Decaf之符号表

语义分析的阶段:

建立符号表的信息，实现静态语义检查。

抽象语法树—>带标注的抽象语法树

一、符号表的组织

符号表的作用：管理符号信息

符号表的两种基本属性：**符号的名字、符号有效的作用域。**

1、 单表形式

PL/0:结构体数组

作用域的层次是用一个计数器来记录的，每进入一 个新的作用域，计数器就增 1；每退出一个作用域，计数器就减 1。

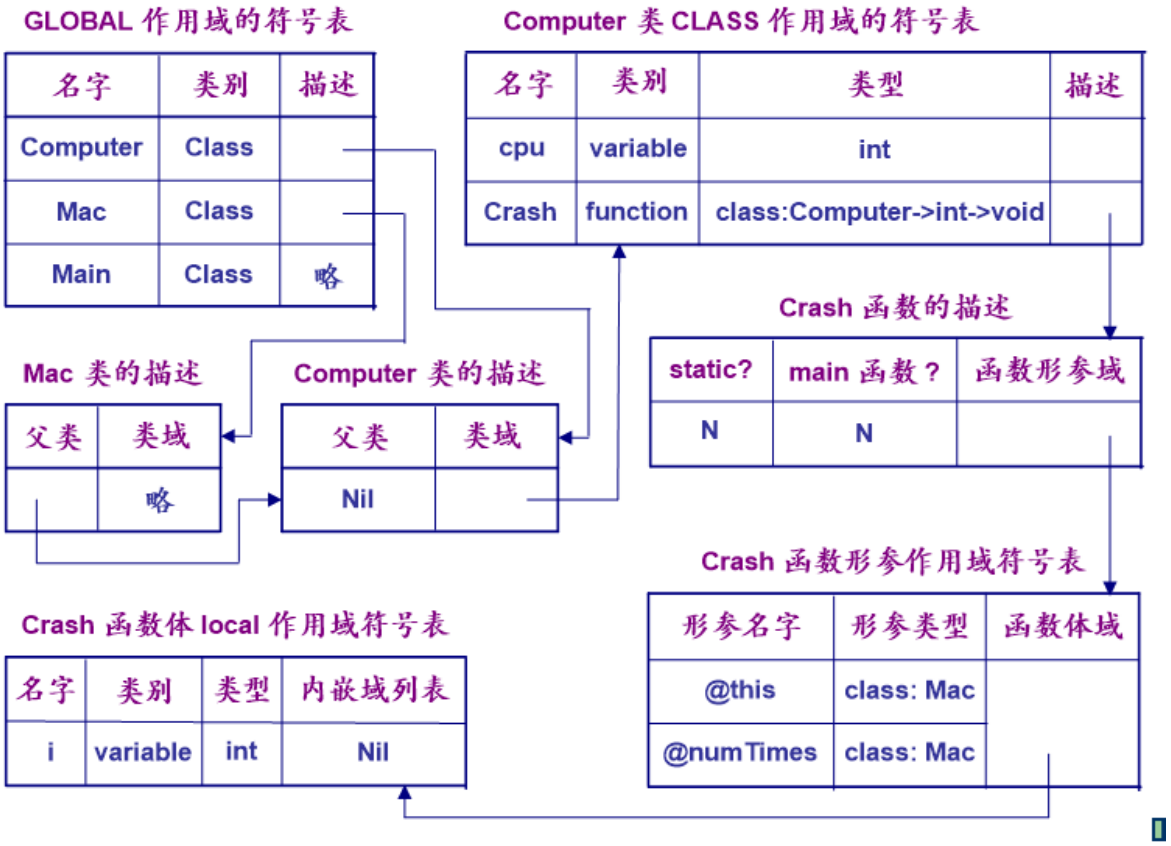
2、 多级符号表

- 为每个作用域单独建立一个符号表，仅记录当前作用域中声明的标识符。
- 同时建立一个 栈来管理整个程序的作用域：每打开一个作用域，就把该作用域压入栈中；每关闭一个作用域，就从栈顶弹出该作用域。
- 有4中类型的作用域：①全局作用域（Global）②类作用域（Class）③形参作用域（Formal）④局部作用域

