互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

第 21 章　组合模式

本章内容

- 公司的人事架构是这样的吗
- 组合模式的定义
- 组合模式的应用
- 组合模式的扩展
- 最佳实践

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

21.1 公司的人事架构是这样的吗

各位读者，大家在上学的时候应该都学过“数据结构”这门课程吧，还记得其中有一节叫“二叉树”吧，我们上学那会儿这一章节是必考内容，左子树，右子树，什么先序遍历后序遍历，重点就是二叉树的遍历，我还记得当时老师就说，考试的时候一定有二叉树的构建和遍历，现在想起来还是觉得老师是正确的，树状结构在实际中应用非常广泛，想想看你最经常使用的 XML 格式是不是就是一个树形结构。

咱就先说个最常见的例子，公司的人事管理就是一个典型的树状结构，你想想你公司的组织架构是不是如图 21-1 所示。

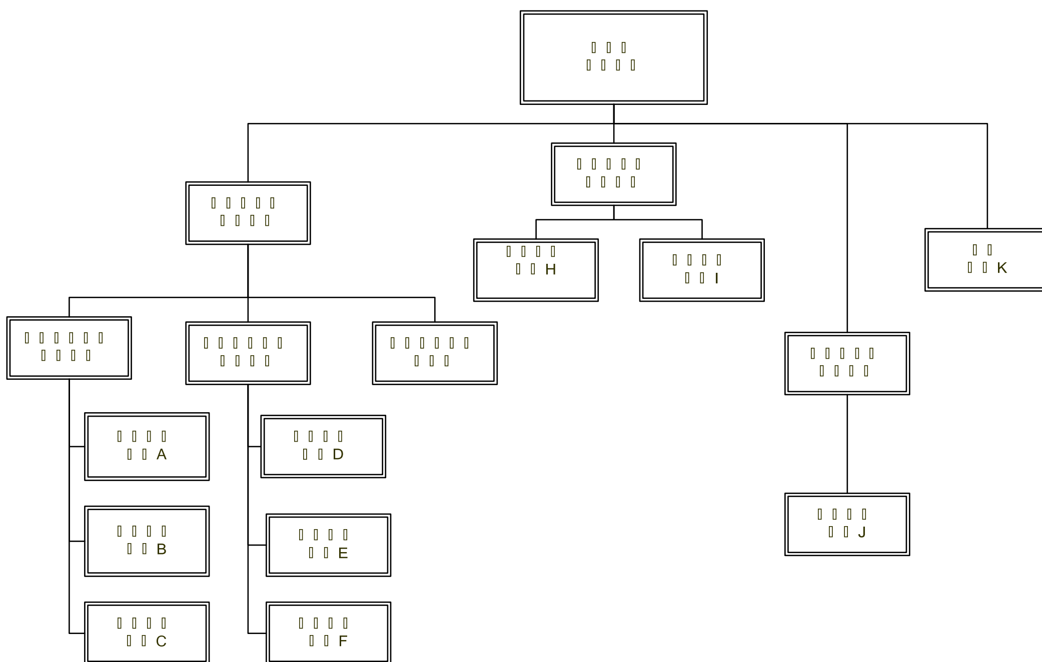

图21-1　普遍的组织架构

从最高的老大，往下一层一层的管理，最后到我们这层小兵……很典型的树状结构（说明一下，这不是二叉树，有关二叉树的定义可以翻翻以前的教科书），我们今天的任务就是要把这个树状结构实现出来，并且还要把它遍历一遍，就类似于阅读你公司的人员花名册。

从该树状结构上分析，有两种不同性质的节点：有分支的节点（如研发部经理）和无分支的节点（如员工 A、员工 D 等），我们增加一点学术术语上去，总经理叫做根节点(是不是想到 XML 中的那个根节点 root，那就对了)，类似研发部经理有分支的节点叫做树枝节点，类似员工 A 的无分支的节点叫做树叶节点，都很形象，三个类型的节点，那是不是定义三个类就可以？好，我们按照这

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

个思路走下去，先看我们自己设计的类图，如图 21-2 所示。

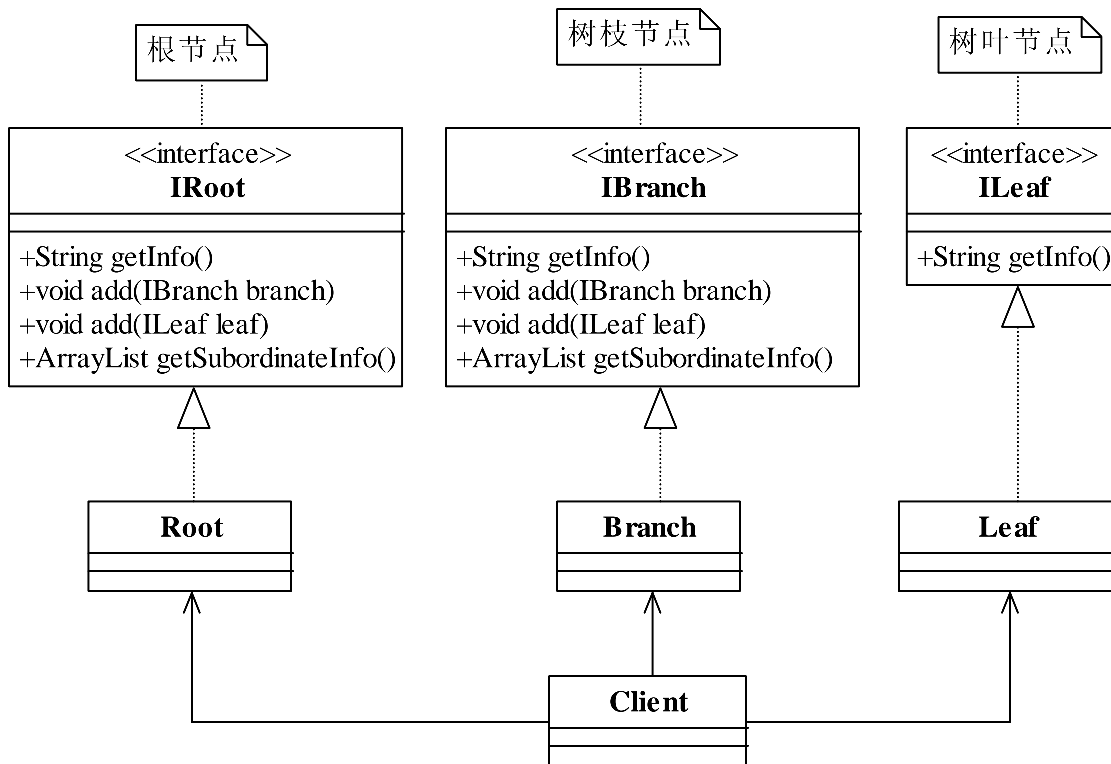

图21-2　最容易想到的组织架构类图

这个类图是初学者最容易想到的类图（首先声明，这个类图是有缺陷的，如果你已经看明白这个类图的缺陷了，该段落就可以一目十行地看下去，我们是循序渐进地讲课，一步一个脚印），非常简单，我们来看一下如何实现，先看最高级别的根节点接口，如代码清单 21-1 所示。

