第 28 章　男人和女人——访问者模式

28.1　男人和女人！

时间：7 月 18 日 21 点　　地点：小菜大鸟住所的客厅　　人物：小菜、大鸟

“……

男人这本书的内容要比封面吸引人，女人这本书的封面通常是比内容更吸引人。

男人的青春表示一种肤浅，女人的青春标志一种价值。

男人在街上东张西望，被称做心怀不轨；女人在路上左瞅右瞧，被叫做明眸善睐。

男人成功时，背后多半有一个伟大的女人。女人成功时，背后大多有一个不成功的男人。

男人失败时，闷头喝酒，谁也不用劝。女人失败时，眼泪汪汪，谁也劝不了。

男人恋爱时，凡事不懂也要装懂。女人恋爱时，遇事懂也装作不懂。

男人结婚时，感慨道：恋爱游戏终结时，‘有妻徒刑’遥无期。女人结婚时，欣慰曰：爱情长跑路漫漫，婚姻保险保平安。

……”

“小菜在发神经呐，念什么男人、女人、恋爱、结婚乱七八糟的东西？”大鸟问道。

“我在网上看到关于男人与女人的区别讨论，很有点意思，抄了几句念着玩玩。”小菜笑着道，“男人结婚时说是判了‘有妻徒刑’，女人结婚时说是买了爱情保险。你说这滑稽吗？”

“本来吗，男人和女人就是完全不相同的两类人，当然在对待各种问题上会有完全不相同的态度。”

“这样的对比还有很多。你听着，我念给你听……”小菜显得很兴奋。

“行了行了，没有事业的成功，你找出再多的男女差异也找不到女朋友的。还是好好学习吧。”大鸟打断了小菜说话，“今天我们需要聊最后一个模式，叫做访问者模式。”

“哦，那我们上课吧。”小菜不得不停止。

“访问者模式讲的是表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。”大鸟开始如夫子般念叨起来。

“我觉得男女对比这么多的原因主要就是因为人类在性别上就只有男人和女人两类。”小菜意犹未尽。

“小菜！你又打断了我。”大鸟一声大喝，“大鸟很生气，后果很严重。”接着说，“你还要不要学，到底是学访问者模式还是讨论男女关系？”

“最好是混在一起同时学习。”小菜调皮地说道。

“狗屁，这是一回事吗？男人女人与访问者模式有屁的关……”大鸟气得脏话都脱口而出，拿起书本，对着小菜的头上就拍了过去。突然，大鸟手停在了空中，“等等，你刚才说什么？”

“我说什么？我说混在一起学习。”小菜快速躲开。

“前面一句，”大鸟似乎想到了什么。

“前面我说了什么，哦，我是说人类只分为男人和女人，所以才会有这么多的对比。”

“OK，就是这句话了。今天咱们就来讨论讨论这男人与女人的问题。”大鸟把举着书本的手放了下来，微笑着说道。

“大鸟，不学习设计模式了？”轮到小菜疑惑了。

“学呀，不过如你所愿，我们混在一起学习。”

“这之间也有关系？”小菜更加丈二和尚摸不着头脑。

“先不谈模式，你能不能把刚才的那些对比，用控制台的程序写出来，打到屏幕上？”大鸟避而不答。

“这个，应该不难实现。不过有什么意义呢？”

“少来废话，写出来再讲。”

“哦。”

28.2 最简单的编程实现

十分钟后，小菜的程序就写出来了。

```
static void Main(string[] args)
{
    Console.WriteLine("男人成功时，背后多半有一个伟大的女人。");
    Console.WriteLine("女人成功时，背后大多有一个不成功的男人。");
    Console.WriteLine("男人失败时，闷头喝酒，谁也不用劝。");
    Console.WriteLine("女人失败时，眼泪汪汪，谁也劝不了。");
    Console.WriteLine("男人恋爱时，凡事不懂也要装懂。");
    Console.WriteLine("女人恋爱时，遇事懂也装作不懂。");

    Console.Read();
}
```

