

Java 编译器前端的探究分析

蒋仕彪 3170102587

指导老师: 冯雁

日期: 2020 年 6 月 27 日

摘要

本文是编译原理课程的期末论文，参考了宫文学的公开课，介绍了 Java 前端编译器的成熟技术（有关编译原理的基础技术，可以在我整理的 [编译原理课程总结](#) 里看到）。作为一个成熟的高级语言，在常规的词法分析和语法分析后，Java 会进行复杂的语义分析，包括建立符号表、处理注解、属性计算、数据流分析、解除语法糖等。这些内容都在本文有一个介绍。

关键词: Java 编译 前端 词法分析 语法分析 语义分析 符号表 文法属性 数据流分析

1 简介

1.1 Java 语言

Java 是一种广泛使用的计算机编程语言，拥有跨平台、面向对象、泛型编程的特性，广泛应用于企业级 Web 应用开发和移动应用开发。

Java 不同于一般的编译语言或解释型语言。它首先将源代码编译成字节码，再依赖各种不同平台上的虚拟机来解释执行字节码，从而具有“**一次编写，到处运行**”的跨平台特性。在早期 JVM 中，这在一定程度上降低了 Java 程序的运行效率。但在 J2SE1.4.2 发布后，Java 的运行速度有了大幅提升。

1.2 自举机制

一个很有趣的事情是，Java 编译器本身也是用 Java 写的，这种现象叫做“**自举**”(Bootstrapping)。我们可以在一个 Java 程序里调用 Java 编译器，如：

```
import Javax.tools.JavaCompiler;
import Javax.tools.ToolProvider;
public class Compile MyClass {
    public static void main(String[] args) {
        JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
        int result = compiler.run(null, null, null, "MyClass.java");
        System.out.println("Compile result code=" + result);
    }
}
```

所以在接下来的章节我们会发现：**Java 的编译器源码依然是 Java 语言！**

1.3 Java 前端编译器的代码结构

`com.sun.source.tree` 存了 Java 语言的 AST 模型。举几个例子：

- ExpressionTree 指的是表达式，各种不同的表达式继承了这个接口，比如 BinaryTree 代表了所有的二元表达式
- StatementTree 代表了语句，它的下面又细分了各种不同的语句，比如，IfTree 代表了 If 语句，而 BlockTree 代表的是一个语句块。

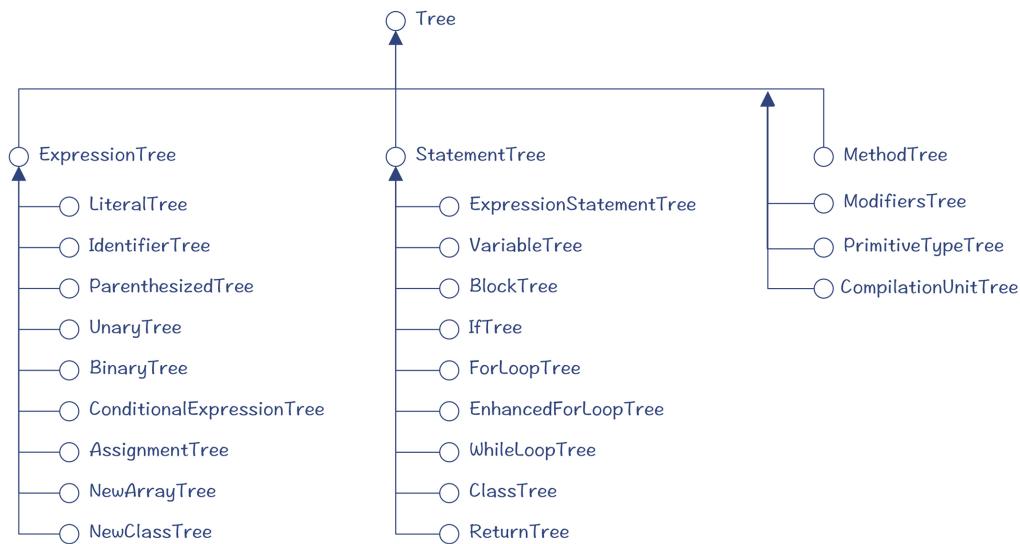


图 1: AST 节点的详细定义

`com.sun.tools.Javac.parser.Lexer` 是词法解析器接口，它可以把字符流变成一个个的 Token，具体的实现 in Scanner 和 JavaTokenizer 类中。

`com.sun.tools.Javac.parser.Parser` 是语法解析器接口，它能够解析类型、语句和表达式，具体的实现 in JavacParser 类中。

总结起来，Java 语言中与编译有关的功能放在了两个模块中：其中，`Java.compiler` 模块主要是对外的接口，而 `jdk.compiler` 中有具体的实现。

