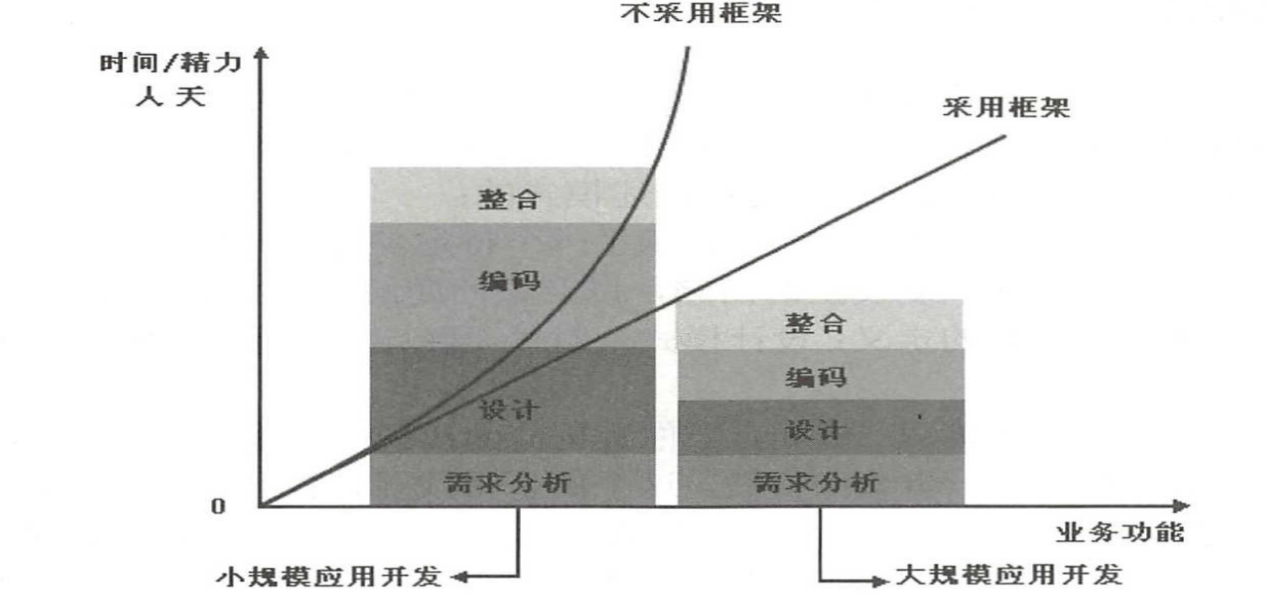
企业级应用开发的思考和策略

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。

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，从而能极好的提高程序的解耦。XML文件常用语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，也便面了在更改数据库时，需要更改程序代码，从而提供更好的适应性。

高效性：

1. 如果采用缓冲池的技术。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。
2. 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，这将导致系统的性能下降。



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。每一个模式描述了不断重复发生的问题，以及该问题解决方案的核心技术。

设计模式分类：

1）创建型:创建对象时，不在直接实例化对象;而是根据特定场景，由程序来确定创建对象的方式，从而保证更高的性能，更好的架构优势。创建型模式主要有简单工厂模式，工厂方法，抽象工厂模式，单例模式，生成器模式和原型模式。

