编写一个最小的 64 位 Hello World

https://cjting.me/2020/12/10/tiny-x64-helloworld/

CJ Ting Fri Dec, 18 06:34

Hello World 应该是每一位程序员的启蒙程序，出自于 Brian Kernighan 和 Dennis Ritchie 的一代经典著作 The C Programming Language。

```
// hello.c
#include <stdio.h>

int main() {
  printf('hello, world\n');
  return 0;
}
```

这段代码我想大家应该都太熟悉了，熟悉到可以默写出来。虽然是非常简单的代码，但是如果细究起来，里面却隐含着很多细节：

- `#include <stdio.h>` 和 `#include 'stdio.h'` 有什么区别？
- `stdio.h` 文件在哪里？里面是什么内容？
- 为什么入口是 `main` 函数？可以写一个程序入口不是 `main` 吗？
- `main` 的 int 返回值有什么用？是谁在处理 `main` 的返回值？
- `printf` 是谁实现的？如果不用 `printf` 可以做到在终端中打印字符吗？

上面这些问题其实涉及到程序的编译、链接和装载，日常工作中也许大家并不会在意。

现代 IDE 在方便我们开发的同时，也将很多底层的细节隐藏了起来。往往写完代码以后，点击「构建」就行了，至于构建在发生什么，具体是怎么构建的，很多人并不关心，甚至根本不知道从源代码到可执行程序这中间经历了什么。

编译、链接和装载是一个巨大的话题，不是一篇博客可以覆盖的。在这篇博客中，我想使用「文件尺寸」作为线索，来介绍从 C 源代码到可执行程序这个过程中，所经历的一系列过程。

Tip: 关于编译、链接和装载，这里想推荐一本书《程序员的自我修养》。不得不说，这个名字起得非常不好，很有哗众取宠的味道，但是书的内容是不错的，值得一看。

我们先来编译上面的程序：

```
$ gcc hello.c -o hello
$ ./hello
hello, world
$ ll hello
-rwxr-xr-x 1 root root 16712 Nov 24 10:45 hello
```

Tip: 后续所有的讨论都是基于 64 位 CentOS7 操作系统。

我们会发现这个简单的 hello 程序大小为 16K。在今天看来，16K 真的没什么，但是考虑到这个程序所做的事情，它真的需要 16K 吗？

在 C 诞生的上个世纪 70 年代，PDP-11 的内存为 144K，如果一个 hello world 就要占 16K，那显然是不合理的，一定有办法可以缩减体积。

Tip:

说起 C 语言，我想顺带提一下 UNIX。没有 C 就没有 UNIX 的成功，没有 UNIX 的成功也就没有 C 的今天。诞生于上个世纪 70 年代的 UNIX 不得不说是一项了不起的创造。

这里推荐两份关于 UNIX 的资料：

- The UNIX Time-Sharing System 1974 年由 Dennis Ritchie 和 Ken Thompson 联合发表的介绍 UNIX 的论文。不要被“论文”二字所吓到，实际上，这篇文章写得非常通俗易懂，由 UNIX 的作者们向你娓娓道来 UNIX 的核心设计理念。
- The UNIX Operating System 一段视频，看身着蓝色时尚毛衣的 Kernighan 演示 UNIX 的特性，不得不说，Kernighan 简直太帅了。

接下来我们来玩一个游戏，目标是：在 CentOS7 64 位操作系统上，编写一个体积最小的打印 hello world 的可执行程序。

Executable

我们先来看「可执行程序」这个概念。

什么是可执行程序？按照字面意思来理解，那就是：可以执行的程序。

ELF

上面用 C 编写的 hello 当然是可执行程序，毫无疑问。

实际上，我们可以说它是真正的“可执行”程序（区别于后文的脚本），或者说“原生”程序。

因为它里面包含了可以直接用于 CPU 执行的机器代码，它的执行无需借助外部。

hello 的存储格式叫做 ELF，全称为 Executable and Linkable Format，看名称可以知道，它既可以用于存储目标文件，又可以用于存储可执行文件。

ELF 本身并不难理解，`/usr/include/elf.h` 中含有 ELF 结构的详细信息。难理解的是由 ELF 所掀开的底层世界，目标文件是什么？和执行文件有什么区别？链接在干什么？目标文件怎样变成可执行文件等等等等。