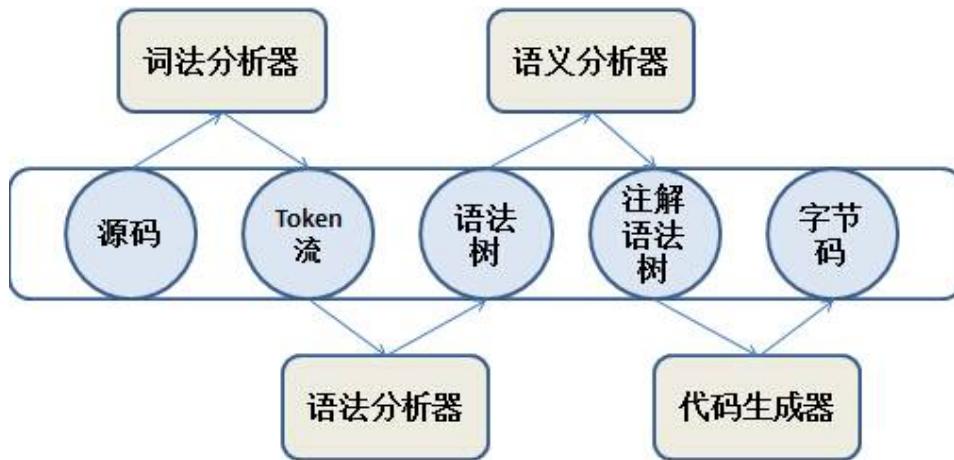


图 2: Java 的总体编译流程

2 Java 的词法分析

词法分析器的具体实现在于 JavaTokenizer 类中。其中，`readToken()` 是一个很重要的函数：

```
循环读取字符
case 空白字符
    处理，并继续循环
case 行结束符
    处理，并继续循环
case A-Za-z$_
    调用 scanIden() 识别标识符和关键字，并结束循环
case 0之后是X或x，或者1-9
    调用 scanNumber() 识别数字，并结束循环
case , ; () [] 等字符
    返回代表这些符号的 Token，并结束循环
case isSpecial(), 也就是% * + - | 等特殊字符
    调用 scanOperator() 识别操作符
...
...
```

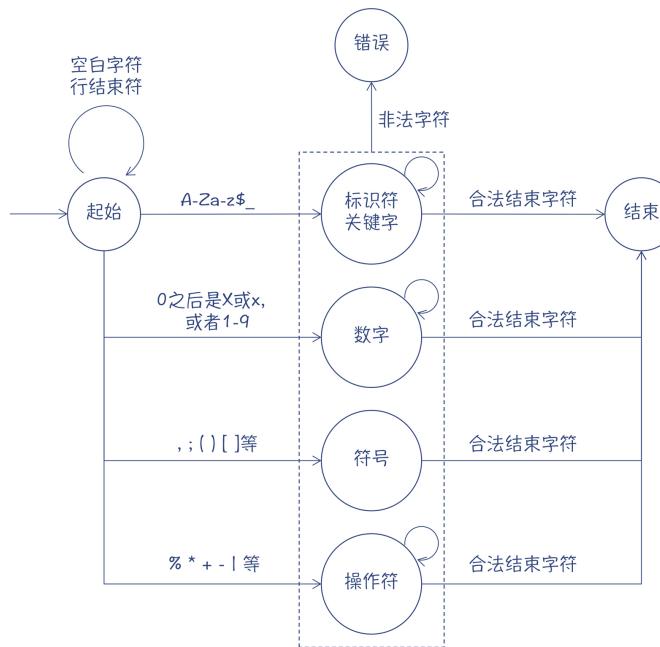


图 3: `readToken()` 的流程图示

Java 的词法解析程序在主干上是遵循有限自动机的算法的，但在很多局部的地方，为了让词法分析的过程更简单高效，采用了手写的算法。

比如**关键字和标识符的规则冲突**是词法分析的一个技术点，因为到最后我们不知道读入的是一个关键字，还是一个普通的标识符。如果单纯按照有限自动机的算法去做词法分析，我们需要对每个关键字都开若干个节点，会使有限自动机变得很复杂。

Java 编译器的处理方式比较简单，分成了两步：首先把所有的关键字和标识符都作为标识符识别出来，然后再从里面把所有预定义的关键字挑出来。这比构造一个复杂的有限自动机实现起来更简单。

3 Java 的语法分析

3.1 AST 和递归下降

跟所有的语法分析器一样，Java 的语法分析器会把词法分析器生成的 Token 流，生成一棵 AST。下面的 AST 就是 MyClass.java 示例代码对应的 AST¹。

Listing 1: MyClass.java 示例代码

```
public class MyClass {  
    public int a = 2+3;  
    public int foo(){  
        int b = a + 10;  
        return b;  
    }  
}
```

