## 第26章 尊重个体差异，提供个性化服务——访问者模式

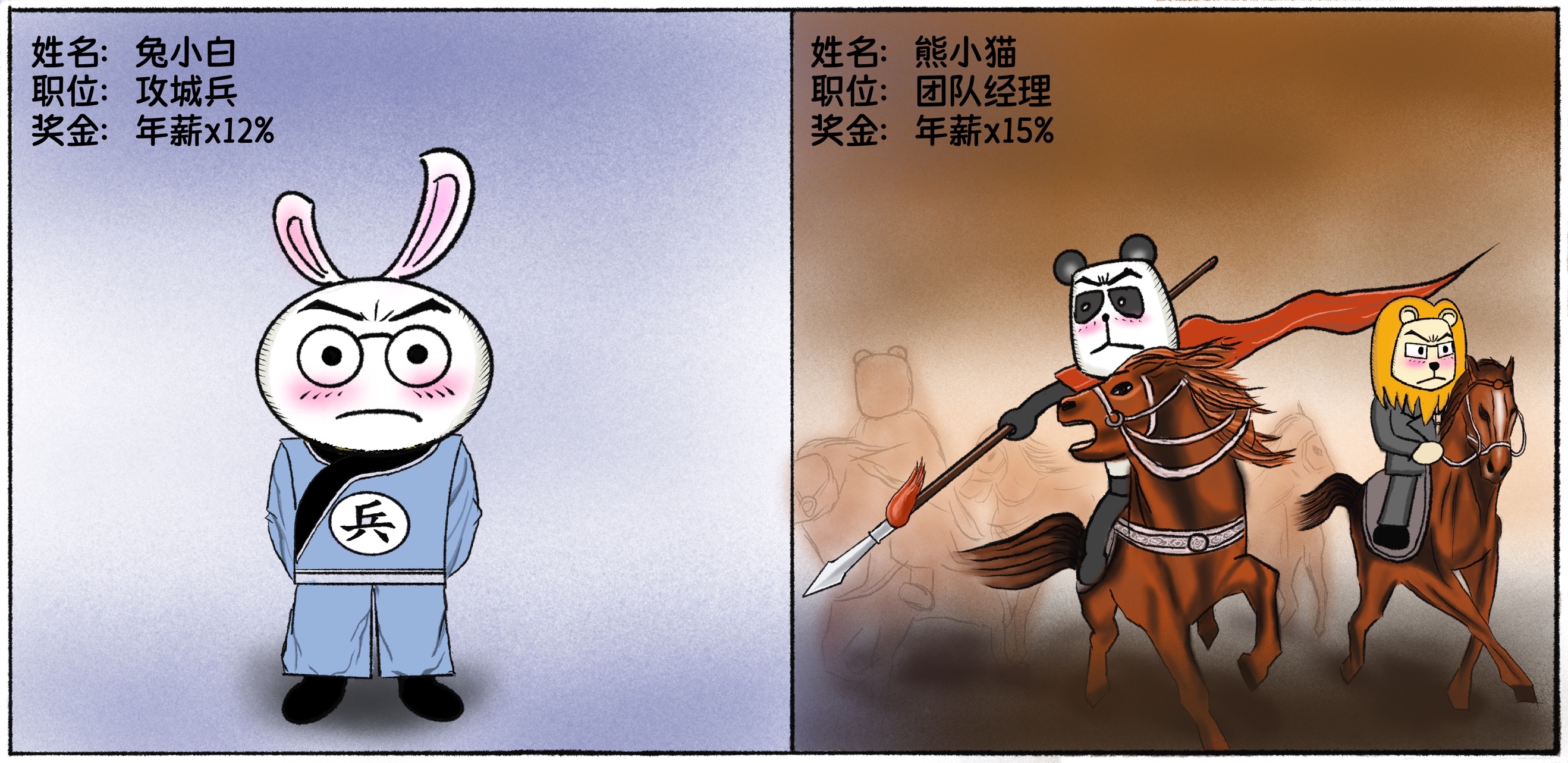
## 26.1 员工的奖金算法居然不一样

熊小猫：这个月就要发年终奖了!辛苦了一年，准备怎么奖励一下自己？

兔小白：这是我工作以来第一次拿年终奖，想想还有点兴奋！不过我的工资基数低，年终奖也不会太多。我计划过年回家给父母买点礼物，剩下的钱先存起来。你呢，是不是已经计划好了？

