第 10 章　组合模式

我们知道地球和一些其他行星围绕着太阳旋转，也知道在一个原子中，有许多电子围绕着原子核旋转。我曾经想象，我们的太阳系也许是一个更大世界里的一个原子，地球只是围绕着太阳原子的一个电子。而我身上的每个原子又是一个星系，原子核就是这个星系中的恒星，电子是围绕着恒星旋转的行星。一个电子中也许还包含了另一个宇宙，虽然这个宇宙还不能被显微镜看到，但我相信它的存在。

也许这个想法有些异想天开，但在程序设计中，也有一些和“事物是由相似的子事物构成”类似的思想。组合模式就是用小的子对象来构建更大的对象，而这些小的子对象本身也许是由更小的“孙对象”构成的。

10.1 回顾宏命令

我们在第9章命令模式中讲解过宏命令的结构和作用。宏命令对象包含了一组具体的子命令对象，不管是宏命令对象，还是子命令对象，都有一个execute 方法负责执行命令。现在回顾一下这段安装在万能遥控器上的宏命令代码：

```
var closeDoorCommand = {
    execute: function(){
          console.log( '关门' );
    }
};

var openPcCommand = {
    execute: function(){
        console.log( '开电脑' );
    }
};

var openQQCommand = {
    execute: function(){
        console.log( '登录QQ' );
    }
};

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

};

var macroCommand = MacroCommand();

macroCommand.add( closeDoorCommand );
macroCommand.add( openPcCommand );
macroCommand.add( openQQCommand );
```

```
macroCommand.execute();
```

通过观察这段代码，我们很容易发现，宏命令中包含了一组子命令，它们组成了一个树形结构，这里是一棵结构非常简单的树，如图10-1所示。

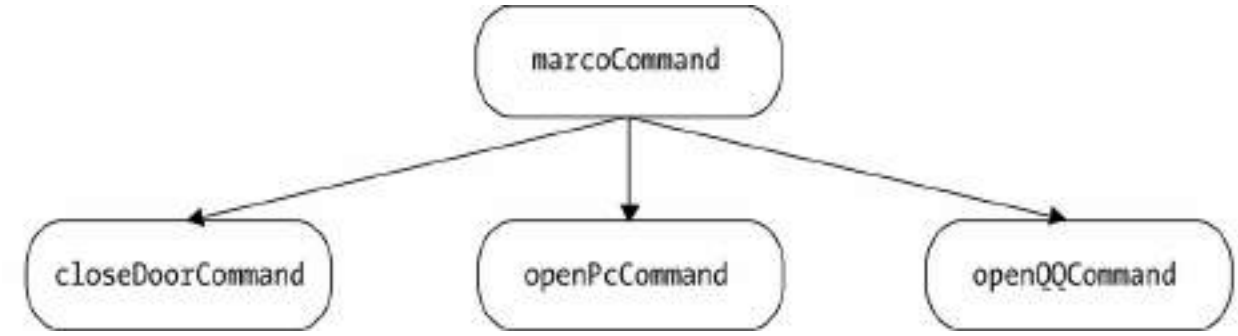

图　10-1

其中，marcoCommand 被称为组合对象，closeDoorCommand 、openPcCommand 、openQQCommand 都是叶对象。
在macroCommand 的execute 方法里，并不执行真正的操作，而是遍历它所包含的叶对象，把真正的execute 请求委托给这些叶对象。

macroCommand 表现得像一个命令，但它实际上只是一组真正命令的“代理”。并非真正的代理，虽然结构上相似，但macroCommand 只负责传递请求给叶对象，它的目的不在于控制对叶对象的访问。

10.2 组合模式的用途

组合模式将对象组合成树形结构，以表示“部分-整体”的层次结构。 除了用来表示树形结构之外，组合模式的另一个好处是通过对象的多态性表现，使得用户对单个对象和组合对象的使用具有一致性，下面分别说明。

- 表示树形结构。通过回顾上面的例子，我们很容易找到组合模式的一个优点：提供了一种遍历树形结构的方案，通过调用组合对象的 `execute` 方法，程序会递归调用组合对象下面的叶对象的`execute` 方法，所以我们的万能遥控器只需要一次操作，便能依次完成关门、打开电脑、登录QQ这几件事情。组合模式可以非常方便地描述对象部分-整体层次结构。

