

# 反转控制IoC

浙江大学城市学院

彭彬

[pengb@zucc.edu.cn](mailto:pengb@zucc.edu.cn)

# IoC

IoC：反转控制（Inversion of Control）

## 控 制 [ kòng zhì ]



[基本释义](#) [详细释义](#)

1. 掌握住不使任意活动或超出范围。

《魏书·太祖道武帝纪》：“昔朕远祖，总御幽都，控制遐国。”宋 苏洵 《衡论上·重远》：“其地控制东南夷、氐、蛮最为要害，土之所产又极富。”清 毛世楷 《武昌》诗：“枝梧蜀汉争持角，控制东南欲建瓴。”魏巍 《东方》第三部第五章：“他咬着牙控制着自己的感情，终于没掉下一滴眼泪。”

2. 指把持。

《北齐书·祖珽传》：“士开、文遥、彦深等专弄威权，控制朝廷，与吏部尚书尉瑾内外交通，共为表里。”

你觉得程序员在开发的时候控制的含义是什么？

## 看看我们前面的一个例子

我们在之前讲解的JDBC例子中，对Service的创建有下面的例子

```
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws S
    response.setStatus(HttpServletResponse.SC_OK);
    response.setContentType("text/plain");

    String userid = request.getParameter(s: "userid");
    String password = request.getParameter(s: "password");
    PrintWriter out = response.getWriter();
    //ILogin service = LoginServiceFactory.createService(LoginServiceFactory.LEVEL.L1);
    //ILogin service = LoginServiceFactory.createService(LoginServiceFactory.LEVEL.L2);
    ILogin service = LoginServiceFactory.createService(LoginServiceFactory.LEVEL.L3);
    . . .
```

上面的例子中，我们根据需要，通过一个“工厂”创建出我们需要的对象，而这个对象实现了一个接口，满足我们的某种需要。这个就是常规的使用对象的编程方法，我们在代码中完成了“用什么对象”，“什么时间创建”这样的“控制”；

上面这个方法是典型的编程方法，也是很好的方法，但是有一个什么问题？

# 看看我们前面的一个例子—有什么问题？

问题就在于耦合度，并不是说上面的耦合度导致了结构不好，而是说我们还可以追求一种更灵活的方式，而这种方式可以带给我们更低的耦合度，这种思考结构下，1996年Michael Mattson提出了IoC的概念

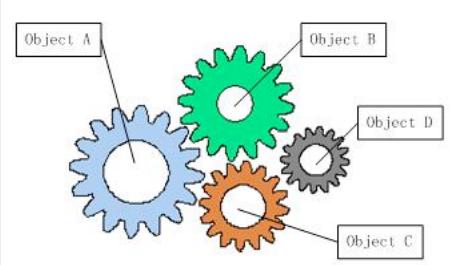
I introduced the concept of IoC in the Avalon community in 1998 and this later influenced all the other IoC-oriented containers. Some believed that I was the one to come up with the concept but this is not true, the first reference I ever found was on Michael Mattson's thesis on "Object Oriented Frameworks: a survey on methodological issues" published in 1996 that on page 96, in the conclusions, reads:

An object-oriented framework is a (generative) architecture designed for maximum reuse, represented as a collective set of abstract and concrete classes; encapsulated potential behaviour for subclassed specializations.

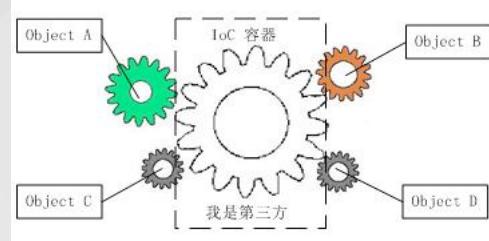
The major difference between an object-oriented framework and a class library is that the framework calls the application code. Normally the application code calls the class library. This inversion of control is sometimes named the Hollywood principle, Do not call us, we call You .

