CHAPTER 9

第9章 企业应用开发的思考和策略

本章要点

- 企业应用开发的挑战
- 解决企业应用开发中挑战的思考方式
- 设计模式的背景
- 单例模式
- 简单工厂
- 工厂方法和抽象工厂
- 代理模式
- 命令模式
- 策略模式
- 门面模式
- 桥接模式
- 观察者模式
- 软件架构设计的原则
- 贫血模式
- 领域对象模型
- 领域对象模型的简化设计

企业级应用的开发平台相当多，如 Java EE、Dot Net、PHP 和 Ruby On Rails 等。这些平台为企业级应用的开发提供了丰富的支持，都实现了企业应用底层所需的功能：缓冲池、多线程及持久层访问等。虽然企业开发有如此之多的选择，企业级应用的开发依然困难重重。

本章将会简要介绍企业开发应用过程中所面临的困难，以及面对这些困难时常用的思考方式和应对策略。开发一个大型企业级应用时，常常必须面对各种各样的问题，而这些问题常常具有特定的场景，而且往往会重复出现，借助于前人已有的、较为成熟的解决方案来解决这些问题，既可提供应用开发的效率，也可保证应用开发的质量。这些前人已有的、较为成熟的解决方案就是所谓的设计模式，本章将会深入介绍 Java EE 应用中常用的设计模式。

所有企业级应用的开发平台都提供了高级、抽象的 API，但仅依靠这些 API 构建企业级的应用远远不够。在这些高级 API 基础上，搭建一个良好的开发体系，也是企业级应用开发必不可少的步骤。本章将从理论上介绍如何搭建一个良好、可维护、可扩展、高稳定性且能够快速开发的应用架构，本章还会重点介绍 Java EE 应用中常用的架构模型。

9.1 企业应用开发面临的挑战

企业应用的开发是相当复杂的，这种复杂除了表现在技术方面外，还表现在行业本身所蕴含的专业知识上。企业级应用的开发往往需要面对更多的问题：大量的并发访问，复杂的环境，网络的不稳定，还有外部的 Crack 行为等。因此企业级应用必须提供更好的多线程支持，具备良好的适应性及良好的安全性等。

由于各行业的应用往往差别非常大，因此企业级应用往往具有很强的行业规则，尤其是优良的企业级应用往往更需要丰富的行业知识。企业应用的开发成功，也需要很多人的共同协作。

下面对企业应用开发面临的挑战作具体分析。

➤➤9.1.1 可扩展性、可伸缩性

市场是瞬息万变的，企业也是随之而变的。而信息化系统是为企业服务的，随着企业需求的变化，企业应用的变化也是必然的。

笔者在多年开发过程中，经常听到软件开发者对于需求变更的抱怨。当开发进行到中间时，大量的工作需要重新开始，确实给人极大的挫败感，难免软件开发者会抱怨。不过，笔者认为，一个积极的软件开发者应该可以正确对待需求的变更。需求的变更，表明有市场前景，只有有变化的产品才是有市场的产品。

优秀的企业级应用必须具备良好的可扩展性和可伸缩性。因为良好的可扩展性可允许系统动态增加新功能，而不会影响原有的功能。

良好的可扩展性建立在高度的解耦之上。使用 Delphi、PowerBuilder 等工具的软件开发人员对 ini 文件一定不会陌生。使用 ini 文件是一种基本的解耦方式，将运行所需资源、模块的耦合等从代码中分离出来，放入配置文件管理。这是一种优秀的设计思路，最理想的情况是允许使用可插拔式的模块（类似于 Eclipse 的插件方式）。

在 Java EE 应用里，大多采用 XML 文件作为配置文件。使用 XML 配置文件可以避免修改代码，从而能极好地提高程序的解耦。XML 文件常用于配置数据库连接信息，通过使用 XML 文件的配置方式，可以让应用在不同的数据库平台上轻松切换。从而避免在程序中使用硬编码的方式来定义数据库的连接，也避免了在更改数据库时，需要更改程序代码，从而提供更好的适应性。

下面是使用 Spring 的 Bean 定义数据源的代码。

```
<!-- 定义数据源 Bean，使用 C3P0 数据源实现 -->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
```

```
        destroy-method="close">
        <!-- 指定连接数据库的驱动 -->
        <property name="driverClass" value="com.mysql.jdbc.Driver"/>
        <!-- 指定连接数据库的 URL -->
        <property name="jdbcUrl" value="jdbc:mysql://localhost/auction"/>
        <!-- 指定连接数据库的用户名 -->
        <property name="user" value="root"/>
        <!-- 指定连接数据库的密码 -->
        <property name="password" value="32147"/>
        <!-- 指定连接数据库连接池的最大连接数 -->
        <property name="maxPoolSize" value="40"/>
        <!-- 指定连接数据库连接池的最小连接数 -->
        <property name="minPoolSize" value="1"/>
        <!-- 指定连接数据库连接池的初始化连接数 -->
        <property name="initialPoolSize" value="1"/>
        <!-- 指定连接数据库连接池的连接最大空闲时间 -->
        <property name="maxIdleTime" value="20"/>
</bean>
```

上面的配置文件可用于建立数据库的连接，可等同于如下代码：

```
//创建数据源实例
ComboPooledDataSource ds = new ComboPooledDataSource();
//设置连接数据库的驱动
ds.setDriverClass("com.mysql.jdbc.Driver");
//设置数据库库服务的 URL
ds.setJdbcUrl("jdbc:mysql://localhost:3306/javaee");
//设置数据库用户名
ds.setUser("root");
//设置数据库连接密码
ds.setPassword("32147");
//指定连接数据库连接池的最大连接数
ds.setMaxPoolSize(40);
//指定连接数据库连接池的最小连接数
ds.setMinPoolSize(1);
//指定连接数据库连接池的初始化连接数
ds.setInitialPoolSize(1);
//指定连接数据库连接池的连接最大空闲时间
ds.setMaxIdleTime(true);
```

可以看出，第一种方式明显比第二种方式更优秀。因为当系统的数据库发生变化时（这是相当常见的情形），开发用的数据库与实际应用的数据库不可能是同一个数据库，当软件系统由客户使用时，其数据库系统也是需要改变的。采用第一种方式则无须修改系统源代码，仅通过修改配置文件就可以让系统适应数据库的改变。

使用 XML 配置文件提高解耦的方式，是目前企业级应用最常用的解耦方式，而依赖注入的方式则提供了更高层次的解耦。使用依赖注入可以将各模块之间的调用从代码中分离出来，并通过配置文件来装配组件。此处的依赖注入并非特指 Spring，事实上，依赖注入容器很多，如 HiveMind 等。

➤➤9.1.2 快捷、可控的开发

如果没有时间限制，任何一个软件系统在理论上都是可实现的。但这样的条件不存在，软件系统必须要及时投放市场。对于企业级应用，时间的限制则更加严格。正如前文介绍的，企业的信息是瞬息万变的，与之对应的系统必须能与时俱进。因此快捷、可控是企业信息化系统必须面对的挑战。

软件开发人员常常乐于尝试各种新的技术，总希望将各种新的技术带入项目的开发中，因而难免有时会将整个项目陷入危险的境地。

当然，采用更优秀、更新颖的技术，通常可以保证软件系统的性能更加稳定。例如，从早期的 C/S 架构向 B/S 架构的过渡，以及从 Model 1 到 Model 2 的过渡等。这些都提高了软件系统的可扩展性及可伸缩性。

但采用新的技术所带来的风险也是不得不考虑的，开发架构必须重新论证，开发人员必须重新培

训，这都需要成本投入。如果整个团队缺乏精通该技术的领导者，项目的开发难免会陷入技术难题，从而导致软件的开发过程变成不可控的——这是非常危险的事情。

成功的企业级应用，往往是保证其良好的可扩展性及可伸缩性，并建立在良好的可控性的基础上。

9.1.3 稳定性、高效性

企业级应用还有个显著特点：并发访问量大，访问频繁。因此稳定性、高效性是企业级信息化系统必须达到的要求。

企业级应用必须有优秀的性能，如采用缓冲池的技术。缓冲池专用于保存那些创建开销大的对象，如果对象的创建开销大，花费时间长，该技术可将这些对象缓存，避免了重复创建，从而提高系统性能。典型的应用是数据连接池。

提高企业级应用性能的另一个方法是——数据缓存。但数据缓存有其缺点：数据缓存虽然在内存中，可极好地提高系统的访问速度；但缓存的数据占用了相当大的内存空间，这将会导致系统的性能下降。因此，数据缓存必须根据实际硬件设施制定，最好使用配置文件来动态管理缓存的大小。

9.1.4 花费最小化，利益最大化

这是个永恒的话题，任何一个商业组织都希望尽可能地降低开销。对开发者而言，降低开销主要是如何使在开发上的投资更有保值效果。即开发的软件系统具有很好的复用性，而不是每次面临系统开发任务时，总是需要重复开发。

尽可能让软件可以有高层次的复用，这也是软件行业的发展趋势。早期软件多采用结构化的程序设计语言，此时的软件复用多停留在代码复用的层次。面向对象的程序设计语言的出现，使代码复用提高到了类的复用中。

在良好的 Java EE 架构设计中，复用是一个永恒的追求目标。架构设计师希望系统中大部分的组件可以复用，甚至能让系统的整个层可以复用。对于采用 DAO 模式的系统架构，如果数据库不发生大的改变，整个 DAO 层都不需要变化。

9.2 如何面对挑战

除了在上文介绍的所面临的各种技术挑战之外，企业级应用还有更多的挑战。每个行业都有各自复杂的规则，软件开发者往往缺乏对行业规则的了解。企业级应用的开发通常需要软件开发者和行业专家齐心协作，但系统开发中沟通成本又相当地高，因为软件开发者与行业专家之间的沟通往往存在不少障碍，这些都会影响系统的开发。

面对这些挑战，笔者有如下建议。

9.2.1 使用建模工具

此处的建模工具不一定是 ROSE 等，可以是简单的手画草图。当然，借助于专业的建模工具可以更好地确定系统模型。

任何语言的描述都很空洞，而且具有很大的歧义性。使用图形则更加直观，而且意义更加明确。推荐使用建模工具主要出于如下两个方面的考虑。

用于软件开发者与行业专家之间沟通，正如前文所介绍的，行业专家与软件开发者之间对系统的理解可能存在少许差异。使用图形来帮助交流是不错的主意，通过建模工具绘制的各种图形，可使软件系统的模型更加清晰化。

用于软件开发者之间的沟通。即使在软件开发者内部，对于软件的模型往往也不是非常统一的。使用建模工具可以减少软件开发者对于系统的理解分歧，从而降低沟通成本。

关于建模工具，推荐采用统一建模语言：UML。但 UML 的使用也需要掌握分寸，在软件开发人员内部使用时，尽可能使用规范的 UML；但用于与行业专家沟通时，则应该尽量增加文字说明，而不要拘泥于 UML 图形的表现上，切忌仅将一个图形生硬摆出。

➤➤9.2.2 利用优秀的框架

使用框架可以大大提高系统的开发效率。除非开发一个非常小的系统，而且是开发后无须修改的系统，才可以完全抛弃框架。

优秀的框架本身就是从实际开发中抽取的通用部分，使用框架就可以避免重复开发通用部分。使用优秀的框架不仅可以直接使用框架中的基本组件和类库，还可以提高软件开发人员对系统架构设计的把握。使用框架有如下几个优势。

1. 提高生产效率

框架是在实际开发过程中抽取出来的通用部分。使用框架可以避免开发重复的代码，看下面的 JDBC 数据库访问代码。

```
//注册数据库驱动
Class.forName("com.mysql.jdbc.Driver");
//数据服务的 URL
String url = "jdbc:mysql://localhost/javaee";
//数据库的用户名
String username = "root";
//数据库密码
String password = "32147";
//获取数据库连接
Connection conn = DriverManager.getConnection(url,username,password);
String sql="...";
//创建 PreparedStatement
PreparedStatement pstmt =c onn. prepareStatement(sql);
//为 SQL 语句传入参数
for(int i = 0 ; i < args.length ; i++)
{
     pstmt.setObject(i + 1 , args[i]);
}
//执行更新
pstmt.executeUpdate();
```

上面的代码是连接数据库执行数据更新的代码。而这个过程的大部分都是固定的，包括连接数据库、创建 Statement 及执行更新等，唯一需要变化的是 SQL 语句。

在实际的开发过程中，不可能总是采用这种步骤进行数据库访问，为避免代码重复，可在实际的开发中提取出如下方法：

```
public Class DbBean
{
     //用于执行更新的方法
     public void update(String sql , String[] args)
     {
          //创建 Statement 对象
          PreparedStatement pstmt= getConnection().prepareStatement(sql);
          //为 SQL 语句传入参数
          for(int i = 0 ; i < args.length ; i++)
          {
               pstmt.setObject(i + 1 , args[i]);
          }
          //执行更新
          pstmt.executeUpdate(sql);
     }
```

```
        //获取数据库连接
        private Connection getConnection()
        {
              if (conn == null)
              {
                    //注册数据库驱动
                    Class.forName("com.mysql.jdbc.Driver");
                    //数据服务的 URL
                    String url="jdbc:mysql://localhost/javaee";
                    //数据库的用户名
                    String username="root";
                    //数据库密码
                    String password="32147";
                    //获取数据库连接
                    conn= DriverManager.getConnection(url,username,password);
              }
        }
        ...
  }
```

上面的代码可以大大减少代码的重复量，但依然需要开发者完成连接数据库、创建 Prepared Statement 等步骤。如果使用 Spring 的 JDBC 抽象框架，上面的代码则可以简化为如下：

```
JdbcTemplate jt = new JdbcTemplate();
//为 JdbcTemplate 指定 DataSource
jt.setDataSource(ds);
//更新所使用的 SQL 语句
String sql="...";
//执行更新
jt.update(sql);
```

借助于 Spring 的 JDBC 抽象框架，数据库访问无须手动获取连接，无须创建 Statement 等对象。只需要传入一个 DataSource 对象，由 JdbcTemplate 完成 DataSource 获取数据库连接；创建 Statement 对象；执行数据库更新的通用步骤。而软件开发者只需要提供简单的 SQL 语句即可。

另外，使用框架可以缩短系统的开发时间，特别是对于大型项目的开发，使用框架的优势将更加明显。根据 JavaWorld 社区的调查，使用框架和不使用框架的时间对比如图 9.1 所示。

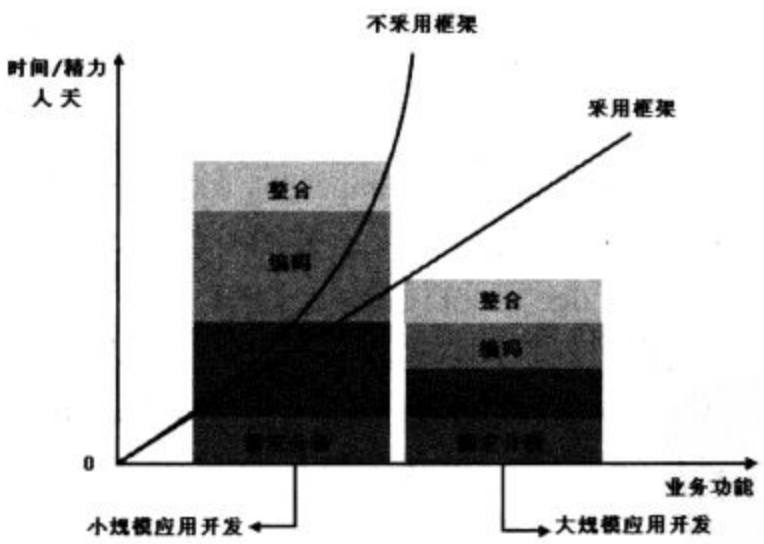

图 9.1　使用框架和不使用框架的对比

2. 具有更稳定、更优秀的性能

如果不使用已有的框架，系统开发者将面临着需要自己完成所有的底层部分。除非开发者丝毫不遵守软件复用的原则，总是重复书写相同代码。

系统开发者从系统开发中提取出的共同部分，也可成为框架。不可否认，完全由开发者自己提取框架有自己的优势，开发人员更加熟悉框架的运行，无须投入成本学习新的技术；但借助于已有框架的优势更加明显，已有的框架通常已被非常多的项目验证过，框架的性能等通常更有保障，而开发者自己提取的框架则可能包含许多未知的隐患。

因此为了更好地了解框架底层的运行，建议使用开源框架。

3. 更好的保值性

采用框架开发的系统使模块组织更加一致，从而降低了软件开发者之间的沟通成本，使系统具有更好的可读性，从而让软件系统具有更好的保值性。

后期的更新、维护也是企业级应用开发的重要组成部分。而使用框架的系统具有很大的相似性，从而有利于后期的更新及维护。

➤➤9.2.3 选择性地扩展

软件的需求千变万化，任何框架不可能总是那么完美，难免需要扩展现有的框架。

在许多项目中，开发者往往喜欢实现自己的框架，认为一个固定的框架会限制其发挥，事实上，他们没有意识到如何扩展框架。虽然开发自己的框架可以获得全部的控制权，但是这也意味着需要很多资源来实现它。正如前文讨论过的，实现自己的框架将需要开发者保证框架的稳定性及性能。

