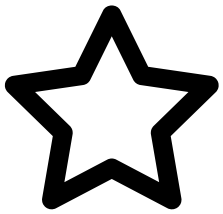
揭密X86架构C可变参数函数实现原理

原创 展开



海枫 最后发布于2018-04-01 00:48:13 阅读数 4016 收藏

前两天公司论坛有同事在问C语言**可变参数函数**中的va_start，va_arg 和 va_end 这些接口怎么实现的？我毫不犹豫翻开箱底，将多年前前（算算有十年了）写的文章「亲密接触C可变参数函数」发给他，然后开始了深入的技术交流，才有现在这篇文章，所以这篇文算是写给同事的，也分享给大家。

「亲密接触C可变参数函数」这篇文章讲的是i386架构下可变参数函数的实现原理，但是从i386到X86架构，两者的函数调用约定发生了天翻地覆的变化。X86不再完全依赖栈进行传递参数，而是通过寄存器传参数，这给运运行库实现**va_list**、**va_start**、**va_arg**和**va_end**接口带来更大挑战。

从样本程序开始

老规矩，为了更好说明可变参数的实现原理，先写个样本程序，后续所有分析都以它为蓝本，方便读者的理解，更好理解方案的本质。

```
#include <stdarg.h>
#include <stdio.h>

long sum(long num, ...)
{
        va_list ap;
        long sum = 0;

        va_start(ap, num);

        while(num--) {
                sum += va_arg(ap, long);
        }

        va_end(ap);
        return sum;
}

int main()
{
        long ret;
        ret = sum(8L, 1L, 2L, 3L, 4L, 5L, 6L, 7L, 8L);
        printf("sum(1..8) = %ld\n", ret);

        return 0;
}
```

为了减少讨论的噪音，函数参数全是long类的，即64位整型，同时没有交杂着浮点类型参数。如果读者对C语言可变参数函了解不多，可参考拙文「亲密接触C可变参数函数」，本文不再讲述C语言可变参数函数本身的定义，以及va_start、va_arg和va_end接口的语义，重点讲解X86架构下**va_start**和 **va_arg**的实现原理。

X86架构函数调用约定

回到上述程序，main函数调用sum求和函数，一共传递了9个long类型参数。在X86架构下，函数调用通过call指令实现，但函数的参数是如何传递的呢？估计有很多读者理解不深，这里简单科普一下。

64位架构下，前6个基础类型（long、int、short 、 char和pointer）参数通过寄存器传递，第7个参数开始用栈传递。通过寄存器传递可减少压栈和弹栈开销，提升性能。具体来说，前6个参数分别通过寄存器rdi，rsi，rdx，rcx，r8和r9寄存器传递，剩下的参数，依次通过调用者的rsp + 0，rsp + 8，rsp + 16，……，rsp + 8*N 这些地址空间传递。

