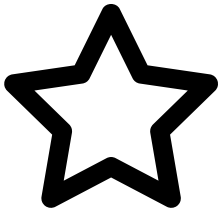
C与汇编语言

原创

展开



孙大圣666 发布于2019-06-19 14:32:28 阅读数 297

收藏

目录

一、汇编指令

汇编指令是指特定CPU架构的指令码的助记符，比如Intel x86 32位下ADD指令对应的16进制机器码有04/05/80/81等，被操作对象不同同一个指令对应的指令码不同，操作对象通常是寄存器，内存地址，I/O端口等，具体操作对象通过指令码后面的一个或者多个的1字节辅助操作码指定，通过objdump命令将生成的可执行程序进行了反汇编可查看汇编指令对应的指令码。

参考：简单学习看机器码的方法

Intel汇编指令格式解析

x86指令格式

二、汇编器和链接器

汇编器（assembler）的作用是将用汇编语言编写的源程序转换成二进制形式的目标代码。Linux 平台的标准汇编器是 GAS， 使用标准的 AT&T 汇编语法，它是 GCC 所依赖的后台汇编工具，通常包含在 binutils 软件包中，对应的命令是as。Linux 平台上另一个经常用到的汇编器是 NASM，与GAS最大的不同是，NASM使用Intel 汇编语法，除此之外NASM和GAS都有自己特定的汇编器指令和宏结构。MSAM是Intel平台上所有汇编器的鼻祖，微软的VS中自带该汇编器，同样使用Intel汇编语法。

汇编器生成的目标代码只局限于单个源代码文件，对源代码中依赖的来自其他源代码文件中的全局变量或者函数未做解析，这时需要**链接器**，将这些未解析的符号翻译成实际的内存地址，最终将存在依赖关系的多个目标代码文件合并成一个可执行文件，GCC使用的链接器是ld。

参考：Java程序员自我修养——编译过程和ELF文件

Linux 汇编器：对比 GAS 和 NASM

三、 汇编语法

汇编语法与汇编器的关系，类似于C/C++标准与C/C++编译器的关系，主要有两种语法，Intel 和AT&T 汇编语法，最大的区别是后者是与平台无关的，前者只局限于Inter平台。特定CPU架构下汇编指令集和寄存器名称相当于汇编语法中的关键字，支持多平台的GAS汇编器会根据当前系统的CPU架构自动识别并转换成对应平台下的机器码。

参考：Linux 汇编语言开发指南

四、C中嵌入汇编代码

C标准中并未规定如何嵌入汇编代码，所以嵌入汇编代码的语法只适用于特定的编译器，下面主要介绍GCC中的内联汇编，其格式为：__asm__ __volatile__("Instruction List" : Output : Input : Clobber/Modify)。

1、__asm__和__volatile__宏

__asm__可简写为asm，用于标识一段汇编指令，__volatile__可简写为volatile，用于告诉编译器不需要对asm中的汇编指令做优化，原封不动的保留即可。

2、Instruction List

Instruction List是指汇编指令列表，可以为空，多条指令时，通常一条指令一行，以分号";"或者换行符"\n"表示结束，并用双引号包住，注意指令中使用寄存器时前面必须加两个%。

3、Output

Output指定汇编语句的输出，即将寄存器中的值存入到哪个变量中，如"=a"(a)，表示将a对应寄存器的值赋值给变量a，a是寄存器eax/ax /al的简写，编译器根据汇编指令自动推测是eax还是ax。

4、Input

Input指定汇编语句的执行参数，可以是一个常量，也可以是一个复杂的表达式，执行参数可以通过占位符或者寄存器的方式传入，占位符按照Input/Output表达式出现的顺序从0开始编号，最多十个表达式，编号最大9。

5、操作数约束

Output表达式"=a"(a) 中a就是操作数约束中的寄存器约束，其他约束如下，其中I O表示Input和Output适用：

- r I,O 表示使用一个通用寄存器，由GCC在%eax/%ax/%al, %ebx/%bx/%bl, %ecx/%cx/%cl, %edx/%dx/%dl中选取一个GCC认为合适的。
- q I,O 表示使用一个通用寄存器，和r的意义相同。
- a I,O 表示使用%eax / %ax / %al
- b I,O 表示使用%ebx / %bx / %bl
- c I,O 表示使用%ecx / %cx / %cl
- d I,O 表示使用%edx / %dx / %dl
- D I,O 表示使用%edi / %di
- S I,O 表示使用%esi / %si
- f I,O 表示使用浮点寄存器
- t I,O 表示使用第一个浮点寄存器
- u I,O 表示使用第二个浮点寄存器
- m I,O 表示使用系统所支持的任何一种内存方式，不需要借助寄存器
- i I 表示输入表达式是一个立即数(整数)，不需要借助任何寄存器
- F I 表示输入表达式是一个立即数(浮点数)，不需要借助任何寄存器
- g I,O 表示可以使用通用寄存器，内存，立即数等任何一种处理方式。

测试用例：

```c
#include <stdio.h>

int main()
{
    int result = 0;
    int input = 1;

    int a = 1;
    int b = 2;

    asm volatile (
        "movl %1, %0\n"
        : "=r"(result)
        : "r"(input)
        );

    printf("result = %d\n", result);
    printf("input = %d\n", input);

    asm volatile (
        "movl %%eax, %%ecx;"
        "movl %%ebx, %%eax\n"
        "movl %%ecx, %%ebx;"
        : "=a"(a), "=b"(b)
        : "a"(4), "b"(5)
        );

    printf("a = %d\n", a);
    printf("b = %d\n", b);

    return 0;
}
```

参考：__asm__ volatile 之 C语言嵌入式汇编

《汇编语言程序设计》

五、GDB反汇编

可通过 gcc -S test.c -o test.s 或者objdump -S test.o > test.s查看生成的汇编代码，但是同源代码的对应关系不够直观而且有很多无关的汇编代码，最理想的是通过gdb反汇编命令disassemble查看每行源代码对应的汇编代码，也可通过ni/si命令单步调试汇编指令，查看执行汇编时寄存器和堆栈的状态。