对这样一个程序：

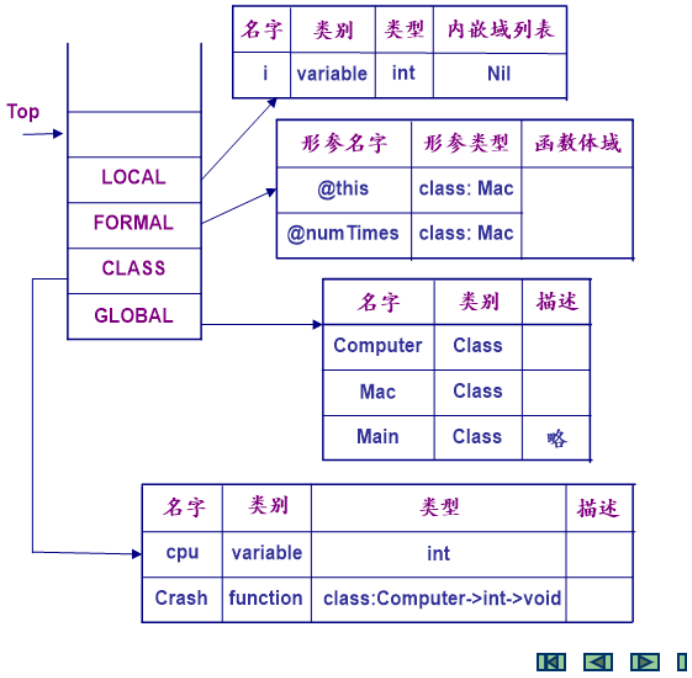
```
class Computer {
   int cpu;
   void Crash(int numTimes) {
      int i;
      for (i = 0; i < numTimes; i = i + 1)
         Print("sad\n");
   }
}

class Mac extends Computer {
   int mouse;
   void Crash(int numTimes) {
      Print("ack!");
   }
}

class Main {
   static void main() {
      class Mac powerbook;
      powerbook = new Mac();
      powerbook.Crash(2);
   }
}
```

符号表结构示意图：



程序运行到下划线语句的时候当前的作用域栈：

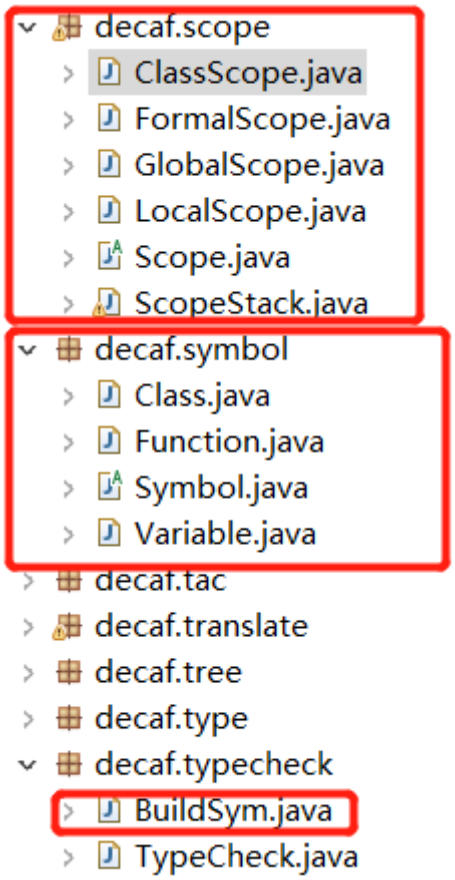
```
class Computer {
   int cpu;
   void Crash(int numTimes) {
      int i;
      for (i = 0; i < numTimes; i = i + 1)
         Print("sad\n");
   }
}

class Mac extends Computer {
   int mouse;
   void Crash(int numTimes) {
      Print("ack!");
   }
}

class Main {
   static void main() {
      class Mac powerbook;
      powerbook = new Mac();
      powerbook.Crash(2);
   }
}
```



对AST进行第一遍扫描建立符号表：（Visitor设计模式）

BuildSym类（继承自Tree.Visitor类）进行符号表的构造：

- 建立符号表的信息
- 检测符号声明冲突
- 跟声明有关的符号引用问题



符号表相关的类：

BuildSym.java | Tree.java | Class.java | Symbol.java | ScopeStack.java

```
1  package decaf.symbol;
2
3  import java.util.Comparator;
8
9  public abstract class Symbol {
10     protected String name;
11
12     protected Scope definedIn;
13
14     protected Type type;
15
16     protected int order;
17
18     protected Location location;
19
20     public static final Comparator<Symbol> LOCATION_COMPARATOR = new Comparator<Symbol>() {
21
22         @Override
23         public int compare(Symbol o1, Symbol o2) {
24             return o1.location.compareTo(o2.location);
25         }
26
27     };
28
29     public static final Comparator<Symbol> ORDER_COMPARATOR = new Comparator<Symbol>() {
30
31         @Override
32         public int compare(Symbol o1, Symbol o2) {
```

Tree.java | Class.java | ScopeStack.java | Scope.java | ClassScope.java | Loc

```java
package decaf.symbol;

import java.util.Iterator;

public class Class extends Symbol {

    private String parentName;

    private ClassScope associatedScope;

    private int order;

    private boolean check;

    private int numNonStaticFunc;

    private int numVar;

    private int size;

    private VTable vtable;

    private Label newFuncLabel;

    public Label getNewFuncLabel() {
        return newFuncLabel;
    }
```

作用域相关的类:

