



下载APP



16 | 生成本地代码第1关：先把基础搭好

2021-09-13 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 17:46 大小 16.28M



你好，我是宫文学。

到目前为止，我们已经初步了解了 CPU 架构和 X86 汇编代码的相关知识点，为我们接下来的让编译器生成汇编代码的工作打下了不错的基础。

不过，我总是相信最好的学习方法就是实践。因为，只有你自己动手尝试过用编译器生成汇编代码，你才会对 CPU 架构和汇编的知识有更深刻的了解，总之，遇到问题，解决问题就好了。



所以呢，今天这节课，我们就开始着手生成 X86 汇编代码。我会带你分析生成汇编代码的算法思路，理解寄存器机与栈机在生成代码上的差别，以及了解如何在内存里表示汇编代码。

看上去工作有点多，不着急，我们一步步来。那首先，我们就通过一个实例，让你对生成汇编代码的算法思路有一个直觉上的认知。

生成汇编代码的算法思路

在前面的课里，我们已经学会了如何生成字节码。你也知道了，基本上，我们只需要通过遍历 AST 就能生成栈机的字节码。

但当时我们也说过，为栈机生成代码是比较简单的，比寄存器机要简单，这具体是为什么呢？

你可以保留这个疑问，先来跟我分析一个例子，看看我们要怎么把它转化成汇编代码，以及在这个过程中会遇到什么问题：

```
1 function foo(a:number, b:number):number{  
2   let c = a+b+10;  
3   return a+c;  
4 }
```

[复制代码](#)

你可以看到，这个函数有两个参数。为了提高程序的运行效率，在参数数量不多的情况下，参数通常都是通过寄存器传递的，我们暂且把传递这两个参数的两个寄存器叫做 r1 和 r2。

接下来，我们要执行运算，也就是 $a+b+10$ 。这该怎么做呢？这里你要注意的是，在生成指令的时候，我们通常不能直接把 b 的值加到 a 上，也就是从 r2 加到 r1 上。因为这样就破坏了 r1 原来的值，而这个值后面的代码有可能用到。

所以呢，我们这里就要生成两条指令。第一条指令，是把 a 的值从 r1 拷贝到一个新的寄存器 r3；第 2 条指令，是把 b 的值从 r2 加到 r3 上，最终结果也就保存到了 r3。

```
1 mov r1, r3  
2 add r2, r3
```

[复制代码](#)

之后，我们要再加 10。这个时候，我们可以放心地把 10 加到 r3 上。因为 r3 是我们自己生成的一个临时变量，我们可以确保其他代码不会用到它，所以我们可以放心地改变它的值。

```
1 add 10, r3
```

[复制代码](#)

接着，我们要把表达式 $a+b+10$ 的值赋给本地变量 c。对于本地变量，我们也是尽可能地把它放到寄存器里。不过呢，由于 r3 作为临时变量的任务已经圆满完成了，所以这个时候，我们可以用 r3 来表示 c 的值，这下我们就节省了一个寄存器。因此，这里我们不需要添加任何新的指令。

再接着，我们要计算 $a+c$ 的值。为了不影响已有寄存器的值，我们又使用了一个新的寄存器 r4，用来保存计算结果：

```
1 mov r1, r4
2 add r3, r4
```

[复制代码](#)

最后，我们要把计算结果返回。通常，根据调用约定，返回值也是放在某个寄存器里的。我们这里假设这个寄存器是 r0。所以，我们可以用这两条指令返回：


```
1 mov r4, r0
2 ret
```

[复制代码](#)

到此为止，我们就已经成功地为 foo 函数生成了汇编代码。当然，这个汇编代码只是示意性的、逻辑性的，如果要让它成为真正可用的汇编代码，还要做一些调整，比如把寄存器的名称换成正式的物理寄存器的名称，如 rax 等。我这样叙述，是为了尽量保持简洁，避免你过早陷入到具体 CPU 架构的细节中去，增加认知负担。