代码清单21-1　根节点接口

```
public interface IRoot {
     //得到总经理的信息
     public String getInfo();
     //总经理下边要有小兵，那要能增加小兵，比如研发部总经理，这是个树枝节点
     public void add(IBranch branch);
     //那要能增加树叶节点
     public void add(ILeaf leaf);
     //既然能增加，那要还要能够遍历，不可能总经理不知道他手下有哪些人
     public ArrayList getSubordinateInfo();
}
```

这个根节点的对象就是我们的总经理 CEO，其具体实现如代码清单 21-2 所示。

代码清单21-2　根节点的实现

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
public class Root implements IRoot {
     //保存根节点下的树枝节点和树叶节点，Subordinate 的意思是下级
     private ArrayList subordinateList = new ArrayList();
     //根节点的名称
     private String name = "";
     //根节点的职位
     private String position = "";
     //根节点的薪水
     private int salary = 0;
     //通过构造函数传递进来总经理的信息
     public Root(String name,String position,int salary){
             this.name = name;
             this.position = position;
             this.salary = salary;
     }
     //增加树枝节点
     public void add(IBranch branch) {
             this.subordinateList.add(branch);
     }
     //增加叶子节点，比如秘书，直接隶属于总经理
     public void add(ILeaf leaf) {
             this.subordinateList.add(leaf);
     }
     //得到自己的信息
     public String getInfo() {
             String info = "";
             info = "名称："+ this.name;;
             info = info + "\t 职位： " + this.position;
             info = info + "\t 薪水： " + this.salary;
             return info;
     }
     //得到下级的信息
     public ArrayList getSubordinateInfo() {
             return this.subordinateList;
     }
}
```

很简单，通过构造函数传入参数，然后获得信息，可以增加子树枝节点（部门经理）和叶子节点（秘书）。我们再来看其他有分支的节点接口，如代码清单 21-3 所示。

代码清单21-3 其他有分支的节点接口

```
public interface IBranch {
     //获得信息
     public String getInfo();
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        //增加数据节点，例如研发部下的研发一组
        public void add(IBranch branch);
        //增加叶子节点
        public void add(ILeaf leaf);
        //获得下级信息
        public ArrayList getSubordinateInfo();
}
```

有了接口，就应该有实现，其具体的实现类如代码清单 21-4 所示。

代码清单21-4　分支的节点实现

```
public class Branch implements IBranch {
        //存储子节点的信息
        private ArrayList subordinateList = new ArrayList();
        //树枝节点的名称
        private String name="";
        //树枝节点的职位
        private String position = "";
        //树枝节点的薪水
        private int salary = 0;
        //通过构造函数传递树枝节点的参数
        public Branch(String name,String position,int salary){
                this.name = name;
                this.position = position;
                this.salary = salary;
        }
        //增加一个子树枝节点
        public void add(IBranch branch) {
                this.subordinateList.add(branch);
        }
        //增加一个叶子节点
        public void add(ILeaf leaf) {
                this.subordinateList.add(leaf);
        }
        //获得自己树枝节点的信息
        public String getInfo() {
                String info = "";
                info = "名称: " + this.name;
                info = info + "\t 职位: "+ this.position;
                info = info + "\t 薪水: "+this.salary;
                return info;
        }
        //获得下级的信息
        public ArrayList getSubordinateInfo() {
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
                return this.subordinateList;
        }
}
```

不管是总经理还是部门经理都是有子节点的存在，最终的子节点就是叶子节点，其接口如代码清单 21-5 所示。

代码清单21-5 叶子节点的接口

```
public interface ILeaf {
        //获得自己的信息呀
        public String getInfo();

}
```

叶子节点的接口简单，实现也非常容易，如代码清单 21-6 所示。

代码清单21-6 叶子节点的实现

```
public class Leaf implements ILeaf {
        //叶子叫什么名字
        private String name = "";
        //叶子的职位
        private String position = "";
        //叶子的薪水
        private int salary=0;
        //通过构造函数传递信息
        public Leaf(String name,String position,int salary){
                this.name = name;
                this.position = position;
                this.salary = salary;
        }
        //最小的小兵只能获得自己的信息了
        public String getInfo() {
                String info = "";
                info = "名称: " + this.name;
                info = info + "\t 职位: "+ this.position;
                info = info + "\t 薪水: "+this.salary;
                return info;
        }
}
```

好了，所有的根节点、树枝节点和叶子节点都已经实现了，从总经理、部门经理到最终的员工都已经实现，然后的工作就是组装成一个树状结构并遍历这颗树，通过什么来完成呢？通过场景类 Client 完成，如代码清单 21-7 所示。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

代码清单21-7 场景类

```
   public class Client {

        public static void main(String[] args) {
                //首先产生了一个根节点
                IRoot ceo = new Root("王大麻子","总经理",100000);
                //产生三个部门经理，也就是树枝节点
                IBranch developDep = new Branch("刘大瘸子","研发部门经理
",10000);
                IBranch salesDep = new Branch("马二拐子","销售部门经理
",20000);
                IBranch financeDep = new Branch("赵三驼子","财务部经理
",30000);

                //再把三个小组长产生出来
                IBranch firstDevGroup = new Branch("杨三乜斜","开发一组组
长",5000);
                IBranch secondDevGroup = new Branch("吴大棒槌","开发二组组
长",6000);