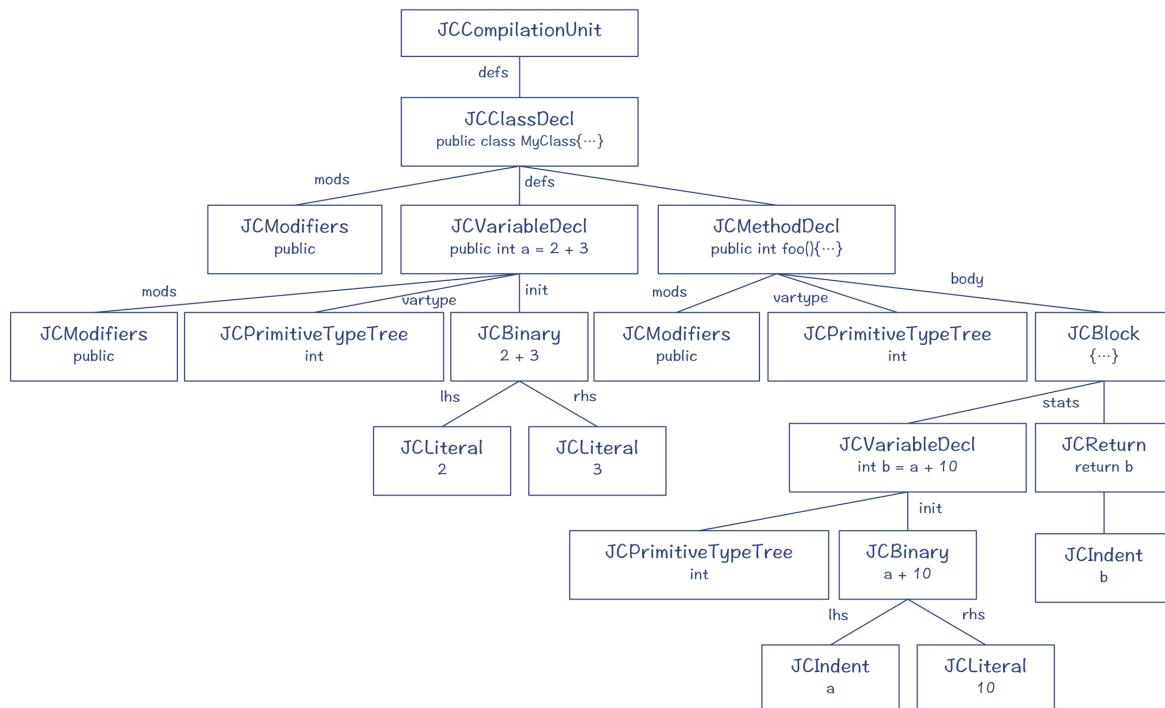


图 4: MyClass.java 对应的 AST 节点

令人惊讶的是，Java 这么成熟的语言也是采用递归下降算法来生成语法规树。当然，Java 采用了典型的消除左递归的算法。比如以下的单句文法会被改成两句。

```
add -> add + mul  
  
add -> mul add'  
add' -> + add' |
```

¹其中的 JCXXX 节点都是实现了 com.sun.source.tree 中的接口，比如 JCBinary 实现了 BinaryTree 接口，而 JCLiteral 实现了 LiteralTree 接口

3.2 优先级和结合性的处理

为了处理运算符优先级，我们通常用语法逐级嵌套的方式来修改文法。比如加法和乘法：

```
add -> mul add,  
add' -> + mul add' |  
mul -> pri mul,  
mul' -> * pri mul' |
```

但是 Java 的运算符优先级“金字塔”有十层，拆成十个 terms 会很麻烦。

Java 编译器里用到了一个先进的技术——**运算符优先级解析器(Operator-Precedence Parser)**。该算法对于多种不同优先级的操作符的表达式都能通过一个循环处理妥当。不仅保证了优先级的正确性，也不用担心左递归问题。

该算法的核心是 **term2Rest()** 函数。term2Rest 维护了一个操作数的栈 (odStack) 和一个操作符的栈 (opStack) 作为工作区。算法会根据 odStack、opStack 和后续操作符这三个信息，决定如何生成优先级正确的 AST。我们以 $2*3+4*5$ 这个表达式为例：

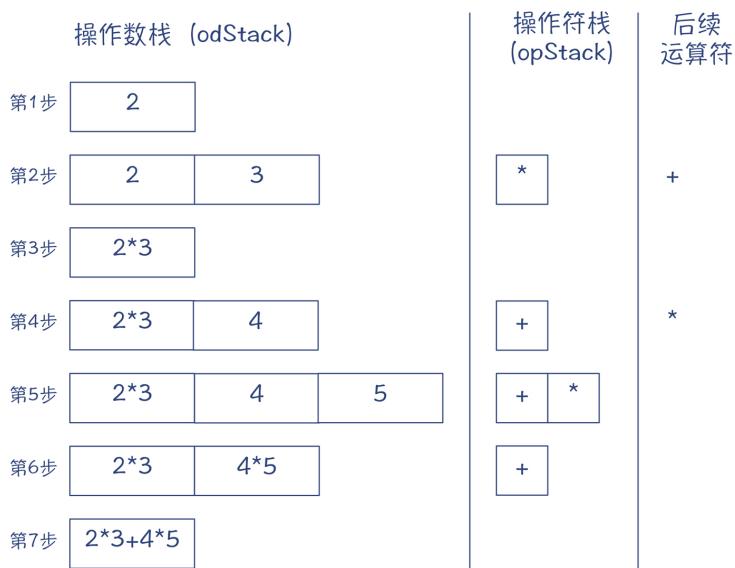


图 5：工作区处理表达式 $2*3+4*5$ 的图示

在一步一步解析的过程中，当 opStack 的栈顶运算符的优先级大于等于后续运算符的优先级时，就会基于 odStack 栈顶的两个元素创建一棵二元表达式的子树；反过来的话，栈顶运算符的优先级小于后续运算符的优先级，就会继续把操作数和操作符入栈，而不是创建二元表达式。这就可以保证，优先级高的操作符形成的子树，总会在最后的 AST 的下层，从而优先级更高。

考虑这个算法的本质：它借助一个工作区，自底向上地组装 AST。很眼熟？其实这就是一个简单 **LR 算法**。操作数栈和操作符栈是工作区，然后要向后预读一个运算符，决定是否做规约。只不过做规约的规则比较简单，依据相邻的操作符的优先级就可以了。

