本文翻译自 GNU 链接器的手册《Red Hat Enterprise Linux 4, Using ld,the Gnu Linker》的第四章。该章完整的介绍了 GNU 链接器脚本的语法与应用。我之所以翻译它是因为我想了解一些困扰过我的问题，比如 linker 是怎样安排可执行文件的内存布局的？在 Linux 下，普通应用程序的内存布局为什么是如此类似？程序为什么总被安排在 0x08048000 开始，就像在 Windows 下，总是开始于 4M 边界？如果我手工定制程序的内存布局，操作系统还能成功载入它吗？…

这一切都与怎么链接你的程序有关，而链接过程又是完全由链接器脚本控制的。

我们是程序员，靠写代码谋生，希望敲入的代码能够正常工作就好，对在代码后面的内幕其实并不太关心。但有时候由不得你，你还是得揭开这层幕布，去看下面“肮脏”的细节。如果你在这领域工作有年，却从来没有过这种需求，那真应该祝福你；但如果你时不时的需要与编译器/链接器作比较亲密的接触，那可能花点时间了解一下这两个天天伴随你，但依然不太熟悉的朋友，还是值得的。

下文中黑色的文字是原文的翻译，而蓝色的则是我的笔记。另外我在文后附录了 3 个实际的例子并对例子做了注释：

1. 链接 Linux 下应用程序的链接器脚本
2. 链接 Linux 内核的链接器脚本
3. 我所在公司的嵌入式系统用到的链接器脚本

翻译此文，我基于如下原则，如果直译出来是能看懂的中国话，我就直译，否则就意译。无论技术还是英文，本人水平有限，力争不误导大家，但如果阁下英文还可以，还是看原版比较好。当然可以参考我文中的笔记，希望能对你的理解有帮助。我的笔记基本都基于我的实际经验而来，还是有一定可信度的。^_^

特别提一句，如果你发觉我翻译得不对或不妥当的地方，请务必指出并发信给我。本人先在此作揖谢过了！在文后有我的 email 地址。收到你的来信是对我的翻译的最大认可，即使是指出错误亦如此。

链接器脚本

由源代码到生成可执行文件一般经历编译与链接两个过程，而每次链接都由链接器脚本控制，该脚本用链接器命令语言写成。链接器脚本的主要目的是描述输入文件的各个节应当怎样被放入输出文件并控制输出文件的内存布局。绝大部分链接器脚本也只是干这个。但当需要的时候，链接器脚本利用下面将要描述的命令来指导链接器执行许多其他操作。

链接器总要用到链接器脚本。如果你本人没有提供一个，则链接器会用一个已被编译入链接器可执行文件中的默认脚本。你可以用 –verbose 这个命令行选项来显示默认的链接器脚本。某些特定的命令行选项，比如 –r 或 –N，将影响默

认的链接器脚本。

ld –verbose 会列出链接器用到的默认链接器脚本。在 Linux 下，当你编译程序时一般用如下命令：

```
gcc –o hello helloworld.c
```

即编译经典的“hello world”，以生成可执行文件 hello。链接器 ld 是 gcc 启动的。下面即是 ld 默认的链接器脚本，在附录中会详细分析。

```
[wzhou@DEBUG wzhou]$ ld –verbose
GNU ld version 2.13.90.0.2 20020802
  Supported emulations:
   elf_i386
   i386linux
   elf_i386_glibc21
using internal linker script:
==================================================
/* Script for -z combreloc: combine and sort reloc sections */
OUTPUT_FORMAT("elf32-i386", "elf32-i386",
        "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)
SEARCH_DIR("/usr/i386-redhat-linux/lib");  SEARCH_DIR("/usr/lib");
SEARCH_DIR("/usr/local/lib"); SEARCH_DIR("/lib");
/* Do we need any of these for elf?
   __DYNAMIC = 0;    */
SECTIONS
{
  /* Read-only sections, merged into text segment: */
  . = 0x08048000 + SIZEOF_HEADERS;
  .interp        : { *(.interp) }
  .hash          : { *(.hash) }
  .dynsym        : { *(.dynsym) }
  .dynstr        : { *(.dynstr) }
  .gnu.version    : { *(.gnu.version) }
  .gnu.version_d  : { *(.gnu.version_d) }
  .gnu.version_r  : { *(.gnu.version_r) }
  .rel.dyn       :
    {
      *(.rel.init)
      *(.rel.text .rel.text.* .rel.gnu.linkonce.t.*)
      *(.rel.fini)
      *(.rel.rodata .rel.rodata.* .rel.gnu.linkonce.r.*)
      *(.rel.data .rel.data.* .rel.gnu.linkonce.d.*)
      *(.rel.tdata .rel.tdata.* .rel.gnu.linkonce.td.*)
```

```
    *(.rel.tbss .rel.tbss.* .rel.gnu.linkonce.tb.*)
    *(.rel.ctors)
    *(.rel.dtors)
    *(.rel.got)
    *(.rel.bss .rel.bss.* .rel.gnu.linkonce.b.*)
  }
.rela.dyn      :
  {
    *(.rela.init)
    *(.rela.text .rela.text.* .rela.gnu.linkonce.t.*)
    *(.rela.fini)
    *(.rela.rodata .rela.rodata.* .rela.gnu.linkonce.r.*)
    *(.rela.data .rela.data.* .rela.gnu.linkonce.d.*)
    *(.rela.tdata .rela.tdata.* .rela.gnu.linkonce.td.*)
    *(.rela.tbss .rela.tbss.* .rela.gnu.linkonce.tb.*)
    *(.rela.ctors)
    *(.rela.dtors)
    *(.rela.got)
    *(.rela.bss .rela.bss.* .rela.gnu.linkonce.b.*)
  }
.rel.plt       : { *(.rel.plt) }
.rela.plt      : { *(.rela.plt) }
.init          :
{
  KEEP (*(.init))
} =0x90909090
.plt           : { *(.plt) }
.text          :
{
  *(.text .stub .text.* .gnu.linkonce.t.*)
  /* .gnu.warning sections are handled specially by elf32.em.  */
  *(.gnu.warning)
} =0x90909090
.fini          :
{
  KEEP (*(.fini))
} =0x90909090
PROVIDE (__etext = .);
PROVIDE (_etext = .);
PROVIDE (etext = .);
.rodata        : { *(.rodata .rodata.* .gnu.linkonce.r.*) }
.rodata1       : { *(.rodata1) }
.eh_frame_hdr : { *(.eh_frame_hdr) }
/* Adjust the address for the data segment.  We want to adjust up
```

```
to
    the same address within the page on the next page up.  */
  . = DATA_SEGMENT_ALIGN(0x1000, 0x1000);
  /* Ensure the __preinit_array_start label is properly aligned.  We
    could instead move the label definition inside the section, but
    the linker would then create the section even if it turns out to
    be empty, which isn't pretty.  */
  . = ALIGN(32 / 8);
  PROVIDE (__preinit_array_start = .);
  .preinit_array    : { *(.preinit_array) }
  PROVIDE (__preinit_array_end = .);
  PROVIDE (__init_array_start = .);
  .init_array    : { *(.init_array) }
  PROVIDE (__init_array_end = .);
  PROVIDE (__fini_array_start = .);
  .fini_array    : { *(.fini_array) }
  PROVIDE (__fini_array_end = .);
  .data         :
  {
    *(.data .data.* .gnu.linkonce.d.*)
    SORT(CONSTRUCTORS)
  }
  .data1        : { *(.data1) }
  .tdata    : { *(.tdata .tdata.* .gnu.linkonce.td.*) }
  .tbss        : { *(.tbss .tbss.* .gnu.linkonce.tb.*) *(.tcommon) }
  .eh_frame      : { KEEP (*(.eh_frame)) }
  .gcc_except_table   : { *(.gcc_except_table) }
  .dynamic       : { *(.dynamic) }
  .ctors        :
  {
    /* gcc uses crtbegin.o to find the start of
       the constructors, so we make sure it is
       first.  Because this is a wildcard, it
       doesn't matter if the user does not
       actually link against crtbegin.o; the
       linker won't look for a file to match a
       wildcard.  The wildcard also means that it
       doesn't matter which directory crtbegin.o
       is in.  */
    KEEP (*crtbegin.o(.ctors))
    /* We don't want to include the .ctor section from
       from the crtend.o file until after the sorted ctors.
       The .ctor section from the crtend file contains the
       end of ctors marker and it must be last */
```

```
    KEEP (*(EXCLUDE_FILE (*crtend.o ) .ctors))
    KEEP (*(SORT(.ctors.*)))
    KEEP (*(.ctors))
  }
  .dtors         :
  {
    KEEP (*crtbegin.o(.dtors))
    KEEP (*(EXCLUDE_FILE (*crtend.o ) .dtors))
    KEEP (*(SORT(.dtors.*)))
    KEEP (*(.dtors))
  }
  .jcr           : { KEEP (*(.jcr)) }
  .got           : { *(.got.plt) *(.got) }
  _edata = .;
  PROVIDE (edata = .);
  __bss_start = .;
  .bss           :
  {
   *(.dynbss)
   *(.bss .bss.* .gnu.linkonce.b.*)
   *(COMMON)
   /* Align here to ensure that the .bss section occupies space up to
      _end.  Align after .bss to ensure correct alignment even if the
      .bss section disappears because there are no input sections.  */
   . = ALIGN(32 / 8);
  }
  . = ALIGN(32 / 8);
  _end = .;
  PROVIDE (end = .);
  . = DATA_SEGMENT_END (.);
  /* Stabs debugging sections.  */
  .stab          0 : { *(.stab) }
  .stabstr       0 : { *(.stabstr) }
  .stab.excl     0 : { *(.stab.excl) }
  .stab.exclstr  0 : { *(.stab.exclstr) }
  .stab.index    0 : { *(.stab.index) }
  .stab.indexstr 0 : { *(.stab.indexstr) }
  .comment       0 : { *(.comment) }
  /* DWARF debug sections.
     Symbols in the DWARF debugging sections are relative to the
beginning
     of the section so we begin them at 0.  */
  /* DWARF 1 */
  .debug         0 : { *(.debug) }
```

```
  .line         0 : { *(.line) }
  /* GNU DWARF 1 extensions */
  .debug_srcinfo  0 : { *(.debug_srcinfo) }
  .debug_sfnames  0 : { *(.debug_sfnames) }
  /* DWARF 1.1 and DWARF 2 */
  .debug_aranges  0 : { *(.debug_aranges) }
  .debug_pubnames 0 : { *(.debug_pubnames) }
  /* DWARF 2 */
  .debug_info     0 : { *(.debug_info .gnu.linkonce.wi.*) }
  .debug_abbrev   0 : { *(.debug_abbrev) }
  .debug_line     0 : { *(.debug_line) }
  .debug_frame    0 : { *(.debug_frame) }
  .debug_str      0 : { *(.debug_str) }
  .debug_loc      0 : { *(.debug_loc) }
  .debug_macinfo  0 : { *(.debug_macinfo) }
  /* SGI/MIPS DWARF 2 extensions */
  .debug_weaknames 0 : { *(.debug_weaknames) }
  .debug_funcnames 0 : { *(.debug_funcnames) }
  .debug_typenames 0 : { *(.debug_typenames) }
  .debug_varnames  0 : { *(.debug_varnames) }
}


==================================================
```

你可以通过用 -T 命令行选项来提供你自己的链接器脚本。当你这样做时，你的链接器脚本将替代默认的脚本。

```
你可以通过如下命令来指定自己的链接器脚本。

gcc -o hello -T my_linker_script helloworld.c
```

你也可以通过命名链接器脚本为输入文件给链接器的方式来隐含的指定链接器脚本。请参考 4.11 节隐含的链接器脚本。

基本的链接器脚本概念

为了描述链接器脚本语言，我们需要定义一些基本的概念和词汇。链接器把多个输入文件合并成单一的输出文件。输出文件和每个输入文件都有某种特定的数据格式，这种格式常被称为对象文件格式。每个文件都被称为一个对象文件，而输出文件经常被称为可执行文件。但为了我们描述的目的，我们将称它为对象文件。每个对象文件都有一个节的列表。我们有时候称输入文件中的节为输入节，类似的在输出文件中的节为输出节。

```
链接器（linker）的输入是编译器（compiler）的输出。而这里指的输入文件也就是俗
```

话说的 obj 文件，它是编译器编译源代码的结果。实际上这还是不太准确，gcc 编译器的输出是汇编码，汇编码再喂给汇编器（assembler），由汇编器翻译成 obj 文件（汇编器只能叫翻译了，因为比较简单）。obj 文件作为链接器的“输入文件”，在链接器脚本的控制下进行链接，得到可执行文件，即这里一再出现的“输出文件”。一般情况下，也就到此为止了，你可以运行该“输出文件”了。但对于嵌入式系统可能还有一步，即把该可执行文件转换成 bin 文件。原因是嵌入式系统没有常规可执行文件那样的运行时系统来支持，有的甚至本身就是运行时系统，这样可执行文件格式的有些东西对它而言是没有意义，甚至是有害的，所以要把这些无用的东西剥离，生成一个可执行的代码块。Linux 内核本身就是典型。

在对象文件中的每个节都有节名和大小，绝大部分节还有被称为节内容的相关数据块。一个节可以被标记成“要载入的”，意思是当输出文件要运行时，该节的内容应当被载入内存。一个没有任何内容的节可以是“要分配的”，意思是在内存中应当为它留出一块空间，但没有任何东西被载入到那儿（在某种状况下，该块空间可能被清零）。也有标记为既不“要载入的”又不是“要分配的”节，典型的是包含某种调试信息的节。

每个“要载入的”或“要分配的”输出节都有两种地址。第一种是 VMA，即虚拟内存地址（Virtual Memory Address），这是当输出文件运行时该节的地址。第二种是 LMA，即载入内存地址（Load Memory Address），这是该节将被载入的地址。在绝大部分情况下，这两种地址是相同的。他们可能不同的一个例子是数据节是被载入在 ROM 中的，当程序启动时要把它拷贝到 RAM 中去（这种技术在以 ROM 为基础的系统中常被用来初始化全局变量）。在这种情况下，该节在 ROM 中的地址为 LMA，而在 RAM 中的地址为 VMA。

通过 objdump 程序中的 -h 命令选项，你可以看到对象文件中的所有节。

```
[wzhou@DEBUG wzhou]$  gcc -c helloworld.c
只编译不链接生成 helloworld.o obj 文件，也就是这里说的输入文件。
[wzhou@DEBUG wzhou]$  objdump -h helloworld.o

helloworld.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000027  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  0000005c  2**2
```

```
                CONTENTS, ALLOC, LOAD, DATA
  2 .bss         00000000  00000000  00000000  0000005c  2**2
                ALLOC
  3 .rodata      0000000e  00000000  00000000  0000005c  2**0
                CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment     00000033  00000000  00000000  0000006a  2**0
                CONTENTS, READONLY
```

