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

函数栈、栈寄存器、栈帧

函数栈：用栈式内存结构，实现函数的顺序调用、嵌套调用。
函数栈，经常被称为线程栈。创建一个线程时，为线程分配一个函数栈。
函数栈，可以放置函数的局部变量。复杂的 struct 变量，如果使用值传递，可以借助函数栈。函数栈的内存操作是内存降序，内存地址从大到小。

栈寄存器：rbp 栈底寄存器，rsp 栈顶寄存器。
入栈，把 rsp 下移，然后在 rsp 地址放入变量。比如，`pushq %r8`，把 r8 的值入栈。
出栈，在 rsp 地址读取变量，然后把 rsp 上移。比如，`popq %r8`，把出栈的值写到 r8。

函数栈的大小限制由操作系统指定。
使用`ulimit -s`查看，单位 KB。

```
[root@192 stack]# ulimit -s
8192
```

使用`ulimit -s num`临时重置，单位 KB。

```
[root@192 stack]# ulimit -s 1024
```

栈帧
调用一个函数时，分配一个栈帧。函数的局部变量，使用栈帧的内存。
栈帧的大小，与局部变量的数量、大小有关。局部变量的数量越多、大小越大，则栈帧相应更大。

栈帧的扩容缩容
编译器在编译时，已经明确某个函数有多少局部变量，需要多少内存，进而把栈帧扩容。扩容需要是 8 的倍数，地址对齐。
扩容与缩容配对出现，保证一个栈帧用完后被正确清理。比如，扩容使用`subq $64, %rsp`，缩容使用`addq $64, %rsp`。扩容使用 sub，因为栈的地址是向下生长的。

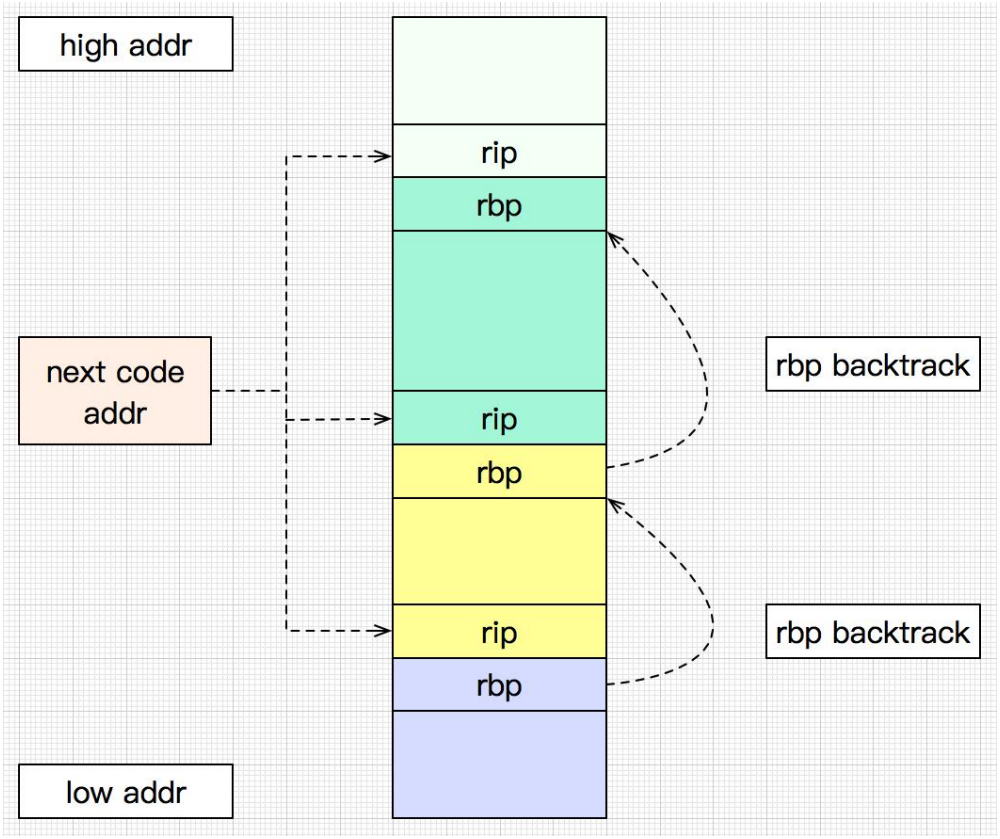
函数栈的常用操作
模拟一个完整的函数调用过程。部分指令可选。

```
callq func_name      调用函数 func_name。入栈，把 rip 压入栈顶。然后把 func_name 的地址，放入 rip。
pushq %rbp           入栈，把 rbp 压入栈顶
movq %rsp, %rbp      把 rsp 的值赋给 rbp。此时 rbp、rsp 指向同一个地址
subq $64, %rsp       扩容，rsp 向下移动 64 个字节
movq %rdi,8(%rsp)    把 rdi 的值，赋给 rsp+8 的地址位置
movq 8(%rsp),%rcx    把 rsp+8 的地址位置的值，赋给 rcx
leaq 16(%rsp),%rbx   把 rsp+16 的地址，赋给 rbx
addq $64, %rsp       缩容，rsp 向上移动 64 个字节
popq %rbp            出栈。把栈顶赋给 rbp
retq                 出栈。把栈顶赋给 rip
```

