# Spring 第一天

# 第1章 Spring 概述

## 1.1 spring 概述

### 1.1.1 spring 介绍

Spring 是分层的 Java SE/EE 应用 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，提供了展现层 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。

### 1.1.2 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）
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）

### 1.1.3 spring 的优势

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。

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。

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。

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。

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。

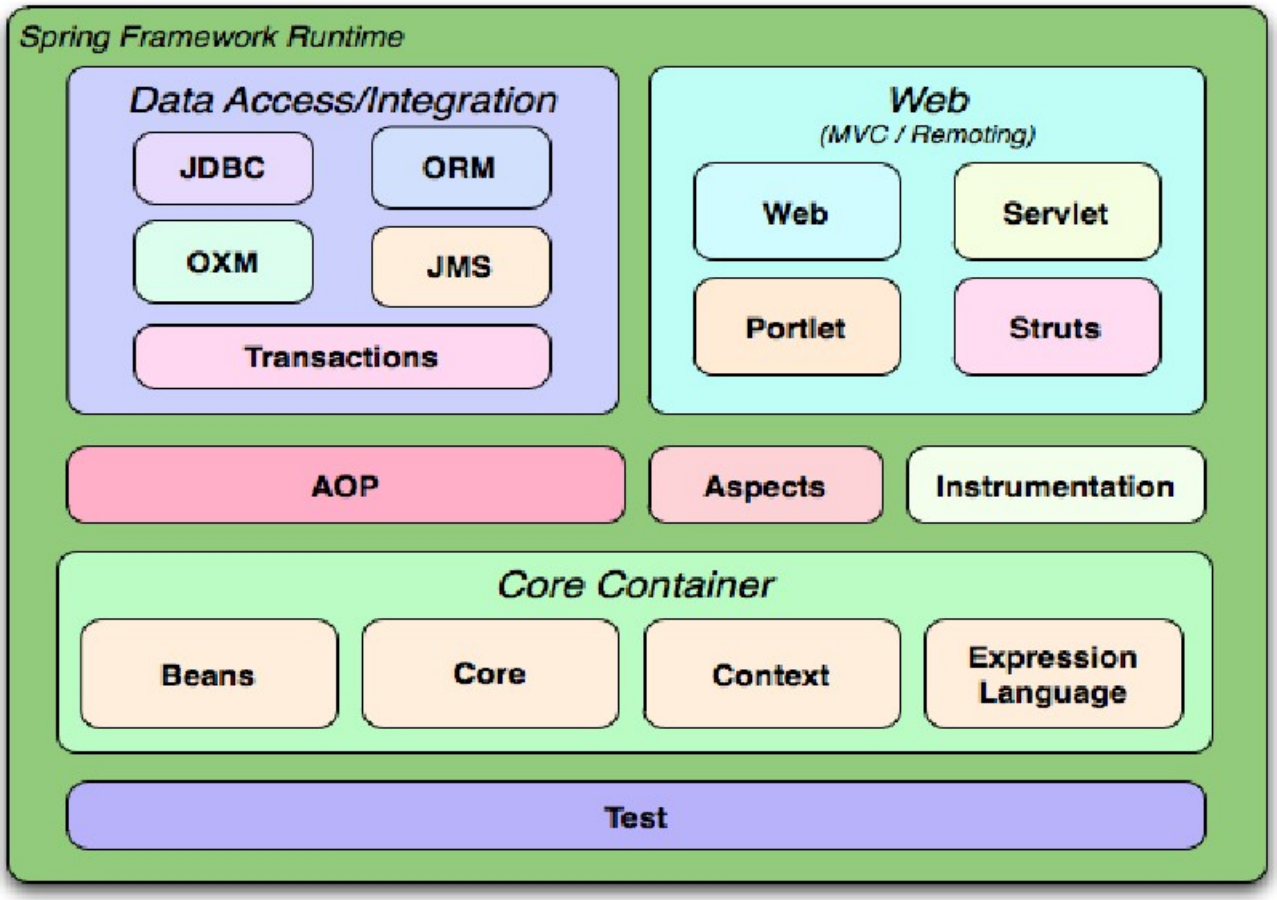
降低 JavaEE API 的使用难度

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。

Java 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。

### 1.1.4 spring 的体系结构



# 1.2 程序的耦合和解耦

## 1.2.1 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。

有些依赖关系是必须的，有些依赖关系可以通过优化代码来解除的。请看下面的示例代码：

```
/**
 * 客户的业务层实现类
 */
public class CustomerServiceImpl implements ICustomerService {

    private ICustomerDao customerDao = new CustomerDaoImpl();
}
```

