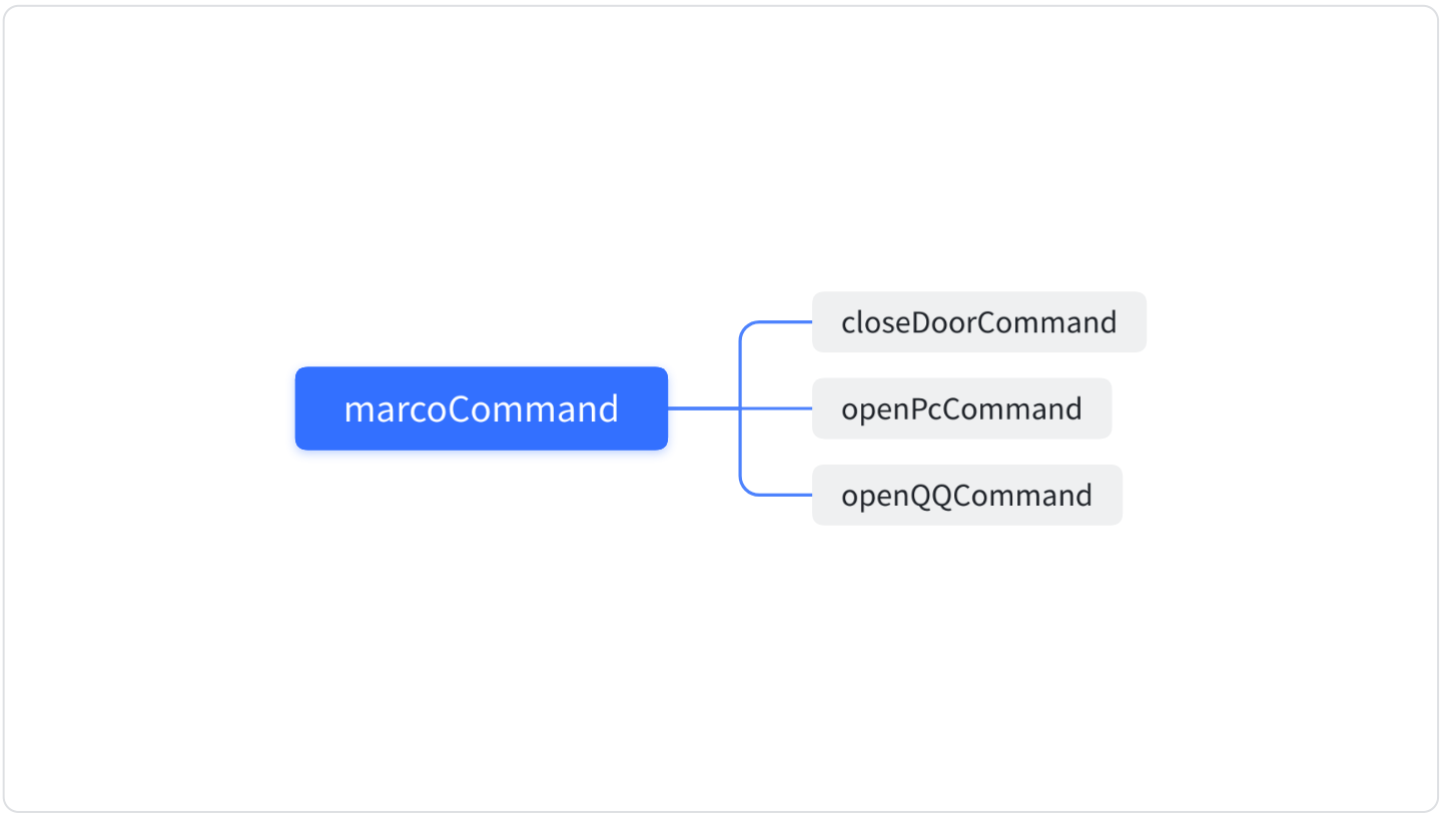
10 组合模式

其实程序设计跟自然一样，也存在“事物是由相似的子事物构成”的思想，例如原子和太阳系的组成一样。组合模式就是借助小的子对象来构建更大的对象

回顾宏命令

之前第九章的宏命令就是一组具体的子命令集合对象，但是不管宏命令自身还是子命令都拥有关键的execute方法，并且如果还有孙命令的话也是同样的道理，最终形成了一级一级的树状结构



marcoCommand虽然变现的像是一个命令，但其实只有内部子命令的代理，只负责传递请求给叶对象

用途

1. 表示树形结构：回顾上面的例子即可，组合模式提供了一种遍历树形结构的方案，例如递归调用叶对象的execute方法来完成抽象操作的实现。非常方便的部署对象部分->整体的层次结构
2. 利用对象多态性统一组合对象和单个对象：利用对象多态性使得客户端可以忽略组合对象和单个对象的不同，对于用户来说透明（上面例子的execute），并不需要关心它究竟是组合对象还是单个对象。例如用户使用遥控器不需要关注它的命令内部实现或者是宏命令还是单个命令，命令只要有execute方法即可嵌入遥控器

