组合模式

题目链接

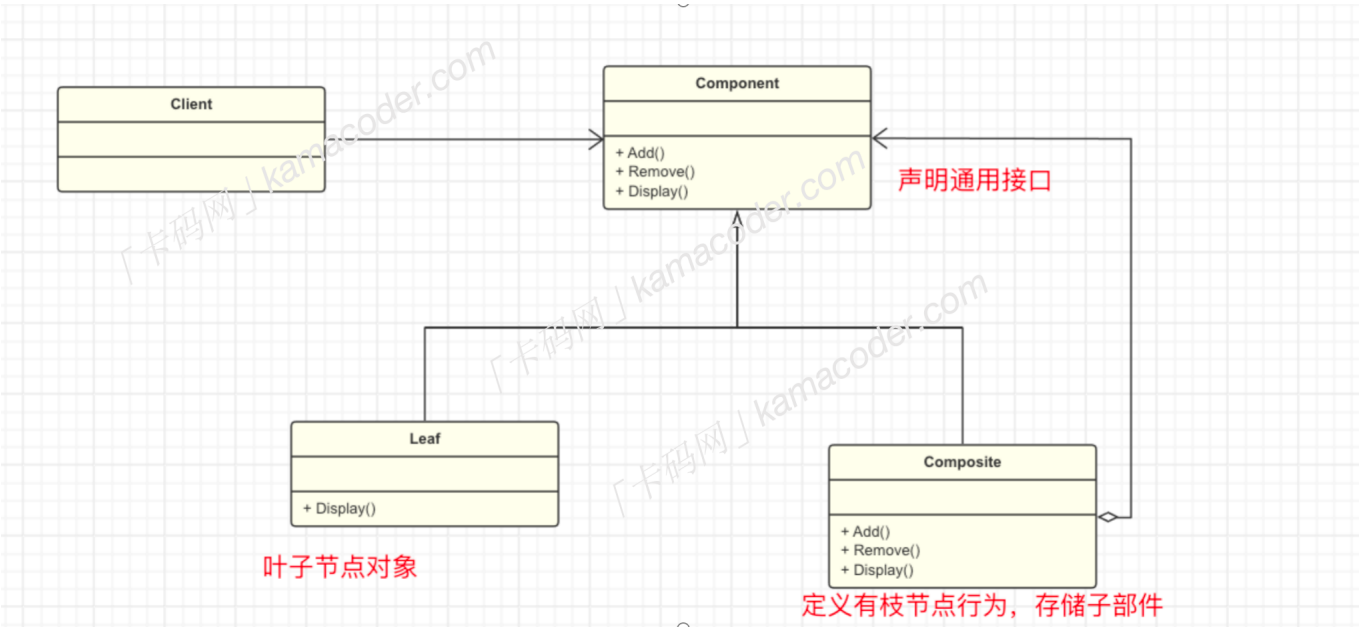
组合模式-公司组织架构

基本概念

组合模式是一种结构型设计模式，它**将对象组合成树状结构**来表示“部分-整体”的层次关系。组合模式使得客户端可以**统一处理单个对象和对象的组合**，而无需区分它们的具体类型。

基本结构

组合模式包括下面几个角色：



理解起来比较抽象，我们用“省份-城市”举个例子，省份中包含了多个城市，如果将之比喻成一个树形结构，城市就是叶子节点，它是省份的组成部分，而“省份”就是合成节点，可以包含其他城市，形成一个整体，省份和城市都是组件，它们都有一个共同的操作，比如获取信息。

- `Component`组件：组合模式的“根节点”，定义组合中所有对象的通用接口，可以是抽象类或接口。该类中定义了子类的共性内容。
- `Leaf`叶子：实现了Component接口的叶子节点，表示组合中的叶子对象，叶子节点没有子节点。
- `Composite`合成：作用是存储子部件，并且在Composite中实现了对子部件的相关操作，比如添加、删除、获取子组件等。

