第29章

访问者（Visitor）模式

要对已存在的类层次进行扩展，通常的做法是为需要的行为增加方法。然而，有时需要增加的行为与现有对象模型并不一致，又或者无法修改现有代码。在这种情况下，不更改类的层次结构，就无法扩展该层次结构的行为。倘若类层次结构的开发者运用了访问者模式，就可以支持其他开发人员扩展该类层次结构的行为。

和解释器模式一样，访问者模式通常是基于合成模式的。你可能需要复习一下合成模式，因为我们会在整章中使用这一模式。

访问者模式的意图是在不改变类层次结构的前提下，对该层次结构进行扩展。

访问者模式机制

在开发类层次结构时，访问者模式的运用为代码的扩展开启了一扇大门，即使后来的开发人员无权访问源代码，仍然可以对类的层次结构进行各种扩展。访问者模式的机制如下：

- 为类层次结构中的部分或全部类增加 `accept()`方法。该方法的每个实现都需要接收一个参数，参数类型为自定义的接口。
- 创建定义了一组操作的接口，这些操作通常都会公用一个名字，如 `visit`，每个操作所需的参数类型并不相同。只要类层次结构中的类需要扩展，就可以为其声明这样的操作。

图 29.1 展示了这样一个类图，它修改了 MachineComponent 的类层次，使其支持访问者模式。

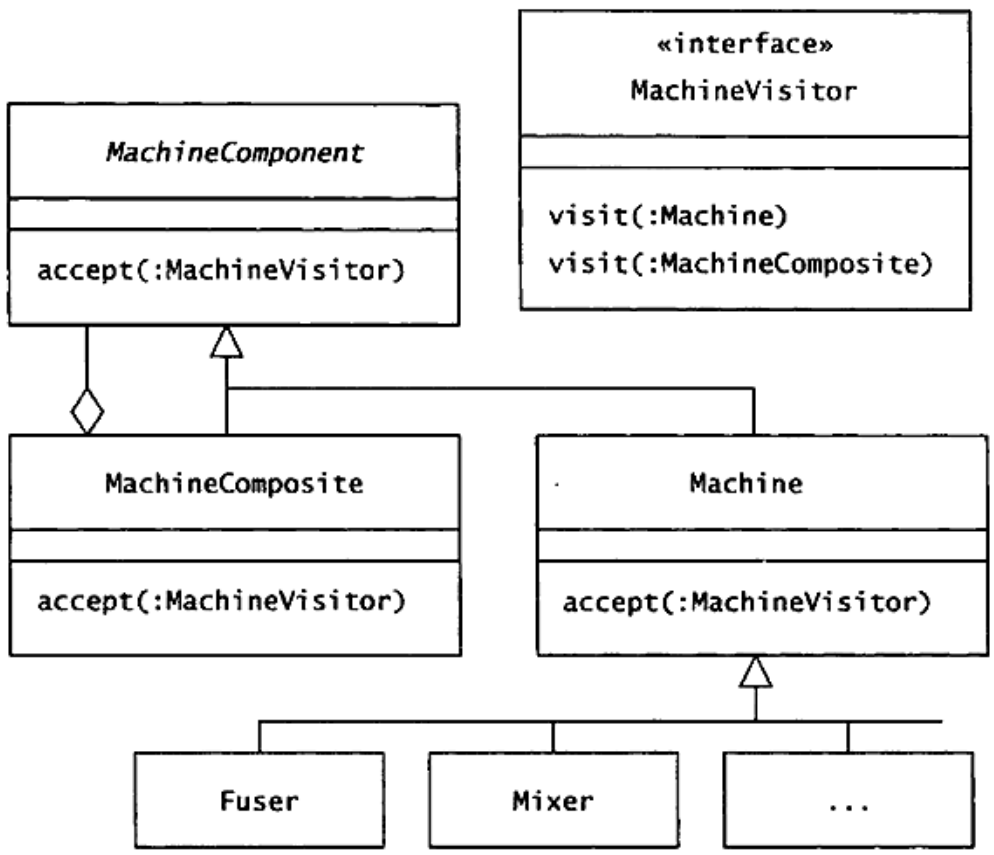

图 29.1　为了让 MachineComponent 层次结构支持访问者模式，在该图中增加了 accept()方法和 MachineVisitor 接口

图 29.1 并未解释访问者模式是如何工作的，在本章的下一小节会详细讲解。该图简单地展示了使用访问者模式的基本原理。

请注意，在 MachineComponent 的类图中，并非所有的类都实现了 accept()方法。访问者模式并不要求层次结构中的每个类都实现自己的 accept()方法。我们将会看到，实现 accept()方法的每个类，都会作为 visit()方法的参数被传递给 visitor 接口。

