



## 06 | 怎么支持条件语句和循环语句？

2021-08-20 宫文学

《手把手带你写一门编程语言》

课程介绍 >



讲述：宫文学

时长 17:30 大小 16.03M



你好，我是宫文学。

我们现在的语言已经支持表达式、变量和函数了。可是你发现没有，到现在为止，我们还没有支持流程控制类的语句，比如条件语句和循环语句。如果再加上这两类语句的话，我们的语言就能做很复杂的事情了，甚至你会觉得它已经是一门比较完整的语言了。

那么今天，我们就来加上条件语句和循环语句。在这个过程中，我们会加深对作用域和栈帧的理解，包括跨作用域的变量引用、词法作用域的概念，以及如何在运行时访问其他作用域的变量。这些知识点会帮助你加深对计算机语言的运行机制的理解。



而这些理解和认知，会有助于我们后面把基于 AST 的解释器升级成基于字节码的解释器，也有助于我们理解编译成机器码后的运行时机制。

好了，首先我们先从语法层面支持一下这两种语句。

## 语法分析：支持一元表达式

按照惯例，我们首先要写下新的语法规则，然后使用 LL 算法来升级语法分析程序。新的语法规则如下：

[复制代码](#)

```
1 ifStatement
2     : If '(' expression ')' statement (Else statement)?
3     ;
4 forStatement
5     :For '(' expression? ';' expression? ';' expression? ')' statement
6     ;
7 statement:
8     : block
9     | functionDecl
10    | variableStatement
11    | expressionStatement
12    | returnStatement
13    | ifStatement
14    | forStatement
15    | emptyStatement
16    ;
```

你从上面的语法可以得到这几个信息：

首先，if 语句中，else 部分是可选的。这样，我们在解析完 if 条件后面的语句以后，要去看看后面跟着的是不是 'else' 关键字，从而决定是否解析 else 后面的语句块。更具体的你可以参见 [parseIfStatement](#) 函数的代码。

第二，在 for 循环语句中，for 括号里用分号分割的三个表达式都是可选的，在解析的时候也要根据 Follow 集合来判断是否需要解析这三个表达式。这点你具体可以参见 [parseForStatement](#) 函数的代码。

最后，从 statement 的语法规则中，我们也可以发现，我们的语言所支持的语句越来越多了，这也使得语言特性越来越丰富了。

现在，升级我们的语法解析程序，对你来说已经没有太大的困难了，你可以参照我的参考实现动手自己做一下。

不过，为了实现 for 语句，我们还有一个语言特性需要升级一下，这就是对一元运算的支持。

哪些是一元运算呢？比如，在 for 语句中，我们经常会使用下面的写法：

```
1 for(i = 0; i < 10; i++)
```

[复制代码](#)

其中 `i++` 就是使用了一元运算。在这里，为了方便，我们干脆就让程序支持所有的一元运算！

一元运算符除了 `++` 以外，还有 `-`、`~`、`!` 等。甚至还有更复杂一点的情况，`+` 号和 `-` 号除了作为二元运算符以外，还可以作为一元运算符使用，比如下面这个例子：

```
1 myAge = +myAge + -10;
```

[复制代码](#)

你甚至可以将多个一元运算符叠加使用，比如我们把上面的例子修改一下，仍然和原来的计算结果相同：

```
1 myAge = + +myAge + - - -10;
```

[复制代码](#)


注意了，这里面的两个 `+` 号或两个 `-` 号之间是留有空格的，否则就会被词法分析程序识别成 `++` 和 `-` 了。

上面的示例程序有一些负数，比如 `-10`。在这里，`-10` 是一个表达式，由一个符号和一个字面量构成，而不是把 `-10` 整体上作为一个字面量。

在第二节介绍词法分析的时候，我曾经留了个思考题，问像 -3 这样的负数，是识别成 - 号和 3，还是把 -3 整体作为一个 Token 识别出来？

答案是前者。原因是，如果不这样处理，2-3 这样的表达式就会识别错。在词法分析阶段，我们只需要把 - 号作为 Token 识别出来就行了，至于它是作为一元运算符还是二元运算符来使用，那就交给语法分析程序来处理吧！

