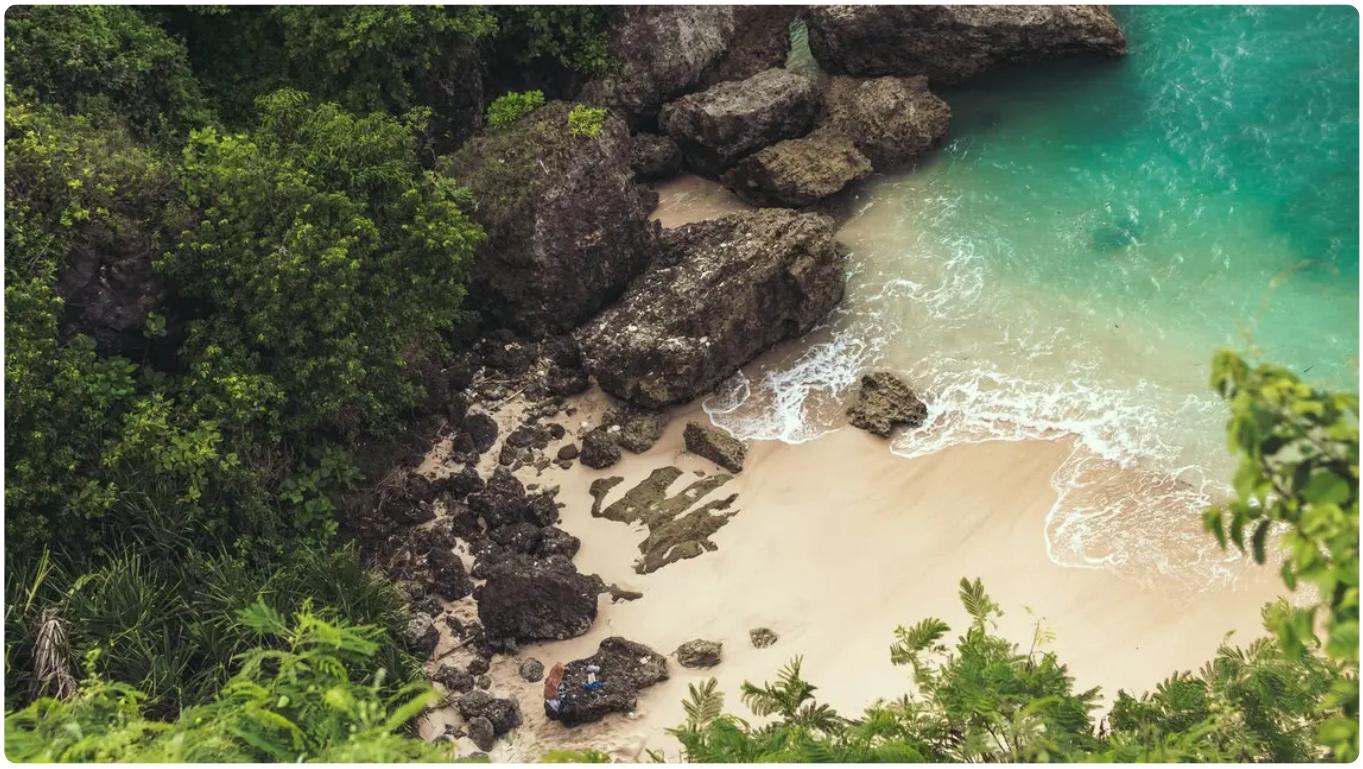
06｜代码封装（下）：函数是如何被调用的？

2021-12-17 于航

《深入C语言和程序运行原理》 课程介绍 >



讲述：于航
时长 10:52　大小 9.96M

你好，我是于航。

在上一讲中，我们主要围绕着 x86-64 平台上 C 函数被调用时需要遵循的一系列规则，即 System V AMD64 ABI 调用规范的内容展开了深入的探讨。而今天，我们将继续讨论有关 C 函数的话题，来看看参数求值顺序、递归调用、旧式声明的相关内容。这些内容将会帮助你更加深入地理解 C 函数的运作细节，写出更加健壮、性能更高的代码。

编写不依赖于参数求值顺序的函数

当一个函数被调用时，传递给它的实际参数应该按照怎样的顺序进行求值，这在 C 标准中并没有被明确规定。因此，对于某些特殊的代码形式，当运行使用不同编译器编译得到的二进制可执行文件时，可能会得到不同的计算结果。比如下面这段代码：