- 利用对象多态性统一对待组合对象和单个对象。利用对象的多态性表现，可以使客户端忽略组合对象和单个对象的不同。在组合模式中，客户将统一地使用组合结构中的所有对象，而不需要关心它究竟是组合对象还是单个对象。

这在实际开发中会给客户带来相当大的便利性，当我们往万能遥控器里面添加一个命令的时候，并不关心这个命令是宏命令还是普通子命令。这点对于我们不重要，我们只需要确定它是一个命令，并且这个命令拥有可执行的`execute` 方法，那么这个命令就可以被添加进万能遥控器。

当宏命令和普通子命令接收到执行`execute` 方法的请求时，宏命令和普通子命令都会做它们各自认为正确的事情。这些差异是隐藏在客户背后的，在客户看来，这种透明性可以让我们非常自由地扩展这个万能遥控器。

10.3　请求在树中传递的过程

在组合模式中，请求在树中传递的过程总是遵循一种逻辑。

以宏命令为例，请求从树最顶端的对象往下传递，如果当前处理请求的对象是叶对象（普通子命令），叶对象自身会对请求作出相应的处理；如果当前处理请求的对象是组合对象（宏命令），组合对象则会遍历它属下的子节点，将请求继续传递给这些子节点。

总而言之，如果子节点是叶对象，叶对象自身会处理这个请求，而如果子节点还是组合对象，请求会继续往下传递。叶对象下面不会再有其他子节点，一个叶对象就是树的这条枝叶的尽头，组合对象下面可能还会有子节点，如图10-2所示。

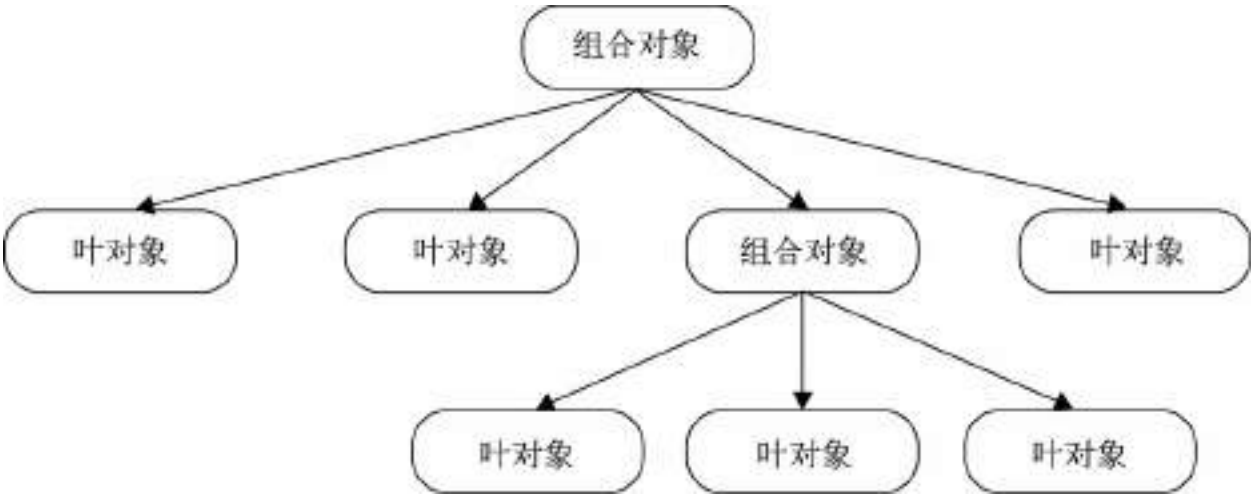

图　10-2

请求从上到下沿着树进行传递，直到树的尽头。作为客户，只需要关心树最顶层的组合对象，客户只需要请求这个组合对象，请求便会沿着树往下传递，依次到达所有的叶对象。

在刚刚的例子中，由于宏命令和子命令组成的树太过简单，我们还不能清楚地看到组合模式带来的好处，如果只是简单地遍历一组子节点，迭代器便能解决所有的问题。接下来我们将创造一个更强大的宏命令，这个宏命令中又包含了另外一些宏命令和普通子命令，看起来是一棵相对较复杂的树。