总结来说，Java 的语法分析总体上是**自顶向下的递归下降算法**。在解决左递归问题时，也采用了标准的改写文法的方法。但是，在处理二元表达式时，局部采用了**自底向上的运算符优先级解析器**，使得算法更简洁。

4 Java 的语义分析

`com.sun.tools.Javac.comp` 包里的 `com.sun.tools.Javac.comp.CompileStates` 类里描述了 Java 在编译阶段的每一个状态。我们会发现，除了开始的词法和语法分析，以及最终的生成字节码，中间的**八个环节**均可视为语义分析。足见其语义分析的复杂性。

Listing 2: CompileStates 里的具体状态信息

```
public enum CompileState {
    INIT(0),           // 初始化
    PARSE(1),          // 词法和语法分析
    ENTER(2),          // 建立符号表
    PROCESS(3),         // 处理注解
    ATTR(4),           // 属性计算
    FLOW(5),           // 数据流分析
    TRANSTYPES(6),      // 去除语法糖：泛型处理
    TRANSPATTERNS(7),   // 去除语法糖：模式匹配处理
    UNLAMBDA(8),        // 去除语法糖：LAMBDA 处理（转换成方法）
    LOWER(9),           // 去除语法糖：内部类、foreach 循环、断言等。
    GENERATE(10);       // 生成字节码
    ...
}
```

语法糖的存在简化了用户的编程体验，在为 Java 增色的同时，也增加了语义分析的步骤。因为不同编译器对语法糖的设计和解语法糖的操作都不太一样，所以在这篇报告里，我总结了除解语法糖外的语义分析，包括以下四个环节：

- `enterTrees()`: 对应 ENTER 阶段，它的主要工作是建立符号表。
- `processAnnotations()`: 对应 PROCESS 阶段，它的工作是处理注解。
- `attribute()`: 对应 ATTR 阶段，这个阶段是做属性计算。
- `flow()`: 对应 FLOW 阶段，这个阶段是做数据流分析。

4.1 建立符号表

Enter 阶段最重要的过程就是建立符号表。

我们知道，**符号表 (Symbol Table)** 是一种保存了程序中所有的定义信息的数据结构。不管是变量、类型，还是方法、参数，在符号表里都有一个条目。

在 `Java.compiler` 模块中定义了 Java 语言的构成元素 (**Element**)，包括模块、包、类型、可执行元素、变量元素等。这其中的每个元素，都算是一种符号。

在 `jdk.compiler` 模块中，定义了这些元素的具体实现——**符号 (Symbol)**。符号里记录了一些重要的属性信息，比如名称 (`name`)、类型 (`type`)、分类 (`kind`)、所有者 (`owner`) 等，还有一些标记位，标志该符号是否是接口、是否是本地的、是否是私有的，等等，这些信息在语义分析和后续编译阶段都会使用。另外，不同的符号还有一些不同的属性信息，比如变量符号，会记录其常数值 (`constValue`)，这在常数折叠优化时会用到。

