



下载APP



21 | 加深对栈的理解：实现尾递归和尾调用优化

2021-09-24 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 16:10 大小 14.81M



你好，我是宫文学。

前面几节课，我们在实现生成本地代码的过程中，对汇编语言、栈和栈帧有关的知识点都进行了比较深入的了解。通过这些学习，你应该对程序的运行机制有了更加透彻的理解。

那么今天这节课，作为第一部分起步篇的结尾，我们就来检验一下自己的学习成果吧！具体一点，我们就是要运用我们前面已经学过的知识点，特别是关于栈和栈帧的知识点，来实现两个有用的优化功能，也就是尾递归和尾调用的优化。



这两个优化有助于我们更好地利用栈里的内存空间，也能够提高程序的性能，对于我们后面实现函数式编程特性也具有很重要的意义。另外，这样的练习，也会加深我们对栈和栈帧、对程序的控制流，还有对程序的运行机制的理解。

好了，我们先从尾递归入手吧，说说尾递归是怎么回事，还有它有怎样的运行特点，看看我们为什么需要去优化它。


递归函数和尾递归

学习编程的同学都应该知道，递归是一种重要的思维方式。我们现实世界的很多事物，用递归来表达是非常自然的。**在递归的思维里，解决整体的问题和解决局部问题的思路是相同的。**

在我们这个课程里，我们学习的语法分析的方法，也采用了递归的思维：我们给一个大程序做语法分析，会分解成给每一个函数、每一条语句做语法分析。不管在哪个颗粒度上，算法的思路都是相同的。

递归思想的一个更具体的使用方式，就是**递归函数**。当前的各种高级语言，都会支持递归函数，也就是允许在一个函数内部调用自身。

你可以看看下面这个例子，这个例子是用来实现阶乘的计算的。

 复制代码

```
1 function factorial (n:number):number{
2   if (n < 1)
3     return 1;
4   else
5     return n * factorial(n-1);
6 }
```

在这里， n 的阶乘 $f(n)$ ，就等于 $n * f(n-1)$ 。这是典型的递归思维，解决一个整体问题 $f(n)$ ，能够被转化为解决其局部问题 $f(n-1)$ 。 n 的值变化了，但解决问题的思路是一致的。

最近几年，函数式编程的思想又重新流行起来。在一些纯函数式的编程语言中，递归是其核心编程机制，被大量使用。

不过，递归函数的大量使用，对程序的运行时机制是一个挑战。因为我们已经知道，在标准的程序运行模式下，每一次函数调用，都要为这个函数创建一个栈帧。如果递归的层次很深，那么栈帧的数量就会非常多，最终引起“stack overflow”，也就是栈溢出的错误，这是我们在使用栈的时候最怕遇到的问题。

另外，我们还知道，**当我们在进行函数调用的时候，还会产生比较大的性能开销**。这些开销包括：设置参数、设置返回地址、移动栈顶指针、保护相关的寄存器，等等。特别是，在这个过程中，一般都会产生内存读写的动作，这会对性能产生比较大的影响。

所以说，虽然递归函数很有用，但你在学习编程的时候，可能你的老师会告诉你，如果对性能和内存占用有较高的要求，那么我们尽量不用递归算法实现，而是把递归算法改成等价的非递归算法。

不过，现代编译器也在努力帮助解决这个问题。比如在上一节课中，我们就已经见到了 C 语言编译器的一个功能，它在编译斐波那契数列的过程中，能够把其中一半的递归调用转变成一个循环语句，从而减少了递归调用导致的开销。

但在这一节课呢，我们不会试图一下子就实现这么复杂的编译优化功能，而是先针对递归调用中的一个特殊情况而进行优化，这个特殊情况就是**尾递归**。

那什么是尾递归呢？**尾递归就是在 return 语句中，return 后面只跟了一个递归调用的情况**。在上面的例子中，你会看到 return 后面跟着的是 $n * \text{factorial}(n-1)$ ，这种情况不是尾递归。不过，我们可以把示例程序改写成尾递归的情形，我写在了下面：

[复制代码](#)

```
1 function factorial(n:number, total:number):number{
2   if (n <= 1)
3     return total;
4   else
5     return factorial(n-1, n*total);
6 }
```

