

第 25 章

解释器（Interpreter）模式

和命令模式一样，解释器模式也产生一个可执行的对象。不同的是，解释器模式创建了一个类层次，该层次中的每个类都实现或者解释了一个公共操作，并且该操作的名称与类名相同。从这点来看，解释器模式与状态模式和策略模式都很相似。在这些模式各自的类集合中，都有一个公共的方法，但是每个类对该方法的实现都不一样。

解释器模式和合成模式也很相似，合成模式通常会定义一个公共接口，该接口用于单个对象或者复合对象。合成模式并不要求必须以不同的方式组织结构，尽管该模式本身支持这种做法。例如第 5 章中的 `ProcessComponent` 层级，就允许顺序和可交替的两种工作流程。合成模式的要点是组合不同的类型（解释器模式通常是基于合成模式的）。一个类组合其他组件的方式定义了解释器类的实现方式。

要了解解释器模式，极具挑战。你可能需要重温合成模式，因为在本章，我们会经常用到该模式。

解释器模式的意图是让你根据事先定义好的一系列组合规则，组合可执行对象。

一个解释器示例

Oozinoz 公司使用解释器来控制机器人沿着生产线移动材料。你或许会认为解释器是一门

编程语言，其实它的核心是提供一组类，可以组合指令。Oozinoz 公司的机器人解释器就是一组封装了机器人指令集类层次。该层次的起点是一个抽象的 `Command` 类。分布在层次中的是一个 `execute()` 方法。图 25.1 展示了 `Robot` 类以及机器人解释器支持的两个命令。

