

Linux 桌面玩家指南：08. 使用 GCC 和 GNU Binutils 编写能在 x86 实模式运行的 1...

阅读目录

- [前言 ##](#)
- [GCC 和 GNU Binutils 的 16 位代码之旅 ##](#)
- [总结 ##](#)
- [版权申明 ##](#)

特别说明：要在我的随笔后写评论的小伙伴们注意了，我的博客开启了 MathJax 数学公式支持，MathJax 使用 $\$$ 标记数学公式的开始和结束。如果某条评论中出现了两个 $\$$ ，MathJax 会将两个 $\$$ 之间的内容按照数学公式进行排版，从而导致评论区格式混乱。如果大家的评论中用到了 $\$$ ，但是又不是为了使用数学公式，就请使用 $\$$ 转义一下，谢谢。

想从头阅读该系列吗？下面是传送门：

- [Linux 桌面玩家指南：01. 玩转 Linux 系统的方法论](#)
- [Linux 桌面玩家指南：02. 以最简洁的方式打造实用的 Vim 环境](#)
- [Linux 桌面玩家指南：03. 针对 Gnome 3 的 Linux 桌面进行美化](#)
- [Linux 桌面玩家指南：04. Linux 桌面系统字体配置要略](#)
- [Linux 桌面玩家指南：05. 发博客必备的图片处理和](#)

[视频录制脚本](#)

- [Linux 桌面玩家指南：06. 优雅地使用命令行及 Bash 脚本编程语言中的美学与哲学](#)
- [Linux 桌面玩家指南：07. Linux 中的 Qemu、KVM、VirtualBox、Xen 虚拟机体验](#)

[回到顶部](#)

前言

这是一个非常复古的话题，希望能够吸引到你们。x86 实模式、16 位的代码、汇编语言，这些古老的东西还有谁会关注呢？只能是浪漫的程序员。网上传说，程序员有三大浪漫：操作系统、编译原理和计算机图形学。在如此浪漫情怀的支持下，我也曾尝试写一个自己的操作系统。后来呢？当然是从入门到放弃啦。不过在 x86 实模式下编写 16 位的引导扇区，仍然给我留下了深刻的记忆。而且，在自己写操作系统的过程中，我一直强烈希望能早一点脱离汇编语言，早一点进入 C 语言的世界。因此，有了这篇随笔。不可否认，这次的标题有点长。之所以把标题写得这么详细，主要是为了搜索引擎能够准确地把握确实需要了解 GCC 生成 16 位实模式代码方法的朋友带到我的博客。

要运行 x86 实模式的程序，目前我知道的只有两种方式，一种是使用 DOS 系统，另一种是把它写成引导扇区的代码，在系统启动时直接运行。很显然，许多讲自己实现操作系统的书籍都会讲到 x86 实模式，也只有自己实现操作系统引导的朋友需要用到 x86 实模式，所以我这篇随笔的阅读用户数肯定很少，虽然我自认为它填补了网上关于该话题相关资料缺乏的空白。因此，凡是逛到我这篇文章的朋友，请点一下推荐，谢谢。

为什么说这篇博客填补了相关话题的空白呢？那是因为不管是那些写书的，还是网上写文章的，一旦需要编写 16 位的实模式代码，都喜欢拿 NASM 说事儿，一点也不顾 GNU AS 的感受。当然，这是有历史原因的，因为 Linux 自从其诞生起就是 32 位、就是多用户多任务操作系统，所以 GCC 和

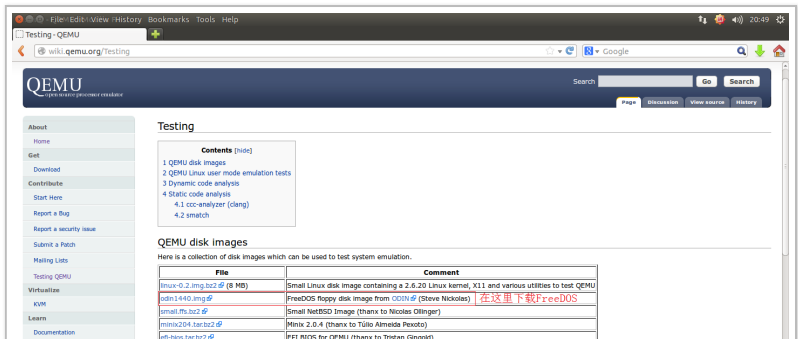
Gnu AS 一移植到 Linux 上就是用来编写 32 位保护模式的代码的。而且，ELF 可执行文件格式也只有 ELF32 和 ELF64，没听说过有 ELF16 的。即使是 Linux 自己，刚诞生的时候（1991 年），也只有使用 as86 汇编器来编写自己的 16 位启动代码，直到 1995 年以后，GNU AS 才逐步加入编写 16 位代码的能力。

