几种常用的设计模式介绍

1. 设计模式的起源

最早提出“设计模式”概念的是建筑设计大师亚力山大 Alexander。在 1970 年他的《建筑的永恒之道》里描述了投计模式的发现，因为它已经存在了千百年之久，而现代才被通过大量的研究而被发现。

在《建筑的永恒之道》里这样描述：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**。

在他的另一本书《建筑模式语言》中提到了现在已经定义了 253 种模式。比如：

说明城市主要的结构：亚文化区的镶嵌、分散的工作点、城市的魅力、地方交通区

住宅团组：户型混合、公共性的程度、住宅团组、联排式住宅、丘状住宅、老人天地室内环境和室外环境、阴和阳总是一气呵成

针对住宅：夫妻的领域、儿童的领域、朝东的卧室、农家的厨房、私家的沿街露台、个人居室、起居空间的序列、多床卧室、浴室、大储藏室

针对办公室、车间和公共建筑物：灵活办公空间、共同进餐、共同小组、宾至如归、等候场所、小会议室、半私密办公室

尽管亚力山大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。“软件设计模式”，这个术语是在 1990 年代由 Erich Gamma 等人从建筑设计领域引入到计算机科学中来的。目前主要有 23 种。

2. 软件设计模式的分类

2.1. 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。

2.2. 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**、亨元模式和**代理模式**。

2.3. 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。

3. 常见设计模式介绍

3.1. 单例模式(singleton)

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。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。

```
public class Singleton {

    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);
    }

    //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(){
        System.out.println("Singleton被构造！");
    }

    //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;
    }
}
```

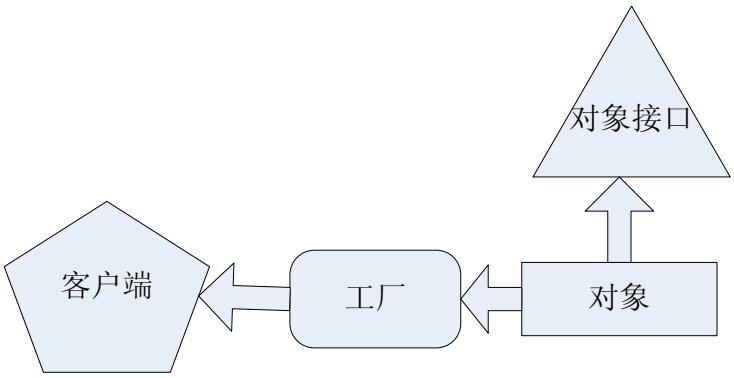
单例模式主要有如下两个优势：

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 实例的生命周期、实例状态等。

3.2. 简单工厂(StaticFactory Method)

简单工厂模式是由一个工厂对象决定创建出哪一种产品类的实例。简单工厂模式是工厂模式家族中最简单实用的模式，可以理解为是不同工厂模式的一个特殊实现。

A 实例调用 B 实例的方法，称为 A 依赖于 B。如果使用 new 关键字来创建一个 B 实例（硬编码耦合），然后调用 B 实例的方法。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。而用工厂模式则不需要关心 B 对象的实现、创建过程。



Output，接口

```
public interface Output
{
    //接口里定义的属性只能是常量
    int MAX_CACHE_LINE = 50;
    //接口里定义的只能是public的抽象实例方法
    void out();
    void getData(String msg);
}
```

Printer，Output 的一个实现

```
//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;
        }
    }
}
```

BetterPrinter，Output 的一个实现

```
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;
        }
    }
}
```

OutputFactory，简单工厂类

```
    public Output getPrinterOutput(String type) {
        if (type.equalsIgnoreCase("better")) {
            return new BetterPrinter();
```

```
        } else {
            return new Printer();
        }
    }
```

Computer

```
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.getPrinterOutput("normal"));
        c.keyIn("建筑永恒之道");
        c.keyIn("建筑模式语言");
        c.print();


        c = new Computer(of.getPrinterOutput("better"));
        c.keyIn("建筑永恒之道");
        c.keyIn("建筑模式语言");
        c.print();
    }
```

使用简单工厂模式的优势：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。

3.3. 工厂方法(Factory Method)和抽象工厂(Abstract Factory)

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，不同的工厂类和产不同的产品。

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，如：

```
//工厂类的定义 1
public class BetterPrinterFactory
    implements OutputFactory
{
    public Output getOutput()
    {
        //该工厂只负责产生BetterPrinter对象
        return new BetterPrinter();
    }
}
//工厂类的定义 2
public class PrinterFactory
    implements OutputFactory
{
    public Output getOutput()
    {
        //该工厂只负责产生Printer对象
        return new Printer();
    }
}
//工厂类的调用
//OutputFactory of = new BetterPrinterFactory();
OutputFactory of = new PrinterFactory();
Computer c = new Computer(of.getOutput());
```

