1．addr2line 能够把程序地址转换为文件名和行号，前提是这个可执行文件包括调试符号

```
1 #include <stdio.h>
2
3 void foo()
4 {
5       printf("The address of foo() is %p\n",foo);
6 }
7 int main()
8 {
9       foo();
10      return 0;
11 }
```

运行如下命令，得到：

```
linux@ubuntu:~/A8/test/addr2line$ gcc -g test.c -o test
linux@ubuntu:~/A8/test/addr2line$ ls
test  test.c
linux@ubuntu:~/A8/test/addr2line$ ./test
The address of foo() is 0x80483c4
```

现在，我们可以用这一地址来看看 addr2line 是如何使用的。在终端中运行如下命令，从命令的运行结果可以看出，addr2line 工具正确指出了 0x80483c4 所对应的程序的具体位置以及所对应的函数名。在调用 addr2line 工具时，要使用 -e 选项来指定可执行映像是 test。通过使用 -f 选项，可以告诉工具输出函数名。

```
linux@ubuntu:~/A8/test/addr2line$ addr2line 0x80483c4 -f -e test
foo
/home/linux/A8/test/addr2line/test.c:4
```

2．nm 可以列出目标文件中的符号。用法虽然简单，但是功能很强大。符号是指函数名或变量。

nm 所列出的每一行有三部分组成：第一列是指程序运行时的符号所对应的地址，对于函数则地址表示的是函数的开始地址，对于变量则表示变量的存储地址；第二列是指对应符号放在哪一个段；而最后一列则是指符号的名称。在前面我们讲解 addr2line 时，我们提到 addr2line 是将程序地址转换成这一地址所对应的具体函数是什么，而 nm 则是全面的列出这些信息。但是，nm 不具备列出符号所在源文件及其行号这一功能，因此，我们说每一个工具有其特定功能。

字母	说　明
A	表示符号所对应的值是绝对的且在以后的连接过程中也不会改变
B 或 b	表示符号位于未初始化的数据段（.bss 段）中
C	表示没有被初始化的共公符号
D 或 d	表示符号位于初始化的数据段（.data 段）中
N	表示符号是调试用的
p	表示符号位于一个栈回朔段中
R 或 r	表示符号位于只读数据段（.rdata 段）中
T 或 t	表示符号位于代码段（.text 段）中
U	表示符号没有被定义

为了更清楚的理解 nm 中的符号和我们程序中的关系，我们看一下下列程序其所对应的 nm 输出结果。

```
1 #include <stdio.h>
  2 int gloable1;
  3 int gloable2 = 9;
  4
  5 static int static_gloable1;
  6 static int static_gloable2 = 99;
  7
  8 void foo()
  9 {
 10         static int intermal1;
 11         static int intermal2 = 999;
 12 }
 13
 14 static void bar()
 15 {
 16
 17 }
 18
 19 int main()
 20 {
 21         int local;
 22         int local2 = 9999;
 23         foo();
 24         return 0;
 25 }
```

```
linux@ubuntu:~/A8/test/objdump$ nm -n test.o
00000000 t foo
00000000 D gloable2
00000000 b static_gloable1
00000004 C gloable1
00000004 b intermal1.1708
00000004 d static_gloable2
00000005 t bar
00000008 d intermal2.1709
0000000a T main
linux@ubuntu:~/A8/test/objdump$
```

从 nm 输出的信息，我们可以看出：

- 不论一个静态变量是定义在函数内还是函数外，其在程序段中的分配方式都是一样的。如果这一静态变量是初始化好的，那么被分配在 data 段中，否则就在 bss 段中
- 非静态的全局变量，其分配的段也是和是否初始化有关。如果被初始化了，分配在 data 段中，否则就在 bss 段中。
- 函数无论是静态还是非静态的，其总是被分配在 text 段，但 T(t)的大小写代表这一符号所对应的函数是否是静态函数。
- 函数内的局部变量并不是分配在 data，bss 和 text 段中，其分配在栈上，nm 是看不到的。

