

控制反转技术分析

仲红艳

(上海师范大学 数理信息学院, 上海 200234)

摘 要:轻量级容器的解耦模式被称为“控制反转”或者“依赖注入”,组件之间的依赖关系由容器(运行环境)在运行期决定,从而在相当程度上降低了组件之间的耦合。该文详细论述了这种解耦模式的原理,以及依赖注入的3种主要形式,并且对几种形式进行了对比总结。

关键词:轻量级容器;控制反转;依赖注入

中图分类号:TP311.5

文献标识码:A

文章编号:1673-629X(2006)11-0059-03

An Analysis of Inversion of Control

ZHONG Hong-yan

(School of Science & Information, Shanghai Normal Univ., Shanghai 200234, China)

Abstract: The decoupled module of lightweight containers is called “inversion of control” or “dependency injection pattern”. The dependency of components is decided at runtime by the container(runtime environment). So the coupling of components is relieved to a great extent. The paper first expatiates the theorem of the decoupled module, then describes three main styles of dependency injection. Besides, the paper compares and summarizes these styles.

Key words: lightweight containers; inversion of control; dependency injection pattern

0 引言

在Java世界里,“轻量级容器”越来越受到广泛关注。轻量级容器能够帮助开发者将来自不同项目的组件组装成为一个内聚的应用程序。它们有着同一个模式,这个模式决定了这些容器进行组件装配的方式。通常这个模式被称之为“控制反转”(IoC)或者“依赖注入”^[1]。

轻量级容器可以定义更为细粒度的组件,甚至这个组件只有一个对象。以依赖注入为代表的解耦模式,可以让组件不去依赖容器的API。轻量级容器通过反转控制让容器具有主动权,去管理插进来的组件。只要组件是符合标准的,就可以被轻量级容器管理。

轻量级容器已经得到了广泛的认可,作为下一代J2EE构架的基础有着无法比拟的优势。

1 控制反转原理

轻量级容器中目前最引人注目的就是名为控制反转或者依赖注入的设计思想。

IoC,就是由容器控制程序之间的关系,而非传统实现中由程序代码直接操控。控制权由应用代码中转到了外部容器,控制权的转移,称为反转。相对IoC而言,“依赖注入”更加准确地描述了这种设计理念。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,形象地来说,即由容器动态地将某种依赖关系注入到组件之中。

依赖注入机制,可以在运行期为组件配置所需资源,而无需在编写组件代码时就加以指定,从而在相当程度上降低了组件之间的耦合。依赖注入的目标并非为软件系统带来更多的功能,而是为了提升组件重用的概率,并为系统搭建一个灵活、可扩展的平台。

在J2EE应用开发中,经常遇到的问题就是:如何将不同的组件组装成为一个内聚的应用程序?IoC模式可以解决这个问题,其目标是将组件的配置与使用分离开。所有的组件都是被动的,组件初始化和调用都由容器负责。组件处在一个容器当中,由容器负责管理。

用下面的代码来说明IoC模式。

```
class ClassA...  
public String aMethod(String arg)  
{  
    String result = instanceOfClassB.bMethod();  
    return result;  
}
```

上边的代码里,要解决的问题是:ClassA如何获得ClassB的实例?一个最直接的方法是给instanceOfClassB定义一个接口。

```
public interface IClassB  
{  
    String bMethod();  
}
```

收稿日期:2006-02-23

作者简介:仲红艳(1983-),女,江苏扬中人,硕士研究生,主要研究方向为虚拟现实。

在 aMethod 里声明:

```
private IClassB instanceOfClassB;
```

当执行 bMethod() 方法时,就要用到 IClassB 的某个具体实例。可见仍然需要通过某种途径获得合适的 I-ClassB 实现类的实例。ClassA 类既依赖于 IClassB 接口,也依赖于具体的实现类。当然希望 ClassA 类只依赖于接口,但要如何获得一个 IClassB 子类的实例,使得 ClassA 类能够与 IClassB 的任何实现类配合工作,并且允许在运行期插入具体的实现类。这里的问题就是:如何设计这个连接过程,使 ClassA 类在不知道实现类细节的前提下与其实例协同工作。

