<https://www.cnblogs.com/LiuYanYGZ/p/5535982.html>

**一 常用脚本**

1 打包脚本

脚本如下，下面附上ar 和 ranlib命令参考（命令来自于网络）

ALLLIB=\*.a  
FILE=`ls \*.a`  
#原来的库解压重命名

for F in $FILE  
do  
        ar x $F  
        OBJ=`ar t $F`  
        for O in $OBJ  
        do  
                mv $O ${F}\_${O}  
        done  
done

#ar c 创建一个库，ar r 插入文件。ar s ==ranlib 向库中插入文件或者更新库

ar cr $ALLLIB \*.o  
ranlib $ALLLIB  
mv $ALLLIB ../

mkdir -p tmp  
mv \*.o tmp

2 更新静态库

使用ar r

3 合并静态库

ar 高级用法---使用ar脚本

第一步：  
我们在命令终端中一次输入  
echo CREATE libyuerapi.a > ar.mac 回车  
echo SAVE >> ar.mac 回车  
echo END >> ar.mac 回车  
ar -M < ar.mac   
  
我们可一个通过cat ar.mac看到ar.mac文件中的内容，而且我们也可以看到有一个libyuerapi.a生成了。目前其实里面什么都没有。  
  
第二步：  
上一步我们已经成功的创建了libyuerapi.a文件，现在我们向其中添加.o文件  
ar -q libyuerapi.a yuer1.o  
ar -q libyuerapi.a yuer2.o  
ar -q libyuerapi.a yuer3.o  
  
第三步：  
把libyucom.a添加到libyuerapi.a库文件中  
我们以同样的方式创建一个ar.mac文件  
echo OPEN libyuerapi.a > ar.mac 回车  
echo ADDLIB libyucom1.a >> ar.mac 回车  
echo SAVE >> ar.mac 回车  
echo END >> ar.mac 回车  
ar -M < ar.mac 回车

**二 具体说明**

当我们的程序中有经常使用的模块，而且这种模块在其他程序中也会用到，这时按照软件重用的思想，我们应该将它们生成库，使得以后编程可以减少开发代码量。这里介绍两个命令ar和nm，用来对库操作。

**ar基本用法**

　当我们的程序中有经常使用的模块，而且这种模块在其他程序中也会用到，这时按照软件重用的思想，我们应该将它们生成库，使得以后编程可以减少开发代码量。这里介绍两个命令ar和nm，用来对库操作。

　　ar命令可以用来创建、修改库，也可以从库中提出单个模块。库是一单独的文件，里面包含了按照特定的结构组织起来的其它的一些文件（称做此库文件的member）。原始文件的内容、模式、时间戳、属主、组等属性都保留在库文件中。

　　下面是ar命令的格式：

　　ar [-]{dmpqrtx}[abcfilNoPsSuvV] [membername] [count] archive files...

　　例如我们可以用**ar rv libtest.a hello.o hello1.o** 来 生成一个库，库名字是test，链接时可以用-ltest链接。该库中存放了两个模块hello.o和hello1.o。选项前可以有‘-'字符，也可以 没有。下面我们来看看命令的操作选项和任选项。现在我们把{dmpqrtx}部分称为操作选项，而[abcfilNoPsSuvV]部分称为任选项。

　　{dmpqrtx}中的操作选项在命令中只能并且必须使用其中一个，它们的含义如下：

* d：从库中删除模块。按模块原来的文件名指定要删除的模块。如果使用了任选项v则列出被删除的每个模块。
* m：该操作是在一个库中移动成员。当库中如果有若干模块有相同的符号定义(如函数定义)，则成员的位置顺序很重要。如果没有指定任选项，任何指定的成员将移到库的最后。也可以使用'a'，'b'，或'I'任选项移动到指定的位置。
* p：显示库中指定的成员到标准输出。如果指定任选项v，则在输出成员的内容前，将显示成员的名字。如果没有指定成员的名字，所有库中的文件将显示出来。
* q：快速追加。增加新模块到库的结尾处。并不检查是否需要替换。'a'，'b'，或'I'任选项对此操作没有影响，模块总是追加的库的结尾处。如果使用了任选项v则列出每个模块。 这时，库的符号表没有更新，可以用'ar s'或ranlib来更新库的符号表索引。
* r：在库中插入模块(替换)。当插入的模块名已经在库中存在，则替换同名的模块。如果若干模块中有一个模块在库中不存在，ar显示一个错误消息，并不替换其他同名模块。默认的情况下，新的成员增加在库的结尾处，可以使用其他任选项来改变增加的位置。
* t：显示库的模块表清单。一般只显示模块名。
* x：从库中提取一个成员。如果不指定要提取的模块，则提取库中所有的模块。