                //剩下的及时我们这些小兵了,就是路人甲，路人乙
                ILeaf a = new Leaf("a","开发人员",2000);
                ILeaf b = new Leaf("b","开发人员",2000);
                ILeaf c = new Leaf("c","开发人员",2000);
                ILeaf d = new Leaf("d","开发人员",2000);
                ILeaf e = new Leaf("e","开发人员",2000);
                ILeaf f = new Leaf("f","开发人员",2000);
                ILeaf g = new Leaf("g","开发人员",2000);
                ILeaf h = new Leaf("h","销售人员",5000);
                ILeaf i = new Leaf("i","销售人员",4000);
                ILeaf j = new Leaf("j","财务人员",5000);
                ILeaf k = new Leaf("k","CEO 秘书",8000);
                ILeaf zhengLaoLiu = new Leaf("郑老六","研发部副总",20000);

                //该产生的人都产生出来了，然后我们怎么组装这棵树
                //首先是定义总经理下有三个部门经理
                ceo.add(developDep);
                ceo.add(salesDep);
                ceo.add(financeDep);
                //总经理下还有一个秘书
                ceo.add(k);
                //定义研发部门 下的结构
                developDep.add(firstDevGroup);
                developDep.add(secondDevGroup);
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
                //研发部经理下还有一个副总
                developDep.add(zhengLaoLiu);
                //看看开发两个开发小组下有什么
                firstDevGroup.add(a);
                firstDevGroup.add(b);
                firstDevGroup.add(c);
                secondDevGroup.add(d);
                secondDevGroup.add(e);
                secondDevGroup.add(f);
                //再看销售部下的人员情况
                salesDep.add(h);
                salesDep.add(i);
                //最后一个财务
                financeDep.add(j);
                //树状结构写完毕，然后我们打印出来
                System.out.println(ceo.getInfo());
                //打印出来整个树形
                getAllSubordinateInfo(ceo.getSubordinateInfo());
        }
        //遍历所有的树枝节点，打印出信息
        private static void getAllSubordinateInfo(ArrayList
subordinateList){
                int length = subordinateList.size();
                for(int m=0;m<length;m++){  //定义一个 ArrayList 长度，不要
在 for 循环中每次计算
                        Object s = subordinateList.get(m);
                        if(s instanceof Leaf){  //是个叶子节点，也就是员工
                                ILeaf employee = (ILeaf)s;
                                System.out.println(((Leaf)
s).getInfo());
                        }else{
                                IBranch branch = (IBranch)s;
                                System.out.println(branch.getInfo());
                                //再递归调用

        getAllSubordinateInfo(branch.getSubordinateInfo());
                        }
                }
        }
    }
```

这个程序比较长，如果在我们的项目中有这样的程序，肯定是要被拉出来做典型的，你写一大坨的程序给谁呀，以后还要维护，程序要短小精悍！幸运的是，我们这是作为案例来讲解，而且就

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

是指出这样组装这棵树是有问题的，等会我们深入讲解，先看运行结果：

```
名称：王大麻子   职位：总经理       薪水： 100000
名称：刘大瘸子   职位：研发部门经理   薪水：10000
名称：杨三乜斜   职位：开发一组组长   薪水：5000
名称：a          职位：开发人员     薪水：2000
名称：b          职位：开发人员     薪水：2000
名称：c          职位：开发人员     薪水：2000
名称：吴大棒槌   职位：开发二组组长   薪水：6000
名称：d          职位：开发人员     薪水：2000
名称：e          职位：开发人员     薪水：2000
名称：f          职位：开发人员     薪水：2000
名称：郑老六     职位：研发部副总     薪水：20000
名称：马二拐子   职位：销售部门经理   薪水：20000
名称：h          职位：销售人员     薪水：5000
名称：i          职位：销售人员     薪水：4000
名称：赵三驼子   职位：财务部经理     薪水：30000
名称：j          职位：财务人员     薪水：5000
名称：k          职位：CEO 秘书     薪水：8000
```

和我们期望的结果一样，一棵完整的树就生成了，而且我们还能够遍历。不错，不错，但是看类图或程序的时候，你有没有发觉有问题？getInfo 每个接口都有，为什么不能抽象出来？Root 类和 Branch 类有什么差别？根节点本身就是树枝节点的一种，为什么要定义成两个接口两个类？如果我要加一个任职期限，你是不是每个类都需要修改？如果我要后序遍历（从员工找到他的上级领导）能做到吗？——彻底晕菜了！

问题很多，我们一个一个解决，先说抽象的问题。我们确实可以把 IBranch 和 IRoot 合并成一个接口，确认无疑的事我们先做，那我们就修改一下类图，如图 21-3 所示。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

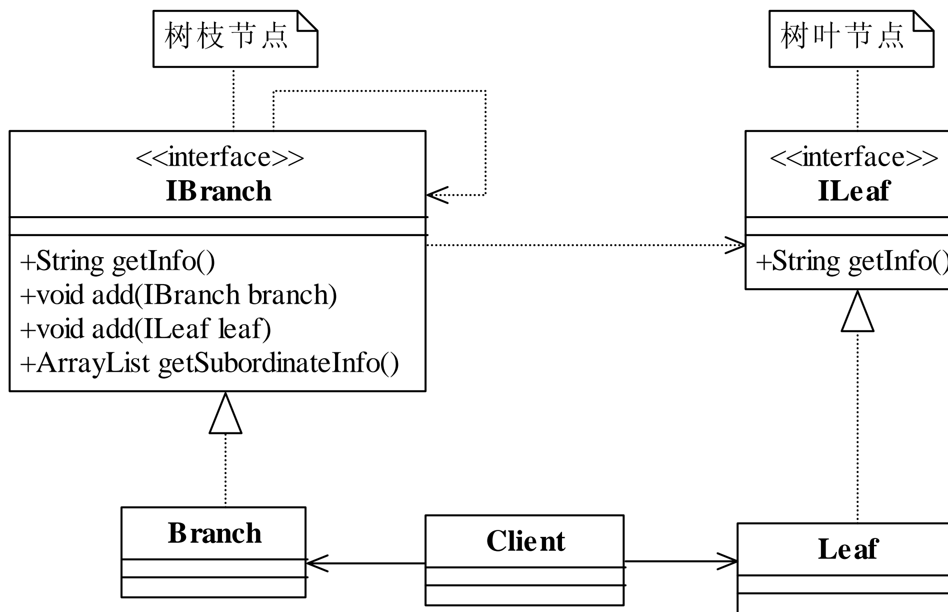

图21-3　整合根节点和树枝节点后的类图

仔细看看这个类图，还能不能发现点问题。想想看接口的作用是什么？定义一类事物所具有的共性，那 ILeaf 和 IBranch 是不是也有共性呢？有，getInfo 方法！我们是不是要把这个共性也封装起来呢？是的，是的，提炼事物的共同点，然后封装之，这是我们作为设计专家的拿手好戏，修改后的类图如图 21-4 所示。

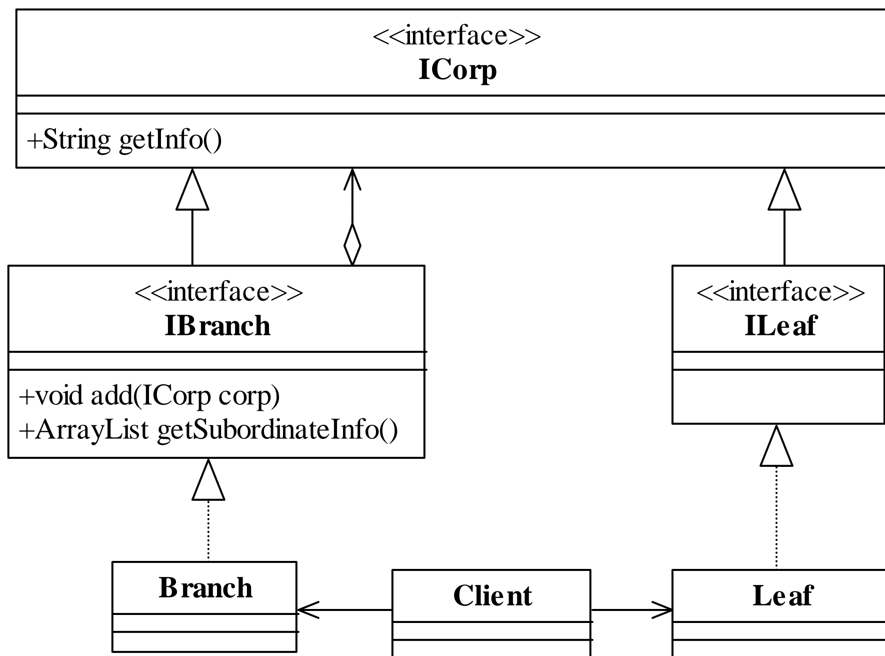

图21-4　修改后的类图

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

类图上增加了一个 ICorp 接口，它是公司所有人员信息的接口类，不管你是经理还是员工，你都有名字、职位、薪水，这个定义成一个接口没有错，但是你可能对于 ILeaf 接口持怀疑状态，空接口有何意义呀？有意义!它是每个树枝节点的代表，系统扩容的时候你就会发现它是多么“栋梁”。我们先来看新增加的接口 ICorp，如代码清单 21-8 所示。

代码清单21-8　公司人员接口