好了，回顾我们的工作成果，你可能很快会发现一个问题：**这么小的一个程序就占据了 4 个寄存器，如果我再加多点参数或者本地变量，那寄存器岂不是很快就会被用光？**

这个担心是很有道理的，你可以先看看下面的例子：

 复制代码

```
1 function foo(a:number, b:number, d:number, f:number):number{
2   let c = a+b+10;
3   let g = ...
4   let h = ...
5   return g+h;
6 }
```

在这个例子中，参数 a、b、d、f 和本地变量 c、g、h，都会额外占用一个寄存器，并且每个变量的计算过程都有可能消耗额外的寄存器来保存临时变量，所以寄存器很快就会被用光。

到这里，你可能已经体会到了为什么给寄存器机生成代码会更难了。在栈机里，根本没有这样的问题，因为我们可以用操作数栈来保存中间结果，而操作数栈的大小是没有限制的。

那我们要如何解决寄存器数量有限的问题呢？

这就涉及到寄存器分配算法了。寄存器分配算法能够让程序最大限度地利用有限的物理寄存器，在物理寄存器数量不够的情况下，会把数据写到内存（通常是在栈里），把寄存器腾出来。

不过，我们现在还不是学习寄存器分配算法的时候，我们会在后面会有专门一节课来实现寄存器分配算法。我们现在的目标，是用相对简单的办法来生成汇编代码和可执行程序。

那在不用寄存器算法的情况下，我们还能怎么办呢？

其实，现代编译器都有一些算法可以快速生成汇编代码（机器码），而不用去做太多优化。特别是在 JIT 编译器中，有时编译速度本身是最重要的，要求能够快速生成机器码，对优化程度反而要求不高，比如 JVM 中的 C1 编译器就是这样的。有合适的机会，虚拟机才会调用优化编译器，对代码进行深度地优化。

在上一节课，我们已经见识到了编译器生成的未经优化的汇编代码。在那样的代码里，你可以把所有的参数和变量全部写到栈里，每次用到的时候就从栈帧里读出来。用内存来保存变量，那当然就不受寄存器数量的限制了，所以我们很容易就能生成汇编代码。

不过，在进行运算的时候，还是要用到寄存器的，比如加减乘数等操作，目标操作数要放在寄存器里，而不能直接在内存进行运算。所以，假设本地变量 a、b、c 都是内存中的变量，那么 $c=a+b$ 对应的汇编代码就需要引入一个寄存器，也就是：

```
1 mov a, r1
2 add b, r1
3 mov r1, c
```

[复制代码](#)

你看到，r1 正是我们在前面的分析过程中所引入的临时变量。所以，在算法中只需要为临时变量分配物理寄存器就好了。

好，大的算法思路已经有了。那么在生成汇编代码之前，我们还需要设计一套数据结构。这套数据结构用于在内存里表示汇编代码，编译程序要先生成这些结构化的数据对象，然后再把这些数据对象输出成文本格式，才能获得汇编代码文件。

