第3章 数据、运算与控制

在经过第 2 章的学习，读者对 C 程序的生成过程有了相对完整的认识。但是对于可执行文件本身的细节，还未构建出合理正确的认识。下面我们先来学习 C 语言程序经过 GCC 生成的可执行文件中的数据、运算和控制的具体汇编代码实现，达到阅读汇编代码时不存在大的障碍。

注意，本章仅是对 C 程序中数据运算和控制如何映射到汇编代码，而对于可执行文件的完整格式则等第 4 章到才讨论。

3.1. 数据

计算机处理任何问题都归结为数据的处理，无论是数值、图像、或者机电设备的运动在计算机内部都抽象为数据。我们这里并不对整数、浮点数等数据类型进行讨论，而是着重于数据所占空间大小、所在内存区间位置、边界对齐、字节顺序等问题，本章主将在汇编级进行观察和分析，对数据对象的讨论也是在汇编级展开。

3.1.1. 数据大小、字节序

由于我们在汇编级讨论数据，因此首先可以观察数据所占据的空间大小、数据的字节顺序。首先来回顾一下 C 语言数据类型的存储空间大小，然后在代码中查看变量所占据的空间。对于整数的数据类型，其数据表示范围和所占字节空间大小，如表 3-1 所示。浮点数则有两种，32 位的单精度浮点数和 64 位的双精度浮点数。

表 3-1 64 位系统 C 语言整数

C 数据类型	空间	最小值	最大值
char	8b	-128	127
		0x80	0x7F
usigned char	8b	0	255
		0x00	0xFF
short [int]	16b	-32768	32767
		0x80 00	0x7F FF
unsigned short [int]	16b	0	65535
		0x00 00	0xFF FF
int32t, int	32b	-2, 147, 483, 648	2, 147, 483, 647
		0x80 00 00 00	0xFF FF FF FF
uint32t, unsigned [int]	32b	0	4, 294, 967, 295
		0x00 00 00 00	0xFF FF FF FF
long [int]	64b	-9, 223, 372, 036, 854, 775, 808	9, 223, 372, 036, 854, 775, 807
		0x80 00 00 00 00 00 00 00	0x7F FF FF FF FF FF FF FF
unsigned long [int]	64b	0	18, 446, 744, 073, 709, 551, 615
		0x00 00 00 00 00 00 00 00	0x8F FF FF FF FF FF FF FF
int64t, long long [int]	64b	-9, 223, 372, 036, 854, 775, 808	9, 223, 372, 036, 854, 775, 807
		0x80 00 00 00 00 00 00 00	0x7F FF FF FF FF FF FF FF
uint64t, unsigned long long [int]	64b	0	18, 446, 744, 073, 709, 551, 615
		0x00 00 00 00 00 00 00 00	0x8F FF FF FF FF FF FF FF

下面我们用一小段 C 程序来展示数据大小和字节顺序，如代码 3-1 所示。

代码 3-1 data_size.c

```
1    char var_char1=0x11;
2    int  var_int1=0x12345678;
3    short var_short1=0x2323;
4    long long  var_64int1=0xF1AAAAAAAAAAAAF2;
5
6    long long  main()
7    {
8        long long  var_64int2;
9        var_64int2=var_64int1+var_short1+var_int1+var_char1;
10       return var_64int2;
11   }
```

编辑上述代码，然后用 gcc -O0 -g data-size.c -o data-size 命令编译成 data-size 可执行文件，并用 gdb data-size 启动调试。启动后用 p 命令逐个打印代码中的 4 个全局变量的数值和地址，最后用 x 命令查看数据所在的地址空间，其中 x/32x 命令中的 32 表示显示 32 个数值，而最后的 x 表示用 16 进制方式显示，如屏显 3-1 所示。

屏显 3-1 gdb 查看 data-size 的数据变量

```
(gdb) p/x var_char1
$1 = 0x11
(gdb) p/x &var_char1
$2 = 0x601030
(gdb) p/x var_int1
$3 = 0x12345678
(gdb) p/x &var_int1
$4 = 0x601034
(gdb) p/x var_short1
$5 = 0x2323
(gdb) p/x &var_short1
$6 = 0x601038
(gdb) p/x var64int1
No symbol "var64int1" in current context.
(gdb) p/x var_64int1
$7 = 0xf1aaaaaaaaaaaf2
(gdb) p/x &var_64int1
$8 = 0x601040
(gdb) x/32x 0x601030
0x601030 <var_char1>:      0x11 0x00 0x00 0x00 0x78 0x56 0x34 0x12
0x601038 <var_short1>:     0x23 0x23 0x00 0x00 0x00 0x00 0x00 0x00
0x601040 <var_64int1>:     0xf2 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xf1
0x601048 <completed.6344>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
(gdb)
```

对上面的 gdb 输出进行分析，可以得到数据变量的大小和相应的布局（以及对齐关系）：

- ✧ 变量 var_char1 位于 0x601030 地址的 1 个字节（后面三个字节都为 0x00），其值为 0x11；
- ✧ 变量 var_int1 位于 0x601034 地址开始的 4 个字节，数值是 0x12345678，按字节地址从低到高分别为 0x78、0x56、0x34 和 0x12，因此是小端字节序。变量 var_int1 并不是紧接着前面的变量 var_char1 存储的，而是对齐在 4 字节边界上，因此前面有 3 个值为 0x00 的填充字节；
- ✧ 变量 var_short1 位于 0x601038 地址开始的 2 字节，数字为 0x2323。后面有 6 个出于对齐目的的填充字节；

✧ 变量 var_64int1 位于 0x601040 地址开始的 8 个字节，数值为 0xf1aaaaaaaaaaaaf2，其中高位在地址高端，因此是小端字节序。

如果使用-Os 选项（优化代码的存储空间）重新编译，则 GCC 会尽量节约存储空间，而将变量存储地址做调整。此时变量数据排列如屏显 3-2 所示，实际占用空间与原来的空间相比减小了一些。

屏显 3-2 用-Os 选项编译后的数据

```
(gdb) x/24x 0x601030
0x601030 <var_64int1>:    0xf2 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xf1
0x601038 <var_short1>:    0x23 0x23 0x00 0x00 0x78 0x56 0x34 0x12
0x601040 <var_char1>:     0x11 0x00 0x00 0x00 0x00 0x00 0x00 0x00
(gdb)
```

x86-64 的数据除了在内存中还会出现在寄存器中，这些寄存器可以按照数据大小的不同来使用，另外在 C 程序的函数参数传递方面也有一些使用约定，如表 3-2 所示。注意表 3-2 中寄存器 rax 的低 32 位命名为 eax，eax 的低 16 位命名为 ax，而 ax 的高 8 位和低 8 位各自命名为 ah 和 al。其他 rbx、rcx 寄存器等也有这样的命名约定。

表 3-2 x86-64 数据寄存器

63-0	31-0	15-0	8-15	7-0	使用惯例
rax	eax	ax	ah	al	保存返回值
rbx	ebx	bx	bh	bl	被调用者保存
rcx	ecx	cx	ch	cl	第 4 个参数
rdx	edx	dx	dh	dl	第 3 个参数
rsi	esi	si	无	sil	第 2 个参数
rdi	edi	di	无	dil	第 1 个参数
rbp	ebp	bp	无	bpl	被调用者保存
rsp	esp	sp	无	spl	栈指针
r8	r8d	r8w	无	r8b	第 5 个参数
r9	r9d	r9w	无	r9b	第 6 个参数
r10	r10d	r10w	无	r10b	调用者保存
r11	r11d	r11w	无	r11b	调用者保存
r12	r12d	r12w	无	r12b	被调用者保存
r13	r13d	r13w	无	r13b	被调用者保存
r14	r14d	r14w	无	r14b	被调用者保存
r15	r15d	r15w	无	r15b	被调用者保存

3.1.2. 数组、结构体和联合体

3.1.3. 数据布局

在前面 3.1.1 查看过数据变量的大小、排列和字节序之后，我们来看看它们在进程影像中的布局。由于我们还没有学习可执行文件的格式和进程影像，因此先简单地看一下进程的虚存空间使用情况，然后就可以定位到各种不同类型（指全局变量、局部变量等）变量是如何布局在进程虚存空间的。

进程空间的使用

进程的影像将在 4.1 小节分析，这里只是通过 proc 文件系统简单地看一下进程各种属性不同的段的地址区间即可。下面以为例，通过/proc/PID/maps 和 gdb 打印的变量地址来展示相关现象。

首先我们编写代码 3-2 和代码 3-3，用 gcc -O0 data-place.c data-place-func.c -o data-place 命令编译成 data-place 可执行文件。这两个代码很简单，读者可以自行阅读。我们主要是展示全局变量、局部变量（堆栈变量、自动变量）和动态分配的内存空间在进程空间中的位置。

代码 3-2 data-place.c

```
1   #include <stdlib.h>
2   int a;
3   int b=100;
4
5   int func(int c, int d);
6
7   int main()
8   {
9       int *buf;
10      a=func(b,5);
11      buf=(int *)malloc(1024);
12      return a;
13  }
```

为了是的代码有更广泛的代表性，例子中使用了两个 C 程序，第二个程序如代码 3-3 所示，完成一些简单的算术运算。

代码 3-3 data-place-func.c

```
1   #include <stdlib.h>
2   int a;
3   int b=100;
4
5   int func(int c, int d);
6
7   int main()
8   {
9       int *buf;
10      a=func(b,5);
11      buf=(int *)malloc(1024);
12      return a;
13  }
```

我们用 gdb -static data-place 启动 data-place 程序，然后在其他终端上用 ps -a |grep data 查找到其进程号 6873，然后用 cat /proc/6873/maps 查看该进程的进程空间布局信息，如屏显 3-3 所示。我们暂时不需要对进程空间有太细致的认识，当前大致知道:

- ✧ 地址区间 0x00400000~0x00401000 是代码段（其标志 r-xp 表示可读、可执行以及私有）;
- ✧ 地址区间 0x00600000~0x00601000 是只读数据段（其标志 r—p 表示可读以及私有）;
- ✧ 地址区间 0x00601000~0x00602000 是可读可写的数据段;

  （以上三个段的内容都来源于磁盘文件/home/lqm/cs2/data-place。）
- ✧ 地址区间 0x00602000~0x00623000 是堆区;
- ✧ 地址区间 0x7ffffffde000-0x7fffffff000 是用户态堆栈。

✧ 中间还有几个动态库所映射的区间，暂时不做解释。

屏显 3-3 data-place 进程布局信息

```
[root@localhost cs2]# ps -a |grep data-place
 6873 pts/0    00:00:00 data-place
[root@localhost cs2]# cat /proc/6873/maps
00400000-00401000 r-xp 00000000 fd:00 2462491                            /home/lqm/cs2/data-place
00600000-00601000 r--p 00000000 fd:00 2462491                            /home/lqm/cs2/data-place
00601000-00602000 rw-p 00001000 fd:00 2462491                            /home/lqm/cs2/data-place
00602000-00623000 rw-p 00000000 00:00 0                                  [heap]
7ffff7a1c000-7ffff7bd2000 r-xp 00000000 fd:00 105236                     /usr/lib64/libc-2.17.so
7ffff7bd2000-7ffff7dd2000 ---p 001b6000 fd:00 105236                     /usr/lib64/libc-2.17.so
7ffff7dd2000-7ffff7dd6000 r--p 001b6000 fd:00 105236                     /usr/lib64/libc-2.17.so
7ffff7dd6000-7ffff7dd8000 rw-p 001ba000 fd:00 105236                     /usr/lib64/libc-2.17.so
7ffff7dd8000-7ffff7ddd000 rw-p 00000000 00:00 0
7ffff7ddd000-7ffff7dfd000 r-xp 00000000 fd:00 105229                     /usr/lib64/ld-2.17.so
7ffff7fe3000-7ffff7fe6000 rw-p 00000000 00:00 0
7ffff7ff9000-7ffff7ffa000 rw-p 00000000 00:00 0
7ffff7ffa000-7ffff7ffc000 r-xp 00000000 00:00 0                          [vdso]
7ffff7ffc000-7ffff7ffd000 r--p 0001f000 fd:00 105229                     /usr/lib64/ld-2.17.so
7ffff7ffd000-7ffff7ffe000 rw-p 00020000 fd:00 105229                     /usr/lib64/ld-2.17.so
7ffff7ffe000-7ffff7fff000 rw-p 00000000 00:00 0
7ffffffde000-7ffffffff000 rw-p 00000000 00:00 0                          [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
[root@localhost cs2]#
```

数据的布局

用 gdb 启动 data-place 之后，可以用 p 命令查看 data-place.c 中的全局变量 a、b 以及 data-place-func.c 中的全局变量 c、d，如屏显 3-4 开头几行所示。它们的地址 a:0x601040、b:0x601034、c:601044 和 d:601038，都位于 0x00601000~0x00602000 区间——即普通数据区。而此时的运行环境不是 buf 的作用域（No symbol "buf" in current context），这是因为 buf 是 main()函数内部的局部变量，而程序还没有开始执行到 main()函数。

在执行 start 命令之后，程序进入 main()函数，此时再检查 buf 变量及其地址，可以发现 buf 初始值为 0（即空指针），地址 0x7fffffffe0b8 位于堆栈区。这正好验证了我们说的局部变量是堆栈变量的说法，后面 3.3.3 小节会讨论函数内部的局部变量是如何函数栈帧中分配空间的。同理，此时还未进入到 func()函数，因此查看 func()内部的变量 x、y 和 k 时提示变量不可见。

在 func()入口处和 data-place.c 的 11 行设置断点，再用 c 命令继续运行到 func()入口处。此时检查发现参数已经传入：x=100、y=5，而且 x 的地址 0x7fffffffe08c、y 的地址 0x7fffffffe088 以及 k 的地址 0x7fffffffe09c，都位于堆栈区间内，而且在 main()函数的局部变量下方（地址更低端的位置）。

继续执行 c 命令，程序返回到 main()函数并且在断点处（data-place.c 的 11 行）暂停。执行 n 命令，程序将执行完 11 行的代码完成内存分配，此时在检查所分配的地址空间 buf=0x602010 位于堆区（heap，请回顾屏显 3-3）。如果所分配的空间继续增长，则堆区会扩展，如果一次分配空间超过一定大小，则会以文件映射的内存区来分配。