“小菜呀，这样的代码你也拿得出手？”大鸟讥讽地说道。

“你不是要把那些对比打在屏幕上吗？我做到了呀。”小菜理直气壮。

“但这和打印‘Hello World’有什么区别，难道你是第一天学习编程？”

“那你要我怎么样写才可以呢？”

“你至少要分析一下，这里面有没有类可以提炼，有没有方法可以共享什么的。”

“哦，你是这个意思，早说呀。这里面至少男人和女人应该是两个不同的类，男人和女人应该继承人这样一个抽象类。所谓的成功、失败或恋爱都是指人的一个状态，是一个属性。成功时如何如何，失败时如何如何不过是一种反应。好了，我写得出来了。”小菜开始得意道。

“这还有点面向对象的意思。快点写吧。”

28.3　简单的面向对象实现

半小时后，小菜写出了第二版的程序。

“人”类，是“男人”和“女人”类的抽象类

```
abstract class Person
{
    protected string action;
    public string Action
    {
        get { return action; }
        set { action = value; }
    }
    //得到结论或反应
    public abstract void GetConclusion();
}
```

“男人”类

```
class Man : Person
{
    //得到结论或反应
    public override void GetConclusion()
    {
        if (action == "成功")
        {
            Console.WriteLine("{0}{1}时，背后多半有一个伟大的女人。", this.GetType().Name,
            action);
        }
        else if (action == "失败")
        {
            Console.WriteLine("{0}{1}时，闷头喝酒，谁也不用劝。", this.GetType().Name, action);
        }
        else if (action == "恋爱")
        {
            Console.WriteLine("{0}{1}时，凡事不懂也要装懂。", this.GetType().Name, action);
        }
    }
}
```

this.GetType().Name 是获得当前类的名称，比如这里就是‘男人’

“女人”类

```
class Woman : Person
{
    //得到结论或反应
    public override void GetConclusion()
    {
        if (action == "成功")
        {
            Console.WriteLine("{0}{1}时，背后大多有一个不成功的男人。", this.GetType().Name,
            action);
        }
        else if (action == "失败")
        {
            Console.WriteLine("{0}{1}时，眼泪汪汪，谁也劝不了。",this.GetType().Name,action);
        }
        else if (action == "恋爱")
        {
            Console.WriteLine("{0}{1}时，遇事懂也装作不懂。", this.GetType().Name, action);
        }
    }
}
```

客户端代码

```
static void Main(string[] args)
{
    IList<Person> persons = new List<Person>();

    Person man1 = new Man();
    man1.Action = "成功";
    persons.Add(man1);
    Person woman1 = new Woman();
    woman1.Action = "成功";
    persons.Add(woman1);

    Person man2 = new Man();
    man2.Action = "失败";
    persons.Add(man2);
    Person woman2 = new Woman();
```

```
        woman2.Action = "失败";
        persons.Add(woman2);

        Person man3 = new Man();
        man3.Action = "恋爱";
        persons.Add(man3);
        Person woman3 = new Woman();
        woman3.Action = "恋爱";
        persons.Add(woman3);

        foreach (Person item in persons)
        {
            item.GetConclusion();
        }

        Console.Read();
    }
```

结果显示

```
男人成功时，背后多半有一个伟大的女人。
女人成功时，背后大多有一个不成功的男人。
男人失败时，闷头喝酒，谁也不用劝。
女人失败时，眼泪汪汪，谁也劝不了。
男人恋爱时，凡事不懂也要装懂。
女人恋爱时，遇事懂也装作不懂。
```

“大鸟，现在算是面向对象的编程了吧。”

“粗略看，应该是算，但你不觉得你在‘男人’类与‘女人’类当中的那些 if……else……很是碍眼吗？”

“不这样不行呀，反正也不算多。”

