

## 08 | 作用域和生存期：实现块作用域和函数

2019-08-30 宫文学

编译原理之美

[进入课程 >](#)



**讲述：宫文学**

时长 13:36 大小 12.46M



目前，我们已经用 Antlr 重构了脚本解释器，有了工具的帮助，我们可以实现更高级的功能，比如函数功能、面向对象功能。当然了，在这个过程中，我们还要克服一些挑战，比如：

如果要实现函数功能，要升级变量管理机制；

引入作用域机制，来保证变量的引用指向正确的变量定义；

提升变量存储机制，不能只把变量和它的值简单地扔到一个 HashMap 里，要管理它的生存期，减少对内存的占用。

本节课，我将借实现块作用域和函数功能，带你探讨作用域和生存期及其实现机制，并升级变量管理机制。那么什么是作用域和生存期，它们的重要性又体现在哪儿呢？

“作用域”和“生存期”是计算机语言中更加基础的概念，它们可以帮你深入地理解函数、块、闭包、面向对象、静态成员、本地变量和全局变量等概念。

而且一旦你深入理解，了解作用域与生存期在编译期和运行期的机制之后，就能解决在学习过程中可能遇到的一些问题，比如：

闭包的机理到底是什么？

为什么需要栈和堆两种机制来管理内存？它们的区别又是什么？

一个静态的内部类和普通的内部类有什么区别？

了解上面这些内容之后，接下来，我们来具体看看什么是作用域。

## 作用域 (Scope)

作用域是指计算机语言中变量、函数、类等起作用的范围，我们来看一个具体的例子。

下面这段代码是用 C 语言写的，我们在全局以及函数 fun 中分别声明了 a 和 b 两个变量，然后在代码里对这些变量做了赋值操作：

 复制代码


```
1  /*
2  scope.c
3  测试作用域。
4  */
5  #include <stdio.h>
6
7  int a = 1;
8
9  void fun()
10 {
11     a = 2;
12     //b = 3;    // 出错，不知道 b 是谁
13     int a = 3; // 允许声明一个同名的变量吗？
14     int b = a; // 这里的 a 是哪个？
15     printf("in fun: a=%d b=%d \n", a, b);
16 }
17
18 int b = 4; //b 的作用域从这里开始
19
20 int main(int argc, char **argv){
21     printf("main--1: a=%d b=%d \n", a, b);
22 }
```

```

23     fun();
24     printf("main--2: a=%d b=%d \n", a, b);
25
26     // 用本地变量覆盖全局变量
27     int a = 5;
28     int b = 5;
29     printf("main--3: a=%d b=%d \n", a, b);
30
31     // 测试块作用域
32     if (a > 0){
33         int b = 3; // 允许在块里覆盖外面的变量
34         printf("main--4: a=%d b=%d \n", a, b);
35     }
36     else{
37         int b = 4; // 跟 if 块里的 b 是两个不同的变量
38         printf("main--5: a=%d b=%d \n", a, b);
39     }
40
41     printf("main--6: a=%d b=%d \n", a, b);
42 }

```

这段代码编译后运行，结果是：

 复制代码

```

1 main--1: a=1 b=4
2 in fun: a=3 b=3
3 main--2: a=2 b=4
4 main--3: a=5 b=5
5 main--4: a=5 b=3
6 main--6: a=5 b=5

