

大咖助阵 | 海纳：C 语言是如何编译执行的？（三）

2022-03-18 海纳

《深入C语言和程序运行原理》

课程介绍 >



讲述：海纳

时长 13:15 大小 12.15M



你好，我是海纳。今天是“C 语言是如何编译执行的？”这一加餐系列的最后一讲。

一个编译器通常分为前端、中端和后端三个典型模块。前端主要包括词法分析和文法分析两个步骤，它的作用是把源文件转换成抽象语法树（**Abstract Syntax Tree, AST**）。在前面两期加餐中，我讲解了预处理、词法分析、文法分析的编译过程基本步骤，带你实现了一个小型 C 语言编译器前端。它将源代码翻译成了 **AST**，而且支持了变量定义和赋值，**if** 语句和 **while** 语句。

中端则主要是将 **AST** 转成中间表示（**Intermediate Representation, IR**），常见的中间表示有三地址码、基于图的静态单赋值表示等等，例如 **LLVM IR** 就是最常见的一种中间表示。编译器的优化主要集中在中端。

而后端的作用是将中间表示翻译成目标平台的机器码，并生成相应平台的可执行程序。机器码是由 **CPU** 指令集决定的，例如 **x86** 平台就要使用 **x86** 指令集，而 **arm64** 平台就应该使用

领资料



aarch64 指令集。华为的鲲鹏平台也是采用了 aarch64 指令集。可执行程序的格式则是由操作系统决定，例如在 Windows 系统上，可执行程序是 PE 格式的，exe 文件或者 dll 文件都是 PE 格式；而 Linux 系统上，可执行程序则是 ELF 文件。

如果完全按顺序来的话，这节课应该继续介绍中端和后端。但是中端优化和后端代码生成这两个话题都涉及很多内容，展开来讲的话，往往需要一整本书的篇幅。为了帮你通过有限的篇幅快速理解编译器的结构，这节课我会介绍一种最简单的执行模型：基于栈的字节码和虚拟机。

字节码和虚拟机

和静态编译一样，字节码也是一种非常常见的策略。由于字节码具有跨平台的特性，所以它得到了广泛的普及，其中最具有影响力的就是 Java 和 Python 字节码。“javac”这个工具会把 Java 源代码翻译成 class 文件，这个 class 文件就是字节码文件，它的代码部分全部由 Java 字节码组成。Python 也是一样的，编译器也会先把 py 文件翻译成 pyc 文件，而 pyc 文件的代码部分则由 Python 字节码组成。

Java 和 Python 的相似之处在于，它们都采用了一种基于栈的计算模型。我们来看下这个 Python 源文件：

 复制代码

```
1 a = 1
2 b = 2
3 def foo(c):
4     return a + b + 3 * c
```

如果使用 dis 方法对 foo 函数进行反编译，就会得到如下输出：

 复制代码

```
1 >>> dis.dis(foo)
2      2           0 LOAD_GLOBAL           0 (a)
3      3           3 LOAD_GLOBAL           1 (b)
4      4           6 BINARY_ADD
5      5           7 LOAD_CONST             1 (3)
6      6          10 LOAD_FAST              0 (c)
7      7          13 BINARY_MULTIPLY
8      8          14 BINARY_ADD
9      9          15 RETURN_VALUE
```

领资料

其中 `LOAD_XX` 指令的作用是把值加载到栈顶。`GLOBAL` 表示当前字节码加载的是全局变量，`CONST` 代表常量，`FAST` 代表局部变量。

`BINARY_ADD` 的作用是把栈上的两个变量出栈，然后把这两个值相加的和再压入栈中，这是对两个值进行求和。`BINARY_MULTIPLY` 的作用是对两个值求积。这个过程比较简单，我就不再画图表示了，你可以想象一下每执行一个字节码，栈会发生怎样的变化。欢迎在评论区交流你的想法。

上面提到的每一条指令，在文件中都有一个编号。还是以 `Python` 虚拟机为例，它的指令编码可以在 [🔗 这里](#) 查看。指令总数不超过 256，所以可以使用一个字节进行编码，这就是 **字节码 (bytecode)** 这个名称的由来。

