基于Spring框架的IoC微内核的实现机制与应用

谌桂枝[1]，沈晓建[2]，龚兴艳[2]

（1.株洲职业技术学院，湖南 株洲 412001；2.湖南工业大学，湖南 株洲 412008）

摘 要：通过模拟实现Spring的IoC容器服务，揭示了容器的微内核实现机制，利用反射原理，编写且分析了Spring用依赖注入（Dependency Injection）实现IoC服务的过程。

关键词：控制反转（IoC）；依赖注入（DI）；解耦；高内聚低耦合；Spring容器

中图分类号：TP311.52 **文献标志码**：A **文章编号**：1673-9833(2009)03-0050-04

Investigation on the Realization Mechanism and Application of Spring Framework-Based IoC Micro-Core

Shen Guizhi[1]，Shen Xiaojian[2]，Gong Xingyan[2]

（1. Vocational Technical College of Zhuzhou，Zhuzhou Hunan 412001，China；
2. Hunan University of Technology，Zhuzhou Hunan 412008，China）

Abstract：The realization mechanism for the micro-core of IoC containers was revealed by simulation. By means of reflection principles, the IoC service realization process using dependency injection in Spring was compiled and analyzed.

Keywords：inversion of control（IoC）；dependency injection（DI）；decoupling；highly cohesive and low coupling；Spring container

传统的J2EE软件开发过程中，由于层与层间高度耦合，不仅使得软件开发项目复杂，而且给后续维护带来了困难。对于一些软件开发项目，即使经过反复修改，也只有永远的Beta版。采用基于Spring框架的IoC微内核，能使修改工作变得更轻松、更简单。

高度耦合会带来很多问题，如代码1：

```
class Vehicle{
    public void fly( ){System.out.println("Vehicle  fly ");}
}
class Boy{
    public void fly( ){Vehicle v=new Vehicle ( ); v.fly( ); }
}
public class Test{
    public static void main(String[] args){
        Boy b=new Boy( );     b.fly( );
  }
}
```

可以看出，通过“new”创建一个Boy的实例时也必须创建一个Vehicle，即若Vehicle不存在或者出现问题，则Boy无法实例化，即Boy对Vehicle产生了一种依赖，这种依赖就是耦合。编程的要求是“高内聚低耦合”，显然，程序1不符合高内聚低耦合要求。

高耦合一般有2种解决方案，方案1：使用接口，接口和实现相互分离的设计模式使得程序易于修改，不易产生“水波效应”，是控制依赖关系的有效方式。如代码2——Vehicle实现接口Flyable

```
Public class Boy{
  Private Flyable flyable;
  Public Boy( ){flyable=new Vehicle( );}
  Public void fly( ){flyable.fly( );}
}
```

使用接口虽然可以更换Vehicle，但还是要自己创建Vehicle对象，所以，接口和实现相互分离的设计模

收稿日期：2009-04-20

作者简介：谌桂枝（1970-），女，湖南株洲人，株洲职业技术学院教师，在职硕士研究生，主要从事J2EE软件方面的研究，E-mail：zzsgz@163.com

式虽然能控制依赖关系，但创建具体类的对象时仍会造成对具体实现的依赖，方案 2 便是使用 Spring 框架。Spring 框架提供了 IoC 服务，代码 1 中 Boy 建立自己的 Vehicle，IoC 容器将组件间的依赖关系提取出来，使用了更灵活的办法，将 Vehicle 类作为插件，只要插件遵循一定规则，一个独立的组装模块就能将插件的具体实现“注射”到应用程序中，设计好的类直接由系统控制中的容器来具体配置，即 Vehicle 类对象的产生反转到容器中，不由 Boy类来创建。可见，采用基于Spring 框架的 IoC 服务可更理想地解决高耦合度问题。

1　DI 实现 IoC 服务

IoC 是个很大的概念，就是控制权反转，可用不同的方式来实现[1]。其主要实现方式有 2 种：1）依赖查找（Dependency Lookup）——容器提供回调接口和上下文环境给组件，EJB 和 Apache Avalon 都使用这种方式；2）依赖注入（DI）——组件不做定位查询，只提供普通的 Java 方法让容器去决定依赖关系。后者是时下最流行的 IoC 类型，其又有接口注入（Interface Injection），设值注入（Setter Injection）和构造子注入（Constructor Injection）3 种方式[2]。