屏显 3-4 用 gdb 查看 data-place 的数据布局情况

```
[root@localhost cs2]# gdb -silent data-place
Reading symbols from /home/lqm/cs2/data-place...done.
(gdb) p &a
$1 = (int *) 0x601040 <a>
     63
```

```
(gdb) p &b
$2 = (int *) 0x601034 <b>
(gdb) p &c
$3 = (int *) 0x601044 <c>
(gdb) p &d
$4 = (int *) 0x601038 <d>
(gdb) p buf
No symbol "buf" in current context.
(gdb) start
Temporary breakpoint 1 at 0x400535: file data-place.c, line 10.
Starting program: /home/lqm/cs2/data-place

Temporary breakpoint 1, main () at data-place.c:10
10          a=func(b,5);
(gdb) p buf
$5 = (int *) 0x0
(gdb) p &buf
$6 = (int **) 0x7fffffffe0b8
(gdb) p x
No symbol "x" in current context.
(gdb) p y
No symbol "y" in current context.
(gdb) p k
No symbol "k" in current context.
(gdb) b func
Breakpoint 2 at 0x40056d: file data-place-func.c, line 7.
(gdb) b 11
Breakpoint 3 at 0x40054d: file data-place.c, line 11.
(gdb) c
Continuing.

Breakpoint 2, func (x=100, y=5) at data-place-func.c:7
7           c=x+d;
(gdb) p x
$7 = 100
(gdb) p y
$8 = 5
(gdb) p &x
$9 = (int *) 0x7fffffffe08c
(gdb) p &y
$10 = (int *) 0x7fffffffe088
(gdb) p &k
$11 = (int *) 0x7fffffffe09c
(gdb) c
Continuing.

Breakpoint 3, main () at data-place.c:11
11          buf=(int *)malloc(1024);
(gdb) n
12          return a;
(gdb) p buf
$12 = (int *) 0x602010
(gdb) p &buf
$13 = (int **) 0x7fffffffe0b8
(gdb)
```

内存泄漏 XXXXXXXXXXXXXXXXXXXXXXXXXx

在 C 程序中，除了动态分配的内存，其他变量的存储空间分配都是由 GCC 负责，我们认为基本不会出问题。只有用户自行分配的空间（如前面代码 3-2 例子中 11 行的 malloc()函数），就有可能出现内存泄漏和释放空指针等问题。GCC 工具 mtrace 可以帮忙发现这类问题——mtrace 为内存分配函数（malloc、realloc、memalign 和 free 等）安装 hook 钩子函数，这些 hook 函数记录内存的申请和释放的 trace 信息，并依据这些信息判定内存分配和释放管理上的问题。

环境变量 MALLOC_TRACE 用于记录 trace 信息文件的路径名，当调用 mtrace()后，该信息文件被打开且被截断为 0 长度。如果 MALLOC_TRACE 没有设置，或者设置的文件不可用或者不可写，那么将不会安装 hook 函数使得 mtrace 无法工作。也可以直接在命令行中指出 trace 信息文件名。

下面以代码 3-4 来展示未释放内存的例子，以及用**错误!未找到引用源。**展示如何使用 mtrace 工具发现这些问题。代码 3-4 的第 14 行将执行 2 次循环，每次调用 client()函数进而在第 6 行代码处执行 malloc()函数分配 100 个字节的空间，另外在第 7 行执行 calloc()分配了 16*16 字节的空间。每次调用 client()一次将进行三次分配而且都没有释放。想象一下，如果主程序是一个服务器程序，在长时间运行后因客户服务不断到达引起 client()函数不断被执行，最后这些未释放的空间将消耗掉全部的可以内存空间。不过这个例子仅分配一点点的空间，并没有引起严重问题就已经结束了。

代码 3-4 leak-demo.c

```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   int  client()
5   {
6       malloc(100);            /* Never freed--a memory leak */
7       calloc(16, 16);             /* Never freed--a memory leak */
8       return 0;
9   }
10
11  int  main(int argc, char *argv[])
12  {
13      int j;
14      for  (int j = 0; j < 2; j++)
15          client();
16
17      exit(EXIT_SUCCESS);
18  }
```

如果想通过工具自动发现这些问题，而不是人工检查或等的系统崩溃时才发现，则可以用 mtrace 工具——我们将代码 3-4 代码增加 mtrace 功能从而修改为**错误!未找到引用源。**。从代码中可以看出首先需要包含头文件 mcheck.h，然后在需要跟踪的代码前加上 mtrace()启动跟踪、在需要跟踪的代码结束处调用 muntrace()结束跟踪——毕竟跟踪会增加额外的运行时间，因此对没有问题的代码就不要开启跟踪功能。

代码 3-5 mt-demo.c

```
1   #include <mcheck.h>
2   #include <stdlib.h>
3   #include <stdio.h>
4
5   int  client()
6   {
7       malloc(100);            /* Never freed--a memory leak */
8       calloc(16, 16);             /* Never freed--a memory leak */
9       return 0;
10  }
11
12  int  main(int argc, char *argv[])
13  {
```

```
14      int j;
15
16      mtrace();
17
18     for  (int j = 0; j < 2; j++)
19          client();
20
21      muntrace();
22
23      exit(EXIT_SUCCESS);
24  }
```

https://blog.csdn.net/sunnydogzhou/article/details/6532436

3.2. 运算

在了解了数据的布局后，我们简单地查看一下 GCC 中如何实现各种运算，由于我们只考察 C 语言中的基本运算，因此转换成汇编也是比较简单和直接的。由于我们并不打算完整的讲授 X86-64 汇编语言，因此只会用样例代码展示一部分的汇编指令。

3.2.1. 数据传送

虽然数据传送算不上真正的运算，但是它也是运算中必不可少的操作，因此首先来观察数据传送。读者如果已经了解过 x86-64 汇编，即可知道数据传送指令有三类。

首先是常见的 mov 指令，将源操作数赋值给目的操作数。根据数据尺寸的不同可以在 mov 后面跟上后缀 b、w、l、q——分别对应 1 个字节的 Byte、2 个字节的 Word、4 个字节的 Double Word 和 8 字节的 Quad Word。如果目的操作数比源操作数的字长更长，则需要分别指明各自的尺寸和扩展方式（比如高位按 0 扩展还是按符号扩展）——例如 movsbw 指令表明将一个字节的源操作数按照符号扩展（即 s:sign）方式扩展到双字的目的操作数上，类似地 movzwq 指令表明将一个双字的源操作数按照 0 扩展（即 z:zero）方式扩展到双字的目的操作数上。对于上述的扩展传送指令，目的操作数必须是寄存器。

第二类指令是堆栈操作，分别是 pushq 和 popq 用于入栈和出栈操作。

第三类指令是 ctlq 用于将 32 位的%eax 按符号扩展到 64 位的%rax 中，以及 clto 用于将 64 位的%rax 扩展到 128 位的%rdx:%rax。

下面以代码 3-6 的 swap.c 为例，简单地了解一下 C 语言中的简单赋值语句如何与汇编进行映射，帮助读者建立直观认识。swap.c 的两个输入参数是变量 x、y 的指针（地址），在函数内先通过指针获得两个变量的值并保存到临时变量 t0、t1 中，然后通过写入对方的地址单元从而完成值的交换。

代码 3-6 swap.c

```
1   void swap  (long *xp, long *yp)
2   {
3       long t0 = *xp;
4       long t1 = *yp;
5       *xp = t1;
```

```
6       *yp = t0;
7   }
```

用 gcc -Og -S swap.c 命令产生出 swap.s 汇编程序，如代码 3-7 代码 3-7 swap.s 所示。GCC 在生成 x86-64 代码时，函数的前 6 个参数将按顺序使用 rdi、rsi、rdx、rcx、r8、r9（关于函数参数传递方法在 3.3.3 小节分析）。去除掉所有的伪代码后，仅剩下其中四条用灰色标注的指令。其中前两个是从内存单元将变量 x、y 的数值（分别对应“(%rdi)”和“(%rsi)”）读入到%rax（对应 t0）和%rdx（对应 t1）中，后两条指令分别将变量 x、y 的数值存储到对方的地址单元即可。

代码 3-7 swap.s

```
1       .file   "swap.c"
2       .text
3       .globl  swap
4       .type   swap, @function
5   swap:
6   .LFB0:
7       .cfi_startproc
8       movq (%rdi), %rax
9       movq (%rsi), %rdx
10      movq %rdx, (%rdi)
11      movq %rax, (%rsi)
12      ret
13      .cfi_endproc
14  .LFE0:
15      .size   swap, .-swap
16      .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
17      .section .note.GNU-stack,"",@progbits
```

其他类型的数据传送指令，请读者自行编写 C 代码进行验证和观察。

3.2.2. 算术/逻辑运算

算术逻辑运算可分为单操作数指令和双操作数指令，读者可以根据表 3-3 回顾一下以前所学的 x86-64 常用算术逻辑运算汇编指令。需要注意的是表中没有给出指令的后缀，实际上所有指令都需要指明操作数大小，例如 inc 实际上包括 incb、incw、incl 和 incq 四种指令。

表 3-3 常用算术/逻辑指令（可带后缀 b/w/l/q）

单操作数指令	作用	双操作数指令	作用
inc	+1	**add**	加法
dec	-1	**sub**	减法
neg	取负	**imul**	乘法
not	取非	**sal**	算术左移
		shl	逻辑左移
		sar	算术右移
		shr	逻辑右移
		xor	异或
		and	与
		or	或

下面用一段代码，将 C 程序与汇编代码的直观联系起来。代码 3-8 在 C 语言函数 arith()种进行了加法、乘法、按位与、整数乘的运算。

代码 3-8 arith.c

```
1   long arith(long x,long y,long z)
```

```
2   {
3       long t1=x+y;
4       long t2=z*48;
5       long t3=t1&0xFFFF;
6       long t4=t2*t3;
7       return t4;
8   }
```

用 gcc –Og –S arith.c –o arith.s 后生成汇编程序如代码 3-9 所示。此处参数 x 这里看到 t1=x+y 是通过 addq 指令完成的。z*48 并未有直接用乘法指令实现，而是利用第 9 行的 leaq 指令完成 z+2z 的计算，然后在第 10 行完成（3z）<< 4=（3*z） *16=48*z 的计算。

代码 3-9 arith.s

```
1       .file   "arithm.c"
2       .text
3       .globl  arith
4       .type   arith, @function
5   arith:
6   .LFB0:
7       .cfi_startproc
8       addq %rdi, %rsi
9       leaq (%rdx,%rdx,2), %rax
10      salq $4, %rax
11      movzwl  %si, %esi
12      imulq   %rsi, %rax
13      ret
14      .cfi_endproc
15  .LFE0:
16      .size   arith, .-arith
17      .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
18      .section .note.GNU-stack,"",@progbits
```

3.2.3. 加载有效地址

加载有效地址（Load Effective Address）指令 leaq 是将源操作数的地址传送/加载的寄存器中。例如 leaq 7(%rdx,%rcx,4),%rax 就是将源操作数地址 7+%rdx+%rcx*4 保存到%rax 中。GCC 也常将该指令用于算术运算，例如代码 3-10 中的算术表达式 t=x+4*y+12*z 转换成汇编后如代码 3-11 所示。

代码 3-10 leaq.c

```
1   long scale(long x,long y, long z)
2   {
3           long t=x+4*y+12*z;
4           return t;
5   }
```

其中 t 的运算被分解为两部分，先求 x+4*7，然后求 12*z，最后将上述两个部分和加起来。第一个部分和 x+4y 是通过代码 3-11 第 8 行的 leaq(%rdi,%rsi,4), %rcx 汇编指令来计算的，而 12*z 则是转换成 4*（3*z）并通过 leaq (%rdx,%rdx,2), %rax 和 salq $2, %rax 两条指令来完成计算的。

代码 3-11 leaq.s

```
1       .file   "leaq.c"
2       .text
3       .globl  scale
```

```
4       .type   scale, @function
5   scale:
6   .LFB0:
7       .cfi_startproc
8       leaq (%rdi,%rsi,4), %rcx
9       leaq (%rdx,%rdx,2), %rax
10      salq $2, %rax
11      addq %rcx, %rax
12      ret
13      .cfi_endproc
14  .LFE0:
15      .size   scale, .-scale
16      .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
17      .section .note.GNU-stack,"",@progbits
```

3.3. 控制

关于 C 语言的控制部分，我们这里讨论循环、分支和 switch 结构，以及函数的调用和返回过程，特别是函数调用和返回过程中的堆栈结构和相应的变化。由于控制结构和函数调用映射到到汇编时要比数据表示和运算的表示要略复杂一点，因此读者需要有点耐心来完成阅读。如果说数据和运算还比较简单，那么控制就略有一点精彩了。

由于控制流是依据特定条件的，所以在 X86-64 处理器用了 RFLAGS 寄存器设置特定标志来表示某些特定条件的发生，下面首先来看看 RFLAGS 寄存器的内容。

3.3.1. 条件跳转和 RFLAGS

代码中除了串行顺序执行的指令序列外，还需要根据不同条件选择执行不同的执行流。对这个控制流的支持是通过条件跳转指令来实现的。条件跳转指令依据标志寄存器中的标志来决定是顺序执行还是跳转到指定代码处执行。

标志寄存器 RFLAGS

表 3-4 给出了 x86 处理器的标志寄存器各位的说明，实际上我们这一章主要关注的是 CF、OF、SF、ZF 等几个有关运算结果的标志——条件跳转指令将使用这些作为判定条件。因此，读者可以先了解这几个标志的作用，而暂时无需将表中所有标志位都弄清楚，读者可以有选择地阅读下面的标志位说明。

- **运算结果标志位**

1、进位标志 CF（Carry Flag）

进位标志 CF 主要用来反映运算是否产生进位或借位。如果运算结果的最高位产生了一个进位或借位，那么其值为 1，否则其值为 0。 使用该标志位的情况有：多字（字节）数的加减运算，无符号数的大小比较运算，移位操作，字（字节）之间移位，专门改变 CF 值的指令等。

2、奇偶标志 PF（Parity Flag）

奇偶标志 PF 用于反映运算结果中“1”的个数的奇偶性。如果“1”的个数为偶数，则 PF 的值为 1，否则其值为 0。 利用 PF 可进行奇偶校验检查，或产生奇偶校验位。在数据传送过程中，为了提供传送的可靠性，如果采用奇偶校验的方法，就可使用该标志位。

3、辅助进位标志 AF（Auxiliary Carry Flag）

在发生下列情况时，辅助进位标志 AF 的值被置为 1，否则其值为 0：

(1)、在字操作时，发生低字节向高字节进位或借位时；

(2)、在字节操作时，发生低 4 位向高 4 位进位或借位时。

4、零标志 ZF（Zero Flag）

零标志 ZF 用来反映运算结果是否为 0。如果运算结果为 0，则其值为 1，否则其值为 0。在判断运算结果是否为 0 时，可使用此标志位。

5、符号标志 SF（Sign Flag）

符号标志 SF 用来反映运算结果的符号位，它与运算结果的最高位相同。运算结果为正数时，SF 的值为 0，否则其值为 1。

6、溢出标志 OF（Overflow Flag）

溢出标志 OF 用于反映有符号数加减运算所得结果是否溢出。如果运算结果超过当前运算位数所能表示的范围，则称为溢出，OF 的值被置为 1，否则，OF 的值被清为 0。