使用简单工厂类，需要在工厂类里做逻辑判断。而工厂类虽然不用在工厂类做判断。但是带来了另一种耦合：客户端代码与不同的工厂类耦合。

为了解决客户端代码与不同工厂类耦合的问题。在工厂类的基础上再增加一个工厂类，该工厂类不制造具体的被调用对象，而是制造不同工厂对象。如：

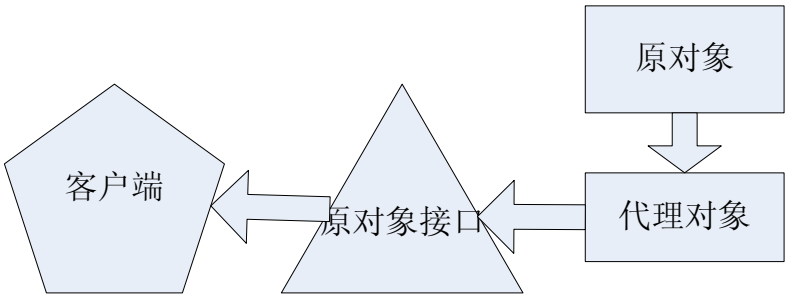
```
//抽象工厂类的定义，在工厂类的基础上再建一个工厂类
public class OutputFactoryFactory
{
    //仅定义一个方法用于返回输出设备。
    public static OutputFactory getOutputFactory(
        String type)
    {
        if (type.equalsIgnoreCase("better"))
        {
```

```
            return new BetterPrinterFactory();
        }
        else
        {
            return new PrinterFactory();
        }
    }
}

//抽象工厂类的调用
OutputFactory of = OutputFactoryFactory.getOutputFactory("better");
Computer c = new Computer(of.getOutput());
```

3.4. 代理模式(Proxy)

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）。



代理就是一个 Java 对象代表另一个 Java 对象来采取行动。如：

```
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();
    }
}
```

调用时，先不创建：

```
Image image = new ImageProxy(null);
```

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。

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以作为目标对象使用，而且增强了目标对象的功能。如：

Panther

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

GunPanther

```
public class GunPanther implements Panther
{
    //info方法实现，仅仅打印一个字符串
    public void info()
    {
        System.out.println("我是一只猎豹！");
    }
    //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");
    }
}
```

MyProxyFactory，创建代理对象

```
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);
    }
}
```

MyInvokationHandler，增强代理的功能

```
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;
    }
}
```

TxUtil

```
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=====");
    }
}
```

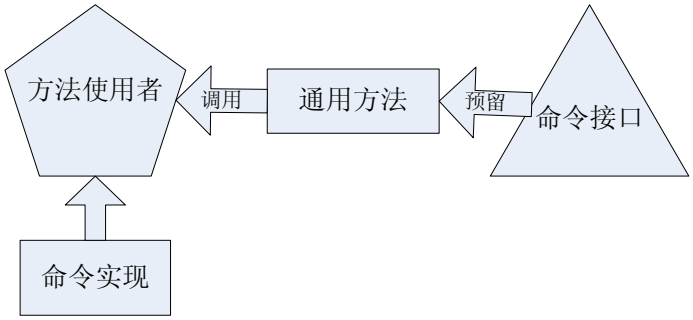
测试

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

Spring 所创建的 AOP 代理就是这种动态代理。但是 Spring AOP 更灵活。

3.5. 命令模式(Command)

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。（在某些编程语言如 Ruby、Perl 里，允许传入一个代码块作为参数。但 Jara 暂时还不支持代码块作为参数）。在 Java 中，传入该方法的是一个对象，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。



如：

Command

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

ProcessArray

```
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);
    }
}
```