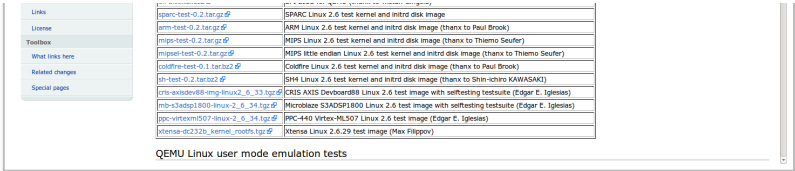
按照惯例，我当然是要给大家安利参考资料的啦。首先，是 [GNU C 语言手册](#)，C 语言是 Linux 操作系统的母语，也是我接触计算机后学习的第一门语言，所以是有非常深厚的感情的。而且，C 语言非常简洁。还记得我前面给大家推荐的 Bash 手册吗？我说它只有 171 页，算是非常简短的了。我这里给大家推荐的 [GNU C 语言手册](#)，加上封面、目录和索引，总共也才 91 页。有 PDF 版本，下载到手机上慢慢看吧。其次，是 AT&T 汇编语言的资料，以及 Binutils 的资料，这个只能上 [Gnu Binutils 官网](#) 上看它的官方文档了，而且没有 PDF 版，没有那种从头读到尾的畅快感。最后，我送大家一本 [Professional Assembly Language](#) 和一本 [Programming from the ground up](#)，希望大家学得愉快。

[回到顶部](#)

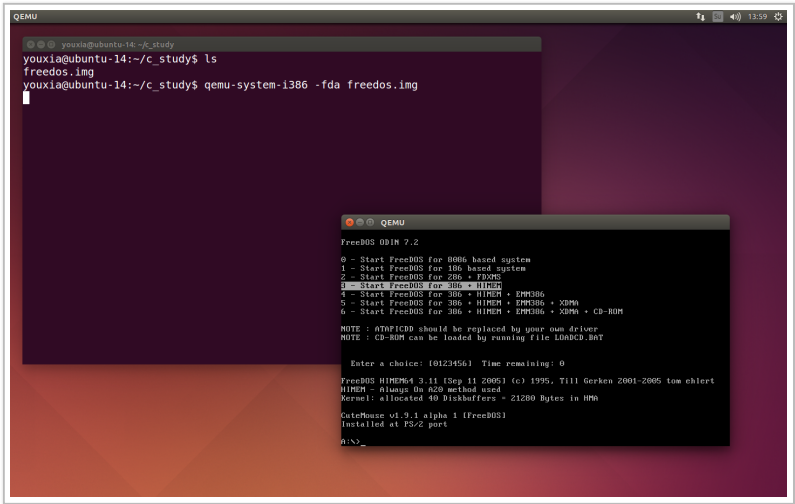
GCC 和 GNU Binutils 的 16 位代码之旅

我决定使用 DOS 作为我的测试环境，所以最后生成的可执行文件都把它制作成 DOS 系统中可运行的 Plain Binary 格式。第一步安装一个 Qemu 虚拟机来运行 FreeDOS，安装虚拟机在 Ubuntu 中只需要一个 `sudo aptitude install qemu` 命令就可以完成，所以我不截图了。但是 FreeDOS 的软盘镜像文件需要到 Qemu 的官网上面去下载，下载地址如下图：

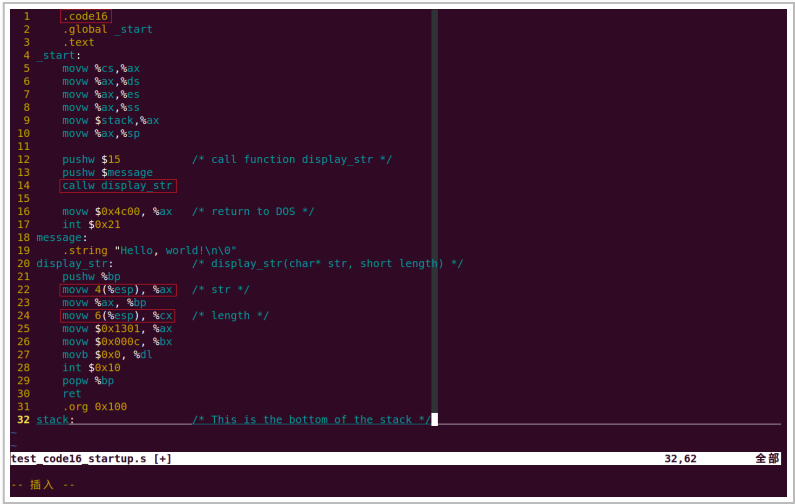




使用 `qemu-system-i386 -fda freedos.img` 可以运行 Qemu 虚拟机和 FreeDOS 系统，如下图：



因为汇编语言更接近底层，而 C 语言更高级，所以先从汇编语言开始，逐步过渡到 C 语言。先写一个简单的、能在 DOS 中显示一个“Hello, world!”的汇编语言程序，考虑到我之后会使用该程序调用 C 语言的 `main` 函数，并且该程序负责让程序运行结束后顺利返回 DOS 系统，所以我把这个程序命名为 `test_code16_startup.s`。其代码如下：