以上述main函数调用sum为例子：

```
1    ret = sum(8L, 1L, 2L, 3L, 4L, 5L, 6L, 7L, 8L)
```

参数传递如图1所示：

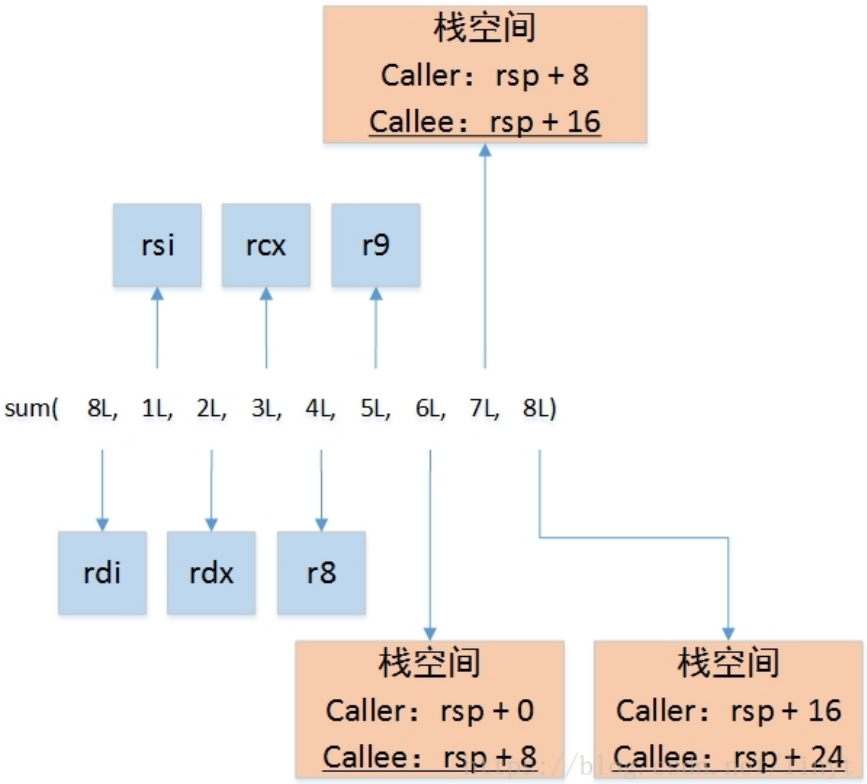


图1：main调用sum的参数传递约定

图1中通过栈传递的参数，都标识了两个地址。caller为调用者，表示main函数在在执行时到的地址，而callee则表示被调用者，为sum函数执行第一条指令时看到的地址。图2展示了进入sum时栈空间结构，可能清晰看到传递参数阶段时栈的空间结构。

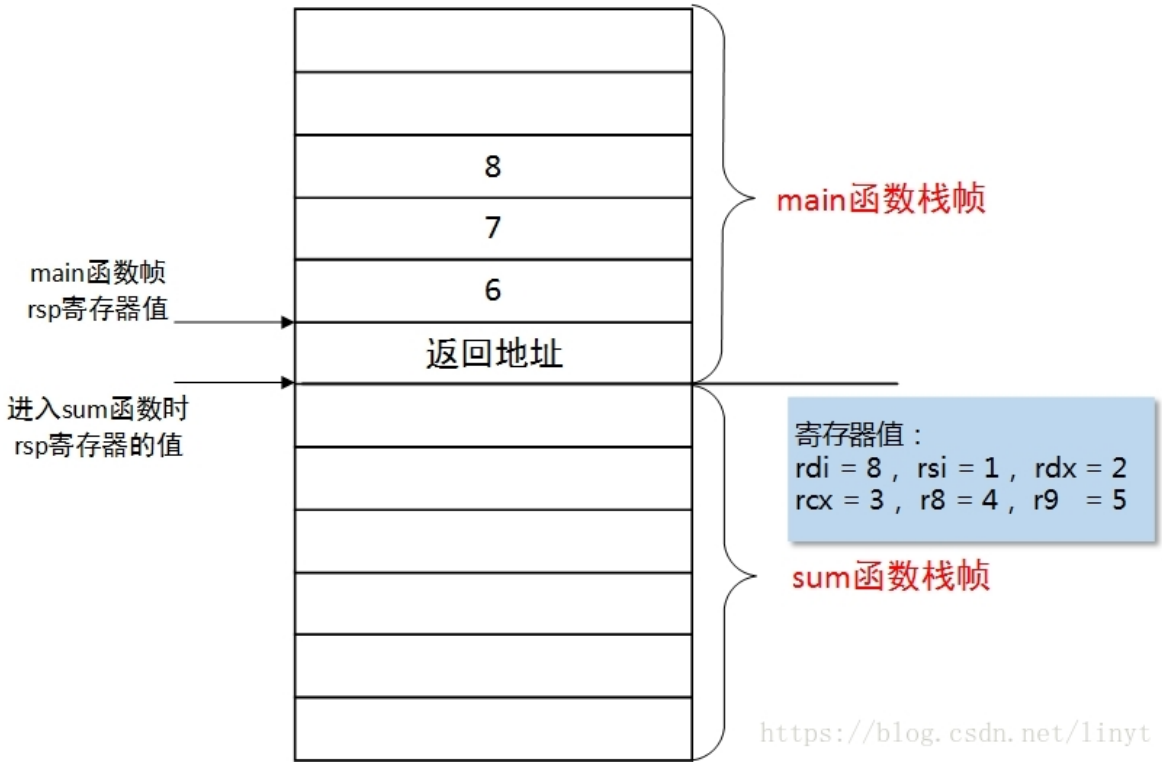


图2：进入sum函数时栈布局

main函数在执行call指令时，将返回地址压到栈上，并将rsp寄存减8字节。所以main在执行call前的rsp值，和进入sum时的rsp值刚好相差8字节，这就是为什么图1caller和callee看到参数地址刚好相差8字节。

图2可以清楚看到，传递sum函数的后3个参数，依次存储在**返回地址**顶上的高地址空间，每个参数占8字节，前6个参数通过寄存器传递。对X86调用约定感兴趣的读者，可以参考wiki x86 calling conventions - System V AMD64 ABI 词条。

