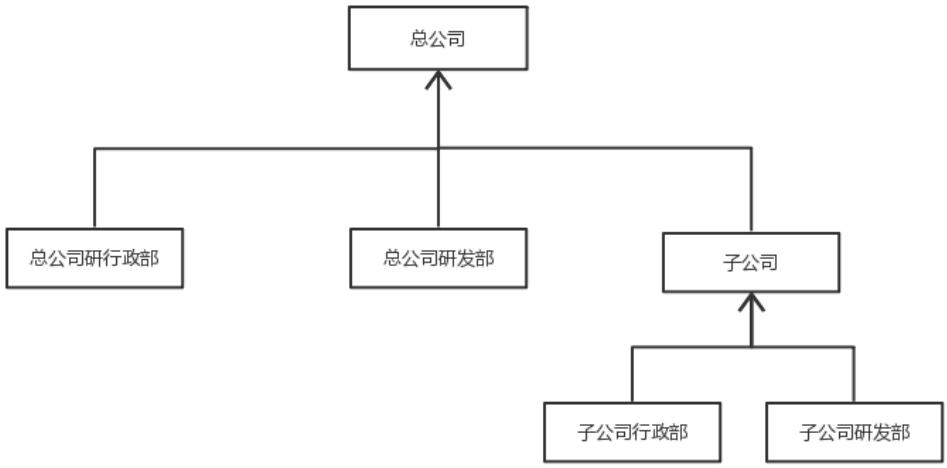
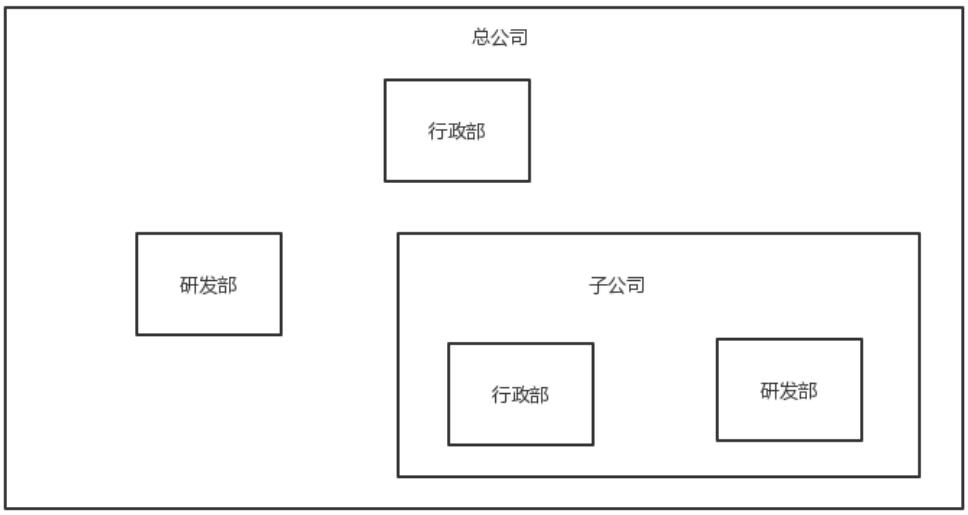
组合模式

1.组合模式介绍

组合模式也称为部分整体模式，结构型设计模式之一，组合模式比较简单，他将一组相似的对象看作一个对象处理，并根据一个书状结构来组合对象，然后提供一个统一的方法去访问相应的对象，以此忽略掉对象与对象集合之间的差别。生活中比较典型的例子就是组织结构的树状图，如图：