注意，创建符号表的过程其实是对 AST 进行遍历的过程，所以 Java 采用了 Visitor 模式。

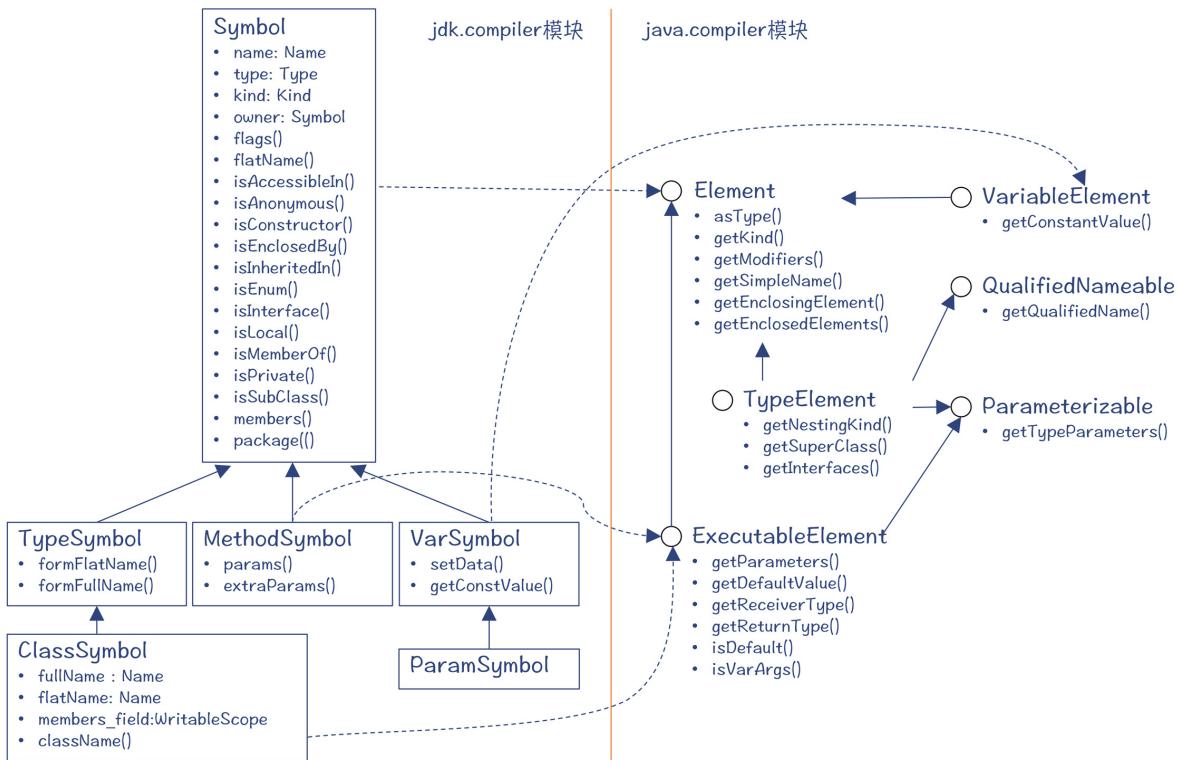


图 6: Java 中 Symbol 及其子类的设计

以之前的 MyClass.Java 为例，我们来考察为它创建符号表的 `visitVarDef()` 函数：

1. 第一行，是创建 Symbol。
2. 第二行，是把 Symbol 关联到对应的 AST 节点（这里是变量声明的节点 JCVariableDecl）。
3. 第三行，是把 Symbol 添加到 Scope 中。Java 编译器把符号被直接保存进了它所在的词法作用域，即符号表采用与作用域同构的带层次的表格。

Listing 3: visitVarDef() 处理 MyClass.Java 的一些步骤

```

...
// 创建 Symbol
VarSymbol v = new VarSymbol(0, tree.name, vartype, enclScope.owner);
...
tree.sym = v;           // 关联到 AST 节点
...
enclScope.enter(v);   // 添加到 Scope 中
...

```

其中，`com.sun.tools.Javac.code.Scope.$ScopeImpl` 类是真正用来存放 Symbol 的容器类。通过 `next` 属性来指向上一级作用域，形成嵌套的树状结构。

`Env< AttrContext>` 是一个辅助类，用来保存编译过程中的一些上下文信息，其中就有当前节点所处的作用域 (`Env.info.scope`)。

`com.sun.source.tree.Scope` 接口是对作用域的一个抽象，可以获取当前作用域中的元素、上一级作用域、上一级方法以及上一级类。

4.2 ENTER 阶段

ENTER 阶段是分两个阶段完成的。

1. 第一个阶段：只是扫描所有的类（包括内部类），建立类的符号，并添加到作用域中。但是每个类定义的细节并没有确定，包括类所实现的接口、它的父类，以及所使用的类型参数。类的内部细节也没有去扫描，包括其成员变量、方法，以及方法的内部实现。
2. 第二个阶段：确定一个类所缺失的所有细节信息，并加入到符号表中。

一个很直接的问题是：**为什么需要两个阶段？只用一个阶段不可以吗？**

在 C/C++ 体系里，如果要实现函数/类之间的相互调用，每个结构调用之前都必须进行声明。但是 Java 是支持“前面用后面定义”的代码模式的，所以必须用两阶段扫描。

前面说的符号表，保存了用户编写的程序中的符号。可是还有一些符号是系统级的，可以在不同的程序之间共享，比如原始数据类型、Java.lang.Object 和 Java.lang.String 等基础对象、缺省的模块名称、顶层的包名称等。Java 编译器在 Symtab 类中保存这些系统级的符号。系统符号表在编译的早期就被初始化好，并用于后面的编译过程中。

4.3 PROCESS 阶段

注解是 Java 语言中的一个重要特性，也是 Java 元编程能力²的重要组成部分。在 Java 编译器中，注解被看作是符号的元数据，用 SymbolMetadata 类记录对应符号的注解信息。

下面来举一个具体的例子：我们新建一个叫做 HelloWorld 的注解。

Listing 4: 定义注解 HelloWorld

```
@Retention(RetentionPolicy.SOURCE) // 注解用于编译期处理
@Target(ElementType.TYPE)           // 注解是针对类型的
public @interface HelloWorld {
}
```

针对这个注解，需要写一个注解处理器 HelloWorldProcessor.java（当编译器在处理该注解的时候，就会调用该注解处理器）。它里面的主要逻辑是获取被注解的类的名称，比如说叫 Foo，然后生成一个 HelloFoo.java 的程序。这个程序里有一个 sayHello() 方法，能够打印出“Hello Foo”。如果被注解的类是 Bar，那就生成一个 HelloBar.java，并且打印“Hello Bar”。

然后我们就可以在 Foo 类里调用该注解了。

Listing 5: 调用注解功能的 Foo 类

```
@HelloWorld
public class Foo {
    // HelloFoo 类是处理完注解后才生成的。
    static HelloFoo helloFoo = new HelloFoo();
    public static void main(String args[]){
        helloFoo.sayHello();
    }
}
```

² 所谓元编程，简单地说，就是用程序生成或修改程序的能力。

4.4 ATTR 阶段

Java 的属性分析定义在 com.sun.tools.Javac.comp.Attr 类里，主要分为四个步骤：

1. Check：做类型检查。
2. Resolve：做名称的消解，也就是对于程序中出现的变量和方法，关联到其定义。
3. ConstFold：常量折叠，比如提前计算出“2+3”这种在编译期就可以计算出结果。
4. Infer：用于泛型中的类型参数推导。