1、x86_64通用寄存器

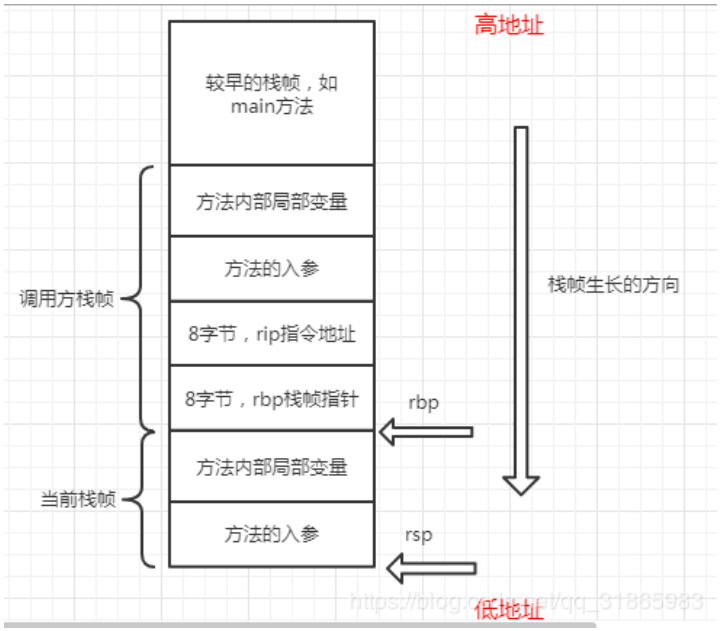
X86-64中，所有寄存器都是64位，相对32位的x86来说，标识符变了，比如从原来的ebp变成了rbp，eax变成了rax。为了向后兼容性，ebp依然可以使用，不过指向了rbp的低32位，即汇编代码中保存到ebp实际是保存到rbp的低32位中。除此之外，新增加寄存器r8到r15，加上x86的原有8个，一共16个通用寄存器，具体用途如下：

- %rax 通常用于存储函数调用的返回结果，同时也用于乘法和除法指令中。
- %rsp 是堆栈指针寄存器，通常会指向栈顶位置
- %rbp 是栈帧指针寄存器，用于标识当前栈帧的起始位置
- %rdi，%rsi，%rdx，%rcx，%r8，%r9 用来传递函数参数，依次对应第1参数，第2参数至第6参数
- %rbx，%r12，%r13，%14，%15 ，%r10，%r11 用作数据存储，属于通用性更为广泛的寄存器，编译器或汇编程序可以根据需要存储任何数据。

2、 调用栈

调用栈按照进程所处的状态分为用户栈和内核栈，这两个栈都是操作系统分配给进程的一片连续内存空间，64位下通常是10k，可通过ulimit -s查看，由操作系统负责维护这两个调用栈的栈顶地址和当前栈帧地址，进程切换时由操作系统负责将当前执行进程的调用栈的栈顶地址放到rbp寄存器中，上一次执行时的当前栈帧地址放到rsp寄存器中，从而使CPU恢复到上一次执行时的调用栈状态。

每次进入到一个新方法，就产生一个新的栈帧，这时需要保存调用方的rbp栈帧指针和方法结束后下一条执行指令的地址，即rip指令寄存器中的指令地址，将这两个压入栈，然后将调用方rsp栈顶指针复制到rbp中，表示栈帧指针往下移动到了原来的栈顶指针处，如果新方法中有调用其他的方法，rsp指针也会从高地址往低地址往下移动足够的空间，方便新方法在该空间中初始化方法内的局部变量，如果不调用其他方法则不需要移动rsp栈顶指针。新方法执行完毕后把rbp保存的调用方的rsp堆栈指针恢复到rsp中，把调用栈中保存的调用方的rbp栈帧指针和rip指令弹出，恢复到rbp寄存器和rip指令寄存器中，即完成了方法调用，如下图：



下面结合测试代码和gdb反汇编代码具体说明。

3、导出汇编代码

测试代码如下，保存为asm2.c:

```
#include <stdio.h>

int outer=12;

int add(int a,int b);

int mult(int a,int b);

int main(){
   int a=123;
   int b=456;
   int c=add(a,a+b);
   int d=234;
   c=mult(c,d);
   printf("c=%d\n",c);
}

int add(int a,int b){
        int c=mult(a,b);
        for(int i=0;i<3;i++){
          c+=outer;
        }
        return c;
```

```
}
int mult(int a,int b){
        return (a+b)*2;
}
```

1、执行sudo gcc -g -std=c99 asm2.c -o asm2.o，生成可执行文件asm2.o，因为for循环中初始化变量i是从C99开始支持的，如果gcc版本较低需要指定C标准版本。

2、执行gdb -q ./asm2.o |tee asm2.s 启动gdb，-q表示不输出gdb的版本信息等，tee命令用于将gdb的输出保存到指定文件asm2.s。

3、执行disassemble main 输出main方法的汇编代码，如下图：

```
[supdev@YZ-222-109-115 gdbTest]$ sudo gdb -q ./asm2.o |sudo tee asm2.s
Reading symbols from /home/openjdk/gdbTest/asm2.o...done.
(gdb) disassemble main
Dump of assembler code for function main:
   0x000000000040052d <+0>:     push   %rbp
   0x000000000040052e <+1>:     mov    %rsp,%rbp
   0x0000000000400531 <+4>:     sub    $0x10,%rsp
   0x0000000000400535 <+8>:     movl   $0x7b,-0x4(%rbp)
   0x000000000040053c <+15>:    movl   $0x1c8,-0x8(%rbp)
   0x0000000000400543 <+22>:    mov    -0x8(%rbp),%eax
   0x0000000000400546 <+25>:    mov    -0x4(%rbp),%edx
   0x0000000000400549 <+28>:    add    %eax,%edx
   0x000000000040054b <+30>:    mov    -0x4(%rbp),%eax
   0x000000000040054e <+33>:    mov    %edx,%esi
   0x0000000000400550 <+35>:    mov    %eax,%edi
   0x0000000000400552 <+37>:    callq  0x40058e <add>
   0x0000000000400557 <+42>:    mov    %eax,-0xc(%rbp)
   0x000000000040055a <+45>:    movl   $0xea,-0x10(%rbp)
```


4、执行disassemble /m main 输出main方法的汇编代码和对应的源码，disassemble 还有一个/r选项，这个是默认值，和不带的效果一样。如下图：

```
(gdb) disassemble /m main
Dump of assembler code for function main:
9       int main(){
   0x000000000040052d <+0>:     push   %rbp
   0x000000000040052e <+1>:     mov    %rsp,%rbp
   0x0000000000400531 <+4>:     sub    $0x10,%rsp

10          int a=123;
   0x0000000000400535 <+8>:     movl   $0x7b,-0x4(%rbp)

11          int b=456;
   0x000000000040053c <+15>:    movl   $0x1c8,-0x8(%rbp)

12          int c=add(a,a+b);
   0x0000000000400543 <+22>:    mov    -0x8(%rbp),%eax
   0x0000000000400546 <+25>:    mov    -0x4(%rbp),%edx
   0x0000000000400549 <+28>:    add    %eax,%edx
   0x000000000040054b <+30>:    mov    -0x4(%rbp),%eax
   0x000000000040054e <+33>:    mov    %edx,%esi
   0x0000000000400550 <+35>:    mov    %eax,%edi
   0x0000000000400552 <+37>:    callq  0x40058e <add>
   0x0000000000400557 <+42>:    mov    %eax,-0xc(%rbp)
```

