



下载APP



05 | 函数实现：是时候让我们的语言支持函数和返回值了

2021-08-18 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 15:14 大小 13.96M



你好，我是宫文学。

不知道你还记不记得，我们在 [第一节](#) 就支持了函数功能。不过那个版本的函数功能是被高度简化了的，比如，它不支持声明函数的参数，也不支持函数的返回值。

在上一节课实现了对变量的支持以后，我们终于可以进一步升级我们的函数功能了。为什么要等到这个时候呢？因为其实函数的参数的实现机制跟变量是很类似的。

为了升级我们的函数功能，我们需要完成几项任务：



1. 参考变量的机制实现函数的参数机制；


2. **支持在函数内部声明和使用本地变量**，这个时候，我们需要能够区分函数作用域和全局作用域，还要能够在退出函数的时候，让本地变量的生命期随之结束；
3. **要支持函数的返回值。**

你可以想象到，在实现了这节课的功能以后，我们的语言就越来像样了。你甚至可以用这个语言来实现一点复杂的功能了，比如设计个函数，用来计算圆的周长、面积什么的。

好吧，让我们赶紧动手吧。首先，像上节课一样，我们还是要增强一下语法分析功能，以便解析函数的参数和返回值、并支持在函数内部声明本地变量。

增强语法分析功能

我们原来的函数声明的语法是比较简陋的，现在，我们采用一下 TypeScript 完整的函数声明语法。采用该语法，函数可以有 0 到多个参数，每个参数都可以指定类型，就像变量一样，并且还可以指定函数返回值的类型。

 复制代码

```
1 //函数声明，由'function'关键字、函数名、函数签名和函数体构成。
2 functionDeclaration
3     : 'function' Identifier callSignature '{' functionBody '}';
4
5 //函数签名，也就是参数数量和类型正确，以及函数的返回值类型正确
6 callSignature
7     : '(' parameterList? ')' typeAnnotation?
8     ;
9
10 //参数列表，由1到多个参数声明构成。
11 parameterList : parameter (',' parameter)* ;
12
13 //参数，由参数名称和可选的类型标注构成
14 parameter : Identifier typeAnnotation? ;
15
16 //返回语句
17 returnStatement: 'return' expression? ';' ;
```

采用该规则以后，你可以声明一个像下面的函数，比如，你给这个函数传入圆的半径的值，它会给你计算出圆的面积：

 复制代码

```
1 //计算圆的面积
```

```
2 function circleArea(r : number):number{
3   let area : number = 3.14*r*r;
4   return area;
5 }
6 let r:number =4;
7 println("r=" + r + ", area="+circleArea(r));
8 r = 5;
9 println("r=" + r + ", area="+circleArea(r));
```

好了，修改好语法规则以后，我们就按照该语法规则来升级一下语法分析程序，跟 [04 讲](#) 一样，我们同样需要计算一下相关元素的 First 和 Follow 集合。在这里，我就不再演示计算 First 集合和 Follow 集合了，而是把它们留到了思考题的部分，让你自己来计算一个语法成分的 Follow 集合，这样能让你对 LL 算法理解得更加深入。

这里，我贴上几个代码片段，更完整的代码，你可以阅读 [parser.ts](#)。

[复制代码](#)

```
1 //解析语句
2 parseStatement():Statement{
3   let t = this.scanner.peek();
4   //根据'function'关键字，去解析函数声明
5   if (t.code == Keyword.Function){
6     return this.parseFunctionDecl();
7   }
8   //根据'return'关键字，解析return语句
9   else if (t.code == Keyword.Return){
10    return this.parseReturnStatement();
11  }
12  //...
13 }
14
15 //解析函数签名
16 parseCallSignature():CallSignature{
17   let beginPos = this.scanner.getNextPos();
18   //跳过 '('
19   let t = this.scanner.next();
20
21   let paramList = null;
22   if (this.scanner.peek().code != Seperator.CloseParen){ //')'
23     paramList = this.parseParameterList();
24   }
25
26   //看看后面是不是 ')'
27   t = this.scanner.peek();
28   if (t.code == Seperator.CloseParen){ //')'
29     //跳过 ')'