4.4.1 类型检查

在类型检查之前，我们要先明确一点：Java 有哪些类型？

Java.compile 模块里定义了 Java 的语言模型，其中有一个包对 Java 的类型体系做了设计：

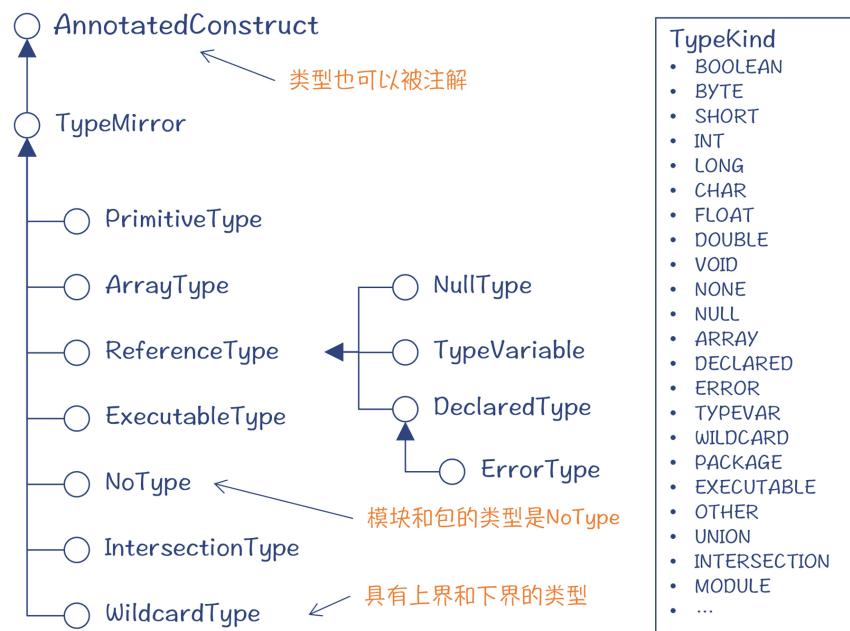


图 7: Java 的类型体系

模块和包的类型是 **NoType**，而方法的类型是 **可执行类型 (ExecutableType)**。而且刻画一个可执行类型是比较复杂的，需要 5 个要素：

1. **returnType**: 返回值类型；
2. **parameterTypes**: 参数类型的列表；
3. **receiverType**: 接收者类型，也就是这个方法是定义在哪个类型（类、接口、枚举）上的；
4. **thrownTypes**: 所抛出异常的类型列表；
5. **typeVariables**: 类型参数的列表。

我们可以和 C 语言做个比较，它只需要这个列表的前两项：返回值类型和参数类型。

通过一个接口体系来刻画类型还是不够细致，Java 又提供了一个 **TypeKind** 的枚举类型，把某些类型做进一步的细化，比如原始数据类型进一步细分为 **BOOLEAN**、**BYTE**、**SHORT** 等。这种设计方式可以减少接口的数量，使类型体系更简洁。

对于 `int a = "Hello"` 这样的语句，Java 的类型检查过程分了四步，如下图所示：

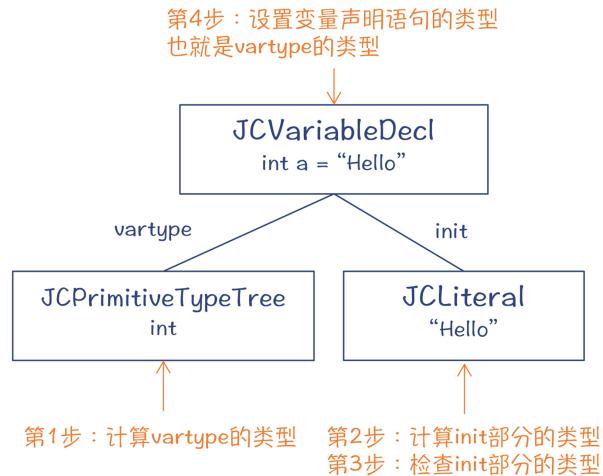


图 8: Java 类型检查的步骤

1. **计算 vartype 子节点的类型。** 这一步是在把 a 加入符号表的时候（MemberEnter）就顺便一起做了（调用的是“Attr.attribType() 方法”）。计算结果是 int 型。
2. **在 ATTR 阶段正式启动以后，深度优先地遍历整棵 AST，自底向上计算每个节点的类型。** 自底向上是 S 属性的计算方式。此时能知道 init 部分的类型是字符串（Java.lang.String）。
3. **检查 init 部分的类型是否正确。** 这个时候，比对的就是 vartype 和 init 这两棵子树的类型。具体实现是在 Check 类的 checkType() 方法，这个方法要用到下面这两个参数：
 - **final Type found :** “发现”类型，也就是“Hello”字面量的类型，这里的值是 Java.lang.String。这个是自底向上计算出来的，属于 S 属性。
 - **final Type req :** “需要”类型，这里的值是 int。也就是说，a 这个变量需要初始化部分的类型是 int 型的。这个变量是自顶向下传递下来的，属于 I 属性。
4. **继续自底向上计算类型属性。** 会把变量声明语句 JCVariableDecl 的类型设为 vartype。