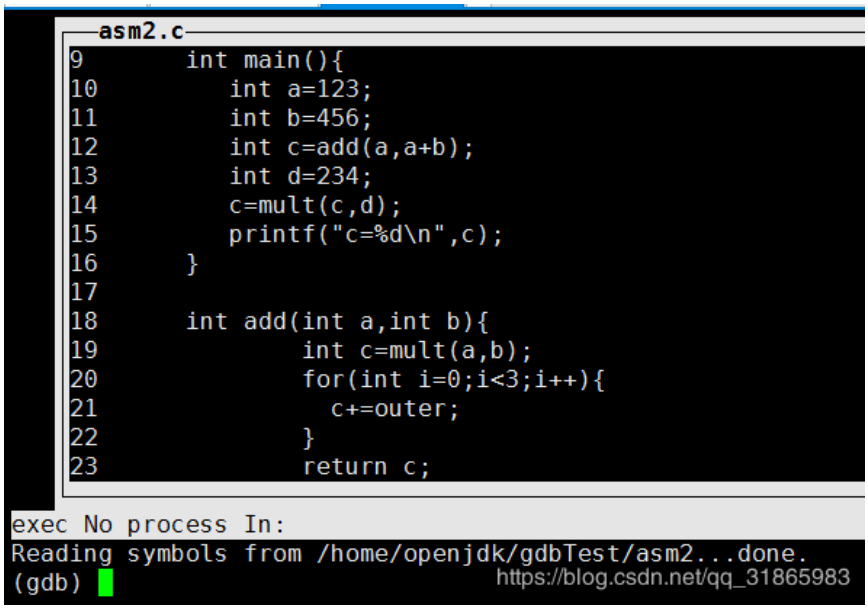

5、适当编辑asm2.s文件，去掉多余的输出，就可以得到可读性很好的汇编代码文件了

4、反汇编调试详解

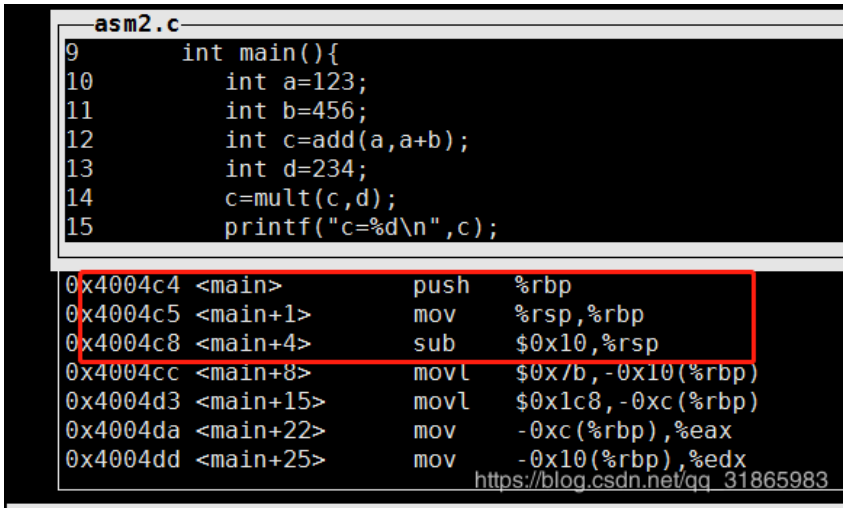
测试代码同上，反汇编调试步骤如下：

1、执行 sudo gcc -g asm2.c -o asm2 -std=c99 生成可执行文件，

2、执行gdb -q -tui ./asm2 启动gdb，-tui表示启动用于显示源码和汇编代码等的文本窗口，也可用gdbtui -q ./asm2替代。



3、执行layout split：显示源代码和汇编窗口，主要是为了查看执行main方法前的汇编指令，因为断点调试只能从int a=123; 这行代码开始，之前的汇编指令在开始断点调试后看不到。



push %rbp，将rbp寄存器保存的栈帧指针压入栈，栈帧指针用于标识当前栈帧的起始位置

mov %rsp,%rbp，将rsp寄存器保存的堆栈指针复制到rbp寄存器，堆栈指针指向栈顶位置，因为栈帧是从上往下，从高地址往低地址生长的，栈顶是指生长方向上的栈顶，所以栈顶地址是整个堆栈的最低地址。