[https://en.wikipedia.org/wiki/Talk:Inversion\\_of\\_control/Archive\\_Oct\\_2007](https://en.wikipedia.org/wiki/Talk:Inversion_of_control/Archive_Oct_2007)

# 降低耦合度



正常的程序设计



降低耦合度的方法

降低耦合度的核心方法是引入一个对象（这里称为IoC容器）来“组合”对象，不让对象之间直接发生“关系”

## 进一步的讨论

上面例子中的耦合度体现为什么？

怎么降低耦合度？

## 进一步的讨论—耦合度

上面耦合度的核心是：对象的创建是在编译时确定的，也就是具体使用什么对象已经被“固化”在源代码编写的时候；

```
ILogin service = LoginServiceFactory.createService(LoginServiceFactory.LEVEL.L3);
```

```
public static ILogin createService(LEVEL level) {  
    switch (level) {  
        case L1:  
            return new LoginServiceV1();  
        case L2:  
            return new LoginServiceV2();  
        case L3:  
            return new LoginServiceV3();  
    }  
}
```

上面代码已经在编写的时候被固化为“LoginServiceV3”，这里的耦合度就是指当源代码编写好之后，如果要修改使用者和LoginServiceV3的“关系”，必须要修改源代码并重新编译部署，这就是这里所讨论的“耦合度”。

## 进一步的讨论—降低耦合度

按照之前的讨论，我们用“*IoC*”反转控制的思想来降低耦合度，那么到底是怎么回事呢？

2004年，*Martin Fowler*探讨了这个问题，既然*IOC*是控制反转，那么到底是“哪些方面的控制被反转了呢？”，经过详细地分析和论证后，他得出了答案：“获得依赖对象的过程被反转了”。控制被反转之后，获得依赖对象的过程由自身管理变为了由*IOC*容器主动注入。于是，他给“控制反转”取了一个更合适的名字叫做“依赖注入(*Dependency Injection*)”。他的这个答案，实际上给出了实现*IOC*的方法：注入。所谓依赖注入，就是由*IOC*容器在运行期间，动态地将某种依赖关系注入到对象之中。

所以，依赖注入(*DI*)和控制反转(*IOC*)是从不同的角度的描述的同一件事情，就是指通过引入*IOC*容器，利用依赖关系注入的方式，实现对象之间的解耦。

## Martin的例子（2004年）

在这个例子中，我编写了一个组件，用于提供一份电影清单，清单上列出的影片都是由一位特定的导演执导的。实现这个伟大的功能只需要一个方法：

```
class MovieLister...

    public Movie[] moviesDirectedBy(String arg)
    {
        List allMovies = finder.findAll();
        for (Iterator it = allMovies.iterator(); it.hasNext();)
        {
            Movie movie = (Movie) it.next();
            if (!movie.getDirector().equals(arg))
            {
                it.remove();
            }
        }
        return (Movie[]) allMovies.toArray(new Movie[allMovies.size()]);
    }
}
```

## Martin的例子—功能接口

我们真正想要考察的是finder对象，或者说，如何将MovieLister对象与特定的finder对象连接起来。为什么我们对这个问题特别感兴趣？因为我希望上面这个漂亮的moviesDirectedBy方法完全不依赖于影片的实际存储方式。所以，这个方法只能引用一个finder对象，而finder对象则必须知道如何对findAll方法作出回应。为了帮助读者更清楚地理解，我给finder定义了一个接口：

```
public interface MovieFinder
{
    List findAll();
}
```

## Martin的例子—接口对象创建

现在，两个对象之间没有什么耦合关系。但是，当我要实际寻找影片时，就必须涉及到MovieFinder的某个具体子类。在这里，我把涉及具体子类的代码放在MovieLister类的构造函数中。

```
class MovieLister...
    private MovieFinder finder;
    public MovieLister()
    {
        finder = new ColonDelimitedMovieFinder("movies1.txt");
    }
```

# Martin的例子—使用不同的接口对象

如果用完全不同的方式——例如SQL 数据库、 XML 文件、 web service， 或者另一种格式的文本文件——来存储影片清单呢？在这种情况下，我们需要用另一个类来获取数据。由于已经定义了MovieFinder接口，我可以不用修改moviesDirectedBy方法。但是，我仍然需要通过某种途径获得合适的MovieFinder实现类的实例。