```

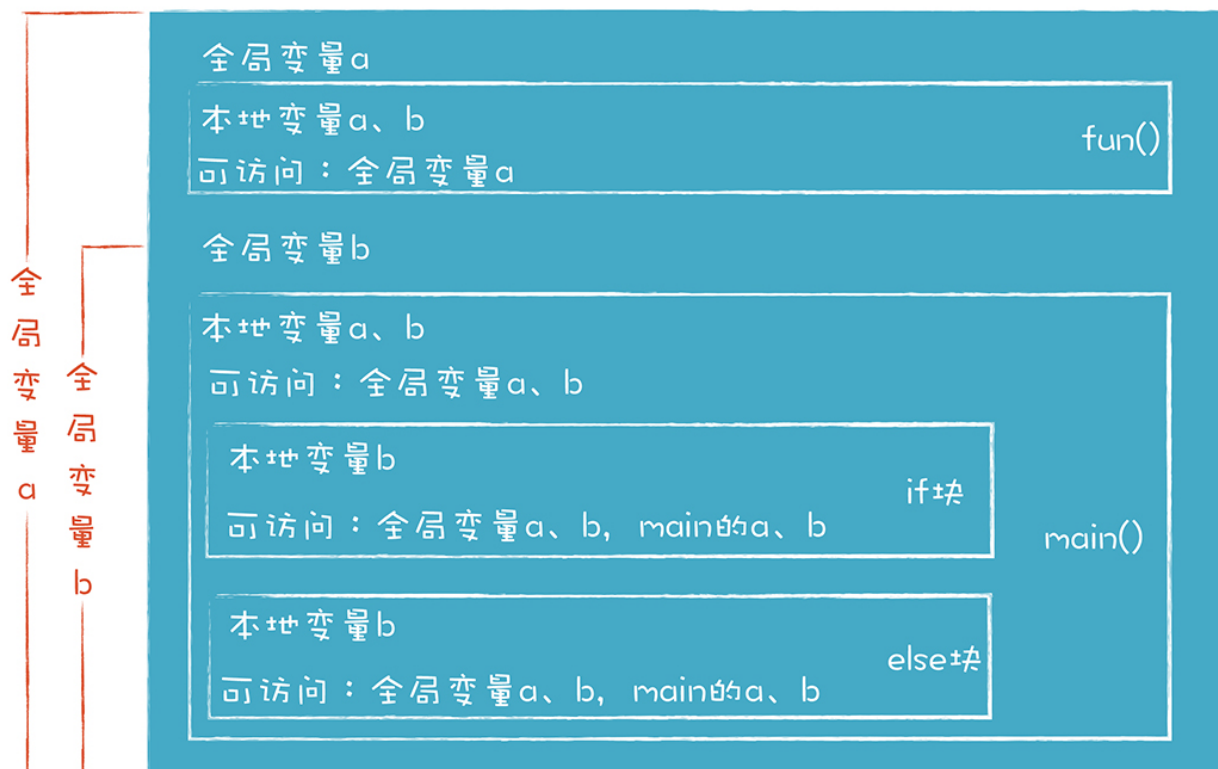
我们可以得出这样的规律：

变量的作用域有大有小，外部变量在函数内可以访问，而函数中的本地变量，只有本地才可以访问。

变量的作用域，从声明以后开始。


在函数里，我们可以声明跟外部变量相同名称的变量，这个时候就覆盖了外部变量。

下面这张图直观地显示了示例代码中各个变量的作用域：



另外，C 语言里还有块作用域的概念，就是用花括号包围的语句，if 和 else 后面就跟着这样的语句块。块作用域的特征跟函数作用域的特征相似，都可以访问外部变量，也可以用本地变量覆盖掉外部变量。

你可能会问：“其他语言也有块作用域吗？特征是一样的吗？”其实，各个语言在这方面的设计机制是不同的。比如，下面这段用 Java 写的代码里，我们用了一个 if 语句块，并且在 if 部分、else 部分和外部分别声明了一个变量 c：

 复制代码


```
1 /**
2  * Scope.java
3  * 测试 Java 的作用域
4  */
5 public class ScopeTest{
6
7     public static void main(String args[]){
8         int a = 1;
9         int b = 2;
10
11         if (a > 0){
12             //int b = 3; // 不允许声明与外部变量同名的变量
13             int c = 3;
14         }
15         else{
```

```

16         int c = 4;    // 允许声明另一个 c，各有各的作用域
17     }
18
19     int c = 5;    // 这里也可以声明一个新的 c
20 }
21 }

```

你能看到，Java 的块作用域跟 C 语言的块作用域是不同的，它不允许块作用域里的变量覆盖外部变量。那么和 C、Java 写起来很像的 JavaScript 呢？来看一看下面这段测试 JavaScript 作用域的代码：