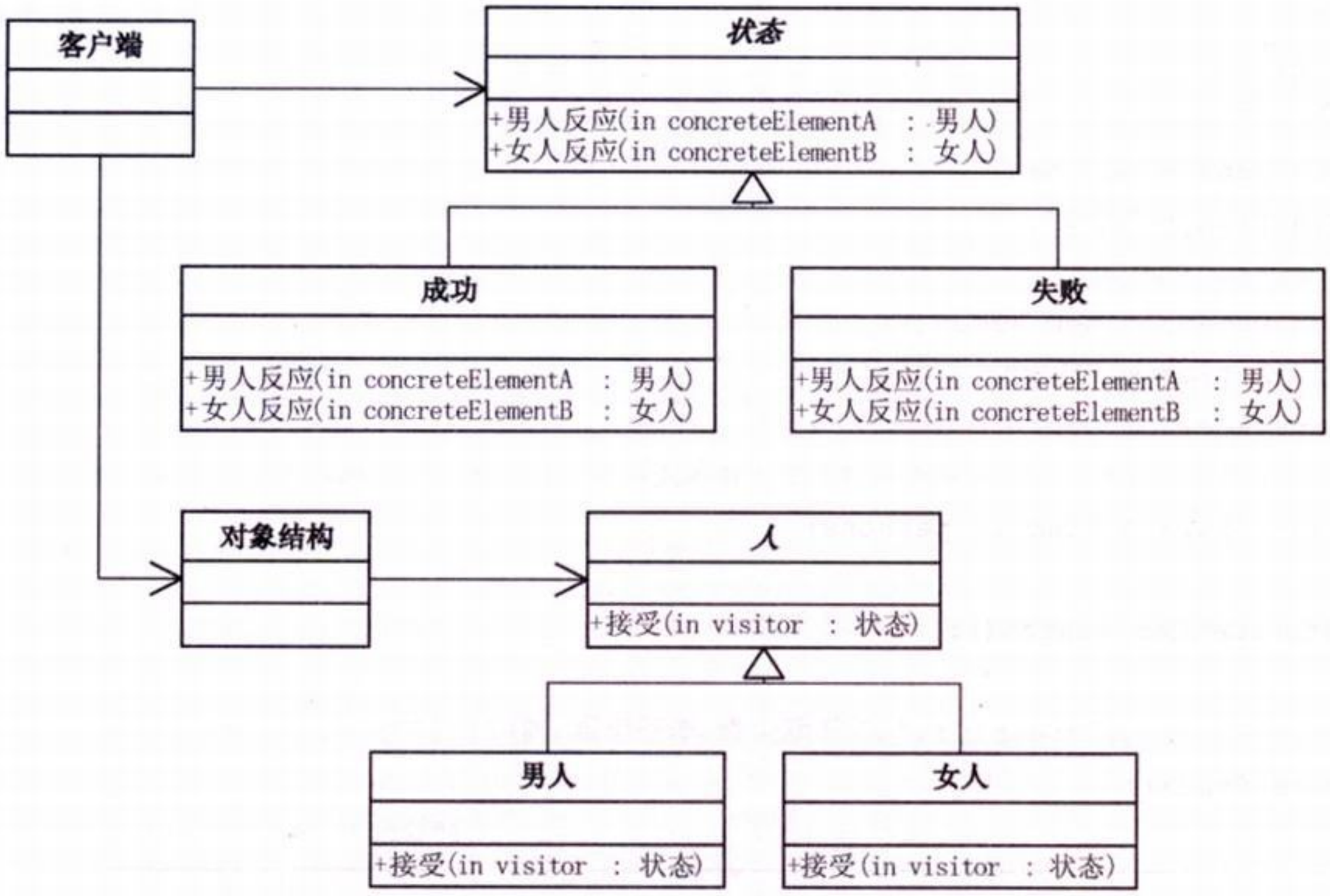
“如果我现在要增加一个‘结婚’的状态，你需要改什么？”

“那这两个类都需要增加分支判断了。”小菜无奈地说，“你说的意思我知道，可是我真的没有办法去处理这些分支，我也想过，把这些状态写成类，可是那又如何处理呢？没办法。”

“哈，办法总是有的。只不过复杂一些。”