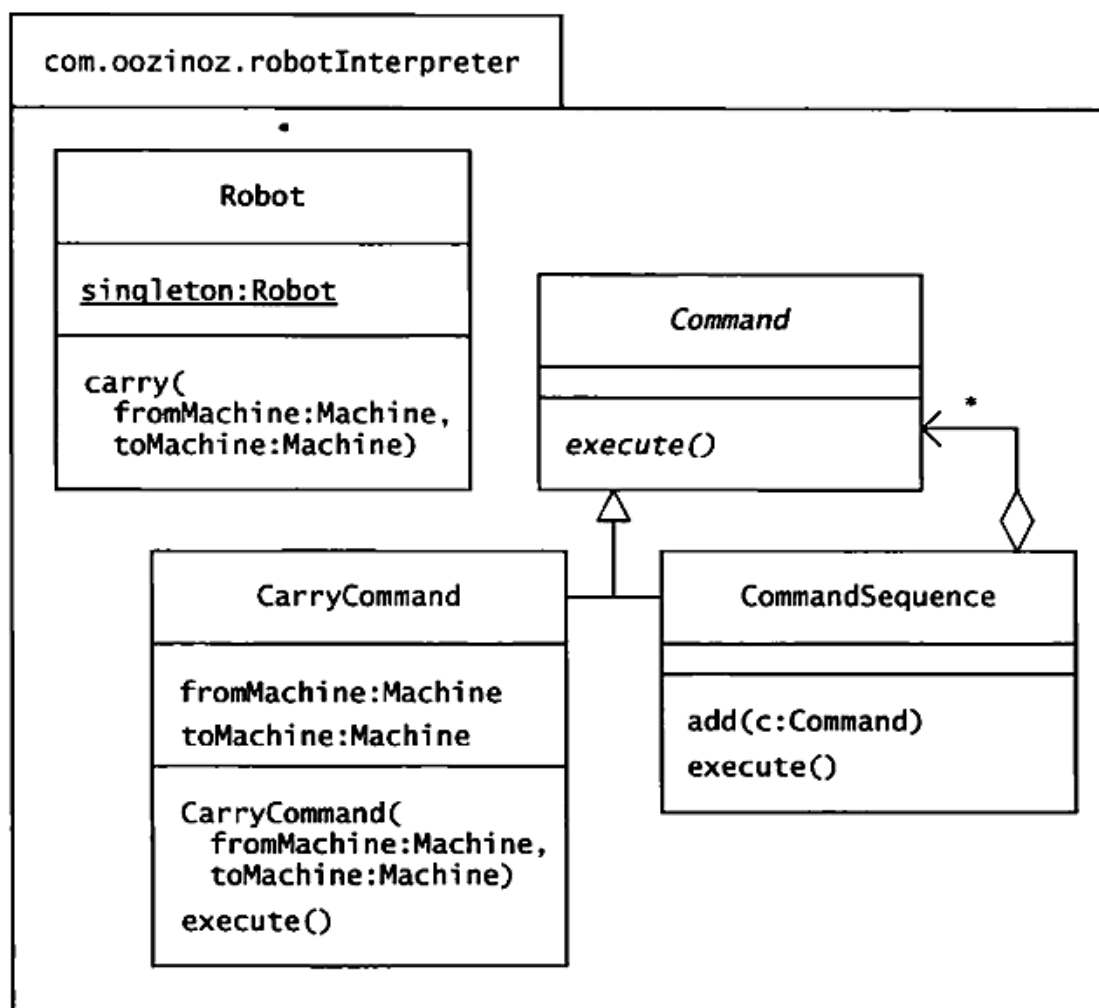


图 25.1 该解释器层次提供了对工厂机器人运行时编程的能力

粗略一看，你会以为图 25.1 是在介绍命令模式，因为层次结构的顶端定义了一个 `Command` 类。然而，命令模式的意图是将方法封装在对象的内部，而图 25.1 所示的 `Command` 层次并非如此，设计层次的目的是要求 `Command` 子类重新阐释 `execute()` 操作的含义。这正是解释器模式的意图：允许你组合可执行的对象。

典型的解释器层次包含的子类通常超过两个，我们马上就会扩展 `Command` 层次。图 25.1 所示的两个类作为第一个例子来讲已经足够了：

```

package app.interpreter;
import com.oozinoz.machine.*;
import com.oozinoz.robotInterpreter.*;
  
```

```
public class ShowInterpreter {
    public static void main(String[] args) {
        MachineComposite dublin = OozinozFactory.dublin();
        ShellAssembler assembler =
            (ShellAssembler) dublin.find("ShellAssembler:3302");
        StarPress press = (StarPress) dublin.find("StarPress:3404");
        Fuser fuser = (Fuser) dublin.find("Fuser:3102");

        assembler.load(new Bin(11011));
        press.load(new Bin(11015));

        CarryCommand carry1 = new CarryCommand(assembler, fuser);
        CarryCommand carry2 = new CarryCommand(press, fuser);

        CommandSequence seq = new CommandSequence();
        seq.add(carry1);
        seq.add(carry2);

        seq.execute();
    }
}
```

该示意代码让工厂机器人将两箱原材料从操作机上转移到卸载缓冲区里，需要用到 `OozinozFactory` 类的 `dublin()` 方法所返回的机器组合对象。该数据模型代表 Oozinoz 公司在爱尔兰都柏林的一个新的工厂设备。代码定位了工厂里的三台机器，将材料箱放在其中两台机器上，然后创建 `Command` 层次中的命令。该程序的最后一句调用了 `CommandSequence` 对象的 `execute()` 方法，让机器人按照 `seq` 中的指令执行操作。

`CommandSequence` 对象通过依次调用各个子命令来完成对 `execute()` 操作的解释。

```
package com.oozinoz.robotInterpreter;

import java.util.ArrayList;
import java.util.List;

public class CommandSequence extends Command {
    protected List commands = new ArrayList();
}
```

```
public void add(Command c) {
    commands.add(c);
}

public void execute() {
    for (inti = 0; i<commands.size(); i++) {
        Command c = (Command) commands.get(i);
        c.execute();
    }
}
}
```

CarryCommand 类的 execute() 方法实现如下：与工厂机器人交互，将箱子从一个机器人移到另一个机器人。

```
package com.oozinoz.robotInterpreter;
import com.oozinoz.machine.Machine;

public class CarryCommand extends Command {
    protected Machine fromMachine;
    protected Machine toMachine;

    public CarryCommand(
        Machine fromMachine, Machine toMachine) {
        this.fromMachine = fromMachine;
        this.toMachine = toMachine;
    }

    public void execute() {
        Robot.singleton.carry(fromMachine, toMachine);
    }
}
```

CarryCommand 类是为机器人控制生产线领域提供的专用设计。我们很容易想到与另一个领域相关的类，比如 StartupCommand 类或者 ShutdownCommand 类，用来控制机器。创建能操作一组机器的 ForCommand 类也很有用。图 25.2 展示了 Command 类层次的这些扩展。