内存数据结构

目前，我们自己的编译器已经把文本格式的源代码，编译成 AST 和符号表这两个内部数据结构了。而我们现在的任务，是要把这些内部数据结构编译成汇编代码那样文本结构。

在之前的课程里，我们曾经基于 AST 和符号表来生成字节码，那时候我们遍历一下 AST 就生成了，算法比较简单。

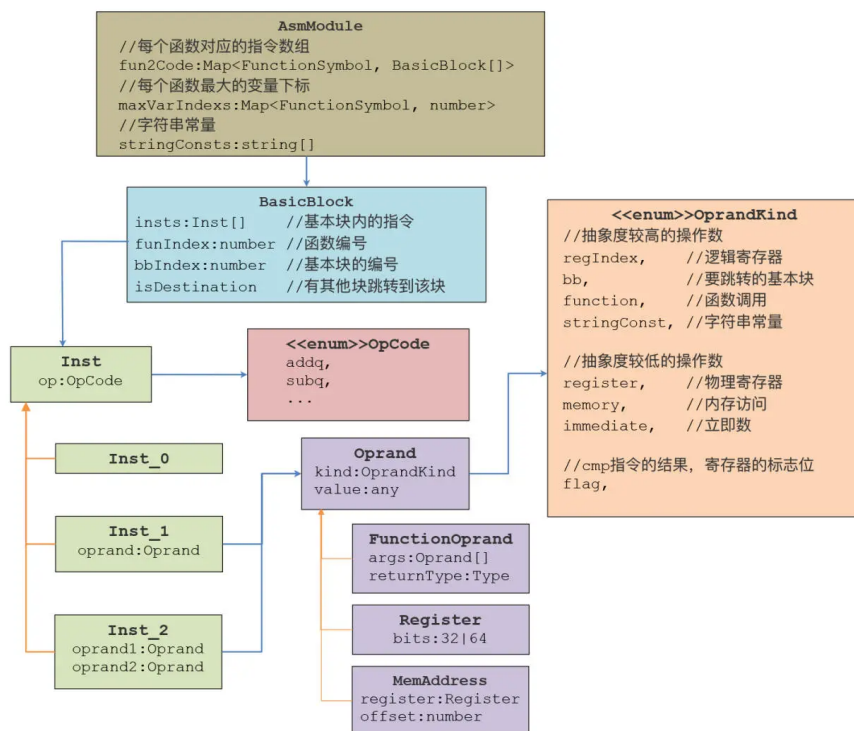
但是生成汇编代码的过程就要复杂一些了，我们如果要直接通过遍历 AST 生成汇编代码，是比较困难的。所以，我们还要再设计一些中间的数据结构，那我们现在就先基于 AST 和符号表来生成这些中间的数据结构。

基于这个中间的数据结构，我们可以做多步的处理工作。比如，我们给每个函数生成的汇编代码要添加序曲和尾声部分，在序曲部分呢，我们要把这个函数用到的需要 Callee 保护的寄存器压到栈里。

而这个函数会用到哪些 Callee 保护的寄存器呢？这需要我们把函数体的代码生成后才能知道。像这样比较多步的处理过程，基于一个内存数据结构来做就会比较容易一些。等都处理完毕了，我们再基于这个内存数据结构生成汇编代码。

注意，我们有时候也把这种内存数据结构看做是一种 IR，也就是中间代码。因为它也是我们程序的一种表示方式。这种表示方式比较贴近底层物理机的机制，所以也被叫做 LIR。

我把当前用于在内存里表示汇编代码的数据结构画成了一张类图，你可以看一下。



我一层层地给你分析一下这个数据结构，它包含了模块、基本块、指令、操作数这些对象。

先看第一层，AsmModule 代表了一个模块，或者说相当于一个汇编代码的文件。

然后是第二层，BasicBlock 代表了一个基本块。

什么是基本块呢？基本块是一系列顺序执行的指令。也就是说，你只能从第一条指令开始运行这个基本块，从最后一条指令离开基本块。没有办法从中间的某一条开始执行，也没有办法从中间离开。如果有一个跳转指令跳到这个基本块，那也只能跳转到基本块的第一条指令。基本块的最后一条指令可能是个跳转指令，可以跳转到别的基本块或者当前基本块。

在我们上一节课的例子中，一个函数只有一个基本块。但如果我们用了 if 语句和循环语句，那就会形成多个基本块，多个块之间通过跳转指令跳转，这个现象我们在生成字节码时已经看到了。

再进一步，因为从一个基本块可以跳转到另一个基本块，那么一个函数的多个基本块通过跳转语句就会连成一个图，这个图就叫做**控制流图 (CFG)**。CFG 是用于程序优化的一种重要的数据结构。Illum (C 和 C++ 等语言的编译器) 和 Go 语言的优化算法，就是典型的基于 CFG 的结构。

我们接着看第三层，Inst 代表了一条指令。每个指令都有一个操作码 (OpCode)。它的子类 Inst_0、Inst_1 和 Inst_2 则分别代表没操作数的指令、一个操作数的指令和两个操作数的指令。