用汇编分析栈帧的结构

栈有回溯功能。函数开始处，把前一个栈帧的 rbp 入栈。函数退出时，用当前栈帧中记录的 rbp，赋给 rbp 寄存器，回溯到前一个栈帧。
CPU 使用 rip 寄存器管理运行指令的地址。函数调用会打断指令的运行顺序，所以把 rip(下一个指令的地址，即返回地址)入栈。函数退出时，用栈帧记录的返回地址，赋给 rip 寄存器，CPU 就能继续执行后续指令。

栈帧的示意图



编写代码： frame.s

```
.global main

.data
print_main :
    .string "main()      curr_rbp = %#x  curr_rsp = %#x  frame_rbp = %#x  frame_rip = %#x \n"
print_func_a :
    .string "func_a()    curr_rbp = %#x  curr_rsp = %#x  frame_rbp = %#x  frame_rip = %#x \n"
print_func_aa :
    .string "func_aa()   curr_rbp = %#x  curr_rsp = %#x  frame_rbp = %#x  frame_rip = %#x \n"

.text

# 函数  ------------------------------
func_aa :
    pushq %rbp
    movq %rsp, %rbp
```

```
    subq $16, %rsp      # 扩容

    movq $print_func_aa, %rdi
    movq %rbp, %rsi     # 当前的 rbp
    movq %rsp, %rdx     # 当前的 rsp
    movq 0(%rbp), %rcx  # 上一个 rbp
    movq 8(%rbp), %r8   # rip
    callq printf

    addq $16, %rsp      # 缩容

    popq %rbp
    retq                # 退出函数。rip 出栈

# 函数  ----------------------------
func_a :
    pushq %rbp
    movq %rsp, %rbp

    subq $16, %rsp      # 扩容

    movq $print_func_a, %rdi
    movq %rbp, %rsi     # 当前的 rbp
    movq %rsp, %rdx     # 当前的 rsp
    movq 0(%rbp), %rcx  # 上一个 rbp
    movq 8(%rbp), %r8   # rip
    callq printf

    callq func_aa       # 调用函数。rip 入栈

    addq $16, %rsp      # 缩容

    popq %rbp
    retq                # 退出函数。rip 出栈

# 函数  ----------------------------
main :
    pushq %rbp
    movq %rsp, %rbp

    movq $print_main, %rdi
    movq %rbp, %rsi     # 当前的 rbp
    movq %rsp, %rdx     # 当前的 rsp
    movq 0(%rbp), %rcx  # 上一个 rbp
    movq 8(%rbp), %r8   # rip
    callq printf

    callq func_a        # 调用函数。rip 入栈
```

```
    movq $0, %rax     # 返回值

    popq %rbp
    retq                # 退出函数。rip 出栈
```