下面对以上代码进行简单解释：

1、GNU AS 汇编器使用的汇编语言采用的是 AT&T 语法，该语法和 Intel 语法不同。我更喜欢 AT&T 的语法，原因有两

个，一是 AT&T 语法是 Linux 世界中通用的标准，二是 AT&T 语法在某些概念方面确实理解起来更简单（比如内存寻址模式）。有汇编语言基础的人，AT&T 语法学起来也很快，主要有以下几条：①汇编指令后面跟有操作数长度的后缀，比如 `mov` 指令，如果操作数是 8 位，则用 `movb`，如果操作数是 16 位，则用 `movw`，如果操作数是 32 位，则用 `movl`，如果操作数是 64 位，则用 `movq`，其余指令依此类推；②操作数的顺序是源操作数在前，目标操作数在后，比如 `movw %cs, %ax` 表示把 `cs` 寄存器中的数据移动到 `ax` 寄存器中，这个顺序和 Intel 汇编语法正好相反；③所有的寄存器使用 `%` 前缀，如 `%ax`，`%di`，`%esp` 等；④对于立即数，需要使用 `$` 前缀，比如 `$4`，`$0x0c`，而且如果一个数字是以 0 开头，则是 8 进制，以其它数字开头，是 10 进制，以 `0x` 开头则是 16 进制，标号当立即数使用时，需要 `$` 前缀，比如上面的 `pushw $message`，而标号当函数名使用时，不需要 `$` 前缀，比如上面的 `callw display_str`；⑤内存寻址方式，众所周知，x86 寻址方式众多，什么直接寻址、间接寻址、基址寻址、基址变址寻址等等让人眼花缭乱，而 AT&T 语法对内存寻址方式做了一个很好的统一，其格式为

`section:displacement(base, index, scale)`，其中 `section` 是段地址，`displacement` 是位移，`base` 是基址寄存器，`index` 是索引，`scale` 是缩放因子，其计算方式为 `线性地址 = section + displacement + base + index*scale`，最重要的是，可以省略以上格式中的一个或多个部分，比如 `movw 4, %ax` 就是把内存地址 4 中的值移动到 `ax` 寄存器中，`movw 4(%esp), %ax` 就是把 `esp+4` 指向的地址中的值移动到 `ax` 寄存器中，依此类推。我上面的介绍是不是全网络最简明的 AT&T 汇编语法教程？

2、在以上代码中我全部使用的都是 16 位的指令，如

`movw`、`pushw`、`callw` 等，并且直接在代码中定义了字符串 `"Hello, world!"`。

3、在以上代码中使用了函数 `display_str`，在调用

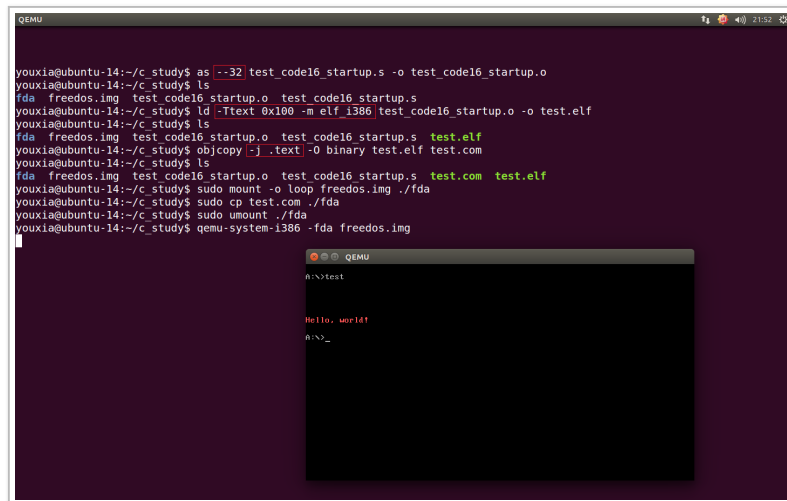
`display_str` 之前，我使用 `pushw $15` 和 `pushw $message` 将参数从右向左依次压栈，然后使用 `callw` 指令调用函数，这和 C 语言的函数调用约定是一样的。调用 `callw` 指令会自动将 `%ip` 寄存器压栈，而在函数开始时，我又用 `pushw`

`%bp` 将 `%bp` 寄存器压栈，所以 `%esp` 又向下移动了 4 个字节，所以在函数中使用 `0x4(%esp)` 和 `0x6(%esp)` 可以访问到这两个参数。在 32 位代码中，由于调用函数时压栈的是 `%ebp` 和 `%ebp`，所以需要使用 `0x8(%esp)` 和 `0xc(%esp)` 来依次访问压栈的参数。关于汇编语言函数调用的细节，可以参考《Programming from the ground up》。

4、以上代码使用 BIOS 中断 `int 0x10` 来输出字符串，使用 DOS 中断 `int 0x21` 来返回 DOS 系统。

5、最重要的是，需要使用 `.code16` 指令让汇编器将程序汇编成 16 位的代码。

代码完成后，使用下面一串命令就可以把它进行汇编、链接，然后转换成 DOS 下的纯二进制格式（Plain Binary），最后复制到 FreeDOS.img 中，使用 Qemu 虚拟机执行 FreeDOS，然后运行该 16 位实模式程序。这一串命令及其运行效果如下图：



```
youxia@ubuntu-14:/c_study$ as --32 test_code16_startup.s -o test_code16_startup.o
youxia@ubuntu-14:/c_study$ ls
fda freedos.img test_code16_startup.o test_code16_startup.s
youxia@ubuntu-14:/c_study$ ld -Ttext 0x100 -m elf_i386 test_code16_startup.o -o test.elf
youxia@ubuntu-14:/c_study$ ls
fda freedos.img test_code16_startup.o test_code16_startup.s test.elf
youxia@ubuntu-14:/c_study$ objcopy -j .text -O binary test.elf test.com
youxia@ubuntu-14:/c_study$ ls
fda freedos.img test_code16_startup.o test_code16_startup.s test.com test.elf
youxia@ubuntu-14:/c_study$ sudo mount -o loop freedos.img ./fda
youxia@ubuntu-14:/c_study$ sudo umount ./fda
youxia@ubuntu-14:/c_study$ qemu-system-i386 -fda freedos.img

QEMU
A:\>test

Hello, world!
A:\>
```