而对已有的框架进行扩展，则可最大限度地利用已有的框架，即使是扩展已有的框架，笔者也不建议盲目扩展，因为新增的部分有时会引入新的风险。笔者建议应对已有框架深入研究，尽量利用已有组件，除非无法使用已有框架时，才考虑选择性地扩展。

➤➤9.2.4 使用代码生成器

使用代码生成器可以自动生成部分程序，不但可以省去许多重复性的劳动，而且在系统开发过程中可以大大节省时间。程序生成器的效率很高，在开发软件的许多环节都有很好的作用，如数据持久化、界面及中间件等。

代码生成器还有个最大的作用：在原型开发期间可以大量重复利用代码生成器。原型系统通常在需求不十分明确时非常有用，此时的需求尚未确定，而软件功能业务无须十分完备，仅提供大致的软件功能，此时的代码生成器就非常有用。

9.3 常见设计模式精讲

设计模式的概念最早起源于建筑设计大师 Alexander 的《建筑的永恒方法》一书，尽管 Alexander 的著作是针对建筑领域的，但他的观点实际上适用于所有的工程设计领域，其中也包括软件设计领域。在《建筑的永恒方法》一书中，Alexander 是这样描述模式的：

模式是一条由三个部分组成的通用规则：它表示了一个特定环境、一类问题和一个解决方案之间的关系。每一个模式描述了一个不断重复发生的问题，以及该问题解决方案的核心设计。

软件领域的设计模式也有类似的定义：设计模式是对处于特定环境下，经常出现的某类软件开发问题的，一种相对成熟的设计方案。

对应到日常生活中，我们无时无刻不在使用模式。例如，当你看前面出现一条水沟时，较为成熟的做法是跳过去——无须重新思索、重新设计下一步该如何做。这个例子很浅显，看上没有什么值得研究的，但试想：如果你从未到过地球（从火星过来？），如果你突然看到一条水沟，你是否需要重新想一想呢？换个思路，当你对软件开发还未熟悉、且刚刚踏入软件开发行业，与一个火星人刚刚踏上地球是否有一定相似之处？所以每次你遇到一个问题时，可能都需要想一想，但通过设计模式就可以直接运用前面成功的经验，从而避免重复设计。

所有资深软件设计师，他们积累了足够的经验，这些经验可以让他们快速、优雅地解决软件开发中的大量重复问题。而设计模式的最终目标就是帮助人们利用软件设计师的集体经验，从而设计出更

加优秀的软件。

很多人容易把设计模式想象成非常高深的概念，笔者还见过一些学员喜欢抱着一本设计模式的书研究，以期成为一个“高手”（笔者估计他肯定是武侠小说看多了），实际上设计模式的理解必须以足够的代码积累量作为基础。因为如果你没有足够的走路、跳跃经验，现在直接告诉你：当遇到一条水沟时，一定要跳过去，你能理解吗？因此学习设计模式之前，必须积累足够的代码量（就像必须走过足够的路一样）。

本节与专门介绍设计模式的图书存在显著的区别：本节的重点不是全面地介绍各种设计模式，而是从实用主义角度来谈设计模式。本节并不仅仅介绍设计模式的实现，因为笔者认为：仅仅研究设计模式的实现是没有意义的。本节将会联系实际 Java EE 应用开发来介绍设计模式，并深入分析 Spring、Hibernate 等框架，以及 Java EE 应用中常用设计模式的应用场景。

根据 Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides（他们是软件设计模式的奠基人）的说法，设计模式常常被分成如下三类。

- 创建型：创建对象时，不再由我们直接实例化对象；而是根据特定场景，由程序来确定创建对象的方式，从而保证更大的性能、更好的架构优势。创建型模式主要有简单工厂模式（并不是 23 种设计模式之一）、工厂方法、抽象工厂模式、单例模式、生成器模式和原型模式。
- 结构型：用于帮助将多个对象组织成更大的结构。结构型模式主要有适配器模式、桥接模式、组合器模式、装饰器模式、门面模式、享元模式和代理模式。
- 行为型：用于帮助系统间各对象的通信，以及如何控制复杂系统中流程。行为型模式主要有命令模式、解释器模式、迭代器模式、中介者模式、备忘录模式、观察者模式、状态模式、策略模式、模板模式和访问者模式。

9.3.1 单例模式

有些时候，允许自由创建某个类的实例没有意义，还可能造成系统性能下降（因为创建对象所带来的系统开销问题）。例如整个系统只有一个窗口管理器，只有一个假脱机打印设备；在 Java EE 应用中可能只需要一个数据库引擎访问点，Hibernate 访问时只需要一个 SessionFactory 实例，如果在系统中为它们创建多个实例就没有太大的意义。

如果一个类始终只能创建一个实例，则这个类被称为单例类，这种模式就被称为单例模式。

对 Spring 框架而言，可以在配置 Bean 实例时指定 scope="singleton"来配置单例模式。不仅如此，如果配置<bean .../>元素时没有指定 scope 属性，则该 Bean 实例默认是单例的行为方式。

Spring 推荐将所有业务逻辑组件、DAO 组件、数据源组件等配置成单例的行为方式，因为这些组件无须保存任何用户状态，故所有客户端都可共享这些业务逻辑组件、DAO 组件，因此推荐将这些组件配置成单例模式的行为方式。

如果不借助 Spring 框架，我们也可手动实现单例模式。为了保证该类只能产生一个实例，程序不能允许自由创建该类的对象，而是只允许为该类创建一个对象。为了避免程序自由创建该类的实例，我们使用 private 修饰该类的构造器，从而将该类的构造器隐藏起来。

将该类的构造器隐藏起来，则需要提供一个 public 方法作为该类的访问点，用于创建该类的对象，且该方法必须使用 static 修饰（因为调用该方法之前还不存在对象，因此调用该方法的不可能是对象，只能是类）。

除此之外，该类还必须缓存已经创建的对象，否则该类无法知道是否曾经创建过实例，也就无法保证只创建一个实例。为此该类需要使用一个静态属性来保存曾经创建的实例，且该属性需要被静态方法访问，所以该属性也应使用 static 修饰。

基于上面的介绍，下面的程序创建了一个单例类。

程序清单：codes\09\9.3\Singleton\SingletonTest.java

```
class Singleton
{
    //使用一个变量来缓存曾经创建的实例
    private static Singleton instance;
    //将构造器使用 private 修饰，隐藏该构造器
    private Singleton(){}
    //提供一个静态方法，用于返回 Singleton 实例
    //该方法可以加入自定义的控制，保证只产生一个 Singleton 对象
    public static Singleton getInstance()
    {
        //如果 instance 为 null，表明还不曾创建 Singleton 对象
        //如果 instance 不为 null，则表明已经创建了 Singleton 对象，将不会执行该方法
        if (instance == null)
        {
            //创建一个 Singleton 对象，并将其缓存起来
            instance = new Singleton();
        }
        return instance;
    }
}
public class SingletonTest
{
    public static void main(String[] args)
    {
        //创建 Singleton 对象不能通过构造器，只能通过 getInstance 方法
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();
        //将输出 true
        System.out.println(s1 == s2);
    }
}
```

上面的程序中第一行粗体字代码使用了一个静态属性来保存已创建的 Singleton 实例，程序第二段粗体字代码用于判断系统是否已经创建过 Singleton 实例——如果已经创建过 Singleton 实例，则直接返回该 Singleton 实例即可。

正是通过上面第二段粗体字代码提供的控制逻辑，从而保证了 Singleton 类只能产生一个实例。所以在 TestSingleton 类的 main 方法中看到两次产生的 Singleton 对象实际上是同一个对象。

在 Java EE 应用中，单例模式是一种应用非常广泛的设计模式，应用中许多组件都只需要单个实例，下面介绍的工厂模式里的工厂也只需要单个实例……

使用单例模式主要有如下两个优势：

- 减少创建 Java 实例所带来的系统开销。
- 便于系统跟踪单个 Java 实例的生命周期、实例状态等。

➤➤9.3.2 简单工厂

对于一个典型的 Java 应用而言，应用之中各实例之间存在复杂的调用关系（Spring 把这种调用关系称为依赖关系，例如 A 实例调用 B 实例的方法，则称为 A 依赖于 B）。

当 A 对象需要调用 B 对象的方法时，许多初学者会选择使用 new 关键字来创建一个 B 实例，然后调用 B 实例的方法。从语法的角度来看，这种做法没有任何问题，这种做法的坏处在于：A 类的方法实现直接调用了 B 类的类名（这种方式也被称为硬编码耦合），一旦系统需要重构：需要使用 C 类来代替 B 类时，程序不得不改写 A 类代码。如果应用中有 100 个或 10000 个类以硬编码方式耦合了 B 类，则需要重新改写 100 个、10000 个地方……这显然是一种非常可怕的事情。

换一个角度来看这个问题：对于 A 对象而言，它只需要调用 B 对象的方法，并不是关心 B 对象的实现、创建过程。考虑让 B 类实现一个 IB 接口，而 A 类只需要 IB 接口耦合——A 类并不直接使用

new 关键字来创建 B 实例，而是重新定义一个工厂类：IBFactory，由该工厂类来负责创建 IB 实例；而 A 类通过调用 IBFactory 工厂的方法来得到 IB 的实例。

通过改用上面设计，则 A 类需要与 IBFactory 耦合，还需要与 IB 接口耦合，如果系统需要重构：需要使用 C 类代替 B 类，则只需要让 C 类也实现 IB 接口，并改写 IBFactory 工厂中创建 IB 实例的实现代码，让该工厂产生 C（实现了 IB 接口）实例即可。由于所有依赖 IB 实例的对象都是通过工厂来获取 IB 实例的，所以它们都将改为获得 C 实例，这就完成了系统重构。

这种将多个类对象交给工厂类来生成的设计方式被称为简单工厂模式。

下面以一个简单的场景来介绍简单工厂模式，假设程序中有个 Computer 对象需要依赖一个输出设备，现在有两个选择：直接让 Computer 对象依赖一个 Printer（实现类）对象，或者让 Computer 依赖一个 Output（接口）属性。

在这种应用场景下，使用简单工厂模式可以让系统具有更好的可维护性、可扩展性。根据工厂模式，程序应该让 Computer 依赖一个 Output 属性，将 Computer 类与 Printer 实现类分离开来。Computer 对象只需面向 Output 接口编程即可；而 Computer 具体依赖于 Output 的哪个实现类则完全透明。

下面是这个 Computer 类定义的代码。

程序清单：codes\09\9.3\SimpleFactory\Computer.java

```
public class Computer
{
    private Output out;
    public Computer(Output out)
    {
        this.out = out;
    }
    //定义一个模拟获取字符串输入的方法
    public void keyIn(String msg)
    {
        out.getData(msg);
    }
    //定义一个模拟打印的方法
    public void print()
    {
        out.out();
    }
    public static void main(String[] args)
    {
        //创建 OutputFactory
        OutputFactory of = new OutputFactory();
        //将 Output 对象传入，创建 Computer 对象
        Computer c = new Computer(of.getOutput());
        c.keyIn("轻量级 Java EE 企业应用实战");
        c.keyIn("疯狂 Java 讲义");
        c.print();
    }
}
```

从上面粗体字代码可以看出，该 Computer 类已经完全与 Output 实现类分离了，它只与该接口耦合。而且，Computer 不再负责创建 Output 对象，系统将提供一个 Output 工厂来负责生成 Output 对象。这个 OutputFactory 工厂类代码如下。

程序清单：codes\09\9.3\SimpleFactory\OutputFactory.java

```
public class OutputFactory
{
    public Output getOutput()
    {
        //下面两行代码用于控制系统到底使用 Output 的哪个实现类
        return new Printer();
    }
}
```

在该 OutputFactory 类中包含了一个 getOutput 方法，该方法返回一个 Output 实现类的实例，该方法负责创建 Output 实例，具体创建哪一个实现类的对象由该方法决定，具体由该方法中粗体部分控制，当然也可以增加更复杂的控制逻辑。程序中粗体字代码创建了一个 Printer 对象，Printer 类代码如下。

程序清单：codes\09\9.3\SimpleFactory\Printer.java

```
//让 Printer 类实现 Output
public class Printer implements Output
{
    private String[] printData = new String[MAX_CACHE_LINE];
    //用以记录当前需打印的作业数
    private int dataNum = 0;
    public void out()
    {
        //只要还有作业，继续打印
        while(dataNum > 0)
        {
            System.out.println("打印机打印：" + printData[0]);
            //把作业队列整体前移一位，并将剩下的作业数减 1
            System.arraycopy(printData , 1, printData, 0, --dataNum);
        }
    }
    public void getData(String msg)
    {
        if (dataNum >= MAX_CACHE_LINE)
        {
            System.out.println("输出队列已满，添加失败");
        }
        else
        {
            //把打印数据添加到队列里，已保存数据的数量加 1。
            printData[dataNum++] = msg;
        }
    }
}
```

上面的 Printer 类模拟了一个简单的打印机，如果系统需要重构，需要使用 BetterPrinter 来代替 Printer 类，则只需要让 BetterPrinter 实现 Output 接口，并改写 OutputFactory 类的 getOutput()方法即可。

下面是 BetterPrinter 实现类的代码。BetterPrinter 只是对原有的 Printer 进行简单修改，以模拟系统重构后的改进。

程序清单：codes\09\9.3\SimpleFactory\BetterPrinter.java

```
public class BetterPrinter implements Output
{
    private String[] printData = new String[MAX_CACHE_LINE * 2];
    //用以记录当前需打印的作业数
    private int dataNum = 0;
    public void out()
    {
        //只要还有作业，继续打印
        while(dataNum > 0)
        {
            System.out.println("高速打印机正在打印：" + printData[0]);
            //把作业队列整体前移一位，并将剩下的作业数减 1
            System.arraycopy(printData , 1, printData, 0, --dataNum);
        }
    }
    public void getData(String msg)
    {
        if (dataNum >= MAX_CACHE_LINE * 2)
        {
            System.out.println("输出队列已满，添加失败");
        }
        else
        {
```

```
                //把打印数据添加到队列里，已保存数据的数量加 1。
                printData[dataNum++] = msg;
            }
        }
    }
```

上面的程序中 BetterPrinter 类与 Printer 并无太大区别，仅仅略微改变了 Printer 实现，且 BetterPrinter 也实现了 Output 接口，因此也可当成 Output 对象使用，于是我们只要把 OutputFactory 工厂类的 getOutput()方法中粗体部分改为如下代码：

```
return new BetterPrinter();
```

再次运行前面的 Computer.java 程序，发现 Computer 所依赖的 Output 对象已改为 BetterPrinter 对象，而不再是原来的 Printer 对象。

通过这种方式，我们把所有生成 Output 对象的逻辑集中在 OutputFactory 工厂类中管理，而所有需要使用 Output 对象的类只需与 Output 接口耦合，而不是与具体的实现类耦合。即使系统中有很多类依赖了 Printer 对象，只要 OutputFactory 类的 getOutput()方法返回 BetterPrinter 对象，则它们全部将会改为依赖 BetterPrinter 对象，而其他程序无须修改，只需要修改 OutputFactory 工厂的 getOutput 的方法实现即可。

使用简单工厂模式的优势是：让对象的调用者和对象创建过程分离，当对象调用者需要对象时，直接向工厂请求即可。从而避免了对象的调用者与对象的实现类以硬编码方式耦合，以提高系统的可维护性、可扩展性。工厂模式也有一个小小的缺陷：当产品修改时，工厂类也要做相应的修改。

对 Spring 容器而言，它首先是一个巨大的工厂，它负责创建所有 Bean 实例，整个应用的所有组件都由 Spring 容器负责创建。不仅如此，Spring 容器扩展了这种简单工厂模式，它还可以管理 Bean 实例之间的依赖关系；而且，如果容器中 Bean 实例具有 singleton 行为特征，则 Spring 容器还会缓存该 Bean 实例，从而保证程序通过 Spring 工厂来获取该 Bean 实例时，Spring 工厂将会返回同一个 Bean 实例。

下面的示例提供一份类似于 Spring 配置文件的 XML 文件，程序提供一个扩展的工厂类，该工厂类也可提供类似于 Spring IoC 容器的功能。

程序清单：codes\09\9.3\IoC\bean.xml

```
<?xml version="1.0" encoding="GBK"?>
<beans>
    <bean id="computer" class="lee.Computer">
        <!-- 为 name 属性注入普通属性值 -->
        <property name="name" value="李刚的电脑"/>
        <!-- 为 out 属性注入普通属性值 -->
        <property name="out" ref="betterPrinter"/>
    </bean>
    <!-- 配置两个 Bean 实例 -->
    <bean id="printer" class="lee.Printer"/>
    <bean id="betterPrinter" class="lee.BetterPrinter"/>
    <!-- 配置一个 prototype 行为的 Bean 实例 -->
  <bean id="now" class="java.util.Date" scope="prototype"/> <!--①-->
</beans>
```