 复制代码

```

1  /**
2   * Scope.js
3   * 测试 JavaScript 的作用域
4   */
5  var a = 5;
6  var b = 5;
7  console.log("1: a=%d b=%d", a, b);
8
9  if (a > 0) {
10     a = 4;
11     console.log("2: a=%d b=%d", a, b);
12     var b = 3; // 看似声明了一个新变量，其实还是引用的外部变量
13     console.log("3: a=%d b=%d", a, b);
14 }
15 else {
16     var b = 4;
17     console.log("4: a=%d b=%d", a, b);
18 }
19
20 console.log("5: a=%d b=%d", a, b);
21
22 for (var b = 0; b < 2; b++){ // 这里是否能声明一个新变量，用于 for 循环？
23     console.log("6-%d: a=%d b=%d", b, a, b);
24 }
25
26 console.log("7: a=%d b=%d", a, b);

```

这段代码编译后运行，结果是：

 复制代码

```
1 1: a=5 b=5
2 2: a=4 b=5
3 3: a=4 b=3
4 5: a=4 b=3
5 6-0: a=4 b=0
6 6-1: a=4 b=1
7 7: a=4 b=2
```

你可以看到，JavaScript 是没有块作用域的。我们在块里和 for 语句试图重新定义变量 b，语法上是允许的，但我们每次用到的其实是同一个变量。

对比了三种语言的作用域特征之后，你是否发现原来看上去差不多的语法，内部机理却不同？这种不同其实是语义差别的一个例子。**你要注意的是，现在我们讲的很多内容都已经属于语义的范畴了，对作用域的分析就是语义分析的任务之一。**


## 生存期 (Extent)

了解了什么是作用域之后，我们再理解一下跟它紧密相关的生存期。它是变量可以访问的时间段，也就是从分配内存给它，到收回它的内存之间的时间。

在前面几个示例程序中，变量的生存期跟作用域是一致的。出了作用域，生存期也就结束了，变量所占用的内存也就被释放了。这是本地变量的标准特征，这些本地变量是用栈来管理的。

但也有一些情况，变量的生存期跟语法上的作用域不一致，比如在堆中申请的内存，退出作用域以后仍然会存在。

下面这段 C 语言的示例代码中，fun 函数返回了一个整数的指针。出了函数以后，本地变量 b 就消失了，这个指针所占用的内存 (&b) 就收回了，其中 &b 是取 b 的地址，这个地址是指向栈里的一小块空间，因为 b 是栈里申请的。在这个栈里的小空间里保存了一个地址，指向在堆里申请的内存。这块内存，也就是用来实际保存数值 2 的空间，并没有被收回，我们必须手动使用 free() 函数来收回。

 复制代码


```
1 /*
2 extent.c
3 测试生存期。
```

```

4  */
5  #include <stdio.h>
6  #include <stdlib.h>
7
8  int * fun(){
9      int * b = (int*)malloc(1*sizeof(int)); // 在堆中申请内存
10     *b = 2; // 给该地址赋值 2
11
12     return b;
13 }
14
15 int main(int argc, char **argv){
16     int * p = fun();
17     *p = 3;
18
19     printf("after called fun: b=%lu *b=%d \n", (unsigned long)p, *p);
20
21     free(p);
22 }

```

类似的情况在 Java 里也有。Java 的对象实例缺省情况下是在堆中生成的。下面的示例代码中，从一个方法中返回了对象的引用，我们可以基于这个引用继续修改对象的内容，这证明这个对象的内存并没有被释放：

 复制代码

```

1  /**
2   * Extent2.java
3   * 测试 Java 的生存期特性
4   */
5  public class Extent2{
6
7      StringBuffer myMethod(){
8          StringBuffer b = new StringBuffer(); // 在堆中生成对象实例
9          b.append("Hello ");
10         System.out.println(System.identityHashCode(b)); // 打印内存地址
11         return b; // 返回对象引用，本质是一个内存地址
12     }
13
14     public static void main(String args[]){
15         Extent2 extent2 = new Extent2();
16         StringBuffer c = extent2.myMethod(); // 获得对象引用
17         System.out.println(c);
18         c.append("World!"); // 修改内存中的内容
19         System.out.println(c);
20
21         // 跟在 myMethod() 中打印的值相同