初窥Linux实现方案

我们知道va_start的作用是告知哪个参数之后才是可变参数，以 sum函数为例：

```
1    va_start(ap, num);
```

这句话告诉我们，num参数之后的，就是可变参数了，即ap描述的参数列从是从第二个参数开始的。num为第一个参数，它通过rdi传递，所以第一次执行va_arg(ap, long)时，获取的是第二个参数，应该从rsi上获取，再次调用va_arg(ap，long)时应该从rdx获取……，如果通过va_arg(ap, long)获取第七个参数时，应该通过rsp + 8去获取。

怎么样，简单吧！

停……停……停……

如果稍为思考一下，发现有不可愈越的技术问题。首当其冲的是，**va_start怎么知道num是第几个参数？**这的确是个难点，因为传给**va_start**的只是个变量名。另一问题是，当调用到va_arg(ap, long)获取第7个参数时，rsp的值是否早已发生变化了，rsp + 8 是否正确，早已无法保正。

i386架构没有这个问题，因为它规定了必须所有参数都放在栈上，并且按顺序挨在一起，不必依赖num是第几个参数，反正num参数的地址值（即&num)，加上sizeof(num)，就是下个参数的首地址，跟esp寄存器没有任何关系。

而在x86通过寄存器传参标准下，这个方案失效了。于是，翻阅了Linux的stdarg.h文件定义，想看看它是怎么实现的。在ubuntu 12.04下面找到了/usr/lib/gcc/x86_64-linux-gnu/4.6/include/stdarg.h文件，却一无所获。下面是Linux glibc的实现方案：

```
#define va_start(v,l)   __builtin_va_start(v,l)
#define va_end(v)       __builtin_va_end(v)
#define va_arg(v,l)     __builtin_va_arg(v,l)
```

头文件使用了gcc提供的**内建 __builtin_va_xxx语句来实现相应的功能**，而__builtin_语句是gcc编译器在编译阶段处理的，想要搞清楚它的实现原理，必须要翻过gcc这座大山。

这就是网上所有资料，讲可变参数原理到这里，就嘎然止步了。然而这里才开始本文想跟大家讨论的原理，好戏在后头，让我们开始漫漫的长征路吧，打造第一个讲述X86可变参数函数实现原理的博文

当然，gcc代码不是我所能把握的，既然不能正向分析，那就来个逆向，一探gcc深处的秘密。

想细想一下，通常标准库语义是**ANSI C标准**来制定，Linux下的glibc只需遵守ANSI C标准实现就可以了，但为什么偏偏**va_start**，**va_arg**和**va_end**需要gcc出来搭把手呢？这里是有原因的。

原因是上面提及的两个技术问题，如果放到编译器里面，这两个问题根本算不上是问题。num参数是第几个参数，对编译器来说，这不是废话嘛。编译器看到sum函数签名时，就知道num是第一个参数，屈指一算，就知道va_start(ap, num)语句初始化ap时，标记从第二个参数开始取**可变参数**。编译器也可以在函prologue处，偷偷生成指令，将**返回地址**高地址处的栈参数区（即rsp + 8) 记录到ap变量内，无论后面rsp怎么变化，都不用操心。

