透明思考

7月 27 2018 •

动态代理的前世今生

（本文发表于《程序员》杂志2005年第1期，是我多年技术文章写作中难得的佳作，故此特意从杂志扫描文件转录为文字，以备散轶。）

> *文字写作有一个坏处在这里，斐多，在这一点上它很象图画。图画所描写的人物站在你面前，好像是活的，但是等到人们向他们提出问题，他们却板着尊严的面孔，一言不发。写的文章也是如此。你可以相信文字好像有知觉在说话，但是等你想向它们请教，请它们把某句所说的话解释明白一点，它们却只能复述原来的那同一套话。*
>
> *——苏格拉底*

遭遇这种困扰的不仅有图画和文字。实际上在我们这个领域里，也有一些技术曾经或正在处于类似的遭遇这种困扰的不仅有图画和文字。实际上在我们这个领域里，也有一些技术曾经或正在处于类似的窘境：它们的发明者赋予了它们连篇累牍的文档，但这些文档终于——尴尬地——无法向它们的读者说清自己的功用。它们只能一遍遍地重复着那些晦涩的官样文章，让读者们大打哈欠。然后，直到某一天，仿如天启一般，一位阐释者出现了，惟有等他——而不是这技术的创造者——把这技术的用途解说得清清楚楚。人们才终于发现了这种新技术的价值。

这并非是天方夜谭。在Java语言中有一种名为动态代理（dynamic proxy）的技术，如今人们将它奉为天物，如果一个容器没有使用动态代理（或者更高级的代替物），简直就会立即被时代的潮流淘汰（梢后我们会看到这样的例子）。然而，动态代理技术原本是随J2SE 1.3于2000年发布的，为何要到两三年之后才突然变得炙手可热，这里发生了怎样曲折的故事？今天，笔者不揣冒昧，想带领读者来了解动态代理背后的故事。

什么是动态代理

在JDK 1.4.2的JavaDoc文档中，我们找到了`java.lang.reflect.Proxy`类，然后就是一篇长长的介绍，其中这样写着：

> *可以在运行的、在创建这个类的时候为指定它所实现的接口，这些被代理类实现的接口被称为**动态接口**（dynamic interface）。代理类的实例被称为代理实例（proxy instance）。每个代理实例都有一个对应的**调用处理器**（invocation handler）对象，该对象实现 `java.lang.reflect.InvocationHandler` 接口。当用户通过代理接口调用代理实例的方法时，该方法调用会被分发到该实例对应的调用处理器的 `invoke()` 方法，同时传入动态接口、代表被调用方法的 `java.lang.reflect.Method` 对象、以及代表方法调用参数的一个对象数组。调用处理器可以在 `invoke()` 方法中对接收到的方法调用进行相应的处理，该方法返回的结果应该是代理实例被调用得到的结果。*

你明白这是怎么回事了吗？噢，不用感到惭愧，即便是当我翻译这段文字时，我都很难读懂它到底想说什么。还好旁边就有一个例子。假如我们有一个接口 `Foo`：

```
public interface Foo {
    void doSomething();
}
```

你就可以直接创建一个实现了 `Foo` 接口的对象实例，而不必为它创建一个实现类：

```
InvocationHandler handler=new MyInvocationHandler();
Class proxyClass = Proxy.getProxytClass(
    Foo.class.getClassLoader(), new Class[]{ Foo.class });
Foo f = (Foo) proxyClass,
         getConstructor(new Class[] { InvocationHandler.class }).
               newInstance(new Object[] { handler });
f.doSomething();
```

当然，你也注意到了，这里有一个新的类：`MyInvocationHandler`，这就是与动态代理对应的调用处理器。毕竟，我们总得有一个地方放置真正的实现逻辑吧。这个类的实现如下：

```
public class MyInvocationHandler implements InvocationHandler
{
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwab
        System.Out.println("Hello!”);
        return null;
    }
 }
```

一个具体实现类的实例传给调用处理器，然后把具体业务的实现委派给这个对象去完成。

```
private Object _target;
public MyInvocationHandler(Object target) {
    _target = target;
}
```

然后在`invoke()`方法中委派`_target`对象处理实际的业务逻辑：

```
private Object invoke(...) {
      return method.invoke(_target, args);
```

当然使用者也必须做一些修改（`FooImpl`是一个实现了`Foo`接口的具体类）：

```
InvocationHandler handler = new MyInvocationHandler(new FooImpl());
```