28.4　用了模式的实现

大鸟帮助小菜画出了结构图并写出了代码



‘状态’的抽象类和‘人’的抽象类

```
abstract class Action
{
    //得到男人结论或反应
    public abstract void GetManConclusion(Man concreteElementA);
    //得到女人结论或反应
    public abstract void GetWomanConclusion(Woman concreteElementB);
}
abstract class Person
{
    //接受
    public abstract void Accept(Action visitor);
}
```

它是用来获得‘状态’对象的

“这里关键就在于人就只分为男人和女人，这个性别的分类是稳定的，所以可以在状态类中，增加‘男人反应’和‘女人反应’两个方法，方法个数是稳定的，不会很容易的发生变化。而‘人’抽象类中有一个抽象方法‘接受’，它是用来获得‘状态’对象的。每一种具体状态都继承‘状态’抽象类，实现两个反应的方法。”

具体“状态”类

```
//成功
class Success : Action
{
```

```
public override void GetManConclusion(Man concreteElementA)
{
    Console.WriteLine("{0}{1}时，背后多半有一个伟大的女人。",
                    concreteElementA.GetType().Name, this.GetType().Name);
}

public override void GetWomanConclusion(Woman concreteElementB)
{
    Console.WriteLine("{0}{1}时，背后大多有一个不成功的男人。",
                    concreteElementB.GetType().Name, this.GetType().Name);
    }
}
//失败
class Failing : Action
{
    //与上面代码类同，省略
}
//恋爱
class Amativeness : Action
{
    //与上面代码类同，省略
}
```

“男人”类和“女人”类

```
//男人
class Man : Person
{
    public override void Accept(Action visitor)
    {
        visitor.GetManConclusion(this);
    }
}

//女人
class Woman : Person
{
    public override void Accept(Action visitor)
    {
        visitor.GetWomanConclusion(this);
    }
}
```

首先在客户程序中将具体状态作为参数传递给“男人”类完成了一次分派，然后“男人”类调用作为参数的“具体状态”中的方法“男人反应”，同时将自己（this）作为参数传递进去。这便完成了第二次分派

“这里需要提一下当中用到一种双分派的技术，首先在客户程序中将具体状态作为参数传递给“男

人”类完成了一次分派，然后“男人”类调用作为参数的“具体状态”中的方法“男人反应”，同时将自己（this）作为参数传递进去。这便完成了第二次分派。双分派意味着得到执行的操作决定于请求的种类和两个接收者的类型。‘接受’方法就是一个双分派的操作，它得到执行的操作不仅决定于‘状态’类的具体状态，还决定于它访问的‘人’的类别。”

对象结构类 由于总是需要‘男人’与‘女人’在不同状态的对比，所以我们需要一个‘对象结构’类来针对不同的‘状态’遍历‘男人’与‘女人’，得到不同的反应。

```
//对象结构
class ObjectStructure
{
    private IList<Person> elements = new List<Person>();

    //增加
    public void Attach(Person element)
    {
        elements.Add(element);
    }
    //移除
    public void Detach(Person element)
    {
        elements.Remove(element);
    }
    //查看显示
    public void Display(Action visitor)
    {
        foreach (Person e in elements)
        {
            e.Accept(visitor);
        }
    }
}
```

遍历方法

客户端代码

```
static void Main(string[] args)
{
    ObjectStructure o = new ObjectStructure();
    o.Attach(new Man());
    o.Attach(new Woman());

    //成功时的反应
    Success v1 = new Success();
    o.Display(v1);
```

在对象结构中加入要对比的“男人”和“女人”

查看在各种状态下，“男人”和“女人”的反应

```
    //失败时的反应
    Failing v2 = new Failing();
    o.Display(v2);

    //恋爱时的反应
    Amativeness v3 = new Amativeness();
    o.Display(v3);

    Console.Read();
}
```

“这样做到底有什么好处呢？”小菜问道。

“你仔细看看，现在这样做，就意味着，如果我们现在要增加‘结婚’的状态来考查‘男人’和‘女人’的反应。只需要怎么就可以了？”