对以上 6 个运算结果标志位，在一般编程情况下，标志位 CF、ZF、SF 和 OF 的使用频率较高，而标志位 PF 和 AF 的使用频率较低。另外需要注意 “溢出”和“进位”是两个不同含义的概念，不要混淆。前者用于无符号数，后者用于无符号数。

- **状态控制标志位**

状态控制标志位是用来控制 CPU 操作的，它们要通过专门的指令才能使之发生改变。

1、跟踪标志 TF（Trap Flag）

当追踪标志 TF 被置为 1 时，CPU 进入单步执行方式，即每执行一条指令，产生一个单步中断请求。这种方式主要用于程序的调试。指令系统中没有专门的指令来改变标志位 TF 的值，但程序员可用其它办法来改变其值。

2、中断允许标志 IF（Interrupt-enable Flag）

中断允许标志 IF 是用来决定 CPU 是否响应 CPU 外部的可屏蔽中断发出的中断请求。但不管该标志为何值，CPU 都必须响应 CPU 外部的不可屏蔽中断所发出的中断请求，以及 CPU 内部产生的中断请求。具体规定如下：

(1)、当 IF=1 时，CPU 可以响应 CPU 外部的可屏蔽中断发出的中断请求；

(2)、当 IF=0 时，CPU 不响应 CPU 外部的可屏蔽中断发出的中断请求。

CPU 的指令系统中有专门的指令来改变标志位 IF 的值。

3、方向标志 DF(Direction Flag)

方向标志 DF 用来决定在串操作指令执行时有关指针寄存器发生调整的方向（数值增或减）。

- **32 位标志寄存器增加的标志位**

1、I/O 特权标志 IOPL（I/O Privilege Level）

I/O 特权标志用两位二进制位来表示，也称为 I/O 特权级字段。该字段指定了要求执行 I/O 指令的特权级。如果当前的特权级别在数值上小于等于 IOPL 的值，那么，该 I/O 指令可执行，否则将发生一个保护异常。

2、嵌套任务标志 NT（Nested Task）

嵌套任务标志 NT 用来控制中断返回指令 IRET 的执行。具体规定如下：

(1)、当 NT=0，用堆栈中保存的值恢复 EFLAGS、CS 和 EIP，执行常规的中断返回操作；

(2)、当 NT=1，通过任务转换实现中断返回。

3、重启动标志 RF（Restart Flag）

重启动标志 RF 用来控制是否接受调试故障。规定：RF=0 时，表示“接受”调试故障，否则拒绝之。在成功执行完一条指令后，处理机把 RF 置为 0，当接受到一个非调试故障时，处理机就把它置为 1。

4、虚拟 8086 方式标志 VM(Virtual 8086 Mode)

如果该标志的值为 1，则表示处理机处于虚拟的 8086 方式下的工作状态，否则，处理机处于一般保护方式下的工作状态。

■ 64 位标志

X86-64 位处理器的 RFLAGS 仅仅是将 x86-32 的 EFLAGS 扩展到 64 位，但是并未增加任何实质性可用的标志。

表 3-4 x86 FLAGS 寄存器

bit	Mask	名称	说明	类别
			FLAGS	
0	0x0001	**CF**	进位	Status
1	0x0002		Reserved, always 1 in EFLAGS	
2	0x0004	**PF**	奇偶位	Status
3	0x0008		Reserved	
4	0x0010	**AF**	辅助进位	Status
5	0x0020		保留	
6	0x0040	**ZF**	零标志	Status
7	0x0080	**SF**	符号	Status
8	0x0100	**TF**	Trap flag (single step)	Control
9	0x0200	**IF**	中断允许位	Control
10	0x0400	**DF**	方向	Control
11	0x0800	**OF**	溢出	Status
12-13	0x3000	**IOPL**	I/O privilege level (286+ only), always 1 on 8086 and 186	System
14	0x4000	**NT**	Nested task flag (286+ only), always 1 on 8086 and 186	System
15	0x8000		Reserved, always 1 on 8086 and 186, always 0 on later models	
			EFLAGS	
16	0x0001 0000	**RF**	Resume flag (386+ only)	System
17	0x0002 0000	**VM**	Virtual 8086 mode flag (386+ only)	System
18	0x0004 0000	**AC**	Alignment check (486SX+ only)	System
19	0x0008 0000	**VIF**	Virtual interrupt flag (Pentium+)	System
20	0x0010 0000	**VIP**	Virtual interrupt pending (Pentium+)	System
21	0x0020 0000	**ID**	Able to use CPUID instruction (Pentium+)	System

22-31	0xFFC0 0000		保留
RFLAGS			
32-63	0xFFFF FFFF 0000 0000		保留

影响标志寄存器

标志寄存器会随着运算的进行而发生改变，也就是说表 3-3 中的所有操作都有可能会影响 RFLAGS 中的某一个或某几个标志。对于其中的逻辑运算，CF 和 OF 都是置为 0；移位操作仅为标志是刚被移出的那个位，溢出标志置为 0；INC 和 DEC 指令设置 OF 和 ZF 标志，但不影响 CF 标志。操作数据传送类指令不会影响标志寄存器，leaq 也不影响标志寄存器。

除了上面已经观察过的指令外，还有两类比较指令 cmpb/cmpw/cmpl/cmpq 以及 testb/testw/testl/testq 会影响标志寄存器。它们都是双操作数指令，仅完成比较或测试而不修改任何这两个操作数。其中 cmpX 指令类似于 subX 指令，但是完成减法操作并设置标志寄存器后，并不将结果保存到目的操作数。同样，testX 指令类似于 andX 指令，也是只影响标志寄存器而不会将结果保存到目的操作数中。

由于 cmpX 指令我们会在 3.3.2 小节反复用到，这里我们仅观察一下 testX 指令。我们参照前面的汇编代码并进行改写形成代码 3-12 testq.s，主要关注第 8 行的 testq 指令。

代码 3-12 testq.s

```
1       .file   "testq.s"
2       .text
3       .globl  testq
4       .type   testq, @function
5   testq:
6   .LFB0:
7       .cfi_startproc
8       testq   %rdi, %rsi
9       movq    %rsi,%rax
10      ret
11      .cfi_endproc
12  .LFE0:
13      .size   testq, .-testq
14      .globl  main
15      .type   main, @function
16  main:
17  .LFB1:
18      .cfi_startproc
19      movabsq $281474976710503, %rsi
20      movl $174, %edi
21      call testq
22      rep ret
23      .cfi_endproc
24  .LFE1:
25      .size   main, .-main
26      .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
27      .section .note.GNU-stack,"",@progbits
```

用 gcc -Og -g testq.s -o test 生成 testq 可执行文件，然后用 gdb -silent testq 跟踪执行，如屏显 3-5 所示。先用 l1 命令显示了汇编代码第 1 行到第 10 行，确定第 8 行是 testq 命令，然后用 b 8 将断点设置在该指令出。执行 run 命令后，程序执行停止在 testq 指令处（该指令还未执行），此时可以用 p 命令检查两个操作数的数值，正如汇编中的数值。检查标志位寄存器可

知当前 PF、ZF 和 IF 置位。用 n 执行 testq 指令后，再检查发现两个源操作数并未有任何改变，但是标志位却因为 test 操作而将 CF 和 ZF 清零了——因为$0xffffffffff67&0xae=0x26=0x0010 0110 有奇数个 1 而且不为 0。

屏显 3-5 用 gdb 观察 testq.s 中的 test 指令影响 RFLAGS

```
[root@localhost cs2]# gdb -silent testq
Reading symbols from /home/lqm/cs2/testq...done.
(gdb) l 1
1         .file"testq.s"
2         .text
3         .globl    testq
4         .typetestq, @function
5     testq:
6     .LFB0:
7         .cfi_startproc
8         testq%rdi, %rsi
9         movq    %rsi,%rax
10        ret
(gdb) b 8
Breakpoint 1 at 0x4004ed: file testq.s, line 8.
(gdb) run
Starting program: /home/lqm/cs2/testq

Breakpoint 1, testq () at testq.s:8
8         testq%rdi, %rsi
(gdb) p/x $rdi
$1 = 0xae
(gdb) p/x $rsi
$2 = 0xffffffffff67
(gdb) p $eflags
$4 = [ PF ZF IF ]
(gdb) p/x $eflags
$5 = 0x246
(gdb) n
9         movq    %rsi,%rax
(gdb) p/x $rsi
$6 = 0xffffffffff67
(gdb) p/x $rdi
$7 = 0xae
(gdb) p $eflags
$8 = [ IF ]
(gdb)
```

使用标志寄存器

当前面的操作或运算完成后，后面的代码可能会需要检查标志寄存器从而判定后续执行什么操作。我们这里讨论的访问条件码并不是直接修改标志寄存器，而是读取和使用标志寄存器中的标志位。

- **访问条件码**

Set 指令等

- **跳转/条件跳转**

PC 相对寻址概念在这里讨论

- **条件赋值语句**

3.3.2. C 语言控制语句

GCC 对各种 C 语言的控制结构采用模板的方式将它们翻译汇编代码，因此相应的汇编代码有固定的“框架”部分和可变的部分。依靠其中的固定的“框架”，我们有可能可以从汇编代码尝试反向恢复成 C 语言代码（即所谓的逆向工程）。

需要注意，编译器优化工作可能会将一些没有实质作用的代码直接忽略掉。因此读者按照后面例子自行编写验证性代码时，注意不要太过简略，例如在函数中循环体内的语句所产生的变量和函数返回值如果没有任何关系，那么整个循环体可能被直接忽略掉，从而看不到你所希望的代码。

if-else

对于 C 语言中的 if-else 条件分支语句，GCC 使用模板来完成编译的变换。C 语言中的条件分支语句在经过语法分析后在 GCC 内部表示为代码 3-13 形式的中间表示，其中有固定的语法结构部分 if 和 else，以及可变语句序列的三个部分：分支条件 test-expr、成功分支 then-statement 和不成功分支 else-statement。需要注意的是上述三个可变部分都可以是多条语句的组合。

代码 3-13 if-else 的语法结构

```
1    if (test-expr)
2        then-statement
3    else
4        else-statement
```

由于代码 3-13 的语义无法直接对接 x86-64 汇编，因此必须做一个目标模板，使得目标模板等价于原来的 if-else 语句——但是模板里面每一个操作都可以由汇编指令的语义所支持。GCC 采用如代码 3-14 所示的模板。其中 if(!t) goto false 语句可以用 cmp 比较指令和 je/jne/js/jns/jg/jge/jl/jle/ja/jae/jb/jbe 条件跳转指令的组合来实现，此时在模板中也不再需要对 else 语法结构进行变换了。

代码 3-14 if-else 编译时采用的模板

```
1        t = test-expr;
2        if (!t)
3           goto false;
4        then-statement
5        goto done;
6    false:
7        else-statement
8    done:
```

下面用代码 3-15 所示的一个简单程序来观察 if-else 语言的编译过程。代码中就只有一个条件分支语句，使用 gcc –Og –S if-else.c 进行编译，产生出的汇编如代码 3-16 所示。

代码 3-15 if-else.c

```
1    long ifelse(long x, long y)
2    {
3        if (x<y)
4            x=y+3;
5        else
```

```
6               y=10*x;
7       return x+y;
8   }
```

从代码 3-16 我们可以还原出模板中的各个组成部分，例如 L2 相当于模板中的“false”标号。因此第 8 行的 cmpq %rsi,%rdi 指令结合第 9 行的 jge .L2 指令，一起完成了 if (!t) goto false 的 C 语句功能。第 15 行的 jmp .L3 而 L3 相当于模板中的“done”标号，因此在成功分支结束处有 jmp .L3（第 11 行）。

需要注意的是，C 代码中的 y=10*x，被转换成了一条 leaq 指令和一个 addq 指令：前面完成 x+4*x=5x 的计算，后一条指令完成 5x+5x 的计算。

如果成功分支有更复杂的代码段，则用他们取代第 10 行的代码；同理，如果有更复杂的不成功分支，则用这些指令需要列取代现有的第 13、14 行。

代码 3-16 if-else.s

```
1       .file   "if-else.c"
2       .text
3       .globl  ifelse
4       .type   ifelse, @function
5   ifelse:
6   .LFB0:
7       .cfi_startproc
8       cmpq %rsi, %rdi
9       jge .L2
10      leaq 3(%rsi), %rdi
11      jmp .L3
12  .L2:
13      leaq (%rdi,%rdi,4), %rsi
14      addq %rsi, %rsi
15  .L3:
16      leaq (%rdi,%rsi), %rax
17      ret
18      .cfi_endproc
19  .LFE0:
20      .size   ifelse, .-ifelse
21      .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
22      .section .note.GNU-stack,"",@progbits
```

因此读者在汇编中发现类似的结构，可以反向猜测原来的 C 程序是一个 if-else 分支结构，然后再进一步验证并完成逆向推导出具体的 C 语句。另外也必须认识到，模板并非是唯一的，不同编译器可能回采用不同的模板，即便是 GCC 也会因为版本不同或优化级别不同而采用不同的模板。

我们给增加一个 main()函数，使得它可以编译生成可执行文件，然后尝试用 gdb 观察其运行。

代码 3-17 if-else-main.c

```
1   long ifelse(long x, long y)
2       {
3       if (x<y)
4           x=y+3;
5       else
6           y=10*x;
7       return x+y;
8   }
9
```

```
10  long main()
11  {
12      long k;
13      k=ifelse(3,5);
14      return k;
15  }
```

用 gcc -Og -S if-else-main 命令产生代码 3-18 所示的汇编程序代码。

代码 3-18 if-else-main.s

```
1       .file   "if-else.c"
2       .text
3       .globl  ifelse
4       .type   ifelse, @function
5   ifelse:
6   .LFB0:
7       .cfi_startproc
8       cmpq %rsi, %rdi
9       jge .L2
10      leaq 3(%rsi), %rdi
11      jmp .L3
12  .L2:
13      leaq (%rdi,%rdi,4), %rsi
14      addq %rsi, %rsi
15  .L3:
16      leaq (%rdi,%rsi), %rax
17      ret
18      .cfi_endproc
19  .LFE0:
20      .size   ifelse, .-ifelse
21      .globl  main
22      .type   main, @function
23  main:
24  .LFB1:
25      .cfi_startproc
26      movl $5, %esi
27      movl $3, %edi
28      call ifelse
29      rep ret
30      .cfi_endproc
31  .LFE1:
32      .size   main, .-main
33      .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
34      .section .note.GNU-stack,"",@progbits
```