复制代码

```
#include <stdio.h>
int main(void) {
  int n = 1;
  printf("%d %d %d", n++, n++, n++);
  return 0;
}
```

这里，我们使用 printf 函数，连续打印出了表达式 n++ 的值。当使用 Clang 13.0.0 编译器进行编译并运行这段代码时，可以得到输出结果 “1 2 3” 。而换成 GCC 11.2 时，则得到了不同的结果 “3 2 1” 。通过查看汇编代码，我们能够看到：Clang 按照从左到右的顺序来依次计算表达式 n++ 的值，而 GCC 则与之相反。

因此，你需要注意的是：**为了保证 C 程序的健壮性及可移植性，不要编写需依赖特定函数参数求值顺序才能够正常运行的代码逻辑。**

尾递归调用优化

对于“递归函数” ，相信你并不陌生。简单来说，递归函数就是一种自己可能会调用自己的函数。比如在下面的 C 代码中，factorial 函数便是一个递归函数。

复制代码

```
int factorial(int num) {
  if (num == 1 || num == 0)
    return 1;
  return num * factorial(num - 1);
}
```

factorial 函数主要用于计算给定数的阶乘。你可以在上述代码的第四行，看到它对自己的调用过程。接下来，我们使用 GCC 在默认优化等级情况下编译这段 C 代码，可以得到如下图所示的汇编代码：

```
1  int factorial(int num) {
2    if (num == 1 || num == 0)
3      return 1;
4    return num * factorial(num - 1);
5  }
6
```

```
1  factorial:
2          push    rbp
3          mov     rbp, rsp
4          sub     rsp, 16
5          mov     DWORD PTR [rbp-4], edi
6          cmp     DWORD PTR [rbp-4], 1
7          je      .L2
8          cmp     DWORD PTR [rbp-4], 0
9          jne     .L3
10 .L2:
11         mov     eax, 1
12         jmp     .L4
13 .L3:
14         mov     eax, DWORD PTR [rbp-4]
15         sub     eax, 1
16         mov     edi, eax
17         call    factorial
18         imul    eax, DWORD PTR [rbp-4]
19 .L4:
20         leave
21         ret
```

这里，在上图右侧的第 17 行处，我们可以看到 factorial 函数对自己的调用过程。

通过上一讲的学习我们得知，函数调用过程中所需要的数据是以栈帧的形式被存放在进程的栈内存中的。而对栈内存的清理工作，只有当被调用函数执行完毕，准备通过 `ret` 指令返回前，才能够通过调用 `leave` 指令等方式进行。

而对于正常的递归函数来说，由于函数不断调用自己，导致先前调用产生的函数栈帧只有在后续调用的函数正常返回后，才能够得到清理。随着函数的不断调用，产生的栈帧越来越多，因此在栈内存无法再继续增长的情况下，便会发生溢出，进而导致程序出现“Segmentation Fault” 等错误。

除此之外，每次的函数调用都会进行栈帧的创建和销毁过程，而随着函数调用次数的增加，这部分开销也可能逐渐影响程序的外部可观测性能。

那有没有办法解决这两个问题呢？答案是有的，它正是我在这里要介绍的“尾递归调用优化（Tail-Call Optimization）”。

尾递归调用优化是指在一定条件下，编译器可以直接**利用跳转指令取代函数调用指令**，来“模拟”函数的调用过程。而这样做，便可以省去函数调用栈帧的不断创建和销毁过程。而且，递归函数在整个调用期间都仅在栈内存中维护着一个栈帧，因此只使用了有限

的栈内存。对于函数体较为小巧，并且可能会进行较多次递归调用的函数，尾递归调用优化可以带来可观的执行效率提升。

尾递归调用的一个重要条件是：递归调用语句必须作为函数返回前的最后一条语句。怎样理解这个约束条件呢？我们来看下面这个例子：

```
1  int factorial(int n, int acc) {
2    if (n == 0) {
3      return acc;
4    } else {
5      return factorial(n - 1, acc * n);
6    }
7  }
8
```

```
1  factorial:
2          mov     eax, esi
3          test    edi, edi
4          je      .L5
5  .L2:
6          imul    eax, edi
7          sub     edi, 1
8          jne     .L2
9  .L5:
10         ret
```

