

个, `Position3D` 类独有的 `z` 字段则作为数组的第 3 个元素保存, 以此类推。

我们来讨论一下不采用这种设计可能会带来什么问题。假如 `Position` 类的 `z` 字段保存在数组的第 2 个元素, `Position3D` 类的 `z` 字段保存在数组的第 1 个元素中, 于是, `xmove` 方法要引用 `x` 字段时, 就必须在执行过程中判断目标对象是 `Position` 类还是 `Position3D` 类, 降低了程序的运行速度。

此外, 我们还可以分别为 `Position` 类与 `Position3D` 类准备不同的 `xmove` 方法及相应的抽象语法树。在与 `Position` 类对应的抽象语法树中, `x` 字段保存在数组的第 2 个元素中, 在与 `Position3D` 类对应的抽象语法树中, 该字段保存在第 1 个元素中。这种做法虽然不会降低运行性能, 但内存使用量较大。

F 由于 Stone 语言仅支持单一继承, 所以能够使用上面的方法。如果是多重继承, 即使我们想让字段保存在同一个位置, 也无法实现。

A C++ 的虚函数表 (virtual function table) 虽然与之类似, 但同时也支持多重继承。不过它实际上仍然无法将数据保存在同一位置, 只是通过多种手段使用户以为相关数据都在同一位置保存。

H 对了, 虚函数表是虚函数的指针数组, 它的本质就是一个方法数组, A 君。

C Java 语言的接口也是一种特殊的多重继承类, 因此具有相同的问题。早期的 Java 语言如果调用了接口的方法, 执行速度就会很慢。随着研究的深入, 这一问题才得以解决。

12.3 定义 lookup 方法

上一章为抽象语法树节点类定义的 `lookup` 方法将在程序执行前查找语法树, 确定所有需要使用的变量的保存位置。这些信息将同时保存至引用了变量的相应节点对象中, 之后语言处理器将能通过编号而非变量名获取变量值, 程序运行过程中的变量引用速度得到了提升。

`lookup` 方法的参数中包含一个 `Symbols` 对象, 该方法在遇到赋值表达式的左侧时, 会将左侧的变量名传递给 `Symbols` 对象的 `put` 方法。如果这个名称是第一次出现, 它将被添加至 `Symbols` 对象之中。于是, 在执行完 `lookup` 方法后, `Symbols` 对象中会新增一些变量名。语言处理器可以通过它们获取程序所需变量的名称与相应的保存位置一览。

本章将扩展 `class` 语句, 使它能够正确处理 `lookup` 方法。通过这次扩展, `lookup` 方法将能查找 `class` 语句中与由大括号括起的类定义体对应的抽象语法树, 向 `Symbols` 对象添加该类中所有的方法名与字段名, 同时为它们分配保存位置。`lookup` 方法执行结束后, 语言处理器将能通过 `Symbols` 对象获得方法名与字段名一览。

在扩展 `lookup` 方法时, 我们可以直接使用上一章定义的 `lookup` 方法的大部分代码。大括号括起的类定义体相当于类的构造函数, 它在执行过程中使用的局部变量可以视为字段。在通过闭包实现对象机制时, 我们也利用了这一特性。因此, 我们可以借助已有的 `lookup` 方法来查找所有需要使用的变量。加之方法和函数都能由 `def` 语句定义, Stone 语言处理器具有相当高的

代码复用比例。

对 lookup 方法进行的扩展还未完成，我们需要设法让它不仅能够在引用了变量的抽象语法树节点对象中记录该变量的保存位置，还能记录方法与字段的相应保存位置。这里同样能够复用之前的代码，简化实现的难度。该复用只适用于语言处理器在调用 this 对象的方法，或访问 this 对象的字段时语句省略了 this. 的情况。也就是说，lookup 方法不支持 this.x 这样的形式，如果要正确引用，就只能单独使用一个 x。

由于这与通常的变量引用形式相同，因此我们可以利用上一章中的 lookup 方法实现。唯一需要修改的是，Symbols 对象返回的保存位置，应当通过保存方法定义与字段值的数组表示。

F 我想起来了，之前第 9 章（第 9 天）也利用了闭包与类在定义上的相似性呢。

C 嗯，这里也是同样的思路。因为 lookup 的实现与 eval 的实现的基本结构相同。

本章的 lookup 方法实现将尽可能复用之前的代码，不过，参数中的 Symbols 对象需要做一些调整。这里的 Symbols 对象相当于 eval 方法接收的 Environment 对象。它们都能通过多个串连的方式来表示作用域的嵌套结构。

图 12.2 是 lookup 方法在查找大括号括起的类定义体时使用的 Symbols 对象。这 4 个对象通过 outer 字段连接，分别记录了不同类型的名称。除了最后一个，这些对象都属于 Symbols 类的子类。

代码清单 12.4 是图中第一个出现的 SymbolThis 对象的定义。它用于记录在类定义体中有效的局部变量的名称。然而，与函数不同，类定义体中新增的名称并非局部变量，而是字段名称，因此该作用域内的有效局部变量就只有一个 this。SymbolThis 对象仅会记录 this 的信息。

下面这两个 MemberSymbols 对象分别用于记录字段名与方法名。代码清单 12.5 是它们的定义。如果 lookup 方法在类定义中遇到了用于定义方法的 def 语句，就会直接将该方法名称添加至第二个 MemberSymbols 对象中。

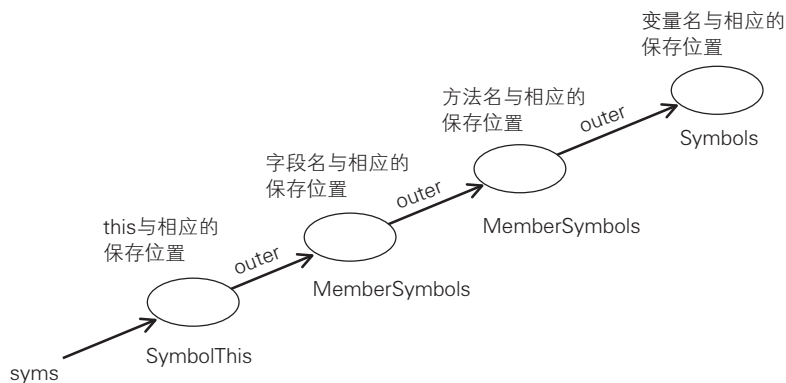


图 12.2 相互串连的 Symbols 对象

字段名的添加过程有些复杂。lookup 方法如果在类定义体中遇到了赋值表达式，将首先检查表达式左侧是否含有新出现的名称，如果该名称是第一次出现，lookup 方法将调用图 12.2 中第一个 SymbolThis 对象的 put 方法来添加名称。

由于类定义体中出现的名称不是局部变量，而是字段名，因此 put 方法将调用 outer 指向的 MemberSymbols 对象提供的 put 方法，而非直接通过自身 (SymbolThis 对象的 put 方法) 来添加名称。outer 字段所指的对象用于记录字段名。由此可知，之所以图 12.2 中的对象链不得不以 SymbolThis 代替 Symbol 对象起始，是由于 put 方法必须按这样的形式做相应修改。

MemberSymbols 对象用于记录方法名、字段名及与之相应的保存位置。这些保存位置的值由 Location 对象表示。该对象具有 nest 字段与 index 字段，且由 MemberSymbols 对象返回的 Location 对象中，nest 字段的值只可能是 METHOD 或 FIELD（它们是两个不同的负整数常量）。通常，nest 字段用于表示该名称属于从最内层数起的第几个作用域，index 字段用于表示与该名称对应的值保存于数组中的第几个元素。通过 METHOD 与 FIELD 这两个特殊的常量，MemberSymbols 对象能够仅凭 nest 字段的值来判断一个名称是否是通常的变量名。Location 对象无需记录更多额外信息，例如，它不必知道方法或字段具体属于哪一个类。这是因为，只有在对 this 对象调用方法或引用字段时，语言处理器才需要记录它们的保存位置。

代码清单 12.4 SymbolThis.java

```
package chap12;
import stone.StoneException;
import chap11.Symbols;

public class SymbolThis extends Symbols {
    public static final String NAME = "this";
    public SymbolThis(Symbols outer) {
        super(outer);
        add(NAME);
    }
    @Override public int putNew(String key) {
        throw new StoneException("fatal");
    }
    @Override public Location put(String key) {
        Location loc = outer.put(key);
        if (loc.nest >= 0)
            loc.nest++;
        return loc;
    }
}
```

代码清单 12.5 MemberSymbols.java

```
package chap12;
import chap11.Symbols;

public class MemberSymbols extends Symbols {
    public static int METHOD = -1;
    public static int FIELD = -2;
```

```

protected int type;
public MemberSymbols(Symbols outer, int type) {
    super(outer);
    this.type = type;
}
@Override public Location get(String key, int nest) {
    Integer index = table.get(key);
    if (index == null)
        if (outer == null)
            return null;
        else
            return outer.get(key, nest);
    else
        return new Location(type, index.intValue());
}
@Override public Location put(String key) {
    Location loc = get(key, 0);
    if (loc == null)
        return new Location(type, add(key));
    else
        return loc;
}
}

```

12.4 整合所有修改并执行

代码清单 12.6 根据前一节的实现思路设计了修改器，它们将对语言处理器进行修改与扩展。其中，为与 class 语句相关的抽象语法树节点类添加相应的 eval 与 lookup 方法是最主要的改动。

首先，修改器为直接与 class 语句对应的 ClassStmnt 类添加了空的 lookup 方法。eval 方法将执行与 lookup 方法功能相当的操作。如果程序定义的需要继承一个父类，但环境中没有记录这一父类，语言处理器就找不到父类的定义，从而不能确定需要继承的方法与字段，这样一来，lookup 方法就无法执行。因此，eval 方法需要接收一个环境参数，同时 lookup 方法的执行也将推迟。ClassStmnt 类的 eval 方法将创建一个 OptClassInfo 对象并添加至环境中，用于保存当前定义的类的信息。如果该类继承了父类的方法或字段，OptClassInfo 对象中也将添加这些信息。完成以上这些操作之后，语言处理器将对类定义体调用 lookup 方法。

ClassBody 类新增的 lookup 方法仅会对 def 语句做特殊处理。类定义体中的 def 语句用于定义方法，此处定义的方法将保存在 MemberSymbols 对象内，即图 12.2 中左起第 3 个椭圆表示的对象。lookup 方法将检查该方法是否已经存在，并根据情况判断是否应该覆盖已有的方法，执行恰当的处理。最后，lookup 方法将对 def 语句调用 lookupAsMethod 方法。由于方法定义与函数定义稍有不同，需要做一些特别的处理，因此这里不能直接使用 lookup，而要通过另外的方法来完成操作。

代码清单 12.6 中的修改器为 `DefStmt` 类添加了 `lookupAsMethod` 方法，它将在 `def` 语句进行方法定义时执行 `lookup` 处理。`lookupAsMethod` 方法将创建一个与方法作用域对应的 `Symbols` 对象，并以此为参数调用方法本身的 `lookup` 方法。该方法与 `def` 语句在定义函数时使用的由 `DefStmt` 类提供的 `lookup` 方法，或 `Fun` 类的 `lookup` 方法的不同之处在于，它将在创建 `Symbols` 对象后首先为它添加一个变量名 `this`。

接下来，我们只需为剩下的类添加 `eval` 或与之相当的方法即可。`Dot` 类新增的 `eval` 方法将在 `.`（点运算符）的左侧是类名而右侧是 `new` 表达式时，创建一个新的 `Stone` 语言对象，否则读取左侧对象的字段值。该字段的值能够通过 `OptStoneObject` 类的 `read` 方法获得。这里的 `eval` 方法与第 9 章（第 9 天）代码清单 9.6 中为 `Dot` 类添加的 `eval` 方法大同小异，请读者比较一下两者的不同。

`Dot` 类新增的 `eval` 方法在创建 `Stone` 语言对象时，将首先创建一个 `OptStoneObject` 对象，然后调用 `initObject` 方法初始化该对象。`initObject` 方法将把 `class` 语句中大括号括起的类定义体作为构造函数执行。如果该类具有父类，`initObject` 方法将先执行父类的构造函数。由于构造函数内部需要使用独立的作用域，因此 `initObject` 方法将创建一个新的环境来执行初始化处理。

`initObject` 方法创建的环境是一个长度为 1 的数组，它仅保存了 `this` 的值。`eval` 方法通过下面这条语句完成数组的初始化，并将 `this` 的值保存至数组的第 1 个元素中。

```
newEnv.put(0, 0, so);
```

由于新创建的环境 `newEnv` 的 `outer` 字段指向用于记录全局变量的值的环境，因此全局变量能够在构造函数中直接引用。该环境也用于类的定义，能够通过调用 `OptClassInfo` 对象的 `environment` 方法获得。

`Name` 类新增的 `eval` 方法能够从之前由 `lookup` 方法记录的保存位置获取与名称对应的值。这里的 `lookup` 方法已由上一章代码清单 11.4 中的 `NameEx` 修改器添加。

如果需要的值保存在对象中，语言处理器将通过 `getThis` 方法从环境中取得 `this` 指向的对象。这种情况下，程序必须知道 `this` 自身的保存位置。因此，我们规定 `this` 的值总是保存在数组的第 1 个元素中。代码清单 12.3 中 `OptMethod` 类的 `makeEnv` 方法的作用是准备一个用于执行新方法的环境，它将在环境数组的第 1 个元素中保存 `this` 的值。

```
e.put(0, 0, self);
```

上面这条语句与构造函数的初始化处理效果相同。

此外，`BinaryExpr` 类的 `computeAssign` 方法也需要覆盖。上一章代码清单 11.4 的 `BinaryEx2` 修改器定义的 `computeAssign` 方法已经对它进行了修改，现在我们需要再次覆盖之前的定义。本章新定义的 `computeAssign` 方法仅会在赋值表达式左侧是字段时执行，否则仍将调用之前的版本。

代码清单 12.7 是经过以上这些修改后得到的性能提升的解释器主体程序。代码清单 12.8 是该解释器的启动程序。

代码清单 12.6 ObjOptimizer.java

```
package chap12;
import java.util.ArrayList;
import java.util.List;
import static javassist.gluonj.GluonJ.revise;
import javassist.gluonj.*;
import stone.*;
import stone.ast.*;
import chap6.Environment;
import chap6.BasicEvaluator;
import chap6.BasicEvaluator.ASTreeEx;
import chap7.FuncEvaluator.PrimaryEx;
import chap11.ArrayEnv;
import chap11.EnvOptimizer;
import chap11.Symbols;
import chap11.EnvOptimizer.ASTreeOptEx;
import chap11.EnvOptimizer.EnvEx2;
import chap11.EnvOptimizer.ParamsEx;
import chap12.OptStoneObject.AccessException;

@Require(EnvOptimizer.class)
@Reviser public class ObjOptimizer {
    @Reviser public static class ClassStmntEx extends ClassStmnt {
        public ClassStmntEx(List<ASTree> c) { super(c); }
        public void lookup(Symbols syms) {}
        public Object eval(Environment env) {
            Symbols methodNames = new MemberSymbols(((EnvEx2)env).symbols(),
                                                    MemberSymbols.METHOD);
            Symbols fieldNames = new MemberSymbols(methodNames,
                                                    MemberSymbols.FIELD);
            OptClassInfo ci = new OptClassInfo(this, env, methodNames,
                                                fieldNames);

            ((EnvEx2)env).put(name(), ci);
            ArrayList<DefStmnt> methods = new ArrayList<DefStmnt>();
            if (ci.superClass() != null)
                ci.superClass().copyTo(fieldNames, methodNames, methods);
            Symbols newSyms = new SymbolThis(fieldNames);
            ((ClassBodyEx)body()).lookup(newSyms, methodNames, fieldNames,
                                         methods);

            ci.setMethods(methods);
            return name();
        }
    }

    @Reviser public static class ClassBodyEx extends ClassBody {
        public ClassBodyEx(List<ASTree> c) { super(c); }
        public Object eval(Environment env) {
            for (ASTree t: this)
                if (!(t instanceof DefStmnt))
                    ((ASTreeEx)t).eval(env);
            return null;
        }
    }
}
```

```

    }
    public void lookup(Symbols syms, Symbols methodNames,
                      Symbols fieldNames, ArrayList<DefStmtnt> methods)
    {
        for (ASTree t: this) {
            if (t instanceof DefStmtnt) {
                DefStmtnt def = (DefStmtnt)t;
                int oldSize = methodNames.size();
                int i = methodNames.putNew(def.name());
                if (i >= oldSize)
                    methods.add(def);
                else
                    methods.set(i, def);
                ((DefStmtntEx2)def).lookupAsMethod(fieldNames);
            }
            else
                ((ASTreeOptEx)t).lookup(syms);
        }
    }
}

@Reviser public static class DefStmtntEx2 extends EnvOptimizer.DefStmtntEx {
    public DefStmtntEx2(List<ASTree> c) { super(c); }
    public int locals() { return size; }
    public void lookupAsMethod(Symbols syms) {
        Symbols newSyms = new Symbols(syms);
        newSyms.putNew(SymbolThis.NAME);
        ((ParamsEx)parameters()).lookup(newSyms);
        ((ASTreeOptEx)revise(body())).lookup(newSyms);
        size = newSyms.size();
    }
}

@Reviser public static class DotEx extends Dot {
    public DotEx(List<ASTree> c) { super(c); }
    public Object eval(Environment env, Object value) {
        String member = name();
        if (value instanceof OptClassInfo) {
            if ("new".equals(member)) {
                OptClassInfo ci = (OptClassInfo)value;
                ArrayEnv newEnv = new ArrayEnv(1, ci.environment());
                OptStoneObject so = new OptStoneObject(ci, ci.size());
                newEnv.put(0, 0, so);
                initObject(ci, so, newEnv);
                return so;
            }
        }
        else if (value instanceof OptStoneObject) {
            try {
                return ((OptStoneObject)value).read(member);
            } catch (AccessException e) {}
        }
        throw new StoneException("bad member access: " + member, this);
    }
    protected void initObject(OptClassInfo ci, OptStoneObject obj,
                             Environment env)

```

```

    {
        if (ci.superClass() != null)
            initObject(ci.superClass(), obj, env);
        ((ClassBodyEx)ci.body()).eval(env);
    }
}

@Reviser public static class NameEx2 extends EnvOptimizer.NameEx {
    public NameEx2(Token t) { super(t); }
    @Override public Object eval(Environment env) {
        if (index == UNKNOWN)
            return env.get(name());
        else if (nest == MemberSymbols.FIELD)
            return getThis(env).read(index);
        else if (nest == MemberSymbols.METHOD)
            return getThis(env).method(index);
        else
            return ((EnvEx2)env).get(nest, index);
    }
    @Override public void evalForAssign(Environment env, Object value) {
        if (index == UNKNOWN)
            env.put(name(), value);
        else if (nest == MemberSymbols.FIELD)
            getThis(env).write(index, value);
        else if (nest == MemberSymbols.METHOD)
            throw new StoneException("cannot update a method: " + name(),
                                     this);
        else
            ((EnvEx2)env).put(nest, index, value);
    }
    protected OptStoneObject getThis(Environment env) {
        return (OptStoneObject)((EnvEx2)env).get(0, 0);
    }
}

@Reviser public static class AssignEx extends BasicEvaluator.BinaryEx {
    public AssignEx(List<ASTree> c) { super(c); }
    @Override
    protected Object computeAssign(Environment env, Object rvalue) {
        ASTree le = left();
        if (le instanceof PrimaryExpr) {
            PrimaryEx p = (PrimaryEx)le;
            if (p.hasPostfix(0) && p.postfix(0) instanceof Dot) {
                Object t = ((PrimaryEx)le).evalSubExpr(env, 1);
                if (t instanceof OptStoneObject)
                    return setField((OptStoneObject)t, (Dot)p.postfix(0),
                                   rvalue);
            }
        }
        return super.computeAssign(env, rvalue);
    }
    protected Object setField(OptStoneObject obj, Dot expr, Object rvalue) {
        String name = expr.name();
        try {
            obj.write(name, rvalue);
            return rvalue;
        }
    }
}

```

```

        } catch (AccessException e) {
            throw new StoneException("bad member access: " + name, this);
        }
    }
}

```

代码清单 12.7 ObjOptInterpreter.java

```

package chap12;
import stone.ClassParser;
import stone.ParseErrorException;
import chap11.EnvOptInterpreter;
import chap11.ResizableArrayEnv;
import chap8.Natives;

public class ObjOptInterpreter extends EnvOptInterpreter {
    public static void main(String[] args) throws ParseErrorException {
        run(new ClassParser(),
            new Natives().environment(new ResizableArrayEnv()));
    }
}

```

代码清单 12.8 ObjOptRunner.java

```

package chap12;
import javassist.gluonj.util.Loader;
import chap8.NativeEvaluator;

public class ObjOptRunner {
    public static void main(String[] args) throws Throwable {
        Loader.run(ObjOptInterpreter.class, args, ObjOptimizer.class,
            NativeEvaluator.class);
    }
}

```

12.5 内联缓存

之前已经提到，如果要对非 `this` 对象进行方法调用或字段引用，上一章及本章介绍的 `lookup` 方法将不再有效，执行速度无法得到提升。这样一来，语言处理器在调用函数或引用字段时，就不得不通过名称来查找环境，从哈希表中获取对应的值，使执行速度大幅下降。

为了缓解这一问题，我们将使用一种名为内联缓存（`inline cache`）的方法。首先，我们假设程序需要引用某个对象的字段。正如之前所讲，只有在实际执行后语言处理器才能确定该对象的类型。不过，根据经验可知，同一位置出现的对象通常是同一种类型。即使是采用面向对象思想写成的程序也是如此。我们能够利用这一规律优化处理器的性能。语言处理器可以在执行程序的同时查找字段的保存位置，并将该结果与对象所属的类型结对保存。之后，如果再次执行同一段程序，语言处理器将首先判断对象的类型，如果与之前相同，则直接使用上次的查找结果，减少

了因查找保存位置而造成的性能下降。

A 既然叫内联缓存，缓存值究竟以内联的形式保存到哪里了呢？

F 在抽象语法树里呀。

H 例如，在引用字段时，信息将由与该字段引用表达式对应的抽象语法树缓存，具体来说，是保存在语法树节点对象的相应字段中。

C 通常，除了抽象语法树的节点，中间代码和二进制代码也能用于缓存。这里的核心思想在于，语言处理器将使用分别为程序的代码行、表达式及指令准备的缓存空间。

代码清单 12.9 中的 `InlineCache` 修改器实现了内联缓存机制。它将覆盖 `Dot` 类的 `eval` 方法与 `BinaryExpr` 类的 `setField` 方法。此外，它还会为每个类添加用于实现缓存功能的 `classInfo` 字段与 `index` 字段（其中，`Dot` 类还会额外新增一个 `isField` 字段）。

`Dot` 类的 `eval` 方法用于读取字段的值，或充当方法调用表达式。`setField` 方法用于处理赋值表达式左侧是某个对象的字段时的情况。这里的 `setField` 方法正是代码清单 12.6 中由 `AssignEx` 修改器定义的 `setField` 方法。这两个方法都会将由修改器添加的 `classInfo` 字段的值，与当前正被执行方法调用或字段引用的对象类型进行比较。如果相同，则再次利用上一次保存的值。

经过 `InlineCache` 修改器的修改后，解释器将支持内联缓存功能。解释器本身的程序与代码清单 12.7 中的相同，不过，应用了修改器的解释器需要通过代码清单 12.10 中的启动程序运行。它与代码清单 12.10 及更早的代码清单 12.8 中的启动程序大同小异，唯一的区别在于启动程序中的 `run` 方法将接收不同类型的修改器。

代码清单 12.9 `InlineCache.java`

```
package chap12;
import java.util.List;
import stone.StoneException;
import stone.ast.ASTree;
import stone.ast.Dot;
import chap6.Environment;
import javassist.gluonj.*;

@Require(ObjOptimizer.class)
@Reviser public class InlineCache {
    @Reviser public static class DotEx2 extends ObjOptimizer.DotEx {
        protected OptClassInfo classInfo = null;
        protected boolean isField;
        protected int index;
        public DotEx2(List<ASTree> c) { super(c); }
        @Override public Object eval(Environment env, Object value) {
            if (value instanceof OptStoneObject) {
                OptStoneObject target = (OptStoneObject)value;
                if (target.classInfo() != classInfo)
                    updateCache(target);
            }
        }
    }
}
```

```

        if (isField)
            return target.read(index);
        else
            return target.method(index);
    }
    else
        return super.eval(env, value);
}
protected void updateCache(ObjStoneObject target) {
    String member = name();
    classInfo = target.classInfo();
    Integer i = classInfo.fieldIndex(member);
    if (i != null) {
        isField = true;
        index = i;
        return;
    }
    i = classInfo.methodIndex(member);
    if (i != null) {
        isField = false;
        index = i;
        return;
    }
    throw new StoneException("bad member access: " + member, this);
}
}
@Reviser public static class AssignEx2 extends ObjOptimizer.AssignEx {
    protected OptClassInfo classInfo = null;
    protected int index;
    public AssignEx2(List<ASTree> c) { super(c); }
    @Override protected Object setField(ObjStoneObject obj, Dot expr,
                                         Object rvalue)
    {
        if (obj.classInfo() != classInfo) {
            String member = expr.name();
            classInfo = obj.classInfo();
            Integer i = classInfo.fieldIndex(member);
            if (i == null)
                throw new StoneException("bad member access: " + member,
                                          this);

            index = i;
        }
        obj.write(index, rvalue);
        return rvalue;
    }
}
}

```

代码清单 12.10 InlineRunner.java

```

package chap12;
import javassist.gluonj.util.Loader;
import chap8.NativeEvaluator;

```

```
public class InlineRunner {
    public static void main(String[] args) throws Throwable {
        Loader.run(ObjOptInterpreter.class, args, InlineCache.class,
                   NativeEvaluator.class);
    }
}
```

A 速度变快了吗？

F 测一下代码清单 12.11 的执行时间吧。先测试添加 `this.` 的程序，再测试去除 `this.` 的版本，就能得出结论了。

H 也就是说，分别测试启用与没有启用内联缓存的情况对吧？

F 没错。测试的结果是，第 9 章的实现最终耗时约 6.8 秒。如果不支持内联缓存，本章的优化版本需要执行约 5.7 秒，如果支持内联缓存，则大概只要 5.0 秒。

A 哇，即使没有内联缓存，性能也提高了 20%，有内联缓存之后更是提高了 35% 呢。

C 要我说呀，这些数字没多大的意义。基准测试程序的写法没有一个定数不是嘛？如果内联缓存能发挥作用的机会较少，结果就会比较糟糕了。

A 也是，只有大量运行实际的程序才能知道具体结果是怎样的。没错吧 F 君？

代码清单 12.11 测试斐波那契数的计算时间（面向对象版本）

```
class Fib {
    fib0 = 0
    fib1 = 1
    def fib (n) {
        if n == 0 {
            fib0
        } else {
            if n == 1 {
                this.fib1
            } else {
                fib(n - 1) + this.fib(n - 2)
            }
        }
    }
}

t = currentTime()
f = Fib.new
f.fib 33
print currentTime() - t + " msec"
```

第2部分

性能优化篇

第
13
天

设计中间代码解释器

设计中间代码解释器

之前的 Stone 语言处理器都会一边遍历抽象语法树的节点，一边执行程序。如果对此不太理解，请读者回忆一下 eval 方法的执行方式。

然而，这种对抽象语法树节点的遍历操作是一种很大的性能负担。在之前的章节中，我们采用了事先计算能够计算的值的方针来优化性能。根据该方针，我们应当对抽象语法树的遍历操作做相应的修改，让语言处理器能够预先计算可以计算的部分。由于抽象语法树的形状不会在程序执行过程中发生改变，因此这种思路应该没有什么问题。

为了实现这种思路，我们采用了名为中间代码解释器的方式。这里的中间代码也能称为二进制代码，人们有时也会用虚拟机来指代中间代码解释器。这些名称的含义基本相同。本章将尝试通过这种方式提升 Stone 语言处理器的性能。

在使用中间代码解释器时，我们要先将抽象语法树转换为中间代码。简单来说，中间代码是一种虚拟的机器语言，因此，中间代码的转换方法，其实与编译器将抽象语法树转换为真正的机器语言时采用的方法大体相同。也就是说，本章将会讲解如何为 Stone 语言设计编译器。

13.1 中间代码与机器语言

顾名思义，抽象语法树具有树形结构。尽管我们前面用了遍历这样一个看似平常的词，实际的处理却并不简单。语言处理器需要在节点之间往返操作，读者仅凭直觉也能想象这将是一件费时的工作。因此，如果语言处理器能够事先计算遍历顺序，并以此重新排列节点，执行开销就可能有所降低。这列重新排列的节点将作为中间代码保存，语言处理器在执行程序时将不再使用抽象语法树，而改用这一中间代码。这就是中间代码解释器的基本原理。

通常，语言处理器不会直接将重新排列的抽象语法树节点作为中间代码使用。如果直接保存抽象语法树的节点，多余的无用信息是一种空间上的浪费，因此，我们需要设计一种虚拟的机器语言，并将各个节点转换为与该节点运算逻辑对应的机器语言。大多数语言处理器使用的中间语言都是这种转换后的代码（图 13.1）。

我们把根据中间代码执行实际运算的程序称为中间代码解释器或虚拟机（virtual machine）。中间代码既可以保存在内存中，也能暂时通过文件保存，在实际执行时再次读取至内存。本章将采用前一种实现方式。Java 语言采用了后者，并将中间代码称为 Java 二进制代码。此外，第二种方式中，生成并保存中间代码的过程被称为编译。

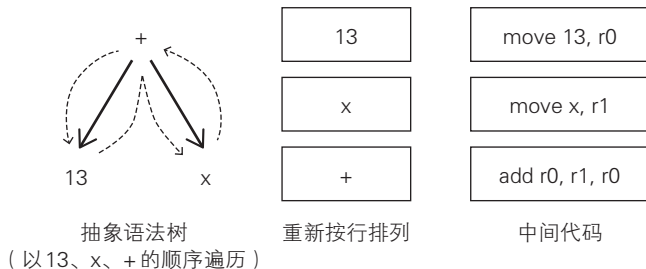


图 13.1 抽象语法树与中间代码

用于表示中间代码的虚拟机器语言不一定要与实际的机器语言相近，通常，我们以能由中间代码解释器高速执行为目标设计虚拟机器语言（否则中间代码转换将没有意义）。不过，由于本章还会讲解 Stone 语言编译器设计的基本概念，因此最终采用了与实际的机器语言类似的虚拟机器语言。编译器同样会执行词法分析与语法分析，并创建抽象语法树。编译器与解释器唯一的区别在于，之后它并不是通过 eval 方法执行程序，而是将抽象语法树转换为机器语言，并以文件形式保存。

H 本章还会讲解 Stone 语言编译器呀？

C 我的确打算介绍一些编译器的设计，不过这里采用的虚拟机器语言和 IA32 之类实际的机器语言相比实在是非常简单，只是基本中的基本。而且编译器设计过程中最重要的代码优化问题也没有涉及。

H 实际的编译器并不仅仅是简单地将抽象语法树的节点重新排列并转换为机器语言就行了，它们还需要通过各种手段尽可能提高机器语言的性能。

C 嗯，这就是代码优化。

A 嗯……IA32 是什么？

F 就是 32 位的英特尔架构呀，你不知道吗？

A 啊，怎么可能，我当然知道啦。

S IA32 太过复杂，没有必要，如果使用与 IA32 类似的虚拟机器语言，反而不容易看清中间代码的本质。因此没必要采用类似的设计。

13.2 Stone 虚拟机

本章设计的中间代码解释器称为 Stone 虚拟机。它处理的中间语言称为虚拟机器语言。

Stone 虚拟机由若干个通用寄存器与内存组成。内存分为四个区域，分别是栈 (stack) 区、堆 (heap) 区、程序代码区与文字常量区。虚拟机器语言保存于程序代码区，字符串字面量保存于文字常量区。

F 这里使用了通用寄存器，说明 Stone 虚拟机是一种寄存器机器，而不是像 Java 虚拟机那样的堆栈结构机器，对吗？

C 嗯，没错，毕竟如今的处理器多是些提供了大量通用寄存器的寄存器机器。

从实际的机器语言的角度来看，计算机能大致分为两种设备，即内存与寄存器访问器，内存是一个巨大的 `byte` 数组，寄存器访问器则用于对若干个通用寄存器进行读写操作。这里暂不考虑其他的输入输出设备。`byte` 类型用于表示 8 位二进制整数，相当于 Java 语言中的 1 字节。

内存虽然是一个 `byte` 数组，但它也能处理其他类型的值。实际的程序通常需要处理 32 位整数、浮点小数或字符串等各种类型的值，而这些值都将通过 8 位整数值的组合表现。例如，32 位整数将以 8 位为一组分解成 4 组，并分别保存至内存，即 `byte` 数组的元素中。这种保存方式称为编码（`encode`）。因此，编译器在从内存中读取这些值时需要进行相应的解码（`decode`）处理。

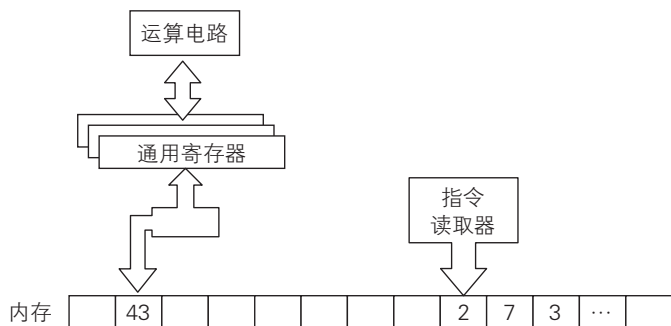


图 13.2 机器语言视点下的计算机

处理器将从指定位置开始依次读取用于表示内存的数组元素，并根据元素的值执行相应的操作（图 13.2）。例如，如果数组元素的值为 1，处理器将对寄存器的值求和；如果为 2，则从内存连续读取 4 个元素，将它们解码为一个 32 位整数后，再保存至寄存器中。这些用于表示操作类型的数字称为机器语言指令。为了实现 `if` 语句等条件判断逻辑，机器语言指令还支持根据不同的条件读取相应地址的指令。

对于有些机器语言指令，仅凭 1 个 `byte` 数组元素（即 1 字节）无法完全表现所要实行的操作内容。例如，在执行四则运算时，除了运算类型，机器语言指令还必须标明需要进行计算的寄存器。因此，大部分机器语言指令将通过多个连续元素值的组合（即若干字节）来表现需要执行的操作。这也是一种编码处理。

F 在图 13.2 中，指令读取器会在执行完 1 条指令后，继续读取执行下一条相邻指令对吧？

C 嗯，在遇到停止指令前这一操作将会不断重复。

通常，从机器语言的角度来看，实际的内存是一个 `byte` 类型的数组。为了简化设计，在 Stone 虚拟机中仅有程序代码区由 `byte` 数组实现，栈区和堆区都是 `Object` 类型的数组，文字常量区则是 `String` 类型的数组。通用寄存器的值也以 `Object` 类型表示。因此，虚拟机在向内存保存各种类型的值时，不必对值进行编码或解码，虚拟机器语言的程序实现得到了简化。

F 这种设计让人觉得这只是一个虚拟机而已，实际的计算机不可能这样实现的吧？

C 当然，实际处理器的寄存器只能保存 32 位或 64 位的比特序列。不过如果要遵循这种设计，虚拟机的实现将变得相当复杂。

Stone 虚拟机除了通用寄存器外还提供了 `pc`、`fp`、`sp` 和 `ret` 这四个寄存器。它们都能保存 `int` 类型的整数值。`pc` 是程序计数器。若 `pc` 的值为 i ，虚拟机将执行程序代码区从前端数起的第 i 个元素中保存的机器语言指令。`fp` 与 `sp` 分别是帧指针（`frame pointer`）与栈指针（`stack pointer`），它们都用于管理栈区。`ret` 用于函数的调用操作。

表 13.1 是虚拟机器语言指令一览。请读者注意，算术运算只能在寄存器之间进行。大部分指令的长度都大于 1 字节，需要由多个字节表示。指令前面的 `iconst` 或 `bconst` 等指令类型（称为操作码）在实际中将由 8 位整数表示。代码清单 13.1 标明了指令的编号以及寄存器编号（称为操作数）。操作数由特定的 8 位整数表示。例如，表中的 `int32` 表示操作码之后连续的 32 位数据将以 8 位为单位分解，组成 4 个字节的数据。其他诸如 `int16` 等同理。

下面的指令表示将整数 67 保存至名为 `r2` 的第二个寄存器中。

```
iconst 67 r2
```

该指令将以 6 个 8 位整数表示，依次保存至内存中程序代码区的相邻元素中。

```
1 0 0 0 67 -3
```

第 1 个数字 1 表示 `iconst`。最后的 -3 表示这是第 2 个寄存器。Stone 虚拟机的第 i 个寄存器将以编号 $-i-1$ 表示。中间 4 个数字用于表示 32 位的整数 67。整数 67 从高位起以 8 位为一组分成四组，每一组都能视为一个 8 位整数（图 13.3）。它们将保存于程序代码区中相邻的四个 `byte` 类型数组元素中（四个 8 位合计 32 位），以二进制数形式表示一个 32 位整数。不难理解，保存的四个元素中有三个值为 0，还有一个值为 67。