为了跟踪查看该程序的执行，我们通过 gcc -Og if-else.c -o if-else 生成可执行文件（对应的机器码如屏显 3-7 所示），让后用 gdb -silent if-else 启动调试，如屏显 3-6 所示。我们将断点设置在 if(x<y)的语句上启动运行，在断点位置限制性 disas（对应完整命令是 disassemble），显示出当前函数的反汇编代码，以及用“=>”标记出下一条要执行的指令（此时是 cmp %rsi,%rdi，对应断点 if(x<y) 位置）。然后用 p 命令检查参数 x（%rdi）为 3、y（%rsi）为 5、标志寄存器含有[PF ZF IF]置位。用 si 执行 cmp 指令之后，在查看标志寄存器为[CF AF SF IF]，比原来多个一个 CF 标志，说明 x-y 产生了借位，因此 jge 的跳转并不发生。因此，执行成功分支——继续执行后面的指令 x=y+3。

屏显 3-6 gdb 调试 if-else-main

```
[root@localhost cs2]# gdb if-else-main -silent
```

```
Reading symbols from /home/lqm/cs2/if-else...done.
(gdb) b 4
Breakpoint 1 at 0x4004ed: file if-else.c, line 4.
(gdb) run
Starting program: /home/lqm/cs2/if-else

Breakpoint 1, ifelse (x=x@entry=3, y=y@entry=5) at if-else.c:4
4         if (x<y)
(gdb) disas
Dump of assembler code for function ifelse:
=> 0x00000000004004ed <+0>:     cmp    %rsi,%rdi
   0x00000000004004f0 <+3>:     jge    0x4004f8 <ifelse+11>
   0x00000000004004f2 <+5>:     lea    0x3(%rsi),%rdi
   0x00000000004004f6 <+9>:     jmp    0x4004ff <ifelse+18>
   0x00000000004004f8 <+11>:    lea    (%rdi,%rdi,4),%rsi
   0x00000000004004fc <+15>:    add    %rsi,%rsi
   0x00000000004004ff <+18>:    lea    (%rdi,%rsi,1),%rax
   0x0000000000400503 <+22>:    retq
End of assembler dump.
(gdb) p $rsi
$1 = 5
(gdb) p $rdi
$2 = 3
(gdb) p $eflags
$3 = [ PF ZF IF ]
(gdb) si
0x00000000004004f0   4         if (x<y)
(gdb) p $eflags
$4 = [ CF AF SF IF ]
(gdb) si
5               x=y+3;
(gdb) si
0x00000000004004f6   5               x=y+3;
(gdb)
```

do-while

C 语言提供了多种循环结构，包括 do-while、while 和 for，并没有汇编指令可以直接支持上述语义。因此跟前面类似，GCC 分析这些循环结构的语法，将它们用模板映射到汇编代码序列。C 语言中的 do-while 语句在经过语法分析后在 GCC 内部表示为代码 3-19 形式的中间表示，其中有固定的语法结构部分 do 和 while，以及可变语句序列的两个部分：循环体 body-statement 和循环条件 test-expr。需要注意的是上述两个可变部分都可以是多条语句的组合。

代码 3-19 do-while 的语法结构

```
1    do
2        body-statement
3    while (test-expr);
```

GCC 采用如代码 3-20 所示的模板，只有条件 t=test-expr 结过为真，才会通过 goto 跳转到 loop 循环开始处。这些语句的语义很容易通过汇编指令来支持。

代码 3-20 do-while 编译时采用的模板

```
1    loop:
2        body-statement
3        t = test-expr;
4        if (t)
5            goto loop;
```

代码 3-21 do-while.c

```
1    long do_while(long n,long k)
2    {
3        do{
```

```
4           k=3*k;
5           n--;
6       }while (n>1);

7       return n+k;
8   }
```

用 gcc -Og -S do-while.c 命令产生代码 3-22 所示的汇编程序代码。

代码 3-22 do-while.s

```
1       .file   "do-while.c"
2       .text
3       .globl  do_while
4       .type   do_while, @function
5   do_while:
6   .LFB0:
7       .cfi_startproc
8   .L2:
9       leaq (%rsi,%rsi,2), %rsi
10      subq $1, %rdi
11      cmpq $1, %rdi
12      jg  .L2
13      leaq (%rdi,%rsi), %rax
14      ret
15      .cfi_endproc
16  .LFE0:
17      .size   do_while, .-do_while
18      .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
19      .section .note.GNU-stack,"",@progbits
```

关于 do-while 的执行观察，请读者仿照 if-else 的 gdb 调试过程，编写 main()函数部分代码，然后自行跟踪器运行状态，并在关键的分支跳转点查看器标志位。

while

while 语句本质上和 do-while 相似，只是 while 甚至可能不执行循环体，而 do-while 至少执行循环体一次。对于代码 3-23 的 while 语法结构，可以首先用代码 3-24 所示的等效语法结构，然后可以使用代码 3-25 所示的。

代码 3-23 while 语法结构

```
1   while (test-expr)
2       body-statement
```

为了用 do-while 来支持 while 语句——因为 while 有不执行循环体的可能性，因此首先要进行一次条件判定，如果不成立则不进入循环体，后面再用一个 do-while 结构完成需要的循环迭代过程，如代码 3-24 所示。

代码 3-24 while 等效的语法结构

```
1   if (!test-expr)
2           goto done;
3       do
4           body-statement
5       while (test-expr);
6   done:
```

然后对为了用 do-while 来支持 while 语句——因为 while 有不执行循环体的可能性，因此首先要进行一次条件判定，如果不成立则不进入循环体，后面再用一个 do-while 结构完成需要的循环迭代过程，如代码 3-24 所示。

代码 3-24 再套用 do-while 的模板，可以得到 while 语句的编译模板如代码 3-25 所示。

代码 3-25 while 编译时可使用的模板

```
1    t = test-expr;
2        if (!t)
3            goto done;
4    loop:
5        body-statement
6        t = test-expr;
7        if (t)
8            goto loop;
9    done:
```

还有另外一种转换模板，称为跳转到中间（Jump to middle）的方式，如代码 3-26 所示。

代码 3-26 while 编译时可使用的模板（跳转到中间）

```
1    goto test;
2    loop:
3        body-statement
4    test:
5        t=test-expr
6        if(t)
7            goto done;
```

我们用代码 3-27 来检验一下 GCC 使用的是哪种模板，该代码非常简单，就是 1~n 的求和。

代码 3-27 while.c

```
1    long my_while(long n)
2    {
3        long a=1;
4        while (n>0){
5            a+=n;
6            n=n-1;
7        }
8        return a;
9    }
```

用 gcc –Og –S while.c 产生出 while.s 如代码 3-28 所示。从中我们可以很容易找出 while 结构位与第 9 行开始，而且 L2 对应于循环条件判定的代码入口，而 L3 则是循环体的入口，第 15 行是循环所需要的前向跳转——以上特征符合代码 3-26 的模板，也就是说此时 GCC 使用了该模板进行编译。

代码 3-28 while.s

```
1        .file    "while.c"
2        .text
3        .globl   my_while
4        .type    my_while, @function
5    my_while:
6    .LFB0:
7        .cfi_startproc
8        movl $1, %eax
9        jmp  .L2
10   .L3:
11       addq %rdi, %rax
12       subq $1, %rdi
13   .L2:
14       cmpq $1, %rdi
15       jg   .L3
16       rep ret
```

```
17      .cfi_endproc
18  .LFE0:
19      .size    my_while, .-my_while
20      .ident   "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
21      .section .note.GNU-stack,"",@progbits
```

关于 while 结构的执行观察，请读者仿照前面的 gdb 调试过程，编写 main()函数部分代码，然后自行跟踪器运行状态，并在关键的分支跳转点查看器标志位即可。

for

与前面的 do-while 或 while 循环虽然也是类似的，但是 for 循环的初始条件、结束条件和循环变量修改三个方面有自己的特定结构，因此也需要在 do-while 基础上做一些扩展。由于 for 的语义和 do-while 差别略大一些，因此模板也略复杂一点。

代码 3-29 for 语法结构

```
1   for (init-expr; test-expr; update-expr)
2       body-statement
```

首先将 for 的初始化代码抽取出来，将循环体语句和循环变量更新语句和冰岛循环体中，形成代码 3-30 所示的 while 语法结构表述。

代码 3-30 for 语法结构用 while 表述

```
1   init-expr;
2   while (test-expr) {
3       body-statement
4       update-expr;
5   }
```

然后再使用代码 3-26 的模板，将上述表述转换为代码 3-31，此时就可以映射到汇编指令了。

代码 3-31 for 编译所用的模板

```
1       init-expr;
2       goto test;
3   loop:
4       body-statement
5       update-expr;
6   test:
7       t=test-expr
8       if(t)
9   goto done;
```

下面以代码 3-32 为例，观察一下 for 循环具体变换后的汇编指令如何。

代码 3-32 for.c

```
1   long my_for(long n)
2   {
3       long i=10;
4       long result=1;
5       for(i=2;i<=n;i++)
6           result*=i;
7       return result;
8   }
```

通过 gcc –Og –S for.c 命令，产生出相应的汇编程序如代码 3-33 所示。从中可以看出，此例子中 GCC 使用了跳转到中间的模板来实现循环的。其中 for 的循环初始化语句 i=2 映射为 movl $2, %edx 汇编指令；循环变量的调整 i++映射为 addq $1,%rdx。

代码 3-33 for.s

```
1       .file   "for.c"
2       .text
3       .globl  my_for
4       .type   my_for, @function
5   my_for:
6   .LFB0:
7       .cfi_startproc
8       movl $1, %eax
9       movl $2, %edx
10      jmp .L2
11  .L3:
12      imulq   %rdx, %rax
13      addq $1, %rdx
14  .L2:
15      cmpq %rdi, %rdx
16      jle .L3
17      rep ret
18      .cfi_endproc
19  .LFE0:
20      .size   my_for, .-my_for
21      .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
22      .section .note.GNU-stack,"",@progbits
```

关于 for 结构的执行观察，请读者仿照前面的 gdb 调试过程，编写 main()函数部分代码，然后自行跟踪器运行状态，并在关键的分支跳转点查看器标志位即可。

switch-case

多重分支 swith-case 语句本质上是 if-else 语句，可以看成是 if-else 的组合扩展，因此其实现方式在效率上会有所考虑——对于不同的 case 编号（整数索引值）情况对应了不同的转换模板。当 case 数量较多，而且所用的索引值比较密集时，GCC 采用跳转表来实现多重分支，否则用多个 if-else 的组合来实现多重分支。

下面用代码 3-34 所示的两码来展示多种分支的编译模板，其中使用了最普通的编号为 10 的 case+break、编号为 12 的 Fall through 的 case、编号为 14 和 15 的相同分支以及 default 等类型的分支。

代码 3-34 switch.c

```
1   long my_switch(long x,long n)
2   {
3       long val=x;
4       switch(n){
5           case 10:
6               val+=12;
7               break;
8           case 12:
9               val*=3;
10
11          case 13:
12              val+=1;
13              break;
14          case 14:
15          case 15:
16              val=8;
17              break;
18          default:
19              val=100;
```

```
20              }
21          val=val+x*20;
22          return val;
23      }
```

将代码 3-34 用 gcc -Og -S switch.c 编译成 switch.s，如代码 3-35 所示。我们将逐个检查各种类型的分支是如何映射为汇编结构的。首先编号 10~15 被转换成 0~5，使用 subq $10, $4si 指令完成。然后用 cmpq $5,%rsi 指令检查编号是否大于 5，如果大于 5 则对应于 default 分支——ja .L8 指令跳转到 default 分支。然后就是利用一个跳转表——起点在.L4，偏移量为编号*指针大小=%rsi*8，使用指令 jmp *.L4(,%rsi,8)指令实现查表和跳转功能。15~21 行是跳转表，其中 17 行对应于编号为 11 的 case——跳转到 default 分支。23 行~39 行对应个中分支指令，每个分支又根据是否 fall through 略有不同，如果带有 break 则直接用 jmp .L2 跳出到 switch-case 的出口处，否则继续往下运行到另一个分支的指令序列中。40~42 行是计算 x*20，43 行计算 val=val+x*20。

代码 3-35 switch.s

```
1       .file   "switch.c"
2       .text
3       .globl  my_switch
4       .type   my_switch, @function
5   my_switch:
6   .LFB0:
7       .cfi_startproc
8       subq $10, %rsi
9       cmpq $5, %rsi
10      ja  .L8
11      jmp *.L4(,%rsi,8)
12      .section .rodata
13      .align 8
14      .align 4
15  .L4:
16      .quad   .L3
17      .quad   .L8
18      .quad   .L5
19      .quad   .L9
20      .quad   .L7
21      .quad   .L7
22      .text
23  .L7:
24      movl $8, %edx
25      jmp .L2
26  .L3:
27      leaq 12(%rdi), %rdx
28      .p2align 4,,2
29      jmp .L2
30  .L5:
31      leaq (%rdi,%rdi,2), %rdx
32      jmp .L6
33  .L9:
34      movq %rdi, %rdx
35  .L6:
36      addq $1, %rdx
37      jmp .L2
38  .L8:
39      movl $100, %edx
```

```
40  .L2:
41      leaq (%rdi,%rdi,4), %rax
42      salq $2, %rax
43      addq %rdx, %rax
44      ret
45      .cfi_endproc
46  .LFE0:
47      .size    my_switch, .-my_switch
48      .ident   "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
49      .section .note.GNU-stack,"",@progbits
```

当 switch-case 语句中的数字索引编号较少或者比较分散，则 GCC 可能不在使用跳转表，若是直接利用跳转指令来实现。代码 3-36 中只有 10 和 19 两个有效的编号，其他编号则对应于 default 分支。

代码 3-36 switch-simp.c

```
1   long my_switch(long x,long n)
2   {
3       long val=x;
4       switch(n){
5           case 10:
6               val+=12;
7               break;
8           case 19:
9               val*=3;
10              break;
11          default:
12              val=100;
13          }
14      val=val+x*20;
15      return val;
16  }
```

用命令 gcc -Og -S switch-simp.c 编译生成 switch-simp.c，如代码 3-37 所示。此时已经没有跳转表了，而是直接和数值 10 和 19 进行比较（第 8 行和第 10 行），进而跳转到.L3 和.L4 对应的分支代码，如果不是上述两个数值则直接跳转到.L6 的缺省 default 分支。从.L2 标号是这个 switch-case 语句的出口位置。