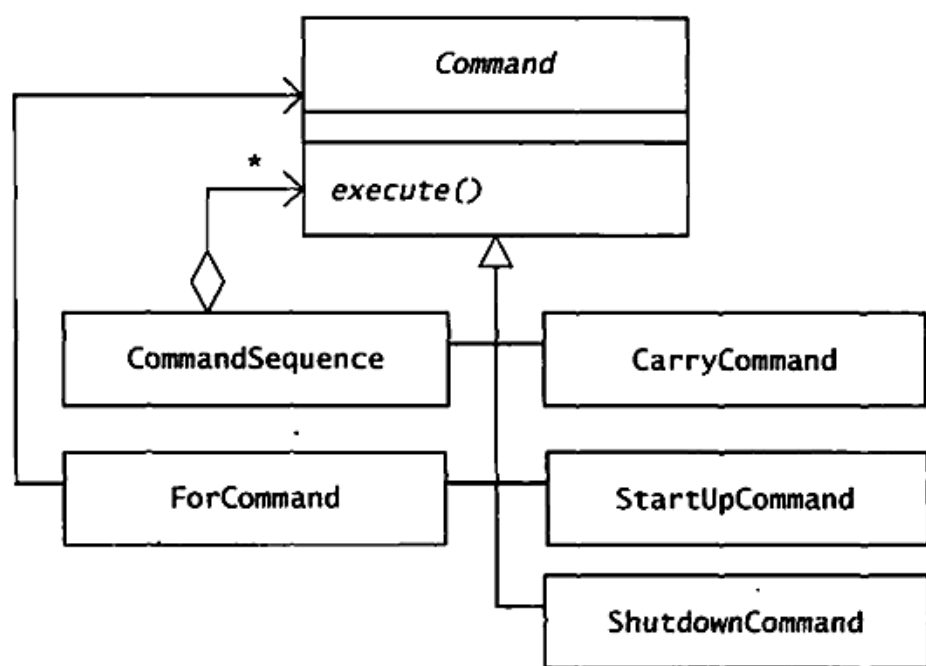


图 25.2 解释器模式允许多个子类重新解释通用操作的意义

`ForCommand` 类的一部分设计清晰明了。该类的构造函数假设接收一组机器和一个 `Command` 对象，以便执行 `for` 循环。该设计更加复杂的部分是如何将循环和方法体关联起来。Java 5 扩展了 `for` 语句，建立了一个变量，每次循环的时候都会给该变量附上新值。我们用下面的语句模拟该方法。

```
for (Command c: commands)
    c.execute();
```

Java 将 `c` 标识符与循环体中的 `c` 变量联系起来。为了创建一个解释器类来模拟该效果，我们需要一个方法对变量进行处理和运算。图 25.3 的 `Term` 层次做到了这点。

`Term` 层次与 `Command` 层次的相似之处在于，都有一个公共方法 `eval()` 贯穿整个层次。你可能会说该层次本身就是解释器模式自身的一个例子，尽管它不包含合成类。像 `CommandSequence` 这样的合成类常常会出现在解释器模式中。

`Term` 层次允许将单台机器都命名为常量，并允许我们将变量赋值给这些常量或者其他变量。该层次允许我们构建出更为灵活的与领域相关的解释器类。例如，`StartUpCommand` 代码可以和 `Term` 对象一起工作，而不仅仅是特定的机器：

```
package com.oozinoz.robotInterpreter2;
import com.oozinoz.machine.Machine;
```

