探讨 Spring 框架与 AOP 思想的研究与应用

【摘 要】

本文对 Spring 框架中所包含的 AOP 思想以及事务管理进行了分析，并通过对一个业务对象实现加锁/解锁的操作，说明了动态代理模式的可行性与有效性。

引言

Aspect Oriented Programming(AOP)是近年来计算机技术中比较热门的话题之一。其发展历史从学术领域和研发机构的运用开始，目前流行的 Spring 应用程序框架将 AOP 思想融入了整个框架的设计开发与应用当中。使用 Spring 框架固然给我们的编程带来了好处与便利，但是同时存在着一个问题，对于初学者来说，所谓的“控制反转”，不是一个能够望文生义的好名称，“依赖注入”也是一样，也正是因为这样，不少初学者很难在短时间内理解和掌握这些名字和他们的用法，而要使用 AOP 的功能也需要理解 AOP，也比较难。基于以上原因，我们就会想到，能否简单地将 Spring 框架中运用到的优秀的理念，巧妙的运用到我们需要使用的地方，而又绕过不容易上手的 Spring 框架，做到一举两得呢？本文就将围绕着上述提出的问题给出作者的看法和观点。

AOP 思想与面向方面的编程

AOP 实际是 GoF 四人组设计模式的一种扩展，设计模式所追求的是降低代码之间的耦合度，增加程序的灵活性和可重用性，AOP 实际上就是设计模式所追求的目标的一种实现。所谓的分离关注就是将某一通用的需求功能从不相关的类之中分离出来；同时，能够使得很多类共享一个行为，一旦行为发生变化，不必修改很多类，只要修改这个行为就可以。AOP 就是这种实现分散关注的编程方法，它将“关注”封装在“方面”中。

面向对象的编程（OOP）方法是在面向过程的编程方法基础上进行的改进，而面向方面编程（AOP）方法又是在面向对象编程（OOP）方法的基础上进行改进而来的一种创新的软件开发方法。AOP 和 OOP 虽然在字面上十分相似，但是却是面向不同领域的两种设计思想。OOP（面向对象编程）针对问题领域中以及业务处理过程中存在的实体及其属性和操作进行抽象和封装，面向对象的核心概念是纵向结构的，其目的是获得更加清晰高效的逻辑单元划分；而 AOP 则是针对业务处理过程中的切面进行提取，例如，某一个操作在各个模块中都有涉及，这个操作就可以看成“横切”存在于系统当中。在许多情况下，这些操作都是与业务逻辑相关性不强或者不属于逻辑操作的必须部分，而面向对象的方法很难对这种情况做出处理。AOP 则将这些操作与业务逻辑分离，使程序员在编写程序时可以专注于业务逻辑的处理，而利用 AOP 将贯穿于各个模块间的横切关注点自动耦合进来。AOP 所面对的是处理过程中的某个步骤或阶段，对不同的阶段领域加以隔离，已获得逻辑过程中各部分之间低耦合性的隔离效果，其与面向方面编程在目标上有着本质的差异。AOP 的核心思想就是将应用程序中的业务逻辑处理部分同对其提供支持的通用服务，即所谓的“横切关注点”进行分离，这些“横切关注点”贯穿了程序中的多个纵向模块的需求。

使用 AOP 机制进行开发，首先要对方面进行了解，将需求分解成一般关注点和横切关注点，即将核心模块级的关注点和系统级的横切关注点分离；然后各自独立的实现这些关注点；最后用工具将业务逻辑代码和横切关注点代码编织到一起，形成最终的程序。通过面向方面的编程可以减少编码时间和重复。

目前已经形成的 Spring 框架

1、Spring 框架的特点

Spring 框架目前如此流行，一方面的原因在于 Spring 提供了一套全面并且十分成熟的轻型应用程序基本框架，并且对复杂的应用开发提供了有力的支持。除此之外，从实际应用开发角度来看，Spring 最大的优势在于它是从实际项目开发经验中抽取的，其提供了丰富的类库，可大大节省编码量，它是一种高效的、可高度重用的应用框架。Spring 框架中目前最吸引人也是该应用框架最具特色的地方就是名为控制反转（IOC=Inverse Of Control）或者依赖注入（DI=Dependence Injection）的设计思想，这是一种相当优秀的设计思想，即“好莱坞”原则：不用你主动来找我，我会通知你。但是，仅仅凭借着这样一个单纯的设计模式并不能使得 Spring 如此成功，Spring 最成功的地方，还是目前使用最为广泛的 AOP 应用，也就是 Spring 中基于 AOP 实现的业务管理机制，也正是由于这一点，使得 Spring AOP 成为应用框架中极其闪光的一个亮点。

2、AOP 思想在 Spring 框架中的体现

文章前面已经讲述了 AOP 的概念以及什么叫做所谓的“横切”关注点，事务管理就是 J2EE 应用中一个横切多个对象的横切关注点的例子。

2.1 事务管理

对于 J2EE 应用程序而言，事务的处理一般有两种模式：依赖特定事务资源的事务处理与依赖容器的参数化事务管理。在这里我们略去对第一种处理方式的说明，直接对第二种方式，即依赖容器的参数化事务管理来阐述笔者的观点。

Spring 事务管理究竟能带给我们什么？

了解 Spring 的人们都知道，对于传统的基于事务资源的事务处理而言，Spring 并不会产生什么影响，我们照样可以成功编写并且运行这样的代码。

对于依赖容器的参数化事务管理而言，Spring 则可以用来帮助实现对事务的管理而无须使用 EJB。Spring 本身也是一个容器，只是相对 EJB 容器所要付出的代价而言，Spring 属于轻量级容器，它能够替代 EJB，通过使用 AOP 来提供声明式事务管理，即可通过 Spring 实现基于容器的事务管理（从本质上来讲，Spring 的事务管理是基于动态 AOP）。Spring 与 EJB 最大的区别在于：

第一，Spring 可以为任意的 Java Class 实现事务管理而无须转换成标准的 EJB；

第二，Spring 事务管理并不依赖特定的事务资源从而使得系统的应用与部署更佳灵活。

2.2 动态代理机制的实现

Spring 框架中所提供的 AOP 支持，是基于动态 AOP 机制实现的，即通过动态 Proxy 模式，在目标对象的方法调用前后插入相应的处理代码。AOP 代理可以是基于 JDK 动态代理，也可以是基于 CGLIB 代理。Spring 默认使用的是基于 Java Dynamic Proxy 模式实现，这样任何的接口都能被代理。基于 Spirng 框架的应用程序开发，程序员会有一种自然的倾向性来实现面向接口编程而不是类，业务对象通常也是实现一个或者多个接口，这也是一种良好的编程习惯。Spring 也可以基于 CGLIB 实现 AOP 代理，这样所代理的是类而不是接口。如果一个业务对象没有实现某一个接口，那么 CGLIB 将被使用。

我们先来分析一下 Spring 事务管理机制的实现原理。由于 Spring 内置 AOP 默认使用动态代理模式实现，我们就先来分析一下动态代理模式的实现方法。动态代理模式的核心就在于代码中不出现与具体应用层相关联的接口或者类引用，如上所说，这个代理类适用于任何接口的实现。下面我们来看一个例子。

```
public class TxHandler implements InvocationHandler {
private Object originalObject;
public Object bind(Object obj) {
    this.originalObject = obj;
    return Proxy.newProxyInstance(obj.getClass().getClassLoader(),obj.getClass().getInterfaces(),this);
}

public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
    Object result = null;
    if (!method.getName().startsWith("save")) {
      UserTransaction tx = null;
      try {
        tx = (UserTransaction) (new InitialContext().lookup("java/tx"));
        result = method.invoke(originalObject, args);
        tx.commit();
      } catch (Exception ex) {
        if (null != tx) {
          try {
            tx.rollback();
          } catch (Exception e) {
        }
      }
    }
} else {
    result = method.invoke(originalObject, args);
}
return result;
}
}
```

下面我们来分析一下上述代码的关键所在。

首先来看一下这段代码：

```
return Proxy.newProxyInstance(
    obj.getClass().getClassLoader(),obj.getClass().getInterfaces(),this);
```

java.lang.reflect.Proxy.newProxyInstance 方法根据传入的接口类型（obj.getClass.getInterfaces()）动态构造一个代理类实例返回，这也说明了为什么动态代理实现要求其所代理的对象一定要实现一个接口。这个代理类实例在内存中是动态构造的，它实现了传入的接口列表中所包含的所有接口。

再来分析以下代码：

```
public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
    ……
    result = method.invoke(originalObject, args);
    ……
    return result;
}
```

InvocationHandler.invoke 方法将在被代理类的方法被调用之前触发。通过这个方法，我们可以在被代理类方法调用的前后进行一些处理，如代码中所示，InvocationHandler.invoke 方法的参数中传递了当前被调用的方法（Method）,以及被调用方法的参数。同时，可以通过 method.invoke 方法调用被代理类的原始方法实现。这样就可以在被代理类的方法调用前后写入任何想要进行的操作。

Spring 的事务管理机制实现的原理，就是通过这样一个动态代理对所有需要事务管理的 Bean 进行加载，并根据配置在 invoke 方法中对当前调用的方法名进行判定，并在 method.invoke 方法前后为其加上合适的事务管理代码，这样就实现了 Spring 式的事务管理。Spring 中的 AOP 实现更为复杂和灵活，不过基本原理是一致的。

3.AOP 思想与动态代理模式的应用实例

综上我们分析了 Spring 框架的事务管理机制的基本现实原理。尽管 Spring 框架集中体现了当前流行框架中未曾关注到的一些内容，但是，Spring 框架存在晦涩难懂的致命问题。以上通过对 Spring 框架的一些基本实现原理的研究，给我们带来了一些启示。我们如果不直接使用庞大的 Spring 框架，而是将融入 Spring 框架中的 AOP 思想直接应用于程序中，既绕过了 Spring 框架这个高门槛，又利用了 Spring 框架中先进的设计理念，这样便达到了一举两得的目的。

下面我们来看一段代码，我们来编写一个 Dynamic Proxy based AOP 实现的实例。假设现在有一个 UserDao 接口和以及其实现类 UserDaoImp。

```
UserDAO.java:

public interface UserDAO {
```

```
    public void saveUser(User user);
}

UserDAOImp.java:

public class UserDAOImp implements UserDAO{
    public void saveUser(User user) {
        ……
    }
}
```

我们需要在 saveUser 方法中添加对一个业务对象的锁，比如在 saveUser 前后加锁和解锁。在不影响外部逻辑和不对现有的代码做任何改动的前提下，代理模式是一个不错的选择。但是如果有多个类似的接口，面对每个接口都要实现一个类似的 Proxy，实在是一个烦琐无味的苦力过程。回想一下 Spring 在处理这个问题上的设计理念我们不难想到，使用动态代理模式，是这个问题的一个聪明的解决方法。

```
public class AOPHandler implements InvocationHandler {
    private static Log logger = LogFactory.getLog(AOPHandler.class);
    private List interceptors = null;
    private Object originalObject;
    public Object bind(Object obj) {
        this.originalObject = obj;
        return Proxy.newProxyInstance(obj.getClass().getClassLoader(),obj.getClass().getInterfaces().this);
    }
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable{Object result=null
        if(method.getName.startsWith( “saveUser” )){
            lock();
            result=method.invoke(this.originalObject,args);
            unlock();
        }
        return result;
    }
    private void lock(){
        logger.info( “lock object!” );
    }
    private void unlock(){
        logger.info( “unlock object!” );
    }
}
```

上述代码中并没有出现与具体应用层相关联的接口以及类的引用，所以对所有的类都适用。这就解决了用静态 Proxy 类实现所产生的弊端。

总结与展望

以上我们讨论了 Spring 框架基于动态 AOP 机制实现以及动态代理机制的应用，围绕着 AOP 的实现与应用，一直有一个热门的话题，即权限管理。Spring 框架目前对 AOP 的支持做得相当出色，但是一直有一个尴尬的问题尚未解决，那就是目前

还没有一个完备的权限管理组件。仔细想想，这并不是 AOP 的瓶颈，而是因为权限管理的形式过于灵活和复杂多变，系统中的权限管理逻辑多种多样各不相同，我们很难做出一个统一的管理与操作。另一方面，权限管理作为一个独立的切面显得过于庞大，需要进一步切分设计，设计过程复杂，实现难度较大。所以我们还在期待着 AOP 思想在权限管理方面能有突破性的应用与扩展。