上面的代码表示：业务层调用持久层，并且此时业务层在依赖持久层的接口和实现类。如果此时没有持久层实现类，编译将不能通过。这种依赖关系就是我们可以通过优化代码解决的。

再比如：

下面的代码中，我们的类依赖了 MySQL 的具体驱动类，如果这时候更换了数据库品牌，我们需要改源码来修改数据库驱动。这显然不是我们想要的。

```
public class JdbcDemo1 {

    /**
     * JDBC 操作数据库的基本入门中存在什么问题？
     *     导致驱动注册两次是个问题，但不是严重的。
     *     严重的问题：是当前类和 mysql 的驱动类有很强的依赖关系。
     *                 当我们没有驱动类的时候，连编译都不让。
     * 那这种依赖关系，就叫做程序的耦合
     *
     * 我们在开发中，理想的状态应该是：
     *     我们应该尽力达到的：编译时不依赖，运行时才依赖。
     *
     * @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
    }
```

```
}
```

### 1.2.2 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");
```

这时的好处是，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。但是因为没有驱动类，所以不能运行。

不过，此处也有个问题，就是我们反射类对象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。

### 1.2.3 工厂模式解耦

在实际开发中我们可以把所有的 dao 和 service 和 action 对象使用配置文件配置起来，当启动服务器应用加载的时候，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。

### 1.2.4 控制反转-Inversion Of Control

上面解耦的思路有 2 个问题：

1、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 action，Service 和 dao 对象。

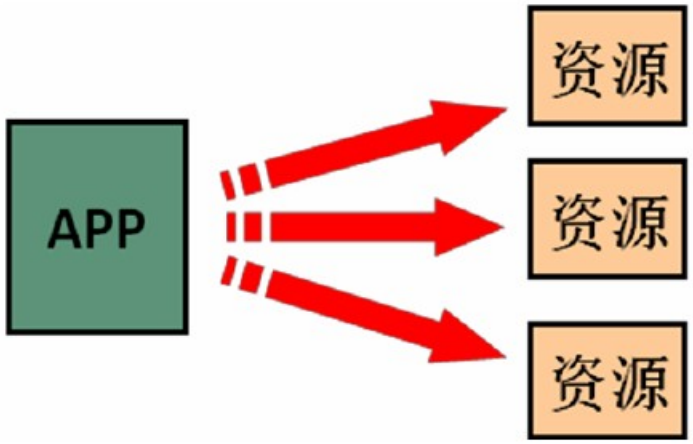
我们把这个 map 称之为容器。

2、还是没解释什么是工厂？

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。

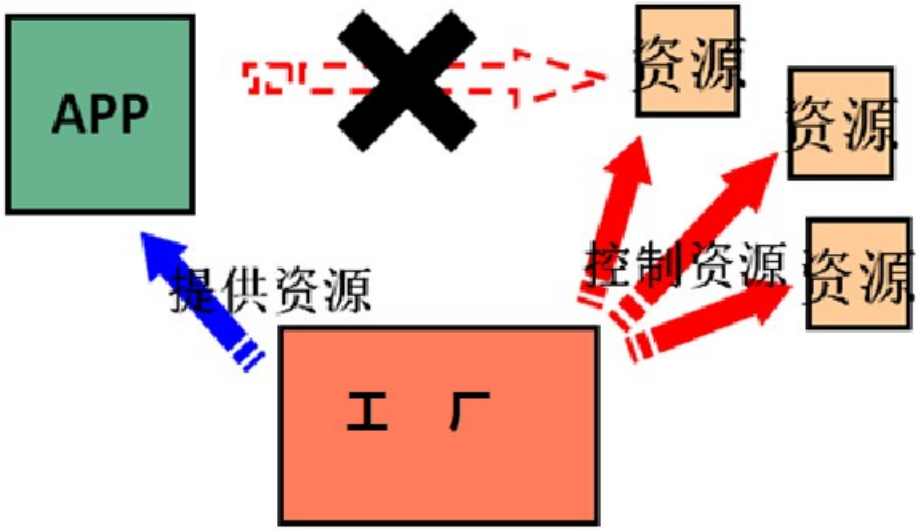
原来：

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。

收藏 0 0

控制反转

同义词 Inversion of Control一般指控制反转

控制反转（Inversion of Control，英文缩写为IoC）是一个重要的面向对象编程的法则来削减计算机程序的耦合问题，也是轻量级的Spring框架的核心。控制反转一般分为两种类型，依赖注入（Dependency Injection，简称DI）和依赖查找（Dependency Lookup）。依赖注入应用比较广泛。

| | | | |
|---|---|---|---|
| 中文名 | 控制反转 | 起源时间 | 2004年 |
| 外文名 | Inverse of Control | 目 的 | 削减计算机程序的耦合问题 |

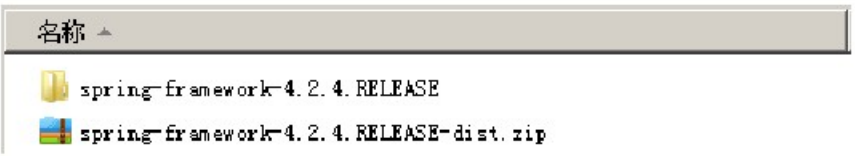
# 第2章 使用 spring 的 IOC 解决程序耦合

## 2.1 案例的前期准备

本章我们使用的案例是，客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伤改查操作，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。

### 2.1.1 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.
```