编译代码：

```
gcc frame.s -o frame
objdump -D frame > frame.dump.txt
```

运行代码：

```
[root@192 stack]# ./frame
main()      curr_rbp = 0x992e2200  curr_rsp = 0x992e2200  frame_rbp = 0  frame_rip = 0x885f5555
func_a()    curr_rbp = 0x992e21f0  curr_rsp = 0x992e21e0  frame_rbp = 0x992e2200  frame_rip = 0x4005a5
func_aa()   curr_rbp = 0x992e21d0  curr_rsp = 0x992e21c0  frame_rbp = 0x992e21f0  frame_rip = 0x40057c
```

分析结果：

函数调用顺序为 `main() > func_a() > func_aa()` 。栈有回溯功能，这里反向分析。

分析 rbp。

func_aa()，栈帧记录的 `frame_rbp = 0x992e21f0` ，等于前一个 func_a()栈帧的 `curr_rbp = 0x992e21f0` 。

func_a()，栈帧记录的 `frame_rbp = 0x992e2200` ，等于前一个 main()栈帧的 `curr_rbp = 0x992e2200` 。

main()，入口函数，栈帧的数据特殊。这里省略。

分析 rip。

分析 rip，需要结合调用函数的代码。查看 frame.dump.txt，这里截取部分代码。

函数	栈帧记录的 rip	调用函数的代码	分析
func_aa()	`frame_rip = 0x40057c`	0000000000400555 <func_a>: 400577: e8 b1 ff ff ff callq 40052d <func_aa> 40057c: 48 83 c4 10 add $0x10,%rsp	`callq func_aa` 的下一个指令为 `addq $16, %rsp` ，其地址为 40057c 。 func_aa()栈帧记录的 rip 为 0x40057c。 两者相等。
func_a()	`frame_rip = 0x4005a5`	0000000000400582 <main>: 4005a0: e8 b0 ff ff ff callq 400555 <func_a> 4005a5: 48 c7 c0 00 00 00 00 mov $0x0,%rax	`callq func_a` 的下一个指令为 `movq $0, %rax` ，其地址为 4005a5。 func_a()栈帧记录的 rip 为 0x4005a5。 两者相等。

顺序调用和嵌套调用的实现

函数调用，是树形结构。程序指令，在内存中是顺序式。顺序式的指令，如何实现树形的函数调用呢？

顺序调用，多个函数依次调用。

```
func_a()
```

```
func_b()
func_c()
```

嵌套调用，一个函数调用一个函数。

```
func_a()
    func_aa()
        func_aaa()
```

混合调用，综合使用顺序调用、嵌套调用。

```
func_a()
    func_aa()
        func_aaa()
    func_ab()
func_b()
    func_bb()
func_c()
```