使用 SPRING AOP 框架和 EJB 组件

摘要

快速发展的开发人员社区、对各种后端技术（包括 JMS、JTA、JDO、Hibernate、iBATIS 等等）的支持，以及（更为重要的）非侵入性的轻量级 IoC 容器和内置的 AOP 运行时，这些因素使得 Spring Framework 对于 J2EE 应用程序开发十分具有吸引力。Spring 托管的组件（POJO）可以与 EJB 共存，并允许使用 AOP 方法来处理企业应用程序中的横切方面——从监控和审计、缓存及应用程序级的安全性开始，直到处理特定于应用程序的业务需求。

本文将向您介绍 Spring 的 AOP 框架在 J2EE 应用程序中的实际应用。

简介

J2EE 技术为实现服务器端和中间件应用程序提供了坚实的基础。J2EE 容器（比如 BEA WebLogic Server）可以管理系统级的元素，包括应用程序生命周期、安全性、事务、远程控制和并发性，而且它可以保证为 JDBC、JMS 和 JTA 之类的常见服务提供支持。然而，J2EE 的庞大和复杂性使开发和测试变得异常困难。传统的 J2EE 应用程序通常严重依赖于通过容器的 JNDI 才可用的服务。这意味着需要大量直接的 JNDI 查找，或者要使用 Service Locator 模式，后者稍微有所改进。这种架构提高了组件之间的耦合度，并使得单独测试某个组件成为几乎不可能实现的事情。您可以阅读 Spring Framework 创建者所撰写的 J2EE Development without EJB 一书，其中深入分析了这种架构的缺陷。

借助于 Spring Framework，可以将使用无格式 Java 对象实现的业务逻辑与传统的 J2EE 基础架构连接起来，同时极大地减少了访问 J2EE 组件和服务所需的代码量。基于这一点，可以把传统的 OO 设计与正交的 AOP 组件化结合在一起。本文稍后将会演示如何重构 J2EE 组件以利用 Spring 托管的 Java 对象，然后应用一种 AOP 方法来实现新特性，从而维护良好的组件独立性和可测试性。

与其他 AOP 工具相比，Spring 提供了 AOP 功能中的一个有限子集。它的目标是紧密地集成 AOP 实现与 Spring IoC 容器，从而帮助解决常见的应用问题。该集成是以非侵入性的方式完成的，它允许在同一个应用程序中混合使用 Spring AOP 和表现力更强的框架，包括 AspectJ。Spring AOP 使用无格式 Java 类，不要求特殊的编译过程、控制类装载器层次结构或更改部署配置，而是使用 Proxy 模式向应该由 Spring IoC 容器托管的目标对象应用通知。

可以根据具体情况在两种类型的代理之间进行选择：

- 第一类代理基于 Java 动态代理，只适用于接口。它是一种标准的 Java 特性，可提供卓越的性能。
- 第二类代理可用于目标对象没有实现任何接口的场景，而且这类接口不能被引入（例如，对于遗留代码的情况）。它基于使用 CGLIB 库的运行时字节码生成。

对于所代理的对象，Spring 允许使用静态的（方法匹配基于确切名称或正则表达式，或者是注释驱动的）或动态的（匹配是在运行时进行的，包括 cflow 切入点类型）切入点定义指派特定的通知，而每个切入点可以与一条或多条通知关联在一起。所支持的通知类型有几种：环绕通知（around advice），前通知（before advice），返回后通知（after returning advice），抛出异常后通知（after throwing advice），以及引入通知（introduction advice）。本文稍后将给出环绕通知的一个例子。想要了解更详细的信息，可以参考 Spring AOP 框架文档。

正如先前提到的那样，只可以通知由 Spring IoC 容器托管的目标对象。然而，在 J2EE 应用程序中，组件的生命周期是由应用服务器托管的，而且根据集成类型，可以使用一种常见的端点类型把 J2EE 应用程序组件公开给远程或本地的客户端：

- 无状态的、有状态的或实体 bean，本地的或远程的（基于 RMI-IIOP）
- 监听本地或外部 JMS 队列和主题或入站 JCA 端点的消息驱动 bean(MDB)
- Servlet(包括 Struts 或其他终端用户 UI 框架、XML-RPC 和基于 SOAP 的接口)

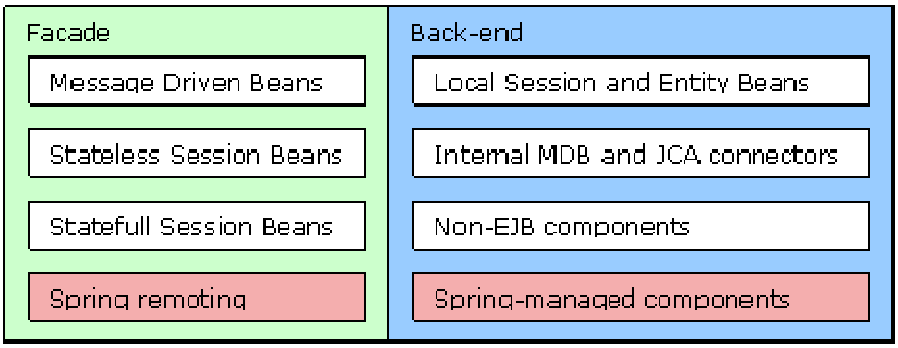

图 1.常见的端点类型

要在这些端点上使用 Spring 的 AOP 框架，必须把所有的业务逻辑转移到 Spring 托管的 bean 中，然后使用服务器托管的组件来委托调用，或者定义事务划分和安全上下文。虽然本文不讨论事务方面的问题，但是可以在“参考资料”部分中找到相关文章。

我将详细介绍如何重构 J2EE 应用程序以使用 Spring 功能。我们将使用 XDoclet 的基于 JavaDoc 的元数据来生成 home 和 bean 接口，以及 EJB 部署描述符。可以在下面的“下载”部分中找到本文中所有示例类的源代码。

重构 EJB 组件以使用 Spring 的 EJB 类

想像一个简单的股票报价 EJB 组件，它返回当前的股票交易价格，并允许设置新的交易价格。这个例子用于说明同时使用 Spring Framework 与 J2EE 服务的各个集成方面和最佳实践，而不是要展示如何编写股票管理应用程序。按照我们的要求，TradeManager 业务接口应该就是下面这个样子：

```
public interface TradeManager {
  public static String ID = "tradeManager";

  public BigDecimal getPrice(String name);
```

```
  public void setPrice(String name, BigDecimal price);

}
```

在设计 J2EE 应用程序的过程中，通常使用远程无状态会话 bean 作为持久层中的外观和实体 bean。下面的 TradeManager1Impl 说明了无状态会话 bean 中 TradeManager 接口的可能实现。注意，它使用了 ServiceLocator 来为本地的实体 bean 查找 home 接口。XDoclet 注释用于为 EJB 描述符声明参数以及定义 EJB 组件的已公开方法。

```
/**
 * @ejb.bean
 *     name="org.javatx.spring.aop.TradeManager1"
 *     type="Stateless"
 *     view-type="both"
 *     transaction-type="Container"
 *
 * @ejb.transaction type="NotSupported"
 *
 * @ejb.home
 *     remote-pattern="{0}Home"
 *     local-pattern="{0}LocalHome"
 *
 * @ejb.interface
 *     remote-pattern="{0}"
 *     local-pattern="{0}Local"
 */
public class TradeManager1Impl implements SessionBean, TradeManager {
  private SessionContext ctx;

  private TradeLocalHome tradeHome;



  /**
   * @ejb.interface-method view-type="both"
   */
  public BigDecimal getPrice(String symbol) {
     try {
        return tradeHome.findByPrimaryKey(symbol).getPrice();
     } catch(ObjectNotFoundException ex) {
```

```
        return null;
    } catch(FinderException ex) {
        throw new EJBException("Unable to find symbol", ex);
    }
}

/**
 * @ejb.interface-method view-type="both"
 */
public void setPrice(String symbol, BigDecimal price) {
    try {
        try {
            tradeHome.findByPrimaryKey(symbol).setPrice(price);
        } catch(ObjectNotFoundException ex) {
            tradeHome.create(symbol, price);
        }
    } catch(CreateException ex) {
        throw new EJBException("Unable to create symbol", ex);
    } catch(FinderException ex) {
        throw new EJBException("Unable to find symbol", ex);
    }
}


public void ejbCreate() throws EJBException {
    tradeHome = ServiceLocator.getTradeLocalHome();
}

public void ejbActivate() throws EJBException, RemoteException {
}

public void ejbPassivate() throws EJBException, RemoteException {
}

public void ejbRemove() throws EJBException, RemoteException {
}
```

```
  public void setSessionContext(SessionContext ctx) throws EJBException, RemoteException {
    this.ctx = ctx;
  }

}
```

如果要在进行代码更改之后测试这样一个组件，那么在运行任何测试（通常是基于专用的容器内测试框架，比如 Cactus 或 MockEJB）之前，必须要经过构建、启动容器和部署应用程序这整个周期。虽然在简单的用例中类的热部署可以节省重新部署的时间，但是当类模式变动（例如，添加域或方法，或者修改方法名）之后它就不行了。这个问题本身就是把所有逻辑转移到无格式 Java 对象中的最好理由。正如您在 TradeManager1Impl 代码中所看到的那样，大量的粘和代码把 EJB 中的所有内容组合在一起，而且您无法从围绕 JNDI 访问和异常处理的复制工作中抽身。然而，Spring 提供抽象的便利类，可以使用定制的 EJB bean 对它进行扩展，而无需直接实现 J2EE 接口。这些抽象的超类允许移除定制 bean 中的大多数粘和代码，而且提供用于获取 Spring 应用程序上下文的实例的方法。

首先，需要把 TradeManager1Impl 中的所有逻辑都转移到新的无格式 Java 类中，这个新的类还实现了一个 TradeManager 接口。我们将把实体 bean 作为一种持久性机制，这不仅因为它超出了本文的讨论范围，还因为 WebLogic Server 提供了大量用于调优 CMP bean 性能的选项。在特定的用例中，这些 bean 可以提供非常好的性能（请参见“参考资料”部分中有关 CMP 性能调优的文章）。我们还将使用 Spring IoC 容器把 TradeImpl 实体 bean 的 home 接口注入到 TradeDao 的构造函数中，您将从下面的代码中看到这一点：

```
public class TradeDao implements TradeManager {
  private TradeLocalHome tradeHome;

  public TradeDao(TradeLocalHome tradeHome) {
    this.tradeHome = tradeHome;
  }

  public BigDecimal getPrice(String symbol) {
    try {
      return tradeHome.findByPrimaryKey(symbol).getPrice();
    } catch(ObjectNotFoundException ex) {
      return null;
    } catch(FinderException ex) {
      throw new EJBException("Unable to find symbol", ex);
    }
  }

  public void setPrice(String symbol, BigDecimal price) {
```

```
    try {
      try {
        tradeHome.findByPrimaryKey(symbol).setPrice(price);
      } catch(ObjectNotFoundException ex) {
        tradeHome.create(symbol, price);
      }
    } catch(CreateException ex) {
      throw new EJBException("Unable to create symbol", ex);
    } catch(FinderException ex) {
      throw new EJBException("Unable to find symbol", ex);
    }
  }

}
```

现在，可以使用 Spring 的 AbstractStatelessSessionBean 抽象类重写 TradeManager1Impl，该抽象类还可以帮助您获得上面所创建的 TradeDao bean 的一个 Spring 托管的实例：

```
/**
 * @ejb.home
 *    remote-pattern="TradeManager2Home"
 *    local-pattern="TradeManager2LocalHome"
 *    extends="javax.ejb.EJBHome"
 *    local-extends="javax.ejb.EJBLocalHome"
 *
 * @ejb.transaction type="NotSupported"
 *
 * @ejb.interface
 *    remote-pattern="TradeManager2"
 *    local-pattern="TradeManager2Local"
 *    extends="javax.ejb.SessionBean"
 *    local-extends="javax.ejb.SessionBean, org.javatx.spring.aop.TradeManager"
 *
 * @ejb.env-entry
 *    name="BeanFactoryPath"
 *    value="applicationContext.xml"
 */
public class TradeManager2Impl extends AbstractStatelessSessionBean implements TradeManager {
```

```
  private TradeManager tradeManager;

  public void setSessionContext(SessionContext sessionContext) {
      super.setSessionContext(sessionContext);
      // make sure there will be the only one Spring bean config
      setBeanFactoryLocator(ContextSingletonBeanFactoryLocator.getInstance());
  }

  public void onEjbCreate() throws CreateException {
     tradeManager = (TradeManager) getBeanFactory().getBean(TradeManager.ID);
  }

  /**
   * @ejb.interface-method view-type="both"
   */
  public BigDecimal getPrice(String symbol) {
     return tradeManager.getPrice(symbol);
  }

  /**
   * @ejb.interface-method view-type="both"
   */
  public void setPrice(String symbol, BigDecimal price) {
     tradeManager.setPrice(symbol, price);
  }

}
```

