

# 53 | 组合模式：如何设计实现支持递归遍历的文件系统目录树结构？

王争 2020-03-04



结构型设计模式就快要讲完了，还剩下两个不那么常用的：组合模式和享元模式。今天，我们来讲一下**组合模式**（Composite Design Pattern）。

组合模式跟我们之前讲的面向对象设计中的“组合关系（通过组合来组装两个类）”，完全是两码事。这里讲的“组合模式”，主要是用来处理树形结构数据。这里的“数据”，你可以简单理解为一组对象集合，待会我们会详细讲解。

正因为其应用场景的特殊性，数据必须能表示成树形结构，这也导致了这种模式在实际的项目开发中并不那么常用。但是，一旦数据满足树形结构，应用这种模式就能发挥很大的作用，能让代码变得非常简洁。

话不多说，让我们正式开始今天的学习吧！

## 组合模式的原理与实现

在 GoF 的《设计模式》一书中，组合模式是这样定义的：

Compose objects into tree structure to represent part-whole hierarchies. Composite lets client treat individual objects and compositions of objects uniformly.

翻译成中文就是：将一组对象组织（Compose）成树形结构，以表示一种“部分 - 整体”的层次结构。组合让客户端（在很多设计模式书籍中，“客户端”代指代码的使用者。）可以统一单个对象和组合对象的处理逻辑。

接下来，对于组合模式，我举个例子来给你解释一下。

假设我们有这样一个需求：设计一个类来表示文件系统中的目录，能方便地实现下面这些功能：

- 动态地添加、删除某个目录下的子目录或文件；
- 统计指定目录下的文件个数；
- 统计指定目录下的文件总大小。

我这里给出了这个类的骨架代码，如下所示。其中的核心逻辑并未实现，你可以试着自己去补充完整，再来看我的讲解。在下面的代码实现中，我们把文件和目录统一用 `FileSystemNode` 类来表示，并且通过 `isFile` 属性来区分。

复制代码

```
1 public class FileSystemNode {
2     private String path;
3     private boolean isFile;
4     private List<FileSystemNode> subNodes = new ArrayList<>();
5
6     public FileSystemNode(String path, boolean isFile) {
7         this.path = path;
8         this.isFile = isFile;
9     }
10
11     public int countNumOfFiles() {
12         // TODO:...
13     }
14
15     public long countSizeOfFiles() {
16         // TODO:...
17     }
18
19     public String getPath() {
20         return path;
21     }
22
23     public void addSubNode(FileSystemNode fileOrDir) {
24         subNodes.add(fileOrDir);
25     }
26
27     public void removeSubNode(FileSystemNode fileOrDir) {
28         int size = subNodes.size();
29         int i = 0;
30         for (; i < size; ++i) {
31             if (subNodes.get(i).getPath().equalsIgnoreCase(fileOrDir.getPath())) {
32                 break;
33             }
34         }
35         if (i < size) {
36             subNodes.remove(i);
37         }
38     }
39 }
```

实际上，如果你看过我的《数据结构与算法之美》专栏，想要补全其中的 `countNumOfFiles()` 和 `countSizeOfFiles()` 这两个函数，并不是件难事，实际上这就是树上的递归遍历算法。对于文件，我们直接返回文件的个数（返回 1）或大小。对于目录，我们遍历目录中每个子目录或者文件，递归计算它们的个数或大小，然后求和，就是这个目录下的文件个数和文件大小。

我把两个函数的代码实现贴在下面了，你可以对照着看一下。

复制代码

```
1 public int countNumOfFiles() {
2     if (isFile) {
3         return 1;
4     }
5     int numOfFiles = 0;
6     for (FileSystemNode fileOrDir : subNodes) {
7         numOfFiles += fileOrDir.countNumOfFiles();
8     }
9     return numOfFiles;
10 }
11
12 public long countSizeOfFiles() {
13     if (isFile) {
14         File file = new File(path);
15         if (!file.exists()) return 0;
16         return file.length();
17     }
18     long sizeofFiles = 0;
19     for (FileSystemNode fileOrDir : subNodes) {
20         sizeofFiles += fileOrDir.countSizeOfFiles();
21     }
22     return sizeofFiles;
23 }
```

单纯从功能实现角度来说，上面的代码没有问题，已经实现了我们想要的功能。但是，如果我们开发的是一个大型系统，从扩展性（文件或目录可能会对对应不同的操作）、业务建模（文件和目录从业务上是两个概念）、代码的可读性（文件和目录区分对待更加符合人们对业务的认知）的角度来说，我们最好对文件和目录进行区分设计，定义为 `File` 和 `Directory` 两个类。

