53讲组合模式：如何设计实现支持递归遍历的文件系统目录树结构

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。

```
public class FileSystemNode {
  private String path;
  private boolean isFile;
  private List<FileSystemNode> subNodes = new ArrayList<>();

  public FileSystemNode(String path, boolean isFile) {
    this.path = path;
    this.isFile = isFile;
  }

  public int countNumOfFiles() {
    // TODO:...
  }

  public long countSizeOfFiles() {
    // TODO:...
  }

  public String getPath() {
    return path;
  }

  public void addSubNode(FileSystemNode fileOrDir) {
    subNodes.add(fileOrDir);
  }

  public void removeSubNode(FileSystemNode fileOrDir) {
    int size = subNodes.size();
    int i = 0;
    for (; i < size; ++i) {
      if (subNodes.get(i).getPath().equalsIgnoreCase(fileOrDir.getPath())) {
        break;
      }
    }
    if (i < size) {
      subNodes.remove(i);
    }
  }
}
```

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。

```
  public int countNumOfFiles() {
    if (isFile) {
      return 1;
    }
    int numOfFiles = 0;
    for (FileSystemNode fileOrDir : subNodes) {
      numOfFiles += fileOrDir.countNumOfFiles();
    }
    return numOfFiles;
  }

  public long countSizeOfFiles() {
    if (isFile) {
      File file = new File(path);
      if (!file.exists()) return 0;
      return file.length();
    }
    long sizeofFiles = 0;
    for (FileSystemNode fileOrDir : subNodes) {
      sizeofFiles += fileOrDir.countSizeOfFiles();
    }
    return sizeofFiles;
  }
```

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：

```
public abstract class FileSystemNode {
  protected String path;

  public FileSystemNode(String path) {
    this.path = path;
  }

  public abstract int countNumOfFiles();
```

```
  public abstract long countSizeOfFiles();

  public String getPath() {
    return path;
  }
}

public class File extends FileSystemNode {
  public File(String path) {
    super(path);
  }

  @Override
  public int countNumOfFiles() {
    return 1;
  }

  @Override
  public long countSizeOfFiles() {
    java.io.File file = new java.io.File(path);
    if (!file.exists()) return 0;
    return file.length();
  }
}

public class Directory extends FileSystemNode {
  private List<FileSystemNode> subNodes = new ArrayList<>();

  public Directory(String path) {
    super(path);
  }

  @Override
  public int countNumOfFiles() {
    int numOfFiles = 0;
    for (FileSystemNode fileOrDir : subNodes) {
      numOfFiles += fileOrDir.countNumOfFiles();
    }
    return numOfFiles;
  }

  @Override
  public long countSizeOfFiles() {
    long sizeofFiles = 0;
```

```
    for (FileSystemNode fileOrDir : subNodes) {
      sizeofFiles += fileOrDir.countSizeOfFiles();
    }
    return sizeofFiles;
  }

  public void addSubNode(FileSystemNode fileOrDir) {
    subNodes.add(fileOrDir);
  }

  public void removeSubNode(FileSystemNode fileOrDir) {
    int size = subNodes.size();
    int i = 0;
    for (; i < size; ++i) {
      if (subNodes.get(i).getPath().equalsIgnoreCase(fileOrDir.getPath())) {
        break;
      }
    }
    if (i < size) {
      subNodes.remove(i);
    }
  }
}
```

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：

```
public class Demo {
  public static void main(String[] args) {
    /**
     * /
     * /wz/
     * /wz/a.txt
     * /wz/b.txt
     * /wz/movies/
     * /wz/movies/c.avi
     * /xzg/
     * /xzg/docs/
     * /xzg/docs/d.txt
     */
    Directory fileSystemTree = new Directory("/");
    Directory node_wz = new Directory("/wz/");
    Directory node_xzg = new Directory("/xzg/");
    fileSystemTree.addSubNode(node_wz);
    fileSystemTree.addSubNode(node_xzg);

    File node_wz_a = new File("/wz/a.txt");
    File node_wz_b = new File("/wz/b.txt");
    Directory node_wz_movies = new Directory("/wz/movies/");
    node_wz.addSubNode(node_wz_a);
    node_wz.addSubNode(node_wz_b);
    node_wz.addSubNode(node_wz_movies);

    File node_wz_movies_c = new File("/wz/movies/c.avi");
    node_wz_movies.addSubNode(node_wz_movies_c);

    Directory node_xzg_docs = new Directory("/xzg/docs/");
    node_xzg.addSubNode(node_xzg_docs);

    File node_xzg_docs_d = new File("/xzg/docs/d.txt");
    node_xzg_docs.addSubNode(node_xzg_docs_d);

    System.out.println("/ files num:" + fileSystemTree.countNumOfFiles());
    System.out.println("/wz/ files num:" + node_wz.countNumOfFiles());
  }
}
```

