第 12 章

职责链（Chain of Responsibility）模式

面向对象的开发者往往力求对象之间保持松散耦合，确保对象各自的责任具体并能最小化。这样的设计可以使得系统更加容易修改，同时降低产生缺陷的风险。从某种程度上讲，Java 语言有利于做出解耦的设计。客户端通常只能访问对象可见的接口，而不了解其实现细节。同时，客户端只需知道哪个对象具有它所需要调用的方法即可。当我们将若干对象按照某种层次结构进行组织时，客户端可能事先并不了解应该使用哪个类。这些对象要么执行该方法，要么将请求传递给下一个对象。

职责链模式的目的是通过给予多个对象处理请求的机会，以解除请求的发送者与接收者之间的耦合。

现实中的职责链模式

当一个人负责某项任务时，可以选择自己做或是让别人做，这就是现实中的职责链模式。以 Oozinoz 公司的情况为例，工程师会负责维护制造焰火的机器。

正如第 5 章中所介绍的那样，Oozinoz 公司把机器、生产线、车间、工厂模型看做“机器组件”。这种方法可以简化并递归实现某些操作，比如关闭某车间中所有的机器，或是简化工厂中工程师职责的模型。在 Oozinoz 公司，通常是特定的工程师负责某些特定的机器组件，尽管这种职责分配可能会分为不同的层次。

举例来说，一个复杂的机器，比如一个星形的冲床，可能由一个具体的工程师直接负责。然而，一个简单的机器可能没有直接负责的工程师，此时，就会由负责机器所在的生产线或车间的工程师来管理。

当客户对象查询负责机器的工程师时，希望能避免查询多个对象。因此，可以使用职责链模式，为每个机器组件分配责任人对象。图 12.1 说明了这种设计思路。

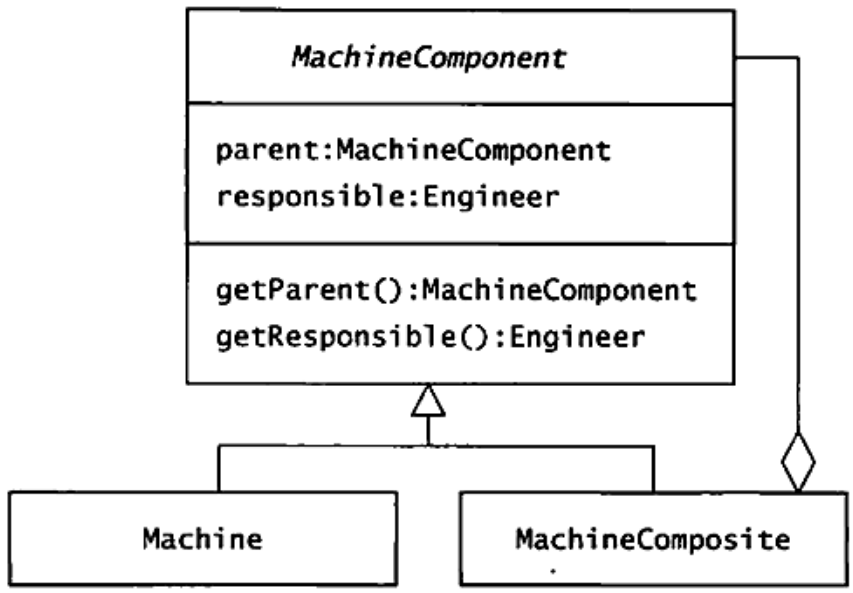

图 12.1　每个 Machine 或 MachineComposite 对象都有父对象（parent）和责任人对象（responsible），它们均继承自 MachineComponent 类

图 12.1 的设计思路允许每个机器组件跟踪负责它的工程师，但这并非强制性的要求。如果某台机器没有分配直接负责的工程师，可以把查询负责人的请求发送给父节点。一般而言，机器的父节点是生产线，生产线的父节点是车间，车间的父节点是工厂。在 Oozinoz 公司中，这个职责链的某处总有一位负责人。

