spring

**第一天**

# 课程计划

1. 组件式开发和依赖问题
2. spring概述
3. spring的IOC

# 组件式开发和依赖问题

## 组件式开发案例

说明：组件式开发指的是各个业务子系统（或者系统中的各个模块）相互相对独立，可以实现根据客户需求定制化，想要哪一个子系统，就集成哪一个子系统。



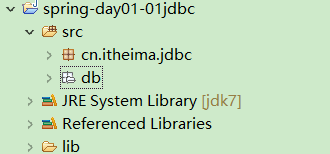
## 依赖问题

### jdbc连接数据库的一些细节问题

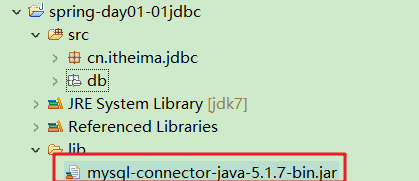
#### 创建数据库和表

|  |
| --- |
| /\*创建客户表\*/  **CREATE** **TABLE** `cst\_customer` (  `cust\_id` bigint(32) **NOT** **NULL** AUTO\_INCREMENT COMMENT '客户编号(主键)',  `cust\_name` **varchar**(32) **NOT** **NULL** COMMENT '客户名称(公司名称)',  `cust\_source` **varchar**(32) **DEFAULT** **NULL** COMMENT '客户信息来源',  `cust\_industry` **varchar**(32) **DEFAULT** **NULL** COMMENT '客户所属行业',  `cust\_level` **varchar**(32) **DEFAULT** **NULL** COMMENT '客户级别',  `cust\_address` **varchar**(128) **DEFAULT** **NULL** COMMENT '客户联系地址',  `cust\_phone` **varchar**(64) **DEFAULT** **NULL** COMMENT '客户联系电话',  **PRIMARY** **KEY** (`cust\_id`)  ) ENGINE=InnoDB AUTO\_INCREMENT=90 **DEFAULT** CHARSET=utf8;  **INSERT** **INTO** `cst\_customer` **VALUES** ('1', '传智播客','6', '1', '22','津安创意园','0208888887');  **INSERT** **INTO** `cst\_customer` **VALUES** ('2', '黑马程序员','6', '1', '22','津安创意园','0208888887');  **INSERT** **INTO** `cst\_customer` **VALUES** ('3', '传智专修学院','6', '1', '22','津安创意园','0208888887');  **INSERT** **INTO** `cst\_customer` **VALUES** ('4', '华山派','7', '2', '23','津安创意园','0208888886');  **INSERT** **INTO** `cst\_customer` **VALUES** ('5', '武当派','7', '2', '23','津安创意园','0208888886');  **INSERT** **INTO** `cst\_customer` **VALUES** ('6', '丐帮','7', '2', '23','津安创意园','0208888886'); |