无论是在 `arm` 平台还是 `x86` 平台上，也不管是在 `macOS` 还是 `Linux` 系统上，相同的源文件翻译成的字节码都是相同的。字节码文件会被虚拟机加载、解析并执行。就像每个 `CPU` 可以执行自己指令集中的指令一样，虚拟机一般是由静态编译语言（例如 `C/C++/Rust`）实现的，它就像一个 `CPU` 一样，逐条执行字节码，而字节码可以看成是虚拟机的指令集。

显然，不同平台的差异被虚拟机屏蔽掉了。正如上面所说，这种屏蔽底层差别，向上提供统一的指令集的办法，就像一个虚拟的计算机。这也正是语言虚拟机名字的由来。

基于栈的字节码，核心操作无非是压栈和出栈，大多数指令的操作数都在栈上，带有操作数的指令最多只带有一个操作数。基于栈的字节码，其指令不带参数或者只带一个参数，无论是编译器的实现还是虚拟机的实现都很简单。

接下来，我们就具体研究一下如何从 `AST` 生成基于栈的字节码。

生成字节码

如果你对后缀表达式求值比较熟悉的话，就会发现，上述字节码的执行过程是和后缀表达式求值相对应的。而且，[🔗 上节课](#) 我讲到了，对表达式的 `AST` 进行后序遍历就可以生成后缀表达式。其实，上节课的程序只要稍加修改就可以变成字节码的生成程序，具体如下所示：

领资料

复制代码

```
1 void code_object_emit_code(CodeObject* co, Node* root, Context* context) {
2     if (root->ntype == NT_INT) {
3         byte param = (byte)code_object_find_constant_index(co, ((IntNode*)root)
```

```

4     code_object_append_bytecode(co, LOAD_CONST, param);
5 }
6 else if (root->ntype == NT_ASN) {
7     AssignNode* node = (AssignNode*)root;
8     code_object_emit_code(co, node->value, context);
9     context->is_store = Store;
10    code_object_emit_code(co, node->var, context);
11    context->is_store = Load;
12 }
13 else if (root->ntype == NT_VAR) {
14     VarNode* node = (VarNode*)root;
15     if (context->is_store == Store) {
16         byte param = (byte)code_object_find_variable_index(co, node->name);
17         code_object_append_bytecode(co, STORE_NAME, param);
18     }
19     else {
20         byte param = (byte)code_object_find_variable_index(co, node->name);
21         code_object_append_bytecode(co, LOAD_NAME, param);
22     }
23 }
24 else if (root->ntype == NT_IF) {
25     IfNode* node = (IfNode*)root;
26     code_object_emit_code(co, node->cond, context);
27     code_object_jump_false(co, IF_ELSE);
28     code_object_emit_code(co, node->then_clause, context);
29     code_object_jump(co, IF_END);
30     code_object_bind(co, IF_ELSE);
31     code_object_emit_code(co, node->else_clause, context);
32     code_object_bind(co, IF_END);
33 }
34 else if (root->ntype == NT_WHILE) {
35     code_object_bind(co, WHILE_HEAD);
36
37     WhileNode* node = (WhileNode*)root;
38     code_object_emit_code(co, node->cond, context);
39
40     code_object_jump_false(co, WHILE_END);
41     code_object_emit_code(co, node->body, context);
42     code_object_jump(co, WHILE_HEAD);
43     code_object_bind(co, WHILE_END);
44 }
45 else if (root->ntype == NT_LIST) {
46     ListNode* node = (ListNode*)root;
47     for (int i = 0; i < node->length; i++) {
48         code_object_emit_code(co, node->array[i], context);
49     }
50 }
51 else if (root->ntype == NT_PRINT) {
52     code_object_emit_code(co, ((PrintNode*)root)->expr, context);
53     code_object_append_bytecode(co, PRINT, 0);
54 }
55 else {

