第五部分

毋哲

代码生成

任务：

代码生成是编译程序过程的最后一段，它的输入是之前编译前端的中间代码和符号表，它的输出是与源程序语义等价的目标程序代码（汇编代码）。应该注意的是，代码优化阶段应该位于代码生成阶段之前，但是在编写编译程序的过程中，一个比较好的顺序是先进行代码生成部分的编写，这是因为一个编译器没有代码优化但是有代码生成也可以生成可运行的目标代码，先写出正确的可以运行的不具有代码优化的程序，可以为后面代码优化阶段减少一些烦恼。如果先写代码优化，再写代码生成，这时如果生成不正确的代码，我们无法知道是优化部分出了错，还是生成部分出了错，这也不利于我们调试。

必须熟悉汇编语言

熟悉汇编语言，不只是要熟悉汇编语言的语法，更要熟悉一个编译器应该生成的汇编语言结构。如每个函数块开始的堆栈结构。

```
0040CAD0    push        ebp  //保存寄存器，要知道 ebp 的作用是用来指示堆栈段栈底的指针，作用是方便寻址
0040CAD1    mov         ebp,esp  //esp 是栈顶指针，此时的栈顶就是栈底，因此将 esp 的值传到 ebp 中。
0040CAD3    sub         esp,44h //给局部变量、临时变量申请空间
0040CAD6    push        ebx   //保存全局寄存器的值
0040CAD7    push        esi   //保存全局寄存器的值
0040CAD8    push        edi   //保存全局寄存器的值
```

以上的语言规则是 VC 6.0 中的汇编语言规则，因此可以被看成是一种比较推荐的语法规则。但是我们完全可以不限于这种方法来设计自己各个寄存器在程序中的用途，但是要保证程序的正确性。（自己的一点小经验，不知是否正确，也不知是否有参考价值）拿我当初的编译器来说，由于我自己实现没有仔细阅读 VC 编译出的汇编代码，基本上程序的结构都是自己设计的。我是先保存了全局寄存器，然后把此时 esp 当做这个堆栈段的栈底，然后将 esp 的值赋给 ebp，之后才给局部变量和临时变量申请了空间。而且为了寻址的简便，我在堆栈段中又给形式参数申请了一个空间，这样再之后的寻址中我只需要找到变量在这个函数内的序号（第几个申请的空间），就可以立马计算出这个变量的地址，而不需要判断这个变量时形参还是实参。而 VC 6 是直接通过[ebp+X]来找出调用这个函数的函数在调用时压入要传递的参数至堆栈的地址。

要想写好代码生成，大家必须熟悉的知识有：汇编语言有关于寻址和堆栈段的知识，仔细观察一些已经比价成熟的编译器生成的汇编的一些规范。

如何给局部变量和临时变量申请空间，以及之后在程序中的寻址问题

给局部变量和临时变量申请空间是在一个函数最开始的时候。这就要求我们首先需要知道这个函数中使用了多少个变量，局部变量的数量比较好知道，可以通过符号表直接获得，而临时变量如果在生成的时候没有保存的话，就需要扫描一遍函数把临时变量的个数数出来了。当数出来变量个数的时候，就可以为变量申请空间了。申请空间很简单，只需要一句话：

sub esp,x*4

X 表示变量的个数，乘以 4 是因为现在使用的空间大多是 32 位，4 个字节，因此需要乘以 4。

要说明的是，这句话可以保证你在内存中为每个临时变量和局部变量申请好空间，但是不能保证你每一个变量都确定了自己空间的地址是什么。比如说一个整型临时变量 a，当你使用它的时候，应该用刚才申请的那么多的空间中的哪一个呢？这就需要我们在编译器中使用一种方法记录每个变量申请的地址是哪一个。我当时是使用了一个数组来记录。

下面需要讨论的一个问题是，我们这个地址应该用什么来表示。是直接用内存中的地址来表示吗？显然比较麻烦，而且我们在编译程序的时候根本无法知道现在所使用的内存是哪一块内存，因此我们需要用另一种方法，以栈底当做一个基址来间接寻址。如果我们知道了栈底的地址，那我们只需要知道当前变量相对于栈底的偏移量即可，基址我们可以在函数最开始通过保存 esp 的值来获得，因此我们所需要保存的便是每个变量相对于栈底的偏移量。

我们在函数的时候都会用 ebp 来保存栈底地址

mov ebp,esp

之后给变量生成空间

sub esp,(变量使用的字节数)

然后在我们需要寻址的时候,假设变量 a 相对于栈底的偏移量是 24,我们便可以用如下方式获得内存中的值：

[ebp-24]

这样我们无法知道变量在内存中地址的问题也就迎刃而解。

编译顺序！=程序运行顺序

这是大多数人一开始都意识不到的一个问题，即编译的顺序并不是程序运行的顺序。编译的顺序是按照程序的书写顺序，从上到下逐句进行编译，而程序的运行顺序是不确定的，这需要根据不同的语句来判断，但是总而言之，程序的运行顺序很有可能不是按照书写顺序从上至下运行。

这个问题本来是我在代码优化的阶段发现的，拿到代码生成的部分来说是因为希望大家早一些意识到这一点，防止在生成代码阶段可能出现不必要的错误。

我当时是这么发现这个错误的。为了希望提高一点生成的汇编语言的可靠性，我在每个基本块(代码优化部分会讲到)结束的时候清空临时寄存器池，并且在函数返回的时候清空所有的寄存器。清空寄存器的操作是将寄存器中的值传回给占用这个寄存器的变量中去。

看下面这个程序

```
1：   a=b;
2：   if (a)
3：      return;//----------------------------------------①
4：   c=a;      //----------------------------------------②
```