代码清单 13.1、代码清单 13.2 与代码清单 13.3 是 Stone 虚拟机的程序实现。其中，代码清单 13.2 是表示堆区的对象接口。

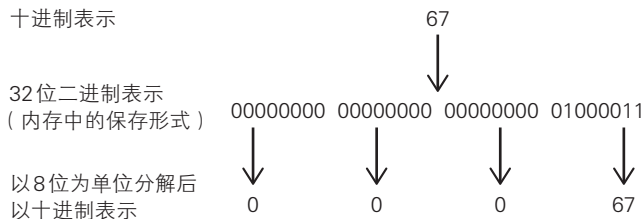


图 13.3 保存在内存中的 67

表 13.1 Stone 虚拟机的虚拟机器语言

<code>iconst int32 reg</code>	将整数值 <code>int32</code> 保存至 <code>reg</code>
<code>bconst int8 reg</code>	将整数值 <code>int8</code> 保存至 <code>reg</code>
<code>sconst int16 reg</code>	将字符常量区的第 <code>int16</code> 个字符串字面量保存至 <code>reg</code>
<code>move src dest</code>	在栈与寄存器，或寄存器之间进行值复制操作 (<code>src</code> 与 <code>dest</code> 可以是 <code>reg</code> 或 <code>int8</code>)
<code>gmove src dest</code>	在堆与寄存器之间进行值复制操作 (<code>src</code> 与 <code>dest</code> 可以是 <code>reg</code> 或 <code>int16</code>)
<code>ifzero reg int16</code>	如果 <code>reg</code> 的值为 0，则跳转至 <code>int16</code> 分支
<code>goto int16</code>	强制跳转至 <code>int16</code> 分支
<code>call reg int8</code>	调用函数 <code>reg</code> ，该函数将调用 <code>int8</code> 个参数 (同时， <code>call</code> 之后的指令地址将被保存至 <code>ret</code> 寄存器)
<code>return</code>	跳转至 <code>ret</code> 寄存器储存的分支地址
<code>save int8</code>	将寄存器的值转移至栈中，并更改寄存器 <code>fp</code> 与 <code>sp</code> 的值
<code>restore int8</code>	还原之前转移至栈中的寄存器值
<code>neg reg</code>	反转 <code>reg</code> 中保存的值的正负号
<code>add reg₁ reg₂</code>	计算 <code>reg₁+reg₂</code> 后保存至 <code>reg₁</code>
<code>sub reg₁ reg₂</code>	计算 <code>reg₁-reg₂</code> 后保存至 <code>reg₁</code>
<code>mul reg₁ reg₂</code>	计算 <code>reg₁×reg₂</code> 后保存至 <code>reg₁</code>
<code>div reg₁ reg₂</code>	计算 <code>reg₁÷reg₂</code> 后保存至 <code>reg₁</code>
<code>rem reg₁ reg₂</code>	计算 <code>reg₁÷reg₂</code> 的余数后将余数保存至 <code>reg₁</code>
<code>equal reg₁ reg₂</code>	如果 <code>reg₁ = reg₂</code> 则将 <code>reg₁</code> 赋值为 1，否则赋值为 0
<code>more reg₁ reg₂</code>	如果 <code>reg₁ > reg₂</code> 则将 <code>reg₁</code> 赋值为 1，否则赋值为 0
<code>less reg₁ reg₂</code>	如果 <code>reg₁ < reg₂</code> 则将 <code>reg₁</code> 赋值为 1，否则赋值为 0

※ 本表中，`int32` 表示 32 位整数，`int16` 表示 16 位整数，`int8` 表示 8 位非负整数，`reg` 表示 8 位寄存器编号。

代码清单 13.1 Opcode.java

```
package chap13;
import stone.StoneException;

public class Opcode {
```

```

    public static final byte ICONST = 1;    // load an integer
    public static final byte BCONST = 2;    // load an 8bit (1byte) integer
    public static final byte SCONST = 3;    // load a character string
    public static final byte MOVE = 4;      // move a value
    public static final byte GMOVE = 5;     // move a value (global variable)
    public static final byte IFZERO = 6;    // branch if false
    public static final byte GOTO = 7;      // always branch
    public static final byte CALL = 8;      // call a function
    public static final byte RETURN = 9;    // return
    public static final byte SAVE = 10;     // save all registers
    public static final byte RESTORE = 11;   // restore all registers
    public static final byte NEG = 12;      // arithmetic negation
    public static final byte ADD = 13;      // add
    public static final byte SUB = 14;      // subtract
    public static final byte MUL = 15;      // multiply
    public static final byte DIV = 16;      // divide
    public static final byte REM = 17;      // remainder
    public static final byte EQUAL = 18;    // equal
    public static final byte MORE = 19;     // more than
    public static final byte LESS = 20;     // less than

    public static byte encodeRegister(int reg) {
        if (reg > StoneVM.NUM_OF_REG)
            throw new StoneException("too many registers required");
        else
            return (byte)-(reg + 1);
    }
    public static int decodeRegister(byte operand) { return -1 - operand; }
    public static byte encodeOffset(int offset) {
        if (offset > Byte.MAX_VALUE)
            throw new StoneException("too big byte offset");
        else
            return (byte)offset;
    }
    public static short encodeShortOffset(int offset) {
        if (offset < Short.MIN_VALUE || Short.MAX_VALUE < offset)
            throw new StoneException("too big short offset");
        else
            return (short)offset;
    }
    public static int decodeOffset(byte operand) { return operand; }
    public static boolean isRegister(byte operand) { return operand < 0; }
    public static boolean isOffset(byte operand) { return operand >= 0; }
}

```

代码清单 13.2 HeapMemory.java

```

package chap13;

public interface HeapMemory {
    Object read(int index);
    void write(int index, Object v);
}

```

代码清单 13.3 StoneVM.java

```

package chap13;
import static chap13.Opcodes.*;
import chap8.NativeFunction;
import stone.StoneException;
import stone.ast.ASTree;
import stone.ast.ASTList;
import java.util.ArrayList;

public class StoneVM {
    protected byte[] code;
    protected Object[] stack;
    protected String[] strings;
    protected HeapMemory heap;

    public int pc, fp, sp, ret;
    protected Object[] registers;
    public final static int NUM_OF_REG = 6;
    public final static int SAVE_AREA_SIZE = NUM_OF_REG + 2;

    public final static int TRUE = 1;
    public final static int FALSE = 0;

    public StoneVM(int codeSize, int stackSize, int stringsSize, HeapMemory hm) {
        code = new byte[codeSize];
        stack = new Object[stackSize];
        strings = new String[stringsSize];
        registers = new Object[NUM_OF_REG];
        heap = hm;
    }

    public Object getReg(int i) { return registers[i]; }
    public void setReg(int i, Object value) { registers[i] = value; }
    public String[] strings() { return strings; }
    public byte[] code() { return code; }
    public Object[] stack() { return stack; }
    public HeapMemory heap() { return heap; }

    public void run(int entry) {
        pc = entry;
        fp = 0;
        sp = 0;
        ret = -1;
        while (pc >= 0)
            mainLoop();
    }

    protected void mainLoop() {
        switch (code[pc]) {
            case ICONST :
                registers[decodeRegister(code[pc + 5])] = readInt(code, pc + 1);
                pc += 6;
                break;
            case BCONST :
                registers[decodeRegister(code[pc + 2])] = (int)code[pc + 1];
                pc += 3;
                break;
        }
    }
}

```

```

        case SCONST :
            registers[decodeRegister(code[pc + 3])]
                = strings[readShort(code, pc + 1)];
            pc += 4;
            break;
        case MOVE :
            moveValue();
            break;
        case GMOVE :
            moveHeapValue();
            break;
        case IFZERO : {
            Object value = registers[decodeRegister(code[pc + 1])];
            if (value instanceof Integer && ((Integer)value).intValue() == 0)
                pc += readShort(code, pc + 2);
            else
                pc += 4;
            break;
        }
        case GOTO :
            pc += readShort(code, pc + 1);
            break;
        case CALL :
            callFunction();
            break;
        case RETURN :
            pc = ret;
            break;
        case SAVE :
            saveRegisters();
            break;
        case RESTORE :
            restoreRegisters();
            break;
        case NEG : {
            int reg = decodeRegister(code[pc + 1]);
            Object v = registers[reg];
            if (v instanceof Integer)
                registers[reg] = -((Integer)v).intValue();
            else
                throw new StoneException("bad operand value");
            pc += 2;
            break;
        }
        default :
            if (code[pc] > LESS)
                throw new StoneException("bad instruction");
            else
                computeNumber();
            break;
    }
}

protected void moveValue() {
    byte src = code[pc + 1];
    byte dest = code[pc + 2];

```

```

Object value;
if (isRegister(src))
    value = registers[decodeRegister(src)];
else
    value = stack[fp + decodeOffset(src)];
if (isRegister(dest))
    registers[decodeRegister(dest)] = value;
else
    stack[fp + decodeOffset(dest)] = value;
pc += 3;
}
protected void moveHeapValue() {
    byte rand = code[pc + 1];
    if (isRegister(rand)) {
        int dest = readShort(code, pc + 2);
        heap.write(dest, registers[decodeRegister(rand)]);
    }
    else {
        int src = readShort(code, pc + 1);
        registers[decodeRegister(code[pc + 3])] = heap.read(src);
    }
    pc += 4;
}
protected void callFunction() {
    Object value = registers[decodeRegister(code[pc + 1])];
    int numOfArgs = code[pc + 2];
    if (value instanceof VmFunction
        && ((VmFunction)value).parameters().size() == numOfArgs) {
        ret = pc + 3;
        pc = ((VmFunction)value).entry();
    }
    else if (value instanceof NativeFunction
        && ((NativeFunction)value).numOfParameters() == numOfArgs) {
        Object[] args = new Object[numOfArgs];
        for (int i = 0; i < numOfArgs; i++)
            args[i] = stack[sp + i];
        stack[sp] = ((NativeFunction)value).invoke(args,
            new ASTList(new ArrayList<ASTree>()));
        pc += 3;
    }
    else
        throw new StoneException("bad function call");
}
protected void saveRegisters() {
    int size = decodeOffset(code[pc + 1]);
    int dest = size + sp;
    for (int i = 0; i < NUM_OF_REG; i++)
        stack[dest++] = registers[i];
    stack[dest++] = fp;
    fp = sp;
    sp += size + SAVE_AREA_SIZE;
    stack[dest++] = ret;
    pc += 2;
}
protected void restoreRegisters() {

```

```

    int dest = decodeOffset(code[pc + 1]) + fp;
    for (int i = 0; i < NUM_OF_REG; i++)
        registers[i] = stack[dest++];
    sp = fp;
    fp = ((Integer)stack[dest++]).intValue();
    ret = ((Integer)stack[dest++]).intValue();
    pc += 2;
}

protected void computeNumber() {
    int left = decodeRegister(code[pc + 1]);
    int right = decodeRegister(code[pc + 2]);
    Object v1 = registers[left];
    Object v2 = registers[right];
    boolean areNumbers = v1 instanceof Integer && v2 instanceof Integer;
    if (code[pc] == ADD && !areNumbers)
        registers[left] = String.valueOf(v1) + String.valueOf(v2);
    else if (code[pc] == EQUAL && !areNumbers) {
        if (v1 == null)
            registers[left] = v2 == null ? TRUE : FALSE;
        else
            registers[left] = v1.equals(v2) ? TRUE : FALSE;
    }
    else {
        if (!areNumbers)
            throw new StoneException("bad operand value");
        int i1 = ((Integer)v1).intValue();
        int i2 = ((Integer)v2).intValue();
        int i3;
        switch (code[pc]) {
            case ADD :
                i3 = i1 + i2;
                break;
            case SUB:
                i3 = i1 - i2;
                break;
            case MUL:
                i3 = i1 * i2;
                break;
            case DIV:
                i3 = i1 / i2;
                break;
            case REM:
                i3 = i1 % i2;
                break;
            case EQUAL:
                i3 = i1 == i2 ? TRUE : FALSE;
                break;
            case MORE:
                i3 = i1 > i2 ? TRUE : FALSE;
                break;
            case LESS:
                i3 = i1 < i2 ? TRUE : FALSE;
                break;
            default:
                throw new StoneException("never reach here");
        }
    }
}

```

```

    }
    registers[left] = i3;
  }
  pc += 3;
}

public static int readInt(byte[] array, int index) {
    return (array[index] << 24) | ((array[index + 1] & 0xff) << 16)
        | ((array[index + 2] & 0xff) << 8) | (array[index + 3] & 0xff);
}
public static int readShort(byte[] array, int index) {
    return (array[index] << 8) | (array[index + 1] & 0xff);
}
}

```

13.3 通过栈实现环境

顾名思义，Stone 虚拟机是一种虚拟的计算机。它虽然能够处理 String 对象，但无法直接操作用于表示环境的 Environment 对象。这是一种有意为之的设计。本章将根据实际处理器的执行方式，通过内存栈区及堆区的形式来实现环境。此外，由于内存的本质是数组，因此本章也会采用第 11 章介绍的方法，事先确定变量值的保存位置，以编号而非名称查找环境。这样一来，环境就能够通过数组实现。

首先，我们通过堆区来实现用于记录全局变量的环境。全局变量只需使用一个环境，因此，我们将直接使用整个堆区。Stone 虚拟机使用的堆区实体，是一个由代码清单 13.2 中的 HeapMemory 接口实现的对象。该接口的 read 与 write 方法能够以数组的形式操作对象。代码清单 13.4 中的 StoneVMEnv 类的对象用于表示堆区。该类继承了第 11 章代码清单 11.2 中的 ResizableArrayEnv 类，并实现了 HeapMemory 接口（图 13.4）。

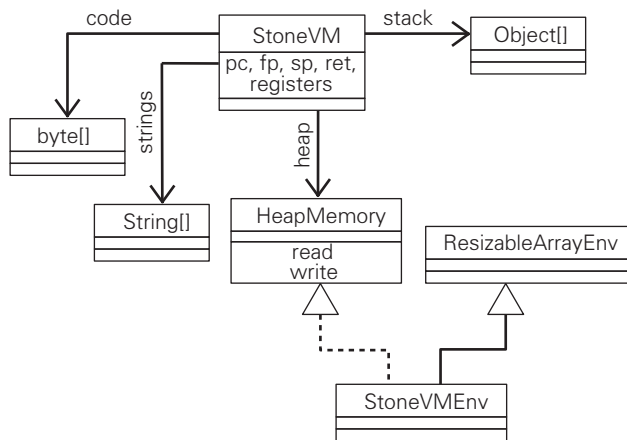


图 13.4 Stone 虚拟机与内存

我们之所以将 `StoneVMEnv` 类设计成 `ResizableArrayEnv` 的子类，而不是其他更简单的类，是因为我们希望能够将该类的对象作为 `Environment` 对象使用。之后也会讲到，虚拟机语言转换仅涉及函数的主体部分，最外层代码中的语句依然会像之前那样通过调用 `eval` 方法执行。因此，用于记录全局变量的环境必须也能以已有的 `Environment` 对象实现。虽说我们也能先将最外层代码中的语句临时转换为虚拟机语言后再去执行，但这种做法可能会转换一些不被执行的语句而造成时间的浪费，运行性能将无法保证。

与之相对地，用于记录局部变量的环境将通过栈区实现。由于 `Stone` 虚拟机需要使用多个用于记录布局变量的环境，因此我们将划分栈区，为每一个环境提供必要的空间。在本章中，`Stone` 语言不支持闭包。如果要为闭包提供支持，基于栈区的环境实现将变得十分复杂。

F C 语言等一些程序设计语言不支持闭包也是出于同样的原因。

代码清单 13.4 StoneVMEnv.java

```
package chap13;
import chap11.ResizableArrayEnv;
public class StoneVMEnv extends ResizableArrayEnv implements HeapMemory {
    protected StoneVM svm;
    protected Code code;
    public StoneVMEnv(int codeSize, int stackSize, int stringsSize) {
        svm = new StoneVM(codeSize, stackSize, stringsSize, this);
        code = new Code(svm);
    }
    public StoneVM stoneVM() { return svm; }
    public Code code() { return code; }
    public Object read(int index) { return values[index]; }
    public void write(int index, Object v) { values[index] = v; }
}
```

为了有序管理从栈区中划分出的空间，明确它们与各个环境的对应关系，我们将采用如下的设计思路。用于记录局部变量的环境将在函数首次调用时创建。由于 `Stone` 虚拟机不支持闭包，因此在函数执行结束，程序返回函数调用位置后，就不再需要该环境。考虑到函数可以嵌套调用，不难想象，最后创建的那个环境总会第一个作废。理由很简单，因为最后调用的函数总会最先结束。利用这一性质，我们就能够通过名为栈的数据结构来管理环境与栈区的对应关系。

H 因为要通过栈来管理，所以我们把它称为栈区，是吗？

C 难道不是因为这些空间将作为栈来使用所以才这么命名的吗？

A 关于这个栈呢……

F 数据结构里的栈是一种元素先进后出的容器。

A 这我当然知道啦。

H 栈的意思是堆积叠放。

c 嗯，像图 13.5 那样堆叠保存数据的话，最后添加的数据就总会被首先取出，非常巧妙。栈区大概是由此得名的吧。

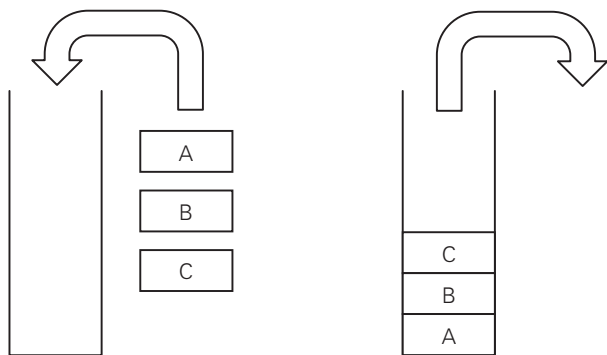


图 13.5 保存于栈中的数据需从上方依次取出，正好与存入的顺序相反

在执行某一函数 f 时， f 只会使用栈区的一部分作为记录局部变量的环境。这部分栈区的起始与末尾地址分别由寄存器 fp 与 sp 标识。例如，如果该函数使用了数组中的第 i 个至第 $j-1$ 个元素， fp 与 sp 的值将分别为 i 与 j (图 13.6)。

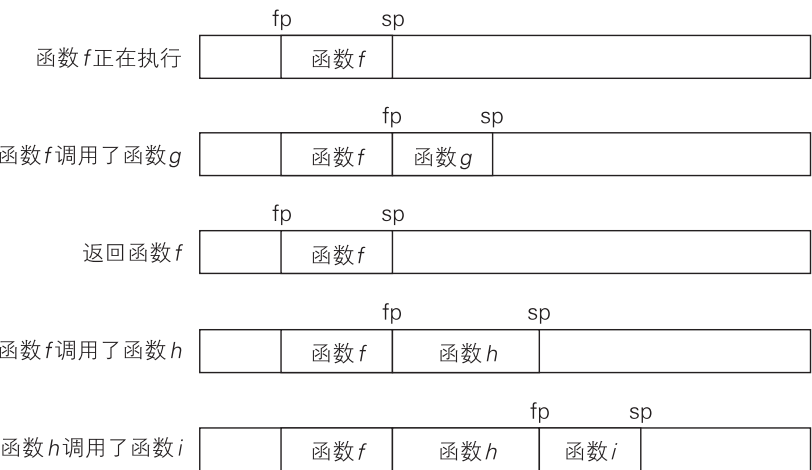


图 13.6 通过寄存器 fp 与 sp 对栈进行管理

c 回想一下，除了通用寄存器外，我们还为 Stone 虚拟机设计了 fp 与 sp 这两个特殊的寄存器。

如果函数 f 调用了另一个函数 g ，寄存器 sp 的值将被复制给寄存器 fp ，同时， sp 的值将根据新调用的函数 g 的需要增加。也就是说， sp 与 fp 的值将做相应的调整，使新调用的函数 g 能够紧接着原函数 f 继续使用栈区。

在新调用的函数 g 结束执行后，程序将返回原先的函数 f ，并还原寄存器 sp 与 fp 的值。于是程序将能重新使用之前的环境。

此时，函数 g 已完成调用，Stone 虚拟机不需要再使用与它对应的环境，函数 g 使用过的栈区空间能够被其他环境再次使用。例如，如果函数 f 之后又调用了另一个函数 h ，新调用的函数 h 将与函数 g 一样，紧接着函数 f 划分栈区，作为与自己对应的环境。函数 h 与函数 g 使用的栈区会有些重叠，不过此时函数 g 已经结束执行，与之对应的环境也不再需要，因此不会发生任何问题。

C 从机器语言的角度来看 Java 语言中的对象，会发现它也是由内存这种巨大的 byte 数组的一部分实现的。

H 虽然名为对象，不过它的具体实现还是一种用于记录字段值的数据结构而已。其实这就是 C 语言的结构体。

C 虚拟机必须时刻管理对象与它们使用的 byte 数组片段之间的对应关系，但要自己设计这样的管理程序可是一件非常不容易的事情。这种程序就是所谓的垃圾回收器。

F 因为最后创建的对象将首先作废不容易实现嘛。

C 内存管理是一个非常复杂的话题，可以另写一本书来讨论。不过，与局部变量的环境相关的内存管理相对简单些。

H 这多亏了“最后创建的环境将首先作废”这样一条性质对吧？

C 没错。如果要支持闭包，这个性质就将不再适用，程序会变得非常复杂。

这种通过栈来实现环境的方法，能将各个环境转化为由寄存器 fp 与 sp 标识的栈区区间。这类区间称为栈帧。函数 f 使用的栈帧称为函数 f 的栈帧，函数 g 使用的栈帧称为函数 g 的栈帧，以此类推。本章会以与第 11 章（第 11 天）类似的方法，将各变量的值保存至栈帧。寄存器 fp 指向的元素是栈帧的前端，Stone 虚拟机将事先确定各个变量的值应当保存在数组从该元素起数起的第几个元素中。可以发现，这与第 11 章的实现稍有不同。由于这里不像第 11 章那样需要使用整个数组，只需要用到从寄存器 fp 所指位置开始的局部数组，因此产生了上述差异。也就是说，寄存器 fp 指向的元素是用于实现当前环境的数组前端。

H 老师，你的意思是说，虚拟机会事先确定变量的值应当保存在从栈帧前端数起的第几个元素是吗？

C 因为只有在实际执行程序之后，我们才能知道某个栈帧具体由栈区中的哪些元素组成，所以虚拟机能事先确定的就只有元素与 fp 指向的栈帧前端的相对位置。

13.4 寄存器的使用

为了将抽象语法树转换为虚拟机器语言，虚拟机需要像图 13.1 那样，一边遍历语法树，一边将与各节点对应的虚拟机器语言片段依次保存至内存的程序代码区中。各节点的虚拟机器语言

之间不存在上下文关系，由事先制定的规则转换。如果注重执行速度，虚拟机就必须通过各种方式对转换后的机器语言进行优化。不过，本章暂时不追求执行速度，只求能够正确运行，因此采用了以上方针。同时，我们对由节点转换得到的虚拟机器语言片段作如下规定。

规定：为了保存由上一条片段计算得到的中间结果，第 $0 \sim i$ 个寄存器将处于占用状态。虚拟机可任意使用第 $i+1$ 个及之后的寄存器来计算当前片段。计算结果将最终保存于第 $i+1$ 个寄存器中。

各个节点将根据该规定转换为虚拟机器语言，并依次排序。最终，整棵抽象语法树都将被转换为虚拟机器语言。例如，图 13.7 是由表达式 $(7 + x) * y$ 转换得到的虚拟机器语言。图的左侧是抽象语法树，右侧是与之对应的虚拟机器语言。

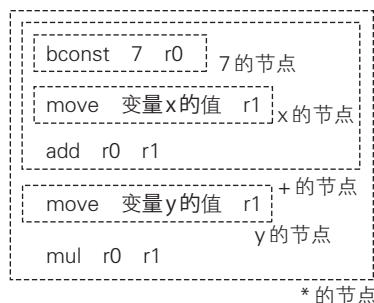
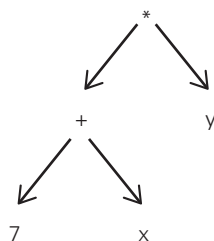


图 13.7 用于计算 $(7 + x) * y$ 的虚拟机器语言

本图中，与节点 7 对应的虚拟机器语言为 `bconst 7 r0`。它将把整数 7 保存至寄存器 `r0` 中。该寄存器现在尚未使用，与以上规定相符。

接下来，我们来看一下节点 `+` 该如何转换为虚拟机器语言。首先，由左侧的节点 7 转换得到的虚拟机器语言将被写入程序代码区。在这条虚拟机器语言之后，紧接着的是由右侧的节点 `x` 转换得到的虚拟机器语言。在执行左侧的虚拟机器语言后，该语句的计算结果必须作为表达式的中间结果，保存至寄存器 `r0`，因此，在将右侧转换为虚拟机器语言时，寄存器 `r0` 将处于占用状态。于是，根据以上规定，右侧的虚拟机器语言的计算结果将保存于寄存器 `r1` 而非 `r0` 中。

H 也就是说，由于第 0 个寄存器被占用，根据规定，虚拟机将只能使用第 1 个及之后的寄存器。

将与左右两侧节点对应的虚拟机器语言写入程序代码区后，语言处理器将接着写入加法指令 `add`。`add` 指令将把保存了中间结果（即左侧的计算结果）的寄存器 `r0` 与保存了上一个计算结果（即右侧的计算结果）的寄存器 `r1` 相加，并保存计算结果至寄存器 `r0` 中。以上就是与节点 `+` 对应的整条虚拟机器语言及相关的寄存器操作。计算结果保存在寄存器 `r0` 中，同样符合之前的规定。由于左侧的计算结果只需保存至加法运算开始为止，因此虚拟机可以放心地将加法的运算结果写入 `r0`。

符合规定的虚拟机器语言片段将随着语法树的遍历递归生成，并依次写入程序代码区，通过这种方式，语言处理器能够轻松地将抽象语法树转换为机器语言。对于上面的例子，语言处理器只要在节点 `+` 的虚拟机器语言后，继续将节点 `y` 的虚拟机器语言写入程序代码区，最后再写入乘法指令 `mul`，就完成了整棵抽象语法树的机器语言转换。整个转换过程的要点在于，用于保存中间计算结果的寄存器必须处于占用状态，不能为之后的虚拟机器语言所用。

C 堆栈结构机器也会用类似的方式将抽象语法树转换为机器语言，这两种方式的本质是相同的。

H 教材上一般只会介绍堆栈结构机器的转换方式。

A 就是那种将表达式以逆波兰表示法改写，然后再把计算结果存入栈中的做法？

F 呀！你记得可真清楚。

A 我可是在考试前通宵去背的，怎么能忘。

C 我说两者本质相同，是因为寄存器组能够被视作一种特殊的栈。可以将第 0 个寄存器理解为栈的底部。

F 如果第 0 个至第 i 个寄存器正在被使用，虚拟机就只能使用第 $i+1$ 个寄存器中的数据，这就好比是这条数据盖在了栈的顶部对吧？

C 没错。如果用逆波兰表示法改写，就相当于后序遍历 (post order) 了抽象语法树。

虽然语言处理器能够通过上述方法轻松地将抽象语法树转换为机器语言，但转换得到的机器语言性能较差。该版本的机器语言存在不少不足，寄存器的使用也有些冗繁。现假定我们要对表达式 $7+x+x$ 进行转换。此时，最佳的虚拟机器语言如下。

```
bconst 7 r0
move 变量 x 的值 r1
add r0 r1
add r0 r1
```

然而，根据以上介绍的方法，我们将得到一个更加冗长的转换结果。虚拟机器语言中将包含两条 move 指令，两次执行变量 x 的值至寄存器的复制操作。

C 机器语言的转换过程本身并不复杂，转换时间 (编译时间) 很短。

F 但要转换成运行性能优秀的机器语言可就不容易了。

S 嗯，谁都知道随意转换得到的机器语言绝对快不起来呀。

F Java 虚拟机也会在运行过程中多次重新编译频繁使用的代码，逐渐提升机器语言的性能。

A 老师，话说回来，如果只有 4 个寄存器，并且都处于占用状态，以上方法就行不通了呀！

H 是呢。根据规定，下一条虚拟机器语言片段的计算结果应该保存在第 5 个寄存器中才行。

A 现在我们就只有 4 个寄存器呀，用不了第 5 个。

F 下面这样的表达式在进行机器语言转换时就该出问题了吧？

```
a + (b * (c + (d * e)))
```

F 如果是堆栈结构机器，栈的容量是无限的，也就没有这个问题了。

S 我们能不能让寄存器也有无限多呢？

C 本章介绍的程序有一个问题，如果寄存器不足，机器语言转换就将以失败告终，同时程序将报错。

A 还真是偷工减料啊。

C 只要好好设计还是能避免这个问题的。我们可以为原来的程序添加一些局部变量，这样表达式的中间计算结果就能够通过这些局部变量记录，不用保存至寄存器。当然，局部变量只能用于临时保存一些寄存器无法容纳的中间结果。

13.5 引用变量的值

在虚拟机器语言中，`move` 或 `gmove` 指令能够将变量的值复制至寄存器中。`move` 指令的格式如下。

```
move 3 r1
```

它将把局部变量的值复制到第 1 个寄存器 `r1` 中。实际复制的值是栈区中的第 `fp+3` 个元素的值。它也是在当前环境中从前往后数第 3 个局部变量的值。

反之，下面的指令能够将寄存器 `r1` 中保存的值复制给同一个局部变量。

```
move r1 3
```

读者可以将其理解为局部变量的复制操作。

H 一般的机器语言是写成下面这样的吧？

```
move r1 3(fp)
```

C 因为我们在这里用的虚拟机只能指定寄存器的相对（间接）地址，所以可以简化写法。

另一个 `gmove` 方法则用于全局变量的复制，它可以将全局变量的值复制到寄存器中。

```
gmove 3 r1
```

这条机器语言能够将堆区从前往后数第 3 个元素的值复制至寄存器 `r1` 中。如果替换 3 与 `r1` 的位置，就能反过来将寄存器的值复制到堆区中。

由于在确定变量值的保存位置时，变量所属的作用域也会一并记录，因此我们能够根据作用域区分目标变量是一个局部变量还是全局变量。只要遵循该规律，语言处理器就能在进行虚拟机器语言转换时确定应当使用 `move` 还是 `gmove`。

13.6 if 语句与 while 语句

语言处理器能够通过分支指令将表示 `if` 语句与 `while` 语句的抽象语法树节点转换为虚拟机器语言。Stone 虚拟机提供了 `ifzero` 与 `goto` 这两种分支指令。

Stone 虚拟机将始终执行程序计数器（寄存器 `pc`）当前指向的机器语言指令。如果 `pc` 的值

为 50，虚拟机将执行程序代码区中从前往后数的第 50 个元素保存的指令。

H 老师，大部分指令的长度都在 2 字节以上哦。

A 程序代码区是一个 byte 类型的数组，因此要完整保存一条指令需要多个元素才行。

C 你们说的没错。准确地说，pc 指向的只是指令的第 1 个字节。

在执行完一条指令后，pc 值将自动增加与该指令长度相同的量，以指向下一条指令。因此，Stone 虚拟机能够依次执行各条指令。同时，pc 的值能够由分支指令更改。ifzero 能在指定寄存器的值为 0 时将某个整数值加至寄存器 pc，goto 则能强制增加 pc 的值。pc 值的增加量中不包括被执行的分支指令的长度。如果 pc 值的增加量为正，程序将向前跳转，如果为负则向后（反方向）跳转。我们将寄存器 pc 的这一用于实现分支跳转的增加量称为偏移量。

if 语句对应的抽象语法树节点的虚拟机器语言转换过程如图 13.8 所示。由 if 语句的条件表达式转换得到的虚拟机器语言将在执行后把结果保存至寄存器 r0 中。在 Stone 语言中，只有 0 为假（false），其他值都为真（true），因此 ifzero 指令能够在条件表达式的值为假时，跳转执行 else 代码块中的语句。

while 语句对应的抽象语法树节点的虚拟机器语言转换过程如图 13.9 所示。与 if 语句一样，该转换通过 ifzero 指令与 goto 指令实现。在本章程序中出现的 while 语句，都将以该图左侧的形式转换为虚拟机器语言。不过，如果能转换为该图右侧形式的虚拟机器语言，虚拟机实际执行的指令总数将会减少。左侧的虚拟机器语言在循环执行时将执行 ifzero 与 goto 这两条指令，右侧则只会执行 ifnonzero 这一条分支指令。ifnonzero 将在条件表达式的计算结果为真（true）时进行跳转，不过，Stone 虚拟机并不支持该指令。因此，本章将把 while 语句转换为左侧的形式。

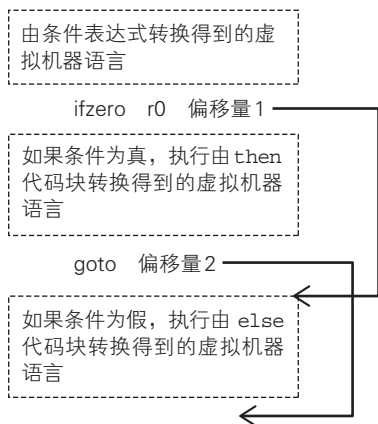
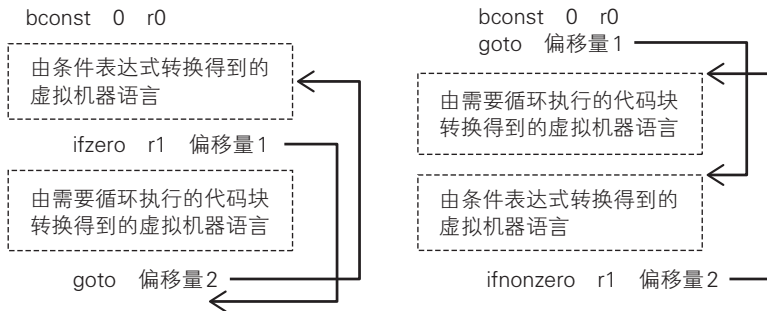


图 13.8 由 if 语句转换得到的虚拟机器语言



※规定“条件表达式的计算结果将保存于寄存器 r1，代码块的计算结果将保存于寄存器 r0”。

图 13.9 由 while 语句转换得到的虚拟机器语言（右侧的指令总数较少）

A 不要那么吝啬，让虚拟机也支持 ifnonzero 嘛！

F 要为虚拟机添加这个功能也不难，只要稍微修改下代码清单 13.3 就好了。

H 我说你们呀，老师不是说过希望让本章介绍的程序尽可能简短易读吗？可别忘了这点。

13.7 函数的定义与调用

在讨论如何将函数体转换为虚拟机器语言之前，我们首先要考虑函数调用所需的实参与返回值的传递方式，以及栈帧的切换机制等问题。在为实际的处理器设计机器语言时也要考虑这些规则，人们通常将它们称为调用惯例 (calling convention)。

函数的调用方需要将实参保存至栈区中。也就是说，该调用方需要直接把实参保存至被调用函数使用的栈帧内。函数的参数将始终保存在栈帧前端，且虚拟机能够事先确定函数的参数与局部变量在栈帧中的保存位置 (图 13.10)。如果参数不止一个，它们将依次存入栈帧前端。第 11 章介绍的实现也是如此。这样一来，函数的调用方能够轻松将参数保存至正确的位置。

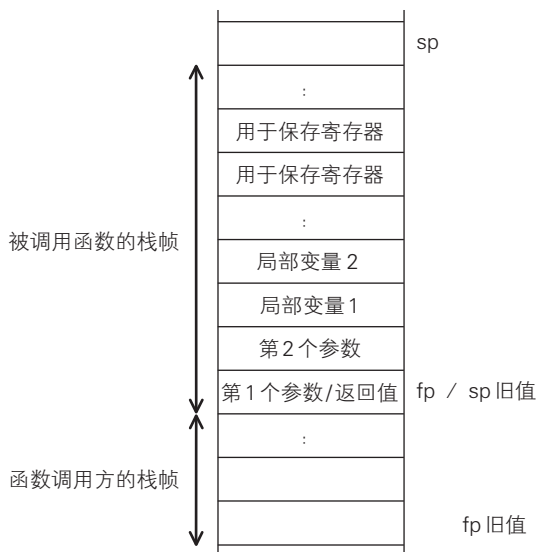


图 13.10 栈帧的使用方式 (假设函数具有两个参数)

Stone 虚拟机能够通过 `call` 指令执行函数调用。函数的调用方将首先把实参保存至被调用函数的栈帧中。被调用函数的栈帧与调用方的栈帧相邻，因此只要知道调用方栈帧的大小，寄存器就能确定实参的保存位置。假设函数调用方栈帧的大小为 s ，第 i ($i \geq 1$) 个参数将保存于栈区的第 $fp + s + i - 1$ 个元素中。

寄存器 `fp` 指向调用方栈帧的前端。被调用函数会首先执行 `save` 指令 (图 13.11)。该指令将在栈帧末端保存所有通用寄存器及寄存器 `ret` 与 `fp` 的值。由于调用方可能会把计算的中间结

果保存在寄存器中，因此虚拟机必须在由被调用函数产生的数据覆盖这些寄存器之前，将它们保存至栈帧中。函数的调用方并不一定会用满所有的寄存器，虚拟机应当仅保存必要的寄存器值。不过，为了便于实现，本章通过 `save` 指令保存了所有的寄存器值。

