## 第11章 数据结构中，也有设计模式——组合模式

## 11.1人力地图中的设计模式

熊小猫：兔小白，你一定在摸鱼吧？对着屏幕笑什么呢？

兔小白：咱们公司最近调整了组织架构。今天系统中的人力地图更新了，我刚才发现我和虎老板之间的层级减少了一层。

熊小猫：调整组织架构不就是为了精简，减少不必要的层级吗？这有什么稀奇的。

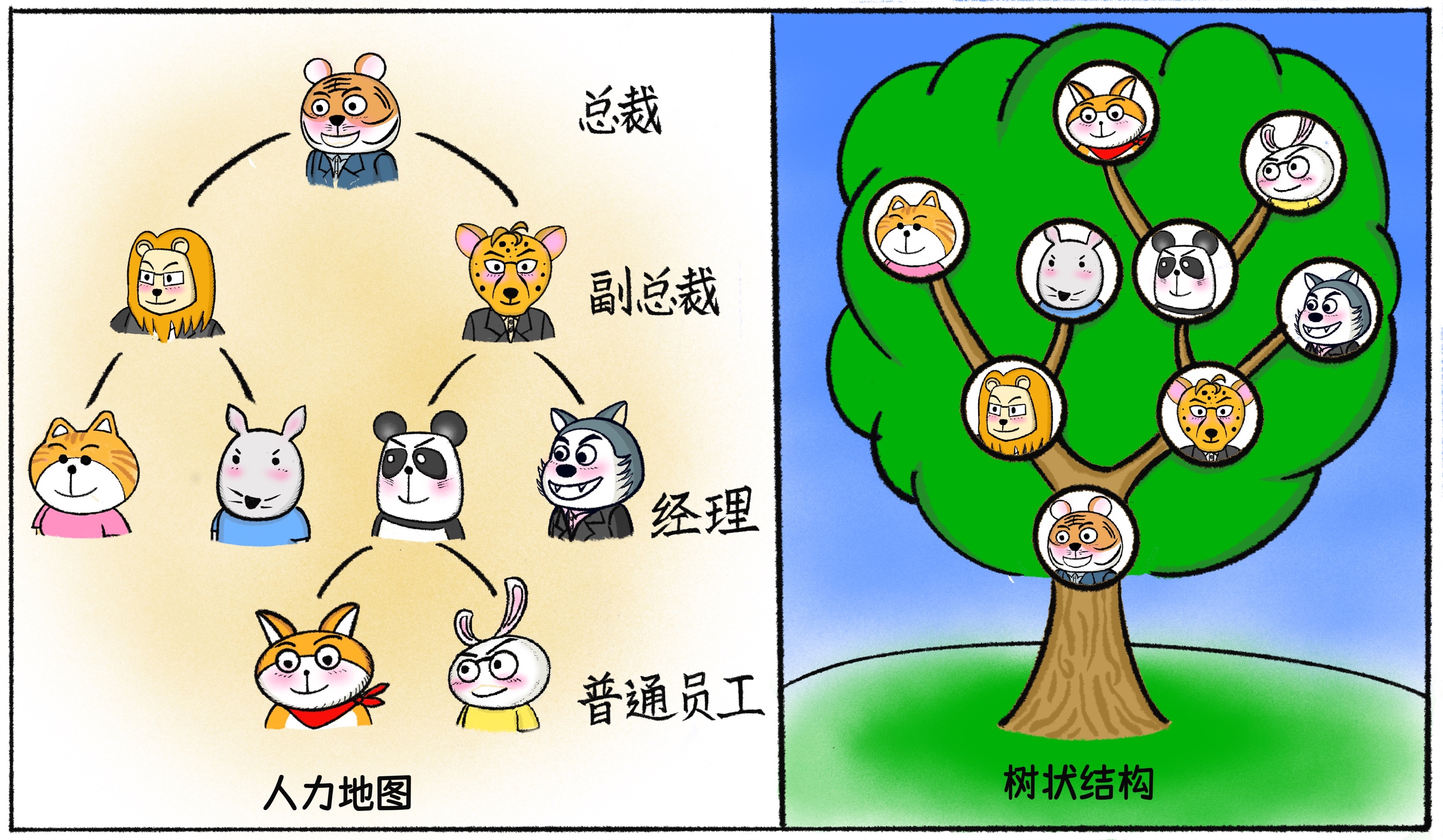
兔小白：我距离老板又近了一级，相当于晋升了呀，所以我很开心！

熊小猫：这算哪门子晋升？晋升得靠你真正的实力。我现在考考你？

兔小白：我只是和你开个玩笑，你怎么还当真考起我来了？既然要考我，我当然也不怕，放马过来吧！

熊小猫：这么自信？那就请听题，人力地图采用了什么数据结构来实现？

兔小白：人力地图是典型的树状结构啊！总裁只有一个， 总裁下面有三个副总裁。每位副总裁下面又有若干名经理。总裁、副总裁、经理都是公司员工，只是级别不同。员工之间的关系为一对多，如果画图表示出来，就像一颗大树的树冠倒转过来，所以称之为树状结构。



熊小猫：没想到你对算法和数据结构还有研究！

兔小白：你今天不是要考我算法吧？

熊小猫：哈哈，被你猜对了！算法先放一放，咱们先集中精力把设计模式学习完。

兔小白：你的意思是树状结构也是一种设计模式？

熊小猫：数据之间构成树状结构，是一种数据结构。对象之间构成树状结构，则是一种设计模式。这种设计模式叫做组合模式。

面向对象语言在实现树结构时，树上的每个节点都会被抽象为对象。比如人力地图中的节点是员工对象。员工对象之间存在一对多的聚合关系。员工通过聚合关系连接在一起，形成一棵树，就是人力地图。

兔小白：说到代码实现我就开始手痒痒！还是动手写一写吧！

熊小猫：哎呀，我刚讲一半。不过也好，咱们可以边写边讲。

## 11.2 只有内部员工的人力地图

熊小猫：看你跃跃欲试的样子，咱们马上开始练习。这次要实现的需求就是人力地图。每个节点的员工可以添加和移除自己的下级员工，并且可以遍历展示这个员工所有下属员工的姓名。

兔小白：等我十分钟，我马上就写一版人力地图。

十分钟后，兔小白完成了人力地图的程序。

兔小白：人力地图中虽然只有员工对象，但还是需要面向接口编程，以防你又提需求变更。我先创建了员工抽象类Staff，定义了维护下属员工的方法和展示下属员工姓名的方法。

public abstract class Staff {  
    protected String name;  
​  
    public Staff(String name){  
        this.name = name;  
   }  
    abstract public void  addSubordinate(Staff staff );  