细心的读者可能已经发现：该配置文件和 Spring 配置文件如此相似。实际上，笔者只是简单修改了 Spring 配置文件。上面的配置文件一样配置了 computer Bean，且为该 Bean 依赖注入了 2 个属性：name 和 out。除此之外，上面的配置文件中①号代码处还配置了一个 prototype 行为的 Bean 实例。

本程序中也提供了一个简化的 ApplicationContext 接口，该接口仅包含一个 getBean()方法。

程序清单：codes\09\9.3\IoC\src\org\crazyit\ioc\ApplicationContext.java

```
public interface ApplicationContext
{
    //获取指定 Bean 实例的方法
    Object getBean(String name)
        throws Exception;
```

```
}
```

本示例将为该接口提供一个简单的实现类，该实现类就是一个功能强大的工厂。它使用 Dom4j 来解析 XML 配置文件，并根据配置文件来创建工厂中的 Bean 实例。下面是该实现类的代码。

程序清单：codes\09\9.3\IoC\src\org\crazyit\ioc\YeekuXmlApplicationContext.java

```
public class YeekuXmlApplicationContext
    implements ApplicationContext
{
    //保存容器中所有单例模式的 Bean 实例
    private Map<String , Object> objPool
        = Collections.synchronizedMap(new HashMap<String , Object>());
    //保存配置文件对应的 Document 对象
    private Document doc;
    //保存配置文件里的根元素
    private Element root;
    public YeekuXmlApplicationContext(String filePath)
        throws Exception
    {
        SAXReader reader = new SAXReader();
        doc = reader.read(new File(filePath));
        root = doc.getRootElement();
        initPool();
        initProp();
    }
    public Object getBean(String name)
        throws Exception
    {
        Object target = objPool.get(name);
        //对于 singleton Bean，容器已经初始化了所有 Bean 实例
        if (target.getClass() != String.class)
        {
            return target;
        }
        else
        {
            String clazz = (String)target;
            //对于 prototype 并未注入属性值
            return Class.forName(clazz).newInstance();
        }
    }
    //初始化容器中所有 singleton Bean
    private void initPool()
        throws Exception
    {
        //遍历配置文件里的每个<bean.../>元素
        for (Object obj : root.elements())
        {
            Element beanEle = (Element)obj;
            //取得<bean.../>元素的 id 属性
            String beanId = beanEle.attributeValue("id");
            //取得<bean.../>元素的 class 属性
            String beanClazz = beanEle.attributeValue("class");
            //取得<bean.../>元素的 scope 属性
            String beanScope = beanEle.attributeValue("scope");
            //如果<bean.../>元素的 scope 属性不存在，或为 singleton
            if (beanScope == null ||
                beanScope.equals("singleton"))
            {
                //以默认构造器创建 Bean 实例，并将其放入 objPool 中
                objPool.put(beanId , Class.forName(beanClazz).newInstance());
            }
            else
            {
                //对于非 singlton Bean，存放该 Bean 实现类的类名。
                objPool.put(beanId , beanClazz);
            }
```

```
        }
    }
    //初始化容器中singleton Bean的属性
    private void initProp()
        throws Exception
    {
        //遍历配置文件里的每个<bean.../>元素
        for (Object obj : root.elements())
        {
            Element beanEle = (Element)obj;
            //取得<bean.../>元素的id属性
            String beanId = beanEle.attributeValue("id");
            //取得<bean.../>元素的scope属性
            String beanScope = beanEle.attributeValue("scope");
            //如果<bean.../>元素的scope属性不存在，或为singleton
            if (beanScope == null ||
                beanScope.equals("singleton"))
            {
                //取出objPool的指定的Bean实例
                Object bean = objPool.get(beanId);
                //遍历<bean.../>元素的每个<property.../>子元素
                for (Object prop : beanEle.elements())
                {
                    Element propEle = (Element)prop;
                    //取得<property.../>元素的name属性
                    String propName = propEle.attributeValue("name");
                    //取得<property.../>元素的value属性
                    String propValue = propEle.attributeValue("value");
                    //取得<property.../>元素的ref属性
                    String propRef = propEle.attributeValue("ref");
                    //将属性名的首字母大写
                    String propNameCamelize = propName.substring(0 , 1). toUpper-
                    Case()
                        + propName.substring(1 , propName.length());
                    //如果<property.../>元素的value属性值存在
                    if (propValue != null && propValue.length() > 0)
                    {
                        //获取设值注入所需的setter方法
                        Method setter = bean.getClass().getMethod(
                            "set" + propNameCamelize , String.class);
                        //执行setter注入
                        setter.invoke(bean , propValue);
                    }
                    if (propRef != null && propRef.length() > 0)
                    {
                        //取得需要被依赖注入的Bean实例
                        Object target = objPool.get(propRef);
                        //objPool池中不存在指定Bean实例
                        if (target == null)
                        {
                            //此处还应处理Singleton Bean依赖prototype Bean的情形
                        }
                        //定义设值注入所需的setter方法
                        Method setter = null;
                        //遍历target对象所所实现的所有接口
                        for (Class superInterface : target.getClass().get-
                        Interfaces())
                        {
                            try
                            {
                                //获取设值注入所需的setter方法
                                setter = bean.getClass().getMethod(
                                        "set" + propNameCamelize ,
                                        superInterface);
                                //如果成功取得该接口对应的方法，直接跳出循环
                                break;
                            }
```

```
                                        catch (NoSuchMethodException ex)
                                        {
                                                //如果没有找到对应的 setter 方法，继续下次循环
                                                continue;
                                        }
                                }
                                //如果 setter 方法依然为 null，
                                //则直接取得 target 实现类对应的 setter 方法
                                if (setter == null)
                                {
                                        setter = bean.getClass().getMethod(
                                                "set" + propNameCamelize , target.
                                                getClass());
                                }
                                //执行 setter 注入
                                setter.invoke(bean , target);
                        }
                }
            }
        }
    }
}
```

上面的 YeekuXmlApplicationContext 类是一个功能强大的工厂类，它根据 XML 配置文件创建 Bean 实例，程序需要 Bean 实例时只需调用该工厂类的 getBean()方法即可。

上面的 YeekuXmlApplicationContext 类当然不能与 Spring 的 ApplicationContext 实现类相比，该容器类仅仅实现了简单的 IoC 功能，而且并未为 prototype 行为的 Bean 的属性提供依赖注入功能。读者可以通过该工厂类大致了解 Spring 底层的实现原理。

下面是测试该工厂类的主类。

程序清单：codes\09\9.3\IoC\src\lee\IoCTest.java

```
public class IoCTest
{
    public static void main(String[] args)
        throws Exception
    {
        //创建 IoC 容器
        ApplicationContext ctx =
            new YeekuXmlApplicationContext("bean.xml");
        //从 IoC 容器中取出 computer Bean
        Computer c = (Computer)ctx.getBean("computer");
        //测试 Computer 对象
        c.keyIn("轻量级 Java EE 企业应用实战");
        c.keyIn("疯狂 Java 讲义");
        c.print();
        System.out.println(ctx.getBean("now"));
    }
}
```

从上面程序中的粗体字代码可以看出，本程序的 IoC 容器具有和 Spring 容器类似的功能，同样可以创建并管理容器中所有的 Bean 实例。

与简单工厂模式类似的还有工厂方法和抽象工厂模式，下面将进一步讲解工厂方式和抽象工厂模式的设计方式。

➤➤9.3.3 工厂方法和抽象工厂

在简单工厂模式里，系统使用 ProductFactory 工厂类生产所有产品实例，且该工厂类决定生产哪个类的实例，即该工厂类负责所有的逻辑判断、实例创建等工作。

如果我们不想在工厂类中进行逻辑判断，程序可以为不同产品类提供不同的工厂，不同的工厂类生产不同的产品。例如我们为上面的 Printer、BetterPrinter 分别提供 PrinterFactory 和 BetterPrinterFactory

工厂类，这就无须在工厂类进行复杂的逻辑判断。

本示例应用将 OutputFactory 改为一个接口，并为该接口提供两个实现类：PrinterFactory.java 和 BetterPrinterFactory.java。下面是 OutputFactory 接口的代码。

程序清单：codes\09\9.3\FactoryMethod\OutputFactory.java

```
public interface OutputFactory
{
    //仅定义一个方法用于返回输出设备。
    Output getOutput();
}
```

上面的 OutputFactory 只是一个接口，该接口提供了一个 getOutput()方法，该方法可直接返回一个输出设备。

下面为 OutputFactory 接口提供一个 PrinterFactory 实现类，该实现类专门负责生成 Printer 实例。

程序清单：codes\09\9.3\FactoryMethod\PrinterFactory.java

```
public class PrinterFactory
    implements OutputFactory
{
    public Output getOutput()
    {
        //该工厂只负责产生 Printer 对象
        return new Printer();
    }
}
```

上面的 PrinterFactory 实现了 OutputFactory 接口，并实现了该接口里的 getOutput()方法，该方法直接返回一个简单的 Printer 对象，如上面的粗体字代码所示。

下面再为 OutputFactory 接口提供一个 BetterPrinterFactory 实现类，该实现类专门负责生成 BetterPrinter 实例。

程序清单：codes\09\9.3\FactoryMethod\BetterPrinterFactory.java

```
public class BetterPrinterFactory
    implements OutputFactory
{
    public Output getOutput()
    {
        //该工厂只负责产生 BetterPrinter 对象
        return new BetterPrinter();
    }
}
```

本示例应用中各类之间的类图如图 9.2 所示。

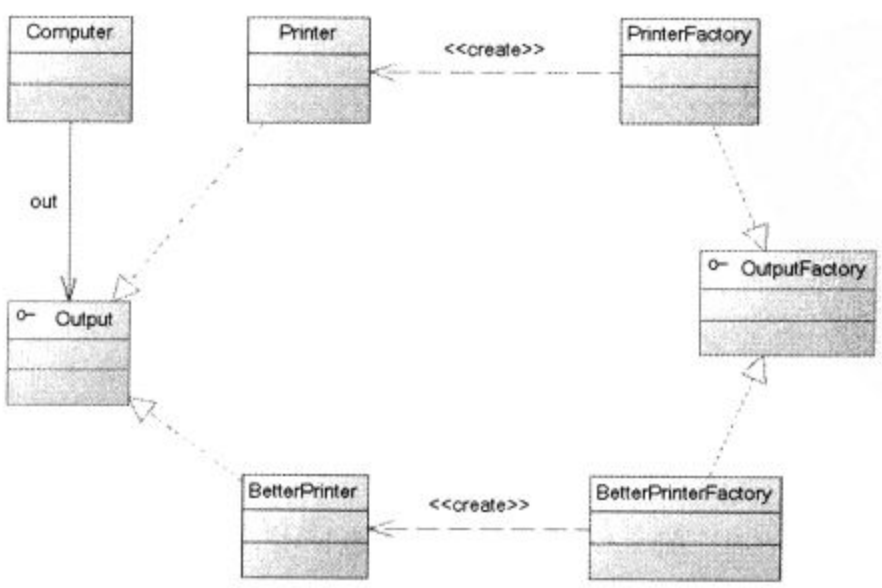

图 9.2　工厂方法中各类的类图

当使用工厂方法设计模式时，对象调用者需要与具体的工厂类耦合：当需要不同对象时，程序需

要调用相应工厂对象的方法来得到所需的对象。如下是 Computer 类中创建 Output 对象并调用该对象方法的代码。

程序清单：codes\09\9.3\FactoryMethod\Computer.java

```
public class Computer
{
    private Output out;
    public Computer(Output out)
    {
        this.out = out;
    }
    //定义一个模拟获取字符串输入的方法
    public void keyIn(String msg)
    {
        out.getData(msg);
    }
    //定义一个模拟打印的方法
    public void print()
    {
        out.out();
    }
    public static void main(String[] args)
    {
        //使用 PrinterFactory 子类来创建 OutputFactory
        OutputFactory of = new PrinterFactory();
        //将 Output 对象传入，创建 Computer 对象
        Computer c = new Computer(of.getOutput());
        c.keyIn("轻量级 Java EE 企业应用实战");
        c.keyIn("疯狂 Java 讲义");
        c.print();
    }
}
```

正如程序中的粗体字代码所示，当客户端代码需要调用 Ouput 对象的方法时，为了得到不同的 Output 实例，程序必须显式创建不同的 OutputFactory 实例，程序中创建的是 PrinterFactory 实例。

从上面的代码可以看出，对于采用工厂方法的设计架构，客户端代码成功与被调用对象的实现类分离，但带来了另一种耦合：客户端代码与不同的工厂类耦合。这依然是一个问题！

为了解决客户端代码与不同工厂类耦合的问题，接着考虑再增加一个工厂类，该工厂类不是生产 Output 对象，而是生产 OutputFactory 实例——简而言之，这个工厂类不制造具体的被调用对象，而是制造不同工厂对象。这个特殊的工厂类被称呼抽象工厂类，这种设计方式也被称为抽象工厂模式。如图 9.3 所示是抽象工厂模式示例的 UML 类图。

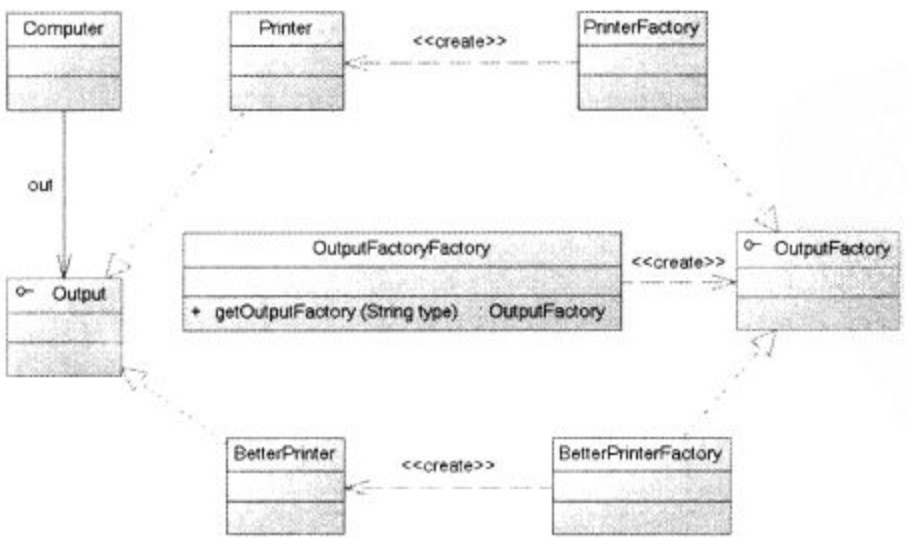

图 9.3　抽象工厂模式的 UML 类图

从图 9.3 中可以看出，在这种模式下系统新增了一个 OutputFactoryFactory 工厂类，该工厂类提供了一个 getOutputFactory(String type)方法，该方法用于返回一个 OutputFactory 工厂实例。下面是该抽

象工厂类的代码。

程序清单：codes\09\9.3\AbstractFactory\OutputFactoryFactory.java

```
public class OutputFactoryFactory
{
    //仅定义一个方法用于返回输出设备。
    public static OutputFactory getOutputFactory(
        String type)
    {
        if (type.equalsIgnoreCase("better"))
        {
            return new BetterPrinterFactory();
        }
        else
        {
            return new PrinterFactory();
        }
    }
}
```

从上面的粗体字代码中可以看出，抽象工厂根据 type 参数进行判断，决定需要生成哪种工厂实例。通过这种设计模式，就可让客户端程序只需与抽象工厂类耦合。下面是客户端调用被调用者对象方法的主方法。

程序清单：codes\09\9.3\AbstractFactory\Computer.java

```
public static void main(String[] args)
{
    //使用 PrinterFactory 子类来创建 OutputFactory
    OutputFactory of = OutputFactoryFactory
        .getOutputFactory("better");
    //将 Output 对象传入，创建 Computer 对象
    Computer c = new Computer(of.getOutput());
    c.keyIn("轻量级 Java EE 企业应用实战");
    c.keyIn("疯狂 Java 讲义");
    c.print();
}
```

上面程序中的粗体字代码用于产生一个 OutputFactory 工厂，但具体产生哪个工厂则由 OutputFactoryFactory 抽象工厂决定，不同的工厂对象将产生不同的 Output 对象。通过采用抽象工厂的设计模式，系统可以让客户端代码与被调用对象的实现类、具体的工厂类分离。