TestCommand

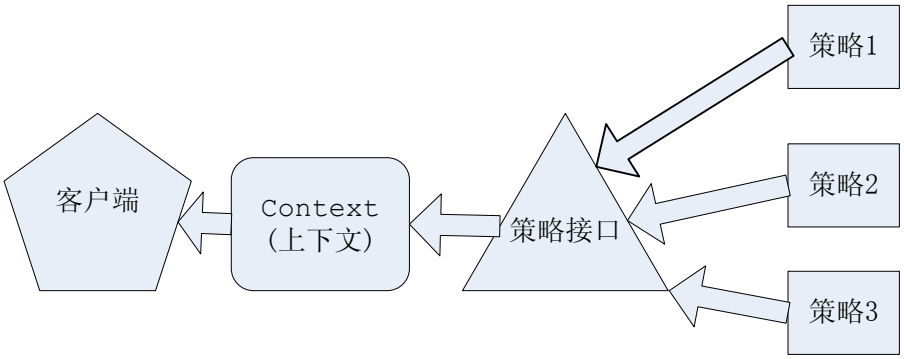
```
public class TestCommand
{
    public static void main(String[] args)
    {
        ProcessArray pa = new ProcessArray();
        int[] target = {3, -4, 6, 4};
        //第一次处理数组，具体处理行为取决于Command对象
        pa.each(target , new Command()
        {
            //重写process()方法，决定具体的处理行为
            public 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);
            }
        });
    }
}
```

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：

```
public interface HibernateCallback
{
    Object doInHibernate(Session session);
}
```

3.6. 策略模式(Strategy)

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 为客户端选择一种最佳算法——使用策略模式的优势是为了支持算法的自由切换。



DiscountStrategy，折扣方法接口

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

OldDiscount，旧书打折算法

```
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;
    }
}
```

VipDiscount，VIP 打折算法

```
//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;
    }
}
```

策略定义

```
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 setDiscount(DiscountStrategy strategy)
    {
        this.strategy = strategy;
    }
}
```

测试

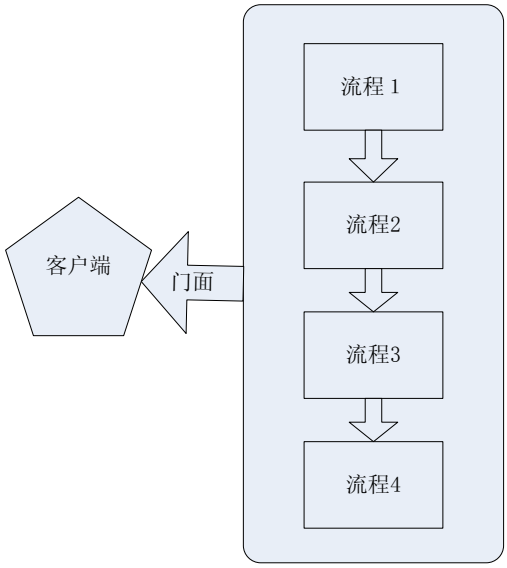
```
    public static void main(String[] args)
    {
        //客户端没有选择打折策略类
        DiscountContext dc = new DiscountContext(null);
        double price1 = 79;
        //使用默认的打折策略
        System.out.println("79元的书默认打折后的价格是："
            + dc.getDiscountPrice(price1));
        //客户端选择合适的VIP打折策略
     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

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。

3.7. 门面模式(Facade)

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。

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。



原来的方式

```
        // 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);
```

门面模式

```
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();
    }
```

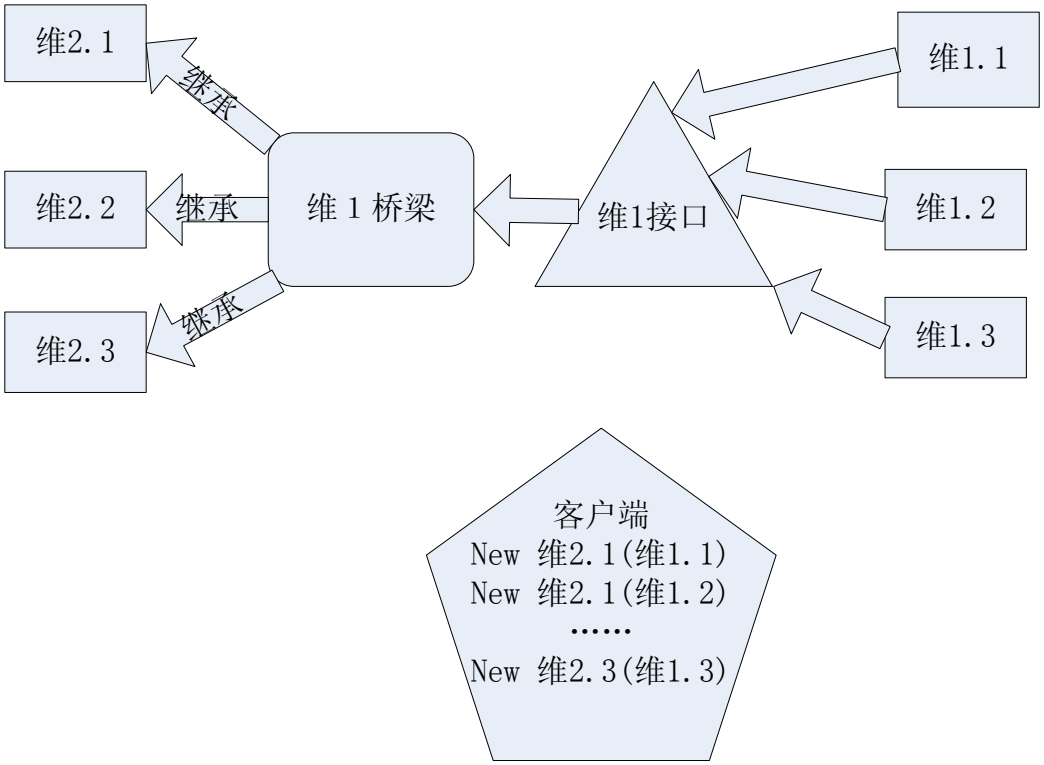
```
    public void serveFood() {
        // 依次调用三个部门的方法，封装成一个serveFood()方法
        String food = pay.pay();
        food = cook.cook(food);
        waiter.serve(food);
    }
}
```

门面模式调用

