



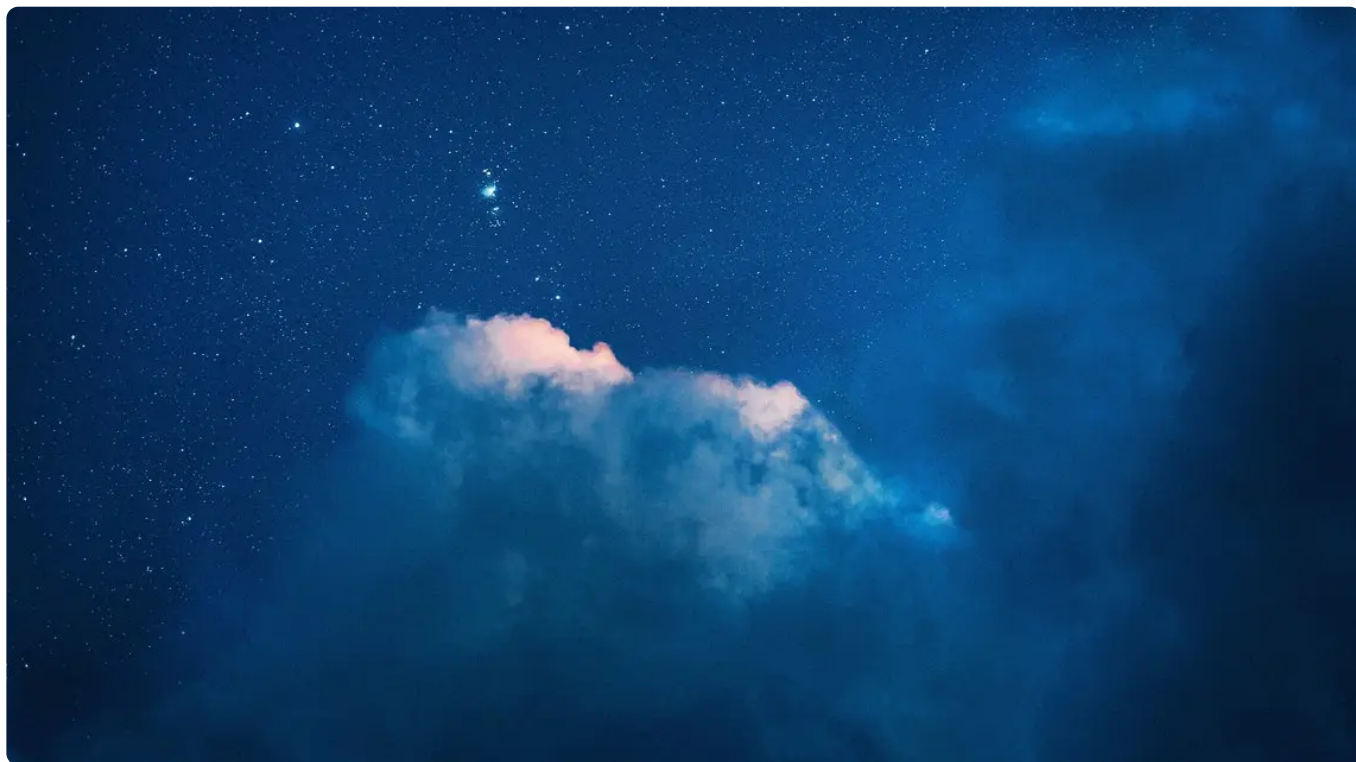
下载APP



## 18 | 生成本地代码第3关：实现完整的功能

2021-09-17 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 12:41 大小 11.62M



你好，我是宫文学。

到目前为止，我们已经把挑战生成本地代码的过程中会遇到的各种难点都解决了，也就是说，我们已经实现基本的寄存器分配算法，并维护好了栈帧。在这个基础上，我们只需要再实现其他的语法特性就行了。

所以，在今天这节课，我们要让编译器支持条件语句和循环语句。这样的话，我们就可以为前面一直在使用的一些例子，比如生成斐波那契数列的程序，生成本地代码了。然后我们可以再比较一次不同运行时机制下的性能表现。



还记得吗？我们在前面已经分别使用了 AST 解释器、基于 JavaScript 的虚拟机和基于 C 语言的虚拟机来生成斐波那契数列。现在我们就看看用我们自己生成的本地代码，性能上