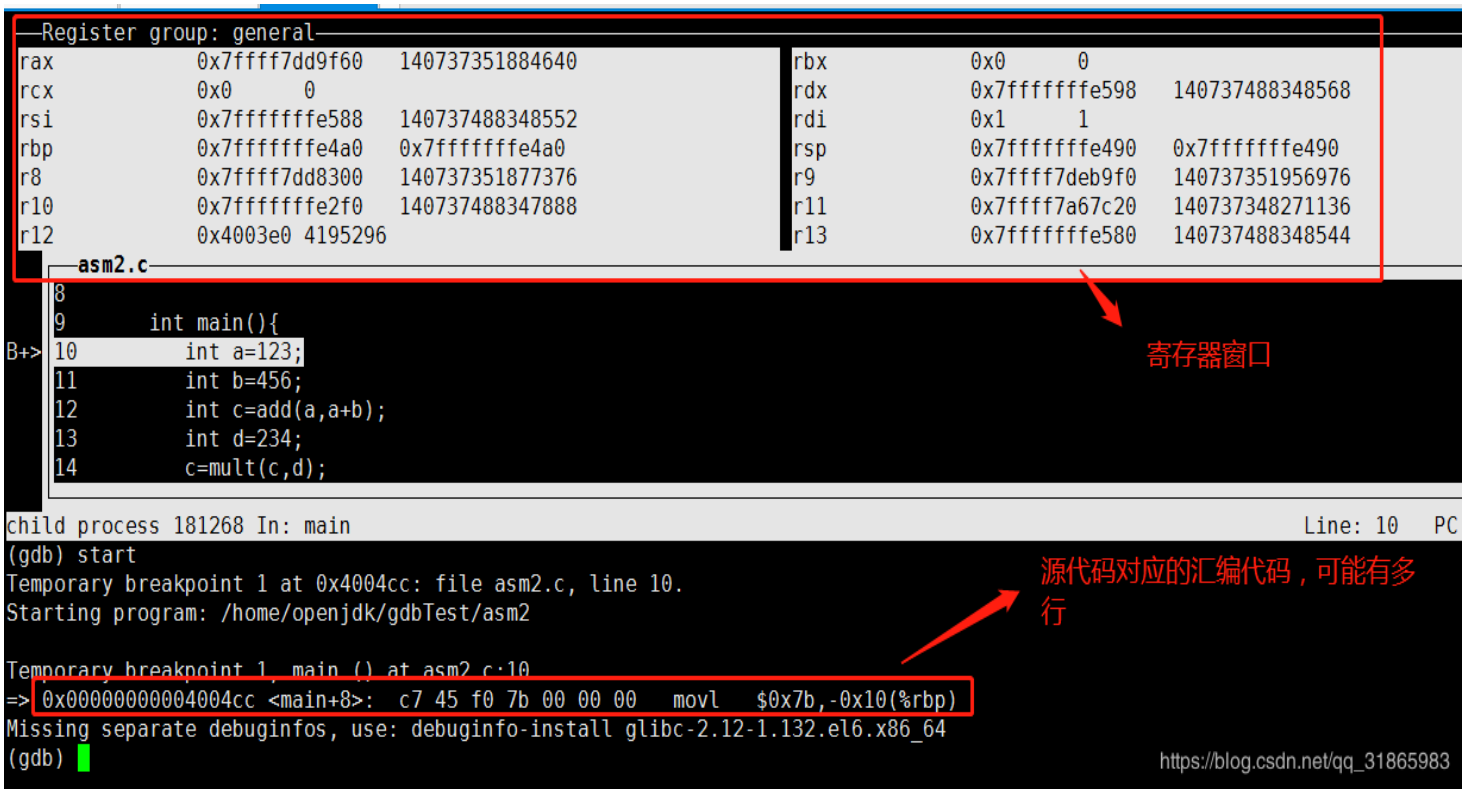
sub $0x10,%rsp，将rsp寄存器中的堆栈指针地址减去16字节

4、执行layout regs，显示寄存器窗口，可直接查看所有寄存器的值，layout src显示源码窗口，查看当前执行的源码，如果寄存器窗口没了，再执行layout regs

5、执行show disassembly-flavor 查看显示的汇编语法类型，默认是att，即AT&T 汇编语法，也可通过set disassembly-flavor intel设置成Intel汇编语法。

6、执行set disassemble-next-line on，当程序暂停时反汇编下一行代码，默认是off

7、执行start， gdb自动停在main方法的第一行代码上，如下图：



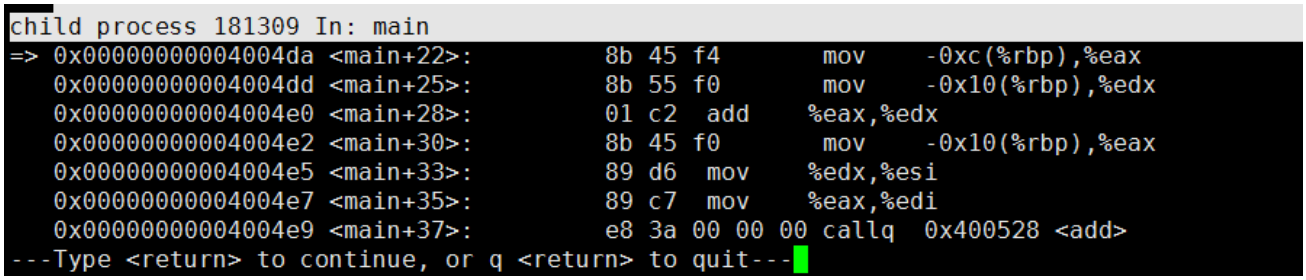
寄存器窗口中，第一列是寄存器的名称，第二列是寄存器中的值的16进制，第三列是对应的10进制整数，rbp和rsp保存的都是内存地址，所以第二列和第三列相同都是16进制形式的，两者相差16字节。

movl $0x7b,-0x10(%rbp)，将堆栈指针减去16字节的起始地址之后的4字节内存的值初始化为0x7b，即123

8、执行ni可查看下一步的汇编代码

movl $0x1c8,-0xc(%rbp) ，将堆栈指针减去12字节的起始地址之后的4字节内存的值初始化为0x1c8，即456

9,、执行到int c=add(a,a+b)；时，汇编代码会显示有多行：



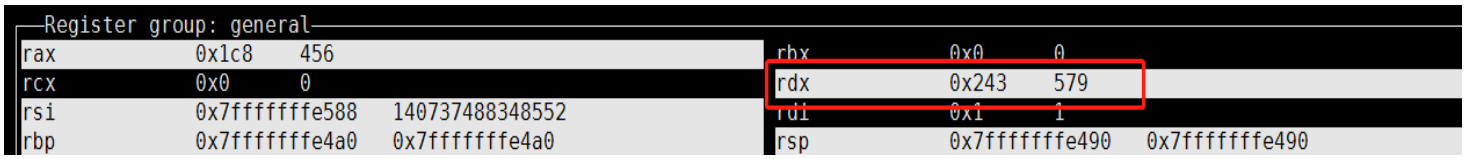
因为多行代码未显示完所以出现最下面的一行---Type <return> to continue, or q <return> to quit---，点击回车往下查看未显示的部分，全部显示完自动退出，也可输入q手动退出，然后执行ni继续下一步调试，当执行到callq的时候可以，执行si可以进入到这个方法的汇编代码执行调试

mov -0xc(%rbp),%eax 因为eax保存32位即4字节的数据，这里将堆栈指针减去8个字节的起始地址往后的4个字节的数据，即456复制到eax寄存器中，64位下实际是rax，汇编代码使用eax主要是告诉CPU操作的字节数

mov -0x10(%rbp),%edx，将堆栈指针减去16个字节的起始地址往后的4个字节的数据，即123复制到edx寄存器中，64位下实际是rdx。

此时可通过info registers查看当前所有寄存器的值，也可通过寄存器窗口查看：

add %eax,%edx，将eax中的值加到edx中的值上，结果保存到edx中，64位下是rdx中，如下：

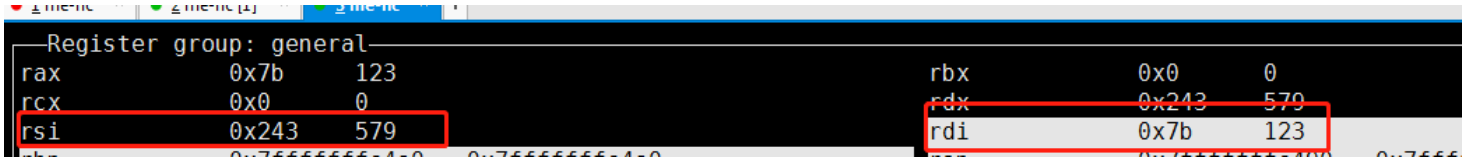


mov -0x10(%rbp),%eax 将堆栈指针减去16个字节的起始地址往后的4个字节的数据，即123复制到eax寄存器中，实际是rax

mov %edx,%esi 将edx的值即579复制到esi寄存器，实际是rsi寄存器，保存函数的第二个参数

mov %eax,%edi 将eax的值即123复制到edi寄存器，实际是rdi寄存器，保存函数的第一个参数

此时寄存器状态：



10、注意此时rbp的变化，进入前rbp是0x7fffffffe4a0，rsp是0x7fffffffe490，执行si 进入到add方法的汇编代码调试，进入后rbp的值并未改变，rsp变成0x7fffffffe488，即往下移动了8个字节，这是call指令保存的方法调用结束后下一条执行的指令的地址，即保存rip指令寄存器中的数据。