好了，下面开始一段逆向之旅，请各位绑好安全带。

反编译破解gcc的密秘

我们先对代码进行编译链接：

```
$ gcc -Wall -g -O2  -o va va.c
```

同样为了减少噪音，采用-O2优化选项，在理解整个原理后，感兴趣的同学可不使用优化选项，再分析。

为防止程序编写不满足预期，先看运行结果：

```
$ ./va
sum(1..8) = 36
```

初步看来，没有什么问题。下面使用gdb对sum函数进行反编译：

```
ivan@ivan:~$ gdb -q ./va
Reading symbols from /home/ivan/va...done.
(gdb) disassemble sum
Dump of assembler code for function sum:
   0x0000000000400590 <+0>:     lea    0x8(%rsp),%rax
   0x0000000000400595 <+5>:     mov    %rsi,-0x28(%rsp)
   0x000000000040059a <+10>:    mov    %rdx,-0x20(%rsp)
   0x000000000040059f <+15>:    mov    %rcx,-0x18(%rsp)
   0x00000000004005a4 <+20>:    mov    %r8,-0x10(%rsp)
   0x00000000004005a9 <+25>:    mov    %rax,-0x40(%rsp)
   0x00000000004005ae <+30>:    lea    -0x30(%rsp),%rax
   0x00000000004005b3 <+35>:    mov    %r9,-0x8(%rsp)
   0x00000000004005b8 <+40>:    movl   $0x8,-0x48(%rsp)
   0x00000000004005c0 <+48>:    mov    %rax,-0x38(%rsp)
   0x00000000004005c5 <+53>:    xor    %eax,%eax
   0x00000000004005c7 <+55>:    test   %rdi,%rdi
   0x00000000004005ca <+58>:    je     0x40061d <sum+141>
```

```
   0x00000000004005cc <+60>:    sub    $0x1,%rdi
   0x00000000004005d0 <+64>:    mov    -0x38(%rsp),%rsi
   0x00000000004005d5 <+69>:    jmp    0x4005f9 <sum+105>
   0x00000000004005d7 <+71>:    nopw   0x0(%rax,%rax,1)
   0x00000000004005e0 <+80>:    mov    %edx,%ecx
   0x00000000004005e2 <+82>:    sub    $0x1,%rdi
   0x00000000004005e6 <+86>:    add    $0x8,%edx
   0x00000000004005e9 <+89>:    add    %rsi,%rcx
   0x00000000004005ec <+92>:    mov    %edx,-0x48(%rsp)
   0x00000000004005f0 <+96>:    add    (%rcx),%rax
   0x00000000004005f3 <+99>:    cmp    $0xffffffffffffffff,%rdi
   0x00000000004005f7 <+103>:   je     0x40061d <sum+141>
   0x00000000004005f9 <+105>:   mov    -0x48(%rsp),%edx
   0x00000000004005fd <+109>:   cmp    $0x30,%edx
   0x0000000000400600 <+112>:   jb     0x4005e0 <sum+80>
   0x0000000000400602 <+114>:   mov    -0x40(%rsp),%rcx
   0x0000000000400607 <+119>:   sub    $0x1,%rdi
   0x000000000040060b <+123>:   lea    0x8(%rcx),%rdx
   0x000000000040060f <+127>:   mov    %rdx,-0x40(%rsp)
   0x0000000000400614 <+132>:   add    (%rcx),%rax
   0x0000000000400617 <+135>:   cmp    $0xffffffffffffffff,%rdi
   0x000000000040061b <+139>:   jne    0x4005f9 <sum+105>
   0x000000000040061d <+141>:   repz retq
End of assembler dump.
(gdb)
```

到这里，一切都还顺利，接下来是烧脑时刻了，准备好了吗？

sum函数大卸八块分析

顺着sum汇编，我们将该函数分成多个阶段：
1. 先分析进入sum函数时，看看栈空间结构
2. 函数prologue阶段，如何建立栈结构的
3. va_start语句，是怎样初始化ap变量的
4. 最后分析va_arg功能是怎么样实现的

阶段1：进入sum函数阶段

执行sum函数第一条指令时，栈结构已在图2展示了，这里不再赘述。

阶段2：prologue阶段

前面提到，前6个参数放在rdi，rsi，rdx，rcx，r8和r9寄存器里，尽管编译器知道num为第一个参数，通过rdi传递，剩下的可变参数从rsi寄存器开始。但sum函数代码里，全是通过va_arg(ap, long)访问可变参数，一样是很难遍历rsi，rdx，rcx，r8和r9寄存器，必须让ap记录当前是第几个，并且通过不停地switch…case做选择，才能实现第一次调用va_arg时从rsi访问，第二次从rdx，……，再到r9。switch…case语句块，本身就是一个查找表，那么能将代码表转换成数据表，让va_arg实现代码简单一些呢，答案是可以的。

gcc是采用讨巧的办法，就是在函数prologue阶段，将6个寄存器，传递可变参数的那些寄存器，全部压到称为**参数保存区**的栈空间上，这个空间刚好位于sum函数帧的高地址处。