Spring 框架通过属性注入，实现了控制反转（IoC），反转的是“如何定位插件的具体实现”[3]，将这种关系的实现变得隐晦同时又很容易实现、易于扩展，如代码 3 ——源代码 Vehicle.java 源代码 Boy.java

```
package org.inject.IoC;
public class Vehicle {
    public void fly( ){System.out.println("Vehicle ");}
}
package org.inject.IoC;
public class Boy{
    public void fly( ){
        Vehicle vehicle;
        public void setVehicle(Vehicle vehicle)
            {this. vehicle = vehicle;}
        vehicle.fly( );
}
}
```

代码 3 定义了 Vehicle 类的对象 vehicle，同时为其提供了 setter 方法 setVehicle，代码中并未指定 vehicle 的值。Spring 框架中 vehicle 对象的创建过程可以反转到容器中，由 Spring 容器完成。

代码 4 是 Spring 的配置文件 applicationContext.xml 文件，在这个配置文件中，可以设定 Boy 和其它类之间的依赖关系。Vehicle 类已经成了 Boy 类的一个属性结点，如代码 4 —— applicationContext.xml

```
<bean id="v"class="org.inject.IoC.Vehicle">
</bean>
<bean id="boy"class="org.inject.IoC.Boy">
    <property name="vehicle">
        <ref bean="v">
    </property>
</bean>
```

文件中定义了一个 id 为 v 的节点，v 为 org. inject. IoC.Vehicle 的对象，定义一个 boy 节点，该节点依赖于 v 对象，对应的在 Boy 类中必须有一个 vehicle 的属性，并提供 setter 方法。每个 bean 的 id 属性都是该 bean 的唯一标识，程序通过 id 访问 bean，bean 与 bean 的依赖关系也通过 id 属性完成。

通过以上分析可知，依赖关系的控制已经被反转到了 Spring 的容器中，创建 Vehicle 的工作不再由 Boy 来完成，而是交给了 Spring 容器来完成。Sping 使用 DI 中设值注入方式实现 IoC，让类与类之间以配置文件的方式组织在一起，若需要取消模块中的某个功能，只需要从配置文件中取消某个节点即可，无须修改应用代码，就可以将该功能直接从系统中“卸载”，而不会对其它类产生影响，从而实现了解耦。

2　Spring 容器功能的模仿实现

Sping 使用 DI 中设值注入方式实现 IoC，从而实现了解耦。下面用反射原理模拟 Spring 用 DI 实现 IoC 服务的过程。

以代码 5 为例。代码 5 中的 xml 数据格式描述了一个层次结构数据集，该配置文件中 class Address 成为了 class Dept 的一个节点，而 class Dept 成为了 class User 的一个节点，即 class Address 和 class Dept 成为插件，可插拔，在程序运行时动态注入，改变了传统意义上的互相依赖。

代码 5 ——配置文件 applicationContext.xml

```
…  <bean id="user1" class="org.aaa.well.bean.User"
singleton="true">
        <property name="uid" value="135"/>
        <property name="age" value="25"/>
        <property name="dept" ref="dept2" />
    </bean>
        <bean id="dept2" class="org.aaa.well.bean.Dept" >
        <property name="deptname" value=" 开发部 " />
        <property name="deptid" value="0010"/>
        <property name="address" ref ="address"/>
    </bean>
<bean id="address" class="org.aaa.well.bean.Address">
        <property name="addname" value=" 华人街 "/>
    </bean>
```

配置文件中涉及到 3 个 JavaBean（Address.java、Dept.java、User.java），再增加 2 个 JavaBean，1 个 Bean（属性有 String id，String cls，boolean singleton，List<

Property> list），还有1个是Property类（属性有String name，String value，String ref），分别对应配置文件中的bean标签和property标签。

