# Spring教程

作者：钱安川（Moxie）



注：后面的内容我将不再完善，但网上的朋友可以帮助完善，只需注明住处即可。

[Spring教程 1](#_Toc128230972)

[Spring框架概述 3](#_Toc128230973)

[Spring是什么？ 3](#_Toc128230974)

[Spring的历史 4](#_Toc128230975)

[Spring的使命（Mission Statement） 4](#_Toc128230976)

[Spring受到的批判 4](#_Toc128230977)

[Spring包含的模块 5](#_Toc128230978)

[总结 6](#_Toc128230979)

[Spring的IoC容器 6](#_Toc128230980)

[用户注册的例子 7](#_Toc128230981)

[面向接口编程 8](#_Toc128230982)

[（用户持久化类）重构第一步——面向接口编程 8](#_Toc128230983)

[重构第二步——工厂（Factory）模式 9](#_Toc128230984)

[重构第三步——工厂（Factory）模式的改进 10](#_Toc128230985)

[重构第四步－IoC容器 11](#_Toc128230986)

[控制反转（IoC）/依赖注入（DI） 11](#_Toc128230987)

[什么是控制反转/依赖注入？ 11](#_Toc128230988)

[依赖注入的三种实现形式 12](#_Toc128230989)

[BeanFactory 14](#_Toc128230990)

[BeanFactory管理Bean（组件）的生命周期 15](#_Toc128230991)

[Bean的定义 16](#_Toc128230992)

[Bean的之前初始化 19](#_Toc128230993)

[Bean的准备就绪（Ready）状态 21](#_Toc128230994)

[Bean的销毁 21](#_Toc128230995)

[ApplicationContext 21](#_Toc128230996)

[Spring的AOP框架 21](#_Toc128230997)

[Spring的数据层访问 22](#_Toc128230998)

[Spring的声明式事务 22](#_Toc128230999)

[Spring对其它企业应用支持 22](#_Toc128231000)

名词解释

容器：

框架：

容器

框架

组件：

服务：

# Spring框架概述

* **主要内容**：介绍Spring的历史，Spring的概论和它的体系结构，重点阐述它在J2EE中扮演的角色。
* **目的**：让学员全面的了解Spring框架，知道Spring框架所提供的功能，并能将Spring框架和其它框架（WebWork/Struts、hibernate）区分开来。

## Spring是什么？

Spring是一个开源框架，它由Rod Johnson创建。它是为了解决企业应用开发的复杂性而创建的。Spring使用基本的JavaBean来完成以前只可能由EJB完成的事情。然而，Spring的用途不仅限于服务器端的开发。从简单性、可测试性和松耦合的角度而言，任何Java应用都可以从Spring中受益。

* 目的：解决企业应用开发的复杂性
* 功能：使用基本的JavaBean代替EJB，并提供了更多的企业应用功能
* 范围：任何Java应用

**简单来说，Spring是一个轻量级的控制反转(IoC)和面向切面(AOP)的容器框架。**

* 轻量——从大小与开销两方面而言Spring都是轻量的。完整的Spring框架可以在一个大小只有1MB多的JAR文件里发布。并且Spring所需的处理开销也是微不足道的。此外，Spring是非侵入式的：典型地，Spring应用中的对象不依赖于Spring的特定类。
* 控制反转——Spring通过一种称作控制反转（IoC）的技术促进了松耦合。当应用了IoC，一个对象依赖的其它对象会通过被动的方式传递进来，而不是这个对象自己创建或者查找依赖对象。你可以认为IoC与JNDI相反——不是对象从容器中查找依赖，而是容器在对象初始化时不等对象请求就主动将依赖传递给它。
* 面向切面——Spring提供了面向切面编程的丰富支持，允许通过分离应用的业务逻辑与系统级服务（例如审计（auditing）和事务（）管理）进行内聚性的开发。应用对象只实现它们应该做的——完成业务逻辑——仅此而已。它们并不负责（甚至是意识）其它的系统级关注点，例如日志或事务支持。
* 容器——Spring包含并管理应用对象的配置和生命周期，在这个意义上它是一种容器，你可以配置你的每个bean如何被创建——基于一个可配置原型（prototype），你的bean可以创建一个单独的实例或者每次需要时都生成一个新的实例——以及它们是如何相互关联的。然而，Spring不应该被混同于传统的重量级的EJB容器，它们经常是庞大与笨重的，难以使用。
* 框架——Spring可以将简单的组件配置、组合成为复杂的应用。在Spring中，应用对象被声明式地组合，典型地是在一个XML文件里。Spring也提供了很多基础功能（事务管理、持久化框架集成等等），将应用逻辑的开发留给了你。

所有Spring的这些特征使你能够编写更干净、更可管理、并且更易于测试的代码。它们也为Spring中的各种模块提供了基础支持。

## Spring的历史

Spring的基础架构起源于2000年早期，它是Rod Johnson在一些成功的商业项目中构建的基础设施。

在2002后期，Rod Johnson发布了《Expert One-on-One J2EE Design and Development》一书，并随书提供了一个初步的开发框架实现——interface21开发包，interface21就是书中阐述的思想的具体实现。后来，Rod Johnson 在interface21 开发包的基础之上，进行了进一步的改造和扩充，使其发展为一个更加开放、清晰、全面、高效的开发框架——Spring。