通过组合模式，整个省份的获取信息操作可以一次性地执行，而无需关心省份中的具体城市。这样就实现了对国家省份和城市的管理和操作。

简易实现

```
// 组件接口
interface Component {
    void operation();
}

// 叶子节点
class Leaf implements Component {
    @Override
    public void operation() {
        System.out.println("Leaf operation");
    }
}

// 组合节点：包含叶子节点的操作行为
class Composite implements Component {
    private List<Component> components = new ArrayList<>();

    public void add(Component component) {
        components.add(component);
    }

    public void remove(Component component) {
        components.remove(component);
    }

    @Override
    public void operation() {
        System.out.println("Composite operation");
        for (Component component : components) {
            component.operation();
        }
    }
}

// 客户端代码
public class Client {
    public static void main(String[] args) {
        // 创建叶子节点
        Leaf leaf = new Leaf();
        // 创建组合节点，并添加叶子节点
        Composite composite = new Composite();
```

```
        composite.add(leaf);

        composite.operation(); // 统一调用
    }
}
```

使用场景

组合模式可以使得客户端可以统一处理单个对象和组合对象，无需区分它们之间的差异，比如在图形编辑器中，图形对象可以是简单的线、圆形，也可以是复杂的组合图形，这个时候可以对组合节点添加统一的操作。

总的来说，组合模式适用于任何需要构建具有部分-整体层次结构的场景，比如组织架构管理、文件系统的文件和文件夹组织等。

本题代码

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

interface Component {
    void display(int depth);
}

class Department implements Component {
    private String name;
    private List<Component> children;

    public Department(String name) {
        this.name = name;
        this.children = new ArrayList<>();
    }

    public void add(Component component) {
        children.add(component);
    }

    @Override
    public void display(int depth) {
        StringBuilder indent = new StringBuilder();
        for (int i = 0; i < depth; i++) {
            indent.append("  ");
        }
        System.out.println(indent + name);
        for (Component component : children) {
```

```
            component.display(depth + 1);
        }
    }
}

class Employee implements Component {
    private String name;

    public Employee(String name) {
        this.name = name;
    }

    @Override
    public void display(int depth) {
        StringBuilder indent = new StringBuilder();
        for (int i = 0; i < depth; i++) {
            indent.append("  ");
        }
        System.out.println(indent + "  " + name);
    }
}

class Company {
    private String name;
    private Department root;

    public Company(String name) {
        this.name = name;
        this.root = new Department(name);
    }

    public void add(Component component) {
        root.add(component);
    }

    public void display() {
        System.out.println("Company Structure:");
        root.display(0);  // 从 1 开始，以适配指定的缩进格式
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 读取公司名称
```

```
        String companyName = scanner.nextLine();
        Company company = new Company(companyName);

        // 读取部门和员工数量
        int n = scanner.nextInt();
        scanner.nextLine();

        // 读取部门和员工信息
        for (int i = 0; i < n; i++) {
            String type = scanner.next();
            String name = scanner.nextLine().trim();

            if ("D".equals(type)) {
                Department department = new Department(name);
                company.add(department);
            } else if ("E".equals(type)) {
                Employee employee = new Employee(name);
                company.add(employee);
            }
        }

        // 输出公司组织结构
        company.display();
    }
}
```

其他语言版本

C++

```
#include <iostream>
#include <vector>
#include <sstream>

class Component {
public:
    virtual void display(int depth) = 0;
};

class Department : public Component {
private:
    std::string name;
    std::vector<Component*> children;

public:
```

```
    Department(const std::string& name) : name(name) {}

    void add(Component* component) {
        children.push_back(component);
    }

    void display(int depth) override {
        std::string indent(depth * 2, ' ');
        std::cout << indent << name << std::endl;
        for (Component* component : children) {
            component->display(depth + 1);
        }
    }
};

class Employee : public Component {
private:
    std::string name;

public:
    Employee(const std::string& name) : name(name) {}

    void display(int depth) override {
        std::string indent((depth + 1) * 2, ' ');
        std::cout << indent << name << std::endl;
    }
};

class Company {
private:
    std::string name;
    Department* root;

public:
    Company(const std::string& name) : name(name), root(new
Department(name)) {}

    void add(Component* component) {
        root->add(component);
    }

    void display() {
        std::cout << "Company Structure:" << std::endl;
        root->display(0);
    }
};
```

```cpp
int main() {
    std::string companyName;
    std::getline(std::cin, companyName);