2）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。

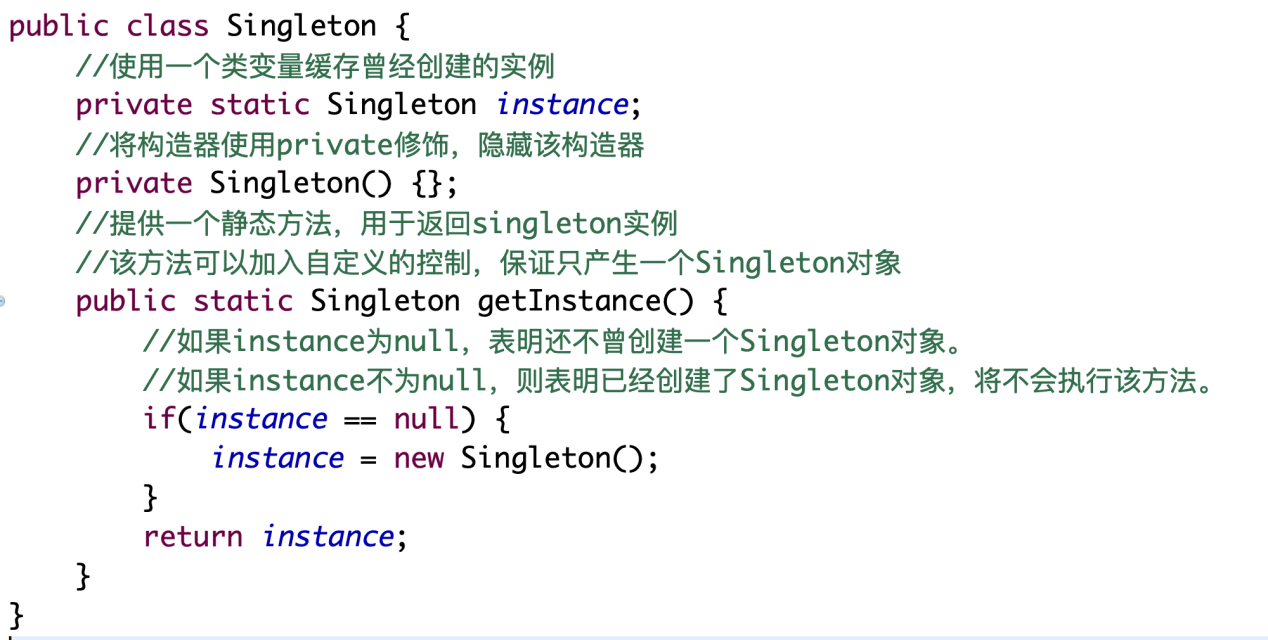
3）行为型:用于帮助系统间各对象的通信，以及如何控制复杂系统中的流程。行为型模式主要有命名模式，解释器模式，迭代器模式，中介者模式，备忘录模式，观察者模式，状态模式，策略模式，模版模式和访问者模式。