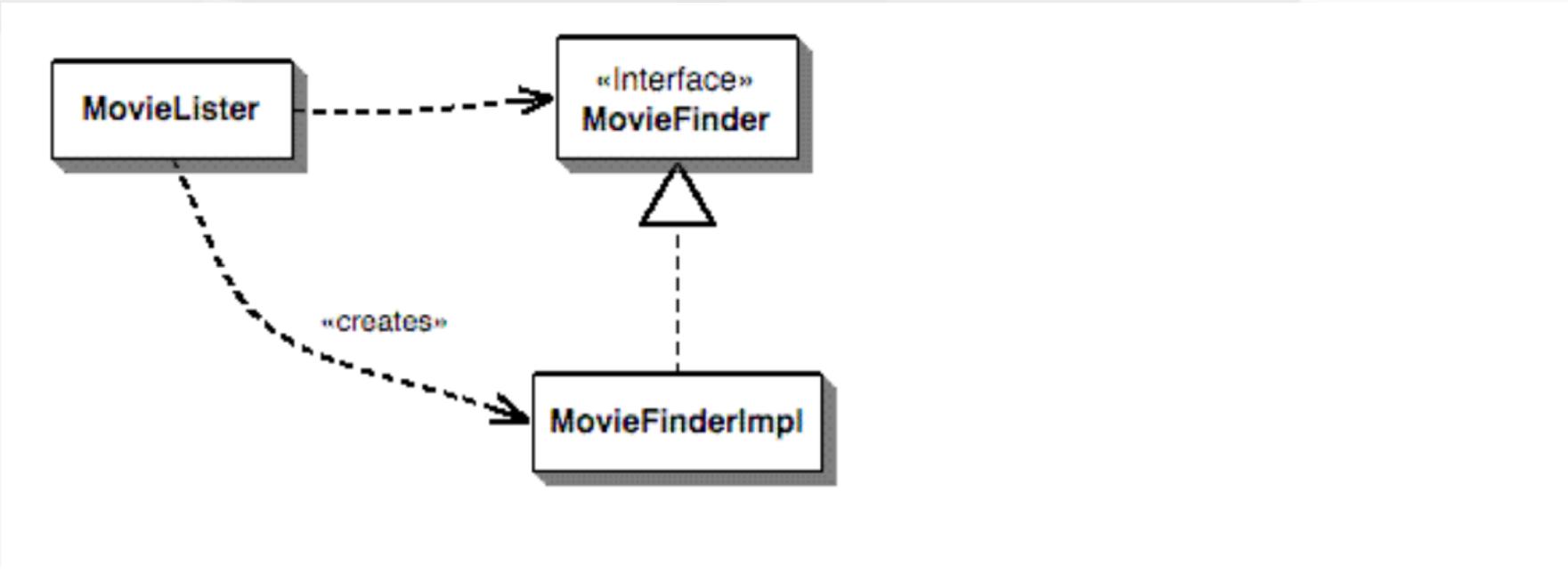