读者掌握了这种抽象工厂模式后，应该对 Spring IoC 容器感到迷惑：它到底是简单工厂？还是抽象工厂？实际上，笔者倾向于认为 Spring IoC 容器是抽象工厂，因为 Sping IoC 容器可以包括万象，它不仅可以管理普通 Bean 实例，也可管理工厂实例。

很多时候，笔者不喜欢纠缠于简单工厂模式、抽象工厂模式这些概念，笔者喜欢把它们统称为工厂模式。如果工厂直接生产被调用对象，那就是简单工厂模式；如果工厂生产了工厂对象，那就会升级成抽象工厂模式。

➤➤9.3.4 代理模式

代理模式是一种应用非常广泛的设计模式，当客户端代码需要调用某个对象时，客户端实际上也不关心是否准确得到该对象，它只要一个能提供该功能的对象即可，此时我们就可返回该对象的代理（Proxy）。

在这种设计方式下，系统会为某个对象提供一个代理对象，并由代理对象控制对源对象的引用。代理就是一个 Java 对象代表另一个 Java 对象来采取行动。在某些情况下，客户端代码不想或不能够直接调用被调用者，代理对象可以在客户和目标对象之间起到中介的作用。

对客户端而言，它不能分辨出代理对象与真实对象的区别，它也无须分辨代理对象和真实对象的

区别。客户端代码并不知道真正的被代理对象，客户端代码面向接口编程，它仅仅持有一个被代理对象的接口。

总而言之，只要客户端代码不能或不想直接访问被调用对象——这种情况有很多原因，比如需要创建一个系统开销很大的对象，或者被调用对象在远程主机上，或者目标对象的功能还不足以满足需求……，而是额外创建一个代理对象返回给客户端使用，那么这种设计方式就是代理模式。

下面示范一个简单的代理模式，程序首先提供了一个 Image 接口，代表大图片对象所实现的接口。

程序清单：codes\09\9.3\Proxy\Image.java

```
public interface Image
{
    void show();
}
```

该接口提供了一个实现类，该实现类模拟了一个大图片对象，该实现类的构造器使用 Thread.sleep() 方法来暂停 3s。下面是该 BigImage 的程序代码。

程序清单：codes\09\9.3\Proxy\BigImage.java

```
//使用该 BigImage 模拟一个很大图片
public class BigImage implements Image
{
    public BigImage()
    {
        try
        {
            //程序暂停 3s 模式模拟系统开销
            Thread.sleep(3000);
            System.out.println("图片装载成功...");
        }
        catch (InterruptedException ex)
        {
            ex.printStackTrace();
        }
    }
    //实现 Image 里的 show()方法
    public void show()
    {
        System.out.println("绘制实际的大图片");
    }
}
```

上面程序的粗体字代码暂停了 3s，这表明创建一个 BigImage 对象需要 3s 的时间开销——程序使用这种延迟来模拟装载此图片所导致的系统开销。如果不采用代理模式，当程序中创建 BigImage 时，系统将会产生 3s 的延迟。为了避免这种延迟，程序为 BigImage 对象提供一个代理对象，BigImage 类的代理类如下所示。

程序清单：codes\09\9.3\Proxy\ImageProxy.java

```
public class ImageProxy implements Image
{
    //组合一个 image 实例，作为被代理的对象
    private Image image;
    //使用抽象实体来初始化代理对象
    public ImageProxy(Image image)
    {
        this.image = image;
    }
    /**
     * 重写 Image 接口的 show()方法
     * 该方法用于控制对被代理对象的访问，
     * 并根据需要负责创建和删除被代理对象
     */
    public void show()
    {
```

```
            //只有当真正需要调用 image 的 show 方法时才创建被代理对象
            if (image == null)
            {
                image = new BigImage();
            }
            image.show();
        }
    }
```

上面的 ImageProxy 代理类实现了与 BigImage 相同的 show()方法，这使得客户端代码获取到该代理对象之后，可以将该代理对象当成 BigImage 来使用。

在 ImageProxy 类的 show()方法中增加了粗体字代码的控制逻辑，这段控制逻辑用于控制当系统真正调用 image 的 show()时，才会真正创建被代理的 BigImage 对象。下面程序需要使用 BigImage 对象，但程序并不是直接返回 BigImage 实例，而是先返回 BigImage 的代理对象，如下面的程序所示。

程序清单：codes\09\9.3\Proxy\BigImageTest.java

```
public class BigImageTest
{
    public static void main(String[] args)
    {
        long start = System.currentTimeMillis();
        //程序返回一个 Image 对象，该对象只是 BigImage 的代理对象
        Image image = new ImageProxy(null);
        System.out.println("系统得到 Image 对象的时间开销:" +
                (System.currentTimeMillis() - start));
        //只有当实际调用 image 代理的 show()方法时，程序才会真正创建被代理对象。
        image.show();
    }
}
```

上面的程序初始化 image 非常快，因为程序并未真正创建 BigImage 对象，只是得到了 ImageProxy 代理对象——直到程序调用 image.show()方法时，程序需要真正调用 BigImage 对象的 show()方法，程序此时才真正创建 BigImage 对象。运行上面程序，看到如图 9.4 所示的结果。

看到如图 9.4 所示的运行结果，读者应该能认同：使用代理模式提高了获取 Image 对象的系统性能。但可能有读者会提出疑问：程序调用 ImageProxy 对象的 show()方法时一样需要创建 BigImage 对象啊，系统开销并未真正减少啊？只是这种系统开销延迟了而已？

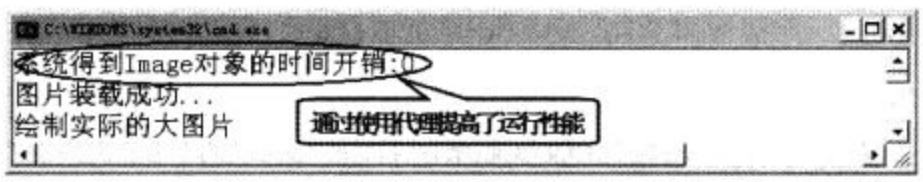

图 9.4　使用代理模式提高性能

我们可以从如下两个角度来回答这个问题：

- 把创建 BigImage 推迟到真正需要它时才创建，这样能保证前面程序运行的流畅性，而且能减少 BigImage 在内存中的存活时间，从宏观上节省了系统的内存开销。
- 在有些情况下，也许程序永远不会真正调用 ImageProxy 对象的 show()方法——意味着系统根本无须创建 BigImage 对象。在这种情形下，使用代理模式可以显著地提高系统运行性能。

第二种情况正是 Hibernate 延迟加载所采用的设计模式，相信读者还记得前面介绍 Hibernate 关联映射时的知识，当 A 实体和 B 实体之间存在关联关系时，Hibernate 默认启用延迟加载，当系统加载 A 实体时，A 实体关联的 B 实体并未被加载出来，A 实体所关联的 B 实体全部是代理对象——只有等到 A 实体真正需要访问 B 实体时，系统才会去数据库里抓取 B 实体所对应的记录。

Hibernate 的延迟加载充分体现了代理模式的优势：当系统加载 A 实体时，也许只需要访问 A 实体对应的记录，根本不会访问 A 的关联实体。如果不采用代理模式，系统需要在加载 A 实体时，同时加载 A 实体的所有关联实体——这是多么大的系统开销啊！

除了上面出于性能考虑使用代理模式之外，代理模式还有另一种常用场景：当目标对象的功能不足以满足客户端需求时，系统可以为该对象创建一个代理对象，而代理对象可以增强原目标对象的功能。

借助于 Java 提供的 Proxy 和 InvocationHandler，可以实现在运行时生成动态代理的功能，而动态代理对象就可作为目标对象使用，而且增强了目标对象的功能。

由于 JDK 动态代理只能创建指定接口的动态代理，所以下面先提供一个 Dog 接口，该接口代码非常简单，仅仅在该接口里定义了两个方法。

程序清单：codes\09\9.3\DynaProxy\Dog.java

```
public interface Dog
{
    //info 方法声明
    public void info();
    //run 方法声明
    public void run();
}
```

上面接口里只是简单定义了两个方法，并未提供方法实现。下面程序先为该接口提供一个实现类，该实现类的实例将会作为被代理的目标对象。下面是该接口实现类的代码。

程序清单：codes\09\9.3\DynaProxy\GunDog.java

```
public class GunDog implements Dog
{
    //info 方法实现，仅仅打印一个字符串
    public void info()
    {
        System.out.println("我是一只猎狗");
    }
    //run 方法实现，仅仅打印一个字符串
    public void run()
    {
        System.out.println("我奔跑迅速");
    }
}
```

上面的代码没有丝毫的特别之处，该 Dog 的实现类仅仅为每个方法提供了一个简单实现。现在假设该目标对象（GunDog）实例的两个方法不能满足实际需要，因此客户端不想直接调用该目标对象。假设客户端需要在 GunDog 为两个方法增加事务控制：在目标方法被调用之前开始事务，在目标方法被调用之后结束事务。

为了实现该功能，我们可以为 GunDog 对象创建一个代理对象，该代理对象提供与 GunDog 对象相同的方法，而代理对象增强了 GunDog 对象的功能。

下面先提供一个 TxUtil 类（这个类通常被称为拦截器），该类里包含两个方法，分别用于开始事务、提交事务，下面是 TxUtil 类的源代码。

程序清单：codes\09\9.3\DynaProxy\TxUtil.java

```
public class TxUtil
{
    //第一个拦截器方法:模拟事务开始
    public void beginTx()
    {
        System.out.println("=====模拟开始事务=====");
    }
    //第二个拦截器方法:模拟事务结束
    public void endTx()
    {
        System.out.println("=====模拟结束事务=====");
    }
}
```

借助于 Proxy 和 InvocationHandler 就可以实现：当程序调用 info()方法和 run()方法时，系统可以

“自动”将 beginTx()和 endTx()两个通用方法插入 info()和 run()方法执行中。

JDK 动态代理的关键在于下面的 MyInvokationHandler 类，该类是一个 InvocationHandler 实现类，该实现类的 invoke 方法将会作为代理对象的方法实现。

程序清单：codes\09\9.3\DynaProxy\MyInvokationHandler.java

```
public class MyInvokationHandler
      implements InvocationHandler
{
      //需要被代理的对象
      private Object target;
      public void setTarget(Object target)
      {
            this.target = target;
      }
      //执行动态代理对象的所有方法时，都会被替换成执行如下的 invoke 方法
      public Object invoke(Object proxy, Method method, Object[] args)
            throws Exception
      {
            TxUtil tx = new TxUtil();
            //执行 TxUtil 对象中的 beginTx。
            tx.beginTx();
            //以 target 作为主调来执行 method 方法
            Object result = method.invoke(target , args);
            //执行 TxUtil 对象中的 endTx。
            tx.endTx();
            return result;
      }
}
```

上面的 invoke()方法将会作为动态代理对象的所有方法的实现体。上面方法中第一行粗体字代码调用了开始事务的方法，第二行粗体字代码通过反射回调了被代理对象的目标方法，第三行粗体字代码调用了结束事务的方法。通过这种方式，使得代理对象的方法既回调了被代理对象的方法，并为被代理对象的方法增加了事务功能。

下面再为程序提供一个 MyProxyFactory 类，该对象专为指定的 target 生成动态代理实例。

程序清单：codes\09\9.3\DynaProxy\MyProxyFactory.java

```
public class MyProxyFactory
{
      //为指定 target 生成动态代理对象
      public static Object getProxy(Object target)
            throws Exception
      {
            //创建一个 MyInvokationHandler 对象
            MyInvokationHandler handler =
                  new MyInvokationHandler();
            //为 MyInvokationHandler 设置 target 对象
            handler.setTarget(target);
            //创建、并返回一个动态代理
            return Proxy.newProxyInstance(target.getClass().getClassLoader()
                  , target.getClass().getInterfaces(), handler);
      }
}
```

上面的动态代理工厂类提供了一个 getProxy 方法，该方法为 target 对象生成一个动态代理对象，这个动态代理对象与 target 实现了相同的接口，所以具有相同的 public 方法——从这个意义上来看，动态代理对象可以当成 target 对象使用。当程序调用动态代理对象的指定方法时，实际上将变为执行 MyInvokationHandler 对象的 invoke 方法。例如调用动态代理对象的 info()方法，程序将开始执行 invoke 方法，其执行步骤如下：

1. 创建 TxUtil 实例。
2. 执行 TxUtil 实例的 beginTx()方法。

③ 使用反射以 target 作为调用者执行 info()方法。

④ 执行 TxUtil 实例的 endTx ()方法。

看到上面的执行过程，读者应该已经发现：当我们使用动态代理对象来代替被代理对象时，代理对象的方法就实现了前面的要求：程序执行 info()、run()方法时增加事务功能。而且这种方式有一个额外的好处：GunDog 的方法中没有以硬编码的方式调用 beginTx()和 endTx()——这就为系统扩展增加了无限可能性：当系统需要扩展 GunDog 实例的功能时，程序只需要提供额外的拦截器类，并在 MyInvokationHandler 的 invoke()方法中回调这些拦截器方法即可。

下面提供一个主程序来测试动态代理的结果。

程序清单：codes\09\9.3\DynaProxy\Test.java

```
public class Test
{
    public static void main(String[] args)
        throws Exception
    {
        //创建一个原始的 GunDog 对象，作为 target
        Dog target = new GunDog();
        //以指定的 target 来创建动态代理
        Dog dog = (Dog)MyProxyFactory.getProxy(target);
        //调用代理对象的 info()和 run()方法
        dog.info();
        dog.run();
    }
}
```

上面程序中的 dog 对象实际上是动态代理对象，只是该动态代理对象也实现了 Dog 接口，所以也可以当成 Dog 对象使用。程序执行 dog 的 info()和 run()方法时，实际上会先执行 TxUtil 的 beginTx()，再执行 target 对象的 info()和 run()方法，最后再执行 TxUtil 的 endTx()。执行上面的程序，看到如图 9.5 所示结果。

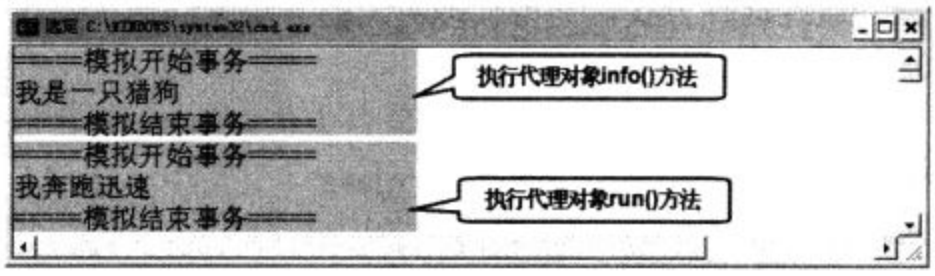

图 9.5　动态代理

通过如图 9.5 所示的运行结果来看，不难发现采用动态代理可以非常灵活地实现解耦。通过使用这种动态代理，程序就为被代理对象增加了额外的功能。

这种动态代理在 AOP（Aspect Orient Program，面向切面编程）里被称为 AOP 代理，AOP 代理可代替目标对象，AOP 代理包含了目标对象的全部方法。但 AOP 代理中的方法与目标对象的方法存在差异：AOP 代理里的方法可以在执行目标方法之前、之后插入一些通用处理。

AOP 代理所包含的方法与目标对象所包含的方法示意图如图 9.6 所示。

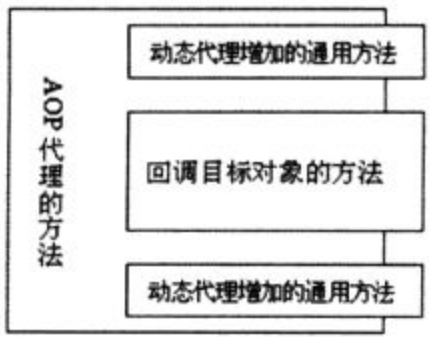

图 9.6　AOP 代理的方法与目标对象的方法

看到此处，相信读者应该对 Spring 的 AOP 框架有点感觉了：当 Spring 容器中的被代理 Bean 实现了一个或多个接口时，Spring 所创建的 AOP 代理就是这种动态代理。Spring AOP 与此示例应用的区别在哪里呢？Spring AOP 更灵活，当 Sping 定义 InvocationHandler 类的 invoke()时，它并没有以硬编码方式决定调用哪些拦截器，而是通过配置文件来决定在 invoke()方法中要调用哪些拦截器，这就实现了更彻底的解耦——当程序需要为目标对象扩展新功能时，根本无须改变 Java 代理，只需要在配置文件中增加更多的拦截器配置即可。

9.3.5 命令模式

考虑这样一种场景：某个方法需要完成某一个功能，完成这个功能的大部分步骤已经确定了，但可能有少量具体步骤无法确定，必须等到执行该方法时才可以确定。具体一点：假设有个方法需要遍历某个数组的数组元素，但无法确定在遍历数组元素时如何处理这些元素，需要在调用该方法时指定具体的处理行为。