更强大的宏命令

这里我们实现一个更强大的遥控器，同时了解请求在复杂组合树中的传递过程：

- 打开空调
- 打开电视和音响
- 关门，开电脑，登录QQ

```
/**
  * 更强大宏命令
  */
 let closeDoorCommand = {
    execute: () => {
        console.log('关门');
    }
}

let openPCCommand = {
    execute: () => {
        console.log('打开电脑');
    }
}

let openQQCommand = {
    execute: () => {
        console.log('打开QQ');
    }
}

let openAcCommand = {
    execute: () => {
        console.log('开空调');
    }
}

let openTvCommand = {
    execute: () => {
        console.log('打开电视');
    }
}

let openSoundCommand = {
    execute: () => {
        console.log('打开音响');
    }
```

```
}

const MacroCommand = () => {
    return {
        commandsList: [],
        add: (command) => {
            this.commandsList.push(command);
        },
        execute: () => {
            for(let i = 0, command; command = this.commandsList[i++];){
                command.execute();
            }
        }
    }
}

const macroCommand1 = MacroCommand();
macroCommand1.add(openTvCommand);
macroCommand1.add(openSoundCommand);

const macroCommand2 = MacroCommand();
macroCommand.add(closeDoorCommand);
macroCommand.add(openPCCommand);
macroCommand.add(openQQCommand);
macroCommand.execute();

const macroCommand = MacroCommand();
macroCommand.add(openAcCommand);
macroCommand.add(macroCommand1);
macroCommand.add(macroCommand2);

// 给遥控器绑定超级命令
var setCommand = (function(command){
    document.getElementById('button').onclick = function(){
        command.execute();
    }
})(macroCommand)
```

最终这个复杂的宏命令组合树的运转非常简单，只需要调用最上层的execute即可，创建组合对象的程序员也不需要关心内在的execute细节，同时根据需求往里面更改节点对象也非常方便

抽象类在组合模式的作用

前面提到组合模式最大优点可以一致地对待组合对象和基本对象，只要它是一个命令，它有execute方法就可以。

这种透明性带来的便利，在静态类型语言中尤为明显。例如Java中的关键就是他们必须继承一个抽象类，这个抽象类即代表组合对象又代表叶对象，保证拥有同样名字的方法以及相同的反馈。

```
public abstract class Component {
    public void add(Component child){}

    public void remove(Component child){}
}
// 具体类
public class Composite extends Component {
    public void add(Component child){}

    public void remove(Component child){}
}

public class Leaf extends Component {
    public void add(Component child){throw new UnsupoortedOperationException()}

    public void remove(Component child){}
}
```

然而在JS这种动态语言中对象的多态性是与生俱来的，没有编译器去检查（现在可以通过TS来实现），一般可以用鸭子类型的思想对他们检查

缺点是不严谨，不太安全，但是好处是快速且自由

透明性带来的安全问题

组合模式的透明性使得用户发起请求时不需要关注组合对象和叶子对象的区别，但他们本质上是有区别的。

例如我们可能会试图往叶子对象里面添加子节点，针对这种误操作，一种方案是给叶对象也统一增加add方法，并且在具体实现中抛出一个异常来提醒用户。

```
let openSoundCommand = {
    execute: () => {
        console.log('打开音响');
    },
    add: () => {throw new Error('叶对象不能添加子节点')}
}
```

组合模式的例子-扫描文件夹

文件夹和文件之间的关系非常适合用组合模式来描述，组合模式在文件夹的应用中有如下好处：

- 在用户进行各种诸如复制，粘贴操作时，不需要考虑这批文件的类型，也不需要关系他们是文件还是文件夹，直接使用Ctrl+C与Ctrl+V的统一操作即可
- 当使用杀毒软件等工具尝试扫描文件夹时，也不需要关心有多少文件和子文件夹，只需要操作最外层的文件夹扫描即可

我们看一下实际的代码：

```
/**
 * 组合模式实现扫描文件夹
 */
// 定义Folder与File类
class Folder {
    constructor(name) {
        this.name = name;
        this.files = [];
    }
    add(file) {
        this.files.push(file);
    }
    scan() {
        for (let i = 0, file, files = this.files; files = files[i++];) {
            file.scan();
        }
    }
}

class File {
    constructor(name) {
        this.name = name;
    }
    add(file) {
        throw new Error('文件下面不能添加文件')
    }
    scan() {
        for (let i = 0, file, files = this.files; files = files[i++];) {
            file.scan();
        }
    }
}

// 创建实例，组合
const folder = new Folder('学习资料');
const folder1 = new Folder('JS');

```

```
const file1 = new File('JS设计模式');
const file2 = new File('精通Jquery');

// 用户操作对象，无需分辨是文件还是文件夹，很容易添加到整个组合树中 --- 符合开闭原则
folder1.add(file1);
folder1.add(file2);
folder.add(folder1);

// 扫描操作也很简单
folder.scan()
```

一些值得注意的地方

组合模式不是父子关系

它的树型结构并不意味着组合对象和叶对象是父子关系，他们只是一种HAS-A的聚合关系，而非IS-A的关系，Composite并非Leaf的父类

对叶对象操作的一致性

组合模式除了要求组合对象和叶对象有相同的接口之外，还有一个必要条件，就是一组叶对象的操作必须具有一致性。

例如针对公司所有员工发放过节费，可以运用组合模式，但是只是针对今天过生日的发送邮件就不行了。

双向映射关系

上述发放过节费运用组合模式可以从公司 -> 部门 -> 小组 -> 员工，但是如果员工可能从属多个组织结构，这样对象之间关系不符合严格意义的层次结构，这个时候不适用组合模式，否则员工可能收到两份过节费

此时可以尝试引入中介者模式来管理小组和员工的相互引用关系

用职责链模式提高组合模式性能

如果树的结构复杂且节点很多，遍历时也许性能会不太好。

有一种方案是利用职责链模式，组合模式中不需要手动设置职责链，这个基于父子层次结构是天然的，然后让请求顺着链条从父到子传递，直到遇到可以处理该请求的对象为止，这也是职责链的经典应用