```

```
30     this.scanner.next();
31
32     //解析typeAnnotation
33     let theType:string = 'any';
34     if (this.scanner.peek().code == Seperator.Colon){ //':'
35         theType = this.parseTypeAnnotation();
36     }
37     return new CallSignature(beginPos,this.scanner.getLastPos(),paramList,
38 }
39 else{
40     console.log("Expecting a ')' after for a call signature");
41     return new CallSignature(beginPos,this.scanner.getLastPos(),paramList,
42 }
43 }
```

做完语法分析后，按照惯例，我们还是要再迭代一下语义分析程序。在这一节中，我们在语义分析环节，开始接触作用域的概念。

语义分析：作用域

我们在学习任何一门语言的时候，都会涉及到作用域的概念。

作用域就是变量能够起作用的代码范围。当我们声明一个变量的时候，这个变量起作用的范围是有限的，比如，在一个函数体内声明的变量，在函数之外就不能引用了。区分了作用域，我们就能保护函数内部的变量的值不会被外部的代码所改变。同时，我们在函数外面使用变量的时候，也不用担心跟函数内部的变量名称冲突。

上一节课，我们已经建立了符号表，并且能够存储每个变量的值。可是，我们当时并没有区分变量的作用域，也没有限制局部变量的生存期。所以，对于下面的程序，由于函数内部和外部都有一个相同名称的变量 `a`，可能就会出现错误：

```
1 function foo(){
2     //局部变量a
3     let a:number = 3;
4 }
5
6 //全局变量a
7 let a:number;
8 println(a);    //打印出3来。
```

[复制代码](#)

所以，在语义分析阶段，我们要区分开不同变量的作用域。目前我们只需要支持全局作用域和函数内部的作用域两种就可以了，我们后面还会针对语句块、类等引入更多的作用域。

在具体实现上，我们需要修改符号表的设计，引入一个作用域类，也就是 Scope。这样，全局的符号和每个函数的符号就可以分别保存在各自的 Scope 对象中，也就不会冲突了。

另外，你要注意，作用域是一层套一层，形成一个树状结构的，比如，函数的作用域就是全局作用域的子作用域。所以，我们在 Scope 的属性中，能够发现作用域所形成的层级结构。

[复制代码](#)

```
1 export class Scope{
2     //以名称为key存储符号
3     name2sym:Map<string,Symbol> = new Map();
4     //上级作用域
5     enclosingScope: Scope|null; //顶级作用域的上一级是null
6     //...
7 }
```

我们再把建立符号表的程序更新一下（参见 [Enter 类](#)），重点看一下 visitBlock 方法。

[复制代码](#)

```
1 visitBlock(block:Block):any{
2     //创建下一级scope
3     let currentScope = new Scope(this.scope);
4     block.scope = currentScope;
5
6     // 修改当前的Scope
7     this.scope = currentScope;
8
9     //调用父类的方法，遍历所有的语句
10    super.visitBlock(block);
11
12    //重新设置当前的Scope
13    if (this.scope.enclosingScope != null){
14        this.scope = this.scope.enclosingScope;
15    }
16    return currentScope;
17 }
```

注意，我们创建新的作用域，都是在遇到 Block 的时候。因为每个函数的函数体都是一个 Block，所以会确保每个函数都对应一个新的 Scope。这样的话，在函数体中声明的变量，就会添加到函数的作用域中，而不是全局作用域。