编写代码： call_func.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

// 函数名的数字后缀，表示函数的调用顺序
// 为了便于讲解，这里没有调用外部函数

void func_1()
{
}

void func_2_1_1()
{
}

void func_2_1_2()
{
}

void func_2_1()
{
    func_2_1_1();
    func_2_1_2();
}

void func_2()
{
    func_2_1();
}

int main()
```

```
{
    func_1();
    func_2();
    return 0;
}
```

编译代码：

```
gcc call_func.c -o call_func
gcc call_func.c -S -o call_func.s
```

调用关系：

从 main 函数开始，把多个函数的调用关系表示出来。

层级 1	层级 2	层级 3	层级 4
main()			
	func_1()		
	func_2()		
		func_2_1()	
			func_2_1_1()
			func_2_1_2()

分析结果：

查看 call_func.c、call_func.s，这里只截取核心代码。由调用关系，分析每个函数的栈帧。

分类	源码	汇编代码	分析

main()	int main()
	{
	    func_1();
	    func_2();
	    return 0;
	}
		main:
		    pushq %rbp
		    movq %rsp, %rbp
		    call func_1
		    call func_2
		    movl $0, %eax
		    popq %rbp
		    ret
			main 依次调用 func_1、func_2。
			返回 0。

func_1()	void func_1()
	{
	}
		func_1:
		    pushq %rbp
		    movq %rsp, %rbp
		    popq %rbp
		    ret
			空函数，依然有栈帧，使用 push、pop 操作栈。

func_2()	void func_2()
	{
	    func_2_1();
	}
		func_2:
		    pushq %rbp
		    movq %rsp, %rbp
		    call func_2_1
		    popq %rbp
		    ret
			func_2 调用 func_2_1。

func_2_1()	void func_2_1()
	{
	    func_2_1_1();
	    func_2_1_2();
	}
		func_2_1:
		    pushq %rbp
		    movq %rsp, %rbp
		    call func_2_1_1
		    call func_2_1_2
		    popq %rbp
			func_2_1 依次调用 func_2_1_1、func_2_1_2。
			没有返回值。

		ret	

func_2_1_1()	void func_2_1_1()
	{
	}
		func_2_1_1:
		pushq %rbp
		movq %rsp, %rbp
		popq %rbp
		ret
			空函数，依然有栈帧，使用 push、pop 操作栈。

func_2_1_2()	void func_2_1_2()
	{
	}
		func_2_1_2:
		pushq %rbp
		movq %rsp, %rbp
		popq %rbp
		ret
			空函数，依然有栈帧，使用 push、pop 操作栈。

分析规律。
不同的函数，如果函数定义相似，汇编代码也相似，函数栈的操作也相似。
main 和 func_2_1，在源码的位置不同，汇编代码相似，都依次调用 2 个函数。
func_1 和 func_2_1_1，在源码的调用层次不同，汇编代码相似，都是空函数，依然有栈帧。

用 C 和汇编分析函数调用的深度

编写代码： depth.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

void func_6()
{
    int tmp = 100;
    printf("func_6  param addr = %p  %llu \n", &tmp, (uint64_t)&tmp);
}

void func_5()
{
    int tmp = 100;
    printf("func_5  param addr = %p  %llu \n", &tmp, (uint64_t)&tmp);
    func_6();
}

void func_4()
{
    int tmp = 100;
    printf("func_4  param addr = %p  %llu \n", &tmp, (uint64_t)&tmp);
    func_5();
}

void func_3()
```

```
{
    int tmp = 100;
    printf("func_3  param addr = %p  %llu \n", &tmp, (uint64_t)&tmp);
    func_4();
}

void func_2()
{
    int tmp = 100;
    printf("func_2  param addr = %p  %llu \n", &tmp, (uint64_t)&tmp);
    func_3();
}

void func_1()
{
    int tmp = 100;
    printf("func_1  param addr = %p  %llu \n", &tmp, (uint64_t)&tmp);
    func_2();
}

int main()
{
    int tmp = 100;
    printf("main    param addr = %p  %llu \n", &tmp, (uint64_t)&tmp);
    func_1();
    return 0;
}
```

编译代码：

```
gcc depth.c -o depth
gcc -S depth.c -o depth.s
```

运行代码：

```
[root@192 stack]# ./depth
main    param addr = 0x7ffeb74ef77c  140731973826428
func_1  param addr = 0x7ffeb74ef75c  140731973826396
func_2  param addr = 0x7ffeb74ef73c  140731973826364
func_3  param addr = 0x7ffeb74ef71c  140731973826332
func_4  param addr = 0x7ffeb74ef6fc  140731973826300
func_5  param addr = 0x7ffeb74ef6dc  140731973826268
func_6  param addr = 0x7ffeb74ef6bc  140731973826236
```

分析结果：

查看 depth.c、depth.s，这里截取核心代码。

函数的调用顺序 `main() > func_1() > func_2() > func_3() > func_4() > func_5() > func_6()` 。