单例模式: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实例默认为单利的行为方式。

手动创建单例模式：1:使用private修饰该类的构造器。

2: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。

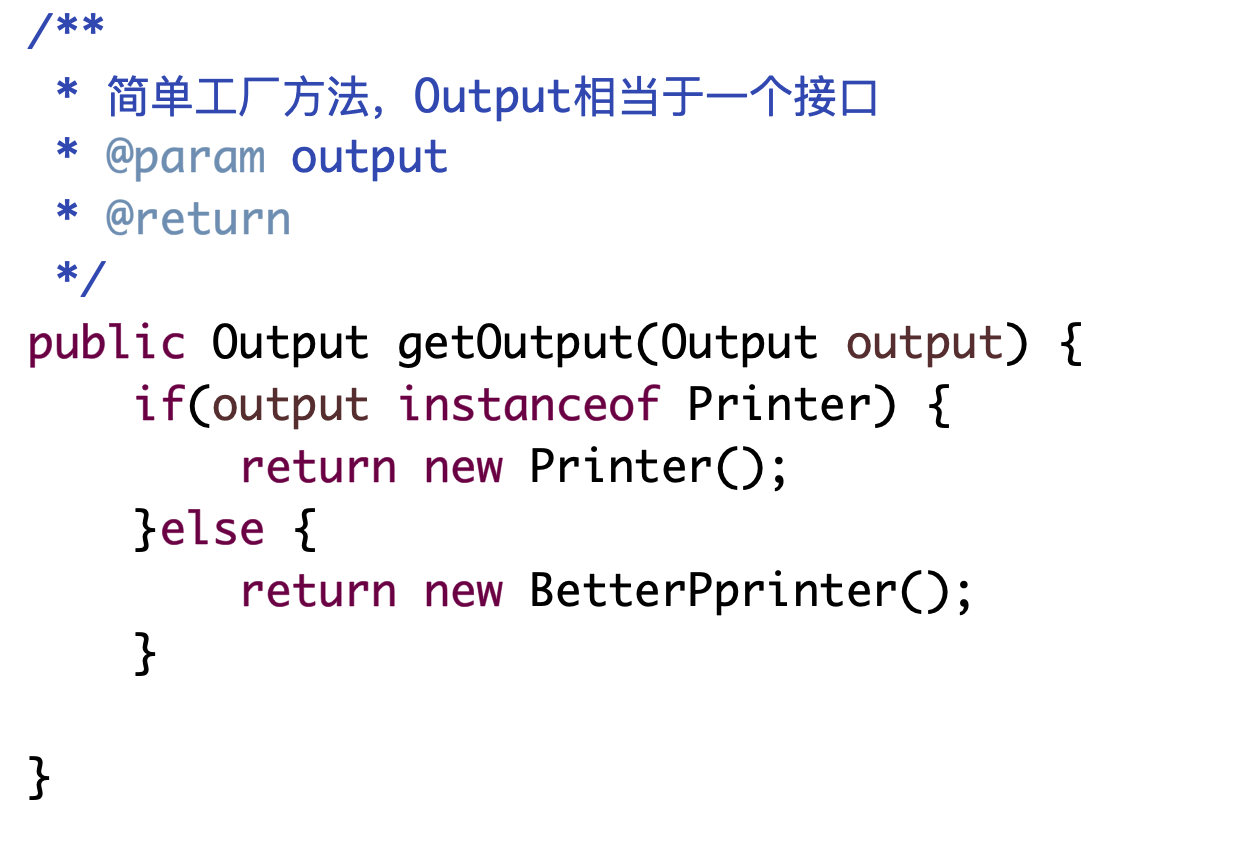


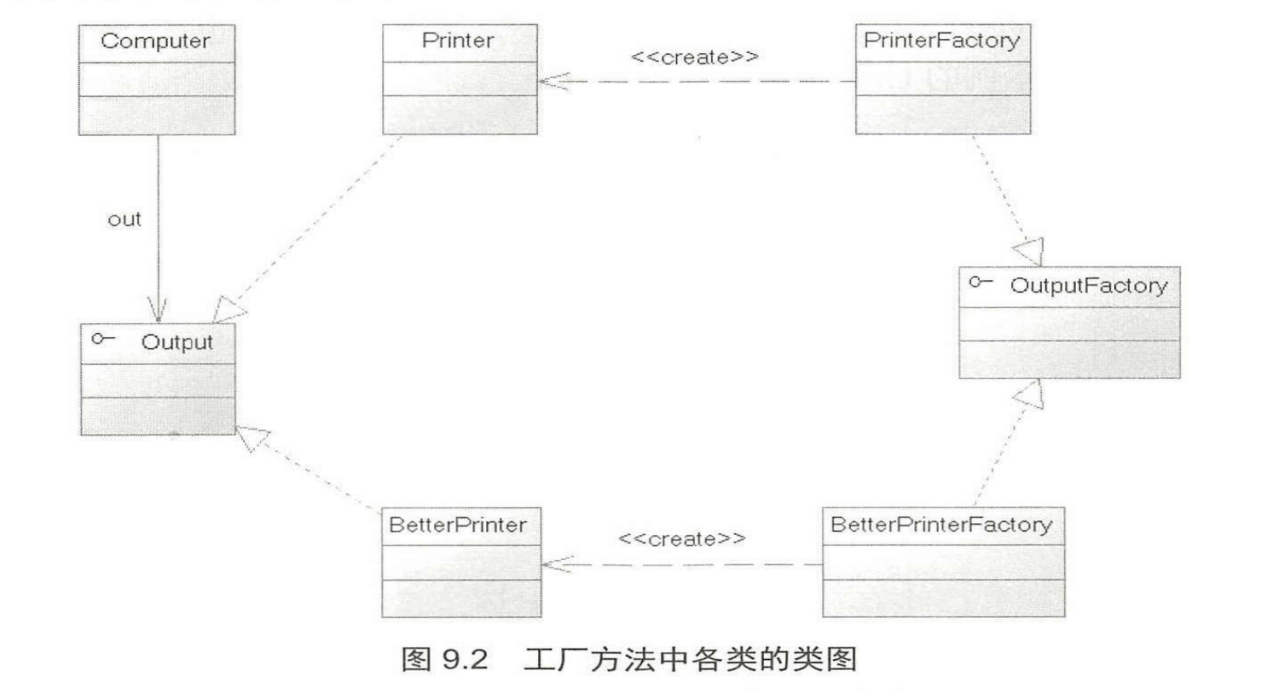
1)减少创建Java实例所带来的系统开销。

2)便于系统跟踪单个Java实例的声明周期，实例状态等。

工厂方法:将多个类对象交给工厂来生成的设计方式被称为简单工厂模式。

使用简单工厂模式的优势是:让对象的调用者和对象的创建过程分离，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，工厂类也要做对应的修改。