10.4　更强大的宏命令

目前的万能遥控器，包含了关门、开电脑、登录QQ这3个命令。现在我们需要一个“超级万能遥控器”，可以控制家里所有的电器，这个遥控器拥有以下功能：

- 打开空调
- 打开电视和音响
- 关门、开电脑、登录QQ

首先在节点中放置一个按钮button来表示这个超级万能遥控器，超级万能遥控器上安装了一个宏命令，当执行这个宏命令时，会依次遍历执行它所包含的子命令，代码如下：

```
<html>
    <body>
        <button id="button">按我</button>
    </body>

    <script>

    var MacroCommand = function(){
        return {
            commandsList: [],
            add: function( command ){
                this.commandsList.push( command );
            },
            execute: function(){
                for ( var i = 0, command; command = this.commandsList[ i
                    command.execute();
                }
            }
         }
    };

    var openAcCommand = {
        execute: function(){
            console.log( '打开空调' );
        }
    };
```

```
/**********家里的电视和音响是连接在一起的，所以可以用一个宏命令来组合打开电视和打开音

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

    var openSoundCommand = {
        execute: function(){
            console.log( '打开音响' );
        }
    };

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

/*********关门、打开电脑和打登录QQ的命令***************/

    var closeDoorCommand = {
        execute: function(){
              console.log( '关门' );
        }
    };

    var openPcCommand = {
        execute: function(){
            console.log( '开电脑' );
        }
    };

    var openQQCommand = {
        execute: function(){
            console.log( '登录QQ' );
        }
    };

    var macroCommand2 = MacroCommand();
    macroCommand2.add( closeDoorCommand );
    macroCommand2.add( openPcCommand );
    macroCommand2.add( openQQCommand );

/*********现在把所有的命令组合成一个“超级命令”**********/

    var macroCommand = MacroCommand();
    macroCommand.add( openAcCommand );
    macroCommand.add( macroCommand1 );
    macroCommand.add( macroCommand2 );
```

```
/*********最后给遥控器绑定“超级命令”**********/

    var setCommand = (function( command ){
        document.getElementById( 'button' ).onclick = function(){
            command.execute();
        }
    })( macroCommand );

    </script>
</html>
```

当按下遥控器的按钮时，所有命令都将被依次执行，执行结果如图10-3所示。

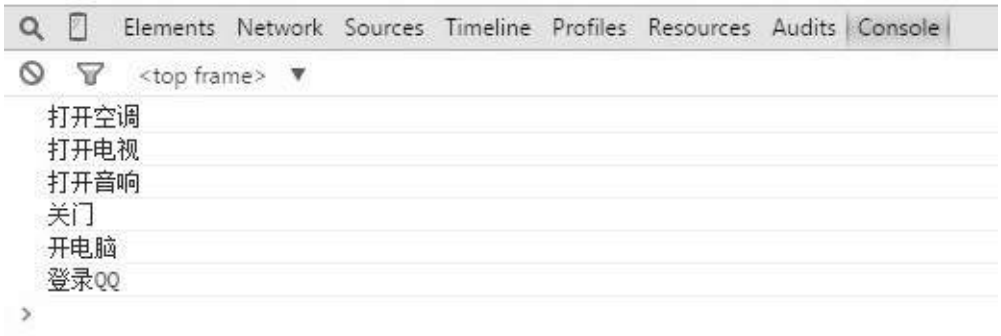

图　10-3

从这个例子中可以看到，基本对象可以被组合成更复杂的组合对象，组合对象又可以被组合，这样不断递归下去，这棵树的结构可以支持任意多的复杂度。在树最终被构造完成之后，让整颗树最终运转起来的步骤非常简单，只需要调用最上层对象的execute 方法。每当对最上层的对象进行一次请求时，实际上是在对整个树进行深度优先的搜索，而创建组合对象的程序员并不关心这些内在的细节，往这棵树里面添加一些新的节点对象是非常容易的事情。

10.5 抽象类在组合模式中的作用

前面说到，组合模式最大的优点在于可以一致地对待组合对象和基本对象。客户不需要知道当前处理的是宏命令还是普通命令，只要它是一个命令，并且有execute 方法，这个命令就可以被添加到树中。

这种透明性带来的便利，在静态类型语言中体现得尤为明显。比如在Java中，实现组合模式的关键是Composite 类和Leaf 类都必须继承自一个Compenent 抽象类。这个Compenent 抽象类既代表组合对象，又代表叶对象，它也能够保证组合对象和叶对象拥有同样名字的方法，从而可以对同一消息都做出反馈。组合对象和叶对象的具体类型被隐藏在Compenent 抽象类身后。

