

解释器模式

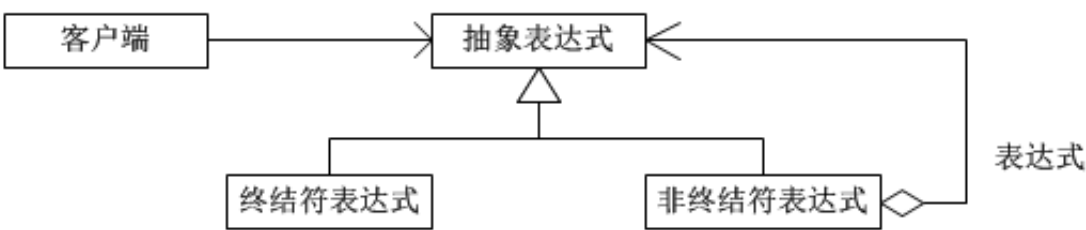
整理自：《java与模式》之解释器模式

在阎宏博士的《JAVA与模式》一书中开头是这样描述解释器（Interpreter）模式的：

解释器模式是类的行为模式。给定一个语言之后，解释器模式可以定义出其文法的一种表示，并同时提供一个解释器。客户端可以使用这个解释器来解释这个语言中的句子。

解释器模式的结构

下面就以一个示意性的系统为例，讨论解释器模式的结构。系统的结构图如下所示：



模式所涉及的角色如下所示：

- **抽象表达式(Expression)角色：**声明一个所有的具体表达式角色都需要实现的抽象接口。这个接口主要是一个interpret()方法，称做解释操作。
- **终结符表达式(Terminal Expression)角色：**实现了抽象表达式角色所要求的接口，主要是一个interpret()方法；文法中的每一个终结符都有一个具体终结表达式与之相对应。比如有一个简单的公式 $R=R1+R2$ ，在里面 $R1$ 和 $R2$ 就是终结符，对应的解析 $R1$ 和 $R2$ 的解释器就是终结符表达式。
- **非终结符表达式(Nonterminal Expression)角色：**文法中的每一条规则都需要一个具体的非终结符表达式，非终结符表达式一般是文法中的运算符或者其他关键字，比如公式 $R=R1+R2$ 中，“+”就是非终结符，解析“+”的解释器就是一个非终结符表达式。
- **环境(Context)角色：**这个角色的任务一般是用来存放文法中各个终结符所对应的具体值，比如 $R=R1+R2$ ，我们给 $R1$ 赋值100，给 $R2$ 赋值200。这些信息需要存放到环境角色中，很多情况下我们使用Map来充当环境角色就足够了。