这些命令中比较重要的选项我都特意标出来了。由于我用的是 64 位的环境，所以调用 `as` 命令的时候需要指定 `--32` 选项，调用 `ld` 命令的时候需要指定 `-m elf_i386` 选项。指定以上选项后，生成的是 32 位的 ELF 目标文件，否则默认会生成 64 位的 ELF 目标文件，如果目标文件是 64 位，以后和 C 语言生成的目标文件连接时会出问题。使用 32 位环境的朋友们不用特意指定这两个选项。由于 DOS 系统总是把 Plain Binary 文件载入到 0x100 地址处执行，所以调用 `ld` 命令时，需要指定 `-Ttext 0x100` 选项。ld 命令执行完成

Linux 桌面玩家指南: 08. 使用 GCC 和 GNU Binutils 编写能在 x86 实模式运行的 16 位代码 - 京山游侠 - 博客园

后, 生成的是 ELF 格式的可执行文件 `test.elf`, 最后需要调用 `objcopy` 生成纯二进制文件, `-j .text` 选项的意思是只需要代码段, 因为我把 "Hello, world!" 也是定义在代码段中的, `-O binary` 选项指定输出格式为纯二进制文件, 输出文件为 `test.com`。最后, 将 `freedos.img` 镜像文件 mount 到 Ubuntu 中, 将 `test.com` 拷贝到其中, 然后

`umount`, 然后运行虚拟机, 在 DOS 中运行 `test`, 就可以看到效果了。

除了 `as` 和 `ld`, GNU Binutils 中的其它程序也是写程序和分析程序时的好帮手。可以使用 `readelf -S` 查看 `test.elf` 文件中的所有段, 也可以使用 `objdump -s` 命令将 `test.elf` 中的数据以 16 进制形式输入, 如下图:

```
youxia@ubuntu-14:~/c_study$ readelf -S test.elf
共有 5 个节头, 从偏移量 0x224 开始:

节头:
[Nr] Name      Type             Addr             Off             Size             ES Flg Lk Inf Al
[ 0]          NULL                00000000          00000000          00000000          00  0  0  0
[ 1] .text       PROGBITS          00000100          000100          000100          AX  0  0  1
[ 2] .shstrtab   STRTAB            00000000          000200          000021          00  0  0  1
[ 3] .symtab     SYMTAB            00000000          0002ec          0000b0          10  4  7  4
[ 4] .strtab     STRTAB            00000000          00039c          000050          00  0  0  1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)

youxia@ubuntu-14:~/c_study$ objdump -s test.elf

test.elf: 文件格式 elf32-i386

Contents of section .text:
0100 8cc8ed8 8cc8ed0 b800289 c46a0f68 .....j.h
0110 1a01e815 00b8004c cd214865 6c6cf2c .....L.Hello,
0120 20776f72 6cd4210a 00005567 8b442404 .....world!..Ug.D$.
0130 89c5678b 4c2406b8 0113bb8c 00b200cd .....g.L$.
0140 105dc300 00000000 00000000 00000000 .....
0150 00000000 00000000 00000000 00000000 .....
0160 00000000 00000000 00000000 00000000 .....
0170 00000000 00000000 00000000 00000000 .....
0180 00000000 00000000 00000000 00000000 .....
0190 00000000 00000000 00000000 00000000 .....
01a0 00000000 00000000 00000000 00000000 .....
01b0 00000000 00000000 00000000 00000000 .....
01c0 00000000 00000000 00000000 00000000 .....
01d0 00000000 00000000 00000000 00000000 .....
01e0 00000000 00000000 00000000 00000000 .....
01f0 00000000 00000000 00000000 00000000 .....
youxia@ubuntu-14:~/c_study$
youxia@ubuntu-14:~/c_study$
```