    Company company(companyName);

    int n;
    std::cin >> n;
    std::cin.ignore();
    for (int i = 0; i < n; i++) {
        std::string type, name;
        std::cin >> type;
        std::getline(std::cin >> std::ws, name);

        if (type == "D") {
            Department* department = new Department(name);
            company.add(department);
        } else if (type == "E") {
            Employee* employee = new Employee(name);
            company.add(employee);
        }
    }

    company.display();

    return 0;
}
```

Python

```python
from typing import List

# 步骤1: 创建实现化接口
class Component:
    def display(self, depth: int):
        pass

# 步骤2: 创建具体实现化类
class Department(Component):
    def __init__(self, name: str):
        self.name = name
        self.children: List[Component] = []

    def add(self, component: Component):
        self.children.append(component)
```

```
    def display(self, depth: int):
        indent = "  " * depth
        print(indent + self.name)
        for component in self.children:
            component.display(depth + 1)

class Employee(Component):
    def __init__(self, name: str):
        self.name = name

    def display(self, depth: int):
        indent = "  " * depth
        print(indent + "  " + self.name)

class Company:
    def __init__(self, name: str):
        self.name = name
        self.root = Department(name)

    def add(self, component: Component):
        self.root.add(component)

    def display(self):
        print("Company Structure:")
        self.root.display(0)

if __name__ == "__main__":
    # 读取公司名称
    company_name = input()
    company = Company(company_name)

    # 读取部门和员工数量
    n = int(input())

    # 读取部门和员工信息
    for _ in range(n):
        type_str, name = input().split(maxsplit=1)

        if type_str == "D":
            department = Department(name.strip())
            company.add(department)
        elif type_str == "E":
            employee = Employee(name.strip())
            company.add(employee)
```

```
    # 输出公司组织结构
    company.display()
```

Go

```go
package main

import (
    "bufio"
    "fmt"
    "os"
    "strings"
)

// 步骤1: 创建组件接口
type Component interface {
    display(depth int)
}

// 步骤2: 创建部门类实现组件接口
type Department struct {
    name     string
    children []Component
}

func NewDepartment(name string) *Department {
    return &Department{
        name:     name,
        children: make([]Component, 0),
    }
}

func (d *Department) add(component Component) {
    d.children = append(d.children, component)
}

func (d *Department) display(depth int) {
    indent := strings.Repeat("  ", depth*2)
    fmt.Println(indent + d.name)
    for _, child := range d.children {
        child.display(depth + 1)
    }
}

// 步骤3: 创建员工类实现组件接口
type Employee struct {
```

```
    name string
}

func NewEmployee(name string) *Employee {
    return &Employee{
        name: name,
    }
}

func (e *Employee) display(depth int) {
    indent := strings.Repeat("  ", depth*2)
    fmt.Println(indent + "  " + e.name)
}

// 步骤4: 创建公司类
type Company struct {
    name string
    root *Department
}

func NewCompany(name string) *Company {
    return &Company{
        name: name,
        root: NewDepartment(name),
    }
}

func (c *Company) add(component Component) {
    c.root.add(component)
}

func (c *Company) display() {
    fmt.Println("Company Structure:")
    c.root.display(0) // 从 0 开始，以适配指定的缩进格式
}

func main() {
    scanner := bufio.NewScanner(os.Stdin)

    // 读取公司名称
    scanner.Scan()
    companyName := scanner.Text()
    company := NewCompany(companyName)

    // 读取部门和员工数量
    scanner.Scan()
```

```
    n := 0
    fmt.Sscanf(scanner.Text(), "%d", &n)

    // 读取部门和员工信息
    var currentDepartment *Department

    for i := 0; i < n; i++ {
        scanner.Scan()
        line := scanner.Text()
        fields := strings.Fields(line)

        if len(fields) < 2 {
            continue
        }

        typeStr := fields[0]
        name := strings.Join(fields[1:], " ")

        if typeStr == "D" {
            department := NewDepartment(name)
            company.add(department)
            currentDepartment = department
        } else if typeStr == "E" {
            employee := NewEmployee(name)
            currentDepartment.add(employee)
        }
    }

    // 输出公司组织结构
    company.display()
}
```