熊小猫：我也没多少奖金，先存起来吧，花钱的地方太多了。

兔小白：你可别谦虚了！你工作了这么多年，工资基数大。而且我听说普通员工按年薪的12%发奖金，但管理人员按年薪的15%发奖金。你的奖金一定少不了！



熊小猫：管理人员的薪酬构成中，奖金的比例更大，这是为了刺激管理层协助公司取得更高的业绩。但是万一公司业绩不达标，导致奖金被打了折扣，管理人员的损失也会更大。

兔小白：今年公司的业绩还不错，你应该可以全额拿到奖金。太羡慕你了！

熊小猫：只要你脚踏实地，学好技术，早晚会超过我。咱们学习了这么久设计模式，只差最后一个就学完了。等把设计模式全部攻克，咱们继续学点新知识。

兔小白：今天就把最后一个设计模式学完吧！我期待能早日升职加薪呢！

熊小猫：没问题，最后一个设计模式叫做访问者模式。计算奖金这个场景正好适合使用访问者模式。

## 26.2 循规蹈矩的程序实现

熊小猫：老规矩，先从练习开始，咱们来编写奖金计算程序。首先，假设员工拿到全额奖金。普通员工的奖金是年薪的12%，管理人员的奖金是年薪的15%。奖金计算程序要解决的核心问题是，同样都为员工，但计算奖金的算法不同。你来想想程序怎么实现。

兔小白：这没什么复杂的，等我10分钟。

10分钟后，兔小白写完了代码。

兔小白：我先创建了抽象员工类Employee，包含姓名和薪资属性，还定义了计算奖金的方法calculateBonus。

public abstract class Employee {  
    private int salary;  
    private String name;  
​  
    public Employee(int salary, String name) {  
        this.salary = salary;  
        this.name = name;  
   }  
​  
    public int getSalary() {  
        return salary;  
   }  
​  
    public String getName() {  
        return name;  
   }  
​  
    public abstract double calculateBonus();  
}

Employee有两个子类，普通员工类和管理人员类。普通员工类OrdinaryEmployee的calculateBonus方法按照年薪的12%计算奖金。

public class OrdinaryEmployee extends Employee {  
    public OrdinaryEmployee(int salary, String name) {  
        super(salary, name);  
   }  
​  
    public double calculateBonus() {  
        return getSalary() \* 12 \* 0.12;  
   }  
}

管理人员类Manager的奖金按照年薪的15%计算奖金。

public class Manager extends Employee {  
    public Manager(int salary, String name) {  
        super(salary, name);  
   }  
​  
    public double calculateBonus() {  
        return getSalary() \* 12 \* 0.15;  
   }  
}

客户端代码，首先构建了一个Employee类型对象的列表，里面存放的既有OrdinaryEmployee对象，也有Manager对象。然后迭代计算每个Employee的奖金。

List<Employee> employees = new ArrayList<>();  
​  
Employee rabbit = new OrdinaryEmployee(8000, "兔小白");  
Employee panda = new Manager(10000, "熊小猫");  
employees.add(rabbit);  
employees.add(panda);  
​  
for (Employee employee : employees) {  
  System.out.println(employee.getName() + " 奖金：" + employee.calculateBonus());  
}