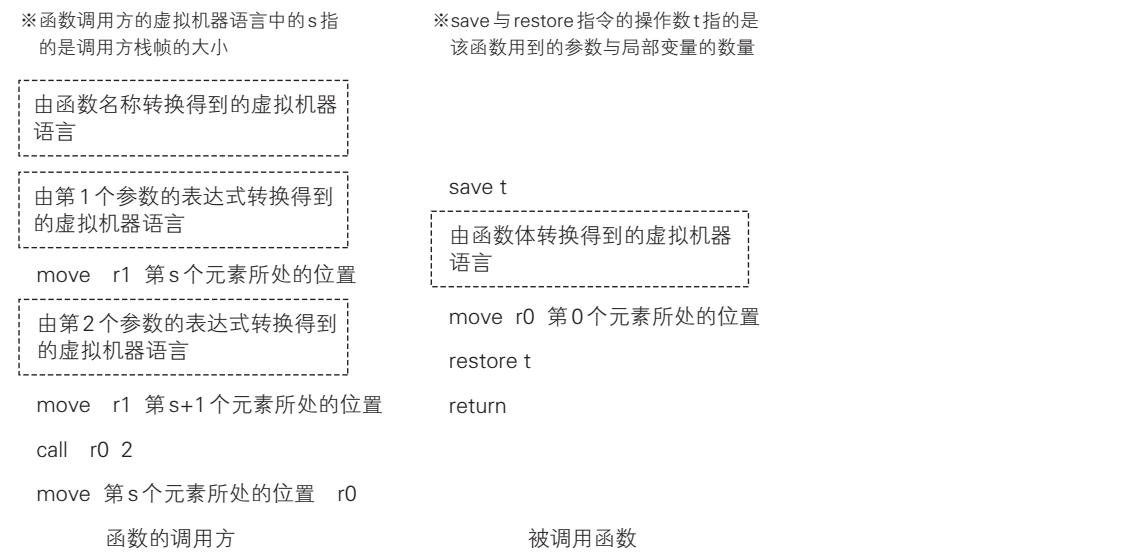


图13.11 用于实现函数调用的虚拟机器语言（假设函数调用方没有占用寄存器，函数具有两个参数）

- A

栈帧的末尾指的是哪里？从图 13.10 看，好象是头部嘛。
- H

A 君，图 13.10 下面是前端，上面才是末尾哦。新的栈帧会从上面进入。
- S

保存所有的寄存器有些浪费呢。
- C

你说的没错，遵循实际的调用惯例，虚拟机一般不应该保存所有的寄存器。
- F

调用方函数的职责是将那些还没有转移保存的寄存器值保存起来吧。

`save` 指令将更改寄存器 `fp` 与 `sp` 的值。在保存原本的寄存器值后，它首先将把 `sp` 的旧值复制给 `fp`，之后为 `sp` 加上一个指定的值。这样一来，寄存器 `fp` 与 `sp` 就会指向（被调用函数使用的）新的栈帧。

虚拟机应当在函数体执行完成后将返回值返回给函数的调用方，它需要将由 `save` 指令保存的值还原至寄存器中。Stone 虚拟机会把返回值保存在被调用函数的栈帧前端。调用方能够由此获取返回值，并根据寄存器的使用规则将它们复制给指定的寄存器。

由 `save` 指令保存的值能够通过 `restore` 指令还原。`restore` 指令将在函数调用结束前执行，它将把寄存器 `fp` 的值复制给 `sp`，并恢复通用寄存器与寄存器 `ret` 和寄存器 `fp` 在转移之前的原值。于是，寄存器值将还原为执行 `save` 指令之前的状态，寄存器 `fp` 与 `sp` 将再次指向原来（函数调用方使用）的栈帧。虚拟机将在执行 `restore` 指令后继续执行 `return` 指令，返

回函数的调用方。准确地说，虚拟机将强制跳转至调用方函数 `call` 指令之后的那条命令。寄存器 `ret` 中保存了 `call` 指令的返回位置。`return` 指令将通过（由 `restore` 指令还原的）寄存器 `ret` 的值来确定返回位置。

13.8 转换为虚拟机器语言

本章设计的 Stone 语言处理器将在执行过程中以对话的形式获取程序输入，之后先把它转换为抽象语法树，再进一步将抽象语法树转换为虚拟机器语言并执行。因此虚拟机器语言的转换时间也包含在执行时间中。由于这个原因，我们不需要将整个程序都转换为虚拟机器语言，仅需转换函数的定义部分。也就是说，Stone 语言处理器在通常情况下只要像之前一样通过 `eval` 方法执行程序，只有在遇到函数调用时才需要通过虚拟机执行函数体。

C C 语言会事先将程序转换为机器语言，并以文件保存，因此不必在意转换开销。

H Java 语言也是一样。只不过它不是把程序转换为机器语言，而是将它转换为二进制代码后再保存。

F 不过，Java 虚拟机会在执行过程中将二进制代码转换为机器语言，这就是所谓的动态编译。

C 嗯，说到动态编译，我们要注意的是，Java 语言并不会把所有的二进制代码都转换为机器语言。

`def` 语句用于定义函数，`DefStmtnt` 类是与之对应的抽象语法树节点类，该类的 `eval` 方法和原先一样，将返回一个表示函数的对象。与此同时，虚拟机会将函数体转换为虚拟机器语言，并把用于表示函数的对象记录至虚拟机器语言的前端。代码清单 13.5 是用于表示函数的对象的类定义。它是第 7 章（第 7 天）代码清单 7.8 中的 `Function` 类的子类。`entry` 字段表示虚拟机器语言前端所处的位置。

代码清单 13.5 VmFunction.java

```
package chap13;
import stone.ast.BlockStmtnt;
import stone.ast.ParameterList;
import chap6.Environment;
import chap7.Function;

public class VmFunction extends Function {
    protected int entry;
    public VmFunction(ParameterList parameters, BlockStmtnt body,
                      Environment env, int entry)
    {
        super(parameters, body, env);
        this.entry = entry;
    }
    public int entry() { return entry; }
}
```

抽象语法树各节点类的 `compile` 方法将实际执行把抽象语法树转换为虚拟机器语言的操作。该方法与 `eval` 及 `lookup` 方法类似，它也会一边依次遍历抽象语法树的节点，一边生成虚拟机器语言。

`DefStmtnt` 的 `eval` 方法会在内部对函数体的抽象语法树调用 `compile` 方法，将其转换为虚拟机器语言。`compile` 方法需要用到 `lookup` 方法的计算结果，与第 11 章的情况相同，`DefStmtnt` 的 `lookup` 方法将在 `eval` 方法（以及 `eval` 方法中的 `compile` 方法）之前调用。`lookup` 方法能够事先确定各变量值在环境中的保存位置。

代码清单 13.6 是抽象语法树各个类的 `compile` 方法。请读者注意，该程序一并修改了 `Arguments` 类的 `eval` 方法。与 `Arguments` 类的对象对应的抽象语法树节点用于表示实参序列。虚拟机将通过该类的 `eval` 方法执行最外层代码中的函数调用，因此，虚拟机器语言的执行将在该方法中开始。函数调用结束后，程序将返回至该 `eval` 方法，并把从栈区取得的返回值设定为 `eval` 方法自身的返回值。

H 老师，不讲解下 `BlockStmtnt` 类的 `compile` 方法吗？

F 它只不过是编译了代码块中各条语句，然后将得到的机器语言依次排列而已。

H `c.nextReg = initReg` 是什么意思？

C 这条语句将正在使用的寄存器的数量还原为了代码块执行之前的数量。按理说，在语句执行结束结束后，计算结果将被保存至寄存器中。不过，由于代码块只需使用最后一条语句的计算结果，因此不必保存其他的中间结果。

F 占用一个寄存器来保存用不到的计算结果，也没什么意义呢。

H 再讲解一下 `else` 之后逻辑行吗？

C 这段代码用于将空代码块转换为虚拟机器语言。Stone 语言中空代码块的计算结果为 0，因此我们会得到这样的虚拟机器语言。

`compile` 方法将接收一个 `Code` 对象作为参数。代码清单 13.7 是该对象的类定义。该对象用于保存虚拟机器语言转换过程中必需的信息。例如，Stone 虚拟机的引用（`svm`）、当前正在转换的函数的栈帧大小（`frameSize`），以及当前正在使用的寄存器数量（`nextReg`）等信息都将通过 `Code` 对象保存。

代码清单 13.6 VmEvaluator.java

```
package chap13;
import java.util.List;
import stone.StoneException;
import stone.Token;
import chap11.EnvOptimizer;
import chap6.Environment;
import chap6.BasicEvaluator.ASTreeEx;
import chap7.FuncEvaluator;
import javassist.gluonj.*;
```

```

import static chap13.Opcode.*;
import static javassist.gluonj.GluonJ.revise;
import static stone.ast.*;

@Require(EnvOptimizer.class)
@Reviser public class VmEvaluator {
    @Reviser public static interface EnvEx3 extends EnvOptimizer.EnvEx2 {
        StoneVM stoneVM();
        Code code();
    }
    @Reviser public static abstract class ASTreeVmEx extends ASTree {
        public void compile(Code c) {}
    }
    @Reviser public static class ASTListEx extends ASTList {
        public ASTListEx(List<ASTree> c) { super(c); }
        public void compile(Code c) {
            for (ASTree t: this)
                ((ASTreeVmEx)t).compile(c);
        }
    }
    @Reviser public static class DefStmntVmEx extends EnvOptimizer.DefStmntEx {
        public DefStmntVmEx(List<ASTree> c) { super(c); }
        @Override public Object eval(Environment env) {
            String funcName = name();
            EnvEx3 vmenv = (EnvEx3)env;
            Code code = vmenv.code();
            int entry = code.position();
            compile(code);
            ((EnvEx3)env).putNew(funcName, new VmFunction(parameters(), body(),
                env, entry));

            return funcName;
        }
        public void compile(Code c) {
            c.nextReg = 0;
            c.frameSize = size + StoneVM.SAVE_AREA_SIZE;
            c.add(SAVE);
            c.add(encodeOffset(size));
            ((ASTreeVmEx)revise(body())).compile(c);
            c.add(MOVE);
            c.add(encodeRegister(c.nextReg - 1));
            c.add(encodeOffset(0));
            c.add(RESTORE);
            c.add(encodeOffset(size));
            c.add(RETURN);
        }
    }
    @Reviser public static class ParamsEx2 extends EnvOptimizer.ParamsEx {
        public ParamsEx2(List<ASTree> c) { super(c); }
        @Override public void eval(Environment env, int index, Object value) {
            StoneVM vm = ((EnvEx3)env).stoneVM();
            vm.stack()[offsets[index]] = value;
        }
    }
    @Reviser public static class NumberEx extends NumberLiteral {
        public NumberEx(Token t) { super(t); }
    }
}

```

```

    public void compile(Code c) {
        int v = value();
        if (Byte.MIN_VALUE <= v && v <= Byte.MAX_VALUE) {
            c.add(BCONST);
            c.add((byte)v);
        }
        else {
            c.add(ICONST);
            c.add(v);
        }
        c.add(encodeRegister(c.nextReg++));
    }
}

@Reviser public static class StringEx extends StringLiteral {
    public StringEx(Token t) { super(t); }
    public void compile(Code c) {
        int i = c.record(value());
        c.add(SCONST);
        c.add(encodeShortOffset(i));
        c.add(encodeRegister(c.nextReg++));
    }
}

@Reviser public static class NameEx2 extends EnvOptimizer.NameEx {
    public NameEx2(Token t) { super(t); }
    public void compile(Code c) {
        if (nest > 0) {
            c.add(GMOVE);
            c.add(encodeShortOffset(index));
            c.add(encodeRegister(c.nextReg++));
        }
        else {
            c.add(MOVE);
            c.add(encodeOffset(index));
            c.add(encodeRegister(c.nextReg++));
        }
    }
}

public void compileAssign(Code c) {
    if (nest > 0) {
        c.add(GMOVE);
        c.add(encodeRegister(c.nextReg - 1));
        c.add(encodeShortOffset(index));
    }
    else {
        c.add(MOVE);
        c.add(encodeRegister(c.nextReg - 1));
        c.add(encodeOffset(index));
    }
}
}

@Reviser public static class NegativeEx extends NegativeExpr {
    public NegativeEx(List<ASTree> c) { super(c); }
    public void compile(Code c) {
        ((ASTreeVmEx)operand()).compile(c);
        c.add(NEG);
        c.add(encodeRegister(c.nextReg - 1));
    }
}

```

```

    }
}

@Reviser public static class BinaryEx extends BinaryExpr {
    public BinaryEx(List<ASTree> c) { super(c); }
    public void compile(Code c) {
        String op = operator();
        if (op.equals("=")) {
            ASTree l = left();
            if (l instanceof Name) {
                ((ASTreeVmEx)right()).compile(c);
                ((NameEx2)l).compileAssign(c);
            }
            else
                throw new StoneException("bad assignment", this);
        }
        else {
            ((ASTreeVmEx)left()).compile(c);
            ((ASTreeVmEx)right()).compile(c);
            c.add(getOpcode(op));
            c.add(encodeRegister(c.nextReg - 2));
            c.add(encodeRegister(c.nextReg - 1));
            c.nextReg--;
        }
    }

    protected byte getOpcode(String op) {
        if (op.equals("+"))
            return ADD;
        else if (op.equals("-"))
            return SUB;
        else if (op.equals("*"))
            return MUL;
        else if (op.equals("/"))
            return DIV;
        else if (op.equals("%"))
            return REM;
        else if (op.equals("=="))
            return EQUAL;
        else if (op.equals(">"))
            return MORE;
        else if (op.equals("<"))
            return LESS;
        else
            throw new StoneException("bad operator", this);
    }
}

@Reviser public static class PrimaryVmEx extends FuncEvaluator.PrimaryEx {
    public PrimaryVmEx(List<ASTree> c) { super(c); }
    public void compile(Code c) {
        compileSubExpr(c, 0);
    }

    public void compileSubExpr(Code c, int nest) {
        if (hasPostfix(nest)) {
            compileSubExpr(c, nest + 1);
            ((ASTreeVmEx)revise(postfix(nest))).compile(c);
        }
    }
}

```

```

        else
            ((ASTreeVmEx)operand()).compile(c);
    }
}
@Reviser public static class ArgumentsEx extends Arguments {
    public ArgumentsEx(List<ASTree> c) { super(c); }
    public void compile(Code c) {
        int newOffset = c.frameSize;
        int numOfArgs = 0;
        for (ASTree a: this) {
            ((ASTreeVmEx)a).compile(c);
            c.add(MOVE);
            c.add(encodeRegister(--c.nextReg));
            c.add(encodeOffset(newOffset++));
            numOfArgs++;
        }
        c.add(CALL);
        c.add(encodeRegister(--c.nextReg));
        c.add(encodeOffset(numOfArgs));
        c.add(MOVE);
        c.add(encodeOffset(c.frameSize));
        c.add(encodeRegister(c.nextReg++));
    }
    public Object eval(Environment env, Object value) {
        if (!(value instanceof VmFunction))
            throw new StoneException("bad function", this);
        VmFunction func = (VmFunction)value;
        ParameterList params = func.parameters();
        if (size() != params.size())
            throw new StoneException("bad number of arguments", this);
        int num = 0;
        for (ASTree a: this)
            ((ParamsEx2)params).eval(env, num++, ((ASTreeEx)a).eval(env));
        StoneVM svm = ((EnvEx3)env).stoneVM();
        svm.run(func.entry());
        return svm.stack()[0];
    }
}
@Reviser public static class BlockEx extends BlockStmnt {
    public BlockEx(List<ASTree> c) { super(c); }
    public void compile(Code c) {
        if (this.numChildren() > 0) {
            int initReg = c.nextReg;
            for (ASTree a: this) {
                c.nextReg = initReg;
                ((ASTreeVmEx)a).compile(c);
            }
        }
        else {
            c.add(BCONST);
            c.add((byte)0);
            c.add(encodeRegister(c.nextReg++));
        }
    }
}
}

```

```

@Reviser public static class IfEx extends IfStmnt {
    public IfEx(List<ASTree> c) { super(c); }
    public void compile(Code c) {
        ((ASTreeVmEx)condition()).compile(c);
        int pos = c.position();
        c.add(IFZERO);
        c.add(encodeRegister(--c.nextReg));
        c.add(encodeShortOffset(0));
        int oldReg = c.nextReg;
        ((ASTreeVmEx)thenBlock()).compile(c);
        int pos2 = c.position();
        c.add(GOTO);
        c.add(encodeShortOffset(0));
        c.set(encodeShortOffset(c.position() - pos), pos + 2);
        ASTree b = elseBlock();
        c.nextReg = oldReg;
        if (b != null)
            ((ASTreeVmEx)b).compile(c);
        else {
            c.add(BCONST);
            c.add((byte)0);
            c.add(encodeRegister(c.nextReg++));
        }
        c.set(encodeShortOffset(c.position() - pos2), pos2 + 1);
    }
}

@Reviser public static class WhileEx extends WhileStmnt {
    public WhileEx(List<ASTree> c) { super(c); }
    public void compile(Code c) {
        int oldReg = c.nextReg;
        c.add(BCONST);
        c.add((byte)0);
        c.add(encodeRegister(c.nextReg++));
        int pos = c.position();
        ((ASTreeVmEx)condition()).compile(c);
        int pos2 = c.position();
        c.add(IFZERO);
        c.add(encodeRegister(--c.nextReg));
        c.add(encodeShortOffset(0));
        c.nextReg = oldReg;
        ((ASTreeVmEx)body()).compile(c);
        int pos3 = c.position();
        c.add(GOTO);
        c.add(encodeShortOffset(pos - pos3));
        c.set(encodeShortOffset(c.position() - pos2), pos2 + 2);
    }
}
}

```

代码清单 13.7 Code.java

```

package chap13;

public class Code {

```

```

protected StoneVM svm;
protected int codeSize;
protected int numOfStrings;
protected int nextReg;
protected int frameSize;

public Code(StoneVM stoneVm) {
    svm = stoneVm;
    codeSize = 0;
    numOfStrings = 0;
}
public int position() { return codeSize; }
public void set(short value, int pos) {
    svm.code()[pos] = (byte)(value >>> 8);
    svm.code()[pos + 1] = (byte)value;
}
public void add(byte b) {
    svm.code()[codeSize++] = b;
}
public void add(short i) {
    add((byte)(i >>> 8));
    add((byte)i);
}
public void add(int i) {
    add((byte)(i >>> 24));
    add((byte)(i >>> 16));
    add((byte)(i >>> 8));
    add((byte)i);
}
public int record(String s) {
    svm.strings()[numOfStrings] = s;
    return numOfStrings++;
}
}

```

13.9 通过虚拟机执行

最后，我们来看一下通过 Stone 虚拟机执行程序的解释器与它的启动程序。代码清单 13.8 与代码清单 13.9 分别是这两个程序的代码。本章代码清单 13.6 中的 `VmEvaluator` 修改器利用了第 11 章实现的 `lookup` 方法。因此，修改器之前需要添加 `@Require` 来表示它依赖于第 11 章代码清单 11.4 中的 `EnvOptimizer` 修改器。程序将在启动后自动应用该修改器。

在解释器启动后，程序中定义并调用的函数将通过虚拟机执行。在函数定义时，虚拟机器语言转换仅需执行一次，也就是说，一旦完成定义，虚拟机就能反复使用转换得到的虚拟机器语言，不存在额外的开销。

如前所述，实际执行编译操作的是定义了函数的 `def` 语句的 `eval` 方法。该方法将进一步调用 `compile` 方法。函数在编译后，将由 `Arguments` 类的 `eval` 方法执行。该类的对象是一种用于表示函数调用表达式的抽象语法树节点。如果函数又调用了其他函数，解释器不会再次使

用 Arguments 类的 eval 方法。由该类的 compile 方法生成的虚拟机器语言含有一条 call 指令，该指令将直接调用新的函数。

- A** 程序在编译为机器语言后，执行速度能提高多少呢？结果真是令人期待呀。
- C** 嗯，只计算斐波那契数的话，其实没太大差别。甚至会像第 11 章那样稍慢一些。
- A** 怎么这样呀，白费劲了。
- F** 之所有会有这个问题，是因为虚拟机的实现语言是 Java 吗？
- C** 很难讲具体问题出在哪里。倒不如说是因为之前设计的时候，我们没有有意识地考虑虚拟机器语言的性能优化。
- S** 嗯，抽象语法树节点的遍历处理，其实并不会明显影响速度。
- C** 没错。真正拖累性能的原因和第 11 章大同小异，比方说，在 Stone 虚拟机中，整数是通过 Integer 对象来表示的，诸如此类。
- S** 而且这个虚拟机还存在内存泄漏问题。
- F** 呀！还有这种错误啊。
- S** 从函数返回时，虚拟机没有处理那些不再使用的栈帧。于是其中包含的对象没能成功 GC（垃圾回收）。
- F** 哦，也就是说没有把所有元素都赋值为 null 对吧。
- H** 老师，如果经常 GC，会不会影响计算斐波那契数的速度呢？
- C** 其实我尝试过为虚拟机的 restore 指令添加清空栈帧（将数组元素赋值为 null）的功能，不过从结果来看，没什么变化。

代码清单 13.8 VmInterpreter.java

```
package chap13;
import stone.FuncParser;
import stone.ParseException;
import chap11.EnvOptInterpreter;
import chap8.Natives;

public class VmInterpreter extends EnvOptInterpreter {
    public static void main(String[] args) throws ParseException {
        run(new FuncParser(),
            new Natives().environment(new StoneVMEnv(100000, 100000, 1000)));
    }
}
```

代码清单 13.9 VmRunner.java

```
package chap13;
import javassist.gluonj.util.Loader;
import chap8.NativeEvaluator;

public class VmRunner {
```

```
public static void main(String[] args) throws Throwable {  
    Loader.run(VmInterpreter.class, args, VmEvaluator.class,  
              NativeEvaluator.class);  
}
```

专栏 4 副业



第 2 部分

性能优化篇

第
14
天

为 Stone 语言添加静态类型 支持以优化性能

第
14
天为 Stone 语言添加静态类型支持以
优化性能

前几章设计的 Stone 语言是一种所谓的动态类型语言，本章将把它修改为一种静态类型语言，并通过静态类型信息优化程序的运行性能。

上一章借助 Java 语言设计了一种专用的虚拟机，用于执行中间代码。从内部来看，该虚拟机通过 Java 语言的 Object 类型来表示所有类型的值，整数也将由 Integer 对象表现。不只是虚拟机，本书设计的解释器也采用了这种设计方式。不可否认，这是程序执行速度较慢的一个重要原因。

本章将利用静态数据类型，尽可能以 int 类型的值来表示整数值。同时，我们将不再使用专用的虚拟机，而会直接使用 Java 虚拟机。抽象语法树需要预先转换为 Java 二进制代码，不过，由于整数改以 int 类型表示，转换得到的 Java 二进制代码执行效率也会较高。Java 虚拟机能够体现实际的硬件，并将整数等基本数据类型作为非对象的特殊数据类型处理。因此，用 int 类型的值而非 Integer 对象表示整数时，程序的执行效率更高。如果不采用这种设计，虚拟机内部就将在 int 类型的值与 Integer 对象间频繁执行转换，影响性能。

F 最终，Stone 语言也具有数据类型了啊。与其说是增强，倒不如说是一种倒退。

A 每次都要声明数据类型真麻烦呀。

C 那么让我们为了 A 君，再添加函数型语言中常见的类型推论功能吧。

F Scala 也支持类型推论呢。

C 是的。支持类型推论之后，大部分变量都无需声明类型，Stone 语言将能自动推测并选择合适的类型。

A 原来如此，也就是说会选择最佳答案对吧。

14.1 指定变量类型

支持静态数据类型的程序设计语言（静态类型语言，静态语言）的特点是，它需要在声明变量与参数的同时指定它们的数据类型。如果语言支持类型推论，一部分甚至大部分的类型指定就能省略。不过，能省略并不表示它们就不需要指定数据类型。

静态语言有一些缺点。例如，即使有需要，数据类型不同的变量值之间也无法相互赋值，而且某些变量的类型可能较为复杂，不易理解。不过，静态类型语言也有一些重要的优点。

- 通过数据类型检查，它能在一定程度上确保程序的正确性
- 静态数据类型信息有助于提高程序的执行速度

前者能够在程序执行前，确保程序不会在执行途中，因发生诸如函数不存在，或对非数值类型的值进行乘法计算等错误，而中止执行。不过，类型检查具体能确保多大程度上的正确性，取决于该语言采用的数据类型系统。如果语言采用了强数据类型系统，类型检查机制将能确保很高的正确性，否则就只能检查出部分错误。

A 虽说类型检查能够确保程序正确，但并不是说用不着调试了。

C 不过，如果能够通过机器检查确保一定程度的正确性，开发效率也能得到很大的提升。

F 也就是说，要在书写数据类型不便与程序的正确性之间做出取舍对吧？

C 不光是写起来不方便，要将一段能够正常运行的动态类型语言程序改为静态类型语言程序，使其能通过类型检查，是一件很不容易，甚至不可能的事。这也是静态数据类型的一个不足。

H 这要看数据类型系统研究的发展了呢。

C 嗯，确实。现在的数据类型系统有时还不能很好地表达程序员的设计意图。

按照惯例，我们先来改进 Stone 语言的语法，使它能够支持数据类型声明。首先增加的是 var 语句。它用于定义一个新的变量，并指定该变量的初始值与数据类型。

```
var x: Int = 7
```

上例中的语句对变量 x 做了定义，它是一个 Int 类型的变量，初始值为 7。读者需要注意，变量声明时初始值不得省略。变量名之后跟有冒号与数据类型，它们都可以省略。

```
var x = 7
```

这条 var 语句省略了变量 x 的数据类型。如果不需要指定数据类型，语句中的 var 也能省略，仅需书写 x=7 即可。也就是说，之前那种通过赋值表达式定义新变量的写法依然有效。

F 以 :Int 的方式来表达数据类型，跟 Scala 很像。

我们规定，var 语句支持声明 Int、String 与 Any 这三种类型。Int 类型表示整数，String 类型表示字符串。Any 同时包含整数与字符串这两种类型。也就是说，一个整数值既可以由 Int 类型表示，也能由 Any 类型表示。字符串也是如此。

Any 类型的变量能够被赋以 Int 或 String 类型的值，反之则不行。在将 Any 类型的值赋值给一个 Int 类型的变量时，如果该值不是一个整数而是字符串，就会发生问题，因此 Stone 语言不支持这种操作。否则，Int 与 String 类型的变量值就能相互转换，为变量指定数据类型的做法将失去意义。

C Any 类型的变量，既能用整数赋值，也能用字符串赋值。

F 总而言之，Any 类型是这两种类型的超类对吧？

A 就像是 Java 中的 Object 类那样？

H 老师，是不是说之前的 Stone 语言里，所有变量都属于 Any 类型呀？

F 意思是有点像，但还是不太一样呢。

S 嗯，如果将变量定义为 Any 类型，很多程序就无法正常执行了。本章之前的 Stone 语言就不存在这个问题。例如，对于下面的代码，

```
x = "three"
print x
x = 3
print x * 2
```

如果 x 的类型是 Any，就会发生数据类型错误。

H 哦， $x * 2$ 中的 x 如果是 Any 类型的变量，就会出错了。

F 就算没有后两行，仍然会发生数据类型错误哦。

A 小 H，所以说在之前的 Stone 语言中，变量类型并不都是 Any，可要注意了。

如果语句没有指定数据类型，变量的类型将取决于类型推论的结果。不过，在此我们暂先不考虑类型推论，规定没有指定数据类型的变量皆为 Any 类型。在实现了类型推论功能之后，我们会修改这一规则。

虽然 Any 类型也包含整数，但它只支持有限的算术操作。Any 类型的值只能进行 + 运算。另外，原生函数 `toInt` 能接收 Any 类型的值作为参数，返回相应的 Int 值。除此以外，-（减法）等运算都无法用于 Any 类型的值。例如，对于 Any 类型的变量 x，即使它的值为整数 3，表达式 $x - 1$ 依然会引起数据类型错误。这是因为，变量 x 属于 Any 类型并不能确保它的值一定是一个整数。

H 老师，话说回来，函数的参数不需要指定数据类型吗？

除了 var 语句，我们还将为函数定义语句添加参数及返回值的数据类型指定功能。例如，语句

```
def inc(n: Int): Int { n + 1 }
```

定义了函数 `inc`，它将接收一个 Int 类型的参数 n，并返回一个 Int 类型的返回值。右括号) 之后跟着的返回值的类型。与 var 语句类似，冒号 : 与后接的数据类型名称用于指定参数或返回值的类型，它们同样能够省略。

代码清单 14.1 列出了对 Stone 语法规则的一些修改，经过这些修改，Stone 语言将能支持上述数据类型指定功能。与 var 语句对应的非终结符是 `variable`。同时，我们需要为 `statement` 增加 `variable` 这一可能情况。此外，`param` 与 `def` 的定义也需要修改，以支持非终结符 `type_tag`。

代码清单 14.2 是与修改后的语法规则对应的语法分析器程序。代码清单 14.3 与代码清单

14.4 是新定义的抽象语法树节点的对象类型。

代码清单 14.2 中使用的方法大多已经在第 7 章（第 7 天）的 7.1 节中作了说明。首次出现的只有 `reset` 方法。该方法用于删除由超类 `FuncParser` 继承而来的 `def` 定义。在代码清单 14.2 中，原有的 `def` 定义将被 `reset` 方法暂时删除，之后，它将由构造函数重新定义。

为了执行新增的 `var` 语句，我们通过代码清单 14.5 所示的修改器为 `VarStmnt` 类添加了 `eval` 方法。与第 11 章（第 11 天）一样，本章也将通过编号而非名称查找环境，事先确定变量值的保存位置。因此，`VarStmnt` 类还需要新增一个 `lookup` 方法。需要注意的是，该修改器也能为其他的类添加或修改方法。由于 `def` 语句的语法有些变化，与之对应的抽象语法树的形式也要相应更改。为此，我们需要修改或新增一些抽象语法树节点类中的方法。代码清单 14.5 中的修改器虽然尚不支持实际的数据类型指定处理，且不会执行数据类型检查，但已经能够正确执行变量具有数据类型的 Stone 语言程序。

代码清单 14.1 与数据类型相关的语法规则

```
type_tag : ":" IDENTIFIER
variable : "var" IDENTIFIER [ type_tag ] "=" expr
param    : IDENTIFIER [ type_tag ]
def       : "def" IDENTIFIER param_list [ type_tag ] block
statement : variable | "if" ... | "while" ... | simple
```

代码清单 14.2 TypedParser.java

```
package stone;
import static stone.Parser.rule;
import stone.ast.*;

public class TypedParser extends FuncParser {
    Parser typeTag = rule(TypeTag.class).sep(":").identifier(reserved);
    Parser variable = rule(VarStmnt.class)
        .sep("var").identifier(reserved).maybe(typeTag)
        .sep("=").ast(expr);

    public TypedParser() {
        reserved.add(":");
        param.maybe(typeTag);
        def.reset().sep("def").identifier(reserved).ast(paramList)
            .maybe(typeTag).ast(block);
        statement.insertChoice(variable);
    }
}
```

代码清单 14.3 VarStmnt.java

```
package stone.ast;
import java.util.List;

public class VarStmnt extends ASTList {
    public VarStmnt(List<ASTree> c) { super(c); }
```

```

    public String name() { return ((ASTLeaf)child(0)).token().getText(); }
    public TypeTag type() { return (TypeTag)child(1); }
    public ASTree initializer() { return child(2); }
    public String toString() {
        return "(var " + name() + " " + type() + " " + initializer() + ")";
    }
}

```

代码清单 14.4 TypeTag.java

```

package stone.ast;
import java.util.List;

public class TypeTag extends ASTList {
    public static final String UNDEF = "<Undef>";
    public TypeTag(List<ASTree> c) { super(c); }
    public String type() {
        if (numChildren() > 0)
            return ((ASTLeaf)child(0)).token().getText();
        else
            return UNDEF;
    }
    public String toString() { return ":" + type(); }
}

```

代码清单 14.5 TypedEvaluator.java

```

package chap14;
import java.util.List;
import javassist.gluonj.*;
import stone.ast.*;
import chap11.EnvOptimizer;
import chap11.Symbols;
import chap11.EnvOptimizer.ASTreeOptEx;
import chap6.Environment;
import chap6.BasicEvaluator.ASTreeEx;

@Require(EnvOptimizer.class)
@Reviser public class TypedEvaluator {
    @Reviser public static class DefStmntEx extends EnvOptimizer.DefStmntEx {
        public DefStmntEx(List<ASTree> c) { super(c); }
        public TypeTag type() { return (TypeTag)child(2); }
        @Override public BlockStmnt body() { return (BlockStmnt)child(3); }
        @Override public String toString() {
            return "(def " + name() + " " + parameters() + " " + type() + " "
                + body() + ")";
        }
    }
    @Reviser public static class ParamListEx extends EnvOptimizer.ParamsEx {
        public ParamListEx(List<ASTree> c) { super(c); }
        @Override public String name(int i) {
            return ((ASTLeaf)child(i).child(0)).token().getText();
        }
    }
}

```

```

    }
    public TypeTag typeTag(int i) {
        return (TypeTag)child(i).child(1);
    }
}
@Reviser public static class VarStmntEx extends VarStmnt {
    protected int index;
    public VarStmntEx(List<ASTree> c) { super(c); }
    public void lookup(Symbols syms) {
        index = syms.putNew(name());
        ((ASTreeOptEx)initializer()).lookup(syms);
    }
    public Object eval(Environment env) {
        Object value = ((ASTreeEx)initializer()).eval(env);
        ((EnvOptimizer.EnvEx2)env).put(0, index, value);
        return value;
    }
}
}

```

14.2 通过数据类型检查发现错误

经过一番努力，Stone 语言终于实现了变量数据类型指定的功能，接下来我们将根据这些信息，对程序进行数据类型检查。数据类型检查将分析程序包含的表达式中所有的静态数据类型，对变量赋值操作中的数据类型是否匹配、字符串之间是否执行了乘法等非法运算，以及函数调用的参数类型是否正确等问题进行检查。具体来说，它将根据变量与参数的数据类型，检查程序中所有的表达式（包括子表达式与语句）是否遵循类型指派规则指定了相应的静态数据类型，确保程序上下文没有矛盾。

为了实现数据类型检查功能，我们要为抽象语法树的节点类添加 `typeCheck` 方法。该方法能够对数据类型进行检查，它与用于执行程序的 `eval` 方法非常类似，都会从抽象语法树的根节点开始递归调用自身，完成整棵语法树的遍历。

`eval` 方法将接收一个环境作为参数，并根据该环境计算表达式的值后返回。这里计算的表达式，是与以 `eval` 方法的调用对象为根节点的抽象语法树对应的表达式。例如，对与表达式 `x-2` 对应的抽象语法树的根节点对象调用 `eval` 方法，返回值就是该表达式的值。变量 `x` 的值由 `eval` 的环境参数决定。对于用于执行数据类型检查的 `typeCheck` 方法，它将接收一个数据类型环境作为参数，计算表达式的静态数据类型并返回结果。以 `x-2` 为例，调用根节点对象的 `typeCheck` 方法将返回该表达式的数据类型。变量 `x` 的数据类型由 `typeCheck` 方法的数据类型环境参数决定。`eval` 方法的环境参数是由变量名与对应的值组成的名值对，`typeCheck` 方法的数据类型环境则是由变量名与对应的数据类型组成的名值对。

类型指派规则看似复杂，其实它的核心是怎样根据各条表达式的子表达式类型，计算该表达式自身的类型。正如 `eval` 方法实现了表达式值的计算，`typeCheck` 方法将实现对各个抽象语

法树节点类的数据类型的计算。

例如，对于单目减法运算表达式 $-(x * 2)$ ，其子表达式 $(x * 2)$ 是 $-$ 运算符的操作数，必须为 `Int` 类型。于是，整个单目运算表达式也将是 `Int` 类型。这就是类型指派规则。

C 准确来讲，类型指派规则指的是“由于操作数的类型是 `Int`，因此整个表达式的类型也是 `Int`”。

A 咦，这跟前面哪里不一样了？

C 也就是说，类型指派规则并没有规定如果操作数不是 `Int` 类型，就一定会立即报错。

H 虽说即使操作数不是 `Int` 类型也不会立即出错，但由于没有其他适用的指派规则，类型检查只好中途结束，最终还是会发生数据类型错误呢。

A 哦，这样啊，我觉得这种小问题无所谓啦。

具体的类型指派规则因程序设计语言而异。在 `Stone` 语言中，双目运算表达式的类型指派规则稍有些复杂，下面将进行详细说明。首先，如果运算符两侧的子表达式都是 `Int` 类型，整个双目运算表达式将也是 `Int` 类型。对于 $+$ 表达式，如果两侧都是 `String` 类型，整个表达式则是 `String` 类型，否则为 `Any` 类型。这使 $+$ 运算符能够用于字符串的连接运算。还有一种特殊情况是 $=$ 运算符。对于 $=$ 运算符，如果左右两侧的类型一致，整个赋值表达式的类型将与子表达式相同。如果左侧是一个新出现的变量，尚无特定的数据类型，该变量将被指定为与右侧相同的类型。

A 还真复杂啊。

代码清单 14.6 是用于添加 `typeCheck` 方法的修改器。代码清单 14.7 是用于表示数据类型环境的 `TypeEnv` 类的定义。`typeCheck` 方法在遇到类型错误时将抛出 `TypeException` 异常。代码清单 14.8 是该异常的定义。

用于表示数据类型环境的 `TypeEnv` 类保存的不是由变量名与类型组成的名值对，而是由变量的保存位置（用于表示该位置的整数）与数据类型组成的名值对。在此我们将沿用与第 11 章（第 11 天）相似的方式，使该类在执行时通过编号而非名称查找数据类型环境。