```

```

56     BinOpNode* binop = (BinOpNode*)root;
57     code_object_emit_code(co, binop->left, context);
58     code_object_emit_code(co, binop->right, context);
59
60     enum NodeType tt = root->ntype;
61     byte param = 0;
62     if (tt == NT_ADD) {
63         code_object_append_bytecode(co, BINARY_ADD, param);
64     }
65     else if (tt == NT_SUB) {
66         code_object_append_bytecode(co, BINARY_SUB, param);
67     }
68     else if (tt == NT_DIV) {
69         code_object_append_bytecode(co, BINARY_DIV, param);
70     }
71     else if (tt == NT_MUL) {
72         code_object_append_bytecode(co, BINARY_MUL, param);
73     }
74     else if (tt == NT_LT) {
75         code_object_append_bytecode(co, COMPARE, COMPARE_LT);
76     }
77 }
78

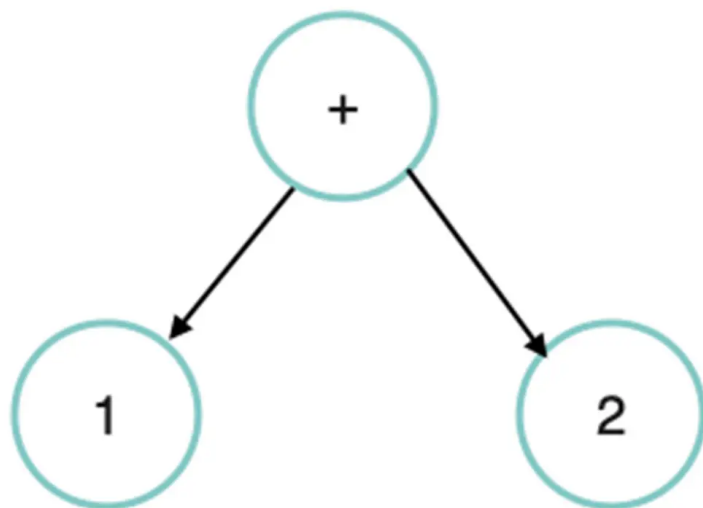
```

我知道，当你第一次读到这段代码的时候，内心一定是崩溃的。但不要害怕，其实用 C 语言写出来的代码比使用 C++ 实现的访问者模式还要直观。接下来，我带着你一起来分析这段代码，你就会发现，这段程序的逻辑是简明直接的。

下面，我举四个例子来说明这段程序的工作原理。

先来看第一个例子，表达式“1+2”的 AST 如下图所示：





`code_object_emit_code` 在访问加号结点时，会先访问它的左子树（代码第 57 行），再访问它的右子树（第 58 行），这将保证加法的两个操作数都在栈上。对于“1+2”这个例子，左子树是整型，所以程序会在常量池里找到整型的序号（第 3 行），把它作为 `LOAD_CONST` 的参数（第 4 行）。之所以使用常量池序号作为 `LOAD_CONST` 的参数，而不是直接使用整数，是因为参数是一个字节，而表示一个整数需要 4 个字节。这同时也意味着常量池的大小不能超过 256。

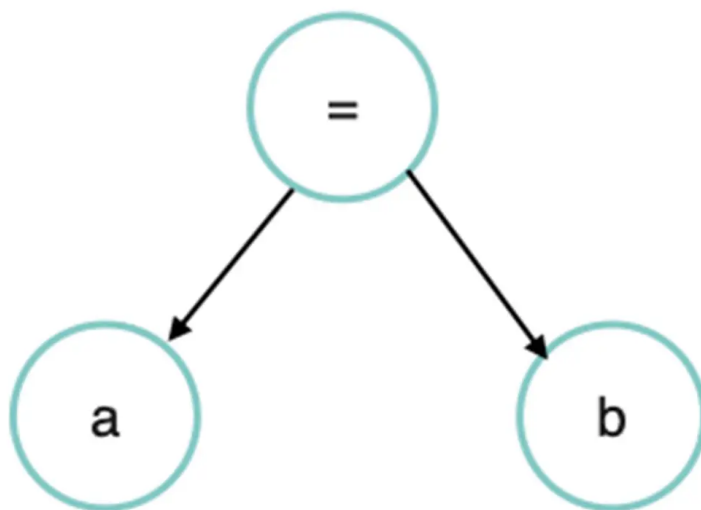
综上所述，“1+2”这个表达式对应的字节码如下：

```
1 LOAD_CONST 0 # 代表数字1在常量池中的序号
2 LOAD_CONST 1 # 代表数字2在常量池中的序号
3 BINARY_ADD
```

