合成模式

> *整理自：《java与模式》之合成模式*

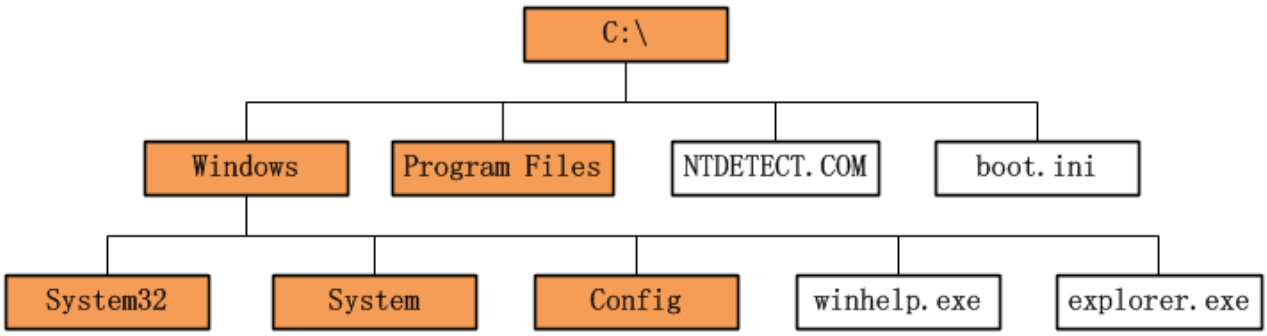
在阎宏博士的《JAVA与模式》一书中开头是这样描述合成（Composite）模式的：

合成模式属于对象的结构模式，有时又叫做“部分——整体”模式。合成模式将对象组织到树结构中，可以用来描述整体与部分的关系。合成模式可以使客户端将单纯元素与复合元素同等看待。

合成模式

合成模式把部分和整体的关系用树结构表示出来。合成模式使得客户端把一个个单独的成分对象和由它们复合而成的合成对象同等看待。

比如，一个文件系统就是一个典型的合成模式系统。下图是常见的计算机XP文件系统的一部分。



从上图可以看出，文件系统是一个树结构，树上长有节点。树的节点有两种，一种是树枝节点，即目录，有内部树结构，在图中涂有颜色；另一种是文件，即树叶节点，没有内部树结构。