MachineComponent 类的 accept()方法是一个抽象方法，每个子类对该方法的实现几乎完全相同：

```
public void accept(MachineVisitor v) {
    v.visit(this);
}
```

你可能会想，既然该方法在 Machine 和 MachineComposite 类中完全相同，就可以将其提

取到抽象的 MachineComponent 类中。然而，编译器却会认为这两份“完全相同的代码”其实并不相同。

> **挑战 29.1**
>
> Java 编译器会认为 Machine 和 MachineComposite 类的 accept()方法有何不同？（不要试图忽略这一点，这是理解访问者模式的关键。）
>
> 答案参见第 363 页

MachineVisitor 接口要求实现者定义访问 machines 和 machine composites 的方法：

```
package com.oozinoz.machine;
public interface MachineVisitor {
    void visit(Machine m);
    void visit(MachineComposite mc);
}
```

MachineComponent 类的 accept()方法与 MachineVisitor 接口的结合，使得开发者能够为层次结构提供新的操作。

常规的访问者模式

假设你就职于 Oozinoz 公司在爱尔兰都柏林新成立的工厂。这里的开发人员已经为新工厂的机器组合建立了对象模型，并且可以通过 OozinozFactory 类的静态方法 dublin()访问该模型。为了能正常显示该合成模型，开发人员创建了一个 MachineTreeModel 类，它可以为 JTree 对象提供需要的信息。（MachineTreeModel 类的代码在 com.oozinoz.dublin 包中。）

为了显示工厂的机器，需要从工厂组合中创建一个 MachineTreeModel 类的实例，并将其包装为 Swing 组件：

```
package app.visitor;
import javax.swing.JScrollPane;
import javax.swing.JTree;
import com.oozinoz.machine.OozinozFactory;
```

```
import com.oozinoz.ui.SwingFacade;

public class ShowMachineTreeModel {
    public ShowMachineTreeModel() {
        MachineTreeModel model = new MachineTreeModel(
            OozinozFactory.dublin());
        JTree tree = new JTree(model);
        tree.setFont(SwingFacade.getStandardFont());
        SwingFacade.launch(
            new JScrollPane(tree),
            " A New Oozinoz Factory");
    }

    public static void main(String[] args) {
        new ShowMachineTreeModel();
    }
}
```

运行这一程序，显示结果如图 29.2 所示。

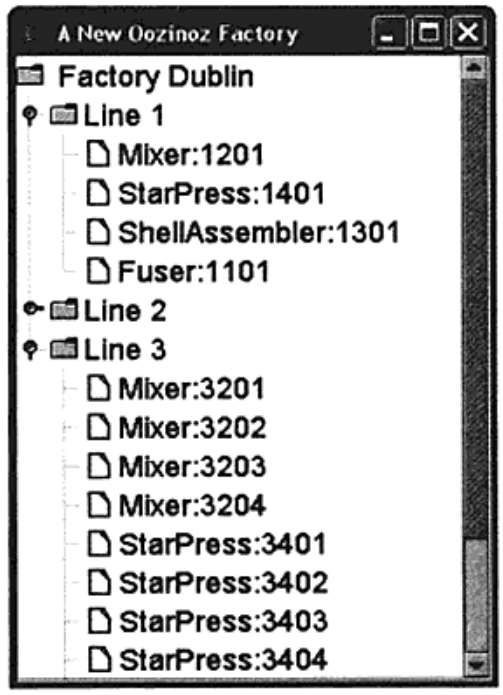

图 29.2　该 GUI 程序展示了在都柏林新工厂中的机器组合

机器组合有许多有用的行为，假设需要在工厂模型中找到一台指定的机器，在不改变 `MachineComponent` 层次结构的前提下增加这项功能，就可以创建一个如图 29.3 所示的 `FindVisitor` 类。