上述代码增加 ClassA 和 ClassB 的耦合度,以致于无法在不修改 ClassA 的情况下变更 IClassB 的具体实现^[2]。IoC 模式用于解决这样的问题。如前所述, IoC 容器负责初始化组件(如 IClassB),并将实例交给使用者。使用代码或配置文件以声明的方式将接口与实例关联起来, IoC 容器负责进行实际的调用处理。对于调用者,只需要关注接口就行了。这样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和重用性。

将这个例子推而广之,在一个真实的系统中,可能有数十个服务和组件。在任何时候,总可以对使用组件的情形加以抽象,通过接口与具体的组件交流。但是,如果希望以不同的方式部署这个系统,就需要用插件机制来处理服务之间的交互过程,这样才可能在不同的部署方案中使用不同的实现。

控制反转模式是框架系统的基础,任何框架系统都离不开控制反转模式^[2]。问题的关键在于:究竟反转了哪方面的控制?对于新生的轻量级容器,反转的是“如何定位插件的具体实现”。在前面的例子里,ClassA 类负责定位 IClassB 的具体实现。如果它直接实例化 IClassB 的一个子类, IClassB 就不称其为一个插件了,因为它并不是在运行期插入应用程序的。而这些轻量级的容器则使用了更为灵活的办法,只要插件遵循一定的规则,一个独立的组装模块就能够将插件的具体实现“注入”到应用程序中。

2 依赖注入的几种形式

依赖注入模式的基本思想是:用一个单独的对象来获得 IClassB 的一个合适的实现,并将其实例赋给 ClassA 类的一个字段。依赖注入的形式主要有 3 种,分别为接口注入、设值方法注入和构造子注入^[3]。

2.1 接口注入

需要定义一个接口,组件的注入将通过这个接口进行。在本例中,这个接口的用途是将一个 IClassB 实例注入继承该接口的对象。

```
public interface GetClassB
```

```
{
```

```
void getClassB(IClassB instanceOfClassB);
```

这个接口应该由提供 IClassB 接口的人一并提供。任何想要使用 IClassB 实例的类(例如 ClassA 类)都必须实现这个接口。

```
class ClassA implements GetClassB...
```

```
IClassB instanceOfClassB;
```

```
void getClassB(IClassB instanceOfClassB)
```

```
{
```

```
    this.instanceOfClassB = instanceOfClassB;
```

```
}
```

对于一个接口注入型 IoC 容器而言,加载接口实现并创建其实例的工作由容器完成。接口注入型 IoC 发展较早,在实际中得到了普遍应用。Apache Avalon 是一个较为典型的 IoC 容器。

2.2 设值注入

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广泛的应用。典型的设值注入,即通过类的 setter 方法完成依赖关系的设置,这种注入机制更加直观,也更加自然。

为了让 ClassA 类接受注入,需要为它定义一个设值方法,该方法接受类型为 IClassB 的参数:

```
class ClassA...
```

```
IClassB instanceOfClassB;
```

```
public void setFinder(IClassB instanceOfClassB)
```

```
{
```

```
    this.instanceOfClassB = instanceOfClassB;
```

```
}
```

类似地,在运行期 IClassB 的实例由容器提供。设值注入型 IoC 容器的典型代表是 Spring Framework。

2.3 构造子注入

构造子注入,即通过构造函数完成依赖关系的设定。在这种类型的依赖注入机制中,依赖关系是通过类构造函数建立的,容器通过调用类的构造方法,将其所需的依赖关系注入其中。

```
class ClassA...
```

```
IClassB instanceOfClassB;
```

```
public classA(IClassB instanceOfClassB)
```

```
{
```

```
    this.instanceOfClassB = instanceOfClassB;
```

```
}
```

PicoContainer(另一种实现了依赖注入模式的轻量级容器)首先实现了构造子注入的依赖注入模式。

3 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被得到应用。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。

后二种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。

3.1 设值注入的优势

1) 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,那么构造子注入模式的构造函数也会相当复杂(需要在构造函数中设定所有依赖关系),而设值注入模式往往更为简单。