push %rbp 把rbp的栈帧指针压入栈中，rbp不变，rsp变成0x7fffffffe480，即又往下移动了8字节，此时栈顶指针保存在rsp上方的8个字节中，此时rbp下方的8字节还是空白的

mov %rsp,%rbp 把rsp的栈顶指针复制到rbp中，即栈帧指针往下移动到原来的栈顶指针处，此时rbp变成0x7fffffffe480

sub $0x20,%rsp 把rsp的栈顶指针减去20字节，即栈顶指针往下移动32个字节，此时rsp变成0x7fffffffe460

mov %edi,-0x14(%rbp) 把edi中的函数参数拷贝到rbp栈帧指针减去20个字节的起始地址之后的4个字节，即123

mov %esi,-0x18(%rbp) 把esi中的函数参数拷贝到rbp栈帧指针减去24个字节的起始地址之后的4个字节，即579

mov -0x18(%rbp),%edx 把rbp栈帧指针减去24个字节的起始地址之后的4个字节的数据拷贝到edx，64位下是rdx

mov -0x14(%rbp),%eax 把rbp栈帧指针减去20个字节的起始地址之后的4个字节的数据拷贝到eax，64位下是rax

mov %edx,%esi 把edx寄存器中的数据拷贝到esi中，作为函数的第二个参数

mov %eax,%edi 把eax寄存器中的数据拷贝到edi中，作为函数的第一个参数

上述四条指令实际是无意义的，如果开启编译器优化，这四条指令可能就没了。

11、执行si 进入到mult方法的汇编代码调试，rbp不变，rsp变成0x7fffffffe458，即往下移动了8个字节，用于保存方法调用结束后下一条执行指令的地址。

push %rbp，把rbp的栈帧指针压入栈中，rbp不变，rsp变成0x7fffffffe450，即往下移动了8个字节，用于保存rbp栈帧指针

mov %rsp,%rbp 把rsp的栈顶指针拷贝到rbp中，rbp和rsp都变成0x7fffffffe450

mov %edi,-0x4(%rbp) 把edi的值拷贝到rbp栈帧指针减去4字节的起始地址之后的4字节，即123

mov %esi,-0x8(%rbp) 把edi的值拷贝到rbp栈帧指针减去8字节的起始地址之后的4字节，即579

mov -0x8(%rbp),%eax 把rbp栈帧指针减去8字节的起始地址之后的4字节的数据拷贝到eax中，即579

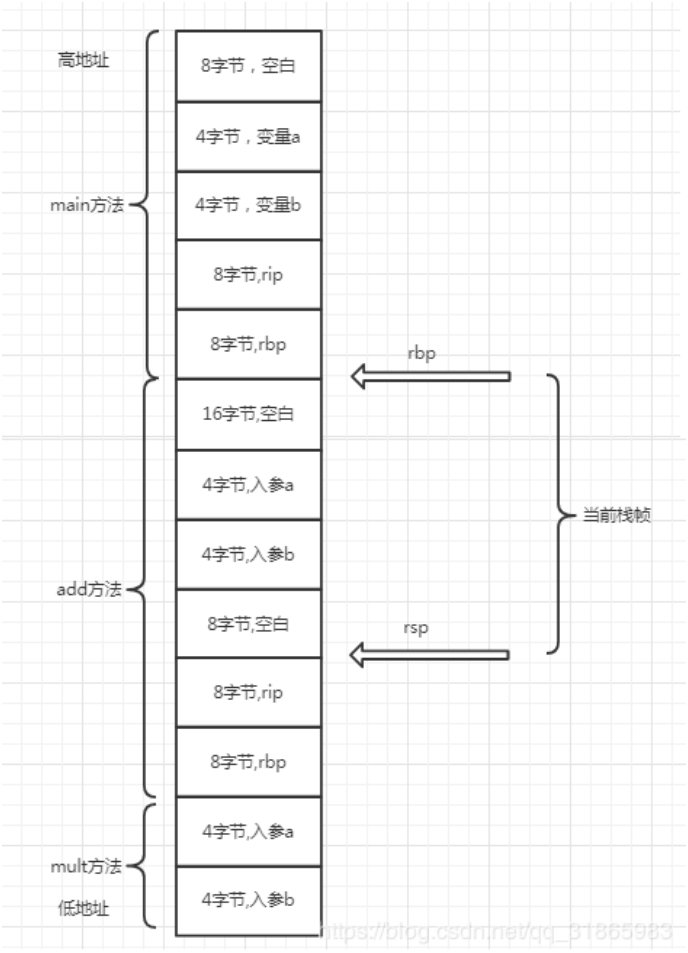
mov -0x4(%rbp),%edx 把rbp栈帧指针减去4字节的起始地址之后的4字节的数据拷贝到edx中，即123

lea (%rdx,%rax,1),%eax，把rdx的值加上rax的值，结果放到eax中

add %eax,%eax 把eax的值加上eax的值

leaveq 相当于两条指令mov %rbp, %rsp和pop %rbp，即把rbp的值复制到rsp中，即0x7fffffffe450，然后把rbp的地址弹出放到rbp中，rsp自动加上保存rbp的8个字节，rbp的值为0x7fffffffe480，rsp的值为0x7fffffffe458

retq 相当于popq %rip，即把调用方法结束后下一条执行的指令的地址弹出并放到rip寄存中，rsp自动加上保存rip的8字节，rsp的值变成0x7fffffffe460，rbp变成0x7fffffffe480，即恢复到调用mult方法前的状态，在mult方法执行过程中分配的两个4字节可以被重新分配给其他方法，相当于自动释放掉了。注意，因为mult方法是最下面的一个栈帧，所以没有往下移动rsp栈顶指针，方法执行期间rsp和rbp指针是一样的，此时栈帧状态如下：