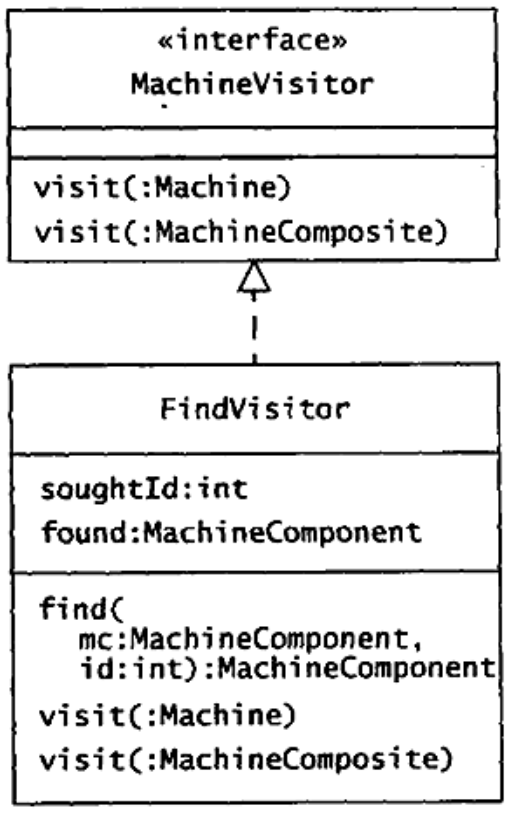

图 29.3 FindVisitor 类向 MachineComponent 层次结构添加了 find()方法

visit()方法不返回任何对象，因此 FindVisitor 类在 found 实例变量中记录搜索状态。

```
package app.visitor;

import com.oozinoz.machine.*;
import java.util.*;

public class FindVisitor implements MachineVisitor {
    private intsoughtId;
    private MachineComponent found;

    public MachineComponent find(
            MachineComponent mc, int id) {
        found = null;
        soughtId = id;
        mc.accept(this);
        return found;
    }

    public void visit(Machine m) {
        if (found == null &&m.getId() == soughtId)
```

```
            found = m;
        }

        public void visit(MachineComposite mc) {
            if (found == null &&mc.getId() == soughtId) {
                found = mc;
                return;
            }
            Iterator iter = mc.getComponents().iterator();
            while (found == null &&iter.hasNext())
                ((MachineComponent) iter.next()).accept(this);
        }
    }
```

visit()方法会检查 found 变量，在找到目标组件后，树的遍历就会结束。

挑战 29.2

写一个程序用来查找并打印出 OozinozFactory.dublin()方法所返回的 MachineComponent 实例中的 StarPress:3404 对象。

答案参见第 363 页

find()方法并不关心接收到的 MachineComponent 对象究竟是 Machine 类型，还是 MachineComposite 类型。该方法只是简单地调用 accept()，而 accept()方法则负责轮流调用 visit()方法。

注意，visit(:MachineComposite)方法内的循环，同样不关心它的子组件的实例是 Machine 类型，还是 MachineComposite 类型。visit()方法简单地调用每个组件的 accept()操作。究竟调用会执行哪个方法，取决于子组件的类型。图 29.4 展示了方法调用的典型顺序。

执行 visit(:MachineComposite)方法时，它会调用每个合成子对象的 accept()方法。子对象会通过调用 Visitor 对象的 visit()方法来回应。图 29.4 展示了从 Visitor 对象到接收 accept()调用的对象，以及各自接收对象之间的来回调用。这种技术被称为双重委派（double d:spatch），用于确保正确的 Visitor 类的 visit()方法被调用。