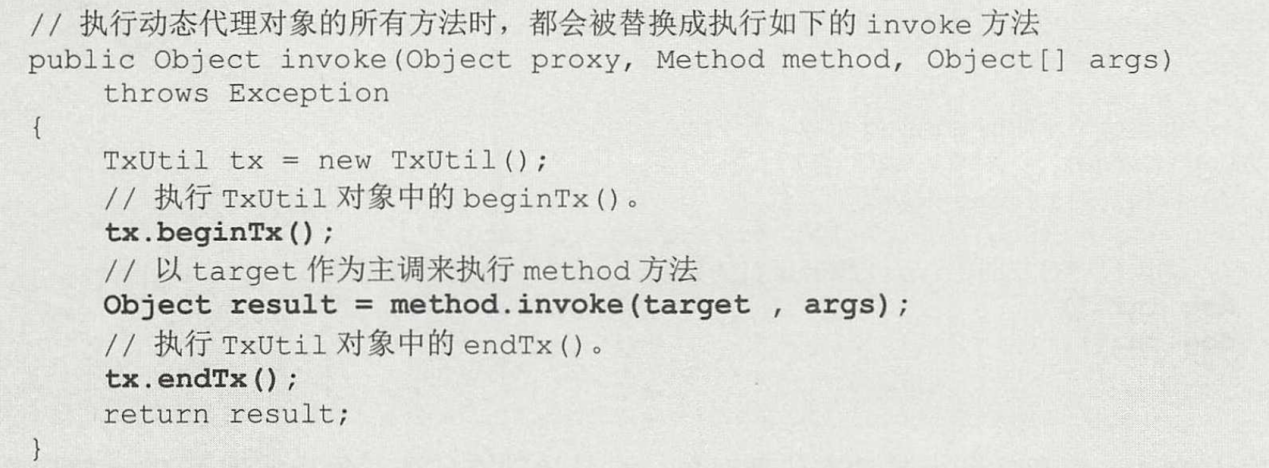


不要过分纠缠简单工厂模式，抽象工厂模式这些概念，可以把它们统称为工厂模式。如果工厂直接生被调用对象，那就是简单工厂模式；如果工厂生产了工厂对象，那就会升级成周详工厂模式。

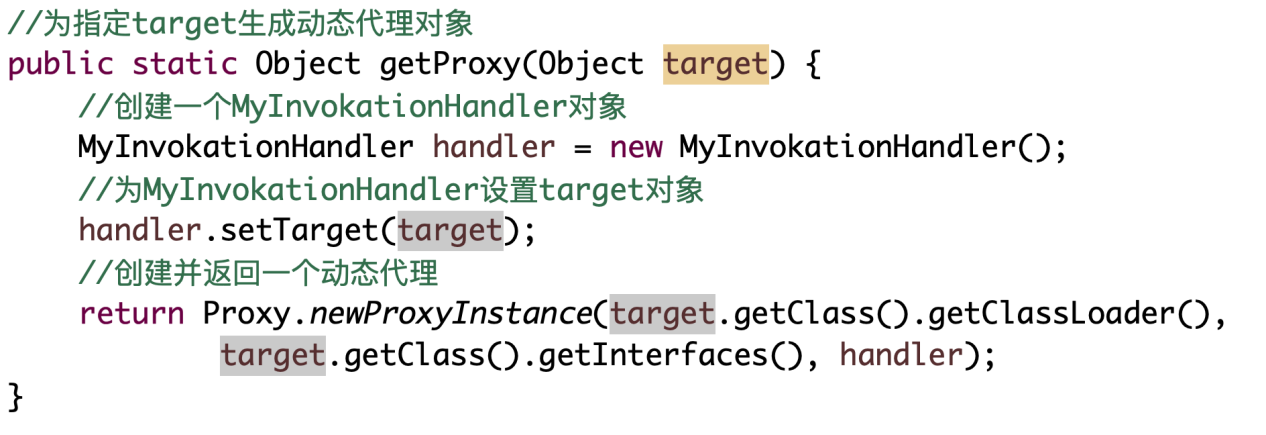
代理模式:是一种应用非常广泛的设计模式，当客户端代码需要调用某个对象时，客户端实际上也不关心是否准确得到该对象，它只要一个能提供该功能的对象即可，此时就可返回该对象的代理(Proxy)。在这种设计方式下，系统会为某个对象提供一个代理对象，并由代理对象控制对源对象的引用。代理就是一个Java对象代理另一个Java对象来采取行动。