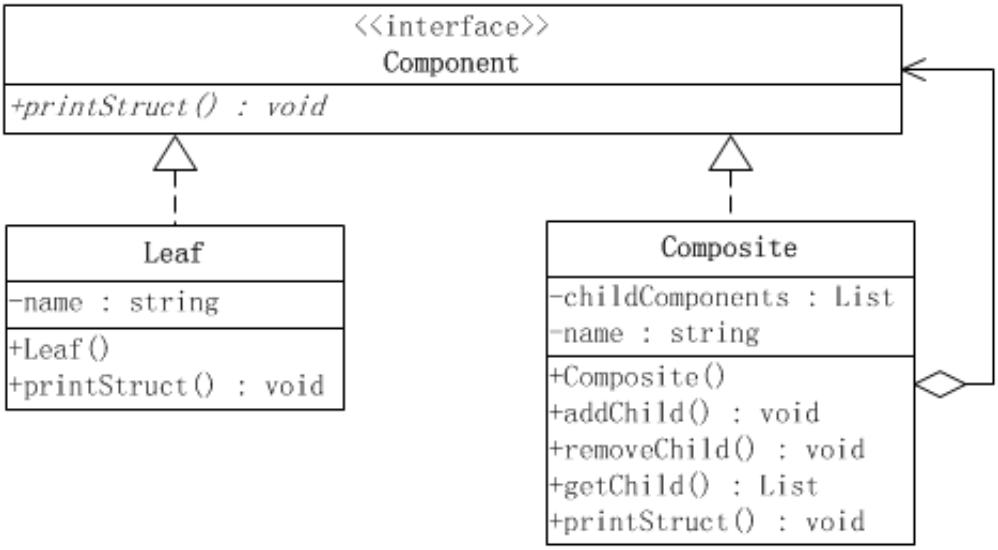
显然，可以把目录和文件当做同一种对象同等对待和处理，这也就是合成模式的应用。

合成模式可以不提供父对象的管理方法，但是合成模式必须在合适的地方提供子对象的管理方法，诸如：add()、remove()、以及getChild()等。

合成模式的实现根据所实现接口的区别分为两种形式，分别称为 **安全式** 和 **透明式**。

安全式合成模式的结构

安全模式的合成模式要求管理聚集的方法只出现在树枝构件类中，而不出现在树叶构件类中。



这种形式涉及到三个角色：

- **抽象构件(Component)角色：**这是一个抽象角色，它给参加组合的对象定义出公共的接口及其默认行为，可以用来管理所有的子对象。合成对象通常把它所包含的子对象当做类型为Component的对象。在安全式的合成模式里，构件角色并不定义出管理子对象的方法，这一定义由树枝构件对象给出。
- **树叶构件(Leaf)角色：**树叶对象是没有下级子对象的对象，定义出参加组合的原始对象的行为。
- **树枝构件(Composite)角色：**代表参加组合的有下级子对象的对象。树枝构件类给出所有的管理子对象的方法，如add()、remove()以及getChild()。

源代码

抽象构件角色类

```
public interface Component {
    /**
     * 输出组建自身的名称
     */
    public void printStruct(String preStr);
```

```
}
```

树枝构件角色类

```
public class Composite implements Component {
    /**
     * 用来存储组合对象中包含的子组件对象
     */
    private List<Component> childComponents = new ArrayList<Component>()
;
    /**
     * 组合对象的名字
     */
    private String name;
    /**
     * 构造方法，传入组合对象的名字
     * @param name    组合对象的名字
     */
    public Composite(String name){
        this.name = name;
    }
    /**
     * 聚集管理方法，增加一个子构件对象
     * @param child 子构件对象
     */
    public void addChild(Component child){
        childComponents.add(child);
    }
    /**
     * 聚集管理方法，删除一个子构件对象
     * @param index 子构件对象的下标
     */
    public void removeChild(int index){
        childComponents.remove(index);
    }
    /**
     * 聚集管理方法，返回所有子构件对象
     */
    public List<Component> getChild(){
        return childComponents;
```

```
    }
    /**
     * 输出对象的自身结构
     * @param preStr 前缀，主要是按照层级拼接空格，实现向后缩进
     */
    @Override
    public void printStruct(String preStr) {
        // 先把自己输出
        System.out.println(preStr + "+" + this.name);
        //如果还包含有子组件，那么就输出这些子组件对象
        if(this.childComponents != null){
            //添加两个空格，表示向后缩进两个空格
            preStr += "  ";
            //输出当前对象的子对象
            for(Component c : childComponents){
                //递归输出每个子对象
                c.printStruct(preStr);
            }
        }

    }

}
```

树叶构件角色类

```
public class Leaf implements Component {
    /**
     * 叶子对象的名字
     */
    private String name;
    /**
     * 构造方法，传入叶子对象的名称
     * @param name 叶子对象的名字
     */
    public Leaf(String name){
        this.name = name;
    }
    /**
     * 输出叶子对象的结构，叶子对象没有子对象，也就是输出叶子对象的名字
```

```
     * @param preStr 前缀，主要是按照层级拼接的空格，实现向后缩进
     */
    @Override
    public void printStruct(String preStr) {
        // TODO Auto-generated method stub
        System.out.println(preStr + "-" + name);
    }

}
```