访问者模式中的双重委派使你可以创建访问类，这些访问类包含的方法可以指向被访问的类结构中不同的类型。

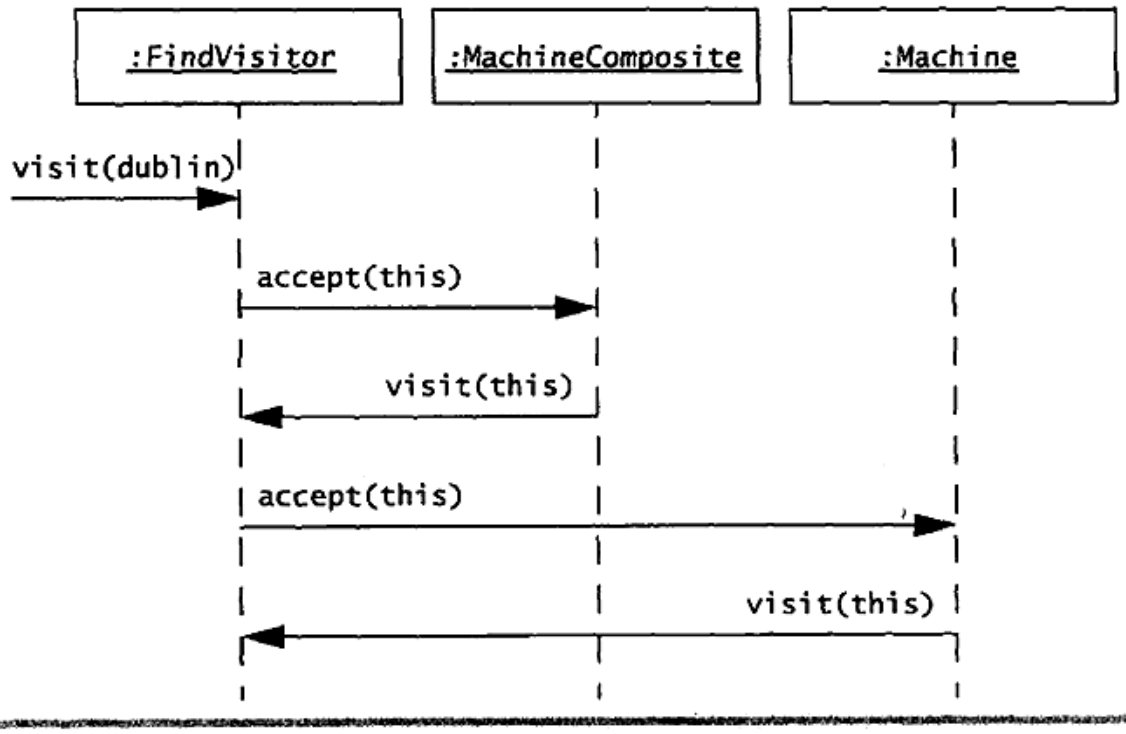

图 29.4 FindVisitor 对象调用 accept()方法，以决定执行哪个 visit()方法

倘若你能够控制源代码，通过运用访问者模式，几乎可以为源代码增加任意的行为。例如，可以增加一个访问者，它能够查找机器组件中的所有机器（叶子节点）：

```
package app.visitor;
import com.oozinoz.machine.*;
import java.util.*;

public class RakeVisitor implements MachineVisitor {
    private Set leaves;

    public Set getLeaves(MachineComponent mc) {
        leaves = new HashSet();
        mc.accept(this);
        return leaves;
    }

    public void visit(Machine m) {
        // 挑战!
    }

    public void visit(MachineComposite mc) {
        // 挑战!
    }
}
```

挑战 29.3

完成 RakeVisitor 类的代码，使其可以收集机器组件中的所有叶子节点。

答案参见第 364 页

这段简短的程序可以找到机器组件的叶子节点，并将其打印出来：

```
package app.visitor;

import com.oozinoz.machine.*;
import java.io.*;
import com.oozinoz.filter.WrapFilter;

public class ShowRakeVisitor {
    public static void main(String[] args)
            throws IOException {
        MachineComponent f = OozinozFactory.dublin();
        Writer out = new PrintWriter(System.out);
        out = new WrapFilter(new BufferedWriter(out), 60);
        out.write(
            new RakeVisitor().getLeaves(f).toString());
        out.close();
    }
}
```

这段程序使用逗号作为过滤器，产生如下输出：

```
[StarPress:3401, Fuser:3102, StarPress:3402, Mixer:3202,
Fuser:3101, StarPress:3403, ShellAssembler:1301,
ShellAssembler:2301, Mixer:1201, StarPress:2401, Mixer:3204,
Mixer:3201, Fuser:1101, Fuser:2101, ShellAssembler:3301,
ShellAssembler:3302, StarPress:1401, Mixer:3203, Mixer:2202,
StarPress:3404, Mixer:2201, StarPress:2402]
```

FindVisitor 类和 RakeVisitor 类都向 MachineComponent 类结构中添加了新的行为。看起来，这些类似乎工作正常。然而，在编写访问者类时可能存在障碍：这些访问者需要知道你想要扩展的类层次结构。一旦类层次结构发生变化，就可能破坏这些访问者类；但往往在一开