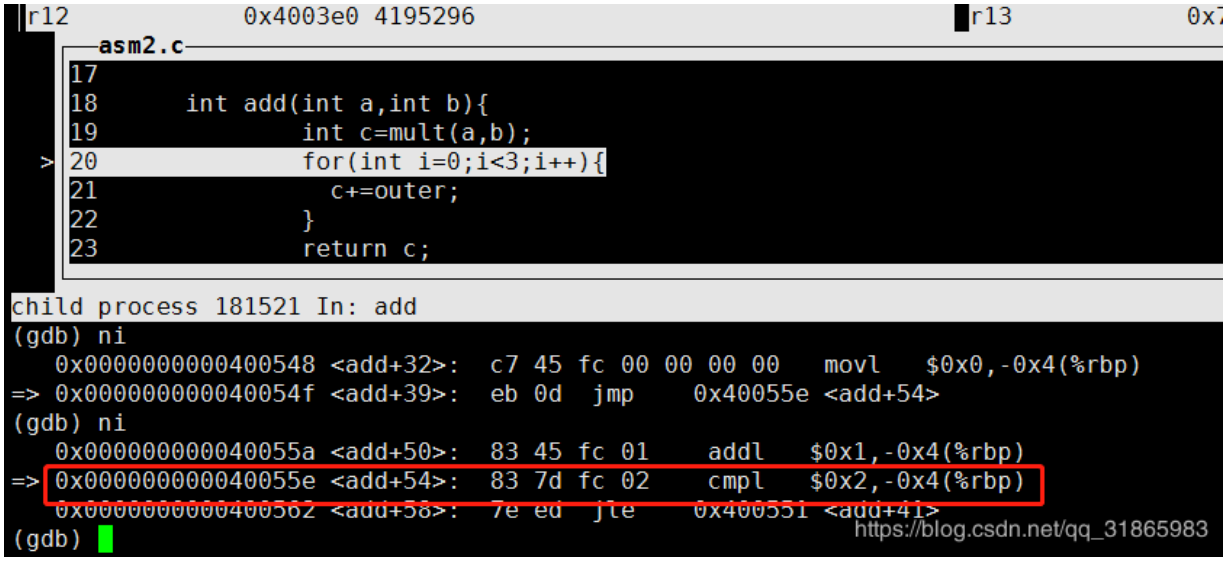


12、mult方法执行完成，继续ni

mov %eax,-0x8(%rbp) 把mult方法调用结果从eax拷贝到rbp栈帧指针减去8个字节的起始地址之后的4字节中，即1404

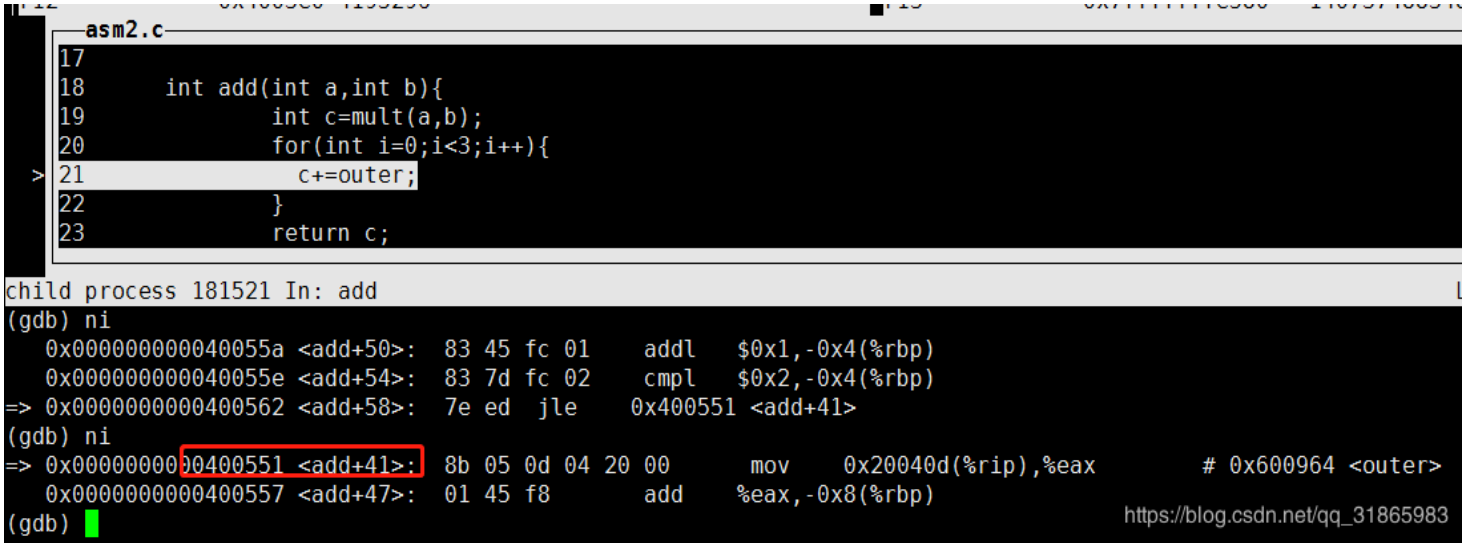
movl $0x0,-0x4(%rbp) 把rbp栈帧指针减去4个字节的起始地址之后的4字节内存初始化为0，即初始化for循环的变量i

jmp 0x40055e 跳转到执行i<3比较的指令，如下图：



cmpl $0x2,-0x4(%rbp) cmp比较指令，实际是减法，用2减去rbp栈帧指针减去4字节的起始地址之后的4字节数据，即变量i，只是结果不保存，只影响CPU内部的标志位，其他指令根据标志位判断比较结果