是否会有巨大的变化。


在这个过程中，我们还会再次认识 CFG 这种数据结构，也会考察一下如何支持一元运算，让我们的语言特性更加丰富。

那首先我们来看一下，如何实现 if 语句和 for 循环语句。

## 支持 if 语句和 for 循环语句

在前面的课程中，我们曾经练习过为 if 语句和 for 循环语句生成字节码。不知道你还记不记得，其中的难点就是生成跳转指令。在今天这节课，我们会完成类似的任务，但采用的是一个稍微不同的方法。

我们还是先来研究一下 if 语句，看看针对 if 语句，编译器需要生成什么代码。我们采用下面一个 C 语言的示例程序，更详细的代码你可以参见代码库中的 [if.c](#)：

 复制代码

```
1 int foo(int a){
2     if (a > 10)
3         return a + 8;
4     else
5         return a - 8;
6 }
```

C 语言的编译器针对这段示例程序，会生成下面的汇编代码（参见代码库中的 [if.s](#)），我对汇编代码进行了整理，并添加了注释。

这段汇编代码是未经优化的。不过，我相信你经过前面课程的训练，应该可以看出来很多可以用手工优化的地方。不过现在我们关注的重点是跳转指令，所以你可以重点看一下代码中的 `cmpl` 指令、`jle` 指令和 `jmp` 指令。

```

foo:                                     ## @foo
    ## 序曲
    pushq    %rbp
    movq     %rsp, %rbp

    movl     %edi, -8(%rbp)  #把参数1保存到栈里，此处可优化
    cmpl     $10, -8(%rbp)  #把参数1与立即数10作比较，也就是计算a-10的值
    jle LBB0_2              #如果a<=10,跳转到else块
## %bb.1:                          #这里是if块。如果a>10,会直接执行这个块。
    movl     -8(%rbp), %eax
    addl     $8, %eax
    movl     %eax, -4(%rbp)
    jmp LBB0_3              #跳转到尾声
LBB0_2:                          #else块
    movl     -8(%rbp), %eax
    subl     $8, %eax
    movl     %eax, -4(%rbp)
LBB0_3:                          ##尾声
    movl     -4(%rbp), %eax
    popq     %rbp
    retq

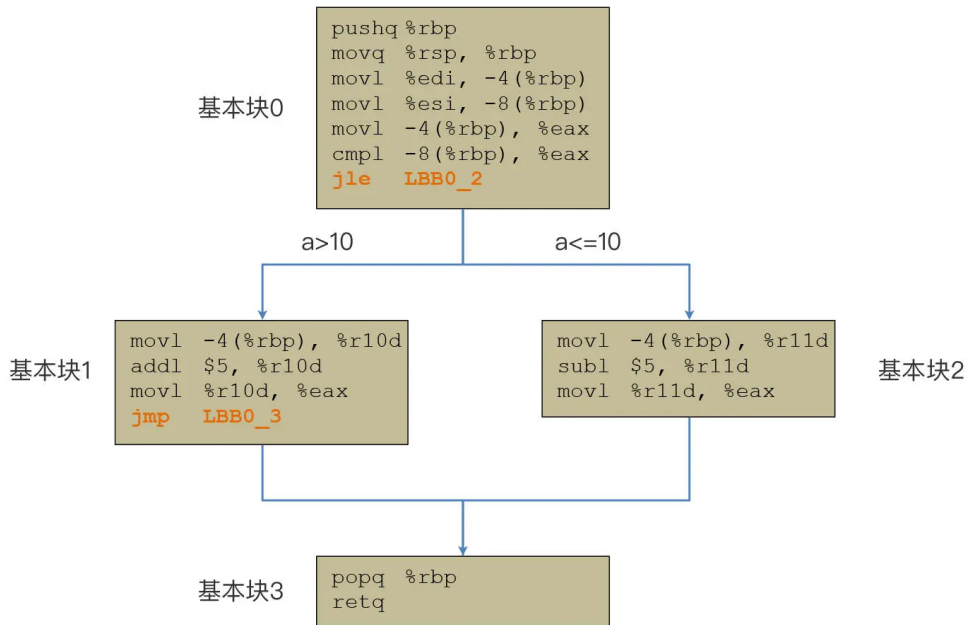
```

我们现在来分析一下。第一个 `cmpl` 指令的作用是比较两个整数的大小。在这个例子中，是比较 `a` 和 10 的大小，计算结果会设置到 `eflags` 寄存器中相应的标志位。