2003年2月Spring框架正式成为一个开源项目，并发布于SourceForge中。

## Spring的使命（Mission Statement）

* J2EE应该更加容易使用。
* 面向对象的设计比任何实现技术（比如J2EE）都重要。
* 面向接口编程，而不是针对类编程。Spring将使用接口的复杂度降低到零。（面向接口编程有哪些复杂度？）
* 代码应该易于测试。Spring框架会帮助你，使代码的测试更加简单。
* JavaBean提供了应用程序配置的最好方法。
* 在Java中，已检查异常（Checked exception）被过度使用。框架不应该迫使你捕获不能恢复的异常。

## Spring受到的批判

* + **Spring不是一个“标准”。**Spring不是J2EE规范的一部分，没有通过JCP（Java Community Process）的审核认可。

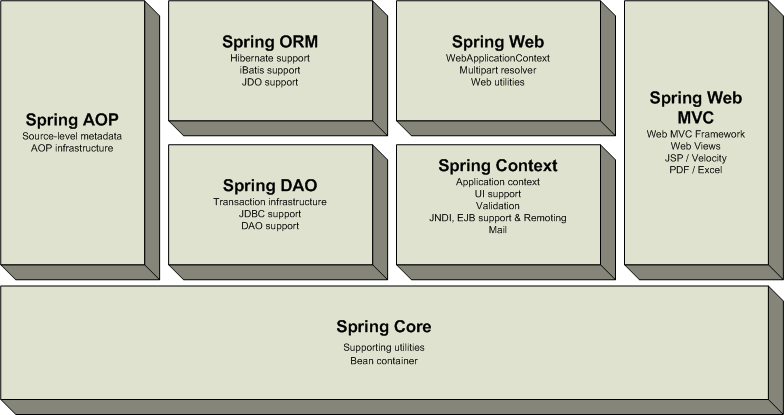
批判来源于EJB的支持者，他们认为EJB是一个标准，是J2EE规范的一部分。当然，标准最主要的目的是希望在应用服务器之间是可移植的，可是EJB的移植却并不轻松，不同应用服务器的ejb部署描述文件总是有着差异。而且EJB开发的类完全依赖于EJB容器。而Spring对其管理的Bean没有任何形式的侵入，这样的Bean是普通Java对象(POJO)，那么它就遵循Java标准，可以到处移植。

* + **Spring是“超重量级”的。**

Spring涉及的内容确实很多（例如：提供了对jdbc、ORM、远程访问等等的支持），但其本质还是Java技术的庞大。Spring只是为了这些技术提供更好的使用方案而已。同时，你可以只选取你需要使用的部分。

## Spring包含的模块

Spring框架由七个定义明确的模块组成（图1.1）。



（Spring框架概览图）

如果作为一个整体，这些模块为你提供了开发企业应用所需的一切。但你不必将应用完全基于Spring框架。你可以自由地挑选适合你的应用的模块而忽略其余的模块。

就像你所看到的，所有的Spring模块都是在核心容器之上构建的。容器定义了Bean是如何创建、配置和管理的——更多的Spring细节。当你配置你的应用时，你会潜在地使用这些类。但是作为一名开发者，你最可能对影响容器所提供的服务的其它模块感兴趣。这些模块将会为你提供用于构建应用服务的框架，例如AOP和持久性。

**核心容器**

这是Spring框架最基础的部分，它提供了依赖注入（Dependency Injection）特征来实现容器对Bean的管理。这里最基本的概念是BeanFactory，它是任何Spring应用的核心。BeanFactory是工厂模式的一个实现，它使用IoC将应用配置和依赖说明从实际的应用代码中分离出来。

**应用上下文（Context）模块**

核心模块的BeanFactory使Spring成为一个容器，而上下文模块使它成为一个框架。这个模块扩展了BeanFactory的概念，增加了对国际化（I18N）消息、事件传播以及验证的支持。

另外，这个模块提供了许多企业服务，例如电子邮件、JNDI访问、EJB集成、远程以及时序调度（scheduling）服务。也包括了对模版框架例如Velocity和FreeMarker集成的支持。

**Spring的AOP模块**

Spring在它的AOP模块中提供了对面向切面编程的丰富支持。这个模块是在Spring应用中实现切面编程的基础。为了确保Spring与其它AOP框架的互用性， Spring的AOP支持基于AOP联盟定义的API。AOP联盟是一个开源项目，它的目标是通过定义一组共同的接口和组件来促进AOP的使用以及不同的AOP实现之间的互用性。通过访问他们的站点http://aopalliance. sourceforge.net，你可以找到关于AOP联盟的更多内容。

Spring的AOP模块也将元数据编程引入了Spring。使用Spring的元数据支持，你可以为你的源代码增加注释，指示Spring在何处以及如何应用切面函数。

**JDBC抽象和DAO模块**

使用JDBC经常导致大量的重复代码，取得连接、创建语句、处理结果集，然后关闭连接。Spring的JDBC和DAO模块抽取了这些重复代码，因此你可以保持你的数据库访问代码干净简洁，并且可以防止因关闭数据库资源失败而引起的问题。

这个模块还在几种数据库服务器给出的错误消息之上建立了一个有意义的异常层。使你不用再试图破译神秘的私有的SQL错误消息！

另外，这个模块还使用了Spring的AOP模块为Spring应用中的对象提供了事务管理服务。

**对象/关系映射集成模块**

对那些更喜欢使用对象/关系映射工具而不是直接使用JDBC的人，Spring提供了ORM模块。Spring并不试图实现它自己的ORM解决方案，而是为几种流行的ORM框架提供了集成方案，包括Hibernate、JDO和iBATIS SQL映射。Spring的事务管理支持这些ORM框架中的每一个也包括JDBC。