```
public interface ICorp {
	//每个员工都有信息，你想隐藏，门儿都没有！
	public String getInfo();
}
```

接口很简单，只有一个方法，就是获得员工的信息，树叶节点是最基层的构件，我们先来看看它的接口，空接口，对的，如代码清单 21-9 所示。

代码清单21-9　树叶接口

```
public interface ILeaf extends ICorp {

}
```

树叶接口的实现类如代码清单21-10所示。

代码清单21-10　　树叶接口

```
public class Leaf implements ILeaf {
	//小兵也有名称
	private String name = "";
	//小兵也有职位
	private String position = "";
	//小兵也有薪水，否则谁给你干
	private int salary = 0;
	//通过一个构造函数传递小兵的信息
	public Leaf(String name,String position,int salary){
		this.name = name;
		this.position = position;
		this.salary = salary;
	}
	//获得小兵的信息
	public String getInfo() {
		String info = "";
		info = "姓名：" + this.name;
		info = info + "\t 职位："+ this.position;
		info = info + "\t 薪水：" + this.salary;
		return info;
	}
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
}
```

小兵就只有这些信息了，我们是具体干活的，我们是管理不了其他同事的，我们来看看那些经理和小组长是怎么实现的，也就是 IBranch 接口，如代码清单 21-11 所示。

代码清单21-11　树枝接口

```
public interface IBranch extends ICorp {
	//能够增加小兵(树叶节点）或者是经理（树枝节点）
	public void addSubordinate(ICorp corp);
	//我还要能够获得下属的信息
	public ArrayList<ICorp> getSubordinate();
	/*本来还应该有一个方法 delSubordinate(ICorp corp)，删除下属
	 * 这个方法我们没有用到就不写进来了
	 */
}
```

接口也很简单，其实现类也不可能太复杂，如代码清单 21-12 所示。

代码清单21-12　树枝实现类

```
public class Branch implements IBranch {
	//领导也是人，也有名字
	private String name = "";
	//领导和领导不同，也是职位区别
	private String position = "";
	//领导也是拿薪水的
	private int salary = 0;
	//领导下边有那些下级领导和小兵
	ArrayList<ICorp> subordinateList = new ArrayList<ICorp>();
	//通过构造函数传递领导的信息
	public Branch(String name,String position,int salary){
		this.name = name;
		this.position = position;
		this.salary = salary;
	}
	//增加一个下属，可能是小头目，也可能是个小兵
	public void addSubordinate(ICorp corp) {
		this.subordinateList.add(corp);
	}
	//我有哪些下属
	public ArrayList<ICorp> getSubordinate() {
		return this.subordinateList;
	}
	//领导也是人，他也有信息
	public String getInfo() {
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
                String info = "";
                info = "姓名：" + this.name;
                info = info + "\t 职位："+ this.position;
                info = info + "\t 薪水：" + this.salary;
                return info;
        }
}
```

实现类也很简单，不多说，程序写得好不好，就看别人怎么调用了，我们看场景类 Client，如代码清单 21-13 所示。

代码清单21-13　　场景类