“哦，我明白你意思了，由于用了双分派，使得我只需要增加一个‘状态’子类，就可以在客户端调用来查看，不需要改动其他任何类的代码。”

“来，写出来试试看。”

结婚状态类

```
class Marriage : Action
{
    public override void GetManConclusion(Man concreteElementA)
    {
        Console.WriteLine("{0}{1}时，感慨道：恋爱游戏终结时，‘有妻徒刑’遥无期。",
                          concreteElementA.GetType().Name, this.GetType().Name);
    }

    public override void GetWomanConclusion(Woman concreteElementB)
    {
        Console.WriteLine("{0}{1}时，欣慰曰：爱情长跑路漫漫，婚姻保险保平安。",
                          concreteElementB.GetType().Name, this.GetType().Name);
    }
}
```

客户端代码，增加下面一段代码就可以完成

```
.....
Marriage v4 = new Marriage();
o.Display(v4);
.....
```

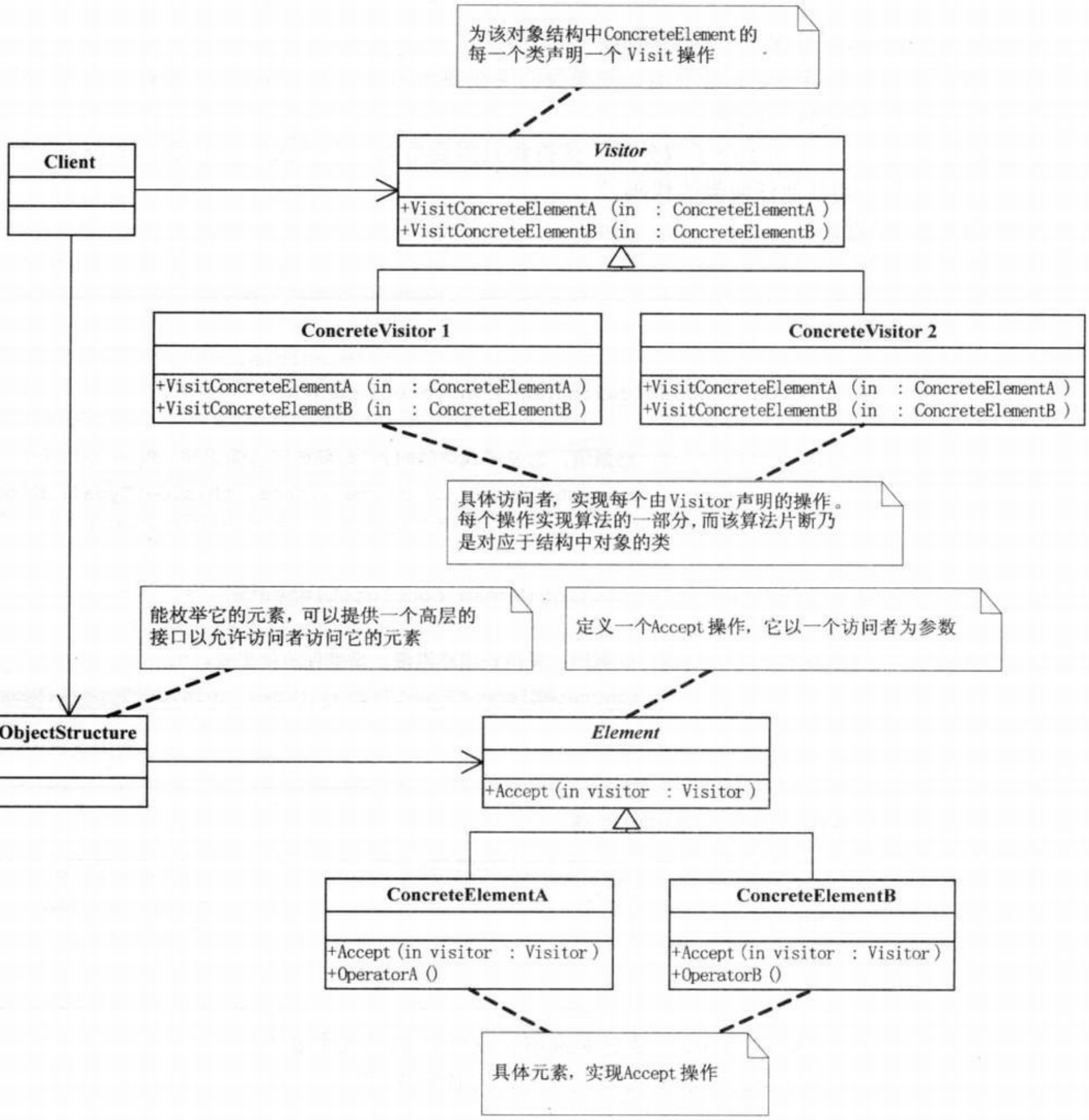
“哈，完美的体现了开放-封闭原则，实在是高呀。这叫什么模式来着？”