```

```
22         System.out.println(System.identityHashCode(c));
23     }
24 }
```

因为 Java 对象所采用的内存超出了申请内存时所在的作用域，所以也就没有办法自动收回。所以 Java 采用的是自动内存管理机制，也就是垃圾回收技术。

那么为什么说作用域和生存期是计算机语言更加基础的概念呢？其实是因为它们对应到了运行时的内存管理的基本机制。虽然各门语言设计上的特性是不同的，但在运行期的机制都很相似，比如都会用到栈和堆来做内存管理。

好了，理解了作用域和生存期的原理之后，我们就来实现一下，先来设计一下作用域机制，然后再模拟实现一个栈。

## 实现作用域和栈

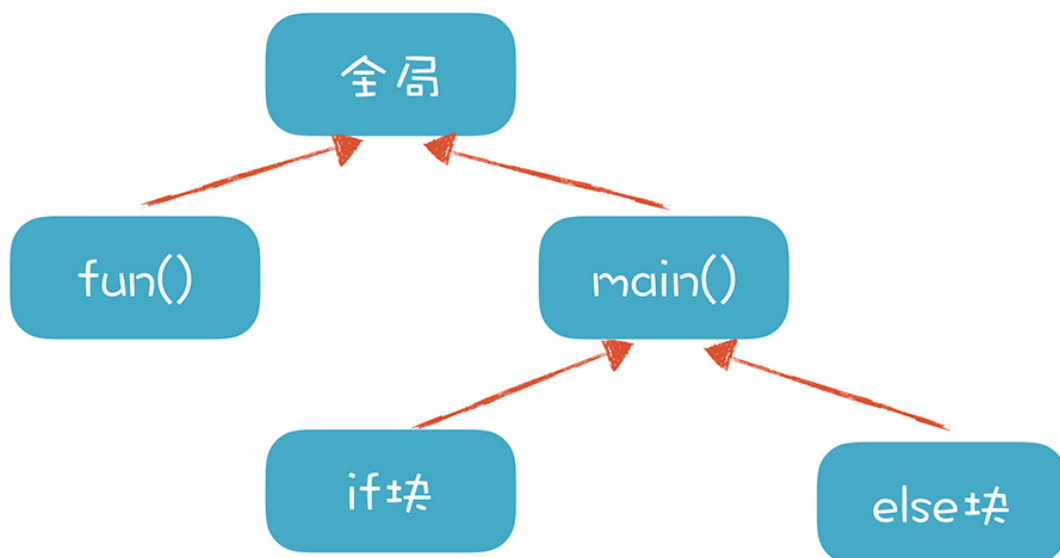
在之前的 PlayScript 脚本的实现中，处理变量赋值的时候，我们简单地把变量存在一个哈希表里，用变量名去引用，就像下面这样：

 复制代码


```
1 public class SimpleScript {
2     private HashMap<String, Integer> variables = new HashMap<String, Integer>();
3     ...
4 }
```

但如果变量存在多个作用域，这样做就不行了。这时，我们就要设计一个数据结构，区分不同变量的作用域。分析前面的代码，你可以看到作用域是一个树状的结构，比如 Scope.c 的作用域：





面向对象的语言不太相同，它不是一棵树，是一片树林，每个类对应一棵树，所以它也没有全局变量。在我们的 playscript 语言中，我们设计了下面的对象结构来表示 Scope：

 复制代码

```
1 // 编译过程中产生的变量、函数、类、块，都被称作符号
2 public abstract class Symbol {
3     // 符号的名称
4     protected String name = null;
5
6     // 所属作用域
7     protected Scope enclosingScope = null;
8
9     // 可见性，比如 public 还是 private
10    protected int visibility = 0;
11
12    //Symbol 关联的 AST 节点
13    protected ParserRuleContext ctx = null;
14 }
15
16 // 作用域
17 public abstract class Scope extends Symbol{
18     // 该 Scope 中的成员，包括变量、方法、类等。
19     protected List<Symbol> symbols = new LinkedList<Symbol>();
20 }
21
22 // 块作用域
23 public class BlockScope extends Scope{
24     ...
25 }
26
27 // 函数作用域
28 public class Function extends Scope implements FunctionType{
```

```
29     ...
30 }
31
32 // 类作用域
33 public class Class extends Scope implements Type{
34     ...
35 }
```