这里的 C 代码和上面那段功能完全相同，只不过我们修改了函数 factorial 的实现逻辑，并且在编译时指定了最高的编译优化等级 “-O3” 。通过查看右侧的汇编代码，你可以发现，编译器并没有进行任何 `call` 指令的调用过程。而这就是因为它使用了尾递归调用优化。

尾递归调用优化的一个最显著特征，就是编译器会**使用跳转指令（如je、jne、jle等）来替换函数调用时所使用的 call 指令**。这里函数 factorial 在执行 `ret` 指令返回前，会判断寄存器 edi 的值是否为 0（ZF=1），来决定是跳转到 “.L2” 标签处继续“递归”执行该函数，还是直接返回。当然，由于这里“实现递归”的方式是通过跳转指令而非函数的再次调用，在函数 factorial 执行的整个过程中，栈内存中仅有其对应的一个栈帧（是由调用 factorial 的函数通过 `call` 指令创建的）。

此时，如果我们尝试违背尾递归优化的重要前提，会有什么结果呢？来看个例子：在 factorial 函数的第一种实现方式中，由于函数的前一次调用结果依赖于函数下一次调用的返回值，导致存放在栈帧中的局部变量 num 的值无法被清理，因此编译器也就无法通过消除历史函数调用栈帧的方式，来模拟函数的递归调用过程。

而这就是尾递归调用优化以“递归调用语句必须作为函数返回前的最后一条语句”为前提条件的原因。在这种情况下，编译器才能够确定函数的返回值没有被上一个栈帧所使用。

但还有一点需要注意：现代编译器具备十分强大的程序执行流分析能力。在很多情况下，它能够直接提取出程序中可以使用循环进行表达的部分，同时避免 `call` 指令的调用过程。因此，编译器是否采用了尾递归优化，在大多数情况下已经很难直接从程序对应的汇编代码中看出了。而我们能做的，只是根据编译器实现尾递归优化的理论基础，来尽可能地从代码层面优化我们的程序。但实际执行时的效果如何，就要取决于具体编译器的能力了。毕竟，与如今强大的 GCC 与 Clang 等编译器相比，还有很多开源编译器甚至连基本的 C 标准特性都没有完全支持。

尾递归调用优化可以帮助我们减少函数调用栈帧的创建与销毁次数，这个过程涉及到寄存器的保存与恢复、栈内存的分配与释放等。但需要注意的是，**尾递归调用优化的效果在那些函数体本身较小，且递归调用次数较多的函数上体现得更加明显。**这里我们需要平衡的一点是：函数自身的执行时间与栈帧的创建和销毁时间，二者哪个占比更大。很明显，选择优化对性能影响更大的因素，通常会得到更大的收益。

废弃的 K&R 函数声明

在 1989 年 ANSI C 标准出现之前，我们在声明一个 C 函数时，可以不为其指定参数列表。而对于这种方式，我们一般称其为 K&R 函数声明。比如下面这个例子：

复制代码

```
1 #include <stdio.h>
2 int add();
3 int main(void) {
4   printf("%d", add(1));  // ?
5   return 0;
6 }
7 int add(int x, int y) {
8   return x + y;
9 }
```

这里你可以看到，在代码第 2 行函数 add 的声明中，我们并没有为其指定任何形式参数。但在代码第 7 行函数 add 的实现中，该函数在执行时实际上会接收两个整型参数。虽然函数在其声明与定义中使用的参数列表并没有完全匹配，但为了保证程序的兼容性，现代编译器都默认支持这种代码形式。

在继续学习之前，你可以先猜一猜代码第 4 行对函数 add 的调用结果是多少。注意，这里在调用时，我们仅为 add 函数传入了一个实参，即一个整型字面量值 1。

经过实践，理想情况下你会得到结果 1，不过也可能会得到看起来毫不相关的随机数。但无论如何，程序的运行确实偏离了预期，而这也正是 C 语言被标准化前，K&R 函数声明被人诟病的一个原因。