Shebang

接下来我们来看另外一种形式的可执行程序，脚本。

```
$ cat > hello.sh <<EOF
#!/bin/bash
echo 'hello, world'
EOF
$ chmod +x hello.sh
$ ./helo.sh
hello, world
```

按照定义，因为这个脚本可以直接从命令行执行，所以它是可执行程序。

那么 hello 和 hello.sh 的区别在哪里？

可以发现 hello.sh 的第一行比较奇怪，这是一个叫做 Shebang 的东西 `#!/bin/bash`，这个东西表明当前文件需要 `/bin/bash` 程序来执行。

所以，hello 和 hello.sh 的区别就在于：一个可以直接执行不依赖于外部程序，而另一个需要依赖外部程序。

我曾经有一个误解，认为 Shebang 是 Shell 在处理，当 Shell 执行脚本时，发现第一行是 Shebang，然后调用相应的程序来执行该脚本。

实际上并不是这样，对 Shebang 的处理是内核在进行。当内核加载一个文件时，会首先读取文件的前 128 个字节，根据这 128 个字节判断文件的类型，然后调用相应的加载器来加载。

比如说，内核发现当前是一个 ELF 文件（ELF 文件前四个字节为固定值，称为魔数），那么就调用 ELF 加载器。

而内核发现当前文件含有 Shebang，那么就会启动 Shebang 指定的程序，将当前路径作为第一个参数传入。所以当我们执行 `./hello.sh` 时，在内核中会被变为 `/bin/bash ./hello.sh`。

这里其实有一个小问题，如果要脚本可以从命令行直接执行，那么第一行必须是 Shebang。Shebang 的形式固定为 `#!` 开头，对于使用 # 字符作为注释的语言比如 Python, Ruby, Elixir 来说，这自然不是问题。但是对于 # 字符不是注释字符的语言来说，这一行就是一个非法语句，必然带来解释错误。

比如 JavaScript，它就不使用 # 作为注释，我们来写一个带 Shebang 的 JS 脚本看看会怎么样。

```
$ cat <<EOF > test.js
#!/usr/bin/env node
console.log('hello world')
EOF
$ chmod +x test.js
$ ./test.js
hello world
```

并没有出错，所以这里是怎么回事？按道理来说第一行是非法的 JS 语句，解释器应该要报错才对。

如果把第一行的 Shebang 拷贝一份到第二行，会发现报了 `SyntaxError`，这才是符合预期的。所以必然是 Node 什么地方对第一行的 Shebang 做了特别处理，否则不可能不报错。

大家可以在 Node 的代码里面找一找，看看在什么地方😉

答案是什么地方都没有，或者说在最新的 Node 中，已经没有地方在处理 Shebang 了。

在 Node v11 中，我们可以看到相应的代码在 这里。

`stripShebang` 函数很明显，它的作用在于启动 JS 解释器的时候，将第一行的 Shebang 移除掉。

但是在 Node v12 以后，Node 更新了 JS 引擎 V8 到 7.4，V8 在这个版本中实现一个叫做 Hashbang grammar 的功能，也就是说，从此以后，V8 可以处理 Shebang 了，因此 Node 删除了相关代码。

因为 Shebang 是 V8 在处理了，所以我们在浏览器中也可以加载带有 Shebang 的 JS 文件，不会有任何问题~

我们可以得出结论，支持作为脚本使用的语言，如果不使用 # 作为注释字符，那么必然要特别处理 Shebang，否则使用起来就太不方便了。

/usr/bin/env

上面的 test.js 文件中，不知道大家是否注意到，解释器路径写的是 `/usr/bin/env node`。

这样的写法如果经常写脚本，应该不陌生，我之前一直这样用，但是没有仔细去想过为什么。

首先我们来看 `/usr/bin/env` 这个程序是什么。

根据 `man env` 返回的信息：env - run a program in a modified environment.