由于是 obj 文件，是一堆浮动代码，即根本还没有地址的概念（有也是相对地址），所以 LMA 与 VMA 都是零。

这里“ALLOC”就是“要分配的”，“LOAD” 就是“要载入的”。

.text 节存放代码，.data 节存放初始化数据，.bss 节存放未初始化数据。.comment 节就是即不“要载入的”，又不“要分配的”。

下面看一下被链接后的可执行文件

```
[wzhou@DEBUG wzhou]$  objdump –h helloworld

helloworld:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .interp       00000013  080480f4  080480f4  000000f4  2**0
                CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000020  08048108  08048108  00000108  2**2
                CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .hash         00000028  08048128  08048128  00000128  2**2
                CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .dynsym       00000050  08048150  08048150  00000150  2**2
                CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .dynstr       0000004c  080481a0  080481a0  000001a0  2**0
                CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .gnu.version  0000000a  080481ec  080481ec  000001ec  2**1
                CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .gnu.version_r 00000020  080481f8  080481f8  000001f8  2**2
                CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .rel.dyn      00000008  08048218  08048218  00000218  2**2
                CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .rel.plt      00000010  08048220  08048220  00000220  2**2
                CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .init         00000018  08048230  08048230  00000230  2**2
                CONTENTS, ALLOC, LOAD, READONLY, CODE
 10 .plt          00000030  08048248  08048248  00000248  2**2
                CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .text         000000fc  08048278  08048278  00000278  2**2
                CONTENTS, ALLOC, LOAD, READONLY, CODE
 12 .fini         0000001c  08048374  08048374  00000374  2**2
```

```
                CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .rodata      00000016  08048390  08048390  00000390  2**2
                CONTENTS, ALLOC, LOAD, READONLY, DATA
 14 .data        0000000c  080493a8  080493a8  000003a8  2**2
                CONTENTS, ALLOC, LOAD, DATA
 15 .eh_frame    00000004  080493b4  080493b4  000003b4  2**2
                CONTENTS, ALLOC, LOAD, DATA
 16 .dynamic     000000c8  080493b8  080493b8  000003b8  2**2
                CONTENTS, ALLOC, LOAD, DATA
 17 .ctors       00000008  08049480  08049480  00000480  2**2
                CONTENTS, ALLOC, LOAD, DATA
 18 .dtors       00000008  08049488  08049488  00000488  2**2
                CONTENTS, ALLOC, LOAD, DATA
 19 .jcr         00000004  08049490  08049490  00000490  2**2
                CONTENTS, ALLOC, LOAD, DATA
 20 .got         00000018  08049494  08049494  00000494  2**2
                CONTENTS, ALLOC, LOAD, DATA
 21 .bss         00000004  080494ac  080494ac  000004ac  2**2
                ALLOC
 22 .comment     00000132  00000000  00000000  000004ac  2**0
                CONTENTS, READONLY
 23 .debug_aranges 00000058  00000000  00000000  000005e0  2**3
                CONTENTS, READONLY, DEBUGGING
 24 .debug_pubnames 00000025  00000000  00000000  00000638  2**0
                CONTENTS, READONLY, DEBUGGING
 25 .debug_info   00000c85  00000000  00000000  0000065d  2**0
                CONTENTS, READONLY, DEBUGGING
 26 .debug_abbrev 00000127  00000000  00000000  000012e2  2**0
                CONTENTS, READONLY, DEBUGGING
 27 .debug_line   000001f2  00000000  00000000  00001409  2**0
                CONTENTS, READONLY, DEBUGGING
 28 .debug_frame  00000014  00000000  00000000  000015fc  2**2
                CONTENTS, READONLY, DEBUGGING
 29 .debug_str    0000098a  00000000  00000000  00001610  2**0
                CONTENTS, READONLY, DEBUGGING
```

经过链接后，代码就定址了，所以 VMA 与 LMA 都有值了。

凡是即不“要载入的”，又不“要分配的”节，象.comment，.debug_XXX 系列节，VMA 和 LMA 都没有意义，所以为零。

每个对象文件还有被称为符号表（symbol table）的符号列表。Symbol 可能被定义也可能没有被定义。每个 symbol 都有名字，每个被定义的 symbol 都有地址和其它信息。如果你把 C 或者 C++的程序编译成对象文件，对于每一个

定义的函数，全局变量或静态变量，你都会获得一个被定义的 symbol。每一个在输入文件中用到的未定义的函数或全局变量将会产生一个未定义的 symbol。

利用 nm 程序或带 -t 选项的 objdump 程序，你能够看到对象文件中的 symbol。

helloworld 的代码如下：

```
#include <stdio.h>

int main(int argc, char** argv)
{
    printf("Hello world!\n");
    return 0;
}
```

用 nm 来看一下该 obj 文件的符号表。

```
[wzhou@DEBUG wzhou]$ nm helloworld.o
00000000 T main
         U printf
```

这里有两个符号，main 和 printf。main 自然是代码中的 main 函数的 symbol，前面的 T 表示在代码段。而 printf 前的 U 表示这是一个被引用到但并没有被定义的 symbol。很正常，printf 自然不是定义在该处，而是在标准 C 库中。由于是 obj 文件，所以没有地址的概念，所以这里的 main 函数的地址栏位零。

再看一下生成的链接器的“输出文件”的 symbol

```
[wzhou@DEBUG wzhou]$ nm helloworld
080494ac A __bss_start
0804829c t call_gmon_start
080494ac b completed.1
08049484 d __CTOR_END__
08049480 d __CTOR_LIST__
080493a8 D __data_start
080493a8 W data_start
08048350 t __do_global_ctors_aux
080482c0 t __do_global_dtors_aux
080493ac d __dso_handle
0804948c d __DTOR_END__
08049488 d __DTOR_LIST__
080493b8 D _DYNAMIC
080494ac A _edata
080493b4 d __EH_FRAME_BEGIN__
080494b0 A _end
```

```
08048374 T _fini
08048390 R _fp_hw
080482fc t frame_dummy
080493b4 d __FRAME_END__
08049494 D _GLOBAL_OFFSET_TABLE_
       w __gmon_start__
08048230 T _init
08048394 R _IO_stdin_used
08049490 d __JCR_END__
08049490 d __JCR_LIST__
       w _Jv_RegisterClasses
       U __libc_start_main@@GLIBC_2.0
08048328 T main
080493b0 d p.0
       U printf@@GLIBC_2.0
08048278 T _start
```

搞大发了，怎么这么多 symbol?

因为链接器在连接时引入了其他的库和 obj 文件，自然连带着它们的 symbol 也进来了。

链接器脚本格式

链接器脚本是纯文本文件。

你写的链接器脚本包含了一系列命令。每条命令可跟带一些参数的关键字或者是对某个 symbol 的赋值。你可以用分号来分隔命令，而空格通常是被忽略的。

象文件或格式名这样的字符串通常能直接被写入脚本。如果文件名中包含有如逗号这样的字符，由于逗号同样也用于分隔文件名，所以你要在这种文件名上加双引号，在文件名中不能包含双引号。

你可以象在 C 语言中一样用/* 和*/在链接器脚本中包含注释。在语法上注释等同于空格，会被链接器忽略。

简单的链接器脚本例子

许多链接器脚本是很简单的。

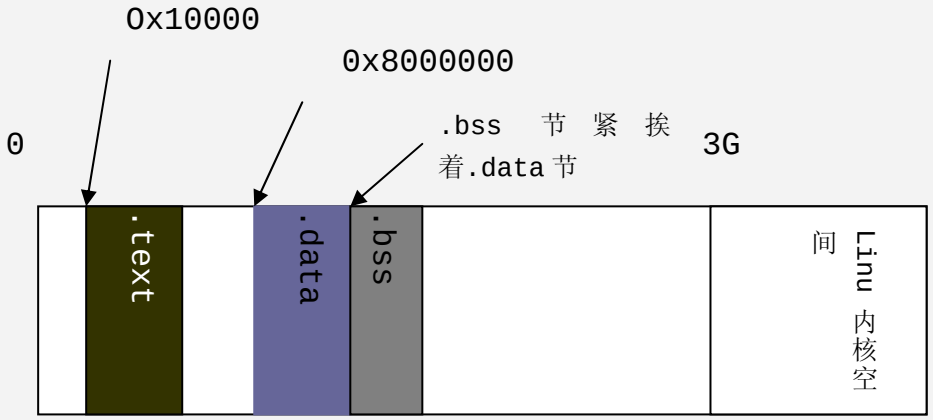
可能最简单的链接器脚本只有一条命令：**SECTIONS**。你用**SECTIONS**命令来描述输出文件的内存布局。

SECTIONS命令是一条强大的命令。在这里我们将描述它的一个简单应用。让我们假设你的程序只包含代码，初始化数据和未初始化数据，它们将分别在.text，.data和.bss节中。让我们进一步假设它们是你的输入文件中包含的仅有的节。

在这个例子中，让我们安排代码应当被载入 0x10000 地址处，而数据应当被载入 0x8000000 地址处。下面是用于这样做的链接器脚本：

```
SECTIONS
{
   . = 0x10000;
   .text : { *(.text) }
   . = 0x8000000;
   .data : { *(.data) }
   .bss : { *(.bss) }
}
```

相对 Linux 系统而言，上面的例子脚本将创建如下的内存映射：



如果.text 节的大小超过了 0x8000000 – 0x10000，则链接就会失败。在.text 节与.data 节之间有空隙，这是合法的。

你写的**SECTIONS**关键字就是**SECTIONS**命令，后面跟着一系列的symbol赋值和被括在花括号里的输出节的描述。

上面例子中在**SECTIONS**命令中的第一行为特殊的symbol.(dot)赋值，.是位置指示器。如果你不用某些方式（这些方式会在后面介绍）说明输出节的地址，则地址会被设置成位置指示器的当前值。位置指示器是随输出节的大小递增的。在**SECTIONS**命令的开始处，他的初始值为 0。

第二行定义了.text 输出节。在这里冒号是必需的，现在你可以忽略它。在输出节名后面的花括号中，你列出了输入节名，这些输入节应当被链接器放入输出节中。*是通配符，用于匹配任何输入文件。表达式 *(.text) 意思是所有输入文件中的所有节名为 .text 的输入节。

由于在定义.text 输出节时位置指示器被置成 0x10000，链接器将设置输出文件中的.text 节的地址从 0x10000 开始。

剩下的行定义了输出文件中的.data 和.bss 节。链接器将把.data 输出节放到 0x8000000 开始的地址处。在链接器放置好.data 输出节以后，位置指示器的值将被调整为 0x8000000 加上.data 输出节的大小。在内存中的效果就是链接器把.bss 输出节放在紧挨着.data 输出节的后面。

如果有必要，链接器会通过递增位置指示器来确保每个输出节对齐在必要的地址上。在上面的例子中，指定.text 和.data 节的地址几乎可以满足任何对齐限制，但链接器可能不得不在.data 与.bss 节之间创建一定的小间隙。

看，这就是一个简单但完整的链接器脚本。

简单的链接器脚本命令

在该节中我们将介绍简单的链接器脚本命令。

设置入口点

程序执行的第一条指令被称为入口点。你可以用**ENTRY**这条链接器脚本命令来设置入口点。该命令的参数是符号名：

ENTRY（symbol）

有好几种方法来设置程序的入口点。链接器将依次尝试用如下列出的方法来设置入口点，直到其中的一个成功为止。

- -e entry 的命令行选项
- 在链接器脚本中通过 ENTRY（symbol）来指定
- 如果定义了 start 符号
- .text 节第一个字节的地址，当然.text 节要存在
- 零地址

通过 ENTRY 设置的程序入口点是该程序开始运行的第一条指令，这句话不太准确。对静态链接的程序而言，这是正确的。当 OS 把程序载入内存后，就会设置该入口点为从内核态返回到用户态后执行的第一条指令。但对动态链接的程序而言，程序的真正入口点根本不在你写的代码中，而在 C 库中的动态链接器中。只有在动态链接器初始化好本身，并为你的程序载入隐含链接的动态库后，才会跳转到所谓的入口点。
对 C/C++程序员而言的入口点而言是 main 函数，这里的 ENTRY 命令设置的入口点对他们是隐性的，只有汇编程序员才要考虑这个问题。
在 ld 的默认链接器脚本中设置的入口点如下：

```
ENTRY(_start)
```

即入口点位于 symbol _start 处。_start 不在用户代码中，而是位于 C 库提供的包含启动代码的 obj 文件中。如果你想调试 main 函数以前的过程，则可以设置断点在该处，不过你看到的只能是汇编码了。

处理文件的命令

一些链接器脚本命令用来处理文件。

INCLUDE filename

在该点上包括链接器脚本文件filename。该文件将在当前目录或用-L选项

指定的任何目录内查找。你可以嵌套的调用**INCLUDE**，直到 10 层。

INPUT(file, file, …)

INPUT(file file …)

INPUT 命令指示链接器包括入指定的文件，即使这些文件在命令行上也指定了。

比如，如果你任何时候都要包括入subr.o文件，但你想避免每次都在命令行上指定该文件的麻烦，你就可以把**INPUT**(subr.o)这条命令写入你的链接器脚本。

实际上，只要你愿意，你可以在链接器脚本中列出所有的输入文件，然后只需要带-T 选项来调用链接器就可以了。

在sysroot前缀被配置，文件名以 / 字符开始，并且被处理的脚本在以sysroot为前缀的目录下时，由**INPUT**指定的文件将在sysroot前缀指定的目录下查找。否则，链接器试图在当前目录下打开指定的文件。如果链接器不能找到文件，它将在存放静态库文件目录下寻找。具体请见命令行选项一节的中关于对 –L 的介绍。[1]

如果你使用**INPUT**(-lfile)的形式，ld会把它转换成libfile.a的文件名，跟在命令行上用-l指定文件一样。

GROUP(file, file, …)

GROUP(file file …)

GROUP命令类似于**INPUT**命令，除了指定的文件都应是静态库文件，并且会反复查找指定的文件，直到没有新的未定义引用被创建。请见 3.1 节的命令行选项中的介绍。

[1] 这段翻译我基本上是意译，而非直译，希望没有理解错。

OUTPUT(filename)

OUTPUT 命令用来命名输出文件。在链接器脚本中应用 **OUTPUT**(filename)的作用同在命令行上的-o filename是相同的(请参见 3.1 节命令行选项)。如果两者都指定了，则命令行上的优先。

你可以用 OUTPUT 命令来定义默认的输出文件名。因为通常情况下在不指定输出文件名时，默认的文件名为 a.out。

SEARCH_DIR(path)

SAERCH_DIR 命令添加指定的 path 到 ld 查找的静态库路径列表中。应用 SEARCH_DIR(path)的效果同在命令行上指定-L path 是相同的(请参见 3.1 节的命令行选项的介绍)。如果两者都指定，则链接器会查找两者指定的路径，但在命令行上指定的路径优先查找。

```
在 ld 的默认链接器脚本中有
SEARCH_DIR("/usr/i386-redhat-linux/lib");  SEARCH_DIR("/usr/lib");
SEARCH_DIR("/usr/local/lib"); SEARCH_DIR("/lib");
```

STARTUP(filename)

STARTUP命令类似于**INPUT**命令，除了该命令指定的filename将变成在链接时被链接的第一个输入文件的功能，就像该文件是在命令行上第一个被指定的一样。在某些系统中，当程序的入口点总是在第一个文件的开始处时可能有用。

处理对象文件格式的命令

一些链接器脚本命令用于处理对象文件格式。

OUTPUT_FORMAT(bfdname)

OUTPUT_FORMAT(default, big, little)

OUTPUT_FORMAT 命令用于指定输出文件的 BFD 格式（请参见第六章

BFD）。用 OUTPUT_FORMAT(bfdname)完全就同在命令行上指定 -oformat bfdname 一样（请参见 3.1 节 命令行选项）。如果两者都指定，命令行选项优先。

你可以用带三个参数的 OUTPUT_FORMAT 命令并基于 -EB 与 -EL 命令行选项来指定不同的格式。这将可以让链接器脚本按用户意愿来设置输出格式。

如果命令行既没有指定 -EB 也没有指定 -EL,则输出格式即是该命令第一个参数(default)指定的格式。如果在命令行上指定了 -EB，则输出格式将被设置成第二个参数(big)指定的格式。如果在命令行上指定了 -EL-，则输出格式将被设置成第二个参数(little)指定的格式。

比如，MIPS ELF 目标的默认的链接器脚本中就有下面的命令：

```
UTPUT_FORMAT(elf32-bigmips, elf32-bigmips, elf32-littlemips)
```

这就是说输出文件的默认格式是 elf-bigmips，但如果用户在命令行上输入了 -EL，输出文件将按 elf-littlemips 格式来创建。

```
在 ld 的默认链接器脚本中有
OUTPUT_FORMAT("elf32-i386", "elf32-i386",
        "elf32-i386")
这样你在命令行上是否指定-EB 或-EL 都一样。
```

TARGET(bfdname)

TARGET命令用于在读入输入文件时指定BFD格式。它会影响到该命令之后的 **INPUT** 和 **GROUP** 命令。该命令类似于在命令行上指定的 -b bfdname(请参见 3.1 节 命令行选项)。如果在脚本中有TARGET命令而没有OUTPUT_FORMAT命令，则最后一次出现的TARGET命令同样用于指定输出文件的格式。

另外的链接器脚本命令

下面再介绍一些另外的链接器脚本命令。

ASSERT(exp, message)

用于确认 exp 是否非零。如果该表达式(exp)为零，则链接器停止链接并返回一个错误号，同时打印 message 消息。

EXTERN(symbol symbol …)