这个要求看起来有点奇怪：这个方法不仅要求参数可以变化，甚至要求方法执行体的代码也可以变化，难道我们能把“处理行为”作为一个参数传入该方法？

提示： 在某些编程语言（如 Ruby、Perl 等）里，确实允许传入一个代码块作为参数。但 Java 暂时还不支持代码块作为参数，这可能是现阶段 Java 存在的一个小小的缺陷。

对于这样的需求，我们必须把“处理行为”作为参数传入该方法，而“处理行为”用编程来实现就是一段代码。那如何把这段代码传入某个方法呢？Java 暂时又不支持代码块参数。

因为 Java 不允许代码块单独存在，因此我们必须把该代码块封装成一个方法。在 Java 语言中，类才是一等公民，方法也不能独立存在，所以我们实际传入该方法的应该是一个对象，该对象通常是某个接口的匿名实现类的实例，该接口通常被称为命令接口，这种设计方式也被称为命令模式。

下面的程序先定义一个 ProcessArray 类，该类里包含一个 each()方法用于处理数组，但具体如何处理暂时不能确定，所以 each()方法里定义了一个 Command 参数。

程序清单：codes\09\9.3\Command\ProcessArray.java

```
public class ProcessArray
{
    //定义一个 each()方法，用于处理数组，
    public void each(int[] target , Command cmd)
    {
        cmd.process(target);
    }
}
```

上面我们定义 each()方法时，指定了一个 Command 形参，这个 Command 接口用于定义一个 process()方法，该方法用于封装对数组的“处理行为”。下面是该 Command 接口代码。

程序清单：codes\09\9.3\Command\Command.java

```
public interface Command
{
    //接口里定义的 process 方法用于封装“处理行为”
    void process(int[] target);
}
```

上面的 Command 接口里定义了一个 process 方法，这个方法用于封装“处理行为”，但这个方法没有方法体——因为现在还无法确定这个处理行为。

当主程序调用 ProcessArray 对象的 each()方法来处理数组时，每次处理数组需要传入不同的“处理行为”——也就是要为 each()方法传入不同的 Command 对象，不同的 Command 对象封装了不同的“处理行为”。

下面是主程序调用 ProcessArray 对象 each()方法的程序。

程序清单：codes\09\9.3\Command\CommandTest.java

```
public class CommandTest
{
    public static void main(String[] args)
    {
        ProcessArray pa = new ProcessArray();
        int[] target = {3, -4, 6, 4};
```

```
            //第一次处理数组，具体处理行为取决于 Command 对象
            pa.each(target , new Command()
            {
                //重写 process()方法，决定具体的处理行为
                public void process(int[] target)
                {
                    for (int tmp : target )
                    {
                        System.out.println("迭代输出目标数组的元素:" + tmp);
                    }
                }
            });
            System.out.println("------------------");
            //第二次处理数组，具体处理行为取决于 Command 对象
            pa.each(target , new Command()
            {
                //重写 process 方法，决定具体的处理行为
                public void process(int[] target)
                {
                    int sum = 0;
                    for (int tmp : target )
                    {
                        sum += tmp;
                    }
                    System.out.println("数组元素的总和是:" + sum);
                }
            });
        }
    }
```

正如上面的程序中两段粗体字代码所示，程序两次调用 ProcessArray 对象的 each()方法来处理数组对象，每次调用 each()方法时传入不同的 Command 匿名实现类的实例，不同的 Command 实例封装了不同的“处理行为”。

运行上面程序，看到如图 9.7 所示的结果。

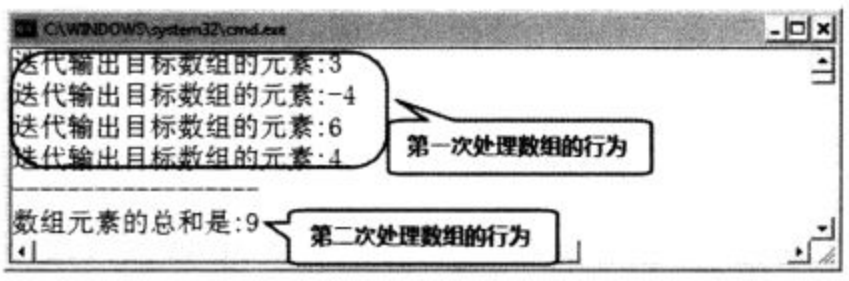

图 9.7　两次处理数组的结果

图 9.7 显示了两次不同处理行为的结果，也就实现了 process 方法和“处理行为”的分离，两次不同的处理行为分别由两个不同的 Command 对象来提供。

理解了这个命令模式后，相信读者对 Spring 框架中 HibernateTemplate 的 executeXxx()方法找到了一点感觉，HibernateTemplate 使用了 executeXxx()方法弥补了 HibernateTemplate 的不足，该方法需要接受一个 HibernateCallback 接口，该接口的代码如下：

```
//定义一个 HibernateCallback 接口，该接口封装持久化处理行为
public interface HibernateCallback
{
    Object doInHibernate(Session session);
}
```

上面的 HibernateCallback 接口就是一个典型的 Command 接口，一个 HibernateCallback 对象封装了自定义的持久化处理。

对 HibernateTemplate 而言，大部分持久化操作都可通过一个方法来实现，HibernateTemplate 对象简化了 Hibernate 的持久化操作，但丢失了使用 Hibernate 持久化操作的灵活性。

通过 HibernateCallback 就可以弥补 HibernateTemplate 灵活性不足的缺点，当调用 HibernateTemplate

的 executeXxx()方法时，传入 HibernateCallback 对象的 doInHibernate()方法就是自定义的持久化处理——即将自定义的持久化处理传入了 executeXxx()方法。如下面的代码片段所示：

```
List list = getHibernateTemplate()
      .executeFind(new HibernateCallback()
      {
            //实现 HibernateCallback 接口必须实现的方法
            public Object doInHibernate(Session session)
                  throws HibernateException, SQLException
            {
                  //执行 Hibernate 分页查询
                  List result = session.createQuery(hql)
                  .setFirstResult(offset)
                  .setMaxResults(pageSize)
                  .list();
                  return result;
            }
      });
return list;
```

上面的程序中粗体字代码块将直接传给 HibernatTemplate，HibernatTemplate 将直接使用该代码块来执行持久化查询，并将查询得到的结果作为 executeFind()方法的返回值。

➤➤9.3.6 策略模式

策略模式用于封装系列的算法，这些算法通常被封装在一个被称为 Context 的类中，客户端程序可以自由选择其中一种算法，或让 Context 为客户端选择一个最佳的算法——使用策略模式的优势是为了支持算法的自由切换。

考虑如下场景：现在我们正在开发一个网上书店，该书店为了更好地促销，经常需要对图书进行打折促销，程序需要考虑各种打折促销的计算方法。

为了实现书店现在所提供的各种打折需求，程序考虑使用如下方式来实现。

```
//一段实现 discount()方法代码
public double discount(double price)
{
      //针对不同情况采用不同的打折算法
      switch(getDiscountType())
      {
            case VIP_DISCOUNT:
                  return vipDiscount(price);
                  break;
            case OLD_DISCOUNT:
                  return oldDiscount(price);
                  break;
            case SALE_DISCOUNT:
                  return saleDiscount(price);
                  break;
            ...
      }
}
```

上面的粗体字代码会根据打折类型来决定使用不同的打折算法，从而满足该书店促销打折的要求。从功能实现的角度来看，这段代码没有太大的问题。但这段代码有一个明显的不足，程序中各种打折方法都被直接写入了 discount(double price)方法中。如有一天，该书店需要新增一种打折类型呢？那开发人员必须修改至少 3 处代码：首先需要增加一个常量，该常量代表新增的打折类型；其次需要在 switch 语句中增加一个 case 语句；最后开发人员需要实现 xxxDiscount()方法，用于实现新增的打折算法。

为了改变这种不好的设计，下面将会选择使用策略模式来实现该功能。下面先提供一个打折算法的接口，该接口里包含一个 getDiscount ()方法，该接口代码如下。

程序清单：codes\09\9.3\Strategy\DiscountStrategy.java

```
public interface DiscountStrategy
{
    //定义一个用于计算打折价的方法
    double getDiscount(double originPrice);
}
```

下面为该打折接口提供两个策略类，它们分别实现了不同的打折算法。

程序清单：codes\09\9.3\Strategy\VipDiscount.java

```
//实现 DiscountStrategy 接口，实现对 VIP 打折的算法
public class VipDiscount
    implements DiscountStrategy
{
    //重写 getDiscount()方法，提供 VIP 打折算法
    public double getDiscount(double originPrice)
    {
        System.out.println("使用 VIP 折扣...");
        return originPrice * 0.5;
    }
}
```

程序清单：codes\09\9.3\Strategy\OldDiscount.java

```
public class OldDiscount
    implements DiscountStrategy
{
    //重写 getDiscount()方法，提供旧书打折算法
    public double getDiscount(double originPrice)
    {
        System.out.println("使用旧书折扣...");
        return originPrice * 0.7;
    }
}
```

提供了如上两个折扣策略类之后，程序还应该提供一个 DiscountContext 类，该类用于为客户端代码选择合适折扣策略，当然也允许用户自由选择折扣策略。下面是该 DiscountContext 类的代码。

程序清单：codes\09\9.3\Strategy\DiscountContext.java

```
public class DiscountContext
{
    //组合一个 DiscountStrategy 对象
    private DiscountStrategy strategy;
    //构造器，传入一个 DiscountStrategy 对象
    public DiscountContext(DiscountStrategy strategy)
    {
        this.strategy  = strategy;
    }
    //根据实际所使用的 DiscountStrategy 对象得到折扣价
    public double getDiscountPrice(double price)
    {
        //如果 strategy 为 null，系统自动选择 OldDiscount 类
        if (strategy == null)
        {
            strategy = new OldDiscount();
        }
        return this.strategy.getDiscount(price);
    }
    //提供切换算法的方法
    public void changeDiscount(DiscountStrategy strategy)
    {
        this.strategy = strategy;
    }
}
```

从上面的程序的粗体字代码可以看出，该 Context 类扮演了决策者的角色，它决定调用哪个折扣

策略来处理图书打折。当客户端代码没有选择合适的折扣时，该 Context 会自动选择 OldDiscount 折扣策略；用户也可根据需要选择合适的折扣策略。

下面的程序示范了使用该 Context 类来处理图书打折的任何情况。

程序清单：codes\09\9.3\Strategy\StrategyTest.java

```
public class StrategyTest
{
    public static void main(String[] args)
    {
        //客户端没有选择打折策略类
        DiscountContext dc = new DiscountContext(null);
        double price1 = 79;
        //使用默认的打折策略
        System.out.println("79 元的书默认打折后的价格是："
            + dc.getDiscountPrice(price1));
        //客户端选择合适的 VIP 打折策略
        dc.changeDiscount(new VipDiscount());
        double price2 = 89;
        //使用 VIP 打折得到打折价格
        System.out.println("89 元的书对 VIP 用户的价格是："
            + dc.getDiscountPrice(price2));
    }
}
```

上面的程序第一行粗体字代码创建了一个 DiscountContext 对象，客户端并未指定实际所需的打折策略类，故程序将使用默认的打折策略类；程序第二行粗体字代码指定使用 VipDiscount 策略类，故程序将改为使用 VIP 打折策略。

再次考虑前面的需求：当业务需要新增一种打折类型时，系统只需要新定义一个 DiscountStrategy 实现类，该实现类实现 getDiscount()方法，用于实现新的打折算法即可。客户端程序需要切换为新的打折策略时，则需要先调用 DiscountContext 的 changeDiscount()方法切换为新的打折策略。

从上面的介绍中可以看出，使用策略模式可以让客户端代码在不同的打折策略之间切换，但也有一个小小的遗憾：客户端代码需要和不同的策略类耦合。为了弥补这个不足，我们可以考虑使用配置文件来指定 DiscountContext 使用哪种打折策略——这就彻底分离客户端代码和具体打折策略类。

介绍到这里，相信读者对 Hibernate 的 Dialect 会有一点感觉了，这个 Dialect 类代表各数据库方言的抽象父类，但不同数据库的持久化访问可能存在一些差别，尤其在分页算法上存在较大的差异，Dialect 不同子类就代表了一种特定的数据库访问策略。为了让客户端代码与具体的数据库、具体的 Dialect 实现类分离，Hibernate 需要在 hibernate.cfg.xml 文件中指定应用所使用的 Dialect 子类。

与此类似的是，Spring 的 Resource 接口也是一个典型的策略接口，不同的实现类代表了不同的资源访问策略。当然 Spring 可以非常“智能”地选择合适的 Resource 实现类，通常来说，Spring 可以根据前缀来决定使用合适的 Resource 实现类；还可根据 ApplicationContext 的实现类来决定使用合适的 Resource 实现类。具体请参考本书 8.3 节的介绍。

➤➤9.3.7 门面模式

随着系统的不断改进和开发，它们会变得越来越复杂，系统会生成大量的类，这使得程序流程更难被理解。门面模式可为这些类提供一个简化的接口，从而简化访问这些类的复杂性，有时这种简化可能降低访问这些底层类的灵活性，但除了要求特别苛刻的客户端之外，它通常都可以提供所需的全部功能，当然，那些苛刻的用户仍然可以直接访问底层的类和方法。

门面模式（Facade）也被称为正面模式、外观模式，这种模式用于将一组复杂的类包装到一个简单的外部接口中。

现在考虑这样的场景：我们有一个顾客需要到饭店用餐，这就需要定义一个 Customer 类，并为该

类定义一个 haveDinner()方法。考虑该饭店有三个部门：收银部、厨师部和服务生部，用户就餐需要这三个部门协调才能完成。

本示例程序先定义一个收银部，用户需要调用该部门的 pay()方法来支付用餐费。

程序清单：codes\09\9.3\Facade\PaymentImpl.java

```
public class PaymentImpl
      implements Payment
{
      //模拟顾客支付费用的方法
      public String pay()
      {
            String food = "快餐";
            System.out.println("你已经向收银员支付了费用，您购买的食物是："
                  + food);
            return food;
      }
}
```

程序接下来要定义一个厨师部门，用户需要调用该部门的 cook ()方法来烹调食物。

程序清单：codes\09\9.3\Facade\CookImpl.java

```
public class CookImpl
      implements Cook
{
      //模拟烹调食物的方法
      public String cook(String food)
      {
            System.out.println("厨师正在烹调:" + food);
            return food;
      }
}
```

程序还要定义一个服务生部门，用户需要调用该部门的 serve ()方法来得到食物。

程序清单：codes\09\9.3\Facade\WaiterImpl.java

```
public class WaiterImpl
      implements Waiter
{
      //模拟服务员上菜的方法
      public void serve(String food)
      {
            System.out.println("服务器员已将" + food
                  + "端过来了，请慢用...");
      }
}
```

接下来实现 Customer 类的 haveDinner()方法时，系统将有如下代码实现。

程序清单：codes\09\9.3\Facade\Customer.java

```
public class Customer
{
      public void haveDinner()
      {
            //依次创建三个部门实例
            Payment pay = new PaymentImpl();
            Cook cook = new CookImpl();
            Waiter waiter = new WaiterImpl();
            //依次调用三个部门实例的方法来实现用餐功能
            String food = pay.pay();
            food = cook.cook(food);
            waiter.serve(food);
            //直接依赖于 Facade 类来实现用餐方法
            Facade f = new Facade();
            f.serveFood();
      }
}
```

正如上面的粗体字代码所示，Customer 需要依次调用三个部门的方法才可实现这个 havaDinner()方法。实际上，如果这个饭店有更多的部门，那么程序就需要调用更多部门的方法来实现这个 haveDinner()方法——这就会增加 haveDinner()方法的实现难度了。

为了解决这个问题，我们可以为 Payment、Cook、Waiter 三个部门提供一个门面（Facade），使用该 Facade 来包装这些类，对外提供一个简单的访问方法。下面是该 Facade 类的代码。

程序清单：codes\09\9.3\Facade\Facade.java

```
public class Facade
{
    //定义被 Facade 封装的三个部门
    Payment pay;
    Cook cook;
    Waiter waiter;
    //构造器
    public Facade()
    {
        this.pay = new PaymentImpl();
        this.cook = new CookImpl();
        this.waiter = new WaiterImpl();
    }
    public void serveFood()
    {
        //依次调用三个部门的方法，封装成一个 serveFood()方法
        String food = pay.pay();
        food = cook.cook(food);
        waiter.serve(food);
    }
}
```

从 Facade 代码可以看出，该门面类保证了 Payment、Cook、Waiter 三个部门，程序的粗体字代码对外提供了一个简单的 serveFood()方法，该方法对外提供了一个用餐的方法，而底层则依赖于三个部门的 pay()、cook()、serve()三个方法。