这种设计思路的优点是机器组件的客户（代码）无须了解负责的工程师是如何分配的，客户（代码）可以查询任何机器组件的负责人。机器组件把客户（代码）与职责的分配标准相互独立。当然，在另一方面，这种设计思路也存在不合理的地方。

挑战 12.1

指出图 12.1 所示的设计思路中的两个缺陷。

答案参见第 323 页

职责链模式可以帮助我们简化客户端代码，尤其当客户端代码不清楚对象组中哪个对象负责处理查询请求时。如果事先没有建立职责链模式，也许要借助其他方法来简化之前复杂的设计。

重构为职责链模式

如果发现客户代码在发出调用请求前对调用做了判断，就应该通过代码重构来改善代码设计。若要应用职责链模式，必须事先明确这一组类对象是否支持该模式。例如，Oozinoz 中的机器组件有时提供对负责人的引用。把期望的操作添加到这组类对象的每个类中，并用链策略（Chaining Strategy）来实现该操作，以满足这一请求。

考虑 Oozinoz 代码库中对工具和工具车的建模。工具不属于 `MachineComponent` 类层次，但是在某些地方与机器类似。特别的，工具始终分配给工具车，并且工具车会包含一位负责的工程师。假设某可视化程序可以显示特定车间的所有工具和机器，并且提供弹出式信息以显示特定项的责任人。图 12.2 显示了在查询指定设备的责任人时所涉及的类。