目前我们划分了三种作用域，分别是块作用域（Block）、函数作用域（Function）和类作用域（Class）。

我们在解释执行 playscript 的 AST 的时候，需要建立起作用域的树结构，对作用域的分析过程是语义分析的一部分。也就是说，并不是有了 AST，我们马上就可以运行它，在运行之前，我们还要做语义分析，比如对作用域做分析，让每个变量都能做正确的引用，这样才能正确地执行这个程序。

解决了作用域的问题以后，再来看看如何解决生存期的问题。还是看 Scope.c 的代码，随着代码的执行，各个变量的生存期表现如下：

进入程序，全局变量逐一生效；

进入 main 函数，main 函数里的变量顺序生效；

进入 fun 函数，fun 函数里的变量顺序生效；

退出 fun 函数，fun 函数里的变量失效；

进入 if 语句块，if 语句块里的变量顺序生效；

退出 if 语句块，if 语句块里的变量失效；

退出 main 函数，main 函数里的变量失效；

退出程序，全局变量失效。

通过下面这张图，你能直观地看到运行过程中栈的变化：



代码执行时进入和退出一个个作用域的过程，可以用栈来实现。每进入一个作用域，就往栈里压入一个数据结构，这个数据结构叫做**栈帧 (Stack Frame)**。栈帧能够保存当前作用域的所有本地变量的值，当退出这个作用域的时候，这个栈帧就被弹出，里面的变量也就失效了。

你可以看到，栈的机制能够有效地使用内存，变量超出作用域的时候，就没有用了，就可以从内存中丢弃。我在 `ASTEvaluator.java` 中，用下面的数据结构来表示栈和栈帧，其中的 `PlayObject` 通过一个 `HashMap` 来保存各个变量的值：

[复制代码](#)

```
1 private Stack<StackFrame> stack = new Stack<StackFrame>();
2
3 public class StackFrame {
4     // 该 frame 所对应的 scope
5     Scope scope = null;
6
7     //enclosingScope 所对应的 frame
8     StackFrame parentFrame = null;
9
10    // 实际存放变量的地方
11    PlayObject object = null;
12 }
13
14 public class PlayObject {
15     // 成员变量
16     protected Map<Variable, Object> fields = new HashMap<Variable, Object>();
17 }
```

目前，我们只是在概念上模仿栈帧，当我们用 Java 语言实现的时候，`PlayObject` 对象是存放在堆里的，Java 的所有对象都是存放在堆里的，只有基础数据类型，比如 `int` 和对象引用是放在栈里的。虽然只是模仿，这不妨碍我们建立栈帧的概念，在后端技术部分，我们会实现真正意义上的栈帧。

要注意的是，栈的结构和 Scope 的树状结构是不一致的。也就是说，栈里的上一级栈帧，不一定是 Scope 的父节点。要访问上一级 Scope 中的变量数据，要顺着栈帧的 parentFrame 去找。我在上图中展现了这种情况，在调用 fun 函数的时候，栈里一共有三个栈帧：全局栈帧、main() 函数栈帧和 fun() 函数栈帧，其中 main() 函数栈帧的 parentFrame 和 fun() 函数栈帧的 parentFrame 都是全局栈帧。

## 实现块作用域

目前，我们已经做好了作用域和栈，在这之后，就能实现很多功能了，比如让 if 语句和 for 循环语句使用块作用域和本地变量。以 for 语句为例，visit 方法里首先为它生成一个栈帧，并加入到栈中，运行完毕之后，再从栈里弹出：

 复制代码


```
1 BlockScope scope = (BlockScope) cr.node2Scope.get(ctx); // 获得 Scope
2 StackFrame frame = new StackFrame(scope); // 创建一个栈帧
3 pushStack(frame); // 加入栈中
4
5 ...
6
7 // 运行完毕，弹出栈
8 stack.pop();
```