一旦程序提供了这个门面类 Facade 之后，Cutsomer 类实现 haveDinner()方法就变得更加简单了。下面是通过 Facade 类实现 haveDinner()方法的代码。

```
public void haveDinner()
{
    //直接依赖于 Facade 类来实现用餐方法
    Facade f = new Facade();
    f.serveFood();
}
```

从上面的程序可以看出，如果不采用门面模式，客户端需要自行决定需要调用哪些类、哪些方法，并需要按合理的顺序来调用它们才可实现所需的功能。不采用门面模式时，程序有如图 9.8 所示的结构。

从图 9.8 中可以看出，两个客户端需要和底层各对象形成错综复杂的网络调用，无疑增加了客户端编程的复杂度。使用正面模式后的程序结构如图 9.9 所示。

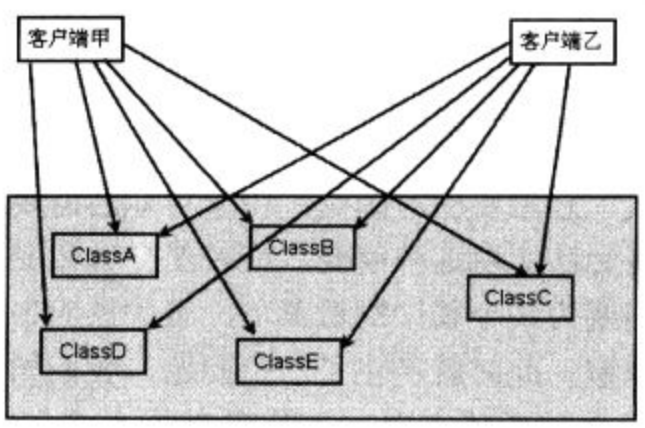

图 9.8　未使用门面模式的结构图

从图 9.9 可以看出，当程序使用了门面模式之后，客户端代码只需要和门面类进行交互，客户端代码变得极为简单。

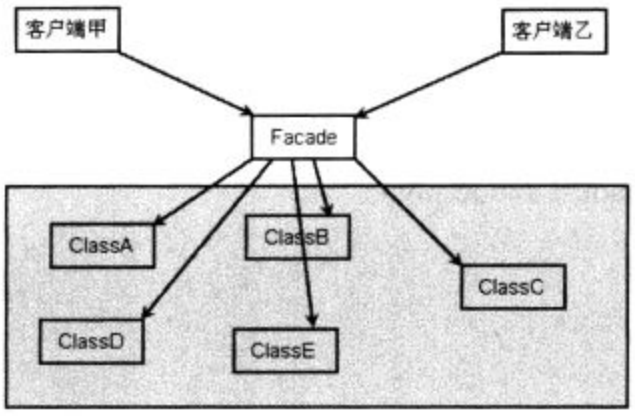

图 9.9　使用门面模式的程序结构图

阅读到此处相信读者对 Spring 的 HibernateTemplate 类有点感觉了，当我们的程序使用 HibernateTemplate 的 find()方法时，程序只要此一行代码即可得到查询返回的 List。但实际上该 find()方法后隐藏了如下代码：

```
Session session = sf.openSession();
Query query = session.createQuery(hql);
for(int i = 0 ; i < args.length ; i++)
{
     query.setParameter(i ,args[i]);
}
query.list();
```

因此，我们可以认为 HibernateTemplate 是 SessionFactory、Session、Query 等类的门面，当客户端程序需要进行持久化查询时，程序无须调用这些类，而是直接调用 HibernateTemplate 门面类的方法即可。

除此之外，Java EE 应用里使用业务逻辑组件来封装 DAO 组件也是典型的门面模式——每个业务逻辑组件都是众多 DAO 组件的门面，系统的控制器类无须直接访问 DAO 组件，而是由业务逻辑方法来组合多个 DAO 方法以完成所需功能，而 Action 只需与业务逻辑组件交互即可。在这种设计方式下，Java EE 应用的各组件有如图 9.10 所示的结构图。

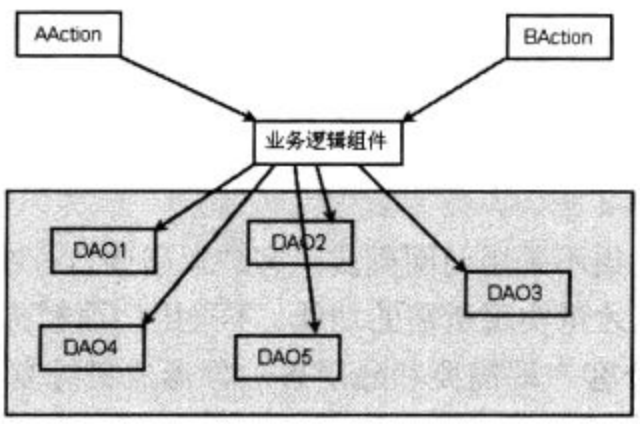

图 9.10　Java EE 应用结构图

➤➤9.3.8　桥接模式

桥接模式是一种结构型模式，它主要应对的是：由于实际的需要，某个类具有两个或两个以上的维度变化，如果只是使用继承将无法实现这种需要，或者使得设计变得相当臃肿。

举例来说，假设现在我们需要为某个餐厅制造菜单，餐厅供应牛肉面、猪肉面……，而且顾客可根据自己的口味选择是否添加辣椒。此时就产生了一个问题，我们如何来应对这种变化：我们是否需要定义辣椒牛肉面、无辣牛肉面、辣椒猪肉面、无辣猪肉面 4 个子类？如果餐厅还供应羊肉面、韭菜面……呢？如果添加辣椒时可选择无辣、微辣、中辣、重辣……风味呢？那程序岂非一直忙于定义

子类？

为了解决这个问题，我们可以使用桥接模式，桥接模式的做法是把变化部分抽象出来，使变化部分与主类分离开来，从而将多个维度的变化彻底分离。最后提供一个管理类来组合不同维度上的变化，通过这种组合来满足业务的需要。

下面以一个简单的示例来示范桥接模式的使用。程序首先提供了一个 Peppery 接口，该接口代表了面条是否添加辣椒。

程序清单：codes\09\9.3\Bridge\Peppery.java

```
public interface Peppery
{
    String style();
}
```

接着程序为该接口提供两个实现类，第一个实现类代表辣椒的风格。

程序清单：codes\09\9.3\Bridge\PepperySytle.java

```
public class PepperySytle implements Peppery
{
    //实现"辣味"风格的方法
    public String style()
    {
        return "辣味很重，很过瘾...";
    }
}
```

下面一个实现类代表不添加辣椒的风格。

程序清单：codes\09\9.3\Bridge\PlainStyle.java

```
public class PlainStyle implements Peppery
{
    //实现"不辣"风格的方法
    public String style()
    {
        return "味道清淡，很养胃...";
    }
}
```

从上面的程序可以看出，该 Peppery 接口代表了面条在辣味风格这个维度上的变化，不论面条在该维度上有多少种变化，程序只需要为这几种变化分别提供实现类即可。对于系统而言，辣味风格这个维度上的变化是固定的，程序必须面对的，程序使用桥接模式将辣味风格这个维度的变化分离出来了，避免与牛肉、猪肉材料风格这个维度的变化耦合在一起。

接着程序提供了一个 AbstractNoodle 抽象类，该抽象类将会持有一个 Peppery 属性，该属性代表该面条的辣味风格。程序通过 AbstractNoodle 组合一个 Peppery 对象，从而运行了面条在辣味风格这个维度上的变化；而 AbstractNoodle 本身可以包含很多实现类，不同实现类则代表了面条在材料风格这个维度上的变化。下面是 AbstractNoodle 类的代码。

程序清单：codes\09\9.3\Bridge\AbstractNoodle.java

```
public abstract class AbstractNoodle
{
    //组合一个 Peppery 变量，用于将该维度的变化独立出来
    protected Peppery style;
    //每份 Noodle 必须组合一个 Peppery 对象
    public AbstractNoodle(Peppery style)
    {
        this.style = style;
    }
    public abstract void eat();
}
```

正如上面的程序中粗体字代码所示，上面的 AbstractNoodle 实例将会与一个 Peppery 实例组合，

不同的 AbstractNoodle 实例与不同的 Peppery 实例组合，就可完成辣味风格、材料风格两个维度上变化的组合了。

由此可见，AbstractNoodle 抽象类可以看做是一个桥梁，它被用来“桥接”面条的材料风格的改变与辣味风格的改变，使面条的特殊属性得到无绑定的扩充。

接下来为 AbstractNoodle 提供一个 PorkyNoodle 子类，该子类代表猪肉面。

程序清单：codes\09\9.3\Bridge\PorkyNoodle.java

```
public class PorkyNoodle extends AbstractNoodle
{
    public PorkyNoodle(Peppery style)
    {
        super(style);
    }
    //实现 eat()抽象方法
    public void eat()
    {
        System.out.println("这是一碗稍嫌油腻的猪肉面条。"
            + super.style.style());
    }
}
```

再提供一个 BeefMoodle 子类，该子类代表牛肉面。

程序清单：codes\09\9.3\Bridge\BeefNoodle.java

```
public class BeefNoodle extends AbstractNoodle
{
    public BeefNoodle(Peppery style)
    {
        super(style);
    }
    //实现 eat()抽象方法
    public void eat()
    {
        System.out.println("这是一碗美味的牛肉面条。"
            + super.style.style());
    }
}
```

从 PorkyNoodle.java 和 BeefMoodle.java 中可以看出：AbstractNoodle 的两个具体类实现 eat()方法时，既组合了材料风格的变化，也组合了辣味风格的变化，从而可表现出两个维度上的变化。桥接模式下这些接口和类之间的结构关系如图 9.11 所示。

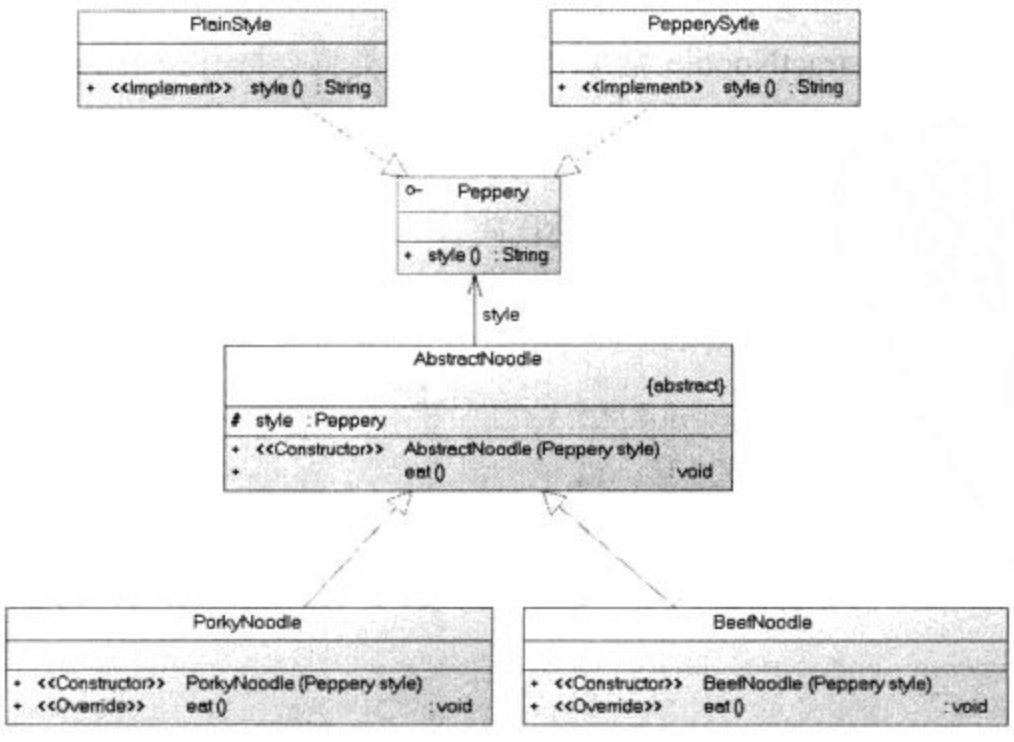

图 9.11　桥接模式的类图

下面提供一个主程序，可以分别产生辣椒牛肉面、无辣牛肉面、辣椒猪肉面、无辣猪肉面 4 种风格的面条。

程序清单：codes\09\9.3\Bridge\Test.java

```
public class Test
{
    public static void main(String[] args)
    {
        //下面将得到“辣味”的牛肉面
        AbstractNoodle noodle1 = new BeefNoodle(
            new PepperySytle());
        noodle1.eat();
        //下面将得到“不辣”的牛肉面
        AbstractNoodle noodle2 = new BeefNoodle(
            new PlainStyle());
        noodle2.eat();
        //下面将得到“辣味”的猪肉面
        AbstractNoodle noodle3 = new PorkyNoodle(
            new PepperySytle());
        noodle3.eat();
        //下面将得到“不辣”的猪肉面
        AbstractNoodle noodle4 = new PorkyNoodle(
            new PlainStyle());
        noodle4.eat();
    }
}
```

上面程序的 main 方法中得到了 4 种面条，这 4 种面条就满足了面条在两个维度上的变化，但程序结构又比较简洁。

桥接模式在 Java EE 架构中有非常广泛的用途，由于 Java EE 应用需要实现跨数据库的功能，程序为了在不同数据库之间迁移，因此系统需要在持久化技术这个维度上存在改变；除此之外，系统也需要在不同业务逻辑实现之间迁移，因此也需要在逻辑实现这个维度上存在改变，这正好符合桥接模式的使用场景。因此，Java EE 应用都会推荐使用业务逻辑组件和 DAO 组件分离的结构，让 DAO 组件负责持久化技术这个维度上的改变，让业务逻辑组件负责业务逻辑实现这个维度上的改变。由此可见，Java EE 应用中常见的 DAO 模式正是桥接模式的应用。

注意：

可能有读者会感到奇怪，刚才我们还提到用业务逻辑组件来包装 DAO 组件是门面模式，怎么现在又说这种方式是桥接模式呢？其实这两种说法都没有问题，称这种方式为门面模式，是从每个业务逻辑组件底层包装了多个 DAO 组件这个角度来看的，从这个角度来看，业务逻辑组件就是 DAO 组件的门面；如果从 DAO 组件的设计初衷来看，设计 DAO 组件是为了让应用在不同持久化技术之间自由切换，也就是分离系统在持久化技术这个维度上的变化，从这个角度来看，Java EE 应用中分离出 DAO 组件本身就是遵循桥接模式的。

不要以为每段代码、每个应用只能使用一种设计模式！实际上，一个设计优良的项目，本身就是设计模式最好的教科书，例如 Spring 框架，当你深入阅读其源代码时，你会发现这个框架处处充满了设计模式的应用场景。

笔者在此处介绍设计模式，并不是为了让读者满足于设计模式的实现方式，而是希望读者能掌握设计模式的应用场景，能使用设计模式来解决实际开发问题。真正掌握设计模式的要求是：当进行系统开发时，无须刻意思索需要运用哪种设计模式；而是信手写来，而整个应用的设计充满灵性，大量设计模式应用其中。

9.3.9 观察者模式

观察者模式定义了对象间的一对多依赖关系，让一个或多个观察者对象观察一个主题对象。当主题对象的状态发生变化时，系统能通知所有的依赖于此对象的观察者对象，从而使得观察者对象能够自动更新。

在观察者模式中，被观察的对象常常也被称为目标或主题（Subject），依赖的对象被称为观察者（Observer）。

下面以一个简单的示例来示范观察者模式，程序先提供一个观察者接口。

程序清单：codes\09\9.3\Observer\Observer.java

```
public interface Observer
{
    void update(Observable o , Object arg);
}
```

上面 Observer 接口是一个观察者接口，程序中所有观察者都应该实现该接口。在该接口的 update()方法中包含了一个 Observable 类型的参数，该参数代表被观察对象，也就是前面介绍的目标或主题。此处的 Observable 是一个抽象基类，程序中被观察者应该继承该抽象基类。Observable 类的代码如下。

程序清单：codes\09\9.3\Observer\Observable.java

```
public abstract class Observable
{
    // 用一个 List 来保存该对象上所有绑定的事件监听器
    List<Observer> observers =
        new ArrayList<Observer>();
    //定义一个方法，用于从该主题上注册观察者
    public void registObserver(Observer o)
    {
        observers.add(o);
    }
    //定义一个方法，用于从该主题中删除观察者
    public void removeObserver(Observer o)
    {
        observers.remove(o);
    }
    //通知该主题上注册的所有观察者
    public void notifyObservers(Object value)
    {
        //遍历注册到该被观察者上的所有观察者
        for (Iterator it = observers.iterator();
            it.hasNext(); )
        {
            Observer o = (Observer)it.next();
            //显式每个观察者的 update 方法
            o.update(this , value);
        }
    }
}
```