代码 3-37 switch-simp.s

```
1       .file    "switch-simp.c"
2       .text
3       .globl   my_switch
4       .type    my_switch, @function
5   my_switch:
6   .LFB0:
7       .cfi_startproc
8       cmpq $10, %rsi
9       je   .L3
10      cmpq $19, %rsi
11      je   .L4
12      jmp  .L6
13  .L3:
14      leaq 12(%rdi), %rdx
15      .p2align 4,,3
16      jmp  .L2
17  .L4:
18      leaq (%rdi,%rdi,2), %rdx
19      .p2align 4,,3
```

```
20        jmp  .L2
21    .L6:
22        movl $100, %edx
23    .L2:
24        leaq (%rdi,%rdi,4), %rax
25        salq $2, %rax
26        addq %rdx, %rax
27        ret
28        .cfi_endproc
29    .LFE0:
30        .size    my_switch, .-my_switch
31        .ident   "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
32        .section .note.GNU-stack,"",@progbits
```

从上面可以看出 switch-case 语句比循环语句略微复杂一些，而且有两类比较明显不同的实现方式。关于 switch-case 语句的跟踪观察，读者请自行用 gdb 进行尝试，这里不再展开。

3.3.3. 函数调用

关于函数调用方面我们首先来观察函数跳转和返回；然后是参数的传递过程，这分成两种情况，即参数个数少于 6 个和参数个数多余 6 个两种情况；再往后讨局部变量的存储空间；最后讨论和分析缓冲区溢出现象。

函数跳转和返回

首先来看函数调用的最基本操作，即跳转到目标函数和返回到调用处的下一条语句指令。我们在已经看过很多函数跳转的例子，前面 if-else 程序的汇编代码 3-18 的 28 行就是 call 指令跳转到 ifelse 地址(ifelse 函数入口)。前面讨论的 if-else.s 是汇编程序，我们在这里用 objdump 工具查看一下可执行文件中的机器码（及反汇编），如屏显 3-7 所示。注意这里的反汇编 callq 4004ed <ifelse>和代码 3-18 的 call ifelse 形式上并不完全相同，这里有了被调用函数的入口地址 4004ed。但是看该指令对应的机器码是 e8 da ff ff ff，可以看出目标地址是补码表示的 PC 相对跳转地址 0xffffffda 实际上对应-0x26，也就是说跳转地址是 PC-16=0x400513-0x26=4004ed，即<ifelse>函数入口地址。

屏显 3-7 if-else 可执行文件的机器码（部分）

```
[root@localhost cs2]# objdump -d if-else

...

00000000004004ed <ifelse>:
  4004ed:  48 39 f7                cmp    %rsi,%rdi
  4004f0:  7d 06                   jge    4004f8 <ifelse+0xb>
  4004f2:  48 8d 7e 03             lea    0x3(%rsi),%rdi
  4004f6:  eb 07                   jmp    4004ff <ifelse+0x12>
  4004f8:  48 8d 34 bf             lea    (%rdi,%rdi,4),%rsi
  4004fc:  48 01 f6                add    %rsi,%rsi
  4004ff:  48 8d 04 37             lea    (%rdi,%rsi,1),%rax
  400503:  c3                      retq

0000000000400504 <main>:
  400504:  be 05 00 00 00          mov    $0x5,%esi
  400509:  bf 03 00 00 00          mov    $0x3,%edi
  40050e:  e8 da ff ff ff          callq  4004ed <ifelse>
  400513:  f3 c3                   repz retq
```

```
  400515:  66 2e 0f 1f 84 00 00       nopw   %cs:0x0(%rax,%rax,1)
  40051c:  00 00 00
  40051f:  90                         nop
...
```

call 指令除了跳转到指定的地址外，还将返回地址压入到堆栈中。我们用 gdb 跟踪 if-else 可执行文件的运行，如屏显 3-8 所示。程序启动启动后第一条指令就是函数调用 k=ifelse(3,5)，用 disassemble main 和 disassemble ifelse 将 main 函数和 ifelse()函数的机器码打印出来，以便后面设置用于观察的断点。此时打印出程序计数器 PC 寄存器的值为 0x400504——即对应于屏显 3-7 的 mov $0x5,%esi 指令（ifelse(3,5)中的参数准备）。用 p $rsp 发现此时的堆栈指针 RSP 寄存器指向 0x7fffffffe0a8。执行 step 命令将进入到 ifelse()函数中，此时再看 PC 值 0x4004ed 对应 ifelse()函数入口出的第一条指令。此时堆栈指针为 0x7fffffffe0a8-8=0x7fffffffe0a0，用 x 0x7fffffffe0a0 命令查看到堆栈指针指向的内存空间保存了返回地址 0x00400513——对照屏显 3-7 可知改地址是 call 指令的下一条指令。

屏显 3-8 if-else 中函数调用的跳转和返回地址的压栈

```
[root@localhost cs2]# gdb -silent if-else
Reading symbols from /home/lqm/cs2/if-else...done.
(gdb) disassemble main
Dump of assembler code for function main:
=> 0x0000000000400504 <+0>:     mov    $0x5,%esi
   0x0000000000400509 <+5>:     mov    $0x3,%edi
   0x000000000040050e <+10>:    callq  0x4004ed <ifelse>
   0x0000000000400513 <+15>:    repz retq
End of assembler dump.
(gdb) disassemble ifelse
Dump of assembler code for function ifelse:
   0x00000000004004ed <+0>:     cmp    %rsi,%rdi
   0x00000000004004f0 <+3>:     jge    0x4004f8 <ifelse+11>
   0x00000000004004f2 <+5>:     lea    0x3(%rsi),%rdi
   0x00000000004004f6 <+9>:     jmp    0x4004ff <ifelse+18>
   0x00000000004004f8 <+11>:    lea    (%rdi,%rdi,4),%rsi
   0x00000000004004fc <+15>:    add    %rsi,%rsi
   0x00000000004004ff <+18>:    lea    (%rdi,%rsi,1),%rax
   0x0000000000400503 <+22>:    retq
End of assembler dump.
(gdb) start
Temporary breakpoint 1 at 0x400504: file if-else.c, line 14.
Starting program: /home/lqm/cs2/if-else

Temporary breakpoint 1, main () at if-else.c:14
14          k=ifelse(3,5);
(gdb) p $pc
$1 = (void (*)()) 0x400504 <main>
(gdb) b *main+10
Breakpoint 2 at 0x40050e: file if-else.c, line 14.
(gdb) c
Continuing.

Breakpoint 2, 0x000000000040050e in main () at if-else.c:14
14          k=ifelse(3,5);
(gdb) p $pc
$1 = (void (*)()) 0x40050e <main+10>
(gdb) p $rsp
$2 = (void *) 0x7fffffffe0a8
(gdb) b ifelse
Breakpoint 3 at 0x4004ed: file if-else.c, line 4.
(gdb) c
Continuing.
```

```
Breakpoint 3, ifelse (x=x@entry=3, y=y@entry=5) at if-else.c:4
4          if (x<y)
(gdb) p $pc
$3 = (void (*)()) 0x4004ed <ifelse>
(gdb) p $rsp
$4 = (void *) 0x7fffffffe0a0
(gdb) x  0x7fffffffe0a0
0x7fffffffe0a0: 0x00400513
(gdb) b *ifelse+22
Breakpoint 4 at 0x400503: file if-else.c, line 9.
(gdb) c
Continuing.

Breakpoint 4, ifelse (x=8, x@entry=3, y=y@entry=5) at if-else.c:9
9      }
(gdb) p $pc
$5 = (void (*)()) 0x400503 <ifelse+22>
(gdb) p $rsp
$6 = (void *) 0x7fffffffe0a0
(gdb) b *main+15
Breakpoint 5 at 0x400513: file if-else.c, line 16.
(gdb) c
Continuing.

Breakpoint 5, main () at if-else.c:16
16     }
(gdb) p $pc
$7 = (void (*)()) 0x400513 <main+15>
(gdb) p $rsp
$8 = (void *) 0x7fffffffe0a8
(gdb) x 0x7fffffffe0a0
0x7fffffffe0a0: 0x00400513
(gdb)
```

参数传递

在 x86-32 位系统上，函数的参数传递是通过堆栈来完成，而 x86-64 位系统（以及龙芯 MIPS64 系统）采用寄存器结合堆栈的方法来传递参数——但参数数量较少时由特定寄存器完成参数传递，参数数量超过一定数量后剩余的参数则通过堆栈来传递。当参数不超过 6 个时，C 程序的函数将顺序使用 RDI、RSI、RDX、RCX、R8 和 R9 来传递（参见表 3-2 的寄存器使用惯例），若参数继续增加则在堆栈中保存所传递的参数。

对于参数较少的情况，我们前面已经遇到过，例如代码 3-17 的 if-else-main.c（以及相应的汇编代码 3-18）。当 main()执行代码 3-17 的 13 行 k=ifelse(3,5)语句的时候就需要传递参数 3 和 5，对应于汇编代码 3-18 的 26~28 行，先后使用了 RDI 和 RSI 两个寄存器。同样在 ifelse()函数的汇编代码 3-18 的第 10 行和 13 行正是使用的这两个寄存器作为操作数的。

那么对应于参数多于 6 个的情况，我们用代码 3-38 的例子来观察相应的细节。除了前 6 个参数用寄存器传递外，后面的参数都按照 8 字节对齐方式在堆栈种顺序存放（第 7 个参数位于栈顶，即最后才压入到栈中）。

代码 3-38 call-params.c

```
1    int func(int a,int b, char c, long d,long k,char * j,long p_s_1,short
     p_s_2)
2    {
3        int local_A;
4        long  local_B;
5
```

```
6       local_A=a+b;
7       if(k<0)
8           local_A=local_A+c+d+*j;
9       local_B=p_s_1-p_s_2;
10      return local_A+local_B;
11  }
12
13  long main()
14  {
15      long mainA;
16      char  mainB;
17      mainA=func(1,2,3,4,5,&mainB,0xaa,0xcc);
18      return mainA;
19  }
```

gcc -Og -S call-params.c 生成相应的汇编程序，代码 3-39 如所示。从 main()函数的 37~41 行，可以看到前 6 个参数传递情况——a=1 在%edi、b=2 在%esi、c=3 在%edx、d=4 在%ecx、k=5 在%r8d、j 指针的数值 31+%rsp 保存在%r9，这个顺序与表 3-2 中的寄存器使用约定相一致。另外两个参数通过堆栈传递，例如 p_s_1=0xaa=170 通过第 35 行的指令保存到了(%rsp)栈顶的位置，而 p_s_2 则通过第 34 行指令保存到了堆栈的 8(%rsp)的空间中。此时还未执行 call func 指令，对应堆栈中的数据布局如图 3-1 左边所示。执行 call func 之后，将会把返回地址压入堆栈，此时堆栈数据如图 3-1 的中部所示。

代码 3-39 call-params.s

```
1       .file   "call-params.c"
2       .text
3       .globl  func
4       .type   func, @function
5   func:
6   .LFB0:
7       .cfi_startproc
8       movzwl  16(%rsp), %eax     //short p_s_2=204, 参见图 3-1 的中部
9       addl %edi, %esi                    //a+b, local_A 映射到 esi
10      testq   %r8, %r8                   // 判定 k<0
11      jns .L2
12      movsbl  %dl, %edx                  //c->edx
13      addl %edx, %esi                    //local_A+=c
14      addl %ecx, %esi                    //local_A+=d
15      movsbl  (%r9), %edi                //取*j
16      addl %edi, %esi                    //local_A+=*j
17  .L2:
18      movswq  %ax, %rax                  //p_s_2, 前面已经转入 eax
19      movq 8(%rsp), %rcx                 //p_s_1, 参见图 3-1 的中部
20      subq %rax, %rcx                //p_s_1-p_s_2, local_B 映射到%rcx
21      movq %rcx, %rax                // local_B 保存到%rax
22      addl %esi, %eax                //local_A+local_B 保存到%eax
23      ret
24      .cfi_endproc
25  .LFE0:
26      .size   func, .-func
27      .globl  main
28      .type   main, @function
29  main:
30  .LFB1:
31      .cfi_startproc
32      subq $32, %rsp
```

```
33        .cfi_def_cfa_offset 40
34        movl $204, 8(%rsp)
35        movq $170, (%rsp)
36        leaq 31(%rsp), %r9
37        movl $5, %r8d
38        movl $4, %ecx
39        movl $3, %edx
40        movl $2, %esi
41        movl $1, %edi
42        call func
43        cltq
44        addq $32, %rsp
45        .cfi_def_cfa_offset 8
46        ret
47        .cfi_endproc
48    .LFE1:
49        .size    main, .-main
50        .ident   "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
51        .section .note.GNU-stack,"",@progbits
```

图 3-1 给出了 main()函数调用 func()函数是堆栈的变化以及相应的数据布局情况，其中的 rsp 数值以及返回地址将会在后面说明。

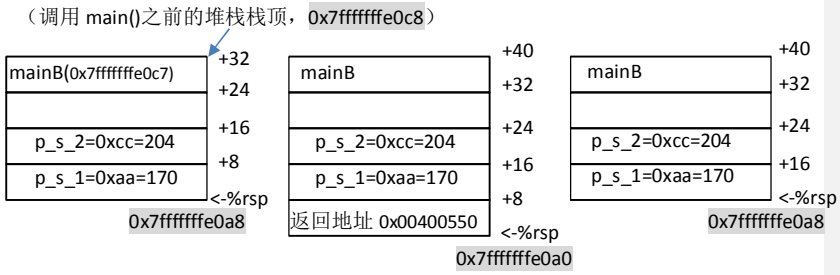

图 3-1 main()执行 call func 指令前后堆栈变化

在阅读 func 的汇编代码时，注意 main()保存 p_s_1 和 p_s_2 用的是(%rsp)和 8(%rsp)，而到了 func()中使用这两个参数的时候，由于堆栈压入了返回地址而下移了 9 个字节，因此其引用表达式为 8(%rsp)和 16(%rsp)。另外，还要注意其中有数据类型的转变所使用的 movzwl 和 movswq 指令。

gcc -Og -g call-params.c -o call-params 生成可执行文件，并用 gdb –silent call-params 跟踪，查看堆栈传递参数的情况（由于寄存器传递的参数比较简单，不再单独观察）。启动 start 命令后程序在 main()入口处暂停，此时堆栈仍是初始化代码所设定的堆栈位置，从屏显 3-9 的第一个 p $rsp 可以看出当前的栈顶位于 0x7fffffffe0c8（这就是图 3-1 顶部指出的栈顶位置）。接着检查两个局部变量，发现 mainA 是映射到寄存器%rax，而 mainB 由于需要地址作为参数，因此必须分配空间——在堆栈中且地址为 0x7fffffffe0c7（参见图 3-1 中 mainB 相应位置）。然后将断点设在*main+51=40054b 的 call 指令处，继续执行到断点并检查堆栈——此时堆栈指针