X86有6个寄存器传递参数，每个寄存器位宽是8字节，所以**参数保存区**是48位字。图3是执行完sum函数prologue指令后的栈结构。

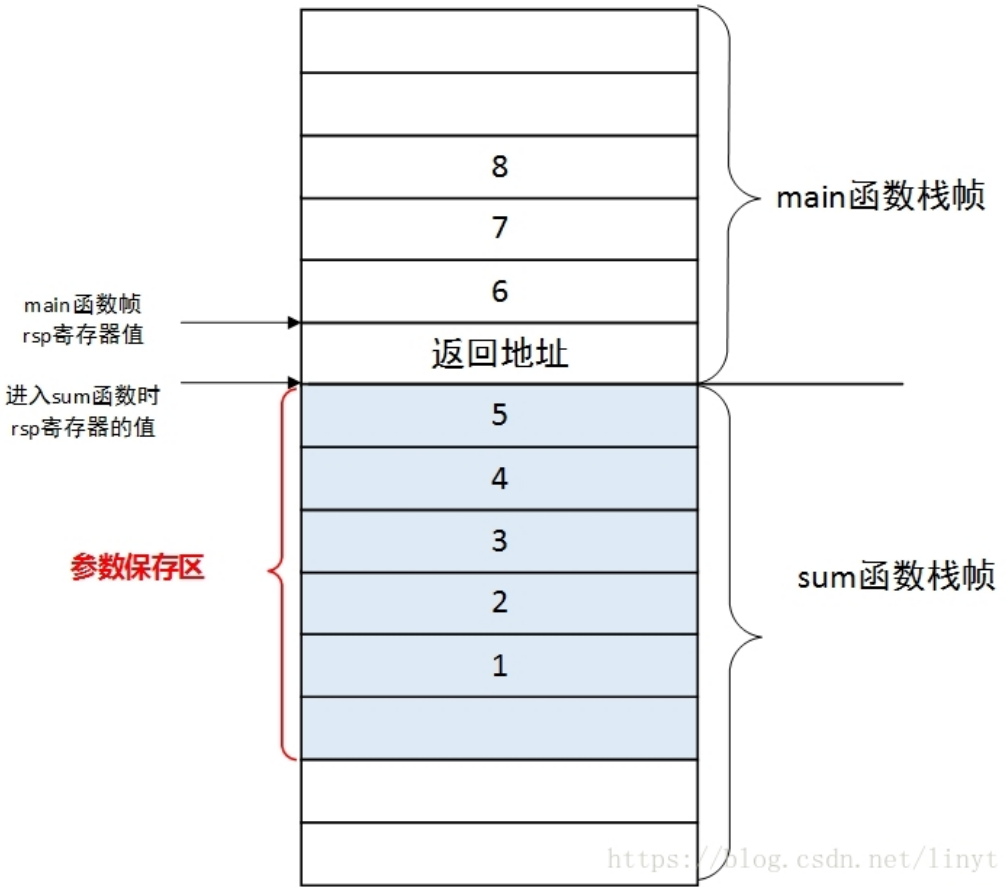


图3：sum函数prologue指令执行后栈结构

尽管gcc生成的参数保存区都是6个寄存空间，48字节，但实际上最多只可能有5个可变参数，所以总是用不完的。每个寄存器压到相应的位置，对于sum函数，第一个参数是固定参数，后5个为可变参数，所以**参数保存区**中第一个8字节留空，后面依赖保存rsi，rdx，rcx，r8和r9。

构建好**参数保存区**后，按顺序访问可变参数就变得轻而易举了。

阶段3：va_start阶段

噢，对了，va_list类型到底是什么类型呢？如果没搞清楚这个结构，上述的汇编指令根本无法入手。OK，有gdb帮助，难题轻而易举。

```
(gdb) ptype va_list
type = struct __va_list_tag {
    unsigned int gp_offset;
    unsigned int fp_offset;
    void *overflow_arg_area;
    void *reg_save_area;
} [1]
```

va_list类型可不简单，仔细对照反编译出来的指令， 发现va_list类型别的洞天，说明如下：

成员名称	描述
reg_save_area	顾名思义是**寄存器保存区**，是个指针，指向prologue指令建好的**参数保存区**
overflow_arg_area	顾名思义是**其它参数保存区**，是个指针，指向栈传递的**参数区**
gp_offset	顾名思义是**通用寄存器偏移量**，是指下个va_arg(ap, xxx)调用要获取的参数，在**参数保存区**的offset
fp_offset	顾名思义是**浮点寄存器偏移量**，本文不讨论浮点寄存器，这个变量暂时略过，有兴趣的读者可思考它是怎么用的