代码清单 14.9 中，`TypeInfo` 类的对象表示数据类型。该类定义了 `ANY`、`INT` 与 `STRING` 这三个 `static` 字段。它们都是 `TypeInfo` 类型的值，表示各自对应的数据类型。此外，该类还定义了 `UnknownType` 与 `FunctionType` 这两个嵌套子类。前者表示程序省略了类型指定，并暂且采用了与 `ANY` 相同的实现逻辑。在之后实现类型推论时，我们将修改该类的实现。

第二个嵌套类 `FunctionType` 用于表示函数的类型。函数的类型通过参数序列的类型与返回值的类型表现。例如，假设某个函数将接收 `Int` 类型与 `Any` 类型的参数，并返回 `String` 类型的返回值。这样一来，我们就可以称它是一个“依次接收 `Int` 与 `Any` 类型的参数且返回值为 `String` 类型”的函数。

抽象语法树各节点类的 `typeCheck` 方法在计算类型时，将首先递归调用子表达式的 `typeCheck` 方法。如果没有子表达式，则调用 `TypeInfo` 对象的 `assertSubtypeOf` 方法，确认它是否满足类型指派规则的前提条件。例如，在检查由 `=` 运算符构成的赋值表达式时，`typeCheck` 方法将像下面这样，对表示左侧类型的 `type` 与表示右侧类型的 `valueType` 调用 `assertSubtypeOf` 方法。

```
valueType.assertSubtypeOf(type, tenv, this);
```

这条语句出自代码清单 14.6 中的 `NameEx2` 修改器。其中，`type` 与 `valueType` 都是 `TypeInfo` 类型的对象。`assertSubtypeOf` 方法将判断 `valueType` 表示的类型是否与 `type` 表示的相同，或是它的子类（子类型），如果不是，该方法将抛出 `TypeException` 异常，否则不执行任何操作。

C 以防万一，我还是再重新说一下，如果 `S` 类型是 `T` 类型的子类，`T` 就是 `S` 的父类。

F A 君，如果一个类型是 `T` 类型的子类，就表示我们可以放心地用它来代替 `T` 类型使用。

S 嗯，如果是 `T` 类型的子类，就算用它替换与 `T` 类型相关的表达式，其他表达式与值的类型也不会产生冲突。

A 这两种说法听着没区别啊。由于一个 `Int` 类型的值同时也属于 `Any` 类型，因此 `Int` 是 `Any` 的子类，这样理解没问题吧？

在对赋值表达式进行类型检查时，右侧表达式的类型只要与左侧的变量类型相同，或是它的子类即可。因此，像上面那样调用 `assertSubtypeOf` 方法就能完成检查。例如，假设 `=` 运算符右侧的表达式为 `Int` 类型，这时，即使左侧变量的类型是 `Any`，也不会发生数据类型错误。

C 从现在起，在使用 `Stone` 语言时，不仅是表达式，语句也必须都指定数据类型才行。

由于 `Stone` 语言不支持 `return` 语句或类似的语法功能，因此不仅是表达式，所有语句都必须指定数据类型。首先，对于由大括号 `{ }` 括起的代码块，它的数据类型与其中最后一条语句或表达式的类型相同。代码块中其余表达式与语句的类型将被忽略。

`if` 语句中的条件表达式为 `Int` 类型。整条 `if` 语句的类型，与最终执行的代码块类型相同，由条件表达式的结果决定。如果这两个代码块的类型不同，整条 `if` 语句的类型将是这两种类型的父类。它由 `TypeInfo` 类的 `union` 方法确定。需要注意的是，对于条件表达式不同的值，`if` 语句将分别执行特定的代码块，因而具有不同的数据类型。如果 `if` 语句的类型是这两种代码块类型的父类，那无论最终执行哪一个代码块，都不会发生问题。

C 其实，`if` 语句的条件表达式并不一定非要是一条整数运算表达式。当且仅当计算结果为 0 时表达式值为假，除此之外，任意非零整数也好，字符串也好，表达式的值都为真。

H 不过，要是将表达式的类型限定为 `Int` 类型，之后把 `if` 语句转换为 `Java` 语言会比较容易。

`while` 语句的条件表达式同样是 `Int` 类型。不过，整条 `while` 语句的类型并不直接沿用代码块的类型。我们规定，`while` 语句的类型是代码块类型与 `Int` 类型的父类。因此，`while` 语句只能是 `Any` 类型或 `Int` 类型两者其一。之所以这样规定，是因为如果 `while` 语句的代码块尚未执行，语句的计算结果为 0，必然属于 `Int` 类型。

如果类型检查的对象是 `def` 语句，整条语句的类型与它所定义的函数类型相同。它将根据该函数的类型创建一个 `FunctionType` 对象，并添加至数据类型环境。之后，`typeCheck` 方法将另外新建一个环境，用于对函数体内部做类型检查。环境创建后，`typeCheck` 方法将把函数参数的类型添加至该数据类型环境，并检查函数体，也就是代码块的类型是否与函数返回值的类型一致，或是它的子类。

C 这里很重要的一点在于，`typeCheck` 方法首先需要将 `FunctionType` 对象添加至数据类型环境中。如果之后没有对函数体做类型检查，递归调用将无法执行。

`Arguments` 类用于表示函数调用表达式，它的 `typeCheck` 方法将递归调用参数的 `typeCheck` 方法，计算实参表达式的类型，并检查它们与形参的类型是否一致。形参的类型能够很容易通过函数调用方获知。整个函数调用表达式的类型与被调用函数的返回值类型相同。

C 在调用函数时，如果不知道函数调用方的类型，就无法执行类型检查了。因此，为了让函数能够实现递归调用，我们必须把它的数据类型先添加至数据类型环境中。

S 嗯，那在遇到所谓的相互递归时，我们该怎么做呢？

A 相互递归？

F 我们将函数 `f` 中调用了函数 `g`，`g` 中又调用了 `f` 的情况称为相互递归。

C 在本章中还不能实现对相互递归的支持。我把它留作读者的课后练习了。

S 是要添加类似于 Scheme 和 OCaml 中 `let rec` 那样的语法功能吗？

C 总之要使函数能够被重新定义。现在这种做法将以数据类型错误处理，但为了实现相互递归调用，我们应该允许程序在不改变函数数据类型的前提下重新定义函数。

F 也就是说，我们可以像下面这样，先临时定义 `odd`，之后再覆盖该定义，对吧？

```
def odd(x: Int): String { "not implemented" }
def even(x: Int): String {
  if x == 0 { "Yes" } else { odd(x - 1) }
}
def odd(x: Int): String {
  if x == 0 { "No" } else { even(x - 1) }
}
```

H 老师，之前的语言处理器理论上就已经支持重新定义函数了呀，为什么算作数据类型错误呢？

C 这是为了之后把它转换为 Java 二进制代码时能更方便些。

代码清单 14.6 TypeChecker.java

```

package chap14;
import java.util.List;
import chap7.FuncEvaluator;
import chap11.EnvOptimizer;
import stone.Token;
import static javassist.gluonj.GluonJ.revise;
import stone.ast.*;
import javassist.gluonj.*;

@Require(TypedEvaluator.class)
@Reviser public class TypeChecker {
    @Reviser public static abstract class ASTreeTypeEx extends ASTree {
        public TypeInfo typeCheck(TypeEnv tenv) throws TypeException {
            return null;
        }
    }
    @Reviser public static class NumberEx extends NumberLiteral {
        public NumberEx(Token t) { super(t); }
        public TypeInfo typeCheck(TypeEnv tenv) throws TypeException {
            return TypeInfo.INT;
        }
    }
    @Reviser public static class StringEx extends StringLiteral {
        public StringEx(Token t) { super(t); }
        public TypeInfo typeCheck(TypeEnv tenv) throws TypeException {
            return TypeInfo.STRING;
        }
    }
    @Reviser public static class NameEx2 extends EnvOptimizer.NameEx {
        protected TypeInfo type;
        public NameEx2(Token t) { super(t); }
        public TypeInfo typeCheck(TypeEnv tenv) throws TypeException {
            type = tenv.get(nest, index);
            if (type == null)
                throw new TypeException("undefined name: " + name(), this);
            else
                return type;
        }
        public TypeInfo typeCheckForAssign(TypeEnv tenv, TypeInfo valueType)
            throws TypeException
        {
            type = tenv.get(nest, index);
            if (type == null) {
                type = valueType;
                tenv.put(0, index, valueType);
                return valueType;
            }
            else {
                valueType.assertSubtypeOf(type, tenv, this);
                return type;
            }
        }
    }
}

```

```

    }
}
@Reviser public static class NegativeEx extends NegativeExpr {
    public NegativeEx(List<ASTree> c) { super(c); }
    public TypeInfo typeCheck(TypeEnv tenv) throws TypeException {
        TypeInfo t = ((ASTreeTypeEx)operand()).typeCheck(tenv);
        t.assertSubtypeOf(TypeInfo.INT, tenv, this);
        return TypeInfo.INT;
    }
}
@Reviser public static class BinaryEx extends BinaryExpr {
    protected TypeInfo leftType, rightType;
    public BinaryEx(List<ASTree> c) { super(c); }
    public TypeInfo typeCheck(TypeEnv tenv) throws TypeException {
        String op = operator();
        if ("=".equals(op))
            return typeCheckForAssign(tenv);
        else {
            leftType = ((ASTreeTypeEx)left()).typeCheck(tenv);
            rightType = ((ASTreeTypeEx)right()).typeCheck(tenv);
            if ("+".equals(op))
                return leftType.plus(rightType, tenv);
            else if ("==" .equals(op))
                return TypeInfo.INT;
            else {
                leftType.assertSubtypeOf(TypeInfo.INT, tenv, this);
                rightType.assertSubtypeOf(TypeInfo.INT, tenv, this);
                return TypeInfo.INT;
            }
        }
    }
    protected TypeInfo typeCheckForAssign(TypeEnv tenv)
        throws TypeException
    {
        rightType = ((ASTreeTypeEx)right()).typeCheck(tenv);
        ASTree le = left();
        if (le instanceof Name)
            return ((NameEx2)le).typeCheckForAssign(tenv, rightType);
        else
            throw new TypeException("bad assignment", this);
    }
}
@Reviser public static class BlockEx extends BlockStmnt {
    TypeInfo type;
    public BlockEx(List<ASTree> c) { super(c); }
    public TypeInfo typeCheck(TypeEnv tenv) throws TypeException {
        type = TypeInfo.INT;
        for (ASTree t: this)
            if (!(t instanceof NullStmnt))
                type = ((ASTreeTypeEx)t).typeCheck(tenv);
        return type;
    }
}

```

```

@Reviser public static class IfEx extends IfStmtnt {
    public IfEx(List<ASTree> c) { super(c); }
    public TypeInfo typeCheck(TypeEnv tenv) throws TypeException {
        TypeInfo condType = ((ASTreeTypeEx)condition()).typeCheck(tenv);
        condType.assertSubtypeOf(TypeInfo.INT, tenv, this);
        TypeInfo thenType = ((ASTreeTypeEx)thenBlock()).typeCheck(tenv);
        TypeInfo elseType;
        ASTree elseBk = elseBlock();
        if (elseBk == null)
            elseType = TypeInfo.INT;
        else
            elseType = ((ASTreeTypeEx)elseBk).typeCheck(tenv);
        return thenType.union(elseType, tenv);
    }
}

@Reviser public static class WhileEx extends WhileStmtnt {
    public WhileEx(List<ASTree> c) { super(c); }
    public TypeInfo typeCheck(TypeEnv tenv) throws TypeException {
        TypeInfo condType = ((ASTreeTypeEx)condition()).typeCheck(tenv);
        condType.assertSubtypeOf(TypeInfo.INT, tenv, this);
        TypeInfo bodyType = ((ASTreeTypeEx)body()).typeCheck(tenv);
        return bodyType.union(TypeInfo.INT, tenv);
    }
}

@Reviser public static class DefStmtntEx2 extends TypedEvaluator.DefStmtntEx {
    protected TypeInfo.FunctionType funcType;
    protected TypeEnv bodyEnv;
    public DefStmtntEx2(List<ASTree> c) { super(c); }
    public TypeInfo typeCheck(TypeEnv tenv) throws TypeException {
        TypeInfo[] params = ((ParamListEx2)parameters()).types();
        TypeInfo retType = TypeInfo.get(type());
        funcType = TypeInfo.function(retType, params);
        TypeInfo oldType = tenv.put(0, index, funcType);
        if (oldType != null)
            throw new TypeException("function redefinition: " + name(),
                                    this);
        bodyEnv = new TypeEnv(size, tenv);
        for (int i = 0; i < params.length; i++)
            bodyEnv.put(0, i, params[i]);
        TypeInfo bodyType
            = ((ASTreeTypeEx)revise(body())).typeCheck(bodyEnv);
        bodyType.assertSubtypeOf(retType, tenv, this);
        return funcType;
    }
}

@Reviser
public static class ParamListEx2 extends TypedEvaluator.ParamListEx {
    public ParamListEx2(List<ASTree> c) { super(c); }
    public TypeInfo[] types() throws TypeException {
        int s = size();
        TypeInfo[] result = new TypeInfo[s];
        for (int i = 0; i < s; i++)
            result[i] = TypeInfo.get(typeTag(i));
    }
}

```

```

        return result;
    }
}

@Reviser public static class PrimaryEx2 extends FuncEvaluator.PrimaryEx {
    public PrimaryEx2(List<ASTree> c) { super(c); }
    public TypeInfo typeCheck(TypeEnv tenv) throws TypeException {
        return typeCheck(tenv, 0);
    }
    public TypeInfo typeCheck(TypeEnv tenv, int nest) throws TypeException {
        if (hasPostfix(nest)) {
            TypeInfo target = typeCheck(tenv, nest + 1);
            return ((PostfixEx)postfix(nest)).typeCheck(tenv, target);
        }
        else
            return ((ASTreeTypeEx)operand()).typeCheck(tenv);
    }
}

@Reviser public static abstract class PostfixEx extends Postfix {
    public PostfixEx(List<ASTree> c) { super(c); }
    public abstract TypeInfo typeCheck(TypeEnv tenv, TypeInfo target)
        throws TypeException;
}

@Reviser public static class ArgumentsEx extends Arguments {
    protected TypeInfo[] argTypes;
    protected TypeInfo.FunctionType funcType;
    public ArgumentsEx(List<ASTree> c) { super(c); }
    public TypeInfo typeCheck(TypeEnv tenv, TypeInfo target)
        throws TypeException
    {
        if (!(target instanceof TypeInfo.FunctionType))
            throw new TypeException("bad function", this);
        funcType = (TypeInfo.FunctionType)target;
        TypeInfo[] params = funcType.parameterTypes;
        if (size() != params.length)
            throw new TypeException("bad number of arguments", this);
        argTypes = new TypeInfo[params.length];
        int num = 0;
        for (ASTree a: this) {
            TypeInfo t = argTypes[num] = ((ASTreeTypeEx)a).typeCheck(tenv);
            t.assertSubtypeOf(params[num++], tenv, this);
        }
        return funcType.returnType;
    }
}

@Reviser public static class VarStmntEx2 extends TypedEvaluator.VarStmntEx {
    protected TypeInfo varType, valueType;
    public VarStmntEx2(List<ASTree> c) { super(c); }
    public TypeInfo typeCheck(TypeEnv tenv) throws TypeException {
        if (tenv.get(0, index) != null)
            throw new TypeException("duplicate variable: " + name(), this);
        varType = TypeInfo.get(type());
        tenv.put(0, index, varType);
        valueType = ((ASTreeTypeEx)initializer()).typeCheck(tenv);
    }
}

```

```

        valueType.assertSubtypeOf(varType, tenv, this);
        return varType;
    }
}

```

代码清单 14.7 TypeEnv.java

```

package chap14;
import java.util.Arrays;
import stone.StoneException;

public class TypeEnv {
    protected TypeEnv outer;
    protected TypeInfo[] types;
    public TypeEnv() { this(8, null); }
    public TypeEnv(int size, TypeEnv out) {
        outer = out;
        types = new TypeInfo[size];
    }
    public TypeInfo get(int nest, int index) {
        if (nest == 0)
            if (index < types.length)
                return types[index];
            else
                return null;
        else if (outer == null)
            return null;
        else
            return outer.get(nest - 1, index);
    }
    public TypeInfo put(int nest, int index, TypeInfo value) {
        TypeInfo oldValue;
        if (nest == 0) {
            access(index);
            oldValue = types[index];
            types[index] = value;
            return oldValue; // may be null
        }
        else if (outer == null)
            throw new StoneException("no outer type environment");
        else
            return outer.put(nest - 1, index, value);
    }
    protected void access(int index) {
        if (index >= types.length) {
            int newLen = types.length * 2;
            if (index >= newLen)
                newLen = index + 1;
            types = Arrays.copyOf(types, newLen);
        }
    }
}

```

代码清单 14.8 TypeException.java

```
package chap14;
import stone.ast.ASTree;

public class TypeException extends Exception {
    public TypeException(String msg, ASTree t) {
        super(msg + " " + t.location());
    }
}
```

代码清单 14.9 TypeInfo.java

```
package chap14;
import stone.ast.ASTree;
import stone.ast.TypeTag;

public class TypeInfo {
    public static final TypeInfo ANY = new TypeInfo() {
        @Override public String toString() { return "Any"; }
    };
    public static final TypeInfo INT = new TypeInfo() {
        @Override public String toString() { return "Int"; }
    };
    public static final TypeInfo STRING = new TypeInfo() {
        @Override public String toString() { return "String"; }
    };

    public TypeInfo type() { return this; }
    public boolean match(TypeInfo obj) {
        return type() == obj.type();
    }
    public boolean subtypeOf(TypeInfo superType) {
        superType = superType.type();
        return type() == superType || superType == ANY;
    }
    public void assertSubtypeOf(TypeInfo type, TypeEnv env, ASTree where)
        throws TypeException
    {
        if (!subtypeOf(type))
            throw new TypeException("type mismatch: cannot convert from "
                                     + this + " to " + type, where);
    }
    public TypeInfo union(TypeInfo right, TypeEnv tenv) {
        if (match(right))
            return type();
        else
            return ANY;
    }
    public TypeInfo plus(TypeInfo right, TypeEnv tenv) {
        if (INT.match(this) && INT.match(right))
            return INT;
        else if (STRING.match(this) || STRING.match(right))
            return STRING;
        else

```

```

        return ANY;
    }
    public static TypeInfo get(TypeTag tag) throws TypeException {
        String tname = tag.type();
        if (INT.toString().equals(tname))
            return INT;
        else if (STRING.toString().equals(tname))
            return STRING;
        else if (ANY.toString().equals(tname))
            return ANY;
        else if (TypeTag.UNDEF.equals(tname))
            return new UnknownType();
        else
            throw new TypeException("unknown type " + tname, tag);
    }
    public static FunctionType function(TypeInfo ret, TypeInfo... params) {
        return new FunctionType(ret, params);
    }
    public boolean isFunctionType() { return false; }
    public FunctionType toFunctionType() { return null; }
    public boolean isUnknownType() { return false; }
    public UnknownType toUnknownType() { return null; }
    public static class UnknownType extends TypeInfo {
        @Override public TypeInfo type() { return ANY; }
        @Override public String toString() { return type().toString(); }
        @Override public boolean isUnknownType() { return true; }
        @Override public UnknownType toUnknownType() { return this; }
    }
    public static class FunctionType extends TypeInfo {
        public TypeInfo returnType;
        public TypeInfo[] parameterTypes;
        public FunctionType(TypeInfo ret, TypeInfo... params) {
            returnType = ret;
            parameterTypes = params;
        }
        @Override public boolean isFunctionType() { return true; }
        @Override public FunctionType toFunctionType() { return this; }
        @Override public boolean match(TypeInfo obj) {
            if (!(obj instanceof FunctionType))
                return false;
            FunctionType func = (FunctionType)obj;
            if (parameterTypes.length != func.parameterTypes.length)
                return false;
            for (int i = 0; i < parameterTypes.length; i++)
                if (!parameterTypes[i].match(func.parameterTypes[i]))
                    return false;
            return returnType.match(func.returnType);
        }
        @Override public String toString() {
            StringBuilder sb = new StringBuilder();
            if (parameterTypes.length == 0)
                sb.append("Unit");
            else
                for (int i = 0; i < parameterTypes.length; i++) {

```

```

        if (i > 0)
            sb.append(" * ");
            sb.append(parameterTypes[i]);
        }
        sb.append(" -> ").append(returnType);
        return sb.toString();
    }
}

```

14.3 运行程序时执行类型检查

至此，Stone 语言处理器已经能够支持类型检查，我们先来尝试一下在运行程序时执行类型检查。代码清单 14.10 是新的解释器。它与之前的解释器大同小异，唯一的区别在于会在调用 eval 方法执行输入的程序之前，通过 typeCheck 方法对数据类型执行检查。也就是说，该解释器会依次对输入的程序调用 lookup、typeCheck 与 eval 方法。此外，在最终显示 eval 方法的返回值时，它将同时显示 typeCheck 方法的返回值，即 eval 方法返回值的数据类型。

代码清单 14.11 是该解释器的启动程序。它应用了 TypeChecker 修改器（及其依赖的其他修改器）。为了对整个程序执行类型检查，它必须知道原生函数的数据类型信息。在第 8 章（第 8 天）已经说明过，原生函数指的是由 Java 语言写成的函数。由于原生函数没有以 Stone 语言定义，因此我们必须事先将它们的定义告诉启动程序。

在向环境添加原生函数时，我们还要向数据类型环境添加该原生函数的数据类型。代码清单 14.12 中的程序能实现这一处理。代码清单 14.10 中的解释器没有像之前那样使用第 8 章代码清单 8.3 中的 Natives 类，将使用代码清单 14.13 的程序。代码清单 14.13 ~ 代码清单 14.17 是新增的原生函数。与之前不同，这些原生函数各自具有单独的类，且函数本身都被命名为 m。这是为了之后将 Stone 语言的程序转换为 Java 二进制代码而做的准备。

H 老师，那些表示原生函数的类的名称都与原生函数的名称一样呢。

F 不过这些类名都是以小写字母起始的，没有遵循 Java 的命名习惯。

C 这些都是为了方便之后把程序转换为 Java 代码，别太在意。

上述代码实现了新的语言处理器。接下来，我们启动解释器，试着执行以下的程序。

```

def fact(n: Int): Int {
    if n > 1 { n * fact(n - 1) } else { 1 }
}
fact 5

```

执行结果如下所示。

```

=> fact : Int -> Int
=> 120 : Int

```

第 1 行定义了函数 `fact`，并表明它是一个接收 `Int` 类型参数，返回 `Int` 类型结果的函数。第 2 行以 5 为参数调用了 `fact`，得到一个 `Int` 类型的返回值 120。

- A** 话说回来，我们一直在讲类型检查，其实能够检查的就只有静态类型语言对吧？
- F** 动态类型语言不支持类型检查，因为在执行之前我们无法知道具体的数据类型呀。
- S** 嗯，动态类型语言只是不支持静态类型检查而已。在每次执行计算前它同样会检查操作数的数据类型。
- C** 例如，在 Ruby 中，如果要对字符串做减法运算，就会发生运行错误。这就好比在执行计算前对 `-` 运算符两侧的数据类型做了检验。
- S** 嗯，由于 Ruby 的减法运算通过调用相应的方法实现，因此实际检查的是左侧的对象是否含有用于减法运算的方法。
- C** 对，你说的没错。
- H** Java 也会在强制类型转换或数组赋值等操作时执行动态数据类型检查。
- C** 毕竟静态数据类型检查有其局限性。静态类型只能表示某个表达式的计算结果起码是某种类型而已。
- A** 在数组赋值操作中也会执行类型检查的吗？
- F** 嗯，举个例子。

```
String[] sa = new String[1];
Object[] oa = sa;           // OK
oa[0] = new Integer(0); // error!
```

变量 `oa` 指向的是一个 `String` 数组，所以不能用一个 `Integer` 对象为 `oa` 的元素赋值。

- C** 除了动态类型检查，我们还能让程序在第 2 行将 `sa` 赋值给 `oa` 时，通过静态数据类型检查发现错误。不过 Java 的设计者没有采用这种设计，因此数组赋值操作必须执行动态数据类型检查。
- F** 我觉得在将 `sa` 赋值给 `oa` 时应该引发静态数据类型错误才对。
- S** 这可不太好，要是 `java.util.Arrays.sort(Object[] a)` 不能接收 `String` 数组作为参数，使用起来会很不方便呢。

代码清单 14.10 TypedInterpreter.java

```
package chap14;
import stone.BasicParser;
import stone.CodeDialog;
import stone.Lexer;
import stone.Token;
import stone.TypedParser;
import stone.ParseException;
import stone.ast.ASTree;
import stone.ast.NullStmnt;
```

```

import chap11.EnvOptimizer;
import chap11.ResizableArrayEnv;
import chap6.BasicEvaluator;
import chap6.Environment;

public class TypedInterpreter {
    public static void main(String[] args) throws ParseException, TypeException {
        TypeEnv te = new TypeEnv();
        run(new TypedParser(),
            new TypedNatives(te).environment(new ResizableArrayEnv()),
            te);
    }
    public static void run(BasicParser bp, Environment env, TypeEnv typeEnv)
        throws ParseException, TypeException
    {
        Lexer lexer = new Lexer(new CodeDialog());
        while (lexer.peek(0) != Token.EOF) {
            ATree tree = bp.parse(lexer);
            if (!(tree instanceof NullStmnt)) {
                ((EnvOptimizer.ASTreeOptEx)tree).lookup(
                    ((EnvOptimizer.EnvEx2)env).symbols());
                TypeInfo type
                    = ((TypeChecker.ASTreeTypeEx)tree).typeCheck(typeEnv);
                Object r = ((BasicEvaluator.ASTreeEx)tree).eval(env);
                System.out.println("=> " + r + " : " + type);
            }
        }
    }
}

```

代码清单 14.11 TypedRunner.java

```

package chap14;
import javassist.gluonj.util.Loader;
import chap8.NativeEvaluator;

public class TypedRunner {
    public static void main(String[] args) throws Throwable {
        Loader.run(TypedInterpreter.class, args, TypeChecker.class,
            NativeEvaluator.class);
    }
}

```

代码清单 14.12 TypedNatives.java

```

package chap14;
import chap6.Environment;
import chap8.Natives;
import chap11.EnvOptimizer.EnvEx2;

public class TypedNatives extends Natives {
    protected TypeEnv typeEnv;
    public TypedNatives(TypeEnv te) { typeEnv = te; }
    protected void append(Environment env, String name, Class<?> clazz,

```

```

        String methodName, TypeInfo type, Class<?> ... params)
    {
        append(env, name, clazz, methodName, params);
        int index = ((EnvEx2)env).symbols().find(name);
        typeEnv.put(0, index, type);
    }
    protected void appendNatives(Environment env) {
        append(env, "print", chap14.java.print.class, "m",
            TypeInfo.function(TypeInfo.INT, TypeInfo.ANY),
            Object.class);
        append(env, "read", chap14.java.read.class, "m",
            TypeInfo.function(TypeInfo.STRING));
        append(env, "length", chap14.java.length.class, "m",
            TypeInfo.function(TypeInfo.INT, TypeInfo.STRING),
            String.class);
        append(env, "toInt", chap14.java.toInt.class, "m",
            TypeInfo.function(TypeInfo.INT, TypeInfo.ANY),
            Object.class);
        append(env, "currentTime", chap14.java.currentTime.class, "m",
            TypeInfo.function(TypeInfo.INT));
    }
}

```

代码清单 14.13 currentTime.java

```

package chap14.java;
import chap11.ArrayEnv;

public class currentTime {
    private static long startTime = System.currentTimeMillis();
    public static int m(ArrayEnv env) { return m(); }
    public static int m() {
        return (int)(System.currentTimeMillis() - startTime);
    }
}

```

代码清单 14.14 length.java

```

package chap14.java;
import chap11.ArrayEnv;

public class length {
    public static int m(ArrayEnv env, String s) { return m(s); }
    public static int m(String s) { return s.length(); }
}

```

代码清单 14.15 print.java

```

package chap14.java;
import chap11.ArrayEnv;

public class print {
    public static int m(ArrayEnv env, Object obj) { return m(obj); }
    public static int m(Object obj) {

```

```

        System.out.println(obj.toString());
        return 0;
    }
}

```

代码清单 14.16 read.java

```

package chap14.java;
import chap11.ArrayEnv;
import javax.swing.JOptionPane;

public class read {
    public static String m(ArrayEnv env) { return m(); }
    public static String m() {
        return JOptionPane.showInputDialog(null);
    }
}

```

代码清单 14.17 toInt.java

```

package chap14.java;
import chap11.ArrayEnv;

public class toInt {
    public static int m(ArrayEnv env, Object value) { return m(value); }
    public static int m(Object value) {
        if (value instanceof String)
            return Integer.parseInt((String)value);
        else if (value instanceof Integer)
            return ((Integer)value).intValue();
        else
            throw new NumberFormatException(value.toString());
    }
}

```

14.4 对类型省略的变量进行类型推论

在实现了类型检查之后，接下来我们将设计类型推论功能。之前，如果没有明确指定变量或参数的类型，我们将默认它们是 Any 类型。然而，Any 类型的值无法进行减法与乘法计算，非常不便。

- C** 还有一种做法是，如果没有指定数据类型，就以与之前采用动态数据类型的 Stone 语言同样的方式处理，变量可以被赋值为整数、字符串或其他任意类型的值，并能参与各种四则运算。
- H** 老师，这样的话，只要在执行四则运算前做下动态类型检查就行了对吧？
- C** 嗯，渐进类型指派 (gradual typing) 就是这么做的。无论如何，静态数据类型与动态数据类型将来都会进一步融合吧。

为此，如果没有明确指定数据类型，我们就需要调查该变量或参数的使用方式，推测恰当的数据类型。这就是类型推论 (type inference)。例如，如果某个变量出现在减法表达式的左侧或右侧，我们就能推测出它是一个 `Int` 类型的变量。之后，如果数据类型省略，我们将暂时把它记为 `Unknown` 类型 (类型不明的类型)，并通过类型推论确定它具体是什么类型。

A 如果某个变量出现在减法表达式的……真绕呀，直接说加法不好吗？

C 嗯，但是加法也可能是用于字符串连接运算哦。也就是说 `+` 运算符的两侧不一定非要是 `Int` 类型的值。

类型推论算法与类型检查算法大同小异。在执行类型检查时，语言处理器常需要确认 - 运算符左右两侧的子表达式是否都是 `Int` 类型，如果不是 `Int` 类型就会发生类型错误。不难想象，为了避免发生类型错误，我们需要将 `Unknown` 类型的子表达式视为 `Int` 类型处理。这样一来，最初是 `Unknown` 类型的值将随着类型检查的进行，逐渐被指定为具体的数据类型。

H 类型检查与类型推论之间还真是有着千丝万缕的联系呢。

C 没错。至少从实现上来看，类型检查与类型推论是同步进行的，类型推论就像是随着类型检查一起执行似的。

对于上面的减法表达式，我们可以很容易地推测出 `Unknown` 具体指代的类型，但是有时，要确定一个值的类型并非易事。例如，在下面的赋值表达式中，变量 `x` 与 `y` 都没有被指定类型。

```
x = y
```

这时，变量 `y` 要么与 `x` 的类型相同，要么是 `x` 类型的子类。然而，如果无法确定具体的类型，仅凭这些条件，我们无法推测出更加具体的结果。

因此，我们只能推迟类型推论处理，等待获取进一步的信息。我们需要暂时记录这条赋值表达式中包含的信息，通常，这些信息以方程式的形式表现。首先，对于没有明确指定且尚不能推测出数据类型的变量与参数，我们将以 t_x 、 t_y 等变量表示。于是，该赋值表达式包含的信息就与下面的式子等价。

```
 $t_y \leq t_x$ 
```

这里的 $<$ 表示子类关系。将数据类型信息以方程式的形式表现之后，类型推论的适用范围更广。例如，如果在执行类型检查时遇到形如 `x - 1` 的表达式，我们可以通过同样的思路，用下面的方程式表示其中包含的类型条件。

```
 $t_x = \text{Int}$ 
```

减法两侧必须都是 `Int` 类型的值。连立两个方程式可得 $t_y \leq \text{Int}$ ，又由于 `Int` 不含子类，因此可知 t_x 、 t_y 都是 `Int` 类型的值。

不难发现，类型推论的本质其实就是连立含有数据类型条件的方程式后求解。该连立方程式的解就是各个 Unknown 类型变量的具体数据类型。

A 方程式也不总是有解的对吧？

F 如果没有解就说明有数据类型错误了呀。也就是说，类型检查失败。

A 那含有多个解的情况呢？

有些方程式可能含有多个解，我们无法据此确定某个变量的具体类型。方便起见，对于这种情况，Stone 语言将把变量指定为 Any 类型。例如，下面的函数 id 将在接收参数 x 后直接返回 x 的值，我们设参数 x 的类型为 t_x ，函数 id 返回值的类型为 t_{ret} ，并连立方程式，解得 $t_x \leq t_{ret}$ 。

```
def id(x) { x }
```

满足该方程式的 t_x 、 t_y 类型组合不只一种。除了 Any、Any 外，Int、Int 及 Int、Any 也都符合条件。不过如上所述，Stone 语言将把 Any、Any 作为方程式的最终解。

H 老师，不增加对多态函数 (polymorphic function) 的支持吗？

F 那样的话，我们就能对上面函数 id 的类型下定义了。即，它是一种接收 α 类型的参数并返回 α 类型返回值的函数。

A α 类型？

F 是 α 类型变量。与 Java 中泛型 (generics) 的类型变量相同。

C 多态函数是一个很有意思的概念，不过它的类型检查与类型推论比较复杂，这里就不具体展开了。

代码清单 14.18 中的修改器将修改与类型推论处理相关的类。它修改了 TypeEnv 类、TypeInfo 类及其子类 UnknownType 类。在此之前，TypeInfo 类的 assertSubtypeOf 方法用于确认两种类型是否相同，或是否具有子类关系，如果不符合则抛出异常。修改后，如果遇到 Unknown 类型的值，该方法将为这两种类型建立方程式，并添加至数据类型环境 TypeEnv 对象中。新增的方程式本应可以表示两种类型一致或具有子类关系，不过简单起见，这里添加的方程式仅能表示两种类型相同。例如，如果赋值表达式需要将 t_1 类型的值赋值给 t_2 类型的变量，该方法将向数据类型环境添加方程式 $t_1 = t_2$ ，而非 $t_1 \leq t_2$ 。同样地，union 方法也做了类似的简化。

S 唉，也就是说，本来能够通过类型推论避免类型错误的程序，现在可能仍会发生类型错误是吗？

C 嗯，错是没错啦，不过如果要解不等式，程序就会变得很复杂了。

此外，由 + 运算符构成的双目运算表达式的类型指派规则也需要简化。按照之前的类型指派规则，如果运算符左右两侧都是 Int 类型，整个双目运算表达式也是 Int 类型，除此之外，无论两侧的值是什么类型，双目运算表达式的类型都需要视具体情况进一步分析决定。与该类型指

派规则对应的方程式非常复杂，因此我们需要简化该规则。如果无法确定 + 运算符左右两侧的数据类型，我们将始终推测它们是 Int 类型。

F 还真是简化了很多啊。

S 嗯，和 Standard ML 一样呢。

H 没关系，如果遇到问题，只要显式地指定数据类型就好了对吧，老师？

C 或者也可以像 OCaml 那样，通过 + 与 +. 等不同的运算符来区分数据类型。

要最终完成类型推论的实现，除了代码清单 14.18 中提供的，我们还需要再使用一个修改器。前面提到过，对于函数内部的局部变量或参数，如果仅凭函数内部类型推论，无法确定 Unknown 类型的具体结果，Stone 语言将默认采用 Any 类型。代码清单 14.19 中的修改器实现了这一逻辑。该修改器覆盖了 DefStmnt 类的 typeCheck 方法，在函数体的类型检查结束时，尚且无法确定具体数据类型的 Unknown 类型全都置为了 Any 类型。

H 老师，如果不使用代码清单 14.19 中的修改器，会有什么问题呢？

C 嗯，比如说，我们来看一下下面这段代码。

```
def id(x) { x }
print id(3)
print id("three")
```

其中，函数 id 本应接收 Any 类型的参数并返回 Any 类型的值，但从第 2 行起，它接收的是 Int 类型的值，返回的却是一个 Any 类型的值。

F 于是第 3 行就会发生类型错误了吧。

S 嗯，函数类型会中途改变，还真奇怪啊。

C 语言处理器在第 2 行调用 id 时将对 id(3) 执行类型检查，新增一个含有 id 的参数类型的方程式，并推测出参数是一个 Int 类型的值。

F 也就是说，如果没有代码清单 14.19 中的修改器，就会一直不停地做类型推论呢。

代码清单 14.20 是解释器的启动程序。为了支持类型推论功能，它在运行 Stone 语言解释器前将首先应用代码清单 14.18 与代码清单 14.19 中的修改器。代码清单 14.20 虽然没有显式地应用 InferTypes 修改器，但应用了 InferFuncTypes 修改器。由于 InferFuncTypes 修改器通过 @Require 隐式地应用了 InferTypes 等修改器，因此它们都会被一起应用于新的解释器。