**Spring的Web模块**

Web上下文模块建立于应用上下文模块之上，提供了一个适合于Web应用的上下文。另外，这个模块还提供了一些面向服务支持。例如：实现文件上传的multipart请求，它也提供了Spring和其它Web框架的集成，比如Struts、WebWork。

**Spring的MVC框架**

Spring为构建Web应用提供了一个功能全面的MVC框架。虽然Spring可以很容易地与其它MVC框架集成，例如Struts，但Spring的MVC框架使用IoC对控制逻辑和业务对象提供了完全的分离。

它也允许你声明性地将请求参数绑定到你的业务对象中，此外，Spring的MVC框架还可以利用Spring的任何其它服务，例如国际化信息与验证。

## 总结

Spring带来了复杂的J2EE开发的春天。它的核心是轻量级的IoC容器，它的目标是为J2EE应用提供了全方位的整合框架，在Spring框架下实现多个子框架的组合，这些子框架之间可以彼此独立，也可以使用其它的框架方案加以代替，Spring希望为企业应用提供一站式（one-stop shop）的解决方案。

# Spring的IoC容器

* **主要内容**：从最基本的面向接口编程逐步引入IoC设计模式（以银行卡：Card为例，接口－单例－工厂方法－IoC）；详细介绍IoC的三种实现，并对其优、缺点进行比较；之后开始引入Spring的IoC容器，详细介绍如何使用Spring的IoC容器组织业务组件。
* **目的**：使学员真正理解IoC的概念、优点，并掌握Spring IoC容器的使用。

## 用户注册的例子

我们先看看更进一步的需求：实现一个用户注册信息持久化的类。

功能：

1. 保存用户注册的信息；
2. 根据用户的名称获得该注册用户。

虽然功能简单，但它对持久化方式的要求却非常的灵活：

1. 在内存中持久化，供测试、演示使用。
2. 如果用户的数据很少，将用户信息持据化到文本文件中。
3. 如果用户信息很多，并需要一些灵活的查询，则需要使用JDBC技术将用将用户信息持久化到数据库中。
4. 面对企业复杂关联的数据，甚至需要使用持久层框架来实现用户信息的持久化，比如：iBATIS、Hibernate等。

如何去设计、实现我们这个持久化类呢？

我们遵循软件开发的原则“首先让它跑起来，再去优化（重构）它”，我们首先实现最简单的在内存中持久化用户信息。

既然我们要保存和取得用户信息，首先应该设计用户类。代码如下：

User.java

**public class User {**

**private Long id;**

**private String name;**

**private String password;**

**private String group;**

**public User(String name,String password){**

**this.name = name;**

**this.password = password;**

**}**

**//相应的get/set方法**

**………..**

**}**

持久化类有两个方法，分别在内存中保存和获取User对象。代码如下：

MemoryUserPersist.java

**public class MemoryUserPersist {**

**private static Map users = new HashMap();**

**static{**

**User defaultAdmin = new User("Moxie","pass");**

**users.put(defaultAdmin.getName(),defaultAdmin);**

**}**

**public MemoryUserPersist (){**

**}**

**public void saveUser(User user){**

**users.put(user.getName(),user);**

**}**

**public User LoadUser(String userName){**

**return (User)users.get(userName);**

**}**

**}**

用户持久化类完成之后，我们就可以在客户端UserRegister中使用它了。例如：用户注册时，UserRegister代码片断如下：

MemoryUserPersist userPersist = new MemoryUserPersist ();

userPersist.saveUser(user);

可是，现在如果要在文本文件中持久化User，又该如何实现呢？实现一个TextUserPersist类，这个并不困难。但客户端代码将面临重大灾难：找到所有使用过MemoryUserPersist的客户端类，将他们中的MemoryUserPersist逐个手工修改为 TextUserPersist，并且重新编译，当然以前的测试也必须全部从头来过！

人生的浩劫只是刚刚开始，因为根据前面的需求我们至少要分别实现四种持久化方式！这时，你一定和我一样在期待着救世主的早日降临——接口（Interface）。

## 面向接口编程

什么是接口？

* 接口定义了行为的协议，这些行为在继承接口的类中实现。
* 接口定义了很多方法，但是没有实现它们。类履行接口协议并实现所有定义在接口中的方法。
* 接口是一种只有声明没有实现的特殊类。

接口的优点：

* Client不必知道其使用对象的具体所属类。
* 一个对象可以很容易地被（实现了相同接口的）的另一个对象所替换。
* 对象间的连接不必硬绑定（hardwire）到一个具体类的对象上，因此增加了灵活性。
* 松散藕合（loosens coupling）。
* 增加了重用的可能性。

接口的缺点：

设计的复杂性略有增加

## （用户持久化类）重构第一步——面向接口编程

1. 设计用户持久化类的接口UserDao，代码如下：

public interface UserDao {

public void save(User user);

public User load(String name);

}

1. 具体的持久化来必须要继承UserDao接口，并实现它的所有方法。我们还是首先实现内存持久化的用户类：

public class MemoryUserDao implements UserDao{

private static Map users = new HashMap();;

static{

User user = new User("Moxie","pass");

users.put(user.getName(),user);

}

public void save(User user) {

users.put(user.getId(),user);

}

public User load(String name) {

return (User)users.get(name);

}

}

MemoryUserDao的实现代码和上面的MemoryUserPersist基本相同，唯一区别是MemoryUserDao类继承了UserDao接口，它的save()和load()方法是实现接口的方法。