[复制代码](#)

```
1  /**
2   * 把变量声明加入符号表
3   * @param functionDecl
4   */
5  visitVariableDecl(variableDecl : VariableDecl):any{
6      //重复变量声明的检查
7      if (this.scope.hasSymbol(variableDecl.name)){
8          console.log("Duplicate symbol: " + variableDecl.name);
9          return;
10     }
11     //把变量加入当前的符号表
12     let sym = new VarSymbol(variableDecl.name, variableDecl.theType, variableDecl)
13     this.scope.enter(variableDecl.name, sym);
14
15     //把本地变量也加入函数符号中，可用于后面生成代码
16     this.functionSym?.vars.push(sym);
17 }
```

更新了建立符号表的程序以后，我们再更新一下引用消解的程序（参见 [RefResolver](#) 类）。在下面的示例程序中，你会注意到，程序是沿着作用域的层级结构，逐级查找符号的。这是因为，TypeScript 或 JavaScript 中的函数，允许访问函数外面声明的变量。

[复制代码](#)

```
1  /**
2   * 变量引用消解
3   * @param variable
4   */
5  visitVariable(variable: Variable):any{
6      //从当前作用域逐级向上查找，确定该变量的符号
7      let symbol = (this.scope as Scope).getSymbolCascade(variable.name);
8      if (symbol != null && symbol.kind == SymKind.Variable){
9          variable.sym = symbol as VarSymbol;
10     }
11     else{
12         console.log("Error: cannot find declaration of variable " + variable.name);
13     }
14 }
```

好了，通过更新建立符号表的程序和引用消解的程序，我们可以把函数的作用域跟全局作用域区分开了。我们现在可以针对这节课一开篇的那个 `circleArea` 示例程序运行一下语义分析程序，会输出下面的符号表：

```
Scope of _main
  circleArea{Function, local var count:2}
  r{Variable}
  Scope of circleArea
    r{Variable}
    area{Variable}
```

你会看到，现在作用域已经分成了两级：主函数 (`_main`) 和 `circleArea` 函数。两级作用域都有一个本地变量 `r`，其中 `circleArea` 中的 `r` 是函数参数。这证明我们划分不同变量的作用域的努力是成功的。

不过，虽然区分了变量的作用域，我们还需要给函数内的本地变量设置正确的生存期。在函数运行结束以后，它的本地变量所占据的内存就应该被回收，避免造成内存使用上的浪费，甚至导致内存泄漏。

为了正确管理本地变量的生存期，我们需要更新当前的解释器，并引入一个栈帧 (Stack Frame) 的机制。

解释器：实现栈帧

在现代语言中，本地变量基本上都是通过栈来管理的，栈又是由一个个的栈帧组成的，所以函数的调用层次，就体现在了栈帧上。

每个函数对应着一个栈帧，在调用一个函数时，就会往栈里压入一个新的栈帧，用来保存支撑该函数运行的相关信息。在退出函数时，该栈帧就会被弹出。这样，随着函数的调用和退出，栈就会不停地伸缩。

我们以下面的示例程序为例：

```
1 function foo(){
```

[复制代码](#)


```
2   some statement
3 }
4
5 function bar(){
6   foo();
7 }
8 bar();
```

当我们在主程序中调用 `bar` 的时候，`bar` 又会调用 `foo`，在程序运行过程中栈帧的变化如下：



图1：示例程序运行时栈帧的变化


栈帧保存着与一个函数正确运行有关的各种信息，其中最重要的就是本地变量的值以及参数的值，其他还有返回值等信息。不过，对于不同的语言来说，栈帧的具体设计可以是不同的，但基本原理是一样的。

在某些编译原理的教科书上，你还会看到“活动记录 (Activation Record)”这样一个概念。它跟栈帧的意思是差不多的。

只不过，栈帧有时候指的是更加物理层面的设计。当程序以本地代码的形式运行时，为了提高性能，我们会把尽量多的数据放在寄存器，而不是放在内存的栈帧里。

而相对来说，活动记录是逻辑意义上一个函数运行过程中所需要维护的状态信息，不管放在栈帧里还是寄存器里，它们都属于该函数的活动记录。


好，回到我们自己的解释器上来。我们通过引入栈帧这么一个数据结构（参见 [StackFrame 类](#)），通过动态的构建和释放栈帧，我们就能管理好本地变量的生存期。栈帧的设计也很简单，包括变量的值和返回值，你可以看看下面这个代码：