Tree.java | ScopeStack.java | Scope.java | ClassScope.java | FormalScope.... | LocalS

```java
package decaf.scope;

import java.util.ListIterator;

public class ScopeStack {
    private Stack<Scope> scopeStack = new Stack<Scope>();

    private GlobalScope globalScope;

    public Symbol lookup(String name, boolean through) {
        if (through) {
            ListIterator<Scope> iter = scopeStack.listIterator(scopeStack
                    .size());
            while (iter.hasPrevious()) {
                Symbol symbol = iter.previous().lookup(name);
                if (symbol != null) {
                    return symbol;
                }
            }
            return null;
        } else {
            return scopeStack.peek().lookup(name);
        }
    }
```

Tree.java ScopeStack.java Scope.java ClassScope.java FormalScope.... LocalScop

```java
package decaf.scope;

import java.util.Iterator;

public abstract class Scope {
    public enum Kind {
        GLOBAL, CLASS, FORMAL, LOCAL
    }

    protected Map<String, Symbol> symbols = new LinkedHashMap<String, Symbol>();

    public abstract Kind getKind();

    public abstract void printTo(IndentPrintWriter pw);

    public boolean isGlobalScope() {
        return false;
    }

    public boolean isClassScope() {
        return false;
    }

```

Tree.java ScopeStack.java Scope.java ClassScope.java FormalScope.... Lc

```java
package decaf.scope;

import java.util.TreeSet;

public class ClassScope extends Scope {

    private Class owner;

    public ClassScope(Class owner) {
        super();
        this.owner = owner;
    }

    @Override
    public boolean isClassScope() {
        return true;
    }

    public ClassScope getParentScope() {
        Class p = owner.getParent();
        return p == null ? null : p.getAssociatedScope();
    }

```

主函数：

```
87    private void compile() {
88
89        Tree.TopLevel tree = parser.parseFile();
90        checkPoint();
91        if (option.getLevel() == Option.Level.LEVEL0) {
92            IndentPrintWriter pw = new IndentPrintWriter(option.getOutp
93            tree.printTo(pw);
94            pw.close();
95            return;
96        }
97        BuildSym.buildSymbol(tree);
98        checkPoint();
99        TypeCheck.checkType(tree);
00        checkPoint();
01        if (option.getLevel() == Option.Level.LEVEL1) {
```

BuildSym类：

```
27  public class BuildSym extends Tree.Visitor {
28
29      private ScopeStack table;
30
31      private void issueError(DecafError error) {
32          Driver.getDriver().issueError(error);
33      }
34
35      public BuildSym(ScopeStack table) {
36          this.table = table;
37      }
38
39      public static void buildSymbol(Tree.TopLevel tree) {
40          new BuildSym(Driver.getDriver().getTable()).visitTopLevel(tree);
41      }
42
```

visitTopLevel方法：

```
43      // root
44      @Override
45      public void visitTopLevel(Tree.TopLevel program) {
46          program.globalScope = new GlobalScope();
47          table.open(program.globalScope);
48          for (Tree.ClassDef cd : program.classes) {
49              Class c = new Class(cd.name, cd.parent, cd.getLocation());
50              Class earlier = table.lookupClass(cd.name);
51              if (earlier != null) {
52                  issueError(new DeclConflictError(cd.getLocation(), cd.name,
53                          earlier.getLocation()));
54              } else {
55                  table.declare(c);
56              }
57              cd.symbol = c;
58          }
59
60          for (Tree.ClassDef cd : program.classes) {
61              Class c = cd.symbol;
62              if (cd.parent != null && c.getParent() == null) {
63                  issueError(new ClassNotFoundError(cd.getLocation(), cd.parent));
64                  c.dettachParent();
65              }
66              if (calcOrder(c) <= calcOrder(c.getParent())) {
67                  issueError(new BadInheritanceError(cd.getLocation()));
68                  c.dettachParent();
69              }
70          }
71
72          for (Tree.ClassDef cd : program.classes) {
73              cd.symbol.createType();
74          }
```

根节点

设置global作用域

遍历所有类节点

<检测符号声明冲突>

将当前这个类加入当前的开作用域中

构建起这个类节点的符号表

<跟声明有关的符号引用问题>

符号表中添加当前类型

```
75
76          for (Tree.ClassDef cd : program.classes) {
77              cd.accept(this);
78              if (Driver.getDriver().getOption().getMainClassName().equals(
79                      cd.name)) {
80                  program.main = cd.symbol;
81              }
82          }
83
84          for (Tree.ClassDef cd : program.classes) {
85              checkOverride(cd.symbol);
86          }
87
88          if (!isMainClass(program.main)) {
89              issueError(new NoMainClassError(Driver.getDriver().getOption()
90                      .getMainClassName()));
91          }
92          table.close();
93      }
94
```

遍历每个类节点，执行相应的visit方法

visitClassDef方法以及类似的重载方法：

```
95      // visiting declarations
96      @Override
97      public void visitClassDef(Tree.ClassDef classDef) {
98          table.open(classDef.symbol.getAssociatedScope());
99          for (Tree f : classDef.fields) {
100             f.accept(this);
101         }
102         table.close();
103     }
104
```

重载visit方法

入栈

遍历类定义的每个域

出栈