 复制代码

第二个例子是赋值语句“`int a = b`”，这条语句所产生的 AST 如下图所示：





从图中可以看出，赋值结点的左孩子和右孩子都是变量名，但这两个变量却是有所区别的。对变量 `a` 的访问，应该生成 `STORE` 指令，而对变量 `b` 的访问则应该生成 `LOAD` 指令。所以在这里我使用了一个 `Context` 变量，来指定当访问的是赋值语句的左端变量时，就使用 `STORE_NAME` 指令（第 9、16、17 行），如果是访问普通的变量，则使用 `LOAD_NAME` 指令（第 20、21 行）。

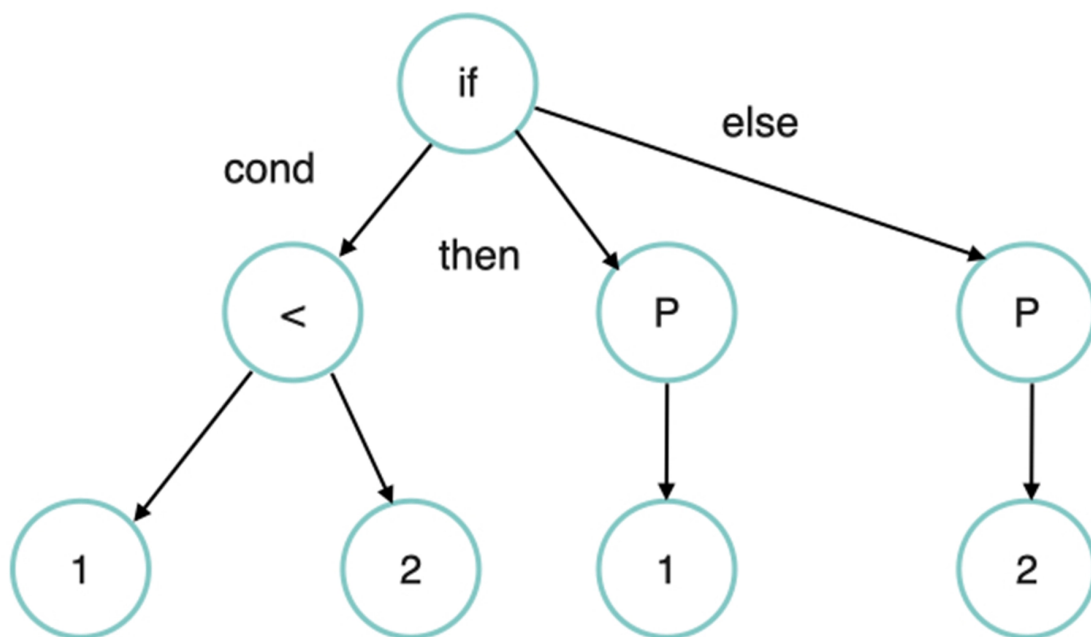
和访问常量类似，我在生成代码的时候，也引入一个变量表。变量表和常量表都是一个列表，这里你也可以把它设计成哈希表，以加快查找速度。但总之，这两个表都是一种容器。所以，“`int a = b`”这个赋值语句所对应的字节码是：

复制代码

```
1 LOAD_NAME 0 # 变量 b 在变量表中的序号
2 STORE_NAME 1 # 变量 a 在变量表中的序号
```

第三个例子是分支语句，代码“`if (1 < 2) { print(1); } else { print(2); }`”，它的 AST 如下图所示：





这里说明下，`print` 是我为了方便打印值而引入的一种非常规手段。由于支持函数调用需要的基础设施太多，我不可能在这短短的几节课内讲清楚，所以就把 `print` 当成了关键字来处理。遇到 `Print` 结点时，生成 `PRINT` 指令即可（第 52、53 行）。