假设 a 被分配了全局寄存器，按照我原来的做法，当从上至下，顺序生成目标代码的时候，在第 1 句的地方，b 的值给了 a，实际上是给了 a 所占用的全局寄存器。第 2 句是判断语句，第 3 句是 return，这时我会清空全局寄存器，这样 a 的值又会写回内存中，并且 a 原来所占用的寄存器不会被占用。到了第 4 句，由于 a 原先占用的寄存已经不被 a 占用，因此此时应该将 a 内存中的值赋值给 c。

这时我们看程序执行的顺序，这个程序有 2 种执行的顺序，1 种是 1-2-3，另一种是 1-2-4。第一种的结果不会出现错误，但是我们看第二种。当运行到第四句时，由于在①这个地方，我们已经清空了全部的寄存器，因此运行到②的时候我们以为 a 不占有寄存器，进而直接去

操作内存，但是事实上程序运行的过程中不会运行到①，所以 a 所占用的寄存器并没有将其中的值传回给 a 内存中的单元，因此在②这个地方，c 被赋的值其实是 a 最一开始的值（a 在刚分配了全局寄存器时的值），因此便会出现错误。

这类错误出现的原因就是我们误将编译顺序当成了程序的执行顺序，因此我们需要在编写编译器的过程中注意这点。

何时清空寄存器？

通过上面讨论的问题，我们自然而然想到了到底什么时候需要清空寄存器的占用呢？大家都知道的是每当基本块结束的时候要清空临时寄存器（这里不会出现上面错误的原因是因为在每一个基本块中，程序就是按照书写顺序从上至下来运行的）。

那对于上例中带来诸多麻烦的全局寄存器呢？其实全局寄存器不需要有清空这个操作，因为全局变量并不会参与到全局寄存器分配的过程中，而且根据全局寄存器分配的算法，如果一个局部变量放弃全局寄存器以后，它的值也就不会再被用到，因此也就不需要有写回这步操作。事实上被分配了全局寄存器的变量根本不需要申请内存空间，直接用寄存器操作即可。

所以，所有清空寄存器的操作只有一个，那便是在基本块结束的时候清空所有临时寄存器的占用即可。

记得给代码优化留出代码段，以防到时候改动过大容易出错

大部分同学在编写代码生成的时候都会使用一种中间代码对应一组汇编代码的方法，由于我们是先写代码生成，后写代码优化，因此在之前很容易忽略代码优化在代码生成的地方产生的影响。我认为在写代码生成的时候，及早地为代码优化空出一些空间是非常有效的解决代码杂乱的办法。下面以我自己的一个例子来解释一下如何在代码生成阶段为代码优化留出一些空间以便后来修改。

```
int AddAsm()//+语句四元式对应的汇编代码生成
{
    GetStr(Middle[AsmIndex].op2,s1);//获得第 2 元所对应的汇编字符串，可能是寄存器，可能是全局变量，可能是立即数，可能是地址
    GetStr(Middle[AsmIndex].op3,s2); //获得第 3 元所对应的汇编字符串，可能是寄存器，可能是全局变量，可能是立即数，可能是地址
    addr=GetStr(Middle[AsmIndex].op4,s3);//获得第 4 元所对应的地址,-2 表示占用了全局寄存器，这里不需要考虑立即数的情况。
    fprintf(outAsm,"\t\tmov\t\teax , %s\n",s1);
    fprintf(outAsm,"\t\tadd\t\teax , %s\n",s2);
    if (!IfOptimize)//如果没有优化的情况下，直接 eax 中的值放回 s3 表示的地址中。
        fprintf(outAsm,"\t\tmov\t\t%s , eax\n",s3);
    else
        //在这里加临时寄存器分配的优化，开优化的情况下，eax 的值可能放回到全局寄存器中或临时寄存器中
    {}
    return 1;
}

*********************************************************************************
```

```
int GetStr(char op[],char s[])//获得 op 参与汇编运算的字符串，取值可为寄存器，全局变量，地址，立即数
{
    addr=-1;
    RegNum=GotWholeReg(op);//查看是否占用全局寄存器，如果占用的话返回寄存器编号
    if (RegNum>=0)
    {
        addr=-2;
    }
    else
    {
        RegNum=OccupyReg(op);//查看是否占用临时寄存器，如果占用的话返回寄存器编号
        if (RegNum>=0)//已占用寄存器
        {
        }
        else
        {
            addr=FindAddress(op);//寻找变量对应的地址
            if (addr>-1)
            {//说明是局部变量
                //返回地址字符串：“[ebp-X]”
            }
            else
            {
                if (('a'<=op[0] && 'z'>=op[0])||
                    ('A'<=op[0] && 'Z'>=op[0])||
                    (op[0]=='_'))
                {//说明是全局变量
                    //返回变量名
                }
                else
                {//说明是立即数
                    //返回立即数的字符串。
                }
            }
        }
    }
    return addr;
}
```

在上面的代码中，如果将红色部分去掉，程序并没有错，而且如果不考虑代码优化的情况下，大多数人都会先写黑色部分的样子，但是一旦加入了代码优化（红色字）,我们会发现程序要改动的地方会有很多，对于简单判断语句的修改其实还是比较好改的，比如只需加

入一个变量和一些判断语句，即可表示是否开启优化。如 if (!IfOptimize)。但是对于那些有关语义方面的修改，比如上例中返回值 addr 所表示的意思，这就比较难改了，而且很容易搞乱一个人的思路。所以最好的办法就是在最一开始就设置好所有情况，即使我们还没有编写代码优化，但是我们在运行程序的时候完全可以不开启代码优化，这样也不会影响到程序的正确性。

编写代码生成程序的大致流程

1，学习相关汇编知识

2，设计运行栈的结构

3，编写内存分配模块和变量初值的存储

4，为每种四元式编写一个模块（注意留出代码优化的空间）