始时，我们对类结构机制的了解并非一清二楚，尤其是当需要访问的组合结构存在环时，更需要额外的处理。

Visitor 环

Oozinoz 公司使用 ProcessComponent 层次结构对工作流建模，该层级结构是另一种合成结构，并可以让其支持访问者模式而获益。与机器组合不同，它天生需要支持包含环结构的工作流，访问者必须保证在遍历过程组件时，不会陷入死循环。图 29.5 展示了 ProcessComponent 的层次结构。

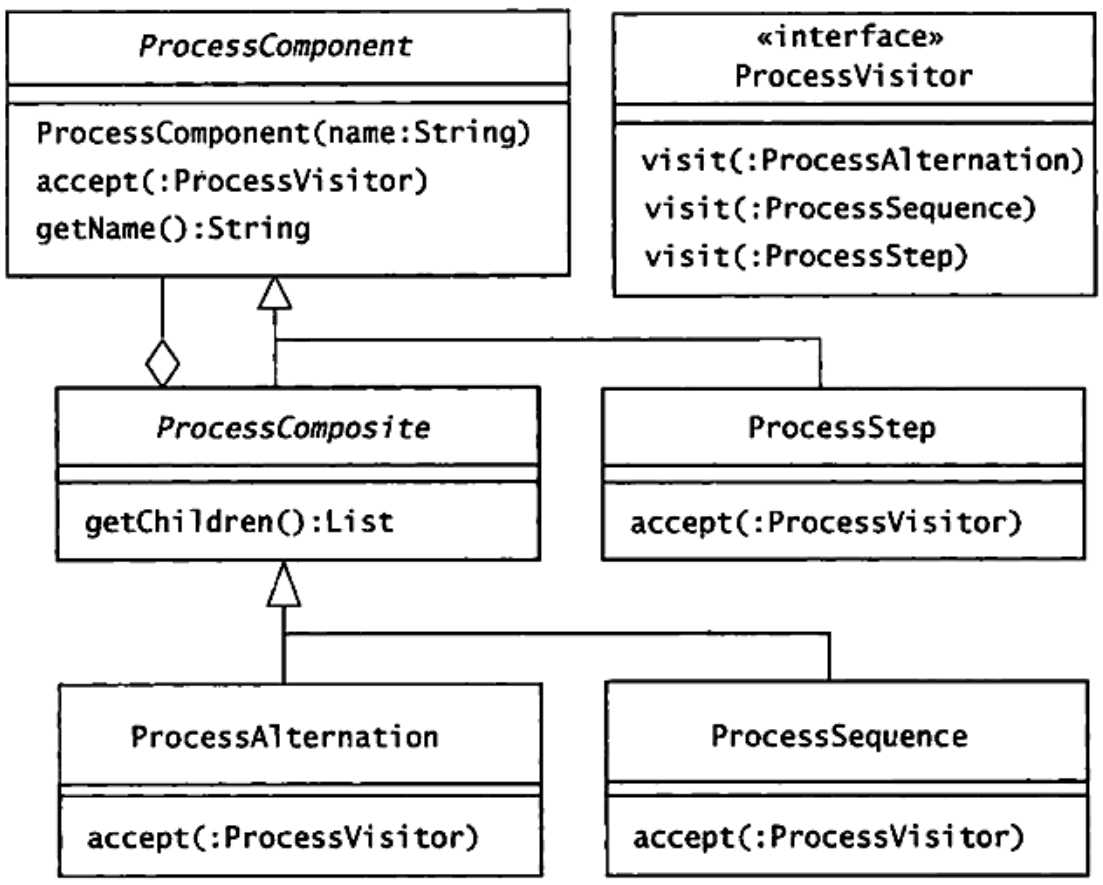

图 29.5 和 MachineComponent 层次结构一样，ProcessComponent 的层次结构可以支持访问者模式

假设希望打印出来的过程组件具有缩进的格式。在第 28 章的迭代器模式中，使用了迭代器来打印出一个工作流的步骤。打印输出如下所示：

```
Make an aerial shell
    Build inner shell
```

```
Inspect
Rework inner shell, or complete shell
    Rework
        Disassemble
    Finish: Attach lift, insert fusing, wrap
```