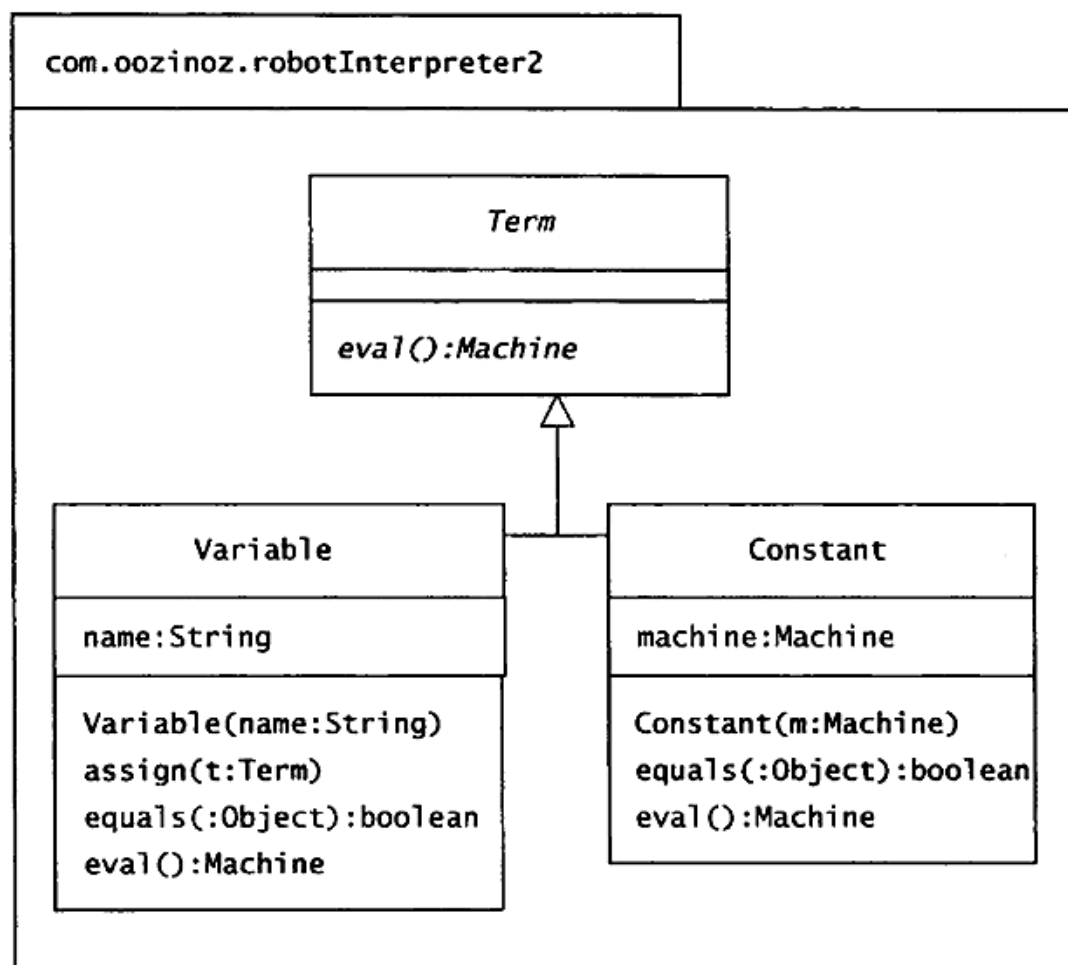


图 25.3 Term 类层次允许变量对机器进行处理

```

public class StartupCommand extends Command {
    protected Term term;

    public StartupCommand(Term term) {
        this.term = term;
    }

    public void execute() {
        Machine m = term.eval();
        m.startup();
    }
}
  
```

类似的，可以通过修改 CarryCommand 类，使其变得更加灵活：

```
package com.oozinoz.robotInterpreter2;
```

```
public class CarryCommand extends Command {
    protected Term from;
    protected Term to;

    public CarryCommand(Term fromTerm, Term toTerm) {
        from = fromTerm;
        to = toTerm;
    }

    public void execute() {
        Robot.singleton.carry(from.eval(), to.eval());
    }
}
```

一旦将 Command 层次设计成可以操作 term 对象，就可以写一个 ForCommand 类，使之允许设置变量的值，并能够在循环中执行方法体命令：

```
package com.oozinoz.robotInterpreter2;

import java.util.List;
import com.oozinoz.machine.Machine;
import com.oozinoz.machine.MachineComponent;
import com.oozinoz.machine.MachineComposite;

public class ForCommand extends Command {
    protected MachineComponent root;
    protected Variable variable;
    protected Command body;

    public ForCommand(
        MachineComponent mc, Variable v, Command body) {
        this.root = mc;
        this.variable = v;
        this.body = body;
    }

    public void execute() {
        execute(root);
    }
}
```

```

private void execute(MachineComponent mc) {
    if (mc instanceof Machine) {
        // 挑战!
        return;
    }

    MachineComposite comp = (MachineComposite) mc;
    List children = comp.getComponents();
    for (inti = 0; i<children.size(); i++) {
        MachineComponent child =
            (MachineComponent) children.get(i);
        execute(child);
    }
}
}