还未发生变化%rsp=0x7fffffffe0a8。在经过 s 命令执行 call 指令跳转到 func 代码中，此时堆栈向下调整 8 个字节变为 0x7fffffffe0a0，用 x 命令查看当前栈顶内容，发现刚才压入堆栈的是 0x00400550——即 call func 指令的后一条指令（返回地址）。最后用 p 命令查看 p_s_1 和 p_s_2 的地址分别为 0x7fffffffe0a8 和 0x7fffffffe0b0，并用 x 命令查看相应地址上的数值分别是 0xaa 和 0xcc，可以验证出它们的位置与图 3-1 相一致。

屏显 3-9 call-params 函数调用中的堆栈变化及变量位置

```
[root@localhost cs2]# gdb -silent call-params
Reading symbols from /root/cs2/call-params...done.
(gdb) start
Temporary breakpoint 1 at 0x400518: file call-params.c, line 14.
Starting program: /root/cs2/call-params

Temporary breakpoint 1, main () at call-params.c:14
14    {
(gdb) p $rsp
$1 = (void *) 0x7fffffffe0c8
(gdb) p &mainA
Address requested for identifier "mainA" which is in register $rax
(gdb) p &mainB
$2 = 0x7fffffffe0c7 ""
(gdb) disassemble
Dump of assembler code for function main:
=> 0x0000000000400518 <+0>:     sub    $0x20,%rsp
   0x000000000040051c <+4>:     movl   $0xcc,0x8(%rsp)
   0x0000000000400524 <+12>:    movq   $0xaa,(%rsp)
   0x000000000040052c <+20>:    lea    0x1f(%rsp),%r9
   0x0000000000400531 <+25>:    mov    $0x5,%r8d
   0x0000000000400537 <+31>:    mov    $0x4,%ecx
   0x000000000040053c <+36>:    mov    $0x3,%edx
   0x0000000000400541 <+41>:    mov    $0x2,%esi
   0x0000000000400546 <+46>:    mov    $0x1,%edi
   0x000000000040054b <+51>:    callq  0x4004ed <func>
   0x0000000000400550 <+56>:    cltq
   0x0000000000400552 <+58>:    add    $0x20,%rsp
   0x0000000000400556 <+62>:    retq
End of assembler dump.
(gdb) b *main+51
Breakpoint 2 at 0x40054b: file call-params.c, line 17.
(gdb) c
Continuing.

Breakpoint 2, 0x000000000040054b in main () at call-params.c:17
17          mainA=func(1,2,3,4,5,&mainB,0xaa,0xcc);
(gdb) p $rsp
$3 = (void *) 0x7fffffffe0a8
(gdb) s
func (a=a@entry=1, b=b@entry=2, c=c@entry=3 '¥003', d=d@entry=4, k=k@entry=5,
    j=j@entry=0x7fffffffe0c7 "", p_s_1=p_s_1@entry=170, p_s_2=p_s_2@entry=204)
    at call-params.c:2
2     {
(gdb) p $rsp
$4 = (void *) 0x7fffffffe0a0
(gdb) x 0x7fffffffe0a0
0x7fffffffe0a0: 0x00400550
(gdb) p &p_s_1
$1 = (long *) 0x7fffffffe0a8
(gdb) p &p_s_2
$2 = (short *) 0x7fffffffe0b0
(gdb) x 0x7fffffffe0a8
0x7fffffffe0a8: 0x000000aa
(gdb) x 0x7fffffffe0b0
```

```
0x7fffffffe0b0: 0x000000cc
(gdb)
```

当 func 函数返回后，堆栈指针会上调 8 个字节，如图 3-1 右部所示，请读者自行观察。

栈帧结构

从前面例子我们看到了函数跳转后，系统会在堆栈中存放返回地址、参数和局部变量。其实系统会为每一次函数调用准备一个堆栈中的数据结构，我们称之为栈帧。在 x86-32 系统上，栈帧由 EBP 寄存器指出，而在 x86-64 位系统上则无需独立的栈帧指针（RSP 寄存器可以用于其它用途），而是由 RSP 统一管理栈帧中的数据。

下面先讨论一个函数的栈帧结构，然后再讨论函数嵌套调用，最后简单展示一下缓冲区溢出现象。

■ 栈帧

图 3-2 给出了函数调用中栈帧的结构示意图，描绘了函数 P 如何在自己的栈帧中准备好调用参数和返回地址，以及被调用函数 Q 如何构建自己的栈帧（甚至继续调用下一级函数）。

首先来看上级函数的栈帧处理。函数 P 将前 6 个参数用寄存器传递，而第 7 个参数及以后的参数则在保存到堆栈进行传递，在执行 call 指令调用 Q 函数时将返回地址压入堆栈，从而完成了 P 函数在调用 Q 函数时的完整栈帧。此时栈顶由堆栈指针 RSP 指出，该位置也是 P 函数和 Q 函数栈帧分界点。

接着来看被调用函数如何建立自己的栈帧。Q 函数首先将堆栈指针下移若干字节（8 字节对齐）形成自己的栈帧空间，其内部空间布局自上而下分别是被保存的寄存器内容、局部变量、调用下一级函数（例如 R 函数）所需要的参数。关于被保存的寄存器、局部变量和下一级函数的参数都不是必须的，如果不需要则可以不分配相应的空间。但是函数的返回地址是必须的，会出现在每一个函数的栈帧里。从 Q 函数返回到 P 函数的地址，是 P、Q 栈帧的分界点，因此出现有些文献将返回地址归属到 P 的栈帧，另一些则将它归属到 Q 栈帧。

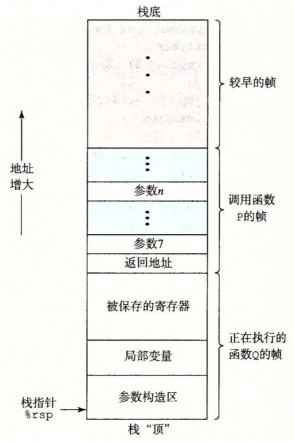

图 3-2 栈帧结构

在 call-params.c 的例子中，我们已经看过 P 函数（main 函数）准备参数的——包括 6 个寄存器参数和两个堆栈传递的参数 p_s_1 和 p_s_2，以及返回地址的压栈过程。例子中展示了 Q 函数（func）的两个局部变量（local_A、local_B）映射到寄存器（%esi、%rcx）的实现，还没有观察到 Q 函数局部变量在堆栈中分配。虽然例子中也展示了 main 的局部变量（mainB）在 main 函数的栈帧出现，但是我们还使用新的例子来展示局部变量及可变数组在栈帧中的使用情况。

✧ 局部变量

我们以代码 3-40 为例，将 main 函数作为 P 函数，而将 func 作为 Q 函数，现在观察 Q 函数中的局部变量 i 和 a1[20]。

代码 3-40 call-localvar1.c

```
1   int leaf_fun(long lf1,long lf2)
2   {   int k;
3       k=lf1+lf2;
4       return k;
5   }
6   int func(long f1,long f2)
7   {
8       int i;
9       int a1[20];
10      for(i=0;i<19;i++)
11          a1[i]=leaf_fun(i,i);
12      return a1[2];
13  }
14
15  long main()
16  {
17      int mf1=20;
18      int mf2=11;
19      mf2=func(mf1,mf2);
20      return mf2;
21  }
```

用 gcc -Og -S call-localvar1.c 产生相应的汇编代码如代码 3-41 所示。可以看到 func 函数入口处将 rbp 和 rbx 压入堆栈进行保存，并在返回前的 41 和 43 行从堆栈中弹出恢复原值，这对应于图 3-2 所示的 Q 函数栈帧中的“被保存的寄存器”。在保存 rbp 和 rbx 之后的第 24 行 subq $80, %rsp 指令将堆栈指针下拉 80 个字节建立栈帧，以及函数返回前第 39 行的 addq $80, %rsp 指令将堆栈指针上移 80 个字节撤销栈帧。func 函数的局部变量 i 用作循环变量，被映射到 ebx 寄存器（因此未在栈帧中占用空间）。func 函数的局部数组 a1[20]则使用栈帧中所分配 80 个字节。在 func 调用 leaf_fun 的时候，由于只需要 2 个参数，因此 func 的栈帧中并未出现图 3-2 所示的 Q 函数的“参数构造区”（这个已经在前面代码 3-18 中分析过）。

代码 3-41 call-localvar1.s

```
22      .file   "call-localvar1.c"
23      .text
24      .globl  leaf_fun
25      .type   leaf_fun, @function
26  leaf_fun:
27  .LFB0:
28      .cfi_startproc
29      leal (%rdi,%rsi), %eax          // lf1+lf2
30      ret
```

```
31      .cfi_endproc
32  .LFE0:
33      .size    leaf_fun, .-leaf_fun
34      .globl   func
35      .type    func, @function
36  func:
37  .LFB1:
38      .cfi_startproc
39      pushq    %rbp                    //保存 rbp
40      .cfi_def_cfa_offset 16
41      .cfi_offset 6, -16
42      pushq    %rbx                    //保存 rbx
43      .cfi_def_cfa_offset 24
44      .cfi_offset 3, -24
45      subq $80, %rsp                   //建立栈帧
46      .cfi_def_cfa_offset 104
47      movl $0, %ebx
48      jmp  .L3
49  .L4:
50      movslq   %ebx, %rbp
51      movq %rbp, %rsi
52      movq %rbp, %rdi
53      call leaf_fun
54      movl %eax, (%rsp,%rbp,4)         //%rsp+%rbp*4 指向 a1[i]
55      addl $1, %ebx
56  .L3:
57      cmpl $18, %ebx
58      jle  .L4
59      movl 8(%rsp), %eax
60      addq $80, %rsp                       //撤销栈帧
61      .cfi_def_cfa_offset 24
62      popq %rbx                            //恢复 rbx
63      .cfi_def_cfa_offset 16
64      popq %rbp                            //恢复 rbp
65      .cfi_def_cfa_offset 8
66      ret
67      .cfi_endproc
68  .LFE1:
69      .size    func, .-func
70      .globl   main
71      .type    main, @function
72  main:
73  .LFB2:
74      .cfi_startproc
75      movl $11, %esi
76      movl $20, %edi
77      call func
78      cltq
79      ret
80      .cfi_endproc
81  .LFE2:
82      .size    main, .-main
83      .ident   "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
84      .section .note.GNU-stack,"",@progbits
```

下面对代码 3-40 做一点小修改，使得 func 就是最底层的函数（不再调用其它函数），形成代码 3-42。这时候 gcc 为了避免不必要的操作，并未将堆栈指针下移——但是并不影响局

部变量使用堆栈下面的空间（地址低于堆栈指针 RSP 指向的位置），这种差别如图 3-3 所示，下面将分析汇编并展示这种差异。

代码 3-42 call-localvar2.c

```
1   int func(long f1,long f2)
2   {
3       int i;
4       int a1[20];
5       for(i=0;i<19;i++)
6           a1[i]=f1*i+f2;
7       return a1[2];
8   }
9
10  long main()
11  {
12      int mf1=20;
13      int mf2=11;
14      mf2=func(mf1,mf2);
15      return mf2;
}
```

用 gcc -Og -S call-localvar2.c 生成汇编代码如代码 3-43 所示。这是我们看到 func 函数入口出并不存在将 rsp 下移的操作（请回顾代码 3-41 中的 45 行用 subq $80, %rsp 指令将 rsp 下移），同样在函数返回前也未看到调整堆栈指针的操作。但是在第 15 行对 a1[i]赋值的指令中，可以推出测 a1[0]位于%rsp-80 的位置，类推得出 a1[1]位于%rsp-76 等等。这里的局部变量 a1[]和前面们一样都是占用的 80 字节并且占用了相同的空间，区别在于这里的 rsp 并未随之调整——相应地利用 rsp 访问该数组时起点地址表达方式略有不同。

代码 3-43 call-localvar2.s

```
1       .file   "call-localvar2.c"
2       .text
3       .globl  func
4       .type   func, @function
5   func:
6   .LFB0:
7       .cfi_startproc
8       movl $0, %eax
9       jmp .L2
10  .L3:
11      movslq  %eax, %rdx
12      movl %eax, %ecx
13      imull   %edi, %ecx
14      addl %esi, %ecx
15      movl %ecx, -80(%rsp,%rdx,4)             //%rsp-80+rdx*4 指向a1[i]
16      addl $1, %eax
17  .L2:
18      cmpl $18, %eax
19      jle .L3
20      movl -72(%rsp), %eax
21      ret
22      .cfi_endproc
23  .LFE0:
24      .size   func, .-func
25      .globl  main
26      .type   main, @function
27  main:
```

```
28  .LFB1:
29      .cfi_startproc
30      movl $11, %esi
31      movl $20, %edi
32      call func
33      cltq
34      ret
35      .cfi_endproc
36  .LFE1:
37      .size    main, .-main
38      .ident   "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
39      .section .note.GNU-stack,"",@progbits
```

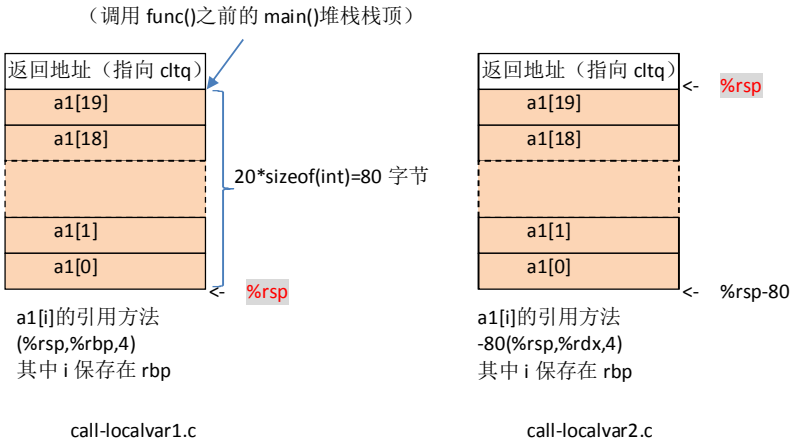

图 3-3 函数调用中最底层函数的栈帧差异

也就是说对于 P 角色的函数，其内部局部变量的分配和使用并不一定要移动 rsp 指针，但是如果 P 函数要调用下一级函数则有必要将 rsp 下移，以便下一级的 Q 函数可以建立自己的栈帧。