#### 创建项目



#### 加入数据库驱动包

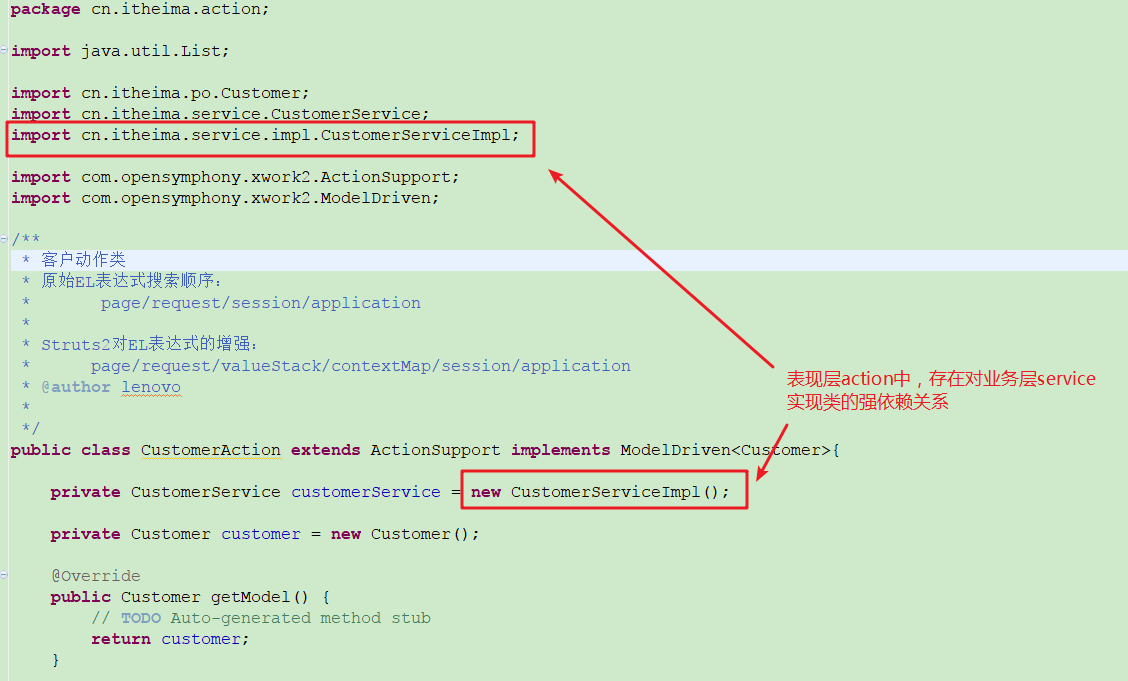


#### 编写jdbc案例代码

|  |
| --- |
| /\*\*  \* 讲解jdbc连接数据库的细节问题  \* **@author** lenovo  \*  \*/  **public** **class** JdbcDemo {  /\*\*  \* 存在的问题：  \* 1.DriverManager.registerDriver(new com.mysql.jdbc.Driver());  \*  \* 当我们注册驱动时，当前类和mysql数据库驱动类有很强的依赖关系，紧密耦合。  \*  \* 面向对象编程原则：  \* 高类聚，低耦合，在程序开发中，需要做到编译时不依赖，运行时才依赖  \*  \* 目前我们知道的解决方式：  \* 通过反射根据全限定类名，获取注册驱动的功能实现  \*  \* 依然存在的问题：  \* 1.当前类还是依赖mysql数据库驱动：com.mysql.jdbc.Driver  \* 2.如果此时我们换成oracle数据库驱动，还是要修改java代码  \* 3.有没有更好的解决方式，实现不管使用mysql数据库，还是oracle数据库，都不需要修改java代码  \* **@param** args  \*/  **public** **static** **void** main(String[] args){  // **TODO** Auto-generated method stub  Connection con = **null**;  PreparedStatement psmt = **null**;  ResultSet rs = **null**;  **try**{  // 加载驱动  //DriverManager.registerDriver(new com.mysql.jdbc.Driver());  Class.*forName*("com.mysql.jdbc.Driver");    // 创建数据库连接对象  con = DriverManager.*getConnection*("jdbc:mysql://127.0.0.1:3308/crm", "root", "admin");    // 定义sql语句  String sql = "select \* from cust\_customer";    // 创建statement语句对象  psmt = con.prepareStatement(sql);  // 设置参数    // 执行  rs = psmt.executeQuery();  // 处理结果集  **while**(rs.next()){  System.*out*.println("客户id："+rs.getInt("cust\_id")+",客户名称："+rs.getString("cust\_name"));  }  }**catch**(Exception e){  e.printStackTrace();  }**finally**{  // 释放资源  **try**{  **if**(rs != **null**) rs.close();  **if**(psmt != **null**) psmt.close();  **if**(con != **null**) con.close();  }**catch**(Exception e){  e.printStackTrace();  }    }  }  } |

### 曾经项目中的强依赖关系案例

#### 表现层action中，存在对业务层service实现类的强依赖关系



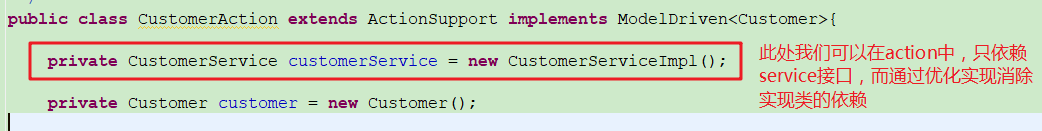
#### 业务层service中，存在对持久层dao实现类的强依赖关系



## 依赖问题解决

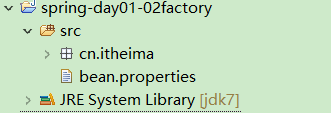
在（1.2.依赖问题）中，我们发现应用程序的类和类之间，不可避免都会产生依赖关系，我们把这种依赖关系，称为耦合。在实际的项目开发中，有些依赖关系是必须的，有些依赖关系可以通过优化实现方式清除。

比如在action中，对业务层的依赖，我们可以优化代码，实现只依赖service层的接口，而不依赖于具体的实现类。从而降低应用系统的耦合性。



### 工厂模式解耦实现

#### 创建项目



#### 编写持久层对象

##### 客户dao接口

|  |
| --- |
| **public** **interface** CustomerDao {    /\*\*  \* 保存客户数据  \*/  **void** saveCustomer();  } |

##### 客户dao实现类

|  |
| --- |
| **public** **class** CustomerDaoImpl **implements** CustomerDao {  /\*\*  \* 保存客户  \*/  **public** **void** saveCustomer() {  System.*out*.println("保存了一个客户数据");  }  } |

#### 编写业务层对象

##### 客户service接口

|  |
| --- |
| **public** **interface** CustomerService {    /\*\*  \* 保存客户  \*/  **void** saveCustomer();  } |

##### 客户service实现类

|  |
| --- |
| **public** **class** CustomerServiceImpl **implements** CustomerService {    // 定义客户dao对象  **private** CustomerDao customerDao = (CustomerDao) BeanFactory.*getBeanFactory*().getInstance("CUSTOMERDAO");  /\*\*  \* 保存客户  \*/  **public** **void** saveCustomer() {  customerDao.saveCustomer();  }  } |

##### 编写表现层对象

|  |
| --- |
| **public** **class** CustomerAction {  /\*\*  \*  \*/  **public** **static** **void** main(String[] args) {  /\*\*  \* 保存客户操作  \*/  CustomerService customerService = **new** CustomerServiceImpl();  customerService.saveCustomer();  }  } |

#### 编写工厂类

|  |
| --- |
| **public** **class** BeanFactory {    /\*\*  \* 编写工厂类的四个基本步骤：  \* 1.声明一个私有的工厂对象引用  \* 2.把构造方法私有化  \* 3.使用静态代码块初始化  \* 4.提供公有的静态方法，获取工厂类对象的引用  \*/  // 1.声明一个私有的工厂对象引用  **private** **static** BeanFactory *beanFactory*= **null**;  **private** **static** Properties *prop*;    // 2.把构造方法私有化  **private** BeanFactory(){    }    // 3.使用静态代码块初始化  **static**{  *beanFactory* = **new** BeanFactory();    InputStream inStream = BeanFactory.**class**.getClassLoader().getResourceAsStream("bean.properties");  *prop* = **new** Properties();  **try** {  *prop*.load(inStream);  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }    // 4.提供公有的静态方法，获取工厂类对象的引用  **public** **static** BeanFactory getBeanFactory(){  **return** *beanFactory*;  }  /\*\*  \* 工厂创建对象：  \* 1.通过配置文件，配置具体的实现类  \* 2.如果需要修改为其它的实现类对象，不需要再修改java代码，只需要修改配置文件即可  \* 3.这里可以统一创建我们需要的全部目标对象  \*  \*/  **public** Object getInstance(String beanName){  String className = (String) *prop*.get(beanName);  Object result = **null**;  **try** {  result = Class.*forName*(className).newInstance();  } **catch** (Exception e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }    **return** result;  }  } |