现在，EJB 把所有调用都委托给在 onEjbCreate()方法中从 Spring 获得的 TradeManager 实例，这个方法是在 AbstractEnterpriseBean 中实现的，它处理所有查找和创建 Spring 应用程序上下文所需的工作。但是，必须在 EJB 部署描述符中为 EJB 声明 BeanFactoryPath env-entry，以便将配置文件和 bean 声明的位置告诉 Spring。上面的例子使用了 XDoclet 注释来生成这些信息。

此外还要注意，我们重写了 setSessionContext()方法，以便告诉 AbstractStatelessSessionBean 跨所有 EJB bean 使用 Sping 应用程序上下文的单个实例。

现在，可以在 applicationContext.xml 中声明一个 tradeManager bean。基本上需要创建一个上面 TradeDao 的新实例，把从 JNDI 获得的 TradeLocalHome 实例传递给它的构造函数。下面给出了可能的定义：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "spring-beans.dtd">

<beans>

  <bean id="tradeManager" class="org.javatx.spring.aop.TradeDao">
    <constructor-arg index="0">
      <bean class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName">
          <bean id="org.javatx.spring.aop.TradeLocalHome.JNDI_NAME"
                class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean"/>
        </property>
        <property name="proxyInterface" value="org.javatx.spring.aop.TradeLocalHome"/>
      </bean>
    </constructor-arg>
  </bean>

</beans>
```

在这里，我们使用了一个匿名定义的 TradeLocalHome 实例，这个实例是使用 Spring 的 JndiObjectFactoryBean 从 JNDI 获得的，然后把它作为一个构造函数参数注入到 tradeManager 中。我们还使用了一个 FieldRetrievingFactoryBean 来避免硬编码 TradeLocalHome 的实际 JNDI 名称，而是从静态的域（在这个例子中为 TradeLocalHome.JNDI_NAME）获取它。通常，使用 JndiObjectFactoryBean 时声明 proxyInterface 属性是一个不错的主意，如上面的例子所示。

还有另一种简单的方法可以访问会话 bean。Spring 提供一个 LocalStatelessSessionProxyFactoryBean，它允许立刻获得一个会话 bean 而无需经过 home 接口。例如，下面的代码说明了如何使用通过 Spring 托管的另一个 bean 中的本地接口访问的 MyComponentImpl 会话 bean：

```
<bean id="tradeManagerEjb"
        class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
    <property name="jndiName">
      <bean id="org.javatx.spring.aop.TradeManager2LocalHome.JNDI_NAME"
            class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean"/>
    </property>
    <property name="businessInterface" value="org.javatx.spring.aop.TradeManager"/>
</bean>
```

这种方法的优点在于，可以很容易地从本地接口切换到远程接口，只要使用 SimpleRemoteStatelessSessionProxyFactoryBean 修改 Spring 上下文中的一处 bean 声明即可。例如：

```
<bean id="tradeManagerEjb"
        class="org.springframework.ejb.access.SimpleRemoteStatelessSessionProxyFactoryBean">
    <property name="jndiName">
      <bean id="org.javatx.spring.aop.TradeManager2Home.JNDI_NAME"
            class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean"/>
    </property>
    <property name="businessInterface" value="org.javatx.spring.aop.TradeManager"/>
    <property name="lookupHomeOnStartup" value="false"/>
  </bean>
```

注意，lookupHomeOnStartup property 被设置为 false，以支持延迟初始化。

下面，我总结一下到此为止所学习的内容：

- 上面的重构已经为使用高级的 Spring 功能（也就是依赖性注入和 AOP）奠定了基础。
- 在没有修改客户端 API 的情况下，我把所有业务逻辑都移出外观会话 bean，这就使得这个 EJB 不惧修改，而且易于测试。
- 业务逻辑现在位于一个无格式 Java 对象中，只要该 Java 对象的依赖性不需要 JNDI 中的资源，就可以在容器外部对其进行测试，或者可以使用存根或模仿（mock）来代替这些依赖性。
- 现在，可以代入不同的 tradeManager 实现，或者修改初始化参数和相关组件，而无需修改 Java 代码。

至此，我们已经完成了所有准备步骤，可以开始解决对 TradeManager 服务的新需求了。

通知由 Spring 托管的组件

在前面的内容中，我们重构了服务入口点，以便使用 Spring 托管的 bean。现在，我将向您说明这样做将如何帮助改进组件和实现新功能。

首先，假定用户想看到某些符号的价格，而这些价格并非由您的 TradeManager 组件所托管。换句话说，您需要连接到一个外部服务，以便获得当前您不处理的所请求符号的当前市场价格。您可以使用雅虎门户中的一个基于 HTTP 的免费服务，但是实际的应用程序将连接到提供实时数据的供应商（比如 Reuters、Thomson、Bloomberg、NAQ 等等）的实时数据更新服务（data feed）。

首先，需要创建一个新的 YahooFeed 组件，该组件实现了相同的 TradeManager 接口，然后从雅虎金融门户获得价格信息。自然的实现可以使用 HttpURLConnection 发送一个 HTTP 请求，然后使用正则表达式解析响应。例如：

```
public class YahooFeed implements TradeManager {
  private static final String SERVICE_URL = "http://finance.yahoo.com/d/quotes.csv?f=k1&s=";
```

```
private Pattern pattern = Pattern.compile("\"(.*) - (.*)\"");

public BigDecimal getPrice(String symbol) {
    HttpURLConnection conn;
    String responseMessage;
    int responseCode;
    try {
        URL serviceUrl = new URL(SERVICE_URL+symbol);
        conn = (HttpURLConnection) serviceUrl.openConnection();
        responseCode = conn.getResponseCode();
        responseMessage = conn.getResponseMessage();
    } catch(Exception ex) {
        throw new RuntimeException("Connection error", ex);
    }

    if(responseCode!=HttpURLConnection.HTTP_OK) {
        throw new RuntimeException("Connection error "+responseCode+" "+responseMessage);
    }

    String response = readResponse(conn);
    Matcher matcher = pattern.matcher(response);
    if(!matcher.find()) {
        throw new RuntimeException("Unable to parse response ["+response+"] for symbol "+symbol);
    }
    String time = matcher.group(1);
    if("N/A".equals(time)) {
        return null;   // unknown symbol
    }
    String price = matcher.group(2);
    return new BigDecimal(price);
}

public void setPrice(String symbol, BigDecimal price) {
    throw new UnsupportedOperationException("Can't set price of 3rd party trade");
}

private String readResponse(HttpURLConnection conn) {
```

```
    // ...
    return response;
  }

}
```

完成这种实现并测试(在容器外部!)之后,就可以把它与其他组件进行集成。传统的做法是向 TradeManager2Impl 添加一些代码,以便检查 getPrice()方法返回的值。这会使测试的次数至少增加一倍,而且要求为每个测试用例设定附加的先决条件。然而,如果使用 Spring AOP 框架,就可以更漂亮地完成这项工作。您可以实现一条通知,如果初始的 TradeManager 没有返回所请求符号的值,该通知将使用 YahooFeed 组件来获取价格(在这种情况下,它的值是 null,但是也可能会得到一个 UnknownSymbol 异常)。

要把通知应用到具体的方法,需要在 Spring 的 bean 配置中声明一个 Advisor。有一个方便的类叫做 NameMatchMethodPointcutAdvisor,它允许通过名称选择方法,在本例中还需要一个 getPrice 方法:

```
<bean id="yahooFeed" class="org.javatx.spring.aop.YahooFeed"/>

  <bean id="foreignTradeAdvisor"
          class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
    <property name="mappedName" value="getPrice"/>
    <property name="advice">
      <bean class="org.javatx.spring.aop.ForeignTradeAdvice">
        <constructor-arg index="0" ref="yahooFeed"/>
      </bean>
    </property>
  </bean>
```

正如您所看到的,上面的 advisor 指派了一个 ForeignTradeAdvice 给 getPrice()方法。针对通知类,Spring AOP 框架使用了 AOP Alliance API,这意味着环绕通知的 ForeignTradeAdvice 应该实现 MethodInterceptor 接口。例如:

```
public class ForeignTradeAdvice implements MethodInterceptor {
  private TradeManager tradeManager;

  public ForeignTradeAdvice(TradeManager manager) {
    this.tradeManager = manager;
  }

  public Object invoke(MethodInvocation invocation) throws Throwable {
    Object res = invocation.proceed();
    if(res!=null) return res;
```

```
    Object[] args = invocation.getArguments();
    String symbol = (String) args[0];
    return tradeManager.getPrice(symbol);
  }

}
```

上面的代码使用 invocation.proceed()调用了一个原始的组件，而且如果它返回 null，它将调用另一个在通知创建时作为构造函数参数注入的 tradeManager。参见上面 foreignTradeAdvisor bean 的声明。

现在可以把在 Spring 的 bean 配置中定义的 tradeManager 重新命名为 baseTradeManager，然后使用 ProxyFactoryBean 把 tradeManager 声明为一个代理。新的 baseTradeManager 将成为一个目标，我们将使用上面定义的 foreignTradeAdvisor 通知它：

```
<bean id="baseTradeManager" class="org.javatx.spring.aop.TradeDao">
    ... same as tradeManager definition in the above example
  </bean>

  <bean id="tradeManager" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces" value="org.javatx.spring.aop.TradeManager"/>
    <property name="target" ref="baseTradeManager"/>
    <property name="interceptorNames">
      <list>
        <idref local="foreignTradeAdvisor"/>
      </list>
    </property>
  </bean>
```

基本上，就是这样了。我们实现了附加的功能而没有修改原始的组件，而且仅使用 Spring 应用程序上下文来重新配置依赖性。要想不借助于 Spring AOP 框架在典型的 EJB 组件中实现类似的修改，要么必须为 EJB 添加附加的逻辑（这会使其难以测试），要么必须使用 decorator 模式（实际上增加了 EJB 的数量，同时也提高了测试的复杂性，延长了部署时间）。在上面的例子中，您可以看到，借助于 Spring，可以轻松地不修改现有组件而向这些组件添加附加的逻辑。现在，您拥有的是几个轻量级组件，而不是紧密耦合的 bean，您可以独立测试它们，使用 Spring Framework 组装它们。注意，使用这种方法，ForeignTradeAdvice 就是一个自包含的组件，它实现了自己的功能片断，可以当作一个独立单元在应用服务器外部进行测试，下面我将对此进行说明。

测试通知代码

您可能注意到了，代码不依赖于 TradeDao 或 YahooFeed。这样就可以使用模仿对象完全独立地测试这个组件。模仿对象测试方法允许在组件执行之前声明期望，然后验证这些期望在组件调用期间是否得到满足。要了解有关模仿测试的更多信息，请参见“参考资料”部分。下面我们将会使用 jMock 框架，该框架提供了一个灵活且功能强大的 API 来声明期望。

测试和实际的应用程序使用相同的 Spring bean 配置是个不错的主意，但是对于特定组件的测试来说，不能使用实际的依赖性，因为这会破坏组件的孤立性。然而，Spring 允许在创建 Spring 的应用程序上下文时指定一个 BeanPostProcessor，从而置换选中的 bean 和依赖性。在这个例子中，可以使用模仿对象的一个 Map，这些模仿对象是在测试代码中创建的，用于置换 Spring 配置中的 bean：

```
public class StubPostProcessor implements BeanPostProcessor {
  private final Map stubs;

  public StubPostProcessor( Map stubs) {
    this.stubs = stubs;
  }

  public Object postProcessBeforeInitialization(Object bean, String beanName) {
    if(stubs.containsKey(beanName)) return stubs.get(beanName);
    return bean;
  }

  public Object postProcessAfterInitialization(Object bean, String beanName) {
    return bean;
  }

}
```

在测试用例类的 setUp()方法中，我们将使用 baseTradeManager 和 yahooFeed 组件的模仿对象来初始化 StubPostProcessor，而这两个组件是使用 jMock API 创建的。然后，我们就可以创建 ClassPathXmlApplicationContext（配置其使用 BeanPostProcessor）来实例化一个 tradeManager 组件。产生的 tradeManager 组件将使用模仿后的依赖性。

这种方法不仅允许孤立要测试的组件，还可以确保在 Spring bean 配置中正确定义通知。实际上，要在不模拟大量容器基础架构的情况下使用这样的方法来测试在 EJB 组件中实现的业务逻辑是不可能的：