`env` 的主要作用是修改程序运行的环境变量，比如说

```
$ export name=shell
$ node
> process.env.name
'shell'
$ env name=env node
> process.env.name
'env'
```

通过 env 我们修改了 node 运行时的环境变量。但是这个功能和我们为什么要在 Shebang 中使用 env 有什么关系？

在 Shebang 中使用 env 其实是因为另外一个原因，那就是 env 会在 PATH 中搜索程序并执行。

当我们执行 `env abc` 时，env 会在 PATH 中搜索 abc 然后执行，就和 Shell 一样。

这就解释了为什么我们要在脚本中使用 `/usr/bin/env node`。对于想要给他人复用的脚本，我们并不清楚他人系统上 node 的路径在哪里，但是我们清楚的是，它一定在 PATH 中。

而同时，绝大部分系统上，`env` 程序的位置是固定的，那就是 `/usr/bin/env`。所以，通过使用 `/usr/bin/env node`，我们可以保证不管其他用户将 node 安装在何处，这个脚本都可以被执行。

binfmt_misc

前面我们提到过，内核对于文件的加载其实是有一套“多态”机制的，即根据不同的类型来选择不同的加载器。

那么这个过程我们可以自己定制吗？

当然可以，内核中有一个加载器叫做 `binfmt_misc`，看名字可以知道，这个加载器用于处理各种各样非标准的其他类型。

通过一套语法，我们可以告知 binfmt_misc 加载规则，实现自定义加载。

比如我们可以通过 binfmt_misc 实现直接运行 Go 文件。

```
# 运行 Go 文件的指令是 `go run`，不是一个独立的程序
# 所以，我们先要写一个脚本包装一下
$ cat <<EOF > /usr/local/bin/rungo
#!/bin/bash
go run $1
EOF
# 接下来写入规则告诉 binfmt_misc 使用上面的程序来加载所有
# 以 .go 结尾的文件
$ echo ':golang:E::go::/usr/local/bin/rungo:' > /proc/sys/fs/binfmt_misc/register
# 现在我们就可以直接运行 Go 文件了
$ cat << EOF > test.go
package main
import 'fmt'

func main() {
  fmt.Println('hello, world')
}
EOF
$ chmod +x test.go
$ ./test.go
hello, world
```

Tiny Script

根据上面的知识，如果我们想要编写一个体积最小的打印 hello world 的脚本，我们要在这两方面着手：

- 解释器路径要尽量短
- 脚本本身用于打印的代码要尽量短

解释器的路径很好处理，我们可以使用链接。

脚本本身的代码要短，这就很考验知识了，我一开始想到的是 Ruby，`puts 'hello, world'` 算是非常短的代码了，没有一句废话。但是后来 Google 才发现，还有更短的，那就是 PHP 😉

PHP 中 打印 hello world 的代码就是 `hello, world`，对的，你没看错，连引号都不用。

所以，最终我们的结果如下：

```
# 假设 php 在 /usr/local/bin/php
$ cd /
$ ln -s /usr/local/bin/php p
$ cat <<EOF > final.php
#!/p
hello, world
EOF
$ chmod +x final.php
$ ./final.php
hello, world
$ ll final.php
-rwxr-xr-x 1 root root 18 Dec  2 22:32 final.php
```

在脚本模式下，我们的成绩是 18 个字节，使用的解释器是 PHP。

其实在脚本模式下编写最小的 hello world 没有太大意义，因为我们完全可以自己写一个输出 hello world 的程序作为解释器，然后脚本里面只要 `#!/x` 就行了。

Tiny Native

上面的脚本只是抛砖引玉，接下来我们进入正题，怎样编写一个体积最小的打印 hello world 的原生可执行程序？

网上有很多关于这个话题的讨论，但基本都是针对 x86 的。现如今 64 位机器早就普及了，所以我们这里针对的是 64 位的 x64。

Tip: 64 位机器可以执行 32 位的程序，比如我们可以使用 `gcc -m32` 来编译 32 位程序。但这只是一个后向兼容，并没有充分利用 64 位机器的能力。

Step0

首先，我们使用上文提到的 hello.c 作为基准程序。

```
// hello.c
#include <stdio.h>

int main() {
  printf('hello, world\n');
  return 0;
}
```

`gcc hello.c -o hello.out` 编译以后，它的大小是 16712 个字节。

Step1: Strip Symbols

第一步，也是最容易想到的一步，剔除符号表。

符号是链接器工作的的基本元素，源代码中的函数、变量等被编译以后，都变成了符号。

如果经常从事 C 开发，一定遇到过 `ld: symbol not found` 的错误，往往是忘记链接了某个库导致的。

使用 `nm` 我们可以查看一个二进制程序中含有哪些符号。

Tip:

`nm` 是“窥探”二进制的一个有力工具。记得之前有一次苹果调整了 iOS 的审核策略，不再允许使用了 UIWebView 的 App 提交。我们的 IPA 里面不知道哪个依赖使用了 UIWebView，导致苹果一直审核不过，每次都要二分注释、打包、提交审核，然后等待苹果的自动检查邮件告知结果，非常痛苦。

后来我想到了一个办法，就是使用 `nm` 查看编译出来的可执行程序，看看里面是否有 UIWebView 相关的 symbol，这大大简化了调试流程，很快就定位到问题了。

对 step0 中的 hello.out 程序使用 `nm`，输出如下：

```
$ nm hello.out
0000000000404038 B __bss_start
0000000000404038 b completed.6949
0000000000404028 D __data_start
0000000000404028 W data_start
0000000000401090 t deregister_tm_clones
0000000000401110 t __do_global_dtors_aux
0000000000403df8 d __do_global_dtors_aux_fini_array_entry
0000000000404030 D __dso_handle
0000000000403e08 d _DYNAMIC
0000000000404038 D _edata
0000000000404040 B _end
00000000004011e4 T _fini
0000000000401130 t frame_dummy
0000000000403df0 d __frame_dummy_init_array_entry
0000000000402154 r __FRAME_END__
0000000000404000 d _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
0000000000402014 r __GNU_EH_FRAME_HDR
0000000000401000 T _init
0000000000403df8 d __init_array_end
0000000000403df0 d __init_array_start
0000000000402000 R _IO_stdin_used
0000000000403e00 d __JCR_END__
0000000000403e00 d __JCR_LIST__
00000000004011e0 T __libc_csu_fini
0000000000401170 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
0000000000401156 T main
                 U puts@@GLIBC_2.2.5
00000000004010d0 t register_tm_clones
0000000000401060 T _start
0000000000404038 D __TMC_END__
```

可以看到有一个符号叫做 `main`，这个对应的就是我们的 main 函数。但是很奇怪没有看到 `printf`，而是出现了一个叫做 `puts@@GLIBC_2.2.5` 的符号。

这里其实是 GCC 做的一个优化，如果没有使用格式字符串调用 `printf`，GCC 会将它换成 `puts`。

这些符号都存储在了 ELF 中，主要用于链接，对于可执行文件来说，符号并没有什么太大作用，所以我们首先可以通过剔除符号表来节省空间。

有两个方法，第一是通过 `strip`，第二是通过 GCC 参数。

这里我们使用第二个方法，`gcc -s hello.c -o hello.out` 得到新的不含符号表的可执行程序，它的大小是 14512 字节。

虽然结果还是很大，但是我们省了 2K 左右，不错，再接再厉。

Step2: Optimization

第二个比较容易想到的办法就是优化，开启优化以后编译器会生成更加高效的指令，从而减小文件体积。

使用 `gcc -O3` 编译我们的程序，然后会发现，结果没有任何变化😂。

其实也非常合理，因为这个程序太简单了，没什么好优化的。

看来要再想想别的办法。

Step3: Remove Startup Files

之前我们提到过一个问题，是谁在调用 `main` 函数?

