基于容器的 IoC 控制反转模式的研究

杨 扬 侯 红 郝克刚
(西北大学信息学院 陕西 西安 710127)

摘 要 目前控制反转 IoC(Inversion of Control)模式广泛应用于各种应用程序框架,像 Spring、PicoContainer 和 Avalon 等这些流行的轻量级容器,都分别支持不同方式的控制反转模式的实现。首先介绍控制反转模式的设计思想及依赖注入、关注分离、依赖倒置原则等相关概念,并通过三个具体的实例就控制反转模式在以上三种框架下的具体实现进行分析,比对控制反转模式在适应各自容器需求的基础上所体现的优点和不足之处,探讨其优劣,展望其发展前景。

关键词 依赖注入 依赖倒置原则 分离关注

ON INVERSION OF CONTROL PATTERN BASED ON CONTAINERS

Yang Yang Hou Hong Hao Kegang
(*School of Information Science and Technology, Northwest University, Xi' an 710127, Shaanxi, China*)

Abstract IoC (Inversion of Control) pattern is widely used in various program application frameworks, such as Spring, PicoContainer and Avalon, etc., these popular lightweight containers at present all support the realisation of IoC pattern of different ways respectively. This paper will firstly give a brief introduction to the design idea of IoC pattern, then extends to its related concepts of Dependency Injection, Separation of Concerns and Dependency Inversion Principle, also there are three concrete instances presented in the paper, through them the specific realizations of IoC within three frameworks listed above are analyzed, the advantages and disadvantages the IoC pattern embodied are compared based on its adaptation to each containers' demand, their superior and inferior degree are discussed, and their development prospects are previewed.

Keywords Dependency injection Dependency inversion principle Separation of concerns

0 引 言

面向对象的思想已经深入人心,但是要利用面向对象的思想开发出优秀的应用程序却不是一件容易的事情。正是基于面向对象的思想,人们对各种应用程序进行了大量的分析、总结,归纳出了设计模式。Alexander 给出模式的经典定义是:每个模式都描述了一个在我们的环境中不断出现的问题,然后描述了该问题的解决方案的核心[1]。通过这种方式,你可以无数次地使用那些已有的解决方案,无需再重复相同的工作。设计模式技术面世以来,得到了广泛的关注、研究与应用。

在企业级 Java 的世界里,开发者常遇到的一个问题就是如何组装不同的程序元素:如果 Web 控制器体系结构和数据库接口是由不同的团队所开发的,彼此几乎一无所知,你应该如何让它们配合工作?很多框架尝试过解决这个问题,有几个框架索性朝这个方向发展,提供了更通用的"组装各层组件"的方案。这样的框架通常被称为"轻量级容器",PicoContainer 和 Spring 都在此列。在这些容器背后,一些设计模式发挥着作用。这些模式已经超越了特定容器的范畴,甚至已经超越了 Java 平台的范畴。时下涌现的各种框架都是利用设计模式的典范,其中 IoC 模式更是广泛应用于各种应用程序框架中,将组件的配置与使用分离开,就是 IoC 模式存在的意义。

1 反向控制/依赖注入

1.1 IoC 模式简介

IoC 模式,顾名思义,即控制反转,就像著名的好莱坞原则:"Don't Call us, We will call you",就是"不用向容器要资源,容器会自动给你所需要的"。RobertC. Martin 在其敏捷软件开发中所描述的依赖倒置原则[2] DIP(Dependency Inversion Principle)都是这一思想的体现。依赖注入(Dependency Injection)是 Martin Flower 对 IoC 模式的一种扩展的解释[3]。IoC 是一种用来解决组件(实际上也可以是简单的 Java 类)之间依赖关系、配置及生命周期的设计模式,其中对组件依赖关系的处理是 IoC 的精华部分。IoC 的实际意义就是把组件之间的依赖关系提取出来,由容器来具体配置。这样,各个组件之间就不存在直接的关联,任何组件都可以最大程度的得到重用。运用了 IoC 模式后我们不再需要自己管理组件之间的依赖关系,只需要声明由容器去实现这种依赖关系。就好像把对组件之间依赖关系的控制