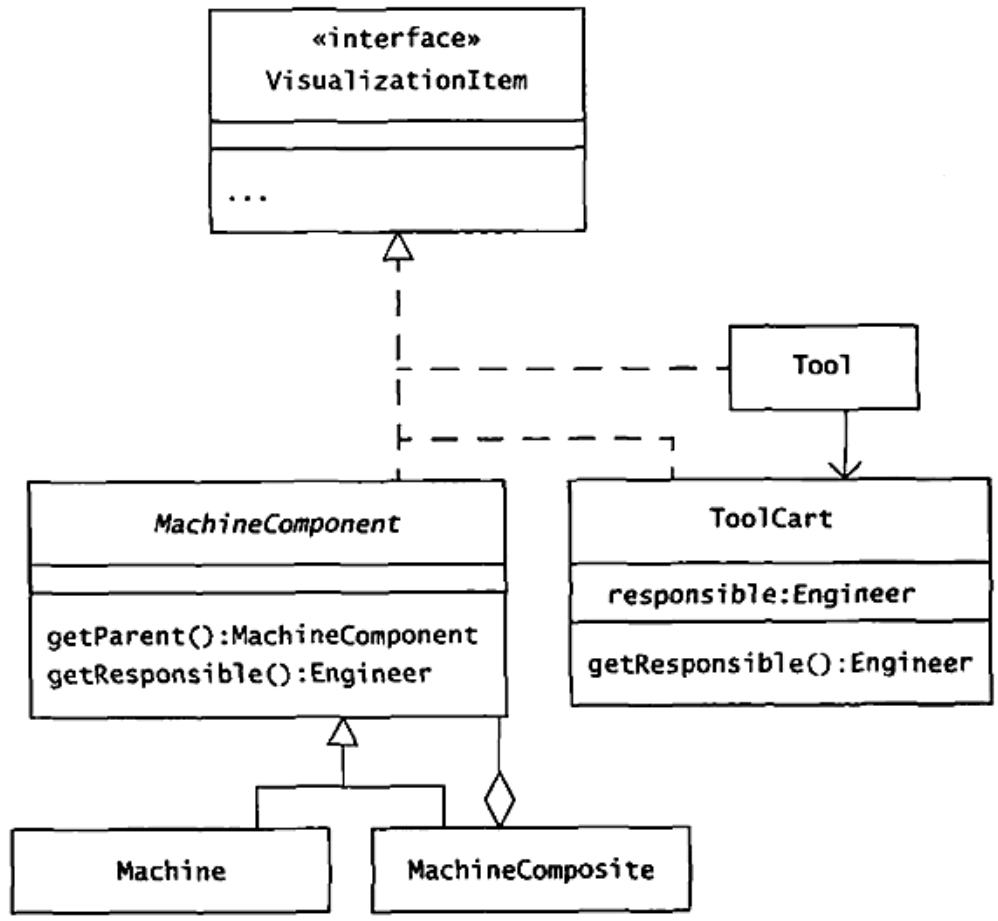

图 12.2　一个模拟环境下包含的设备项，包括各种机器、机器组合、工具以及工具车

VisualizationItem 接口具体指定了类需要的部分行为，以便于实现可视化，同时没有定义 getResponsible()方法。事实上，并不是 VisualizationItem 的所有项都知道具体的负责人。当可视化程序需要决定哪个工程师对特定设备负责时，取决于所选的那个设备项。机器、机器组和工具车都有 getResponsible()方法，而工具没有。为确定工具的责任工程师，程序代码必须查看这个工具属于哪个工具车，然后查看谁负责这个工具车。为确定指定设备项的负责人，应用程序的菜单代码使用了一些 if 语句和类型判断。这些特性表明重构可能有助于改善代码，具体代码如下：

```
package com.oozinoz.machine;

public class AmbitiousMenu {
    public Engineer getResponsible(VisualizationItem item) {
        if (item instanceof Tool) {
            Tool t = (Tool) item;
            return t.getToolCart().getResponsible();
        }
        if (item instanceof ToolCart) {
            ToolCart tc = (ToolCart) item;
            return tc.getResponsible();
        }
        if (item instanceof MachineComponent) {
            MachineComponent c = (MachineComponent) item;
            if (c.getResponsible() != null)
                return c.getResponsible();
            if (c.getParent() != null)
                return c.getParent().getResponsible();
        }
        return null;
    }
}
```

职责链模式的意图在于减轻调用者的压力，使它们无须了解哪个对象可以处理调用请求。在本例中，菜单是个调用者，它要求找到对应的负责人。根据当前设计，调用者必须了解哪个设备项具有 getReponsible()方法。借助于职责链模式为所有模拟项提供责任人，我们可以借此来简化代码。这样可以将了解哪些对象知道其负责人的相关职责从菜单代码中转移到模拟项。

挑战 12.2

重新绘制类图 12.2，将 getReponsible()方法移入 VisualizationItem 接口，并将该方法加入到 Tool 类。

答案参见第 324 页

现在，菜单代码变得更加简单，可以直接通过调用 VisualizationItem 对象的方法来查找负责的工程师，具体如下：

```
package com.oozinoz.machine;

public class AmbitiousMenu2 {
    public Engineer getResponsible(VisualizationItem item) {
        return item.getResponsible();
    }
}
```

每个设备项的 getResponsible()方法也会变得更加容易实现。

挑战 12.3

为下面的类分别写出对应的 getResponsible()方法。

A. MachineComponent

B. Tool

C. ToolCart

答案参见第 325 页

固定职责链

在为 MachineComponent 类编写 getResponsible()方法时，必须考虑到它的父对象可能为空。一种解决方法是，让每个 MachineComponent 对象都有一个非空父对象。这样做可以让

我们的对象模型更加紧凑。为实现这一目的，可以为 MachineComponent 类的构造函数添加一个参数来提供父对象（如果提供的父对象为空，甚至可以抛出一个异常，以便知道异常被捕获的位置）。此外，还要考虑到位于根部的对象——该对象没有父对象。一种合理的做法是创建一个 MachineRoot 类，并让该类继承 MachineComposite 类（而不是 MachineComponent）。为保证每个 MachineComponent 对象都对应一名负责人，可以这样做：

- MachineRoot 类的构造函数需要一个 Engineer 对象。
- MachineComponent 类的构造函数需要一个类型为 MachineComponent 的父对象。
- 只有 MachineRoot 使用 null 作为其父对象的值。