1)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。

2)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。这这种情况下，使用代理模式可以显著提高系统运行性能。(Hibernate延迟加载所采用的设计模式)



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。



上面的动态代理工厂类提供了一个getProxy()方法，该方法为target对象生产一个动态代理对象，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，实际上将变成执行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。

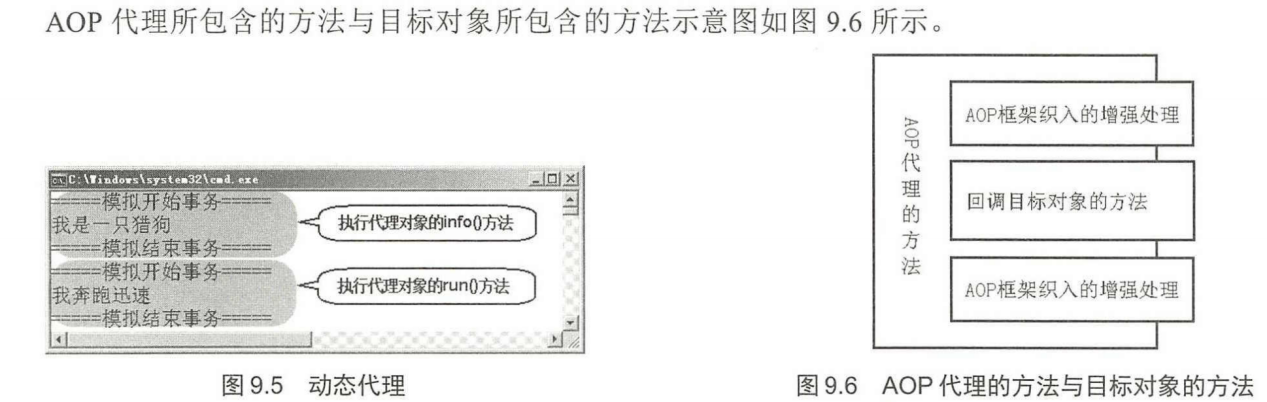
1）创建TxUtil实例。

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。

这种动态代理在AOP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。



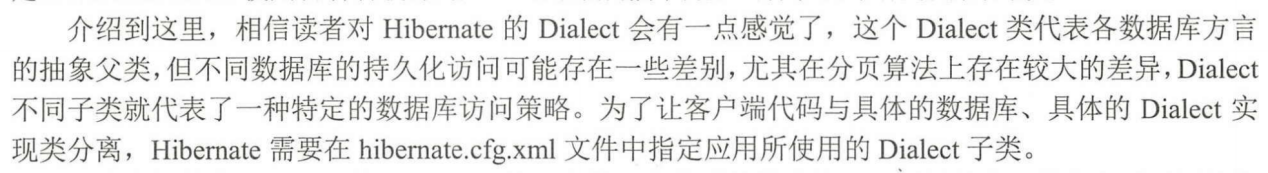
策略模式:用于封装系列的算法，这些算法通常被封装在一个被称为Context的类中，客户端程序可以自由选择其中一种算法，或让Context为客户端选择一个最佳的算法——使用策略模式的优势是为了支持算法的自由切换。





上面程序第一行穿件一个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，第二行粗体指定使用VIP打折策略，