这时，客户端UserRegister的代码又该如何实现呢？

UserDao userDao = new MemoryUserDao();

userDao.save(user);

（注：面向对象“多态”的阐述）

如果我们再切换到文本的持久化实现TextUserDao，客户端代码仍然需要手工修改。虽然我们已经使用了面向对象的多态技术，对象userDao方法的执行都是针对接口的调用，但userDao对象的创建却依赖于具体的实现类，比如上面MemoryUserDao。这样我们并没有完全实现前面所说的“Client不必知道其使用对象的具体所属类”。

如何解决客户端对象依赖具体实现类的问题呢？

下面该是我们的工厂（Factory）模式出场了！

## 重构第二步——工厂（Factory）模式

我们使用一个工厂类来实现userDao对象的创建，这样客户端只要知道这一个工厂类就可以了，不用依赖任何具体的UserDao实现。创建userDao对象的工厂类UserDaoFactory代码如下：

public class UserDaoFactory {

public static UserDao createUserDao(){

return new MemoryUserDao();

}

}

客户端UserRegister代码片断如下：

UserDao userDao = UserDaoFactory. CreateUserDao();

userDao.save(user);

现在如果再要更换持久化方式，比如使用文本文件持久化用户信息。就算有再多的客户代码调用了用户持久化对象我们都不用担心了。因为客户端和用户持久化对象的具体实现完全解耦。我们唯一要修改的只是一个UserDaoFactory类。

## 重构第三步——工厂（Factory）模式的改进

到这里人生的浩劫已经得到了拯救。但我们仍不满足，因为假如将内存持久化改为文本文件持久化仍然有着硬编码的存在——UserDaoFactory类的修改。代码的修改就意味着重新编译、打包、部署甚至引入新的Bug。所以，我们不满足，因为它还不够完美！

如何才是我们心目中的完美方案？至少要消除更换持久化方式时带来的硬编码。具体实现类的可配置不正是我们需要的吗？我们在一个属性文件中配置UserDao的实现类，例如：

在属性文件中可以这样配置：userDao = com.test.MemoryUserDao。UserDao的工厂类将从这个属性文件中取得UserDao实现类的全名，再通过Class.forName(className).newInstance()语句来自动创建一个UserDao接口的具体实例。UserDaoFactory代码如下：

public class UserDaoFactory {

public static UserDao createUserDao(){

String className = "";

// ……从属性文件中取得这个UserDao的实现类全名。

UserDao userDao = null;

try {

userDao = (UserDao)Class.forName(className).newInstance();

} catch (Exception e) {

e.printStackTrace();

}

return userDao;

}

通过对工厂模式的优化，我们的方案已近乎完美。如果现在要更换持久化方式，不需要再做任何的手工编码，只要修改配置文件中的userDao实现类名，将它设置为你需要更换的持久化类名即可。

我们终于可以松下一口气了？不，矛盾仍然存在。我们引入了接口，引入了工厂模式，让我们的系统高度的灵活和可配置，同时也给开发带来了一些复杂度：1、本来只有一个实现类，后来却要为这个实现类引入了一个接口。2、引入了一个接口，却还需要额外开发一个对应的工厂类。3、工厂类过多时，管理、维护非常困难。比如：当UserDao的实现类是JdbcUserDao，它使用JDBC技术来实现用户信息从持久化。也许要在取得JdbcUserDao实例时传入数据库Connection，这是仍少UserDaoFactory的硬编码。

当然，面接口编程是实现软件的可维护性和可重用行的重要原则已经勿庸置疑。这样，第一个复杂度问题是无法避免的，再说一个接口的开发和维护的工作量是微不足道的。但后面两个复杂度的问题，我们是完全可以解决的：工厂模式的终极方案——IoC模式。

## 重构第四步－IoC容器

使用IoC容器，用户注册类UserRegister不用主动创建UserDao实现类的实例。由IoC容器主动创建UserDao实现类的实例，并注入到用户注册类中。我们下面将使用Spring提供的IoC容器来管理我们的用户注册类。

用户注册类UserRegister的部分代码如下：

public class UserRegister {

private UserDao userDao = null;//由容器注入的实例对象

public void setUserDao(UserDao userDao){

this.userDao = userDao;

}

// UserRegister的业务方法

}

在其它的UserRegister方法中就可以直接使用userDao对象了，它的实例由Spring容器主动为它创建。但是，如何组装一个UserDao的实现类到UserRegister中呢？哦，Spring提供了配置文件来组装我们的组件。Spring的配置文件applicationContext.xml代码片断如下：

<bean id="userRegister" class="com.dev.spring.simple.UserRegister">

<property name="userDao"><ref local="userDao"/></property>

</bean>

<bean id="userDao" class="com.dev.spring.simple.MemoryUserDao"/>

## 控制反转（IoC）/依赖注入（DI）

### 什么是控制反转/依赖注入？

控制反转（IoC=Inversion of Control）IoC，用白话来讲，就是由容器控制程序之间的（依赖）关系，而非传统实现中，由程序代码直接操控。这也就是所谓“控制反转”的概念所在：（依赖）控制权由应用代码中转到了外部容器，控制权的转移，是所谓反转。

IoC也称为好莱坞原则（Hollywood Principle）：“Don’t call us, we’ll call you”。即，如果大腕明星想演节目，不用自己去找好莱坞公司，而是由好莱坞公司主动去找他们（当然，之前这些明星必须要在好莱坞登记过）。