代码清单 14.18 InferTypes.java

```
package chap14;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
```

```

import stone.ast.ASTree;
import javassist.gluonj.Reviser;
import chap14.TypeInfo.UnknownType;

@Reviser public class InferTypes {
    @Reviser public static class TypeInfoEx extends TypeInfo {
        @Override
        public void assertSubtypeOf(TypeInfo type, TypeEnv tenv, ASTree where)
            throws TypeException
        {
            if (type.isUnknownType())
                ((UnknownTypeEx)type.toUnknownType()).assertSupertypeOf(this,
                                                                           tenv, where);
            else
                super.assertSubtypeOf(type, tenv, where);
        }
        @Override public TypeInfo union(TypeInfo right, TypeEnv tenv) {
            if (right.isUnknownType())
                return right.union(this, tenv);
            else
                return super.union(right, tenv);
        }
        @Override public TypeInfo plus(TypeInfo right, TypeEnv tenv) {
            if (right.isUnknownType())
                return right.plus(this, tenv);
            else
                return super.plus(right, tenv);
        }
    }
    @Reviser public static class UnknownTypeEx extends TypeInfo.UnknownType {
        protected TypeInfo type = null;
        public boolean resolved() { return type != null; }
        public void setType(TypeInfo t) { type = t; }
        @Override public TypeInfo type() { return type == null ? ANY : type; }
        @Override public void assertSubtypeOf(TypeInfo t, TypeEnv tenv,
                                               ASTree where) throws TypeException
        {
            if (resolved())
                type.assertSubtypeOf(t, tenv, where);
            else
                ((TypeEnvEx)tenv).addEquation(this, t);
        }
        public void assertSupertypeOf(TypeInfo t, TypeEnv tenv, ASTree where)
            throws TypeException
        {
            if (resolved())
                t.assertSubtypeOf(type, tenv, where);
            else
                ((TypeEnvEx)tenv).addEquation(this, t);
        }
        @Override public TypeInfo union(TypeInfo right, TypeEnv tenv) {
            if (resolved())
                return type.union(right, tenv);

```

```

        else {
            ((TypeEnvEx)tenv).addEquation(this, right);
            return right;
        }
    }
    @Override public TypeInfo plus(TypeInfo right, TypeEnv tenv) {
        if (resolved())
            return type.plus(right, tenv);
        else {
            ((TypeEnvEx)tenv).addEquation(this, INT);
            return right.plus(INT, tenv);
        }
    }
}
@Reviser public static class TypeEnvEx extends TypeEnv {
    public static class Equation extends ArrayList<UnknownType> {}
    protected List<Equation> equations = new LinkedList<Equation>();
    public void addEquation(UnknownType t1, TypeInfo t2) {
        // assert t1.unknown() == true
        if (t2.isUnknownType())
            if (((UnknownTypeEx)t2.toUnknownType()).resolved())
                t2 = t2.type();
        Equation eq = find(t1);
        if (t2.isUnknownType())
            eq.add(t2.toUnknownType());
        else {
            for (UnknownType t: eq)
                ((UnknownTypeEx)t).setType(t2);
            equations.remove(eq);
        }
    }
    protected Equation find(UnknownType t) {
        for (Equation e: equations)
            if (e.contains(t))
                return e;
        Equation e = new Equation();
        e.add(t);
        equations.add(e);
        return e;
    }
}
}

```

代码清单 14.19 InferFuncTypes.java

```

package chap14;
import java.util.List;
import chap14.TypeInfo.FunctionType;
import chap14.TypeInfo.UnknownType;
import chap14.InferTypes.UnknownTypeEx;
import stone.ast.ASTree;
import javassist.gluonj.Require;
import javassist.gluonj.Reviser;

```

```

@Require({TypeChecker.class, InferTypes.class})
@Reviser public class InferFuncTypes {
    @Reviser public static class DefStmntEx3 extends TypeChecker.DefStmntEx2 {
        public DefStmntEx3(List<ASTree> c) { super(c); }
        @Override public TypeInfo typeCheck(TypeEnv tenv) throws TypeException {
            FunctionType func = super.typeCheck(tenv).toFunctionType();
            for (TypeInfo t: func.parameterTypes)
                fixUnknown(t);
            fixUnknown(func.returnType);
            return func;
        }
        protected void fixUnknown(TypeInfo t) {
            if (t.isUnknownType()) {
                UnknownType ut = t.toUnknownType();
                if (!((UnknownTypeEx)ut).resolved())
                    ((UnknownTypeEx)ut).setType(TypeInfo.ANY);
            }
        }
    }
}

```

代码清单 14.20 InferRunner.java

```

package chap14;
import javassist.gluonj.util.Loader;
import chap8.NativeEvaluator;

public class InferRunner {
    public static void main(String[] args) throws Throwable {
        Loader.run(TypedInterpreter.class, args, InferFuncTypes.class,
            NativeEvaluator.class);
    }
}

```

14.5 Java 二进制代码转换

获得了静态数据类型信息之后，我们终于可以考虑如何将抽象语法树转换成 Java 二进制代码了。Java 二进制代码是 Java 虚拟机使用的机器语言。如果直接从正面入手，我们可以像上一章讲的那样通过 `compile` 方法实现。不过，本章将利用现有的库，用一种不同的方式实现 Java 二进制代码的转换。

为了对 Java 二进制代码进行操作，本章采用了一种名为 Javassist 的库。该库能够在程序执行过程中创建并载入新的类，并调用其中的方法。由于新增方法的定义能以 Java 源代码的形式传递给 Javassist，因此我们无需在程序中生成 Java 二进制代码，非常方便。Javassist 能自动编译接收的源代码，并将其转换为二进制代码。

代码清单 14.21 中的程序能够通过 Javassist 来定义新的方法。JavaLoader 类的 `load` 方

法将在接收类名与方法的定义后，对方法进行定义，并生成二进制代码，最后载入 Java 虚拟机。load 方法的返回值是该类的 Class 对象。只要有了 Class 对象，我们就能利用第 8 章介绍的 Java 中的反射机制来执行新定义的方法。

F 每一个方法都需要定义一个新的类吗？

C 没错，毕竟对 Java 来说，一个类一旦被载入虚拟机，就无法再添加新的方法了。因此如果我们用 Stone 语言定义了新的函数，就不得不新定义一个 Java 的类。

JavaLoader 类中使用的 ClassPool 对象由 Javassist 提供，用于管理类名与二进制代码（类文件）的对应关系。它会同时管理类的路径，而且在必要时能够从文件系统中取得类文件，读取二进制代码。

我们可以通过调用 ClassPool 对象的 makeClass 方法，在 JavaLoader 类中定义新的类。该方法将创建一个没有任何类与字段的空类。该方法的返回值是一个 CtClass 对象，用于表示 Javassist 中的类。这里的 CtClass 是 compile-time 类的简称。该对象与 java.lang.Class 有些相像，且提供了一些类似的方法。我们可以通过 addMethod 方法向 CtClass 对象添加希望定义的方法。在调用 toClass 之后，该 CtClass 对象将被转换为二进制代码并载入虚拟机，以获取与之对应的 Class 对象。addMethod 方法需要接收一个 String 对象，作为方法的定义。

代码清单 14.21 JavaLoader.java

```
package chap14;
import stone.StoneException;
import javassist.CannotCompileException;
import javassist.ClassPool;
import javassist.CtClass;
import javassist.CtMethod;

public class JavaLoader {
    protected ClassLoader loader;
    protected ClassPool cpool;
    public JavaLoader() {
        cpool = new ClassPool(null);
        cpool.appendSystemPath();
        loader = new ClassLoader(this.getClass().getClassLoader()) {};
    }
    public Class<?> load(String className, String method) {
        // System.out.println(method);
        CtClass cc = cpool.makeClass(className);
        try {
            cc.addMethod(CtMethod.make(method, cc));
            return cc.toClass(loader, null);
        } catch (CannotCompileException e) {
            throw new StoneException(e.getMessage());
        }
    }
}
```

F CtClass 是 compile-time 的简称呀？其实应该算是 Load-time，叫 LtClass 不是更好嘛？

C 的确，我现在也觉得 Lt 更好。

F 还有就是，为什么 toClass 要接收一个类载入器呢？

H 你是说 ClassLoader 对象对吧？每个 JavaLoader 对象都有专门的类载入器，它们用于载入新定义的类。

C 如果替换成 JavaLoader，就能载入具有相同名称的类了。在 Java 中，对于两个名称相同但定义不同的类，只要它们的类载入器不同，就能同时在程序中使用。

出于与上一章相同的理由，我们将仅对抽象语法树中的函数部分进行 Java 二进制代码转换。非函数调用部分将照常通过 eval 方法执行。函数调用表达式由 Arguments 类表示，该类的 eval 方法用于执行已转换为 Java 二进制代码的函数。

代码清单 14.22 中 JavaFunction 类的对象用于表示函数。该类的对象能够保存已被转换为 Java 二进制代码的 Stone 语言函数信息，并同时在环境中记录这些信息及对应的函数名称。由于函数调用之外的代码仍将由 eval 方法执行，因此本章依然需要使用用于记录全局变量与函数的环境。这些环境将以第 11 章介绍的方式实现，语言处理器在执行过程中将通过编号而非名称来查找信息。另一方面，函数体已被转换为 Java 二进制代码，因此语言处理器将借助 Java 语言的参数与局部变量来记录函数参数及局部变量的值。这些值将不会被记录至环境中。

代码清单 14.22 JavaFunction.java

```
package chap14;
import stone.StoneException;
import chap7.Function;

public class JavaFunction extends Function {
    protected String className;
    protected Class<?> clazz;
    public JavaFunction(String name, String method, JavaLoader loader) {
        super(null, null, null);
        className = className(name);
        clazz = loader.load(className, method);
    }
    public static String className(String name) {
        return "chap14.java." + name;
    }
    public Object invoke(Object[] args) {
        try {
            return clazz.getDeclaredMethods()[0].invoke(null, args);
        } catch (Exception e) {
            throw new StoneException(e.getMessage());
        }
    }
}
```

A 所以说，最后该怎样把抽象语法树转换为 Java 二进制代码呢？

由于本章使用了 Javassist，因此与其说是将抽象语法树转换为 Java 二进制代码，不如说是将它转换为 Java 源代码。例如，假设我们定义了这样一个 Stone 语言函数 `fact`。

```
def fact(n) {
  if n < 2 { 1 } else { n * fact(n - 1) }
}
```

该函数将被转换为名为 `chap14.java.fact` 的类中的一个 `static` 方法，如下所示。

```
public static int m(chap11.ArrayEnv env, int v0){
  int res;
  if ((v0 < 2 ? 1 : 0) != 0) {
    res = 1;
  } else {
    res = (v0 * chap14.java.fact.m(env, (v0 - 1)));
  }
  return res;
}
```

`chap14.java.fact` 类仅包含一个方法 `m`。`m` 这个方法名并没有什么特殊含义。该方法第一个参数 `env` 是一个用于引用全局变量的环境，不过在该例中，`fact` 函数不需要使用这个环境。`m` 方法的第二个参数 `v0` 是 `fact` 函数的参数。在转换为 Java 语言的方法后，`if` 语句有一条稍显冗长的条件表达式。该表达式由 `fact` 函数的定义直接翻译，几乎没有改动。

C 如果想知道函数被转换成了怎样的 Java 语言方法，只要在代码清单 14.21 中 `JavaLoader` 类的 `load` 方法的前部输出参数 `method` 的值就能看到了。

F 哦，原来如此，这部分现在被注释掉了呢。

代码清单 14.23 中的修改器用于将函数转换为 Java 二进制代码。修改器起始处的 `ranslateExpr` 与 `returnZero` 是两个辅助方法，需由其他方法调用。`EnvEx3` 与 `ArrayEnvEx` 修改器将向环境中添加新的字段，并通过它们保存 `JavaLoader` 对象。此外，相应的访问器方法也将被添加，用于获取这些对象。

其他的修改器将分别修改抽象语法树的相应节点，为它们添加 `translate` 方法。该方法将在被调用后以调用了它的对象为根节点遍历子树，并在生成与该子树对应的源代码后返回。与 `eval` 方法及 `compile` 方法相同，该方法也会通过递归调用的形式实现语法树节点的遍历。

在 Java 语言的转换过程中，Stone 语言的 `Int`、`String` 与 `Any` 类型分别对应 Java 中的 `int`、`String` 与 `Object` 类型。由于 Stone 语言的语法与 Java 较为相近，所以转换逻辑并不复杂，只需为每个变量添加静态数据类型即可。不过，Javassist 内置的编译器不支持 Java5 引入

的 `autoboxing` 功能，因此我们必须显式地执行该处理。该处理由 `translateExpr` 方法实现。例如，假设有下面的表达式。

```
x = i + 3
```

其中变量 `x` 为 `Any` 类型，变量 `i` 为 `Int` 类型。将其转换为 Java 语言后，它们将分别是 `Object` 与 `int` 类型的值，于是，该表达式将把一个 `int` 类型的值赋值给一个 `Object` 类型的变量。对于下面的表达式，如果语言处理器支持 `autoboxing` 功能，转换得到的 Java 语言表达式就能直接运行。如果不支持，由于 `=` 左右两侧类型不同，语言处理器将发生数据类型错误。

```
x = i + 3
```

为此，我们必须像下面这样，在代码中显式地执行数据类型转换。

```
x = new Integer(i + 3)
```

此外，为了执行显式的类型转换，除了变量与参数，语言处理器还需要知道子表达式的静态数据类型。因此，我们需要在执行类型检查的同时，为抽象语法树的节点对象添加相关字段，以保存已知的子表达式类型。Java 语言的转换处理将根据需要使用这些信息。

一个变量名不但可以指向局部变量，还可以指向全局变量或函数。我们必须根据情况将它们转换为相应的 Java 语言代码。由 `NameEx3` 修改器向 `Name` 类添加的 `translate` 方法能够实现这一功能。如果变量名指向的是局部变量或参数，它们将被转换为形如 `v0`、`v1` 的名称。`v` 之后数字代表变量在环境中的保存位置的编号。该编号能够通过 `index` 字段获取。

A 咦，环境又不是通过数组实现的，保存位置的编号是指什么？好乱啊。

C 环境确实不是通过数组实现的……

S 我记得用于记录全局变量的环境的确是由数组实现的呀。

C 虽然函数中的环境不是由数组实现的，不过我们还是会通过编号来管理局部变量，而不使用它们的名称。编号的分配由第 11 章介绍的修改器完成，这种实现较为简单。

另一方面，全局变量的变量名在转换后将引用作为参数传递的环境 `env`。由于数据类型的转换较为麻烦，因此实际的处理将交由代码清单 14.24 中 `Runtime` 类的相应方法实现，这里的代码仅需调用该方法即可。对于函数名称，它们将在函数转换为 Java 二进制代码后，被同时转换为对应的 `static` 方法的名称，如下所示。

```
chap14.java.fact.m
```

其中，`chap14.java` 是包名，`m` 是方法名。类名则与原来的函数名相同。

在转换过程中，我们还需要对比较运算符多加注意。Stone 语言与 Java 的运行机制稍有不同。Stone 语言的 `==` 运算符能同时对整数与字符串进行比较，但在 Java 中，字符串比较必须通过调用 `equals` 方法实现。此外，在 Stone 语言中，所有种类的比较结果都是 `Int` 类型

的 0 或 1, 但 Java 则通过 `boolean` 值表示结果。因此, 我们在转换中使用了下面这样的三目运算表达式。

```
(v0 < 2 ? 1 : 0)
```

类似地, `+` 运算符的计算结果如果是 `Any` 类型, 转换逻辑也会相对复杂些。这两种运算符的计算由代码清单 14.24 中 `Runtime` 类的 `eq` 与 `plus` 方法实现, 语言处理器将调用相应的代码进行实际的处理。详细信息请读者参见代码清单 14.23 中的 `BinaryEx2` 修改器。

A 要分别处理这些细节差异还真不容易啊。从一开始就以 Java 为标准来设计 Stone 语言不好嘛。

我们还要小心处理代码块与 `if` 语句的转换。在 Stone 语言中, 代码块内最后一条语句(或表达式)的计算结果将被作为整个代码块的结果。但 Java 语言并非如此, 它需要通过 `return` 语句来确定返回结果。此外, 虽然下面的表达式在 Stone 语言中合法, 但它既没有执行赋值操作, 也没有进行方法调用, 因此无法作为独立的语句在 Java 语言中使用。

```
x + 1;
```

为此, 我们为整个函数准备了一个名为 `res` 的局部变量, 并将可能作为代码块最后一条语句的结果的值, 即函数的返回值, 保存在这一 `res` 变量中。事实上, `translate` 方法的参数表示的是当前正在转换的 Stone 语言函数的返回值类型。如果该参数为 `null`, 就意味着(调用了 `translate` 方法的)语句或代码块的计算结果不是函数的返回值, 而不必保存在变量 `res` 中。

在本章中, 我们禁止在 Stone 语言的代码块内单独使用形如 `x+1` 这样的既非赋值也没有调用行数的表达式语句, 除非它位于代码块的最后。`BlockEx2` 修改器的 `translateStmt` 方法将对此进行检查, 如发现违例, 则会抛出异常。

A 这还真是偷懒, 完全是怎么方便怎么改嘛。这样一来, Stone 语言的语法兼容性就会很差了吧。

C 在这种地方花太多心思也没用啊。其实我在其他一些地方也做了简化处理, 比如, 函数无法重新定义。如果要支持这个功能, Java 语言转换就会变得很复杂了。

H 哦, 所以前面才提到 Stone 语言不支持相互递归对吧?

F 知道了这些后, 我真是体会到了实用程序设计语言设计者们的辛苦呢。他们就不能那么简单地放弃兼容性了。

代码清单 14.23 中的修改器覆盖了 `DefStmt` 类与 `Arguments` 类中的 `eval` 方法, 改变了函数定义与函数调用的执行逻辑。在定义函数时, 语言处理器将首先调用 `DefStmt` 类中修改过的 `eval` 方法。该方法会调用 `translate` 方法, 将函数体转换为 Java 语言的方法, 并创建一个 `JavaFunction` 对象。该对象用于表示函数, 它将与函数名一起被保存至环境中。

另一方面，在调用最外层函数时，语言处理器将调用 `Arguments` 类中修改过的 `eval` 方法。它将计算实参的值，并调用 `JavaFunction` 对象的 `invoke` 方法，执行已被转换为 Java 二进制代码的函数。

代码清单 14.23 ToJava.java

```
package chap14;
import java.util.ArrayList;
import java.util.List;
import chap11.ArrayEnv;
import chap11.EnvOptimizer;
import chap6.Environment;
import chap7.FuncEvaluator;
import stone.StoneException;
import stone.Token;
import stone.ast.*;
import javassist.gluonj.Require;
import javassist.gluonj.Reviser;
import static javassist.gluonj.GluonJ.revise;

@Require(TypeChecker.class)
@Reviser public class ToJava {
    public static final String METHOD = "m";
    public static final String LOCAL = "v";
    public static final String ENV = "env";
    public static final String RESULT = "res";
    public static final String ENV_TYPE = "chap11.ArrayEnv";

    public static String translateExpr(ASTree ast, TypeInfo from, TypeInfo to) {
        return translateExpr(((ASTreeEx)ast).translate(null), from, to);
    }
    public static String translateExpr(String expr, TypeInfo from,
                                      TypeInfo to)
    {
        from = from.type();
        to = to.type();
        if (from == TypeInfo.INT) {
            if (to == TypeInfo.ANY)
                return "new Integer(" + expr + ")";
            else if (to == TypeInfo.STRING)
                return "Integer.toString(" + expr + ")";
        }
        else if (from == TypeInfo.ANY)
            if (to == TypeInfo.STRING)
                return expr + ".toString()";
            else if (to == TypeInfo.INT)
                return "(" + Integer.parseInt(expr) + ".intValue()";
        return expr;
    }
    public static String returnZero(TypeInfo to) {
        if (to.type() == TypeInfo.ANY)
            return RESULT + "=new Integer(0)";
    }
}
```

```

        else
            return RESULT + "=0;";
    }

    @Reviser public static interface EnvEx3 extends EnvOptimizer.EnvEx2 {
        JavaLoader javaLoader();
    }

    @Reviser public static class ArrayEnvEx extends ArrayEnv {
        public ArrayEnvEx(int size, Environment out) { super(size, out); }
        protected JavaLoader jloader = new JavaLoader();
        public JavaLoader javaLoader() { return jloader; }
    }

    @Reviser public static abstract class ASTreeEx extends ASTree {
        public String translate(TypeInfo result) { return ""; }
    }

    @Reviser public static class NumberEx extends NumberLiteral {
        public NumberEx(Token t) { super(t); }
        public String translate(TypeInfo result) {
            return Integer.toString(value());
        }
    }

    @Reviser public static class StringEx extends StringLiteral {
        public StringEx(Token t) { super(t); }
        public String translate(TypeInfo result) {
            StringBuilder code = new StringBuilder();
            String literal = value();
            code.append('');
            for (int i = 0; i < literal.length(); i++) {
                char c = literal.charAt(i);
                if (c == '')
                    code.append("\\\\");
                else if (c == '\\')
                    code.append("\\\\\\\\");
                else if (c == '\n')
                    code.append("\\\n");
                else
                    code.append(c);
            }
            code.append('');
            return code.toString();
        }
    }

    @Reviser public static class NameEx3 extends TypeChecker.NameEx2 {
        public NameEx3(Token t) { super(t); }
        public String translate(TypeInfo result) {
            if (type.isFunctionType())
                return JavaFunction.className(name()) + "." + METHOD;
            else if (nest == 0)
                return LOCAL + index;
            else {
                String expr = ENV + ".get(0," + index + ")";
                return translateExpr(expr, TypeInfo.ANY, type);
            }
        }
    }

```

```

    }
}
public String translateAssign(TypeInfo valueType, ASTree right) {
    if (nest == 0)
        return "(" + LOCAL + index + "="
            + translateExpr(right, valueType, type) + ")";
    else {
        String value = ((ASTreeEx)right).translate(null);
        return "chap14.Runtime.write" + type.toString()
            + "(" + ENV + "," + index + "," + value + ")";
    }
}
}
}
@Reviser public static class NegativeEx extends NegativeExpr {
    public NegativeEx(List<ASTree> c) { super(c); }
    public String translate(TypeInfo result) {
        return "-" + ((ASTreeEx)operand()).translate(null);
    }
}
@Reviser public static class BinaryEx2 extends TypeChecker.BinaryEx {
    public BinaryEx2(List<ASTree> c) { super(c); }
    public String translate(TypeInfo result) {
        String op = operator();
        if ("=".equals(op))
            return ((NameEx3)left()).translateAssign(rightType, right());
        else if (leftType.type() != TypeInfo.INT
            || rightType.type() != TypeInfo.INT) {
            String e1 = translateExpr(left(), leftType, TypeInfo.ANY);
            String e2 = translateExpr(right(), rightType, TypeInfo.ANY);
            if ("==".equals(op))
                return "chap14.Runtime.eq(" + e1 + "," + e2 + ")";
            else if ("+".equals(op)) {
                if (leftType.type() == TypeInfo.STRING
                    || rightType.type() == TypeInfo.STRING)
                    return e1 + "+" + e2;
                else
                    return "chap14.Runtime.plus(" + e1 + "," + e2 + ")";
            }
            else
                throw new StoneException("bad operator", this);
        }
        else {
            String expr = ((ASTreeEx)left()).translate(null) + op
                + ((ASTreeEx)right()).translate(null);
            if ("<".equals(op) || ">".equals(op) || "==" .equals(op))
                return "(" + expr + "?1:0)";
            else
                return "(" + expr + ")";
        }
    }
}
}
@Reviser public static class BlockEx2 extends TypeChecker.BlockEx {

```

```

public BlockEx2(List<ASTree> c) { super(c); }
public String translate(TypeInfo result) {
    ArrayList<ASTree> body = new ArrayList<ASTree>();
    for (ASTree t: this)
        if (!(t instanceof NullStmnt))
            body.add(t);
    StringBuilder code = new StringBuilder();
    if (result != null && body.size() < 1)
        code.append(returnZero(result));
    else
        for (int i = 0; i < body.size(); i++)
            translateStmnt(code, body.get(i), result,
                           i == body.size() - 1);
    return code.toString();
}
protected void translateStmnt(StringBuilder code, ASTree tree,
                              TypeInfo result, boolean last)
{
    if (isControlStmnt(tree))
        code.append(((ASTreeEx)tree).translate(last ? result : null));
    else
        if (last && result != null)
            code.append(RESULT).append('=')
                .append(translateExpr(tree, type, result)).append(";\\n");
        else if (isExprStmnt(tree))
            code.append(((ASTreeEx)tree).translate(null)).append(";\\n");
        else
            throw new StoneException("bad expression statement", this);
}
protected static boolean isExprStmnt(ASTree tree) {
    if (tree instanceof BinaryExpr)
        return "=".equals(((BinaryExpr)tree).operator());
    return tree instanceof PrimaryExpr || tree instanceof VarStmnt;
}
protected static boolean isControlStmnt(ASTree tree) {
    return tree instanceof BlockStmnt || tree instanceof IfStmnt
        || tree instanceof WhileStmnt;
}
}
@Reviser public static class IfEx extends IfStmnt {
    public IfEx(List<ASTree> c) { super(c); }
    public String translate(TypeInfo result) {
        StringBuilder code = new StringBuilder();
        code.append("if(");
        code.append(((ASTreeEx)condition()).translate(null));
        code.append("! = 0) {\\n");
        code.append(((ASTreeEx)thenBlock()).translate(result));
        code.append("} else {\\n");
        ASTree elseBk = elseBlock();
        if (elseBk != null)
            code.append(((ASTreeEx)elseBk).translate(result));
        else if (result != null)

```

```

        code.append(returnZero(result));
        return code.append("}\n").toString();
    }
}

@Reviser public static class WhileEx extends WhileStmnt {
    public WhileEx(List<ASTree> c) { super(c); }
    public String translate(TypeInfo result) {
        String code = "while(" + ((ASTreeEx)condition()).translate(null)
            + "!=0){\n" + ((ASTreeEx)body()).translate(result)
            + "}\n";
        if (result == null)
            return code;
        else
            return returnZero(result) + "\n" + code;
    }
}

@Reviser public static class DefStmntEx3 extends TypeChecker.DefStmntEx2 {
    public DefStmntEx3(List<ASTree> c) { super(c); }
    @Override public Object eval(Environment env) {
        String funcName = name();
        JavaFunction func = new JavaFunction(funcName, translate(null),
            ((EnvEx3)env).javaLoader());
        ((EnvEx3)env).putNew(funcName, func);
        return funcName;
    }
    public String translate(TypeInfo result) {
        StringBuilder code = new StringBuilder("public static ");
        TypeInfo returnType = funcType.returnType;
        code.append(javaType(returnType)).append(' ');
        code.append(METHOD).append(" (chap11.ArrayEnv ").append(ENV);
        for (int i = 0; i < funcType.parameterTypes.length; i++) {
            code.append(',').append(javaType(funcType.parameterTypes[i]))
                .append(' ').append(LOCAL).append(i);
        }
        code.append(")\n");
        code.append(javaType(returnType)).append(' ').append(RESULT)
            .append(";\n");
        for (int i = funcType.parameterTypes.length; i < size; i++) {
            TypeInfo t = bodyEnv.get(0, i);
            code.append(javaType(t)).append(' ').append(LOCAL).append(i);
            if (t.type() == TypeInfo.INT)
                code.append("=0;\n");
            else
                code.append("=null;\n");
        }
        code.append(((ASTreeEx)revise(body())).translate(returnType));
        code.append("return ").append(RESULT).append("};");
        return code.toString();
    }
    protected String javaType(TypeInfo t) {
        if (t.type() == TypeInfo.INT)
            return "int";
        else if (t.type() == TypeInfo.STRING)

```

```

        return "String";
    else
        return "Object";
    }
}

@Reviser public static class PrimaryEx2 extends FuncEvaluator.PrimaryEx {
    public PrimaryEx2(List<ASTree> c) { super(c); }
    public String translate(TypeInfo result) { return translate(0); }
    public String translate(int nest) {
        if (hasPostfix(nest)) {
            String expr = translate(nest + 1);
            return ((PostfixEx)postfix(nest)).translate(expr);
        }
        else
            return ((ASTreeEx)operand()).translate(null);
    }
}

@Reviser public static abstract class PostfixEx extends Postfix {
    public PostfixEx(List<ASTree> c) { super(c); }
    public abstract String translate(String expr);
}

@Reviser public static class ArgumentsEx extends TypeChecker.ArgumentsEx {
    public ArgumentsEx(List<ASTree> c) { super(c); }
    public String translate(String expr) {
        StringBuilder code = new StringBuilder(expr);
        code.append('(').append(ENV);
        for (int i = 0; i < size(); i++)
            code.append(',').append(translateExpr(child(i), argTypes[i],
                funcType.parameterTypes[i]));
        return code.append(')').toString();
    }
}

public Object eval(Environment env, Object value) {
    if (!(value instanceof JavaFunction))
        throw new StoneException("bad function", this);
    JavaFunction func = (JavaFunction)value;
    Object[] args = new Object[numChildren() + 1];
    args[0] = env;
    int num = 1;
    for (ASTree a: this)
        args[num++] = ((chap6.BasicEvaluator.ASTreeEx)a).eval(env);
    return func.invoke(args);
}

@Reviser public static class VarStmntEx3 extends TypeChecker.VarStmntEx2 {
    public VarStmntEx3(List<ASTree> c) { super(c); }
    public String translate(TypeInfo result) {
        return LOCAL + index + "="
            + translateExpr(initializer(), valueType, varType);
    }
}
}

```

代码清单 14.24 Runtime.java

```

package chap14;
import chap11.ArrayEnv;

public class Runtime {
    public static int eq(Object a, Object b) {
        if (a == null)
            return b == null ? 1 : 0;
        else
            return a.equals(b) ? 1 : 0;
    }
    public static Object plus(Object a, Object b) {
        if (a instanceof Integer && b instanceof Integer)
            return ((Integer)a).intValue() + ((Integer)b).intValue();
        else
            return a.toString().concat(b.toString());
    }
    public static int writeInt(ArrayEnv env, int index, int value) {
        env.put(0, index, value);
        return value;
    }
    public static String writeString(ArrayEnv env, int index, String value) {
        env.put(0, index, value);
        return value;
    }
    public static Object writeAny(ArrayEnv env, int index, Object value) {
        env.put(0, index, value);
        return value;
    }
}

```

14.6 综合所有修改再次运行程序

代码清单 14.25 是新版的启动程序。该启动程序在运行 Stone 语言时将同时执行类型检查、类型推论以及 Java 二进制代码的转换。虽然本章需要使用 Javassist 库，不过它已经包含在了 GlueJ 中，因此我们无需做额外的处理。

A 这次多少会快一些了吧。

F 第 8 章代码清单 8.6 中的 fib 函数在经过类型推论处理后，所有的变量都将具有数据类型。
(小心翼翼地定义了函数)

H 是嘛，所有变量都具有类型了啊。那么这就不需要添加 :Int 之类的声明了呢。

F 什么？还真能直接运行呀。

S 嗯，因为代码中只有 Int 与 String，非常简单，所以添加类型也很容易。

A 那么执行速度如何？试着计算下 fib 33 吧。(试着多次执行 fib 33)

- F** 在函数定义完成后有点慢啊。第一次要花 0.2 到 0.7 秒才行。不过之后每次执行时就只要 0.03 至 0.04 秒了。
- A** 比 Ruby 要快多了嘛。
- C** 不能这么说。不要仅凭斐波那契数的计算速度来判断一种语言的性能，这话我说了多少次了，可别忘了。
- H** 老师，第一次比较慢是因为 JIT 编译器的关系吗？
- C** Stone 语言处理器本身的 JIT 似乎很费时。不过如果是 Server 模式，第一次也会很快。0.03 秒的结果和直接用 Java 写成的斐波那契数计算程序差不多了。
- A** 话说用 C 语言来计算 fib 33 要花多少时间？
- C** 编译未经优化的话，gcc 需要 0.1 秒左右吧。
- A** 竟然比 C 语言还快？
- F** 不要仅凭斐波那契数的计算速度来判断一种语言的性能呀，A 君。
- C** 先不说这个，如果开启了 O2 编译优化，gcc 只要不到 0.5 毫秒就能完成计算了。
- H** 毫秒吗，有 100 倍的差距了呀。
- C** 斐波那契数是一个例外，所以差别会很大。不过要评论语言的性能是很难的。只是不断优化一个简单的程序的话，最后就不知道到底在测什么了。事实上，在 O4 优化下，gcc 只要 4 微妙就能算好了。

代码清单 14.25 JavaRunner.java

```
package chap14;
import javassist.gluonj.util.Loader;
import chap8.NativeEvaluator;

public class JavaRunner {
    public static void main(String[] args) throws Throwable {
        Loader.run(TypedInterpreter.class, args, ToJava.class,
            InferFuncTypes.class, NativeEvaluator.class);
    }
}
```

专栏5 Twitter



第 3 部分

解说篇（自习时间）

第
15
天

手工设计词法分析器

第
15
天

手工设计词法分析器

在第3天（第3章），我们借助正则表达式库实现了词法分析器。由于程序设计语言通常都需具备正则表达式库，因此这种方式不会产生不便。不过在本章中，我们将讨论一种不使用正则表达式库的词法分析器设计方法。具体来讲，我们将不再通过正则表达式库来实现字符串匹配，正则表达式的处理逻辑将全部手工完成。

15.1 修改自动机

首先，我们来看一个例子，试考虑下面的正则表达式。

```
[0-9]+
```

该正则表达式用于匹配整型字面量。它能匹配由0至9这些数字至少重复一次组成的字符串。元字符+表示至少重复一次，元字符*表示至少重复0次，含有元字符*的正则表达式有时可以借助元字符+简化表述。

```
[0-9][0-9]*
```

这条正则表达式将匹配由1个数字或1个数字后接若干数字构成的字符串。从结果上来看，这两种正则表达式将始终匹配相同的字符串。

为了设计能够执行这一正则表达式匹配的程序，我们首先需要创建与之等价的自动机（automaton，准确来讲，这是一种确定有限状态自动机）。

A 确定什么的……名字可真长啊。

C 还有其他很多种自动机，为了区分它们，只能用那么长的名字了。

自动机类似于一种极为简单的计算机。它的内部包含了一个仅能记录有限类型的值的内存，在接收新的输入后，新值将由输入值与当前值共同决定，并更新至内存中。自动机不支持包括四则运算或分支运算等在内的任何其他类型的运算。自动机程序实质是一张对应关系表，根据该表，我们能由输入值及当前内存值的组合，得到需要保存至内存中的新值。

图15.1中的自动机与正则表达式`[0-9][0-9]*`等价。图中，圆圈内的数字表示自动机内存记录的当前值。圆圈（或其中的数字）称为状态。图中的箭头表示自动机在某一状态下，如果收到箭头旁标识的输入，将转换至箭头指向的状态（这一过程称为转换）。也就是说，如果圆圈内的数字是内存的当前值，且箭头旁的数字是自动机接受的输入，那么内存的值将被更新为箭头

指向的圆圈内的数字。

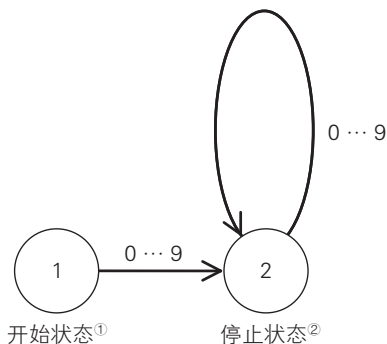


图 15.1 用于表示整型字面量的自动机

事实上，圆圈内的数字并没有特殊的含义。它们仅仅用于区分不同的圈圈。因此，有些图中圆圈内不写数字。该图最为根本的一点在于，圆圈的数量有限（因此称为有限状态自动机）。读者千万不能误以为圆圈的个数可以无限增加。

A 说是不能无限，那万一有的正则表达式只能与含有无限个圆圈的自动机等价的话，该怎么办呢？

F A君，你好好听课了吗？任何正则表达式都能转化为与之等价的有限状态机哦，也就是说自动机定能仅包含有限个圆圈，这可是常识呀。

字符串匹配的执行将从起始状态，即图中的状态 1 开始。自动机将从字符串头部开始逐一输入字符，并据此改变状态，最终抵达停止状态，即图中的状态 2。执行途中如果找不到符合要求的箭头就会出现错误，字符串匹配失败。例如，假设自动机在状态 1 下接受了输入 A，由于图中的箭头只有在接受数字输入时有效，因此本次匹配将以出错告终。

状态 2 是一个停止状态，其中，我们应当关注的是状态 2 上标有的箭头。可以看到，该自动机的状态 2 依然能够接受数字输入并继续执行。由于箭头从状态 2 出发又回到该状态，因此无论输入几次，自动机将始终处于状态 2。只要输入内容是数字，自动机的执行将不断循环。

要判断正则表达式是否与整个字符串匹配，程序需要检查字符串的最后一个字符输入后，自动机是否处于停止状态。如果没有到达停止状态，或中途出错，则表示字符串与正则表达式不匹配。

在词法分析过程中，语言处理器需要判断正则表达式与字符串头部的多少字符匹配。例如，假设有以下字符串。

37+41

① 亦有起始状态、初始状态等译法。——译者注

② 亦有接受状态、终止状态、结束状态等译法。——译者注

该字符串的前两个字符，即子字符串 37，与正则表达式 $[0-9][0-9]^*$ 匹配。为了得到这一结果，自动机必须不断运行，直至无法再接受新的输入，到达停止状态。无法再接受新的输入指的是，任何新的输入都会引起错误。对于上面的例子，自动机将接受 2 个字符，至数字 7 为止。此时，自动机处于状态 2，且已抵达停止状态，如果继续接受第 3 个字符 + 就会报错。该状态下，已经接受的字符将能与正则表达式匹配。由于这里最后接受的字符是 7，因此子字符串 37 能够与正则表达式匹配。

