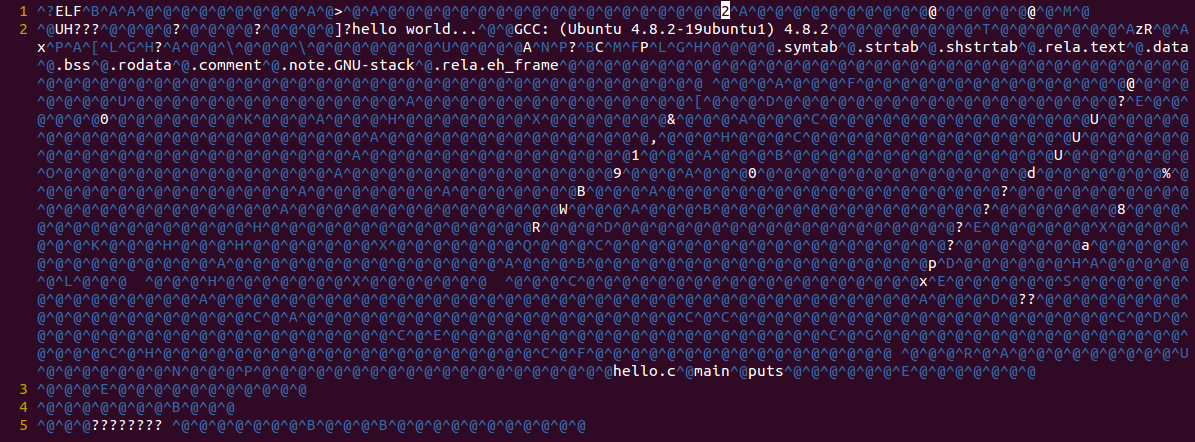
# Linux系统典型文件格式ELF

在Linux系统使用过程中，我们经常会看到elf32-i386、ELF 64-bit LSB等字样。那么究竟ELF是什么呢？

当我们使用gcc编译工具编译c程序会得到一个二进制的文件，想当然的使用vim编辑工具将其打开，结果看到如下内容：

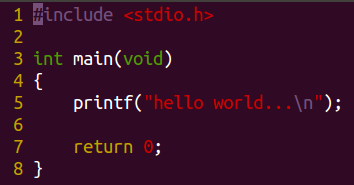


当然了，大部分同学不会这样做。数据是以二进制形式存储的，而vi只是一个文本编辑工具。那么数据究竟是怎样存储，以什么样的格式存储成二进制文件呢？是一个一个挨着排吗？从左向右，还是从右向左？这就需要我们深入了解下ELF文件了。

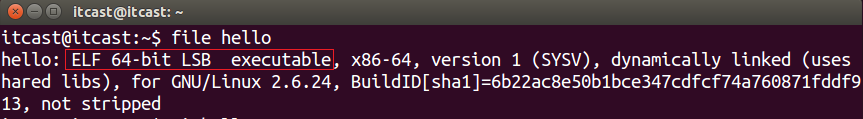
ELF文件格式是一个开放标准，各种UNIX系统的可执行文件都采用ELF格式，它有三种不同的类型：

1. 可重定位的目标文件（Relocatable，或者Object File）
2. 可执行文件（Executable）
3. 共享库（Shared Object，或者Shared Library）

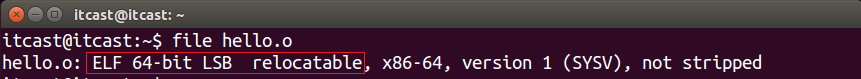
从我们最不畏惧的hello world入手吧。



很常见的，当我们gcc hello.c -o hello 编译这个c源程序的时候就得到了一个ELF格式的文件。可以使用file命令来查看。数据显示，该文件是一个64位的，小尾端存储的，可执行文件。



而当我们使用gcc -c hello.c -o hello.o编译生成的则是一个可重定位的目标文件，也可以使用file命令来查看它。



同样，我们也得到了一个ELF格式的文件。但是两者略有不同，前者是Executable可执行文件，而后者是可重定位的Relocatable。如果你感兴趣也可以试试共享库文件，其格式依然是ELF，或许会是这样ELF 32-bit LSB shared object。

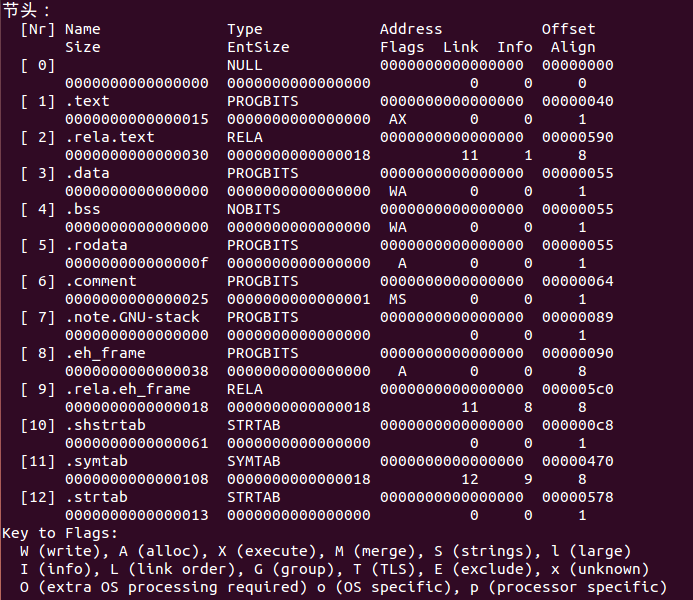
那么ELF文件内部是怎样存储数据的呢？当然不能再使用vi啦，我们可以使用readelf工具来查看下，以目标文件hello.o为例：readelf -a hello.o