```
public class Client {

        public static void main(String[] args) {
                //首先是组装一个组织结构出来
                Branch ceo = compositeCorpTree();
                //首先把 CEO 的信息打印出来：
                System.out.println(ceo.getInfo());
                //然后是所有员工信息
                System.out.println(getTreeInfo(ceo));
        }
        //把整个树组装出来
        public static Branch compositeCorpTree(){
                //首先产生总经理 CEO
                Branch root = new Branch("王大麻子","总经理",100000);
                //把三个部门经理产生出来
                Branch developDep = new Branch("刘大瘸子","研发部门经理
",10000);
                Branch salesDep = new Branch("马二拐子","销售部门经理
",20000);
                Branch financeDep = new Branch("赵三驼子","财务部经理
",30000);
                //再把三个小组长产生出来
                Branch firstDevGroup = new Branch("杨三乜斜","开发一组组长
",5000);
                Branch secondDevGroup = new Branch("吴大棒槌","开发二组组
长",6000);

                //把所有的小兵都产生出来
                Leaf a = new Leaf("a","开发人员",2000);
                Leaf b = new Leaf("b","开发人员",2000);
                Leaf c = new Leaf("c","开发人员",2000);
                Leaf d = new Leaf("d","开发人员",2000);
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
            Leaf e = new Leaf("e","开发人员",2000);
            Leaf f = new Leaf("f","开发人员",2000);
            Leaf g = new Leaf("g","开发人员",2000);
            Leaf h = new Leaf("h","销售人员",5000);
            Leaf i = new Leaf("i","销售人员",4000);
            Leaf j = new Leaf("j","财务人员",5000);
            Leaf k = new Leaf("k","CEO 秘书",8000);
            Leaf zhengLaoLiu = new Leaf("郑老六","研发部副经理",20000);

            //开始组装
            //CEO 下有三个部门经理和一个秘书
            root.addSubordinate(k);
            root.addSubordinate(developDep);
            root.addSubordinate(salesDep);
            root.addSubordinate(financeDep);
            //研发部经理
            developDep.addSubordinate(zhengLaoLiu);
            developDep.addSubordinate(firstDevGroup);
            developDep.addSubordinate(secondDevGroup);
            //看看开发两个开发小组下有什么
            firstDevGroup.addSubordinate(a);
            firstDevGroup.addSubordinate(b);
            firstDevGroup.addSubordinate(c);
            secondDevGroup.addSubordinate(d);
            secondDevGroup.addSubordinate(e);
            secondDevGroup.addSubordinate(f);
            //再看销售部下的人员情况
            salesDep.addSubordinate(h);
            salesDep.addSubordinate(i);
            //最后一个财务
            financeDep.addSubordinate(j);

            return root;
      }
      //遍历整棵树,只要给我根节点，我就能遍历出所有的节点
      public static String getTreeInfo(Branch root){
            ArrayList<ICorp> subordinateList =
root.getSubordinate();
            String info = "";
            for(ICorp s :subordinateList){
                  if(s instanceof Leaf){ //是员工就直接获得信息
                        info = info + s.getInfo()+"\n";
                  }else{ //是个小头目
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
                                info = info + s.getInfo() +"\n"+
getTreeInfo((Branch)s);
                        }
                }
                return info;
        }
    }
```

运行结果完全相同，不再赘述。通过这样构件，一个非常清晰的树状人员资源管理图出现了，那我们的程序是否还可以优化？可以!你看 Leaf 和 Branch 中都有 getInfo 信息，是不是可以抽象？好，我们抽象一下，如图 21-5 所示。

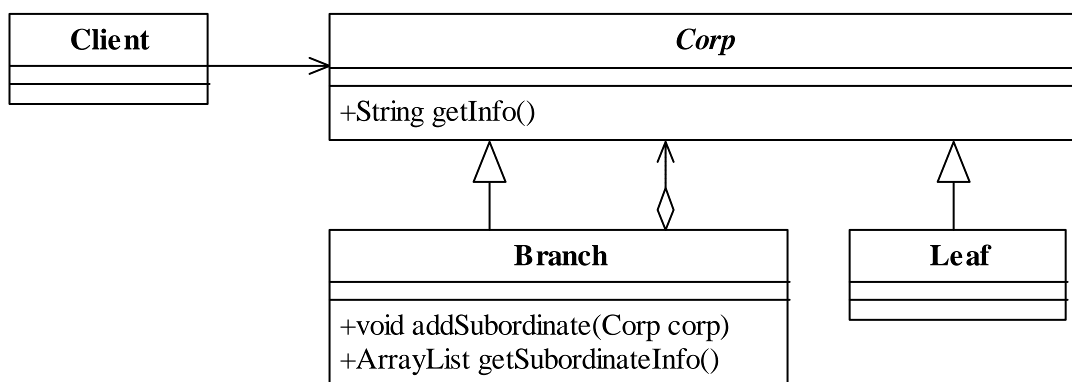

图21-5　精简的类图

你一看这个图，乐了。能不乐嘛，减少很多工作量了，接口没有了，改成抽象类了，IBranch 接口也没有了，直接把方法放到了实现类中了，太精简了！而且场景类只认定抽象类 Corp 就成，那我们首先来看抽象类 ICorp，如代码清单 21-14 所示。

代码清单21-14　抽象公司职员类

```
    public abstract class Corp {
         //公司每个人都有名称
         private String name = "";
         //公司每个人都职位
         private String position = "";
         //公司每个人都有薪水
         private int salary =0;

         public Corp(String _name,String _position,int _salary){
                 this.name = _name;
                 this.position = _position;
                 this.salary = _salary;
         }
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        //获得员工信息
        public String getInfo(){
                String info = "";
                info = "姓名：" + this.name;
                info = info + "\t 职位："+ this.position;
                info = info + "\t 薪水：" + this.salary;
                return info;
        }
    }
```

抽象类嘛，就应该抽象出一些共性的东东出来，然后看两个具体的实现类，树叶节点如代码清单 21-15 所示。

代码清单21-15　树叶节点

```
    public class Leaf extends Corp {
        //就写一个构造函数，这个是必须的
        public Leaf(String _name,String _position,int _salary){
                super(_name,_position,_salary);
        }
    }
```

这个精简得比较多，几行代码就完成了，确实就应该这样，下面是小头目的实现类，如代码清单 21-16 所示。

代码清单21-16　树枝节点