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	……	……

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。

```
public abstract class HumanResource {
  protected long id;
  protected double salary;

  public HumanResource(long id) {
    this.id = id;
  }
```

```
  public long getId() {
    return id;
  }

  public abstract double calculateSalary();
}

public class Employee extends HumanResource {
  public Employee(long id, double salary) {
    super(id);
    this.salary = salary;
  }

  @Override
  public double calculateSalary() {
    return salary;
  }
}

public class Department extends HumanResource {
  private List<HumanResource> subNodes = new ArrayList<>();

  public Department(long id) {
    super(id);
  }

  @Override
  public double calculateSalary() {
    double totalSalary = 0;
    for (HumanResource hr : subNodes) {
      totalSalary += hr.calculateSalary();
    }
    this.salary = totalSalary;
    return totalSalary;
  }

  public void addSubNode(HumanResource hr) {
    subNodes.add(hr);
  }
}

// 构建组织架构的代码
public class Demo {
```

```
  private static final long ORGANIZATION_ROOT_ID = 1001;
  private DepartmentRepo departmentRepo; // 依赖注入
  private EmployeeRepo employeeRepo; // 依赖注入

  public void buildOrganization() {
    Department rootDepartment = new Department(ORGANIZATION_ROOT_ID);
    buildOrganization(rootDepartment);
  }

  private void buildOrganization(Department department) {
    List<Long> subDepartmentIds = departmentRepo.getSubDepartmentIds(department.getId());
    for (Long subDepartmentId : subDepartmentIds) {
      Department subDepartment = new Department(subDepartmentId);
      department.addSubNode(subDepartment);
      buildOrganization(subDepartment);
    }
    List<Long> employeeIds = employeeRepo.getDepartmentEmployeeIds(department.getId());
    for (Long employeeId : employeeIds) {
      double salary = employeeRepo.getEmployeeSalary(employeeId);
      department.addSubNode(new Employee(employeeId, salary));
    }
  }
}
```

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。

精选留言

test

把计算逻辑放在addSubNode和removeSubNode里面

2020-03-04 09:02

八戒

课堂讨论

可以把计算文件数量和大小的逻辑抽出来，定义两个成员变量文件大小和文件数量；

在每次addSubNode()和removeSubNode()的时候去调用计算逻辑，更新文件大小和文件数量；

这样在调用countNumOfFiles和countSizeOfFiles的时候直接返回我们的成员变量就好了；

当然如果这么做的话，那countNumOfFiles和countSizeOfFiles这两个方法的名字也不合适了，应该叫numOfFiles和sizeOfFiles

2020-03-04 08:04

辣么大

我想的一个思路是：每个节点新增一个field：parent，父链接指向它的上层节点，同时增加字段numOfFiles，sizeOfFiles。对于File节点：numOfFiles=1， sizeOfFiles=它自己的大小。对于Directory节点，是其子节点的和。删除、增加subnode时，只需要从下向上遍历一个节点的parent link，修改numOfFiles和sizeOfFiles。这样的话删除、新增subnode修改值的复杂度为树的深度，查询返回numOfFiles和sizeOfFiles复杂度为O(1)。

2020-03-04 08:06

小晏子

Directory中缓存子节点数量和大小的信息，每次addSubNode和removeSubNode时，失效缓存的节点数量和大小的信息，这样每次查询的时候，如果缓存的信息有效，那么就直接返回，反之就遍历一遍，有点类似于数据库和cache数据同步的cache-aside方式，另外如果file本身大小如果有变化，也要有办法去失效Directory中的缓存信息，这就需要实现新的接口通知机制。

2020-03-04 09:51

小兵

课堂讨论首先想到了使用缓存，对于一个文件系统来说，文件的数量应当远高于文件夹的数量，可以在文件夹类增加一个成员变量，维护该层级下的文件数量，遍历的时候只需要遍历文件夹就可以了。

2020-03-04 07:43

唐朝农民

那个算薪资的在实际生产中也不回这么用吧，虽然使用设计模式提高代码的可扩展性，但是需要循环，递归调用数据仓储层，如果员工一多肯定造成很大的性能影响，这也是我经常纠结的地方，有个时候为了减少访问数据库的次数，而不得不放弃更优雅的代码，请问这种情况该怎么破?

2020-03-04 19:56

Wh1

重构之后的FileSystemNode的子类Directory中的递归方法 countNumOfFiles() 是不是少了结束判断语句?

2020-03-04 16:41

南山

真的是没有最适合，只有更适合

实际工作中碰到过一个场景需要抽象出条件和表达式来解决的。一个表达式可以拥有N个子表达式以及条件，这个表达式还有一个属性and、or来决定所有子表达式/条件是全部成立还是只要有一个成立，这个表达式就成立。

当时做的时候真是各种绕，这种场景真的非常适合组合模式，能大大简化代码的实现难度，提高可读、可维护性

2020-03-04 12:40

Algo

给每个目录进行分片，当要增加目录/文件或删除目录/文件时，根据分片找到对应的部分，然后增加或删除，且更新该分片的文件个数。总数是根据各分片进行汇总的数字。。。。

2020-03-04 11:17

陆老师

可以加入fileSize，和fileCount字段，并用volatile修饰。文件的增删改操作，重新计算并赋值两个成员变量。其他线程读取到的数值也是最新的数值。

2020-03-04 10:58

，

课后题:

这种方案应该多用于在服务启动时,从数据库/配置文件取出数据,按照格式缓存起来,外部调用的时候直接从缓存中去取,可以添加变量size,维护在各个节点下面,在add/remove时同时更新缓存和数据库(ps:这种数据一般很少变化吧?)

2020-03-04 10:52

Jackey

目录结构中存储这两项数据是否可行？每次新增或删除文件时就更新父节点的数据。这样的话就需要在结构体中增加一个“父指针”。

2020-03-04 09:51

守拙

组合模式与其说是设计模式,不如说是数据结构与算法的抽象.

就像小野二郎只做寿司一样, 组合模式专注于树形结构中单一对象(叶子节点)与组合对象(树节点)的递归遍历.

2020-03-04 09:44