 复制代码

```
1  /**
2   * 栈帧
3   * 每个函数对应一级栈帧.
4   */
5  class StackFrame{
6      //存储变量的值
7      values:Map<string, any> = new Map();
8
9      //返回值，当调用函数的时候，返回值放在这里
10     retVal:any = undefined;
11 }
```

由于我们目前的运行时是基于 node.js 实现的，所以当我们释放栈帧时，实际上是通过 V8 的垃圾收集机制回收内存的。如果我们把运行时改为用 C 语言单独实现，就可以实时释放了。

接着再修改我们的解释器（参见 [Interpreter 类](#)），让它支持栈帧。这其中，最关键的就是调用函数的程序，它要负责栈帧的创建和释放。请看下面的参考实现：

 复制代码

```
1  /**
2   * 运行函数调用。
3   * 原理：根据函数定义，执行其函数体。
4   * @param functionCall
5   */
6  visitFunctionCall(functionCall:FunctionCall):any{
7      if (functionCall.name == "println"){ //内置函数
8          return this.println(functionCall);
9      }
10
11     if(functionCall.sym != null){
12         //清空返回值
13         this.currentFrame.retVal = undefined;
14
15         //1.创建新栈帧
16         let frame = new StackFrame();
17         //2.计算参数值，并保存到新创建的栈帧
18         let functionDecl = functionCall.sym.node as FunctionDecl;
19         if (functionDecl.callSignature.paramList != null){
20             let params = functionDecl.callSignature.paramList.params;
21             for (let i = 0; i < params.length; i++){
22                 let variableDecl = params[i];
23                 let val = this.needLeftValue(this.visit(functionCall.arguments
24                     frame.values.set(variableDecl.name, val); //设置到新的frame里。
```

```

25         }
26     }
27
28     //3.把新栈帧入栈
29     this.pushFrame(frame);
30
31     //4.执行函数
32     this.visit(functionDecl.body);
33
34     //5.弹出当前的栈帧
35     this.popFrame();
36
37     //6.函数的返回值
38     return this.currentFrame.retVal;
39 }
40 else{
41     console.log("Runtime error, cannot find declaration of " + functionCal
42     return;
43 }
44 }

```

在上面这个示例程序中，你会看到调用一个函数的完整的过程，包括传递参数的过程，以及在调用函数前后将栈帧入栈和出栈的过程。

并且，在退出函数的时候，如果我们执行了一个 Return 语句，并且带有返回值，那么该返回值就会被设置到上一级栈帧中。

[复制代码](#)

```

1  /**
2   * 处理Return语句时，要把返回值封装成一个特殊的对象，用于中断后续程序的执行。
3   * @param returnStatement
4   */
5  visitReturnStatement(returnStatement: ReturnStatement):any{
6      let retVal:any;
7      if (returnStatement.exp != null){
8          retVal = this.getLeftValue(this.visit(returnStatement.exp));
9          this.setReturnValue(retVal);
10     }
11     return new ReturnValue(retVal); //这里是传递一个信号，让Block和for循环等停止执行
12 }
13
14 //把返回值设置到上一级栈帧中（也就是调用者的栈帧）
15 private setReturnValue(retVal:any){
16     let frame = this.callStack[this.callStack.length-2];
17     frame.retVal = retVal;
18 }


```

不过，关于 Return 语句，我们还有一个重要的机制需要注意。当程序遇到 Return 语句的时候，后面的代码就不执行了，直接退出函数。

可是，我们当前的解释器，是通过遍历 AST 来实现的。那么，缺省我们是会遍历每棵子树，也就是执行每个语句。可是，return 语句又要求程序跳过某些语句，这应该如何实现呢？