3． readelf –h test

这条命令查看可执行文件“test”的 section 的头信息。

```
linux@ubuntu:~/A8/test/nm$ readelf -h test
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x80482e0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          5644 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         8
  Size of section headers:           40 (bytes)
  Number of section headers:         37
  Section header string table index: 34
```

每一个头信息都是一个 Elf32_Shdr 结构，其成员含义如下所述:

```
typedef struct {
    Elf32_Word sh_name;         //指定了这个 section 的名字
    Elf32_Word sh_type;         //把 sections 按内容和意义分类
    Elf32_Word sh_flags;        //sections 支持位的标记，用来描述多个属性
    Elf32_Addr sh_addr;     //该 section 在内存中的位置
    Elf32_Off  sh_offset;       //该 section 的字节偏移量
    Elf32_Word sh_size;     //该 section 的字节大小
    Elf32_Word sh_link;     //该 section 报头表的索引连接
    Elf32_Word sh_info;     //保存着额外的信息
    Elf32_Word sh_addralign;        //地址对齐的约束
    Elf32_Word sh_entsize;      //保存着一张固定大小入口的表
 } Elf32_Shdr;
```

4．ar 用来管理档案文件，在嵌入式系统当中，ar 主要用来对静态库进行管理。现在先让我们看看静态库里有些什么。我们采用 lib.a 为例，对其用 ar -t 来查看，如下所示：

```
linux@ubuntu:~/A8/test/nm$ cp /home/linux/A8/linux-2.6.35/lib/lib.a .
linux@ubuntu:~/A8/test/nm$ ar -t lib.a
argv_split.o
cmdline.o
ctype.o
dec_and_lock.o
decompress.o
decompress_bunzip2.o
decompress_inflate.o
decompress_unlzma.o
decompress_unlzo.o
dump_stack.o
extable.o
flex_array.o
idr.o
int_sqrt.o
ioremap.o
irq_regs.o
is_single_threaded.o
klist.o
kobject.o
kobject_uevent.o
kref.o
plist.o
prio_heap.o
prio_tree.o
proportions.o
radix-tree.o
ratelimit.o
rbtree.o
reciprocal_div.o
rwsem-spinlock.o
sha1.o
show_mem.o
string.o
vsprintf.o
linux@ubuntu:~/A8/test/nm$
```

采用 GNU 工具集进行开发时，一个静态库其实是将所有的.o 文件打成一个档案包。现在，我们就来看看如何使用 ar 来生成静态库。

foo.c

```
#include   <stdio.h>
void foo()
{
    printf("This is foo()\n");
}
```

bar.c

```
#include   <stdio.h>
void   bar()
{
    printf("This is bar()\n");
}
```

我们希望将 foo（）和 bar（）函数做成一个库，为此先要将它们编译成.o 目标文件。有了目标文件之后，我们采用 ar 命令来生成 libmy.a 库，如下所示。其中，ar 的 c

参数表示创建一个档案文件，而 r 参数表示增加文件到创建的库文件中，s 参数是为了生成库索引以提高连接速度。

```
linux@ubuntu:~/A8/test/ar$ gcc -c foo.c
linux@ubuntu:~/A8/test/ar$ gcc -c bar.c
linux@ubuntu:~/A8/test/ar$ ar crs libmy.a foo.o bar.o
linux@ubuntu:~/A8/test/ar$
```

现在可以在当前目录下看到一个 libmy.a 文件，这就是我们的静态库。下面验证库是否可用，

```
#include <stdio.h>
2 int  main()
3
4 {
5        foo();
6        bar();
7        return 0;
8 }
```

编译我们的验证程序，并与 libmy.a 进行连接，运行程序，从运行的运行结果可以看出，我们的 libmy.a 是起作用的。

```
linux@ubuntu:~/A8/test/ar$ gcc main.c libmy.a
linux@ubuntu:~/A8/test/ar$ ls
a.out  bar.o  foo.o  libmy.a  main.o  mylib.a
bar.c  foo.c  lib.a  main.c   mylib
linux@ubuntu:~/A8/test/ar$ ./a.out
This is the foo
This is the bar
linux@ubuntu:~/A8/test/ar$
```