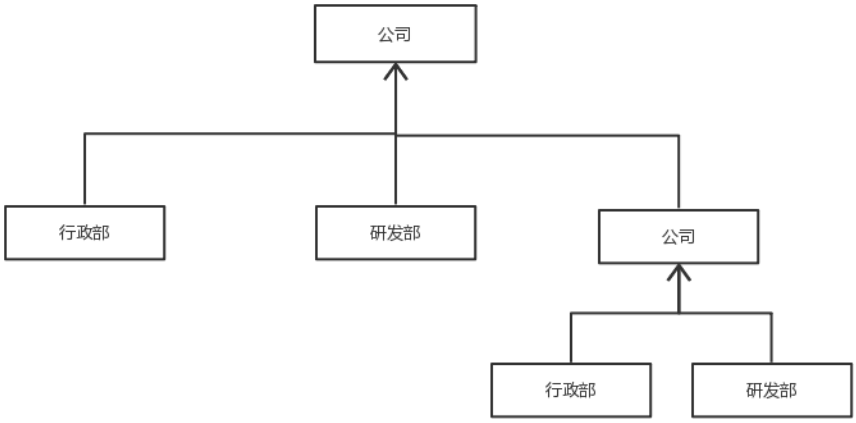


上面是一个公司的组织结构树状图，其中总公司下有行政部和研发部，但是从总公司的角度来看子公司就是一个独立的个体，与总公司所属的行政和研发部平级，如果我们以一个嵌套盒子的形式来展示将会更加直观，比如我们将上述的树形结构转为一个嵌套盒子，总公司则是最外层的盒子，里面包含3个小盒子，分别表示总公司的行政部与研发部，如图：



在这么一个结构中大家可以看到虽然总公司和子公司其本质不一样，但是他在我们的组织结构中是一样的，我们可以把他们看作一个抽象的公司，在组合模式中我们将这样的一个拥有分支的节点称之为枝干构件，位于树状结构顶部的

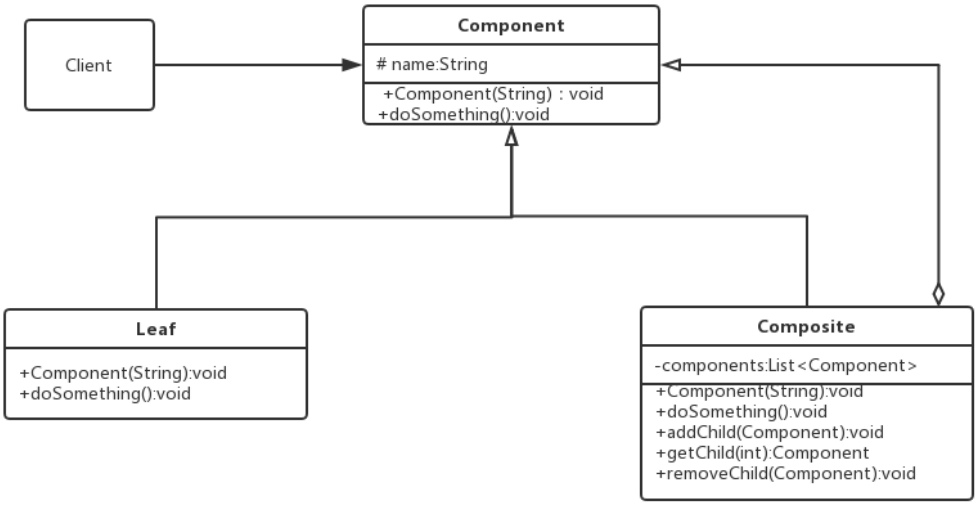
枝干比较特殊，我们称之为枝干构件，因为其为整个树状图的始端，同样对于像行政部和研发部这样没有分支的结构，我们称为叶子构件，这样的一个结构就是组合模式的雏形，如图：