为了说明解释器模式的实现办法，这里给出一个最简单的文法和对应的解释器模式的实现，这就是模拟Java语言中对布尔表达式进行操作和求值。

在这个语言中终结符是布尔变量，也就是常量true和false。非终结符表达式包含运算符and，or和not等布尔表达式。这个简单的文法如下：

Expression ::= Constant | Variable | Or | And | Not

And ::= Expression ‘AND’ Expression

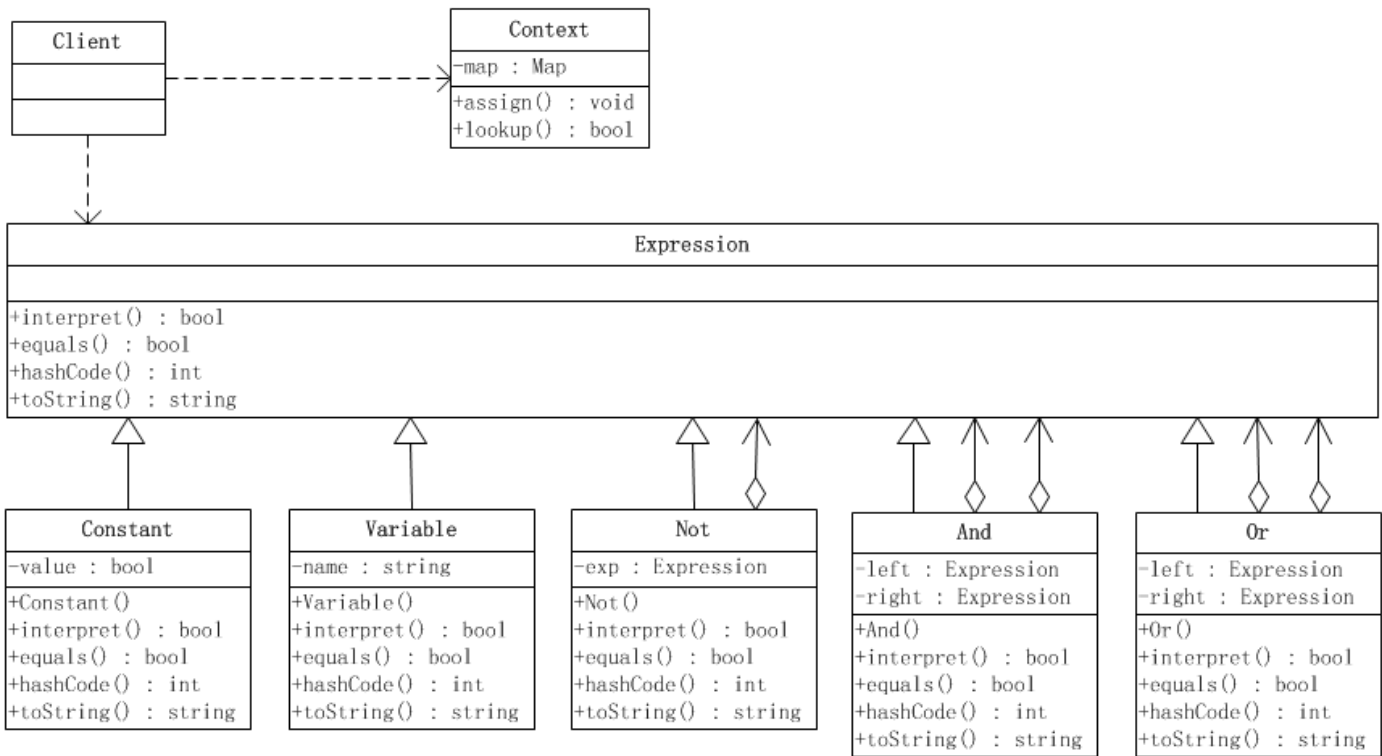
Or ::= Expression ‘OR’ Expression

Not ::= ‘NOT’ Expression

Variable ::= 任何标识符

Constant ::= ‘true’ | ‘false’

解释器模式的结构图如下所示：



源代码

抽象表达式角色

```
public abstract class Expression {
    /**
     * 以环境为准，本方法解释给定的任何一个表达式
    */
}
```

```

    */
    public abstract boolean interpret(Context ctx);
    /**
     * 检验两个表达式在结构上是否相同
     */
    public abstract boolean equals(Object obj);
    /**
     * 返回表达式的hash code
     */
    public abstract int hashCode();
    /**
     * 将表达式转换成字符串
     */
    public abstract String toString();
}

```

一个Constant对象代表一个布尔常量

```

public class Constant extends Expression{

    private boolean value;

    public Constant(boolean value){
        this.value = value;
    }

    @Override
    public boolean equals(Object obj) {

        if(obj != null && obj instanceof Constant){
            return this.value == ((Constant)obj).value;
        }
        return false;
    }

    @Override
    public int hashCode() {
        return this.toString().hashCode();
    }
}

```

```

@Override
public boolean interpret(Context ctx) {

    return value;
}

@Override
public String toString() {
    return new Boolean(value).toString();
}

}

```

一个Variable对象代表一个有名变量

```

public class Variable extends Expression {

    private String name;

    public Variable(String name){
        this.name = name;
    }
    @Override
    public boolean equals(Object obj) {

        if(obj != null && obj instanceof Variable)
        {
            return this.name.equals(
                ((Variable)obj).name);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return this.toString().hashCode();
    }

    @Override
    public String toString() {

```

```

        return name;
    }

    @Override
    public boolean interpret(Context ctx) {
        return ctx.lookup(this);
    }
}