若要重新做一个焰火弹，需要先拆除，然后再重新组装。“拆除”之后的步骤是“制作焰火弹”。然而，输出的结果并未体现出这一步，因为迭代器发现该步骤在之前已经出现过一次。如果输出的步骤名称能够展现当前工作的步骤形成了一个环，将更有意义。若能标示出那些交替出现而非顺序出现的组件，同样很有帮助。

为了支持对工作流程的直观输出，可以创建一个访问者类，它会初始化一个StringBuilder对象，并把访问者类访问工作组件的过程记录在该对象中。为了标示出交替出现的组合步骤，访问者可以在这些步骤的名称前标记一个问号（?）。为了标示已经出现过的步骤，访问者可以在该步骤的名称后面加上省略号（…）。经过这些转换，焰火弹的生产过程会产生如下输出：

```
Make an aerial shell
    Build inner shell
    Inspect
    ?Rework inner shell, or complete shell
        Rework
            Disassemble
            Make an aerial shell…
        Finish: Attach lift, insert fusing, wrap
```

工作流组件的访问者类需要处理环的情况，通过使用Set对象来跟踪访问者业已访问的节点，可以轻而易举地实现。该类的代码如下：

```
package app.visitor;
import java.util.List;
import java.util.HashSet;
import java.util.Set;
import com.oozinoz.process.*;

public class PrettyVisitor implements ProcessVisitor {
    public static final String INDENT_STRING = " ";
    private StringBuffer buf;
    private int depth;
```

```
    private Set visited;
    public StringBuffer getPretty(ProcessComponent pc) {
        buf = new StringBuffer();
        visited = new HashSet();
        depth = 0;
        pc.accept(this);
        return buf;
    }

    protected void printIndentedString(String s) {
        for (inti = 0; i< depth; i++)
            buf.append(INDENT_STRING);
        buf.append(s);
        buf.append("\n");
    }
    // ... visit() 方法 ...
}
```

该类使用了 getPretty()方法来初始化实例变量，并实现访问算法。该算法使用 printIndentedString()方法来处理缩进。当访问 ProcessStep 对象时，代码只是打印出该步骤的名字：

```
public void visit(ProcessStep s) {
    printIndentedString(s.getName());
}
```

从图 29.5 中你可能会注意到，ProcessComposite 类没有实现 accept()方法，但它的子类却实现了。访问交替执行的流程与顺序执行的流程在逻辑上几乎完全相同，如下所示：

```
public void visit(ProcessAlternation a) {
    visitComposite("?", a);
}

public void visit(ProcessSequence s) {
    visitComposite("", s);
}
protected void visitComposite(
        String prefix, ProcessComposite c) {
    if (visited.contains(c)) {
        printIndentedString(prefix + c.getName() + "...");
```

```
        } else {
            visited.add(c);
            printIndentedString(prefix + c.getName());
            depth++;

            List children = c.getChildren();
            for (inti = 0; i<children.size(); i++) {
                ProcessComponent child =
                    (ProcessComponent) children.get(i);
                child.accept(this);
            }

            depth--;
        }
    }
```

二者的区别在于，交替节点会打印问号作为前缀。无论是哪一种类型，只要当前访问的节点曾经被访问过，就打印出该节点的名称和省略号。否则，就将该节点添加到已访问的节点集合中，并打印出前缀——问号或者什么都没有，然后调用子节点的 accept()方法。对于一个典型的访问者模式而言，代码使用多态来决定子节点是 ProcessStep、ProcessAlternation 还是 ProcessSequence 类的实例。

如下的小程序可以很好地打印出工作流：

```
package app.visitor;

import com.oozinoz.process.ProcessComponent;
import com.oozinoz.process.ShellProcess;

public class ShowPrettyVisitor {
    public static void main(String[] args) {
        ProcessComponent p = ShellProcess.make();
        PrettyVisitor v = new PrettyVisitor();
        System.out.println(v.getPretty(p));
    }
}
```

运行该程序，产生如下输出：

```
Make an aerial shell
```

```
Build inner shell
Inspect
?Rework inner shell, or complete shell
    Rework
        Disassemble
        Make an aerial shell...
    Finish: Attach lift, insert fusing, wrap
```