收稿日期:2007-09-29。国家 863 计划资助项目(2004AA115090)。
杨扬,硕士生,主研领域:软件工程,软件理论。

进行了倒置,不再由组件自己来建立这种依赖关系而交给容器(比如 Spring)去管理。考虑一个 Button 控制 Light 的例子:

```
public class Button {
  private Light light;
  public void push() {
    light = new Light();
    light.turnOn();
  }
}
```

但是马上发现这个设计的问题,Button 类直接依赖于 Light 类,这个依赖关系意味着当 Light 类修改时,Button 类会受到影响。此外,想重用 Button 类来控制类似于 Light 的(比如同样具有 turnOn 功能的其他类)另外一个对象则是不可能的。即 Button 控制 Light,并且只能控制 Light。显然违反了"高层模块不应该依赖于低层模块,两者都应该依赖于抽象;抽象不应该依赖于具体实现,细节应该依赖于抽象"这一原则(DIP 原则)[2]。

考虑到上述问题,自然的想到应该抽象出一个接口,于是下面我们将 Light 改作为接口使用,可以使 Button 类控制实现 Light 接口的类。再深入考虑一下,虽然我们的 Button 现在可以控制实现 Light 接口的类,但是 Button 和 Light 接口之间还是存在 create 这样的依赖关系。为了解决这种依赖关心,一般会采用 Factory 模式,将对象的创建交给 Factory 类来创建,但是这种创建仍是显示的,组件变化了仍然需要重新编译程序。而采用 J2EE 经典的 ServiceLocator 模式,如果你要把 Button 组件拿到另一个系统里面用,你就必须修改它的源码,让它使用另一个系统的 ServiceLocator。换句话说,这个组件不具备可移植性。这就是为什么需要依赖注入的道理,让组件的创建、配置及生命周期总是由外部容器来管理,在这里,就是让容器把 Button、Light 以及 Light 的实现类 LightImpl 组装起来。

1.2 IoC 的分离关注

分离关注[5]SoC(Separation of Concerns)通过功能分解可得到关注点,这些关注可以是组件 Components、方面 Aspects 或服务 Services。从设计模式中,我们已经习惯一种思维编程方式:Interface Driven Design 接口驱动,接口驱动有很多好处,可以提供不同灵活的子类实现、增加代码稳定和健壮性等等,但是接口一定是需要实现的,也就是如下语句迟早要执行:AInterface a = new AInterfaceImp(); AInterfaceImp 是接口 AInterface 的一个子类,IoC 模式可以延缓接口的实现,根据需要实现,有个比喻:接口如同空的模型套,在必要时,需要向模型套注射石膏,这样才能成为一个模型实体,因此,我们将人为控制接口的实现成为"注射"。其实 IoC 模式也是解决调用者和被调用者之间的一种关系,上述 AInterface 实现语句表明当前是在调用被调用者 AInterfaceImp,由于被调用者名称写入了调用者的代码中,这产生了一个接口实现的原罪:彼此联系,调用者和被调用者有紧密联系,在 UML 中是用依赖 Dependency 表示。但是这种依赖在分离关注的思维下是不可忍耐的,必须切割,实现调用者和被调用者解耦,新的 IoC 模式 Dependency Injection 模式由此产生了。Dependency Injection 模式[4]是依赖注射的意思,也就是将依赖先剥离,然后在适当的时候再注射进入。

1.3 IoC 的类型

依赖注入的形式主要有三种,分别叫做构造子注入(Constructor Injection)、设值方法注入(Setter Injection)和接口注入(Interface Injection)。这三种注入形式分别就是 type1 IoC(接口注入)、type2 IoC(设值方法注入)和 type3 IoC(构造子注入)。

1.3.1 PicoContainer 中的构造子注入

首先,我们来看 PicoContainer 容器如何利用构造子注入来完成依赖注入的。PicoContainer 通过构造子来判断如何将 Light 实例注入 Button 类。因此,Button 类必须声明一个构造子,并在其中包含所有需要注入的元素:

```
class Button {
  private Light light;
  public Button (Light light) {
      this.light = light;
  }
  public void push () {
      light.turnOn ();
  }
}
```