函数	输出	与下一个栈内变量的地址差值
main()	main param addr = 0x7ffeb74ef77c 140731973826428	32
func_1()	func_1 param addr = 0x7ffeb74ef75c 140731973826396	32
func_2()	func_2 param addr = 0x7ffeb74ef73c 140731973826364	32

func_3()	func_3 param addr = 0x7ffeb74ef71c 140731973826332	32
func_4()	func_4 param addr = 0x7ffeb74ef6fc 140731973826300	32
func_5()	func_5 param addr = 0x7ffeb74ef6dc 140731973826268	32
func_6()	func_6 param addr = 0x7ffeb74ef6bc 140731973826236	无

问题：前后 2 个栈内变量的地址差值，为什么都是 32?
首先，这里的函数的结构都相似，进而编译后的函数的汇编代码也相似，函数栈的栈帧逻辑也相似。

```
func_curr()
    int tmp
    printf
    func_next()
```

其次，分析汇编代码 depth.s，以 func_1 调用 func_2 为例。

```
func_1:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $100, -4(%rbp)
    call    func_2
```

分析指令。

```
subq    $16, %rsp    扩容，把当前栈帧扩大 16 字节。
call    func_2       入栈，把 rip 入栈占用 8 字节。
pushq   %rbp         入栈，把 rbp 入栈占用 8 字节。
movl    $100, -4(%rbp)  变量放在栈帧的上部。前后 2 个变量的放置逻辑一样，相对偏移可以省略。
```

计算地址差值=16+8+8=32。
最后，如果扩容为 `subq $64, %rsp` ，则地址差值不是 32。栈帧的大小，影响局部变量的地址生成。

用 C 分析局部变量的地址顺序

全局变量，在数据区，地址升序。
局部变量，在栈区，地址降序。
局部变量，内存地址很高。结合进程的内存布局，可以看出。

编写代码： param_addr.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

// 变量在数据区
int64_t out_tmp5 = 555;
int64_t out_tmp6 = 666;

int main()
{
    // 局部变量。变量在函数栈
    int64_t tmp3 = 333;
```

```
    int64_t tmp4 = 444;
    printf("main 的局部变量 : \n");
    printf("  tmp3  addr = %p  value = %lld \n", &tmp3, tmp3);
    printf("  tmp4  addr = %p  value = %lld \n", &tmp4, tmp4);
    printf("\n");

    printf("全局变量 :  \n");
    printf("  out_tmp5  addr = %p  value = %lld \n", &out_tmp5, out_tmp5);
    printf("  out_tmp6  addr = %p  value = %lld \n", &out_tmp6, out_tmp6);
    printf("\n");

    // 休眠进程。便于查看进程的内存布局
    sleep(9999);
    return 0;
}
```

编译代码：

```
gcc param_addr.c -o param_addr
```

运行代码：

```
[root@192 stack]# ./param_addr
main 的局部变量 :
  tmp3  addr = 0x7fff6e9bd168  value = 333
  tmp4  addr = 0x7fff6e9bd160  value = 444

全局变量 :
  out_tmp5  addr = 0x601050  value = 555
  out_tmp6  addr = 0x601058  value = 666
```

查看内存布局：

```
[root@192 stack]# ps aux | grep param_
root      50684  0.0  0.0   4216   356 pts/3    S+   01:33   0:00 ./param_addr
root      50760  0.0  0.0 112812   980 pts/4    S+   01:34   0:00 grep --color=auto param_
[root@192 stack]# cat /proc/50684/maps
00400000-00401000               r-xp               00000000               08:03               34489132
/root/code/x86-asm/common2/stack/param_addr
00600000-00601000               r--p               00000000               08:03               34489132
/root/code/x86-asm/common2/stack/param_addr
00601000-00602000               rw-p               00001000               08:03               34489132
/root/code/x86-asm/common2/stack/param_addr
7fd2d17eb000-7fd2d19af000 r-xp 00000000 08:03 15928                      /usr/lib64/libc-2.17.so
7fd2d19af000-7fd2d1bae000 ---p 001c4000 08:03 15928                      /usr/lib64/libc-2.17.so
7fd2d1bae000-7fd2d1bb2000 r--p 001c3000 08:03 15928                      /usr/lib64/libc-2.17.so
7fd2d1bb2000-7fd2d1bb4000 rw-p 001c7000 08:03 15928                      /usr/lib64/libc-2.17.so
7fd2d1bb4000-7fd2d1bb9000 rw-p 00000000 00:00 0
7fd2d1bb9000-7fd2d1bdb000 r-xp 00000000 08:03 611075                     /usr/lib64/ld-2.17.so
7fd2d1dcf000-7fd2d1dd2000 rw-p 00000000 00:00 0
7fd2d1dd8000-7fd2d1dda000 rw-p 00000000 00:00 0
7fd2d1dda000-7fd2d1ddb000 r--p 00021000 08:03 611075                     /usr/lib64/ld-2.17.so
```