我给你一个解决方案。上面的示例代码中，在 visitReturnStatement 方法的最后一句，你会看到示例代码中返回了一个 ReturnValue 对象。这里，我相当于做了一个特殊的标记。在遍历树的时候，告诉上一级函数，这里遇到了一个 return 语句，这样就可以跳过后面的语句了。

我们再来看一下 visitBlock 中的代码，当检测出某个语句返回的是 ReturnValue 对象的时候，就会中断该 Block 的执行。并且，你还看到，visitBlock 接着把这个 ReturnValue 对象往外抛，这是考虑到 Block 会嵌套的情况。在下一节，当我们使用 if 语句和 for 循环语句的时候，就会遇到这种情况。但是，不管嵌套了多少个 Block，程序的执行流程都会一直往外跳，直到整个当前函数停止运行。

 复制代码

```
1  /**
2   * 遍历一个块
3   * @param block
4   */
5  visitBlock(block:Block):any{
6      let retVal:any;
7      for(let x of block.stmts){
8          retVal = this.visit(x);
9          //如果当前执行了一个返回语句，那么就直接返回，不再执行后面的语句。
10         //如果存在上一级Block，也是中断执行，直接返回。
11
12         if (typeof retVal == 'object' &&
13             ReturnValue.isReturnValue(retVal)){
14             return retVal;
15         }
16     }
17     return retVal;
18 }
```

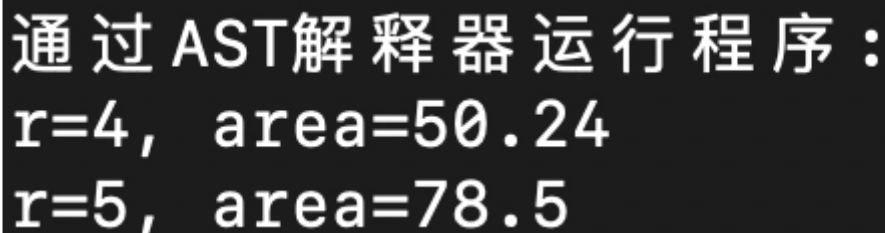
在这里，你也会看到，基于 AST 的解释器在处理像 return 这样的流程跳转语句的时候，其实是比较困难的。其他流程跳转语句还包括 break、continue 这种，处理起来也都比较麻烦。这也是后面我们会引入一个新的解释器——基于字节码的解释器的原因之一。

好了，到目前为止，我们这节课的任务就完成了。我们的函数可以支持参数和返回值了，成为了真正意义上的函数。现在，你就可以试着写几个函数，来验证一下我们语言的功能了，比如，我运行了一下下面的示例程序，用来打印不同半径的圆的面积：

 复制代码

```
1 function circleArea(r : number):number{
2   let area : number = 3.14*r*r;
3   return area;
4 }
5 let r:number =4;
6 println("r=" + r + ", area="+circleArea(r));
7 r = 5;
8 println("r=" + r + ", area="+circleArea(r));
```

运行该程序，可以得到下面的输出结果：



```
通过AST解释器运行程序：
r=4, area=50.24
r=5, area=78.5
```

看来，函数的传参功能和返回值功能运行都正常。

如果你喜欢动手，还可以在解释器里加一些调试代码，打印程序运行期间栈和栈帧的情况，加深对栈帧机制的理解。

最后，我们再加一个小彩蛋。也就是不知不觉间，其实我们的语言已经支持在函数内部声明函数了。

彩蛋：在函数内部声明的函数

仔细审视我们当前的语法规则，你会发现，函数声明是一种语句。而我们在任何一个 Block 里面，包括函数体里面，允许各种类型的语句，那当然也就允许声明函数。在下面的示例代码中，我把 circleArea 函数改写了一下，里面声明了一个 inner 函数：