jle 0x400551 **jle**是转移跳转指令，如果cmpl指令返回true，则跳转到0x400551处的指令，否则继续执行下一条指令，如下图：



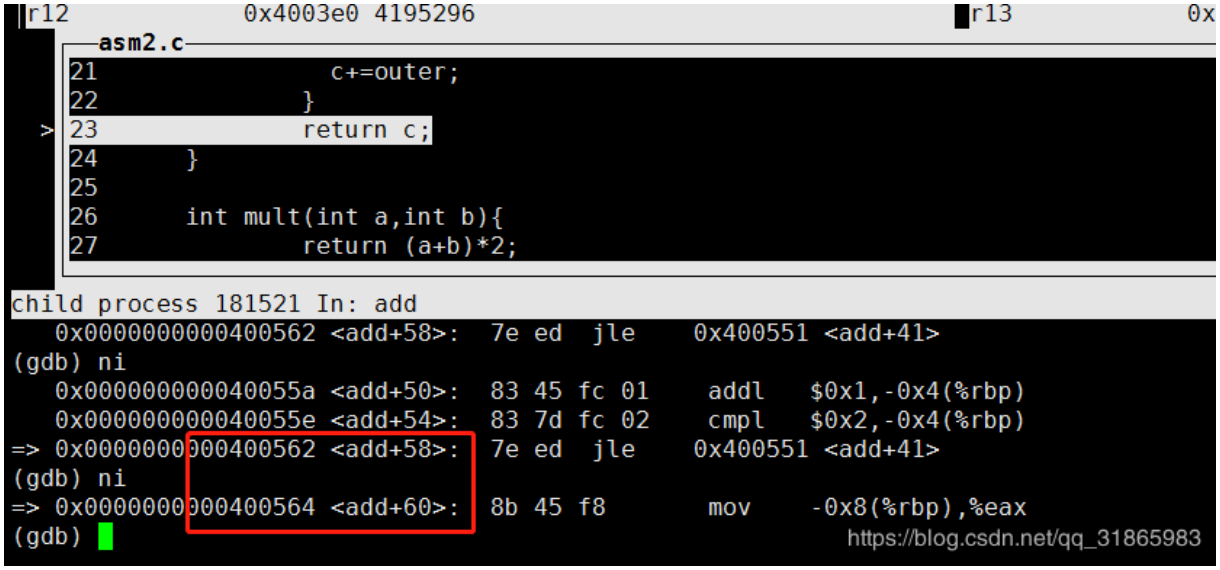
mov 0x20040d(%rip),%eax 把rip指令地址加上0x20040d字节的起始地址之后的4字节数据拷贝到eax寄存器，即全局变量outer拷贝到eax中，全局变量是在单独的.data段中保存，在程序加载时初始化，这里使用相对地址获取该变量的值。

add %eax,-0x8(%rbp) 把rbp栈帧地址减去8个字节的起始地址之后的4字节的数据同eax中的值相加，即c变量加上outer，结果保存到c变量中。

addl $0x1,-0x4(%rbp) 把变量i加上1

cmpl $0x2,-0x4(%rbp) 比较变量i和2

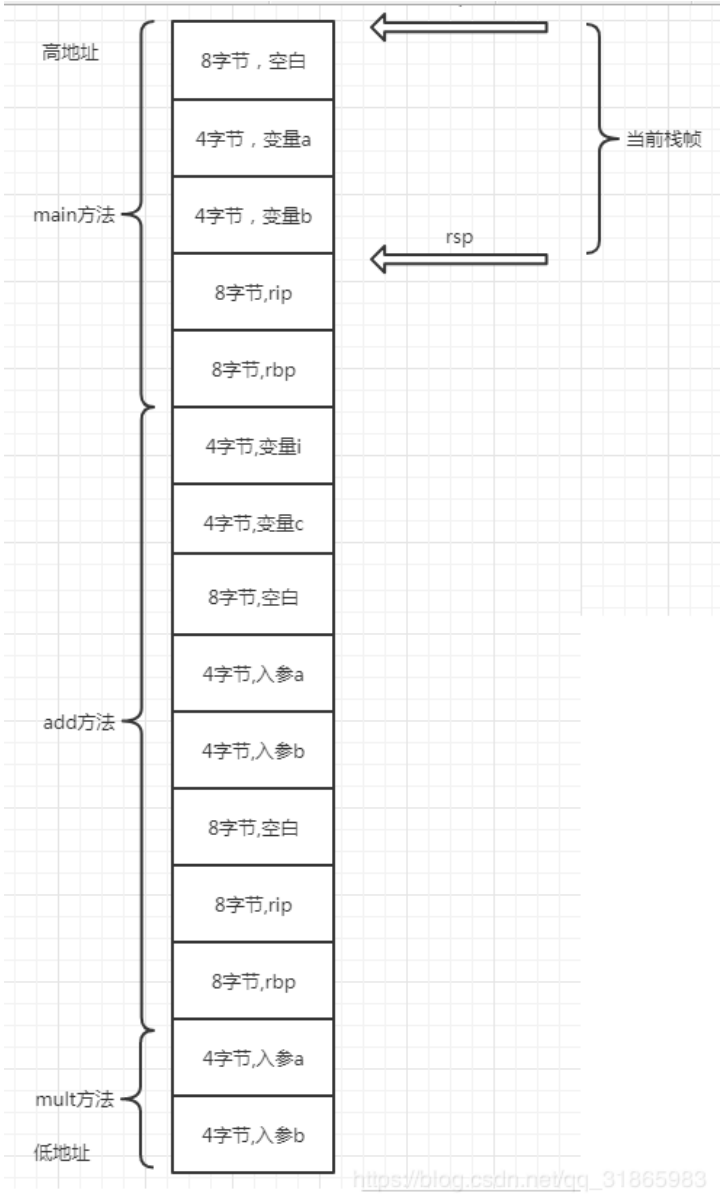
jle 0x400551 跳转到c+=outer，后面的就是重复的，直到cmpl比较返回-1，jre就跳到jre下面的即for循环结束后的一条指令，如下图：



mov -0x8(%rbp),%eax 将变量c的值拷贝到eax中

leaveq 同上，恢复rbp

retq 同上，恢复rip，至此add 方法执行完毕，此时栈帧状态如下：



13、add方法执行完成，继续ni：

mov %eax,-0x8(%rbp) 把eax中的值拷贝到rbp栈帧指针减去8字节的起始地址之后的4字节，即变量c的初始化

movl $0xea,-0x4(%rbp) 把rbp栈帧指针减去4字节的起始地址之后的4字节初始化为0xea，即变量d初始化

mov -0x4(%rbp),%edx 把变量d复制到edx中

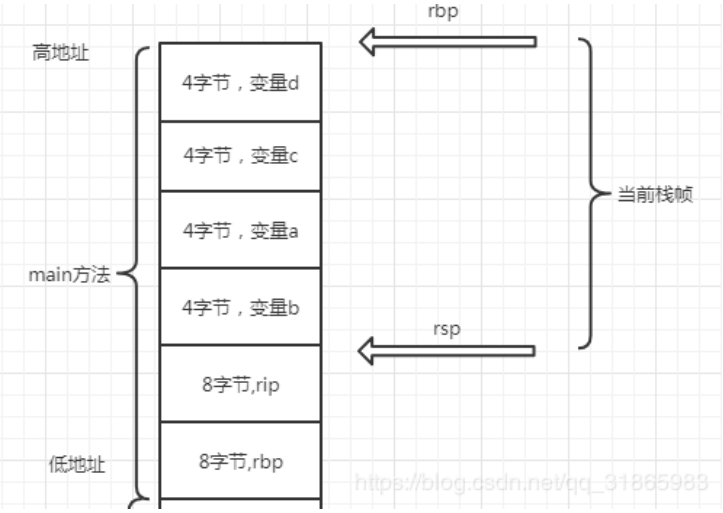
mov -0x8(%rbp),%eax 把变量c复制到eax中

mov %edx,%esi 把edx中的值复制到esi中，作为函数的第二个参数

mov %eax,%edi 把eax中的值复制到edi中，作为函数的第一个参数

callq 0x400569 <mult> 调用mult方法，执行指令同add方法调用mult方法，因为mult方法未调用其他方法，所以执行过程中rbp和rsp一样，rsp未向下移动，直接使用rsp下面的两个4字节储存变量。