```
    public class Branch extends Corp {
        //领导下边有那些下级领导和小兵
        ArrayList<Corp> subordinateList = new ArrayList<Corp>();
        //构造函数是必须的了
        public Branch(String _name,String _position,int _salary){
                super(_name,_position,_salary);
        }
        //增加一个下属，可能是小头目，也可能是个小兵
        public void addSubordinate(Corp corp) {
                this.subordinateList.add(corp);
        }
        //我有哪些下属
        public ArrayList<Corp> getSubordinate() {
                return this.subordinateList;
        }
    }
```

场景类中构建树形结构，并进行遍历。组装没有变化，遍历组织机构数稍有变化，如代码清单 21-17 所示。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

代码清单21-17　稍稍修改的场景类

```
public class Client {
    //遍历整棵树,只要给我根节点，我就能遍历出所有的节点
    public static String getTreeInfo(Branch root){
        ArrayList<Corp> subordinateList = root.getSubordinate();
        String info = "";
        for(Corp s :subordinateList){
            if(s instanceof Leaf){ //是员工就直接获得信息
                info = info + s.getInfo()+"\n";
            }else{ //是个小头目
                info = info + s.getInfo() +"\n"+
getTreeInfo((Branch)s);
            }
        }
        return info;
    }
}
```

场景类中 main 方法没有变动，请参考代码清单 21-7 所示，不再赘述。遍历组织机构树的 getTreeInfo 稍有修改，就是把用到 ICorp 接口的地方修改为 Corp 抽象类，仅仅修改了粗体部分，其他保持不变，运行结果相同。这就是组合模式。

21.2 组合模式的定义

组合模式(Composite Pattern)也叫合成模式，有时又叫做部分—整体模式（Part-Whole），主要是用来描述部分与整体的关系，其定义如下：Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. 将对象组合成树形结构以表示“部分-整体”的层次结构，使得用户对单个对象和组合对象的使用具有一致性。

组合模式的通用类图如图 21-6 所示。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

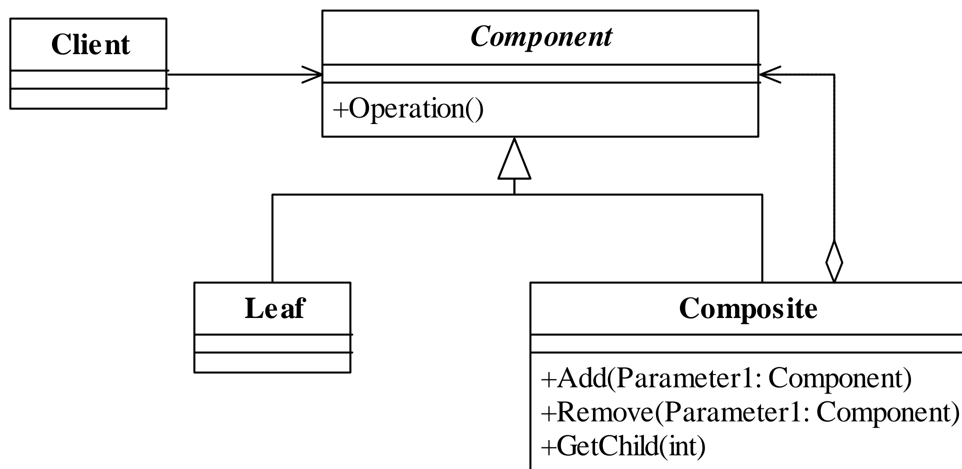

图21-6　组合模式通用类图

我们先来说说组合模式的几个角色：

- Component 抽象构件角色

定义参加组合对象的共有方法和属性，可以定义一些默认的行为或属性，比如我们例子中的 getInfo 就封装到了抽象类中。

- Leaf 叶子构件

叶子对象，其下再也没有其他的分支，也就是遍历的最小单位。

- Composite 树枝构件

树枝对象，它的作用是组合树枝节点和叶子节点形成一个树形结构。

我们来看组合模式的通用源代码，首先看抽象构件，如代码清单 21-18 所示。

代码清单21-18　抽象构件

```
public abstract class Component {
     //个体和整体都具有的共享
     public void doSomething(){
             //编写业务逻辑
     }
}
```

组合模式的重点就在树枝构件，其通用代码如代码清单 21-19 所示。

代码清单21-19　树枝构件

```
public class Composite extends Component {
     //构件容器
     private ArrayList<Component> componentArrayList = new 
ArrayList<Component>();
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
    //增加一个叶子构件或树枝构件
    public void add(Component component){
            this.componentArrayList.add(component);
    }
    //删除一个叶子构件或树枝构件
    public void remove(Component component){
            this.componentArrayList.remove(component);
    }
    //获得分支下的所有叶子构件和树枝构件
    public ArrayList<Component> getChildren(){
            return this.componentArrayList;
    }
}
```

树叶节点是没有子下级对象的对象，定义参加组合的原始对象行为，其通用源代码如代码清单 21-20 所示。

代码清单21-20　树叶构件

```
public class Leaf extends Component {
    /*
     * 可以覆写父类方法
     * public void doSomething(){
     *
     * }
     */
}
```

场景类负责树状结构的建立，并可以通过递归方式遍历整个树，如代码清单 21-21 所示。

代码清单21-21　场景类

```
public class Client {

    public static void main(String[] args) {
            //创建一个根节点
            Composite root = new Composite();
            root.doSomething();
            //创建一个树枝构件
            Composite branch = new Composite();
            //创建一个叶子节点
            Leaf leaf = new Leaf();
            //建立整体
            root.add(branch);
            branch.add(leaf);
    }
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
    //通过递归遍历树
    public static void display(Composite root){