该 Observable 抽象类是所有被观察者的基类，它主要提供了 registObserver()方法用于注册一个新的观察者；并提供了一个 removeObserver()方法用于删除一个已注册的观察者；当具体被观察对象的状态发生改变时，具体被观察对象会调用 notifyObservers 方法来通知所有观察者。

下面提供一个具体的被观察者类：Product，该产品有两个属性，它继承了 Observable 抽象基类。

程序清单：codes\09\9.3\Observer\Product.java

```
public class Product extends Observable
{
    //定义两个属性
    private String name;
    private double price;
```

```
        //无参数的构造器
        public Product(){}
        public Product(String name , double price)
        {
              this.name = name;
              this.price = price;
        }
        public String getName()
        {
              return name;
        }
        //当程序调用 name 的 setter 方法来修改 Product 的 name 属性时
        //程序自然触发该对象上注册的所有观察者
        public void setName(String name)
        {
              this.name = name;
              notifyObservers(name);
        }
        public double getPrice()
        {
              return price;
        }
        //当程序调用 price 的 setter 方法来修改 Product 的 price 属性时
        //程序自然触发该对象上注册的所有观察者
        public void setPrice(double price)
        {
              this.price = price;
              notifyObservers(price);
        }
}
```

正如程序中两行粗体字代码所示，当程序调用 Product 对象的 setName()、setPrice()方法来改变 Product 的 name、price 属性时，这两个方法将自动触发 Observable 基类的 notifyObservers 方法。

接下来程序提供两个观察者，一个用于观察 Product 对象的 name 属性，另一个用于观察 Product 对象的 price 属性。

程序清单：codes\09\9.3\Observer\NameObserver.java

```
public class NameObserver implements Observer
{
      //实现观察者必须实现的 update 方法
      public void update(Observable o , Object arg)
      {
            if (arg instanceof String )
            {
                  //产品名称改变值在 name 中
                  String name = (String)arg;
                  //启动一个 JFrame 窗口来显示被观察对象的状态改变
                  JFrame f = new JFrame("观察者");
                  JLabel l = new JLabel("名称改变为: " + name);
                  f.add(l);
                  f.pack();
                  f.setVisible(true);
                  System.out.println("名称观察者:" +
                        o + "物品名称已经改变为: " + name);
            }
      }
}
```

程序清单：codes\09\9.3\Observer\PriceObserver.java

```
public class PriceObserver implements Observer
{
      //实现观察者必须实现的 update 方法
      public void update(Observable o , Object arg)
      {
            if(arg instanceof Double)
```

```
            {
                System.out.println("价格观察者:" +
                    o + "物品价格已经改变为: " + arg);
            }
        }
    }
```

接着主程序创建一个 Product 对象（被观察的目标对象），然后向该被观察对象上注册两个观察者对象，当主程序调用 Product 对象的 setter 方法来改变该对象的状态时，注册在 Product 对象上的两个观察者将被触发。主程序代码如下。

程序清单：codes\09\9.3\Observer\Test.java

```
public class Test
{
    public static void main(String[] args)
    {
        //创建一个被观察者对象
        Product p = new Product("电视机" , 176);
        //创建两个观察者对象
        NameObserver no = new NameObserver();
        PriceObserver po = new PriceObserver();
        //向被观察对象上注册两个观察者对象
        p.registObserver(no);
        p.registObserver(po);
        //程序调用 setter 方法来改变 Product 的 name 和 price 属性
        p.setName("书桌");
        p.setPrice(345f);
    }
}
```

运行上面的程序，我们将可看到当 Product 的属性值发生改变时，注册在该 Product 上的 NameObserver 和 PriceObserver 将被触发。

纵观上面介绍的观察者模式，我们发现观察者模式通常包含如下 4 个角色。

- 被观察者的抽象基类：它通常会持有多个观察者对象的引用。Java 提供了 java.util.Observable 基类来代表被观察者的抽象基类，所以实际开发中无须自己开发这个角色。
- 观察者接口：该接口是所有被观察对象应该实现的接口，通常它只包含一个抽象方法 update()。Java 同样提供了 java.util.Observer 接口来代表观察者接口，实际开发中也无须开发该角色。
- 被观察者实现类：该类继承 Observable 基类。
- 观察者实现类：实现 Observer 接口，实现 update 抽象方法。

理解了上面观察者模式的实现思路之后，可能有读者会感到疑惑：观察者模式的实现方式与 Java 事件机制的底层实现何其相似啊？实际上，我们完全可以把观察者接口理解成事件监听接口，而被观察者对象也可当成事件源来处理——换个角度来思考：监听，观察，这两个词语之间有本质的区别吗？Java 事件机制的底层实现，本身就是通过观察者模式来实现的。

除此之外，观察者模式在 Java EE 应用中也有广泛应用，主题/订阅模式下的 JMS（Java Message Service，Java 消息服务）本身就是观察者模式的应用。图 9.12 显示了主题/订阅模式下 JMS 的示意图。

从图 9.12 中可以看出，当 Topic 主题收到发布者（Publisher）发布的消息时，注册到该主题的所有订阅者（Subscriber）都可收到该消息。实际上，Java EE 把这个 Topic 设计成一个被观察者，而所有订阅者都注册到该被观察者，当发布者发布消息时，该消息将会引起 Topic 主题的改变，这种改变将会触发注册到该 Topic 上的所有观察者。

提示： 如果读者希望了解有关 JMS 的相关内容以及编程的详细步骤，可以参考本书的姊妹篇《经典 Java EE 企业应用实战》，该书专门有一章用于介绍 JMS。

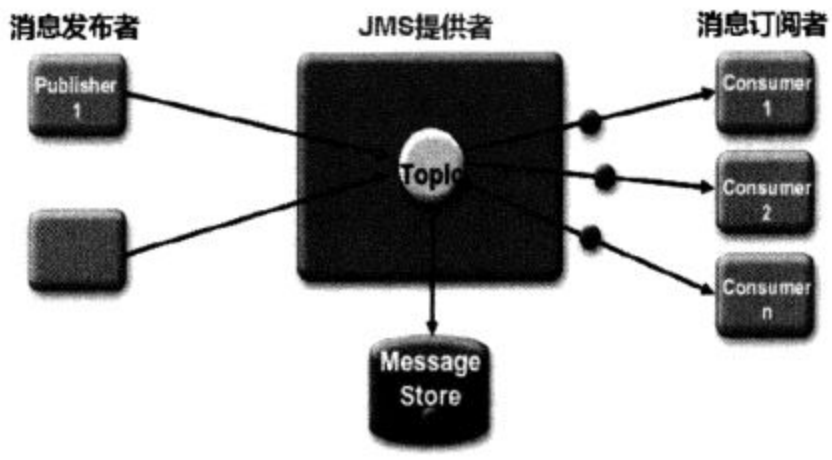

图 9.12　主题/订阅模式下 JMS 的示意图

9.4　常见的架构设计策略

目前流行的轻量级 Java EE 应用的架构基本比较统一，通常会使用 Spring 作为核心，向上整合 MVC 框架，向下整合 ORM 框架。使用 Spring 的 IoC 容器来管理各组件之间的依赖关系时，Spring 的声明事务将负责业务逻辑层组件的事务管理。

虽然大体的结构设计保持相似，但在具体的技术组合上可能存在较小的变化，当我们决定采用某种架构设计时，我们主要考虑这种架构是否成功地将规范和实现分离了，从而可以提供较好的可扩展性、可修改性。最理想的情况是，当我们修改企业级应用的某个组件时，应用中其他组件受到影响很小，或者不受任何影响——这就为整个系统后期的扩展提供了无限可能性。

➤➤9.4.1　贫血模型

贫血模型是最常用的应用架构，也是最容易理解的架构。为了让读者通过本书顺利进入轻量级 Java EE 企业应用开发，本书后面的范例都将采用这种简单的架构设计。

所谓贫血，指 Domain Object 只是单纯的数据类，不包含业务逻辑方法，即每个 Domain Object 类只包含相关属性，并为每个属性提供基本的 setter 和 getter 方法。所有的业务逻辑都由业务逻辑组件实现，这种 Domain Object 就是所谓的贫血的 Domain Object，采用这种 Domain Object 的架构即所谓的贫血模型。

下面以一个简单的消息发布系统的代码为例来介绍贫血模型。

在贫血模型里，所有的 Domain Object 只是单纯的数据类，只包含每个属性的 setter 和 getter 方法，如下是两个持久化类。

第一个 Domain Object 是消息，其代码如下：

```
public class News
{
    //主键
    private Long id;
    //消息标题
    private String title;
    //消息内容
    private String content;
    //消息的发布时间
    private Date postDate;
    //消息的最后修改时间
    private Date lastModifyDate;
    //消息所属分类
    private Category category;
    //消息发布者
    private User poster;
    //消息对应的回复
```

```
    private Set newsReviews;
    //下面省略各属性的 setter 和 getter 方法
    ...
    //重写 Domain Object 的 equals 方法
    public boolean equals(Object object)
    {
        if(this == object)
        {
            return true;
        }
        if (object != null &&
            object.getClass() == News.class)
        {
            News rhs = (News) object;
            return this.poster.equals(rhs.getPoster())
                && this.postDate.equals(rhs.getPostDate());
        }
        return false;
    }
    //重写 News 类的 hashCode 方法
    public int hashCode()
    {
        return this.poster.hashCode() +
            this.postDate.hashCode() * 29;
    }
    //重写 News 类的 toString 方法
    public String toString()
    {
        return new ToStringBuilder(this).append("id", this.id)
            .append("title" , this.title)
            .append("postDate" , this.postDate)
            .append("content" , this.content)
            .append("lastModifyDate" , this.lastModifyDate)
            .append("poster" , this.poster)
            .append("category" , this.category)
            .append("newsReviews" , this.newsReviews)
            .toString();
    }
}
```

第二个 Domain Object 是消息对应的回复，其代码如下：

```
public class NewsReview
{
    //消息回复的主键
    private Long id;
    //消息回复的内容
    private String content;
    //消息回复的回复时间
    private Date postDate;
    //回复的最后修改时间
    private Date lastModifyDate;
    //消息回复对应的消息
    private News news;
    //此处省略了各属性的 setter 和 getter 方法
    ...
    //重写 NewsReview 的 equals 方法
    public boolean equals(Object object)
    {
        if(this == object)
        {
            return true;
        }
        if (object != null &&
            object.getClass() == NewsReview.class)
        {
            NewsReview rhs = (NewsReview) object;
            return this.poster.equals(rhs.getPoster())
```

```
                && this.postDate.equals(rhs.getPostDate());
        }
        return false;
    }
    //重写 NewsReview 的 hashCode 方法
    public int hashCode()
    {
        return this.poster.hashCode()
            + this.postDate.hashCode() * 29;
    }
    //重写 NewsReview 的 toString 方法
    public String toString()
    {
        return new ToStringBuilder(this)
            .append("id" , this.id)
            .append("postDate" , this.postDate)
            .append("lastModifyDate" , this.lastModifyDate)
            .append("content" , this.content)
            .append("poster" , this.poster)
            .append("news" , this.news).toString();
    }
}
```

从上面贫血模型的 Domain Object 可以看出，其类代码中只为每个属性提供 setter 和 getter 方法，这种 Domain Object 只是单纯的数据体，类似于 C 语言的数据结构。虽然它的名字是 Domain Object，却没有包含任何业务对象的相关方法。Martin Fowler 认为，这是一种不健康的建模方式，Domain Model 既然代表了业务对象，就应该包含相关的业务方法。从语义的角度上来看，Domain Model 在这里被映射为持久化实体（一般都是针对 ORM 框架），这种 Domain Object 应该是数据与动作的集合，贫血模型相当于抛弃了 Java 面向对象的性质。

Rod Johnson 和 Martin Fowler 一致认为：贫血的 Domain Object 实际上以数据结构代替了对象。他们认为 Domain Object 应该是个完整的 Java 对象，既包含基本的数据，也包含操作数据相应的业务逻辑方法。

下面是 NewsDAOHibernate 的源代码，该 DAO 对象用于操作 News 对象。

```
//NewsDaoHibernate 继承 HibernateDaoSupport，实现 NewsDao 接口
public class NewsDaoHibernate
    extends HibernateDaoSupport implements NewsDao
{
    //根据主键加载消息
    public News getNews(Long id)
    {
        return (News)getHibernateTemplate()
            .get(News.class, id);
    }
    //保存新的消息
    public void saveNews(News news)
    {
        getHibernateTemplate().saveOrUpdate(news);
    }
    //根据主键删除消息
    public void removeNews(Long id)
    {
        getHibernateTemplate().delete(getNews(id));
    }
    //查找全部的消息
    public List findAll()
    {
        getHibernateTemplate().find("from News");
    }
}
```

既然 DAO 对象完成具体的持久化操作，因此基本的 CRUD 操作都应该在 DAO 对象中实现。但

DAO 对象应该包含多少个查询方法，并不是确定的。因此，根据业务逻辑的不同需要，不同的 DAO 对象可能有数量不等的查询方法。

对于现实中 News，应该包含一个业务方法（addNewsReviews 方法）。在贫血模型下，News 类的代码并没有包含该业务方法，只是将该业务方法放到业务逻辑对象中实现。下面是业务逻辑对象实现 addNewsReviews 的代码。

```
public class FacadeManagerImpl
      implements FacadeManager
{
      //业务逻辑对象依赖的 DAO 对象
      private CategoryDao categoryDao;
      private NewsDao newsDao;
      private NewsReviewDao newsReviewDao;
      private UserDao userDao;

      //此处还应该增加依赖注入 DAO 对象必需的 setter 方法
      ...
      //此处还应该增加其他业务逻辑方法
      ...
      //下面是增加新闻回复的业务方法
      public NewsReview addNewsReview(Long newsId , String content)
      {
            //根据新闻 id 加载新闻
            News news = newsDao.getNews(newsId);
            //以默认构造器创建新闻回复
            NewsReview review = new NewsReview();
            //设置新闻与新闻回复之间的关联
            review.setNews(news);
            //设置新闻回复的内容
            review.setContent(content);
            //设置回复的回复时间
            review.setPostDate(new Date());
            //设置新闻回复的最后修改时间
            review.setLastModifyDate(new Date());
            //保存回复
            newsReviewDao.saveNewsReview(review);
            return review;
      }
}
```

在贫血模型下，业务逻辑对象作为 DAO 组件的门面，封装了全部的业务逻辑方法，Web 层仅与业务逻辑组件交互即可，无须访问底层的 DAO 对象。Spring 的声明式事务管理将负责业务逻辑对象方法的事务性。

贫血模型的分层非常清晰。Domain Object 并不具备领域对象的业务逻辑功能，仅仅是 ORM 框架持久化所需的持久化实体类，仅是数据载体。贫血模型容易理解，开发便捷，但背离了面向对象的设计思想，所有的 Domain Object 并不是完整的 Java 对象。

总结起来，贫血模型存在如下缺点：

- 项目需要书写大量的贫血类，当然也可以借助某些工具自动生成。
- Domain Object 的业务逻辑得不到体现。由于业务逻辑对象的复杂度大大增加，许多不应该由业务逻辑对象实现的业务逻辑方法，完全由业务逻辑对象实现，从而使业务逻辑对象的实现类变得相当庞大。

贫血模型的优点是：开发简单、分层清晰、架构明晰且不易混淆；所有的依赖都是单向依赖，解耦优秀。适合于初学者及对架构把握不十分清晰的开发团队。

9.4.2 领域对象模型

根据更完整的面向对象规则，每个 Java 类都应该提供其相关的业务方法，如果我们在系统中设计

更完备的 Domain Object 对象，则 Domain Object 不再是单纯的数据载体，Domain Object 包含了相关的业务逻辑方法。例如，News 类包含了 addNewsView 方法等。

下面是修改后的 News 类的源代码。

```
public class News
{
    //此处省略了其他的属性
    //此处省略了属性对应的 setter 和 getter 方法
    ...
    //增加新闻回复的业务逻辑方法
    public NewsReview addNewsReview(String content)
    {
        //以默认构造器创建新闻回复实例
        NewsReview review = new NewsReview();
        //设置回复内容
        review.setContent(content);
        //设置回复的发布日期
        review.setPostDate(new Date());
        //设置回复的最后修改日期
        review.setLastModifyDate(new Date());
        //设置回复与消息的关联
        review.setNews(this);
        return review;
    }
    //此处省略了重写的 hashCode，equals 等方法
    ...
}
```

在上面的 Domain Object 中，包含了相应的业务逻辑方法，这是一种更完备的建模方法。

注意：

不要在 Domain Object 中对消息回复完成持久化，如需完成持久化，必须调用 DAO 组件；一旦调用 DAO 组件，将造成 DAO 对象和 Domain Object 的双向依赖；另外，Domain Object 中的业务逻辑方法还需要在业务逻辑组件中代理，才能真正实现持久化。

