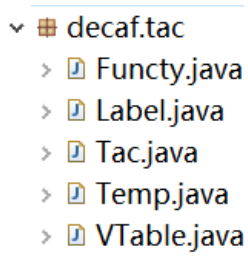
中间代码生成阶段

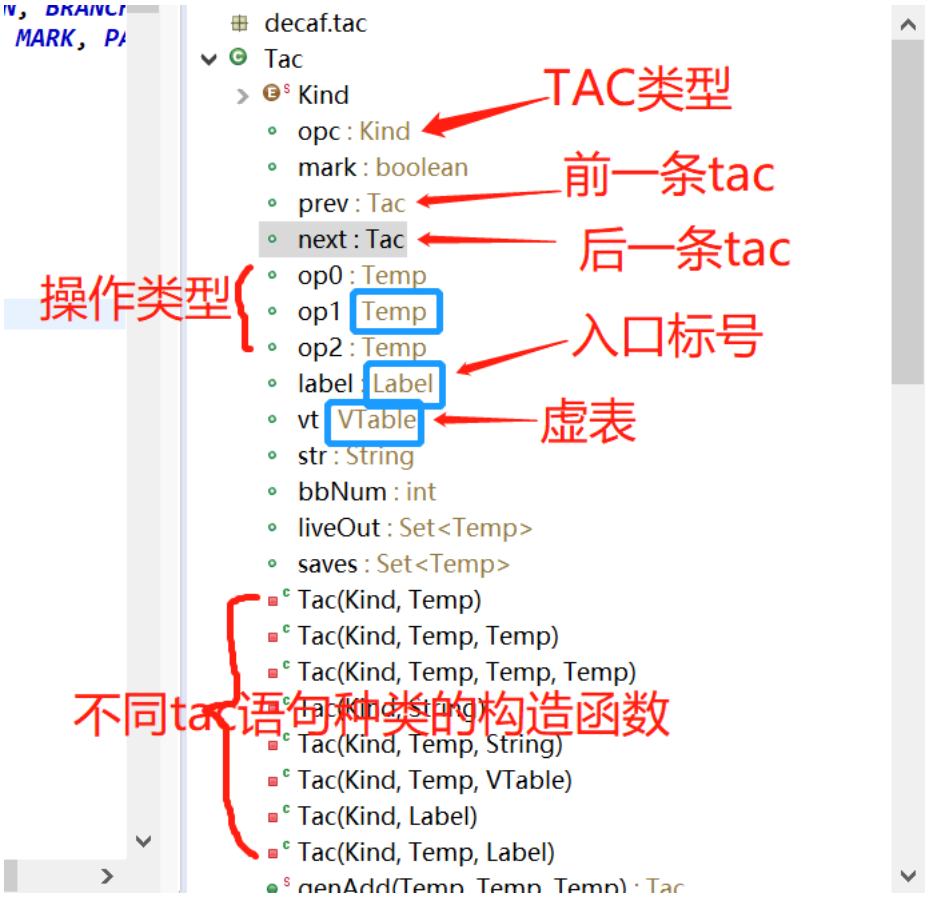
## 将 AST->TAC

TAC 结构




- **tac 包 包含 TAC 结构和 TAC 表示中用到的数据对象**

### 1. TAC 结构



2. TAC 中的几类数据对象：

---

## Temp（临时变量）

```
/*临时变量——与实际机器中的寄存器对应，表示函数的形式参数以及函数的局部变量
 *  （但是不表示类的成员变量）
 * 是用于函数体内的数据对象
*/
```

---

## Label（标号）

```
/*
*表示标号，即代码序列中的特定位置（也称为行号）。
*在实验框架中有两种标号，一种是函数的入口标号，另一种是一般的跳转目标标号
*/
```

---

## Functy（函数块）

```
/*函数块—表示源程序中的一个函数定义
 * 与符号表中的FUNCTION对象不同，functy对象并不包括函数的返回值、参数表等信息
 * 而仅包括函数的入口标号以及函数体的语句序列
*/

public class Functy {
    public Label label;      //入口/跳转目标标号

    public Tac paramMemo;    //指导命令（临时量（对应形参）的固定偏移量）

    public Tac head;     //指向tac语句头指针

    public Tac tail;     //指向tac语句尾指针

    public Function sym;     //对应function的符号表信息
}
```

每个 Functy 里的 Tac 是**链式存储结构**。由“头指针”“尾指针”就可以遍历对应 function 里所有语句生成的 Tac。

---

## VTable（类的虚函数表）

```
/* 类的虚函数表
 * 即一个存放着各虚函数入口标号的线性表
 */
public class VTable {
    public String name;      //虚表名?

    public VTable parent;    //继承的父类的虚表

    public String className;     //   类名

    public Label[] entries;      //各方法（非静态方法）的入口标号
}
```