那么，这个东西又有什么特别之处呢？请再仔细看代理类的创建过程，尤其是`getProxyClass()`方法的第二个参数：

```
Class proxyClass = Proxy.getProxyClass(
    Foo.class.getClassLoader(), new Class[]{ Foo.class });
```

这个参数传入的是代理类将要实现的所有接口。也就是说，我们的`MyInvocationHandler`不仅可以用于创建`Foo`接口的实例，同样可以创建其他任何接口的实例。再看看创建代理实例时传入的第三个参数，这是一个`MyInvocationHandler`的实例，我们也可以在运行时改变这个实例内部包含的实现对象（即`_target`对象）。现在我们可以知道“动态代理”这个名字从何而来了，在《设计模式》书中这样写着：

> *Proxy（代理）：对象结构型模式*
> *意图：为其他对象提供一种代理以控制对这个对象的访问。*
> *别名：Surrogate（替身）*

在这里，被“控制访问”的对象（即`_target`对象）可以在运行时改变，需要控制的接口可以在运行时改变，控制的方式（即`InvocationHandler`的具体实现）也可以在运行时改变。所以在这个代理模式中，代理者和被代理者的关系是完全动态的，这就是“动态代理”名称的由来。

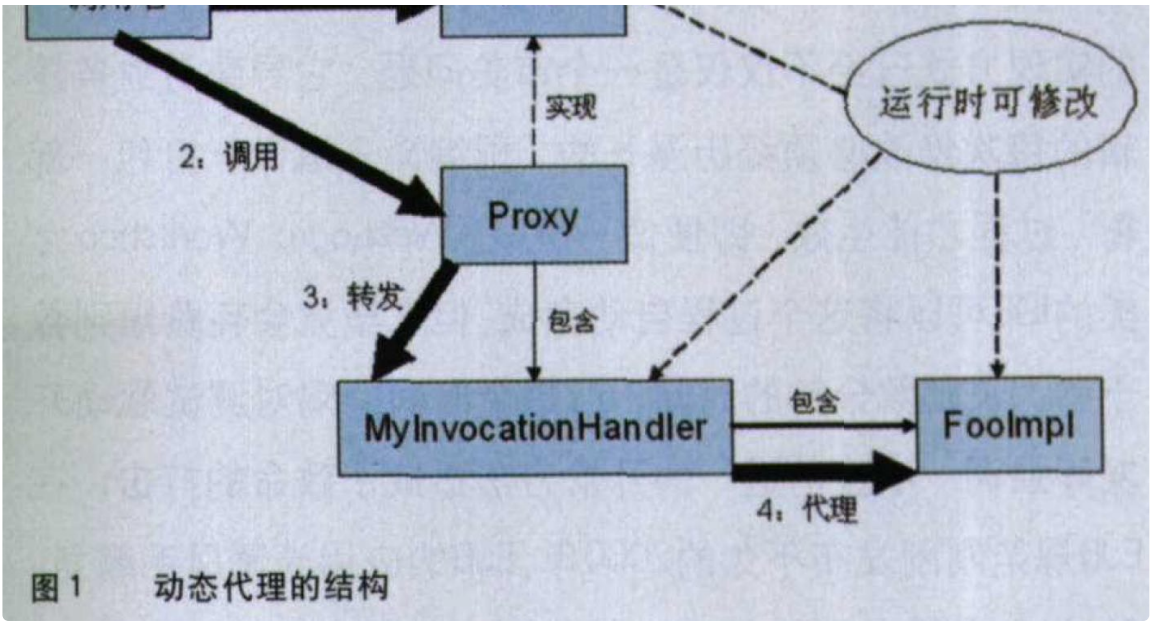

图1 动态代理的结构

RMI的故事

也许你要说了：“好吧，我现在知道动态代理是怎么回事。可是它有什么用呢？”在2000年，刚拿到JDK 1.3的很多人都发出了同样的问题，其中就包括有“Java神童”之称的Rickard Oberg。当时Oberg正在设计JBoss的EJB容器——准确说是EJB容器中的RMI部分。我们都知道，如果一个EJB实现了本地接口，对它的调用将是普通Java方法调用；而如果它实现了远程接口，对它的调用就会变成RMI远程调用。从Sun的参考实现开始。绝大多数EJB容器（包括IBM WebSphere、BEA WebLogic等业界领先的产品）都采用了预编译的做法：首先根据业务代码生成对应的RMI skeleton，然后把 skeleton和业务代码一起编译，由容器在运行时将方法调用分发给合适的skeleton去执行。