第二个 `jle` 指令，它的作用是根据 `eflags` 寄存器中标志位决定是否进行跳转，如果发现是小于等于的结果，那么就进行跳转，这里是跳转到 `else` 块。如果是大于呢，就会顺着执行下面的指令，也就是 `if` 块的内容。

最后我们来看 `jmp` 指令，这是无条件跳转指令，相当于我们前面学过的字节码中的 `goto` 指令。

认识了这三个指令以后，我们就知道程序的跳转逻辑了。在这个 C 语言的示例程序中，一共有四个基本块，我把它们之间的跳转关系画成了图，可以更加直观一些：



在分析清楚了整个思路以后，为 if 语句生成本地代码的逻辑也就很清楚了，我们现在就动手吧。完整的代码你可以查看代码库里的 [visitIfStmt 方法](#)，我这里挑重点和你分析一下。

首先，我们要生成 4 个基本块：

复制代码

```

1 //条件
2 let bbCondition = this.getCurrentBB();
3 let compOprand = this.visit(ifStmt.condition) as Oprand;
4
5 //if块
6 let bbIfBlock = this.newBlock();
7 this.visit(ifStmt.stmt);
8
9 //else块
10 let bbElseBlock:BasicBlock|null = null
11 if (ifStmt.elseStmt != null){
12     bbElseBlock = this.newBlock();
13     this.visit(ifStmt.elseStmt);
14 }
15
16 //最后，要新建一个基本块,用于If后面的语句。
17 let bbFollowing = this.newBlock();
  
```

接着，我们要添加跳转指令，在 4 个基本块之间建立正确的跳转关系。这其中，最关键的就是我们怎么来为基本块 0，也就是 if 条件所在的基本块生成跳转指令。

你会看到，在 [visitBinary](#) 函数中，我们为 if 条件中的比较表达式生成了一条 `cmpl` 指令。那这个时候，`visitBinary` 为上级调用者返回什么操作数呢？

回忆一下，在上一节课，当我们为其他表达式生成指令之后，会返回一个操作数，代表该指令的执行结果会放在该操作数中，这个操作数通常会是一个逻辑寄存器。不过，`cmpl` 指令跟加减乘除等指令不同，该指令不会改变目标寄存器的值，而是改变 `eflags` 寄存器的值。

为了体现这个逻辑，我就使用了一种新的操作数类型，也就是 `flag` 类型。这样的话，比较运算就返回一个 `flag` 类型的操作数就好了，而且这个操作数里也会记录下比较运算符。

[复制代码](#)

```
1 case Op.G:  
2 case Op.L:  
3 case Op.GE:  
4 case Op.LE:  
5 case Op.EQ:  
6 case Op.NE:  
7     insts.push(new Inst_2(OpCodes.cmpl, right, dest));  
8     dest = new Oprand(OprandKind.flag, this.getOpsiteOp(bi.op));  
9     break;
```

现在，根据这个操作数中记录的比较运算符，我们可以生成正确的跳转指令了。比如，如果比较运算符是大于，那么生成的跳转指令就是小于等于，也就是 `jle`。跳转到哪里呢？跳转到基本块 2，对应的指令是 `jle LBB0_2`。其中，`0_2` 代表第 0 个函数的第 2 个基本块。

不过，在这个阶段，我并没有基于计算出这个标签值来，而是采用了类型为 `“bb”` 的一个操作数，指向目标基本块即可。接下来，我们会再通过一个 [Lower 过程](#)，计算出每个基本块准确的标签值。在这里，我们又一次使用了抽象类型的操作数，这会使指令生成过程得到简化。

这里你要注意，在编译器生成基本块的过程中，有些基本块可能是空的块，也就是块中没有代码。这也很容易理解，因为有些 Block 可能就是空的。那么在 Lower 的过程中，这些空块也会被去除掉。