va_list类型语义明了，那va_start初始化ap的处理过程也跃然纸上，图4是va_start(ap, num)执行完成后的栈结构和ap变量值说明。

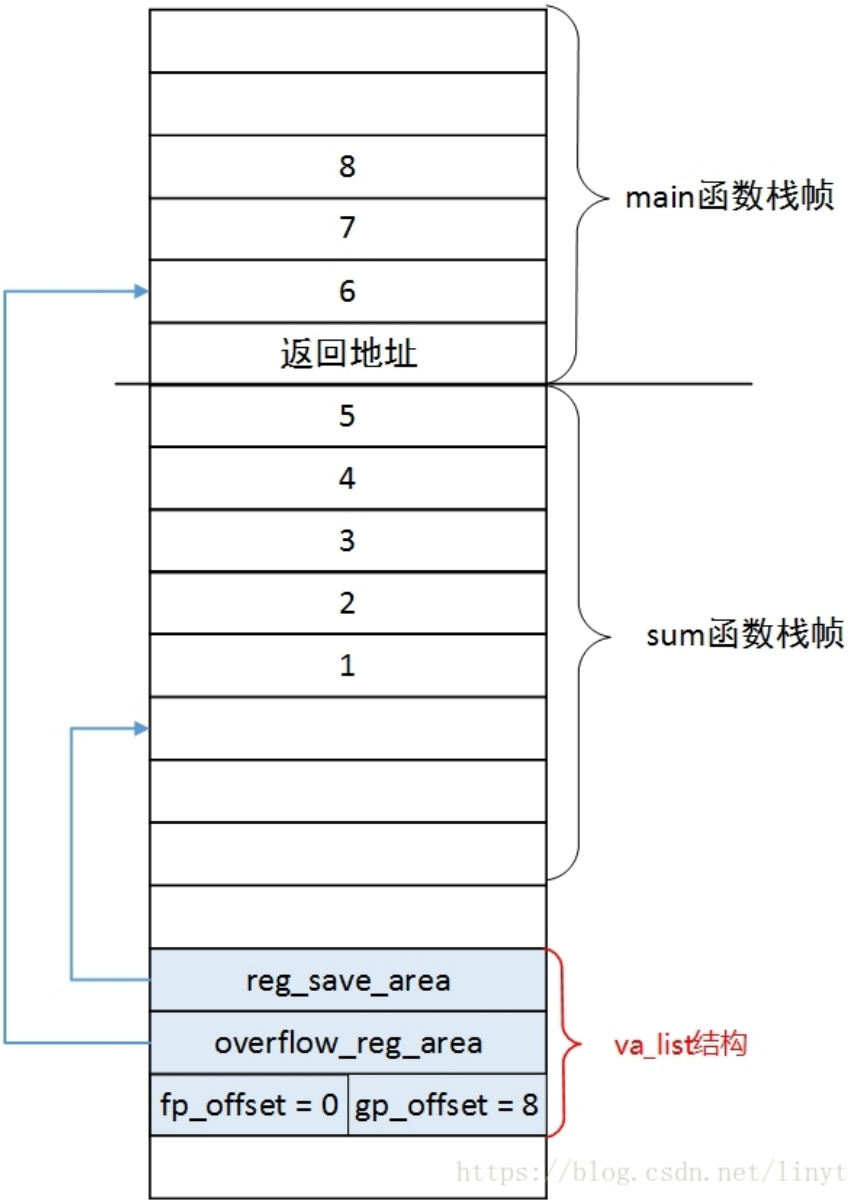


图4：va_start(ap, num)执行完成后栈结构和ap变量说明

唯一需要说明的是，gp_offset 值为8，表示下个参数存放在reg_save_area + 8这个地址上，同时也说明下一个要处理的参数是sum的第二个参数。其实这个gp_offset初始值，是编译器知道num为第一个参数，所以可变参数第二个算起，将偏移量初始化为8。

阶段4: va_arg阶段

理解完va_list结构和初始化过程，剩下的事情太简单了，每次执行va_start(ap, long)时，先读取gp_offset，然后计算gp_offset + reg_save_area地址，根据该地址就可以从**参数保存区**读取参数值，然后将gp_offset的值加8（即sizeof(long)），以便读取下个参数。

噢，慢……如果**参数保存区**读完了怎么办？不用担心，overflow_reg_area帮你顶着呢，有了它就可以访问那些通过栈传递的参数了。那怎么知道**参数保存区**读完了呢？不用急，有gp_offset当门卫，问问它就知道了。如果它的值大于等48（即6个寄存器位宽），就说明读完了，那开始从overflow_reg_area处读吧。

然而，overflow_reg_area没有对应的offset指标了，每次读完直接更新指针，指向下个参数即可。图5展示了读取第一个可变参数的执行过程，而图6展示读取栈传递参数的过程。