针对Compenent 抽象类来编写程序，客户操作的始终是Compenent 对象，而不用去区分到底是组合对象还是叶对象。所以我们往同一个对象里的add 方法里，既可以添加组合对象，也可以添加叶对象，代码如下：

```
// Java代码

public abstract class Component{
    // add方法，参数为Component类型
    public void add( Component child ){}
    // remove方法，参数为Component类型
    public void remove( Component child ){}
}

public class Composite extends Component{
    // add方法，参数为Component类型
    public void add( Component child ){}
    // remove方法，参数为Component类型
    public void remove( Component child ){}
}

public class Leaf extends Component{
    // add方法，参数为Component类型
    public void add( Component child ){
        throw new UnsupportedOperationException()    // 叶对象不能再添加子
    }
    // remove方法，参数为Component类型
    public void remove( Component child ){
    }
}
```

```
public class client(){

    public static void main( String args[] ){
        Component root = new Composite();

        Component c1 = new Composite();
        Component c2 = new Composite();

        Component leaf1 = new Leaf();
        Component leaf2 = new Leaf();

        root.add(c1);
        root.add(c2);

        c1.add(leaf1);
        c1.add(leaf2);

        root.remove();
    }
}
```

然而在JavaScript这种动态类型语言中，对象的多态性是与生俱来的，也没有编译器去检查变量的类型，所以我们通常不会去模拟一个“怪异”的抽象类，JavaScript中实现组合模式的难点在于要保证组合对象和叶对象对象拥有同样的方法，这通常需要用鸭子类型的思想对它们进行接口检查。

在JavaScript中实现组合模式，看起来缺乏一些严谨性，我们的代码算不上安全，但能更快速和自由地开发，这既是JavaScript的缺点，也是它的优点。

10.6 透明性带来的安全问题

组合模式的透明性使得发起请求的客户不用去顾忌树中组合对象和叶对象的区别，但它们在本质上有是区别的。

组合对象可以拥有子节点，叶对象下面就没有子节点，所以我们也许会发生一些误操作，比如试图往叶对象中添加子节点。解决方案通常是给叶对象也增加add 方法，并且在调用这个方法时，抛出一个异常来及时提醒客户，代码如下：

```
var MacroCommand = function(){
    return {
        commandsList: [],
        add: function( command ){
            this.commandsList.push( command );
        },
        execute: function(){
            for ( var i = 0, command; command = this.commandsList[ i++ ]
               command.execute();
            }
        }
    }
};

var openTvCommand = {
    execute: function(){
        console.log( '打开电视' );
    },
    add: function(){
        throw new Error( '叶对象不能添加子节点' );
    }
};

var macroCommand = MacroCommand();

macroCommand.add( openTvCommand );
openTvCommand.add( macroCommand )    // Uncaught Error: 叶对象不能添加子节点
```

10.7　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。文件夹里既可以包含文件，又可以包含其他文件夹，最终可能组合成一棵树，组合模式在文件夹的应用中有以下两层好处。

- 例如，我在同事的移动硬盘里找到了一些电子书，想把它们复制到F盘中的学习资料文件夹。在复制这些电子书的时候，我并不需要考虑这批文件的类型，不管它们是单独的电子书还是被放在了文件夹中。组合模式让Ctrl+V、Ctrl+C成为了一个统一的操作。

- 当我用杀毒软件扫描该文件夹时，往往不会关心里面有多少文件和子文件夹，组合模式使得我们只需要操作最外层的文件夹进行扫描。

现在我们来编写代码，首先分别定义好文件夹Folder 和文件File 这两个类。见如下代码:

```
/****************************** Folder ******************************/
var Folder = function( name ){
    this.name = name;
    this.files = [];
};

Folder.prototype.add = function( file ){
    this.files.push( file );
};

Folder.prototype.scan = function(){
    console.log( '开始扫描文件夹: ' + this.name );
    for ( var i = 0, file, files = this.files; file = files[ i++ ]; ){
        file.scan();
    }
};

/****************************** File ******************************/
var File = function( name ){
    this.name = name;
};

File.prototype.add = function(){
    throw new Error( '文件下面不能再添加文件' );
};
```

```
File.prototype.scan = function(){
    console.log( '开始扫描文件: ' + this.name );
};
```

