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（HollywoodPrinciple）：“Don't callus, 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 ControlContainers 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。

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（InterfaceInjection）

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 implementsInjectUserDao{

   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（SetterInjection）

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（ConstructorInjection）

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

    publicUserRegister(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 SpringFramework 开发的应用而言，Type2 使用更加广泛。

作者：zhang1206214477
来源：CSDN
原文：https://blog.csdn.net/chidoncheung/article/details/46576921
版权声明：本文为博主原创文章，转载请附上博文链接！