正在业界为IoC争吵不休时，大师级人物Martin Fowler也站出来发话，以一篇经典文章《Inversion of Control Containers and the Dependency Injection pattern》为IoC正名，至此，IoC又获得了一个新的名字：“依赖注入 （Dependency Injection）”。

相对IoC 而言，“依赖注入”的确更加准确的描述了这种古老而又时兴的设计理念。从名字上理解，所谓依赖注入，即组件之间的依赖关系由容器在运行期决定，形象的来说，即由容器动态的将某种依赖关系注入到组件之中。

例如前面用户注册的例子。UserRegister依赖于UserDao的实现类，在最后的改进中我们使用IoC容器在运行期动态的为UserRegister注入UserDao的实现类。即UserRegister对UserDao的依赖关系由容器注入，UserRegister不用关心UserDao的任何具体实现类。如果要更改用户的持久化方式，只要修改配置文件applicationContext.xm即可。

依赖注入机制减轻了组件之间的依赖关系，同时也大大提高了组件的可移植性，这意味着，组件得到重用的机会将会更多。

### 依赖注入的三种实现形式

我们将组件的依赖关系由容器实现，那么容器如何知道一个组件依赖哪些其它的组件呢？例如用户注册的例子：容器如何得知UserRegister依赖于UserDao呢。这样，我们的组件必须提供一系列所谓的回调方法（这个方法并不是具体的Java类的方法），这些回调方法会告知容器它所依赖的组件。根据回调方法的不同，我们可以将IoC分为三种形式：

#### Type1－接口注入（Interface Injection）

它是在一个接口中定义需要注入的信息，并通过接口完成注入。Apache Avalon是一个较为典型的Type1型IOC容器，WebWork框架的IoC容器也是Type1型。

当然，使用接口注入我们首先要定义一个接口，组件的注入将通过这个接口进行。我们还是以用户注册为例，我们开发一个InjectUserDao接口，它的用途是将一个UserDao实例注入到实现该接口的类中。InjectUserDao接口代码如下：

public interface InjectUserDao {

public void setUserDao(UserDao userDao);

}

UserRegister需要容器为它注入一个UserDao的实例，则它必须实现InjectUserDao接口。UserRegister部分代码如下：

public class UserRegister implements InjectUserDao{

private UserDao userDao = null;//该对象实例由容器注入

public void setUserDao(UserDao userDao) {

this.userDao = userDao;

}

// UserRegister的其它业务方法

}

同时，我们需要配置InjectUserDao接口和UserDao的实现类。如果使用WebWork框架则配置文件如下：

<component>

<scope>request</scope>

<class>com.dev.spring.simple.MemoryUserDao</class>

<enabler>com.dev.spring.simple.InjectUserDao</enabler>

</component>

这样，当IoC容器判断出UserRegister组件实现了InjectUserDao接口时，它就将MemoryUserDao实例注入到UserRegister组件中。

#### Type2－设值方法注入（Setter Injection）

在各种类型的依赖注入模式中，设值注入模式在实际开发中得到了最广泛的应用（其中很大一部分得力于Spring框架的影响）。

基于设置模式的依赖注入机制更加直观、也更加自然。前面的用户注册示例，就是典

型的设置注入，即通过类的setter方法完成依赖关系的设置。

#### Type3－构造子注入（Constructor Injection）

构造子注入，即通过构造函数完成依赖关系的设定。将用户注册示例该为构造子注入，UserRegister代码如下：

public class UserRegister {

private UserDao userDao = null;//由容器通过构造函数注入的实例对象

public UserRegister(UserDao userDao){

this.userDao = userDao;

}

//业务方法

}

#### 几种依赖注入模式的对比总结

接口注入模式因为历史较为悠久，在很多容器中都已经得到应用。但由于其在灵活性、易用性上不如

其他两种注入模式，因而在IOC的专题世界内并不被看好。

Type2和Type3型的依赖注入实现则是目前主流的IOC实现模式。这两种实现方式各有特点，也各具优势。

Type2 设值注入的优势

1． 对于习惯了传统JavaBean开发的程序员而言，通过setter方法设定依赖关系显得更加直观，更加自然。

2． 如果依赖关系（或继承关系）较为复杂，那么Type3模式的构造函数也会相当庞大（我们需要在构造函数中设定所有依赖关系），此时Type2模式往往更为简洁。

3． 对于某些第三方类库而言，可能要求我们的组件必须提供一个默认的构造函数（如Struts中的Action），此时Type3类型的依赖注入机制就体现出其局限性，难以完成我们期望的功能。

Type3 构造子注入的优势：

1． “在构造期即创建一个完整、合法的对象”，对于这条Java设计原则，Type3无疑是最好的响应者。

2． 避免了繁琐的setter方法的编写，所有依赖关系均在构造函数中设定，依赖关系集中呈现，更加易读。

3． 由于没有setter方法，依赖关系在构造时由容器一次性设定，因此组件在被创建之后即处于相对“不变”的稳定状态，无需担心上层代码在调用过程中执行setter方法对组件依赖关系产生破坏，特别是对于Singleton模式的组件而言，这可能对整个系统产生重大的影响。

4． 同样，由于关联关系仅在构造函数中表达，只有组件创建者需要关心组件内部的依赖关系。对调用者而言，组件中的依赖关系处于黑盒之中。对上层屏蔽不必要的信息，也为系统的层次清晰性提供了保证。

5． 通过构造子注入，意味着我们可以在构造函数中决定依赖关系的注入顺序，对于一个大量依赖外部服务的组件而言，依赖关系的获得顺序可能非常重要，比如某个依赖关系注入的先决条件是组件的UserDao及相关资源已经被设定。