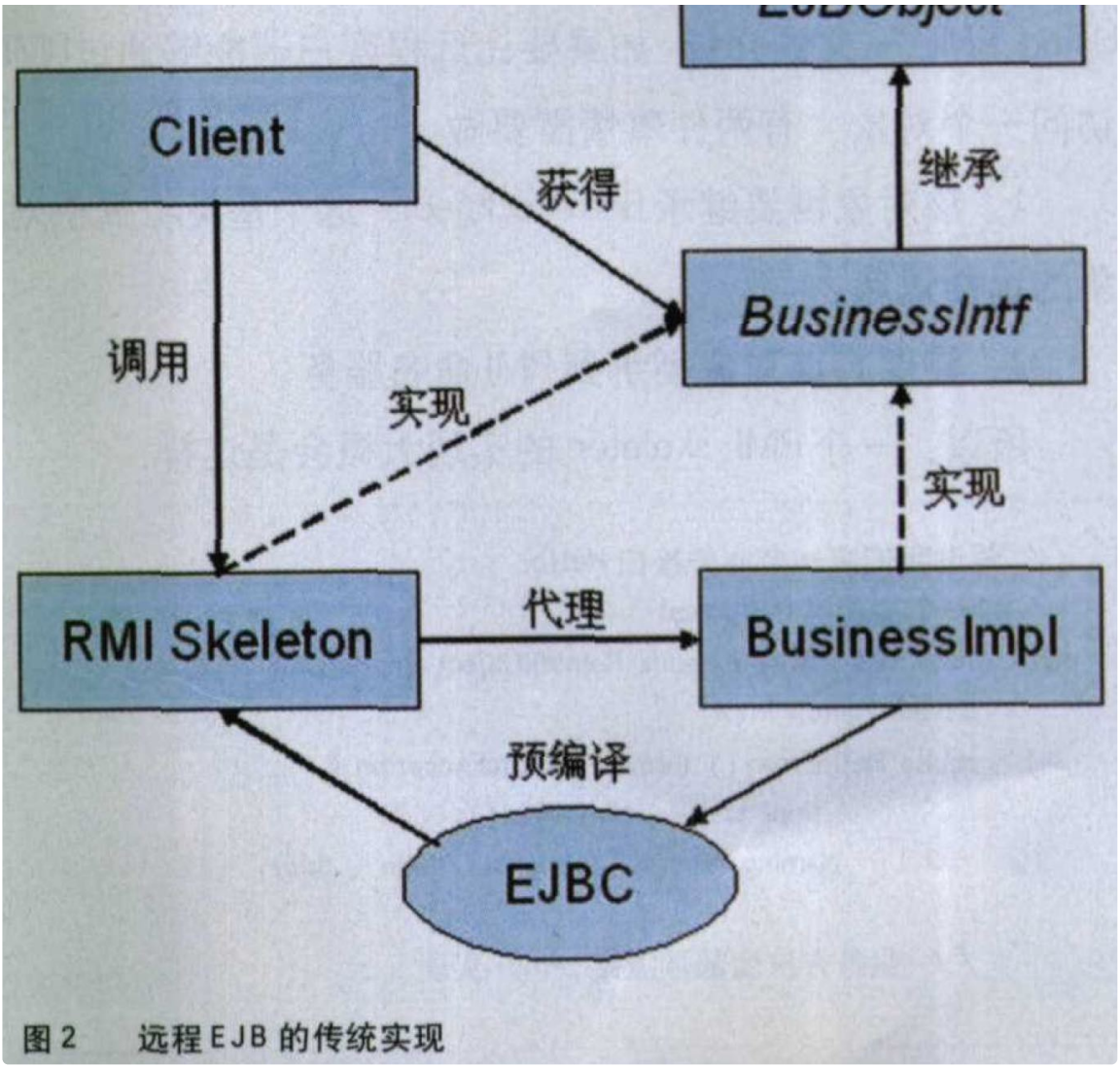

图 2　远程 EJB 的传统实现

但是在Oberg敏锐而又挑剔的眼里，这种传统的实现方式是极度缺乏美感的。既然所有的远程EJB都必须提供RMI访问的途径，既然这种途径本身是与业务逻辑毫无关系的，那么凭直觉，这种“RMI访问的途径”就应该是一个单独的模块，而不是为每个EJB分别生成一份这样的代码。几年以后，当我们重新审视EJB的设计时，我们不得不敬佩Oberg独到的眼光。当EJB投入大规模应用之后，这种基于预编译的实现方式已经不仅仅是一个审美问题。它导致对业务逻辑的每次修改必须经历漫长的“预编译—编译—打包—部署”过程才能生效，即便像WSAD、WebLogic Workshop之类的IDE可以将这个过程自动完成，但它毕竟会耗费短则数十秒、长则数分钟的时间，而这个时间间隔对测试驱动开发等强调“小步前进”的开发方法造成了致命的打击。在EJB规范刚刚发布不久的2000年，EJB的应用尚属凤毛麟角，Oberg完全凭直觉和审美就察觉了这种实现方式的缺陷，让我们不得不相信：很多时候具有美感的技术就是好的技术。当然，这是题外话了。

但是，要把“RMI访问的途径”抽象到一个单独的模块中并不是一件容易的事情，否则别的容器实现者也不会选择代码生成的方式了。按照JDK文档中

1. 该对象需要继承`RemoteObject`，这个基类将提供对象的远程语意。
2. 需要将该对象捆绑到RMI命名服务。

所以，一个RMI skeleton的实现大概会是这样：

```
// 假设我们有一个业务接口Hello
// 和一个实现类HelloImpl
public class HelloProxy extends RemoteObject implements Hello {
    private Hello _impl;
    public HelloProxy() throws RemoteException {
        _impl = new Hellolmpl();
        Naming.rebind("//myhost/hello", this);
    }
    // 业务方法全部委派给_impl实现
}
```