C 正则表达式被改为自动机后，似乎就能用程序实现了对吧？

H 毕竟自动机的执行方式多少和计算机有些相似呢。

A 不过，改起来难度不小吧？

F 老师，您要是不先讲下怎么把正则表达式改为非确定有限状态自动机，再介绍怎么把它进一步改写为确定有限状态机吧？一般的教材都是这么安排的。

C 如果允许任意的正则表达式，并改写与之相应的匹配程序的话，确实是要这样……

H 意思是我们要从头自己手工设计正则表达式库？

C 如果已知正则表达式，要直接把它转换成确定有限状态自动机也不是件难事。稍微想想就能做出来了。

H 那是因为老师您已经理解得很透彻了所以才能做到吧……

下面我们再看一个自动机的例子。这次我们尝试将一条更加复杂的正则表达式改写为自动机吧。

```
\s*([0-9][0-9]*|[A-Za-z][A-Za-z0-9]*|=|==)
```

它与第 3 章使用的正则表达式十分相似。在该正则表达式中，`\s*` 之后跟有由 `|` 连接的 4 种模式。其中，第 1 个模式用于与整形字符串匹配。第 2 个模式用于与标识符匹配。如果一个字符串的首个字符是 `A` 至 `z` 中的某个字母，且之后跟有若干个字母或数字，抑或不后接任何内容，它就会与该模式匹配。最后两个模式用于匹配 `=` 以及 `==`。由于 `\s` 表示空白符，因此最终该正则表达式匹配的字符串将由两部分组成。首先，空白符将重复出现多次或完全不出现，之后，再接有一个与四种模式中的某模式相匹配的字符串。

图 15.2 是与该正则表达式等价的自动机，它含有 5 种状态。其中，状态 1 是开始状态，其余都是停止状态。当自动机处于状态 1 时，它能根据输入的内容在 4 种模式中选择，并转换至相应的状态。如果输入的是空白符，自动机将保持状态 1，直至接受到与某种模式匹配的字符。

由于模式 `=` 与 `==` 的首字符相同，因此对于这两种模式，自动机都将转换至状态 4。如果下一个字符不是 `=`，匹配将就此结束。如果自动机继续接受了一个字符 `=`，就将转换至状态 5 并结束。状态 5 没有转换至其他状态的箭头，因此无论之后的输入是什么，匹配过程都会直接结束，不会进行判断。

15.2 自动机程序

如果能够将正则表达式改写为自动机，就不难将其进一步改写为用于执行基于原正则表达式的字符串匹配程序。代码清单 15.1 是根据图 15.2 中的自动机改写而来的程序。其中包含了用于测试的 main 方法。

在调用 Lexer 类的 read 方法后，词法分析器将从 Reader 对象中读取字符，并返回前端与正则表达式匹配的子字符串。不过，起始的空白符不会被返回。再次调用 read 方法时，词法分析器将从上次返回的字符串之后接着继续处理，返回与正则表达式匹配的字符串。

分析 read 方法的内部可以发现，它仅是自动机逻辑的一种直接表述。自动机中的箭头在程序中以 if 或 while 语句表示。如果箭头返回的是原有状态，则由 while 语句表示。如果正则表达式不是很复杂，这种级别的简单程序就足以表达它的匹配逻辑。

词法分析器通过 Lexer 类的 getChar 方法从 Reader 对象中逐一读取字符。ungetChar 方法将重置字符的读取状态，使词法分析器能够重新读取该字符。在重置了字符的读取状态后，词法分析器能再次使用 getChar 方法重新读取该字符。

词法分析器之所以需要支持这种操作，是因为它必须从读取的字符串前端获取与正则表达式匹配的尽可能长的部分。为此，正如之前所讲，词法分析器不得不持续运行，直至无法再接受新的输入字符，到达停止状态。此时，Reader 对象中可能还留有一些字符。由于词法分析器在继续读取字符时将发生错误，因此它必须能够通过 ungetChar 方法取消读取。此外，这些字符如果不是空白符，将在下次调用 read 方法时成为字符串的起始字符。

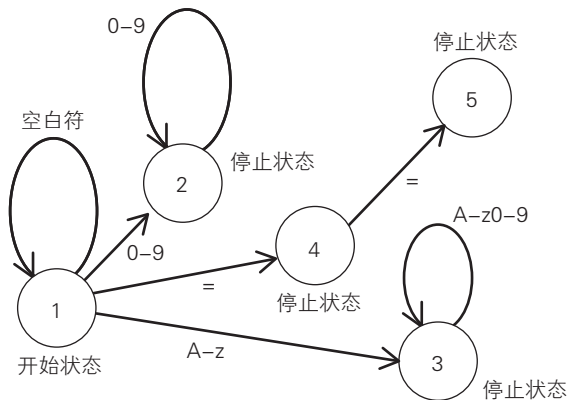


图 15.2 用于识别整型字面量、标识符与等号的自动机

代码清单 15.1 由图 15.2 中的自动机改写得到的程序

```

package chapA;
import java.io.IOException;
import java.io.Reader;
import stone.CodeDialog;

```

```

public class Lexer {
    private Reader reader;
    private static final int EMPTY = -1;
    private int lastChar = EMPTY;
    public Lexer(Reader r) { reader = r; }
    private int getChar() throws IOException {
        if (lastChar == EMPTY)
            return reader.read();
        else {
            int c = lastChar;
            lastChar = EMPTY;
            return c;
        }
    }
    private void ungetChar(int c) { lastChar = c; }
    public String read() throws IOException {
        StringBuilder sb = new StringBuilder();
        int c;
        do {
            c = getChar();
        } while (isSpace(c));
        if (c < 0)
            return null; // end of text
        else if (isDigit(c)) {
            do {
                sb.append((char)c);
                c = getChar();
            } while (isDigit(c));
        }
        else if (isLetter(c)) {
            do {
                sb.append((char)c);
                c = getChar();
            } while (isLetter(c) || isDigit(c));
        }
        else if (c == '=') {
            c = getChar();
            if (c == '=')
                return "==";
            else {
                ungetChar(c);
                return "=";
            }
        }
        else
            throw new IOException();

        if (c >= 0)
            ungetChar(c);

        return sb.toString();
    }
    private static boolean isLetter(int c) {
        return 'A' <= c && c <= 'Z' || 'a' <= c && c <= 'z';
    }
}

```

```

private static boolean isDigit(int c) { return '0' <= c && c <= '9'; }
private static boolean isSpace(int c) { return 0 <= c && c <= ' '; }
public static void main(String[] args) throws Exception {
    Lexer l = new Lexer(new CodeDialog());
    for (String s; (s = l.read()) != null; )
        System.out.println("-> " + s);
}
}

```

15.3 正则表达式的极限

正则表达式能够表述非常复杂的字符串模式匹配逻辑，但它并非万能。例如，我们来考虑一下与由 `/*` 与 `*/` 括起的注释语句匹配的正则表达式。乍一看，它可以通过正则表达式表述，但只要在这样的注释中还嵌套了由 `/*` 与 `*/` 标识的注释，正则表达式就无能为力了。

```
/* java /* ruby */ scala */
```

试考虑上述注释。该注释中嵌套了另一条注释 `/*ruby*/`。我们无法写出一条能够与整条注释匹配的正则表达式。正则表达式最多能匹配下面这样的注释，`scala*/` 将无法被识别。

```
/* java /* ruby */
```

- C** 其实上面的说法不太准确，含有嵌套的模式并非完全无法由正则表达式表述。如果嵌套只有有限的几层，正则表达式还是能表述的。但如果嵌套是无限的，就没办法了。
- A** 为什么会这样呢？
- F** 因为正则表达式一定能以有限状态自动机表示呀。也就是说，它只能处理有限的状态。因此能够处理有限层嵌套的正则表达式还是存在的，但能处理无限层嵌套的正则表达式从理论上就不可能实现。

第 3 部分

解说篇（自习时间）

第
16
天

语法分析方式

第
16
天

语法分析方式

第5章(第5天)通过 Parser 库轻松实现了语法分析逻辑。本章将介绍怎样才能不使用这类外部的库,手工设计语法分析程序。

16.1 正则表达式与BNF

上一章已经介绍了如何设计基于正则表达式的字符串匹配程序。由代码清单 15.2 的程序可知,该匹配程序比想象中简单得多。

如果由 BNF 定义的语法的复杂程度不高,基于该语法的语法分析程序同样会直观明了。事实上,正则表达式也能视作一种语法表述方式,它能够表达某些由 BNF 定义的语法。能够由 BNF 表述的语法称为上下文无关文法,而能通过正则表达式表述的语法称为正则文法。如果一种语法是正则文法,我们就能通过与上一章类似的方式,设计出语法分析程序。

C 顺便提一句,也有上下文有关文法这种语法。

H 嗯,那是怎样的语法来着?

C BNF 语法规则的左侧是一个非终结符。如果它的前后含有终结符,比如 `"/" pat "/"` 这样,该语法就是一种上下文有关文法。也就是说,只有被字符 `/` 前后括起时,非终结符 `pat` 才能扩展成这条语法规则右侧的模式。

正则表达式用于字符串的匹配,BNF 则用于单词序列的匹配。乍一看两者区别很大,其实,如果将字符串中的每个字符视作一个单词,就不难发现正则表达式与 BNF 的本质相同,都能用于模式匹配。例如,下面的正则表达式与由 BNF 定义的语法规则相同。

```
[0-9][0-9]*
digit: "0" | "1" | "2" | ... | "8" | "9"
number: digit | digit number
```

非终结符 `number` 的语法规则以递归形式定义,它表示非终结符 `digit` (0 至 9 的数字)将至少出现一次,本质与上面的正则表达式相同。如上所示,如果语法规则的右侧同时包含终结符与非终结符,且除了最右端是非终结符外,右侧的剩余部分都是终结符,该语法就是一种正则文法。也就是说,该语法能由正则表达式表述。

C 右侧的非终结符表示递归,因此这里的关键是语法规则必须在尾部递归。

H 也就是说能够循环展开的就是正则文法对吧?

我们顺便通过一个具体的例子来深入探讨一下 `number` 的定义。`number` 可以由一个 `digit` 构成，也能由 `digit` 后接 `number` 构成。

在第一种情况下，`number` 就是一个 `digit`。在第二种情况下，后接的 `number` 可以是一个 `digit`，也可以是一个 `digit` 再后接 `number`。

```
digit
digit number
```

于是，以 `digit number` 为例，对于第一种情况，加上原有的 `digit`，整个 `number` 由两个 `digit` 组成。对于第二种情况，最后的 `number` 既可以是一个 `digit`，也可以是 `digit` 再后接 `number`。由于这可以无限循环，因此最终，`number` 可以仅含有一个 `digit`，也可以由任意个连续的 `digit` 组成。

```
digit digit
digit digit number
```

16.2 语法分析算法

如果一种由 BNF 定义的语法不是正则文法，我们就不得不使用更加复杂的算法来实现语法分析。遗憾的是，绝大部分的程序设计语言语法都不是正则文法。只要语言支持在表达式中使用无限嵌套的括号，它就无法通过正则文法表述。

现已存在大量能够对非正则文法进行语法分析的算法。这些算法各有特色。其中一些算法较为简单，只能处理与正则文法区别不大的语法。另外一些算法非常复杂，但能够对各种不同类型的语言进行语法分析。

F 这部分理论非常深奥呢。

A 嗯，是呀。

F 你嗯个什么啊，编译课上不是讲过吗？

A 哎呀，当时我完全在梦游了。

常见的语法分析算法可以分为向上分析算法与向下分析算法两类。前者称为自底向上语法分析，后者称为自顶向下语法分析。向上分析算法将首先组合相邻单词创建子表达式，再组合这些子表达式，逐步构造出整体结构。LR 语法分析（Left-to-right, Rightmost derivation）是一种著名的自底向上分析算法。LR 语法分析非常强大，但基于 LR 算法的语法分析程序很难实现，因此，人们通常会使用自动工具将 BNF 语法转换为语法分析器。其中，`yacc` 是长久以来为人们所熟知的典型。不过需要注意的是，`yacc` 实现的其实是 LALR 语法分析，它的语法分析能力稍逊于 LR 语法分析。

A 老师，我们就直接用工具来生成吧，不用讲内部的原理了。

F A 君，把工具当黑盒直接使用可不好啊。

另一种向下分析语法将从整体结构开始向下分析。LL 语法分析 (Left-to-right, Leftmost derivation) 是其中的代表。LL 语法分析器程序较为简单，是一种充分利用了递归调用的递归下降语法分析器 (recursive-descent parser)。这种分析器也可称为基于递归的自顶向下语法分析器。本书讲解 LL 语法分析。然而遗憾的是，LL 语法分析算法并非万能，它无法处理所有 LR 语法分析能够胜任的情况。不过，如果程序设计语言的语法不是特别复杂，LL 语法分析就已足够。因此，如果不使用自动生成工具，人们常会采用 LL 语法分析来处理语法。

C 只要多预读几次，LL 语法分析的性能就能与 LR 语法分析不相上下，因此我选择讲解 LL 语法分析。之后还会详细介绍这里提到的预读机制。

F 这里说 LL 语法分析弱于 LR 语法分析的前提是只允许预读一次。也就是说，比较的对象是 LL(1) 与 LR(1) 的分析能力^①。

16.3 LL 语法分析

接下来，我们来分析一下支持 LL 语法分析的递归下降语法分析器。我们将以第 4 章代码清单 4.7 中的语法规则为例，设计一个能对四则运算表达式进行语法分析的程序。

为方便阅读，代码清单 16.1 重新列出了代码清单 4.7 中的语法规则。代码清单 16.2 是相应的程序，其中包含了用于测试的 main 方法。该程序使用了第 3 章 (第 3 天) 代码清单 3.3 中的 Lexer 类，作为语言处理器的词法分析器。

阅读程序后不难理解，代码清单 16.1 中的各条语法规则分别由 factor、term 与 expression 这 3 种方法实现。这些方法分别与 : 左侧的各个非终结符对应。它们将通过词法分析器读取一条

^① 东京工业大学的佐佐政孝老师为这一点添加了补充说明。

从这段对话来看，似乎 LL(k) 的分析能力弱于 LR(1)。然而，情况并没有那么简单。

严格来讲，能够通过 LL(k) 分析的语言，即 LL(k) 语言，只要方法得当，总能够转换为能够以 LR(1) 分析的语言。也就是说，假设有某种程序设计语言，如果它能通过 LL(k) 分析，我们只要稍微修改一下它的语法，就能使用 LR(1) 来分析该语言。这里所说的语法修改自然不能改变语言本来的语法含义。因此，如果一段程序符合 (或不符合) 某种语法，它同样也符合 (或不符合) 修改后的版本。

反之，能够由 LR(1) 分析的语言并不一定都能改写为可以通过 LL(k) 分析的版本。与 LR(1) 相比，LL(k) 分析适用性更强。

不过，这一切的前提都是语言需要经过改写。LL(k) 语法在被改写为能通过 LR(1) 分析的版本后，常常会变得难以理解。从易读性的观点来看，基于 LL(k) 分析的版本更加合适。

如果不允许转换语法，有些语法就只能通过 LL(k) 方式分析，而无法使用 LR(1) 分析。此时，最好选择 LL(k)。此外，尽管能由 LL(k) 分析的语法必定能通过 LR(k) 分析，但 LR(k) 分析的算法很难手工设计，LR(k) 语法分析器的生成工具也较为少见，LL(k) 分析依然有一定的优势。

这个问题较为复杂，读者如有兴趣，请参考其他相关书籍。

与非终结符对应的单词序列，并以抽象语法树的形式返回语法分析结果。

每一个方法的内部都与自动机十分类似，它们将根据规则对输入进行词法分析。首先，我们来分析一下各条语法规则表示的铁路图。第 4 章的图 4.6 是与代码清单 16.1 中语法规则对应的铁路图。自动机的箭头旁标注的是输入的类型，铁路图则把这些标在箭头指向的圆圈或矩形中，两者本质上十分相似。因此，基于两者实现的基本程序架构也大同小异，都将通过词法分析器获取单词，并以此确定箭头的前进方向。

与自动机不同的是，铁路图中的非终结符分别与不同的方法对应。在铁路图中沿箭头前进时，如果中途遇到非终结符（以矩形表示），程序将调用与之对应的方法，对该部分进行语法分析。

代码清单 16.1 四则运算表达式的语法规则（与代码清单 4.7 相同）

```
factor:      NUMBER | "(" expression ")"
term:       factor { ("*" | "/" ) factor }
expression: term { ("+" | "-") term }
```

代码清单 16.2 ExprParser.java

```
package chapB;
import java.util.Arrays;
import stone.*;
import stone.ast.*;

public class ExprParser {
    private Lexer lexer;

    public ExprParser(Lexer p) {
        lexer = p;
    }

    public ASTree expression() throws ParseException {
        ASTree left = term();
        while (isToken("+") || isToken("-")) {
            ASTLeaf op = new ASTLeaf(lexer.read());
            ASTree right = term();
            left = new BinaryExpr(Arrays.asList(left, op, right));
        }
        return left;
    }

    public ASTree term() throws ParseException {
        ASTree left = factor();
        while (isToken("*") || isToken("/")) {
            ASTLeaf op = new ASTLeaf(lexer.read());
            ASTree right = factor();
            left = new BinaryExpr(Arrays.asList(left, op, right));
        }
        return left;
    }

    public ASTree factor() throws ParseException {
        if (isToken("(")) {
            token("(");
```

```

        ASTree e = expression();
        token("");
        return e;
    }
    else {
        Token t = lexer.read();
        if (t.isNumber()) {
            NumberLiteral n = new NumberLiteral(t);
            return n;
        }
        else
            throw new ParseException(t);
    }
}

void token(String name) throws ParseException {
    Token t = lexer.read();
    if (!(t.isIdentifier() && name.equals(t.getText())))
        throw new ParseException(t);
}

boolean isToken(String name) throws ParseException {
    Token t = lexer.peek(0);
    return t.isIdentifier() && name.equals(t.getText());
}

public static void main(String[] args) throws ParseException {
    Lexer lexer = new Lexer(new CodeDialog());
    ExprParser p = new ExprParser(lexer);
    ASTree t = p.expression();
    System.out.println("=> " + t);
}
}

```

在本例中，`expression` 方法将在中途调用 `term` 方法，`term` 方法又会中途调用 `factor` 方法。这是由于表示各个非终结符的语法规则的右侧都含有其他的非终结符。`factor` 方法甚至可能会调用 `expression` 方法。这就是所谓的递归调用。

C 与正则表达式不同，BNF 支持递归定义，这也是它的一大特征。

A 为了让语法规则支持递归，每条规则都需要通过不同的方法来实现对吧？这我还是知道的。

C 真的明白了吗？反过来说，由于正则表达式不支持递归，因此我们不得不想方设法通过 `while` 语句来实现循环处理。

H 代码清单 16.2 的程序也通过 `while` 语句实现了对 BNF 中 `{}` 部分的循环处理。

C 嗯，`{}` 是 BNF 的一种元字符，它是从正则表达式中引入的。

程序在铁路图中遇到箭头分支时，将调用 `isToken` 方法来分析下一个单词，确定箭头的走向。例如，`expression` 方法将通过 `isToken` 方法判断非终结符 `term` 之后跟的是否是 `+` 或 `-`。如果接受的输入是 `+` 或 `-`，根据语法规则，之后应该跟有非终结符 `term`，因此程序将进一步调用 `term` 方法。如果不是，处理将就此结束。`isToken` 方法将调用 `Lexer` 类的 `peek` 方法来查

找之后的单词。请读者注意，由于该方法只是预读单词，因此在 `isToken` 方法确定了前进路线后，程序还需要再次调用 `lexer` 的 `read` 方法执行实际的读取。不过，`factor` 方法不需要直接调用 `read` 方法，它能通过 `token` 方法间接完成调用。

C 词法分析器的基本输出形式是数据流。请大家注意 `read` 与 `peek` 的区别。3.3 节也提到过这点。

如果在与语法对应的铁路图中遇到箭头分支，为使 LL 语法分析顺利执行，程序必须能够仅凭下一个单词就确定分支的选择。如果一种语法无法达到这一要求，就不能由 LL 语法分析执行解析。查看下一个单词的内容的操作称为预读。

在代码清单 16.2 中，`peek` 方法能够预读下一个单词，确定箭头的走向。对于有些语法规则，语法分析器不能仅凭下一个单词就确定箭头的走向。这时，上述方法无法完成语法分析。不过，如果预读多个单词后能够确定结果，语法分析器只需预读必需的单词即可完成分析。这类 LL 语法分析称为 LL(k)。为方便区分，至多预读一次的 LL 语法分析称为 LL(1)。LL(k) 算法自然会比 LL(1) 更为强大，能够分析更加复杂的语法。

H 什么时候需要预读 2、3 个单词来着？

C 看一下下面的语法吧。

表达式：`"(" 标识符 ")" 表达式 | "(" 表达式 ")" | 项 { "+" 项 }`

F 根据该定义，表达式可以是类型转换表达式、括号括起的表达式，或是加法运算表达式等，没错吧？

H 也就是说，在预读得到的单词是左括号时，程序必须进一步预读，才能知道当前表达式究竟是一条类型转换表达式还是一条由括号括起的表达式对吗？

C 假设有程序 (`shape`)。你们觉得这是类型转换表达式还是由括号括起的表达式？

F 如果 `shape` 是变量，这就是一个由括号括起的表达式。不过词法分析器可不知道这些。

A 在这个例子里，不进一步预读的话可不知道具体怎么处理呀。

C 说的对。只预读左右括号括起的内容还不够，还要继续预读单词。如果下一个单词是一个标识符，比如 (`shape`) `s`，那这段程序就是一条类型转换表达式。

H 如果程序是 (`shape`) `+` `3`，即右括号之后是一个 `+` 号，那 `shape` 就是一个变量名，(`shape`) 则是一条括号表达式。

C 没错。但如果 `shape` 是一个 `double` 类型的名称，(`shape`) `+` `3` 就该是类型转换表达式了。

A 这么说，无论预读多少都没用了咯。说到底，这里的语法规则设计得有问题。

C 正是如此。由于这里的语法定义有歧义，因此无法通过 LL 语法分析处理。

A 那该怎么办才好？

C 暂且不必深究，类型转换也好，括号表达式也好，都行。先创建语法树，在完成语法分析后再做修正就行了吧。

对于有些语法，语法分析器无论预读多少个单词也不能确定箭头的走向。如果预读了一个单词后仍有多种可选走向，语法分析器就无法做出正确选择。这时，我们只需让语法分析器任意选择走向，如果之后走不通（发生语法错误），返回（回溯）至分支处选择其他路线即可。不过，这已经超越了 LL 语法分析的范畴。不断的回溯也会影响语法分析速度。为了避免这种情况，在设计程序设计语言时，我们应尽可能将预读次数限制在一次以内。如果语法需要多次预读才能分析，它的定义通常比较模棱两可，我们应该先考虑修改这些定义。

C 我们也可以用其他方法来解决这个问题。在词法分析器中进一步细分单词的类型后，语法中的歧义就有可能自动消失。例如，如果下一个单词是一个标识符，只要词法分析器能明确它是变量名、类型名还是保留字，语法歧义就会减少。

F 不过，能够做到这些的词法分析器还算是词法分析器吗？

H 在完成分析前不停循环回溯的 LL 语法分析其实是一种 packrat 语法分析对吧？

S 嗯，packrat 语法分析会先将数据存于内存（即 memoization 处理），不会循环回溯同一段逻辑，因此分析性能依然是线性的，并没有在不停地费力回溯哦。

F 只要在分析过程中经过一次，该箭头及之后的处理结果都将全部记录在案。因此同一条分支无需反复计算判断。

S 嗯，如果在源代码中的位置及箭头与之前相同，就不必重新执行一遍了。

A 啊，真糟糕，感觉要占用大量内存。

C 现在内存容量不断增加，这种分析方式也有了实用价值。

F 这种算法虽然很耗内存，但分析能力很强。即使单词是由单个字符构成的，执行速度也相当快哦。下面这样的语法定义也完全没问题。

```
def_keyword : "d" "e" "f"
```

H 如果单词都是由单个字符构成的，LL(1) 就无能为力了。因此这是一个很重要的优势。

C 如果一个字符就能构成一个单词，我们就不需要词法分析器了。觉得哪种情况更好啊？

A 如果可以设计词法分析器，肯定是不设计比较好咯。

H 如果语言可以根据上下文改变标识符的含义就好了。

C AspectJ 之类的能够实现这个要求。AspectJ 提供了切入点的概念。假设有一个切入点 `foo*bar`，由于其中含有通配符 `*`，因此 `foo*bar` 将被整个识别为独立的标识符。对于其他情况，`*` 都用于表示乘号。也就是说它被分解为了 `foo`、`*` 与 `bar` 这三个单词。

LL 语法分析需要根据预读结果决定之后的箭头走向。如果不习惯这种算法，可能会有些难以理解。为了理解 LL 语法分析算法，我们可以列举所有符合语法规则、有可能接着出现的单词。由于各条语法规则间可能相互依赖，因此我们在列举可能的情况时必须多加小心。如果箭头一端是非终结符，我们还要考虑该非终结符之后连接的会是什么单词。

在设计语法分析器程序时还可能出现下面这样的错误。明明仅靠一次预读无法判断箭头的走

向，却误以为能够确定分支路线，最终导致语法分析出错。因此，我们必须仔细检查，避免出现这种问题。

16.4 算符优先分析法与自底向上语法分析

LL 语法分析是一种典型的自顶向下语法分析算法。LR 语法分析则是一种具有代表性的自底向上语法分析算法。由于 LR 语法分析较为复杂，我们将介绍另一种名为算符优先分析法的算法。这种算法非常简单，甚至已经脱离了自底向上语法分析的范畴。

顾名思义，算符优先分析法（operator precedence parsing）是一种基于运算符优先级的语法分析算法。它是 LR(1) 的弱化版本，类似于一种至多预读一次的 LR 语法分析算法，因此只能对数学运算之类的简单语句执行语法分析，而不会被单独用于通常的程序设计语言语法分析。

不过，由于这种算法非常适合对数学算式执行语法分析，因此基于其他方法的语法分析器常常会结合使用算符优先分析法，专门对表达式执行语法分析。算符优先分析法不仅能用于自底向上语法分析器，也能嵌入自顶向下语法分析器中使用。事实上，本书使用的 Parser 库也在一定程度上利用了算符优先分析法。

接下来，我们看一下如何在 LL 语法分析这种自顶向下分析法中结合使用算符优先分析法。算符优先分析法其实能同时对双目运算符及括号等字符执行语法分析，不过我们在此将仅用算符优先分析法来处理双目运算符。其他诸如 factor（因子）等成分将交由 LL 语法分析执行。

A 为什么要费这么大劲儿使用算符优先分析法呀？

F 肯定还是为了给我们讲解它的原理吧？

C 不，我这么决定是有现实原因的。

将算符优先分析法与 LL 语法分析，尤其是递归下降语法分析器结合后，双目运算表达式的分析程序将变得非常简单。代码清单 16.1 中含有两种用于处理双目运算的非终结符，其中非终结符 `expression` 表示加减法算式，`term` 表示乘除法算式。这种设计是为了区分运算符之间的优先级。

通常，抽象语法树的节点与语法规则中的非终结符对应。为了表现运算符之间的优先级，与含有较高优先级运算符的表达式对应的节点将位于更接近抽象语法树叶节点的位置。反之，如果节点对应的表达式中运算符优先级较低，该节点就会更靠近根节点。为了实现这一机制，语法规则将使各个运算符两侧的表达式都具有比该运算符更高的优先级。对于本例来说，运算符 `+`、`-` 的两侧是 `term`。`term` 表达式中含有运算符 `*` 与 `/`，具有更高的优先级。

因此，随着具有不同优先级的运算符的增加，语法规则也会相应变得复杂。递归下降语法分析器将为每一条语法规则定义一个方法，于是语法分析器中将同时存在大量相似的方法。例如，代码清单 16.3 除了四则运算，还包含了其他运算符的语法规则。语法规则的数量与优先级的数量一致。因此，语法分析器中也含有相同数量的类似方法。

只要语法分析器结合使用了算符优先分析法，就能避免这样的问题。我们来看一个例子。代码清单 16.4 改写了代码清单 16.2 中的递归下降语法分析器的程序，将双目运算符处理逻辑替换为了算符优先分析法，并添加了用于测试的 main 方法。

代码清单 16.3 含有不同优先级运算符的表达式

```
factor:  NUMBER | "(" expression ")"
term:   factor { ("*" | "/" ) factor }
add_expr: term { ("+" | "-" ) term }
rel_expr: add_expr { ("<" | ">" ) add_expr }
eq_expr: rel_expr { ("==" | "!=" ) rel_expr }
and_expr: eq_expr { "&&" eq_expr }
or_expr:  and_expr { "||" and_expr }
```

代码清单 16.4 使用了算符优先分析法的语法分析器 (OpPrecedenceParser.java)

```
package chapB;
import java.util.Arrays;
import java.util.HashMap;
import stone.*;
import stone.ast.*;

public class OpPrecedenceParser {
    private Lexer lexer;
    protected HashMap<String,Precedence> operators;

    public static class Precedence {
        int value;
        boolean leftAssoc; // left associative
        public Precedence(int v, boolean a) {
            value = v; leftAssoc = a;
        }
    }

    public OpPrecedenceParser(Lexer p) {
        lexer = p;
        operators = new HashMap<String,Precedence>();
        operators.put("<", new Precedence(1, true));
        operators.put(">", new Precedence(1, true));
        operators.put("+", new Precedence(2, true));
        operators.put("-", new Precedence(2, true));
        operators.put("*", new Precedence(3, true));
        operators.put("/", new Precedence(3, true));
        operators.put("^", new Precedence(4, false));
    }

    public ASTree expression() throws ParseException {
        ASTree right = factor();
        Precedence next;
        while ((next = nextOperator()) != null)
            right = doShift(right, next.value);

        return right;
    }

    private ASTree doShift(ASTree left, int prec) throws ParseException {
```

```

        ASTLeaf op = new ASTLeaf(lexer.read());
        ASTree right = factor();
        Precedence next;
        while ((next = nextOperator()) != null && rightIsExpr(prec, next))
            right = doShift(right, next.value);

        return new BinaryExpr(Arrays.asList(left, op, right));
    }
    private Precedence nextOperator() throws ParseException {
        Token t = lexer.peek(0);
        if (t.isIdentifier())
            return operators.get(t.getText());
        else
            return null;
    }
    private static boolean rightIsExpr(int prec, Precedence nextPrec) {
        if (nextPrec.leftAssoc)
            return prec < nextPrec.value;
        else
            return prec <= nextPrec.value;
    }
    public ASTree factor() throws ParseException {
        if (isToken("(")) {
            token("(");
            ASTree e = expression();
            token(")");
            return e;
        }
        else {
            Token t = lexer.read();
            if (t.isNumber()) {
                NumberLiteral n = new NumberLiteral(t);
                return n;
            }
            else
                throw new ParseException(t);
        }
    }
    void token(String name) throws ParseException {
        Token t = lexer.read();
        if (!(t.isIdentifier() && name.equals(t.getText())))
            throw new ParseException(t);
    }
    boolean isToken(String name) throws ParseException {
        Token t = lexer.peek(0);
        return t.isIdentifier() && name.equals(t.getText());
    }
    public static void main(String[] args) throws ParseException {
        Lexer lexer = new Lexer(new CodeDialog());
        OpPrecedenceParser p = new OpPrecedenceParser(lexer);
        ASTree t = p.expression();
        System.out.println("=> " + t);
    }
}

```

代码清单 16.2 与代码清单 16.4 的主要区别在于 `expression` 方法。代码清单 16.4 中的 `expression` 方法使用了 `doShift`、`nextOperator` 及 `rightIsExpr` 这三个辅助方法，并且移除了代码清单 16.2 中含有的 `term` 方法。`factor` 方法、`token` 方法及 `isToken` 方法则没有变化。

由于代码清单 16.4 中的语法分析器使用了算符优先分析法，因此我们能很容易地为语法规则添加双目运算符。在添加新的运算符时，我们无需新增 `term` 之类的方法，只需将运算符添加至 `operators` 字段即可。这里的 `operators` 字段是一张运算符表。事实上，代码清单 16.4 中根本就没有 `term` 方法。

运算符表由哈希表实现。其中，键名是运算符的名称，与之对应的值是一个 `Precedence` 对象。`Precedence` 类的构造函数的第一个参数用于接收运算符的优先级。优先级由一个大于零的整数表示，数值越大，优先级就越高。第二个参数用于标识运算符采用的是左结合还是右结合。如果该值为 `true`，就表示这是一个左结合的运算符。

A 所以说，算符优先分析法具体是怎样执行的呢？

代码清单 16.5 四则运算表达式的语法规则（递归定义）

```
factor:    NUMBER | "(" expression ")"
term:      factor | term ("*" | "/" ) factor
expression: term | expression ("+" | "-") term
```

`expression` 方法的执行方式如下。现假设对于代码清单 16.5 所示的语法规则，我们有以下单词序列。

```
NUMBER "+" NUMBER "*" NUMBER
```

这也是一条由 BNF 终结符组成的序列。虽然上面的语法规则包含递归，但它的实际含义与代码清单 16.1 所示的语法规则相同。

在对这条单词序列执行语法分析时，我们必须能够识别左起第 2 个 `NUMBER` 究竟是其左侧 `+` 运算符的右操作数，还是右侧 `*` 运算符的左操作数。算符优先分析法将通过比较左右两个运算符的优先级来解决这个问题。代码清单 16.4 中，`doShift` 方法内 `while` 语句的条件表达式用于执行这一判断。如果 `rightIsExpr` 方法的返回值为 `true`，中间的 `NUMBER` 就是右侧运算符的左操作数。

在上例中，`*` 运算符的优先级较高，因此第二个 `NUMBER` 是其右侧 `*` 的左操作数。整个乘法表达式是 `+` 运算符的右操作数。

自底向上分析算法将相邻单词合并为子表达式，再将相邻的子表达式合并为表达式，逐步构造完整的结构。中途创建的子表达式能在语法规则中找到对应的非终结符。如果单词或子表达式序列与某个非终结符的模式匹配，序列就将被合并，作为与该非终结符对应的子表达式。在上例中，单词及子表达式的合并过程如下所示。

```

NUMBER "+" NUMBER "*" NUMBER
factor "+" factor "*" factor
term "+" term "*" factor
expression "+" term
expression

```

例如，由于单词 NUMBER 与非终结符 factor 的模式匹配，因此 NUMBER 能够被视为与 factor 对应的子表达式。为了表现这种对应关系，第 2 行用 factor 替换了上一行中出现的所有 NUMBER。类似地，第 4 行将第 3 行右侧的三个单词替换为了一个 term。同时，第 3 行左侧的 term 被替换为了 expression。之所以能够这样替换，是因为原来的单词与相应的非终结符匹配。如上例所示，在语法分析器将所有单词最终合并为一个非终结符后，语法分析结束。

查找与模式相匹配的单词序列，并将其替换为非终结符的操作称为归约 (reduce)。除非语法特别简单，否则语法分析器不应在执行分析时，没有章法地任意寻找与模式匹配的序列。例如，在上面的例子中，从第 3 行起也能采用以下归约方式。

```

term "+" term "*" factor
expression "+" term "*" factor
expression          "*" factor

```

如果采用这种方式，归约将提前在中途结束。没有模式能与最后一行的单词序列匹配。之所以出现这种结果，是由于该方式首先归约了 + 运算符及其两侧的三个单词。由这次归约创建的抽象语法树中，+ 运算符的优先级高于 * 运算符，逻辑整个错了。我们在制定语法规则时，应事先考虑到这一问题，使归约操作在遇到这类语法分析不当的情况时无法顺利完成。

F 我们也可以设计一种语法分析算法，用蛮力法把所有可能的归约情况都尝试一遍，再找出其中完成了所有归约的方法。

C 没错。不过计算机科学存在的意义就是为了找出更加合理高效的解决方法呀。

大部分语法分析算法都试图从前往后逐一读取单词序列，并依次执行归约。这些算法不会从后往前，或一次读取全部单词。因此，每读取一个新的单词后，只可能有三种结果：该单词符合某种模式的最后一部分，单词序列将归约为该模式；该单词是某种模式的起始部分；该单词处于某种模式的中段。其中，后两种结果称为移进 (shift)。

在上例中，语法分析器在读取左起第二个单词 NUMBER 后，立即将其归约为了 factor 与 term。之后，新得到的 term 有两种可能的合并方式。它既可以与左侧的 + 运算符共同作为非终结符 expression 模式的末尾部分进行归约，也能与右侧的 * 运算符一起充当非终结符 term 模式的起始部分进行移进。

算符优先分析法将在读取单词后比较其前后运算符的优先级，并以此决定执行归约还是移进操作。由于比较方式较为简单，因此这种算法只适用于对数学算式执行语法分析，难以应对其他类型的表达式。同为自底向上方式的 LR 语法分析采用了更复杂的方式来选择操作，因而能对

更复杂的语法规则进行分析。

在上例中，`*` 运算符的优先级更高，因此语法分析器在读取左起第二个单词 `NUMBER` 后应当执行移进操作。只需递归调用代码清单 16.4 中的 `doShift` 方法就能实现移进。语法分析器原本正在匹配非终结符 `expression` 的模式，此时该匹配将暂时中断，以执行方法的递归调用。`doShift` 方法的递归调用结束后（`*` 运算符被归约为 `term` 后），匹配将重新开始。在代码清单 16.4 中，`doShift` 方法在执行完成后将返回原调用处，这即是归约操作。