当然, 也可以使用 `objdump -d` 或者 `objdump -D` 将程序进行反汇编, 查看是否真正生成了 16 位代码, 如下图: (反汇编时一定要指定 `-m i8086` 选项)

```
youxia@ubuntu-14:~/c_study$ objdump -d -m i8086 test.elf

test.elf: 文件格式 elf32-i386

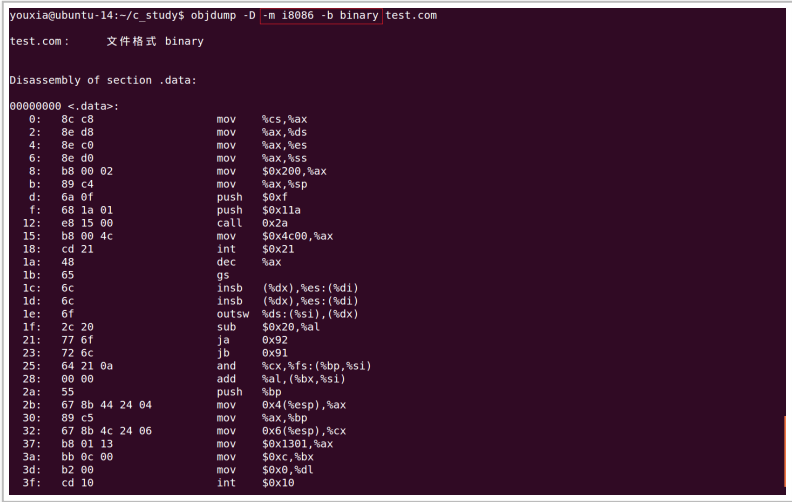
Disassembly of section .text:

00000100 <start>:
100: 8c c8          mov     %cs, %ax
102: 8e d8          mov     %ax, %ds
104: 8e c0          mov     %ax, %es
106: 8e d0          mov     %ax, %ss
108: b8 00 02      mov     $0x200, %ax
10b: 89 c4          mov     %ax, %esp
10d: 6a 0f          push    $0xf
10f: 68 1a 01      push    $0x11a
112: e8 15 00      call    12a <display_str>
115: b8 00 4c      mov     $0x4c0, %ax
118: cd 21          int     $0x21

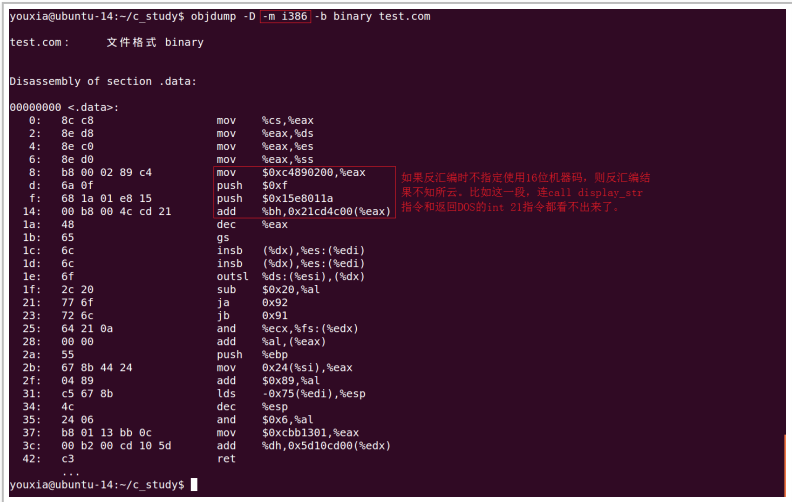
0000011a <message>:
11a: 48            dec     %ax
11b: 65            gs
11c: 6c            insb    (%dx), %es: (%di)
11d: 6c            insb    (%dx), %es: (%di)
11e: 6f            outsw   %ds: (%si), (%dx)
11f: 2c 20          sub     $0x20, %al
121: 77 0f          ja      192 <display_str+0x68>
123: 72 6c          jb      191 <display_str+0x67>
125: 64 21 0a      and     %cx, %fs: (%bp, %si)
...

0000012a <display_str>:
12a: 55            push    %bp
12b: 67 8b 44 24 04 mov     0x4(%esp), %ax
12c: 89 c5          mov     %ax, %bp
12e: 67 8b 4c 24 06 mov     0x6(%esp), %cx
```

也可以对纯二进制格式的文件进行反汇编，必须指定 `-b` `binary` 选项，如下图，对 `test.com` 进行反汇编：



反汇编时，一定要指定 `-m i8086` 选项，否则 `objdump` 不知道反汇编的是 16 位代码。（前面提到过 Linux 从诞生起就是 32 位，所以 ELF 只有 32 位和 64 位两种，没有 16 位的 ELF 格式。）如下图，如果使用 `-m i386` 选项进行反汇编，反汇编结果将不知所云：



下面进入 C 语言的世界。为了搞清楚 C 语言生成的 16 位代码的汇编指令有哪些特别之处，先写一个简单的 C 语言程序进行调研，如下图：


```
1  __asm__ (".code16\n");
2
3  void display_str(char* str, short length){
4      char* p = str;
5      short l = length;
6  }
7
8  void main(){
9      char* str = "Hello world from C language.\n";
10     display_str(str, 29);
11 }
```

test_code16.c 9,49 全部

test_code16.c" 11L, 201C

该程序有以下特点：

- 1、程序的开头使用了 `__asm__ (".code16\n")` 嵌入汇编指令，以指示 `as` 生成 16 位代码；
- 2、`display_str` 函数的签名和之前汇编语言中的相同，可以使用它来观察 C 语言生成的代码如何传递参数。