```

ForCommand 类的 execute() 方法遍历了机器的整棵组件树。第 28 章的迭代器模式将提供另一种更为优雅的方式来迭代组合结构。对解释器模式而言，要点是正确地解释树中每个节点的 execute() 请求。

挑战 25.1

完成 ForCommand 类中 execute() 方法的代码。

答案参见第 356 页

ForCommand 类允许我们为工厂编写“程序”或“脚本”命令。例如，如下的程序就编写了一个解释器，用来关闭工厂的所有机器：

```

package app.interpreter;

import com.oozinoz.machine.*;
import com.oozinoz.robotInterpreter2.*;

class ShowDown {
    public static void main(String[] args) {
        MachineComposite dublin = OozinozFactory.dublin();
        Variable v = new Variable("machine");
        Command c = new ForCommand(

```



```
        dublin, v, new ShutDownCommand(v));  
        c.execute();  
    }  
}
```

当程序调用 `execute()` 方法时, `ForCommand` 对象 `c` 会通过遍历所提供的机器组件, 为每个机器解释 `execute()` 方法:

- 为变量 `v` 赋值。
- 调用已经提供的 `ShutDownCommand` 对象的 `execute()` 方法。

如果再添加一些控制逻辑流程的类, 比如 `IfCommand` 类和 `whileCommand` 类, 就能构建一个完善的解释器了。这些类需要某些方法来建模 `Boolean` 条件。例如, 可能需要对机器变量是否等于某台特殊机器进行建模。我们可以为此引入一套新的术语层次结构, 但也可以借用 C 语言的想法来简化设计: 让 `null` 代表假, 其余值都代表真。有了这个想法, 就可以扩展 `Term` 类的层次结构, 如图 25.4 所示。