3) 对于某些第三方类库而言,可能要求组件必须提供一个默认的构造函数,此时构造子注入的依赖注入机制就体现出其局限性,难以完成所期望的功能。

3.2 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”,构造子注入很好地贯彻了这条 Java 设计原则。

2) 如果依赖关系比较简单,构造子注入模式避免了繁琐的 setter 方法的编写,所有依赖关系均在构造函数中直接设定,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) 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。

可见,构造子注入和设值注入模式各有千秋,而 Spring Framework、PicoContainer 等轻量级容器都对这两种类型的依赖注入机制提供了良好支持。这也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^[4]。

4 小 结

IoC 是构架松耦合应用的很好的实例。IoC 还有几个重要的好处,例如:因为组件不需要在运行时间寻找合作者,所以它们可以更简单地编写和维护。同样原因,应用代码更容易测试。大部分业务对象不依赖于 IoC 容器的 API。这使得很容易使用遗留下来的代码,且很容易地使用对象,无论在容器内或不在容器内。IoC 不同于传统容器的体系结构(EJB),代码最小程度地依赖于容器。这意味着业务对象可以潜在地被运行在不同的 IoC 框架上,或者在任何框架之外并且不需要任何代码的改动^[5]。

目前, IoC 在 Spring Framework、PicoContainer、Castle Windsor 等轻量级容器中都有很好的应用。

参考文献:

- [1] 透 明. IoC 容器和 Dependency Injection 模式[EB/OL]. 2004. <http://gigix.blogdriver.com/gigix/inc/Dependency-Injection.pdf>.
- [2] 林 润. 什么是 IoC[EB/OL]. 2004-04. <http://tech.blogbus.com/logs/2004/04/129472.html>.
- [3] Fowler M. Inversion of Control Containers and the Dependency Injection pattern[EB/OL]. 2004. <http://martinfowler.com/articles/injection.html>.
- [4] 夏 昕. Spring 开发指南[EB/OL]. 2004. http://www.xiaxin.net/Spring_Dev_Guide.rar.
- [5] Johnson R. Introducing to Spring Framework[EB/OL]. 2005. <http://www.theserverside.com/articles/article.tss?l=SpringFramework>.

(上接第 3 页)

需要使用能够精确读出旋转角度的旋转平台,使用普通的旋转平台就可以进行测量,从而减少了测量设备的成本。并且测量过程中不需要人工读取平台旋转角度,简化了测量过程。但是当所测量的物体为圆柱体或球体且旋转平台旋转轴过其对称轴时这种方法失效,因为任意两个过旋转轴的平面与物面相交所成的曲线都是完全匹配的,但是这种情况极少发生。除此之外,融合结果的精确程度与计算过程中插值精度选择有关,精度过小或者过大都不能得到最优结果。具体精度要求应该按照实际要求确定。

参考文献:

- [1] Hartley R, Zisserman A. 计算机视觉中的多视图几何[M]. 韦 穗,杨尚骏,章权兵,胡茂林,译. 合肥:安徽大学出版社,2002:223-224.
- [2] Rioux M. Laser ranger finder based upon synchronized[J].

Appl Opt, 1984, 23(21): 3837-3844.

- [3] Vandop E R, Regtien P L. Volumetric segmentation of range images for printed circuit board inspection[C]//In: Frederik Y W, Ye Shenghua. Proceedings of SPIE Int Conf on Automated Optical Inspection for Industry. Washington: SPIE, 1966: 697-694.
- [4] Yu Xiaoyang, Zhang Jian, Wu Liying, et al. Laser scanning device used in spaceencoding rangefinder[C]//In: Frederik Y W, Ye Shenghua. Proceedings of SPIE Int Conf on Automated Optical Inspection for Industry. Washington: SPIE, 1996: 490-495.
- [5] 龙 玺,钟约先,李仁举,等. 结构光三维扫描测量的三维拼接技术[J]. 清华大学学报:自然科学版,2002,42(4):477-480.
- [6] 翟 鸣. 基于结构光的三维数据测量方法研究[D]. 合肥:安徽大学电子工程与信息科学学院,2005.