按照这个设计思路，我们对代码进行重构。重构之后的代码如下所示：

复制代码

```
1 public abstract class FileSystemNode {
2     protected String path;
3
4     public FileSystemNode(String path) {
5         this.path = path;
6     }
7
8     public abstract int countNumOfFiles();
9     public abstract long countSizeOfFiles();
10
11     public String getPath() {
12         return path;
13     }
14 }
15
16 public class File extends FileSystemNode {
17     public File(String path) {
18         super(path);
19     }
20
21     @Override
22     public int countNumOfFiles() {
23         return 1;
24     }
25
26     @Override
27     public long countSizeOfFiles() {
28         java.io.File file = new java.io.File(path);
29         if (!file.exists()) return 0;
30         return file.length();
31     }
32 }
33
34 public class Directory extends FileSystemNode {
35     private List<FileSystemNode> subNodes = new ArrayList<>();
36
37     public Directory(String path) {
38         super(path);
39     }
40
41     @Override
42     public int countNumOfFiles() {
43         int numOfFiles = 0;
44         for (FileSystemNode fileOrDir : subNodes) {
45             numOfFiles += fileOrDir.countNumOfFiles();
46         }
47         return numOfFiles;
48     }
49
50     @Override
51     public long countSizeOfFiles() {
52         long sizeofFiles = 0;
53         for (FileSystemNode fileOrDir : subNodes) {
54             sizeofFiles += fileOrDir.countSizeOfFiles();
55         }
56         return sizeofFiles;
57     }
58
59     public void addSubNode(FileSystemNode fileOrDir) {
60         subNodes.add(fileOrDir);
61     }
62
63     public void removeSubNode(FileSystemNode fileOrDir) {
64         int size = subNodes.size();
65         int i = 0;
66         for (; i < size; ++i) {
67             if (subNodes.get(i).getPath().equalsIgnoreCase(fileOrDir.getPath())) {
68                 break;
69             }
70         }
71         if (i < size) {
72             subNodes.remove(i);
73         }
74     }
75 }
```

文件和目录类都设计好了，我们来看，如何用它们来表示一个文件系统中的目录树结构。具体的代码示例如下所示：

复制代码

```
1 public class Demo {
2     public static void main(String[] args) {
3         /**
4          * /
5          * /wz/
6          * /wz/a.txt
7          * /wz/b.txt
8          * /wz/movies/
9          * /wz/movies/c.avi
10         * /xzg/
11         * /xzg/docs/
12         * /xzg/docs/d.txt
13         */
14         Directory fileSystemTree = new Directory("/");
15         Directory node_wz = new Directory("/wz/");
16         Directory node_xzg = new Directory("/xzg/");
17         fileSystemTree.addSubNode(node_wz);
18         fileSystemTree.addSubNode(node_xzg);
19
20         File node_wz_a = new File("/wz/a.txt");
21         File node_wz_b = new File("/wz/b.txt");
22         Directory node_wz_movies = new Directory("/wz/movies/");
23         node_wz.addSubNode(node_wz_a);
24         node_wz.addSubNode(node_wz_b);
25         node_wz.addSubNode(node_wz_movies);
26
27         File node_wz_movies_c = new File("/wz/movies/c.avi");
28         node_wz_movies.addSubNode(node_wz_movies_c);
29
30         Directory node_xzg_docs = new Directory("/xzg/docs/");
31         node_xzg.addSubNode(node_xzg_docs);
32
33         File node_xzg_docs_d = new File("/xzg/docs/d.txt");
34         node_xzg_docs.addSubNode(node_xzg_docs_d);
35
36         System.out.println("/ files num:" + fileSystemTree.countNumOfFiles());
37         System.out.println("/wz/ files num:" + node_wz.countNumOfFiles());
38     }
39 }
```

我们对照着这个例子，再重新看一下组合模式的定义：“将一组对象（文件和目录）组织成树形结构，以表示一种‘部分 - 整体’的层次结构（目录与子目录的嵌套结构）。组合模式让客户端可以统一单个对象（文件）和组合对象（目录）的处理逻辑（递归遍历）。”

实际上，刚才讲的这种组合模式的设计思路，与其说是一种设计模式，倒不如说是对业务场景的一种数据结构和算法的抽象。其中，数据可以表示成树这种数据结构，业务需求可以通过在树上的递归遍历算法来实现。

## 组合模式的应用场景举例

刚刚我们讲了文件系统的例子，对于组合模式，我这里再举一个例子。搞懂了这两个例子，你基本上就算掌握了组合模式。在实际的项目中，遇到类似的可以表示成树形结构的业务场景，你只要“照葫芦画瓢”去设计就可以了。

假设我们在开发一个 OA 系统（办公自动化系统）。公司的组织结构包含部门和员工两种数据类型。其中，部门又可以包含子部门和员工。在数据库中的表结构如下所示：

部门表（Department）				
部门ID	隶属上级部门ID	.....	.....	.....
id	parent_department_id	.....	.....	.....

员工表（Employee）				
员工ID	隶属上级部门ID	员工薪资	.....	.....
id	department_id	salary	.....	.....