    abstract public void removeSubordinate(Staff staff);  
    abstract public void display(int myDepth);  
}

员工类Employee继承自Staff类。成员变量subordinates是Employee类型对象的列表，用来保存下属员工。removeSubordinate和addSubordinate方法用来维护直属的下属员工。display方法通过递归展示所有下属员工的名字。

public class Employee extends Staff {  
    private ArrayList<Staff> subordinates = new ArrayList();  
​  
    public Employee(String name) {  
        super(name);  
   }  
​  
    public void addSubordinate(Staff staff) {  
        subordinates.add(staff);  
   }  
​  
    public void removeSubordinate(Staff staff) {  
        subordinates.remove(staff);  
   }  
​  
    public void display(int myDepth) {  
        for (int i = 0; i < myDepth; i++) {  
            System.out.print("-");  
       }  
        System.out.println(name);  
​  
        for (Staff subordinate:subordinates) {  
            subordinate.display(myDepth+1);  
       }  
   }  
}

熊小猫：学聪明了呀！这么简单的程序还记得面向接口编程。

兔小白：我记得你说过，程序是否需要更为灵活的设计，取决于成本和未来变化的可能性。我既然知道你一定会提需求变更，而且加抽象类也不费事，我当然就提前做了。不然一会还要改。

熊小猫：不错，不错！需求确实要变更，刚才我已经说过，我才讲了一半。现在人力地图的需求确实比较简单，你实现的也没有问题。咱们公司除正式员工外，还有一些合作单位派遣的外协员工，现在人力地图中要加入外协员工。首先为了方便管理，外协员工也会从属于某个正式员工。但是我们公司并不需要管理外协员工内部的层级关系，因此在人力地图中，外协员工不会有下级。