#### 准备bean.properties文件

|  |
| --- |
| CUSTOMERDAO=cn.itheima.dao.impl.CustomerDaoImpl  CUSTOMERSERVICE=cn.itheima.service.impl.CustomerServiceImpl |

### 工厂模式解耦理解

在实际项目开发中，我们可以通过配置文件把action、service、dao对象配置起来，当启动服务器加载应用的时候，读取配置文件，创建配置文件中的对象并且保存起来。在接下来实际使用的地方，直接拿过来使用即可。

此时我们需要考虑两个问题：

* 将对象存放在什么地方？

由于在一个应用中，会需要大量的对象，首先考虑的是使用集合Map或者List来存储对象。那么这里选择Map还是List，考虑我们有没有查找需求，有则选择Map。

此处我们的选择是：

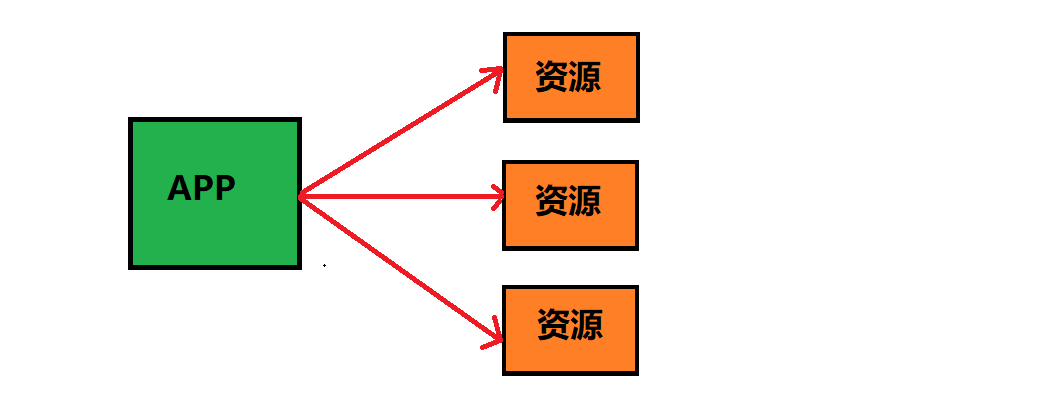
在应用加载时候，创建一个Map，用于存放action、service、dao对象。这个Map我们把它称为容器。

* 什么是工厂呢？

工厂就是负责从容器中，帮助我们获取指定的对象。此时我们获取对象的方式发生了改变。

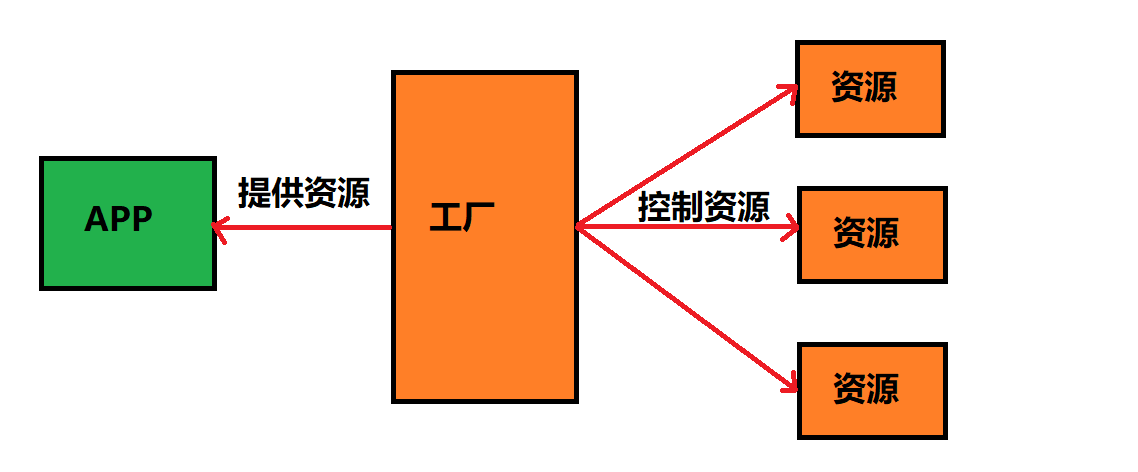
原来：

我们获取对象时，都是采用new的方式。是主动获取。



现在：

我们获取对象时，找工厂要，由工厂创建或者提供给我们。是被动接收。



说明：

这种将对象创建的权利，由在程序代码中主动new对象，转变为由工厂创建提供，我们使用的地方被动接收的方式。称为控制反转。

# spring概述

## spring介绍

spring是分层的Java SE/EE应用的full-stack轻量级开源框架。它是以IOC（Inversion Of Control）控制反转和AOP（Aspect Oriented Programming）面向切面编程为核心，提供了表现层springmvc和持久层spring JDBC以及业务层的事务管理等企业级应用解决方案。还能实现将开源世界中众多优秀的第三方框架和类库整合，成为越来越受欢迎的Java EE企业级应用框架。