```
        Facade f = new Facade();
        f.serveFood();
```

3.8. 桥接模式(Bridge)

由于实际的需要，某个类具有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。而桥接模式的做法是**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。



Peppery 口味风格接口：

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

口味之一

```
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...";
    }
}
```

口味之二

```
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...";
    }
}
```

口味的桥梁

```
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();
}
```

材料之一，继承口味

```
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());
    }
}
```

材料之二，继承口味

```
public class BeefMoodle extends AbstractNoodle
{
```

```
    public BeefMoodle(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());
    }
}
```

主程序

```
public class Test
{
    public static void main(String[] args)
    {
        //下面将得到"辣味"的牛肉面
        AbstractNoodle noodle1 = new BeefMoodle(
            new PepperySytle());
        noodle1.eat();
        //下面将得到"不辣"的牛肉面
        AbstractNoodle noodle2 = new BeefMoodle(
            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();
    }
}
```

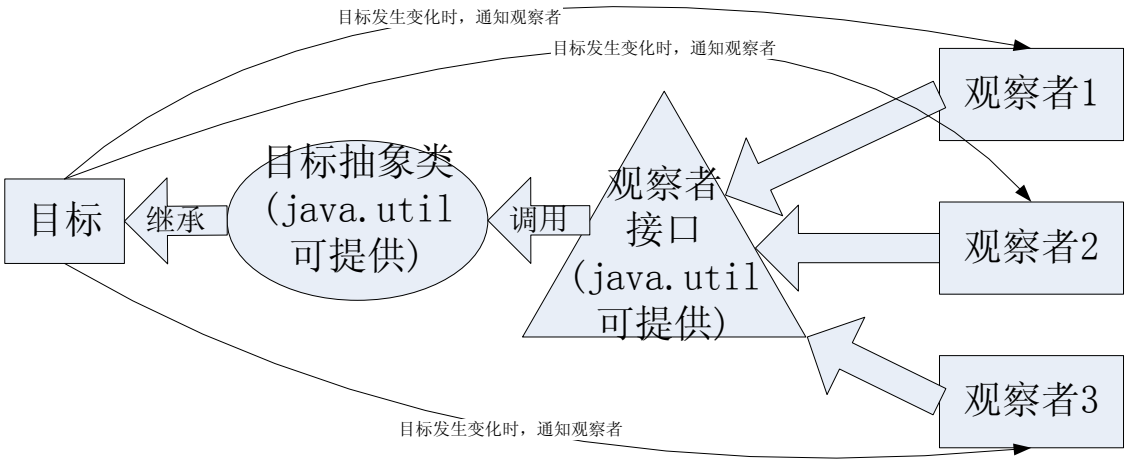
Java EE 应用中常见的 DAO 模式正是桥接模式的应用。

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。

3.9. 观察者模式(Observer)

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）。



Observer，观察者接口：

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

Observable，目标或主题：

```
import java.util.ArrayList;
import java.util.List;
import java.util.Iterator;

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.add(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);
```

```
        }
    }
}
```

Product 被观察类：

```
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);
    }
}
```

NameObserver 名称观察者：

```
import javax.swing.JFrame;
import javax.swing.JLabel;
```

```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);
        }
    }
}
```

PriceObserver 价格观察者：

```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);
        }
    }
}
```

测试：

```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);
    }
}
```

其中 Java 工具类提供了被观察者抽象基类：java.util.Observable。观察者接口：java.util.Observer。

我们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，主题/订阅模式下的 JMS 本身就是观察者模式的应用。