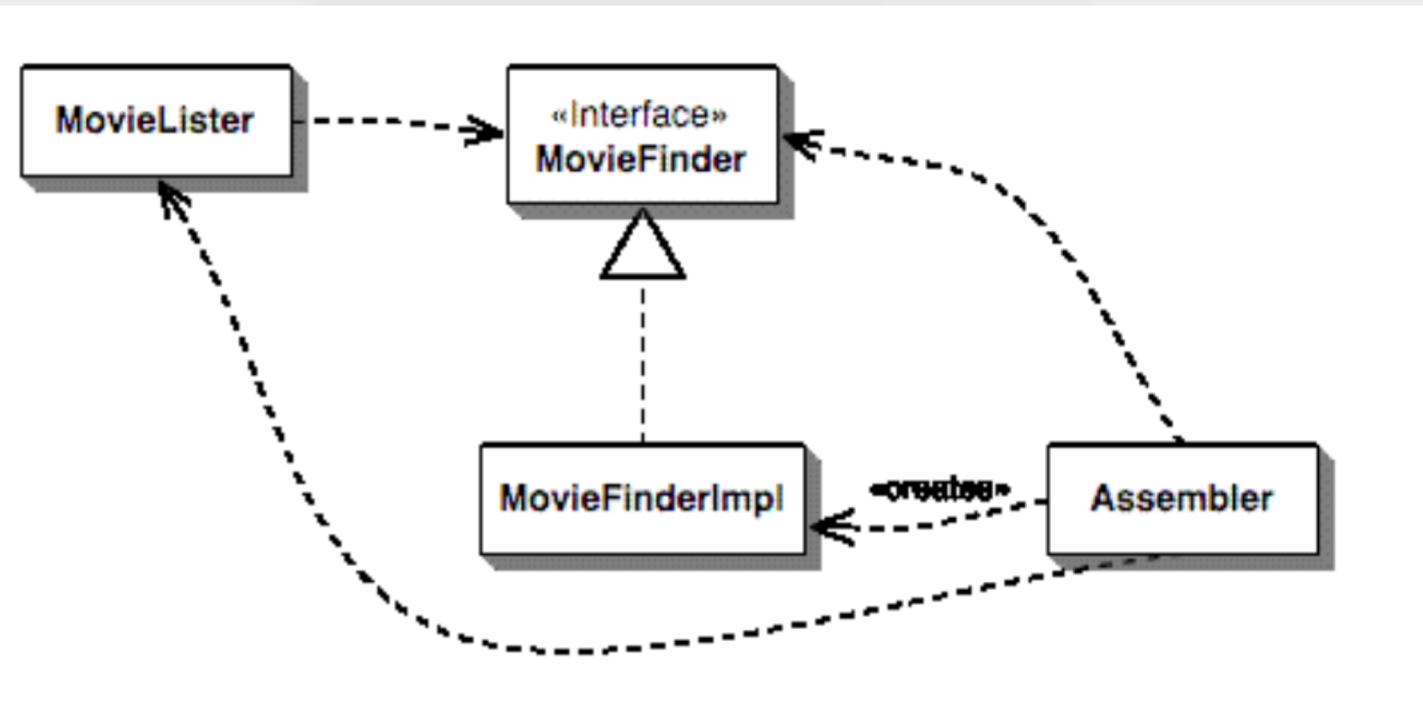