可以看到，尽管各个RMI skeleton的实现非常相似，但它们有一个最大的差异：它们必须实现不同的业务接口。熟悉C++的读者或许会说，如果有泛型（generic）技术，这个问题就不成问题了：我们可以设计一个`RMIProxy`泛型类，并把`Hello`接口作为类型参数传入。话是没错，但别忘了，我们现在身处2000年，J2SE 1.3才刚刚发布，拥有泛型技术的J2SE 1.5自然不能为我们所用了。就为了获得实现不同业务接口的skeleton，EJB的实现者们才不得不选择了代码生成的手段。于是，这里的问题就集中为一个：

> *如何创建这样一个代理类：它的实例可以实现任意的业务接口，并且可以在运行时决定一个实例究竟实现哪个业务接口。*

啊哈，你开心地说，这不正是动态代理能提供的吗？2000年9月，Rickard Oberg也有同样的欣喜若狂：他突然发现，原来JDK 1.3提供的这个叫动态代理的东西可以这样来用。剩下的事情就是按部就班地写程序了。最后，Oberg完成了这个优雅的EJB实现，也奠定了JBoss直到4.0版本之前没有改变的架构基础。这是全世界第一个基于动态代理机制的EJB实现，JBoss一向引以为傲的热发布能力很大程度上正是得利于这个优雅的架构。如果读者对Oberg的这个“尤里卡”式的发现有兴趣，不妨找出JBoss 3.x的源码，亲眼看看RMI这部分的实现。

作为XDoclet的作者，Rickard Oberg不应该被看作代码生成技术的反对者。然而，相比之下，他对动态代理的热情绵延数年不绝，对于XDoclet倒是疏于维护，似乎只当它是一个玩具，这又是为什么呢？沿着这个问题，我们又拉开了另一个故事的帷幕……

完成了JBoss的核心实现之后，Oberg并没有停止思考。一方面，他深感EJB的繁复给开发者带来了巨大的不便，于是开发了一个名为“EJBDoclet”的工具。这个小工具可以根据业务代码中特定的JavaDoc注释生成部署描述符、Home对象实现类等文件，大大减轻了EJB开发者的负担。2001年初，EJBDoclet的beta版本甫一发布，便受到众多EJB开发者的青睐。然后，Oberg又对EJBDoclet的架构重新做了设计，赋予了它灵活的插件机制，让开发者们可以创建各种插件插入其中，从而支持其他用途的代码生成。2001年9月，Oberg将EJBDoclet改名为XDoclet，此时它不仅可以支持EJB代码生成，还支持Struts、JSP Taglib、WebWork、Apache-SOAP和JMX。时至今日，几乎凡是需要配置文件的地方，你都可以找到XDoclet的支持，XDoclet已经成了J2EE开发者工具箱里必不可少的一件利器。这是后话，按下不表。