该命令使得symbol成为输出文件中的未定义符号。它可以触发对标准函数库另外模块的链接。你可以在单个**EXTERN**中列多个symbol，也可以一个**EXTERN**指定一个symbol，并多次应用该命令来达到同样的效果。该命令同命令行选项-u有相同的效果。

这同在 C/C++代码中的功用差不多。

FORCE_COMMON_ALLOCATION

该命令同在命令行上指定 -d 选项是相同的。即使是对一个可重定位的输出文件，ld 也要为 common symbol 指定空间。

INHIBIT_COMMON_ALLOCATION

该命令同在命令行上指定 -no-define-common 选项是相同的。即使对非重定位的输出文件，ld 也不为 common symbol 指定空间。

NOCROSSREFS(section section …)

该命令可用于让 ld 发现在特定的输出节之间有任何的引用时报错。

对于某类特定的程序，尤其是在用到了覆盖技术的嵌入式系统上的程序，当某个节被载入内存时，即意味另一个节将不能被载入。任何在这两个节之间的直接引用将是错误的。比如在某个节内的代码调用另一个节中定义的函

数，那将是一个错误。

NOCROSSREFS命令指定的是输出节的节名。如果ld检测到任何指定的输出节之间的任何引用，它都将报错并返回一个非零的退出码。注意，**NOCROSSREFS**命令指定的是输出节名，而不是输入节名。

恕我孤陋寡闻，在内存便宜如斯的今天，我实在不知道现在还有什么系统要用到 overlay 技术。这里的意思是，如果 A section 有函数 a_func，而 B section 有函数 b_func，同时由于内存紧张，在同一时间，系统只能要么载入 A section，要么载入 B section，反正不能同时载入。则很显然，A section 中的代码不能访问 B section 中的 b_func，同样的 B section 中的代码不能访问 a_func。NOCROSSREFS 命令显然是为这种情况设计的，以便让链接器在链接时能检查出这种相互引用（cross reference）的问题。
用命令表示上面例子中的问题，即是：

NOCROSSREFS(A B)
这里 A 与 B 都是最终可执行文件中的节，而不是要被链接的 obj 文件中的节。

OUTPUT_ARCH(bfdarch)

指定特定的输出 CPU 架构。这里的参数是 BFD 库用到的架构名之一(请参见第六章 BFD)。你可以用带-f 选项的 objdump 工具来查看对象文件是什么架构的。

为 symbol 赋值

你可以在链接器脚本中为 symbol 赋值。这将定义一个全局的 symbol 符号。

简单赋值

你可以用 C 语言中的任何赋值操作来为 symbol 赋值。

- symbol = expression ;
- symbol += expression ;
- symbol -= expression ;
- symbol *= expression ;
- symbol /= expression ;
- symbol <<= expression ;
- symbol >>= expression ;
- symbol &= expression ;
- symbol |= expression ;

上面的第一行定义 symbol 符号，并对其赋值为 expression。在其他行，symbol 符号必须已经被定义，它的值将根据对应操作而被调整。

特殊的符号名.(dot)用来表示位置指示器。你只能在**SECTIONS**命令中用到它。

在表达式 expression 后面的分号(;)是必需的。

表达式定义在文后，请参见 4.10 节链接器脚本中的表达式。

你可以这样对符号赋值，或者写在命令的等号的右边，或者作为 SECTIONS 命令内部的声明，或者像在介绍 SECTIONS 命令是那样作为输出节的一部分。

The section of the symbol will be set from the section of the expression; for more information, refer to Section 4.10.6 The Section of an Expression.

下面是一个例子，在该例中列出了符号赋值可能被用到的三种情况。

```
floating_point = 0;
SECTIONS
{
   .text :
   {
      *.(.text)
      _etext = .;
   }
   _bdata = (. + 3) & ~ 3;
   .data : { *(.data) }
}
```

在上例中，符号 floating_point 被定义成零，符号_etext 被定义成紧接着最后一个.text 输入节的地址，而符号_bdata 则被定义成紧接着.text 输出节并对齐在 4 字节边界的地址。

PROVIDE 命令

在某些情况下，有这样一种需求，在被链接的所有对象文件中谁也没有定义某个符号，但该符号却又被代码用到了，则实际上是链接器定义了该符号。比如传统的链接器会定义符号etext，而ANSI C却要求程序员把etext作为一个函数名，在一般情况下这当然会报错。而**PROVIDE**关键字就能够解决这种尴尬，因为它只在程序中没有定义但却又引用到该符号的情况下定义该符号，就像etext符号那样。该命令的语法是这样的，**PROVIDE**(symbol = expression)。

下面是一个利用**PROVIDE**来定义etext的例子：

```
SECTIONS
{
   .text :
      {
         *(.text)
         _etext = .;
         PROVIDE(etext = .);
      }
}
```

在上例中，如果程序定义了_etext(带下划线的 etext 符号)，链接器将报多重定义的错。另外，如果程序定义了 etext(不带下划线的 etext 符号)，链接器将默认用到程序员的定义。但如果在代码中没有定义该符号却又用到了该符号，则链接器将用脚本中的 etext 符号的定义。

SECTIONS 命令

SECTIONS命令告诉链接器怎样把输入节组织称输出节，并怎样在内存中布局输出节。

SECTIONS命令的格式如下：

```
SECTIONS
{
   sections-command
   sections-command
   ...
}
```

每行 section-command 可以是如下几种之一：

- **ENTRY**命令(请参见 4.4.1 节设置入口点)
- 符号赋值(请参见 4.5 节为符号赋值)
- 输出节的描述
- 覆盖技术(overlay)描述

ENTRY命令和符号赋值只允许出现在**SECTIONS**命令内部，同时为了方便，这些命令中可以用到位置计数器，即.(dot)。因为你可以在生成输出文件的布局的合适的地方利用这些命令，这样可以让脚本更易于理解。
输出节与覆盖技术在下面介绍。

如果在你的脚本中没有用到**SECTIONS**命令，链接器将按照在输入文件中碰到的节的顺序把它放入输出文件的同名的输出节中。例如，如果在第一个输入文件中包含了其他输入文件的所有输入节名，则输出文件的节的顺序将同第一个文件中的节的顺序相同。第一个节总是开始于零地址。

输出节介绍

完整的输出节的格式看起来是这样的：

```
section [address] [(type)] : [AT(lma)]
{
   output-section-command
   output-section-command
   ...
} [>region] [AT>lma_region] [:phdr :phdr …] [=fillexp]
```

当然很少有输出节会用全上面列出的大部分的可选节属性。

为了避免节名与后面的参数混淆，section 后的空白是必需的。冒号与花括号在这里也是必需的，而换行与其他空白字符则是可选的。
每行 output-section-command 可以是下面的命令之一：

- 为符号赋值(请参见 4.5 节为符号赋值)
- 输入节描述(请参见 4.6.4 节输入节介绍)
- 直接包括的数据(请参见 4.6.5 节输出节数据)
- 特殊的输出节关键字(请参见 4.6.6 节输出节关键字)

输出节的节名

这里的 section 是输出节的节字。Section 必须符合你的输出文件格式的限制。在只支持有限数量的节名的格式中，比如 a.out 格式的可执行文件，节名必须取自该格式支持的节名中(比如 a.out 就只支持.text，.data 和.bss)。

If the output format supports any number of sections, but with numbers and not names (as is the case for Oasys), the name should be supplied as a quoted numeric string。

对于上面一段原文中的“but with numbers and not names”实在无法理解，哪位高手能否指点一下。

节名可以是包含字符，但如果含有特殊字符，例如逗号，则节名需要用双引号引起来。

节名为/DISCARD/的输出节是特殊的节，具体请参见 4.6.7 节丢弃输出节。

输出节介绍

Address 参数是输出节的 VMA（虚拟内存地址）的表达式。如果你不提供 address 参数，在 region 参数存在的情况下，链接器把该节设置成基于 region 的地址，否则（region 没有指定）就把它设置成位置指示器的当前值。

如果你没有提供 address 参数，输出节的地址将被设置成你指定的值。如果你 address 参数和 region 参数两者都没有指定，则输出节的起始地址将被设置成位置计数器的当前值，同时该值会被调整到输出节要求的对齐地址上。输出节对起始地址对齐的要求是包含在该输出节中的输入节对齐的条件中最严格的。
比如，

```
.text . : { *(.text) }
```

和

```
.text : { *(.text) }
```

是有些微差别的。前者设置.text 输出节的地址为位置指示器的当前值，而后者同样把.text 输出节的地址设置为位置指示器的当前值，而位置指示器的当前值的对齐要求已被调整为各个输入节中要求最严格的。

上面一段有点费解，可能是我中文水平比较差，不能用简洁的母语来表达，只能用注释来说明。

前者对输出节起始地址的没有特殊要求，位置指示器当前值在哪儿就哪儿。而后者则是有较高要求。比如输入节 input_a_section 的对齐要求是 0x10，输入节 input_b_section 的对齐要求是 0x100，位置指示器当前值为 0x800010，则前者的输出节的起始地址为 0x800010，而后者的输出节的起始地址为 0x800100，被调整到输入节中更严格的对齐边界上。

参数 address 可以是任意的表达式，请参见 4.10 节链接器脚本中的表达式。

例如，你想使节对齐在 0x10 字节的边界上，也就是该节起始地址的最低 4 位为零，你可以这样做：

```
.text ALIGN(0x10) : { *(.text) }
```

ALIGN 把位置指示器的当前值向上取整到指定的值。在脚本中 address 参数会改变位置指示器的值。

如果位置指示器的当前值为 0x15，则 ALIGN(0x04)，会把其值向上调整为 0x18。所谓向上，即使往高地址方向对齐。

输入节介绍

输出节中最常看到的命令是对输入节的描述命令。
对输入节的描述是链接器脚本中最基本的操作。你通过输出节来告诉链接器怎样布局你的程序的内存安排，而你用输入节描述命令来告诉链接器怎样把输入文件

组织入你的内存布置。

输出节的布局决定你的程序的内存布局，而输入节描述则是指导链接器怎样把输入文件中的各个节放入输出文件的节中。

输入节基础

输入节描述命令包括可选的文件名和其后括在括号里的输入节的列表。文件名和节名可以用通配符，具体在后面介绍（请参见 4.6.4.2 节输入节名通配符）。最常用的输入节描述命令是把特定节名的所有输入节放入输出节中。例如，把所有输入文件中的.text 节放入输出节中，你可以这样写：

```
*(.text)
```

这里的*匹配所有的输入文件。如果要从匹配的文件名中排除一些文件，则 **EXCLUDE_FILE**命令可以用于排除某些特定的文件。例如：

```
(*(EXCLUDE_FILE (*crtend.o *otherfile.o) .ctors))
```

将把除了 crtend.o 和 otherfile.o 外的所有输入文件中的.ctors 节放入输出节。

有两种方法来在一个输出节中放入多种输入节，象：

```
*(.text .rdata)
*(.text) *(.rdata)
```

这两种方法的不同之处在于各个输入文件中的.text 与.rdata 输入节最终在输出节中出现的次序不同。在前者的例子中，输入节是就象它们在脚本中的次序一样是交错放置入输出节的。而在后者的例子中先是放置所有的.text 输入节，然后才是所有的.rdata 输入节。

比如 A 文件有.text，.rdata 节(标记为.text-A，.rdata-A)，B 文件有.text，.rdata 节（标记为.text-B，.rdata-B），C 文件有.text，.rdata 节（标记为.text-C，.rdata-C)。则针对前者的输出节是这样的：

```
.text-A
.rdata-A
.text-B
.rdata-B
.text-C
```

```
.rdata-C

而针对后者的输出节是这样的：

.text-A
.text-B
.text-C
.rdata-A
.rdata-B
.rdata-C
```

你可以通过指定输入文件名来包含如该文件的输入节。如果你想指定把某些输入文件中的含有特殊数据的节放入内存中特定的位置，那就需要这样做。例如：

```
data.o(.data)
```

如果文件名后没有指定输入节名，则该输入文件的所有节将被放入该输出节中。这一般很少用到，但有时候可能有用。例如：

```
data.o
```

当你指定的文件名不包括任何通配字符时，链接器将首先查看该文件名是否在命令行上指定或者由**INPUT**命令指定。如果没有，链接器将试图把该文件作为输入文件打开，就像它在命令行上指定的一样。注意，这是与通过**INPUT**命令指定的有区别的，因为在这种情况下，链接器不会在库文件的搜索目录中去查找该文件（通过**INPUT**命令指定的话则会）。

输入节名通配符

在输入节描述命令中，文件名与节名两部分都可以有通配符。

象在上面很多例子中看到的文件名中的*就是一种简单的文件名通配符。

这里用到的通配符与 Unix Shell 中的差不多。

*

匹配任何字符与任何数目的字符

?

匹配任何单个字符

[chars]

匹配指定的 chars 中的任何一个字符；“-”字符可以用于说明一定的范围，象[a-z]用于匹配任何小写字母。

\

Escape 字符，即引用紧跟在后的字符。

文件名中的通配符字符不会匹配“/”字符（在 Unix 系统中，该字符用于分隔目录名）。但单个的“*”是个例外，它将匹配任何文件名，无论是否包含“/”字符。而在输入节名中，通配符字符可以匹配“/”字符，即没有在文件名中的限制。

```
*(.text)
将匹配任何指定的输入文件，比如 a.o，obj/b.o(这里 obj 是目录名)，../obj/c.o 等等。这里由于文件名部分是单个的“*”通配符字符，所以匹配一切输入文件（输入文件当然可以带目录指定）

aaa*.o(.text)
将匹配 a.o，但 obj/aaab.o 就不能匹配了，因为它包含了路径分割符“/”。
```

文件名通配符只会匹配在命令行上和用**INPUT**命令显示指定的文件，链接器不会搜索目录来扩大匹配的范围。

如果文件名能匹配脚本中多行带通配符的输入文件名描述，链接器将用脚本中第一行来匹配（最早优先）。像下面的例子中，输入节描述命令的顺序可能就有问题，因为包含 data.o 的规则不会被用到（即第二行）：

```
.data : { *(.data) }
.data1 : { data.o(.data) }
```

上面的脚本有问题是因为满足第二行的也满足第一行，又根据最早优先原则，第一行将被采用，所以在输出文件中不会有.data1 这一节。

正常情况下，链接器会按在链接时它看到的次序（也就是在脚本中出现的次序）来把与通配符匹配的文件与节放入输出文件，但你可以通过**SORT**关键字来改变它。**SORT**关键字放置在括住通配符表达式的括号的前面，像SORT(.text*)。

当用到**SORT**关键字后，链接器会对输入的文件名或节名按升序来排列，在把它们放入输出文件中。

下面的例子演示了通配符被利用来分别放置输入文件各节的情况。该链接器脚本指导链接器把所有输入文件中.text 节放入(输出文件的).text 节，所有输入

文件中.data 节放入(输出文件的).data 节。链接器还把以大写字母开头的所有输入文件中的.data 节放入(输出文件的).DATA 节，而除此之外（即文件名不是以大写字母开头）的文件中的.data 节放入(输出文件的).data 节。

```
SECTION {
    .text : { *(.text) }
    .DATA : { [A-Z]*(.data) }
    .data : { *(.data) }
    .bss : { *(.bss) }
}
```

输入节与常见符号（common symbols）

对常用符号需要特殊的记号，因为在许多对象文件格式中，常用符号并没有一个特定的输入节对应。链接器对待常用符号就像它们是存放在一个节名叫 COMMON 的输入节中一样。

你可以把带有 COMMON 节的文件象同其他输入节的文件一样对待。你可以把来自特定输入文件的常用符号放到这个节中，而把来自另一个输入文件的常用符号放入另一个节中。

在绝大部分情况下，输入文件的常用符号将被放到输出文件的.bss 节。例如：

```
.bss { *(.bss) *(COMMON) }
```

有些对象文件格式有多种类型的常用符号。例如，MIPS ELF 对象文件格式区分标准常用符号和小常用符号(small common symbols)。在这种情况下，链接器将为另外类型的常用符号指定不同的较特殊的节名。就 MIPS ELF 而言，链接器命名标准的常用符号为 COMMON，而小常用符号为.scommon。这样就允许你把不同类型的常用符号映射到内存的不同位置。

有时候你会在老的链接器脚本中看到[COMMON]的命令描述。这个记号现在已经

不用了，它等同于现在的 *(COMMON)。

输入节和垃圾收集

When link-time garbage collection is in use (-gc-sections), it is often useful to mark sections that should not be eliminated. This is accomplished by surrounding an input section's wildcard entry with KEEP(), as in KEEP(*(.init)) or KEEP(SORT(*)(.ctors)).