Java 的类型检查，其实就是所需类型（I 属性）和实际类型（S 属性）的比对。如果发现类型不匹配，就记录下错误信息。下面展示的是在做类型检查时整个的调用栈：

```

JavaCompiler.compile()
-> JavaCompiler.attribute() -> Attr.attrib()
-> Attr.attribClass()          // 计算 TypeCheck 的属性
-> Attr.attribClassBody()
-> Attr.attribStat()          // int a = "Hello";
-> Attr.attribTree()           // 遍历声明成员变量 a 的 AST
-> Attr.visitVarDef()          // 访问变量声明节点
-> Attr.attribExpr(TCTree, Env, Type) // 计算 "Hello" 的属性，并传入 vartype 的类型
-> Attr.attribTree()           // 遍历 "Hello" AST，所需类型信息在 ResultInfo 中
-> Attr.visitLiteral()          // 访问字面量节点，所需类型信息在 resultInfo 中
-> Attr.check()                // 把节点的类型跟原型类型（需要的类型）做比对
-> Check.checkType()           // 检查跟预期的类型是否一致
  
```

4.4.2 引用消解

所谓引用消解 (Reference resolution)，就是当我们在程序中用到一个变量的时候，必须知道它确切的定义在哪里。具体到 Java 编译器，引用消解实际上就是把标识符的 AST 节点关联到正确的 Symbol 的过程。

引用消解不仅仅针对变量，还针对类型、包名称等各种用到标识符的地方。引用消解的实现思路很清晰：编译器在 Enter 阶段已经建立了作用域的嵌套结构。那么在这里，编译器只需要沿着这个嵌套结构逐级查找就行了。

com.sun.tools.Javac.comp.Resolve 类的 findIdentInternal 方法是对几种不同的符号做引用消解的入口。以下列举部分代码：

```
if (kind.contains(KindSelector.VAL)) {    // 变量消解
    sym = findVar(env, name);
    ...
}
if (kind.contains(KindSelector.TYP)) {    // 类型消解
    sym = findType(env, name);
    ...
}
if (kind.contains(KindSelector.PCK))        // 包名称消解
    return lookupPackage(env, name);
```

对于 `int b = a + f` 这个变量声明语句，在查找变量 `a` 时，沿着 Scope 的嵌套关系往上查找两级就行。但对于变量 `f`，还需要沿着类的继承关系，在符号表里找到父类（或接口），从中查找有没有名称为 `f` 的成员变量。

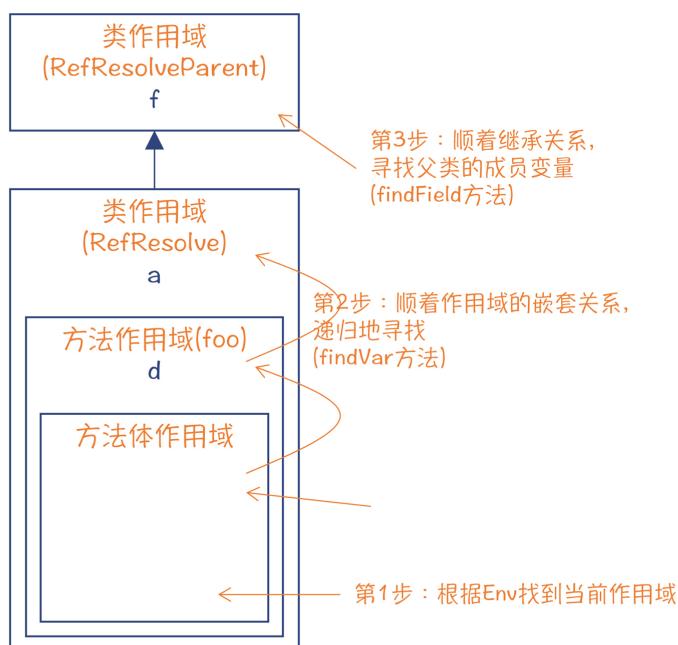


图 9: Scope 寻找的层级示意图

4.4.3 常数折叠

在 ATTR 阶段，还会做一项优化工作：**常数折叠（Constant Fold）**。我们知道，优化工作通常是在编译器的后端去做的。Javac 编译器作为前端编译器，需要保证字节码是比较优化的，减少解释执行的消耗。因为常数折叠借助属性计算就可以实现，所以在 ATTR 阶段顺便就把这个优化做了。

Java 在什么情况下会做常数折叠呢？我们来看看下面这个例子。变量 a 和 b 分别是一个整型和字符串型的常数。这样的话， $c=b+a*3$ 中 c 的值，是可以在编译期就计算出来的。这要做两次常数折叠的计算，最后生成一个 Hello 6 的字符串常数。

```
public class ConstFold {
    public String foo(){
        final int a = 2;           //int 类型的常数
        final String b = "Hello";  //String 类型的常数
        String c = b + a * 3;      //发生两次折叠
        return c;
    }
}
```

触发上述常数折叠的代码，在 com.sun.tools.Javac.comp.Attr 类的 **visitBinary()** 方法中，具体实现是在 com.sun.tools.Javac.comp.ConstFold 类。它的计算逻辑是：针对每个 AST 节点的 type，可以通过 Type.constValue() 方法，看看它是否有常数值。如果二元表达式的两个子节点都有常数值，那么就可以做常数折叠，计算出的结果保存在父节点的 type 属性中。

常数折叠实质上是针对 AST 节点的常数值属性来做属性计算的。

4.4.4 推导类型参数

ATTR 阶段做的最后一项工作是对泛型中的**类型参数推导**。在 Java 语言中（很多语言不支持这一点），如果在前面声明了一个参数化类型的变量，那么在后面的初始化部分里不带这个参数化类型也是可以的（编译器会自动推断出来）。