## spring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）**

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的优点

* IOC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管理，避免硬编码造成的程序间过渡耦合。用户也不必再为了单例模式类，属性文件解析等底层实现编写代码，可以更加专注于业务系统需求的实现。

* AOP面向切面编程支持

通过spring的AOP功能，方便实现面向切面编程，很多使用传统OOP编程不容易实现的业务功能，可以通过AOP轻松实现。比如事务管理，日志功能。

* 声明式事务支持

通过声明式方式灵活实现事务管理，提高开发效率和质量，将我们（程序员）从单调烦闷的事务管理代码中解脱出来。

* 方便程序测试

可以使用非容器依赖的方式进行程序测试工作，让测试工作更加轻松，更加方便。

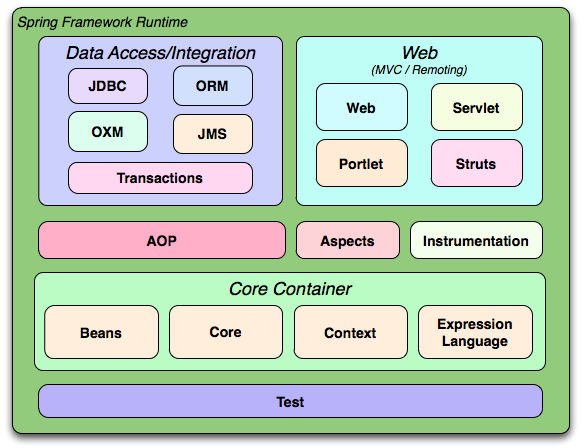
* 集成第三方优秀框架

spring框架降低了第三方框架的集成使用难度，提供了对各种框架（hibernate、struts2、quartz、mybatis等）的直接支持。

* 学习java源码的经典案例

spring的源码设计精妙、结构清晰、匠心独具，处处体现了大师对java设计模式的灵活应用以及java技术的高深造诣。它的源代码无疑是java技术的最佳实践案例。

## spring体系结构



# spring的IOC

## 什么是IOC

IOC（Inversion Of Control）控制反转。是面向对象编程的一个重要法则，用于削减计算机程序间的耦合问题。控制反转中分为两种类型，一种是DI（Dependency Injection）依赖注入；另外一种是DL（Dependency Lookup）依赖查找。实际应用中依赖注入使用更多。

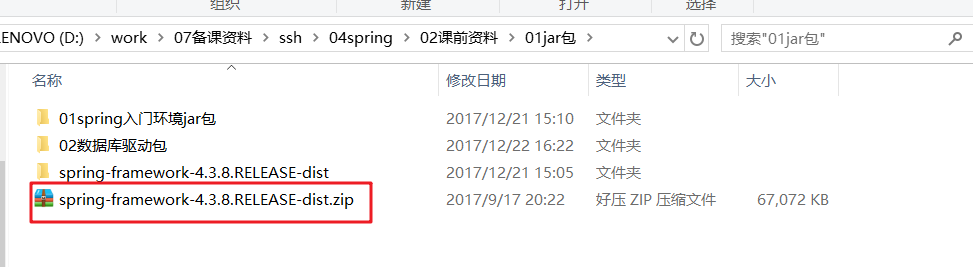
说明：控制反转的简单理解，就等于是在java应用程序中，对象的使用从我们主动new一个对象的方式，转变为由spring创建提供，我们被动接收的方式。即对象的创建管理权利，由程序员创建->转移到spring创建->程序员再被动接收使用对象。

## spring入门案例

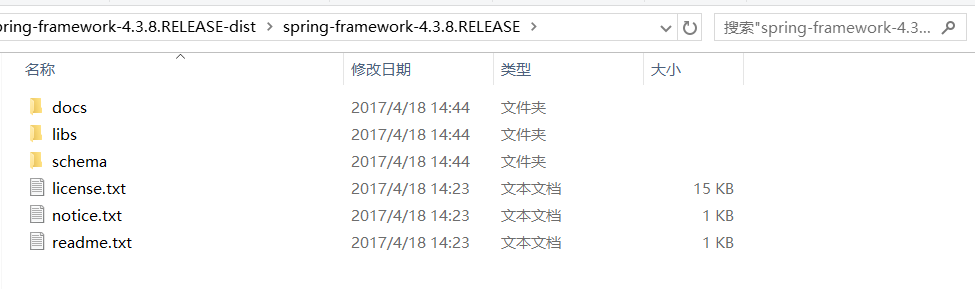
### spring准备

spring官方网址：http://spring.io/

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



解压：



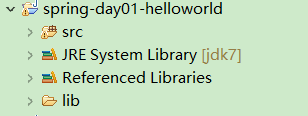
docs：API和开发规范

libs：jar包和源码包

schema：约束

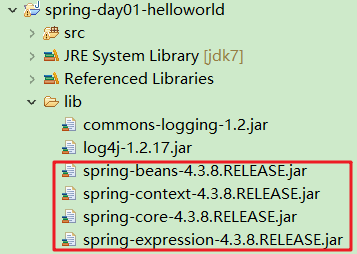
### 创建项目

说明：创建一个普通的java项目。

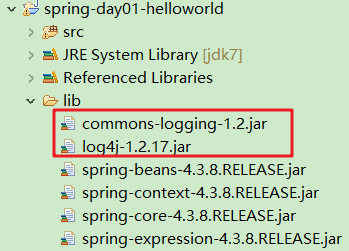


### 加入jar包

复制spring-beans-4.3.8.RELEASE.jar、spring-core-4.3.8.RELEASE.jar、spring-context-4.3.8.RELEASE.jar、spring-expression-4.3.8.RELEASE.jar 包到项目lib目录：