为了支持一元运算，我们需要把与表达式相关的语法规则升级一下：

 复制代码

```
1 expression: assignment;
2 assignment: binary (assignmentOp binary)* ;
3 binary: unary (binOp unary)* ;
4 unary: primary | prefixOp unary | primary postfixOp ;
5 primary: StringLiteral | DecimalLiteral | IntegerLiteral | functionCall | '('
6 prefixOp = '+' | '-' | '++' | '--' | '!' | '~';
7 postfixOp = '++' | '--';
```

在我们升级后的语法规则中，二元表达式能够分解成一元表达式，而一元表达式，又分成下面这三种情况。


第一种情况下，一元表达式就是一个基础表达式。

第二种情况是一个右递归的语法规则，可以由一个前缀一元运算符再加一个一元表达式组成，所以我们前面的 “+ +myage” 的表达式是正确的。

第三种情况，是一个基础表达式后面再跟上后缀一元运算符。后缀一元运算只有 ++ 和 - 两个，并且是不允许递归的，也就是说，像 “myage++ ++” 这样的表达式就是错误的。

具体解析一元表达式的代码你可以参见 [🔗 parseUnary 函数](#)。

我们可以试着用支持一元运算符的解析器解析下面的示例程序，这个示例程序中有很多个 + 号和 - 号，你第一眼看上去可能很难判断出它们分别属于哪个语法成分。

 复制代码

```
1 //示例代码：example_unary.ts
```

```

2 let myAge:number = 18;
3 myAge = + +myAge++ + - - -10;
4 println("mvAge="+mvAge):

```

但基于我们的语法规则，我们的解析器能够准确地分析出每个 + 号或 - 号所属的语法成分。比如，对于第二个语句，我们的解析器输出的 AST 是下图这个样子的，其中 1 个 + 号属于加法表达式，有两个 + 号和 3 个 - 号的都属于前缀一元运算符，而 ++ 号则属于后缀运算符。

```

ExpressionStatement
  Binary:Assign ← 最顶层的表达式是一个赋值表达式
    Variable: myAge, resolved
    Binary:Plus ← 下一级是一个加法表达式
      Prefix Unary:Plus ← 第1个+号, 前缀
        Prefix Unary:Plus ← 第2个+号, 前缀
          PostFix Unary:Inc ← ++, 后缀
            Variable: myAge, resolved
          Prefix Unary:Minus ← 第1个-号, 前缀
            Prefix Unary:Minus ← 第2个-号, 前缀
              Prefix Unary:Minus ← 第3个-号, 前缀
                10

```

那到此为止，我们的语法分析功能就升级完毕了。接下来，我们顺着已经养成的习惯，继续来看看这两个语句在语义方面有没有带来新的内容。

## 语义分析：深化对作用域的理解

你会发现，在程序里用上 if 语句和 for 语句以后，一个直观的表现就是出现了很多的语句块，而这些语句块是能够影响作用域的。我们把这些语句块所构成的作用域，叫做**块作用域**。

在早期的 JavaScript 版本是不支持块作用域的，比如，在下面的程序中，我们在块的外面仍然可以访问变量 i，并且打印出 i 的值。

```

1 function foo(){
2   {
3     var i = 4;
4   }
5   console.log(i);
6 }

```

[复制代码](#)

```
7 foo();
```

但在 ES6 版本以后，JavaScript 也支持了块作用域。你可以用 `let` 关键字在块中声明变量，这个变量的作用域仅限于当前块，如果从作用域外面访问，编译器就会报错。

[复制代码](#)

```
1 function foo(){
2   {
3     let i = 4;
4   }
5   console.log(i); //报编译错误
6 }
7 foo();
```