最后一层，Operand 代表了一个操作数。我们知道，在汇编代码里，操作数只有三类：立即数、寄存器和内存访问。在这里，我又添加了几个抽象度比较高的类型，比如逻辑寄存器、基本块、函数、字符串常量等。在后面的课程里，我会再跟你解释为什么需要这些抽象度较高的种类。这里操作数的类型用枚举量 OperandKind 表示。

好了，这些就是表示汇编代码的内存数据结构。基本的算法思路理好了，内存数据结构也有了，接下来我们终于可以着手生成汇编代码了。

为基础功能生成汇编代码

我们还是先针对我们这节课一开头的那个示例程序，先把生成汇编代码的逻辑走通，生成一些基础功能的汇编代码。之后，我们再来处理 if 语句、for 循环语句、函数调用等等复杂的情况，具体的代码你可以参考代码库里的 AsmGenerator。

你可以看到，我们整个算法仍然是通过遍历 AST 来完成的。在遍历 prog 节点的时候，我们就会生成主程序对应的 LIR，也就是基本块、指令、操作数这些。我们可以把这个主程序看做一个函数，相当于 C 语言的 main 函数。而在遍历到函数声明节点的时候，程序就会生成这个函数所对应的 LIR。

你可以重点看看几个关键节点的处理过程。

首先，我们在处理整数字面量节点（`visitIntegerLiteral`）的时候，只需要生成一个立即数类型的 `Oprand` 返回给上级节点就行了。

[复制代码](#)

```
1 visitIntegerLiteral(integerLiteral: IntegerLiteral): any {  
2     return new Oprand(OprandKind.immediate, integerLiteral.value);  
3 }
```

然后，在处理变量（`visitVariable`）的时候，我们要生成一个变量下标类型的 `Oprand`。

[复制代码](#)

```
1 visitVariable(variable: Variable): any {  
2     if (this.functionSym != null && variable.sym != null) {  
3         return new Oprand(OprandKind.varIndex, this.functionSym.vars.indexOf(v  
4     }  
5 }
```

这里你可能会问：怎么会有一个变量下标类型的操作数呢？我们之前讲汇编代码的时候，并没有这种类型的操作数呀？

其实，这就是按照我们前面的算法思路来的。在一开始，我们会认为当前 CPU 有无限个寄存器，每个寄存器有一个下标，对应一个变量，包括参数、本地变量和临时变量。所以，变量下标就可以看做是一个逻辑寄存器。之后，在寄存器分配算法中，我们再根据物理寄存器的数量，把这些逻辑寄存器对应到物理寄存器。如果逻辑寄存器的数量超出了物理寄存器的数量，我们再到栈里为逻辑寄存器分配内存空间。

接下来我们再看看如何处理二元表达式（`visitBinary`）。思路是这样的，我们首先获得左右两颗子树的返回值，也就是两个 `Oprand`。然后基于这两个 `Oprand` 生成指令，并存入当前的基本块，最后的返回值也是一个 `Oprand`，也就是存放表达式结果的那个寄存器。

[复制代码](#)

```
1 visitBinary(bi: Binary): any {  
2     ...  
3     let left = this.visit(bi.exp1) as Oprand; // 左边操作数  
4     ...  
5     let right = this.visit(bi.exp2) as Oprand; // 右边操作数  
6     ...  
7 }
```



```
8     let dest = ...    //计算出一个目标操作数
9
10    switch(bi.op){
11        case Op.Minus: //'-'
12            insts.push(new Inst_2(OpCode.subl,right, dest));    //组合成一条指令
13            ...
14    }
15 }
```

在这节课前面的分析中，我们就讨论过临时变量的使用场景，以及如何尽量节省临时变量所占用的寄存器数量的问题。这里的做法也是一样的，在我们计算 $a+b+10$ 的时候， a 和 b 各自已经占据了一个逻辑寄存器，但 $a+b$ 的结果还需要用一个临时变量来保存，并且我们也为这个临时变量申请一个逻辑寄存器 t 。那我们在计算 $t+10$ 的时候呢，就不用再申请一个新的逻辑寄存器了，我们只需要把结果保存到 t 里就行了。