F 说到蛮力法，前面提到的 packrat 语法分析就是使用了蛮力法呢。

C 嗯，话是没错，不过那种算法只要遇到了一种可能的路径就会结束查找了。

A 可能的路径？

H 就是铁路图中箭头的轨迹。可能的路径就是中途不会中断，能够不发生语法错误一路走到最后的路径。

C 如果语法有歧义，就可能存在多条可能路径。第一条符合条件的路径将成为分析的结果。

A 这样也可以吗？！

S 嗯，就算有歧义也很难发现不是吗？不过 LR 语法分析能识别 `shift/reduce conflict` 之类的错误。

F 但要调试 `shift/reduce conflict` 可让人头大了。

H F 君，比起直接发布含有歧义的语法，多花点力气调试并执行类型检查才更好呀。

C 如果使用 Packrat 语法分析，我们可以通过 PEG (Parsing Expression Grammar) 来定义语法，明确规定该以怎样的顺序来尝试通过哪些路径。这样一来，就算存在多条可能的路径，我们也能明确知道首先找到的将是哪一条。

F 哦，所以没有用 `|` 而是用了 `/` 呢。

C 不过使用 PEG 之后，虽然语法不再有歧义，但由 PEG 定义的语法是否符合最初的设计意图就不得而知了。这需要我们另外考虑。

专栏6 英勇事迹



第 3 部分

解说篇（自习时间）

第
17
天

Parser 库的内部结构

第
17
天

Parser 库的内部结构

本书在设计语法分析器程序时利用了 Parser 库。第 5 章（第 5 天）已经介绍了该库的基本用法，但没有讲解它的内部结构。本章将通过 Parser 库的源代码，讲解它的内部结构。

在设计 LL 语法分析程序时，我们常会遇到类似的代码模式。Parser 库将这些反复出现的相似代码打包成库，语法分析器只需按需调用库的方法即可执行关键的语法分析操作。这是一种极为简单的解析器组合子（parser combinator）库。

17.1 组合子分析

组合子（combinator）是一个由组合子逻辑（combinatory logic）衍生而来的概念。组合子是一种高阶函数，它将接收若干函数作为参数，并返回它们的组合。此外，组合子不含自由变量（组合子的定义中只能使用参数）。

Y-combinator（fixed-point combinator）是一种著名的组合子，用于表述递归计算。由于组合子无法在定义中使用指代自身的（自由变量）名称，因此无法显式地表述递归调用。Y-combinator 通过某种形式表现了递归逻辑，是一种非常引人注目的组合子。

组合子分析原本是 Haskell 这类函数型语言中的一种程序设计技巧。将多个能对简单语法执行语法分析（parsing）的函数进行组合后，我们就能获得一个能够对更复杂的语法进行分析的新的函数。我们将这种能够执行简单语法分析的函数作为参数，将其组合后返回能够执行复杂语法分析的函数称为解析器组合子。

H 老师，解析器组合子和 Y-combinator 什么的有关系吗？

C 先不管它与跟函数型语言有多少关系，毕竟 Parser 库是用于 Java 语言的，内部逻辑肯定大幅简化了呀，并不高深。

A 说到底，就是在拿些玄妙难懂的概念糊弄我们嘛。

由 Parser 库实现的 Java 版组合子分析非常简单。它将组合多个能够执行简单语法分析的对象，并以此新建一个能够执行较复杂语法分析的对象。本章组合的并非函数，而是对象。库提供了一些基本的类，帮助实现用于执行语法分析的对象。库的使用者只要创建必需的对象，并将其组合，就能得到所需的语法分析器。

17.2 解析器组合子的内部

本章末尾的代码清单 17.1 是 Parser 库的源程序。该库与其他一些复杂的库不同，源程序

的规模很小，主要的类就只有一个 `Parser` 类。不过，在阅读源码后我们不难发现，`Parser` 类还含有 `Element` 类等不少嵌套子类（图 17.1）。

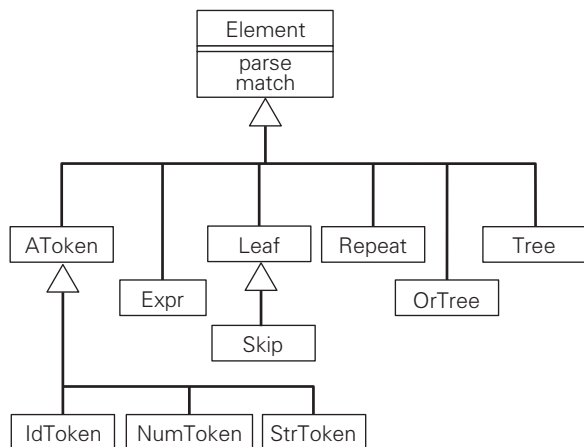


图 17.1 Parser 类中包含的嵌套子类

这些嵌套子类用于表现 `Parser` 对象需要处理的语法规则模式。语言处理器在将语法规则转换为 `Parser` 对象时，会调用 `number` 或 `ast` 等方法来构造模式。这些方法将创建 `Parser` 类中嵌套子类的对象，并将它们添加至 `Parser` 对象的 `elements` 字段指向的 `ArrayList` 对象中。例如，`ast` 方法将创建一个 `Tree` 对象，并将其添加至 `ArrayList` 中。

`Parser` 类的 `parse` 方法将根据构造的模式执行语法分析。它主要采用 LL 语法分析方式，并根据需要，部分使用了算符优先分析法。

`Parser` 类中嵌套子类的对象将被添加至 `elements` 字段指向的 `ArrayList` 对象中。这些新增的对象共同表现了铁路图的线路关系。铁路图是一种语法规则的表述方式，已在之前的第 4 章（第 4 天）中与 BNF 一同介绍过。

例如，图 17.2 通过 `Parser` 类及其嵌套子类的对象来表现第 4 章图 4.6 中铁路图的局部。对象的构造程序如下所示。

```
Parser factor = rule().or(rule().sep("(").ast(expr).sep(")"), rule().number());
```

不难看出，语言处理器构造的对象之间的关系与铁路图颇为相似。铁路图中箭头分支能够由 `OrTree` 对象表示。左括号、表达式、右括号组成的序列分别能由 `elements` 字段中的元素 `Skip`、`Tree` 与 `Skip` 这三个对象表示。

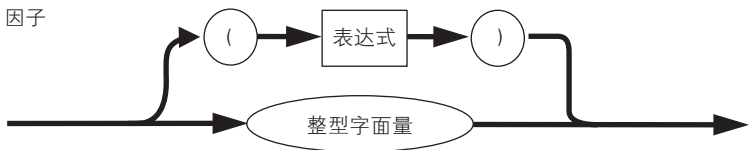
`Parser` 类的 `parse` 方法能够遍历这种形式的铁路图的同时执行语法分析。这就如同词法分析器能够一边遍历自动机，一边执行词法分析。`parse` 方法采用的是 LL 语法分析方式。上一章提到过，LL 语法分析的执行方式与遍历自动机实现的词法分析非常相似。

`parse` 方法将从词法分析器接收单词，遍历铁路图中的箭头，确认整条路径是否能够走通。

如果不能，则会发生语法错误，如果能走通，表示语法分析成功，于是 `parse` 方法将创建并返回相应的抽象语法树。我们能够通过反复调用 `Parser` 类的嵌套子类的 `parse` 方法，来判断箭头路径是否被正确遍历。词法分析器提供的单词最终由嵌套子类的 `parse` 方法接收，于是我们可以借此判断，这些子类自身表示的终结符或非终结符是否与接收的单词匹配。如果不匹配，就说明存在语法错误，`parse` 方法将抛出异常。

嵌套子类提供的 `parse` 方法通常只能完成极为简单的语法分析。但 `Expr` 类的 `parse` 是个例外，它能执行较复杂的分析。该类的对象由 `Parser` 类的 `expression` 方法添加。`Expr` 的 `parse` 方法将通过算符优先分析法对表达式执行语法分析。不过，由于因子部分将由其他嵌套子类对象的 `parse` 方法完成，因此 `Expr` 类复杂的分析并不算太复杂。

铁路图



Parser 库

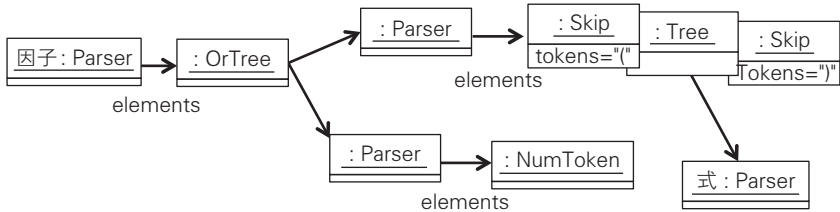


图 17.2 通过 `Parser` 对象来表现铁路图

- H** 总之，嵌套子类的对象是执行语法分析的基本组件，我们将组合这些对象来实现希望的语法分析对吧？
- F** 每一个组件都只能执行非常简单的分析。
- C** 于是，语法分析最终变为了一种单调的处理，语法分析器只需逐一确认接收的单词是否符合语法规则即可。
- F** 还要根据这一确认操作构造抽象语法树。
- C** 要这么说的话，不断接收的单词可能存在多种可能的匹配情况，而这也是确认操作要解决的问题，可没有想象中那么容易。

在 `Parser` 类的嵌套子类提供的 `parse` 方法中，最重要的是 `OrTree` 类的 `parse` 方法。`Parser` 类的 `or` 方法将使用 `OrTree` 来实现或运算逻辑。它与铁路图中的箭头分支对应，

不过该方法需要确定箭头的走向。

如有必要，LL 语法分析将预读单词，以确定箭头的走向。也就是说，语法分析器将事先调查之后将接收什么单词，并选择能够成功执行语法分析（不发生语法错误）的路径。该操作由 Parser 类及其嵌套子类的 match 方法执行。

通过 or 方法的参数传递的 Parser 对象将被 OrTree 对象接收，作为可用的分支选项。OrTree 对象的 parse 方法在调用后，将首先依次调用与各个分支选项对应的 Parser 对象的 match 方法。match 方法将预读一个单词，并将读取的单词与自身表示的模式的头部进行匹配，返回匹配结果。如果 match 方法返回真，OrTree 对象就会选择该 Parser 对象，并调用该对象的 parse 方法，继续执行语法分析。match 方法在遇到第一个符合条件的 Parser 对象后就会停止匹配，不再检查之后的选项。

F 这跟上一章最后提到的 PEG 很像嘛。

由于 OrTree 类的 parse 方法采用了这种方式来选择 Parser 对象，因此 Parser 类的 or 方法的执行逻辑与 BNF 中 |（或关系）稍有差异。于是，在通过 Parser 库执行语法分析时存在一些限制条件。需要分析的语法必须能够仅通过一次单词预读，就完全确定分支选项的选择。也就是说，并不是所有由 BNF 表述的语法定义都能通过这种方式完成语法分析。此外，如果同时存在多种选项，每个选项头部的终结符必须不同。不仅是 or 方法，repeat 及 option 方法也必须遵循这一规定。用于表示循环范围的语法规则必须以终结符起始，且该终结符不能与循环结束后遇到的终结符相同。这样一来，语法分析器只需预读一个单词，就能确定是否应该继续执行循环。

A 感觉最后最后讲了 Parser 库的一些关键限制呢。

C 毕竟超出限制的话就麻烦了。

S 嗯，Scala 的解析器组合子可是会采用更强大的方式来选择分支。

C 话是没错……不过 Stone 语言用这种程度的方法就足够了。

代码清单 17.1 Parser.java

```
package stone;

import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.ArrayList;
import java.lang.reflect.Method;
import java.lang.reflect.Constructor;
import stone.ast.ASTree;
import stone.ast.ASTLeaf;
import stone.ast.ASTList;

public class Parser {
```

```

protected static abstract class Element {
    protected abstract void parse(Lexer lexer, List<ASTree> res)
        throws ParseException;
    protected abstract boolean match(Lexer lexer) throws ParseException;
}

protected static class Tree extends Element {
    protected Parser parser;
    protected Tree(Parser p) { parser = p; }
    protected void parse(Lexer lexer, List<ASTree> res)
        throws ParseException
    {
        res.add(parser.parse(lexer));
    }
    protected boolean match(Lexer lexer) throws ParseException {
        return parser.match(lexer);
    }
}

protected static class OrTree extends Element {
    protected Parser[] parsers;
    protected OrTree(Parser[] p) { parsers = p; }
    protected void parse(Lexer lexer, List<ASTree> res)
        throws ParseException
    {
        Parser p = choose(lexer);
        if (p == null)
            throw new ParseException(lexer.peek(0));
        else
            res.add(p.parse(lexer));
    }
    protected boolean match(Lexer lexer) throws ParseException {
        return choose(lexer) != null;
    }
    protected Parser choose(Lexer lexer) throws ParseException {
        for (Parser p: parsers)
            if (p.match(lexer))
                return p;

        return null;
    }
    protected void insert(Parser p) {
        Parser[] newParsers = new Parser[parsers.length + 1];
        newParsers[0] = p;
        System.arraycopy(parsers, 0, newParsers, 1, parsers.length);
        parsers = newParsers;
    }
}

protected static class Repeat extends Element {
    protected Parser parser;
    protected boolean onlyOnce;
    protected Repeat(Parser p, boolean once) { parser = p; onlyOnce = once; }
    protected void parse(Lexer lexer, List<ASTree> res)
        throws ParseException

```

```

    {
        while (parser.match(lexer)) {
            ASTree t = parser.parse(lexer);
            if (t.getClass() != ASTList.class || t.numChildren() > 0)
                res.add(t);
            if (onlyOnce)
                break;
        }
    }
    protected boolean match(Lexer lexer) throws ParseException {
        return parser.match(lexer);
    }
}

protected static abstract class AToken extends Element {
    protected Factory factory;
    protected AToken(Class<? extends ASTLeaf> type) {
        if (type == null)
            type = ASTLeaf.class;
        factory = Factory.get(type, Token.class);
    }
    protected void parse(Lexer lexer, List<ASTree> res)
        throws ParseException
    {
        Token t = lexer.read();
        if (test(t)) {
            ASTree leaf = factory.make(t);
            res.add(leaf);
        }
        else
            throw new ParseException(t);
    }
    protected boolean match(Lexer lexer) throws ParseException {
        return test(lexer.peek(0));
    }
    protected abstract boolean test(Token t);
}

protected static class IdToken extends AToken {
    HashSet<String> reserved;
    protected IdToken(Class<? extends ASTLeaf> type, HashSet<String> r) {
        super(type);
        reserved = r != null ? r : new HashSet<String>();
    }
    protected boolean test(Token t) {
        return t.isIdentifier() && !reserved.contains(t.getText());
    }
}

protected static class NumToken extends AToken {
    protected NumToken(Class<? extends ASTLeaf> type) { super(type); }
    protected boolean test(Token t) { return t.isNumber(); }
}

protected static class StrToken extends AToken {

```

```

        protected StrToken(Class<? extends ASTLeaf> type) { super(type); }
        protected boolean test(Token t) { return t.isString(); }
    }

    protected static class Leaf extends Element {
        protected String[] tokens;
        protected Leaf(String[] pat) { tokens = pat; }
        protected void parse(Lexer lexer, List<ASTree> res)
            throws ParseException
        {
            Token t = lexer.read();
            if (t.isIdentifier())
                for (String token: tokens)
                    if (token.equals(t.getText())) {
                        find(res, t);
                        return;
                    }

            if (tokens.length > 0)
                throw new ParseException(tokens[0] + " expected.", t);
            else
                throw new ParseException(t);
        }
        protected void find(List<ASTree> res, Token t) {
            res.add(new ASTLeaf(t));
        }
        protected boolean match(Lexer lexer) throws ParseException {
            Token t = lexer.peek(0);
            if (t.isIdentifier())
                for (String token: tokens)
                    if (token.equals(t.getText()))
                        return true;

            return false;
        }
    }

    protected static class Skip extends Leaf {
        protected Skip(String[] t) { super(t); }
        protected void find(List<ASTree> res, Token t) {}
    }

    public static class Precedence {
        int value;
        boolean leftAssoc; // left associative
        public Precedence(int v, boolean a) {
            value = v; leftAssoc = a;
        }
    }

    public static class Operators extends HashMap<String, Precedence> {
        public static boolean LEFT = true;
        public static boolean RIGHT = false;
        public void add(String name, int prec, boolean leftAssoc) {
            put(name, new Precedence(prec, leftAssoc));
        }
    }

```

```

    }

    protected static class Expr extends Element {
        protected Factory factory;
        protected Operators ops;
        protected Parser factor;
        protected Expr(Class<? extends ASTree> clazz, Parser exp,
            Operators map)
        {
            factory = Factory.getForASTList(clazz);
            ops = map;
            factor = exp;
        }

        public void parse(Lexer lexer, List<ASTree> res) throws ParseException {
            ASTree right = factor.parse(lexer);
            Precedence prec;
            while ((prec = nextOperator(lexer)) != null)
                right = doShift(lexer, right, prec.value);

            res.add(right);
        }

        private ASTree doShift(Lexer lexer, ASTree left, int prec)
            throws ParseException
        {
            ArrayList<ASTree> list = new ArrayList<ASTree>();
            list.add(left);
            list.add(new ASTLeaf(lexer.read()));
            ASTree right = factor.parse(lexer);
            Precedence next;
            while ((next = nextOperator(lexer)) != null
                && rightIsExpr(prec, next))
                right = doShift(lexer, right, next.value);

            list.add(right);
            return factory.make(list);
        }

        private Precedence nextOperator(Lexer lexer) throws ParseException {
            Token t = lexer.peek(0);
            if (t.isIdentifier())
                return ops.get(t.getText());
            else
                return null;
        }

        private static boolean rightIsExpr(int prec, Precedence nextPrec) {
            if (nextPrec.leftAssoc)
                return prec < nextPrec.value;
            else
                return prec <= nextPrec.value;
        }

        protected boolean match(Lexer lexer) throws ParseException {
            return factor.match(lexer);
        }
    }

    public static final String factoryName = "create";

```

```

protected static abstract class Factory {
    protected abstract ASTree make0(Object arg) throws Exception;
    protected ASTree make(Object arg) {
        try {
            return make0(arg);
        } catch (IllegalArgumentException e1) {
            throw e1;
        } catch (Exception e2) {
            throw new RuntimeException(e2); // this compiler is broken.
        }
    }
}

protected static Factory getForASTList(Class<? extends ASTree> clazz) {
    Factory f = get(clazz, List.class);
    if (f == null)
        f = new Factory() {
            protected ASTree make0(Object arg) throws Exception {
                List<ASTree> results = (List<ASTree>)arg;
                if (results.size() == 1)
                    return results.get(0);
                else
                    return new ASTList(results);
            }
        };
    return f;
}

protected static Factory get(Class<? extends ASTree> clazz,
                             Class<?> argType)
{
    if (clazz == null)
        return null;
    try {
        final Method m = clazz.getMethod(factoryName,
                                           new Class<?>[] { argType });
        return new Factory() {
            protected ASTree make0(Object arg) throws Exception {
                return (ASTree)m.invoke(null, arg);
            }
        };
    } catch (NoSuchMethodException e) {}
    try {
        final Constructor<? extends ASTree> c
            = clazz.getConstructor(argType);
        return new Factory() {
            protected ASTree make0(Object arg) throws Exception {
                return c.newInstance(arg);
            }
        };
    } catch (NoSuchMethodException e) {
        throw new RuntimeException(e);
    }
}

protected List<Element> elements;
protected Factory factory;

```

```

public Parser(Class<? extends ASTree> clazz) {
    reset(clazz);
}
protected Parser(Parser p) {
    elements = p.elements;
    factory = p.factory;
}
public ASTree parse(Lexer lexer) throws ParseException {
    ArrayList<ASTree> results = new ArrayList<ASTree>();
    for (Element e: elements)
        e.parse(lexer, results);

    return factory.make(results);
}
protected boolean match(Lexer lexer) throws ParseException {
    if (elements.size() == 0)
        return true;
    else {
        Element e = elements.get(0);
        return e.match(lexer);
    }
}
public static Parser rule() { return rule(null); }
public static Parser rule(Class<? extends ASTree> clazz) {
    return new Parser(clazz);
}
public Parser reset() {
    elements = new ArrayList<Element>();
    return this;
}
public Parser reset(Class<? extends ASTree> clazz) {
    elements = new ArrayList<Element>();
    factory = Factory.getForASTList(clazz);
    return this;
}
public Parser number() {
    return number(null);
}
public Parser number(Class<? extends ASTLeaf> clazz) {
    elements.add(new NumToken(clazz));
    return this;
}
public Parser identifier(HashSet<String> reserved) {
    return identifier(null, reserved);
}
public Parser identifier(Class<? extends ASTLeaf> clazz,
    HashSet<String> reserved)
{
    elements.add(new IdToken(clazz, reserved));
    return this;
}
public Parser string() {
    return string(null);
}

```

```

public Parser string(Class<? extends ASTLeaf> clazz) {
    elements.add(new StrToken(clazz));
    return this;
}
public Parser token(String... pat) {
    elements.add(new Leaf(pat));
    return this;
}
public Parser sep(String... pat) {
    elements.add(new Skip(pat));
    return this;
}
public Parser ast(Parser p) {
    elements.add(new Tree(p));
    return this;
}
public Parser or(Parser... p) {
    elements.add(new OrTree(p));
    return this;
}
public Parser maybe(Parser p) {
    Parser p2 = new Parser(p);
    p2.reset();
    elements.add(new OrTree(new Parser[] { p, p2 }));
    return this;
}
public Parser option(Parser p) {
    elements.add(new Repeat(p, true));
    return this;
}
public Parser repeat(Parser p) {
    elements.add(new Repeat(p, false));
    return this;
}
public Parser expression(Parser subexp, Operators operators) {
    elements.add(new Expr(null, subexp, operators));
    return this;
}
public Parser expression(Class<? extends ASTree> clazz, Parser subexp,
    Operators operators) {
    elements.add(new Expr(clazz, subexp, operators));
    return this;
}
public Parser insertChoice(Parser p) {
    Element e = elements.get(0);
    if (e instanceof OrTree)
        ((OrTree)e).insert(p);
    else {
        Parser otherwise = new Parser(this);
        reset(null);
        or(p, otherwise);
    }
    return this;
}
}

```

第 3 部分

解说篇（自习时间）

第
18
天

GluonJ 的使用方法

GluonJ 的使用方法

为了通过 Java 语言实现类似于 Ruby 语言中 open class 的功能，本书采用了一种名为 GluonJ 的工具。为此，我们必须在编译与执行时多注意一些细节问题。本章将简单讲解 GluonJ 的使用方法。

18.1 设定类路径

由于通过 GluonJ 写成的程序中含有 @Reviser 等 Java 标注，因此编译器的类路径中必须包含 gluonj.jar。下面是一条编译示例。

```
javac -cp .:gluonj.jar chap6/BasicEvaluator.java
```

-cp（或 -classpath）是类路径设定选项。如果存在多个路径，Linux 与 Mac OS^① 的路径之间需要通过冒号 : 分隔，对于 Windows，则需使用分号 ;。在本章的所有示例中，默认 gluonj.jar 文件位于当前文件夹。

如果开发环境是 Eclipse，我们需要先打开 Project（项目）菜单中的 Properties（属性）面板，并在 Java Build Path（Java 的编译路径）的 Libraries（库）界面中将 gluonj.jar 作为外部 jar 文件添加进去（图 18.1）。

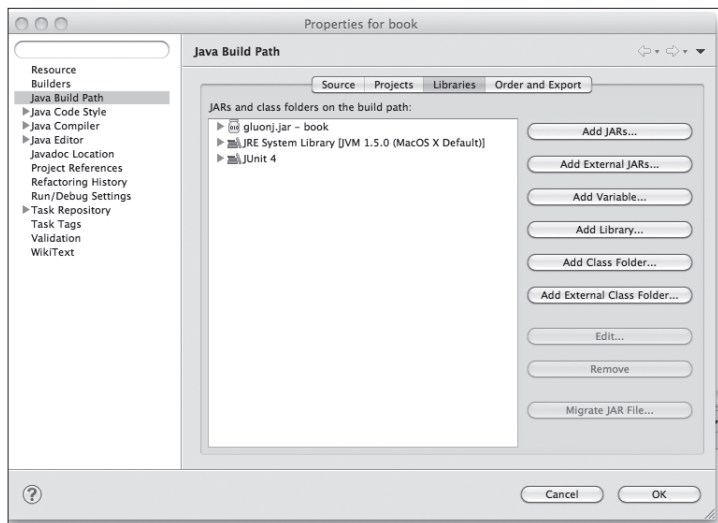


图 18.1 将 gluonj.jar 添加至编译路径

① 现改名为 OS X，是由苹果公司开发的操作系统。——译者注

18.2 启动设定

由于修改器无法直接应用于需要修改的类，因此我们必须在编译后，执行程序前，将修改器应用于相关的类。有多种方式能够确保修改器成功应用。

本书将通过启动程序来实现这一处理。除了原本的 main 方法外，我们还需要另外准备一个 main 方法，通过它调用由 GluonJ 提供的 Loader 类中的 run 方法，以启动原本的 main 方法。代码清单 18.1 展示的是第 6 章（第 6 天）代码清单 6.5 的 Runner 类，我们将以它为例进行讲解。其中，BasicInterpreter 是包含了原有 main 方法的类，我们将为它应用 BasicEvaluator 修改器。

代码清单 18.1 计时器启动程序 Runner.java（与代码清单 6.5 相同）

```
package chap6;
import javassist.gluonj.util.Loader;

public class Runner {
    public static void main(String[] args) throws Throwable {
        Loader.run(BasicInterpreter.class, args, BasicEvaluator.class);
    }
}
```

在通过该方法启动程序时，需要在类路径中包含 gluonj.jar。

```
java -cp ../gluonj.jar chap6.Runner
```

如果使用 Eclipse，由于类路径（编译路径）中已经含有了 gluonj.jar，因此无需做额外处理，只需执行 Runner 类的 main 方法即可。

此外还有一种解决方法，该方法不必使用启动程序，而是通过 Java 虚拟机的启动选项来应用所需的修改器。此时，程序的启动方式如下所示。

```
java -javaagent:gluonj.jar=chap6.BasicEvaluator chap6.BasicInterpreter
```

该指令与代码清单 18.1 的功能相同，即应用 chap6.BasicEvaluator 修改器，并启动 chap6.BasicInterpreter 的 main 方法。请读者注意，该指令启动的 main 方法并不是代码清单 18.1 中由 Runner 类提供的 main 方法，而是 BasicInterpreter 中原本的 main 方法。

我们可以通过下面的选项来启用 GluonJ。= 之后的部分用于指定修改器的应用方式。

```
-javaagent:gluonj.jar=
```

在 Eclipse 中，我们也能通过类似的方式，为程序添加选项并执行。为此，我们需要打开 Run Configurations（执行配置）面板中的 Arguments（参数）标签页，在该标签页的 VM arguments（虚拟机参数）栏中填入选项（图 18.2）。请读者注意，不要将选项误填入 Program arguments（程序参数）栏中。

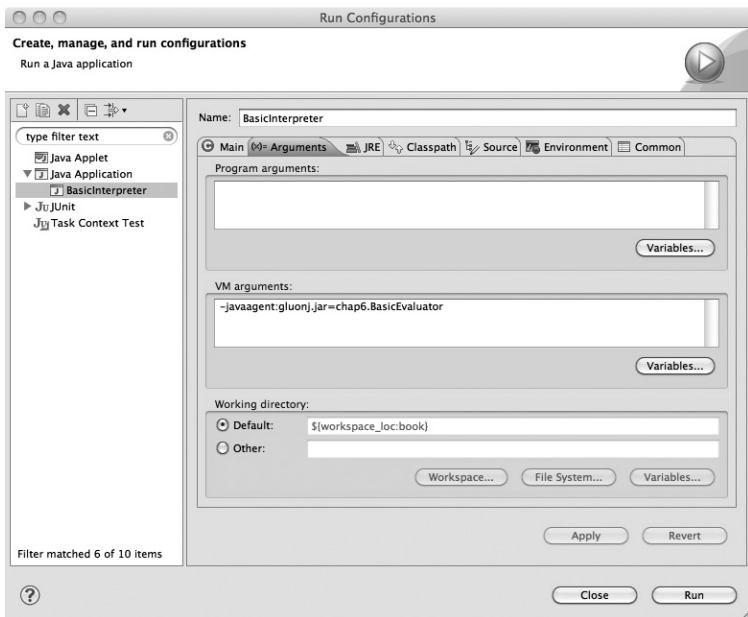


图 18.2 指定虚拟机参数

最后一种方法将对编译生成的类文件（.class 文件）执行后期处理，并在程序执行前事先应用相关的修改器。应用了修改器的类文件无需 GluonJ 也能正常执行，是标准的 Java 语言程序。即使没有代码清单 18.1 中 Runner 类那样的启动程序，文件也能启动执行。-javaagent:gluonj.jar= 这样的选项也不再需要。为了对类文件执行后期处理，我们要运行下面这样的 Java 指令。所有需要处理的类文件（及修改器）都要添加为该指令的参数。

```
java -jar gluonj.jar chap6/*.class stone/*.class stone/ast/*.class
```

该指令将改写参数包含的类文件的内容，指令执行后，即使没有 GluonJ，它们也能正常运行。请读者注意，选项中没有使用 -cp，而是使用了 -jar。由于经过后期处理得到的只是普通的 Java 类文件，因此启动程序时无需使用额外的选项，只要启动原本的 main 方法即可。在本例中，我们将启动 chap6.BasicInterpreter 类的 main 方法。

```
java chap6.BasicInterpreter
```

其中，类路径不必包含 gluonj.jar。

C 如果采用后期处理的方式，由修改器带来的启动开销将几乎为 0，因此不会由于使用了 GluonJ 而大幅拖慢程序的速度。

18.3 GluonJ 语言

本书使用了 GluonJ 框架，它能使 Java 语言实现类似于 Ruby 语言中 open class 那样的功能。不过事实上，GluonJ 是一种独立的程序设计语言，它有专门的用于定义修改器的语法。要使用这些 GluonJ 语言自定义的语法，我们需要通过专门的编译器来执行编译操作。由于 Eclipse 等集成开发环境不适合开发这类语言，因此本书采用了另外的框架，以在 Java 语言中使用 GluonJ 语言提供的功能。

- A** 话说，还是要有用于 GluonJ 语言的 Eclipse 插件才好呀。
- F** 它还是实验室级别的语言，跟能投入实用的语言还是有差距的。
- C** 最近新出现了 Spoofox^① 之类的工具，用于解决这类问题。不过要实现 Eclipse 对 GluonJ 的支持，依然不是一件容易的事。

如代码清单 18.2 所示，GluonJ 语言无需通过 @Reviser 标注来使用修改器。GluonJ 语言的修改器声明与类声明几乎相同，唯一的区别在于原本写于类名前的 class 替换成了 reviser。

代码清单 18.2 GluonJ 语言中的修改器声明

```
public reviser NegativeEx extends NegativeExpr {
    public Object eval(Environment env) {
        Object v = operand().eval(env);
        if (v instanceof Integer)
            return new Integer(-((Integer)v).intValue());
        else
            throw new RuntimeException("bad type " + location());
    }
}
```

- A** 哎呀，没太大区别嘛。
- C** 话不能这么说，这里重要的是在使用时是否需要执行数据类型转换。

此外，如果通过 Java 语言定义修改器，除了需要使用 @Reviser 标注，还必须根据需要显式地执行强制类型转换。例如，ASTree 类的 eval 方法需由 ASTreeEx 修改器添加。如果使用 @Reviser 标注，我们必须在调用 ASTree 对象新增的 eval 方法时，将对象强制转换为与修改器相同的类型。以代码清单 18.2 的第 3 行为例，我们需要改用下面这样的语句。

```
Object v = ((ASTreeEx)operand()).eval(env);
```

之所以要这样修改，是因为 operand 方法返回的是一个 ASTree 类型的值。不过，如果使用 GluonJ 语言专门的语法，我们就能像代码清单 18.2 那样，无需强制类型转换即可直接完成赋值操作。

^① <http://spoofox.org>

如果使用 `@Reviser` 标注，代码中插入的诸如 `(ASTreeEx)` 这样的强制类型转换必定成功，不存在转换失败并抛出 `ClassCastException` 异常的可能。需要修改的类一定能被成功转换为与相应的修改器相同的类型。由此可知，即使使用标注，也不会对 Java 语言的静态类型检查造成多大影响。不过，在使用 GluonJ 语言时，类型检查的效率更高。而且最重要的是，此时程序结构将更加简洁，因此我们更推荐做法。

H GluonJ 除了具有一些与 open class 类似的功能，还支持其他很多别的特性。不一起介绍一下吗？

C 本书用不到的功能就不专门讲解了。有兴趣的读者请自己去了解吧。

代码清单 18.3 通过 `@Reviser` 声明修改器

```
@Reviser public class NegativeEx extends NegativeExpr {
    public NegativeEx(List<ASTree> c) { super(c); }
    public Object eval(Environment env) {
        Object v = ((ASTreeEx)operand()).eval(env);
        if (v instanceof Integer)
            return new Integer(-((Integer)v).intValue());
        else
            throw new StoneException("bad type " + location());
    }
}
```

18.4 功能总结

我们已经林林总总讲解了一些 GluonJ 的功能，本章的最后将重新整理这些内容。GluonJ 语言提供了很多专用语法，不过我们要整理的并非 GluonJ 语言的这些语法，而是在本书中作为 Java 语言框架使用的 GluonJ 语言功能。

● `@Reviser`

标记 `@Reviser` 表示该类是一个修改器，它可以修改 `extends` 之后的类的定义。修改器的字段与方法将被直接添加至需要修改的目标类中。

如果需要修改的类含有构造函数，修改器也会包含具有相同类型签名的构造函数。如果构造函数不止一个，修改器将同时包含所有对应的构造函数。构造函数无法另外添加。

代码清单 18.3 是一个例子，它改写了代码清单 18.2，通过 `@Reviser` 实现了修改器的声明。两种方式的主要区别在于是否使用了构造函数，以及 `eval` 方法在最初调用 `operand()` 的 `eval` 方法时是否需要执行强制类型转换。

● 强制类型转换与 `revise` 方法

在调用由修改器添加的方法时，我们必须将修改后的类强制转换为与修改器一样的类型。不过有时，这样的强制类型转换可能会与 Java 语言的语法冲突，造成编译错误。此时，我们可以

改用 `javassist.gluonj.GluonJ` 类中的 `static` 方法 `revise`。例如，在第 11 章代码清单 11.4 中，`FunEx` 修改的 `lookup` 方法就使用了这一 `revise` 方法。

```
import static javassist.gluonj.GluonJ.revise;
:
((ASTreeOptEx)revise(body)).lookup(newSyms);
```

其中，`body` 是一个 `BlockStmtnt` 类型的变量，并应用了 `ASTreeOptEx` 修改器。`lookup` 方法由 `ASTreeOptEx` 修改器添加。根据 Java 语言的语法，`ASTreeOptEx` 修改器和 `BlockStmtnt` 类都是 `ASTree` 的子类，但两者之间并无继承关系。也就是说，`ASTreeOptEx` 类并不是 `BlockStmtnt` 类的子类。因此下面这样的强制类型转换将引起编译错误。

```
(ASTreeOptEx)body
```

为避免该问题，我们要像下面这样改用 `revise`。

```
(ASTreeOptEx)revise(body)
```

● super 调用

修改器在覆盖修改对象的类方法时，可以在自身的方法中通过 `super` 调用需要修改的类提供的方法。这与子类的方法可以借助 `super` 来调用父类方法的机制相同。

如果一个类应用了多个修改器，由某个修改器添加的方法能被其他修改器再次覆盖。这时，`super` 调用的是被覆盖的修改器提供的方法。

● @Require

如果修改器含有 `@Require` 标注，表示它依赖于由 `@Require` 的参数列出的其他修改器。如果某个修改器 `R` 依赖于另一个修改器 `S`，程序在应用修改器 `R` 之前将首先隐式地应用修改器 `S`。也就是说，修改器 `S` 的应用将先于 `R`。如果两个修改器修改了同一个类，后应用的修改器有可能会覆盖前一个修改器添加的方法。

● 分组

我们能将多个修改器分组。标有 `@Reviser` 但不含 `extends` 部分的类用于表示修改器的分组。分组内的修改器都是该类的嵌套子类。这些嵌套子类都必须是 `static` 类型。

修改器的分组可以作为 `@Require` 标注的参数使用。此时，具有该标注的修改器将依赖于参数中的修改器的分组。这一分组中的所有修改器都将被事先应用。

● 修改器的 extends 操作

除了通常的类，修改器还可以 `extends` 另一个修改器。例如，对于下面这样修改器 `R` 的定义，`S` 如果不是普通的类，而是另一个修改器，修改器 `R` 将修改 `S` 修改的类。

```
@Reviser public class R extends S { ... }
```

此时，修改器 R 将在修改器 S 之后应用。因此，修改器 R 的方法能够覆盖由修改器 S 添加的方法。

专栏7 英勇事迹（续）



第 3 部分

解说篇（自习时间）

第
19
天

抽象语法树与设计模式

抽象语法树与设计模式

本书在实现程序中的抽象语法树时，使用了 GlueJ 语言以及 Ruby 语言中 open class 风格的语法结构。不过，抽象语法树的类设计有一些重要的设计模式，通过这些设计模式，我们可以完全在 Java 语言的框架内设计相关的类。本章将讲解如何通过设计模式设计抽象语法树，并分析这种方式的优缺点，供有兴趣的读者阅读。

19.1 理想的设计

本书已经分阶段逐步构建了 Stone 语言的语言处理器。我们首先选取了一些必需的语法规则，实现了一个极为简单的版本。在确认该处理器能正常工作后，我们又进一步为其添加新功能，并再次确认执行情况，如此反复……本书采用的这种流程称为迭代式开发风格。如今，不只是语言处理器，这种方式广泛应用于 Web 服务等各种类型的开发，十分常见。