好了，到目前为止，我们已经能够为 if 语句正确的生成指令了。

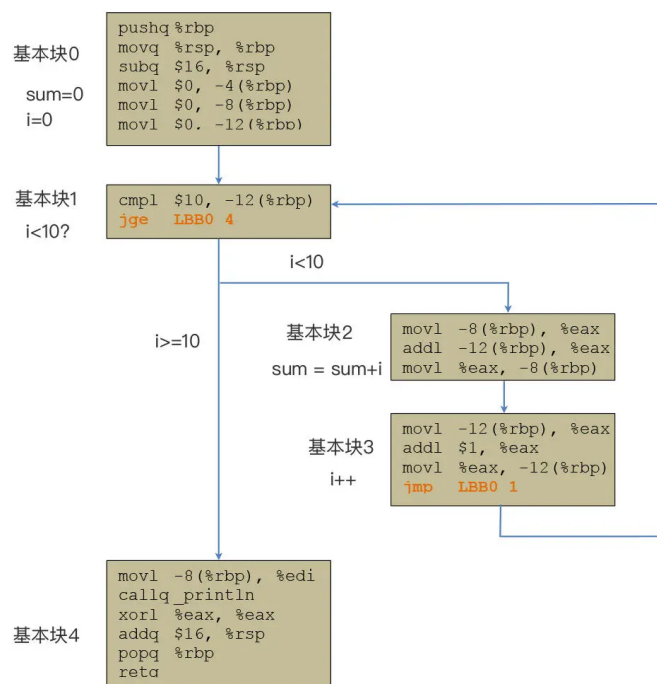
那接下来，我们为 for 循环语句生成指令的要点也是类似的，都是生成多个基本块，并且用跳转指令进行正确的拼接。你可以参考一下代码库里的 [example\\_for.ts](#)。

我这里也摘取了一个关键的示例程序，我们来简单分析一下：

[复制代码](#)

```
1 let sum:number = 0;
2
3 for (let i:number =0; i<10; i++){
4     sum = sum + i;
5 }
6
7 println(sum);
```

这个示例程序的 CFG 是这样的（基于 [for.s](#) 生成）：



你可以多熟悉一下这些 if 和 for 循环结构所对应的 CFG 图，因为我们后面在学习寄存器分配算法的时候，还要跟这样的 CFG 打交道。

在实现 for 循环语句的过程中，我们还会碰到一个小的技术点，也就是一元运算。一元运算，特别是 ++ 和 -- 这样的运算符，实现起来有点特殊，也值得我们稍微探讨一下。

## 实现一元运算

在 for 循环语句中，我们通常都会使用一个递增变量。比如在 [example\\_for.ts](#) 中，你就会看到 i++ 这种表达式。

对于 i++ 来说，我们需要返回 i 的当前值，然后再让 i 的值加 1。但对于 ++i 来说，则是先让 i 的值加 1，然后返回增加后的值。

这里的具体实现你可以参照代码库里 [visitUnary](#) 方法，下面我摘取了其中处理 ++ 和 -- 的代码片段，我们来分析一下。

 复制代码

```
1 let operand = this.visit(u.exp) as Operand;
2
3 //用作返回值的Operand
4 let result:Operand = operand;
5
6 //++和--
7 if(u.op == Op.Inc || u.op == Op.Dec){
8     let tempVar = new Operand(OperandKind.varIndex, this.allocateTempVar());
9     insts.push(new Inst_2(OpCodes.movl, operand, tempVar));
10    if(u.isPrefix){ //前缀运算符
11        result = tempVar;
12    }
13    else{ //后缀运算符
14        //把当前操作数放入一个临时变量作为返回值
15        result = new Operand(OperandKind.varIndex, this.allocateTempVar());
16        insts.push(new Inst_2(OpCodes.movl, operand, result));
17        this.s.deadTempVars.push(tempVar.value);
18    }
19    //做+1或-1的运算
20    let opCode = u.op == Op.Inc ? OpCodes.addl : OpCodes.subl;
21    insts.push(new Inst_2(OpCodes, new Operand(OperandKind.immediate,1), tempVar)
22    insts.push(new Inst_2(OpCodes.movl, tempVar, operand));
23 }
```



对于 `++i` 来说，计算逻辑大致如下面的伪代码。也就是我们先申请一个临时变量 `t1`，把 `t1` 加 1，再赋给 `i`，最后返回 `t1`。这个时候返回值就是 `i+1`。

[复制代码](#)

```
1 t1 = i
2 t1 = t1+1
3 i = t1
4 return t1
```

但对于 `i++` 来说，它的计算逻辑就要多引入一个临时变量 `t2` 了，并且一开始要让 `t1` 和 `t2` 都等于 `i`。`t1` 的作用，仍然是 `i` 增加了 1，但返回值却是 `t2`，也就是 `i` 原来的值。