下面，就让我们来看一看，在这种情况下的函数 add 是如何被调用的。使用默认优化等级进行编译，我们得到了如下图所示的汇编代码：

```
1  #include <stdio.h>
2  int add();
3  int main(void) {
4    printf("%d", add(1));
5    return 0;
6  }
7  int add(int x, int y) {
8    return x + y;
9  }
10
```

```
1  .LC0:
2          .string "%d"
3  main:
4          push    rbp
5          mov     rbp, rsp
6          mov     edi, 1
7          mov     eax, 0
8          call    add
9          mov     esi, eax
10         mov     edi, OFFSET FLAT:.LC0
11         mov     eax, 0
12         call    printf
13         mov     eax, 0
14         pop     rbp
15         ret
16 add:
17         push    rbp
18         mov     rbp, rsp
19         mov     DWORD PTR [rbp-4], edi
20         mov     DWORD PTR [rbp-8], esi
21         mov     edx, DWORD PTR [rbp-4]
22         mov     eax, DWORD PTR [rbp-8]
23         add     eax, edx
24         pop     rbp
25         ret
```

沿着在 main 函数内部调用 add 函数的执行链路进行寻找，我们可以轻松地发现问题所在。

在上一讲中我们已经了解过，SysV 调用约定会使用寄存器 rdi、rsi 来传递用户函数调用时的前两个参数。而这里在 main 函数对应的汇编代码中，可以看到 add 函数在被调用前，编译器仅通过蓝框内的汇编指令，对传入 add 函数的第一个参数进行了处理，将它存放到了寄存器 edi 中。而 add 函数在实际执行时，会通过红框内的指令，同时从寄存器 edi、esi 中初始化它所需要的两个参数。因此，此时寄存器 esi 中的值是什么，便决定了该函数的最终返回值。而它可能是 0，也有可能是各种随机数。

总的来看，**出现问题的原因是编译器并没有强制要求函数声明、函数定义，以及函数调用三者的参数列表必须保持一致。**因此，为了杜绝此类问题，ANSI C 标准化之后的 C 语言提出了新的“函数原型”概念，以取代旧时使用的函数声明方式。

和函数声明不同，函数原型强制程序员显式指出函数参数的使用方式，即使在没有参数时，也需要显式地将参数部分指定为 void。同时，对于函数原型、函数定义，以及函数调用，三者的参数列表必须保持一致，否则将无法通过编译。

上面的 C 代码在使用函数原型改写后如下所示：

复制代码

```
#include <stdio.h>
int add(int x, int y);
int main(void) {
  printf("%d", add(1));  // compiling error!
  return 0;
}
int add(int x, int y) {
  return x + y;
}
```

此时若再次进行编译，编译器将会提示“参数不匹配”的错误。

总而言之，言而总之，为了减少产生这种不必要问题的机会，**请不要在 C 代码中使用古老的 K&R 函数声明**。换句话说，每一个函数参数列表都不应该为空。

总结

好了，讲到这里，今天的内容也就基本结束了。最后我来给你总结一下。

这一讲，我主要和你讨论了有关 C 函数的另外三个话题，分别是函数参数求值顺序、尾递归调用优化，以及 K&R 函数声明。

首先，编译器对函数参数的求值顺序并不固定，因此，不要试图编写需要依赖于特定参数求值顺序才能正常运行的代码逻辑。

其次，对递归函数的不正确使用，可能会导致进程栈内存出现溢出。而通过尾递归优化，编译器可以将函数的递归调用实现由 `call` 指令转换为条件跳转指令，从而大大减少函数调用栈帧的产生，进而避免了栈溢出的问题。不仅如此，这种方式也在一定程度上提高了函数的执行性能。

最后，考虑到兼容性，现代编译器仍然支持旧式的 K&R 函数声明式写法，但这种写法极易引入难以调试的程序问题。因此，请确保为每一个函数参数列表都指明它所需要的参数类型。

思考题

在课程的最后，我们来一起做个思考题吧。

除了我在这两讲中介绍的有关 C 函数的内容，现代 C 语言中还增加了很多有关函数的新特性。比如，C11 中新引入了一个名为 `_Noreturn` 的关键字，可参与函数的定义过程。你可以动手查查它的用处，思考它存在的意义，并在评论区交流。

今天的课程就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

上一篇 05 | 代码封装（上）：函数是如何被调用的？

精选留言 (1)

pedro

2021-12-17

_Noreturn：函数不返回到其调用点

展开