在上面的业务逻辑方法中，并没有进行持久化。如果抛开 DAO 层，这种 Domain Object 也可以独立测试，只是没有进行持久化。DAO 对象是变化最小的对象，它们都是进行基本的 CRUD 操作，在两种模型下的 DAO 对象没有变化。

另外还需要对业务逻辑对象进行改写，虽然 Domain Object 包含了基本业务逻辑方法，但业务逻辑对象还需代理这些方法。修改后的业务逻辑对象的代码如下：

```
public class FacadeManagerImpl
    implements FacadeManager
{
    //业务逻辑对象依赖的 DAO 对象
    private CategoryDao categoryDao;
    private NewsDao newsDao;
    private NewsReviewDao newsReviewDao;
    private UserDAO userDao;
    //此处还应该增加依赖注入 DAO 对象必需的 setter 方法
    ...
    //此处还应该增加其他业务逻辑方法
    ...
    //下面是增加新闻回复的业务方法
    public NewsReview addNewsReview(Long newsId , String content)
    {
        //根据新闻 id 加载新闻
        News news = newsDao.getNews(newsId);
        //通过 News 的业务方法添加回复
        NewsReview review = news.addNewsReview(content);
        //此处必须显示持久化消息回复
        newsReviewDao.saveNewsReview(review);
```

```
            return review;
        }
    }
```

从上面粗体字代码可以看出，此时 FacadeManagerImpl 组件的 addNewsReview()方法由 News 领域对象的 addNewsReview()方法提供实现，而业务逻辑对象仅对该方法进行简单的包装，执行必要的持久化操作。

在这里存在一个问题：业务逻辑方法很多，哪些业务逻辑方法应该放在 Domain Object 对象中实现，而哪些业务逻辑方法完全由业务逻辑对象实现呢？Rod Johnson 认为，可重用度高，与 Domain Object 密切相关的业务方法应放在 Domain Object 对象中实现。

业务逻辑方法是否需要由 Domain Object 实现的标准，从一定程度上说明了采用 Rich Domain Object 模型的原因。由于某些业务方法只是专一地属于某个 Domain Object，因此将这些方法由 Domain Object 实现，能提供更好的软件复用，能更好地体现面向对象的封装性。

Rich Domain Object 模型的各组件之间的关系大致如图 9.13 所示。

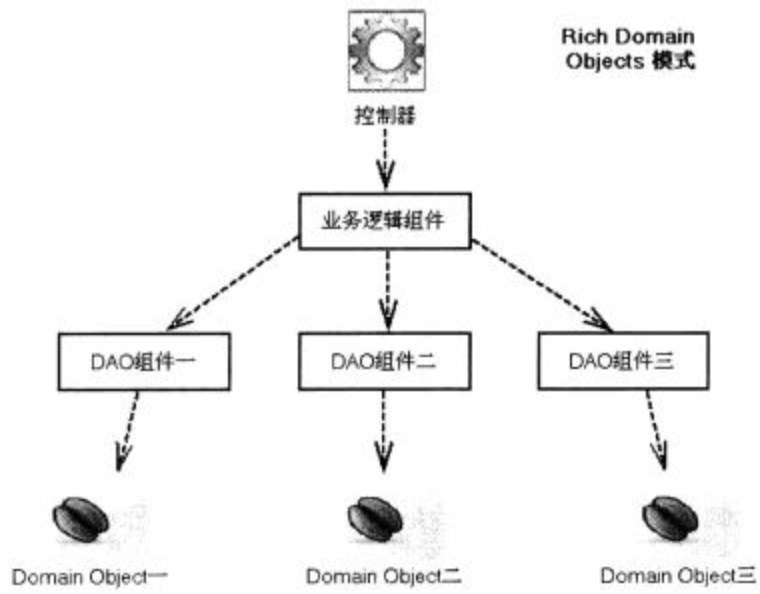

图 9.13　Rich Domain Object 的组件关系图

这种 Rich Domain Object 模型主要的问题是业务逻辑组件比较复杂：业务逻辑组件继续需要作为 DAO 组件的门面，而且还需要包装 Domain Object 里的业务逻辑方法，暴露这些业务逻辑方法，让 Action 组件可以调用这些方法。

为了简化业务逻辑对象的开发，Rich Domain Object 模型可以有如下两个方向的改变：

- 合并业务逻辑对象与 DAO 对象。
- 合并业务逻辑对象和 Domain Object。

9.4.3　合并业务逻辑对象与 DAO 对象

在这种模型下 DAO 对象不仅包含了各种 CRUD 方法，而且还包含各种业务逻辑方法。此时的 DAO 对象，已经完成了业务逻辑对象所有任务，变成了 DAO 对象和业务逻辑对象混合体。此时，业务逻辑对象依赖 Domain Object，既提供基本的 CRUD 方法，也提供相应的业务逻辑方法。

下面是这种架构设计的代码（此处 News 类的实现与前面的领域对象模型中的 News 类代码一样，此处不再给出）。

```
// NewsServiceHibernate 继承 HibernateDaoSupport,
//实现 NewsService 接口，它既是 DAO 组件，也是业务逻辑组件
public class NewsServiceHibernate
      extends HibernateDaoSupport implements NewsService
{
      //根据主键加载消息
      public News getNews(Long id)
      {
```

```
        return (News)getHibernateTemplate()
            .get(News.class, id);
    }
    //保存新的消息
    public void saveNews(News news)
    {
        getHibernateTemplate().saveOrUpdate(news);
    }
    //根据主键删除消息
    public void removeNews(Long id)
    {
        getHibernateTemplate().delete(getNews(id));
    }
    //查找全部的消息
    public List findAll()
    {
        getHibernateTemplate().find("from News");
    }
    //下面是增加新闻回复的业务方法
    public NewsReview addNewsReview(Long newsId
        , String content)
    {
        //根据新闻 id 加载新闻
        News news = newsDao.getNews(newsId);
        //通过 News 的业务方法添加回复
        NewsReview review = news.addNewsReview(content);
        //此处必须显示持久化消息回复
        newsReviewService.saveNewsReview(review);
        return review;
    }
}
```

正如上面见到的，DAO 对象和业务逻辑对象之间容易形成交叉依赖（可能某个业务逻辑方法的实现，必须依赖于原来的 DAO 对象）。当 DAO 对象被取消后，业务逻辑对象取代了 DAO 对象，因此变成了一个业务逻辑对象依赖多个业务逻辑对象。而每个业务逻辑对象都可能包含需要多个 DAO 对象协作的业务方法，从而导致业务逻辑对象之间的交叉依赖。

业务逻辑对象和 DAO 对象合并后的组件关系如图 9.14 所示。

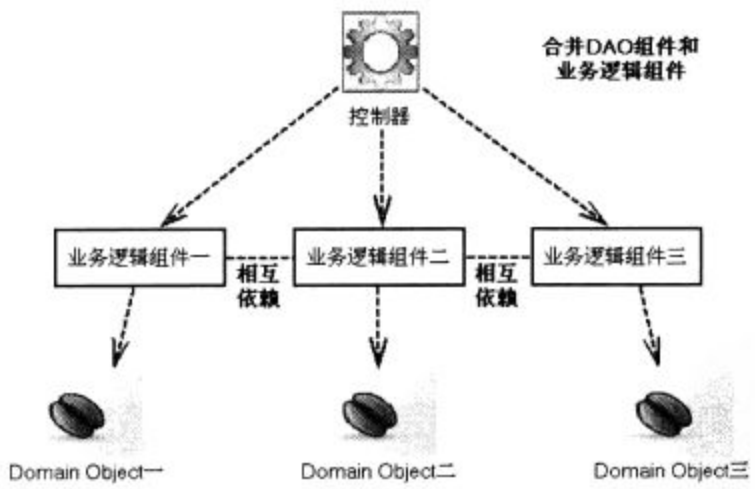

图 9.14　合并 DAO 对象和业务逻辑对象

这种模型也导致了 DAO 方法和业务逻辑方法混合在一起，显得职责不够单一，软件分层结构不够清晰。而且使业务逻辑对象之间交叉依赖，容易产生混乱，未能做到彻底的简化。

9.4.4　合并业务逻辑对象和 Domain Object

在这种架构下，所有的业务逻辑都应该被放在 Domain Object 里面，而此时的业务逻辑层不再是传统的业务逻辑层，它仅仅封装了事务和少量逻辑，不再提供任何业务逻辑的实现。而 Domain Object 依赖于 DAO 对象执行持久化操作，此处 Domain Object 和 DAO 对象形成双向依赖。在这种设计思路

下，业务逻辑层变得非常“薄”，它的功能也变得非常微弱。

在这种设计架构下，几乎不再需要业务逻辑层，而 Domain Object 则依赖 DAO 对象完成持久化操作。下面是在这种架构设计下的 News 类代码。

```
public class News
{
    //此处省略了其他的属性
    ...
    //此处省略了属性对应的 setter 和 getter 方法
    ...
    //增加新闻回复的业务逻辑方法
    public NewsReview addNewsReview(String content)
    {
        //以默认构造器创建新闻回复实例
        NewsReview review = new NewsReview();
        //设置回复内容
        review.setContent(content);
        //设置回复的发布日期
        review.setPostDate(new Date());
        //设置回复的最后修改日期
        review.setLastModifyDate(new Date());
        //设置回复与消息的关联
        review.setNews(this);
        //直接调用 newsReviewsDao 完成消息回复的持久化
        newsReviewsDao.save(review);
        return review;
    }
    //此处省略了重写的 hashCode、equals 等方法
    ...
}
```

正如上面程序中粗体字代码所示，此时 Domain Object 对象直接调用了 DAO 组件的持久化方法，从而使得 Domain Object 本身就可实现整个业务逻辑功能。

在这种设计下，Domain Object 必须使用 DAO 对象完成持久化，因此 Domain Object 必须接收 IoC 容器的注入，而 Domain Object 获取容器注入的 DAO 对象，通过 DAO 对象完成持久化操作。

合并业务逻辑对象和 Domain Object 后各组件的关系如图 9.15 所示。

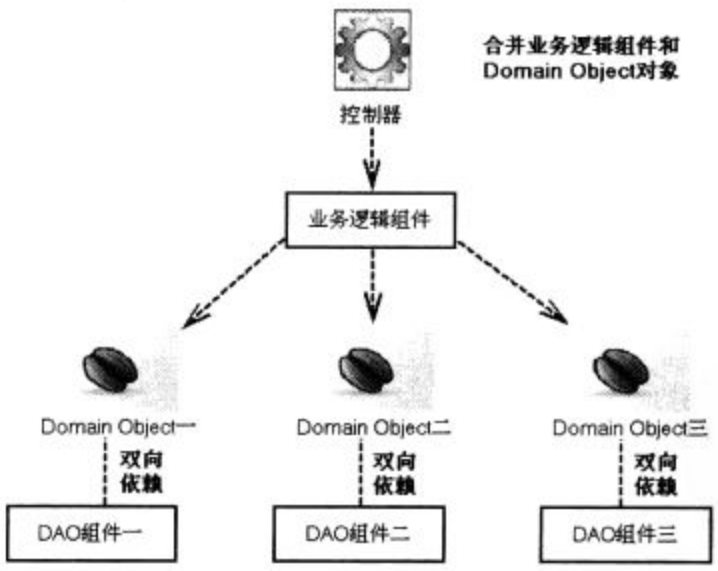

图 9.15　合并业务逻辑组件和 Domain Object

这种架构的优点是：业务逻辑对象非常简单，只提供简单的事务操作，业务逻辑对象无须依赖于 DAO 对象。

但这种架构也存在如下缺点：

- DAO 对象和 Domain Object 形成了双向依赖，其复杂的双向依赖会导致很多潜在的问题。
- 业务逻辑层和 Domain 层的逻辑混淆不清，在实际项目中，极容易导致架构混乱。
- 由于使用业务逻辑对象提供事务封装特性，业务逻辑层必须对所有的 Domain Object 的逻辑提供相应的事务封装，因此业务逻辑对象必须重新定义 Domain Object 实现的业务逻辑，其工作

相当烦琐。

9.4.5 抛弃业务逻辑层

在 Rich Domain Object 架构的各种变化中，虽然努力简化业务逻辑对象，但业务逻辑对象依然存在，使用业务逻辑对象始终是 DAO 组件访问的门面。下面介绍两种更彻底的简化，彻底放弃业务逻辑层。

抛弃业务逻辑层也有两种形式：

- Domain Object 彻底取代业务逻辑对象。
- 由控制器直接调用 DAO 对象。

1. Domain Object 完全取代业务逻辑对象

这种架构设计是 9.4.4 节所介绍的架构的更激进的演化，由于在那种架构中业务逻辑对象的作用仅仅只提供事务封装，因此业务逻辑对象存在的必要性不是很大，考虑对 Domain Object 的业务逻辑方法增加事务管理，而 Web 层的控制器则直接依赖于 Domain Object。

这种架构设计更加简化，Domain Object 与 DAO 组件形成双向依赖，而 Web 层的控制器直接调用 Domain Object 的业务逻辑方法。这种架构设计与 Ruby On Rails 框架的设计极为相似，Ruby On Rails 应用也使用 Domain Object 完整地实现了业务逻辑方法。

这种架构设计的优点是：分层少，代码实现简单。但也存在如下缺点：

- 业务逻辑对象的所有业务逻辑都将在 Domain Object 中实现，势必引起 Domain Object 的混乱。
- Domain Object 必须直接传递到 Web 层，从而将持久化 API 直接传递到 Web 层，因此可能引发一些意想不到的问题。

在这种架构设计下，Domain Object 相当不稳定。如果业务逻辑需要改变，Domain Object 也需要发生改变，而且 DAO 对象与 Domain Object 形成双向依赖，这将导致底层的 Domain Object 和 DAO 对象都需要修改，从而导致这种架构设计的分层失去意义。各层之间以强耦合方式组合在一起，各层对象互相依赖，可扩展性较差。

注意：

Rails 项目与 Java EE 项目不一样，因为 Rails 项目中的 Domain Object 会继承 ActiveRecord::Base 类，而且 Ruby 是一门动态语言，这就使得 Domain Object 本身直接具有了持久化操作访问，相当于 Domain Object 本身就具有了所有 DAO 方法；而且由于 Ruby 语言的动态特征，开发 Rails 项目时几乎无须考虑 DAO 组件，因而 Rails 项目采用这种架构设计依然可包括很好的可扩展性。

2. 控制器完成业务逻辑

在这种模型里，控制器直接调用 DAO 对象的 CRUD 方法，通过调用基本的 CRUD 方法，完成对应的业务逻辑方法。在这种模型下，业务逻辑对象的功能由控制器完成。事务则推迟到控制器中完成，因此对控制器的 execute()方法增加事务控制即可。

对于基本的 CRUD 操作，控制器可直接调用 DAO 对象的方法，省略了业务逻辑对象的封装，这就是这种模型的最大优势。对于业务逻辑简单（当业务逻辑只是大量的 CRUD 操作时）的项目，使用这种模型也未尝不是一种好的选择。

但这种模型将导致控制器变得臃肿，因为每个控制器除了包含原有的 execute 方法之外，还必须包含所需要的业务逻辑方法的实现。极大地省略了业务逻辑层的开发，避免了业务逻辑对象不得不大量封装基本的 CRUD 方法的弊端。

这种模型的缺点极为明显，实际上这种架构模型极少被使用：

- 因为没有业务逻辑层，对于那些需要多个 DAO 参与的复杂业务逻辑，在控制器中必须重复

实现，其效率低，也不利于软件重用。

- Web 层的功能不再清晰，Web 层的控制器相当复杂。Web 层不仅负责实现控制器逻辑，还需要完成业务逻辑的实现，因此必须精确控制何时调用 DAO 方法控制持久化。

这种架构模型还扩大了事务的影响范围。大部分情况下，只有业务逻辑方法需要增加事务控制，而 execute 方法无须增加事务控制。但如果 execute 方法直接调用了 DAO 对象的 CRUD 方法，则会导致这些方法不在事务环境下执行。为了让数据库访问都在事务环境下进行，不得不将事务范围扩大到整个 execute 方法。这必然导致性能降低。

9.5 本章小结

本章主要介绍了笔者在架构设计方面的一些经验。本章从企业应用开发面临的困难讲起，并给出了面对这些困难时应该采用怎样的应对策略。本章重点介绍了 Java EE 应用中常用的设计模式，本章并没有全面介绍 23 种设计模式，也不是仅仅介绍了设计模式的实现方法；而是侧重于介绍设计模式在 Spring、Hibernate 以及 Java EE 应用中的用途，并分析了使用这些模式的优势。

本章最后还介绍了 Java EE 应用的常用架构，如贫血模型、领域对象模型，以及领域对象模型的几种简化方式，并深入分析了几种架构各自的优缺点，让开发者对这些架构有更深刻的认识。