实际上我们编写的程序都会被默认链接到 GCC 提供的 C 运行时库，叫做 `crt` 。

通过 `gcc --verbose` 我们可以查看编译链接的详细日志。

```
$ gcc --verbose hello.c
...
/home/linuxbrew/.linuxbrew/Cellar/gcc/5.5.0_7/libexec/gcc/x86_64-unknown-linux-gnu/5.5.0/collect2 -plugin /home/
linuxbrew/.linuxbrew/Cellar/gcc/5.5.0_7/libexec/gcc/x86_64-unknown-linux-gnu/5.5.0/liblto_plugin.so -plugin-opt=/home/
linuxbrew/.linuxbrew/Cellar/gcc/5.5.0_7/libexec/gcc/x86_64-unknown-linux-gnu/5.5.0/lto-wrapper -plugin-opt=-fresolution=/tmp/
ccALFqFq.res -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc -plugin-opt=-pass-
through=-lgcc -plugin-opt=-pass-through=-lgcc_s --eh-frame-hdr -m elf_x86_64 -dynamic-linker /lib64/ld-linux-x86-64.so.2 --dynamic-
linker /home/linuxbrew/.linuxbrew/lib/ld.so -rpath /home/linuxbrew/.linuxbrew/lib /home/linuxbrew/.linuxbrew/Cellar/gcc/5.5.0_7/lib/
gcc/x86_64-unknown-linux-gnu/5.5.0/crt1.o /home/linuxbrew/.linuxbrew/Cellar/gcc/5.5.0_7/lib/gcc/x86_64-unknown-linux-gnu/5.5.0/
crti.o /home/linuxbrew/.linuxbrew/Cellar/gcc/5.5.0_7/lib/gcc/x86_64-unknown-linux-gnu/5.5.0/crtbegin.o -nostdlib -L/home/
linuxbrew/.linuxbrew/Cellar/gcc/5.5.0_7/lib/gcc/x86_64-unknown-linux-gnu/5.5.0 -L/home/linuxbrew/.linuxbrew/lib /tmp/cc2wNkTa.o -
lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed /home/linuxbrew/.linuxbrew/Cellar/gcc/5.5.0_7/
lib/gcc/x86_64-unknown-linux-gnu/5.5.0/crtend.o /home/linuxbrew/.linuxbrew/Cellar/gcc/5.5.0_7/lib/gcc/x86_64-unknown-linux-gnu/
5.5.0/crtn.o
```

可以发现我们的程序链接了 `crt1.o` , `crti.o` , `crtbegin.o` , `crtend.o` 以及 `crtn.o` 。

其中 `crt1.o` 里面提供的 `_start` 函数是程序事实上的入口，这个函数负责准备 main 函数需要的参数，调用 main 函数以及处理 main 函数的返回值。

上面这些 crt 文件统称为 Start Files。所以，现在我们的思路是，可不可以不用这些启动文件?

`_start` 函数主要功能有两个，第一是准备参数，我们的 main 不使用任何参数，所以这一部分可以忽略。

第二是处理返回值，具体的处理方式是使用 main 函数的返回值调用 `exit` 系统调用进行退出。

所以如果我们不使用启动文件的话，只需要自己使用系统调用退出即可。

因为我们现在不使用 `_start` 了，自然我们的主函数也没必要一定要叫做 `main`，这里我们改个名字突出一下这个事实。

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

int
nomain()
{
  printf('hello, world\n');
  _exit(0);
}
```

`unistd.h` 里面提供系统调用的相关函数，这里我们使用的是 `_exit`。为什么是 `_exit` 而不是 `exit`？可以参考这个回答 What is the difference between using _exit() & exit() in a conventional Linux fork-exec? 。

通过 `gcc -e nomain -nostartfiles` 编译我们的程序，其中 `-e` 指定入口，`--nostartfiles` 作用很明显，告诉 GCC 不必链接启动文件了。

我们得到的结果是 13664 个字节，不错，又向前迈进了一步。

Step4: Remove Standard Library

现在我们已经不使用启动文件了，但是我们还在使用标准库，`printf` 和 `_exit` 函数都是标准库提供的。

可不可以不使用标准库？

当然也可以。

这里就要说到系统调用，用户程序和操作系统的交互通过一系列称为”系统调用“的过程来完成。

比如 syscall_64 是 64 位 Linux 的系统调用表，里面列出了 Linux 提供的所有系统调用。

系统调用工作在最底层，通过约定的寄存器传递参数，然后使用一条特别的指令，比如 32 位 Linux 是 `int 80h`，64 位 Linux 是 `syscall` 进入系统调用，最后通过约定的寄存器获取结果。

C 标准库里面封装了相关函数帮助我们进行系统调用，一般我们不用关心调用细节。

现在如果我们不想使用标准库，那么就需要自己去完成系统调用，在 hello 程序中我们使用了两个系统调用：

- `write`：向终端打印字符实际上就是向终端对应的文件写入数据
- `exit`：退出程序

因为要访问寄存器，所以必须要使用内联汇编。

最终代码如下，在 C 中内联汇编的语法可以参考 这篇文档。