        for(Component c:root.getChildren()){
            if(c instanceof Leaf){ //叶子节点
                c.doSomething();
            }else{ //树枝节点
                display((Composite)c);
            }
        }
    }
}
```

各位可能已经看出一些问题了，组合模式是对依赖倒转原则的破坏，但是它还有其他类型的变形，面向对象就是这么多的形态和变化，请读者继续阅读下去，就会找到解决方案。

21.3 组合模式的应用

21.3.1 组合模式的优点

- 高层模块调用简单

一棵树形机构中的所有节点都是 Component，局部和整体对调用者来说没有任何区别，也就是说，高层模块不必关心自己处理的是单个对象还是整个组合结构，简化了高层模块的代码。

- 节点自由增加

使用了组合模式后，我们可以看看，如果想增加一个树枝节点、树叶节点是不是都很容易呀，只要找到它的父节点就成，非常容易扩展，符合开闭原则，对以后的维护非常有利。

21.3.2 组合模式的缺点

组合模式有一个非常明显的缺点，看到我们在场景类中的定义，提到树叶和树枝使用时的定义了吗？直接使用了实现类！这在面向接口编程上是很不恰当的，与依赖倒置原则冲突，读者在使用的时候要考虑清楚，它限制了你接口的影响范围。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

21.3.3 组合模式的应用

- 维护和展示部分－整体关系的场景，如树形菜单、文件和文件夹管理。
- 从一个整体中能够独立出部分模块或功能的场景。

21.3.4 组合模式的注意事项

只要是树形结构，就要考虑使用组合模式，这个一定要记住，只要是要体现局部和整体的关系的时候，而且这种关系还可能比较深，考虑一下组合模式吧。

21.4 组合模式的扩展

21.4.1 真实的组合模式

什么是真实的组合模式？就是你在实际项目中使用的组合模式，而不是仅仅依照书本上学习到的模式，它是“实践出真知”。在我们的例子中，经过精简后，确实是类、接口减少了很多，而且程序也简单很多，但是大家可能还是很迷茫，这个 Client 程序并没有改变多少呀，非常正确，树的组装你是跑不了的，你要知道在项目中使用关系型数据库来存储这些信息，你从数据库中可以直接提取出哪些人要分配到树枝，哪些人要分配到树叶，树枝与树枝、树叶的关系等等，这些都是由相关的业务人员维护到数据库中的，通常这里是把数据存放到一张单独的表中，表结构如图 21-7 所示。

主键	唯一编码	名称	是否是叶子节点	父节点
1	CEO	总经理	否	NULL
2	developDep	研发部经理	否	CEO
3	salesDep	销售部经理	否	CEO
4	financeDep	财务部经理	否	CEO
5	k	总经理秘书	是	CEO
6	a	员工 A	是	developDep
7	b	员工 B	是	salesDep

图21-7　关系数据库中存储的树形结构

这张数据表定义了一个树形结构，我们要做的就是从数据库中把它读取出来，然后展现到前台

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

上，用 for 循环加上递归就可以完成这个读取。用了数据库后，数据和逻辑已经在表中定义好了，我们直接读取放到树上就可以了，这个还是比较容易做的，大家不妨自己考虑一下。

这才是组合模式的真实引用，它依靠了关系数据库的非对象存储性能，非常方便地保存了一个树形结构。大家可以在项目中考虑采用，想想看现在还有哪个项目不使用关系型数据库呢？

21.4.2 透明的组合模式

组合模式有两种不同的实现：透明模式和安全模式，我们上面讲的就是安全模式，那透明模式是什么样子呢？透明模式的通用类图如图 21-8 所示。

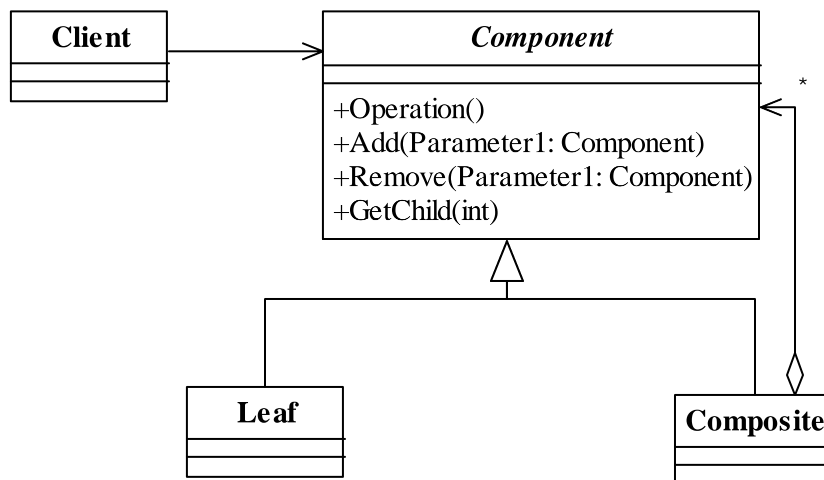

图21-8　透明模式的通用类图

我们与图 21-6 所示的安全模式类图对比一下就非常清楚了，透明模式是把用来组合使用的方法放到抽象类中，比如 add()，remove()以及 getChildren 等方法（顺便说一下，getChildren 一般返回的结果为 Iterable 的实现类，很多，大家可以看 JDK 的帮助），不管叶子对象还是树枝对象都有相同的结构，通过判断是 getChildren 的返回值确认是叶子节点还是树枝节点，如果处理不当，这个会在运行期出现问题，不是很建议的方式；安全模式就不同了，它是把树枝节点和树叶节点彻底分开，树枝节点单独拥有用来组合的方法，这种方法比较安全，我们的例子使用了安全模式。

由于透明模式的使用者还是比较多，我们也把它的通用源代码共享出来，首先看抽象构件，如代码清单 21-22 所示。

代码清单21-22　抽象构件

```
public abstract class Component {
     //个体和整体都具有的共享
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        public void doSomething(){
                //编写业务逻辑
        }
        //增加一个叶子构件或树枝构件
        public abstract void add(Component component);
        //删除一个叶子构件或树枝构件
        public abstract void remove(Component component);
        //获得分支下的所有叶子构件和树枝构件
        public abstract ArrayList<Component> getChildren();
    }
```

抽象构件定义了树枝节点和树叶节点都必须具有的方法和属性，这样树枝节点的实现就不需要任何变化，如代码清单 21-19 所示。

树叶节点继承了 Component 抽象类，不想让它改变有点难呀，它必须实现三个抽象方法，怎么办？好办，给个空方法，如代码清单 21-23 所示。

代码清单21-23　树叶节点

```
    public class Leaf extends Component {

        @Deprecated
        public void add(Component component){
                //空实现
        }

        @Deprecated
        public void remove(Component component){
                //空实现
        }