兔小白：外协员工有点特殊，他的行为和我定义的staff抽象类不一样呀。我得再理理对象间的关系。

## 11.3 外协员工也要一视同仁

十分钟后，兔小白还没写完程序。

兔小白：有个问题给我难住了。由于外协员工没有下属员工，所以外协员工类不应该有subordinates列表以及相关的维护方法。那么，Staff抽象类中的相关方法也要移除掉。如果这样设计，员工类会和外协员工类的行为差别很大，这会导致客户端代码在使用Staff类型的对象前，需要先判断是内部员工还是外协员工，才能知道有哪些方法可以调用。

熊小猫：外协员工没有下属员工，是树中的叶子结点。不过也可以让外协员工具备subordinates的维护方法，只不过实现逻辑为空。这样客户端就不用刻意区分外协员工和内部员工了。

兔小白：可以这样设计吗？我觉得不合理呀！

熊小猫：你可以把外协员工看作是一种特殊的员工。他也拥有维护下属员工的行为，但是因为他不能拥有下属员工，所以什么都不需要做。从这个角度来考虑，是不是就可以说服你了？

兔小白：好吧，虽然有些牵强，我勉强接受。

熊小猫：有的时候软件设计也需要变通。快去改代码吧！

十五分钟后，兔小白修改完了代码。

兔小白：按照你说的，改起来就简单多了，外协员工类OutsourcedStaff也继承自Staff类。虽实现了维护subordinates列表的方法，但实际上没有做任何事情。由于没有下属，所以display方法只打印自己的姓名。

public class OutsourcedStaff extends Staff{  
    public OutsourcedStaff(String name) {  
        super(name);  
   }  
​  
    public void addSubordinate(Staff staff) {  
   }  
​  
    public void removeSubordinate(Staff staff) {  
   }  
​  
    public void display(int myDepth) {  
        for (int i = 0; i < myDepth; i++) {  
            System.out.print("-");  
       }  
        System.out.println(name);  
   }  
}

客户端代码如下。

public class Client {  
    public static void main(String[] args) {  
        Employee tiger = new Employee("虎老板");  
        Employee lion = new Employee("狮哥");  
        Employee leopard = new Employee("豹哥");  
        Employee cat = new Employee("猫花花");  
        Employee panda = new Employee("熊小猫");  
        Employee wolf = new Employee("狼总监");  
        Employee rabbit = new Employee("兔小白");  
        Employee dog = new Employee("汪小黄");  
        OutsourcedStaff mouse = new OutsourcedStaff("鼠小弟");  
​  
        panda.addSubordinate(rabbit);  
        panda.addSubordinate(dog);  
        lion.addSubordinate(cat);  
        lion.addSubordinate(mouse);  
        leopard.addSubordinate(panda);  
        leopard.addSubordinate(wolf);  
        tiger.addSubordinate(lion);  
        tiger.addSubordinate(leopard);  
​  
        tiger.display(1);  
   }  
}

执行结果是这样的。

-虎老板  
--狮哥  
---猫花花  
---鼠小弟  
--豹哥  
---熊小猫  
----兔小白  
----汪小黄  
---狼总监

熊小猫：不错，这就是我想要的效果。

兔小白：但是这样处理外部员工类，我还是觉得不太妥当。

熊小猫：放心，我不会骗你......其实这也不是我发明的,这是组合模式的一种实现方式，叫做透明方式。这种实现方式将子结点的管理方法定义在抽象类中，叶子结点同样实现子结点管理的方法，只不过不做任何事情。客户端在使用节点对象的时候，不用判断是否为叶子结点，因此称之为透明方式。

这种方式存在一些弊端，会导致客户端对叶子结点做一些无意义的操作，例如增加或者删除叶子结点的子节点。透明方式的优势是以牺牲安全性为代价而获取的。

兔小白：看来我觉得不妥，并不是没有依据。

熊小猫：如果只是实现树状数据结构，结点没有任何业务含义，那么并不需要特殊处理叶子结点。如果一个结点的子节点列表为空，它就是叶子结点。但是在面向对象设计时，结点对象被赋予了业务含义。所以当外协员工类中出现对下属员工列表的维护方法时，你会觉得和需求不符，感到不妥。其实在大多数时候，并不会产生问题。