在这种情况下，即使是块中变量的名称与块外的变量名相同也没关系。你看，在下面这个示例程序中，`if` 语句块中的 `myAge` 和外面的 `myAge` 就是两个不同的变量，甚至类型都不相同。

[复制代码](#)

```
1 function bar() {
2   let myAge = 18;
3   if (myAge < 30) {
4     let myAge = '';
5     myAge = 'young';
6     console.log("myAge, inside: " + myAge);
7   }
8   console.log("myAge, outside: " + myAge);
9 }
```

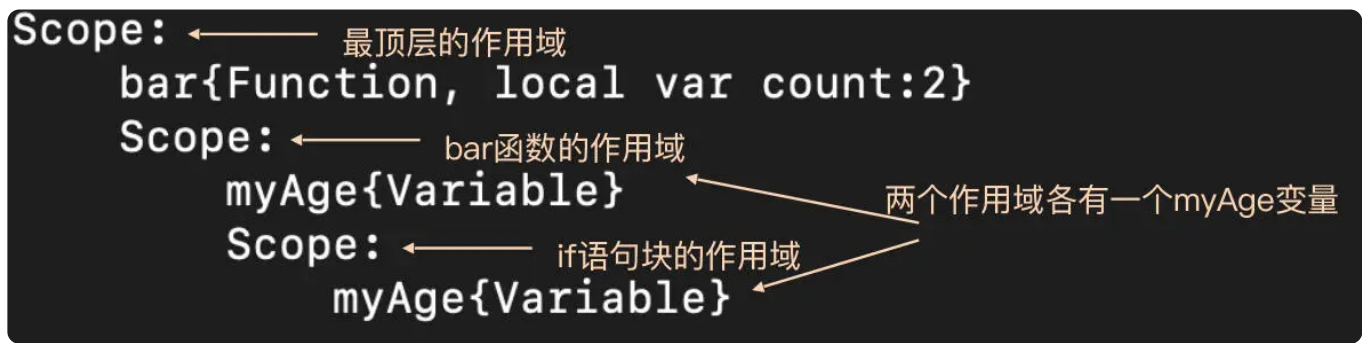
这个程序是可以正常运行的，在 `if` 块的内部和外部，都可以正常的打印 `myAge` 的值，但使用的是两个不同作用域中的变量。

你也看到了，块作用域的机制使得我们在语句块中使用变量名称的时候可以更加自由，不用担心在块中声明的变量会影响到块外面，从而减少了程序出错的概率，也让程序员在书写逻辑的时候更加自由。

而且，上面的示例程序在编译过程中会形成下面这个 `Scope` 结构。你会看到，每个 `Block` 都形成了自己的作用域，所有这些作用域构成了一个树状结构。每个作用域，可以看到在



该作用域中声明的符号，以及上面各级的符号，但看不到兄弟作用域和下级作用域的符号。



你要注意，if 语句的条件部分和 if 下面的块其实属于两个作用域。在 if 条件中的 myAge 变量，指的是外面的这个 myAge，它是 number 型的，因此我们把它用在 “myAge<30” 这样一个条件表达式里。

读到这里，你可能产生一个疑问：在语句块中，如果在声明新的 myAge 之前，是否也可以像在 if 的条件表达式里那样来使用外部的 myAge 变量呢？我们可以试一下，把程序改成下面的样子，看看会产生什么结果。

复制代码

```

1 function bar() {
2   let myAge = 18;
3   if (myAge < 30) {
4     myAge = 30;
5     let myAge = '';
6     myAge = 'young';
7   }
8 }

```

你会发现，如果我们用 Typescript 编译，编译器会报错，说块中的 myAge 在没有初始化的时候，就被访问了：

```

→ 06 node example02.js
/Users/richard/compilers/craft/06/example02.js:8
    myAge = 30;
      ^

ReferenceError: Cannot access 'myAge' before initialization
    at foo (/Users/richard/compilers/craft/06/example02.js:8:15)
    at Object.<anonymous> (/Users/richard/compilers/craft/06/example02.js:17:1)