另一方面，Oberg本人仍然坚信：代码生成（尤其是自动生成Java代码）是一种恶行——虽然或许是必要的恶行。作为Java编程的顶尖高手，他对于“Don’t Repeat Yourself”这句箴言的领悟比大多数人更深刻。于是他开始总结：有哪些事情是在不断重复做的？动态代理能不能对解决这些问题有所帮助？对于第一个问题，他很快找到了不少的答案，例如：

- 日志：如果需要记录所有方法的调用时间、传入参数、返回值，就必须编写大量的重复代码。
- 安全性检查：如果需要给业务方法加上基于角色的访问控制，就必须逐个方法修改。
- 事务管理：如果需要访问事务性资源（例如关系型数据库），必须在每个操作之前开启事务、操作结束之后提文或回滚事务。

这些问题有一个共同的特点：它们都分布在各个对象继承体系中，任何业务对象都可能需要它们；另一方面，它们又与具体的业务逻辑几乎没有关系。也就是说，它们纯粹是基础设施（infrastructure）功能。如果用经典的面向对象手段来解决这类基础设施问题，必须采用Template Method或类似的设计模式：业务对象实现特定的接口（或继承特定的基类），通过接口提供的回调接口提供基础设施服务。

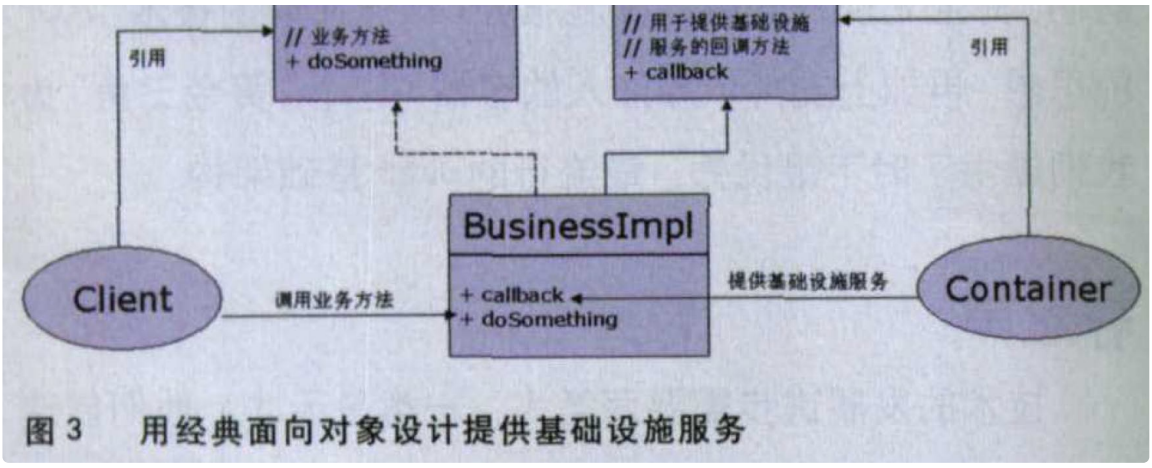

图 3 用经典面向对象设计提供基础设施服务

但这种设计方案有两个缺陷。其一，业务对象除了实现业务接口之外，还必须实现一个容器特有的接口（例如EJB容器规定的SessionBean接口）。甚至是继承容器特有的基类，这对业务对象造成了侵入，使它难以移植到别的应用环境中；其二，由于基础设施都是通过回调接口提供，所以“可以提供哪些基础设施服务”实际上是预先规定好的，不能根据需要定制——比如说，如果你突然需要统计每个业务方法执行的时间，那么很抱歉，容器提供的接口上没有相关的回调方法，所以无法提供这样的基础设施。这两个重大的缺陷正是导致EJB和Apache Avalon等经典容器最终无法满足需要而被淘汰的根本原因。

早在多年之前，一些来自高校和研究机构的学院派人士就提出：对于这种“横切”（crosscut）多个对象继承体系的基础设施问题，可以通过AOP （Aspect—Oriented Programming，面向方面编程）的手段来解决。1999年，帕洛阿尔托研究中心（PARC）的科学家们创造了一个叫做AspectJ的AOP实现品，并且在一些项目中用它来解决基础设施问题，取得了很好的效果。但是和大多数EJB容器一样，AspectJ也采用了代码生成的技术：程序员编写好业务代码和aspect代码之后，必须进行预编译，AspectJ的预编译器会把aspect织入业务逻辑之中，生成新的Java代码，然后才能编译运行。再一次地，代码生成显出了它的缺陷：由于每次修改业务代码之后都必须重新预编译，程序员的“开发—测试—重构”步伐不得不放慢；更糟的是，由于最终编译运行的是预编译之后生成的新代码，一旦有异常抛出，跟踪栈里的行号和业务代码完全不符，调试器也会因此而失效。这些问题使得AspectJ最终也仅仅是一个阳春白雪的实验室产品，没能在J2EE世界获得广泛的应用。