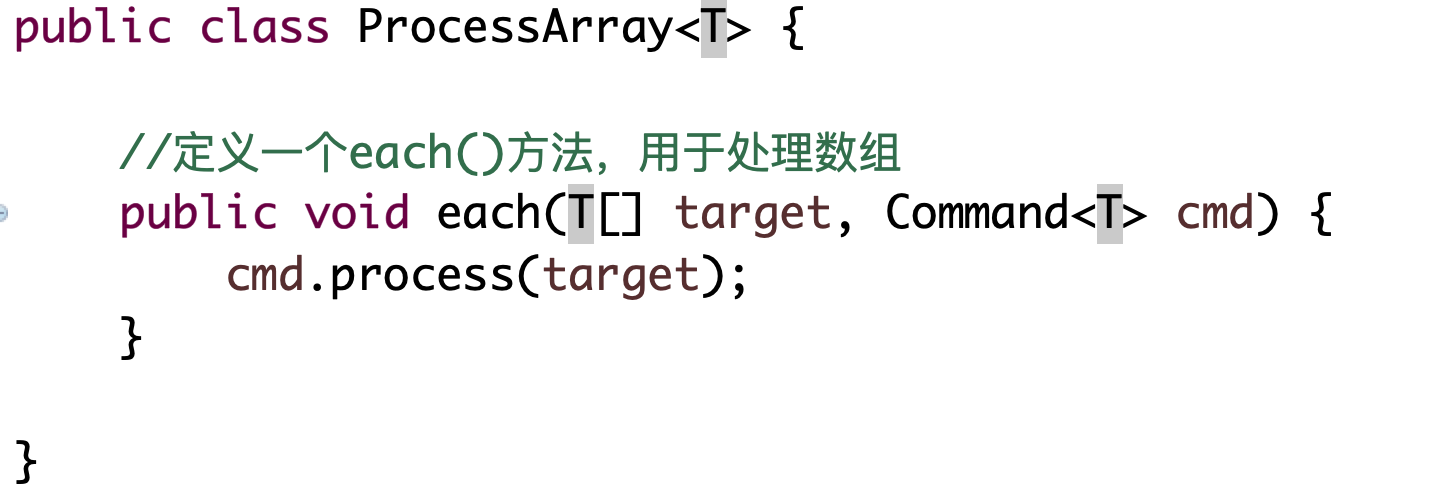
有一个缺点，客户端代码需要和不同的策略类耦合。为了弥补这个不足，可以考虑使用配置文件来指定DiscountContext使用哪种打折策略——这就彻底分离客户端代码和具体打折策略类。



命令模式: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:假设有个方法需要便利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，甚至要求方法执行体的代码也可以变化，需要能把这种“处理行为”作为一个参数传入该方法。

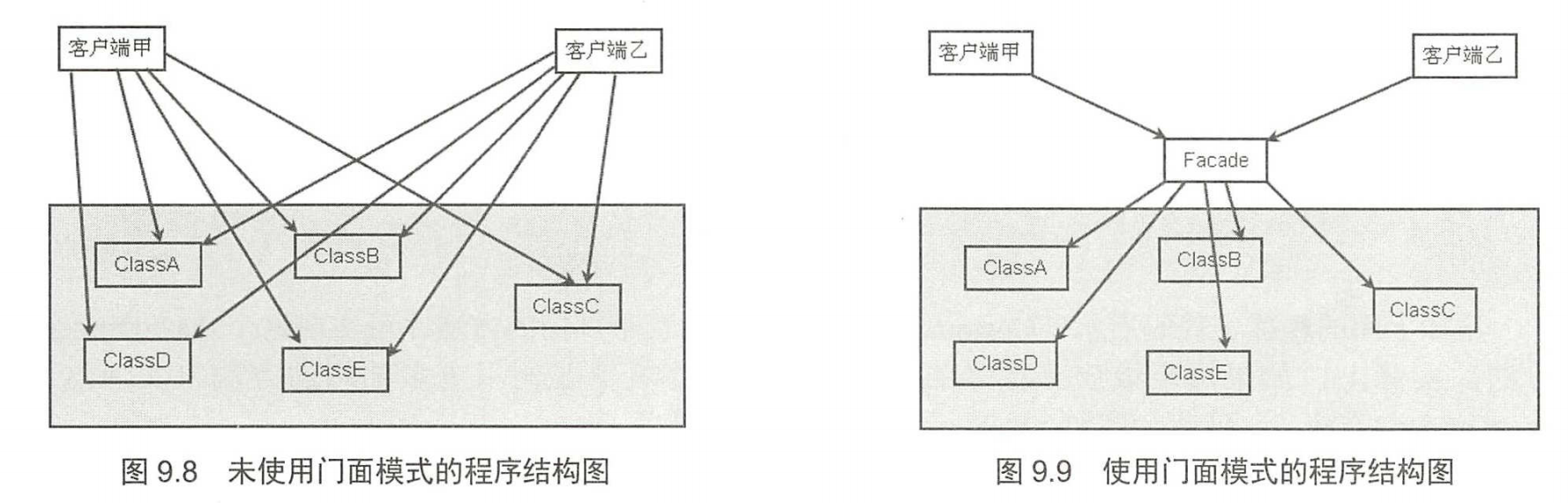
在java语言中，类才是一等公民，方法也不能独立存在，所以实际传入该方法的应该是一个对象，该对象通常是某个接口的匿名实现类的实例，该接口通常被称为命令接口，这种设计方式也被称为命令模式。