2.组合模式的定义

将对象组合成树形结构以表示“部分-整体”的层次机构，使得用户对单个对象和组合对象的使用具有一致性。

3.组合模式的UML类图

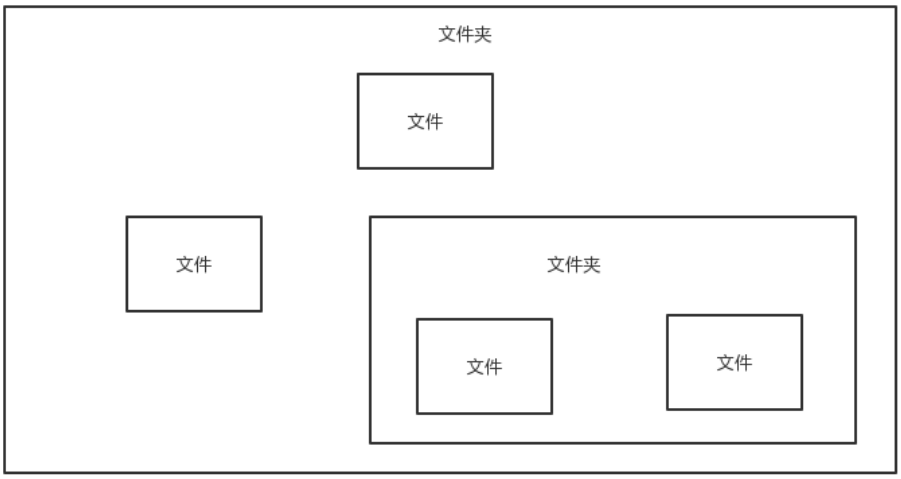


角色介绍：

- Component:抽象根节点，为组合中的对象声明接口。在适当的情况下，实现所有类共有接口的缺省行为。声明一个接口用于访问和管理Component的节点。可在递归结构中定义一个接口，用于访问一个父节点，并在合适的情况下实现它。
- Composite：定义有子节点的那些枝干节点的行为，存储子节点，在Component接口中实现与子节点有关的操作。
- Leaf:在组合中表示叶子节点对象，叶子结点没有子节点，在组合中定义节点对象的行为。
- Client:通过Component接口操纵组合节点的对象。

4.组合模式的简单实现

对于程序员来说，介绍一些组织结构想必一定不会很陌生，但是即便如此，我们也能在其他地方看到组合模式的影子，如果你使用过电脑，知道什么是操作系统，那么对文件和文件夹这两个概念一定不会陌生，什么是文件呢？文件就是可以被具体程序执行的对象，那么什么是文件夹呢？文件夹就是可以存放文件和文件夹的对象 ，如图：



如此一看，操作系统的文件系统其实就是一种典型的组合模式例子，在这里我们就以此为例，看看一个简单的文件系统是如何构成的，首先声明一个Dir抽象类表示文件和文件夹。

表示文件和文件夹的抽象类

```
//表示文件和文件夹的抽象类
public abstract class Dir {
        /*
         * 声明一个List成员变量存储文件夹下的所有元素
         */
        protected List<Dir> dirs = new ArrayList<Dir>();
        private String name;// 当前文件或文件夹名

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

        /*
         * 添加一个文件或文件夹
         */
        public abstract void addDir(Dir dir);

        /*
         * 移除一个文件或文件夹
         */
        public abstract void rmDir(Dir dir);

        /*
         * 清空文件夹下所有元素
         */
        public abstract void clear();

        /*
         * 输出文件夹下所有元素
```

```
         */          public abstract void print();

        /*
         * 获取文件夹下所有的文件或子文件夹
         */
        public abstract List<Dir> getFiles();

        /*
         * 文件或文件夹名
         */
        public String getName() {
                return name;
        }
}
```

在该抽象类中我们定义了相关的抽象方法，大家可以看到这里用到的就是所谓的透明的组合模式，这里要实现的功能很简单，只是简单的打印一下目录结构，因此，声明一个成员变量来存储当签文件或文件夹的名称并提供对象的getter方法。接下来就是具体的文件夹类，该类中我们实现具体的文件夹方法逻辑。

```
//表示文件夹的类
public class Folder extends Dir {

        public Folder(String name) {
                super(name);
        }

        @Override
        public void addDir(Dir dir) {
                dirs.add(dir);
        }

        @Override
        public void rmDir(Dir dir) {
                dirs.remove(dir);
        }

        @Override
        public void clear() {
                dirs.clear();
        }

        @Override
        public void print() {
                System.out.print(getName() + "(");
                Iterator<Dir> iter = dirs.iterator();
                while (iter.hasNext()) {
                        Dir dir = iter.next();
                        dir.print();
                        if (iter.hasNext()) {
                                System.out.print(",");
                        }
                }
                System.out.print(")");
        }
```

```
        @Override
    public List<Dir> getFiles() {
            return dirs;
    }
}
```