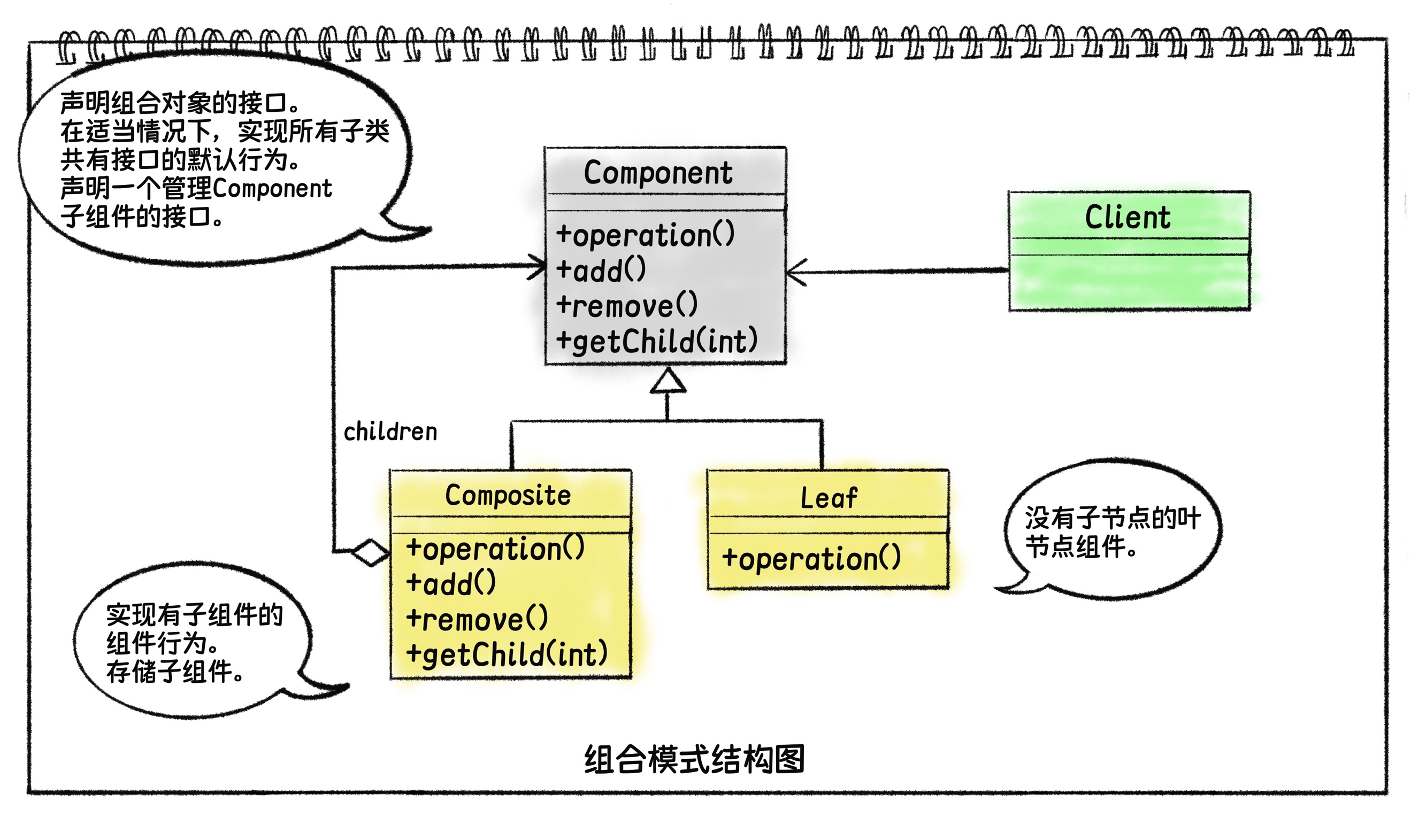
此外，还有一种安全实现方式。从面相对象的角度出发，分析对象的行为。外协员工不应该有下属员工，所以外协员工类中不会实现下属员工的维护方法。但由于其继承自Staff类，Staff类也要移除下属员工的维护方法。下属员工的维护方法只在内部员工类中实现。这样设计，客户端在使用Staff类型对象时，需要先做判断，只有Employee的实例才可以维护下属员工，丧失了透明性。