如果想删除档案文件中的文件，我们可以使用 d 参数。下面是使用 d 参数删除 libmy.a 中的 foo.o 文件。从操作的最后结果看，当执行完 d 操作后，libmy.a 中只存在一个 bar.o 文件了。

```
linux@ubuntu:~/A8/test/ar$ ar -d libmy.a foo.o
linux@ubuntu:~/A8/test/ar$ ar -t libmy.a
bar.o
linux@ubuntu:~/A8/test/ar$
```

现在总结一下 ar 的几个参数：采用 c 参数创建一个档案文件，r 参数表示向档案文件增加文件，t 参数用于显示档案文件中存在哪些文件，s 参数用于指示生成索引以加快查找速度，d 参数用于从档案文件中删除文件，最后 x 参数是用于从档案文件中解压文件。

5．objdump 可以显示一个或者更多目标文件的信息，主要用来反汇编。

如下所示（部分未显示）：

```
linux@ubuntu:~/A8/test/objdump$ objdump -d test

test:     file format elf32-i386


Disassembly of section .init:

08048274 <_init>:
 8048274:       55                      push   %ebp
 8048275:       89 e5                   mov    %esp,%ebp
 8048277:       53                      push   %ebx
 8048278:       83 ec 04                sub    $0x4,%esp
 804827b:       e8 00 00 00 00          call   8048280 <_init+0xc>
 8048280:       5b                      pop    %ebx
 8048281:       81 c3 74 1d 00 00       add    $0x1d74,%ebx
 8048287:       8b 93 fc ff ff ff       mov    -0x4(%ebx),%edx
 804828d:       85 d2                   test   %edx,%edx
 804828f:       74 05                   je     8048296 <_init+0x22>
 8048291:       e8 1e 00 00 00          call   80482b4 <__gmon_start__@plt>
 8048296:       e8 d5 00 00 00          call   8048370 <frame_dummy>
```

6．objcopy 可以进行目标文件格式转换。

arm-linux-objcopy --gap-fill=0xff -O srec u-boot u-boot.srec

arm-linux-objcopy --gap-fill=0xff -O binary u-boot u-boot.bin

7．size 工具很简单，就是列出程序文件中各段的大小。

```
1 #include <stdio.h>
 2 int gloable1;
 3 int gloable2 = 9;
 4
 5 static int static_gloable1;
 6 static int static_gloable2 = 99;
 7
 8 static void foo()
 9 {
10         static int intermal1;
11         static int intermal2 = 999;
12 }
13
14 static void bar()
15 {
16 }
17
18 int main()
19 {
20         int local;
21         int local2 = 9999;
22         foo();
23         return 0;
24 }
```

```
linux@ubuntu:~/A8/test/objdump$ size test.o
   text    data     bss     dec     hex filename
     35      12       8      55      37 test.o
linux@ubuntu:~/A8/test/objdump$
```

8．strip 用来丢弃目标文件中的全部或者特定符号，减小文件体积。对于嵌入式系统，这个命令必不可少。

9．strings 用来打印某个文件的可打印字符串。

全局符号与弱符号之间的区别主要有两点：

(1). 当链接编辑器组合若干可重定位的目标文件时，不允许对同名的 STB_GLOBAL 符号给出多个定义。 另一方面如果一个已定义的全局符号已经存在，出现一个同名的弱符号并不会产生错误。链接编辑器尽关心全局符号，忽略弱符号。 类似地，如果一个公共符号（符号的 st_shndx 中包含 SHN_COMMON），那么具有相同名称的弱符号出现也不会导致错误。链接编辑器会采纳公共定义，而忽略弱定义。

(2). 当链接编辑器搜索归档库（archive libraries）时，会提取那些包含未定义全局符号的档案成员。成员的定义可以是全局符号，也可以是弱符号。连接编辑器不会提取档案成员来满足未定义的弱符号。 未能解析的弱符号取值为 0。