```
7fd2d1ddb000-7fd2d1ddc000 rw-p 00022000 08:03 611075                     /usr/lib64/ld-2.17.so
7fd2d1ddc000-7fd2d1ddd000 rw-p 00000000 00:00 0
7fff6e99e000-7fff6e9bf000 rw-p 00000000 00:00 0                          [stack]
7fff6e9d8000-7fff6e9da000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

分析结果：
栈内变量的顺序。源码中，tmp3 在 tmp4 的前面。内存地址，tmp3 在 tmp4 的后面，即 0x7fff6e9bd168 大于 0x7fff6e9bd160。
全局变量的顺序。源码中，out_tmp5 在 out_tmp6 的前面。内存地址，out_tmp5 在 out_tmp6 的前面，即 0x601050 小于 0x601058。
栈内变量，地址 0x7fff6e9bd168、0x7fff6e9bd160 在内存布局的栈区，地址很高。
全局变量，地址 0x601050、0x601058 在内存布局的数据区，地址很低。

用 C 分析 struct 属性的地址顺序

一个变量的内部，多个属性的内存地址的顺序和代码顺序是一致的。
如果 struct 包含多个属性，属性在代码的顺序是前后顺序，属性在内存地址的顺序是升序。
如果在栈中设置 2 个 struct 变量，则 2 个变量的顺序和代码顺序相反。但是不会影响变量属性的顺序。

编写代码： struct_addr.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

// struct 包含多个属性
typedef struct
{
    // 使用 64 位，更直观
    int64_t score1;
    int64_t score2;
} study_t;

// 变量在数据区
study_t Jack = {
    .score1 = 100,
    .score2 = 200,
};

int main()
{
    // 变量在函数栈
    study_t Bob;
    Bob.score1 = 500;
    Bob.score2 = 600;
```

```
    printf("Jack 全局变量 : \n");
    printf("  struct  addr = %p   \n", &Jack);
    printf("  score1  addr = %p  value = %lld \n", &Jack.score1, Jack.score1);
    printf("  score2  addr = %p  value = %lld \n", &Jack.score2, Jack.score2);
    printf("\n");

    printf("Bob  局部变量 : \n");
    printf("  struct  addr = %p   \n", &Bob);
    printf("  score1  addr = %p  value = %lld \n", &Bob.score1, Bob.score1);
    printf("  score2  addr = %p  value = %lld \n", &Bob.score2, Bob.score2);
    printf("\n");

    return 0;
}
```

编译代码：

```
gcc struct_addr.c -o struct_addr
```

运行代码：

```
[root@192 stack]# ./struct_addr
Jack 全局变量 :
  struct  addr = 0x601050
  score1  addr = 0x601050  value = 100
  score2  addr = 0x601058  value = 200