程序输出如下。普通员工和管理人员分别按照各自的比例计算奖金。

兔小白 奖金：11520.0  
熊小猫 奖金：18000.0

熊小猫：你的程序是比较常见的设计方式，子类实现父类的抽象方法。这样设计虽然不会出什么差错，但不一定是最优的设计方式。现在我要增加新的需求来验验你程序的成色！

兔小白：没问题，你说吧，我对自己写的程序有信心！

熊小猫：我要增加新的薪资项计算——电话费津贴。普通员工每月100元，管理人员每月300元。

兔小白：你不按套路出牌呀！我以为你要增加一种新的员工类型，采用新的奖金计算方式呢。如果是这样，我增加Employee的子类就可以了，完全符合开闭原则。

熊小猫：这样就没意思啦！这次，我要增加的是新操作。

兔小白：这样的话，只能先给Employee增加计算电话费津贴的抽象方法，就叫做getTelephoneAllowance吧。然后它的两个子类分别实现该方法。

熊小猫：如果这样修改，会改动Employee继承体系中所有的类，严重违反了开闭原则。

兔小白：没有办法呀！想要增加类的行为，只能修改类。

熊小猫：我有办法不修改类，也能增加它的行为。

兔小白：哦？这么神奇？

## 26.3可扩展操作的程序实现

熊小猫：你的设计角度是从行为的发出者入手。所以增加行为，必然修改行为的发出者。我们换个角度，从行为出发，将行为抽象成对象。行为对象针对不同的发出者，执行的逻辑也不同。你来尝试将计算奖金的行为抽象为对象，提供个性化服务。它根据计算的对象不同，执行的逻辑也不同。

兔小白：我觉得可以这样修改 ，先定义SalaryCalculator接口，针对不同的计算对象类型，定义2个重载的calculate方法。

public interface SalaryCalculator {  
    double calculate(OrdinaryEmployee employee);  
​  
    double calculate(Manager employee);  
}

再增加奖金计算类BonusCalculator，实现该接口。BonusCalculator分别根据普通员工和管理人员计算奖金的算法，实现这2个重载的calculate方法。

public class BonusCalculator implements SalaryCalculator {  
    public double calculate(OrdinaryEmployee employee) {  
        return employee.getSalary() \* 12 \* 0.12;  
   }  
​   
    public double calculate(Manager employee) {  
        return employee.getSalary() \* 12 \* 0.15;  
   }  
}

熊小猫：很棒！你已经完成了改造的第一步，Calculator支持扩展，也就是说做到了操作可扩展。接下来需要让Employee使用Calculator完成薪资项计算。可以在Employee中定义一个accept方法，让它接收Calculator类型对象，然后使用该对象完成操作。这样的话，accept方法执行的操作取决于传入的Calculator类型。

按照这个思路，我为Employee抽象类增加accept抽象方法。

public abstract class Employee {  
    private final int salary;  
    private final String name;  
​  
    public Employee(int salary, String name) {  
        this.salary = salary;  
        this.name = name;  
   }  
​  
    public int getSalary() {  
        return salary;  
   }  
​  
    public String getName() {  
        return name;  
   }  
​  
    public abstract double accept(SalaryCalculator calculator);  
}

OrdinaryEmployee和Manager对accept方法的实现几乎一样，都是调用BonusCalculator的calculate方法，将自己作为参数传入。但由于自身类型不同，调用的是不同的calculate重载方法。

OrdinaryEmployee代码修改如下。

public class OrdinaryEmployee extends Employee {  
    public OrdinaryEmployee(int salary, String name) {  
        super(salary, name);  
   }  
​  
    public double accept(SalaryCalculator calculator) {  
        return calculator.calculate(this);  
   }  
}

Manager代码修改如下。

public class Manager extends Employee {  
    public Manager(int salary, String name) {  
        super(salary, name);  
   }  
​  
    public double accept(SalaryCalculator calculator) {  
        return calculator.calculate(this);  
   }  
}

现在的程序结构使得Employee可以使用accept方法接收各种操作实现的Calculator子类对象。我们可以为计算电话费津贴定义一个新的Calculator子类TelephoneAllowanceCalculator。

public class TelephoneAllowanceCalculator implements SalaryCalculator {  
    public double calculate(OrdinaryEmployee employee) {  
        return 100;  
   }  
​  
    public double calculate(Manager employee) {  
        return 300;  
   }  
}

客户端代码中，在调用Employee的accept方法时，可以通过改变入参Calculator对象的类型，来改变Employee的行为。

List<Employee> employees = new ArrayList<>();  
​  
Employee rabbit = new OrdinaryEmployee(8000, "兔小白");  
Employee panda = new Manager(10000, "熊小猫");  
employees.add(rabbit);  
employees.add(panda);  
​  
for (Employee employee : employees) {  
  System.out.println(employee.getName() + " 奖金："  
                     + employee.accept(new BonusCalculator()) + " 电话费津贴："  
                     + employee.accept(new TelephoneAllowanceCalculator()));  
}