兔小白：我感觉大多数场景下透明性更重要。如果叶子结点对子节点列表维护的方法实现恰当，例如什么都不做，那么即使被调用也不会产生错误。这意味着削弱了安全性带来的问题。

熊小猫：确实如此，一般情况下，推荐使用透明方式。下面咱们看看组合模式的定义和适用场景。

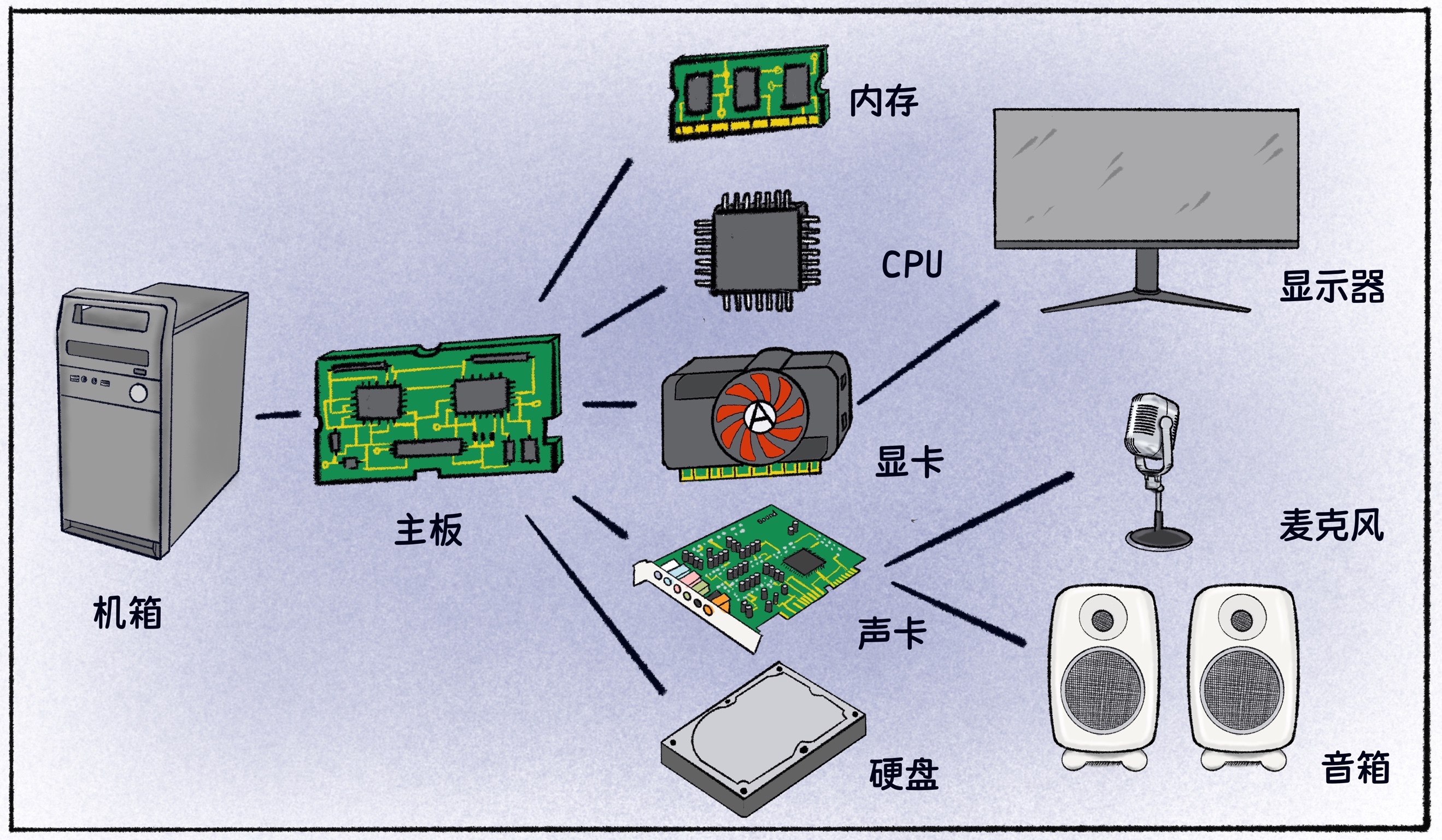
## 11.4 组合模式适用场景

熊小猫：组合模式我觉得不如叫树形模式更为形象。我以前经常把“组合”和对象关系的“合成”弄混。这两个词是同一个英文单词Composite，但其实描述的是不同内容。我们看看组合模式的结构图。