为读取配置文件，写一个解析类ReadContext，在public static Map<String，Bean> analyse( )方法中，读每个bean的属性值以及每个bean所对应的class的property值，代码（代码6）如下：

```
Element e = doc.getRootElement( );
    List<Element> list1 = e.getChildren( );
    for(Element e1:list1) {
        Bean bean = new Bean( );
        bean.setId(e1.getAttributeValue("id"));
        bean.setCls(e1.getAttributeValue("class"));
bean.setSingleton(e1.getAttributeValue("singleton")==null||e1.
getAttributeValue("singleton").equals("true"));
        List<Element> list2 = e1.getChildren( );
        for(Element e2:list2) {
            Property p = new Property( );
            p.setName(e2.getAttributeValue("name"));
            String value = e2.getAttributeValue("value");
            if(value==null && e2.getChild("value")!=null) {
                value = e2.getChild("value").getText( );
            }
            p.setValue(value);
            String ref = e2.getAttributeValue("ref");
            if(ref==null && e2.getChild("ref")!=null) {
                ref = e2.getChild("ref").getAttribute
Value("local");
            }
            p.setRef(ref);
            bean.addProperty(p);
            }
        map.put(bean.getId(), bean);
        }
        return map;
```

接下来是写一个IoC容器与工厂类Application Context。在这个类中先调用ReadContext类（代码6）中的静态方法，得到dom结点树上的结点，放入一个Map对象中，如代码7：

```
Map xml = ReadContext.analyse( );
```

再写一个jnjects（String beanname）方法，这个方法只要传入一个bean的名字就可以取出每个bean的属性值，如代码8：

```
List<Property> list = bean.getList( );
if(list==null || list.size( )<=0) return instance;
for(Property p : list) {
    String name=p.getName( );
    String value=p.getValue( );
    String ref = p.getRef( );
    String methodname = "set"+name.substring(0,1).
toUpperCase( )+name.substring(1);
    }
```

通过反射可以获得每个bean的实例，如代码9：

```
Object instance = Class.forName(bean.getCls( )).newInstance( );
```

一个bean对应一个class，通过以上步骤可以分别得到Address.java、Dept.java、User.java3个类的实例instance，并通过代码8的第7行指明methodname是set方法，还能得到每个bean的property值value。

下面是利用反射原理，动态执行setter方法，它需要将instance、method name、value作为参数。Class类、Method类是与反射有关的类，先得到类的Class实例以及其中的方法名，如代码10：

```
Class cls = instance.getClass( ); // 得到 Class 实例
Method[ ] methods = cls.getDeclaredMethods( );
Method m2 = null;
for(Method m : methods) {
  if(m.getName( ).equals(methodname)) { m2 = m; break;}
}
```

然后在invoke方法中动态执行setter方法，如代码11：

```
m2.invoke(instance, translate(value, m2.getParameterTypes(
)[0]));
```

其中translate是一个方法，它获得动态执行方法的参数列表。Object translate(Object value, Class cls)方法如代码12：

```
if(cls.getName( ).equals("int") || cls.getName( ).equals
("Integer")) {
    return Integer.valueOf(value.toString( ));
}
return value;
```

IoC容器与工厂类运行完毕后，获得的实例对象就注入了值。编写一个测试类，就能用getter得到属性值，如代码13：

```
ApplicationContext ac = new ApplicationContext( );
User u = ac.getBean("user1");
System.out.print(" 工号："+u.getUid( ));
System.out.print(" 年龄："+u.getAge( ));
System.out.print" 部门：" +(u.getDept( ).getDeptname( ));
System.out.print(" 部门地址 "+u.getDept( ).getAddress( ).
getAddname( ));
```

最后成功输出：工号——135；年龄——25；部门——开发部；部门地址——华人街。

以上模拟Sping使用DI中设值注入方式实现IoC，减少了类与类的耦合。DI编程模式遵循好莱坞法则：don't call us—we'll call you[4]。应用的程序编写中可理解为：“你不必创建自己的对象，而只需描述该对象如何被创建；你不必实例化或直接定位你的组件需要的服务，而只需确定哪些服务为哪些组件所需要，然后由其它程序（通常是一个容器）负责把它们‘钩’到一起。”即控制权由应用代码中转到了外部容器。结果