输出结果大致可分为四个部分：ELF Header(ELF头)、Section Headers(节头表)、Relocation section(重定位节)、Symbol table(符号表)，我们依次来看。

第一部分，ELF Header描述整个ELF文件的数据存储概况，如操作系统是UNIX，体系结构是Advanced Micro Devices X86-64，数据存储是二进制补码，小尾端法存储，类型是可重定位文件，Section Header Table中有13个Section Header，从文件地址304开始，每个Section Header占64字节，这个目标文件没有程序头(Program Header)。



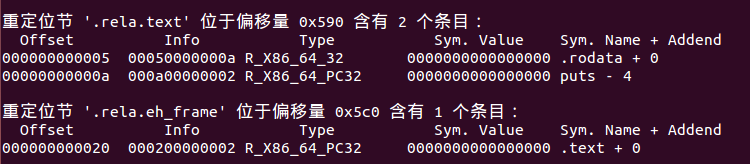
第二部分，挨着ELF头的数据信息是Section Headers(节头表)，顾名思义，它由一定数量的Section Header组成，可从中读出各个Section的描述信息，其中不乏我们编写的C程序源码、全局变量、常量等数据的存储位置。.text Section、.data Section、bss Section、.rodata Section都与我们的程序直接相关，而其它Section是汇编器自动添加的。 Address 是这些Section加载到内存中的地址（当然，程序中的地址都是虚拟地址），加载地址要在链接时填写，现在空缺，由于目标文件尚未做链接操作，所以是全0。 Offset 和 Size 列指出了各Section的起始文件地址和长度。比如 .data 段从文件地址0x55开始，一共0个字节，因为测试的程序中没有定义全局变量，只使用printf函数打印了“hello world…\n”所以后面的 .rodata Section大小为0xf也就是15个字节。



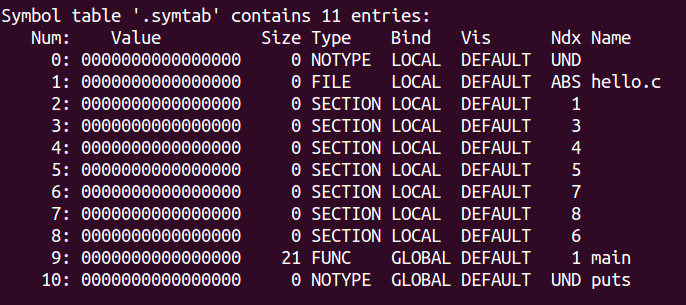
我们知道，C语言的全局变量如果在代码中没有初始化，就会在程序加载时用0初始化。这种数据属于 .bss ，在加载时它和 .data一样都是可读可写的数据，但是在ELF文件中 .data中若有数据则需要占用一部分空间保存初始值，而 .bss却不需要。也就是说，.bss在文件中只占一个Section Header而没有对应的Section，程序加载时 .bss 占多大内存空间在Section Header中描述。在我们这个例子中没有用到 .bss ，因此size也是0。

特别指出的是，.shstrtab 和 .strtab 这两个Section中存放的都是ASCII码，因此，在本文起始使用vi打开的ELF文件，如果仔细看，是能够看到字符串的，而并非通篇皆是“^@”等怪异字符。.shstrtab的全称应该是“Section Header String Table”用来保存各个Section的名字。.strtab Section保存程序中用到的符号的名字，每个名字都是以 '\0' 结尾的字符串。

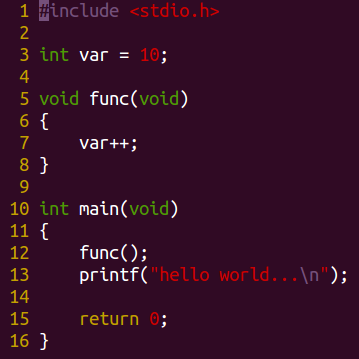
第三部分，可重定位节。该内容主要针对链接器设定，旨在告诉链接器指令中的哪些地方需要做重定位。当链接器完成链接工作后会自动将该Section删除。



第四部分，.symtab 是符号表。 我们在编写程序时定义的变量、函数都是符号，main就是符号的典型代表。当然为了保证程序能正常的编译、加载执行，编译器还帮助我们加入了其他许多必要的符号。这些符号都在.symtab中有所体现。