## 3. 面向对象机制的运行时存储组织（P247）

### ——面向对象语言存储分配策略

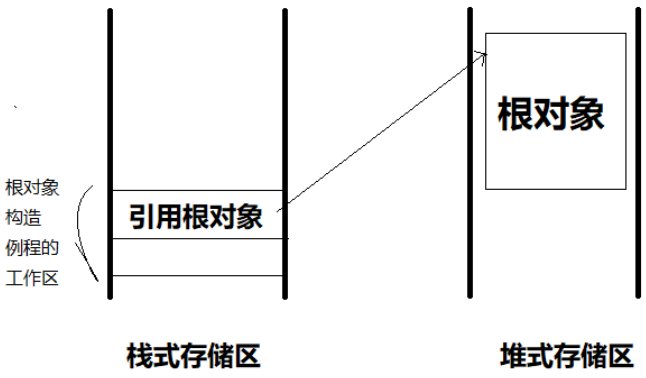
### 1. 角色

**类——程序的静态定义**

**对象——程序运行时的动态结构**

### 2. 面向对象程序运行时特征

- 对象是类的实例，是按需要创建而不是预先分配的
- 执行一个面向对象程序就是创建系统根类（Root Class）的一个实例，并调用该实例的创建过程。创建根对象就是启动 main 函数。
- 创建对象的过程实现该对象初始化；运行根对象构造例程时，在堆区为根对象申请空间并创建根对象，同时在栈区保存引用根对象的存储单元。



## 3. 对象的存储结构

对象存储的方式是：每个对象都对应一个记录对象状态的内存块（存放于堆区），其中包括对象所属类的虚表指针（位于内存块开始的位置），和用于说明对象状态的**属性变量**。

**VTable 格式**

| VTable parent----继承的父类的虚表 |
| --- |
| String className----类名 |
| Label[] entries ----类的例程（函数）入口（标号） |

注：不包含静态方法，因为静态方法的地址是固定的，可以直接调用

所以当我们需要 new 一个类的对象时，首先需要申请适当大小的堆存储空间（调用库函数_Alloc），**将第一个单元置为执行该对象所属类虚表的指针**，后续单元依次存放成员变量（先放继承变量）

Eg.

Computer 类的虚表

| Null　　（没有继承类） |
| --- |
| Computer |
| _Computer.Crash（对象方法） |

new 了一个 Computer 对象

| |
| --- |
| cpu　(属性变量) |

实现运行时函数地址绑定，即所谓的动态绑定机制。

```
class Father {
  int writeName() { print(1); ...}
  int smile() { print(2); ...}
}
class Son extends Father {
  int writeName() { print(3); }
  int laugh() { print(4); }
}
```

这里Father类定义了一个writeName()方法，而其子类Son使用新的writeName()覆盖了这个方法。然后考虑以下的代码片断：

```
class Father a;
class Son b;
class Father c;
a = new Father();
b = new Son();
c = b;
```

对象 a, b, c 分别调用 writeName() 方法，输出的结果是?

```
virtual table of Father:
+0: address of function writeName (the version prints 1)
+4: address of function smile
vitural table of Son:
+0: address of function writeName(the version prints 3)
+4: address of function smile
+8: address of function laugh
```

现在我们考虑c <- b的情况。由于Decaf的对象赋值采用引用赋值，因此这个赋值语句的效果仅仅是让c和b指向同一块内存区域。因此，按照上面的过程，当调用c.writeName()的时候，我们首先通过c所指向的内存区域找到对应的虚函数表（此时是Son的虚函数表），然后在这个虚函数表内找到writeName偏移量即+0对应的那一项。我们发现这一项对应的函数地址是打印3的那个writeName()函数的地址，因此c.writeName()的调用结果是输出3。

- **第一遍翻译**

**所需要做的工作：为每个类生成 VTable、New 函数，计算各类偏移量信息，为每个函数创建 Functy 对象，为函数形参关联 Temp 对象。**

```
@Override
//重载根结点的vivit方法
public void visitTopLevel(Tree.TopLevel program) {
    for (Tree.ClassDef cd : program.classes) {
        cd.accept(this);    //  遍历每个类定义并执行相应的visit方法
    }
    for (Tree.ClassDef cd : program.classes) {
        tr.createVTable(cd.symbol);          //为每个类生成相应的VTable
        tr.genNewForClass(cd.symbol);        //为每个类生成相应的new函数
    }
    for (Tree.ClassDef cd : program.classes) {
        if (cd.parent != null) {         //  为每个子类的VTable设置指向父类VTable的指针
            cd.symbol.getVtable().parent = cd.symbol.getParent()
                    .getVtable();
        }
    }
}
```

```
@Override
public void visitMethodDef(Tree.MethodDef funcDef) {
    Function func = funcDef.symbol;
    if (!func.isStatik()) {
        func.setOffset(2 * OffsetCounter.POINTER_SIZE + func.getOrder()
                * OffsetCounter.POINTER_SIZE);
    }
    tr.createFuncty(func);  //创建functy
    OffsetCounter oc = OffsetCounter.PARAMETER_OFFSET_COUNTER;
    oc.reset();
    int order;
    if (!func.isStatik()) {
        Variable v = (Variable) func.getAssociatedScope().lookup("this");
        v.setOrder(0);
        Temp t = Temp.createTempI4();
        t.sym = v;
        t.isParam = true;
        v.setTemp(t);
        v.setOffset(oc.next(OffsetCounter.POINTER_SIZE));
        order = 1;
    } else {
        order = 0;
    }
```

```
  for (Tree.VarDef vd : funcDef.formals) {
      vd.symbol.setOrder(order++);
      Temp t = Temp.createTempI4();
      t.sym = vd.symbol;
      t.isParam = true;
      vd.symbol.setTemp(t);
      vd.symbol.setOffset(oc.next(vd.symbol.getTemp().size));
  }
```

需要注意的是，第一遍里创建的 Functy 对象，只是初始化了每个 Functy 的入口标号，而函数体的语句序列是在第二遍翻译过程中得到的 Tac 序列。