这个新的阶乘函数使用了两个参数，其中第二个参数保存的是阶乘的累积值。如果要计算 10 的阶乘，那么我们需要函数 `factorial(10, 1)`。你可以仔细看一下 `factorial` 函数的两个不同的版本，它们确实是等价的。但第二个版本中的第二个 `return` 语句呢，就是一个标准的尾递归调用。

我们为什么要谈论尾递归呢？这是因为尾递归在栈帧的使用上有其独特的特点，使得我们可以用很简单的方法就能实现优化。

那么接下来，我们就分析一下递归函数在栈的使用上的特点，这有利于我们制定优化策略。

递归函数对栈的使用

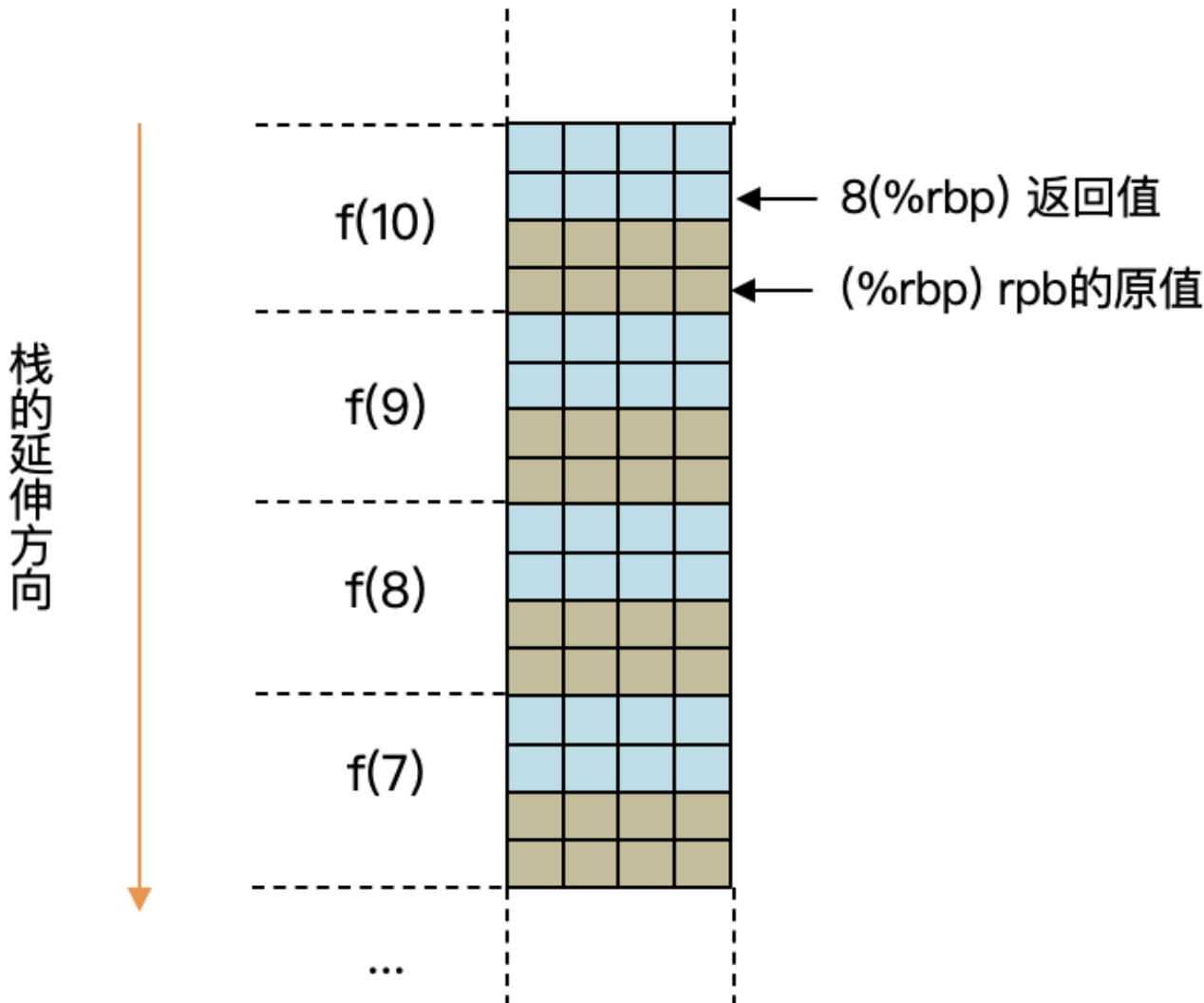
你可以用我们上一节课的 PlayScript 版本，使用 `make example_fact` 命令来生成上面示例程序的汇编代码和可执行文件。