表明，这样的机制使软件实现可插拔，降低了代码耦合度，这就是 Sping 框架实现 IoC 模式的优点。

3　Spring 的 IoC 在软件开发中应用

合理使用设计模式、反射或是 Spring 的 IoC 将使开发程序变得简单[5]。对业务层直接通过“new”创建出 DAO 层的对象，如果变动 DAO，将直接导致业务层和表现层的失效，这种类型的代码的耦合度极高[4]。

Spring 的 IoC 在实际软件开发中应用较广。市场上流行的 DBMS 产品较多，如 SQLServer、Oracle Use 等。作为上层类并不知道数据来源，只要运用面向接口编程，再运用依赖注入就很易实现[5]。IUserDao 作为抽象类，其下可存在 DataFileUserDao、DBUserDao、XMLUserDao，分别代表XML文件、DATA文件和DBMS产品 3 种数据来源。

DataFileUserDao、DBUserDao、XMLUserDao这3种数据来源在applicationContext.xml文件中的层次结构如代码 14：

```
<!-- 数据库 存取方式 -->
    <bean id="dbUserDao"
        class="test.imp.DBUserDao"/>
<!-- 二进制文件 存取方式 -->
    <bean id="datUserDao"
        class="test.imp.DataFileUserDao"/>
<!-- XML 文件 存取方式 -->
    <bean id="xmlUserDao"
        class="test.imp.XMLUserDao"/>
<!-- User 模块代理 -->
    <bean id="userFacade" class="test.poxy.UserFacade">
    <!--<property name="impUserDao" ref="datUserDao"/> -->
      <property name="impUserDao" ref="datUserDao"/>
    <!-- <property name="impUserDao" ref="xmlUserDao"/> -->
    </bean>
```

可以看出，使用哪个数据库，实现类是最后动态注入的，只需在application Content.xml配置文件中选择节点即可。实现 DAO 层有变化并不影响业务层和表现层，这就是轻量级框架的特点。

在以上实例中，User Facade 代理类持有接口 IUserDao，IuserDao，存在接口实现类 DataFileUserDao，XMLUserDao，DBUserDao，用户可根据实际需求扩展自己需要的类。实际只需要实现 IUserDao 接口，通过 Spring 注入到 User Facade 代理类中，并且不需要更改任何现有代码，这便是 IoC 依赖注入带来的好处之一，事实上此处也满足了软件 ocp 开闭原则。

4　结论

Spring 使用 DI 中设值注入方式实现 IoC，使类的依赖关系从代码中脱离开来，通过相应配置文件，适时将类的实例注入到依赖类中，而依赖类事先并不知道最终所使用到的类，从而实现解耦，为软件适应不同需求提供便利性和可扩展性，是构建高可靠性的应用软件基础原则之一。好的 IoC 框架将声明式地（通过一个 XML 配置文件），而不是编程式地（这种方式的可靠性较差）串联起应用程序之间的相互依赖性。

参考文献

[1] 薄　奇，许林英. Spring 框架中 IoC 的实现[J]. 微处理机，2008(29)：147-149.
Bo Qi，Xu Linying. The Implement of IoC in Spring Framework[J]. Microprocessors，2008(29)：147-149.

[2] 娄　锋，孙　涌. 轻量级IoC容器的研究与设计[J]. 计算机技术与发展，2007(1)：91-93.
Lou Feng，Sun Yong. Design and Implementation of Lightweight IoC Container[J]. Computer Technology and Development，2007(1)：91-93.

[3] Martin Fowler. IoC 容器和 Dependency Injection 模式[J]. 程序员，2004(3)：89-96.
Martin Fowler. IoC Container and Dependency Injection Pattern[J]. Programmer，2004(3)：89-96.

[4] 魏学松，张育平. IoC框架的研究与设计[J]. 计算机技术与发展，2006(3)：213-216.
Wei Xuesong，Zhang Yuping. Research and Realization of IoC Framework[J]. Computer Technology and Development，2006(3)：213-216.

[5] 苏　琨. 基于IoC模式的软件开发框架重构[J]. 山西电子技术，2007(2)：82-83，94.
Su Kun. Reconstruction of Software Development Frame-Work Based on IoC Pattern[J]. Shanxi Electronic Technology，2007(2)：82-83，94.

（责任编辑：张亦静）