然而，在新增功能时，我们难免要为一些类改写或添加方法与字段。对已经确认能正常执行的程序做局部修改，也就是修改已有的类定义，或彻底替换某些类的定义，将使程序变得难以理解，提升今后的开发难度。

H 如果随意修改程序，可能会把原本能正常执行的部分也改错呢。

S 嗯，这时需要使用版本管理系统才行，这可是常识。如果在修改前签入过代码，万一发生问题，只要还原之前的代码即可。

C 不过进行多次修改后，合并与移除会很麻烦。

F 老师，通过语言来管理版本问题时，融合也不轻松。不过，这时的粒度较细，比版本管理系统多少要方便些……

理想情况下，程序新增的功能应能作为差分文件^①处理。只要将它们与现有程序结合，就能得到具有新功能的软件。我们希望尽可能不对原有程序做额外的修改。此外，即使是差分文件，也应易于开发者阅读与理解。如果差分文件的表述形式与 diff 指令的输出结果类似，和程序结构本身无关，将给开发造成很大的困难。

实现这一理想，是程序设计语言研究领域的一大目标。在开发程序设计语言的处理器时，我们可以通过多种手法努力向这一目标靠近。本章将介绍 interpreter 模式与 visitor 模式，这两种设计模式是常用的处理器设计手法。应用这些模式后，程序将变得易于修正，且无需在新增功能时

^① 或称为差别文件。——译者注

对代码做太大的改动。

19.2 Interpreter 模式

第 6 章(第 6 天)最初在设计仅含基本功能的 Stone 语言处理器时,通过修改器向抽象语法树的节点类添加了 eval 方法。新增了 eval 方法的类其实采用了 Interpreter 设计模式。所有抽象语法树的节点类共享一个父类 ASTree,各个类又分别定义了各自的 eval 方法,这是典型的 interpreter 模式。本书使用了大量的抽象语法树节点类,方便起见,下面将以一种简化版本为例进行讲解。首先,抽象语法树将仅含 NumberLiteral 与 BinaryExpr 这两种具体类。在第 4 章(第 4 天)的图 4.5 中,它们与 ASTree 类之间还夹有 ASTLeaf 类与 ASTList 类,不过本章将省略这两个类。此外,各类的字段定义也将从简,与 BinaryExpr 类对应的节点仅表示加法或乘法。代码清单 19.1 是具体的程序。图 19.1 是与之对应的类图。

由于采用了 interpreter 模式,各节点类共享的父类 ASTree 也包含 eval 方法。调用该方法后程序将开始执行。BinaryExpr 类含有字段 left 与 right,它们分别表示左侧与右侧的操作数,是 ASTree 类型的变量。因此,我们无需在意实际的对象究竟是 NumberLiteral 还是 BinaryExpr,只需直接调用 eval 方法即可。调用 eval 方法后,语言处理器将根据对象的实际类型选择合适的 eval 方法执行。这种方式称为动态方法分派,是面向对象语言具备的基本功能。

代码清单 19.1 简化后的抽象语法树节点类

```
public abstract class ASTree {
    public abstract int eval() throws Exception;
}

public class NumberLiteral extends ASTree {
    public Token value;
    public int eval() throws Exception {
        return value.getNumber();
    }
}

public class BinaryExpr extends ASTree {
    public Token operator;
    public ASTree left, right;
    public int eval() throws Exception {
        String op = operator.getText();
        if ("+".equals(op))
            return left.eval() + right.eval();
        else if ("*".equals(op))
            return left.eval() * right.eval();
        else
            throw new Exception("bad operator " + op);
    }
}
```

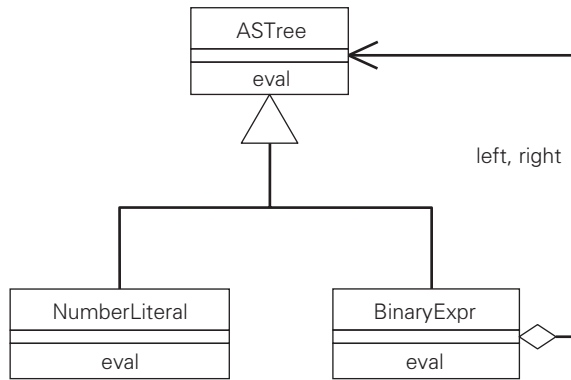


图 19.1 简化后的抽象语法树节点类之间的相互关系

代码清单 19.2 将 eval 方法从抽象语法树的节点类中分离

```

public class Evaluator {
    public static int eval(ASTree tree) throws Exception {
        if (tree instanceof BinaryExpr) {
            BinaryExpr expr = (BinaryExpr)tree;
            String op = expr.operator.getText();
            if ("+".equals(op))
                return expr.left.eval() + expr.right.eval();
            else if ("*".equals(op))
                return expr.left.eval() * expr.right.eval();
            else
                throw new Exception("bad operator " + op);
        }
        else if (tree instanceof NumberLiteral) {
            NumberLiteral num = (NumberLiteral)tree;
            return num.value.getNumber();
        }
        else
            throw new Exception("unknown ASTree");
    }
}

```

C 你们觉得 interpreter 模式好在哪里呢？

F 对于面向对象语言写成的程序来说，这些是理所应当的，没觉得特别好呀。

A 这都已经被归纳为一种设计模式了，只管用就好了，不用考虑为什么要用它。

我们来考虑一下不使用 interpreter 模式的程序，并与使用了该模式的程序进行比较。eval 方法用于执行程序，但它并不一定适合作为抽象语法树节点类中的方法。有观点认为，抽象语法树节点类的方法应仅用于抽象语法树的构建及其他一些相关操作，而不是程序的执行。

代码清单 19.2 将 eval 方法分离出了抽象语法树的节点类。该 eval 方法将接收抽象语法树作为参数，并执行程序。它将分析由参数传递而来的对象 tree 的类型，根据情况执行相应

的处理。实际的处理逻辑与代码清单 19.1 的 eval 方法基本相同。可以说，它只是通过 if 语句显式地完成了由动态方法分派隐式地执行的操作。图 19.2 显示了这种方式与 interpreter 模式之间的差异。interpreter 模式将分别在不同的类中执行相应的处理（在图中以矩形表示），而分离后的 eval 方法则将在一条 if 语句中执行所有的处理。

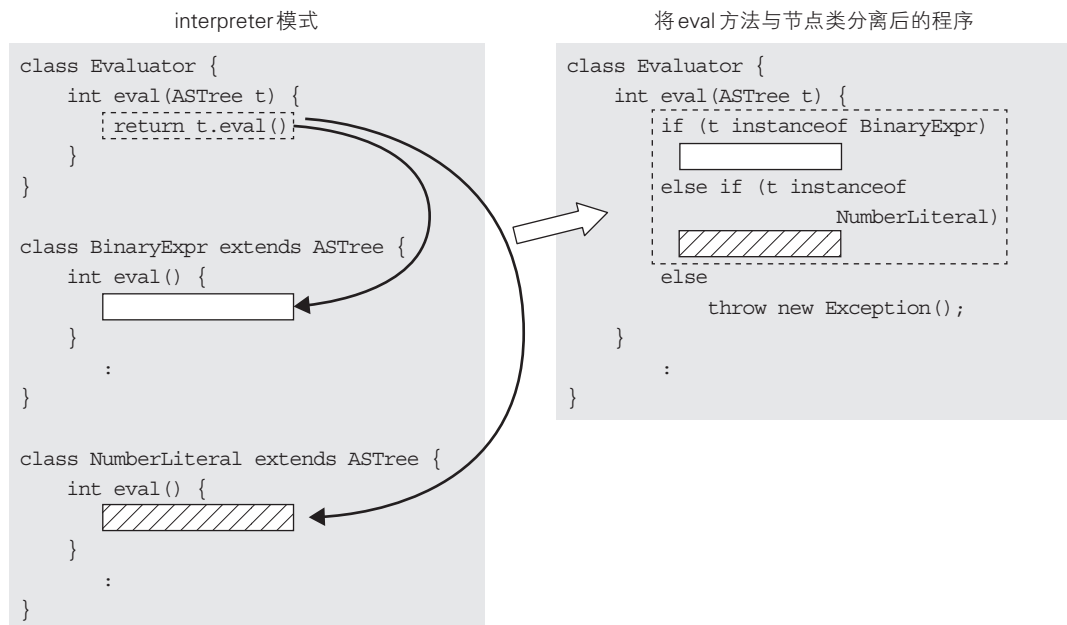


图 19.2 动态方法分派与 if 语句

F 这种结构不佳的代码简直是典型的反面教材呢。

C 不过，在面向对象普及之前，大家都是这么设计程序的哦。

使用了 interpreter 模式的程序，即使之后因扩充语法规则而新增了一些 ASTree 类的子类，原有程序也无需做太多修改，这也是 interpreter 模式的一大优点。例如，假设我们要为语言处理器扩充语法，为其添加对单目负号运算符（即整数前的负号）的支持。此时，原有程序几乎不会有什么改动。

为了表示具有负号的项，我们可以为 ASTree 定义一个新的子类 NegativeExpr。如果程序没有采用 interpreter 模式，if 语句就必须做相应的修改。我们必须为 if 语句添加逻辑，使它能够在参数是一个 NegativeExpr 对象时也做出恰当的处理（图 19.3）。另一方面，如果程序采用了 interpreter 模式，我们只需为 NegativeExpr 类定义合适的 eval 方法即可。BinaryExpr 类等其他一些已经能够正确运行的部分不必做额外的修改。

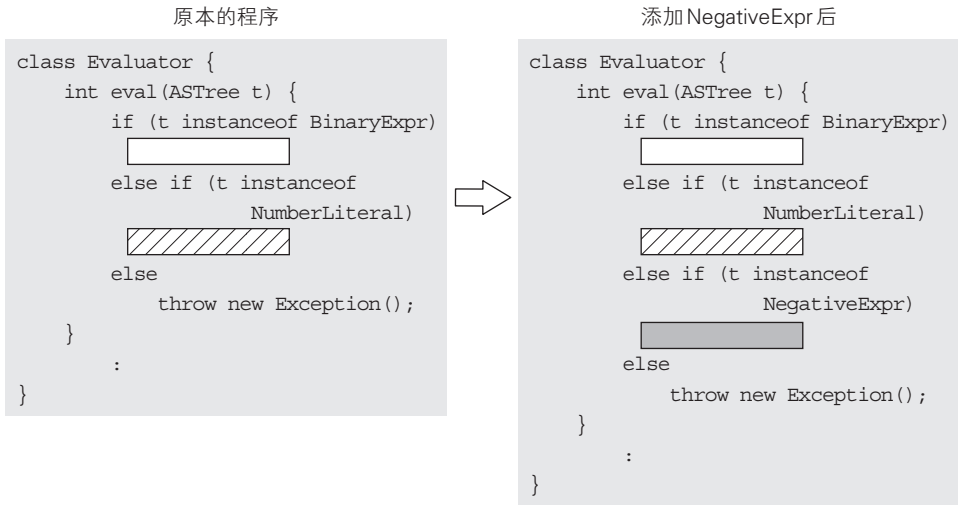


图 19.3 新增抽象语法树的节点类

- A** 说是要修改 if 语句，也不过是添加一条 else 分支嘛，不要紧，不是什么大问题。
- C** 方法内部也需要修改哦。对于现有的这些能够正常执行的类，它们的源代码免不了要做些修改。
- S** 常常看起来容易，真做起来时会发现要改的地方一大堆。
- F** 如果使用 interpreter 模式，就不用修改已有的源代码了呢。
- H** 不过老师您也得讲讲 interpreter 模式的缺点才行啊……

采用 interpreter 模式的程序也存在不足。如果程序不采用该模式，所有的处理都能在 Evaluator 类的 eval 方法中完成。采用 interpreter 模式的程序则需要分别在不同的 eval 方法中执行处理。在上例中，BinaryExpr 类与 NumberLiteral 类各自含有专门的 eval 方法，原本能在一个 eval 方法中执行的处理不得不分为两部分。随着 ASTree 的子类的增加，eval 方法的数量也会不断增加。由于这些方法分别位于不同的源文件中，因此很难理清全体 eval 方法具体实现了怎样的功能，可读性较差。例子中的程序比较简短，这个问题尚不明显，如果程序规模较大，我们就不能再忽略这个缺点了。面向切面程序设计中的横切关注点也有这个问题。

19.3 Visitor 模式

前面介绍的 interpreter 模式还有另一个问题。在第 11 章（第 11 天）的具体实现中，我们为抽象语法树的节点类添加了 lookup 方法。lookup 方法将事先统计程序中出现的变量，并确定变量的保存位置。与 eval 方法相同，该方法也将从抽象语法树的根节点向叶节点遍历整棵语法

树，同时逐步执行必要的计算。

A 噢，本章讨论的抽象语法树中不含变量呀。只有数字而已。

H A 君，就别在意这些小细节了。

C 无论 lookup 方法的内部怎样，总之只要知道 lookup 方法在执行时会遍历语法树，并查找叶节点的内容就好了。

如果使用 interpreter 模式，我们就必须为各个类添加相应的 lookup 方法，其中包含了与 eval 方法同样的程序结构。NumberLiteral 类与 BinaryExpr 类原本已能正常运行，却依然需要添加一些方法，其实质与修改代码无异。这并非是我们所希望的结果。

A 哎呀，interpreter 模式也不行呀。

C 不，上面的例子已经超出了 interpreter 模式的适用范围。所以说上面说的其实有些吹毛求疵了。

此时，我们可以利用 visitor 模式来解决这一问题。该模式下，eval 方法与 lookup 方法将被移至其他的类，抽象语法树的 ASTree 类、NumberLiteral 类或 BinaryExpr 类中无需包含这些方法。在新增方法时，抽象语法树的类定义也不需要修改。我们只要创建新的类，再向其中添加方法即可。

F 说起来，那本书^①里也说这种情况下适合使用 visitor 模式呢。

我们来看一下经改写后采用了 visitor 模式的程序。该模式是一种不易从整体上把握的设计模式，图 19.4 是它的整体结构。大家可以比较一下图 19.2 介绍的 interpreter 模式。不难发现，在扩充功能时，visitor 模式无需向 NumberLiteral 与 BinaryExpr 等抽象语法树节点类添加诸如 eval 或 lookup 这类的方法。

首先，我们无需直接在抽象语法树的相关类中定义 eval 等方法。为此，我们需要为各个类逐一定义 accept 这一通用的方法作为替代。另一方面，具体的处理操作将由 Visitor 类型的对象中的方法实现。accept 方法将接收该对象作为参数，选择该对象中合适的 visit 方法并执行。这样一来，我们只需更改传递给 accept 方法的 Visitor 对象，就能轻松替换抽象语法树各节点类的处理操作。这就是 visitor 模式的核心思想。图 19.4 中的虚线表示由参数接收的 Visitor 对象的替换操作。

^① *Design Pattern*, Erich Gamma 等著，艾迪生-韦斯利出版公司 1995 年出版。中文译本《设计模式》由机械工业出版社于 2000 年 9 月出版，李英军、马晓星、蔡敏、刘建中等译。

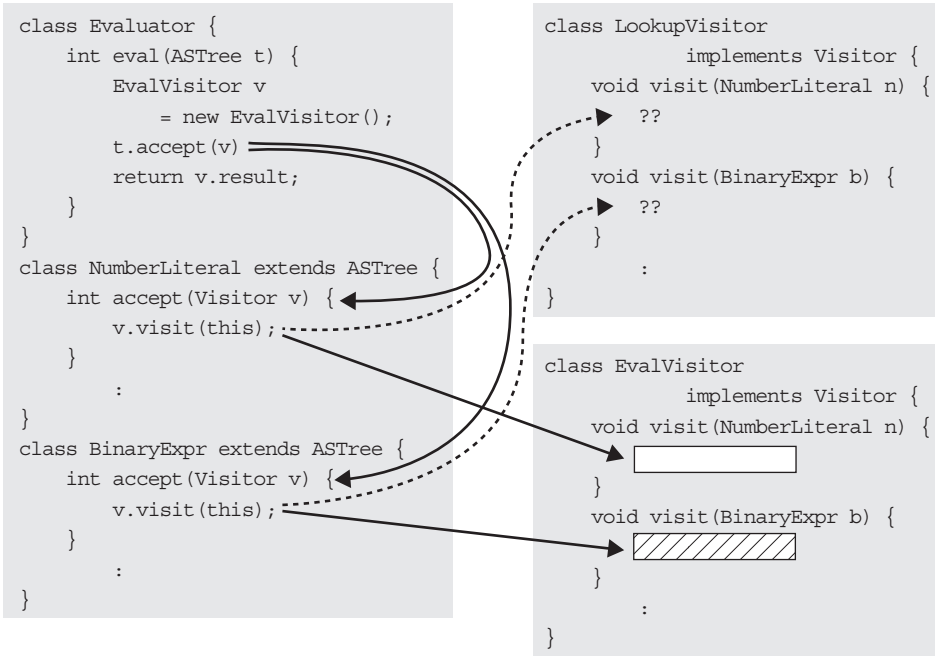


图 19.4 使用 visitor 模式并改写程序

- A** 这段讲解完全听不懂啊。
- C** 果然还是要阅读实际的代码才行。
- S** 嗯，不过就算阅读实际的代码，也没那么容易理解哦。

interpreter 模式将在调用抽象语法树根节点对象的 eval 方法后开始执行程序。visitor 模式在将像下面这样调用 accept 方法执行。其中，变量 t 指向表示抽象语法树根节点的 ASTree 对象。

```

EvalVisitor v = new EvalVisitor();
t.accept(v);
int res = v.result;
    
```

这段代码首先创建了一个用于执行实际处理的 EvalVisitor 对象，再将它作为参数调用 accept 方法。此处，计算结果通过 result 字段获得，而不是 accept 方法的返回值。在 interpreter 模式下，程序将仅执行 t.eval()，visitor 模式则会按上述方式处理。

- S** 通过泛型 (generics) 就能让 accept 方法的返回值作为计算结果哦。
- C** 嗯……之后我会讲的。

代码清单 19.3 是根据 visitor 模式修改得到的抽象语法树节点类。它去除了代码清单 19.1 中的 eval 方法，同时添加了相应的 accept 方法。新增的 accept 方法的参数类型

是 Visitor 接口。代码清单 19.4 是该接口的定义。也就是说，抽象语法树能够通过 accept 方法接收 (accept) 指定的访问者 (visitor)。

除了 ASTree 类之外，代码清单 19.3 的各个类中的 accept 方法都基本相同。然而，由于 visit 方法已被覆盖，因此它们在实际执行时将分别调用不同的 visit 方法。变量 this 的类型将由具体情况决定，在调用 visit 方法时，参数可以是各种类型的值。

代码清单 19.3 根据 visitor 模式改写抽象语法树的节点类

```
public abstract class ASTree {
    public abstract void accept(Visitor v) throws Exception;
}

public class NumberLiteral extends ASTree {
    public Token value;
    public void accept(Visitor v) throws Exception {
        v.visit(this);
    }
}

public class BinaryExpr extends ASTree {
    public Token operator;
    public ASTree left, right;
    public void accept(Visitor v) throws Exception {
        v.visit(this);
    }
}
```

代码清单 19.4 Visitor 接口 (Visitor.java)

```
public interface Visitor {
    void visit(NumberLiteral n) throws Exception;
    void visit(BinaryExpr e) throws Exception;
}
```

于是，以 t.accept(v) 的形式调用 accept 方法后，最终调用的其实是与变量 t 所属的对象类型对应的 visit 方法。另一方面，interpreter 模式将通过 t.eval() 调用与 t 的对象类型对应的 eval 方法，两者表示的含义相同，区别仅在于一个方法名是 visit，另一个是 eval，且这两种方法分别由不同的类定义。

代码清单 19.5 是 EvalVisitor 类的定义。在 interpreter 模式下，NumberLiteral 与 BinaryExpr 类分别含有各自的 eval 方法，在 visitor 模式下，它们将汇总于 EvalVisitor 类，同时，方法名变为 visit。EvalVisitor 类通过不同的方法，描述了它在访问 (visit) NumberLiteral 类或 BinaryExpr 类等各种不同的类时，应具体执行哪些操作。

此外，为提高通用性，visit 方法将返回一个 void 而非 int 类型的值。因此，计算结果的返回方式也将发生变化。visit 方法不通过 return 语句返回结果，而将借助 result 字段传递方法的返回值。被调用的方法将把返回值保存于 result 字段，调用方则通过该字段读取所需的返回值。

代码清单 19.5 EvalVisitor 类 (EvalVisitor.java)

```
public class EvalVisitor implements Visitor {
    public int result;
    public void visit(NumberLiteral num) {
        result = num.value.getNumber();
    }
    public void visit(BinaryExpr e) throws Exception {
        String op = e.operator.getText();
        e.left.accept(this);
        int r1 = result;
        e.right.accept(this);
        if ("+".equals(op))
            result = r1 + result;
        else if ("*".equals(op))
            result = r1 * result;
        else
            throw new Exception("bad operator " + op);
    }
}
```

C 通过对象的字段传递返回值的做法看起来可能有些古怪。

F 哎呀，说明 EvalVisitor 不支持多线程是吧？

H 不会啊。只要为每个线程创建 EvalVisitor 对象，就能正常地并行处理了不是吗？

S 老师，泛型的用法什么时候介绍……

C 对了，这就讲。虽然有些复杂，但我们也能直接通过 accept 方法的返回值获取计算结果。为此，我们首先要将 Visitor 接口改为泛型。

```
public interface Visitor<R> {
    R visit(NumberLiteral n) throws Exception;
    R visit(BinaryExpr e) throws Exception;
}
```

然后，各类中的 accept 方法需做如下改动。

```
public <R> R accept(Visitor<R> v) throws Exception {
    return v.visit(this);
}
```

accept 方法的返回值类型为 R，与参数传来的 Visitor 对象的类型参数相同。R 是一个类型变量。之后，我们只要再修改一下 EvalVisitor 的定义，就能使 accept 方法在接收 EvalVisitor 类型的参数时返回 Integer 类型的值。

```
public class EvalVisitor implements Visitor<Integer> { ... }
```

之所以能这样，是由于 EvalVisitor 也是 Visitor<Integer> 类型的类。顺便一提，如果类型为 Visitor<String>，accept 方法的返回值就是 String 类型。

代码清单 19.6 LookupVisitor 类 (LookupVisitor.java)

```
public class LookuplVisitor implements Visitor {
    private Symbols symbols;
    public LookuplVisitor(Symbols syms) { symbols = syms; }
    public void visit(NumberLiteral num) {
        symbols.put(num.value.getText());
    }
    public void visit(BinaryExpr e) throws Exception {
        e.left.accept(this);
        e.right.accept(this);
    }
}
```

根据 Visitor 模式修改程序后, 虽然类的设计变得更为复杂, 但能够更轻松地为抽象语法树添加新的处理逻辑。应用该模式后, 我们无需再对 NumberLiteral 类及 BinaryExpr 类等抽象语法树的节点类做额外的修改。

例如, 假设我们希望实现 lookup 处理, 对变量执行统计。在 interpreter 模式下, 我们必须修改抽象语法树的各个节点类, 为它们添加 lookup 方法。在 visitor 模式下, 我们不需要进行这类修改。我们只需定义一个实现了 Visitor 接口的 LookupVisitor 类, 并在其中定义 NumberLiteral 类与 BinaryExpr 类所需的 visit 方法即可。各个类的 lookup 处理将由对应的 visit 方法承担。在定义了 LookupVisitor 类之后, 我们只要创建相应的对象, 并以它为参数调用 accept 方法, 就能执行对整棵抽象语法树的 lookup 处理。

代码清单 19.6 是 LookupVisitor 类的定义。lookup 方法原本用于查找表达式中的变量, 不过本章讨论的表达式中不含变量, 只有数值。方便起见, 我们在实现 visit 方法时, 该方法执行的 lookup 处理将只查找表达式中的数值, 并通过 put 方法将其保存至 Symbols 对象之中。

由于 lookup 处理没有返回值, 所以 LookupVisitor 类不必包含 result 字段。此时, 返回值将通过 EvalVisitor 类的 result 字段返回, 因此我们无需更改 Visitor 接口的定义, 就能支持任意类型的返回值。

lookup 处理没有返回值, 查找结果将由 result 字段表示。无论哪种类型的返回值都能通过这种方式处理。例如, 假设返回值为 String 类型, 程序能自动将 result 字段转换为 String 类型。我们不需要对抽象语法树的节点类或 Visitor 接口做额外的修改。在 visitor 模式下, 类设计从整体上具有极高的通用性。

不过, 尽管 lookup 处理没有返回值, 它需要接收参数。第 11 章(第 11 天)添加的 lookup 方法将接收一个 Symbols 类型的参数。LookupVisitor 类改用 symbols 字段来支持这一设计, 该字段将通过构造函数的参数初始化。Symbols 对象在作为参数传递后, 将借由该字段保存。与返回值处理的设计理念相同, 这种方式确保了整个类设计有着很高的通用性。

19.4 使用反射

A visitor 模式还是不行啊。

F 为什么这么说？

A 普通人哪儿能理解这么复杂的东西呀。要能搞懂就都是天才了。

visitor 模式或许能称得上是最难理解的模式之一。不过，得益于 `EvalVisitor` 与 `LookupVisitor`，我们不必修改 `ASTree` 类，或 `NumberLiteral` 与 `BinaryExpr` 等子类，就能为语言处理器添加新的语法功能。这也是 visitor 模式的一大优点。此外，由于添加的部分全都集中于 `EvalVisitor` 与 `LookupVisitor` 类中，处理逻辑的可读性也很强。但另一方面，因为 `ASTree` 类及其子类新增了 `accept` 方法，因此程序的复杂度将会提升。

我们可以通过 Java 语言提供的反射机制降低这种类型的程序复杂性。使用反射之后，程序能在执行过程中查找某个类具有哪些方法，并执行找到的方法。

如果使用了反射，`Visitor` 接口就不再需要。此外，作为抽象语法树节点类的父类，`ASTree` 类仍然需要定义 `accept` 方法，不过它的子类不必再对 `accept` 方法进行定义。程序中其他类的定义与没有使用反射时的 visitor 模式相同。

我们来看一下具体的程序。代码区的 19.7 是除了 `ASTree` 之外的抽象语法树节点类的简单定义。此时，这些节点类中不需要 `eval` 与 `accept` 等方法。`EvalVisitor` 与 `LookupVisitor` 的定义没有变化，不过由于我们不再使用 `Visitor` 接口，因此删去了 `implements` 一句。其他部分与原来相同。

代码清单 19.7 使用了反射机制的抽象语法树节点类

```
public class NumberLiteral extends ASTree {
    public Token value;
}

public class BinaryExpr extends ASTree {
    public Token operator;
    public ASTree left, right;
}
```

A 不用写 `eval` 与 `accept` 方法，反射真是太棒啦！

使用反射机制的关键在于为 `ASTree` 类定义通用的 `accept` 方法。只要设计出通用的 `accept` 方法，`ASTree` 的子类就不必一次次覆盖该方法。代码清单 19.8 是改写后的 `ASTree` 类。

该 `accept` 方法将在由参数传递的 `visitor` 对象中查找与自身所属的类相符的 `visit` 方法。该方法自身所属的类型可以通过 `getClass` 方法获取。之后，程序将对 `visitor` 对象调用找到的 `visit` 方法，参数就是该类自身。例如，假设该对象是一个 `BinaryExpr` 对象，此

时，`accept` 方法将查找方法 `visit(BinaryExpr e)`，并以自身（`this` 对象）作为参数调用该方法。

`findMethod` 方法将负责实际查找适用的 `visit` 方法。它将在由参数传递的 `visitor` 对象中查找以自身（`this` 对象）的类型为参数的 `visit` 方法。如果没有符合条件的结果，`findMethod` 方法将查找以自身的父类作为参数的 `visit` 方法。例如，假设 `this` 对象是一个 `BinaryExpr` 对象，如果 `visitor` 对象中不存在 `visit(BinaryExpr e)` 方法，由于 `BinaryExpr` 存在父类 `ASTree` 类，因此 `findMethod` 方法将进而查找 `visit(ASTree e)` 方法。

A 这样一来 `visitor` 模式就完美了！太棒了！

F 不过要像这样使用反射机制，运行速度可就很慢了昵。

C 确实……不过只要多花些功夫设计 `ASTree` 类的实现，使其支持缓存功能，运行速度倒也不至于太慢。

代码清单 19.8 通用的 `accept` 方法

```
import java.lang.reflect.Method;

public abstract class ASTree {
    public final void accept(Object visitor) throws Exception {
        Method method = findMethod(visitor, getClass());
        if (method != null)
            method.invoke(visitor, this);
    }
    private static Method findMethod(Object visitor, Class<?> type) {
        if (type == Object.class)
            return null;
        else
            try {
                return visitor.getClass().getMethod("visit", type);
            }
            catch (NoSuchMethodException e) {
                return findMethod(visitor, type.getSuperclass());
            }
    }
}
```

`visitor` 模式的逻辑较难理解，不过如果通过反射机制实现，该模式的本质就变得一目了然了。在调用 `ASTree` 对象的 `accept` 方法后，语言处理器将选择并调用与该对象实际所属的类对应的 `visit` 方法。为此，我们可以在与各个类对应的 `visit` 方法中实现相应的处理。由于所有的处理都整合于 `EvalVisitor` 类中，因此可读性得到了改善。此外，在添加新的处理时，我们只需定义一个类似于 `LookupVisitor` 的新类即可，而不必修改已有的类。在当前的设计中，实际执行处理的对象类已经与抽象语法树的节点类分离。

F 这就是所谓的关注点分离呢。

在使用反射机制实现 visitor 模式时，该模式原本存在的另一个问题也将得到改善。在原本的实现中，如果为抽象语法树添加了新的节点类，我们就必须对 Visitor 接口以及所有实现了该接口的类进行修改。

例如，假设我们要为语言处理器扩充单目符号运算符功能，为 ASTree 添加一个新的子类 UnaryExpr。此时，我们必须修改 Visitor 接口，添加一个 visit(UnaryExpr e) 方法。实现该接口的 EvalVisitor 等类自然也需要修改，增加 visit(UnaryExpr e) 方法。事实上，大部分已有的类都需要做一定的修改。

对于抽象语法树节点类的添加问题，interpreter 模式反而更有优势。visitor 模式虽然便于添加抽象语法树的处理，但在添加抽象语法树的节点类时会较为复杂。

不过，在通过反射机制实现 visitor 模式时，我们能在很大程度上避免这个问题。在添加 UnaryExpr 类时，我们只要定义一个 EvalVisitor 的子类，并实现相应的 visit(UnaryExpr e) 方法即可。在这种实现方式下，所谓的静态数据类型检查将不再起效，因此无需执行额外的修改。此外，由于程序不再需要使用 Visitor 接口，自然也就没有修改它的必要。

F 这种实现吸取了动态类型语言的部分优点呢。

C 但反过来讲，如果缺少 visit 方法，编译器也发现不了。

19.5 面向切面语言

A 咦，还有东西要讲啊。

H A 君，只是总结而已，最后再总结一下。

最后，回到问题的核心，我们希望达到的目标有两点。

- 我们希望能在扩展程序功能时，不必修改已有的可以正常运行的类。扩展分为以下两类：
 - 新增诸如 UnaryExpr 这样的抽象语法树节点类；
 - 新增诸如 lookup 这样的针对抽象语法树的处理操作。
- 我们希望能在同一位置实现所有相同类型的方法。

为了实现以上目标，本章介绍了 interpreter 与 visitor 这两种设计模式。它们都是非常优秀的面向对象程序设计手法，但都有优缺点。

在新增 UnaryExpr 等抽象语法树的节点类时，采用 interpreter 模式的程序只需做很少的修改。但如果要为抽象语法树新增 eval 或 lookup 这样的处理操作，该程序就不得不修改各种类型的类，不但修改量较大，可读性也很差。

另一方面，visitor 模式的程序在为抽象语法树添加 eval 与 lookup 处理时，只需修改 EvalVisitor 类即可，修改量较小，且可读性很高。不过，如果要新增 UnaryExpr 之类的抽象语法树节点，程序就必须进行大量的修改。此外，我们无法直接为抽象语法树的节点对象添加新的字段。如果要在 visitor 模式中使用反射机制，程序的复杂性将会上升，这也算是一个不足。

因此，本书采用了面向切面语言，通过 GlueJ 来实现语言处理器。这种方式不但无需勉强设计复杂的 Java 程序，而且能够使语言处理器支持更多的功能。

当然，说是采用了面向切面语言的设计方式，本书其实仅使用了 Ruby 语言中的 open class 功能。不过，很多面向切面语言都支持 open class 或类似的功能。该功能可以有效实现面向切面的程序设计目标。如本节开头所讲，面向切面程序设计的主要目标之一就是希望整合相同类型的代码逻辑，并尽可能减少在功能扩展时所需的修改量。

F 为了得出这一结论，我们讲了好久呢。

C 之前也说过，如果不先试着通过面向对象的方式实现语言处理器的话，读者可能无法理解为什么我们一定要使用 GlueJ。

F 其实这才是老师最想写的内容不是吗？

H 据说老师起初是打算在第 4 章讲解这部分的。

A 哎呀，这可不行。如果是我，读到哪里就该把书扔一边了。

C 其实我草稿都写好了，给编辑一看，说是“第 4 章的 iterator 模式好像有点太难了”，于是只好放弃，挪到后面去了。

F iterator？不是 interpreter 吗？

C 不，我没讲错。他确实是说觉得 iterator 模式太难了。

H 也就是说，interpreter 模式难到连编辑都错看成 iterator 了么，又或是因为实在太枯燥所以搞错了……

A 呀，真是有点无奈呀。

编辑 那时候正好在处理另一篇与 iterator 有关的文章……看错了真是抱歉。

C 对了，这是本书的编辑 I 先生。

H 初次见面，我是 H。

F 您也是 SD 杂志的编辑对吧，我一直在读您的杂志哦。

C 最近那边的工作比较忙，这本书就无暇顾及了呢。

H（好啦，老师）尽管如此，各位读者还是读完了这本书啊。

C 怎么说呢……之后进行了反思，改写了很多。

编辑 我觉得这本书真的非常棒！

A 噢，I 先生你怎么好像在冒汗呀？

版 权 声 明

2 SHUKAN DE DEKIRU! SCRIPT GENGO NO TSUKURIKATA by Shigeru Chiba

Copyright © 2012 Shigeru Chiba

All rights reserved.

Original Japanese edition published by Gijyutsu-Hyoron Co., Ltd., Tokyo

This Simplified Chinese language edition published by arrangement with
Gijyutsu-Hyoron Co., Ltd., Tokyo in care of Tuttle-Mori Agency, Inc., Tokyo

本书中文简体字版由 Gijyutsu-Hyoron Co., Ltd. 授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

关注图灵教育 关注图灵社区

iTuring.cn

在线出版 电子书《码农》杂志 图灵访谈 ……



QQ联系我们

读者QQ群: 218139230



微博联系我们

官方账号: @图灵教育 @图灵社区 @图灵新知

市场合作: @图灵袁野 @图灵刘紫凤

写作本版书: @图灵小花 @陈冰_图书出版人

翻译英文书: @李松峰 @朱巍ituring @楼伟珊

翻译日文书或文章: @图灵乐馨

翻译韩文书: @图灵陈曦

电子书合作: @hi_jeanne

图灵访谈/《码农》杂志: @李盼ituring

加入我们: @王子是好人



微信联系我们



图灵教育
turingbooks



图灵访谈
ituring_interview

本书是一本优秀的编译原理入门读物，它尝试以一种现代的方式设计一种现代的语言，即使读者对编译器已有一定程度的了解，也一定能从中学到很多。

——日本编译器权威专家 中田育男

第1部分 基础篇

设计Stone语言的解释器。第2~8章将实现一个具有基本功能的解释器。第9~10章将介绍一些较深入的内容。

- 第1天 来，我们一起做些什么吧
- 第2天 设计程序设计语言
- 第3天 分割单词
- 第4天 用于表示程序的对象
- 第5天 设计语法分析器
- 第6天 通过解释器执行程序
- 第7天 添加函数功能
- 第8天 关联Java语言
- 第9天 设计面向对象语言
- 第10天 无法割舍的数组

第2部分 性能优化篇

第2部分将对第1部分设计的Stone语言解释器进行性能优化。其中，第13章将介绍如何设计Stone语言的编译器，以提高性能。如果读者仅对编译器的设计方法感兴趣，只需阅读第11章与第13章即可。

- 第11天 优化变量读写性能
- 第12天 优化对象操作性能
- 第13天 设计中间代码解释器
- 第14天 为Stone语言添加静态类型支持以优化性能

第3部分 解说篇 (自习时间)

第3部分将介绍一些在开发Stone语言过程中没能涉及的进阶主题。第15章和第16章的内容是大多语言处理器相关教材中都会讲解的基础知识。

- 第15天 手工设计词法分析器
- 第16天 语法分析方式
- 第17天 Parser库的内部结构
- 第18天 GlueJ的使用方法
- 第19天 抽象语法树与设计模式

图灵社区: iTuring.cn
热线: (010)51095186转600

分类建议 计算机/程序语言与软件开发

人民邮电出版社网址: www.ptpress.com.cn

ISBN 978-7-115-35564-5



ISBN 978-7-115-35564-5

定价: 59.00元

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育：好书、活动每日播报

微博 @图灵社区：电子书和好文章的消息

微博 @图灵新知：图灵教育的科普小组

微信 图灵访谈：[ituring_interview](#)，讲述码农精彩人生

微信 图灵教育：[turingbooks](#)