```
char *str = 'hello, world\n';

void
myprint()
{
  asm('movq $1, %%rax \n'
      'movq $1, %%rdi \n'
      'movq %0, %%rsi \n'
      'movq $13, %%rdx \n'
      'syscall \n'
      : // no output
      : 'r'(str)
      : 'rax', 'rdi', 'rsi', 'rdx');
}

void
myexit()
{
  asm('movq $60, %rax \n'
      'xor %rdi, %rdi \n'
      'syscall \n');
}

int
nomain()
{
  myprint();
  myexit();
}
```

使用 `gcc -nostdlib` 编译我们的程序，结果是 12912 字节。

能去的我们都去掉了，为什么还是这么大？？？

Step5: Custom Linker Script

我们先来看一下上一步得到的结果。

```
$ readelf -S -W step4/hello.out
Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        0000000000401000 001000 00006e 00  AX  0   0 16
  [ 2] .rodata           PROGBITS        0000000000402000 002000 00000e 01 AMS  0   0  1
  [ 3] .eh_frame_hdr     PROGBITS        0000000000402010 002010 000024 00   A  0   0  4
  [ 4] .eh_frame         PROGBITS        0000000000402038 002038 000054 00   A  0   0  8
  [ 5] .data             PROGBITS        0000000000404000 003000 000008 00  WA  0   0  8
  [ 6] .comment          PROGBITS        0000000000000000 003008 000022 01  MS  0   0  1
  [ 7] .shstrtab         STRTAB          0000000000000000 00302a 000040 00      0   0  1
```

可以发现 `Size` 很小但是 `Off` 的值非常大，也就是说每个 Section 的体积很小，但是偏移量很大。

使用 `xxd` 查看文件内容，会发现里面有大量的 0。所以情况现在很明朗，有人在对齐。

这里其实是默认的 Linker Script 链接脚本在做对齐操作。

控制链接器行为的脚本叫做 Linker Script，链接器内置了一个默认脚本，正常情况下我们使用默认的就好。

我们先来看看默认的脚本是什么内容。

```
$ ld --verbose
GNU ld (GNU Binutils) 2.34
...
  . = ALIGN(CONSTANT (MAXPAGESIZE));
...
  . = ALIGN(CONSTANT (MAXPAGESIZE));
...
```

可以看到里面有使用 `ALIGN` 来对齐某些 Section，使得他们的地址是 MAXPAGESIZE 的倍数，这里 MAXPAGESIZE 是 4K。

这就解释了为什么我们的程序那么大。

所以现在解决方案也就很清晰了，我们不使用默认的链接脚本，自行编写一个。

```
$ cat > link.lds <<EOF
ENTRY(nomain)

SECTIONS
{
  . = 0x8048000 + SIZEOF_HEADERS;

  tiny : { *(.text) *(.data) *(.rodata*) }

  /DISCARD/ : { *(*) }
}
EOF
```

使用 `gcc -T link.lds` 编译程序以后，我们得到了 584 字节，巨大的进步！🚀

Step6: Assembly

还有什么办法能进一步压缩吗？

上面我们是在 C 中使用内联汇编，为什么不直接使用汇编，完全抛弃 C？

我们来试试看，其实上面的 C 代码转换成汇编非常直接。

```
section .data
message: db 'hello, world', 0xa

section .text

global nomain
nomain:
  mov rax, 1
  mov rdi, 1
  mov rsi, message
  mov rdx, 13
  syscall
  mov rax, 60
  xor rdi, rdi
  syscall
```

这里我们使用 `nasm` 汇编器，我喜欢它的语法~

`nasm -f elf64` 汇编我们的程序，然后使用 `ld` 配合上面的自定义链接脚本链接以后得到可执行程序。

最后的结果是 440 字节，离终点又进了一步了✌~

Step7: Handmade Binary

还能再进一步吗？还有什么是我们没控制的？

所有的代码都已经由我们精确掌控了，但是最终的 ELF 文件依旧是由工具生成的。

所以，最后一步，我们来手动生成 ELF 文件，精确地控制可执行文件的每一个字节。