```
public class ForeignTradeAdviceTest extends TestCase {
  TradeManager tradeManager;
  private Mock baseTradeManagerMock;
  private Mock yahooFeedMock;
```

```
  protected void setUp() throws Exception {
    super.setUp();

    baseTradeManagerMock = new Mock(TradeManager.class, "baseTradeManager");
    TradeManager baseTradeManager = (TradeManager) baseTradeManagerMock.proxy();

    yahooFeedMock = new Mock(TradeManager.class, "yahooFeed");
    TradeManager yahooFeed = (TradeManager) yahooFeedMock.proxy();

    Map stubs = new HashMap();
    stubs.put("yahooFeed", yahooFeed);
    stubs.put("baseTradeManager", baseTradeManager);

    ConfigurableApplicationContext ctx = new ClassPathXmlApplicationContext(CTX_NAME);
    ctx.getBeanFactory().addBeanPostProcessor(new StubPostProcessor(stubs));

    tradeManager = (TradeManager) proxyFactory.getProxy();
  }
  ...
```

在实际的 testAdvice()方法中，可以为模仿对象指定期望并验证（例如）baseTradeManager 上的 getPrice()方法是否返回 null，然后 yahooFeed 上的 getPrice()方法也将被调用：

```
public void testAdvice() throws Throwable {
    String symbol = "testSymbol";
    BigDecimal expectedPrice = new BigDecimal("0.222");

    baseTradeManagerMock.expects(new InvokeOnceMatcher()).method("getPrice")
      .with(new IsEqual(symbol)).will(new ReturnStub(null));

    yahooFeedMock.expects(new InvokeOnceMatcher()).method("getPrice")
      .with(new IsEqual(symbol)).will(new ReturnStub(expectedPrice));

    BigDecimal price = tradeManager.getPrice(symbol);
    assertEquals("Invalid price", expectedPrice, price);
        baseTradeManagerMock.verify();
    yahooFeedMock.verify();
```

```
}
```

这段代码使用 jMock 约束来指定，baseTradeManagerMock 期望只使用一个等于 symbol 的参数调用 getPrice()方法一次，而且这次调用将返回 null。类似地，yahooFeedMock 也期望对同一方法只调用一次，但是返回 expectedPrice。这允许在 setUp()方法中运行所创建的 tradeManager 组件，并断言返回的结果。

这个测试用例很容易参数化，从而涵盖所有可能的用例。注意，当组件抛出异常时，可以很容易地声明期望。

测试	baseTradeManager			yahooFeed			期望	
	调用	返回	抛出	调用	返回	抛出	结果 t	异常
1	true	0.22	-	false	-	-	0.22	-
2	true	-	e1	false	-	-	-	e1
3	true	null	-	true	0.33	-	0.33	-
4	true	null	-	true	null	-	null	-
5	true	null	-	true	-	e2	-	e2

可以使用这个表更新测试类，使其使用一个涵盖了所有可能场景的参数化序列，：

```
...

  public static TestSuite suite() {
    BigDecimal v1 = new BigDecimal("0.22");
    BigDecimal v2 = new BigDecimal("0.33");

    RuntimeException e1 = new RuntimeException("e1");
    RuntimeException e2 = new RuntimeException("e2");

    TestSuite suite = new TestSuite(ForeignTradeAdviceTest.class.getName());
    suite.addTest(new ForeignTradeAdviceTest(true, v1,     null, false, null, null, v1,     null));
    suite.addTest(new ForeignTradeAdviceTest(true, null, e1,     false, null, null, null, e1));
    suite.addTest(new ForeignTradeAdviceTest(true, null, null, true,   v2,     null, v2,     null));
    suite.addTest(new ForeignTradeAdviceTest(true, null, null, true,   null, null, null, null));
    suite.addTest(new ForeignTradeAdviceTest(true, null, null, true,   null, e2,     null, e2));
    return suite;
  }

  public ForeignTradeAdviceTest(
      boolean baseCall, BigDecimal baseValue, Throwable baseException,
      boolean yahooCall, BigDecimal yahooValue, Throwable yahooException,
```

```
        BigDecimal expectedValue, Throwable expectedException) {
    super("test");

    this.baseCall = baseCall;
    this.baseWill = baseException==null ?
            (Stub) new ReturnStub(baseValue) : new ThrowStub(baseException);
    this.yahooCall = yahooCall;
    this.yahooWill = yahooException==null ?
            (Stub) new ReturnStub(yahooValue) : new ThrowStub(yahooException);
    this.expectedValue = expectedValue;
    this.expectedException = expectedException;
}

public void test() throws Throwable {
    String symbol = "testSymbol";

    if(baseCall) {
        baseTradeManagerMock.expects(new InvokeOnceMatcher())
            .method("getPrice").with(new IsEqual(symbol)).will(baseWill);
    }

    if(yahooCall) {
        yahooFeedMock.expects(new InvokeOnceMatcher())
            .method("getPrice").with(new IsEqual(symbol)).will(yahooWill);
    }

    try {
        BigDecimal price = tradeManager.getPrice(symbol);
        assertEquals("Invalid price", expectedValue, price);
    } catch(Exception e) {
        if(expectedException==null) {
            throw e;
        }
    }
            baseTradeManagerMock.verify();
    yahooFeedMock.verify();
}
```

```
public String getName() {
  return super.getName()+" "+
    baseCalled+" "+baseValue+" "+baseException+" "+
    yahooCalled+" "+yahooValue+" "+yahooException+" "+
    expectedValue+" "+expectedException;
}
...
```

在更复杂的情况下，上面的测试方法可以很容易地扩展为大得多的输入参数集合，而且它仍然会立刻运行且易于管理。此外，把所有参数移入一个外部配置文件或者甚至 Excel 电子表格是合理的做法，这些配置文件或电子表格可以由 QA 团队管理，或者直接根据需求生成。

组合和链接通知

我们已经使用了一个简单的拦截器通知来实现附加的逻辑，并且将其当作一个独立的组件进行了测试。当应该在不进行修改并且与其他组件没有附加耦合的情况下扩展公共执行流时，这种设计十分有效。例如，当价格已经发生变化时，如果需要使用 JMS 或 JavaMail 发送通知，我们可以在 tradeManager bean 的 setPrice 方法上注册另一个拦截器，并使用它来向相关组件通知有关这些变化的情况。在很多情况下，这些方面都适用于非功能性需求，比如许多 AOP 相关的文章和教程中经常用作“hello world”例子的跟踪、登录或监控。

另一个传统的 AOP 应用程序是缓存。例如，一个基于 CMP 实体 bean 的 TradeDao 组件将从 WebLogic Server 提供的缓存功能中受益。然而对于 YahooFeed 组件来说却并非如此，因为它必须通过 Internet 连接到雅虎门户。这明显是一个应该应用缓存的位置，而且它还允许减少外部连接的次数，并最终降低整个系统的负载。注意，基于截至时间的缓存也会在刷新信息时带来一些延迟，但是在很多情况下，它仍然是可以接受的。要应用缓存功能，可以定义一个 yahooFeedCachingAdvisor，它将把 CachingAdvice 附加到 yahooFeed bean 上的 getPrice()方法。在“下载”部分中，您可以找到一个 CachingAdvice 实现的例子。

```
<bean id="getPriceAdvisor" abstract="true"
        class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
    <property name="mappedName" value="getPrice"/>
  </bean>

  <bean id="yahooFeedCachingAdvisor" parent="getPriceAdvisor">
    <property name="advice">
      <bean class="org.javatx.spring.aop.CachingAdvice">
        <constructor-arg index="0" ref="cache"/>
      </bean>
    </property>
```

```
  </bean>
```

因为 getPrice()方法已经成为几种通知的公共联结点，所以声明一个抽象的 getPriceAdvisor bean，然后在 yahooFeedCachingAdvisor 中对其进行扩展，指定具体的通知 CachingAdvice。注意，也可以修改前面的 foreignTradeAdvisor，使其使用同一个 getPriceAdvisor 父 bean。

现在可以更新 yahooFeed bean 的定义，并将它包装在一个 ProxyFactoryBean 中，然后使用 yahooFeedCachingAdvisor 通知它。例如：

```
<bean id="yahooFeed" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces" value="org.javatx.spring.aop.TradeManager"/>
    <property name="target">
      <bean class="org.javatx.spring.aop.YahooFeed">
    </property>
    <property name="interceptorNames">
      <list>
        <value>yahooFeedCachingAdvisor</value>
      </list>
    </property>
  </bean>
```

当请求命中已经保存在缓存中的数据时，上面的修改将极大地提高性能，但是如果传入多个针对同一个符号的请求，而该符号尚未进入缓存或者已经到期，我们将看到多个并发的请求到达服务提供者，请求同一个符号。对此，存在一种显而易见的优化，就是中断对同一个符号的所有请求，直到第一个请求完成为止，然后使用第一个请求获得的结果。EJB 规范（参见“Programming Restrictions”，2.1 版本的 25.1.2 部分）一般不推荐使用这种方法，因为它对运行在多个 JVM 上的集群环境不奏效。然而，至少在单个的节点中这种优化可以改进性能。图 2 所示的图表对比说明了优化之前和优化之后的情况：

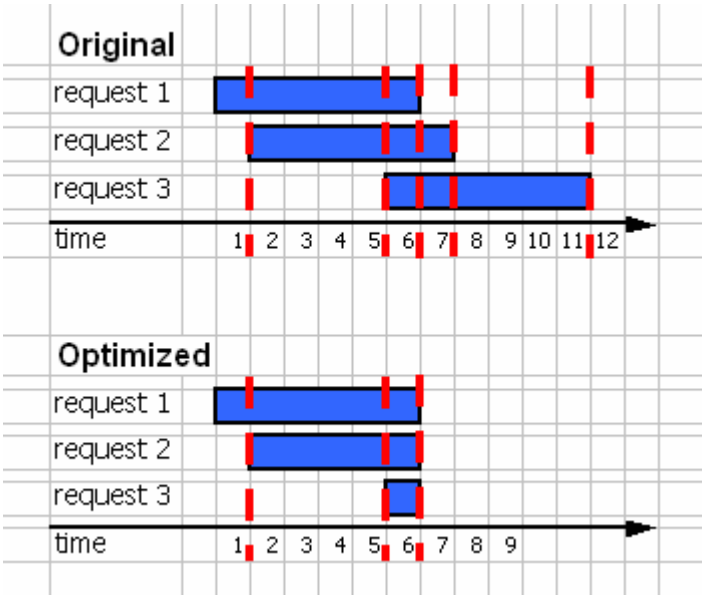

图 2. 优化之前和优化之后

该优化也可以实现为通知，并添加在 yahooFeed bean 中的拦截器链的末端：

```
...
    <property name="interceptorNames">
      <list>
        <idref local="yahooFeedCachingAdvisor"/>
        <idref local="syncPointAdvisor"/>
      </list>
    </property>
```

实际的拦截器实现应该像下面这样：

```
public class SyncPointAdvice implements MethodInterceptor {
  private long DEFAULT_TIMEOUT = 10000L;

  private Map requests = Collections.synchronizedMap(new HashMap());

  public Object invoke(MethodInvocation invocation) throws Throwable {
    String symbol = (String) invocation.getArguments()[0];
    Object[] lock = (Object[]) requests.get(symbol);
    if(lock==null) {
      lock = new Object[1];
      requests.put(symbol, lock);
      try {
        lock[0] = invocation.proceed();
        return lock[0];
```

```
        } finally {
            requests.remove(symbol);
            synchronized(lock) {
                lock.notifyAll();
            }
        }
    }

    synchronized(lock) {
        lock.wait(DEFAULT_TIMEOUT);
    }
    return lock[0];
  }