可见，Type3和Type2模式各有千秋，而Spring、PicoContainer都对Type3和Type2类型的依赖注入机制提供了良好支持。这也就为我们提供了更多的选择余地。理论上，以Type3类型为主，辅之以Type2类型机制作为补充，可以达到最好的依赖注入效果，不过对于基于Spring Framework开发的应用而言，Type2使用更加广泛。

## BeanFactory

BeanFactory是Spring的“心脏”。它就是Spring IoC容器的真面目。Spring使用BeanFactory来实例化、配置和管理Bean。但是，在大多数情况我们并不直接使用BeanFactory，而是使用ApplicationContext。它也是BeanFactory的一个实现，但是它添加了一系列“框架”的特征，比如：国际化支持、资源访问、事件传播等。ApplicationContext我们将在后面章节中介绍。

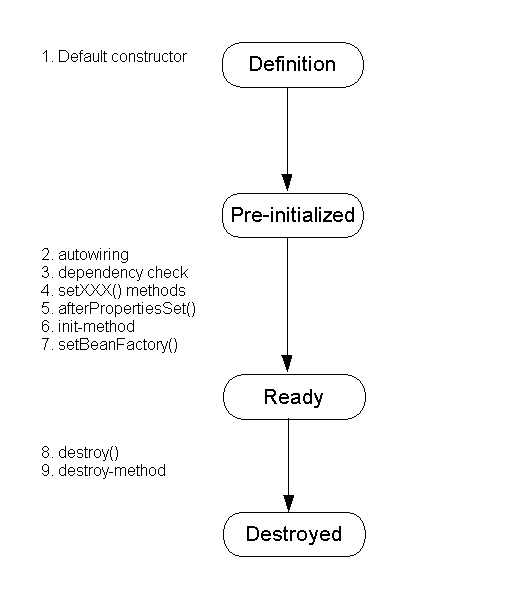
ApplicationContext

BeanFactory其实是一个接口－org.springframework.beans.factory.BeanFactory，它可以配置和管理几乎所有的Java类。当然，具体的工作是由实现BeanFactory接口的实现类完成。我们最常用的BeanFactory实现是org.springframework.beans.factory.xml.XmlBeanFactory。它从XML文件中读取Bean的定义信息。当BeanFactory被创建时，Spring验证每个Bean的配置。当然，要等Bean创建之后才能设置Bean的属性。单例(Singleton)Bean在启动时就会被BeanFactory实例化，其它的Bean在请求时创建。根据BeanFactory的Java文档（Javadocs）介绍，“Bean定义的持久化方式没有任何的限制：LDAP、RDBMS、XML、属性文件，等等”。现在Spring已提供了XML文件和属性文件的实现。无疑，XML文件是定义Bean的最佳方式。

BeanFactory是初始化Bean和调用它们生命周期方法的“吃苦耐劳者”。注意，BeanFactory只能管理单例（Singleton）Bean的生命周期。它不能管理原型(prototype,非单例)Bean的生命周期。这是因为原型Bean实例被创建之后便被传给了客户端,容器失去了对它们的引用。

### BeanFactory管理Bean（组件）的生命周期

下图描述了Bean的生命周期。它是由IoC容器控制。IoC容器定义Bean操作的规则，即Bean的定义（BeanDefinition）。Bean的定义包含了BeanFactory在创建Bean实例时需要的所有信息。BeanFactory首先通过构造函数创建一个Bean实例，之后它会执行Bean实例的一系列之前初始化动作，初始化结束Bean将进入准备就绪（ready）状态，这时应用程序就可以获取这些Bean实例了。最后，当你销毁单例（Singleton）Bean时，它会调用相应的销毁方法，结束Bean实例的生命周期。



（图－Bean的生命周期）

### Bean的定义

前面的用户注册的例子中，我们已经使用Spring定义了一个用户持久化类：

<bean id="userDao" class="com.dev.spring.simple.MemoryUserDao"/>

这是一个最简单的Bean定义。它类似于调用了语句：MemoryUserDao userDao = new MemoryUserDao()。

**id**属性必须是一个有效的XML ID,这意味着它在整个XML文档中必须唯一。它是一个Bean的“终身代号（9527）”。同时你也可以用name属性为Bean定义一个或多个别名（用逗号或空格分开多个别名）。name属性允许出现任意非法的XML字母。例如：

<bean id="userDao" name="userDao\*\_1, userDao\*\_2"

class="com.dev.spring.simple.MemoryUserDao"/>。

**class**属性定义了这个Bean的全限定类名（包名＋类名）。Spring能管理几乎所有的Java类。一般情况，这个Java类会有一个默认的构造函数，用set方法设置依赖的属性。

Bean元素出了上面的两个属性之外，还有很多其它属性。说明如下：

<bean

id="beanId"（1）

name="beanName"（2）

class="beanClass"（3）

parent="parentBean"（4）

abstract="true | false"（5）

singleton="true | false"（6）

lazy-init="true | false | default"（7）

autowire="no | byName | byType | constructor | autodetect | default"（8）

dependency-check = "none | objects | simple | all | default"（9）

depends-on="dependsOnBean"（10）

init-method="method"（11）

destroy-method="method"（12）

factory-method="method"（13）

factory-bean="bean">（14）

</bean>

（1）、id: Bean的唯一标识名。它必须是合法的XML ID，在整个XML文档中唯一。

（2）、name: 用来为id创建一个或多个别名。它可以是任意的字母符合。多个别名之间用逗号或空格分开。

（3）、class: 用来定义类的全限定名（包名＋类名）。只有子类Bean不用定义该属性。

（4）、parent: 子类Bean定义它所引用它的父类Bean。这时前面的class属性失效。子类Bean会继承父类Bean的所有属性，子类Bean也可以覆盖父类Bean的属性。注意：子类Bean和父类Bean是同一个Java类。