```

代表逻辑“与”操作的And类，表示由两个布尔表达式通过逻辑“与”操作给出一个新的布尔表达式的操作

```

public class And extends Expression {

    private Expression left, right;

    public And(Expression left , Expression right){
        this.left = left;
        this.right = right;
    }
    @Override
    public boolean equals(Object obj) {
        if(obj != null && obj instanceof And)
        {
            return left.equals(((And)obj).left) &&
                right.equals(((And)obj).right);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return this.toString().hashCode();
    }

    @Override
    public boolean interpret(Context ctx) {

        return left.interpret(ctx) && right.interpret(ctx);
    }
}

```

```

    }

    @Override
    public String toString() {
        return "(" + left.toString() + " AND " + right.toString() + ")";
    }
}

```

代表逻辑“或”操作的Or类，代表由两个布尔表达式通过逻辑“或”操作给出一个新的布尔表达式的操作

```

public class Or extends Expression {
    private Expression left, right;

    public Or(Expression left , Expression right){
        this.left = left;
        this.right = right;
    }
    @Override
    public boolean equals(Object obj) {
        if(obj != null && obj instanceof Or)
        {
            return this.left.equals(((Or)obj).left) && this.right.equals
(((Or)obj).right);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return this.toString().hashCode();
    }

    @Override
    public boolean interpret(Context ctx) {
        return left.interpret(ctx) || right.interpret(ctx);
    }

    @Override

```

```

    public String toString() {
        return "(" + left.toString() + " OR " + right.toString() + ")";
    }
}

```

代表逻辑“非”操作的Not类，代表由一个布尔表达式通过逻辑“非”操作给出一个新的布尔表达式的操作

```

public class Not extends Expression {

    private Expression exp;

    public Not(Expression exp){
        this.exp = exp;
    }
    @Override
    public boolean equals(Object obj) {
        if(obj != null && obj instanceof Not)
        {
            return exp.equals(
                ((Not)obj).exp);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return this.toString().hashCode();
    }

    @Override
    public boolean interpret(Context ctx) {
        return !exp.interpret(ctx);
    }

    @Override
    public String toString() {
        return "(Not " + exp.toString() + ")";
    }
}

```

```
}
```

环境(Context)类定义出从变量到布尔值的一个映射

```
public class Context {

    private Map<Variable, Boolean> map = new HashMap<Variable, Boolean>();

    public void assign(Variable var , boolean value){
        map.put(var, new Boolean(value));
    }

    public boolean lookup(Variable var) throws IllegalArgumentException{
        Boolean value = map.get(var);
        if(value == null){
            throw new IllegalArgumentException();
        }
        return value.booleanValue();
    }
}
```

客户端类

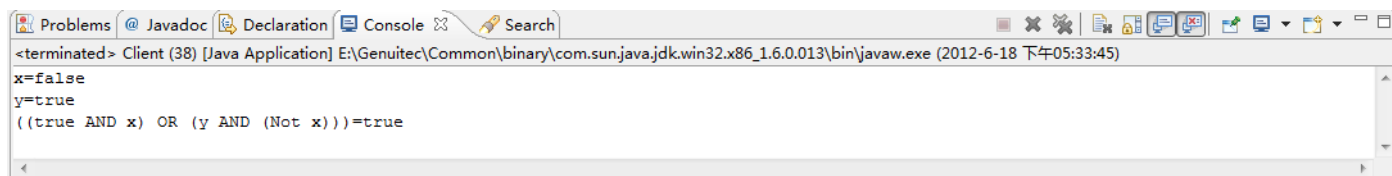
```
public class Client {

    public static void main(String[] args) {
        Context ctx = new Context();
        Variable x = new Variable("x");
        Variable y = new Variable("y");
        Constant c = new Constant(true);
        ctx.assign(x, false);
        ctx.assign(y, true);

        Expression exp = new Or(new And(c, x) , new And(y, new Not(x)));
        System.out.println("x=" + x.interpret(ctx));
        System.out.println("y=" + y.interpret(ctx));
        System.out.println(exp.toString() + "=" + exp.interpret(ctx));
    }
}
```


}

运行结果如下：



The screenshot shows an IDE's console window with the following content:

```
<terminated> Client (38) [Java Application] E:\Genuitec\Common\binary\com.sun.java.jdk.win32.x86_1.6.0.013\bin\javaw.exe (2012-6-18 下午05:33:45)  
x=false  
y=true  
((true AND x) OR (y AND (Not x)))=true
```