这个汇编文件是没有做尾递归优化的，你可以看一下它的内容，看看它的栈帧是什么结构。

[复制代码](#)

```
1 _factorial:
2     .cfi_startproc
3     ## bb.0
4     pushq    %rbp
5     movq     %rsp, %rbp
6     cmpl     $1, %edi                #   cmpl $1, var0
7     jg      LBB0_2
8     ## bb.1
9     movl     %esi, %eax              #   movl var1, returnSlot
10    jmp      LBB0_3
11 LBB0_2:
12    movl     %edi, %r10d              #   movl var0, var2
13    subl     $1, %r10d               #   subl $1, var2
14    movl     %edi, %r11d              #   movl var0, var3
15    imull    %esi, %r11d              #   imull  var1, var3
16    movl     %r10d, %edi
17    movl     %r11d, %esi
18    callq    _factorial
19    movl     %eax, %edx              #   movl returnSlot, var4
20    movl     %edx, %eax              #   movl var4, returnSlot
21 LBB0_3:
22    popq     %rbp
23    retq
24    .cfi_endproc
```

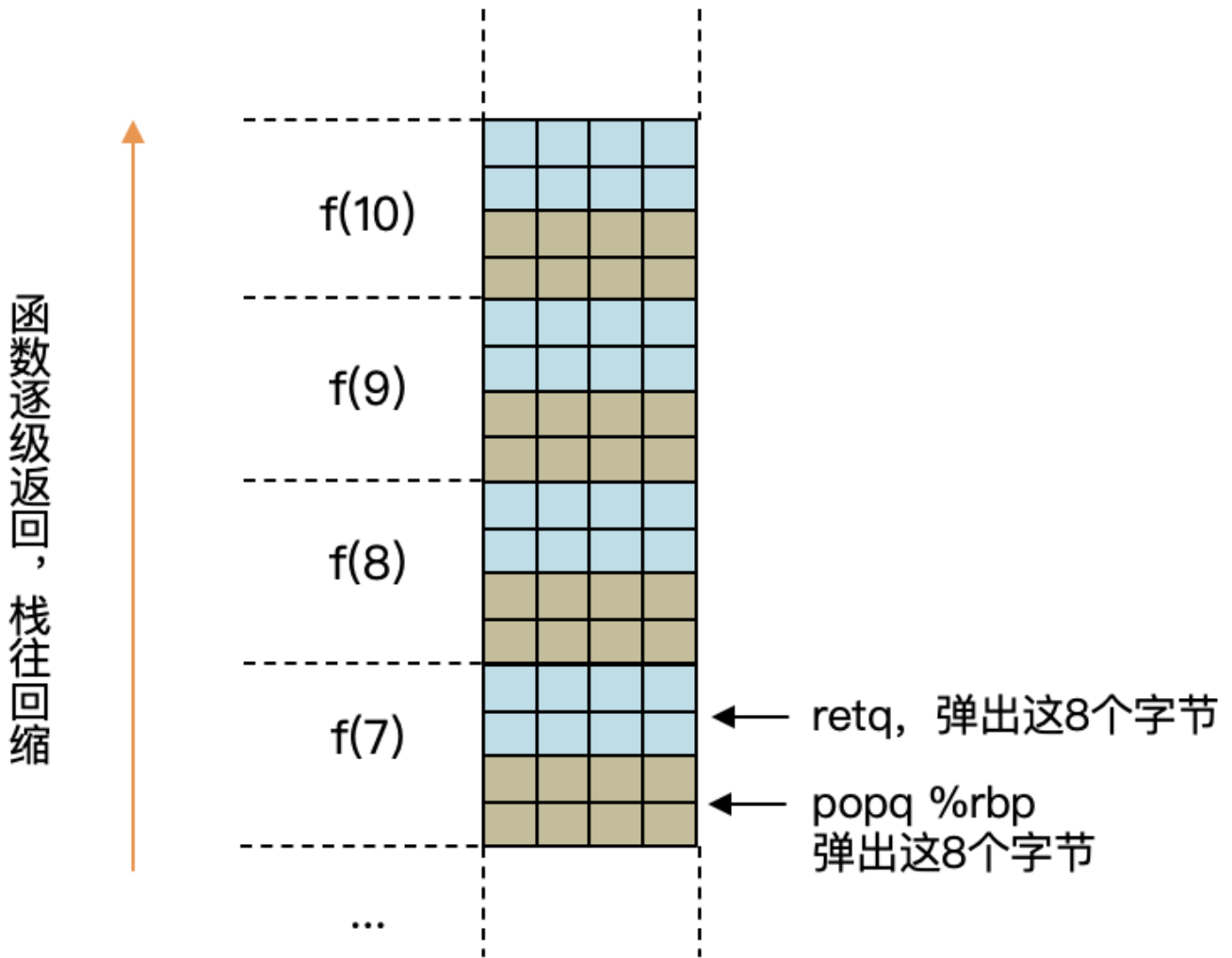
从汇编代码中，你能看出来，这个函数的栈帧特别简单。在 `factorial` 函数中，`n` 和 `total` 两个参数是保存在寄存器中的，在栈帧里只保存了 `rbp` 寄存器的旧值和返回地址。如果递归函数复杂一点，或者我们采用的是比较简单的寄存器分配算法，那么栈帧里可能会保存一些其他信息，比如溢出的变量等。