图1：在MovieLister 类中直接创建MovieFinder 实例时的依赖关系

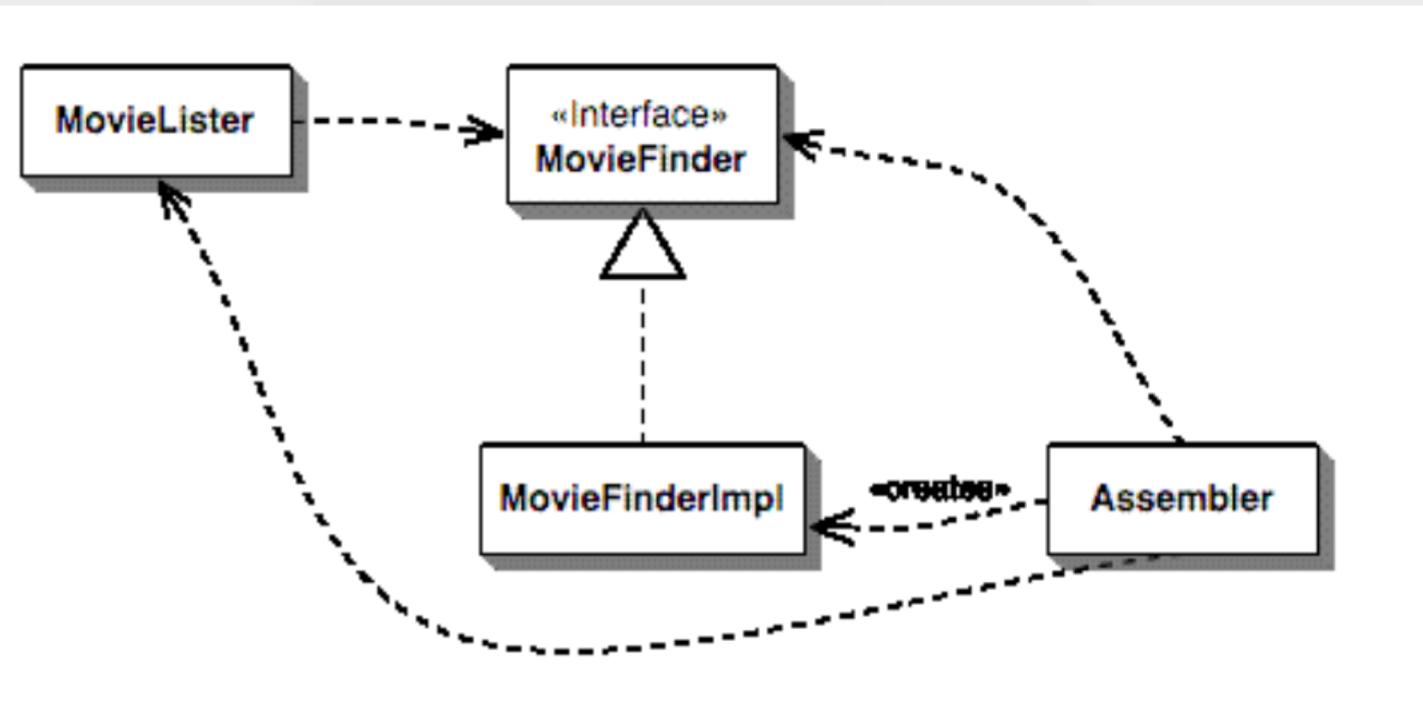
## Martin的例子—对象动态创建（依赖注入）

Dependency Injection模式的基本思想是：用一个单独的对象（装配器）来获得MovieFinder的一个合适的实现，并将其实例赋给MovieLister类的一个字段。这样一来，我们就得到了图2所示的依赖图：



# Martin的例子—对象动态创建（依赖注入）

Dependency Injection模式的基本思想是：用一个单独的对象（装配器）来获得MovieFinder的一个合适的实现，并将其实例赋给MovieLister类的一个字段。这样一来，我们就得到了图2所示的依赖图：



## Martin的例子—依赖注入（用Setter方法）

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

```
class MovieLister...
    private MovieFinder finder;
    public void setFinder(MovieFinder finder)
    {
        this.finder = finder;
    }
```

类似地，在MovieFinder的实现类中，我也定义了一个设值方法，接受类型为String的参数：

```
class ColonMovieFinder...
    public void setFilename(String filename)
    {
        this.filename = filename;
    }
```

## Martin的例子—依赖注入（用Setter方法）

第三步是设定配置文件。Spring 支持多种配置方式，你可以通过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。

```
<beans>
    <bean id="MovieLister" class="spring.MovieLister">
        <property name="finder">
            <ref local="MovieFinder"/>
        </property>
    </bean>
    <bean id="MovieFinder" class="spring.ColonMovieFinder">
        <property name="filename">
            <value>movies1.txt</value>
        </property>
    </bean>
</beans>
```