}
```

可以看出，通知代码相当简单，而且不依赖于其他的组件，这使得 JUnit 测试变得十分简单。在“参考资料”部分，您可以找到 SyncPointAdvice 的 JUnit 测试的完整源代码。对于复杂的并发场景来说，使用 Java 5 中 java.util.concurrent 包的同步机制或者针对老的 JVM 使用其 backport 是一种不错的做法。

下载

sources.zip 包含了本文中使用的所有源代码。如果您希望构建代码，可以遵照 README.txt 中的指导。

结束语

本文介绍了一种把 J2EE 应用程序中的 EJB 转换为 Spring 托管组件的方法，以及转换之后可以采用的强大技术。它还给出了几个实际的例子，说明如何借助于 Spring 的 AOP 框架、应用面向方面的方法来扩展 J2EE 应用程序，并在不修改现有代码的情况下实现新的业务需求。

在 EJB 中使用 Spring Framework 将减少代码间的耦合，并使许多强大的功能即时生效，从而提高可扩展性和灵活性。这还使得应用程序的单个组件变得更加易于测试，包括新引入的 AOP 通知和拦截器，它们用于实现业务功能或者处理非功能性的需求，比如跟踪、缓存、安全性和事务。

参考资料

- Spring Framework 项目页面
- Inversion of Control Containers and the Dependency Injection pattern，作者 Martin Fowler
- Better J2EEing with Spring，作者 Peter Braswell (dev2dev，2005 年 7 月)

- Spring Integration with WebLogic Server，作者 Andy Piper、Rod Johnson、Chris Wall 和 Nick Tran (dev2dev，2005 年 9 月)
- Peak Performance Tuning of CMP 2.0 Entity Beans in WebLogic Server 8.1 and 9.0，作者 Dmitri Maximovich (dev2dev，2005 年 11 月)
- Spring-based AOP Caching，作者 Pieter Coucke
- Acegi security 项目——为 The Spring Framework 提供全面的安全性服务
- Wire Hibernate Transactions in Spring，作者 Binildas Christudas (ONJava，2005 年 7 月)
- Implementing Transaction Suspension in Spring，作者 Juergen Hoeller (dev2dev，2005 年 11 月)
- XDoclet
- Unit testing with mock objects，作者 Alexander Chaffee 和 William Pietri (developerWorks，2005 年 11 月)
- Concurrent Programming with J2SE 5.0，作者 Qusay Mahmoud (Sun Developer Network，2005 年 3 月)
- Getting to Know Synchronizers——简要介绍了 java.util.concurrent 包所支持的同步机制
- Java 1.4 backport of JSR 166 (java.util.concurrent)
- dev2dev EJB 技术中心

Spring Framework 中的面向方面编程（AOP），第一部分

作为这个介绍 Spring 框架中的面向方面编程(Aspect-Oriented Programming，AOP)的系列的第一部分，本文介绍了使您可以使用 Spring 中的面向方面特性进行快速开发的基础知识。使用跟踪和记录方面(面向方面领域的 HelloWorld) 作为例子，本文展示了如何使用 Spring 框架所独有的特性来声明切入点和通知以便应用方面。本系列的第二部分将更深入地介绍如何运用 Spring 中的所有通知类型和切入点来实现更实用的方面和面向方面设计模式。对于 AOP 的更一般性的介绍，请查看 ONJava 站点上 Graham O'Regan 的文章，“Introduction to Aspect-Oriented Programming”。

本文的目的不是要介绍构成模块化 J2EE 系统——即 Spring 框架——的所有重要元素，我们将只把注意力放在 Spring 所提供的 AOP 功能上。由于 Spring 的模块化设计方法，我们可以只使用该框架的 AOP 元素，而无需对构成 Spring 框架的其他模块做太多考虑。

在 AOP 方面，Spring 提供了什么？

“它的目标不是提供最完善的 AOP 实现（虽然 Spring AOP 非常强大）；而是要提供 AOP 实现与 Spring IoC 的紧密集成，以便帮助解决企业应用中的常见问题。”

Spring Framework 参考文档

为了实现这个目标，Spring 框架目前支持一组 AOP 概念，从切入点到通知。本文将展示如何使用 Spring 框架中所实现的如下 AOP 概念：

- **通知（Advice）：**如何将 before 通知、afterReturning 通知和 afterThrowing 通知声明为 bean。

- **切入点（Pointcut）：**如何声明静态切入点逻辑以将 XML Spring Bean Configuration 文件中的所有内容联系在一起。
- **Advisor：**关联切入点定义与通知 bean 的方式。

设置场景：一个简单的例子应用程序

“一般而言，Spring 并不是预描述的。虽然使用好的实践非常容易，但是它避免强制推行一种特定的方法。” Spring Framework 参考文档

要试用 Spring 框架的 AOP 功能，首先我们要创建一个简单的 Java 应用程序。IbusinessLogic 接口和 BusinessLogic 类为 Spring 框架中的 bean 提供了简易构件块。虽然该接口对于我们的简单应用程序逻辑来说不是必需的，但是它是 Spring 框架所推荐的良好实践。

```
public interface IBusinessLogic
{
    public void foo();
}

public class BusinessLogic
    implements IBusinessLogic
{
    public void foo()
    {
        System.out.println(
        "Inside BusinessLogic.foo()");
    }
}
```

可以编写 MainApplication 类，借此练习 BusinessLogic bean 的公有方法。

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class MainApplication
{
    public static void main(String [] args)
    {
        // Read the configuration file
        ApplicationContext ctx =
          new FileSystemXmlApplicationContext(
            "springconfig.xml");
```

```
        //Instantiate an object
        IBusinessLogic testObject =
          (IBusinessLogic) ctx.getBean("businesslogicbean");

        // Execute the public
        // method of the bean
        testObject.foo();
    }
}
```

在 BusinessLogic 类及其关联接口中没有什么需要注意的。但是，MainApplication 类初始化 BusinessLogic 对象的方式很有意思。通过使用ctx.getBean("businesslogicbean")调用，MainApplication将加载和管理BusinessLogic类的 bean 实例的任务转交给了 Spring 框架。

允许 Spring 控制 BusinessLogic bean 的初始化，这使得 Spring 运行时有机会在 bean 被返回给应用程序之前执行 J2EE 系统所需的所有与 bean 相关的管理任务。然后 Spring 运行时配置可以决定对 bean 应用哪些任务和模块。该配置信息由一个 XML 文件提供，类似于下面所示的：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC
    "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

   <!-- Bean configuration -->
   <bean id="businesslogicbean"
   class="org.springframework.aop.framework.ProxyFactoryBean">
      <property name="proxyInterfaces">
         <value>IBusinessLogic</value>
      </property>
      <property name="target">
         <ref local="beanTarget"/>
      </property>
   </bean>
   <!-- Bean Classes -->
   <bean id="beanTarget"
```

```
    class="BusinessLogic"/>

</beans>
```

该配置文件，即 springconfig.xml，指定要加载一个接口与 IbusinessLogic 相匹配的 bean。该 bean 随后被关联到 BusinessLogic 实现类。看起来好像是费了很大力气只为了加载一个简单的 bean 并调用一个方法，但是您要知道，这个配置文件只是使 Spring 框架可以透明地对应用程序应用其组件的众多特性的一个体现。

图 1 显示了基本的顺序图：MainApplication 原样执行，没有应用方面。

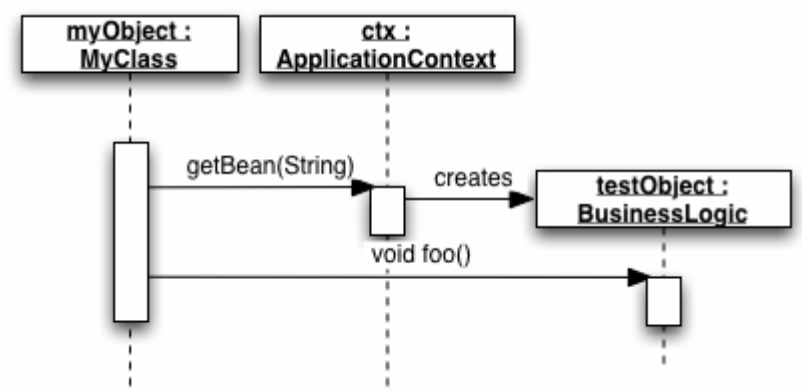

图 1.没有对 BusinessLogic bean 应用方面时的顺序图

请查看本文末尾处的参考资料，获取这个简单 Spring 应用程序的源代码。

应用方法跟踪（Method Tracing）方面

可能最基本的方面就是方法跟踪方面了。这可能是您找得到的最简单的方面了，因此它是研究新的 AOP 实现的一个很好的起点。

方法跟踪方面在一个目标应用程序内捕获对所跟踪的方法的调用以及方法的返回值，并以某种方式显示这种信息。在 AOP 中，通知的 before 和 after 类型用于捕获这些类型的联结点，因为这两种通知可以在方法调用联结点之前或之后触发。使用 Spring 框架，方法跟踪方面的 before 通知是在 TracingBeforeAdvice 类中声明的。

```
import java.lang.reflect.Method;
import org.springframework.aop. MethodBeforeAdvice;

public class TracingBeforeAdvice
    implements MethodBeforeAdvice
{
    public void before(Method m,
                       Object[] args,
                       Object target)
                       throws Throwable
    {
```

```
            System.out.println(
                "Hello world! (by " +
                this.getClass().getName() +
                ")");
        }
}
```

类似地，after 通知可以在 TracingAfterAdvice 类中声明。

```
import java.lang.reflect.Method;
import org.springframework.aop.AfterReturningAdvice;

public class TracingAfterAdvice
    implements AfterReturningAdvice
{
    public void afterReturning(Object object,
                               Method m,
                               Object[] args,
                               Object target)
                               throws Throwable
    {
        System.out.println(
            "Hello world! (by " +
            this.getClass().getName() +
            ")");
    }
}
```

这两个类都通过实现 Spring 框架的适当通知接口而表示了特定的通知。每种类型的通知都指定实现 before(..)或 afterReturning(..)方法，以便使 Spring 运行时可以告诉通知适当的联结点会在何时出现。值得注意的是，TracingAfterAdvice 实际上是从 AfterReturningAdvice 扩展而来的，表示只有在联结点在无异常的情况下获得返回值时才运行通知。

为了将通知与应用程序中的适当联结点关联起来，必须对 springconfig.xml 进行一些修改。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC
    "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
```

```
<beans>

    <!-- Bean configuration -->
    <bean id="businesslogicbean"
    class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>IBusinessLogic</value>
        </property>
        <property name="target">
            <ref local="beanTarget"/>
        </property>
        <property name="interceptorNames">
            <list>
                <value>theTracingBeforeAdvisor</value>
                <value>theTracingAfterAdvisor</value>
            </list>
            </property>
    </bean>
    <!-- Bean Classes -->
    <bean id="beanTarget"
    class="BusinessLogic"/>

    <!-- Advisor pointcut definition for before advice -->
    <bean id="theTracingBeforeAdvisor"
        class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice">
            <ref local="theTracingBeforeAdvice"/>
        </property>
        <property name="pattern">
            <value>.*</value>
        </property>
    </bean>

    <!-- Advisor pointcut definition for after advice -->
    <bean id="theTracingAfterAdvisor"
        class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice">
```

```
            <ref local="theTracingAfterAdvice"/>
        </property>
        <property name="pattern">
            <value>.*</value>
        </property>
    </bean<

    <!-- Advice classes -->
    <bean id="theTracingBeforeAdvice"
        class="TracingBeforeAdvice"/>
    <bean id="theTracingAfterAdvice"
        class="TracingAfterAdvice"/>

</beans>
```

theTracingBeforeAdvisor 和 theTracingAfterAdvisor advisor 被添加到前面所声明的 businesslogicbean。每个 advisor 都可能截获所有 bean 所关联到的联结点。Advisor 本身就是 bean，而它唯一的作用就是将切入点定义与通知 bean 关联起来。本例中的切入点定义是在静态对象层次结构中指定相关联结点的正则表达式。

因为本例中使用了 org.springframework.aop.support.RegexpMethodPointcutAdvisor 切入点 advisor，切入点逻辑是使用正则表达式指定的。正则表达式用于识别公有接口对 IbusinessLogici 接口的联结点。下面是一些可以用来指定 IBusinessLogic 接口上的不同联结点集合的正则表达式例子：

- **<value>.*</value>**：该表达式选择 advisor 所关联到的一个或多个 bean 上的所有联结点。
- **<value>./IBusinessLogic/.foo</value>**：该表达式只选择 IbusinessLogic 接口上的 foo()方法的联结点。如果是 advisor 所关联到的 bean，则该表达式只选择 IBusinessLogic 接口上的联结点。

springconfig.xml 文件中最后的 bean 声明指定实现通知 bean 的类。

既然已经指定了跟踪方面的正确配置，那么下一次执行 MainApplication 时，这些方面就会在初始化过程中被编织进去，而 BusinessLogic bean 中的所有方法都将被跟踪，如图 2 所示。