然后，每次的递归调用，都会创建一个栈帧。递归嵌套多少层，就需要建立多少级的栈帧。如果说我们要计算 10000 的阶乘，那就需要 10000 个栈帧。

这样说的话，到目前为止，我们的编译器所生成的程序，是完全合乎规则的，似乎没啥问题啊？

不过，如果你细看一下，就会发现一个现象：**在每一级调用结束之后，我们都会返回到上一级函数。而且上一级函数也会立即结束，继续返回上一级。这个过程会一直持续下去，直到退出最上面一级的 `factorial` 函数。**



在逐级返回的过程中，其实程序再也没有访问 `n` 和 `total` 这两个本地变量，只有 `%eax` 寄存器是有用的，因为这里面保存了返回值。

这样看起来，这些一层层的上级栈帧，都是没有什么用的，这里的内存空间，完全是浪费的。也就是说，在计算 10000 的阶乘的时候，前 9999 个栈帧其实都是没有用的。另外，寄存器里保存的变量的值，也已经结束了生存期。所以，我们在调用下一级函数的时候，根本没有必要去保护它们原来的值。

既然如此，那我们能不能做一些优化，提高程序的性能呢？

这就需要用到**尾递归优化**的技术了。

尾递归优化

这要怎么优化呢？总的思路是这样的：**既然旧函数的栈帧没有用，那么我们就没有必要为递归调用产生一个新的栈帧，而是复用当前栈帧就行了。**

那怎么实现这一点呢？我先直接说答案吧。我们的解决思路，就是把 call 语句改成一个 jmp 指令，跳到程序的开头重新执行。

这个时候，参数 1 和参数 2 的值已经分别被更新成了 $n-1$ 和 $n * total$ 。所以，当我们跳到程序开头再一次执行的时候，就相当于调用 `factorial(n-1, n * total)`，从而取得了和尾递归调用相同的计算结果，你可以看看下面这个截图。

```

_factorial:
    .cfi_startproc
## bb.0
    pushq    %rbp
    movq     %rsp, %rbp
    cmpl     $1, %edi                # cmpl $1, var0
    jg       LBB0_2                  跳转到程序的开头，重新开始执行。
                                    相当于调用factorial(n-1, n*total)
## bb.1
    movl     %esi, %eax              # movl var1, returnSlot
    jmp      LBB0_3
LBB0_2:
    movl     %edi, %r10d             # movl var0, var2
    subl     $1, %r10d              # subl $1, var2
    movl     %edi, %r11d             # movl var0, var3
    imull    %esi, %r11d             # imull  var1, var3
    movl     %r10d, %edi             参数1, 变成了n-1
    movl     %r11d, %esi             参数2, 变成了
                                    n*total
    callq    _factorial
    movl    %eax, %edx              # movl returnSlot, var4
    movl    %edx, %eax              # movl var4, returnSlot
LBB0_3:
    popq     %rbp
    retq
    .cfi_endproc