“它应该算是 GoF 中最复杂的一个模式了，叫做访问者模式。”

28.5 访问者模式

访问者模式（Visitor），表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。[DP]

访问者模式（Visitor）结构图



“在这里，Element 就是我们的‘人’类，而 ConcreteElementA 和 ConcreteElementB 就是‘男人’和‘女人’，Visitor 就是我们写的‘状态’类，具体的 ConcreteVisitor 就是那些‘成功’、‘失败’、‘恋爱’等等状态。至于 ObjectStructure 就是‘对象结构’类了。”

“哦，怪不得这幅类图我感觉和刚才写的代码类图几乎可以完全对应。”

“本来我是想直接来谈访问者模式的，但是为什么我突然会愿意和你聊男人和女人的对比呢，原因就在于你说了一句话：‘男女对比这么多的原因是因为人类在性别上就只有男人和女人两类。’而这也正是访问者模式可以实施的前提。”

“这个前提是什么呢？”

“你想呀，如果人类的性别不止是男和女，而是可有多种性别，那就意味‘状态’类中的抽象方法就不可能稳定了，每加一种类别，就需要在状态类和它的所有下属类中都增加一个方法，这就不符合开放-封闭原则。”

“哦，也就是说，**访问者模式适用于数据结构相对稳定的系统**？”

“对的，**它把数据结构和作用于结构上的操作之间的耦合解脱开，使得操作集合可以相对自由地演化。**”

“访问者模式的目的是什么？”小菜问道。

“**访问者模式的目的是要把处理从数据结构分离出来。**很多系统可以按照算法和数据结构分开，如果这样的系统**有比较稳定的数据结构，又有易于变化的算法的话，使用访问者模式就是比较合适的，因为访问者模式使得算法操作的增加变得容易。**反之，如果这样的系统的数据结构对象易于变化，经常要有新的数据对象增加进来，就不适合使用访问者模式。”

“那其实**访问者模式的优点就是增加新的操作很容易，因为增加新的操作就意味着增加一个新的访问者。访问者模式将有关的行为集中到一个访问者对象中。**”

“是的，总结得很好。”大鸟接着说，“通常 ConcreteVisitor 可以单独开发，不必跟 ConcreteElementA 或 ConcreteElementB 写在一起。正因为这样，ConcreteVisitor 能提高 ConcreteElement 之间的独立性，如果把一个处理动作设计成 ConcreteElementA 和 ConcreteElementB 类的方法，每次想新增“处理”以扩充功能时就得去修改 ConcreteElementA 和 ConcreteElementB 了。这也就是你之前写的代码，在‘男人’和‘女人’类中加了对‘成功’、‘失败’等状态的判断，造成处理方法和数据结构的紧耦合。”

“**那访问者的缺点其实也就是使增加新的数据结构变得困难了。**”

“所以 GoF 四人中的一个作者就说过：‘大多时候你并不需要访问者模式，但当一旦你需要访问者模式时，那就是真的需要它了。’事实上，我们很难找到数据结构不变化的情况，所以用访问者模式的机会也就不太多了。这也就是为什么你谈到男人女人对比时我很高兴和你讨论的原因，因为人类性别这样的数据结构是不会变化的。”