很遗憾，这一小节，我就根本不知道作者想说明什么。语焉不详而又没有背景知识，只能不翻译了。哪位大侠若能指教，请发信给我。我这里先谢过。

输入节例子

下面的例子是一个完整的链接器脚本文件。它告诉链接器从文件 all.o 中读入所有的节并把它们放入开始于地址 0x10000 处的节名为 outputa 的输出节的起始处。在该节中紧挨着放入的是文件 foo.o 的.input1 节。在其后是 outputb 输出节，里面依次放入文件 foo.o 的.input2 节和文件 foo1.o 的.input1 节。剩下的所有其他文件的.input1 和.input2 节将被放入 outputc 输出节中。

```
SECTIONS {
   outputa 0x10000 :
   {
      all.o
      foo.o (.input1)
   }
   outputb :
   {
      foo.o (.input2)
      foo1.o (.input1)
   }
```

```
    outputc :
    {
        *(.input1)
        *(.input2)
    }
}
```

输出节数据

通过应用**BYTE**，**SHORT**，**LONG**，**QUAD**或**SQUAD**这类输出节命令，你可以显式的在输出节里包含你想放入的数据。这些关键字的后面跟着括在圆括号里的表达式，而这些表达式的值就是你想放入的数据（请参见 4.10 节连接器脚本中的表达式）。表达式的值被放入位置指示器当前指向的位置。

BYTE，**SHORT**，**LONG**，**QUAD**分别用于在输出节中存入一个，两个，四个和八个字节。在存入这些字节后，位置指示器将增加相应存入的字节数以指向后面。

例如，下面的脚本先存入内容为 1 的一个字节，然后是存入内容为符号 addr 地址的 4 个字节。

```
BYTE(1)
LONG(addr)
```

当你用的是 64 位机时，**QUAD**和**SQUAD**两条命令是相同的，他们都存入 8 个字节，64 位值。当在 32 位机上时，表达式按 32 位值来计算。在这种情况下，**QUAD**命令存入一个以零扩展（即无符号扩展）到 64 位的 32 位值，而**SQUAD**命令则存入一个有符号扩展的 64 位的 32 位值。

上面两句是直译，意思可能是在 32 位机上，QUAD 存入的是 4 个字节，但该值是经过无符号扩展到 64 位后的 32 位值；而 SQUAD 存入的也是 4 个字节，但该值是经过有符号扩展到 64 位后的 32 位值。

何为有符号扩展与无符号扩展，找本汇编书看看吧。

如果输出文件的对象文件格式指明了是数据是按什么尾序（所谓大尾小尾，intel

CPU是小尾，SPARC CPU是大尾，有的CPU大小尾可以在启动时设置）的次序存储的，这也是常规情况，则值将按指定的尾序存放。当对象文件格式没有指明尾序时，比如像真实的例子，S-records（我不知道这S-records是啥东东），值将被按第一个输入文件中的尾序存放。

注意，这些命令只能被用在节描述的内部，而不能用在两个节的中间，象下面的脚本会引起链接器报错：

```
SECTIONS {.text : { *(.text) } LONG(1) .data : { *(.data) } }
```

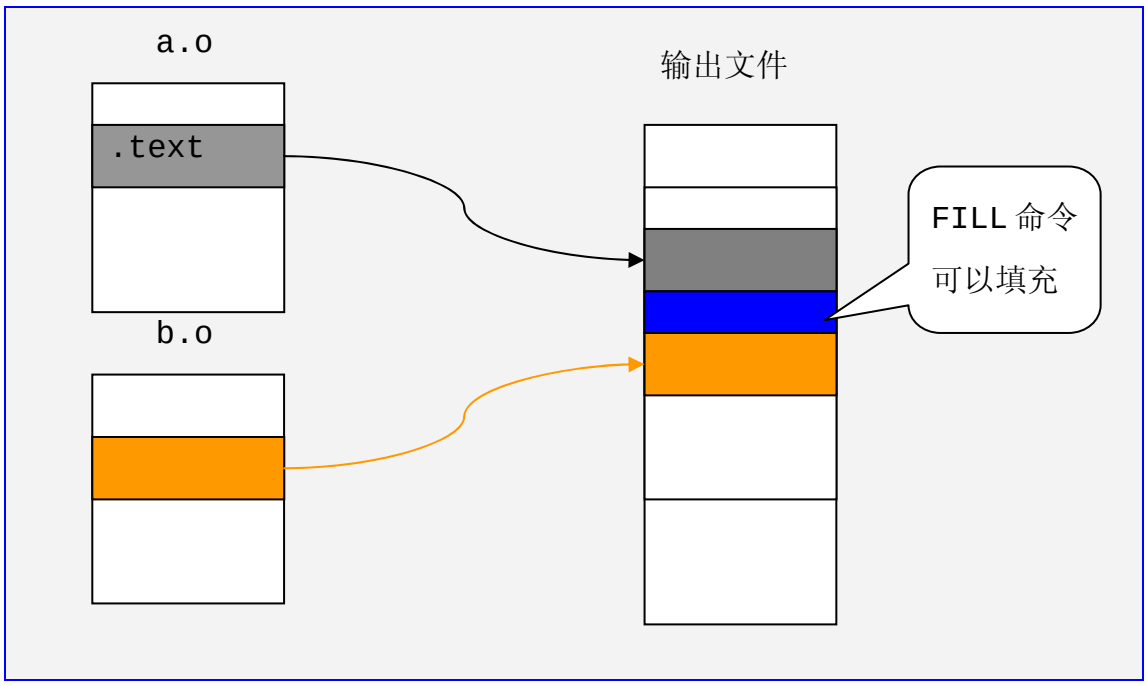
上面的脚本的想法是在.text 输出节与.data 输出节之间放入 4 个字节的空间，内容为 1，但由于 LONG 命令不再任何输出节内，而是在两个输出节之间，所以不符合语法。

而下面的则能工作：

```
SECTIONS {.text : { *(.text); LONG(1) } .data : { *(.data) } }
```

LONG 命令位于输出节.text 内。虽然从效果上看，这四字节的“1”物理上还是处于.text 与.data 之间，但逻辑上该“1”是属于.text 节的。

你可以用**FILL**命令来填充当前的节，该命令后面跟着括在括号里的表达式。在输出节内没有特别指定的区域（比如，由于输入节在映射入输出节时为满足对齐要求而留出的空隙）将被反复填充以该表达式的值。



FILL命令从它声明处的位置开始有效，一直到下一个**FILL**命令或该节结束。

通过在一节中引入多个**FILL**命令，你可以在一个输出节的不同部分填充以不同的内容。

下面的例子演示了怎样在内存的未指定区域已 0x90 填充：

```
FILL(0x90909090)
```

FILL命令与输出节属性**=fillexp**类似，但它只影响输出节中该命令后的部分，而节属性则是对整节都有效的。如果两者在脚本中都指定了，**FILL**命令优先。请参见 4.6.8.5 节输出节填充，里面有关于填充表达式的详细介绍。

输出节关键字

在输出节中可以包含一些关键字。

CREATE_OBJECT_SYMBOLS

该命令指导链接器为每一个输入文件生成对应符号。每个符号的名字就是对应输入文件的文件名。每个这种符号所在的节就是 CREATE_OBJECT_SYMOLS 命令所在的节。这在 a.out 对象格式文件中是常例，但在其他对象格式中使用得很少。

CONSTRUCTORS

当链接a.out格式的对象文件时，为了支持C++中的全局构造函数与全局析构函数，链接器会用一种较少用到的方式来构造。当要链接的对象文件格式不支持任意的节（即只支持预定义的一些节），象ECOFF和XCOFF格式，链接器将根据名字自动地识别全局构造函数与析构函数。对于这些对象文件格式，**CONSTRUCTORS**命令指导链接器把构造函数相关的信息放到该命令所在的输出节中。在其他对象文件格式中该命令被忽略。

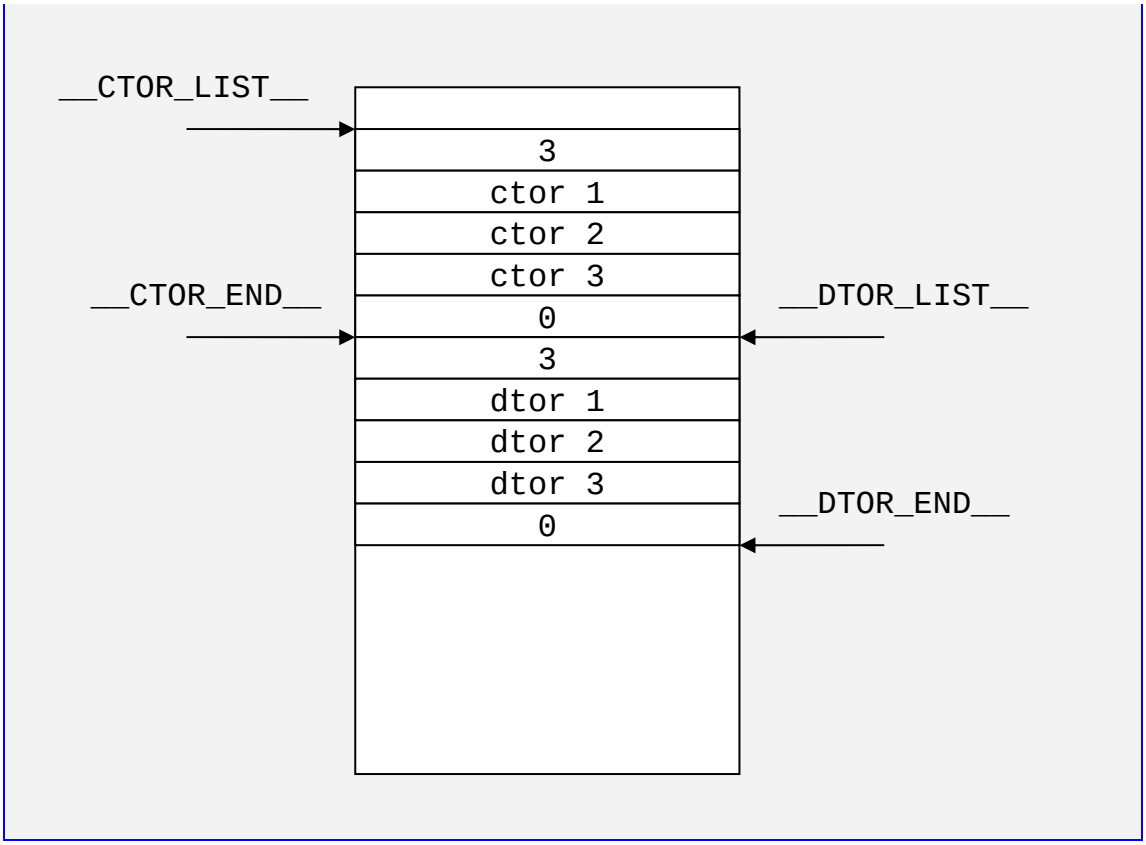
符号 __CTOR_LIST__ 标记全局构造信息列表的头，而符号 __DTOR__LIST__标记该列表的尾。该列表的第一项（4 个字节）是该列表的项数，后面的每一项是每个构造函数或析构函数的地址，最后以空项结尾。

编译器负责安排这些构造/析构的代码的位置（链接器只是把这些函数的地址放入列表中）。对于这些对象文件格式，gnu C++通常会从__main子过程中来调用构造函数，对__main的调用代码会被自动插入到main的启动代码中。通常情况下，gnu C++会在atexit函数中去调用析构函数，或者直接从函数exit中调用。

对于象 COFF 或 ELF 这种支持任意节名的对象文件格式，gnu C++通常会把全局构造函数的地址放入.ctors 节中，而全局析构函数的地址放入.dtors 节中。下面链接器脚本中的命令序列会构造出符合 gnu C++运行代码的有序列表。

```
__CTOR_LIST__ = .;
LONG((__CTOR_END__ - __CTOR_LIST__) / 4 – 2)
*(.ctors)
LONG(0)
__CTOR_END__ = .;
__DTOR_LIST__ = .;
LONG((__DTOR_END__ - __DTOR_LIST__) / 4 – 2)
*(.dtors)
LONG(0)
__DTOR_END__ = .;
```

这个例子中的 4 表示列表的每项占 4 个字节，而 2 是指该列表的头上一项和尾部的空项（即 LONG(0)的那项）。另外__CTOR_END__与__DTOR_LIST__其实是同一个位置两个标记而已。



如果你使用的gnu C++支持调整初始化顺序，即对全局构造函数的执行顺序提供一定控制，那你必须在链接阶段排序好构造函数，以确保它们能按照正确的顺序执行。方法是在原来用**CONSTRUCTORS**命令的地方，用**SORT**(**CONSTRUCTORS**)代替。如果构造函数与析构函数的地址分别是在 .ctors 与 .dtors 节中的，则用 *(**SORT**(.ctors)) 和 *(**SORT**(.dtors))来代替*(.ctors)和*(.dtors)。

通常情况下，编译器和链接器会自动处理这些问题，你不需要关心。但如果你用 C++并自己写链接器脚本的话，你可能需要考虑这些问题了。

```
$ cat construct.cpp
class CObj_1
{
public:
    int dummy_1;
    CObj_1()
    {
        dummy_1 = 1;
    }
```

```
    ~CObj_1()
    {
        dummy_1 = 0;
    }
};

class CObj_2
{
public:
    int dummy_2;
    CObj_2()
    {
        dummy_2 = 2;
    }
    ~CObj_2()
    {
        dummy_2 = 0;
    }
};

class CObj_3
{
public:
    int dummy_3;
    CObj_3()
    {
        dummy_3 = 3;
    }
    ~CObj_3()
    {
        dummy_3 = 0;
    }

};

CObj_1    obj_1;
CObj_2    obj_2;
CObj_3    obj_3;

int main(int argc, char** argv)
{
    return 0;
}
$ g++ -c construct.cpp  生成对象文件 construct.o
```

```
$ g++ -o construct construct.cpp  生成输出文件 construct

基于这个例子来分析一下 linker 是怎么对待构造、析构函数的。

$ objdump –h construct.o
  3 .gnu.linkonce.t._ZN6CObj_1C1Ev 0000000e  00000000  00000000  00000158  2**1
              CONTENTS, ALLOC, LOAD, READONLY, CODE, LINK_ONCE_DISCARD
  这个节只存放了 Cobj_1 的构造函数
  4 .gnu.linkonce.t._ZN6CObj_2C1Ev 0000000e  00000000  00000000  00000166  2**1
              CONTENTS, ALLOC, LOAD, READONLY, CODE, LINK_ONCE_DISCARD
  这个节只存放了 Cobj_2 的构造函数
  5 .gnu.linkonce.t._ZN6CObj_3C1Ev 0000000e  00000000  00000000  00000174  2**1
              CONTENTS, ALLOC, LOAD, READONLY, CODE, LINK_ONCE_DISCARD
  这个节只存放了 Cobj_3 的构造函数
  6 .gnu.linkonce.t._ZN6CObj_1D1Ev 0000000e  00000000  00000000  00000182  2**1
              CONTENTS, ALLOC, LOAD, READONLY, CODE, LINK_ONCE_DISCARD
  这个节只存放了 Cobj_1 的析构函数
  7 .gnu.linkonce.t._ZN6CObj_2D1Ev 0000000e  00000000  00000000  00000190  2**1
              CONTENTS, ALLOC, LOAD, READONLY, CODE, LINK_ONCE_DISCARD
  这个节只存放了 Cobj_2 的析构函数
  8 .gnu.linkonce.t._ZN6CObj_3D1Ev 0000000e  00000000  00000000  0000019e  2**1
              CONTENTS, ALLOC, LOAD, READONLY, CODE, LINK_ONCE_DISCARD
  这个节只存放了 Cobj_3 的析构函数

从上面刻看出 g++把 3 个构造函数与 3 个析构函数放在不同的节中。

在链接时根据 gcc 的默认链接器脚本中如下脚本

  .text           :
  {
    *(.text .stub .text.* .gnu.linkonce.t.*)
    /* .gnu.warning sections are handled specially by elf32.em.  */
    *(.gnu.warning)
  } =0x90909090

由于上面存放构造函数与析构函数的节名是以.gnu.linonce.t 开头的，所以构造、析构
代码最终将放入输出文件的代码节.text。

看一下链接器的输入文件 construct.o 的符号表。
$ nm construct.o
         U __cxa_atexit
         U __dso_handle
0000010a t _GLOBAL__I_obj_1
         U __gxx_personality_v0
```

```
00000000 T main
00000000 B obj_1
00000004 B obj_2
00000008 B obj_3
000000c2 t __tcf_0
000000da t __tcf_1
000000f2 t __tcf_2
00000018 t _Z41__static_initialization_and_destruction_0ii
00000000 W _ZN6CObj_1C1Ev     Cobj_1 ctor 的 symbol
00000000 W _ZN6CObj_1D1Ev     Cobj_1 dtor 的 symbol
00000000 W _ZN6CObj_2C1Ev     Cobj_2 ctor 的 symbol
00000000 W _ZN6CObj_2D1Ev     Cobj_2 dtor 的 symbol
00000000 W _ZN6CObj_3C1Ev     Cobj_3 ctor 的 symbol
00000000 W _ZN6CObj_3D1Ev     Cobj_3 dtor 的 symbol
```