（5）、abstract（默认为”false”）：用来定义Bean是否为抽象Bean。它表示这个Bean将不会被实例化，一般用于父类Bean，因为父类Bean主要是供子类Bean继承使用。

（6）、singleton（默认为“true”）：定义Bean是否是Singleton（单例）。如果设为“true”,则在BeanFactory作用范围内，只维护此Bean的一个实例。如果设为“flase”，Bean将是Prototype（原型）状态，BeanFactory将为每次Bean请求创建一个新的Bean实例。

（7）、lazy-init（默认为“default”）：用来定义这个Bean是否实现懒初始化。如果为“true”，它将在BeanFactory启动时初始化所有的Singleton Bean。反之，如果为“false”,它只在Bean请求时才开始创建Singleton Bean。

（8）、autowire（自动装配，默认为“default”）：它定义了Bean的自动装载方式。

1、“no”:不使用自动装配功能。

2、“byName”:通过Bean的属性名实现自动装配。

3、“byType”:通过Bean的类型实现自动装配。

4、“constructor”:类似于byType，但它是用于构造函数的参数的自动组装。

5、“autodetect”:通过Bean类的反省机制（introspection）决定是使用“constructor”还是使用“byType”。

（9）、dependency-check（依赖检查，默认为“default”）：它用来确保Bean组件通过JavaBean描述的所以依赖关系都得到满足。在与自动装配功能一起使用时，它特别有用。

1. none：不进行依赖检查。
2. objects：只做对象间依赖的检查。
3. simple：只做原始类型和String类型依赖的检查
4. all：对所有类型的依赖进行检查。它包括了前面的objects和simple。

（10）、depends-on（依赖对象）：这个Bean在初始化时依赖的对象，这个对象会在这个Bean初始化之前创建。

（11）、init-method:用来定义Bean的初始化方法，它会在Bean组装之后调用。它必须是一个无参数的方法。

（12）、destroy-method：用来定义Bean的销毁方法，它在BeanFactory关闭时调用。同样，它也必须是一个无参数的方法。它只能应用于singleton Bean。

（13）、factory-method：定义创建该Bean对象的工厂方法。它用于下面的“factory-bean”，表示这个Bean是通过工厂方法创建。此时，“class”属性失效。

（14）、factory-bean:定义创建该Bean对象的工厂类。如果使用了“factory-bean”则“class”属性失效。

#### 配置Bean的属性值和Bean对象的组装

我们可以在Spring的配置文件中直接设置Bean的属性值。例如：你的Bean有一个“maxSize”属性，它表示每页显示数据的最大值，它有一个set方法。代码如下：

private int maxSize;

public void setMaxSize(int maxSize) {

this.maxSize = maxSize;

}

这样，你可以在Bean定义时设置这个属性的值：

<property name="maxSize"><value>20</value></property>

前面介绍了Bean原始类型的属性设置。这种方式已经可以非常有效而便利的参数化应用对象。然而，Bean工厂的真正威力在于：它可以根据bean属性中描述的对象依赖来组装（wire）bean实例。例如：userDao对象的一个属性“sessionFactory”引用了另外一个Bean对象，即userDao对象实例依赖于sessionFactory对象：

<bean id="userDao" class="com.dev.spring.simple.HibernateUserDao">

<property name="sessionFactory"><ref local="sessionFactory"/></property>

</bean>

<bean id="sessionFactory"

class="org.springframework.orm.hibernate.LocalSessionFactoryBean">

…..

</bean>

在这个简单的例子中，使用<ref>元素引用了一个sessionFactory实例。在ref标签中，我们使用了一个“local”属性指定它所引用的Bean对象。除了local属性之外，还有一些其它的属性可以用来指定引用对象。下面列出<ref>元素的所有可用的指定方式：

bean：可以在当前文件中查找依赖对象，也可以在应用上下文（ApplicationContext）中查找其它配置文件的对象。

local：只在当前文件中查找依赖对象。这个属性是一个XML IDREF，所以它指定的对象必须存在，否则它的验证检查会报错。

external：在其它文件中查找依赖对象，而不在当前文件中查找。

总的来说，<ref bean="..."/>和<ref local="..."/>大部分的时候可以通用。“bean”是最灵活的方式，它允许你在多个文件之间共享Bean。而“local”则提供了便利的XML验证。

#### 复杂的属性值

Spring的bean工厂不仅允许用String值和其他bean的引用作为bean组件的属性值，还支持更复杂的值，例如数组、java.util.List、java.util.Map和java.util.Properties。数组、set、list和map中的值不仅可以是String类型，也可以是其他bean的引用；map中的键、Properties的键和值都必须是String类型的；map中的值可以是set、list或者map类型 。

例如：

Null:

<property name=“bar”><null/></property>

List和数组：

<property name=“bar”>

<list>

<value>ABC</value>

<value>123</value>

</list>

</property>

Map:

<property name=“bar”>

<map>

<entry key=“key1”><value>ABC</value></entry>

<entry key=“key2”><value>123</value></entry>

</set>

</property>

### Bean的之前初始化

Bean工厂使用Bean的构造函数创建Bean对象之后，紧接着它会做一件非常重要的工作——Bean的初始化。它会根据配置信息设置Bean的属性和依赖对象，执行相应的初始化方法。