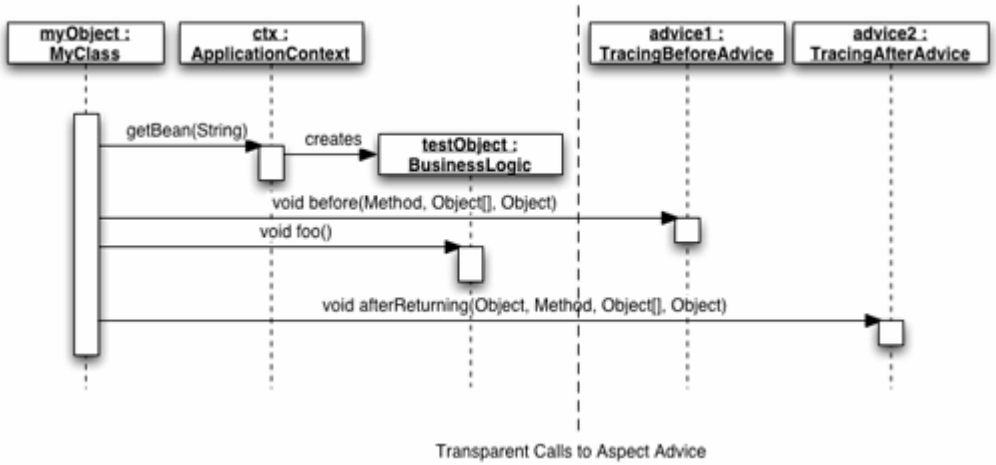

图 2. 方法跟踪方面应用到 BusinessLogic bean 之后的顺序图（单击图像查看大图）

方法跟踪方面和例子应用程序的源代码可在本文末尾的参考资料小节进行下载。

方面的重用

可以对方法跟踪方面进行扩展，提供一个稍微复杂的记录（Logging）方面。记录方面提供了一个很不错的重用例子，因为记录方面所需的许多特性都已经包含在方法跟踪方面中了。

在本例中，记录方面扩展了方法跟踪方面，以便显示附加的与（在应用程序的执行过程中）所引发的异常有关的信息。

要完全使用记录方面，需要对应用程序做一些更改。BusinessLogicException 异常类提供了一个可以由 IBusinessLogicInterface 接口和 BusinessLogic 实现类新增的 void bar()方法引发的异常。

```
public class BusinessLogicException
    extends Exception
{

}

public interface IBusinessLogic
{
    public void foo();

    public void bar()
        throws BusinessLogicException;
}

public class BusinessLogic
    implements IBusinessLogic
{
```

```
    public void foo()
    {
        System.out.println(
            "Inside BusinessLogic.foo()");
    }

    public void bar()
        throws BusinessLogicException
    {
        System.out.println(
        "Inside BusinessLogic.bar()");
            throw new BusinessLogicException();
    }
}
```

MainApplication 类现在将对 void bar()方法进行一次额外的调用，并处理选中的、可能由该方法引发的异常。

```
import org.springframeworkcontext.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class MainApplication
{
    public static void main(String [] args)
    {
        // Read the configuration file
        ApplicationContext ctx =
            new FileSystemXmlApplicationContext(
                "springconfig.xml");

        //Instantiate an object
        IBusinessLogic testObject =
            (IBusinessLogic) ctx.getBean(
                "businesslogicbean");

        //Execute the public methods of the bean
        testObject.foo();

        try
```

```
        {
            testObject.bar();
        }
        catch(BusinessLogicException ble)
        {
            System.out.println(
              "Caught BusinessLogicException");
        }
    }
}
```

来自方法跟踪方面的 TracingBeforeAdvice 和 TracingAfterAdvice 通知可以整体重用。LoggingThrowsAdvice 类为新的异常记录提供了通知。

```
import org.springframework.aop.ThrowsAdvice;
import java.lang.reflect.Method;

public class LoggingThrowsAdvice
    implements ThrowsAdvice
{
    public void afterThrowing(Method method,
                              Object[] args,
                              Object target,
                              Throwable subclass)
    {
        System.out.println(
          "Logging that a " +
          subclass +
          "Exception was thrown.");
    }
}
```

应用记录方面的最后一步是修改 springconfig.xml 配置文件，使其包含新添加的 LoggingThrowsAdvice 通知。

图 3 显示了运行 MainApplication 并使用 Spring 框架应用了记录方面的 UML 顺序图。

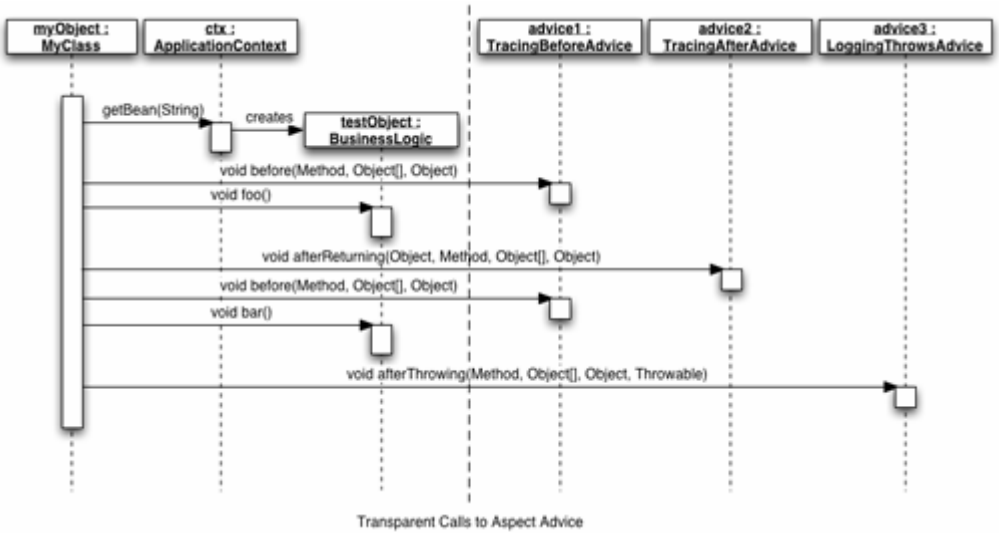

图 3. 记录方面应用到 BusinessLogic bean 之后的顺序图（单击图像查看大图）

此处的记录方面清楚地说明了如何重用现有方面以及如何在 Spring 框架中使用通知的 throws 形式。通过为 before 和 after 通知声明新的通知来重写现有的方法跟踪方面实现，可以实现更复杂的记录方面，记录到更复杂的记录框架，比如 LOG4J。关于记录方面和例子应用程序的源代码，请参见本文末尾的参考资料小节。

结束语

本文展示了使用 Spring 框架中的基本 AOP 结构所应用的一些简单方面。在本系列的下一篇文章中，我们将介绍一些更实用的方面，探讨方面的生命周期，使用 Spring 框架的 around 通知，并使用 Spring 来应用 AOP 模式。

参考资料

- 本文的源代码
- Spring Java/J2EE Framework
- Spring AOP Framework
- "Introduction to Aspect-Oriented Programming"

Spring Framework 中的面向方面编程（AOP），第二部分

在本系列的第一部分，我介绍了如何实现面向方面领域的“HelloWorld”：跟踪和记录方面。利用 Spring 框架所提供的面向方面编程(Aspect-Oriented Programming，AOP)功能，您看到了如何使用 before-、after-和基于异常的通知，以及如何使用基于正则表达式的简单切入点。跟踪和记录方面提供了非常不错的上手例子，而本文将进一步介绍一种新的通知形式：around 通知。

比起第一部分中介绍的那些通知类型，around 形式的通知是一种更具侵入性也更强大的面向对象概念。本文将描述 around 通知的每个特性，以便您可以在自己的 Spring AOP 应用程序中正确地使用它。在本文最后，我将向您展示如何使用 around 通知来截获和改变应用程序中各个特性相互作用的方式，以便实现 Cuckoo's Egg（杜鹃的蛋）面向方面设计模式。

概述 Spring AOP、IoC 和代理

在第一部分，我们快速浏览了 Spring 的一些 AOP 特性，而没有阐明 Spring 如何实现 AOP 的细节。要理解 Spring 框架如何运转，尤其是它如何实现其 AOP 功能，首先您要明白，Spring 是一个依赖于控制反转(Inversion of Control，IoC)设计模式的轻量级框架。

注意：本文的目的不是要深入介绍 IoC 模式，介绍 IoC 只是为了使您明白该设计模式是如何影响 Spring AOP 实现的。有关 IoC 模式的更详细的介绍请参见本文末尾的参考资料。

IoC 设计模式的出现已经有一段时间了。一个最明显的例子就是 J2EE 架构本身。随着企业开发尤其是 J2EE 平台的出现，应用程序开始依赖于由外部容器所提供的一些特性，比如 bean 创建、持久性、消息传递、会话以及事务管理。

IoC 引入了一个新概念：由组件构成的框架，它与 J2EE 容器有许多类似之处。IoC 框架分离了组件所依赖的功能，并且，根据 Sam Newman 文章中的说法，提供了“连接组件的‘胶水’”。

对组件所依赖特性的控制 被反转 了，这样外部框架就可以尽可能透明地提供这些特性了。IoC 模式真正意识到了从传统的由依赖于功能的组件来负责这些功能，到由独立的框架来配置和提供这些功能的方式转变。

图 1 显示了一些构成 IoC 模式的不同组件角色的例子。

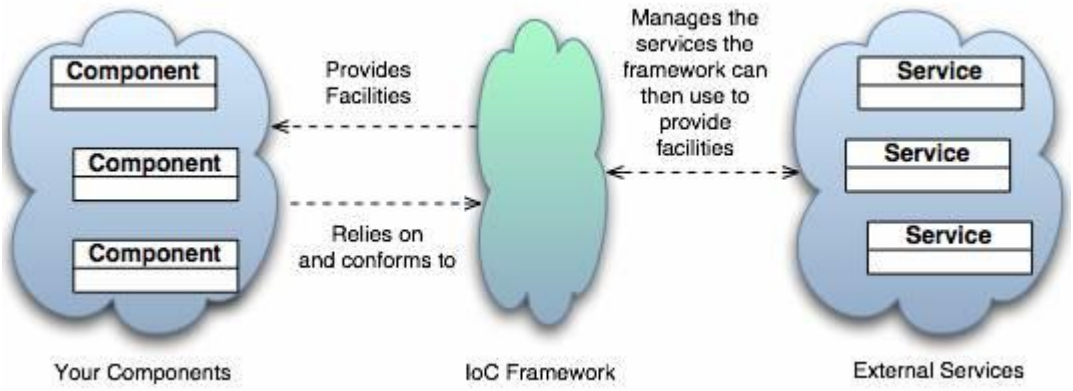

图 1. 没有对 BusinessLogic bean 应用方面时的顺序图.