✧ 变长帧、变长数组

变长数组其实也是函数的局部变量。

使用了 rbp 作为栈帧指针，

允许多个变长数组吗？

普通局部变量将使用 rbp 作为基址来引用

■ **调用栈/多层调用**

Gstack 查看调用关系

Gdb 查看调用关系

https://blog.csdn.net/yangzhongxuan/article/details/6911689

由于函数会呈现多层次的调用关系，因此随着各级函数建立自己的栈帧的过程，在堆栈里面会形成一个所谓的“调用栈”——就是从地址高端往地址低端逐级层叠的栈帧。利用这些层叠的栈帧可以分析出当前所执行的指令所属函数以及如何从 main()函数经过多级调用而到到当当前指令，也包括各级调用时所使用的参数。

需要注意：这里指的函数是指函数的一次执行。同一个函数被多次重复进入则各自建立一次栈帧，典型的就是函数的递归调用，虽然只有一套函数代码，但是递归调用 n 次则形成 n 个独立的栈帧，每个都代表一次调用执行。

下面用一个经典的递归调用例子，即产生费波纳奇数列的程序，如代码 3-44 所示。

代码 3-44 fibonacci.c

```
#include<stdio.h>
#define N 8

int Fibonacci(int n)
{
    int f;

    if(n<=2)
        f=1;
    else
        f = Fibonacci(n-1) + Fibonacci(n-2);
    return f;
}

int main()
{
    int c=0;
    c = Fibonacci(N);
    return c;
}
```

用 gcc -Og -S fibonacci.c 生成相应的汇编代码，如代码 3-45 所示。

代码 3-45 fibonacci.s

```
    .file   "fibonacci.c"
    .text
    .globl  Fibonacci
    .type   Fibonacci, @function
Fibonacci:
.LFB11:
    .cfi_startproc
    pushq   %rbp                        //保存 rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    pushq   %rbx                        //保存 rbx
    .cfi_def_cfa_offset 24
    .cfi_offset 3, -24
    subq $8, %rsp                       //堆栈指针下移 8 字节
    .cfi_def_cfa_offset 32
    movl %edi, %ebx                     //将参数 n 拷贝一份到 ebx
    cmpl $2, %edi                       //检查 n<=2
    jle .L3                             //n<=2 则跳转（返回数值 1）
    leal -1(%rdi), %edi                 //n-1 作为 Fibonacci（）参数
    call Fibonacci                      //eax=Fibonacci(n-1)
    movl %eax, %ebp                     //Fibonacci(n-1)保存到 ebp
    leal -2(%rbx), %edi                 //n-2 作为 Fibonacci()参数
```

```
23      call Fibonacci                          //eax=Fibonacci(n-2)
24      addl %ebp, %eax                 //eax=Fibonacci(n-2)+Fibonacci(n-1)
25      jmp  .L2
26  .L3:
27      movl $1, %eax
28  .L2:
29      addq $8, %rsp                           //堆栈指针上移 8 字节
30      .cfi_def_cfa_offset 24
31      popq %rbx                               //恢复 rbx
32      .cfi_def_cfa_offset 16
33      popq %rbp                               //恢复 rbp
34      .cfi_def_cfa_offset 8
35      ret
36      .cfi_endproc
37  .LFE11:
38      .size    Fibonacci, .-Fibonacci
39      .globl   main
40      .type    main, @function
41  main:
42  .LFB12:
43      .cfi_startproc
44      subq $8, %rsp
45      .cfi_def_cfa_offset 16
46      movl $8, %edi                           //Fibonacci()参数为 8
47      call Fibonacci
48      addq $8, %rsp
49      .cfi_def_cfa_offset 8
50      ret
51      .cfi_endproc
52  .LFE12:
53      .size    main, .-main
54      .ident   "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
55      .section .note.GNU-stack,"",@progbits
```

用 gcc -Og -g -o fibonacci fibonacci.c 命令产生可执行文件，然后用 gdb 调试，如屏显 3-10 所示。我们用 b 11 if n==5 将断点设置在代码 3-44 的第 11 行，并且将条件设置为 n==5，用 c 命令继续运行直到暂停执行。程序暂停后，用 p n 确定当前 n 值为 5，因此应该完成了 n=8、7、6、5 的调用——即 1）main()发出的 Fibonacci(8)、2）Fibonacci(8)发出的 Fibonacci(7)、3）Fibonacci(7)发出的 Fibonacci(6)、4）Fibonacci(6)发出的 Fibonacci(5)。

此时用 bt（对应 backtrace）命令和 bt full 命令查看调用栈，可以看到 4 个完整的栈帧（含返回地址）和 1 个不完整的栈帧（没有进一步调用下一级函数，因此也还没有返回地址）main()函数自己的战阵。bt 和 bt full 两个命令输出的堆栈编号 4~0 分别对应 main()、Fibonacci(8)、Fibonacci(7)、Fibonacci(6)、Fibonacci(5)的栈帧。用 info frame X 可以查看编号 X 的栈帧详细信息。

除了编号为 0 的 Fibonacci(5)正在运行，还没有调用下一级函数，因此没有返回地址外，每个栈帧中都保存一个返回地址。我们查看屏显 3-10 后面的 disassemble 反汇编出来的代码，可以知道 4 号栈帧的返回地址为 0x40052a，即 main()调用 Fibonacci()之后的下一条指令。编号为 3~1 的三个栈帧上的返回地址都是 0x400502，对应于 Fibonacci(n)调用的 Fibonacci(n-1)之后的指令。而编号为 0 的栈帧因为还没有调用下一级函数，因此没有返回地址——info frame 0 所显示的 rip=0x4004fa 是断点处的位置。

我们用 info frame X 只查看了 0 号、1 号和 4 号栈帧的内容。

屏显 3-10 gdb 查看 fibonacci 中的调用栈

```
[root@localhost cs2]# gdb -silent fibonacci
Reading symbols from /home/lqm/cs2/fibonacci...done.
(gdb) start
Temporary breakpoint 1 at 0x40051c: file fibonacci.c, line 16.
Starting program: /home/lqm/cs2/fibonacci

Temporary breakpoint 1, main () at fibonacci.c:16
16    {
(gdb) b 11 if n==5
Breakpoint 2 at 0x4004fa: file fibonacci.c, line 11.
(gdb) c
Continuing.

Breakpoint 2, Fibonacci (n=n@entry=5) at fibonacci.c:11
11              f = Fibonacci(n-1) + Fibonacci(n-2);
(gdb) p n
$1 = 5
(gdb) bt
#0  Fibonacci (n=n@entry=5) at fibonacci.c:11
#1  0x0000000000400502 in Fibonacci (n=n@entry=6) at fibonacci.c:11
#2  0x0000000000400502 in Fibonacci (n=n@entry=7) at fibonacci.c:11
#3  0x0000000000400502 in Fibonacci (n=n@entry=8) at fibonacci.c:11
#4  0x000000000040052a in main () at fibonacci.c:18
(gdb) bt full
#0  Fibonacci (n=n@entry=5) at fibonacci.c:11
        f = <optimized out>
#1  0x0000000000400502 in Fibonacci (n=n@entry=6) at fibonacci.c:11
        f = <optimized out>
#2  0x0000000000400502 in Fibonacci (n=n@entry=7) at fibonacci.c:11
        f = <optimized out>
#3  0x0000000000400502 in Fibonacci (n=n@entry=8) at fibonacci.c:11
        f = <optimized out>
#4  0x000000000040052a in main () at fibonacci.c:18
        c = 0
(gdb) info frame 0
Stack frame at 0x7fffffffe040:
 rip = 0x4004fa in Fibonacci (fibonacci.c:11); saved rip 0x400502
 called by frame at 0x7fffffffe060
 source language c.
 Arglist at 0x7fffffffe020, args: n=n@entry=5
 Locals at 0x7fffffffe020, Previous frame's sp is 0x7fffffffe040
 Saved registers:
  rbx at 0x7fffffffe028, rbp at 0x7fffffffe030, rip at 0x7fffffffe038
(gdb) info frame 1
Stack frame at 0x7fffffffe060:
 rip = 0x400502 in Fibonacci (fibonacci.c:11); saved rip 0x400502
 called by frame at 0x7fffffffe080, caller of frame at 0x7fffffffe040
 source language c.
 Arglist at 0x7fffffffe040, args: n=n@entry=6
 Locals at 0x7fffffffe040, Previous frame's sp is 0x7fffffffe060
 Saved registers:
  rbx at 0x7fffffffe048, rbp at 0x7fffffffe050, rip at 0x7fffffffe058
(gdb) info frame 4
Stack frame at 0x7fffffffe0b0:
 rip = 0x40052a in main (fibonacci.c:18); saved rip 0x7ffff7a39c05
 caller of frame at 0x7fffffffe0a0
 source language c.
 Arglist at 0x7fffffffe098, args:
 Locals at 0x7fffffffe098, Previous frame's sp is 0x7fffffffe0b0
 Saved registers:
  rip at 0x7fffffffe0a8
(gdb) disassemble main
```

```
Dump of assembler code for function main:
   0x000000000040051c <+0>:     sub    $0x8,%rsp
   0x0000000000400520 <+4>:     mov    $0x8,%edi
   0x0000000000400525 <+9>:     callq  0x4004ed <Fibonacci>
   0x000000000040052a <+14>:    add    $0x8,%rsp
   0x000000000040052e <+18>:    retq
End of assembler dump.
(gdb) disassemble Fibonacci
Dump of assembler code for function Fibonacci:
   0x00000000004004ed <+0>:     push   %rbp
   0x00000000004004ee <+1>:     push   %rbx
   0x00000000004004ef <+2>:     sub    $0x8,%rsp
   0x00000000004004f3 <+6>:     mov    %edi,%ebx
   0x00000000004004f5 <+8>:     cmp    $0x2,%edi
   0x00000000004004f8 <+11>:    jle    0x400510 <Fibonacci+35>
=> 0x00000000004004fa <+13>:    lea    -0x1(%rdi),%edi
   0x00000000004004fd <+16>:    callq  0x4004ed <Fibonacci>
   0x0000000000400502 <+21>:    mov    %eax,%ebp
   0x0000000000400504 <+23>:    lea    -0x2(%rbx),%edi
   0x0000000000400507 <+26>:    callq  0x4004ed <Fibonacci>
   0x000000000040050c <+31>:    add    %ebp,%eax
   0x000000000040050e <+33>:    jmp    0x400515 <Fibonacci+40>
   0x0000000000400510 <+35>:    mov    $0x1,%eax
   0x0000000000400515 <+40>:    add    $0x8,%rsp
   0x0000000000400519 <+44>:    pop    %rbx
   0x000000000040051a <+45>:    pop    %rbp
   0x000000000040051b <+46>:    retq
End of assembler dump.
(gdb)
```

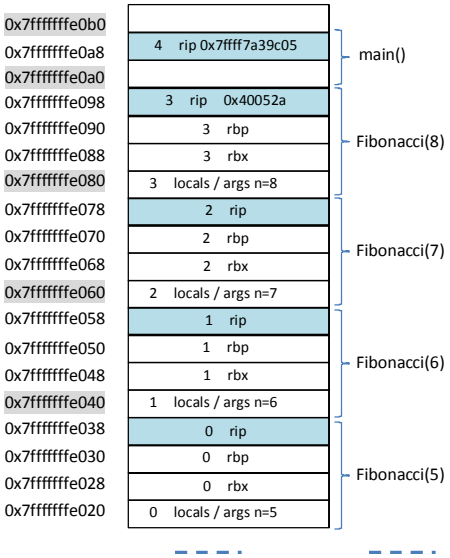

图 3-4 fibonacci 中的调用栈

■ 进程堆栈

在分析完普通函数的调用和栈帧结构后，我们来看看系统为刚创建的进程准备了什么样的 main 函数堆栈的，并且将环境变量也一同分析。

✧ main 函数的参数

虽然我们还未学习链接过程，因此也没有完整的分析程序的内存布局。但是前面 2.1.5 已经大致了解了程序有布局的概念，以及屏显 3-3 给出了一个进程的程序空间（虚存）上的布局情况（当然也包括 stack 占用的区间），后面的图 4-1 则给出了进程影像的直观图示。图 4-1 的%rsp 指向了用户栈的起点，但是整个用户态栈的细节并未展示。我们前面已经分析了栈帧结构，以及分析了函数嵌套调用所形成的层次性的栈帧堆叠——这些就是用户态栈的内部结构。

现在我们来分析用户态栈的起始状态。从屏显 3-3 的 stack 区间 7ffffffde000-7fffffffff000 可以看出，堆栈的栈底应该在 7fffffffff000。参考图 4-2 可知我们所使用的系统是 48 位物理地址的系统，因为堆栈的起点位于 7fffffffff000（占 48bit）与图中 48 位地址划分相一致，否则 56 位或 64 位系统的堆栈起始地址将在更高地址处。在前面的代码 3-38 例子中，进入 main()函数之后 gdb 显示此时的堆栈指针为 0x7fffffffe0c8（屏显 3-9）在 7fffffffff000 地址略低的位置。

但是实际上 Linux 出于安全考虑而选择随机地选择堆栈的起点，以及会在上述地址保存环境变量，因此 main()函数的起点要比 7fffffffff000 更低——并且在略高于 main()函数的栈帧的空间中保存有 shell 传递进来的命令行参数。下面我们用一个例子来展示 main()刚开始工作时，系统为之准备的堆栈中的命令行参数，该例子所对应的初始堆栈结构（以及命令行参数）如图 3-5 所示。