程序输出符合预期。

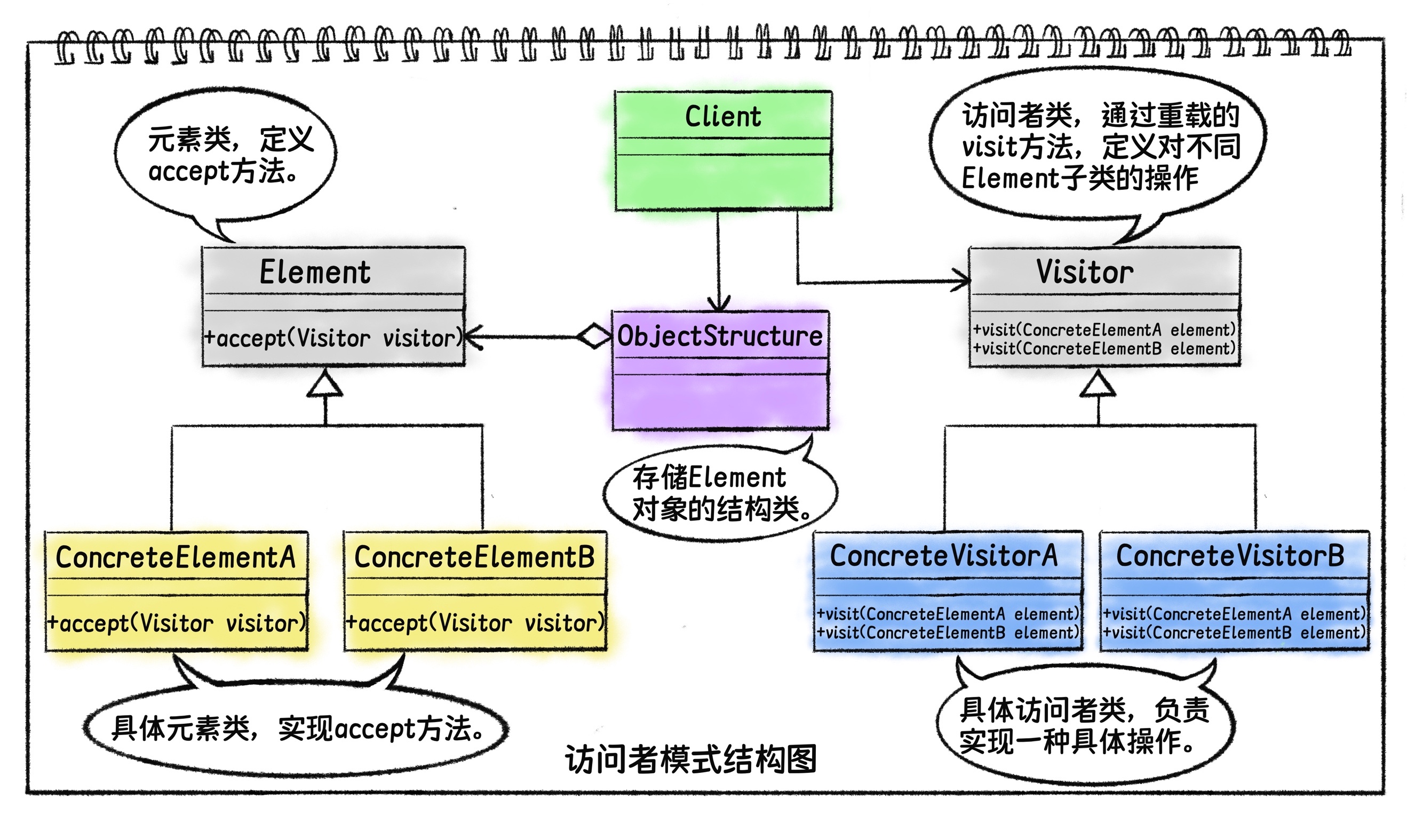
兔小白 奖金：11520.0 电话费津贴：100.0  
熊小猫 奖金：18000.0 电话费津贴：300.0