客户端类

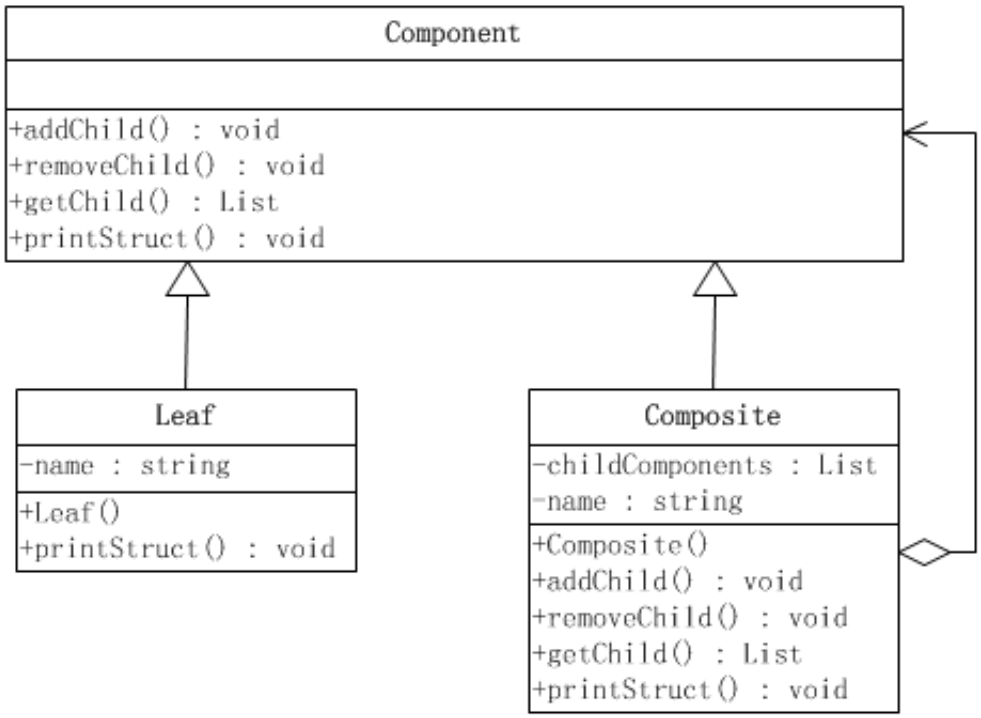
```
public class Client {
    public static void main(String[]args){
        Composite root = new Composite("服装");
        Composite c1 = new Composite("男装");
        Composite c2 = new Composite("女装");

        Leaf leaf1 = new Leaf("衬衫");
        Leaf leaf2 = new Leaf("夹克");
        Leaf leaf3 = new Leaf("裙子");
        Leaf leaf4 = new Leaf("套装");

        root.addChild(c1);
        root.addChild(c2);
        c1.addChild(leaf1);
        c1.addChild(leaf2);
        c2.addChild(leaf3);
        c2.addChild(leaf4);

        root.printStruct("");
    }
}
```

可以看出，树枝构件类(Composite)给出了addChild()、removeChild()以及getChild()等方法的声明和实现，而树叶构件类则没有给出这些方法的声明或实现。这样的做法是安全的做法，由于这个特点，客户端应用程序不可能错误地调用树叶构件的聚集方法，因为树叶构件没有这些方法，调用会导致编译错误。

安全式合成模式的缺点是不够透明，因为树叶类和树枝类将具有不同的接口。

透明式合成模式的结构

与安全式的合成模式不同的是，透明式的合成模式要求所有的具体构件类，不论树枝构件还是树叶构件，均符合一个固定接口。



源代码

抽象构件角色类

```
public abstract class Component {
    /**
     * 输出组建自身的名称
     */
    public abstract void printStruct(String preStr);
    /**
     * 聚集管理方法，增加一个子构件对象
     * @param child 子构件对象
     */
    public void addChild(Component child){
        /**
         * 缺省实现，抛出异常，因为叶子对象没有此功能
         * 或者子组件没有实现这个功能
         */
```

```
        throw new UnsupportedOperationException("对象不支持此功能");
    }
    /**
     * 聚集管理方法，删除一个子构件对象
     * @param index 子构件对象的下标
     */
    public void removeChild(int index){
        /**
         * 缺省实现，抛出异常，因为叶子对象没有此功能
         * 或者子组件没有实现这个功能
         */
        throw new UnsupportedOperationException("对象不支持此功能");
    }

    /**
     * 聚集管理方法，返回所有子构件对象
     */
    public List<Component> getChild(){
        /**
         * 缺省实现，抛出异常，因为叶子对象没有此功能
         * 或者子组件没有实现这个功能
         */
        throw new UnsupportedOperationException("对象不支持此功能");
    }
}
```