图字：

Component：组件

Provides Facilities：提供功能

Relies on and conforms to：依赖于并服从

Manages the services the framework can then use to provide facilities：管理框架随后可以用来提供功能的服务

Service：服务

Your Component：您的组件

IoC Framework：IoC 框架

External services：外部服务

IoC 模式使用 3 种不同的方法来解除组件与服务控制的耦合：类型 1、类型 2 和类型 3。

- 类型 1：接口注入

  这是大部分 J2EE 实现所使用的方法。组件显式地服从于一组接口，带有关联的配置元数据，以便允许框架对它们进行正确的管理。

- 类型 2：Setter 注入

  外部元数据被用来配置组件相互作用的方式。在第一部分中，我们就是使用这种 IoC 方法利用 springconfig.xml 文件来配置 Spring 组件的。

- 类型 3：构造函数注入

  组件（包括构造组件时要用的参数）注册到框架，而框架提供组件的实例以及所有要应用的指定功能。

IoC 在组件开发和企业开发中越来越受欢迎。IoC 的实际例子包括传统的 J2EE 解决方案，比如：JBoss、Apache 基金会的 Avalon 项目以及本文的 Spring 框架。实际上，Spring 框架构建于 IoC 模式的基础上是为了帮助将它的轻量级功能注入到它的相关应用程序的组件中。

那么 IoC 对于 Spring AOP 有何意义呢？Spring 的 IoC 特性是使用 IoC springconfig.xml 配置文件对应用程序应用方面的推动因素之一。springconfig.xml 配置文件通知 Spring 框架运行时有关应用程序的组件要被注入的功能类型的信息，所以自然轻量级的 AOP 功能就以同样的方式应用了。然后 Spring 使用代理模式围绕现有的类和 bean 实现指定的 AOP 功能。

图 2 显示了 Spring 及其 IoC 框架如何使用代理对象提供 AOP 功能(根据 springconfig.xml 文件中的 IoC 配置。)

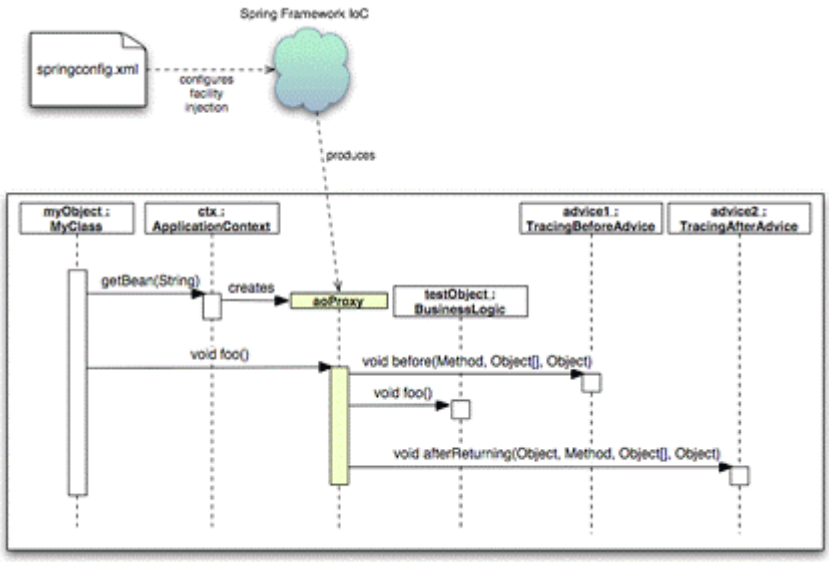

图 2. springconfig.xml 配置文件改变了 Spring 框架 IoC，以便随后向第一部分中的一个顺序图提供 AOP 代理（单击图像查看大图）

在本系列下面的部分，您将不断看到现在包含在顺序图中的代理对象。这只是为了说明对于 Spring AOP 来说没有“魔法”，实际上只有一个面向对象设计模式的良好例子。

回到 AOP：使用 around 通知的积极方面

在第一部分，您看到了如何使用 Spring AOP 来实现跟踪和记录方面。跟踪和记录都是“消极”方面，因为它们的出现并不会对应用程序的其他行为产生影响。它们都使用了消极的 before 和 after 形式的通知。

但是如果您希望改变应用程序的常规行为呢？例如说，您希望重写一个方法？这样的话，您就需要使用更积极的 around 形式的通知。

第一部分的简单例子应用程序包括 IbusinessLogic 接口、BusinessLogic 类和 MainApplication 类，如下所示：

```
public interface IBusinessLogic
{
   public void foo();
}



public class BusinessLogic
   implements IBusinessLogic
{
   public void foo()
   {
       System.out.println(
          "Inside BusinessLogic.foo()");
   }
}

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class MainApplication
{
   public static void main(String [] args)
   {
      // Read the configuration file
      ApplicationContext ctx =
         new FileSystemXmlApplicationContext(
            "springconfig.xml");

      //Instantiate an object
```

```
    IBusinessLogic testObject =
        (IBusinessLogic) ctx.getBean(
            "businesslogicbean");

    // Execute the public
    // method of the bean
    testObject.foo();
  }
}
```

要对一个 BusinessLogic 类的实例彻底重写对 foo()方法的调用 ,需要创建 around 通知 ,如下面的 AroundAdvice 类所示：

```
import org.aopalliance.intercept.MethodInvocation;
import org.aopalliance.intercept.MethodInterceptor;

public class AroundAdvice
    implements MethodInterceptor
{
    public Object invoke(
        MethodInvocation invocation)
        throws Throwable
    {
        System.out.println(
            "Hello world! (by " +
            this.getClass().getName() +
            ")");

        return null;
    }
}
```

要在 Spring 中用作 around 通知，AroundAdvice 类必须实现 MethodInterceptor 接口和它的 invoke(..)方法。每当截获到方法的重写，invoke(..)方法就会被调用。最后一步是改变包含在应用程序的 springconfig.xml 文件中的 Spring 运行时配置，以便可以对应用程序应用 AroundAdvice。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC   "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
```

```
<beans>

    <!-- Bean configuration -->
    <bean id="businesslogicbean"
    class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>IBusinessLogic</value>
        </property>
        <property name="target">
            <ref local="beanTarget"/>
        </property>
        <property name="interceptorNames">
            <list>
                <value>theAroundAdvisor</value>
            </list>
            </property>
    </bean>
    <!-- Bean Classes -->
    <bean id="beanTarget"
    class="BusinessLogic"/>

    <!-- Advisor pointcut definition for around advice -->
    <bean id="theAroundAdvisor"
        class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice">
            <ref local="theAroundAdvice"/>
        </property>
        <property name="pattern">
            <value>.*</value>
        </property>
    </bean>

    <!-- Advice classes -->
    <bean id="theAroundAdvice"
        class="AroundAdvice"/>
```

```
</beans>
```

根据该 springconfig.xml 配置文件，theAroundAdvisor 截获所有对 BusinessLogic 类的方法的调用。接下来，theAroundAdvisor 被关联到 theAroundAdvice，表明当截获一个方法时，就应该使用在 AroundAdvice 类中指定的通知。既然已经指定了 around 通知的正确配置，下一次执行 MainApplication 类时，BusinessLogic bean 的 foo()方法就会被截获并重写，如图 3 所示：

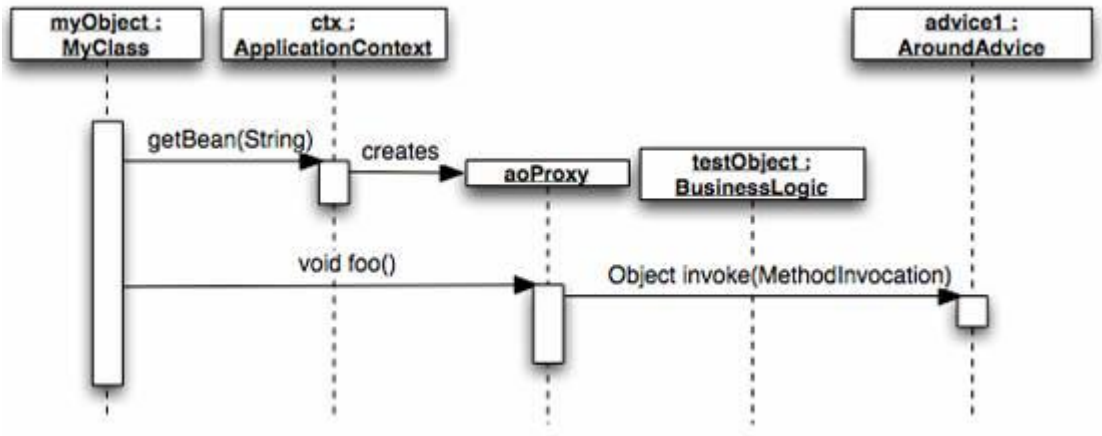

图 3. 使用 around 通知重写对 BusinessLogic 类中的 foo()方法的调用

前面的例子显示，BusinessLogic 类中的 foo()方法可以通过 AroundAdvice 类中的 invoke(..)方法彻底重写。原来的 foo()方法完全不能被 invoke(..)方法调用。如果希望从 around 通知内调用 foo()方法，可以使用 proceed()方法，可从 invoke(..)方法的 MethodInvocation 参数中得到它。

```
public class AroundAdvice
    implements MethodInterceptor
{
    public Object invoke(
        MethodInvocation invocation)
        throws Throwable
    {
        System.out.println(
            "Hello world! (by " +
            this.getClass().getName() +
            ")");

        invocation.proceed();

        System.out.println("Goodbye! (by " +
            this.getClass().getName() +
            ")");
```

```
        return null;
    }
}
```

图 4 显示了对 proceed()的调用如何影响操作的顺序（与图 3 所示的初始 around 通知执行相比较）。

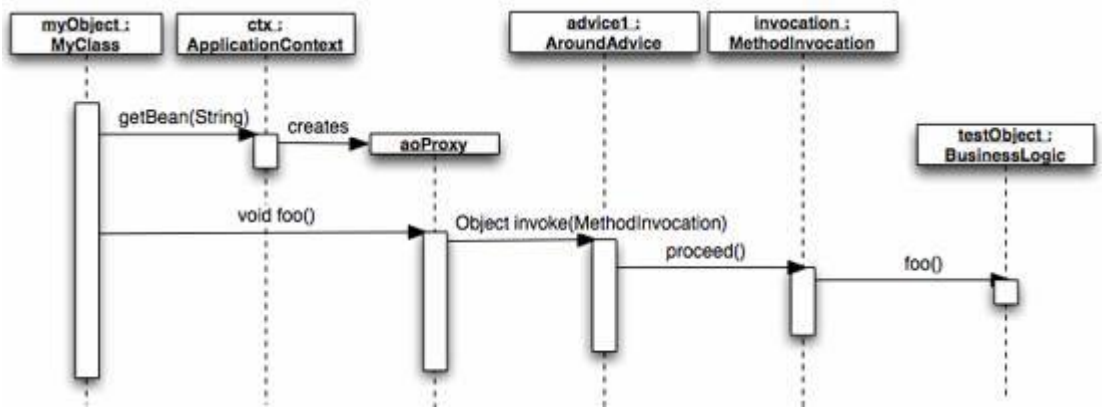

图 4. 从 around 通知内使用 proceed()调用原来的方法

当调用 proceed()时，实际是在指示被截获的方法（在本例中是 foo()方法）利用包含在 MethodInvocation 对象中的信息运行。您可以通过调用 MethodInvocation 类中的其他方法来改变该信息。

您可能希望更改包含在 MethodInvocation 类中的信息，以便在使用 proceed()调用被截获的方法之前对被截获方法的参数设置新值。

通过对 MethodInvocation 对象调用 getArguments()方法，然后在返回的数组中设置其中的一个参数对象，最初传递给被截获的方法的参数可以被更改。

如果 IbusinessClass 和 BusinessLogic 类的 foo()方法被更改为使用整型参数，那么就可以将传递给被截获的调用的值由在 AroundAdvice 的 notify(..)方法中传递改为在 foo(int)中传递。

```
public class AroundAdvice
    implements MethodInterceptor
{
    public Object invoke(
        MethodInvocation invocation)
        throws Throwable
    {
        System.out.println(
            "Hello world! (by " +
            this.getClass().getName() +
            ")");

        invocation.getArguments()[0] = new Integer(20);
```

```
        invocation.proceed();

        System.out.println(
            "Goodbye! (by " +
            this.getClass().getName() +
            ")");