[复制代码](#)

```
1 //计算出一个目标操作数
2 let dest: Oprand;
3
4 //确定应该返回的Oprand，尽量重用已有的临时变量，并且释放不再使用的临时变量
5 if (this.isTempVar(left) || bi.op == Op.Assign){ //'='
6     dest = left;
7     if (this.isTempVar(right)){
8         this.deadTempVars.push(right.value);
9     }
10 }
11 else{
12     dest = new Oprand(OprandKind.varIndex, this.allocateTempVar());
13 }
```

从这个过程中，我们可以得到一个结论：**无论这个表达式有多长，我们其实只需要少量临时变量就可以了，这样我们能最大程度地节省寄存器。**

另外，如果有些临时变量没有用了，那么我们的程序需要能够复用这些临时变量。比如，对于 $a * 3 + b * 4$ 这样的表达式， $+$ 号两侧都会生成两个临时变量，我们先假设它们是 $t1$ 和 $t2$ 。但最后整个表达式的计算结果，只需要保存在 $t1$ 里就行了（把 $t2$ 的值加到 $t1$ 上），那 $t2$ 这个逻辑寄存器就空出来了。我们就可以把它加到一个 `deadTempVars` 的列表中，并在申请下一次申请临时变量时复用。

好了，算法逻辑我就先写这些。基于上面这些逻辑，我们已经可以把简单的函数编译成汇编代码了，你可以运行编译器试一下。编译器会生成以.s 结尾的文件，然后你再用 clang 或 gcc 把这个.s 文件编译成可执行文件就可以了。

而且，我已经把这些操作变成了 Makefile 的配置，比如，你用"make example"命令就可以把 example.ts 编译成 example.s，然后再生成可执行文件 example 了。

课程小结

好了，今天的课就到这里了，我希望你记住下面这几个知识点：

首先，为寄存器机生成代码，最大的难点是物理寄存器的数量是有限的。通常在算法上我们会首先采用逻辑寄存器，假设寄存器的数量是无限的。之后，再把逻辑寄存器映射到物理寄存器或栈中的内存。

其实，**这种先采用比较抽象的指令，然后再转化成更加具体的实现的过程，叫做 Lower。**Lower 的思路是贯穿在整个编译过程中的。源代码的抽象度是最高的，最接近人类理解的，机器码是最具体的，是便于机器理解的。我们把源代码编译成机器码的过程，就是一个不断 Lower 的过程，中间我们可能会设计多级的中间代码，也就是 IR。后面的课程中，我们还会继续体会这种 Lower 的过程。

第二，为了生成汇编代码，我们需要设计一套内存数据结构，用来表示模块、基本块、指令和操作数等。多个基本块会构成 CFG，CFG 是很多编译器中进行代码优化算法的基础。

在内存中的这套数据结构，其实也是一种中间代码，而且是比较靠近物理实现的中间代码，所以被叫做 LIR，L 是 Low Level 的意思。

在下节课里，我们会涉及到更多的知识点，包括如何实现简单的寄存器分配算法（也就是把逻辑寄存器映射到物理寄存器）、为 if 语句和 for 循环语句生成代码，以及维护栈帧等等。

思考题

通过最近几节课的内容，我相信你已经了解寄存器机的特点了。而在之前的课程中，我们设计的虚拟机是一个栈机。那如果我现在再请你实现一个虚拟机，这次用寄存器机，你有

什么设计思路呢？如何设计它的指令？如何让它比栈机具有更高的性能？请分享一下你的想法。

感谢你和我一起学习，也欢迎你把这节课分享给更多对生成本地代码感兴趣的朋友。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 15 | 汇编语言学习（二）：熟悉X86汇编代码

下一篇 17 | 生成本地代码第2关：变量存储、函数调用和栈帧维护

精选留言 (1)

💬 写留言



奋斗的蜗牛

2021-09-14

老师很厉害，我看完这块，都有茅塞顿开的感觉

展开 ∨



👍 1