Spring-IoC容器

极客学院出版

前言

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（所谓 IoC 就是一个用 XML 来定义生成对象的模式）。Spring 的模块化是很强的，各个功能模块都是独立的，因为大多数应用程序都是由两个或是更多的类通过彼此的合作来实现业务逻辑，这使得每个对象都需要获取与其合作的对象（也就是它所依赖的对象）的引用。如果这个获取过程要靠自身实现，那么这将导致代码高度耦合并且难以维护和调试，本教程将通过示例代码，像你展示控制反转及依赖注入的实现方法及 Spring BeanFactory 实例化 Bean 的过程。

参考技术文档地址：http://www.cnblogs.com/linzheng/archive/2011/01/03/1925052.html

Spring-IoC 编程示例地址：http://www.cnblogs.com/linzheng/archive/2011/01/03/1925062.html

适合人群

本教程为 Java 中高级人员准备，可以帮助你更好的理解 Spring 框架里的 IoC 容器的实现，帮助你在采用 Spring 框架做开发时解决程序模块间高耦合的问题。

学习条件

对于本教程，我们假定你已经对 Java 面向对象编程思想有了深刻的了解，对 Spring 框架及 XML 有了一定接触。

目录

1 设计用户持久化类

在介绍面向接口编程前，我们先来了解一下 IoC的基本原理，所谓 IoC，对于 Spring 框架来说，就是由 Spring 来负责控制对象的生命周期和对象间的关系。在一个对象中，如果要使用另外的对象，就必须得到它（自己 new 一个，或者从 JNDI 中查询一个），使用完之后还要将对象销毁（比如 Connection 等）。那么 IoC 是如何做的呢？ Spring 所倡导的开发方式就是如此，所有的类都会在 Spring 容器中登记，告诉 Spring 你是个什 么东西，你需要什么东西，然后 Spring 会在系统运行到适当的时候，把你要的东西主动给你，同时也把你交给其他需要你的东西。所有的类的创建、销毁都由 Spring 来控制，也就是说控制对象生存周期的不再是引用它的对象，而是 Spring 。对于某个具体的对象而言，以前是它控制其他对象，现在是所有对象都被 Spring 控制，所以这叫控制反转。

在应用程序中，用来实现业务问题实体的（如，在电子商务应用程序中的 Customer 和 Order ）类就是持久化类。不能认为所有的持久化类的实例都是持久的状态 —— 一个实例的状态也可能是瞬时的或脱管的。

下面开始设计一个用户持久化类，然后一步步学习如何利用 IoC 容器来处理用户持久化类的。

首先，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);

}
```

具体的持久化类必须要继承 `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!

2 工厂模式

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。

3

工厂模式改进

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。

4

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"/>
```

5

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.xml 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。

6

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：

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 是一个较为典型的接口注入型 IOC 容器，WebWork 框架的 IoC 容器也是接口注入型。

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 InjectUserDad{

   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。

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。

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
}
```

7

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。

设值方法注入和构造子注入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。

设值注入的优势

- 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。
- 如果依赖关系（或继承关系）较为复杂，那么构造子注入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），此时设值方法注入模式往往更为简洁。
- 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），此时构造子注入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。

构造子注入的优势

- “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，构造子注入无疑是最好的响应者。
- 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。
- 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。
- 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。
- 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，构造子注入和设值注入模式各有千秋，而 Spring、PicoContainer 都对构造子注入和设值注入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，以构造子注入类型为主，辅之以设值注入类型机制作为补充，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，设值注入使用更加广泛。

8

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。

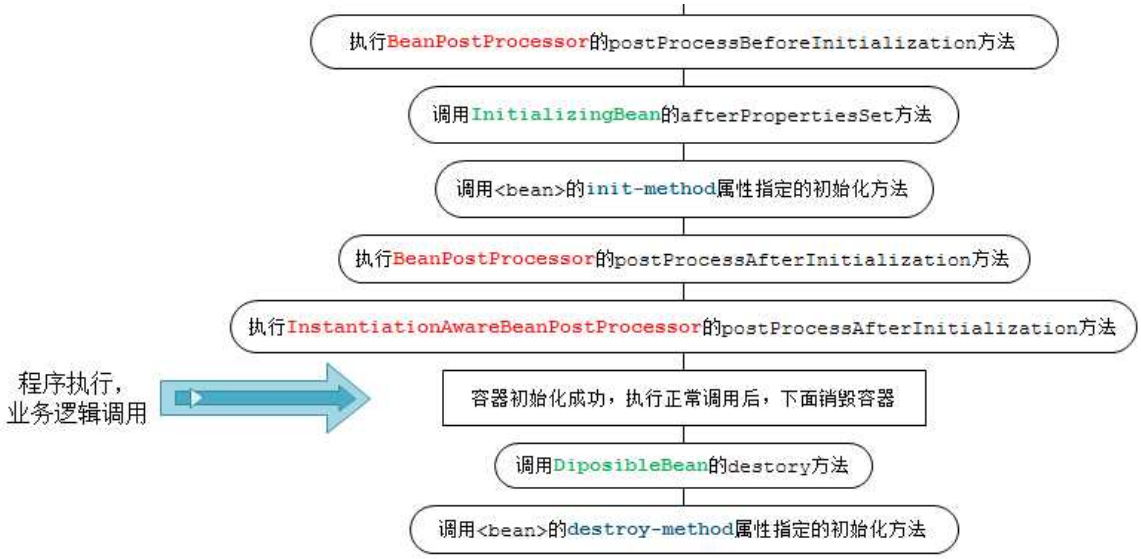
BeanFactory 其实是一个接口－org.springframework.beans.factory.BeanFactory，它可以配置和管理几乎所有的 Java 类。当然，具体的工作是由实现 BeanFactory 接口的实现类完成。我们最常用的 BeanFactory 实现是 org.springframework.beans.factory.xml.XmlBeanFactory。它从 XML 文件中读取 Bean 的定义信息。当 BeanFactory 被创建时，Spring 验证每个 Bean 的配置。当然，要等 Bean 创建之后才能设置 Bean 的属性。单例(Singleton)Bean 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。

9

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。



图片 9.1 图片描述性文字

10

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。

- “no”：不使用自动装配功能。
- “byName”：通过 Bean 的属性名实现自动装配。
- “byType”：通过 Bean 的类型实现自动装配。
- “constructor”：类似于 byType，但它是用于构造函数的参数的自动组装。
- “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。

- none：不进行依赖检查。
- objects：只做对象间依赖的检查。
- simple：只做原始类型和 String 类型依赖的检查
- 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。

11

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，和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。

12

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>
```

13

Bean 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“byTyp e”，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 ses sionFactory 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 Be an 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。

- 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。
- 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>
```

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。

14

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。

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台

更多信息请访问

http://wiki.jikexueyuan.com/project/spring-ioc/