```


去掉这两条代码

这里我稍微岔开一个小话题，在上面的图中，你会看到我还把 callq 后面的两条指令打了叉，去掉了。这个原因也很简单，这两条语句其实是做了无用功。第一条指令把 eax 中的值拷贝到 edx，第二条又从 edx 拷贝回了 eax，作为返回值的 eax 其实并没有变化。

你还记不记得，在上一节课的思考题中，我让你去检查，我们目前生成的汇编代码还有哪些地方需要优化，这里就是其中一个例子。不过，具体如何去完善这些代码，我们以后再说。

我们还是回到尾递归的话题上来。对着上面的汇编代码，你可能会发现，这不就是一个循环语句的结构吗？每次循环，就把 n 减 1。也就是说，上面的汇编代码其实相当于下面的

高级语言代码：

 复制代码

```
1 function factorial(n:number, total:number):number{
2   for (; n>=1; n--){
3     if (n<=1)
4       return total;
5     else
6       total = n*total;
7   }
8 }
```

没错！**尾递归调用是一定能够转化成循环语句的**。也正因为如此，所以我们没有必要为每次调用都生成一个新的栈帧，这样就能既避免了栈溢出的风险，又提升了性能。

那么，我们应该如何升级 PlayScript，来实现尾递归的优化呢？

首先，我们必须能够分析出来什么样的函数调用属于尾递归。这个判断起来也比较简单，如果一个 return 语句中的表达式，只有一个递归调用，那么这个递归调用就是尾递归。

我们看看它的 AST 有什么特点。你用 `node play example_fact.ts -v` 命令，可以打印出程序的 AST 信息。你会发现，ReturnStatement 节点只有一个 FunctionCall 子节点，并且 FunctionCall 所调用的函数正是 factorial 自身，是一个递归调用。


```

FunctionDecl factorial
  Return type: number
  ParamList:
    VariableDecl n(number)
      no initialization.
    VariableDecl total(number)
      no initialization.
  IfStatement
    Condition:
      Binary:LE(boolean)
        Variable: n(number), resolved
        1(integer)
    Then:
      ReturnStatement
        Variable: total(number), resolved
    Else:
      ReturnStatement ← return语句
        FunctionCall (number) factorial, resolved
          Binary:Minus(number)
            Variable: n(number), resolved
            1(integer)
          Binary:Multiply(number)
            Variable: n(number), resolved
            Variable: total(number), resolved

```

只有一个子节点，是一个递归调用。

掌握了 AST 的这个特点后，我们很容易就可以写一个分析程序，识别出哪个函数调用是尾递归，从而改变后面生成汇编代码的逻辑，这里具体代码你可以参考代码库里的 [TailAnalyzer](#)。

TailAnalyzer 的运行结果，是一个 [TailAnalysisResult](#) 对象。这个对象里保存了所有尾递归的 FunctionCall 节点。这样，在生成 Asm 的时候，我们就可以针对尾递归专门生成不同的代码了。

接下来，我们再使用这节课的 PlayScript 版本，再次执行 make example_fact，就会生成针对尾递归优化后的汇编代码和可执行程序。我们看一下新版本的汇编代码：

```

_factorial:
    .cfi_startproc
## bb.0
    pushq    %rbp
    movq     %rsp, %rbp
LBB0_1:  ← 这里新加了一个基本块和标签，用于跳转。
    cmpl     $1, %edi                    #  cmpl $1, var0
    jg      LBB0_3
## bb.2
    movl     %esi, %eax                  #  movl var1, returnSlot
    jmp     LBB0_4
LBB0_3:
    movl     %edi, %r10d                 #  movl var0, var2
    subl     $1, %r10d                  #  subl $1, var2
    movl     %edi, %r11d                 #  movl var0, var3
    imull    %esi, %r11d                 #  imull  var1, var3
    movl     %r10d, %edi
    movl     %r11d, %esi
    jmp     LBB0_1  ←  callq指令变成了jmp指令。
LBB0_4:
    popq     %rbp
    retq
    .cfi_endproc

```

在新版本生成的汇编代码里，你会看到我们新加了一个基本块，作为跳转指令的目标。而原来的 `callq` 指令，则变成跳转到这个基本块的一个 `jmp` 指令。

你可以运行一下新版本的程序，你会发现两个版本的运行结果是相同的，也就是说它们的功能是完全等价的。如果你有兴趣的话还可以做一个测试程序，测一下优化前后的性能差异到底有多大。