没有看到什么__CTOR_LIST__，__CTOR_END__，__DTOR_LIST__，__DTOR_END__之类的符号。很自然，因为这些都是 linker 产生的。

下面看一下链接器的输出文件中的信息。

```
$ objdump -h construct

construct:    file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .interp       00000013  08048114  08048114  00000114  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000020  08048128  08048128  00000128  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .hash         00000030  08048148  08048148  00000148  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .dynsym       00000070  08048178  08048178  00000178  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .dynstr       000000b9  080481e8  080481e8  000001e8  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .gnu.version  0000000e  080482a2  080482a2  000002a2  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .gnu.version_r 00000050  080482b0  080482b0  000002b0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .rel.dyn      00000008  08048300  08048300  00000300  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .rel.plt      00000018  08048308  08048308  00000308  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
```

```
 9 .init         00000018  08048320  08048320  00000320  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
10 .plt          00000040  08048338  08048338  00000338  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
11 .text         0000024c  08048378  08048378  00000378  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
12 .fini         0000001c  080485c4  080485c4  000005c4  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
13 .rodata       00000008  080485e0  080485e0  000005e0  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
14 .eh_frame_hdr 00000034  080485e8  080485e8  000005e8  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
15 .data         0000000c  0804961c  0804961c  0000061c  2**2
                 CONTENTS, ALLOC, LOAD, DATA
16 .eh_frame     000000e8  08049628  08049628  00000628  2**2
                 CONTENTS, ALLOC, LOAD, DATA
17 .dynamic      000000e0  08049710  08049710  00000710  2**2
                 CONTENTS, ALLOC, LOAD, DATA
18 .ctors        0000000c  080497f0  080497f0  000007f0  2**2
                 CONTENTS, ALLOC, LOAD, DATA
19 .dtors        00000008  080497fc  080497fc  000007fc  2**2
                 CONTENTS, ALLOC, LOAD, DATA
20 .jcr          00000004  08049804  08049804  00000804  2**2
                 CONTENTS, ALLOC, LOAD, DATA
21 .got          0000001c  08049808  08049808  00000808  2**2
                 CONTENTS, ALLOC, LOAD, DATA
22 .bss          00000010  08049824  08049824  00000824  2**2
                 ALLOC
23 .comment      00000132  00000000  00000000  00000824  2**0
                 CONTENTS, READONLY
24 .debug_aranges 00000058  00000000  00000000  00000958  2**3
                 CONTENTS, READONLY, DEBUGGING
25 .debug_pubnames 00000025  00000000  00000000  000009b0  2**0
                 CONTENTS, READONLY, DEBUGGING
26 .debug_info   00000c85  00000000  00000000  000009d5  2**0
                 CONTENTS, READONLY, DEBUGGING
27 .debug_abbrev 00000127  00000000  00000000  0000165a  2**0
                 CONTENTS, READONLY, DEBUGGING
28 .debug_line   000001f2  00000000  00000000  00001781  2**0
                 CONTENTS, READONLY, DEBUGGING
29 .debug_frame  00000014  00000000  00000000  00001974  2**2
                 CONTENTS, READONLY, DEBUGGING
30 .debug_str    0000098a  00000000  00000000  00001988  2**0
                 CONTENTS, READONLY, DEBUGGING
```

我们看到了关心的.ctors 与.dtors 节。这两个节的属性都是“要载入”的数据。

```
$ nm construct
08049824 A __bss_start
0804839c t call_gmon_start
08049824 b completed.1
080497f8 d __CTOR_END__
080497f0 d __CTOR_LIST__
       U __cxa_atexit@@GLIBC_2.1.3
0804961c D __data_start
0804961c W data_start
080485a0 t __do_global_ctors_aux
080483c0 t __do_global_dtors_aux
08049620 d __dso_handle
08049800 d __DTOR_END__
080497fc d __DTOR_LIST__
08049710 D _DYNAMIC
08049824 A _edata
08049628 d __EH_FRAME_BEGIN__
08049834 A _end
080485c4 T _fini
080485e0 R _fp_hw
080483fc t frame_dummy
0804970c d __FRAME_END__
08048532 t _GLOBAL__I_obj_1
08049808 D _GLOBAL_OFFSET_TABLE_
       w __gmon_start__
       U __gxx_personality_v0@@CXXABI_1.2
08048320 T _init
080485e4 R _IO_stdin_used
08049804 d __JCR_END__
08049804 d __JCR_LIST__
       w _Jv_RegisterClasses
       U __libc_start_main@@GLIBC_2.0
08048428 T main
08049828 B obj_1
0804982c B obj_2
08049830 B obj_3
08049624 d p.0
08048378 T _start
080484ea t __tcf_0
08048502 t __tcf_1
0804851a t __tcf_2
```

```
08048440 t _Z41__static_initialization_and_destruction_0ii
0804854c W _ZN6CObj_1C1Ev
08048576 W _ZN6CObj_1D1Ev
0804855a W _ZN6CObj_2C1Ev
08048584 W _ZN6CObj_2D1Ev
08048568 W _ZN6CObj_3C1Ev
08048592 W _ZN6CObj_3D1Ev
```

整个列表比较长，摘取感兴趣的。

根据文章中的说法这是构造函数地址列表的头与尾。

```
080497f8 d __CTOR_END__
080497f0 d __CTOR_LIST__
```

根据文章中的说法这是析构函数地址列表的头与尾。

```
08049800 d __DTOR_END__
080497fc d __DTOR_LIST__
```

这是构造函数与析构函数的地址

```
0804854c W _ZN6CObj_1C1Ev
08048576 W _ZN6CObj_1D1Ev
0804855a W _ZN6CObj_2C1Ev
08048584 W _ZN6CObj_2D1Ev
08048568 W _ZN6CObj_3C1Ev
08048592 W _ZN6CObj_3D1Ev
```

再看一下.ctors 与.dtors 节的信息

```
18 .ctors       0000000c  080497f0  080497f0  000007f0  2**2
                CONTENTS, ALLOC, LOAD, DATA
19 .dtors       00000008  080497fc  080497fc  000007fc  2**2
                CONTENTS, ALLOC, LOAD, DATA
```

即.ctors 节开始于文件偏移 0x7f0（2032）处，该节大小为 12（0x0c）个字节；.dtors 节开始于文件偏移 0x7fc（2044）处，该节大小为 8 个字节。

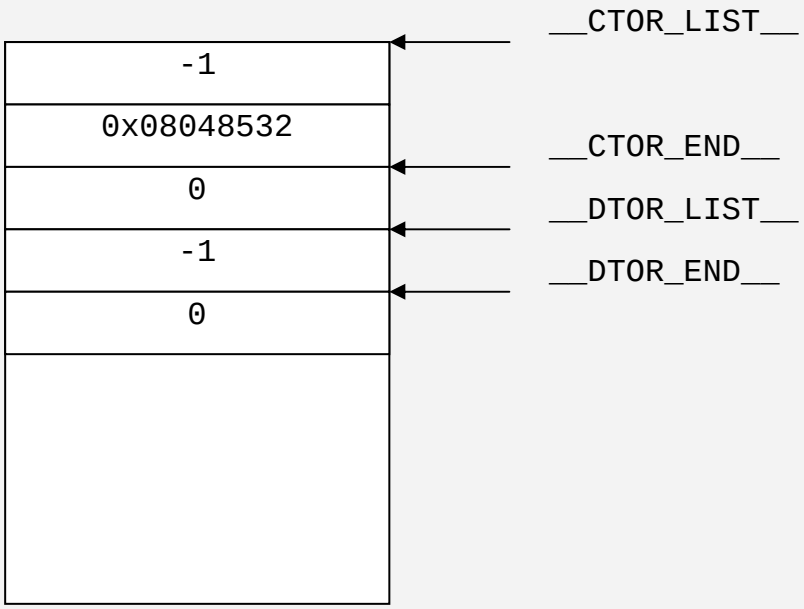
看一下节中的内容。

先看.ctors 节

```
$ hexdump -C -s 2032 -n 12 construct
000007f0  ff ff ff ff 32 85 04 08  00 00 00 00           |....2.......|
```

再看.dtors 节

```
$ hexdump -C -s 2044 -n 8 construct
000007fc  ff ff ff ff 00 00 00 00                       |........|
```



很显然，上面的图与文章中介绍的完全“文不对题”。在符号表中搜索 0x8048532，会发现它是符号_GLOBAL__I_obj_1 的地址。

```
08048532 t _GLOBAL__I_obj_1
```

这里的“t”表示这个符号是在代码段，显然_GLOBAL__I_obj_1 是个函数，不是我们编写，而是链接器送给我们的。反汇编一下：

```
$ objdump -d construct
08048532 <_GLOBAL__I_obj_1>:
 8048532: 55                    push   %ebp
 8048533: 89 e5                 mov    %esp,%ebp
 8048535: 83 ec 08              sub    $0x8,%esp
 8048538: 83 ec 08              sub    $0x8,%esp
 804853b: 68 ff ff 00 00        push   $0xffff
 8048540: 6a 01                 push   $0x1
 8048542: e8 f9 fe ff ff        call                           8048440
<_Z41__static_initialization_and_destruction_0ii>
 8048547: 83 c4 10              add    $0x10,%esp
 804854a: c9                    leave
 804854b: c3                    ret
```

看不出什么名堂，只有一个函数调用。看一下调用的函数：

```
08048440 <_Z41__static_initialization_and_destruction_0ii>:
 8048440: 55                    push   %ebp
 8048441: 89 e5                 mov    %esp,%ebp
 8048443: 83 ec 08              sub    $0x8,%esp
```

```
 8048446: 81 7d 0c ff ff 00 00    cmpl   $0xffff,0xc(%ebp)
 804844d: 75 2d                   jne                            804847c
<_Z41__static_initialization_and_destruction_0ii+0x3c>
 804844f: 83 7d 08 01             cmpl   $0x1,0x8(%ebp)
 8048453: 75 27                   jne                            804847c
<_Z41__static_initialization_and_destruction_0ii+0x3c>
 8048455: 83 ec 0c                sub    $0xc,%esp
 8048458: 68 28 98 04 08          push   $0x8049828
 804845d: e8 ea 00 00 00          call   804854c <_ZN6CObj_1C1Ev>[2]
 8048462: 83 c4 10                add    $0x10,%esp
 8048465: 83 ec 04                sub    $0x4,%esp
 8048468: 68 20 96 04 08          push   $0x8049620
 804846d: 6a 00                   push   $0x0
 804846f: 68 ea 84 04 08          push   $0x80484ea
 8048474: e8 cf fe ff ff          call   8048348 <_init+0x28>
 8048479: 83 c4 10                add    $0x10,%esp
 804847c: 81 7d 0c ff ff 00 00    cmpl   $0xffff,0xc(%ebp)
 8048483: 75 2d                   jne                            80484b2
<_Z41__static_initialization_and_destruction_0ii+0x72>
 8048485: 83 7d 08 01             cmpl   $0x1,0x8(%ebp)
 8048489: 75 27                   jne                            80484b2
<_Z41__static_initialization_and_destruction_0ii+0x72>
 804848b: 83 ec 0c                sub    $0xc,%esp
 804848e: 68 2c 98 04 08          push   $0x804982c
 8048493: e8 c2 00 00 00          call   804855a <_ZN6CObj_2C1Ev>[3]
 8048498: 83 c4 10                add    $0x10,%esp
 804849b: 83 ec 04                sub    $0x4,%esp
 804849e: 68 20 96 04 08          push   $0x8049620
 80484a3: 6a 00                   push   $0x0
 80484a5: 68 02 85 04 08          push   $0x8048502
 80484aa: e8 99 fe ff ff          call   8048348 <_init+0x28>
 80484af: 83 c4 10                add    $0x10,%esp
 80484b2: 81 7d 0c ff ff 00 00    cmpl   $0xffff,0xc(%ebp)
 80484b9: 75 2d                   jne                            80484e8
<_Z41__static_initialization_and_destruction_0ii+0xa8>
 80484bb: 83 7d 08 01             cmpl   $0x1,0x8(%ebp)
 80484bf: 75 27                   jne                            80484e8
<_Z41__static_initialization_and_destruction_0ii+0xa8>
 80484c1: 83 ec 0c                sub    $0xc,%esp
 80484c4: 68 30 98 04 08          push   $0x8049830
 80484c9: e8 9a 00 00 00          call   8048568 <_ZN6CObj_3C1Ev>[4]
```

[2] 这是Cobj_1 class的ctor
[3] 这是Cobj_2 class的ctor
[4] 这是Cobj_3 class的ctor

```
 80484ce: 83 c4 10            add    $0x10,%esp
 80484d1: 83 ec 04            sub    $0x4,%esp
 80484d4: 68 20 96 04 08      push   $0x8049620
 80484d9: 6a 00               push   $0x0
 80484db: 68 1a 85 04 08      push   $0x804851a
 80484e0: e8 63 fe ff ff      call   8048348 <_init+0x28>
 80484e5: 83 c4 10            add    $0x10,%esp
 80484e8: c9                  leave
 80484e9: c3                  ret
```

在该函数中分别调用了 3 个构造函数。也就是说在__CTOR_LIST__与__CTOR_END__之间的是一个链接器提供的初始化函数，它会去调用各个全局对象的构造函数。那么析构函数呢？在__DTOR_LIST__与__DTOR_END__之间没有任何东西，难道不调用析构了？当然不是，析构函数同样在上面的初始化函数中被处理了。

在符号表中的下列 3 个符号是 3 个析构函数的 stub 函数：

```
080484ea t __tcf_0
08048502 t __tcf_1
0804851a t __tcf_2
```

反汇编看一下：

```
080484ea <__tcf_0>:
 80484ea: 55                  push   %ebp
 80484eb: 89 e5               mov    %esp,%ebp
 80484ed: 83 ec 08            sub    $0x8,%esp
 80484f0: 83 ec 0c            sub    $0xc,%esp
 80484f3: 68 28 98 04 08      push   $0x8049828
 80484f8: e8 79 00 00 00      call   8048576 <_ZN6CObj_1D1Ev>[5]
 80484fd: 83 c4 10            add    $0x10,%esp
 8048500: c9                  leave
 8048501: c3                  ret

08048502 <__tcf_1>:
 8048502: 55                  push   %ebp
 8048503: 89 e5               mov    %esp,%ebp
 8048505: 83 ec 08            sub    $0x8,%esp
 8048508: 83 ec 0c            sub    $0xc,%esp
 804850b: 68 2c 98 04 08      push   $0x804982c
 8048510: e8 6f 00 00 00      call   8048584 <_ZN6CObj_2D1Ev>[6]
 8048515: 83 c4 10            add    $0x10,%esp
 8048518: c9                  leave
```