可以看到，其实组合模式中的对象关系是聚合，这是很容易搞混淆的一点。为什么叫做组合模式呢？我觉得有一种应用场景，和组合模式的名字很贴切——描述设备组装所需的结构和组件。

比如说计算机。计算机由各种组件所组成，组件间可以进行组装。有些组件不能再安装其他组件，例如内存，这种组件是Leaf类型对象。有些组件可以安装其他组件，例如主板上可以安装CPU、内存、显卡、声卡等组件。显卡上又可以安装显示器。这种组件对象是Composite类型对象。



我们来看看组合模式有哪些优点。

1. 结点变化容易。一个结点对象由若干结点对象所组成，每个结点都是一棵子树。这意味着很容易就能实现结点组成的变化。例如人力地图中某个职员结点的下属变化，计算机中某个组件连接的下级组件变化，都可以通过子节点的维护来实现。
2. 自由组成对象。Leaf对象可以组合成Composite对象，Composite对象又可以被组合成新的Composite对象。通过组合Component类型对象，很容易就可以得到新的对象。
3. 客户端调用透明。客户端在使用任何Composite对象或者Leaf对象时，没有任何区别。

兔小白：可以自由组成对象？那岂不是任何对象都可以使用Component类型对象组合而成？万物皆Component，只要定义好Component，想要什么对象就组成什么对象！

熊小猫：哈哈，哪有这等美事？组合模式是结构型设计模式，他专注解决的问题是对象结构。你有没有发现，所有由Component组成的对象，虽然组成不同，但是行为完全一样！

兔小白：还真是这样！行为已经定义在Component中，再怎么组合也不能产生新的行为。

熊小猫：组合模式的灵活优势，建立在更高程度的抽象之上。但是抽象程度越高，对象间的差别就会越小。例如人力地图里，不管是内部员工还是外协员工，都被抽象为员工。电脑组装的例子中，主板、CPU、内存、硬盘、显卡都被抽象为组件。高度抽象后，对象间不一样的行为会被消除掉，只剩下少量共有行为。

兔小白：让你这样一说，我又觉得组合模式好像用处不大了......

熊小猫：没有万能的设计模式呀！只有用对了地方，设计模式才能发挥出它的价值！下面这几种场景，比较适合使用组合模式。

1. 对象之间是层级关系。例如人力地图是很典型的层级关系。再比如汽车的车型、车系、车款也是层层细化。还有思考问题时，经常使用的“脑图”，也是把你头脑中的想法分层。常见的还有文件系统、菜单目录等。
2. 对象之间是整体和部分的关系。例如前面提到的计算机组成结构。其实大部分硬件产品都具备这种结构。

使用组合模式是为了获取对象组合的灵活性，通过对象组合形成新的对象。除此之外，还有一个重要的原因，利用树状结构的特点，可以将整体运算分解为部分运算，从而递归求解。例如计算某个产品组件的总成本，可以对构成这个组件的树上的所有组件成本递归求和。人力地图的例子中，显示某个员工的所有下级员工，也使用了递归。

其实以上场景也可以不用组合模式来实现。对象在丧失灵活性的同时，会获得更好的个性化。例如在计算机的例子中，完全可以定义个性化的主板类、硬盘类、CPU类等组件类，每个类都有自己独立的行为。但是由于丧失了统一的接口，客户端在使用这些类的时候不能一视同仁，很难将问题分解为子问题，使用递归求解。

在工作中，即使遇到典型的树状对象关系的场景，也要再多思考一下：你到底想要的是对象构成的灵活、树状结构的便利，还是需要个性化的对象。同一个场景下，不同的设计方式各有利弊。