此时main方法的栈帧状态如下：



14、mult方法执行完成，继续ni

mov %eax,-0x8(%rbp) 将mult方法的调用结果保存到原来的变量c中

mov $0x400678,%eax 把0x400678拷贝到eax中，0x400678应该是字符串"c=%d\n"指针的地址

mov -0x8(%rbp),%edx 把变量c拷贝到edx中

mov %edx,%esi 把edx中的值拷贝到esi中，作为函数的第二参数

mov %rax,%rdi 把rax中的值拷贝到rdi中，作为函数的第一参数

mov $0x0,%eax 用0初始化eax

callq 0x4003b8 <printf@plt> 调用printf方法，printf方法是标准库函数，看不到源码，而且涉及系统调用相关，整体流程比较复杂，不展开了。

mov $0x0,%eax 用0初始化eax，0是返回值

leaveq 同上，恢复rbp

retq 同上，恢复rip，至此main方法结束，返回0，进程自动退出。

5、结构体反汇编

上面讲的是普通变量操作的反汇编，普通变量或者指针都可以通过寄存器传递，但是结构体如何传递了？结构体是如何通过汇编初始化了？

测试代码如下：

```
#include <stdio.h>

struct user{
        int age;
        char name[10];
};

typedef struct user user;

user change(user user);

int main ()
{
   user test={14,"test"};
   user a=change(test);
   int age=a.age+1;
```

```
    printf("age:%d",age);
                            }

user change(user user){
        user.age=12;
        return user;
}
```

结构体初始化的指令如下：

movq $0x0,-0x20(%rbp) 将离rbp栈帧指针32字节的起始地址之后的8字节初始化为0

movq $0x0,-0x18(%rbp) 将离rbp栈帧指针24字节的起始地址之后的8字节初始化为0

movl $0xe,-0x20(%rbp) 将离rbp栈帧指针32字节的起始地址之后的4字节初始化为14，即结构体test中age属性的初始化

movq $0x74736574,-0x1c(%rbp) 将离rbp栈帧指针28字节的起始地址之后的8字节初始化为$0x74736574，这个是字符串test在静态存储区的地址，即结构体中name属性的初始化

movw $0x0,-0x14(%rbp) 将离rbp栈帧指针20字节的起始地址之后的2字节初始化为0

进入函数调用前的指令如下：

mov -0x20(%rbp),%rdx 将离rbp栈帧指针32字节的起始地址之后的8字节拷贝到rdx中

mov -0x18(%rbp),%rax 将离rbp栈帧指针24字节的起始地址之后的8字节拷贝到rax中

mov %rdx,%rdi 将rdx寄存器的数据拷贝到rdi中

mov %rax,%rsi 将rax寄存器的数据拷贝到rsi中

进入到change方法的指令如下：

mov %rdi,%rdx 将rdi中的数据拷贝到rdx中

mov %rsi,%rax 将rsi中的数据拷贝到rax中

mov %rdx,-0x20(%rbp) 将rdx中的数据拷贝到离rbp栈帧指针32字节的起始地址之后的8字节中

mov %rax,-0x18(%rbp) 将rax中的数据拷贝到离rbp栈帧指针24字节的起始地址之后的8字节中

movl $0xc,-0x20(%rbp) 将rdx中的数据拷贝到离rbp栈帧指针32字节的起始地址之后的4字节赋值为12

change方法准备返回的指令如下：

mov -0x20(%rbp),%rax

mov %rax,-0x10(%rbp) 将离rbp栈帧指针32字节的起始地址之后的8字节拷贝到离rbp栈帧指针16字节的起始地址之后的8字节

mov -0x18(%rbp),%rax

mov %rax,-0x8(%rbp) 将离rbp栈帧指针24字节的起始地址之后的8字节拷贝到离rbp栈帧指针8字节的起始地址之后的8字节

mov -0x10(%rbp),%rax 将离rbp栈帧指针16字节的起始地址之后的8字节拷贝到rax中

mov -0x8(%rbp),%rdx 将离rbp栈帧指针8字节的起始地址之后的8字节拷贝到rdx中

方法返回以后执行的指令如下：

mov %rax,%rcx

mov %rdx,%rax

mov %rcx,-0x40(%rbp)

mov %rax,-0x38(%rbp) 上述指令将change方法返回的数据拷贝到离rbp栈帧指针64字节的起始地址之后的16字节中

mov -0x40(%rbp),%rax

mov %rax,-0x30(%rbp)

mov -0x38(%rbp),%rax

mov %rax,-0x28(%rbp) 上述指令将离rbp栈帧指针64字节的起始地址之后的16字节拷贝到离rbp栈帧指针48字节的起始地址之后的16字节，即用change方法的返回值重新初始化了变量a

mov -0x30(%rbp),%eax

add $0x1,%eax

mov %eax,-0x4(%rbp) 上述指令完成a.age+1,并把结果保存到离rbp栈帧指针4字节的起始地址之后的4字节。

至此结构体变量的初始化和参数传递分析完成，从中可以得出结论，传递结构体时实际是将结构体对应的内存以8字节为单位拷贝到寄存器中，将寄存器中的数据拷贝到内存即完成对应结构体的初始化。如果结构体占用的内存超过可用寄存器允许传递的最大字节数怎么办了？把上述结构体中char数组的长度改成200，继续测试。汇编指令比较长，尤其是rep指令不好调试，可通过gcc -S生成汇编代码查看。

其中传递结构体的核心指令如下：

leaq -432(%rbp), %rbx
movq %rsp, %rdx
leaq -224(%rbp), %rax
movl $25, %ecx
movq %rdx, %rdi
movq %rax, %rsi
rep movsq 上述指令将距离rbp栈帧指针224字节的起始地址之后的25*8=200字节拷贝到rsp栈顶指针后面的200字节中，拷贝结束，rdi指向拷贝的最后一个字节，rsi指向离rbp栈帧指针24字节处

movq %rsi, %rax
movq %rdi, %rdx
movl (%rax), %ecx
movl %ecx, (%rdx) 上述指令将离rbp栈帧指针24字节处后的4字节拷贝到200字节的后面，至此结构体的204字节都已拷贝完成。

所以当结构体的大小超过寄存器传递允许的最大字节数时就直接从当前栈帧的内存拷贝至调用方法的栈帧内。
参考：GDB调试之TUI界面

GDB 单步调试汇编

disassemble command