但AspectJ毕竟指出了一个方向、一种可能性，它的问题只在于代码生成。一位年轻人开始思索：如何不使用代码生成技术实现AOP。他就是Jon Tirsen，Rickard Oberg离开JBoss之后的挚友之一。Oberg是一个天才，但和所有的天才一样，他也有一个致命的毛病：他痛恨与智商低的人交流——如果对方跟不上他的节奏，他宁可选择缄默。所以，尽管用动态代理技术如此优雅地解决了EJB容器中RMI的

的交流很快碰出了火花：大卫特Oberg在JBoss里做的事抽象泛化，那就是一个基于动态代理的AOP实现。

既然方向已经明确，Tirsen立即投入了设计和实现的工作。他采用了“拦截器链”（Interceptor Chain）的体系结构：对于每个具有横切性质的问题，用一个拦截器（interceptor）来解决，然后把多个拦截器串成一条链；在一个动态代理的`InvocationHandler`中拦截到所有的方法调用。就可以挨个执行链条上的每个拦截器。这样一来，不管有多少基础设施服务需要提供，也可以应付裕如了。很快，他就完成了这个产品，这就是Nanning框架——Tirsen曾到中国的广西旅游，旖旎的山水令他乐而忘返，所以他用了广西首府南宁的名字为自己的作品命名。2002年11月，Nanning发布了第一个版本。随后，由Jon Tirsen和Rickard Oberg牵头，一群AOP技术的先行者又成立了“AOP联盟”（AOP alliance）组织，制订了Java AOP的标准API。这样，aspect本身也可以在不同的AOP框架之间移植、复用，只要这些框架都支持这套标准。如今所有基于动态代理的开源AOP框架都实现了这套标准，说明Jon Tirsen选择的架构思路已经得到了同行的一致认可。