        return null;
    }
}
```

在本例中，被截获的方法的第一个形参被假设为 int。实参本身是作为对象传递的，所以通过将其包装在 Integer 类实例中的方法，基本的 int 类型的形参被改为对应数组中的新值。如果您将该参数设置为一个非 Integer 对象的值，那么在运行时就会抛出 IllegalArgumentException 异常。

您还将注意到，invoke(..)方法必须包含一个 return 语句，因为该方法需要返回值。但是，被重写的 foo()方法并不返回对象，所以 invoke(..)方法可以以返回 null 结束。如果在 foo()方法不需要的情况下，您仍然返回了一个对象，那么该对象将被忽略。

如果 foo()方法确实需要返回值，那么需要返回一个与 foo()方法的初始返回类型在同一个类或其子类中的对象。如果 foo()方法返回一个简单类型，例如，一个 integer，那么您需要返回一个 Integer 类的对象，当方法被重写时，该对象会自动由 AOP 代理拆箱，如图 5 所示：

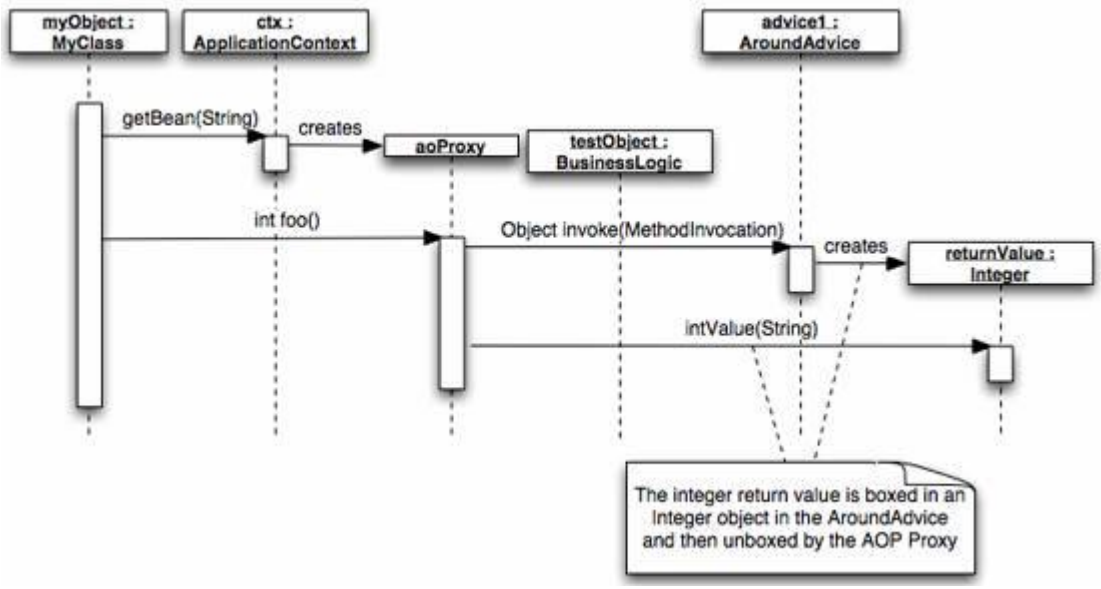

图 5. around 通知的装箱和自动拆箱

图字：

Object invoke：对象调用

The integer return value is boxed in a Integer object in the AroundAdvice and then unboxed by the AOP Proxy：整型返回值被装箱在 AroundAdvic 通知的一个 Integer 对象中，然后由 AOP 代理拆箱。

面向方面编程还是一个比较新的领域，尤其是与衍生出它的面向对象编程相比。设计模式通常被认为是常见问题的通用解决方案，因为面向方面发展的时间还不长，所以已发现的面向方面设计模式比较少。

此处要介绍的是一种正在浮现的模式，即 Cuckoo's Egg 设计模式。该模式还有其他的叫法，它在面向对象领域的对等体包括模仿对象（Mock Object）和模仿测试（Mock Testing），甚至代理模式也与它有一些类似之处。

Cuckoo's Egg 面向方面设计模式可以被定义为应用程序上下文中功能部件的透明和模块化的置换。就像杜鹃偷偷地把自己的蛋放在另一种鸟的巢中一样，Cuckoo's Egg 设计模式用一个替代功能部件实现置换现有的功能部件,而使造成的干扰尽可能少。

这种置换的实现方式可以是静态的、动态的、部分的、完全的，针对一个对象的多个部分，或针对多个组件。使用面向方面的方法可以透明地实现功能部件的置换，而无需对应用程序的其余部分进行更改。要置换应用程序中现有功能部件的替代功能部件就是“杜鹃的蛋”。图 6 显示了 Cuckoo's Egg 设计模式中的主要组成元素。

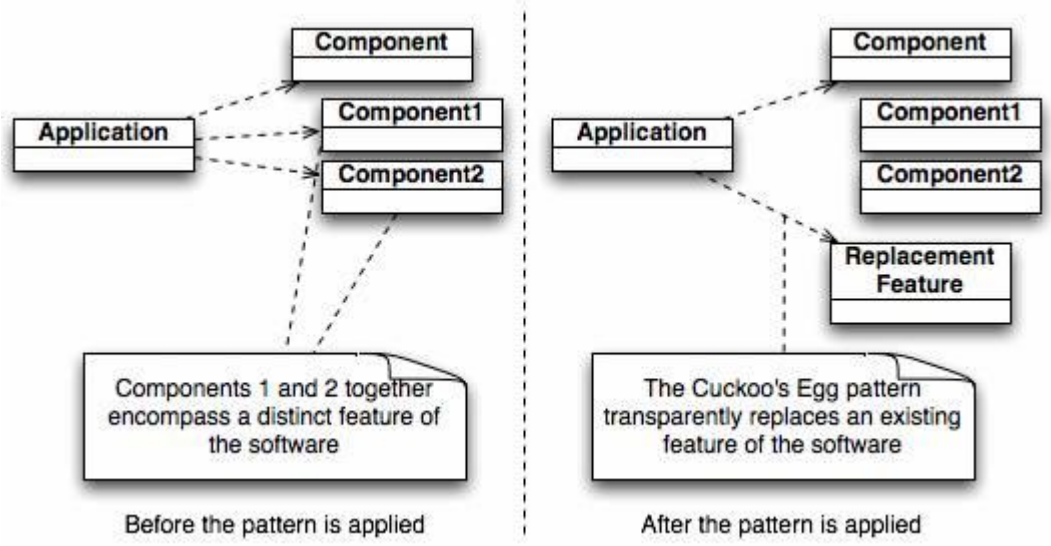

图 6. Cuckoo's Egg 设计模式中的主要组成元素

图字：

Application：应用程序

Component：组件

Replacement Feature：替代功能部件

Component 1 and 2 together encompass a distinct feature of the software：组件 1 和 2 共同包含了软件的一个独立的功能部件

The Cuckoo's Egg pattern transparently replaces an existing feature of the software：Cuckoo's Egg 模式透明地置换了软件现有的功能部件

Before the pattern is applied：应用该模式前

After the pattern is applied：应用该模式后

Cuckoo's Egg 设计模式依赖于 around 通知的概念。您需要借助于积极的和侵入性的 around 通知来截获并有效置换应用程序中现有的功能部件。

有关 Cuckoo's Egg 设计模式的更多信息，以及 AspectJ 中的一个可选实现，请参见《AspectJ Cookbook》（O'Reilly，2004 年 12 月出版）。

要使用 Spring AOP 实现 Cuckoo's Egg 设计模式，需要声明一个 around 通知来截获所有对要置换的功能部件的调用。与 hot-swappable target sources（Spring AOP 的一个功能部件，将在本系列的另一篇文章中介绍）不同，around 通知的显式使用使得 Cuckoo's Egg 实现可以有效地跨越对象边界（因此也可以跨越 bean 边界）进行整个功能部件的置换，如图 7 所示。

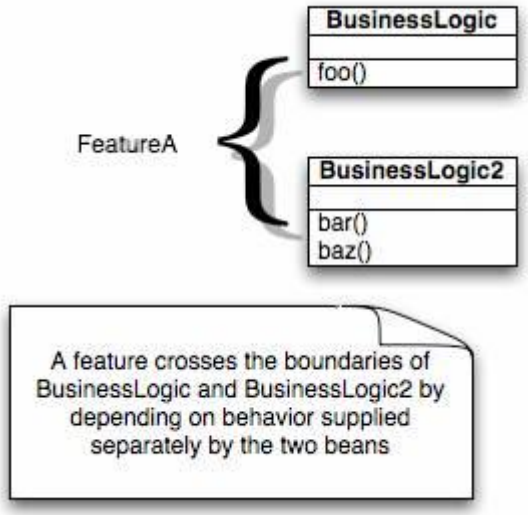

图 7. 一个跨越 bean 边界的组件

图字：

A feature crosses the boundaries of BusinessLogic and BusinessLogic2 by depending on behavior supplied separately by the two beans：一个功能部件通过依赖于由 BusinessLogic 和 BusinessLogic2 各自提供的行为而跨越了这两个 bean 的边界

下面的代码显示了一个具有两个 bean 的简单应用程序，其中有一个功能部件跨越了该应用程序的多个方面。要置换的功能部件可以被视为包含 IBusinessLogic bean 中的 foo()方法和 IBusinessLogic2 bean 中的 bar()方法。IBusinessLogic2 bean 中的 baz()方法不是 该功能部件的一部分，所以不进行置换。

```
public interface IBusinessLogic
{
    public void foo();
}
```

```
public interface IBusinessLogic2
{
    public void bar();

    public void baz();
}
```

该例子的完整源代码可在本文末尾的参考资料小节中下载。

此处，ReplacementFeature 类扮演了“杜鹃的蛋”的角色，它提供了将被透明地引入应用程序的替代实现。ReplacementFeature 类实现了所有在该类引入时要被置换的方法。

```
public class ReplacementFeature
{
    public void foo()
    {
        System.out.println(
            "Inside ReplacementFeature.foo()");
    }

    public void bar()
    {
        System.out.println(
            "Inside ReplacementFeature.bar()");
    }
}
```

现在需要声明一个 around 通知来截获对跨越 bean 的功能部件的方法调用。CuckoosEgg 类提供了某种 around 通知来检查被截获的方法，并将适当的方法调用传递给 ReplacementFeature 类的实例。

```
public class CuckoosEgg implements MethodInterceptor
{
    public ReplacementFeature replacementFeature =
        new ReplacementFeature();

    public Object invoke(MethodInvocation invocation)
        throws Throwable
    {
        if (invocation.getMethod().getName().equals("foo"))
        {
```

```
            replacementFeature.foo();
        }
        else
        {
            replacementFeature.bar();
        }

        return null;
    }
}
```

因为与 Spring 框架关系密切，Cuckoo's Egg 设计的详细信息被放在 springconfig.xml 配置文件中。对 springconfig.xml 文件的更改将确保所有对 IbusinessLogic 和 IBusinessLogic2 bean 的 foo()方法和 bar()方法的调用都将被截获，并传递给 CuckoosEgg 类的 around 通知。

```
...

<!--CONFIG-->
<bean id="businesslogicbean"
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
        <value>IBusinessLogic</value>
    </property>
    <property name="target">
        <ref local="beanTarget"/>
    </property>
    <property name="interceptorNames">
        <list>
            <value>theCuckoosEggAdvisor</value>
        </list>
    </property>
</bean>

<bean id="businesslogicbean2"
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
        <value>IBusinessLogic2</value>
    </property>
```

```
    <property name="target">
        <ref local="beanTarget2"/>
    </property>
    <property name="interceptorNames">
        <list>
            <value>theCuckoosEgg2Advisor</value>
        </list>
    </property>
</bean>

<!--CLASS-->
<bean id="beanTarget" class="BusinessLogic"/>
<bean id="beanTarget2" class="BusinessLogic2"/>

<!--ADVISOR-->
<bean id="theCuckoosEggAdvisor"
    class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
        <ref local="theReplacementFeaturePart1Advice"/>
    </property>
    <property name="pattern">
        <value>IBusinessLogic.*</value>
    </property>
</bean>

<bean id="theCuckoosEgg2Advisor"
    class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
        <ref local="theReplacementFeaturePart2Advice"/>
    </property>
    <property name="pattern">
        <value>IBusinessLogic2.bar*</value>
    </property>
</bean>

<!--ADVICE-->
<bean id="theReplacementFeaturePart1Advice" class="CuckoosEgg"/>
```

```
<bean id="theReplacementFeaturePart2Advice" class="CuckoosEgg"/>

...
```

当使用修改后的 springconfig.xml 文件运行例子应用程序时，要替换的、被指定为功能部件的一部分的方法调用完全被截获并传递给 ReplacementFeature 类。

通常，即使在同一个实现环境中，我们也可以用不同的方法来实现同一种设计模式。实现上例的另一种方法是实现两个独立的通知。

最后需要注意的是，使用 Cuckoo's Egg 设计模式置换的功能部件，不管它是跨越 bean 的还是在一个类中，它的生命周期与它所置换的功能部件的目标生命周期匹配。在上例中这没什么问题，因为只有一个功能部件实例被置换了，而且唯一的 Cuckoo's Egg 通知只维护一个替代功能部件。

这个例子非常简单，而在实践中，您很可能必须处理大量需要用各自的 Cuckoo's Egg 实例置换的功能部件实例。在这种情况下，单个的方面实例需要被关联到单个的要置换的功能部件实例。本系列的下一篇文章将会考虑方面生命周期的用法，届时将解决这个问题。

结束语

本文介绍了如何在 Spring 框架内谨慎使用 around 形式的通知。around 形式的通知常用于实现 Cuckoo's Egg 设计模式时，所以我们引入了一个例子来说明如何使用 Spring AOP 实现这种面向方面设计模式。

在本系列的第三部分中，您将看到如何使用 Spring 框架中其他的 AOP 基本概念。这些概念包括：控制方面生命周期、使用基于 introduction 通知的积极方面改变应用程序的静态结构，以及使用 control flow 切入点实现对方面编织的更细微的控制。

参考资料

- Spring 框架教程——向您展示 Spring 框架的内部运作方式
- 本文的完整源代码
- Spring Java/J2EE 框架
- “Comparing the Inversion of Control and Dependency Injection Design Patterns”
- IoC 模式概述
- Spring 框架参考文档第 5 章——介绍 Spring 中的所有 AOP 特性