```

这个例子说明了，在 TypeScript 中，如果存在和外部同名的变量，那你只能引用块中声明的变量，是没有办法引用外部变量的，无论块中声明的变量的位置是靠前还是靠后。

那是不是所有的语言，对于块作用域都是这样规定的呢？不是的，你可以用不同的语言试验一下。

首先是 Java 语言。Java 语言也支持块定义域，但不允许块中的变量与外部变量名称相同。

[复制代码](#)

```
1 public static int bar(){
2     int myAge = 18;
3     if (myAge < 30) {
4         string myAge = "young"; //错误：变量名称重复
5         int yourAge = 20;      //块中的变量
6     }
7     else{
8         int yourAge = 10;      //第二个块中可以有同名的变量
9     }
10    return myAge + yourAge;    //错误：找不到yourAge
11 }
```

再看看 C 语言。C 语言允许在块中声明新的变量名称之前，引用外部的变量，也就是说，上面的示例代码如果用 C 语言改写一下，也是完全合法的。

[复制代码](#)

```
1 void bar(){
2     int myAge = 18;
3     if (myAge < 30) {
4         myAge = 30;          //给外层的myAge赋值
5         const char* myAge = "young"; //新声明一个myAge
6         printf("myAge=%s\n", myAge); //打印内层的myAge的值
7     }
8     printf("myAge=%d\n", myAge);    //打印外层的myAge的值
9 }
```

那么，对于你日常使用的语言，在块作用域的语义上有什么不同呢？你也可以分析一下。



按照这节课对作用域的剖析方法，下次你再见到一段代码的时候，你可以迅速在大脑中划分出清晰的作用域的范围，形成作用域的树。作为语言的实现者，你也会对语言的特性有更加清晰的了解，这算是你学习这门课的额外收获吧！

那么，上面这些不同语言对于作用域的不同规定，会如何影响到语义分析程序呢？我们来看一下。

在前面的课程中，我们做语义分析的时候，分成了清晰的两个阶段：Enter 阶段是建立符号表，Resolve 阶段是去做引用消解。如果存在块内外变量同名的情况，在块中的变量引用，都会指向在块中声明的变量。

这种处理方法，对于 C 语言是不行的。使用上述方法，会导致第一个 myAge 引用的变量出错。也就是说，我们在做引用消解的时候，必须考虑语句的顺序，要注意我们在块中声明同名称的 myAge 之前，引用的实际是块外面的变量。

这就导致为 C 语言编写引用消解的程序要更复杂一些，建立符号表的工作和引用消解的工作必须是针对每一条语句同步去进行的。随着对每一个语句的扫描，符号表逐渐建立起来后，引用消解时总是指向最近的那一个声明。

我们现在的处理方法，用来处理 Java 程序，其实也是行不通的。在 Java 的块里，虽然不允许变量跟外部变量重名，但也必须是先声明后使用，所以也只能一边建符号表，一边做引用消解，这样才能发现在声明之前使用变量的错误。

那么，我们现在的处理方法，针对 TypeScript 或 JavaScript（指 ES6 以上的版本）是不是就没有问题了呢？

其实还是有问题的，现在的算法没有办法检查出 myAge 在声明在前就被引用的错误，并像在 Node.js 里运行那样报错。


那要如何解决这个问题呢？其实只要把问题分析清楚了，解决起来也不是太难。

在我们的语义分析程序中，我们先用 Enter 类对 AST 做了一遍遍历，把包括变量的每个符号都正确地加入到了符号表中，也就是由 Scope 形成的一个层次数据结构。接下来，我们

用 `RefResolver` 类对 AST 再做一次遍历，来建立引用消解，但在这次遍历中，我们用了一个临时的数据结构，来保存当前已经声明了的变量。

这样做的目的，是在引用消解的时候，我们既要看该变量是否属于该作用域，又要看当前该变量是否被声明了。如果该变量不属于当前作用域，那么我们就去引用外部作用域中的变量；而如果该变量属于当前作用域，那它必须在变量声明之后才能使用。