[复制代码](#)

```
1 t1 = i
2 t2 = i
3 t1 = t+1
4 i = t1
5 return t2;
```

除了 `++` 和 `--` 外，还有其他的一元运算符，如 `+` 和 `-`。

对 `+` 号的处理比较简单，实际上我们直接忽略就行了。其实，忽略 `+` 号的这个动作甚至不用等到生成汇编代码的环节，我们应该在代码优化的环节就把它去掉。

对于 `-` 号，它的计算逻辑则是用 0 去减当前的值。在这个过程中，我们仍然需要用到临时变量，你可以看看下面这个代码：

[复制代码](#)

```
1 //-
2 else if (u.op == Op.Minus){
3     let tempVar = new Operand(OperandKind.varIndex, this.allocateTempVar());
4     //用0减去当前值
5     insts.push(new Inst_2(OpCode.movl, new Operand(OperandKind.immediate,0), tem
6     insts.push(new Inst_2(OpCode.subl, operand, tempVar));
7     result = tempVar;
8     if (this.isTempVar(operand)){
9         this.s.deadTempVars.push(operand.value);
10    }
11 }
```



好了，在实现了 if 语句和 for 循环语句以后，我们几乎已经完成生成本地代码的所有工作了。那现在又到了检验我们工作成果的时候了，我们再做一次性能比拼，看看这一次，我们的程序性能有多高。

## 再一次进行性能比拼

我们还是用斐波那契数列的例子来做测试，你用 `make example_fibo` 命令就可以编译并生成汇编代码和可执行文件，并且能够打印出每次执行所花费的时间。你还可以用 `make fibo` 命令编译 C 语言版本的示例程序，用来做性能比较。

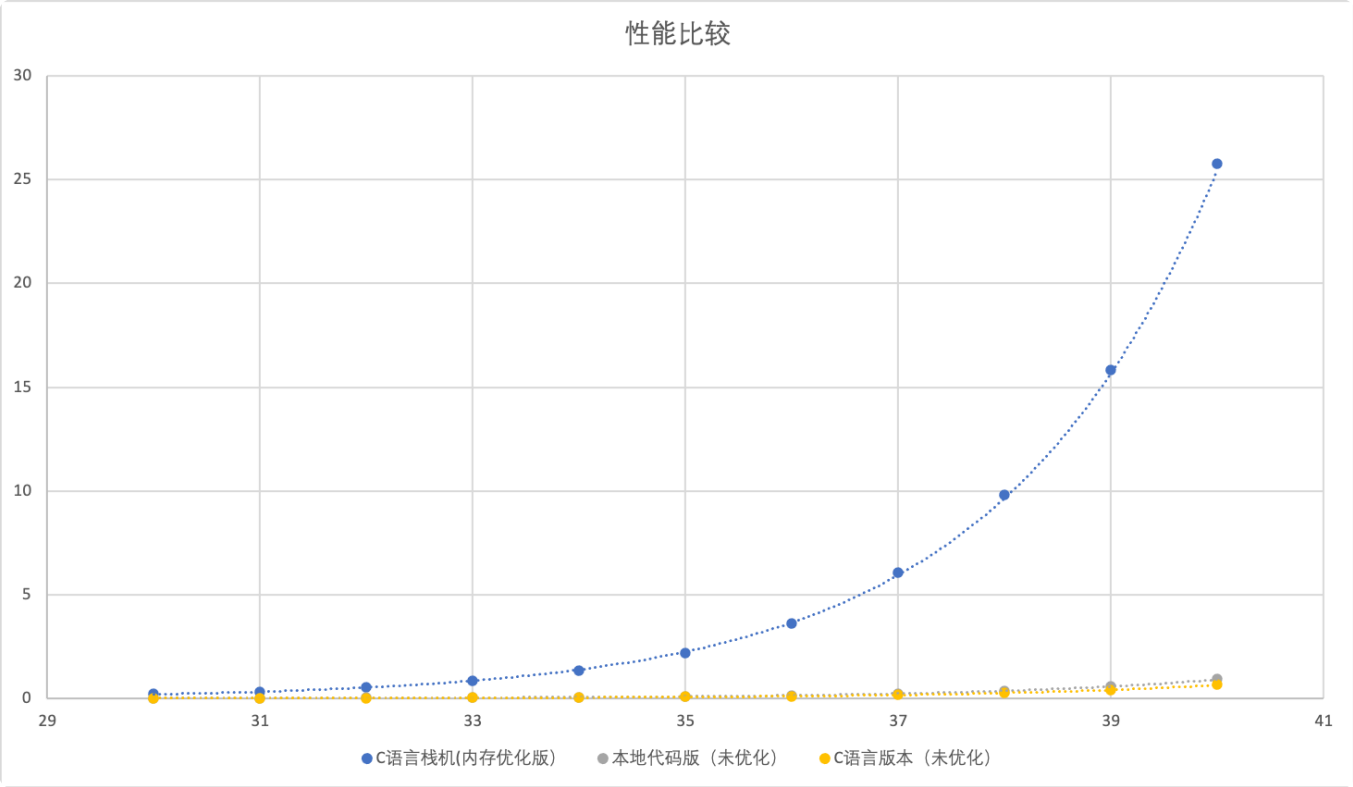
我把比较结果放在下面的表格中，这次，我们只比较 C 语言的栈机和本地代码版就可以了，因为本地代码版速度实在是快太多了，以至于跟 AST 解释器和 TypeScript 版的栈机做比较已经没有太大意义了。