#### 自动装配

一般不推荐在大型的应用系统中使用自动装配。当然，它可以很好的用于小型应用系统。如果一个bean声明被标志为“autowire（自动装配）”，bean工厂会自动将其他的受管对象与其要求的依赖关系进行匹配，从而完成对象的装配——当然，只有当对象关系无歧义时才能完成自动装配。因为不需要明确指定某个协作对象，所以可以带来很多的便利性。

举个例子，如果在Bean工厂中有一个SessionFactory类型的实例，HibernateUserDao的sessionFactory属性就可以获得这个实例。这里可以使用<bean>的autowire属性，就象这样：

<bean id="userDao" class="com.dev.spring.simple.HibernateUserDao" autowire=”byType”>

</bean>

<bean id="sessionFactory"

class="org.springframework.orm.hibernate.LocalSessionFactoryBean">

…..

</bean>

注意，在userDao的定义中并没有明确引用sessionFactory。由于它的“autowire”被设置为“byType”，所有只要userDao有一个类型为SessionFactory的属性并有一个set方法，Bean工厂就会自动将sessionFactory组装到userDao中。

如果一个应用有两个数据库，这时就对应有两个SessionFactory。这时autowire="byType"就无法使用了。我们可以使用另外一种自动装配方式“byName”。它将根据属性的名称来匹配依赖对象，这样如果你的配置文件中可以同时存在多个类型相同的SessionFactory，只要他们定义的名称不同就可以了。这种方式的缺点是：需要精确匹配Bean的名称，即必须要保证属性的名称和它所依赖的Bean的名称相等，这样比较容易出错。

（举例：Aa对象依赖Bb对象。）

**注意：我们还是强烈推荐手工指定Bean之间的依赖关系。这种用法最强大，因为它允许按名称引用特定的Bean实例，即使多个Bean具有相同类型也不会混淆。同时，你可以清楚的知道一个Bean到底依赖哪些其它的Bean。如果使用自动装载，你只能去Bean的代码中了解。甚至，Bean工厂也许会自动装载一些你根本不想依赖的对象。**

#### 依赖检查

如果你希望Bean严格的设置所有的属性，“dependency-check”（依赖检查）属性将会非常有用。它默认为“none”，不进行依赖检查。“simple”会核对所有的原始类型和String类型的属性。“objects”只做对象间的关联检查（包括集合）。“all”会检查所有的属性，包括“simple”和“objects”。

举个例子：一个Bean有如下的一个属性：

private int intVar = 0;

public void setIntVar(int intVar) {

this.intVar = intVar;

}

这个Bean的配置文件设置了“dependency-check=”simple””。如果这个Bean的配置中没有定义这个属性“intVar”，则在进行这个Bean的依赖检查时就会抛出异常：org.springframework.beans.factory.UnsatisfiedDependencyException。

#### setXXX()

set方法非常简单，它会给class注入所有依赖的属性。这些属性都必须是在配置文件中使用<property>元素定义，它们可以是原始类型，对象类型（Integer,Long），null值，集合，其它对象的引用。

#### afterPropertiesSet()

有两种方法可以实现Bean的之前初始化方法。1、使用“init-method”属性，在Spring的配置文件中定义回调方法。下面将会具体描述。2、实现接口InitializingBean并实现它的afterPropertiesSet()方法。接口InitializingBean的代码如下：

public interface InitializingBean {

void afterPropertiesSet() throws Exception;

}

在JavaBean的所有属性设置完成以后，容器会调用afterPropertiesSet()方法，应用对象可以在这里执行任何定制的初始化操作。这个方法允许抛出最基本的Exception异常，这样可以简化编程模型。

在Spring框架内部，很多bean组件都实现了这些回调接口。但我们的Bean组件最好不要通过这种方式实现生命周期的回调，因为它依赖于Spring的API。无疑，第一种方法是我们的最佳选择。

#### init-method

init-method的功能和InitializingBean接口一样。它定义了一个Bean的初始化方法，在Bean的所有属性设置完成之后自动调用。这个初始化方法不用依赖于Spring的任何API。它必须是一个无参数的方法，可以抛出Exception。

例如：我们的Bean组件UserManger中定义一个初始化方法init()。这样，我们就可以在Bean定义时指定这个初始化方法：

<bean id=”userManger” class=”com.dev.spring.um.DefaultUserManager”

init-method=”init”>

……

</bean>

#### setBeanFactory()

### Bean的准备就绪（Ready）状态

Bean完成所有的之前初始化之后，就进入了准备就绪(Ready)状态。这就意味着你的应用程序可以取得这些Bean，并根据需要使用他们。

## Bean的销毁

在你关闭（或重启）应用程序时，单例（Singleton）Bean可以再次获得生命周期的回调，你可以在这时销毁Bean的一些资源。第一种方法是实现DisposableBean接口并实现它的destroy()方法。更好的方法是用“destroy-method”在Bean的定义时指定销毁方法。

## ApplicationContext

# Spring的AOP框架

* **主要内容**：首先介绍什么是AOP编程，为什么要使用AOP编程（以用户登陆验证为例）；当前AOP框架概述、比较；Spring AOP框架的原理，如何使用Spring的AOP框架进行AOP编程？
* **目的**：使学员正确理解AOP的概念、优点，并能使用Spring的AOP框架进行AOP编程。

# Spring的数据层访问

# Spring的声明式事务

# Spring对其它企业应用支持