对于 `If` 结点，它的 `cond`、`then_clause`、`else_clause` 的访问都很简单。难就难在，当条件不成立时，控制流应该跳过 `then_clause`，转而去执行 `else_clause`，这需要通过 `jump` 语句实现。

代码的第 27 行就是这个思路的直接体现。第 27 行的作用是生成 `JUMP_IF_FALSE` 指令，也就是说 `cond` 条件的执行结果为 `false` 时，就进行跳转。但是跳转的目标是哪里呢？答案是在 `then_clause` 结束，`else_clause` 开始之前。

但这时候我们就遇到问题了：在生成 `JUMP` 指令的时候，我们还不知道 `then_clause` 会产生多少字节码，也就是说，我们并不知道 `JUMP` 的目标地址在哪里。所以，在生成 `JUMP` 指令的时候，我们只能将目标地址空着（第 27 行），当目标地址确定了以后，再把目标地址回填到 `JUMP` 指令处，这正是 `bind` 函数所做的事情（第 30 行）。



相信你对这个过程已经不陌生了，因为这就是**重定位（Relocation）**所做的工作：把目标地址回填到 `JUMP`、`CALL` 等指令参数里。为了能够在重定位时正确地找到需要重定位的那条

jump 指令，我们必须要在 code_object_jump_false 里将这条 jump 指令的地址记录下来。这种用于辅助重定位的信息就是重定位信息（Reloc Info）。

所以，经过重定位以后，这个例子中的分支语句最终生成的字节码就是：

 复制代码

```
1  LOAD_CONST 1
2  LOAD_CONST 2
3  COMPARE <
4  JUMP_IF_FALSE 5
5  LOAD_CONST 1
6  PRINT
7  JUMP 3
8  LOAD_CONST 2
9  PRINT
```

其中，第 4 行的字节码 JUMP_IF_FALSE 的作用就是让控制流向后跳 5 个字节。由于第 5 行的 LOAD_CONST 和第 7 行的 JUMP 都带有一个参数，所以跳 5 个字节，控制流就指向了第 8 行，而这正是 else_clause 开始的地方。

第 7 行的 JUMP 指令则是 then_clause 结束的地方，这就意味着当 if 条件为 true 时，else_clause 就被跳过了。

最后一个例子是循环语句，代码“int i = 0; while (i < 10) { i = i + 1; print(i); }”。循环语句和分支语句有相似之处，它也包含两个跳转：第一个是当条件不成立时，要跳过循环体，第二个是循环体结束处，应该再跳转回循环头继续下一次判断。所以第一个跳转是条件跳转，第二个跳转则是绝对跳转。

而这两种跳转，我们都已经支持了。但是循环语句有一个分支语句所没有的特点，那就是循环体内可能会出现 break。当遇到 break 语句时，控制流就应该直接跳到循环体的结尾。而循环体内可能有多个 break 语句，这就要求我们要在每个 break 语句处都记录下需要重定位的地址，所以我把这些重定位信息使用一个双向链表记录在了一起。具体的代码我就不在这里展示了，你可以去 [gitee](#) 上查看。

 领资料

同样地，最后一个例子的字节码展示如下：

 复制代码

```

1  LOAD_CONST 0
2  STORE_NAME i
3  LOAD_NAME i
4  LOAD_CONST 10
5  COMPARE <
6  JUMP_IF_FALSE 12
7  LOAD_NAME i
8  LOAD_CONST 1
9  BINARY_ADD
10 STORE_NAME i
11 LOAD_NAME i
12 PRINT
13 JUMP -20