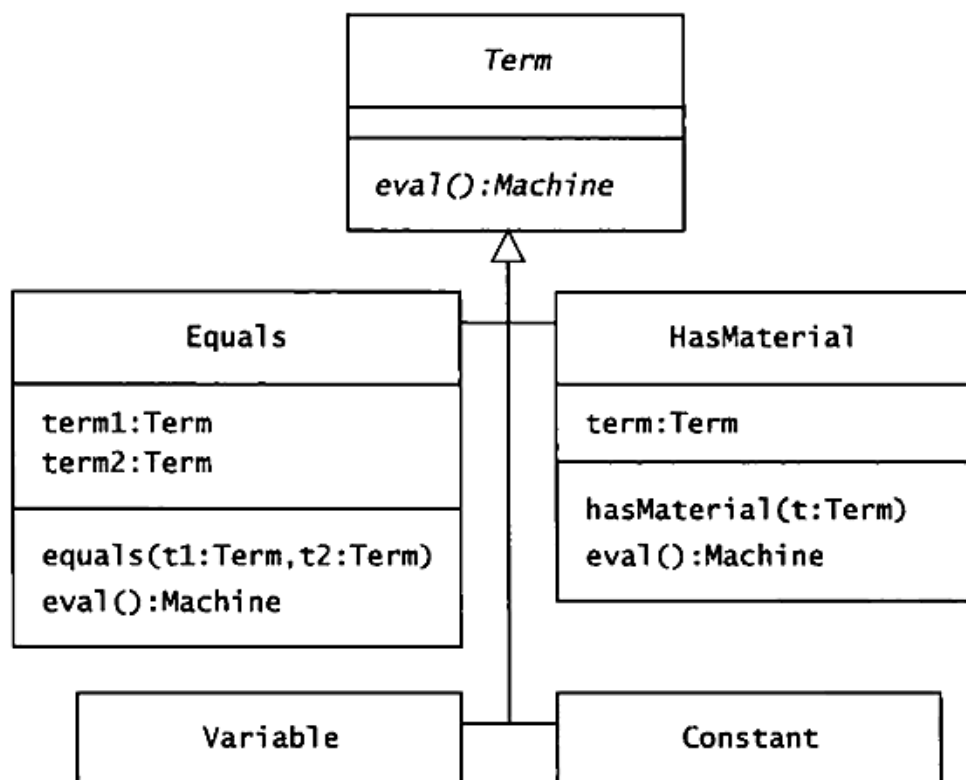


图 25.4 包括建模 `Boolean` 条件的 `Term` 类层次结构

`Equals` 类比较两个 `term` 对象, 并且返回 `null` 表示假。一个合理的设计是让 `Equals` 类包含 `eval()` 方法, 并且在两个 `term` 对象结果相等时, 返回其中一个结果:

```
package com.oozinoz.robotInterpreter2;
import com.oozinoz.machine.Machine;
public class Equals extends Term {
    protected Term term1;
    protected Term term2;
    public Equals(Term term1, Term term2) {
        this.term1 = term1;
        this.term2 = term2;
    }

    public Machine eval() {
        Machine m1 = term1.eval();
        Machine m2 = term2.eval();
        return m1.equals(m2) ? m1 : null;
    }
}
```

HasMaterial 类将 Boolean 类中值的概念扩展到了特定领域，代码如下所示：

```
package com.oozinoz.robotInterpreter2;

import com.oozinoz.machine.Machine;

public class HasMaterial extends Term {
    protected Term term;

    public HasMaterial(Term term) {
        this.term = term;
    }

    public Machine eval() {
        Machine m = term.eval();
        return m.hasMaterial() ? m : null;
    }
}
```

既然已经将 Boolean 条件术语引入到解释器包中，我们就可以添加流程控制类，如图 25.5 所示。

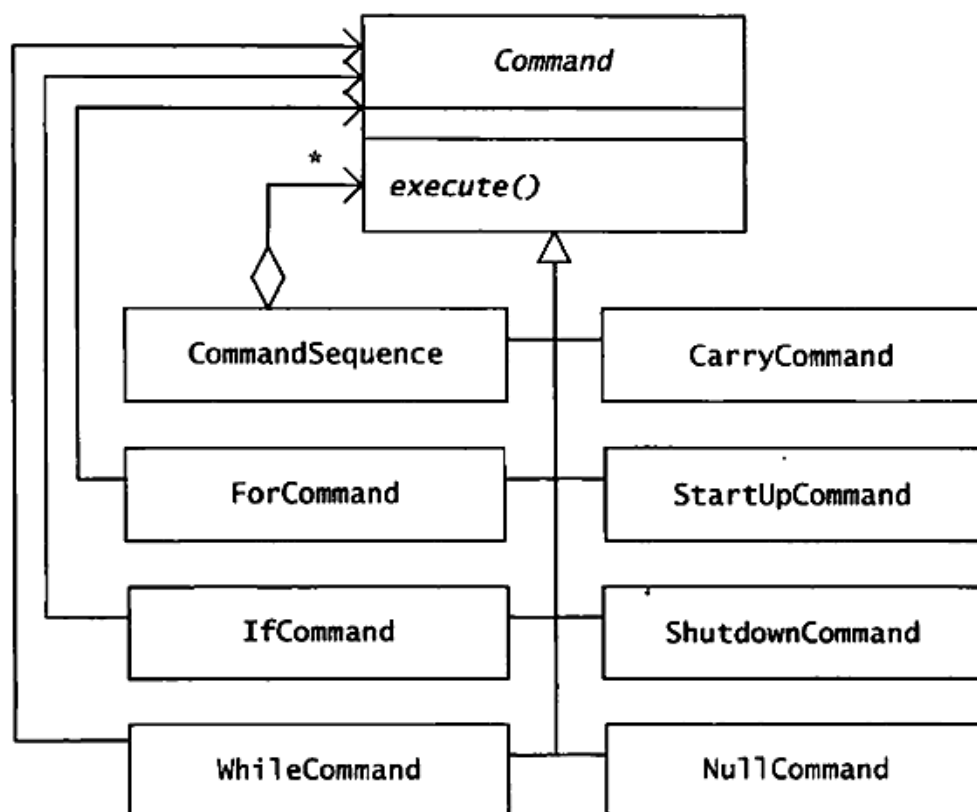


图 25.5 可以通过为解释器层次结构增加一个逻辑控制流类，使其更加完善

当需要一个什么也不执行的命令时，NullCommand 将很有用。例如，if 命令的 else 分支为空时：

```

package com.oozinoz.robotInterpreter2;
public class NullCommand extends Command {
    public void execute() {
    }
}

package com.oozinoz.robotInterpreter2;

public class IfCommand extends Command {
    protected Term term;
    protected Command body;
    protected Command elseBody;

    public IfCommand(
        Term term, Command body, Command elseBody) {
        this.term = term;
        this.body = body;
        this.elseBody = elseBody;
    }
}
  