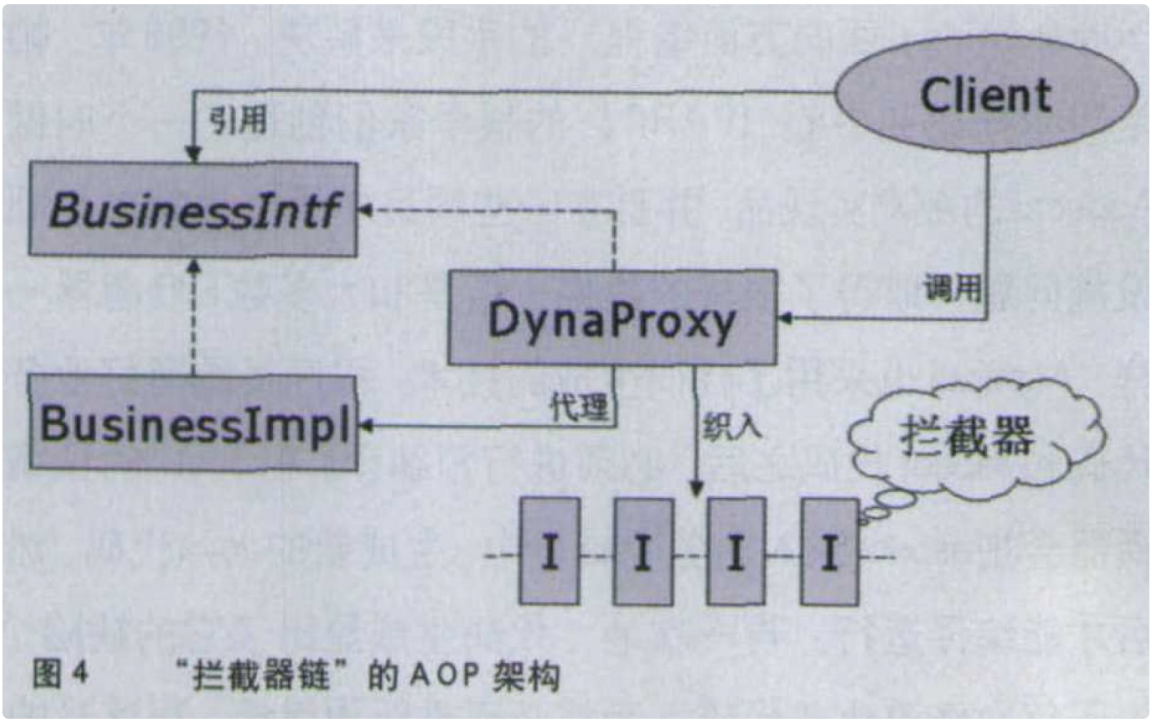

图 4 “拦截器链”的 AOP 架构

但这时的成果还不足以让Tirsen和Oberg满意，因为它的易用性还不够好。在2002年以前，几乎每个J2EE应用中都有大量的工厂（包括Service Locator），这是为了分离接口与实现、分离使用与创建，甚至有很多优秀的架构师声称：“没有大量工厂的应用不是好的应用。”如果要为这些应用引入AOP，势必要大范围修改工厂：即便是在新应用中引入AOP，也需要对工厂的实现方式做很多调整，从而丧失了AOP理想中的“透明性”——即：所有业务代码都不知晓AOP的存

公。

最后的一块拼图很快就出现了。ThoughtWorks公司（也就是Martin Fowler的公司）的Paul Hammant和Aslak Hellesoy等几位员工在业余时间创建了一个开源项目，叫做PicoContainer——极其徽小的容器。这个容器可谓名副其实：它的全部源码只有区区20个类、数百行代码，甚至没有使用JDK之外的任何类库。可是，Tirsen甫一见它，立即为之倾倒，迫不及待地加入了它的开发团队，并把它推荐给好友Oberg，后者也同样如获至宝，在自己的blog上对它赞不绝口。

这个“小东西”的魅力究竟何在？在于它实现了后来被Martin Fowler称为“依赖注入”（Dependency Injection）的机制。简而言之，PicoContainer的核心理念可以提炼为两点：第一，使用者必须通过容器获得组件；第二，组件的创建、生命周期和依赖关系由容器全权负责。抛开别的好处不说，这样的容器能够提供一个全局唯一的对象获取点，实现Tirsen和Oberg希望的、“透明的”AOP也就有了下手之处。于是，Tirsen很快就把Nanning和PicoContainner（准确地说，是PicoContainer的姊妹项目，增加了XML配置等外围功能的NanoContainer）整合起来，得到了一个近乎完美的解决方案：事务管理、安全性检查、异常处理等基础设施都被实现为一个拦截器；一个业务组件是否需要拦截器，需要哪些拦截器都可以在配置文件中表达，不管业务代码还是用户代码都不需要任何修改，用Oberg的话来说，业务代码永远是这样：

```
public class BusinessImpl implements BusinessIntf {
    pubIic void doSomething() {
        //仅仅实现业务逻辑，不考虑基础设施
    }
}
```

而用户代码则永远是这样：

```
public class Client {
    private BusinessIntf _business;
    // Type 3 IoC：构造子注入
    public Client(BusinessIntf business) {
        _business = business
    }
    // 直接使用_business对象，不考虑基础设施
}
```

这样一来，不管业务组件还是客户端都具有更好的可移植性，而且更容易测试：只要直接new出一个对象，通过构造子把它需要依赖的对象传入其中，就可以测试它的业务逻辑；如果需要改变依赖的对象，也只要在配置文件中修改一行即

全检查，业务代码和用户代码始终都是上面这样简单的形式，不会受到丝毫影响。这正是无数人梦寐以求的完全透明、完全无侵入的基础设施服务。动态代理的技术、AOP的思想，再加上支持依赖注入的容器，这个“黄金三角”为我们提供了时下最优秀最流行的J2EE基础架构。