你可以把阶乘的例子变成一个累加的例子来测试，也就是 $f(n) = n + f(n-1)$ 。为什么呢？因为阶乘的值增加得太快了，很快就会超过整数的范围，比如 16 的阶乘就超出了一个 32 位整数的表达能力。

好了，现在我们已经完成了尾递归的优化。但在这节课的开头，我们还提出了另一个概念，就是**尾调用**。那么尾调用又是什么呢？我们把尾递归和尾调用放在一起介绍，是否意味着我们也可以采用尾递归优化的思路来优化尾调用呢？我们接着往下分析。

尾调用优化

什么是尾调用呢？**尾调用就是在 return 语句后面直接跟一个函数调用的情况**。比如，对于下面两个函数 foo 和 bar，foo 中有一个 return 语句，直接调用了 bar，这就是一个尾调用的情况。

[复制代码](#)

```
1 function foo(p1:number, p2:number):number{
2   ...
3   return bar();
4 }
5 function bar():number{
6   ...
7 }
```

如果你利用我们分析尾递归所获得的知识去分析尾调用，就会发现它们是有相似点的。也就是，**在做尾调用的时候，调用者的栈帧和寄存器已经没有用了，所以被调用者完全没有必要新建一个栈帧，而是复用调用者的栈帧就可以了。**

那么这具体要怎么进行优化呢？其实跟尾递归一样，我们都是使用 jmp 指令来代替 callq 指令就行了。只不过，这一次 jmp 跳转目标，是 _bar，也就是 bar 函数的入口位置。

我们之前就说过，在汇编语言里，函数名称也只不过是一个标签而已，所以它们可以作为跳转指令的目标。而原本 callq 指令，也只不过是相当于两条指令：

[复制代码](#)

```
1 pushq 返回地址
2 jmp 函数标签
```

但在尾调用的情形下，我们没必要修改返回地址。因为 bar 返回 foo 以后，接着就会返回 foo 的调用者。所以，我们这里只用一个 jmp 指令就可以了，这样也能减少保存返回地址导致的性能开销。

具体实现，你可以通过 make example_tail 命令，去编译 [example_tail.ts](#) 示例代码，并研究一下它所生成的汇编代码。

课程小结

好了，到这里我们今天这节课就讲完了。这节课，我们借助尾递归和尾调用优化的话题，加深了对栈、栈帧和程序运行机制的了解。我希望你记住并产生以下这几个认知：

首先，对于任何程序来说，当它在执行 `return` 语句的时候，所有的变量的生存期其实都已经结束了，所以栈帧和变量所占据的寄存器就都没用了。而尾递归和尾调用的优化，就是借助了这一特点，复用了调用者的栈帧，从而达到了节省内存和提高性能的目的。

第二，我已经多次提到一个观点，就是**计算机语言的设计者，其实拥有很大的自由度来决定如何使用栈帧**。这节课的内容也能够再次印证我这个观点，我们对栈和栈帧的使用不是僵化的、一成不变的，而是针对不同的语言特性，我们可以做不同的使用。比如，在实现协程机制的时候，我们使用栈的方式跟传统的函数调用方式也有不同。如果你想进一步了解这方面的知识，你可以参考一下《编译原理实战课》的 [🔗 第 34 节](#)。

最后，这节课也是我们第一次接触编译器中的优化技术。对于编译器来说，第一项任务当然是把高级语言的代码翻译成目标代码，而第二项重要的任务就是在翻译的过程中，做各种优化，尽量保证生成的是最高效的机器码。而优化采用的技术也有很多，并且会发生在编译过程的各个阶段。优化工作其实也是编译器中工作量最大、难度最高的工作。在后面的课程中，我们也会接触到更多的优化技术。

思考题

你会发现，在我们 [🔗 example_tail.ts](#) 这个尾调用的例子中，只有一个程序的出口，也就是只有一个 `return` 语句，所以我们做指令的改写还是比较容易的。那如果存在两个或两个以上的 `return` 语句，并改写其中的一个尾调用语句，让它变成 `jmp` 指令，这样生成的汇编代码又会有什么不同呢？

欢迎你研究一下，并在留言区分享你的发现，这个研究会帮助你加深对程序控制流的理解。我是宫文学，我们下节课见。

资源链接

[🔗 这节课的示例代码在这里！](#)

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 20 | 怎么实现一个更好的寄存器分配算法：实现篇

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。