像addDir、rmDir在这里的方法中实现逻辑比较简单，这里主要就是print方法用来输出文件夹的目录结构，该方法逻辑也比较简单，首先输出自己的名字，也就是当前文件夹名，然后迭代遍历子元素，调用子元素的print方法输出其目录结构，如果遇到子元素还是个文件夹，那么递归遍历直至所有的输出元素均为文件为止。接下来看看文件夹的实现。

```
//表示文件的类
public class File extends Dir {

    public File(String name) {
            super(name);
    }

    @Override
    public void addDir(Dir dir) {
            throw new UnsupportedOperationException("文件对象不支持该操作");
    }

    @Override
    public void rmDir(Dir dir) {
            throw new UnsupportedOperationException("文件对象不支持该操作");
    }

    @Override
    public void clear() {
            throw new UnsupportedOperationException("文件对象不支持该操作");
    }

    @Override
    public void print() {
            System.out.print(getName());
    }

    @Override
    public List<Dir> getFiles() {
            throw new UnsupportedOperationException("文件对象不支持该操作");
    }

}
```

文件类相对于文件夹类来说既不支持添加也不支持删除，因为文件不能作为文件夹来使用，他本是文件系统中的最小分割单位，因此，这里我们虽然实现了Dir中的一些添加、删除等操作方法，但是，其实现逻辑只是抛出一个不支持的异常，因为就文件本身来说就不支持对应的操作。最后，在一个客户类中构建目录结构并输出。

```
package compositepattern;
```

```
//客户类
public class Client {
        public static void main(String[] args) {
        //构造一个目录对象表示C盘根目录
                Dir diskC=new Folder("C");

                //C盘根目录下有一个文件ImbaMallLog.txt
                diskC.addDir(new File("ImbaMallLog.txt"));

                //C盘根目录下还有3个子目录Windows、PerfLogs、Program File
                Dir dirWin=new Folder("Windows");

                //Windows目录下有文件explorer.exe
                dirWin.addDir(new File("explorer.exe"));
                diskC.addDir(dirWin);

                //PerfLogs目录
                Dir dirPer=new Folder("PerfLogs");

                //PerfLogs目录下有文件null.txt
                dirPer.addDir(new File("null.ext"));
                diskC.addDir(dirPer);

                //Program File目录
                Dir dirPro=new Folder("Program File");

                //Program File目录下有文件ftp.txt
                 dirPro.addDir(new File("ftp.txt"));
                 diskC.addDir(dirPro);

                 //打印出文件结构
                 diskC.print();
        }
}
```

输出：

```
C(ImbaMallLog.txt,Windows(explorer.exe),PerfLogs(null.ext),Program File(ftp.txt))
```

这里我们以括号作为一个文件夹的内容范围，如上输出所示，C盘文件夹下有3个子文件夹Window、PerfLogs和Program File以及一个文件ImbaMallLog.txt，而且3个文件夹中还给自包含有子文件，一个典型的树状嵌套结构，这就是组合模式。

5.总结

组合模式与我们前面讲的解释器模式有一定的类同，两者在迭代对象时都涉及递归的调用，但是组合模式所提供的属性层次结构使得我们能够一视同仁地对待单个对象和对象集合，不过这是以牺牲单一原则换来的，而且组合模式是通过继承来实现的，这样的做法缺少弹性。所以，和其它许多设计模式一样，在使用设计模式之前一定要想清楚利弊关系，不要造成设计模式的滥用。

组合模式的优点：

- 组合模式可以清楚的定义分层次的复杂对象，表示对象的全部或部分层次，他让高层次模块忽了层次的差异，方便对整个层次结构进行控制。
- 高层模块可以一致地使用一个组合结构或其中单个对象，不必关心处理的是单个对象还是整个组合结构，简化了高层模块的代码。
- 在组合模式中增加新的枝干构件和叶子构件都很方便，无须对现有类库进行任何修改，符合“开闭原则”。
- 组合模式为树形结构的面向对象实现提供了一种灵活的解决方案，通过叶子对象和枝干对象的递归组合，可以形成复杂的树形结构，但对树形结构的控制却非常简单。

组合模式的缺点：

在新增构件时不好对枝干中的构建类型进行限制，不能依赖类型系统来施加这些约束，因为在大多数情况下，他们都来自于相同的抽象层，此时，必须运行时进行类型类型检查来实现，这个实现过程较为复杂。