n	C语言栈机 (内存优化版) (单位：秒)	本地代码版 (未优化) (单位：秒)	C语言版 (未优化) (单位：秒)	C语言栈机/ 本地代码版
30	0.215	0.009852	0.008354	21.82
31	0.318	0.014253	0.009532	22.31
32	0.516	0.023645	0.014249	21.82
33	0.833	0.038226	0.025849	21.79
34	1.349	0.05912	0.038808	22.82
35	2.168	0.087446	0.069654	24.79
36	3.599	0.13645	0.101521	26.38
37	6.086	0.226655	0.161691	26.85
38	9.797	0.365186	0.257798	26.83
39	15.847	0.587676	0.419833	26.97
40	25.773	0.954535	0.681619	27.00

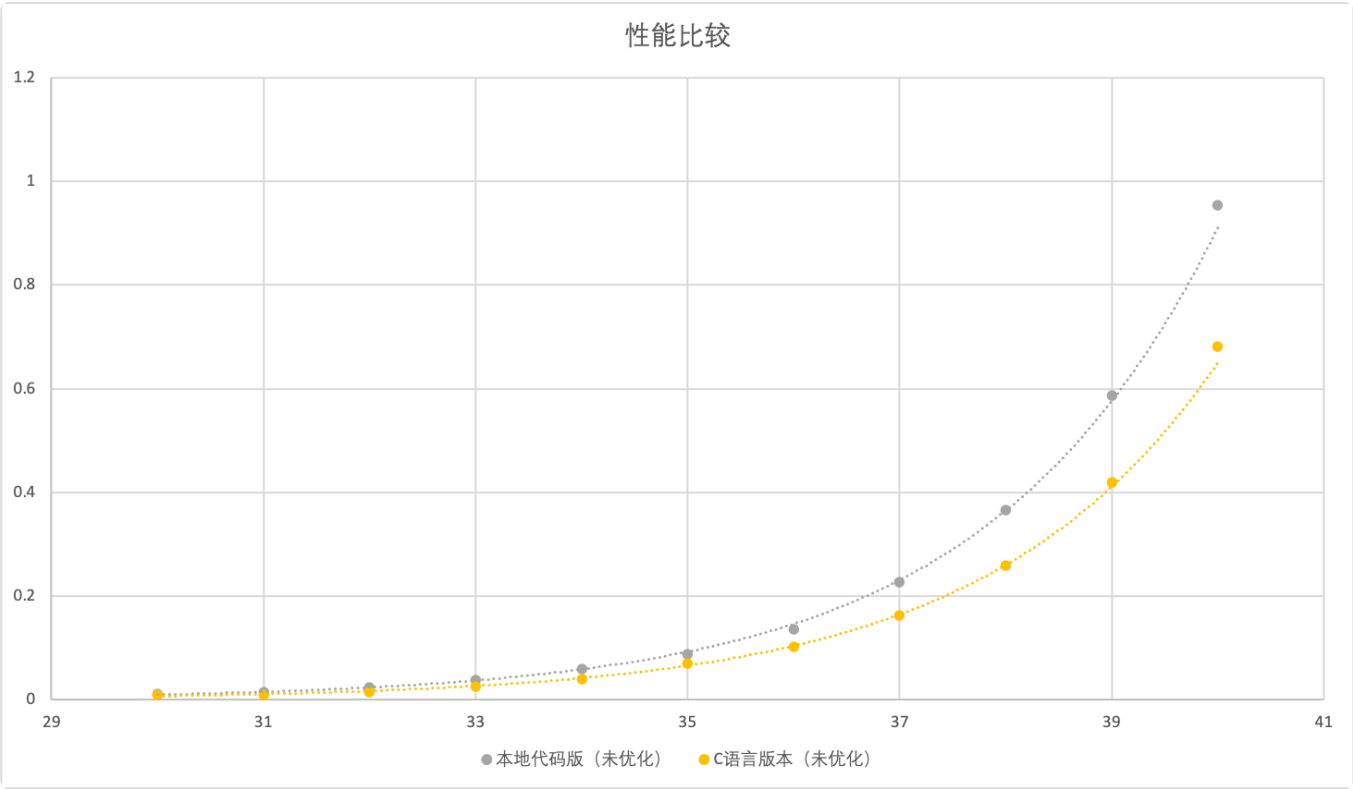


从表格可以看出，本地代码版的速度基本上是栈机版的 20 多倍，可以说性能有了极大的提升。

另外，我们还把用纯 C 语言写的版本 ( [fibonacci.c](#) ) 的性能加到了表格里，作为比较。我还绘制了曲线图。



你可能不放大这张图仔细观察，还注意不到本地代码的曲线在哪里。因为我们生成的本地代码和 C 语言生成的本地代码，性能上的差异已经比较小了，不过也还是有一小点差异的。我把它俩的对比又单独画了一张图，你再看一下。



那这种差异来自于哪里呢？我们不是都用了相同的寄存器分配算法吗？

如果你比较一下二者的汇编代码，还是能看出一小点差别来的。**最主要的差别，就在于如何处理 Caller 需要保护的寄存器。**

在我们这门课当前使用的算法里，我们记录了在一个函数中，如果有 Caller 保护的寄存器被使用了，那么我们在每次调用函数的时候，要把所有被使用的这些寄存器都保存到内存里。

这里有一些浪费。因为其实有些寄存器不再用了，也就意味着它不再需要保护了。如果我们进一步优化一下的话，这里其实需要一个算法，来检测哪些临时变量仍然是在使用中的，然后保护它们正在使用的寄存器就可以了。