我们上课使用的版本是 spring4.2.4。

### 2.1.2 创建业务层接口和实现类

```
/**
 * 客户的业务层接口
*/
public interface ICustomerService {
    /**
     * 保存客户
     * @param customer
     */
    void saveCustomer();
}
```

```
/**
 * 客户的业务层实现类
 */
public class CustomerServiceImpl implements ICustomerService {

    private ICustomerDao customerDao = new CustomerDaoImpl();//此处有依赖关系

    @Override
    public void saveCustomer() {
        customerDao.saveCustomer();
    }
}
```

### 2.1.3 创建持久层接口和实现类

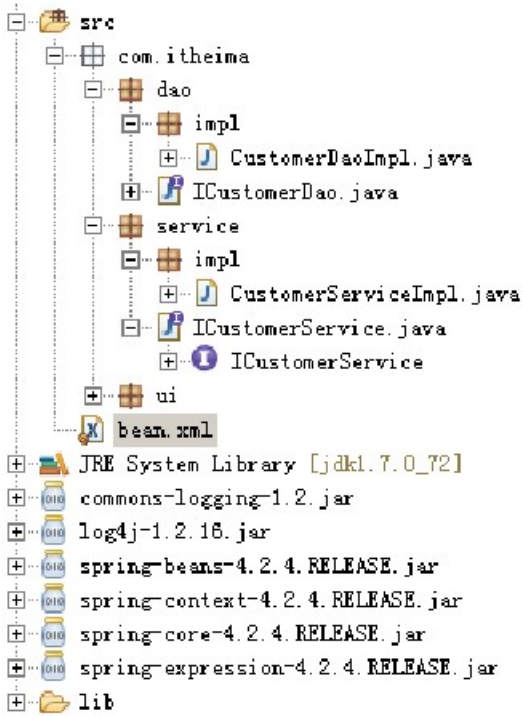
```
/**
 * 客户的持久层接口
 */
public interface ICustomerDao {
    /**
     * 保存客户
     */
    void saveCustomer();
}

/**
 * 客户的持久层实现类
 */
public class CustomerDaoImpl implements ICustomerDao {
    @Override
    public void saveCustomer() {
        System.out.println("保存了客户");
    }
}
```