[复制代码](#)

```
1 function circleArea(a:number):number{
2     function inner(b:number):number{
3         return b*b;
4     }
5     return 3.14*inner(a);
6 }
7 let r:number =4;
8 println("r=" + r + ", area="+circleArea(r));
9 r = 5;
10 println("r=" + r + ", area="+circleArea(r));
```

下图是 circleArea 对应的 AST，你看到，函数声明内部又嵌套了另一个函数声明。

```
Prog
  FunctionDecl circleArea
    Return type: number
    ParamList:
      VariableDecl a, type: number
        no initialization.
    FunctionDecl inner
      Return type: number
      ParamList:
        VariableDecl b, type: number
          no initialization.
      ReturnStatement
        Binary: Multiply
          Variable: b, resolved
          Variable: b, resolved
    ReturnStatement
      Binary: Multiply
        3.14
        FunctionCall inner, resolved
          Variable: a, resolved
```

在函数体内部声明的函数，其作用域只在函数内部，这在程序输出的符号表中能够看出来。inner 函数是 circleArea 中的一个符号，整个程序形成了三级作用域，分别是主函数 (_main)、circleArea 和 inner。

符号表：

```
Scope of _main
  circleArea{Function, local var count:1}
  r{Variable}
  Scope of circleArea
    a{Variable}
    inner{Function, local var count:1}
    Scope of inner
      b{Variable}
```

你可以运行一下使用内部函数版本的 circleArea，它也完全能正常运行，我们并不需要为支持内部函数而做什么特殊的事情！从这里，你也能看出在我们实现计算机语言的过程中，有一个很有魔力的地方：**只要你制定了规则，你的语言就会遵守该规则运行，哪怕有些运行场景你自己都没有意识到。**你在后面也会越来越多的体会到这一点。

不过，目前我们只是支持了内部函数而已，还没有高级函数的特性，也就是把函数本身当做参数和返回值来传递。这个过程中，通常又会涉及到闭包特性。我们将在后面的课程中涉及到这个知识点。那个时候，我们会在已经编译成本地代码的版本中讨论实现函数式编程涉及的知识点。

课程小结

好了，到这里我们今天的课就结束了，让我们来简单回顾一下吧。

这节课我们继续迭代和增强了我们的语言，让它支持了完整的函数功能。在这个过程中，我们仍然要把语法分析功能、语义分析功能和基于 AST 的解释器都升级一遍。

在这个过程中，比较重要的知识点有三个。

首先是作用域。通过 Scope 对象，我们让符号表变成了一个层次结构，让不同的变量和函数归属到不同层次的作用域。在现代语言中，符号表通常都是采用类似的层次结构来保存的。

第二个知识点是栈帧。在运行期，通过采用栈帧，我们可以让函数的本地变量的生存期与函数的生存期相一致，从而达到节省内存的目的。

最后，我们也讨论了 return 语句的实现机制。返回值是被保存到上一级栈帧的返回值字段的。并且，通过特殊设计的机制，我们保证了在遇到 return 语句的时候，程序会跳过其他的语句，直接从函数中退回。

在下一节课里，我们借助 if 语句和 for 循环语句，会进一步加深对作用域的理解。

思考题

今天，我给你留了两个思考题：

1. 在函数声明相关的语法规则中，parameter 的 Follow 集合是什么？有什么作用？
2. 在这节课中，我们是同等看待参数和本地变量的。但是，它们在使用起来真的没有差别吗？请你试着分析一下。

欢迎在留言区给我留言。感谢你和我一起学习，如果你觉得这节课讲得还不错，也欢迎分享给更多感兴趣的朋友。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 04 | 如何让我们的语言支持变量和类型？

精选留言 (1)

写留言



R

2021-08-19

//函数声明，由'function'关键字、函数名、函数签名和函数体构成。
functionDeclaration
: 'function' Identifier callSignature '{' functionBody '}';
...
展开