1 组合模式核心

- 组合模式将对象组合成 **树形结构**，以表示 **“整体与部分”** 的层次结构
- 组合模式包含3个成员：
    - **Leaf：** 叶节点，叶节点没有子节点，是整体与部分中的部分
    - **Composite：** 非叶节点(树枝节点或根节点)，存储子节点，是整体与部分的整体
    - **Component：** 抽象组件，Leaf 和 Composite 都继承于它，这样才使得用户对整体和部分的使用是一致的（用户并不知道哪个是整体，哪个是部分！）。它定义了所有子类公共的缺省行为

2 公司管理系统示例

2.1 Component(抽象公司)

- 抽象组件，定义所有行为，使得整体与部分对用户具有透明性

```
public abstract class AbstractCompany {
    protected String name;
    public AbstractCompany(String name) {
        this.name = name;
    }
    //定义默认方法，抛出异常
    public boolean add(AbstractCompany company) {
        throw new UnsupportedOperationException("不支持的操作");
    }
    public boolean remove(AbstractCompany company) {
        throw new UnsupportedOperationException("不支持的操作");
    }
    public void display(int depth) {
        throw new UnsupportedOperationException("不支持的操作");
    }
    public void LineOfDuty() {
        throw new UnsupportedOperationException("不支持的操作");
    }
}
```

2.2 Composite(整体)

- 整体继承于抽象公司，并实现了所有方法
- 在display、LineOfDuty方法中递归调用了子节点方法，使得用户对整体的操作就像是对部分的操作一样，具有透明性

```
public class CompositeCompany extends AbstractCompany {
    public CompositeCompany(String name) {
        super(name);
    }
    private List<AbstractCompany> children = new ArrayList<>();
    public  boolean add(AbstractCompany company) {
        return children.add(company);
    }
    public boolean remove(AbstractCompany company) {
        return children.remove(company);
    }
    public void display(int depth) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < depth; i++) {
            sb.append("-");
        }
        System.out.println(sb.toString() + name);
        //递归调用子节点
        for (AbstractCompany child : children) {
            child.display(depth + 2);
        }
    }
    public void LineOfDuty() {
        for (AbstractCompany child : children) {
            child.LineOfDuty();
        }
    }
}
```

2.3 Leaf(部分：具体公司)

- Leaf 也继承了抽象公司，且没有实现增加删除操作
- 使得用户调用 Leaf 的增加删除操作时返回异常

```
public class HRDepartment extends AbstractCompany {
    public HRDepartment(String name) {
        super(name);
    }
    public void display(int depth) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < depth; i++) {
            sb.append("-");
        }
        System.out.println(sb.toString() + name);
    }
    public void LineOfDuty() {
```

```
            System.out.println(name + "：员工招聘培训管理");
        }
    }
```

3 适用场景

- 想表示对象的整体与部分的层次结构时
- 当用户不关心使用的是组合对象还是单个对象时（这时用户的意图应该是使用他直接或间接手下的所有单个对象的相关行为，但他只会对他直接手下下命令）

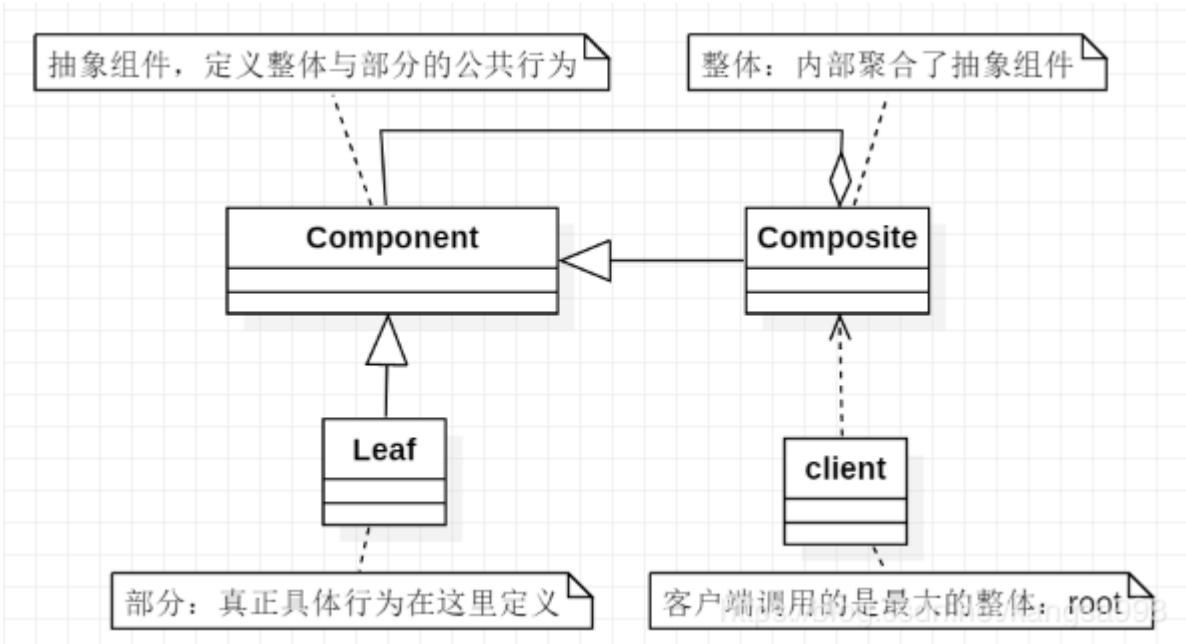
4 安全性与透明性的选择

- **透明性：** 在类层次的根部定义所有子类的接口方法具有良好的透明性（这样就可以一致性的使用所有的子类节点），但是用户可能会做出一些无意义的事情，如在 Leaf 中增加和删除对象，这样就没有安全性
- **安全性：** 在 Composite 类中定义单独的增删改操作具有安全性，这样用户就不可能调用到 Leaf 的增删改行为。但这样就失去了透明性，因为 Leaf 和 Composite 具有不同的接口了
- 一般情况下，都会选择透明性，在基类接口声明的行为中默认抛出异常，这样如果子类不重写它，在用户使用它时就会直接抛出异常

5 组合模式特点

- 部分可以被组合成整体，整体也可以被组合成更大的整体。非叶节点中存储的子节点即可以有叶节点，也可以有非叶节点。
- 用户可以一致的使用组合对象（非叶节点）和单个对象（叶节点）。（当使用组合对象非增删改查行为时，会自动的递归调用组合对象下的所有叶节点的对应行为）
- 可以很容易的添加或删除组件
- 要求较高的抽象性，如果节点和叶子有很多差异性的话，比如很多方法和属性都不一样，难以实现组合模式

6 UML 类图



参考

大话设计模式 Head First 设计模式 设计模式 github 源码地址