当我们在代码中需要获取某个变量的值的时候，首先在当前帧中寻找。找不到的话，就到上一级作用域对应的帧中去找：

 复制代码

```
1 StackFrame f = stack.peek(); // 获取栈顶的帧
2 PlayObject valueContainer = null;
3 while (f != null) {
4     // 看变量是否属于当前栈帧里
5     if (f.scope.containsSymbol(variable)){
6         valueContainer = f.object;
7         break;
8     }
9     // 从上一级 scope 对应的栈帧里去找
10    f = f.parentFrame;
11 }
```

运行下面的测试代码，你会看到在执行完 for 循环以后，我们仍然可以声明另一个变量 i，跟 for 循环中的 i 互不影响，这证明它们确实属于不同的作用域：


 复制代码

```
1 String script = "int age = 44; for(int i = 0;i<10;i++) { age = age + 2;} int i = 8;";
```

进一步的，我们可以实现对函数的支持。

## 实现函数功能

先来看一下与函数有关的语法：


 复制代码

```
1 // 函数声明
2 functionDeclaration
3     : typeTypeOrVoid? IDENTIFIER formalParameters ('[' ' '])*
4       functionBody
5       ;
6 // 函数体
7 functionBody
8     : block
9     | ';'
10    ;
11 // 类型或 void
12 typeTypeOrVoid
13     : typeType
14     | VOID
15     ;
16 // 函数所有参数
17 formalParameters
18     : '(' formalParameterList? ')'
19     ;
20 // 参数列表
21 formalParameterList
22     : formalParameter (',' formalParameter)* (',' lastFormalParameter)?
23     | lastFormalParameter
24     ;
25 // 单个参数
26 formalParameter
27     : variableModifier* typeType variableDeclaratorId
28     ;
29 // 可变参数数量情况下，最后一个参数
30 lastFormalParameter
31     : variableModifier* typeType '...' variableDeclaratorId
```

```
32     ;
33 // 函数调用
34 functionCall
35     : IDENTIFIER '(' expressionList? ')'
36     | THIS '(' expressionList? ')'
37     | SUPER '(' expressionList? ')'
38     ;
```

在函数里，我们还要考虑一个额外的因素：**参数**。在函数内部，参数变量跟普通的本地变量在使用时没什么不同，在运行期，它们也像本地变量一样，保存在栈帧里。

我们设计一个对象来代表函数的定义，它包括参数列表和返回值的类型：

 复制代码

```
1 public class Function extends Scope implements FunctionType{
2     // 参数
3     protected List<Variable> parameters = new LinkedList<Variable>();
4
5     // 返回值
6     protected Type returnType = null;
7
8     ...
9 }
```


在调用函数时，我们实际上做了三步工作：

建立一个栈帧；

计算所有参数的值，并放入栈帧；

执行函数声明中的函数体。

我把相关代码放在了下面，你可以看一下：

 复制代码

```
1 // 函数声明的 AST 节点
2 FunctionDeclarationContext functionCode = (FunctionDeclarationContext) function.ctx;
3
4 // 创建栈帧
5 functionObject = new FunctionObject(function);
```


```

6 StackFrame functionFrame = new StackFrame(functionObject);
7
8 // 计算实参的值
9 List<Object> paramValues = new LinkedList<Object>();
10 if (ctx.expressionList() != null) {
11     for (ExpressionContext exp : ctx.expressionList().expression()) {
12         Object value = visitExpression(exp);
13         if (value instanceof LValue) {
14             value = ((LValue) value).getValue();
15         }
16         paramValues.add(value);
17     }
18 }
19
20 // 根据形参的名称，在栈帧中添加变量
21 if (functionCode.formalParameters().formalParameterList() != null) {
22     for (int i = 0; i < functionCode.formalParameters().formalParameterList().formalParameterList().size(); i++) {
23         FormalParameterContext param = functionCode.formalParameters().formalParameterList().formalParameterList().get(i);
24         LValue lValue = (LValue) visitVariableDeclaratorId(param.variableDeclaratorId());
25         lValue.setValue(paramValues.get(i));
26     }
27 }
28
29 // 调用方法体
30 rtn = visitFunctionDeclaration(functionCode);
31
32 // 运行完毕，弹出栈
33 stack.pop();