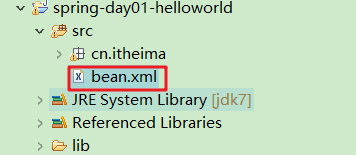


复制commons-logging-1.2.jar、log4j-1.2.17.jar 包到项目lib目录：



### 在类的根路径下创建bean.xml文件

说明：文件的名称可以修改。



导入约束：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!-- 导入schema  约束的位置在:  ..\spring-framework-4.3.8.RELEASE\docs\spring-framework-reference\html\xsd-configuration.html  文件中。    注意：  导入schema约束，并且建议将spring-beans.xsd，增加使用的大版本信息：spring-beans-4.0.xsd  -->  <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-4.0.xsd"*>  </beans> |

### 模拟项目中的三层实现

#### 准备dao层

##### 准备客户dao接口

|  |
| --- |
| /\*\*  \* 客户dao接口  \* **@author** lenovo  \*  \*/  **public** **interface** CustomerDao {    /\*\*  \* 保存客户数据  \*/  **void** saveCustomer();    } |

##### 实现客户dao接口

|  |
| --- |
| /\*\*  \* 客户dao接口实现  \* **@author** lenovo  \*  \*/  **public** **class** CustomerDaoImpl **implements** CustomerDao {    **public** CustomerDaoImpl(){  System.*out*.println("客户dao对象创建了.......");  }  @Override  **public** **void** saveCustomer() {  // **TODO** Auto-generated method stub  System.*out*.println("保存一个客户数据");  }  } |

#### 准备service层

##### 准备客户service接口

|  |
| --- |
| /\*\*  \* 客户service接口  \* **@author** lenovo  \*  \*/  **public** **interface** CustomerService {  /\*\*  \* 保存一个客户数据  \*/  **void** saveCustomer();  } |

##### 实现客户service接口

|  |
| --- |
| /\*\*  \* 客户service接口实现  \* **@author** lenovo  \*  \*/  **public** **class** CustomerServiceImpl **implements** CustomerService {    // 定义客户dao对象  **private** CustomerDao customerDao = **new** CustomerDaoImpl();  @Override  **public** **void** saveCustomer() {  // **TODO** Auto-generated method stub  customerDao.saveCustomer();  }  } |

#### 准备表现层

|  |
| --- |
| /\*\*  \* spring的入门案例  \* **@author** lenovo  \*  \*/  **public** **class** Client {  /\*\*  \* **@param** args  \*/  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  System.*out*.println("模拟应用中的表现层");  }      } |

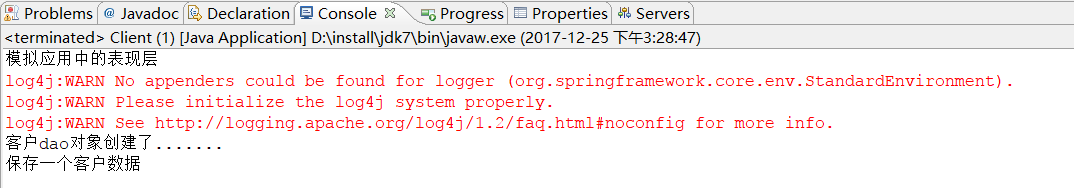
### 把service和dao对象交给spring管理

说明：在bean.xml文件中配置service和dao对象。

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!-- 导入schema  约束的位置在:  ..\spring-framework-4.3.8.RELEASE\docs\spring-framework-reference\html\xsd-configuration.html  文件中。    注意：  导入schema约束，并且建议将spring-beans.xsd，增加使用的大版本信息：spring-beans-4.0.xsd  -->  <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-4.0.xsd"*>  <!-- 配置客户dao实现类对象,说明：  bean标签：配置javaBean对象  id属性：指定对象的唯一标识名称  class属性：指定类的全限定类名称（包名称+类名称） -->  <bean id=*"customerDao"* class=*"cn.itheima.dao.impl.CustomerDaoImpl"*></bean>    <!-- 配置客户service实现类对象 -->  <bean id=*"customerService"* class=*"cn.itheima.service.impl.CustomerServiceImpl"*></bean>    </beans> |