        @Deprecated
        public ArrayList<Component> getChildren(){
                //空实现
                return null;
        }
    }
```

为什么要加个 Deprecated 注解呢？就是告诉调用者，你可以调我这个方法，但是可能出现错误哦，我已经告诉你“该方法已经失效”了，你还使用，那就没招儿了。

在透明模式下，遍历整个树形结构是比较容易的，不用进行强制类型转换，如代码清单 21-24 所示。

代码清单21-24　树结构遍历

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
public class Client {
    //通过递归遍历树
    public static void display(Component root){

        for(Component c:root.getChildren()){
            if(c instanceof Leaf){ //叶子节点
                c.doSomething();
            }else{ //树枝节点
                display(c);
            }
        }
    }
}
```

仅仅在遍历时不再进行牵制的类型转化了，其他的组装则没有任何变化。透明模式的好处就是它基本遵循了依赖倒转原则，方便系统进行扩展。

21.4.3 组合模式的遍历

我们在上面也还提到了一个问题，就是树的遍历问题，从上到下遍历没有问题，但是我要是从下往上遍历呢？比如组织机构这颗树，我从中抽取一个用户，要找到它的上级有哪些，下级有哪些，怎么处理？想想，~~~，再想想！想出来了吧，我们对下答案，类图如图 21-9 所示。

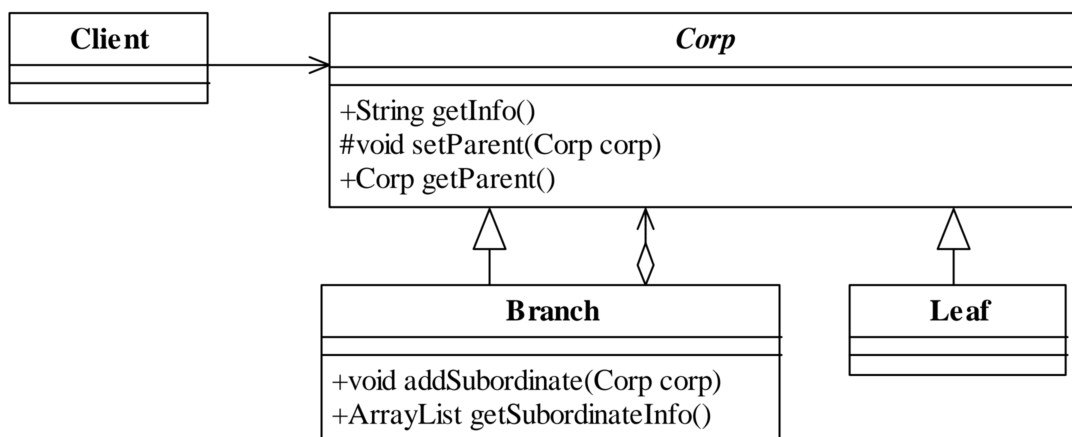

图21-9　增加父查询的类图

看类图中，在 Corp 类中增加了两个方法，setParent 是设置父节点是谁，getParent 是查找父节点是谁，我们来看一下程序的改变，如代码清单 21-25 所示。

代码清单21-25　抽象构件

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
public abstract class Corp {
	//公司每个人都有名称
	private String name = "";
	//公司每个人都职位
	private String position = "";
	//公司每个人都有薪水
	private int salary =0;
	//父节点是谁
	private Corp parent = null;

	public Corp(String _name,String _position,int _salary){
		this.name = _name;
		this.position = _position;
		this.salary = _salary;
	}
	//获得员工信息
	public String getInfo(){
		String info = "";
		info = "姓名：" + this.name;
		info = info + "\t 职位："+ this.position;
		info = info + "\t 薪水：" + this.salary;
		return info;
	}

	//设置父节点
	protected void setParent(Corp _parent){
		this.parent = _parent;
	}

	//得到父节点
	public Corp getParent(){
		return this.parent;
	}
}
```

就增加了粗体部分，然后我们再来看看树枝节点的改变，如代码清单 21-26 所示。

代码清单21-26　树枝构件

```
public class Branch extends Corp {
	//领导下边有那些下级领导和小兵
	ArrayList<Corp> subordinateList = new ArrayList<Corp>();
	//构造函数是必须的了
	public Branch(String _name,String _position,int _salary){
		super(_name,_position,_salary);
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        }
        //增加一个下属，可能是小头目，也可能是个小兵
        public void addSubordinate(Corp corp) {
                corp.setParent(this); //设置父节点
                this.subordinateList.add(corp);
        }
        //我有哪些下属
        public ArrayList<Corp> getSubordinate() {
                return this.subordinateList;
        }
}
```

增加了粗体部分。看懂程序了吗？甭管是树枝节点还是树叶节点，在每个节点都增加了一个属性：父节点对象，这样在树枝节点增加子节点或叶子节点是设置父节点，然后你看整棵树除了根节点外每个节点都有一个父节点，剩下的事情还不好处理吗？每个节点上都有父节点了，你要往上找，那就找呗！大家自己考虑一下，写个 find 方法，然后一步一步往上找，非常简单的方法，这里就不再赘述。

有了这个 parent 属性，什么后序遍历（从下往上找）、中序遍历（从中间某个环节往上或往下遍历）都解决了，这个就不多说了。

再提一个问题，树叶节点和树枝节点是有顺序的，你不能乱排，怎么办？比如我们上面的例子，研发一组下边有 3 个成员，这 3 个成员是要进行排序（在机关里这叫做排位，同样是同事也有个先后升迁顺序）的呀，你怎么处理？问我呀，问你呢，好好想想，以后用得着的！

21.5 最佳实践

组合模式在项目中到处都有，比如现在的页面结构一般都是上下结构，上面放系统的 Logo，下边分为两部分：左边是导航菜单，右边是展示区，左边的导航菜单一般都是树形的结构，比较清晰，有非常多的 JavaScript 源码实现了类似的树形菜单，大家可以到网上搜索一把。

还有，大家常用的 XML 结构也是一个树形结构，根节点、元素节点、值元素这些都与我们的组合模式相匹配，之所以本章节不以 XML 为例子讲解，是因为有人还直接读写 XML 文件吗？一般都是用 JDOM 或者 DOM4J 了。

还有一个非常重要的例子：我们自己本身也是一个树状结构的一个树枝或树叶。根据我能够找到我的父母，根据父亲又能找到爷爷奶奶，根据母亲能够找到外公外婆等等，很典型的树形结构，而且还很规范（这个要是不规范那肯定是乱套了）。