[5] Cobj_1 class的dtor
[6] Cobj_2 class的dtor

```
 8048519: c3                   ret

0804851a <__tcf_2>:
 804851a: 55                   push   %ebp
 804851b: 89 e5                mov    %esp,%ebp
 804851d: 83 ec 08             sub    $0x8,%esp
 8048520: 83 ec 0c             sub    $0xc,%esp
 8048523: 68 30 98 04 08       push   $0x8049830
 8048528: e8 65 00 00 00       call   8048592 <_ZN6CObj_3D1Ev>[7]
 804852d: 83 c4 10             add    $0x10,%esp
 8048530: c9                   leave
 8048531: c3                   ret
```

下面标紫色的就是对析构函数的 stub 的处理：

```
08048440 <_Z41__static_initialization_and_destruction_0ii>:
 8048440: 55                   push   %ebp
 8048441: 89 e5                mov    %esp,%ebp
 8048443: 83 ec 08             sub    $0x8,%esp
 8048446: 81 7d 0c ff ff 00 00    cmpl   $0xffff,0xc(%ebp)
 804844d: 75 2d                jne                            804847c
<_Z41__static_initialization_and_destruction_0ii+0x3c>
 804844f: 83 7d 08 01          cmpl   $0x1,0x8(%ebp)
 8048453: 75 27                jne                            804847c
<_Z41__static_initialization_and_destruction_0ii+0x3c>
 8048455: 83 ec 0c             sub    $0xc,%esp
 8048458: 68 28 98 04 08       push   $0x8049828
 804845d: e8 ea 00 00 00       call   804854c <_ZN6CObj_1C1Ev>
 8048462: 83 c4 10             add    $0x10,%esp
 8048465: 83 ec 04             sub    $0x4,%esp
 8048468: 68 20 96 04 08       push   $0x8049620
 804846d: 6a 00                push   $0x0
 804846f: 68 ea 84 04 08       push   $0x80484ea[8]
 8048474: e8 cf fe ff ff       call   8048348 <_init+0x28>
 8048479: 83 c4 10             add    $0x10,%esp
 804847c: 81 7d 0c ff ff 00 00    cmpl   $0xffff,0xc(%ebp)
 8048483: 75 2d                jne                            80484b2
<_Z41__static_initialization_and_destruction_0ii+0x72>
 8048485: 83 7d 08 01          cmpl   $0x1,0x8(%ebp)
 8048489: 75 27                jne                            80484b2
<_Z41__static_initialization_and_destruction_0ii+0x72>
 804848b: 83 ec 0c             sub    $0xc,%esp
```

[7] Cobj_3 class的dtor
[8] Cobj_1 class dtor的stub

```
 804848e: 68 2c 98 04 08      push   $0x804982c
 8048493: e8 c2 00 00 00      call   804855a <_ZN6CObj_2C1Ev>
 8048498: 83 c4 10            add    $0x10,%esp
 804849b: 83 ec 04            sub    $0x4,%esp
 804849e: 68 20 96 04 08      push   $0x8049620
 80484a3: 6a 00               push   $0x0
 80484a5: 68 02 85 04 08      push   $0x8048502[9]
 80484aa: e8 99 fe ff ff      call   8048348 <_init+0x28>
 80484af: 83 c4 10            add    $0x10,%esp
 80484b2: 81 7d 0c ff ff 00 00    cmpl   $0xffff,0xc(%ebp)
 80484b9: 75 2d               jne                            80484e8
<_Z41__static_initialization_and_destruction_0ii+0xa8>
 80484bb: 83 7d 08 01         cmpl   $0x1,0x8(%ebp)
 80484bf: 75 27               jne                            80484e8
<_Z41__static_initialization_and_destruction_0ii+0xa8>
 80484c1: 83 ec 0c            sub    $0xc,%esp
 80484c4: 68 30 98 04 08      push   $0x8049830
 80484c9: e8 9a 00 00 00      call   8048568 <_ZN6CObj_3C1Ev>
 80484ce: 83 c4 10            add    $0x10,%esp
 80484d1: 83 ec 04            sub    $0x4,%esp
 80484d4: 68 20 96 04 08      push   $0x8049620
 80484d9: 6a 00               push   $0x0
 80484db: 68 1a 85 04 08      push   $0x804851a[10]
 80484e0: e8 63 fe ff ff      call   8048348 <_init+0x28>
 80484e5: 83 c4 10            add    $0x10,%esp
 80484e8: c9                  leave
 80484e9: c3                  ret

上面的函数是初始化函数，当然不可能在这儿调用全局析构函数，这是向程序的退出钩子函数注册当程序从 main 函数返回要调用的函数，这里就是 3 个析构函数。Gnu linker 的真正实现与手册上说的，相差太多。所以有时候光看手册是会害人的。
```

输出节的丢弃

链接器不会创建没有任何内容的输出节。为了书写脚本的方便，引用到的输入文件中的输入节可能存在也可能不存在。比如象下面的例子：

```
.foo { *(.foo) }
```

只有在至少有一个输入文件中有.foo输入节的情况下，链接器才会在输出文件

[9] Cobj_2 class dtor的stub
[10] Cobj_3 class dtor的stub

中创建.foo输出节（也就是在通配符*所代表的所有输入文件中如果没有文件含有.foo为节名的输入节，则这一行等于没有一样。这为书写链接器脚本带来一定方便，因为编写者不需要去判断各个输入文件中是否有某个节了）。

但如果你在输出节命令中用到任何除输入节描述以外的其他描述，比如为符号赋值，则链接器总会创建该输出节，即使实际上没有任何输入节匹配。

特殊的输出节名/DISCARD/可以用于丢弃输入节（即你不想放入任何输出节的内容）。任何被指定放入所谓/DISCARD/输出节的输入节将不会被包括进输出文件。

输出节属性

在前面我们展示过输出节的整体描述是象下面这样的：

```
section [address] [(type)] : [AT(lma)]
{
   output-section-command
   output-section-command
   ...
} [>region] [AT>lma_region] [:phdr :phdr …] [=fillexp]
```

在前面我们已经介绍过 section，address 和 output-section-command 了，下面我们将介绍剩下的节属性。

输出节类型

每个输出节都可以有下面定义的类型，也就是上面被括在括号里的 type。

NOLOAD
　　该类型的输出节不需要被载入，即当程序运行时，该节不会被载入内存。

DSECT
COPY
INFO
OVERLAY
　　支持这些类型是为了向下兼容，很少用到。它们都有同样的效果：该节被标

记为不要分配，即当程序运行时，这种类型的节是不需要分配内存的。

链接器通常基于映射如本节的输入节的属性来设置输出节的属性，当然你可以指定节的类型来覆盖它。比如，在下面的例子脚本中，ROM 节被定位在内存地址零，并且当程序运行时，不需要被载入，但 ROM 节的内容还是会在链接器的输出文件中的。

```
.SECTIONS {
   ROM 0 (NOLOAD) : { … }
}
```

输出节属性 LMA

每个输出节都有一个虚拟地址（VMA）和载入地址（LMA）；请参见 4.1 节基本的链接器脚本概念。出现在输出节描述中的地址表达式设置的是ＶＭＡ的地址。

```
象
SECTION
{
   outputa  0x10000  :
   {
      all.o
      foo.o（.input1）
   }
}
中的 0x10000 指定的是该节(outputa 节)的虚拟地址，即 VMA，也就是当该节的代码在运行时的地址。
```

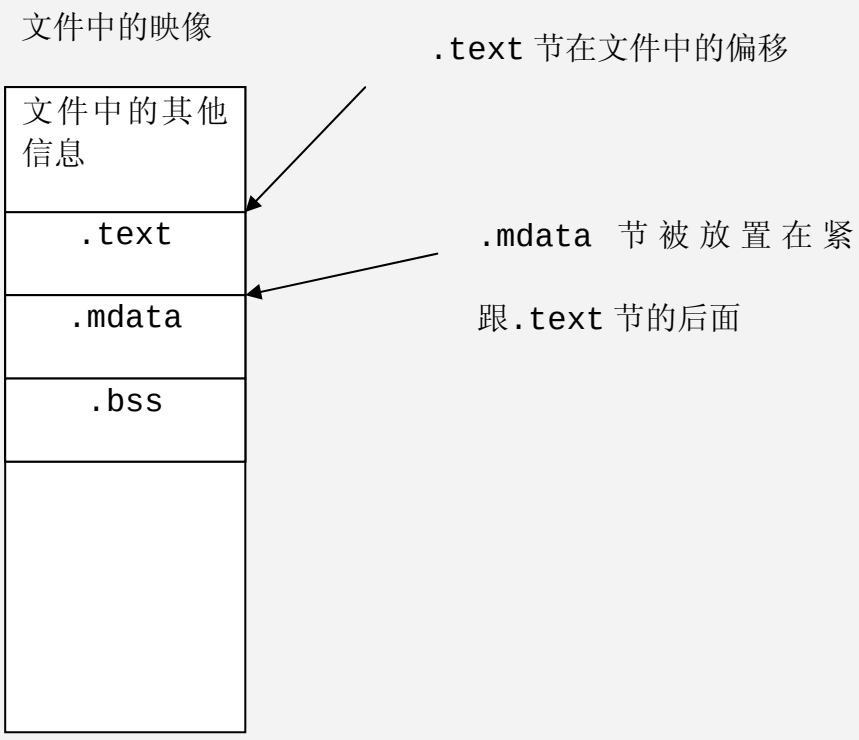
链接器通常把 LMA 设置同 VMA 一样，但你也可以用 AT 关键字来改变。跟在 AT 关键字后面的 lma 表达式可以指定该节的载入地址。同样的，用 AT>lma_region 表达式可以为节的载入地址指定一块内存区域。请参见 4.7 节 MEMORY 命令。

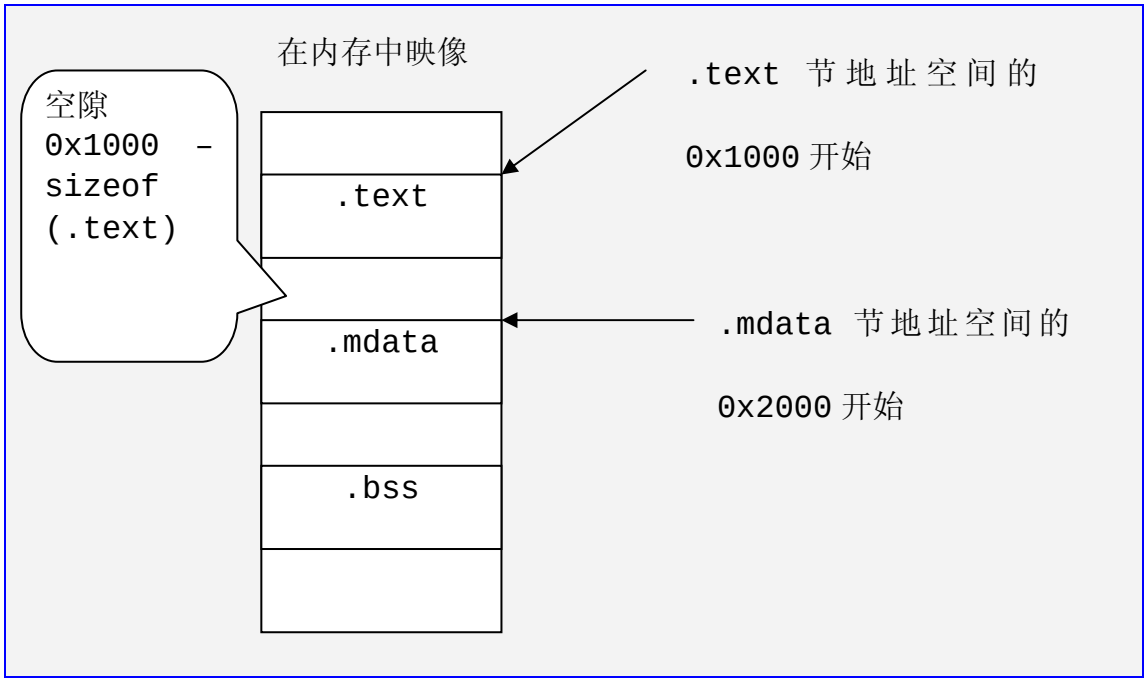
这个特性使得生成 ROM 映像比较方便。比如下面的链接器脚本创建了 3 个输出节：一个节名为.text，开始于地址 0x1000，一个名为.mdata，VMA 虽然是 0x2000，但被载入到紧接着.text 节的后面，另一个是节名为.bss，包含未初

始化数据，开始与地址.0x3000。符号_data 被赋值成 0x2000，可见位置指示器指向的是 VMA 的值，而不是 LMA 的值。

```
SECTIONS
{
   .text  0x1000 : { *(.text) _etext = . ; }
   .mdata 0x2000 :
      AT (ARRD (.text) + SIZEOF(.text) )
      { _data = . ; *(.data); _edata = .; }
   .bss 0x3000
      { _bstart = . ; *(.bss) *(COMMON) ; _bend = . ;}
}
```

可以用下面的图来描述上面的例子





用上面链接器脚本生成的例子运行时初始程序要包括类似下面的运行时初始化代码，即把 ROM 映像中的初始化数据拷贝到它的运行地址。请注意下面的代码是怎样利用链接器脚本定义的符号的。

```
extern char _etext, _data, _edata, _bstart, _bend;
char *src = &_etext;
char *dst = &_data;

/* ROM has data at end of text; copy it. */
while(dst < &_edata) {
   *dst++ = *src++;
}

/* Zero bss*/
for(dst = &_bstart; dst < &_bend; dst++)
   *dst = 0;
```

输出节属性区域

用>region 可以把节赋予前面定义的内存区域。请参见 4.7 节内存命令。

这里有个简单的例子：

```
MEMORY { rom : ORIGN = 0x1000, LENGTH = 0x1000 }

SECTIONS { ROM : { *(.text) } > rom }
```

输出节属性 Phdr

你可以通过:phdr 命令把某个节赋给以前定义的程序段（segment）。请参见 4.8 节 PHDRS 命令。如果某个节被赋给了一个或多个段，则接下来被分配的结同样被赋给了这些段，除非有显式指明新的：phdr 命令来更改段。你可以用：NONE 命令来指导链接器不要把节放入任何段中。

下面有个简单的例子：

```
PHDRS { text PT_LOAD ; }
SECTIONS { .text : { *(.text) } :text }
```

输出节填充属性

你可以用=fillexp来对整个节进行填充，这里fillexp是一个表达式（请参见 4.10 节链接器脚本中的表达式）。输出节中任何未特定指明的区域（比如，由于放入该输出节的输入节之间的对齐而留下的间隙）将被反复填充以该表达式的值。如果填充表达式是简单的十六进制数，比如以 0x开头的数字，同时尾部没有k或M，则任意长度的十六进制数都可以作为填充值。数字的前导零也成为填充数字（比如，0x0011，则前面的两个数学上无意义的零也成为填充值）。另外也可以包括括号和+，填充值取的是表达式的值得最低 4 个字节。所有情况下，填充值都认为是大尾端的。