极客时间

我们希望在内存中构建整个公司的人员架构图（部门、子部门、员工的隶属关系），并且提供接口计算出部门的薪资成本（隶属于这个部门的所有员工的薪资和）。

部门包含子部门和员工，这是一种嵌套结构，可以表示成树这种数据结构。计算每个部门的薪资开支这样一个需求，也可以通过在树上的遍历算法来实现。所以，从这个角度来看，这个应用场景可以使用组合模式来设计和实现。

这个例子的代码结构跟上一个例子的很相似，代码实现我直接贴在了下面，你可以对比着看一下。其中，`HumanResource` 是部门类（`Department`）和员工类（`Employee`）抽象出来的父类，为的是能统一薪资的处理逻辑。`Demo` 中的代码负责从数据库中读取数据并在内存中构建组织架构图。

复制代码

```
1 public abstract class HumanResource {
2     protected long id;
3     protected double salary;
4
5     public HumanResource(long id) {
6         this.id = id;
7     }
8
9     public long getId() {
10         return id;
11     }
12
13     public abstract double calculateSalary();
14 }
15
16 public class Employee extends HumanResource {
17     public Employee(long id, double salary) {
18         super(id);
19         this.salary = salary;
20     }
21
22     @Override
23     public double calculateSalary() {
24         return salary;
25     }
26 }
27
28 public class Department extends HumanResource {
29     private List<HumanResource> subNodes = new ArrayList<>();
30
31     public Department(long id) {
32         super(id);
33     }
34
35     @Override
36     public double calculateSalary() {
37         double totalSalary = 0;
38         for (HumanResource hr : subNodes) {
39             totalSalary += hr.calculateSalary();
40         }
41         this.salary = totalSalary;
42         return totalSalary;
43     }
44
45     public void addSubNode(HumanResource hr) {
46         subNodes.add(hr);
47     }
48 }
49
50 // 构建组织架构的代码
51 public class Demo {
52     private static final long ORGANIZATION_ROOT_ID = 1001;
53     private DepartmentRepo departmentRepo; // 依赖注入
54     private EmployeeRepo employeeRepo; // 依赖注入
55
56     public void buildOrganization() {
57         Department rootDepartment = new Department(ORGANIZATION_ROOT_ID);
58         buildOrganization(rootDepartment);
59     }
60
61     private void buildOrganization(Department department) {
62         List<Long> subDepartmentIds = departmentRepo.getSubDepartmentIds(department.getId());
63         for (Long subDepartmentId : subDepartmentIds) {
64             Department subDepartment = new Department(subDepartmentId);
65             department.addSubNode(subDepartment);
66             buildOrganization(subDepartment);
67         }
68         List<Long> employeeIds = employeeRepo.getDepartmentEmployeeIds(department.getId());
69         for (Long employeeId : employeeIds) {
70             double salary = employeeRepo.getEmployeeSalary(employeeId);
71             department.addSubNode(new Employee(employeeId, salary));
72         }
73     }
74 }
```

我们再拿组合模式的定义跟这个例子对照一下：“将一组对象（员工和部门）组织成树形结构，以表示一种‘部分 - 整体’的层次结构（部门与子部门的嵌套结构）。组合模式让客户端可以统一单个对象（员工）和组合对象（部门）的处理逻辑（递归遍历）。”

## 重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

组合模式的设计思路，与其说是一种设计模式，倒不如说是对业务场景的一种数据结构和算法的抽象。其中，数据可以表示成树这种数据结构，业务需求可以通过在树上的递归遍历算法来实现。

组合模式，将一组对象组织成树形结构，将单个对象和组合对象都看做树中的节点，以统一处理逻辑，并且它利用树形结构的特点，递归地处理每个子树，依次简化代码实现。使用组合模式的前提在于，你的业务场景必须能够表示成树形结构。所以，组合模式的应用场景也比较局限，它并不是一种很常用的设计模式。

## 课堂讨论

在文件系统那个例子中，`countNumOfFiles()` 和 `countSizeOfFiles()` 这两个函数实现的效率并不高，因为每次调用它们的时候，都要重新遍历一遍子树。有没有什么办法可以提高这两个函数的执行效率呢（注意：文件系统还会涉及频繁的删除、添加文件操作，也就是对应 `Directory` 类中的 `addSubNode()` 和 `removeSubNode()` 函数）？

欢迎留言和我分享你的想法。如果有收获，也欢迎你把这篇文章分享给你的朋友。

## 本周热门直播

- 没有代码洁癖的程序员，是不是好程序员？
- 如何成为一名“面霸”？
- 大厂面试问的那些冷门问题，在工作中真就不会用到吗？
- 如何才能学好纷繁复杂的 Spring 技术栈？
- 别焦虑，你得想自己怎么做才能成为“团队骨干”

微信扫码，进入直播观众席>>>