```



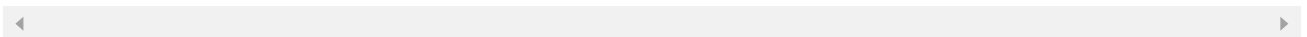
你可以用 playscript 测试一下函数执行的效果，看看参数传递和作用域的效果：

 复制代码

```

1 String script = "int b= 10; int myfunc(int a) {return a+b+3;} myfunc(2);";

```



## 课程小结

本节课，我带你实现了块作用域和函数，还跟你一起探究了计算机语言的两个底层概念：作用域和生存期。你要知道：

对作用域的分析是语义分析的一项工作。Antlr 能够完成很多词法分析和语法分析的工作，但语义分析工作需要我们自己做。

变量的生存期涉及运行期的内存管理，也引出了栈帧和堆的概念，我会在编译器后端技术时进一步阐述。

我建议你在学习新语言的时候，先了解它在作用域和生存期上的特点，然后像示例程序那样做几个例子，借此你会更快理解语言的设计思想。比如，为什么需要命名空间这个特性？全局变量可能带来什么问题？类的静态成员与普通成员有什么区别？等等。

下一讲，我们会尝试实现面向对象特性，看看面向对象语言在语义上是怎么设计的，以及在运行期有什么特点。

## 一课一思

既然我强调了作用域和生存期的重要性，那么在你熟悉的语言中，有哪些特性是能用作用域和生存期的概念做更基础的解读呢？比如，面向对象的语言中，对象成员的作用域和生存期是怎样的？欢迎在留言区与大家一起交流。

最后，感谢你的阅读，如果这篇文章让你有所收获，也欢迎你将它分享给更多的朋友。

今天讲的功能照样能在 playscript-java 项目中找到示例代码，其中还有用 playscript 写的脚本，你可以多玩一玩。

playscript-java (项目目录) : [码云](#) [GitHub](#)

PlayScript.java (入口程序) : [码云](#) [GitHub](#)

PlayScript.g4 (语法规则) : [码云](#) [GitHub](#)

ASTEvaluator.java (解释器) : [码云](#) [GitHub](#)

BlockScope.play (演示块作用域) : [码云](#) [GitHub](#)

function.play (演示基础函数功能) : [码云](#) [GitHub](#)

lab/scope 目录 (各种语言的作用域测试) : [码云](#) [GitHub](#)

---

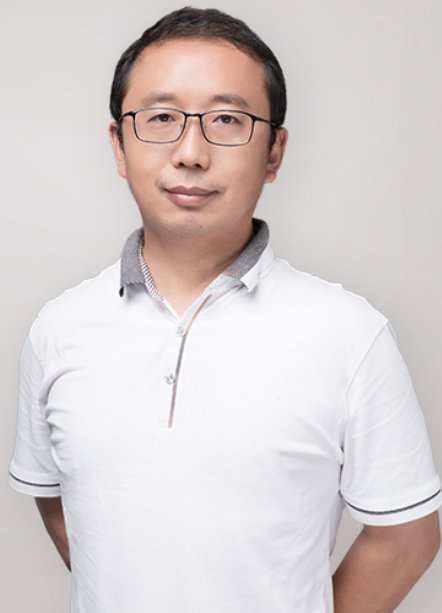


# 编译原理之美

手把手教你实现一个编译器

宫文学

北京物演科技CEO



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 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 07 | 编译器前端工具（二）：用Antlr重构脚本语言

下一篇 09 | 面向对象：实现数据和方法的封装

## 精选留言 (7)

 写留言



Johnson

2019-08-30

现在课程的做法相当于AST之后直接解析执行了，所有的逻辑都堆在AST和紧接着的语义分析，没有把AST转化成IR，然后在这个IR上做各种事情，最后再到interpreter执行。是因为前期为了简单起见，所以先这么直观的来么？