在输出节命令中你也可以用**FILL**命令来改变填充值，请参见 4.6.5 节输出节数据。

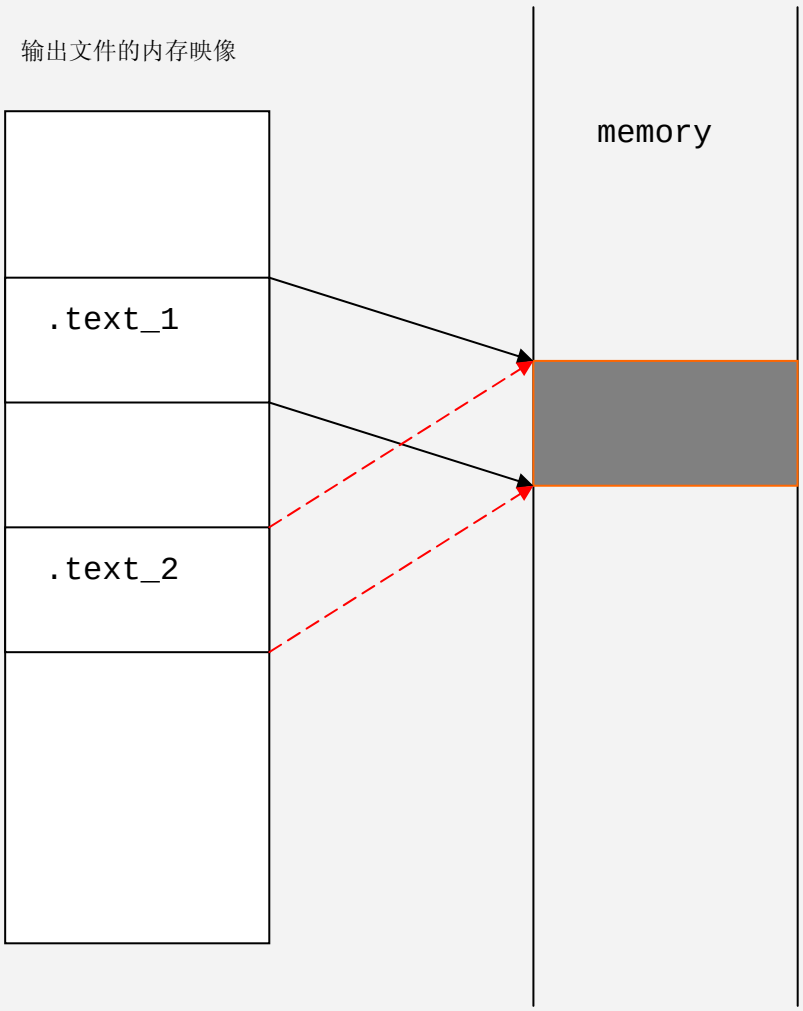
这里有一个简单的例子：

```
SECTIONS { .text : { *(.text) } = 0x90909090 }
```

覆盖（overlay）介绍

下面对 overlay 一词将不做翻译，我觉得可能更易懂。Overlay 技术在现在内存便宜如斯的今天，还有应用的场合吗？恕我孤陋寡闻，我只知道在 60，70 年代有些系统载入可执行文件时由于受限于当时内存的稀缺，不得不应用该技术，在现今，我好象再也没有听说过。其实我当然没有机会去经历这种技术，我的了解也只是在自学链接器实现技术时学到的，而且是作为曾经有过的链接技术历史学到的。

Overlay命令对下面一种需求提供了比较简单的解决方法，即可执行内存映像的不同部分在运行时却要在相同的地址(我不知道除了因为内存紧张的缘故，还会有其他原因会要求这样做吗？下面的图可以解释这种需求)。



所谓“输出文件内存映像”这种文绉绉的词，在一般系统上，就是各种格式的可执行文件，在嵌入式系统上就是最终要烧到 ROM 里的那一大坨 binary 文件。假设上面该内存映像中

有.text_1 和.text_2 这两个 section，静态时它们在物理上当然不在同一个位置，它们分别在“映像”的不同位置。但在运行时，它们必须在同一个地址上。同样的，他们不可能是在同一时间出现在同一个地址，只可能是要么先.text_1 先运行，然后是.text_1 运行，或反之。这就是所谓覆盖技术。这里上图中之所以标“memory”，是因为情况不同，该词意义也不同。对一般的系统而言，这个 memory 表示一个程序的地址空间，而对嵌入式系统而言，可能就是实实在在的内存了。

在运行时，需要overlay管理器介入，在要覆盖的节和内存地址之间拷贝来拷贝去，或许通过简单的操纵寻址位就行（不理解什么叫操纵寻址位）。这种方法可能是有用的，比如，当某些内存速度比另一些快时。

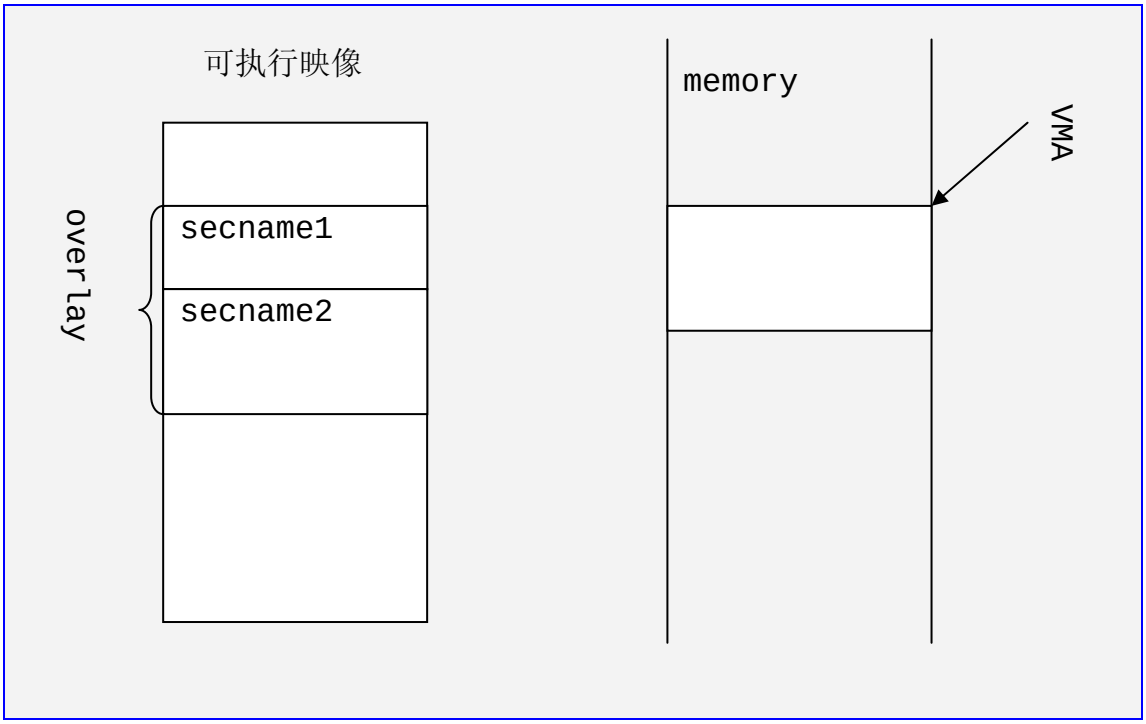
看作者的意思是这样的：比如像上图中，如果不用 overlay 技术，则.text_1 和.text_2 节被载入内存的不同地址，运行各占一块，互不相干。但如果.text_1 处的内存速度远远快于.text_2 出的，则在运行.text_2 代码时最好把代码拷贝到原来.text_1 所占的地方再运行，可能会快一点。我知道在 BIOS 里有个选项，可以把 ROM 中的代码拷贝到 RAM 中，然后当要运行 ROM 中代码时，其实是在运行 RAM 中的 ROM 代码。原因是一般 ROM 的速度都要比 RAM 慢。

Overlay技术用**OVERLAY**命令来实现。**OVERLAY**命令同其他输出节命令一样只能用在**SECTIONS**命令内部。下面是OVERLAY命令的完整语法格式：

```
OVERLAY [start] : [NOCROSSREFS] [AT (ldaddr)]
{
   secname1
   {
      output-section-command
      output-section-command
      …
   } [:phdr…] [=fill]
   secname2
   {
      output-section-command
      output-section-command
      …
   } [:phdr…] [=fill]
   …
} [>region] [:phdr…] [=fill]
```

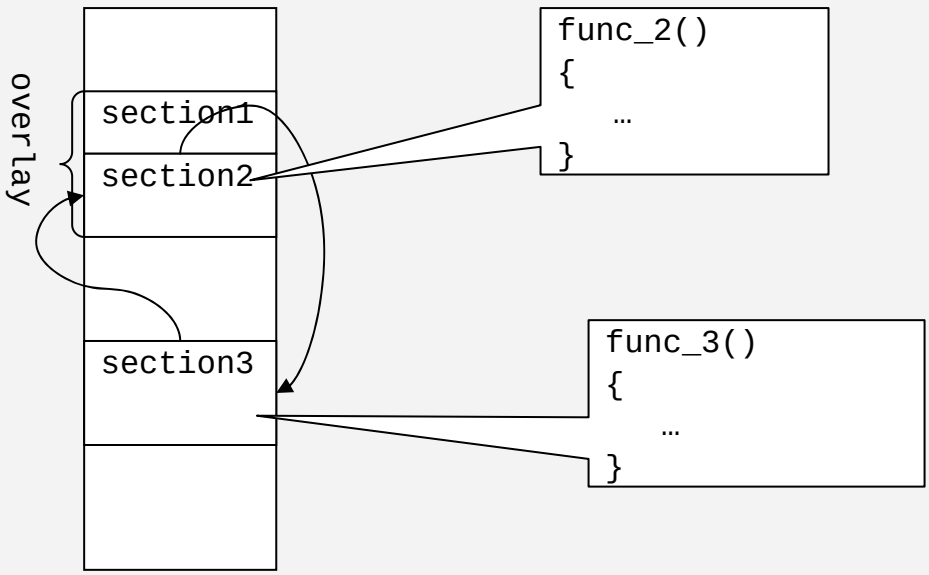
除了**OVERLAY**关键字和每个节必须有的节名（上面例子中的secname1 和secname2）外，其他成员都是可选的。**OVERLAY**结构内的节的定义同一般的节的定义是相同的，例外之处是**OVERLAY**命令内部的节的定义中不能指定地址和内存区域（memory regions）。

OVERLAY命令内的不同节开始于相同的虚拟地址（VMA），而这些节在可执行映像中位置，被安排作为一个整体从载入地址开始连续的存放（象普通的节定义一样，载入地址也是可选的，默认就是VMA地址；同样VMA地址也是可选的，默认情况下是位置指示器的当前值）。



如果在**OVERLAY**命令中用到了**NOCROSSREF**关键字，并且在**OVERLAY**命令中的节之间有互相引用（即比如section1 中的代码有调用section2 中的函数及访问变量，或反之），编译器将报错。由于**OVERLAY**中的节是在相同的VMA运行，所以从一个节中直接访问另一个节是没有意义的。

section1 与 section2 只能同时一个在内存中，当然两者不能互相“直接”访问，但“间接”应该可以的吧？象下图中的情况：



Section1 和 section2 是位于 OVERLAY 命令中的节，而 section3 是普通的节。假设 section1 中的代码访问 section3 中的 func_3 函数，而该函数又访问 section2 中的 func_2 函数，这应当是可以的。

对于**OVERLAY**中的每个节，链接器会为每个节自动生成两个符号。符号__load_start_secname 指向该节载入地址的头，符号__load_stop_secname指向该节载入地址的尾。而在secname中任何不符合C语言中标示符定义可用的字符都被删除。C（或汇编）代码可以利用这些标的出节的边界的符号（在C语言或汇编语言中看作指向某段代码的地址）来完成OVERLAY中的节间覆盖的实现。

在 overlay 的尾部，位置指示器的值被设置成 overlay 的开始地址加上 overlay 中占空间最大的节的大小。

这里有一个例子，注意下面的脚本片段位于**SECTIONS**命令的内部。

```
OVERLAY 0x1000 : AT (0x4000)
{
   .text0 { o1/*.o(.text) }
   .text1 { o2/*.o(.text) }
```

```
}
```

这里的 o1/，o2/是目录。o1.*.o 表示 o1 目录下的所有.o 文件。

例子中定义了两个运行后开始于地址 0x1000 的节，.text0 和.text1。.text 节 0 将被载入到 0x4000，而.text1 节则被载入到紧接着.text0 节之后。同时__load_start_text0，__load_stop_text0，__load_start_text1，__load_stop_text1 这四个符号被定义。

用 C 代码来拷贝.text1 节的内容到覆盖区域可能看上去像下面的代码。

```
extern char __load_start_text1, __load_stop_text1;

memcpy（(char *） 0x1000,

        &__load_start_text1,

        &__load_stop_text1 - &__load_start_text1）;
```

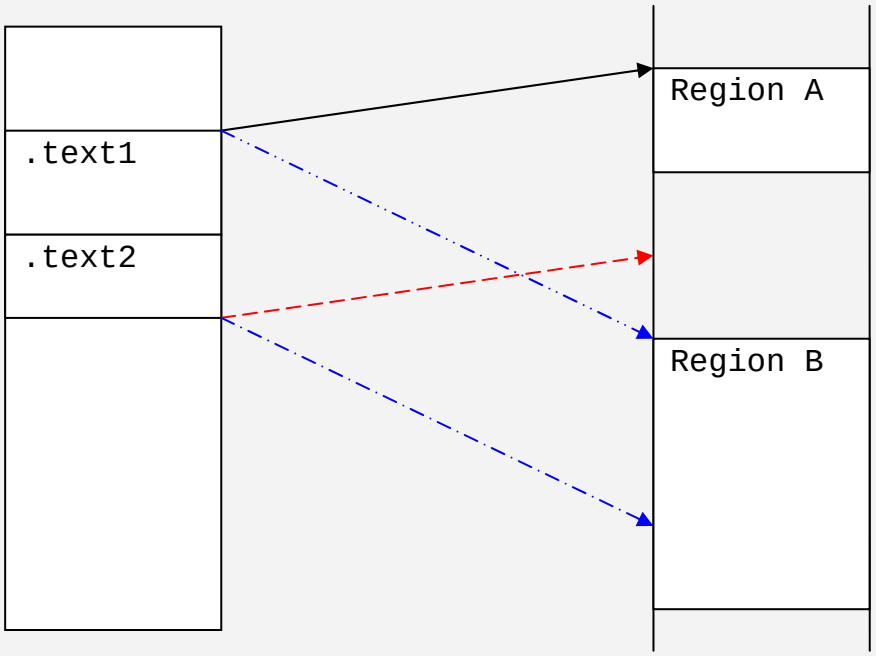
我们注意到，其实OVERLAY命令只是“语法甜头”（syntactic sugar）（这个词组，我的理解是在语法上提供更方便的手段，但不一定是必需的，不知道理解得对不对）。该命令的功能可以用更基本的命令来实现，上面的例子就可以用下面的脚本片段来实现同样的功能。

```
.text0 0x1000 : AT (0x4000) { o1/*.o(.text) }

__load_start_text0 = LOADADDR (.text0);

__load_stop_text0 = LOADADDR (.text0) + SIZEOF(.text0);

.text1 0x1000 : AT (0x4000 + SIZEOF(.text0)) { o2/*.o(.text) }

__load_start_text1 = LOADADDR (.text1);

__load_stop_text1 = LOADADDR (.text1) + SIZEOF(.text1);

. = 0x1000 + MAX (SIZEOF(.text0), SIZOF(.text1));
```

MEMORY 命令

链接器的默认配置允许分配所有可获得的内存（对于所谓“可获得的内存”我的理解是这样的：在一般系统上，无论是大家都熟悉的Windows,Linux或Solaris，Mac OS，可执行文件可获得的最大虚拟内存就是操作系统运行该文件时提供给它的的虚拟地址空间，比如对Windows/Linux而言，就是 4G空间，对 64 位CPU，比如SPARC，则是更大的虚拟空间，但并不一定是 2 的 64 次方，因为很多 64 位CPU没有用足地址空间；而对嵌入式系统而言，一般就没有虚存的概念，这“可获得的内存”就是CPU能寻址的空间了。这样“可获得的内存”其实与我们常规理解的插在内存插槽上的内存条是完全不是一个概念），但你可以用**MEMORY**命令来修改。

MEMORY命令描述了目标机上的内存块的位置和大小。你可以用它去指定哪些内存区域可以被链接器使用，而哪些则必须避免使用。你可以把输出节指定给用**MEMORY**定义的特定内存区域，链接器就会基于该内存区域来设置输出节的地址，并在该区域放不下输出节时报警。（在不能把输出节完整放入指定内存区域的情况下,）链接器不会对在脚本中指定的输出进行重新安排，把它们放入其他可用的内存区域中去。



在上图中，链接器脚本把.text1 和.text2 这两个输出节都指派到内存区域 Region A，但很明显，该区域空间太小，塞不下这两个节；但另一块内存区域 Region B 的空间比较大，能容得下这两个输出节。但链接器不会这么做。因为这不是你程序员在脚本中指定的。链接器不会做你程序员没告诉它的事，它只会报错，链接失败。