```
BITS 64
  org 0x400000

ehdr:           ; Elf64_Ehdr
  db 0x7f, 'ELF', 2, 1, 1, 0 ; e_ident
  times 8 db 0
  dw  2         ; e_type
  dw  0x3e      ; e_machine
  dd  1         ; e_version
  dq  _start    ; e_entry
  dq  phdr - $$ ; e_phoff
  dq  0         ; e_shoff
  dd  0         ; e_flags
  dw  ehdrsize  ; e_ehsize
  dw  phdrsize  ; e_phentsize
  dw  1         ; e_phnum
  dw  0         ; e_shentsize
  dw  0         ; e_shnum
  dw  0         ; e_shstrndx
ehdrsize  equ  $ - ehdr

phdr:           ; Elf64_Phdr
  dd  1         ; p_type
  dd  5         ; p_flags
  dq  0         ; p_offset
  dq  $$        ; p_vaddr
  dq  $$        ; p_paddr
  dq  filesize  ; p_filesz
  dq  filesize  ; p_memsz
  dq  0x1000    ; p_align
phdrsize  equ  $ - phdr

_start:
  mov rax, 1
  mov rdi, 1
  mov rsi, message
  mov rdx, 13
  syscall
  mov rax, 60
  xor rdi, rdi
  syscall

message: db 'hello, world', 0xa

filesize  equ  $ - $$
```

还是使用 nasm，不过这一次，我们使用 `nasm -f bin` 直接得到二进制程序。

最终结果是 170 个字节，这 170 字节的程序发送给任意的 x64 架构的 64 位 Linux，都可以打印出 hello world。

结束了，尘埃落定。

Final Binary Anatomy

最后我们来看一下这 170 字节中每一个字节是什么，在做什么，真正地做到对每一个字节都了然于胸。

```
# ELF Header
00:   7f 45 4c 46 02 01 01 00 # e_ident
08:   00 00 00 00 00 00 00 00 # reserved
10:   02 00 # e_type
12:   3e 00 # e_machine
14:   01 00 00 00 # e_version
18:   78 00 40 00 00 00 00 00 # e_entry
20:   40 00 00 00 00 00 00 00 # e_phoff
28:   00 00 00 00 00 00 00 00 # e_shoff
30:   00 00 00 00 # e_flags
34:   40 00 # e_ehsize
36:   38 00 # e_phentsize
38:   01 00 # e_phnum
3a:   00 00 # e_shentsize
3c:   00 00 # e_shnum
3e:   00 00 # e_shstrndx

# Program Header
40:   01 00 00 00 # p_type
44:   05 00 00 00 # p_flags
48:   00 00 00 00 00 00 00 00 # p_offset
50:   00 00 40 00 00 00 00 00 # p_vaddr
58:   00 00 40 00 00 00 00 00 # p_paddr
60:   aa 00 00 00 00 00 00 00 # p_filesz
68:   aa 00 00 00 00 00 00 00 # p_memsz
70:   00 10 00 00 00 00 00 00 # p_align

# Code
78:   b8 01 00 00 00          # mov    $0x1,%eax
7d:   bf 01 00 00 00          # mov    $0x1,%edi
82:   48 be 9d 00 40 00 00 00 00 00    # movabs $0x40009d,%rsi
8c:   ba 0d 00 00 00          # mov    $0xd,%edx
91:   0f 05                   # syscall
93:   b8 3c 00 00 00          # mov    $0x3c,%eax
98:   48 31 ff                # xor    %rdi,%rdi
9b:   0f 05                   # syscall
9d:   68 65 6c 6c 6f 2c 20 77 6f 72 6c 64 0a # 'hello, world\n'
```

可以发现 ELF Header 是 64 个字节，Program Header 是 56 字节，代码 37 个字节，最后 13 个字节是 `hello, world\n` 这个字符串数据。

从上面的反汇编中我们可以看出 x86-64 和 ARM 比起来一个显著的特点就是 x86-64 是变长指令集，每条指令的长度并不相等。长一点的 movabs 是 10 个字节，而短一点的 syscall 只有 2 个字节。

关于 x86-64，Intel 官方的手册 Intel® 64 and IA-32 Architectures Software Developer Manuals 十分十分详细，是每一个底层爱好者居家旅行的必备之物。

tiny-x64-helloworld 仓库中有上面每一步的代码和编译指令，供大家参考~

最后，编译、链接和装载互联网上有很多资料，这篇博客的目的并不是想要详细地去介绍这里面的知识，更多地是想作为一个楔子，帮助大家建立一个整体的认识，从而挑选自己感兴趣的部分去深入学习，祝大家 Happy Coding~