　　下面在看看可与操作选项结合使用的任选项：

* a：在库的一个已经存在的成员后面增加一个新的文件。如果使用任选项a，则应该为命令行中membername参数指定一个已经存在的成员名。
* b：在库的一个已经存在的成员前面增加一个新的文件。如果使用任选项b，则应该为命令行中membername参数指定一个已经存在的成员名。
* c：创建一个库。不管库是否存在，都将创建。
* f：在库中截短指定的名字。缺省情况下，文件名的长度是不受限制的，可以使用此参数将文件名截短，以保证与其它系统的兼容。
* i：在库的一个已经存在的成员前面增加一个新的文件。如果使用任选项i，则应该为命令行中membername参数指定一个已经存在的成员名(类似任选项b)。
* l：暂未使用
* N：与count参数一起使用，在库中有多个相同的文件名时指定提取或输出的个数。
* o：当提取成员时，保留成员的原始数据。如果不指定该任选项，则提取出的模块的时间将标为提取出的时间。
* P：进行文件名匹配时使用全路径名。ar在创建库时不能使用全路径名（这样的库文件不符合POSIX标准），但是有些工具可以。
* s：写入一个目标文件索引到库中，或者更新一个存在的目标文件索引。甚至对于没有任何变化的库也作该动作。对一个库做ar s等同于对该库做ranlib。
* S：不创建目标文件索引，这在创建较大的库时能加快时间。
* u：一般说来，命令ar r...插入所有列出的文件到库中，如果你只想插入列出文件中那些比库中同名文件新的文件，就可以使用该任选项。该任选项只用于r操作选项。
* v：该选项用来显示执行操作选项的附加信息。
* V：显示ar的版本。

 对于每一个符号，nm列出其值(the symbol value)，类型（the symbol type）和其名字(the symbol name)。

例如，

  对于每一个符号，nm列出其值(the symbol value)，类型（the symbol type）和其名字(the symbol name)。如下例：  
00000024 T cleanup\_before\_linux  
00000018 T cpu\_init  
00000060 T dcache\_disable  
00000054 T dcache\_enable  
0000006c T dcache\_status  
00000000 T do\_reset  
0000003c T icache\_disable  
00000030 T icache\_enable  
00000048 T icache\_status  
    上面的显示是使用nm cpu.o的输出，对于cleanup\_before\_linux这个符号来说，00000024是以16进制显示的其值，T为其类型，而cleanup\_before\_linux是其名字。可以看出，上面显示的cleanup\_before\_linux这个symbol的值实际上是该函数在text section中的偏移。但是，每个符号的值的具体含义依其类型而异。当然，对于每个符号的值，其类型、其值以及它们所属的section是密切相关的。