比如，对于下面程序中的 `"myAge = 30"` 这一行，编译器会认为这个 `myAge` 是块中声明的那个 `string` 类型的 `myAge`，而不是外部作用域中的 `myAge`。同时，由于目前块中的 `myAge` 还没有声明，如果提前使用它是错误的。

 复制代码

```
1 function bar() {  
2     let myAge = 18;  
3     if (myAge < 30) {  
4         myAge = 30; //编译错误  
5         let myAge = '';  
6         myAge = 'young';  
7     }  
8 }
```

我们可以画一张图来表现当编译器遍历到 `"myAge=30;"` 这个语句的时候，相关内部状态信息的情况，以及编译器是如何决策的。

```

function bar() {

    let myAge = 18;      ← 函数体作用域中的变量: {myAge}, 已声明的变量: {}

    if (myAge < 30) {    ← 函数体作用域中的变量: {myAge}, 已声明的变量: {myAge}

        myAge = 30;      ← 块作用域中的变量: {myAge}, 已声明的变量: {}

        let myAge = '';  ← 块作用域中的变量: {myAge}, 已声明的变量: {}
        检查出语义错误!

        myAge = 'young'; ← 块作用域中的变量: {myAge}, 已声明的变量: {myAge}

    }                    ← 块作用域中的变量: {myAge}, 已声明的变量: {myAge}

}                        ← 函数体作用域中的变量: {myAge}, 已声明的变量: {myAge}

```



你看，在遍历 AST 的过程中，我们的语义分析程序会不断更新一个集合中的值，这个集合就是“已声明的变量”。同时，我们的语义分析程序还会知道当前作用域中变量的集合的值。对比这两个集合的值，程序就会发现那些在声明之间就被使用的变量，并报错。

这种随着程序的执行流程，去动态计算一些数据的值，并根据这些值来做分析的方法，叫做**数流分析框架**。数据流分析框架在做语义分析和代码优化的时候都很有用，我们后面还会见到它的更多用途。

好了，理解了原理，写代码就简单了，你可以参考 [RefResolver](#) 类中我给出的示例代码。

不过，关于引用消解，我还要再补充两个语义规则，这两个规则我们也要在算法中有所体现：

第一，在 TypeScript 中，函数的声明和引用的顺序是不受限制的。也就是说，完全可以声明在后，使用在前。所以，在引用消解程序中，处理变量的消解算法和函数的消解的算法是不同的。

第二，在同一个作用域中，不可以有名称相同的符号，变量和函数的名称也不可以冲突。所以，我在 Scope 类的设计中，使用了一个以名称为 key，以符号为 value 的 Map 对象来保存该作用域中的符号，就隐含了名称唯一的要求。

好了，关于语义分析工作，我们就到这里。接下来，我们继续完成这节课的工作，让我们的解释器也能支持 if 语句和 for 语句。

## 升级解释器

首先看 if 语句。在解释器里，我们实现了 visitIfStatement() 方法，用于执行 if 语句。你可以看看下面的这个示例代码：

[复制代码](#)

```
1  /**
2   * 执行if语句
3   * @param ifStmt
4   */
5  visitIfStatement(ifStmt:IfStatement):any{
6      //计算条件
7      let conditionValue = this.needLeftValue(this.visit(ifStmt.condition));
8      //条件为真，则执行then部分
9      if (conditionValue){
10         return this.visit(ifStmt.stmt);
11     }
12     //条件为false，则执行else部分
13     else if (ifStmt.elseStmt !=null){
14         return this.visit(ifStmt.elseStmt);
15     }
16 }
```