挑战 12.4

请写出图 12.3 中各个类的构造函数，从而保证每个 MachineComponent 对象都对应一名负责的工程师。

答案参见第 325 页

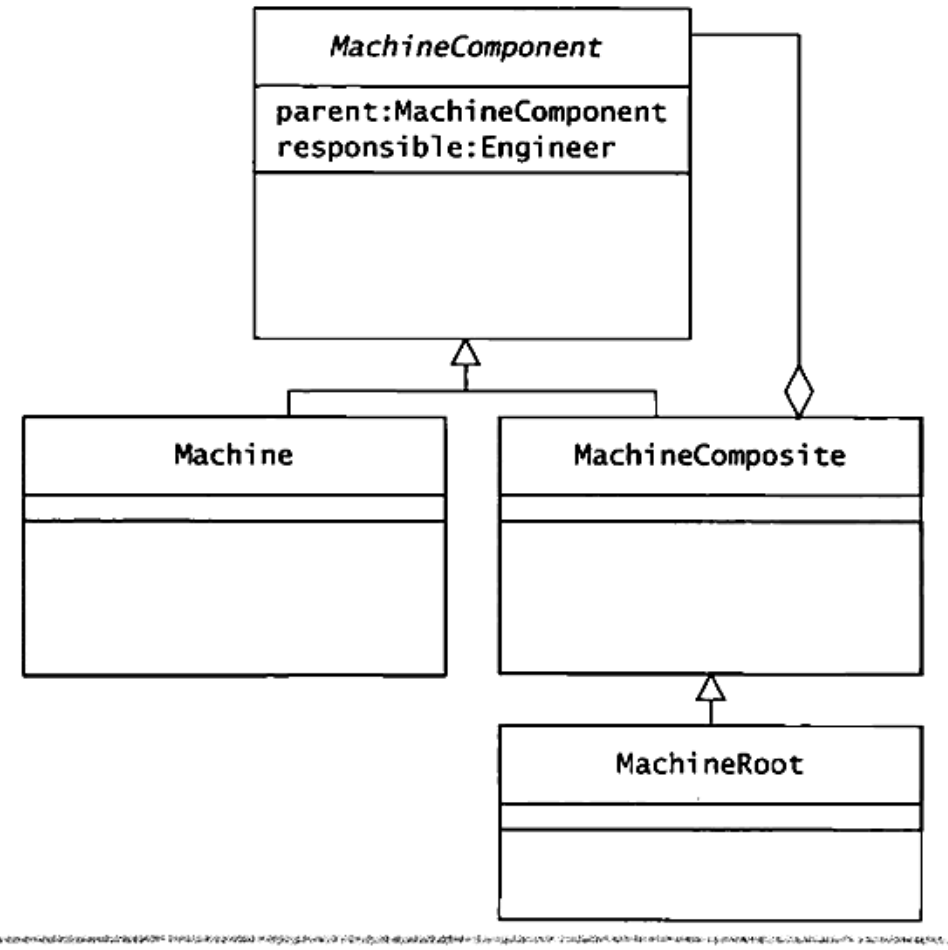

图 12.3　构造函数如何保证每个 MachineComponent 对象都对应一名负责的工程师

通过固定职责链，我们的对象模型变得更加健壮，代码也更为简洁。现在可以实现 MachineComponent 类的 getResponsible()方法了。

```
public Engineer getResponsible() {
    if (responsible != null)
        return responsible;
    return parent.getResponsible();
}
```

没有组合结构的职责链模式

职责链模式需要一种策略来确定处理请求的对象的查询顺序。采用哪种查询策略取决于建模领域的背景知识。倘若对象模型存在某些类型的组合，如 Oozinoz 公司的机器类层次结构，这一情形就极为常见。不过，职责链模式也可以用于不带组合结构的对象模型。

挑战 12.5

举例说明职责链模式可以存在于哪些没有组合结构的对象链模型中。

答案参见第 326 页

小结

在运用职责链模式时，客户端不必事先知道对象集合中哪个对象可提供自己需要的服务。当客户端发出调用请求后，该请求会沿着职责链转发请求，直到找到提供该服务的对象为止。这就可以降低客户端与提供服务的对象之间的耦合度。

如果某个对象链能够应用一系列不同的策略来解决某个问题，如解析用户的输入，这时也可以应用职责链模式。该模式更常见于组合结构，它具有一个包容的层次结构，为对象链提供了一种自然的查询顺序。简化对象链和客户端的代码是职责链模式的一个主要优点。