接下来创建一些文件夹和文件对象，并且让它们组合成一棵树，这棵树就是我们F盘里的现有文件目录结构：

```
var folder = new Folder( '学习资料' );
var folder1 = new Folder( 'JavaScript' );
var folder2 = new Folder ( 'jQuery' );

var file1 = new File( 'JavaScript设计模式与开发实践' );
var file2 = new File( '精通jQuery' );
var file3 = new File( '重构与模式' )


folder1.add( file1 );
folder2.add( file2 );

folder.add( folder1 );
folder.add( folder2 );
folder.add( file3 );
```

现在的需求是把移动硬盘里的文件和文件夹都复制到这棵树中，假设我们已经得到了这些文件对象：

```
var folder3 = new Folder( 'Nodejs' );
var file4 = new File( '深入浅出Node.js' );
folder3.add( file4 );

var file5 = new File( 'JavaScript语言精髓与编程实践' );
```

接下来就是把这些文件都添加到原有的树中：

```
folder.add( folder3 );
folder.add( file5 );
```

通过这个例子，我们再次看到客户是如何同等对待组合对象和叶对象。在添加一批文件的操作过程中，客户不用分辨它们到底是文件还是文件夹。新增加的文件和文件夹能够很容易地添加到原来的树结构中，和树里已有的对象一起工作。

我们改变了树的结构，增加了新的数据，却不用修改任何一句原有的代码，这是符合开放-封闭原则的。

运用了组合模式之后，扫描整个文件夹的操作也是轻而易举的，我们只需要操作树的最顶端对象：

```
folder.scan();
```

执行结果如图10-4所示。

图　10-4

10.8　一些值得注意的地方

在使用组合模式的时候，还有以下几个值得我们注意的地方。

1. 组合模式不是父子关系

组合模式的树型结构容易让人误以为组合对象和叶对象是父子关系，这是不正确的。

组合模式是一种HAS-A（聚合）的关系，而不是IS-A。组合对象包含一组叶对象，但`Leaf` 并不是`Composite` 的子类。组合对象把请求委托给它所包含的所有叶对象，它们能够合作的关键是拥有相同的接口。

为了方便描述，本章有时候把上下级对象称为父子节点，但大家要知道，它们并非真正意义上的父子关系。

2. 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拥有相同的接口之外，还有一个必要条件，就是对一组叶对象的操作必须具有一致性。

比如公司要给全体员工发放元旦的过节费1000块，这个场景可以运用组合模式，但如果公司给今天过生日的员工发送一封生日祝福的邮件，组合模式在这里就没有用武之地了，除非先把今天过生日的员工挑选出来。只有用一致的方式对待列表中的每个叶对象的时候，才适合使用组合模式。

3. 双向映射关系

发放过节费的通知步骤是从公司到各个部门，再到各个小组，最后到每个员工的邮箱里。这本身是一个组合模式的好例子，但要考虑的一种情况是，也许某些员工属于多个组织架构。比如某位架构师既隶属于开发组，又隶属于架构组，对象之间的关系并不是严格意义上的层次结构，在这种情况下，是不适合使用组合模式的，该架构师很可能会收到两份过节费。

这种复合情况下我们必须给父节点和子节点建立双向映射关系，一个简单的方法是给小组和员工对象都增加集合来保存对方的引用。但是这种相互间的引用相当复杂，而且对象之间产生了过多的耦合性，修改或者删除一

个对象都变得困难，此时我们可以引入中介者模式来管理这些对象。

4. 用职责链模式提高组合模式性能

在组合模式中，如果树的结构比较复杂，节点数量很多，在遍历树的过程中，性能方面也许表现得不够理想。有时候我们确实可以借助一些技巧，在实际操作中避免遍历整棵树，有一种现成的方案是借助职责链模式。职责链模式一般需要我们手动去设置链条，但在组合模式中，父对象和子对象之间实际上形成了天然的职责链。让请求顺着链条从父对象往子对象传递，或者是反过来从子对象往父对象传递，直到遇到可以处理该请求的对象为止，这也是职责链模式的经典运用场景之一。

10.9 引用父对象

在11.7节提到的例子中，组合对象保存了它下面的子节点的引用，这是组合模式的特点，此时树结构是从上至下的。但有时候我们需要在子节点上保持对父节点的引用，比如在组合模式中使用职责链时，有可能需要让请求从子节点往父节点上冒泡传递。还有当我们删除某个文件的时候，实际上是从这个文件所在的上层文件夹中删除该文件的。