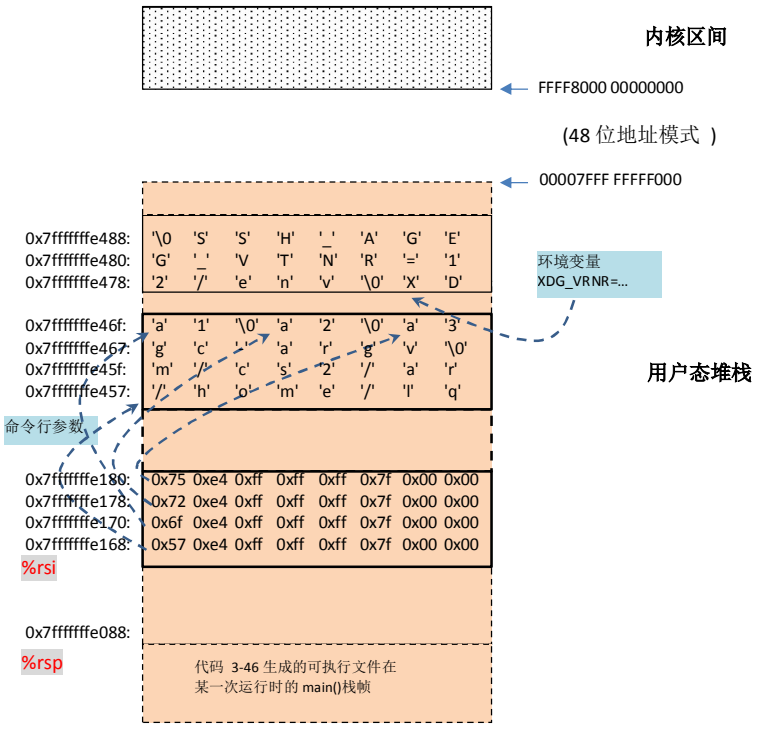

图 3-5 系统为 main 函数准备的堆栈

代码 3-46 是展示命令行参数的例子，首先打印参数个数（含可执行文件名），然后逐个打印命令行参数字符串。

代码 3-46 argc-argv.c

```
1   #include <stdio.h>
2   int main(int argc,char *argv[])
3   {
4       int i;
5       printf("argc = %d \n",argc);
6       for(i=0;i<argc; i++)
7           printf("arg%d:%s\n",i,argv[i]);
8       return 0;
9   }
```

使用命令 gcc -Og -g -o argc-argv argc-argv.c 生成可执行文件并执行，可以得到我们所需的结果，如屏显 3-11 所示。我们在命令行中输入 argc-argv a1 a2 a3，输出提示参数个数为 4 并正确打印出相应的参数字符串。

屏显 3-11 argc-argv 打印命令行参数

```
[root@localhost cs2]# argc-argv  a1 a2 a3
argc = 4
arg0:argc-argv
arg1:a1
arg2:a2
arg3:a3
[root@localhost cs2]#
```

下面用 gdb -silent argc-argv 进行调试，并通过 run a1 a2 a3 将命令行参数传入（本例使用 set args a1 a2 a3 来设置命令行参数），如屏显 3-12 所示。在 start 命令启动程序后，首先检查得到当前的堆栈指针 rsi 的数值为 0x7fffffffe088。

通过 p &argc 和 p &argv 命令，我们发现它们是通过 rdi 和 rsi 传入的（参数少于 6 个，用寄存器传递，请回顾表 3-2 中前 6 个参数的寄存器使用约定）。继续检查 argv（即%rsi）可知其地址为 0x7fffffffe168，该地址指向一个字符串指针列表——用 x/32x $rsi 我们找到对应的四个参数字符串首地址：0x7fffffffe457、0x7fffffffe46f、0x7fffffffe472 和 0x7fffffffe475。我们再用 x/32c 0x7fffffffe457 逐个字节检查上述地址上的数据，可以看见相应地址上存储了 /home/lqm/cs2/argc-argv、a1、a2 和 a3 四个字符串。最后也可以在 gdb 中直接打印 argv[0~3] 也获得相一致的结果。我们将这些分析结果绘制成图，如图 3-5 所示。

屏显 3-12 gdb 查看命令行参数

```
[root@localhost cs2]# gdb -silent argc-argv
Reading symbols from /home/lqm/cs2/argc-argv...done.
(gdb) set args a1 a2 a3
(gdb) start
Temporary breakpoint 1 at 0x40052d: file argc-argv.c, line 3.
Starting program: /home/lqm/cs2/argc-argv a1 a2 a3

Temporary breakpoint 1, main (argc=4, argv=0x7fffffffe168) at argc-argv.c:3
3     {
(gdb) p $rsp
$8 = (void *) 0x7fffffffe088
(gdb) p argc
$1 = 4
(gdb) p argv
```

```
$2 = (char **) 0x7fffffffe168
(gdb) p &argc
Address requested for identifier "argc" which is in register $rdi
(gdb) p &argv
Address requested for identifier "argv" which is in register $rsi
(gdb) x/32x $rsi
0x7fffffffe168: 0x57 0xe4 0xff 0xff 0xff 0x7f 0x00 0x00
0x7fffffffe170: 0x6f 0xe4 0xff 0xff 0xff 0x7f 0x00 0x00
0x7fffffffe178: 0x72 0xe4 0xff 0xff 0xff 0x7f 0x00 0x00
0x7fffffffe180: 0x75 0xe4 0xff 0xff 0xff 0x7f 0x00 0x00
(gdb) x/32c 0x7fffffffe457
0x7fffffffe457: 47 '/'  104 'h'  111 'o'  109 'm'  101 'e'  47 '/'  108 'l'  113 'q'
0x7fffffffe45f: 109 'm'  47 '/'  99 'c'  115 's'  50 '2'  47 '/'  97 'a'  114 'r'
0x7fffffffe467: 103 'g'  99 'c'  45 '-'  97 'a'  114 'r'  103 'g'  118 'v'  0 '\000'
0x7fffffffe46f: 97 'a'  49 '1'  0 '\000'  97 'a'  50 '2'  0 '\000'  97 'a'  51 '3'
(gdb) p argv[0]
$4 = 0x7fffffffe457 "/home/lqm/cs2/argc-argv"
(gdb) p argv[1]
$5 = 0x7fffffffe46f "a1"
(gdb) p argv[2]
$6 = 0x7fffffffe472 "a2"
(gdb) p argv[3]
$7 = 0x7fffffffe475 "a3"
(gdb)
```

✧ 环境变量位置

环境变量也在堆栈区，由 shell 在创建进程的时候填写的。我了探究环境变量在进程空间中的位置，我们用 C 语言的 getenv()获得其中的 PATH 环境变量，然后再设法查看全部环境变量，具体代码如代码 3-47 所示。

代码 3-47 env.c

```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   int main(void)
5   {
6       char *pathvar;
7       pathvar = getenv("PATH");
8       printf("pathvar=%s",pathvar);
9       return 0;
10  }
```

用 gcc -Og -g env.c -o env 生成可执行文件，然后用 gdb -silent env 开始调试。用 n 命令执行完 getenv()之后，用 p &pathvar 查看参数所在位置时，提示由$rax 指出该环境变量的地址。查看$rax 指向的地址空间（0x7fffffffed15），可以看到改进程的 PATH 环境变量为“/usr/local/bin:/usr/local/sbin: ... ”。这确定了 PATH 环境变量所在空间，其他环境变量仍未知。

屏显 3-13 用 gdb 查看 env 进程的 PATH 环境变量

```
[root@localhost cs2]# gdb -silent env
Reading symbols from /home/lqm/cs2/env...done.
(gdb) start
Temporary breakpoint 1 at 0x40057d: file env.c, line 5.
Starting program: /home/lqm/cs2/env

Temporary breakpoint 1, main () at env.c:5
5   {
(gdb) n
7       pathvar = getenv("PATH");
```

```
(gdb) n
8           printf("pathvar=%s",pathvar);
(gdb) p &pathvar
Address requested for identifier "pathvar" which is in register $rax
(gdb) p/x $rax
$3 = 0x7fffffffed15
(gdb) x/32c 0x7fffffffed15
0x7fffffffed15: 47 '/'    117 'u'    115 's'    114 'r'    47 '/'    108 'l'    111 'o'    99 'c'
0x7fffffffed1d: 97 'a'    108 'l'    47 '/'     98 'b'     105 'i'   110 'n'    58 ':'     47 '/'
0x7fffffffed25: 117 'u'   115 's'    114 'r'    47 '/'     108 'l'   111 'o'    99 'c'     97 'a'
0x7fffffffed2d: 108 'l'   47 '/'     115 's'    98 'b'     105 'i'   110 'n'    58 ':'     47 '/'
```

为了找到其他环境变量所在的存储空间，首先需要知道该进程所有还进变量是什么。下面借助 Linux 的 proc 文件系统提供的信息来获得该系统上 env 程序的环境变量，先找到进程号然后查看/proc/PID/environ 文件内容即可，如屏显 3-14 所示。其中可以看到 PATH=/usr/local/bin:... 如我们前面 gdb 在进程 0x7fffffffed15 地址上查看到的相一致。

屏显 3-14 env（pid=28342）可执行文件的环境变量

```
[root@localhost cs2]# ps -A|grep env
28342 pts/0    00:00:00 env
[root@localhost cs2]#  cat /proc/28342/environ
XDG_VTNR=1SSH_AGENT_PID=2904XDG_SESSION_ID=1HOSTNAME=localhost.localdomainIMSETTINGS_INTEGRATE_DESKTOP=yes
GPG_AGENT_INFO=/run/user/1000/keyring/gpg:0:1TERM=xterm-256colorSHELL=/bin/bashXDG_MENU_PREFIX=gnome-
VTE_VERSION=3804HISTSIZE=1000WINDOWID=39845895IMSETTINGS_MODULE=IBusUSER=lqmLS_COLORS=rs=0:di=38;5;27:ln=3
8;5;51:mh=44;38;5;15:pi=40;38;5;11:so=38;5;13:do=38;5;5:bd=48;5;232;38;5;11:cd=48;5;232;38;5;3:or=48;5;232
;38;5;9:mi=05;48;5;232;38;5;15:su=48;5;196;38;5;15:sg=48;5;11;38;5;16:ca=48;5;196;38;5;226:tw=48;5;10;38;5

...

38;5;45:*.au=38;5;45:*.flac=38;5;45:*.mid=38;5;45:*.midi=38;5;45:*.mka=38;5;45:*.mp3=38;5;45:*.mpc=38;5;45
:*.ogg=38;5;45:*.ra=38;5;45:*.wav=38;5;45:*.axa=38;5;45:*.oga=38;5;45:*.spx=38;5;45:*.xspf=38;5;45:SSH_AUT
H_SOCK=/run/user/1000/keyring/sshUSERNAME=lqmSESSION_MANAGER=local/unix:@/tmp/.ICE-
unix/2708,unix/unix:/tmp/.ICE-
unix/2708COLUMNS=151GNOME_SHELL_SESSION_MODE=classicPATH=/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin
:/bin:/sbin:/home/lqm/.local/bin:/home/lqm/bin:MAIL=/var/spool/mail/lqmDESKTOP_SESSION=gnome-
classic_=/usr/bin/gdbQT_IM_MODULE=ibusPWD=/home/lqm/cs2XMODIFIERS=@im=ibusLANG=zh_CN.UTF-
8GDM_LANG=zh_CN.UTF-8LINES=34GDMSESSION=gnome-
classicHISTCONTROL=ignoredupsXDG_SEAT=seat0HOME=/rootSHLVL=3MALLOC_TRACE=/tmp/tGNOME_DESKTOP_SESSION_ID=th
is-is-deprecatedXDG_SESSION_DESKTOP=gnome-
classicLOGNAME=lqmDBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-
bifmq1ORxC,guid=c0adcad74a6305e331e155045ab5b1f4LESSOPEN=||/usr/bin/lesspipe.sh %sWINDOWPATH=1XDG_RUNTIME_
DIR=/run/user/1000DISPLAY=:0XDG_CURRENT_DESKTOP=GNOME-Classic:GNOMEXAUTHORITY=/root/.xauthR3NqWK
[root@localhost cs2]#
```

我们尝试往更低地址处查看，发现了全部的环境变量的起点位于 0x7fffffffe47e 位置上是第一个环境变量 XDG_VTNR（见屏显 3-15），比较屏显 3-14 显示的第一个环境变量 XDG_VTNR 及其内容，可以确定它们是一致的。我们将环境变量的信息绘制到图 3-5 中用户态堆栈的顶部位置。

屏显 3-15 堆栈区顶部的环境变量存储空间

```
(gdb) x/128c 0x7fffffffe478
0x7fffffffe478: 50 '2'    47 '/'    101 'e'   110 'n'   118 'v'   0 '\000'  88 'X'    68 'D'
0x7fffffffe480: 71 'G'    95 '_'    86 'V'    84 'T'    78 'N'    82 'R'    61 '='    49 '1'
0x7fffffffe488: 0 '\000'  83 'S'    83 'S'    72 'H'    95 '_'    65 'A'    71 'G'    69 'E'
0x7fffffffe490: 78 'N'    84 'T'    95 '_'    80 'P'    73 'I'    68 'D'    61 '='    50 '2'
0x7fffffffe498: 57 '9'    48 '0'    52 '4'    0 '\000'  88 'X'    68 'D'    71 'G'    95 '_'
0x7fffffffe4a0: 83 'S'    69 'E'    83 'S'    83 'S'    73 'I'    79 'O'    78 'N'    95 '_'
0x7fffffffe4a8: 73 'I'    68 'D'    61 '='    49 '1'    0 '\000'  72 'H'    79 'O'    83 'S'
0x7fffffffe4b0: 84 'T'    78 'N'    65 'A'    77 'M'    69 'E'    61 '='    108 'l'   111 'o'
0x7fffffffe4b8: 99 'c'    97 'a'    108 'l'   104 'h'   111 'o'   115 's'   116 't'   46 '.'
```

```
0x7fffffffe4c0: 108 'l'    111 'o'    99 'c'     97 'a'     108 'l'    100 'd'    111 'o'    109 'm'
0x7fffffffe4c8: 97 'a'     105 'i'    110 'n'    0 '\000'   73 'I'     77 'M'     83 'S'     69 'E'
0x7fffffffe4d0: 84 'T'     84 'T'     73 'I'     78 'N'     71 'G'     83 'S'     95 '_'     73 'I'
```

- **缓冲区溢出**

在前面栈帧结构中，我们看到函数的局部变量是保存在堆栈中的，这就可能引入一个潜在的安全问题。如果一个程序提供一个函数使用了局部变量，并将给局部变量作为缓冲区使用——调用者……

寄存器现场

简单说明在 P-Q 函数调用中，一部分寄存器可以由 Q 函数任意修改而不做保护，另一些则需要 Q 函数将寄存器原值保存在堆栈中并在返回前恢复原值。对于 P 函数而言，他在调用 Q 函数前，如果使用了调用者保存的寄存器，则需要自己保存这些寄存器原值，并在 Q 函数返回后由 P 函数从自己的堆栈中恢复这些寄存器原值。

3.4. 浮点运算

3.5. 向量指令

3.5.1. gcc 中文材料

3.6. 小结

练习

1. 请读者自行设计一套不同于 0 小节中的 if-else 模板，并用它描述代码 3-15
2. 请设法跟踪 fibonacci.c 函数中 Fibonacci(n-2)分支中的 n=5 的时候的栈帧，并比较返回地址和书中例子给出的返回地址的不同。