使用下面的命令对程序进行编译和反汇编，如下图：

```
youxia@ubuntu-14:~/c_study$ gcc -c -m32 test_code16.c
youxia@ubuntu-14:~/c_study$ objdump -d -m i386 test_code16.o

test_code16.o: 文件格式 elf32-i386

Disassembly of section .text:

00000000 <display_str>:
0: 66 55                push    %ebp
2: 66 89 e5             mov     %esp,%ebp
5: 66 83 ec 14          sub     $0x14,%esp
9: 67 66 8b 45 0c       mov     0xc(%ebp),%eax
e: 67 89 45 ec          mov     %ax,-0x14(%ebp)
12: 67 66 8b 45 08       mov     0x8(%ebp),%eax
17: 67 66 89 45 fc       mov     %eax,-0x4(%ebp)
1c: 67 66 0f b7 45 ec    movzwl -0x14(%ebp),%eax
22: 67 89 45 fa          mov     %ax,-0x6(%ebp)
26: c9                  leave   %eax
27: c3                  ret

00000028 <main>:
28: 66 55                push    %ebp
2a: 66 89 e5             mov     %esp,%ebp
2d: 66 83 ec 18          sub     $0x18,%esp
31: 67 66 c7 45 fc 00 00 movl    $0x0,-0x4(%ebp)
38: 00 00
3a: 67 66 c7 44 24 04 1d movl    $0x1d,0x4(%esp)
41: 00 00
44: 67 66 8b 45 fc       mov     -0x4(%ebp),%eax
49: 67 66 89 04 24       mov     %eax,(%esp)
4e: e8 fe ff             call    4f <main+0x27>
51: c9                  leave   %eax
52: c3                  ret

youxia@ubuntu-14:~/c_study$
```

从上图可以看出，C 语言生成的代码虽然是 16 位，但是它有

如下特点：① 16 位寄存器

② 16 位立即数 ③ 16 位寻址方式

一开始是 `push %ebp`, 而不是 `push %bp`; ②在

`display_str` 函数中获取参数的位置分别为 `0x8(%ebp)` 和 `0xc(%ebp)`, 而不是我在汇编语言中写的 `0x4(%ebp)` 和 `0x6(%ebp)`; ③从生成的 `main` 函数可以看出, 调用 `display_str` 之前, 没有使用 `push` 命令把参数压栈, 而是直接通过 `sub $0x18, %esp` 调整 `%esp` 的位置, 然后使用

`mov` 指令将参数放到指定位置, 和使用 `push` 指令的效果相同; ④虽然我在 `display_str` 函数的定义中故意将长度参数定义为 `short`, 但是从生成的代码中可以看到依然是每隔 4 个字节放一个参数。

另外需要说明的是, 调用 `gcc` 时除了指定 `-c` 选项指示它只编译不连接外, 还要指定 `-m32` 选项, 这样才会生成 32 位的汇编代码, 而只有在 32 位的汇编代码中使用 `.code16` 指令, 才能编译成 16 位的机器码。如果没有指定 `-m32` 选项, 则生成的是 64 位汇编代码, 然后汇编时会出错。使用 `-m32` 选项后, 生成的目标文件是 ELF32 格式。ELF32 格式的目标文件只能和 ELF32 格式的目标文件连接, 这也是为什么前面的 `as` 和 `ld` 需要指定 `--32` 和 `-m elf_i386` 选项的原因。

通过以上分析, 似乎可以得出以下结论: 只需要将汇编代码中的 `pushw %bp` 更改为 `pushl %ebp`, 然后将获取参数的位置调整为 `0x8(%ebp)` 和 `0xc(%ebp)`, 就可以从 C 语言里面成功调用到汇编语言中的函数了。而事实上, 还有一点点小差距。从上面的反汇编代码中可以看到, 函数调用时使用的是 16 位的 `call` 指令, 该指令压栈的是 `%ip`, 而不是 `%eip`, 而 C 语言生成的函数框架中获取的参数位置是按照将 `%eip` 压栈计算出来的, 它们之间差了两个字节。

为了证明我以上判断的准确性, 我将上面的 C 语言程序和汇编程序修改后, 编译连接成一个完整的程序, 看看它究竟能否正确运行。

C 语言程序修改很简单, 就是去掉了 `display_str` 函数的实现, 只保留声明。如下图:

```
1  asm ("code16\n");
2
3  void display_str(char* str, short length);
4
5  void main(){
6      char* str = "Hello world from C language.\n";
7      display_str(str, 29);
8  }
```

test_code16.c 3,42 全部

test_code16.c" 8L, 158C

汇编语言的更改包含以下几个地方：将 `display_str` 函数导出，将 `pushw %bp` 改为 `pushl %ebp`，同时修改获取参数的位置。如下图：

```
1  .code16
2  .global _start
3  .text
4  _start:
5      movw %cs,%ax
6      movw %ax,%ds
7      movw %ax,%es
8      movw %ax,%fs
9      movw $1000,%ax
10     movw %ax,%esp
11
12     call main
13
14     movw $0x4C00,%ax /* return to DOS */
15     int $0x21
16
17     .global display_str
18 display_str: /* display_str(char* str, short length) */
19     pushl %ebp
20     movw [0x0(%esp)],%ax /* str */
21     movw %ax,%bp
22     movw [0xc(%esp)],%cx /* length */
23     movw $0x1301,%ax
24     movw $0x000c,%bx
25     movb $0x0,%dl
26     int $0x10
27     popl %ebp
28     ret
29     .org 0x100
30 stack: /* This is the bottom of the stack */
```

