# 语法分析之错误检测

TypeCheck对抽象语法树进行第二次扫描，检测更对的语义错误，TypeCheck这个类继承于Visitor，重写了Visitor这个类里面所有的visit方法目的是进行语义错误检测。

## 一、架构

首先看向compile这个函数里面如何使用TypeCheck这个类

```
checkPoint();
TypeCheck.checkType(tree);
```

调用TypeCheck里面的checkType这个函数

↓↓↓↓↓

再看向TypeCheck里面的checkType

```
public static void checkType(Tree.TopLevel tree) {
    new TypeCheck(Driver.getDriver().getTable()).visitTopLevel(tree);
}
```

在里面调用了new了一个新的TypeCheck的对象并调用了它的visitTopLevel这个方法

↓↓↓↓↓

再看向visitTopLevel这个方法

```
@Override
public void visitTopLevel(Tree.TopLevel program) {
    table.open(program.globalScope);
    for (Tree.ClassDef cd : program.classes) {
        cd.accept(this);
    }
    table.close();
}
```

在visitTopLevel里面又访问了所有的类并调用其accept方法起始也就是各个类的visit的方法

↓↓↓↓↓

各个类的visit里面又调用类的各个组成成分的visit

```
@Override
public void visitClassDef(Tree.ClassDef classDef) {
    table.open(classDef.symbol.getAssociatedScope());
    for (Tree f : classDef.fields) {
        f.accept(this);
    }
    table.close();
}
```

通过这样的方式古一步一步把问题细化，调用其他的函数进行处理，通过返回值或者直接将错误添加到错误列表里面。

举一个TypeCheck里面的例子：

### Call :: = [Expr.]identifier'('Actuals')'

看到这个的时候语义分析就会想到就会想到

- 检查Expr的语义
- 检查identifier是否为full
- 静态方法中不能调用非静态的函数
- 当Expr存在的时候，是否有identifier这个函数
- 形参和传入的实参是否匹配

.....

```
@Override
public void visitCallExpr(Tree.CallExpr callExpr) {
    if (callExpr.receiver == null) {
        ClassScope cs = (ClassScope) table.lookForScope(Kind.CLASS);
        checkCallExpr(callExpr, cs.lookupVisible(callExpr.method));
        return;
    }
    callExpr.receiver.usedForRef = true;
    callExpr.receiver.accept(this);
    if (callExpr.receiver.type.equal(BaseType.ERROR)) {
        callExpr.type = BaseType.ERROR;
        return;
    }
    if (callExpr.method.equals("length")) {//如果方法名叫length
        if (callExpr.receiver.type.isArrayType()) {
            if (callExpr.actuals.size() > 0) {
                issueError(new BadLengthArgError(callExpr.getLocation(),
                        callExpr.actuals.size()));
            }
            callExpr.type = BaseType.INT;
            callExpr.isArrayLength = true;
            return;
        } else if (!callExpr.receiver.type.isClassType()) {  //接受者不是一个类
            issueError(new BadLengthError(callExpr.getLocation()));
            callExpr.type = BaseType.ERROR;
            return;
        }
    }

    if (!callExpr.receiver.type.isClassType()) {
        issueError(new NotClassFieldError(callExpr.getLocation(),
                callExpr.method, callExpr.receiver.type.toString()));
        callExpr.type = BaseType.ERROR;
        return;
    }

    ClassScope cs = ((ClassType) callExpr.receiver.type)
            .getClassScope();
    checkCallExpr(callExpr, cs.lookupVisible(callExpr.method));
}
```

```
private void checkCallExpr(Tree.CallExpr callExpr, Symbol f) {
    Type receiverType = callExpr.receiver == null ? ((ClassScope) table
            .lookForScope(Scope.Kind.CLASS)).getOwner().getType()
            : callExpr.receiver.type;
    if (f == null) {                //没有那个方法
        issueError(new FieldNotFoundError(callExpr.getLocation(),
                callExpr.method, receiverType.toString()));
        callExpr.type = BaseType.ERROR;
    } else if (!f.isFunction()) {   //f并不是一个方法
        issueError(new NotClassMethodError(callExpr.getLocation(),
                callExpr.method, receiverType.toString()));
        callExpr.type = BaseType.ERROR;
    } else {
        Function func = (Function) f;
        callExpr.symbol = func;
        callExpr.type = func.getReturnType();
        if (callExpr.receiver == null && currentFunction.isStatik()
                && !func.isStatik()) {            //静态方法中不能调用非静态的函数
            issueError(new RefNonStaticError(callExpr.getLocation(),
                    currentFunction.getName(), func.getName()));
        }
        if (!func.isStatik() && callExpr.receiver != null
                && callExpr.receiver.isClass) {    //指通过非类成员变量来访问类成员
            issueError(new NotClassFieldError(callExpr.getLocation(),
                    callExpr.method, callExpr.receiver.type.toString()));
        }
        if (func.isStatik()) {
            callExpr.receiver = null;
        } else {
            if (callExpr.receiver == null && !currentFunction.isStatik()) {  //如果不是静态方法，而且receiver为null，那么就需要把receiver指为this
                callExpr.receiver = new Tree.ThisExpr(callExpr.getLocation());
                callExpr.receiver.accept(this);
            }
        }
        for (Tree.Expr e : callExpr.actuals) {  //检查实参是否存在语义错误
            e.accept(this);
        }
        List<Type> argList = func.getType().getArgList();  //形参
        int argCount = func.isStatik() ? callExpr.actuals.size() //实际传入的参数 ，如果不是静态函数那么会多一个this指针
                : callExpr.actuals.size() + 1;
        if (argList.size() != argCount) {  //对比实参和形参的数目
            issueError(new BadArgCountError(callExpr.getLocation(),
                    callExpr.method, func.isStatik() ? argList.size()
```

```
        } else {
            Iterator<Type> iter1 = argList.iterator();
            if (!func.isStatik()) {  //在访问输入参数的时候掠过this
                iter1.next();
            }
            Iterator<Tree.Expr> iter2 = callExpr.actuals.iterator();
            for (int i = 1; iter1.hasNext(); i++) {
                Type t1 = iter1.next();
                Tree.Expr e = iter2.next();
                Type t2 = e.type;
                if (!t2.equal(BaseType.ERROR) && !t2.compatible(t1)) { //实际参数的类型与形参类型不相同
                    issueError(new BadArgTypeError(e.getLocation(), i,
                            t2.toString(), t1.toString()));
                }
            }
        }
    }
```

## 针对解连锁报错这个问题：

利用 BaseType.ERROR 来表示因为出错而导致无法 推断的数据类型，一旦遇到 BaseType.ERROR 就说明前面已经报过错，不必再报了。 另外需要注意的是，当表达式中一个操作数的数据类型为 BaseType.ERROR 时，如果 这个表达式的运算结果类型只可能有一种（例如&&， >这种操作，返回值只可能是bool）， 那么我们就不需要把 BaseType.ERROR 传递开去（否则后面一些潜在的错误就会被忽 略了），因为我们知道即使程序员把前面的错误更正了，这条表达式的运算结果类型也 只可能是所规定的那种。