展开

作者回复: 因为目前是在讲前端，所以就先不引入IR。

同时也是在告诉同学们，哪怕我们只拿到了AST，也已经能做很多事情了。

IR在后端部分会讲。我会给出一个自己设计的IR的例子，用IR重新实现部分功能。然后再去采用LLVM的IR。



**ZYS**

2019-08-31

宫老师，可否兼顾一下用c++的学员，介绍一下cpp版本playscript如何在visual studio2010或更高的版本运行？

展开 ∨

作者回复: 讲后端部分的时候，主要是用cpp版本实现的。那部分的指导资料我整理一下，写一个README.md，尽快更新到Github和码云上。

先简单说一下：

- 1.如果仅仅用cpp版本的Antlr，这个比较简单，你做练习的时候可以试用一下。
- 2.把Antlr和LLVM一起用的时候，要配置的东西更多一些，好在有cmake。



**许童童**

2019-08-30

变量的使用范围由作用域决定，作用域由词法规则决定，词法分析生成作用域链，之后查找变量就沿着这条作用域链查找，与函数调用栈就没有关系了。一般函数的生存期就是出栈后就结束了，如果是引用对象会在本次GC中回收，如果产生了闭包，那就要等到引用闭包的变量销毁，生存期才结束。

展开 ∨

作者回复: 很准确，很清晰！



**沉淀的梦想**

2019-09-02

用PlayScript的代码运行课程中的示例时会报一个空指针异常

```
String script = "int age = 44; for(int i = 0;i<10;i++) { age = age + 2;} int i = 8;";
```

```
Exception in thread "main" java.lang.NullPointerException  
    at play.ASTEvaluator.visitStatement(ASTEvaluator.java:617)...
```

展开 ∨

- 作者回复: 1.是运行双引号内部的部分，不是连String script= 也带上。  
2.是使用playscript-java这个工程吗？别用错了工程。

如果还有问题，继续给我提问！  
希望不影响你继续动手实践的热情！

1



**Geek\_89bbab**

2019-09-01

```
private void pushStack(StackFrame frame) {  
    // 如果新加入的frame是当前frame的下一级，则入栈  
    if (stack.size() > 0) {  
  
        for (int i = stack.size()-1; i>0; i--){...
```

展开 ∨

作者回复: 我往代码里加了注释，你可以更新一下看看！  
我也把注释拷贝到这里。  
里面有些特性，比如一等公民函数，是还没讲到的，10讲就会讲。

第一个if:

```
/*
```

如果新加入的栈帧，跟某个已有的栈帧的enclosingScope是一样的，那么这俩的parentFrame也一样。

因为它们原本就是同一级的嘛。

比如:

```
void foo();  
void bar(foo());
```

或者:

```
void foo();  
if (...){  
    foo();  
}  
*/
```

第二个if:

```
/*
```

如果新加入的栈帧，是某个已有的栈帧的下一级，那么就把把这个父子关系建立起来。比如:

```
void foo(){  
    if (...){ //把这个块往栈帧里加的时候，就符合这个条件。  
    }  
}
```

再比如,下面的例子:

```
class MyClass{
    void foo();
}
MyClass c = MyClass(); //先加Class的栈帧, 里面有类的属性, 包括父类的
c.foo(); //再加foo()的栈帧
*/
```

第3个if:

```
/*
```

这是针对函数可能是一等公民的情况。这个时候, 函数运行时的作用域, 与声明时的作用域会不一致。

我在这里设计了一个 “receiver” 的机制, 意思是这个函数是被哪个变量接收了。要按照这个receiver的作用域来判断。

```
*/
```



**李懂**

2019-08-30

原来栈里放的栈帧, 栈帧是Scope,类似执行上下文, 里面保存了变量! 以前一直以为进栈, 是放的执行函数体, 跟上脚步!

展开 ▾

作者回复: 这个栈帧还是拿java模拟的, 让大家有个概念。到学后端的时候, 那里有更物理的栈帧实现。到时候你可以进一步加深一下认识:-D



**北冥Master**

2019-08-30

牛逼, 越来越深入了, 看的有点吃力了

展开 ▾

作者回复: 后几讲涉及的都是语义功能, 并涉及了一部分运行期技术 (给后端技术部分提前做铺垫)。

语义上的差别是每种语言真正的差别, 但底层有一些共通的机制。搞搞明白对我们学各种语言都有好处。