现在来改写扫描文件夹的代码，使得在扫描整个文件夹之前，我们可以先移除某一个具体的文件。

首先改写Folder 类和File 类，在这两个类的构造函数中，增加this.parent 属性，并且在调用add 方法的时候，正确设置文件或者文件夹的父节点：

```
var Folder = function( name ){
    this.name = name;
    this.parent = null;     // 增加this.parent属性
    this.files = [];
};

Folder.prototype.add = function( file ){
    file.parent = this;     // 设置父对象
    this.files.push( file );
};

Folder.prototype.scan = function(){
    console.log( '开始扫描文件夹: ' + this.name );
    for ( var i = 0, file, files = this.files; file = files[ i++ ]; ){
        file.scan();
    }
};
```

接下来增加Folder.prototype.remove 方法，表示移除该文件夹：

```
Folder.prototype.remove = function(){
    if ( !this.parent ){     // 根节点或者树外的游离节点
        return;
```

```
    }
    for ( var files = this.parent.files, l = files.length - 1; l >=0; l-
        var file = files[ l ];
        if ( file === this ){
            files.splice( l, 1 );
        }
    }
};
```

在File.prototype.remove方法里，首先会判断this.parent，如果this.parent为null，那么这个文件夹要么是树的根节点，要么是还没有添加到树的游离节点，这时候没有节点需要从树中移除，我们暂且让remove方法直接return，表示不做任何操作。

如果this.parent不为null，则说明该文件夹有父节点存在，此时遍历父节点中保存的子节点列表，删除想要删除的子节点。

File类的实现基本一致：

```
var File = function( name ){
    this.name = name;
    this.parent = null;
};

File.prototype.add = function(){
    throw new Error( '不能添加在文件下面' );
};

File.prototype.scan = function(){
    console.log( '开始扫描文件: ' + this.name );
};

File.prototype.remove = function(){
    if ( !this.parent ){    // 根节点或者树外的游离节点
        return;
    }
    for ( var files = this.parent.files, l = files.length - 1; l >=0; l-
        var file = files[ l ];
        if ( file === this ){
            files.splice( l, 1 );
        }
    }
```

```
};
```

下面测试一下我们的移除文件功能：

```
var folder = new Folder( '学习资料' );
var folder1 = new Folder( 'JavaScript' );
var file1 = new Folder ( '深入浅出Node.js' );

folder1.add( new File( 'JavaScript设计模式与开发实践' ) );
folder.add( folder1 );
folder.add( file1 );

folder1.remove();    // 移除文件夹
folder.scan();
```

执行结果如图10-5所示。

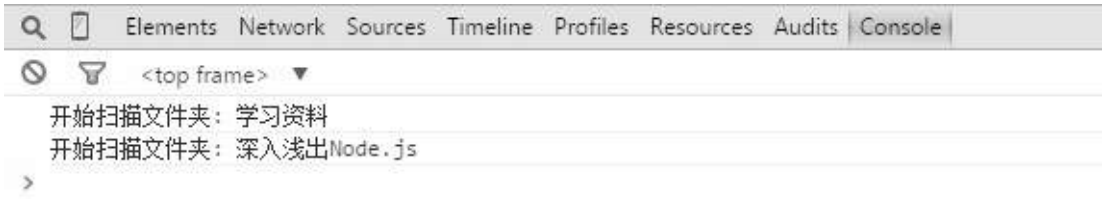

图　10-5

10.10　何时使用组合模式

组合模式如果运用得当，可以大大简化客户的代码。一般来说，组合模式适用于以下这两种情况。

- 表示对象的部分-整体层次结构。组合模式可以方便地构造一棵树来表示对象的部分-整体结构。特别是我们在开发期间不确定这棵树到底存在多少层次的时候。在树的构造最终完成之后，只需要通过请求树的最顶层对象，便能对整棵树做统一的操作。在组合模式中增加和删除树的节点非常方便，并且符合开放-封闭原则。

- 客户希望统一对待树中的所有对象。组合模式使客户可以忽略组合对象和叶对象的区别，客户在面对这棵树的时候，不用关心当前正在处理的对象是组合对象还是叶对象，也就不用写一堆`if`、`else`语句来分别处理它们。组合对象和叶对象会各自做自己正确的事情，这是组合模式最重要的能力。