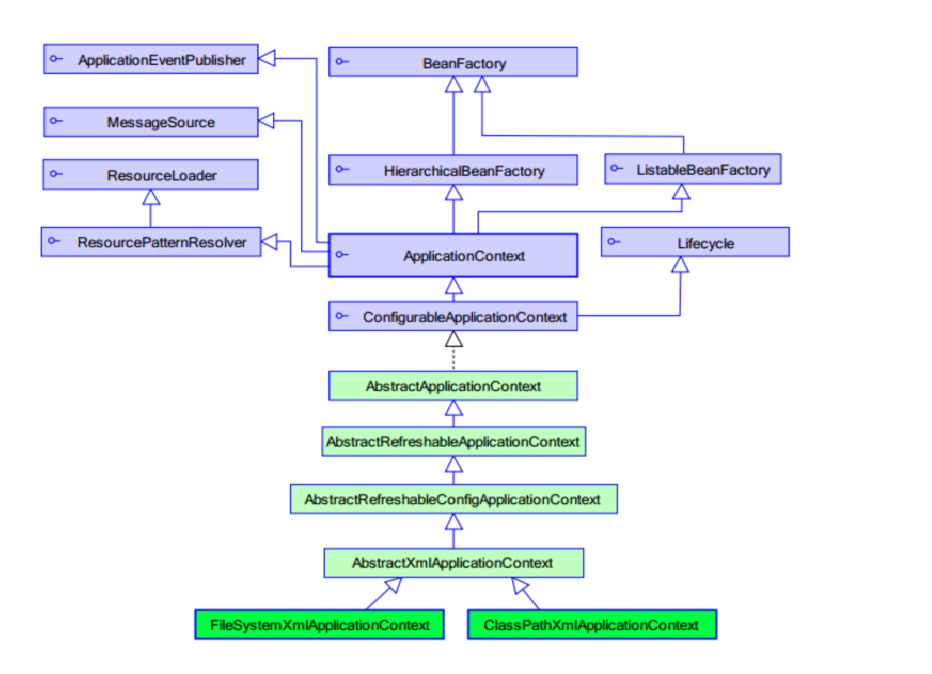
### 在client类中，测试配置

|  |
| --- |
| **public** **class** Client {  /\*\*  \* ApplicationContext接口：  \* 1.它就相当于一个大工厂，提供了根据配置文件中的id属性值，获取对象  \*  \* 2.实现类：  \* ClassPathXmlApplication类，从类的根路径下加载配置文件（企业项目开发中，使用较多）  \* FileSystemXmlApplication类，从文件系统（磁盘）上加载配置文件  \* **@param** args  \*/  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  System.*out*.println("模拟应用中的表现层");    // 1.创建ClassPathXmlApplication对象  // 注意这里的"classpath:"表示在类的根路径下，是一个标识符，加上是使用习惯。也可以不加。  ApplicationContext context = **new** ClassPathXmlApplicationContext("classpath:bean.xml");    // 2.从context中，获取客户dao对象  // getBean方法：获取bean对象  // name参数：配置文件中id属性值  CustomerDao customerDao = (CustomerDao) context.getBean("customerDao");    // 3.调用方法执行  customerDao.saveCustomer();      }      } |



## 基于xml配置的IOC细节

### 工厂类结构图



#### BeanFactory和ApplicationContext区别

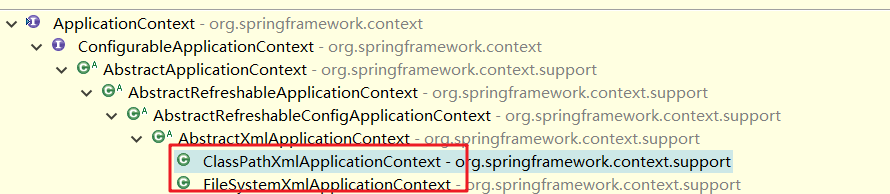
1. BeanFactory是工厂顶层接口
2. ApplicationContext是BeanFactory子接口
3. 创建对象时间点不一样：

BeanFactory创建对象：什么时候使用，什么时候创建。采用延迟创建的思想

ApplicationContext创建对象：加载配置文件，立即创建。采用立即创建的思想

|  |
| --- |
| /\*\*  \* BeanFactory接口：  \* 1.它是spring工厂的顶层接口  \* 2.它采用延迟创建对象方式，即什么时候使用，什么时候创建  \* **@param** args  \*/  @SuppressWarnings("deprecation")  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  System.*out*.println("模拟应用中的表现层");    // 1.创建BeanFactory对象  Resource resource = **new** ClassPathResource("bean.xml");  BeanFactory factory = **new** XmlBeanFactory(resource);    // 2.从工厂中获取客户dao对象  CustomerDao customerDao = (CustomerDao) factory.getBean("customerDao");  customerDao.saveCustomer();  } |

#### ApplicationContext接口实现类



ClassPathXmlApplicationContext：从类根路径下加载配置文件（企业项目开发中使用较多）

FileSystemXmlApplicationContext：从文件系统（磁盘上）加载配置文件

### 配置文件中bean标签细节

#### bean标签

|  |
| --- |
| <bean id=*"customerDaoScope"* class=*"cn.itheima.dao.impl.CustomerDaoImpl"*  scope=*"singleton"*  init-method=*"init"* destroy-method=*"close"*></bean> |

作用：

用于配置对象，让spring创建管理。默认是使用无参的构造方法，如果没有无参的构造方法创建对象会失败。

属性：

id：指定对象的唯一标识名称

class：指定类的全限定类名称（包名称+类名称）

scope：指定bean的作用范围

singleton：单例（默认值）

prototype：多例

request：WEB项目中，spring将创建的对象，存入request域中

session：WEB项目中，spring将创建的对象，存入session域中

globalsession：WEB项目中，如果是服务器集群环境，应用在整个集群中；如果没有服务器集群，相当于session。

init-method：指定类中的初始化方法名称

destory-method：指定类中的销毁方法名称

#### bean的作用范围和生命周期

单例对象：scope=”singleton”

范围：

一个应用只有一个对象实例，范围是整个应用

生命周期：

出生：加载配置文件，容器创建，对象出生

活着：只要容器存在，对象就一直活着

死亡：容器销毁，对象死亡

多例对象：scope=”prototype”

范围：

在一次使用的过程中

生命周期：

出生：第一次获取对象时，对象出生

活着：在一次使用的过程中，对象活着

死亡：当对象不再使用，也没有被其它的对象引用，交给java垃圾回收器执行回收

#### 创建bean的三种方式

方式一：使用无参构造方法

|  |
| --- |
| <!-- 1.使用默认的构造方法创建对象，注意如果默认的构造方法不存在，会报异常，说明：  实际项目中，此种方法使用较多 -->  <bean id=*"customerDao"*  class=*"cn.itheima.dao.impl.CustomerDaoImpl"*></bean> |