比起简单地迭代整个工作模型而言，这种方式的输出结果所能涵盖的信息更多。如果输出中出现问号，则表示该组合步骤是交替出现的。一旦节点出现两次，后面就会加上省略号，这比直接省略该重复步骤表达得更为清楚。

ProcessComponent 类结构的开发者通过定义 ProcessVisitor 接口以及 accept()方法，使得该层级结构支持访问者模式。这些开发者需要竭力避免在遍历工作流时出现死循环。如 PrettyVisitor 类所示，访问者类的开发者需要识别工作流组件中可能存在的环。倘若在开发 ProcessComponent 时，能够在一定程度上支持对环结构的管理，并将其作为支持访问者模式的一部分，就可以规避这些错误。

挑战 29.4

ProcessComponent 的开发者应该如何设计，才能保证该组件在支持访问者模式的同时支持对环的管理？

答案参见第 365 页

访问者模式的危机

一直以来，访问者模式颇具争议。一些开发者反对使用该模式，而另一些人则拥护该模式，并竭力寻找强化该模式的方法，尽管这些方法通常都会增加系统的复杂性。事实上，诸多设计问题都与使用访问者模式有关。

本章给出的例子同样彰显了访问者模式的脆弱之处。例如，在 MachineComponent 的层次结构中，开发者区分了 Machine 节点和 MachineComposite 节点，但却没有区分 Machine 子

类。如果需要区分访问类中不同机器的类型，就需要使用类型检查或其他技术，以辨别 visit() 方法接收的到底是哪种机器类型。或许你会认为，类层次的开发者应该在访问者接口中，提供一个 visit(:Machine)方法，以包含所有的机器类型。然而，随时都可能增加新的机器类型，因而这种设计并不健壮。

访问者模式是否是一个好的选择，取决于系统变化的特征：如果层次结构相对稳定，变化的总是行为，那么就是好的选择。如果行为相对稳定，但是结构总是变化，选择它就不恰当了，因为你需要更新现有的访问者类，使得它们可以支持新的节点类型。

ProcessComponent 层次结构中还存在另一种脆弱性。该结构的开发者知道工作流模型中环的危险性。但是，他们应该如何将这种顾虑告知访问者类的开发者呢？

这些问题可能暴露出访问者模式的一个根本问题：当需要扩展层次结构的行为时，通常需要一些层次设计的专业知识。如果缺乏这些专业知识，就可能陷入陷阱，例如无法避免在工作流中出现死循环。即使你拥有层次结构的专业知识，也可能会引入某种危险的依赖关系。一旦层次结构发生变化，这些依赖关系就可能会失效。这种专业知识以及对代码的控制使得访问者模式很难被应用。

计算机语言解析器是一个经典的运用访问者模式的范例，而且不会产生后续问题。解析器的开发者通常会开发一个抽象语法树，它可以根据语言的语法组织输入文字。如果希望为这棵树开发多种行为，则访问者模式就是最佳选择。在这个例子中，被访问的层次结构通常只有很少的行为，甚至没有行为。访问者类有责任处理所有设计好的行为，而避免像本章中的例子那样分离这些职责。

和任何模式一样，访问者模式不是必需的；如果需要使用该模式，就应该物尽其用。对于访问者模式而言，通常还有其他的替换方案提供更为健壮的设计。

挑战 29.5

请列出两种可替换的方案，用来替换 Oozinoz 公司的机器层次结构和工作流结构。

答案参见第 365 页

因此，若要判断是否应该使用访问者模式，最好满足如下条件：

- 节点类型的集合是稳定的。
- 共同发生的变化是为不同的节点添加新的功能。
- 新功能必须适用于所有节点类型。

小结

访问者模式使你可以在不改变类层次结构的前提下，为该结构增加新的行为。该模式的机制包括为访问者类定义一个接口，为层次关系中的所有访问者增加一个 accept()方法。accept()方法将使用双重委派技术，将其调用委派给访问者。类层次结构中的对象可以根据其类型调用合适的 visit()方法。

访问者类的开发者必须了解被访问者层次结构设计的微妙之处。特别的，访问者类需要注意被访问对象模型中的环结构。归咎于此，一些开发者会避免使用访问者模式，转而运用一些替换方式。是否应该运用访问者模式，通常取决于你的设计理念、团队情况以及具体的应用。