test_code16_startup.s [+] 30,61 全部

test_code16_startup.s" 32L, 645C

编译、连接、运行程序的指令如下：

```
youxia@ubuntu-14:/c_study$ gcc -c -m32 test_code16.c
youxia@ubuntu-14:/c_study$ as --32 test_code16_startup.s -o test_code16_startup.o
youxia@ubuntu-14:/c_study$ ld -Ttext 0x100 -m elf_i386 test_code16_startup.o test_code16.o -o test.elf
youxia@ubuntu-14:/c_study$ sudo mount -o loop freedos.img ./fda
[sudo] password for youxia:
youxia@ubuntu-14:/c_study$ readelf -S test.elf
共有 8 个节头，从偏移量 0x3e8 开始：
节头：
[Nr] Name      Type             Addr             Off
[ 0] .text       PROGBITS         00000000         0000
[ 1] .rodata     PROGBITS         00000100         0010
[ 2] .eh_frame   PROGBITS         00000331         0003
[ 3] .comment    PROGBITS         00000350         0003
[ 4] .shstrtab   STRTAB           00000000         0000
[ 5] .syntab     SYMTAB           00000000         0000
[ 6] .strtab     STRTAB           00000000         0000
[ 7] .strtab     STRTAB           00000000         0000
Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (stri
I (info), L (link order), G (group), T (TLS), E (excl
```

```
0 (extra OS processing required) o (OS specific), p l.
youxia@ubuntu-14:~/c_study$ objcopy -j .text -j .rodata -O binary test.elf test.com
youxia@ubuntu-14:~/c_study$ sudo cp test.com ./fda
youxia@ubuntu-14:~/c_study$ sudo umount ./fda
youxia@ubuntu-14:~/c_study$ qemu-system-i386 -fda freedos.img
```

可以看到 "Hello world from C language." 没有正确显示出来。上面的命令都是前面用过的，不需要多解释，唯一不同的是使用 C 语言写的程序多了一个 `.rodata` 段，所以在 `objcopy` 的时候需要把这个段也包含进来。

由于 C 语言生成的函数框架都是从 `0x8(%ebp)` 开始取参数，它认为 `0x0(%ebp)` 是 old ebp, `0x4(%ebp)` 是 `%eip`，而事实上使用 16 位的 `call` 指令调用函数后，`0x4(%ebp)` 中是 `%ip` 而不是 `%eip`，所以要从 `0x6(%ebp)` 开始取参数。我们不可能修改 C 语言生成的函数框架，只能看看能否将 16 位的 `call` 改成 32 位的 `call`。

办法当然是有的，那就是不使用 `.code16`，而使用 `.code16gcc`。`.code16gcc` 和 `.code16` 不同的地方就在于它生成的汇编代码在使用到 `call`、`ret`、`jump` 等指令时，都生成 32 位的机器码，相当于 `calll`，`retl`，`jumpl`。这也是 `.code16gcc` 叫 `.code16gcc` 的原因，因为它就是配合 GCC 生成的函数框架使用的。

下面再来修改代码，C 语言代码修改很简单，只需要将 `.code16` 改成 `.code16gcc` 即可，如下图：

```
1 __asm__(".code16gcc");
2
3 void display_str(char* str, short length);
4
5 void main(){
6     char* str = "Hello world from C language.\n";
7     display_str(str, 29);
8 }
```

test_code16.c 1,20 全部

通过反汇编，可以看到它使用了 32 位的 `calll` 和 `retl`，如下图：

```
youxia@ubuntu-14:~/c_study$ objdump -d -m i386 test_code16.o
test_code16.o: 文件格式 elf32-i386

Disassembly of section .text:

00000000 <main>:
0: 66 55          push %ebp
2: 66 89 e5       mov %esp,%ebp
5: 66 83 e4 f0    and $0xfffffff0,%esp
9: 66 83 ec 20    sub $0x20,%esp
d: 67 66 c7 44 24 1c 00 movl $0x0,0x1c(%esp)
14: 00 00 00       movl $0x1d,0x4(%esp)
17: 67 66 c7 44 24 04 1d movl $0x1d,0x4(%esp)
1e: 00 00 00
21: 67 66 8b 44 24 1c mov 0x1c(%esp),%eax
27: 67 66 89 04 24 mov %eax,(%esp)
2c: 66 e8 fc ff ff calll 2e <main+0x2e>
32: 66 c9          leavel
34: 66 c3          retl
youxia@ubuntu-14:~/c_study$
```