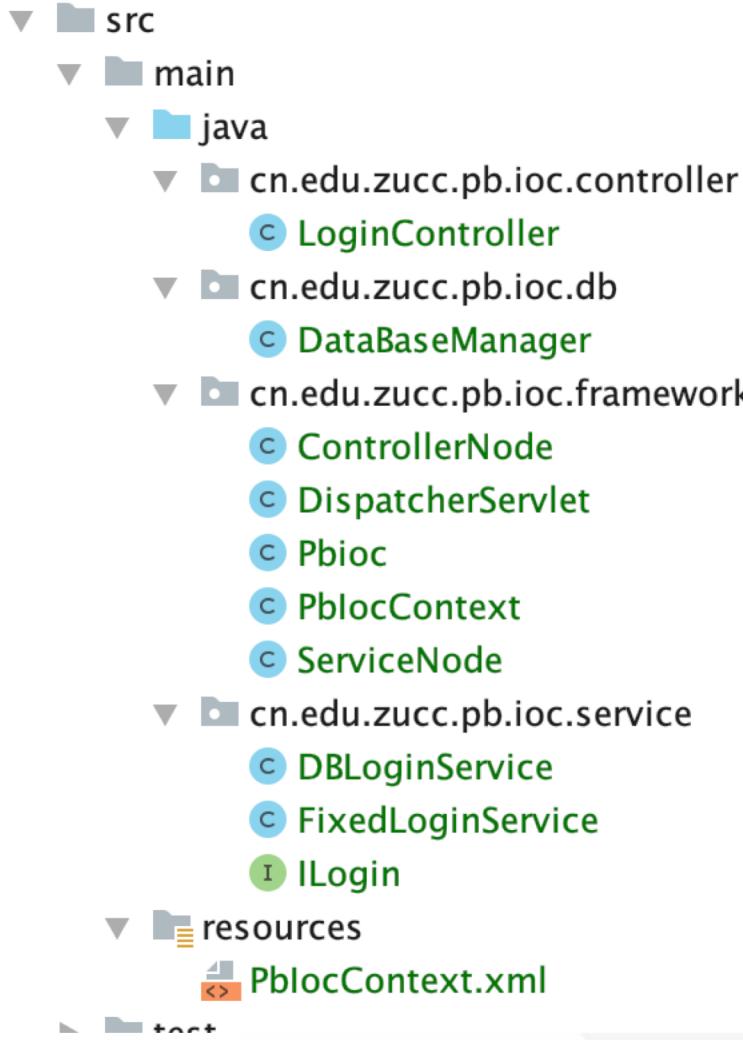
# Martin的例子—想象动态性的实现

接下去让我想象一下根据配置文件实现的动态性

- 1) 通过配置文件，我们知道了“MovieLister”这个对象是“spring.MovieLister”类的实现，所以我们可以用Java的反射Class.forName来加载对应的类并动态创建其实例
- 2) 通过配置文件，我们同样知道了MovieLister需要引用一个称为“MovieFinder”的对象，需要设置到“MovieLister”的“finder”属性
- 3) 通过配置文件，我们知道了“MovieFinder”是class “spring.ColonMovieFinder”的实例，我们可以用Java的反射Class.forName来加载并实例化此对象
- 4) 然后我们通过反射，调用“MovieLister”的setFinder方法设置其对应属性，实现“注入”

\*上面的步骤通过接口设计+配置文件+反射的方法实现了动态的注入对象，这就实现了对象依赖的解耦，如果创建一个中心的角色来实现这套创建并注入的机制，那么我们就实现了“依赖注入”，也就实现了“控制反转”，对象自己不再创建所需的依赖，而依赖于配置并运行时进行注入，这就是反转控制的思想及其实现，也就是Spring的基石。

# IoC示例代码分析~一个自己实现的Mini-Spring



示例代码分析 (L05IoCApp) :

- 1) 入口 Servlet: DispatcherServlet
- 2) 配置管理 PbIocContext类
- 3) 实现通过配置文件的 IoC

Controller/Service的绑定是动态装配的

其它：

使用 JAXB 实现 XML 的快速分析

# 参考

<http://www.cnblogs.com/DebugLZQ/archive/2013/06/05/3107957.html>

<http://www.cnblogs.com/xingyukun/archive/2007/10/20/931331.html>

<https://www.martinfowler.com/articles/injection.html>

<http://insights.thoughtworkers.org/injection/>

[https://en.wikipedia.org/wiki/Talk:Inversion\\_of\\_control/Archive\\_Oct\\_2007](https://en.wikipedia.org/wiki/Talk:Inversion_of_control/Archive_Oct_2007)



END

---

Pb&Lois