兔小白：Employee的accept方法很关键，这是一个“万能”方法，它可以接收任何Calculator类型的对象，然后用该对象完成特定的操作。

熊小猫：accept方法使用的是双分派技术。它所执行的操作取决于两个类型，一是Calculator的类型，二是Employee的类型。下面我们来详细分析访问者模式。

## 26.4 访问者模式的优缺点及适用场景

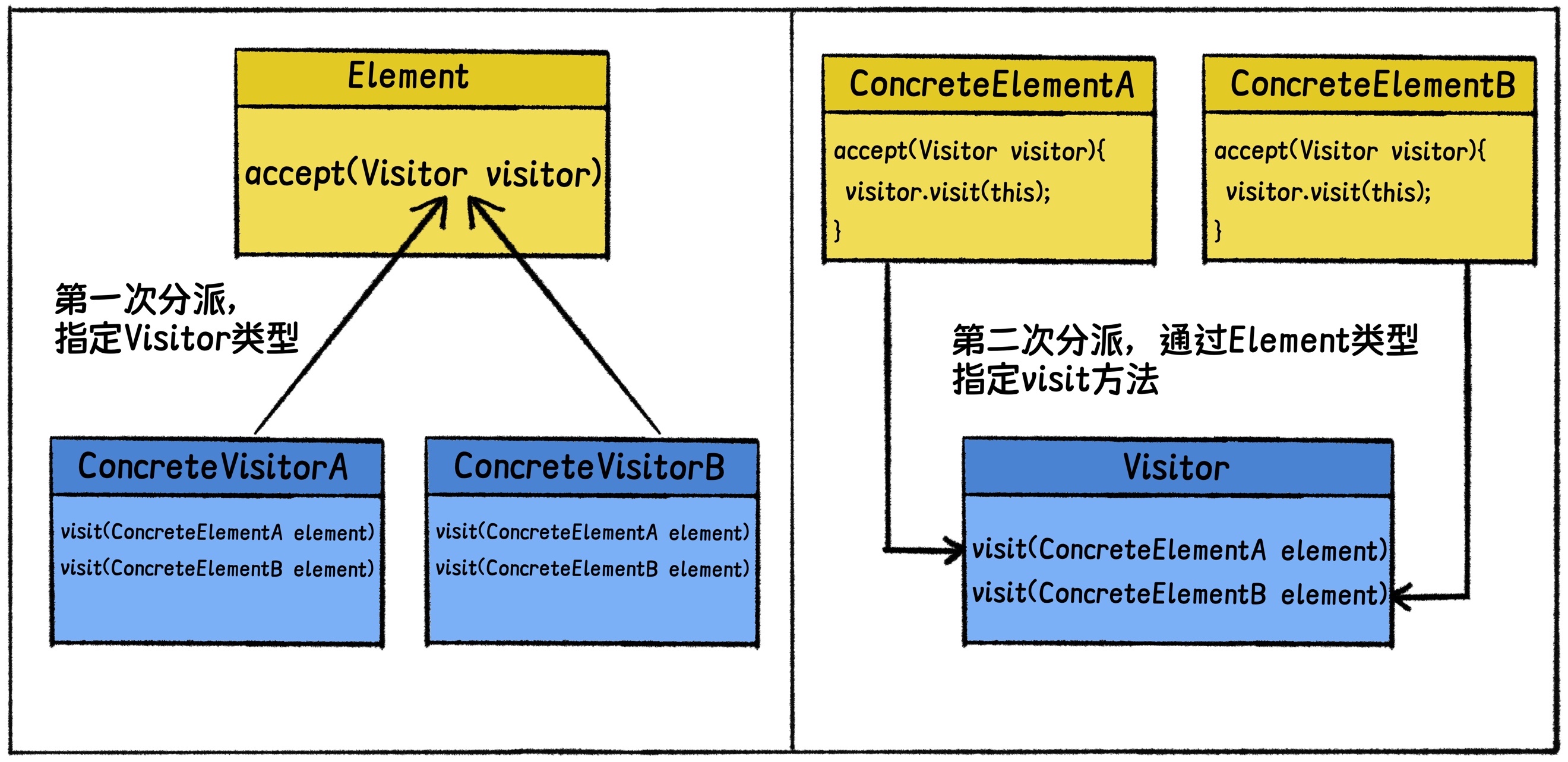
熊小猫：我们先来看看访问者模式的结构图。



访问者模式中有两套继承体系。一套是Visitor，对应练习中的Calculator。另外一套是Element，对应练习中的Employee。

Visitor是访问者类，通过重载的visit方法，定义针对不同Element子类的操作。ConcreteVisitor是具体访问者类，负责实现一种具体操作。ConcreteVisitor将各个Element子类的操作逻辑，分散在visitElement重载方法中实现。访问者模式的名字便来自于这种实现方式，可以理解为Visitor访问的Element对象类型不同，对该Element对象执行的操作逻辑也不同。

Element是元素类，它定义了accept方法。ConcreteElement是具体元素类，实现accept方法。accept方法接收一个Visitor类型的参数。Visitor的类型决定accept方法执行哪种操作，操作的具体逻辑又是由ConcreteElement类型所决定。这就是我之前说到的双分派技术。



访问者模式通过双分派技术分离了元素和操作的，不修改元素，也可以为元素添加操作。

ObjectStructure是一个存储Element对象的结构类，可以是树状结构，也可以是列表等集合。它提供一个能够访问每个Element的方法。ObjectStructure对应练习中的Employee列表。

此外，Visitor提供的visit方法可以接收任何类型的参数，也就是说它可以访问不具有相同父类的对象。[[1]](#footnote-1)1因此，ObjectStructure中存储的Element对象不一定具有相同的父类。