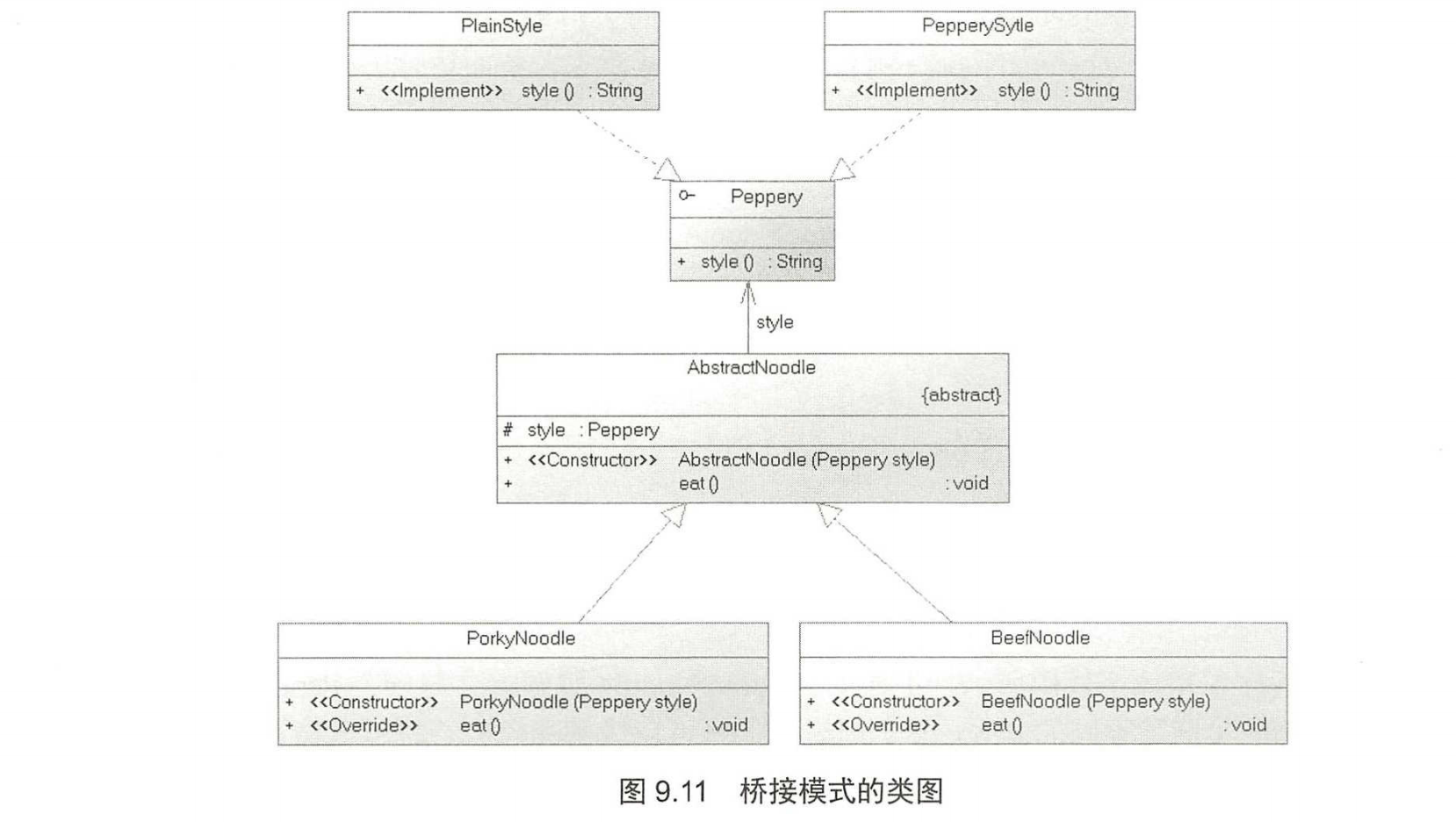


门面模式（Facade）:可为这些类提供一个简化的接口，从而简化访问这些类的复杂性，有时这种简化可能降低访问这些底层类的灵活性，但除了要求特别苛刻的客户端外，它通常都可以提供所需的全部功能，当然，那些苛刻的用户仍然可以直接访问底层的类和方法。