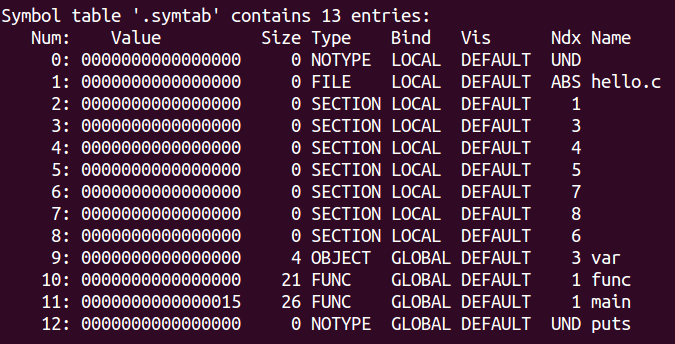


Ndx 列是每个符号所在的Section编号，各Section的编号在Section Header Table中有列出。 Value 列是每个符号所代表的地址，在目标文件中，符号地址都是相对于该符号所在Section的相对地址，如定义全局变量var，那么该符号在.symtab中的Value则是相对于.data Section开头的位置。 main 位于 .text 段的开头，所以地址也是0。但是上例中所有的Value都是0不易看出差异，所以我们适当的修改下我们的测试程序，添加一个初始化为非0的全局变量var和一个函数func。



这时.data Section的Size已经不再为0了，因为我们定义了全局变量var，它是一个int类型的变量，存储于.data Section上，因此 .data Section的Size应该是4，请大家自己验证吧。

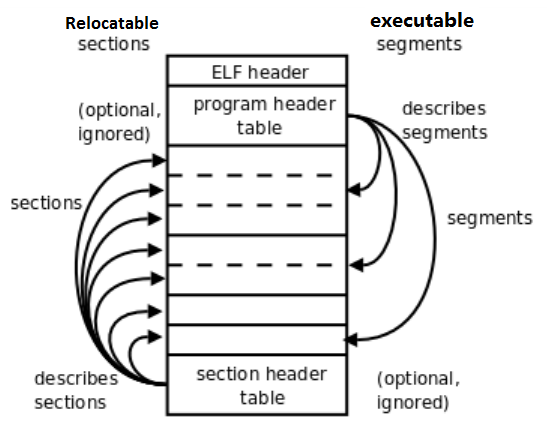
我们继续来看.symtab的变化。由于加入了两个符号var和func，所以 .symtab表的成员多了两个。var是全局变量，存储于.data Section中，编号在Ndx中指出，为3，由于只有这一个全局变量，所以var在的Value为0，相对于 .data Section开头的位置；符号main发生了变化，main是函数名，保存于.text Section中，编号为1，但其Value却不再是0，由于程序中还有另外一个符号func，所以符号main的Value由原来的0变为15，依然是相对于.text Section 起始位置而言。



但请大家注意，Symbol table ‘.symtab’ 中Value记录的是符号对应的值的位置。var是一个变量，值是数据位于.data中，func和main是函数，对应的值是函数入口地址(或者说函数首行指令的地址)，位于.text中。而“var”、“func”、“main”这些符号名本身存在哪里呢？其实这个问题我们在前文阐述过，这些字符串本身保存在 .strtab中。这样来看 .strtab和 .shstrtab的地位是等同的，差别是前者保存程序中用到的符号，而后者保存Section名称。

其实，ELF格式提供了两种不同的视角，链接器把ELF文件看成是Section的集合，而加载器把ELF文件看

成是Segment的集合。这里以Relocatable 的Section为例带大家分析了ELF的数据存储。大家可以结合可重定位Relocatable 的ELF文件数据存储的形式来了解Executable可执行文件的数据存储形式。而二者的关系可以从下图看出。



左边是从链接器的视角来看ELF文件，开头的ELF Header描述了体系结构和操作系统等基本信息，并指出Section Header Table和Program Header Table在文件中的位置，Program Header Table在链接过程中用不到，所以是可有可无的，Section Header Table中保存了所有Section的描述信息，通过Section Header Table可以找到每个Section在文件中的位置。

右边是从加载器的视角来看ELF文件，开头是ELF Header，Program Header Table中保存了所有Segment的描述信息，Section Header Table在加载过程中用不到，所以是可有可无的。从上图可以看出，一个Segment由一个或多个Section组成，这些Section加载到内存时具有相同的访问权限，如 .text Section会和 .rodata Section合并为一个Segment，同时分配只读访问权限，而.data Section通常和 .bss Section合并为一个Segment，分配读写权限。

有些Section只对链接器有意义，在运行时用不到，也不需要加载到内存，那么它可以不属于任何Segment， 如 .rela.text Section 在Executable文件中就消失了。另外，Section Header Table和Program Header Table并不是一定要位于文件的开头和结尾，其位置由ELF Header指出，上图这么画只是为了清晰。目标文件需要链接器做进一步处理，所以一定有Section Header Table；可执行文件需要加载运行，所以一定有Program Header Table；而共享库既要加载运行，又要在加载时做动态链接，所以既有Section Header Table又有Program Header Table。