春之声

技术的发展进步需要两类人：一类是天才，他们创造奇迹、拓展技术的疆域；另一类是传教士，他们读懂天才的思想，并传之于普罗大众。在J2EE的世界里Rickard Oberg是众望所归的天才，而Rod Johnson则是当仁不让的传教士。这位面容清瘦，已经谢顶的中年人是Servlet技术规范专家组的成员，那本厚达1400页的《J2EE编程指南》也有他贡献的章节。不过，Johnson真正名扬四海，还是因为他在2002年底出版的那本*Expert One-on-One J2EE Design and Development*，以及随后推出的Spring开源项目。公平地说，不管在*J2EE Design and Development*书中还是在Spring框架中，以至于在2004年初出版的*Expert One-on-One J2EE Development without EJB*中，Rod Johnson没有提出任何新思想、新技术——他所做的就是把天才们的发现归纳整理分门别类，写成寻常程序员也能看懂、能用上的书和框架。和我们这个故事有关的，AOP的种种用法——例如事务管理、remot ing、安全检查——都是经过Johnson的言传身教才飞入寻常百姓家，从这一点来说，Rod Johnson “J2EE传教士”的名号受之无愧。

既然Spring已经把AOP实现得非常易用，我们不妨来试着亲手编写一些AOP的程序。首先，假设在bean工厂配置文件中声明了这样的一个bean组件：

```
<bean id = “helloBean”class=”my.packge.Hellolmpl”/>
```

使用这个bean的客户端代理如下：

```
BeanFactory factory = ...;
Hello hello = (Hello) factory.getBean(“helloBean”);
System.out.println(hello.sayHello());
```

一切运行正常，控制台上打出了`Hello，World!`的字样。现在，我们想要获得每个方法被调用的时间，正如前文所说，这是一个具有横切性质的基础设施问题，可以用一个拦截器来描述它：

```
    public Object invoke(MethodInvocation invocation) throws Throwable {
        Date now = new Date();
        System.out.printin(
            invocation.getMethod().getName() + " : " + now);
        return invocation.proceed();
    }
}
```

`MethodInterceptor`是AOP联盟API中规定的一个用于方法调用拦截的拦截器接口，在它的`invoke()`方法中，已经把方法调用封装成方便的`MethodInvocation`对象。只要经过适当的注册，被拦截对象的每次方法调用都会首先通过拦截器的`invoke()`方法。现在我们就来配置这个拦截器，Spring提供的`ProxyFactoryBean`可以将拦截器与业务组件结合起来：

```
<bean name="timeInterceptor"
    Class="my.package.InvokeTimeInterceptor" />
<bean name="helloBean" class="my.package.ProxyFactoryBean">
    <property name="proxylnterfaces">
        <value>org.groller.dynamicproxy.Hello</value>
    </property>
    <property name="interceptorNames">
          <value>timeInterceptor</value>
    </property>
    <property name="target">
        <ref Iocal="helloBeanTarget"/>
    </property>
</bean>
```

请注意，为了不影响客户端的使用，我们把这个新的bean也命名为`helloBean`，而原来的业务组件则被改名为`helloBeanTarget`，表示它是“被拦截的目标”。现在，我们再次运行客户端代码，控制台上出现了下列字样：

```
sayHello : Sun Dec 12 13:02:06 CST 2004
Hello, World!
```

说明我们的拦截器已经生效了。用同样的方法，我们可以把任何拦截器施加在任何业务组件之上，从而提供灵活的基础设施服务。而且，对于常用的基础设施，Spring框架已经提供了更便利的代理工厂，例如事务管理可以通过`TransactionProxyFactoryBean`获得，Hessian remoting可以通过`HessianProxyFactoryBean`获得。时至今日，J2EE架构师们已经一致同意：基础设施服务应该以AOP的形式提供，所以是否提供便利的AOP已经成为了挑选容器的重要条件之一。虽然动态AOP并非都是以动态代理技术实现，但其他实现技术

理技术的出现和发展，让基于J2EE世界的架构理念发生了根本性的变化，而Spring框架则是推动这场变革的一阵春风。

尾声

时间走到2004年底，关于动态代理的故事到这里已经讲完了，但故事并没有结束。就在12月8日，当笔者采访“UML三友”之一的Ivar Jacobson时，他略微透漏了自己对AOP的全新认识。在Jacobson看来，AOP已经可以上升到方法学的高度，它与用例的结合将迸发出更加耀眼的火花。大师的新作*Aspect-Oriented Software Development with Use Case*已经面世，我们即将看到AOP思想会给我们带来怎样的新惊喜。

很有趣地，每当想到AOP的眩目前景，每当看到AOP给我们带来的巨大变化，我总是禁不住暗想：如果没有2000年时Rickard Oberg对动态代理技术的顿悟，现在的AOP、乃至整个J2EE社群的技术潮流又会是什么样子呢？或许，这就是“文本”与“阐释者”之间宿命的渊源吧。

站长统计