汇编程序的修改主要是将 .code16 改为 .code16gcc, 然后手动将 callw 改成 calll, 将 retw 改成 retl, 如下图:

```
1  .code16gcc
2  .global _start
3  .text
4  _start:
5  movw %cs,%ax
6  movw %ax,%ds
7  movw %ax,%es
8  movw %ax,%ss
9  movw $stack,%ax
10 movw %ax,%sp
11
12 calll main
13
14 movw $0x4c00,%ax /* return to DOS */
15 int $0x21
16
17 .global display_str
18 display_str: /* display_str(char* str, short length) */
19 pushl %ebp
20 movw 0x8(%esp),%ax /* str */
21 movw %ax,%bp
22 movw 0xc(%esp),%cx /* length */
23 movw $0x1301,%ax
24 movw $0x000c,%bx
25 movb $0x0,%dl
26 int $0x10
27 popl %ebp
28 retl
29 .org 0x100
30 stack: /* This is the bottom of the stack */
```

test_code16 startup.s 30,61 全部

test_code16 startup.s 38L, 609C

最后, 编译连接, 拷贝到 freedos.img, 运行虚拟机, 查看运行效果, 如下图:

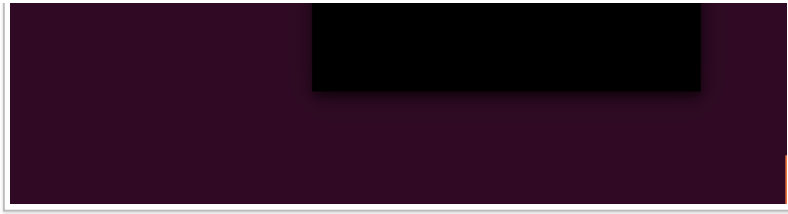
```
QEMU
youxia@ubuntu-14:~/c_study$ gcc -c -m32 test_code16.c
youxia@ubuntu-14:~/c_study$ as --32 test_code16_startup.s -o test_code16_startup.o
youxia@ubuntu-14:~/c_study$ ld -Ttext 0x100 -m elf_i386 test_code16_startup.o test_code16.o -o test.elf
youxia@ubuntu-14:~/c_study$ objcopy -j .text -j .rodata -O binary test.elf test.com
youxia@ubuntu-14:~/c_study$ sudo mount -o loop freedos.img ./fda
youxia@ubuntu-14:~/c_study$ sudo cp test.com ./fda
youxia@ubuntu-14:~/c_study$ sudo umount ./fda
youxia@ubuntu-14:~/c_study$ qemu-system-i386 -fda freedos.img
```

QEMU

A:\>test

Hello world from C language.

A:\>_



大功告成。

[回到顶部](#)

总结

编写运行于 x86 实模式下的 16 位代码是一个很复古的话题，编写能在 DOS 下运行的 Plain Binary 可执行文件是一个更复古的话题。以往，凡是需要使用 x86 的 16 位实模式的时候，作者都喜欢用 NASM 来编程。比如《30 天自制操作系统》、《Orange's 一个操作系统的实现》、《x86 汇编语言——从实模式到保护模式》等书籍都以 NASM 汇编器和 Intel 汇编语法作为示例。而且他们都是在进入 32 位保护模式后，才让汇编语言和 C 语言共同工作。

我用 Linux 操作系统，所以我就是想不管是写 32 位代码，还是 16 位代码，都能使用 GCC 和 GNU AS。我还想即使是在 16 位模式下，也能尽量少用汇编语言，多用 C 语言。经过努力，有了上面的文章。使用 GCC 和 GNU Binutils 编写运行于 x86 实模式的 16 位代码的过程如下：

1、如果只用汇编语言编写 16 位程序，请使用 `.code16` 指令，并保证只使用 16 位的指令和寄存器；如果要和 C 语言一起工作，请使用 `.code16gcc` 指令，并且在函数框架中使用 `pushl` , `calll` , `retl` , `leavel` , `jmp1` , 使用 `0x8(%ebp)` 开始访问函数的参数；很显然，使用 C 语言和汇编语言混编的程序可以在实模式下运行，但是不能在 286 之前的真实 CPU 上运行，因为 286 之前的 CPU 还没有 `pushl` 、 `calll` 、 `retl` 、 `leavel` 、 `jmp1` 等指令。

2、使用 `as` 时，请指定 `--32` 选项，使用 `gcc` 时，请指定 `-m32` 选项，使用 `ld` 时，请指定 `-m elf_i386` 选项。如果是反汇编 16 位代码，在使用 `objdump` 时，请使用 `-m`

3、在 DOS 中运行的 .com 文件会被加载到 0x100 处执行，所以使用 `ld` 连接时需指定 `-Ttext 0x100` 选项；引导扇区的代码会被加载到 0x7c00 处执行，所以使用 `ld` 连接时需指定 `-Ttext 0x7c00` 选项。

4、使用 `gcc` 、 `as` 、 `ld` 生成的程序默认都是 ELF 格式，而在 DOS 下运行的 .com 程序是 Plain Binary 的，在引导扇区运行的代码也是 Plain Binary 的，所以需要使用 `objcopy` 将 ELF 文件中的代码段和数据段拷贝到一个 Plain Binary 文件中，使用 `-O binary` 选项；Plain Binary 文件也可以反汇编，在使用 `objdump` 时需指定 `-b binary` 选项。

[回到顶部](#)

版权声明

该随笔由京山游侠在 2018 年 10 月 14 日发布于博客园，引用请注明出处，转载或出版请联系博主。QQ 邮箱：
1841079@qq.com

全文完

本文由 简悦 SimpRead 优化，用以提升阅读体验

使用了 全新的简悦词法分析引擎 ^{beta}，[点击查看详细说明](#)