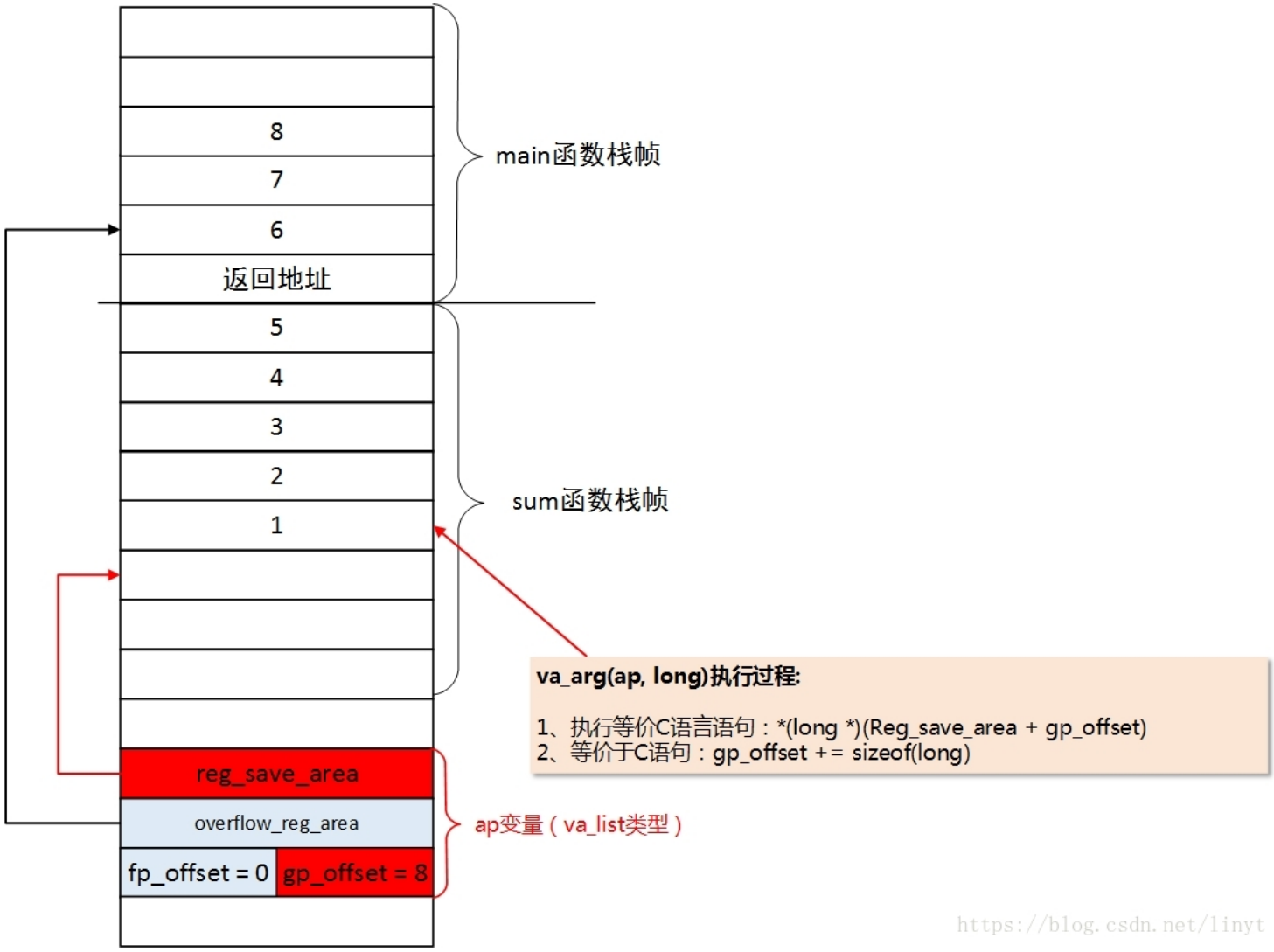


图5：va_arg 访问第一个可变参数过程

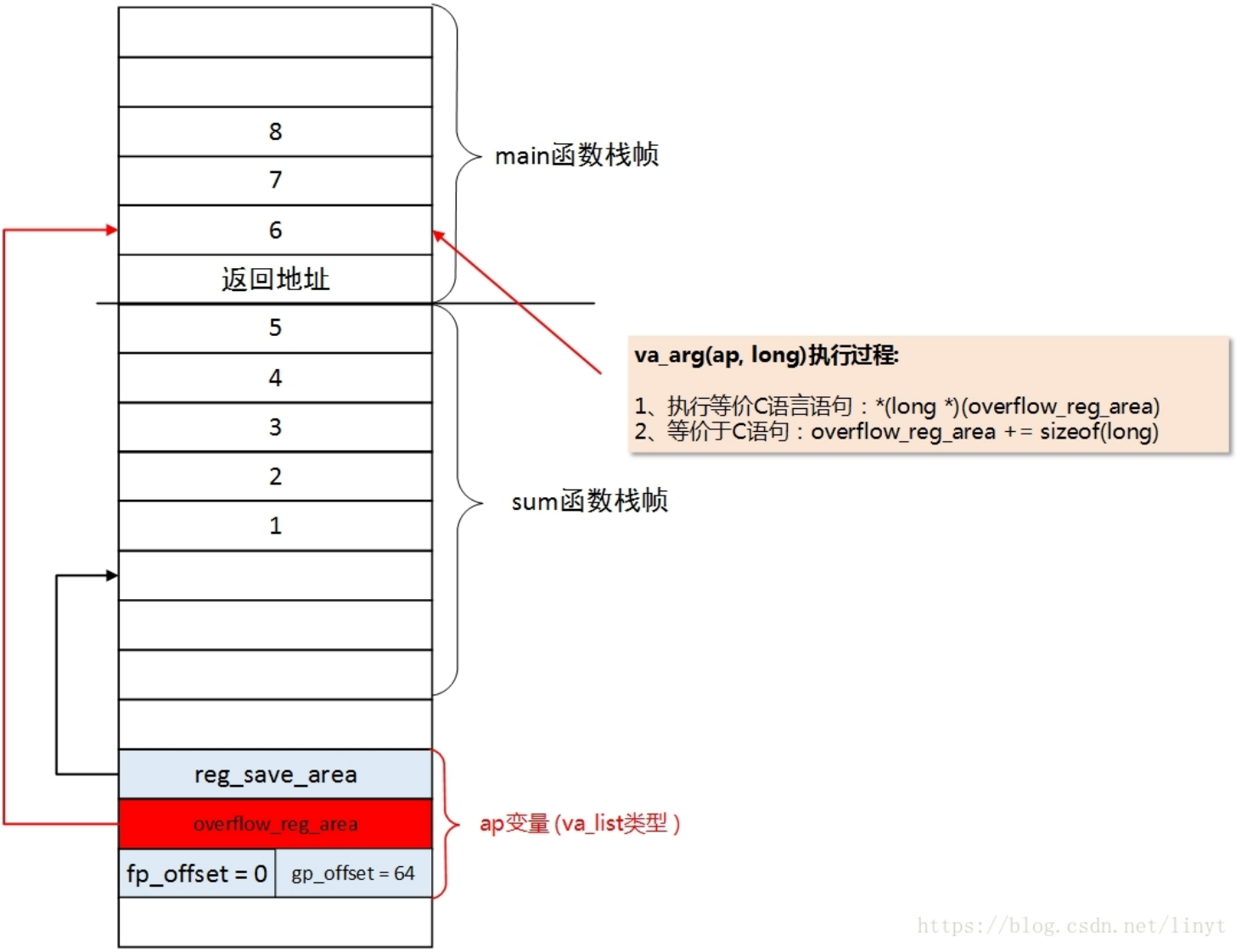


图6：va_arg访问栈传递参数过程

至此，可变参数原理关键过程都分析完了，至于va_end和va_stop原理，不作深入分析。

总结

时隔10年，再度分析C语言可变参数的实现原理，感觉绕了一圈又回到原点。实际上，自己加深了对语言本身和编译器的理解。由于没有阅读gcc源代码的经验，所以没有深入分析gcc提供的__builtin_va_start，__builtin_va_arg这几个内置功能的源代码，而通过二进制来反编译探索。**希望有编译器背影的大牛可以写个姊妹篇，以飨读者。**

那最后总结一下：

1. gcc一旦发现函数内使用va_start、va_arg这些接口，它会生成prologue指令，创建一个称为**参数保存区**的栈空间，并将第N个(N < 6）到第6个参数对应的寄存器保存在**参数保存区**
2. va_list变量结构有两个指针，分别指向参数保存区和栈传递参数区，gp_offset和fp_offset分别保存下个参数在参数保存区的偏移量，当超过48时，通过overflow_reg_area访问参数
3. va_start对va_list做初始化，va_arg根据va_list变量（本文程序是ap）状态，访问下个参数。如果gp_offset < 48，则是reg_save_area + gp_offset，否则是overflow_reg_area，当前访问完后，要更新值指向下个参数。