我们在下节课里，在实现升级版的寄存器分配算法的时候，就会做变量活跃性的检测。所以，这种优化我们会在下一节课一并实现。

## 课程小结

这就是今天这节课的所有内容了。这节课的内容相对简单，你只需要记住以下几个知识点就可以了：

首先，我们又重温了一下如何生成跳转指令。对于条件跳转来说，首先要把两个操作数做比较，比较结果会被设置到 `eflags` 寄存器中。这样，我们后面的条件跳转指令就可以根据 `eflags` 寄存器的内容来做跳转了。并且，对于条件跳转，我们再一次看到了一个现象，就是生成的跳转指令跟比较操作符恰好是相反的。对于 `>` 号的比较运算符，生成的反倒是 `jle` 指令，也就是如果小于等于，则跳转。

不过，我们目前的 `if` 条件仍然只支持一个简单的条件表达式，还不支持把多个条件表达式组合成一个逻辑表达式的情况。你有兴趣的话，可以研究一下怎么扩展这方面的特性。

另外，对于一元运算，我们也稍微展开讲了一下。你要注意，特别是针对 `i++` 这样的表达式，怎样实现不同的当前值和返回值

最后，我们又一次用斐波那契数列做了一下性能的对比测试。相比 C 语言的栈机版本，这次我们的性能提高了 20 多倍。看来，我们这好几节课努力学习底层架构和汇编代码的相关技术，还是很值得的。

掌握了我们当前的生成汇编代码的技术，其实你已经可以去实现很多有用的底层功能了，比如做一个图形化的开发工具，用来生成能够在物联网设备、自动设备上运行的机器码。

不过，我们当然不会满足于现在的成绩。下节课，我们将升级寄存器分配算法，继续提升程序的性能！

## 思考题

今天这节课里，我们画了两个 CFG 的图。CFG 是一种图的数据结构。那么，我们例子中的两个图有什么特点呢？这个特点对于图的处理有什么好处呢？

欢迎在留言区分享你的看法。由于我们后面会经常用到图的数据结构，所以你现在要开始把有关图的算法熟悉起来了！我是宫文学，我们下节课见。

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   17 | 生成本地代码第2关：变量存储、函数调用和栈帧维护

下一篇   19 | 怎么实现一个更好的寄存器算法：原理篇

## 精选留言

💬 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。