## 2.2 基于 XML 的配置（入门案例）

### 2.2.1 第一步：拷贝必备的 jar 包到工程的 lib 目录中


### 2.2.2 第二步：在类的根路径下创建一个任意名称的 xml 文件（不能是中文）




给配置文件导入约束：

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- 导入 schema
约束的位置在：
    ..\spring-framework-4.2.4.RELEASE\docs\spring-framework-reference\html\xsd-configuration.html
    文件中。
注意：要导入 schema 约束
```

```
    -->
    <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>
```

### 2.2.3 第三步：把资源交给 spring 来管理，在配置文件中配置 service 和 dao

```
    <!-- 把资源交给 spring 来管理 -->
    <bean id="customerDao" class="com.itheima.dao.impl.CustomerDaoImpl"/>
    <bean                                                id="customerService"
class="com.itheima.service.impl.CustomerServiceImpl"/>
```

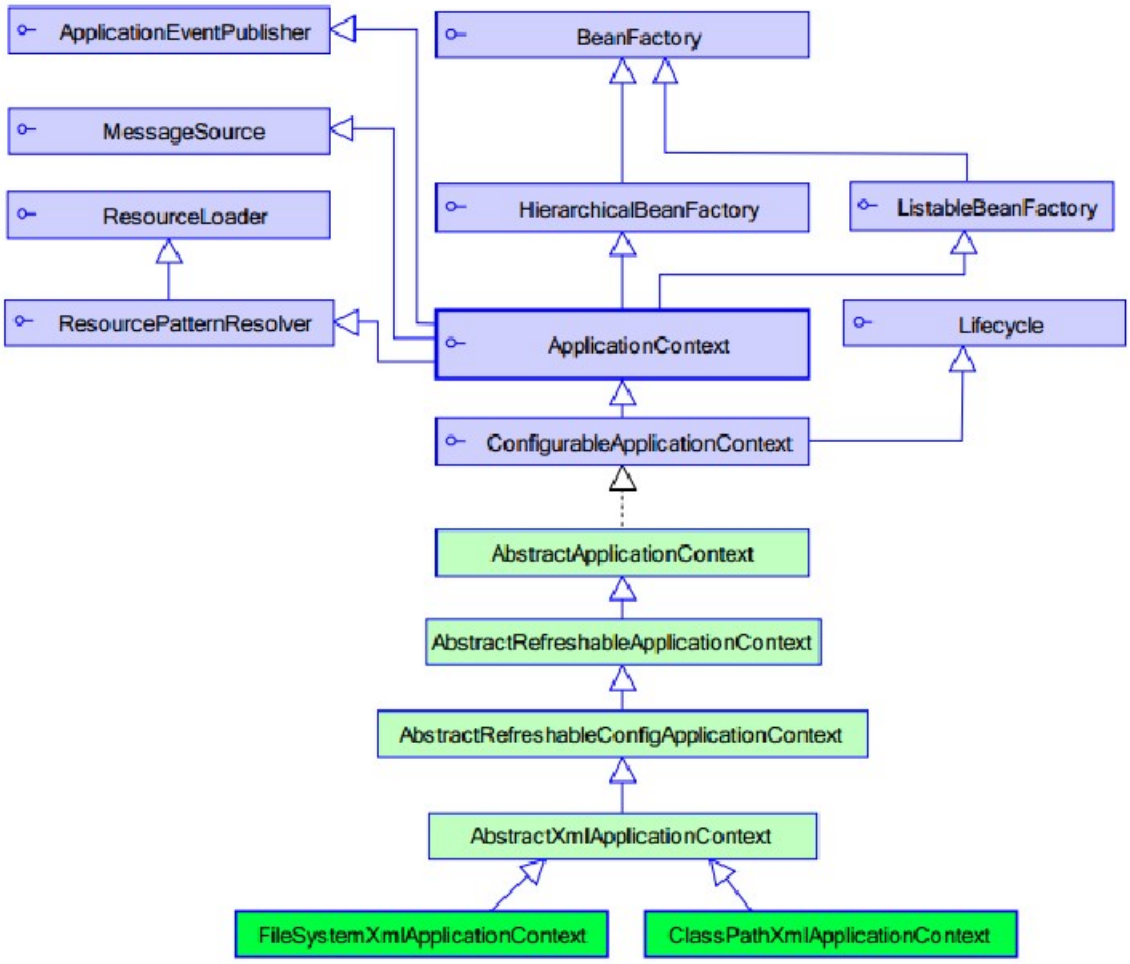
### 2.2.4 测试配置是否成功