方式二：使用静态工厂方法

|  |
| --- |
| <!-- 2.使用静态工厂创建对象，说明：  factory-method属性：指定静态方法的名称 -->  <bean id=*"staticFactoryCustomerDao"*  class=*"cn.itheima.factory.StaticFactory"* factory-method=*"getCustomerDao"*>  </bean> |
| /\*\*  \* 静态工厂创建实例  \* **@author** lenovo  \*  \*/  **public** **class** StaticFactory {    // 静态工厂方法  **public** **static** CustomerDao getCustomerDao(){  **return** **new** CustomerDaoImpl();  }  } |

方式三：使用实例工厂方法

|  |
| --- |
| <!-- 3.使用实例工厂创建对象,说明：  3.1.首先需要创建实例工厂对象  3.2.指定使用实例工厂对象，工厂方法，创建目标对象（客户dao对象），说明：  factory-bean属性：指定实例工厂对象  factory-method属性：指定实例工厂方法-->  <bean id=*"instanceFactory"* class=*"cn.itheima.factory.InstanceFactory"*></bean> |
| <bean id=*"instanceFactoryCustomerDao"* factory-bean=*"instanceFactory"* factory-method=*"getCustomerDao"*></bean> |
| /\*\*  \* 实例工厂创建对象  \* **@author** lenovo  \*  \*/  **public** **class** InstanceFactory {    // 创建对象方法  **public** CustomerDao getCustomerDao(){    **return** **new** CustomerDaoImpl();  }  } |

## 依赖注入

#### 什么是依赖注入

当程序运行的时候，让spring提供层与层之间的依赖对象。在创建容器时，为我们维护好它们之间的关系。我们在使用中，只需要提供构造方法或者set方法即可。简单说就是缺少什么，就传入什么。

#### 构造方法注入

说明：让spring使用类中的构造方法，给成员变量赋值。

|  |
| --- |
| /\*\*  \* 依赖注入知识点service接口  \* **@author** lenovo  \*  \*/  **public** **class** DependencyInjectionServiceImpl **implements** CustomerService {  **private** String name;  **private** Integer age;  **private** Date birthday;      /\*\*  \* 构造方法注入  \* **@param** name  \* **@param** age  \* **@param** birthday  \*/  **public** DependencyInjectionServiceImpl(String name, Integer age,  Date birthday) {  **super**();  **this**.name = name;  **this**.age = age;  **this**.birthday = birthday;  }    /\*\*  \* set方法注入============================start  \*/  // public void setName(String name) {  // this.name = name;  // }  //  //  // public void setAge(Integer age) {  // this.age = age;  // }  //  //  // public void setBirthday(Date birthday) {  // this.birthday = birthday;  // }  /\*\*  \* set方法注入============================start  \*/  @Override  **public** **void** saveCustomer() {  // **TODO** Auto-generated method stub  System.*out*.println("名称：" + name + ",年龄：" + age + ",生日：" + birthday);  }  } |
| <!-- 构造方法注入方式，说明： 实际项目中使用较少，会使用set方式较多    constructor-arg标签：声明调用构造方法注入    属性：  index：指定参数，在构造方法参数列表中的 索引，从0开始。（使用中与name属性，二选一个即可）  name：指定参数，在构造方法参数列表中的 名称。（使用过程中，与index属性，二选一即可）  type：指定参数，在构造方法参数列表中的 数据类型。（默认可以省略）  value：指定参数值，数据类型是基本类型和字符串  ref：指定参数值，数据类型是其它的bean类型    -->  <bean id=*"dependencyInjectionService"* class=*"cn.itheima.service.impl.DependencyInjectionServiceImpl"*>  <constructor-arg name=*"name"* value=*"小明"* ></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> |

#### set方法注入

说明：让spring使用类中的set方法，给成员变量赋值。

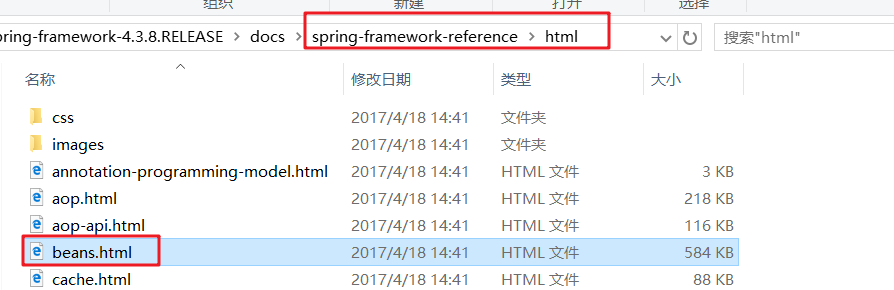
|  |
| --- |
| /\*\*  \* 依赖注入知识点service接口  \* **@author** lenovo  \*  \*/  **public** **class** DependencyInjectionServiceImpl **implements** CustomerService {  **private** String name;  **private** Integer age;  **private** Date birthday;      /\*\*  \* 构造方法注入  \* **@param** name  \* **@param** age  \* **@param** birthday  \*/  // public DependencyInjectionServiceImpl(String name, Integer age,  // Date birthday) {  // super();  // this.name = name;  // this.age = age;  // this.birthday = birthday;  // }    /\*\*  \* set方法注入============================start  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **void** setAge(Integer age) {  **this**.age = age;  }  **public** **void** setBirthday(Date birthday) {  **this**.birthday = birthday;  }  /\*\*  \* set方法注入============================end  \*/  @Override  **public** **void** saveCustomer() {  // **TODO** Auto-generated method stub  System.*out*.println("名称：" + name + ",年龄：" + age + ",生日：" + birthday);  }  } |
| <!-- set方式注入，说明：    property标签：声明调用set方法注入    属性：  name：指定参数在set方法中的参数名称  value：指定参数值，数据类型是基本类型和字符串  ref：指定参数值，数据类型是其它的bean类型    注意事项：  在实际项目中使用set注入，建议将注入的属性名称，与set方法的后面部分一致    -->  <bean id=*"dependencyInjectionService"* class=*"cn.itheima.service.impl.DependencyInjectionServiceImpl"*>  <property name=*"name"* value=*"小花"*></property>  <property name=*"age"* value=*"18"*></property>  <property name=*"birthday"* ref=*"now"*></property>  </bean> |
| <bean id=*"now"* class=*"java.util.Date"*></bean> |

