作者：代兴　邮箱：503268771@qq.com　Vx 公众号：东方架构师

# 函数体的规则

函数体包含实现具体功能的代码、指令。

函数体的规则：
函数体是函数的主体部分。函数体包含的代码，会被编译为汇编指令。
函数体内部可以调用其他函数，实现函数嵌套。
内联函数，把函数体展开到被调用的地方。

# 比较多个函数有不同的函数体

编写 C 程序： body.c

```
// 没有函数体
void func_no_body(int age)
{
}

// 有简单的函数体
void func_small_body(int age)
{
    int plus = age + 5;
}

// 有很大的函数体
void func_big_body(int age)
{
    int plus = age + 5;
    plus = plus / age;
    plus = plus - age;
}
```

编译 C 程序：

```
gcc -S body.c -o body.s -std=gnu99
```

查看汇编文件 body.s，找到 3 个函数对应的汇编代码。

```
5   func_no_body:
6   .LFB0:
7       .cfi_startproc
8       pushq   %rbp
9       .cfi_def_cfa_offset 16
10      .cfi_offset 6, -16
11      movq    %rsp, %rbp
12      .cfi_def_cfa_register 6
13      movl    %edi, -4(%rbp)
14      popq    %rbp
15      .cfi_def_cfa 7, 8
16      ret
```

```
22  func_small_body:
23  .LFB1:
24      .cfi_startproc
25      pushq   %rbp
26      .cfi_def_cfa_offset 16
27      .cfi_offset 6, -16
28      movq    %rsp, %rbp
29      .cfi_def_cfa_register 6
30      movl    %edi, -20(%rbp)
31      movl    -20(%rbp), %eax
32      addl    $5, %eax
33      movl    %eax, -4(%rbp)
34      popq    %rbp
35      .cfi_def_cfa 7, 8
36      ret
```

```
42  func_big_body:
43  .LFB2:
44      .cfi_startproc
45      pushq   %rbp
46      .cfi_def_cfa_offset 16
47      .cfi_offset 6, -16
48      movq    %rsp, %rbp
49      .cfi_def_cfa_register 6
50      movl    %edi, -20(%rbp)
51      movl    -20(%rbp), %eax
52      addl    $5, %eax
53      movl    %eax, -4(%rbp)
54      movl    -4(%rbp), %eax
55      cltd
56      idivl   -20(%rbp)
57      movl    %eax, -4(%rbp)
58      movl    -20(%rbp), %eax
59      subl    %eax, -4(%rbp)
60      popq    %rbp
61      .cfi_def_cfa 7, 8
62      ret
```

分析结果：
函数 func_no_body 没有函数体，有 0 行 C 代码，汇编代码占用 12 行。
函数 func_small_body 有简单的函数体，有 1 行 C 代码，汇编代码占用 15 行。
函数 func_big_body 有很大的函数体，有 3 行 C 代码，汇编代码占用 21 行。
随着函数体的增加，汇编代码的占用行数也增加。
func_small_body 相比 func_no_body，多了 1 行 C 代码，汇编代码多了 3 行。
func_big_body 相比 func_small_body，多了 2 行 C 代码，汇编代码多了 6 行。

问题：没有函数体的函数，为什么占用那么多空间？
函数 func_no_body 没有函数体，却使用了 12 行汇编代码。简化汇编代码，只保留核心部分，如下。

```
func_no_body:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
```

```
    popq    %rbp
    ret
```

汇编代码 `pushq %rbp` 、 `movq %rsp, %rbp` 、 `popq %rbp` 、 `ret` ，属于函数栈操作。函数栈操作，在上述的 3 个方法中都存在，且汇编代码一样。
汇编代码 `movl %edi, -4(%rbp)` ，在这里没有实际作用，可以省略。
空函数，也包含函数栈操作。函数栈操作，一般不能省略，某些情况可以省略。

函数调用有成本。
成本 1，额外的函数栈操作。指令操作 rbp/rsp，给被调用函数分配函数栈的一个栈帧。
成本 2，影响代码缓存。指令和数据在高速缓存，被调用的函数可能不在高速缓存，需要加载。
如何降低成本？ 使用内联函数，把函数体的代码，直接复制到外层函数，省去函数调用这一步。
使用内联函数，有特定规则。函数体必须简单，不能包含循环等复杂操作。内联函数被调用次数多。编译器有其他隐含规则，不保证完全内联。

# 比较内联函数和普通函数

编写代码： inline.c

```
// 内联函数。使用修饰符前缀
inline __attribute__((always_inline)) int func_with_inline(int num)
{
    // 只有 1 行代码。用 333 标识
    return num + 333;
}

// 普通函数。不使用内联。
int func_without_inline(int num)
{
    // 只有 1 行代码。用 555 标识
    return num + 555;
}

int func_test()
{
    // 循环调用函数多次，增加内联的概率
    for (int k = 0; k < 777; ++k)
    {
        // 依次调用 2 个函数。用 2 个局部变量承接，防止编译器优化掉函数调用。
        int tmp1 = func_with_inline(k);
        int tmp2 = func_without_inline(k);
    }
    return 0;
}
```

编译代码：

```
gcc inline.c -S -o inline.s -std=gnu99
```

分析结果：
为了方便对比代码，内联函数用 333 标识，普通函数用 555 标识。
查看汇编文件 inline.s，查询内联函数的名称 func_with_inline 则没有找到，查询普通函数的名称 func_without_inline 则找到了，说明内联函数的符号消失了。
查询内联函数的标识 333，则发现对应的汇编代码在函数 func_test 的函数体，说明内联函数的函数体被展开了。
查询普通函数的标识 555，则发现对应的汇编代码在函数 func_without_inline 的函数体，说明普通函数没有受到内联的影响。

```
23      .type   func_test, @function
24  func_test:
25  .LFB2:
26      .cfi_startproc
27      pushq   %rbp
28      .cfi_def_cfa_offset 16
29      .cfi_offset 6, -16
30      movq    %rsp, %rbp
31      .cfi_def_cfa_register 6
32      subq    $16, %rsp
33      movl    $0, -4(%rbp)
34      jmp .L4
35  .L6:
36      movl    -4(%rbp), %eax
37      movl    %eax, -16(%rbp)
38      movl    -16(%rbp), %eax
39      addl    $333, %eax
40      movl    %eax, -8(%rbp)
41      movl    -4(%rbp), %eax
42      movl    %eax, %edi
43      call    func_without_inline
44      movl    %eax, -12(%rbp)
```

```
4       .type   func_without_inline, @function
5   func_without_inline:
6   .LFB1:
7       .cfi_startproc
8       pushq   %rbp
9       .cfi_def_cfa_offset 16
10      .cfi_offset 6, -16
11      movq    %rsp, %rbp
12      .cfi_def_cfa_register 6
13      movl    %edi, -4(%rbp)
14      movl    -4(%rbp), %eax
15      addl    $555, %eax
16      popq    %rbp
```