```

通过这四个例子，我就把最基本的变量、分支和循环全部实现了。接下来，我们就再看一下这些字节码是如何实现的。也就是说，我们需要再研究一下虚拟机的具体实现。

虚拟机的实现

跟编译器的实现相比，虚拟机的实现更加直观：通过一个循环，不断地取出字节码，然后按照这个字节码的语义对栈进行操作。

按照上面的分析，我在这里展示一种常见的虚拟机实现：

 复制代码

```

1 void interpret(VirtualMachine* vm, CodeObject* co) {
2     for (int i = 0; i < co->bytecodes->length; i++) {
3         byte opcode = co->bytecodes->array[i];
4         byte param = 0;
5         int u, v;
6
7         if (opcode >= OP_CODE_PARAMETER) {
8             param = co->bytecodes->array[++i];
9         }
10
11        switch(opcode) {
12        case LOAD_CONST:
13            PUSH(co->constant_pool->array[param]);
14            break;
15        case STORE_NAME:
16            u = POP();
17            vm->var_table->array[param] = u;
18            break;
19        case LOAD_NAME:
20            PUSH(vm->var_table->array[param]);
21            break;

```

领资料

```

22     case BINARY_ADD:
23         u = POP();
24         v = POP();
25         PUSH(u+v);
26         break;
27     case PRINT:
28         u = POP();
29         printf("%d\n", u);
30         break;
31     case COMPARE:
32         u = POP();
33         v = POP();
34         if (param == COMPARE_LT) {
35             if (v < u) {
36                 PUSH(1);
37             }
38             else {
39                 PUSH(0);
40             }
41         }
42         break;
43     case JUMP_IF_FALSE:
44         u = POP();
45         if (!u) {
46             i += param;
47         }
48         break;
49     case JUMP:
50         i += param;
51         break;
52     default:
53         printf("Error, unrecognized bytecode: %d\n", opcode);
54     }
55 }
56 }

```

这里，我以 `JUMP_IF_FALSE` 为例进行讲解：虚拟机先从栈顶得到一个元素，再判断这个元素是否为 `false`。如果不是 `false`，则什么都不做，直接执行下一条字节码；但如果为 `false`，就会给变量 `i` 加上这个跳转指令的参数。这样一来，`i` 就变成跳转的目标地址了。



其他字节码都比较简单，我就不再一一解释了，你可以在 [这里](#) 查看全部代码。



总结

在这节课里，我介绍了一种最简单的执行模型：基于栈的字节码和虚拟机。

把 AST 转换成字节码的过程，主要是通过序遍历整个语法树，分别生成相应的字节码。其中，最难的部分是重定位的过程。重定位是指**指令在生成的时候，跳转的目标地址尚不能确定，只能先把这条指令记录下来，当目标地址确定以后再将地址回填到指令参数里**。用于记录指令的信息称为重定位信息。条件语句的重定位信息只要有一条就够了，但循环语句因为存在 `break` 语句，所以重定位信息可能有多条，这就需要使用容器对它们进行管理。

字节码的指令不会超过 256 个，一个字节足够编码所有指令。另外，基于栈的字节码还有一个优势，就是参数都在栈上，所以指令参数的个数很少，最多只有一个参数，这就让字节码变得简洁，也让编译器的实现和虚拟机的实现都变得简单很多。

最后，我还展示了一个真实的虚拟机例子。典型的虚拟机的结构就是使用循环语句不断地取出字节码进行解析执行，循环体里包含一个巨大的 `switch-case`。Python 虚拟机和 Java 虚拟机都有类似的实现。


到这里，关于 C 语言编译过程的系列加餐先暂告一段落了。最后想说的是，编译器的设计是一个十分复杂的工程，因此很难在短短几篇文章中把中端后端的相关知识全部介绍给你。希望以后还有机会能在课程里带你从头开始实现一个完整的编译器。

青山不改，绿水长流，后会有期。我们有机会再见。

分享给需要的人，Ta 订阅超级会员，你最高得 50 元

Ta 单独购买本课程，你将得 20 元

 生成海报并分享

 赞 1  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。

 领资料

[上一篇](#) 加餐 | 和 C 语言相比，C++ 有哪些不同的语言特性？

[下一篇](#) 大咖助阵 | Tony Bai: Go 程序员拥抱 C 语言简明指南

操作系统实战 45 讲

从 0 到 1, 实现自己的操作系统

彭东

网名 LMOS

Intel 傲腾项目关键开发者



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。

精选留言 (1)

 写留言



一个工匠

2022-04-02

悟了，感谢



 1

领资料