```

```
    public void execute() {  
        // 挑战!  
    }  
}
```

挑战 25.2

完成 IfCommand 类的 execute() 方法。

答案参见第 357 页

挑战 25.3

写出 whileCommand 类的代码。

答案参见第 357 页

你可能会将 whileCommand 类与卸载火药填压机的解释器一起使用：

```
package app.interpreter;  
  
import com.oozinoz.machine.*;  
import com.oozinoz.robotInterpreter2.*;  
  
public class ShowWhile {  
    public static void main(String[] args) {  
        MachineComposite dublin = OozinozFactory.dublin();  
        Term starPress = new Constant(  
            (Machine) dublin.find("StarPress:1401"));  
        Term fuser = new Constant(  
            (Machine) dublin.find("Fuser:1101"));  
  
        starPress.eval().load(new Bin(77));  
        starPress.eval().load(new Bin(88));  
  
        whileCommand command = new whileCommand(  
            new HasMaterial(starPress),  
            new CarryCommand(starPress, fuser));  
    }  
}
```

```
        command.execute();
    }
}
```

`command` 对象是一个解释器，它的 `execute()` 方法会将所有的箱子从火药填压机 1401 上卸载下来。

挑战 25.4

合上本书，请写出命令模式和解释器模式的区别。

答案参见第 357 页

可以为解释器类层次增加更多的类，以此增加对更多领域相关任务类型控制的支持。也可以扩展 `Term` 层次结构。譬如增加一个 `Term` 的子类，使其可以找到其他机器附近的卸载缓冲区。

`Command` 和 `Term` 类层次的用户可以任意组合丰富而复杂的执行“程序”。例如，可以轻而易举地创建这样一个对象：在执行时，可以卸载工厂中除卸载缓冲区外的其他机器上的原材料。可以用如下伪代码来描述该程序：

```
for (m in factory)
    if (not (m is unloadBuffer))
        ub = findUnload for m
        while (m hasMaterial)
            carry (m, ub)
```

如果用 Java 代码来实现这些功能，会比这些伪代码要冗余复杂得多。那么，为何不创建一个领域相关的解析器，负责将这些伪代码转换成实际的代码，使其可以管理工厂的原材料，并为我们创建相应的解释器对象呢？

解释器、语言和解析器

解释器模式仅仅展现了解释器是如何工作的，却并没有指出应该如何实例化以及如何组合这些对象。在本章，我们通过几行 Java 代码“手工”地创建了一个新的解释器；然而，更为常见的方式却是通过解析器 (Parser) 来创建。解析器对象可以根据一组规则识别和分解文本结构，

使其转换为更加适宜的形式，以便进一步进行处理。例如，可以为早期的伪代码文本程序创建一个机器命令的解释器对象。

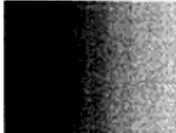
在撰写本书时，市面上关于用 Java 写解析器的资料很少。想要获取更多的资料，可以尝试在网上搜索“Java 解析器工具（Java parser tools）”。大多数的解析器工具都包含一个解析器的生成器。要使用该生成器，需要使用指定的语法或语言来描述该模式，工具会根据你的描述生成一个解析器。生成的解析器会识别你语言中的实例。相对于使用解析器生成器，还可以运用解释器模式编写通用目的的解释器。*Building Parsers with Java™*（由 Metsker 在 2001 年编写）一书使用 Java 语言解释了该技术。

小结

解释器模式可以让你根据创建类层次结构来组合可执行对象。层次结构中的每个类都实现了一个公共的操作，比如 `execute()`。尽管之前的例子没有讨论，但该方法通常都需要一个额外的“上下文”对象，用来存储重要的状态信息。

每个类的名称通常暗指该类所要实现或解释的通用操作。每个类要么定义合成命令的方式，要么表示执行某些操作的终端命令。

设计解释器常常会引入变量，以及布尔或者算术表达式。解释器也经常和解析器一起使用，用来创建一个轻量级的语言，简化新的解释器对象的创建。



第 5 部分

扩展型模式