随后,需要告诉 PicoContainer 去完成 Button、Light 以及 LightImpl 的组装。

```
class PicoContainer {
  private Light light;
  private Button button;
  pubic void configContainer () {
      this.light = new LightImpl ();
      this.button = new Button (this.light);
      this.button.push ();
    }
}
```

这段配置代码可以位于一个类。当然,还可以将这些配置信息放在一个单独的配置文件中,这也是一种常见的做法。你可以编写一个类来读取配置文件,然后对容器进行合适的设置。

1.3.2 Spring 中的设值方法注入

Spring 框架是一个用途广泛的企业级 Java 开发框架,其中包括了针对事务、持久化框架、Web 应用开发和 JDBC 等常用功能的抽象。和 PicoContainer 一样,它也同时支持构造子注入和设值方法注入,但该项目的开发者更推荐使用设值方法注入。为了让 Button 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 Light 的参数:

```
class Button {
  private Light light;
  public void setLight (Light light) {
      this.light = light;
  }
  public void push () {
      light.turnOn ();
  }
}
```

随后,需要告诉 SpringContainer 去完成 Button、Light 以及 LightImpl 的组装。

```
class SpringContainer {
  private Light light;
  private Button button;
  pubic void configContainer () {
```

```
        this. light = new LightImpl ();
        this. button = new Button ();
        this. button. setLight (this. light);
        this. button. push ();
    }
}
```

其实,Spring 支持多种配置方式,可以通过 XML 文件进行配置,也可以直接在代码中配置。不过,XML 文件是比较理想的配置方式。此处我们还是通过类来实现的。

1.3.3　Avalon 中的接口注入

除了前面两种注入技术,还可以在接口中定义需要注入的信息,并通过接口完成注入。Avalon 框架就使用了类似的技术。首先,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 Light 实例注入继承了该接口的对象。

```
public interface InjectLight {
    void injectLighter (Light light);
}
```

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

```
class Button implements InjectLight {
    private Light light;
    public void injectLight (Light light) {
        this. light = light;
    }
    public void push () {
        this. light. turnOn ();
    }
}
```

现在,还需要用一些配置代码将所有的组件实现装配起来。可以在代码中完成配置,并将配置好的 Button 对象保存在名为 Button 的字段中:

```
class AvalonContainer {
    private Light light;
    private Button button;
    pubic void configContainer () {
        this. light = new LightImpl ();
        this. button = new Button ();
        this. button. injectLight (this. light);
        this. button. push ();
    }
}
```

以上三种方式实现了依赖注入,这样的最大好处在于消除了 Button 类对具体 Light 实现类的依赖。这样一来,就可以把 Button 类交给任何使用者,让他们根据自己的环境插入一个合适的 LightImpl 即可。

1.4　三种 IoC 类型比较

Type1　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,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、自然。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。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,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。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 DataSource 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的支持。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 开发的应用而言,Type2 使用更加广泛。

2　结　论

使用 IoC 模式,可以不管将来具体实现,完全在一个抽象层次进行描述和技术架构,因此,Ioc 模式可以为容器、框架之类的软件实现提供了具体的实现手段,属于架构技术中一种重要的模式应用。

当然,IoC 与通常的方法相比,代码不便于理解,因为组件创建是隐含的。所以轻量级的、无侵入性的 IoC 容器仍然有待进一步的研究开发。

参　考　文　献

[1] Christopher Alexander. The Timeless Way of Building[M]. Oxford University Press, Feb. 2002.

[2] Robert CMartin. 敏捷软件开发:原则、模式与实践[M]. 清华大学出版社,2003.

[3] Martin Flower. Patterns of Enterprise Application Architecture[M]. Nov. 2002.

[4] Martin Flower. IoC 容器和 Dependency Injection 模式[J]. Jan. 2004: 1-5.

[5] 段玉聪,顾毓清. 多维关注分离的模型驱动过程框架设计方法[J]. 软件学报,2006,8:2-6.