```
    /**
     * 模拟一个表现层
     */
    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
            ICustomerService          cs          =          (ICustomerService)
ac.getBean("customerService");
            System.out.println(cs);

            ICustomerDao cd = (ICustomerDao) ac.getBean("customerDao");
            System.out.println(cd);
        }
    }
```

# 第3章 Spring 基于 XML 的 IOC 细节

## 3.1 spring 中工厂的类结构图

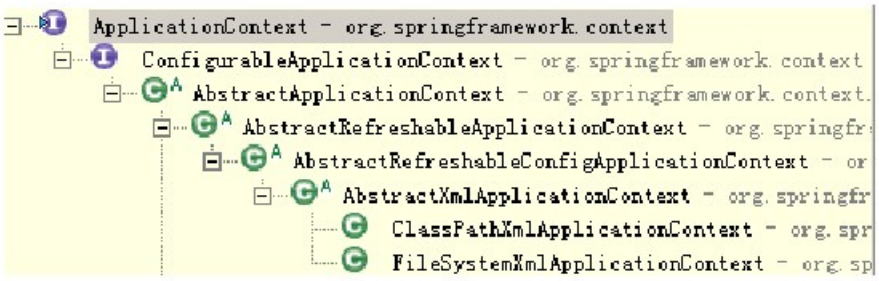



### 3.1.1 BeanFactory 和 ApplicationContext 的区别

```
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。
```

### 3.1.2 ApplicationContext 接口的实现类

**ApplicationContext** 的实现类，如下图：




```
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。
```

## 3.2 IOC 中 bean 标签和管理对象细节

### 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.
        * globalSession :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。
```

### 3.2.2 bean 的作用范围和生命周期

```
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。
```

### 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="customerService"
class="com.itheima.service.impl.CustomerServiceImpl"/>
```

```
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 ICustomerService createCustomerService(){
        return new CustomerServiceImpl();
    }
}
<!-- 此种方式是：
     使用 StaticFactory 类中的静态方法 createCustomerService 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="customerService"
      class="com.itheima.factory.StaticFactory"
```

```
        factory-method="createCustomerService"></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 ICustomerService createCustomerService(){
            return new CustomerServiceImpl();
        }
    }
        <!-- 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="customerService"
              factory-bean="instancFactory"
              factory-method="createCustomerService"></bean>
```

## 3.3 spring 的依赖注入

### 3.3.1 依赖注入的概念

它是 spring 框架核心 ioc 的具体实现方式。简单的说，就是坐等框架把对象传入，而不用我们自己去获取。

### 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 CustomerServiceImpl implements ICustomerService {

        private String name;
        private Integer age;
```

```
        private Date birthday;

        public CustomerServiceImpl(String name, Integer age, Date birthday) {
            this.name = name;
            this.age = age;
            this.birthday = birthday;
        }

        @Override
        public void saveCustomer() {
            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="customerService"
class="com.itheima.service.impl.CustomerServiceImpl">
        <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>
```

### 3.3.3 set 方法注入

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 CustomerServiceImpl implements ICustomerService {

    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 saveCustomer() {
        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="customerService"
class="com.itheima.service.impl.CustomerServiceImpl">
        <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>
```

### 3.3.4 使用 p 名称空间注入数据（本质还是调用 set 方法）

此种方式是通过在 xml 中导入 p 名称空间，使用 p:propertyName 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。

Java 类代码：