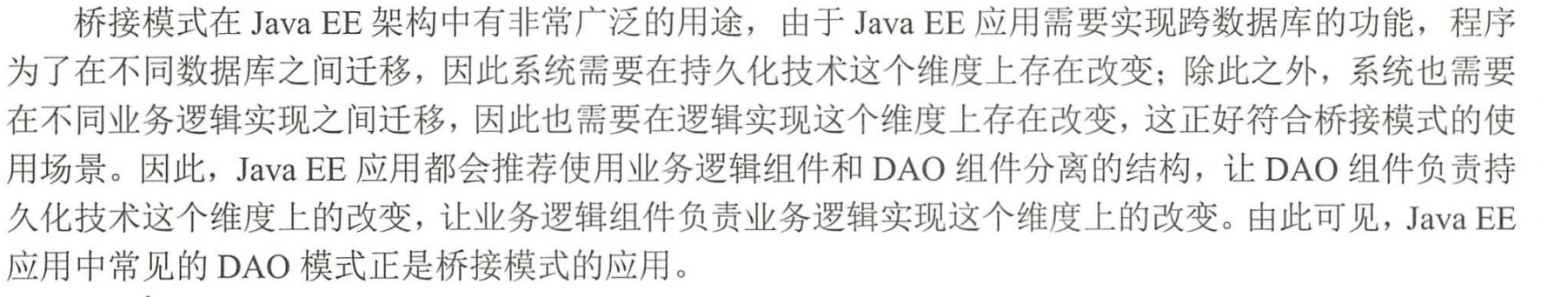
也没称为正面模式，外观模式。这种模式用于将一组负责的类包装到一个简单的外部接口中。

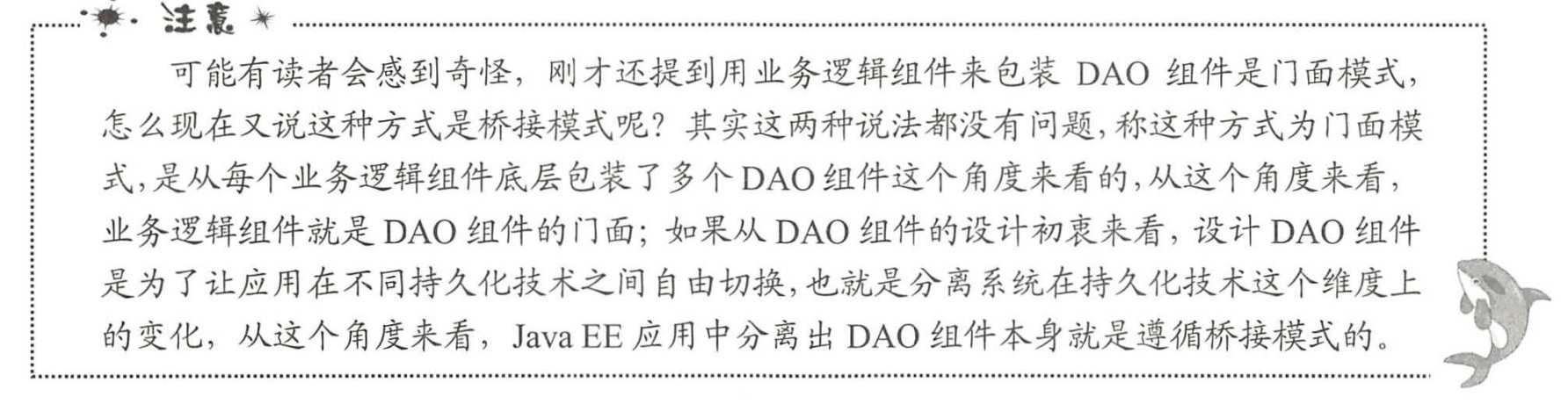


桥接模式:是一种结构型模型，它主要应对的是：由于实际的需要，某个类具有两个或两个以上的纬度变化，如果只是使用继承将无法实现这种需要，或者使得设计变得相当臃肿。









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，从而是的观察者对象能自动更新。

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）。