#### p名称空间注入

说明：p名称空间注入，是指直接在bean标签内使用属性p:name方式，本质上还是set方式注入。

第一步：需要在配置文件中，导入p名称空间。p名称空间内容可以从：

spring-framework-reference/html/bean.xml文件找到：



|  |
| --- |
| <?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-4.0.xsd"*>  ......    </beans> |

第二步：配置bean

|  |
| --- |
| <?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-4.0.xsd"*>  <!-- p名称空间注入，说明：  1.p:属性名称，给某个属性赋值  2.p:属性名称-ref，引用对象给某个属性赋值 -->  <bean id=*"dependencyInjectionService"* class=*"cn.itheima.service.impl.DependencyInjectionServiceImpl"*  p:name=*"小花"* p:age=*"18"* p:birthday-ref=*"now"*>  </bean>    </beans> |

#### 集合属性注入

说明：集合属性注入指给类中的集合成员赋值，使用的set方式注入。我们以下注入数组、List、Set、Map、Properties数据类型。

第一步：准备包含集合属性的bean

|  |
| --- |
| /\*\*  \* 集合属性注入  \* **@author** lenovo  \*  \*/  **public** **class** CollectionDIServiceImpl **implements** CustomerService {    **private** String[] arr;  **private** List<String> list;  **private** Set<String> set;  **private** Map<String,String> map;  **private** Properties prop;  **public** **void** setArr(String[] arr) {  **this**.arr = arr;  }  **public** **void** setList(List<String> list) {  **this**.list = list;  }  **public** **void** setSet(Set<String> set) {  **this**.set = set;  }  **public** **void** setMap(Map<String, String> map) {  **this**.map = map;  }  **public** **void** setProp(Properties prop) {  **this**.prop = prop;  }  @Override  **public** **void** saveCustomer() {  // **TODO** Auto-generated method stub  System.*out*.println(Arrays.*asList*(arr));  System.*out*.println(list);  System.*out*.println(set);  System.*out*.println(map);  System.*out*.println(prop);  }  } |

第二步：配置bean

|  |
| --- |
| <!-- 集合属性注入，说明：  1.数据结构相同，标签可以互换  List结构：  array/list/set  Map结构：  map/entry/props/prop-->  <bean id=*"collectionDIService"* class=*"cn.itheima.service.impl.CollectionDIServiceImpl"*>  <!-- 注入数组 -->  <property name=*"arr"*>  <array>  <value>arr1</value>  <value>arr2</value>  <value>arr3</value>  </array>  </property>    <!-- 注入list -->  <property name=*"list"*>  <list>  <value>list1</value>  <value>list2</value>  <value>list3</value>  </list>  </property>    <!-- 注入set -->  <property name=*"set"*>  <set>  <value>set1</value>  <value>set2</value>  <value>set3</value>  </set>  </property>    <!-- 注入map -->  <property name=*"map"*>  <map>  <entry key=*"map1"* value=*"map-v1"*></entry>  <entry key=*"map2"* value=*"map-v2"*></entry>  <entry key=*"map3"* value=*"map-v3"*></entry>  </map>  </property>    <!-- 注入properties -->  <property name=*"prop"*>  <props>  <prop key=*"prop1"*>prop-v1</prop>  <prop key=*"prop2"*>prop-v2</prop>  <prop key=*"prop3"*>prop-v3</prop>  </props>  </property>  </bean> |

## 练习

说明：处理之前CustomerService中，调用CustomerDao空指针异常。

|  |
| --- |
| <?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-4.0.xsd"*>  <!-- 配置客户dao实现类对象,说明：  bean标签：配置实例化组件对象  id属性：指定实例对象的唯一标识（对象引用的名称）  class属性：指定类的全路径（包名称+类名称） -->  <bean id=*"customerDao"* class=*"cn.itheima.dao.impl.CustomerDaoImpl"*></bean>    <!-- 配置客户service实现类对象 -->  <bean id=*"customerService"* class=*"cn.itheima.service.impl.CustomerServiceImpl"*>  <property name=*"customerDao"* ref=*"customerDao"*></property>  </bean>  ......    </beans> |
| **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  ApplicationContext context = **new** ClassPathXmlApplicationContext("classpath:bean.xml");    // CustomerService customerService = (CustomerService) context.getBean("dependencyInjectionService");  // customerService.saveCustomer();    // 测试集合属性注入  // CustomerService customerService = (CustomerService) context.getBean("collectionDIService");  // customerService.saveCustomer();      // 处理之前CustomerService中，调用CustomerDao空指针异常  CustomerService customerService = (CustomerService) context.getBean("customerService");  customerService.saveCustomer();    } |

## 思考

说明：假如有以下集合属性，如何给它们进行注入？

1. 、List<User> userList;
2. 、Set<User> userSet;
3. 、Map<String,User> userMap;
4. 、User[] userArr;