```
/**
 * 使用 p 名称空间注入，本质还是调用类中的 set 方法
 */
public class CustomerServiceImpl4 implements ICustomerService {

    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 saveCustomer() {
        System.out.println(name+","+age+","+birthday);
    }
}
```

配置文件代码：

```
<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">

    <bean id="customerService"
          class="com.itheima.service.impl.CustomerServiceImpl4"
          p:name="test" p:age="21" p:birthday-ref="now"/>
</beans>
```

### 3.3.5 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 CustomerServiceImpl implements ICustomerService {

    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 saveCustomer() {
        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="customerService"
class="com.itheima.service.impl.CustomerServiceImpl">
        <!-- 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>
```

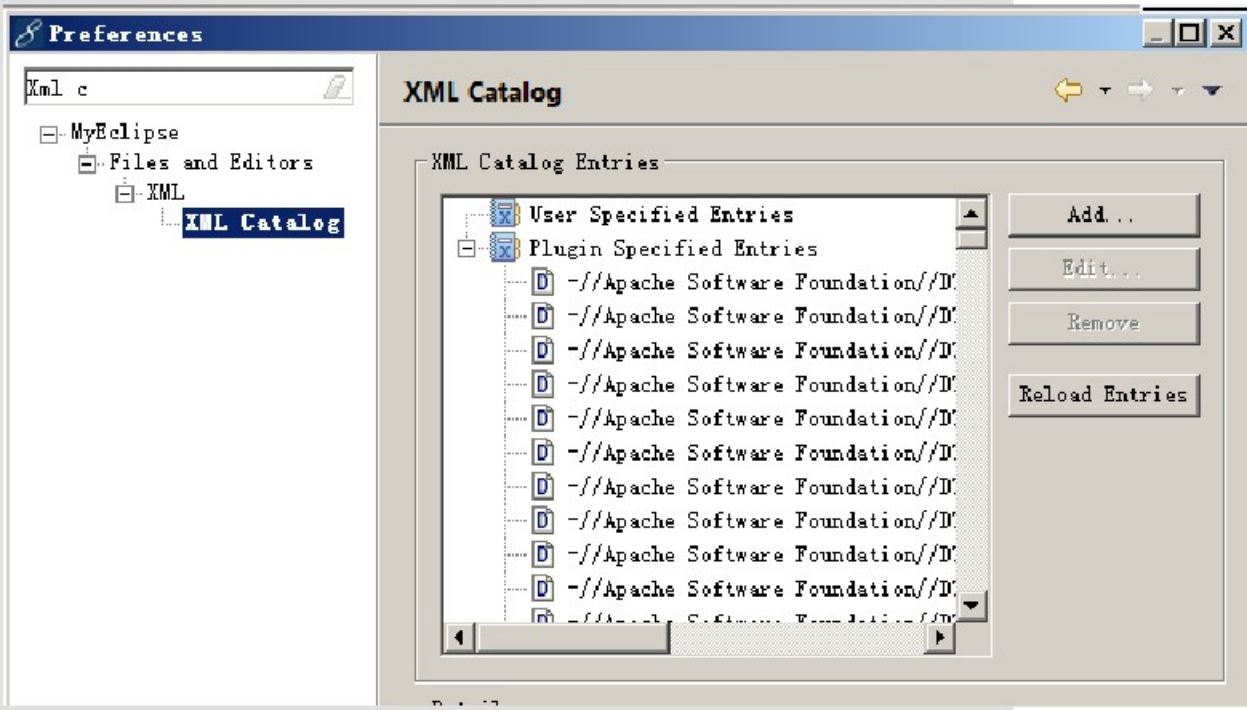
# 第4章 附录

## 4.1 Spring 配置文件中提示的配置

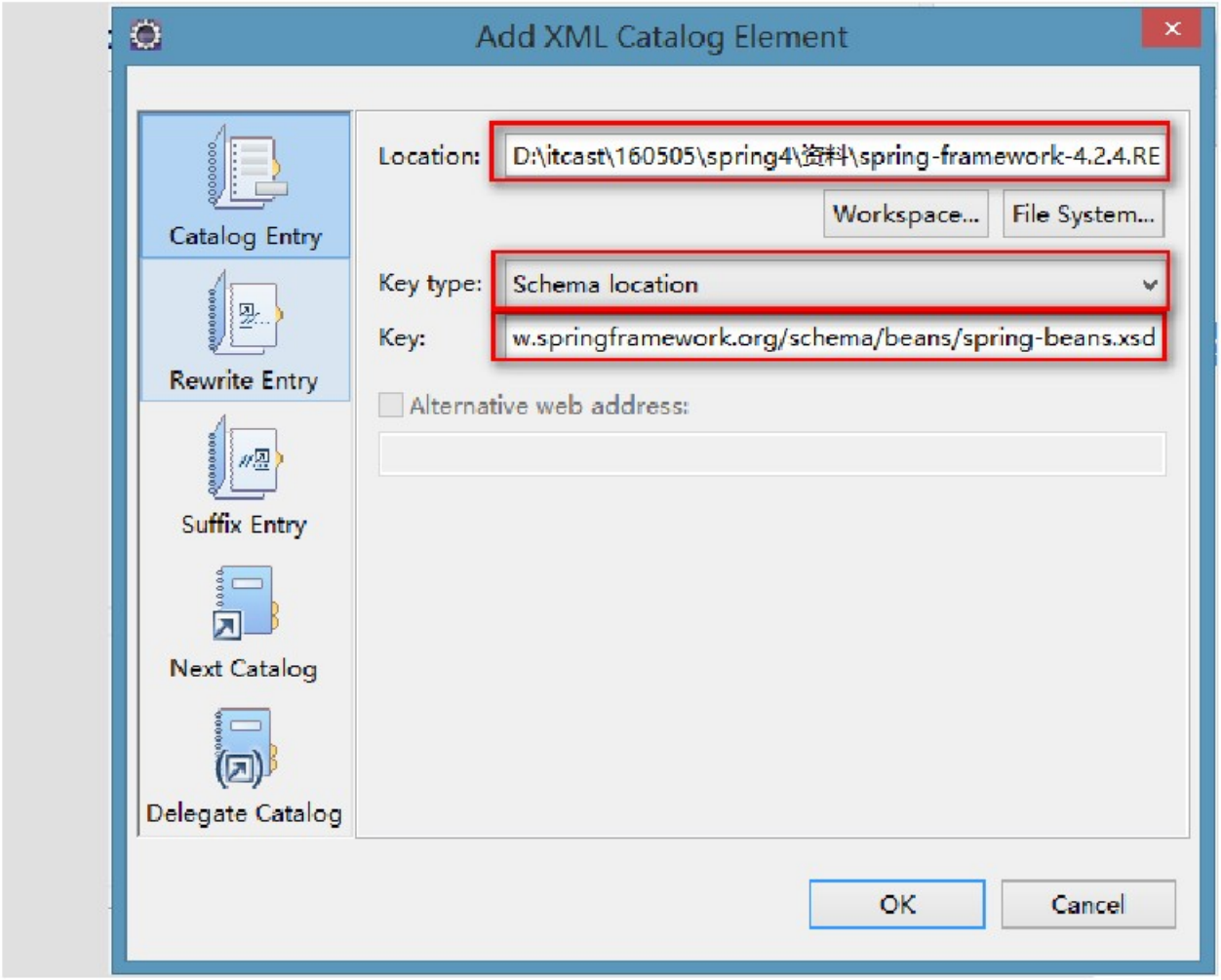
复制路径：

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

查找 XML Catalog:




点击 Add..


Add XML Catalog Element
Catalog Entry
Rewrite Entry
Suffix Entry
Next Catalog
Delegate Catalog
Location:
D:\itcast\160505\spring4\资料\spring-framework-4.2.4.RE
Workspace...
File System...
Key type:
Schema location
Key:
w.springframework.org/schema/beans/spring-beans.xsd
Alternative web address:
OK
Cancel