总结：

|  |  |
| --- | --- |
| 符号  类型 | 说明 |
| A | 该符号的值是绝对的，在以后的链接过程中，不允许进行改变。这样的符号值，常常出现在中断向量表中，例如用符号来表示各个中断向量函数在中断向量表中的位置。 |
| B | 该符号的值出现在非初始化数据段(bss)中。例如，在一个文件中定义全局static int test。则该符号test的类型为b，位于bss section中。其值表示该符号在bss段中的偏移。一般而言，bss段分配于RAM中 |
| C | 该符号为common。common symbol是未初始话数据段。该符号没有包含于一个普通section中。只有在链接过程中才进行分配。符号的值表示该符号需要的字节数。例如在一个c文件中，定义int test，并且该符号在别的地方会被引用，则该符号类型即为C。否则其类型为B。 |
| D | 该符号位于初始话数据段中。一般来说，分配到data section中。例如定义全局int baud\_table[5] = {9600, 19200, 38400, 57600, 115200}，则会分配于初始化数据段中。 |
| G | 该符号也位于初始化数据段中。主要用于small object提高访问small data object的一种方式。 |
| I | 该符号是对另一个符号的间接引用。 |
| N | 该符号是一个debugging符号。 |
| R | 该符号位于只读数据区。例如定义全局const int test[] = {123, 123};则test就是一个只读数据区的符号。注意在cygwin下如果使用gcc直接编译成MZ格式时，源文件中的test对应\_test，并且其符号类型为D，即初始化数据段中。但是如果使用m6812-elf-gcc这样的交叉编译工具，源文件中的test对应目标文件的test,即没有添加下划线，并且其符号类型为R。一般而言，位于rodata section。值得注意的是，如果在一个函数中定义const char \*test = “abc”, const char test\_int = 3。使用nm都不会得到符号信息，但是字符串“abc”分配于只读存储器中，test在rodata section中，大小为4。 |
| S | 符号位于非初始化数据区，用于small object。 |
| T | 该符号位于代码区text section。 |
| U | 该符号在当前文件中是未定义的，即该符号的定义在别的文件中。例如，当前文件调用另一个文件中定义的函数，在这个被调用的函数在当前就是未定义的；但是在定义它的文件中类型是T。但是对于全局变量来说，在定义它的文件中，其符号类型为C，在使用它的文件中，其类型为U。 |
| V | 该符号是一个weak object。 |
| W | The symbol is a weak symbol that has not been specifically tagged as a weak object symbol. |
| - | 该符号是a.out格式文件中的stabs symbol。 |
| ? | 该符号类型没有定义 |

更新静态库的符号索引表   
  
本小节的内容相对简单。前边提到过，静态库文件需要使用“ar”来创建和维护。当给静态库增建一个成员时（加入一个.o文件到静态库中），“ar”可直接 将需要增加的.o文件简单的追加到静态库的末尾。之后当我们使用这个库进行连接生成可执行文件时，链接程序“ld”却提示错误，这可能是：主程序使用了之 前加入到库中的.o文件中定义的一个函数或者全局变量，但连接程序无法找到这个函数或者变量。   
  
这个问题的原因是：之前我们将编译完成的.o文件直接加入到了库的末尾，却并没有更新库的有效符号表。连接程序进行连接时，在静态库的符号索引表中无法定 位刚才加入的.o文件中定义的函数或者变量。这就需要在完成库成员追加以后让加入的所有.o文件中定义的函数（变量）有效，完成这个工作需要使用另外一个 工具“ranlib”来对静态库的符号索引表进行更新。   
  
我们所使用到的静态库（文档文件）中，存在这样一个特殊的成员，它的名字是“\_\_.SYMDEF”。它包含了静态库中所有成员所定义的有效符号（函数名、 变量名）。因此，当为库增加了一个成员时，相应的就需要更新成员“\_\_.SYMDEF”，否则所增加的成员中定义的所有的符号将无法被连接程序定位。完成 更新的命令是：   
ranlib ARCHIVEFILE   
通常在Makefile中我们可以这样来实现：   
libfoo.a: libfoo.a(x.o) libfoo.a(y.o) ...   
ranlib libfoo.a

它所实现的是在更新静态库成员“x.o”和“y.o”之后，对静态库的成员“\_\_.SYMDEF”进行更新（更新库的符号索引表）。   
  
如果我们使用GNU ar工具来维护、管理静态库，我们就不需要考虑这一步。GNU ar本身已经提供了在更新库的同时更新符号索引表的功能（这是默认行为，也可以通过命令行选项控制ar的具体行为。可参考 GNU ar工具的man手册）。

GNU工具中ar是用来制作库文件.a的，但同时还提供了一个ranlib，从手册上看ranlib相当于ar -s，为什么这样呢？

这是由于最早在Unix系统上ar程序是单纯用来打包多个.o到.a（类似于tar做的事情），而不处理.o里的符号表。Linker程序则需 要.a文件提供一个完整的符号表，所以当时就写了单独的ranlib程序用来产生linker所需要的符号信息。也就是说，产生一个对linker合 格的的.a文件需要做ar和ranlib两步 。

很快，Unix厂商就发现ranlib做得事情完全可以合并到ar里面去，于是ar程序的升级版本就包括了ranlib的功能，但早期的很多项目的Makefile都已经是按照两步式的方法生成.a，所以为了保证这些早期文件的兼容性，ranlib被保留下来了。

如今，GNU/Linux系统上，ranlib依然存在，当然大部分项目已经不使用它了，因为ar -s就做了ranlib的工作。  
历史通常是进步和妥协的混合！