“哈，看来是我为你找到了一个好的教学样例。”小菜得意道。

“和往常一样，我们需要书写一些基本的代码来巩固我们的学习。有了 UML 的类图，相信你应该没什么问题了。”

“OK。”

28.6　访问者模式基本代码

Visitor 类，为该对象结构中 ConcreteElement 的每一个类声明一个 Visit 操作。

```
abstract class Visitor
{
    public abstract void VisitConcreteElementA(ConcreteElementA concreteElementA);

    public abstract void VisitConcreteElementB(ConcreteElementB concreteElementB);
}
```

ConcreteVisitor1 和 ConcreteVisitor2 类，具体访问者，实现每个由 Visitor 声明的操作。每个操作实现算法的一部分，而该算法片断乃是对应于结构中对象的类。

```
class ConcreteVisitor1 : Visitor
{
    public override void VisitConcreteElementA(ConcreteElementA concreteElementA)
    {
        Console.WriteLine("{0}被{1}访问",
                    concreteElementA.GetType().Name, this.GetType().Name);
    }

    public override void VisitConcreteElementB(ConcreteElementB concreteElementB)
    {
        Console.WriteLine("{0}被{1}访问",
                    concreteElementB.GetType().Name, this.GetType().Name);
    }
}

class ConcreteVisitor2 : Visitor
{
    //代码与上类类似，省略
}
```

Element 类，定义一个 Accept 操作，它以一个访问者为参数。

```
abstract class Element
{
    public abstract void Accept(Visitor visitor);
}
```

ConcreteElementA 和 ConcreteElementB 类，具体元素，实现 Accept 操作。

```
class ConcreteElementA : Element
{
    public override void Accept(Visitor visitor)
    {
        visitor.VisitConcreteElementA(this);
    }
```

充分利用双分派技术，实现处理与数据结构的分离

```
        public void OperationA()
    { }
}

class ConcreteElementB : Element
{
    public override void Accept(Visitor visitor)
    {
        visitor.VisitConcreteElementB(this);
    }

    public void OperationB()
    { }
}
```

其他的相关方法

ObjectStructure 类，能枚举它的元素，可以提供一个高层的接口以允许访问者访问它的元素。

```
class ObjectStructure
{
    private IList<Element> elements = new List<Element>();

    public void Attach(Element element)
    {
        elements.Add(element);
    }
    public void Detach(Element element)
    {
        elements.Remove(element);
    }
    public void Accept(Visitor visitor)
    {
        foreach (Element e in elements)
        {
            e.Accept(visitor);
        }
    }
}
```

客户端代码

```
static void Main(string[] args)
{
    ObjectStructure o = new ObjectStructure();
    o.Attach(new ConcreteElementA());
    o.Attach(new ConcreteElementB());
```

```
    ConcreteVisitor1 v1 = new ConcreteVisitor1();
    ConcreteVisitor2 v2 = new ConcreteVisitor2();

    o.Accept(v1);
    o.Accept(v2);

    Console.Read();
}
```

28.7 比上不足，比下有余

“啊，访问者模式比较麻烦哦。”

“是的，访问者模式的能力和复杂性是把双刃剑，只有当你真正需要它的时候，才考虑使用它。有很多的程序员为了展示自己的面向对象的能力或是沉迷于模式当中，往往会误用这个模式，所以一定要好好理解它的适用性。”

“哈，大鸟太高估了我们这些菜鸟程序员了。你说得是没错，不过我估计大多数人不去用它的原因绝不是因为怕误用，而是因为它太过于复杂和晦涩，根本不能理解，不熟悉的东西当然就不会想着去应用它了。”

“对，如果不理解，实在是不可能会想到用访问者模式的。”

“不管男人女人，不懂也要装懂的多得是了。”小菜说道，“这其实不能算是男人的专利。”

“你看的那些所谓的男人和女人的对比，都是不准确的，我给个答案吧，男人与女人最大的区别就是，比上不足，比下有余。”

“啊？！”