访问者模式分离了元素和作用于元素的操作，不改变元素，也可以定义作用于元素的新操作。它具备下面这些优点。

1. 对元素的操作可扩展。这是访问者模式的核心优势，增加新的操作，只需要增加新的访问者。
2. 集中同类型操作。将同类型的操作集中在同一个访问者类中，而不是分散在各个元素类中。这样有利于程序的维护。
3. 符合单一职责原则。元素类负责数据维护，每个访问者类只负责对元素的一种操作，职责明确且单一。

兔小白：访问者模式的优点确实很多，但我觉得它对元素类的扩展支持并不好。

熊小猫：你说的没错，鱼和熊掌不能兼得，访问者模式获得了操作的扩展性，但这是以牺牲元素的扩展性为代价。增加新的元素类，需要为每一个访问者子类增加匹配新元素类型的visit方法，供新的元素类调用。此外，访问者类执行操作时，常常需要了解元素类的内部属性，这在一定程度上破坏了元素类的封装。

为了扬长避短，适合使用访问者模式的场景，应具备如下特点。

1. 元素类只负责数据的维护，不提供操作。
2. 元素类的种类固定。由于增加或减少元素类，会影响所有已经存在的访问者类。因此需要稳定的元素类种类。
3. 对元素的操作存在变化的可能。元素操作类型不稳定的场景，访问者模式才能发挥它易于扩展操作的优势。

兔小白：你常说软件设计要贴近现实世界。我觉得访问者模式更适合Element对象天然仅封装数据，而不应刻意塑造。如果某个场景中，想要抽象为Element的对象并不是单纯封装数据，那么强行将它的行为分离成Visitor类，并不可取。

熊小猫：设计模式的运用要自然，不要刻意。如果问题还没彻底分析清楚，就先选定一种设计模式，那么程序设计和开发就会被框定在这种设计模式之中，刻意往上靠。最终写出程序不但难以理解，而且也无法发挥出设计模式的优势。

到今天为止，我们学完了所有23种设计模式。但这只是刚刚上路，学懂不等于会用。今后的工作中多思考、多实践，才能融会贯通，运用自如。加油吧，兔小白！

兔小白：感谢熊老师的辛勤付出，我收获颇丰！明天也终于可以学习新知识了！

熊小猫：哈哈，明天还不行。咱们最后再花点时间来总结设计模式，不能稀里糊涂的就学完了。

1. 1 《设计模式》书中的访问者模式结构图没有体现这一点，但在对访问者模式的效果部分讲解了这一特点。 [↑](#footnote-ref-1)