Bob  局部变量 :
  struct  addr = 0x7ffe86047640
  score1  addr = 0x7ffe86047640  value = 500
  score2  addr = 0x7ffe86047648  value = 600
```

分析结果：
study_t 有 2 个属性，依次为 score1、score2。
全局变量，属性的内存地址的顺序和代码顺序一样，即 0x601050 小于 0x601058，地址相差 8 个字节。
局部变量，属性的内存地址的顺序和代码顺序一样，即 0x7ffe86047640 小于 0x7ffe86047648，地址相差 8 个字节。

2 个属性的内存地址，对于全局变量、局部变量，都是升序。因为 2 个属性属于同一个变量。
一个变量的内部，多个属性的内存地址的顺序和代码顺序是一致的。

栈大小与栈溢出

出于安全考虑、性能考虑，操作系统给函数栈设置了大小限制，比如，8MB。超过栈大小限制，访问栈内存，触发栈溢出。
单个函数，如果分配非常多的局部变量且不回收，可能触发栈溢出。
嵌套函数，调用层次非常多且不退出，可能触发栈溢出。
用指针遍历栈内存，超过边界，可能触发栈溢出。

把函数栈的最大大小临时调小，方便测出栈溢出。

```
[root@192 stack]# ulimit -s 2048
```

编写代码： max_size.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

// 只看一个函数
int main()
{
    // 获得变量的地址
    int tmp = 0;
    uint64_t init_addr = (uint64_t)&tmp;
    uint64_t tmp_addr = init_addr;

    // 向下遍历栈地址，直到触发栈溢出
    for (int k = 0; k < 900000; ++k)
    {
        // 地址的差值。
        uint64_t offset = init_addr - tmp_addr;

        // 访问指定的内存。可能触发栈溢出。
        int tmp2 = *((int *)tmp_addr);

        printf("tmp_addr = %#llx  %llu   offset = %llu \n", tmp_addr, tmp_addr, offset);

        // 地址下移。
        tmp_addr = tmp_addr - 1024;
    }

    return 0;
}
```

编译代码：

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

运行代码：

输出很多行，这里截取最后几行。

```
[root@192 stack]# ./max_size
// 这里省略很多行
tmp_addr = 0x7ffd73ca3d88  140726546087304   offset = 2083840
tmp_addr = 0x7ffd73ca3988  140726546086280   offset = 2084864
tmp_addr = 0x7ffd73ca3588  140726546085256   offset = 2085888
tmp_addr = 0x7ffd73ca3188  140726546084232   offset = 2086912
Segmentation fault
```

分析结果：
Segmentation fault 表示内存访问错误。栈溢出，导致程序终止。
读写内存操作，校验内存权限， `int tmp2 = *((int *)tmp_addr)` 。
地址差值 2086912，约等于 2048KB。大小不完全相等，因为函数栈的高地址存有当前线程的部分信息，占用部分内存。

用汇编分析省略函数栈操作

问题：函数栈操作 push、pop，可以省略吗？
某些情况可以省略。如果某个函数没有使用栈内存，且方法退出后不影响上下文，可以不使用显式的函数栈操作。
编译器开启优化选项，可能省略函数栈操作。

编写代码： ignore.s

```
.global main

.data

str_num :
    .string " rbx = %lld  rcx = %lld \n"

.text

func_ignore :       # 函数。没有 push、pop

    addq $222, %rbx     # 加法。rbx=rbx+222
    subq $111, %rcx     # 减法。rcx=rcx-111

    retq                # ret 和 call 对应。


main :
    pushq %rbp
    movq %rsp, %rbp

    movq $333, %rbx     # 赋值，把 333 赋给 rbx
    movq $888, %rcx     # 赋值，把 888 赋给 rcx

    callq func_ignore   # 调用函数

    movq $str_num, %rdi
    movq %rbx, %rsi
    movq %rcx, %rdx
    callq printf        # 输出

    popq %rbp
```

```
    retq
```

编译代码：

```
gcc ignore.s -o ignore
```

运行代码：

```
[root@192 stack]# ./ignore
 rbx = 555  rcx = 777
```

分析结果：
计算结果正确。rbx 的值 333+222=555 。rcx 的值 888-111=777 。
函数 func_ignore，没有 push、pop 操作。严格的说，没有显式的 push、pop 操作。callq、retq，把 rip 入栈、出栈，有隐式的 push、pop 操作。

问题：为什么函数 func_ignore 缺少 push、pop，结果依然正确？
对于硬件来讲，使用寄存器、内存进行正确的计算，就能得到正确的结果。
某些汇编指令，如果不影响计算过程，可以省略。