树枝构件角色类，此类将implements Conponent改为extends Conponent，其他地方无变化。

```
public class Composite extends Component {
    /**
     * 用来存储组合对象中包含的子组件对象
     */
    private List<Component> childComponents = new ArrayList<Component>()
;
    /**
     * 组合对象的名字
     */
    private String name;
    /**
```

```
     * 构造方法，传入组合对象的名字
     * @param name    组合对象的名字
     */
    public Composite(String name){
        this.name = name;
    }
    /**
     * 聚集管理方法，增加一个子构件对象
     * @param child 子构件对象
     */
    public void addChild(Component child){
        childComponents.add(child);
    }
    /**
     * 聚集管理方法，删除一个子构件对象
     * @param index 子构件对象的下标
     */
    public void removeChild(int index){
        childComponents.remove(index);
    }
    /**
     * 聚集管理方法，返回所有子构件对象
     */
    public List<Component> getChild(){
        return childComponents;
    }
    /**
     * 输出对象的自身结构
     * @param preStr 前缀，主要是按照层级拼接空格，实现向后缩进
     */
    @Override
    public void printStruct(String preStr) {
        // 先把自己输出
        System.out.println(preStr + "+" + this.name);
        //如果还包含有子组件，那么就输出这些子组件对象
        if(this.childComponents != null){
            //添加两个空格，表示向后缩进两个空格
            preStr += "  ";
            //输出当前对象的子对象
            for(Component c : childComponents){
                //递归输出每个子对象
```

```
                c.printStruct(preStr);
            }
        }

    }

}
```

树叶构件角色类，此类将implements Conponent改为extends Conponent，其他地方无变化。

```
public class Leaf extends Component {
    /**
     * 叶子对象的名字
     */
    private String name;
    /**
     * 构造方法，传入叶子对象的名称
     * @param name 叶子对象的名字
     */
    public Leaf(String name){
        this.name = name;
    }
    /**
     * 输出叶子对象的结构，叶子对象没有子对象，也就是输出叶子对象的名字
     * @param preStr 前缀，主要是按照层级拼接的空格，实现向后缩进
     */
    @Override
    public void printStruct(String preStr) {
        // TODO Auto-generated method stub
        System.out.println(preStr + "-" + name);
    }

}
```

客户端类的主要变化是不再区分Composite对象和Leaf对象。

```
public class Client {
    public static void main(String[]args){
        Component root = new Composite("服装");
```

```
        Component c1 = new Composite("男装");
        Component c2 = new Composite("女装");

        Component leaf1 = new Leaf("衬衫");
        Component leaf2 = new Leaf("夹克");
        Component leaf3 = new Leaf("裙子");
        Component leaf4 = new Leaf("套装");

        root.addChild(c1);
        root.addChild(c2);
        c1.addChild(leaf1);
        c1.addChild(leaf2);
        c2.addChild(leaf3);
        c2.addChild(leaf4);

        root.printStruct("");
    }
}
```

可以看出，客户端无需再区分操作的是树枝对象(Composite)还是树叶对象(Leaf)了；对于客户端而言，操作的都是Component对象。

两种实现方法的选择

这里所说的安全性合成模式是指：从客户端使用合成模式上看是否更安全，如果是安全的，那么就不会有发生误操作的可能，能访问的方法都是被支持的。

这里所说的透明性合成模式是指：从客户端使用合成模式上，是否需要区分到底是“树枝对象”还是“树叶对象”。如果是透明的，那就不用区分，对于客户而言，都是Compoent对象，具体的类型对于客户端而言是透明的，是无须关心的。

对于合成模式而言，在安全性和透明性上，会**更看重透明性**，毕竟合成模式的目的是：让客户端不再区分操作的是树枝对象还是树叶对象，而是以一个统一的方式来操作。

而且对于安全性的实现，需要区分是树枝对象还是树叶对象。有时候，需要将对象进行类型转换，却发现类型信息丢失了，只好强行转换，这种类型转换必然是不够安全的。

因此在使用合成模式的时候，建议多采用透明性的实现方式。