比如用户可以把第一行代码简化成第二行代码。

Listing 6: 类型参数推导前后举例

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
Map<String, List<String>> myMap = new HashMap<>();
```

类型推断是一个非常复杂的过程，充满了**启发式（Heuristic）**的思想。比如：

- 如果某类型变量只在方法参数列表或返回值的一处被调用了，可直接根据调用方法时传递的实际类型或方法返回值的类型来确定泛型方法的参数类型。
- 当某个类型变量在方法的参数列表和返回值中被多次利用了，而且在调用方法时这多处的实际类型又是一样的，那么这也可以很明显的知道此泛型方法的参数类型。
- 当某个类型变量在方法的参数列表和返回值中被多次利用了，而且在调用方法时这多处的实际类型又对应不同的类型，且返回值是 void，那么这时取多处实际变量类型的**最大交集**。

4.5 FLOW 阶段

Java 编译器在 FLOW 阶段做了四种数据流分析：活性分析、异常分析、赋值分析和本地变量捕获分析。这里主要介绍活性分析方法。

分析方法	英文	代码位置	说明
活性分析	Liveness Analysis	LiveAnalyzer	检查每个语句是否都是可到达的。
异常分析	Exception Analysis	FlowAnalyzer	检查每个异常要么被捕捉，要么被继续抛出。 这个分析器用到了活性分析的一个结果， <code>finallyCanCompleteNormally</code> 字段。
赋值分析	Assignment Analysis	AssignAnalyzer	有两个赋值分析：确定的赋值分析 (Definite Assignment Analysis, DA)，检查每个变量在使用之前是否已经被赋值，Java 要求在读取一个变量的值之前，它必须至少被赋值过一次；确定的未赋值分析 (Definite Unassignment Analysis, DU)，检查 final 类型的变量不会被赋值 1 次以上。 <code>AssignAnalyzer</code> 依赖活性分析的结果。它还能分析出哪些变量实质上是 final 的 (effectively-final)，也就是只被赋值了一次。
本地变量捕获分析	Local Variable Capture Analysis	CaptureAnalyzer	确保被内部类和 Lambda 所访问的本地变量，要么是 final 的，要么实质上是 final 的。它要在赋值分析之后进行。

图 10: Java 的四种数据流分析

什么是活性分析方法呢？举一个 Return 的例子。下面这段代码里，foo 函数的返回值是 int，而函数体中，只有在 if 条件中存在一个 return 语句。这样，代码在 IDE 中就会报编译错误，提示缺少 return 语句。想要检查是否缺少 return 语句，我们就要进行活性分析。

```
public class NoReturn{
    public int foo(int a){
        if (a > 0){
            return a;
        }
    }
}
```

活性分析的具体实现是在 Flow 的一个内部类 LiveAnalyzer 中，编译器用了一个 alive 变量来代表代码是否会执行到当前位置。

如果方法体里有正确的 return 语句，那么扫描完方法体以后，alive 的取值是 “DEAD”，也就是这之后不会再有可执行的代码了；否则就是 “ALIVE”，这意味着 AST 中并不是所有的分支，都会以 return 结束。

在下面的代码示例中，当递归下降地扫描到 if 语句的时候，只有同时存在 then 的部分和 else 的部分，并且两个分支的活性检查的结果都是 “DEAD”，也就是两个分支都以 return 语句结束的时候，if 节点执行后 alive 就会变成 “DEAD”，也就是后边的语句不会再被执行。除此之外，都是 “ALIVE”，也就是 if 后边的语句有可能被执行。

```

public void visitIf(JCIf tree) {
    scan(tree.cond);           // 扫描 if 语句的条件部分
    // 扫描 then 部分。如果这里面有 return 语句，alive 会变成 DEAD
    scanStat(tree.thenpart);
    if (tree.elsepart != null) {
        Liveness aliveAfterThen = alive;
        alive = Liveness.ALIVE;
        scanStat(tree.elsepart);
        // 只有 then 和 else 部分都有 return 语句，alive 才会变成 DEAD
        alive = alive.or(aliveAfterThen);
    } else { // 如果没有 else 部分，那么把 alive 重新置为 ALIVE
        alive = Liveness.ALIVE;
    }
}

```

再进一步，活跃性分析还可以检测不可到达的语句。

一个简单的例子是，如果我们在 return 语句后面再加一些代码，那么这个时候，alive 已经变成“DEAD”，编译器就会报“语句不可达”的错误。

当然，Java 编译器还能检测更复杂的情况。比如在下面的例子中，a 和 b 是两个 final 类型的本地变量，final 修饰词意味着这两个变量的值已经不会再改变。

```

public class Unreachable{
    public void foo(){
        final int a=1;
        final int b=2;
        while(a>b){
            // a>b 的值可以在编译期计算出来
            System.out.println("Inside while block");
        }
        System.out.println("Outside while block");
    }
}

```

这种情况下，在扫描 while 语句的时候，条件表达式 $a > b$ 会被计算出来（是 false），这意味着 while 块内部的代码不会被执行。这种优化叫做稀疏有条件的常数折叠。

还有一种代码不可达的情况，就是无限循环后面的代码。在上面的例子中，如果把 while 条件的“ $a > b$ ”改成“ $a < b$ ”，编译器会扫描 while 里面有没有合适的 break 语句（通过 resolveBreaks() 方法）。如果找不到，就意味着这个循环永远不会结束，那么循环体后面的语句就永远不会到达，从而导致编译器报错。