这个逻辑是很清晰的。首先我们来计算 if 条件的值，如果为真，则执行 if 后面的语句或语句块；如果为假，则执行 else 后面的语句或语句块。

实现 for 循环的思路也差不多，遵循 for 语句的语义来运行就行了，你可以参考下面的示例代码：

[复制代码](#)

```
1  /**
2   * 执行for语句
3   * @param forStmt
4   */
5  visitForStatement(forStmt:ForStatement):any{
6      //执行init
7      if(forStmt.init !=null){
8          this.visit(forStmt.init);
9      }
10 }
```

```
11 //计算循环结束的条件
12 let notTerminate = forStmt.termination == null ? true : this.visit(forStmt
13 while(notTerminate){
14     //执行循环体
15     let retVal = this.visit(forStmt.stmt);
16     //处理循环体中的Return语句
17     if (typeof retVal == 'object' && ReturnValue.isReturnValue(retVal)){
18         // console.log("is ReturnValue!!")
19         return retVal;
20     }
21
22     //执行递增部分
23     if (forStmt.increment!=null){
24         this.visit(forStmt.increment);
25     }
26
27     //执行循环判断
28     notTerminate = forStmt.termination == null ? true : this.visit(forStmt
29 }
30 }
```

其中，循环的初始化部分在一开始执行的，而且只执行一次，接着我们就要计算循环的终止条件。如果不满足终止条件，则开始进入循环。每次循环，首先执行循环体，然后执行递增部分的语句，最后再检查一遍循环退出条件。

你可以写几个例子来测试 if 语句和 for 循环语句是否能正确运行。在这里，我提供了一个计算斐波那契数列的示例程序给你参考：

[复制代码](#)

```
1 function fibonacci(n:number):number{
2     if (n <= 1){
3         return n;
4     }
5     else{
6         return fibonacci(n-1) + fibonacci(n-2);
7     }
8 }
9
10 for (let i:number = 0; i< 32; i++){
11     println(fibonacci(i));
12 }
```

这个程序不仅演示了 if 语句和 for 循环，还演示了函数的递归调用。运行这个程序，就会打印出一个斐波那契数列，如下图所示：

通过 AST 解释器运行程序：

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
10946
17711
```



```
28657
46368
75025
121393
196418
317811
514229
832040
1346269
```

## 课程小结

好了，到这里我们今天这节课就讲完了，我们来简单回顾一下。

这节课我们增加了对流程控制类语句的支持。在支持了 if 语句和 for 循环语句之后，我们的语言特性已经很丰富了，我们已经可以用这些特性编写很复杂的程序了，比如生成斐波那契数列的示例程序。

在这个过程中，**最重要的知识点是对块作用域的理解**。不同的语言在块作用域上的特性是不同的，所以我们要采用不同的算法来做引用消解。

TypeScript 允许在块中声明新的变量覆盖外部作用域中的变量，块中所有该名称的变量都会引用这个新变量，但必须先声明再使用。TS 的这个特点，就要求我们的算法要采用数据流分析框架，在遍历 AST 的过程中，知道某个变量在当前代码的位置是否已经被声明过了。你先把数据流分析框架记住，它非常有用，我们后面还会有很多场景要用到它。

此外，我们为了支持 For 循环，还增加了对一元运算符的支持。像 ++ 和 - 这样的运算符，在求值和未来生成字节码方面都有一些特殊性，你可以多注意示例代码对它们的处理。

## 思考题

今天的思考题，我想问一下，对于 for 循环语句来说，一般包含几层的作用域？为什么？你可以运行一个例子，看看打印出来的符号表的层次，验证一下你的想法，欢迎在留言区留言。

感谢你和我一起学习编程语言，也欢迎你将这门课分享给更多对编程语言感兴趣的朋友。我是宫文学，我们下节课见。

## 资源链接

[🔗 这节课的示例代码在这里！](#)

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

👍 赞 0    💡 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇    05 | 函数实现：是时候让我们的语言支持函数和返回值了

下一篇    07 | 怎么设计属于我们自己的虚拟机和字节码？

## 精选留言 (1)

💬 写留言



崔伟协

2021-08-20

似乎需要重新编译ts到js,执行node play fib.ts才能成功

展开 ∨