链接器脚本中只能包含一条**MEMORY**命令，但在该命令中你可以定义的内存区域却是你想要多少就可以多少，没有限制。下面是该命令的语法：

```
MEMORY
{
   name [(attr)] : ORIGIN = origin, LENGTH = len
   ...
}
```

name是链接器脚本中定义的区域的名字，该名字出了脚本范围就没什么意义了。

区域名处于一个独立的名字空间，它不会与符号名，文件名或节名冲突（即区域名可以与符号名，文件名或节名同名）。每个区域名必须是不同的。

attr是可选的属性列表，用于把在链接器脚本中没有显式映射的输入节放到特定的内存区域上。象在 4.6 节**SECTIONS**命令中所介绍的，如果你不为某些输入节指定输出节，链接器就会为这些输入节建立同名的输出节。如果你定义了内存区域的属性，链接器将以它们(这些属性)为标准来为建立的输出节选择内存区域。

attr 属性只能包括下面一些字符：

R
    只读节

W
    可读可写节

X
    可执行节

A
    要分配节

I
    初始化节

L
    同 I

!
    对属性取反

如果一个未被映射的节（在链接器脚本中没有显式指定映射到哪一个输出节）的属性匹配了上面列出的除!外的任何属性，它将被放入具有同样属性的内存区域内。!属性用于取反操作，以便一个未被被映射的节只被放入与它的属性不匹配的内存区域。

origin是定义的内存区域起始地址的表达式。该表达式在实施内存区域分配动作以前必须能被计算出具体值，这就意味着在该表达式中你不能用到节的相对符号。**ORIGIN**关键字可以被略写成org或o(但不能是ORG)。

len是定义的内存区域字节数的表达式。象origin一样，该表达式也必须在实施内存区域分配动作以前必须能被计算出具体值。**LENGTH**关键字可以被略写成len或l。

在下面的例子中，我们指定了可获得的两个内存区域：一个开始与 0 地址，大小是 245K，另一个开始与 0x40000000，大小为 4M。链接器将把脚本中没有指定放入哪块内存区域，并且是只读或可执行的节放入rom标记的内存区域中。在脚本中同样没有被指定放入哪块内存区域的剩下的节（不是只读或可执行的节）将被放入ram标记的内存区域中。

```
MEMORY
{
   rom(rx) : ORIGIN = 0, LENGTH = 256K
   ram(!rx) : org = 0x40000000, l = 4M
}
```

一旦你定义了内存区域，你可以通过>region 这样的输出节属性指导链接器把特定的输出节放入到定义的内存区域中去。例如，如果你定义了一个名为 mem 的内存区域，那你就可以在输出节的定义中用>mem 了。具体请参见 4．6．8．3 节输出节区域。如果输出节没有指定地址，则链接器会把该内存区域内下一个可用的地址给该节。如果被指定放入某块内存区域的多个输出节由于太大而塞不

进去，链接器就会报错。

MEMORY 命令在一般系统上根本用不到，它的用武之地主要是嵌入式系统。该命令的的核心概念，内存区域，对 Window/Linux 这样的系统而言，已无意义。因为链接器链接的输出文件将在操作系统虚拟的地址空间里运行，而该空间对所有的应用程序都是一视同仁的。如果你从事嵌入式开发，倒要好好研究一下。不过听说最新的 VxWorks 已经支持虚拟地址空间，不知道与原来的在开发上有什么大变化。

PHDRS 命令

ELF(Executable and Linkable Format)对象文件格式会用到program header(这就不翻了，翻了反而怪怪的)，也就是众所周知的段，segment（如果阁下进入计算机行业比较早，那请不要把segment与DOS时代程序员天天碰到的那个segment相混淆。名称虽然一样，但实际所指完全不是一回事）。program header描述了程序应当怎样被载入内存当中去。你可以用带 -p 命令行选项的objdump来打印出这些信息。

```
$ objdump -p helloworld

helloworld:     file format elf32-i386

Program Header:
    PHDR off    0x00000034 vaddr 0x08048034 paddr 0x08048034 align 2**2
         filesz 0x000000c0 memsz 0x000000c0 flags r-x
  INTERP off    0x000000f4 vaddr 0x080480f4 paddr 0x080480f4 align 2**0
         filesz 0x00000013 memsz 0x00000013 flags r--
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x000003a6 memsz 0x000003a6 flags r-x
    LOAD off    0x000003a8 vaddr 0x080493a8 paddr 0x080493a8 align 2**12
         filesz 0x00000104 memsz 0x00000108 flags rw-
 DYNAMIC off    0x000003b8 vaddr 0x080493b8 paddr 0x080493b8 align 2**2
         filesz 0x000000c8 memsz 0x000000c8 flags rw-
    NOTE off    0x00000108 vaddr 0x08048108 paddr 0x08048108 align 2**2
         filesz 0x00000020 memsz 0x00000020 flags r—
```

上面列出了例子 helloworld.cpp 编译成 ELF 可执行文件后的 segment（段）。
例子中有 6 个段。其中“PHDR”是 program header 本身所在的段。具体你可能要看 ELF Specification 了。

当你在ELF系统上运行一个ELF格式的可执行文件时，系统载入器读入记载着怎

样载入程序的program header（原文中这句有点不太准确，在program header中描述的是ELF文件各个段的属性，比如段的起始地址，大小，段属性等等，至于怎样载入是ELF loader---一般封装在exec的系统调用中---的事情，并且由于操作系统的不同，载入肯定是有差异的）。只有在program header正确的情况下，载入才能成功。本手册不会介绍系统载入器是怎样解释program header的这样的细节；更多信息，请参见ELF ABI。

默认情况下链接器会生成合适的program header。但在有些情况下，为了精细控制生成的程序，你可能需要指定program header。你可以用**PHDRS**命令来达到这个目的。当链接器看到脚本中有**PHDRS**这个命令时，除了你指定的外，它将不会生成任何其他 program header。

只有在生成ELF格式的输出文件时链接器才会注意到**PHDRS**命令，对于其他格式文件，链接器只是简单的忽略它。

下面是PHDRS命令的语法格式，其中**PHDRS**，**FILEHDR**，**AT**和**FLAGS**是关键字。

```
PHDRS
{
   name type [ FILEHDR ] [ PHDRS ] [ AT ( address ) ]
         [ FLAGS ( flags ) ] ;
}
```

name仅在链接器脚本中的**SECTIONS**命令中被引用，而不会存入输出文件中。Program header的名字处于独立名字空间，不会与符号名，文件名或节名冲突。每个program header必须有不同的名字。

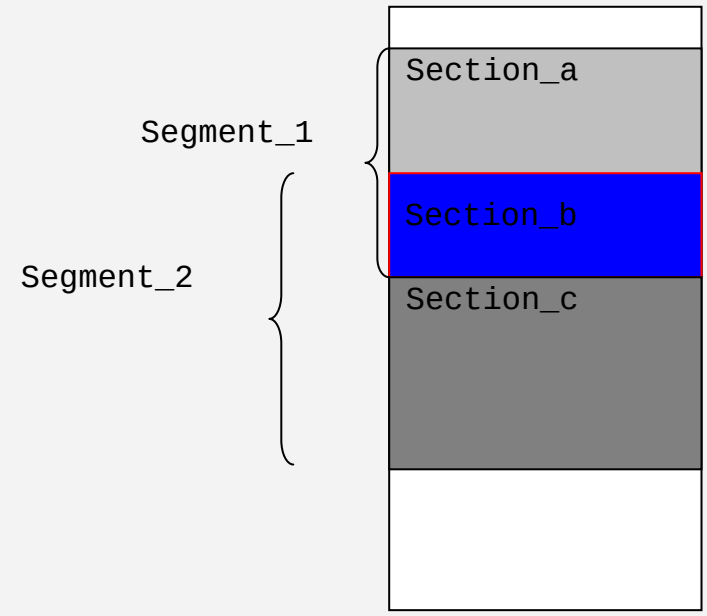
某些类型的program header描述了系统载入器从文件中载入到内存的段的特

性。在链接器脚本中，你可以通过把某些输出节归入[11]一个段来指定段的内容。

你可以用:phdr这个输出节属性来把某个节归入特定段中。请参见 4.6.8.4 节

输出节属性phdr。

把某个节归入多个段是一种很普通的情况，这只意味着内存中的某段包含了另一段（段之间是允许有重合的）。



在上图中 Section_b 既属于 Segment_1，又属于 Segment_2。在实际的内存或可执行文件中，并不会因为 Section_b 属于两个 segment 而又两份。

你可以重复用:phdr 这个属性，using it once for each segment which should contain the section.

如果你用:phdr把一个节归入某个或某几个段中，链接器就会把其后面紧接着的没有用:phdr指明的节都归入同该节所属相同的段中（，一直到出现新的:phdr修饰的节或SECTIONS命令结束）。这是为了方便的缘故，因为通常一系列紧接着的节都会被归入同一个段中。你可以用:NONE来修改这种默认行为，以告诉链接器不要把:NONE修饰的节归入任何段中。

[11]我觉得用“归入”一词比“放入”可能更合适一点。因为这几个节属于这个段，而那几个节属于另外的段，这只是一个逻辑上的归类，而并不是有“段”这么个额外的空间，然后把属于它的“节”放入，就象有个空盒子，然后把东西放入一样。

你还可以用出现在program header类型后面的**FILEHDR**和**PHDRS**关键字进一步说明段的内容。其中**FILEHDR**关键字表示该段包括ELF文件头，而**PHDRS**关键字表示该段应当包括ELF program header本身。

type 可以是下面几种之一，其中括号中的数字是符号的值。

PT_NULL(0)

指示这是一个不用的 program header。

PT_LOAD(1)

指示该 program header 描述的段需要从文件载入。

PT_DYNAMIC(2)

指示该段包含了动态链接的信息。

PT_INTERP(3)

指示该段包含了在系统中能找到的程序解释器的路径名。

PT_NOTE(4)

指示该段包含注释信息。

PT_SHLIB(5)

保留的 program header 类型，由 ELF ABI 定义但没有说明。

PT_PHDR(6)

只是该段包含各个 program header。

expression

一种表示某种program header的数字类型的表达式（即没有象上面那样有PT_XXX那样的符号定义），这可能会被用到。

你可以通过上面**PHDRS**完整命令图中的**AT**（address）表达式来指定某个段应当被载入到内存的某个特定地址，这同输出节属性中的**AT**命令是等效的（请参

见 4.6.8.2 节输出节属性LMA）。而program header中的**AT**命令会覆盖输出节属性中的**AT**。

链接器通常基于段中包含的节的属性来设置段的标志，当然你也可以用**FLAGS**关键字来显式的指定段的标志。flags的值必须是个整数，它将被用于设置program header中的p_flags域。

下面是一个**PHDRS**的例子，这是一个ELF系统上典型的设置program header的例子。

```
PHDRS
{
   headers PT_PHDR PHDRS ;
   interp PT_INTERP ;
   text PT_LOAD FILEHDR PHDRS ;
   data PT_LOAD ;
   dynamic PT_DYNAMIC ;
}

SECTIONS
{
   . = SIZEOF_HEADERS;
   .interp : { *(.interp) } : text : interp
   .text : { *(.text） } : text
   .rodata : { *(.rodata) } /* default  to :text */
   ...
   . = . + 0x1000; /* move to a new page in memory */
   .data : { *(.data) } : data
   .dynamic : { *(.dynamic) } :data :dynamic
```

```
   ...
}
```

VERSION 命令

对于ELF格式文件，链接器支持符号版本（即带版本信息的符号）。符号版本仅对共享库有用。当运行一个被链接到较早版本的共享库的程序时，动态链接器（不是本文讨论的链接器脚本的“链接器”，不要误会）可以利用符号的版本信息来选择（该共享库中输出的）某个函数的特定版本。

你可以在主链接脚本中直接包括一个“版本”脚本，或者你可以提供一个“版本”脚本作为隐含的链接器脚本。你也可以用-version-script命令行选项（来指定版本脚本）。

VERSION 命令的语法是很简单的：

```
VERSION { version-script-commands }
```

版本脚本中的命令格式同 Sun 的 Solaris 2.5 的链接器用到的是一样的。版本脚本定义了一棵版本节点树，在脚本中你指定节点的名字和依赖性。你可以指定哪些符号被绑定到哪些版本节点，你可以限制指定的某些符号的范围，以便在共享库外它们不是全局可见的。

演示版本脚本语言的最简单方法是用一些例子。

```
VERS_1.1 {
   global:
      foo1;
   local:
      old*;
      original*;
      new*;
};
```

```
VERS_1.2 {
   foo2;
} VERS_1.1;

VERS_2.0 {
   bar1; bar2;
} VERS_1.2
```

上例中的版本脚本定义了三个版本节点。第一个版本节点是 VERS_1.1，它不依赖其他版本。脚本把符号 foo1 绑定到 VERS_1.1，它把一些符号限制在局部范围以便这些符号在共享库外不可见。在例子中是用通配符实现的，任何以 old，original 或 new 开头的符号都是局部的。通配符的效果同 shell 中在匹配文件名时用到的是一样的(在 shell 中用通配符匹配文件名被称为“globbing”)。

接下来，版本脚本定义了节点 VERS_1.2，该节点依赖于 VERS_1.1。脚本把符号 foo2 绑定到版本节点 VERS_1.2。

最后，版本脚本定义了节点 VERS_2.0，该节点依赖于 VERS_1.2。脚本绑定符号 bar1 和 bar2 到版本节点 VERS_2.0。

当链接器发现定义在库中的符号没有被指定绑定到某个版本节点时，它将把它们绑定到库中的一个未指定的基础版本上。你可以通过在版本脚本中某处利用 global: *;的声明来把除了未指定的符号外的所有符号绑定到某个给定的版本节点。

版本节点的名字除了供人阅读以外没有特殊意义。名字叫 2.0 的版本节点可能出现在 1.1 与 1.2 之间（即版本节点的名字并不用来表示真正意义上的“版本”，不要

认为 2 在 1.1 和 1.2 之后，这只是名字而已）。然而，对书写版本脚本而言，这可能是一种使人混淆的方法。

节点名字可以被省略，提供它仅仅是为了命名版本脚本中的版本节点。下面的版本脚本没有赋予任何版本给符号，只是选择哪些符号是全局可见的，而哪些则不是。

```
{ global: foo; bar; local: *; }
```

上面的例子中除了 foo 和 bar 符号外，其他符号都是在该共享库外不可见的。

当你把一个应用程序与某个带有版本信息符号的共享库链接时，该应用程序本身知道它需要哪一个版本的符号，它也知道在它要链接的共享库中它需要哪一个版本节点。那样在运行期，动态链接器能够做快速检查，以确定你所链接的库确实提供了应用程序需要用到的所有动态符号对应的所有版本节点。用这种方法是可行的，即动态链接器能确定地知道它需要的所有外部符号在不用搜索遍库中每一个符号的情况下就能被找到。

实际上，相比SunOS做的，符号的版本化是一种更复杂的小版本检查的方法。这里对该基本问题是这样解决的，只在确实需要时才对外部函数的引用进行绑定，而不是在程序启动阶段就绑定所有引用（按需绑定）。如果一个共享库过期了，某个被调用的接口可能没有提供。当应用程序试图调用该接口时，它可能以突然和出人意料的方式失败。而对于已符号版本化的库，当用户启动程序，而其用到的库又太老时，用户将得到一个警告。

在 Sun 的版本化方法上，GNU 链接器做了一些扩展。其中第一项是提供了在源代码中把符号绑定到某个版本节点的能力，即符号定义的场所在源代码文件中，而不是象上面的版本脚本中。这主要减轻了库维护者的负担。你可以在 C 源代码中象下面例子中那样写：

```
__asm__(“.symver original_foo, foo@VERS_1.1”);
```

该例把函数 original_foo 改名成绑定到版本节点 VERS_1.1 的别名 foo。而 local:指令可以用来防止符号 original_foo 被输出。.symver 指令优先于版本脚本中的指定。

GNU 链接器的第二项扩展是允许相同函数的多个版本出现在共享库中。用这种方

法，你可以对某个接口做不兼容的改动，且不需要递增接口所在共享库的主版本号，而依然允许应用程序链接到老的接口。

要这样做，你要在源代码文件中用到多条.symver 指令。这里有个例子：

```
__asm__(“.symver original_foo, foo@”);
__asm__(“.symver old_foo, foo@VERS_1.1”);
__asm__(“.symver old_foo1, foo@VERS_1.2”);
__asm__(“.symver new_foo, foo@VERS_2.0”);
```

在上例中，foo@代表符号 foo 绑定到该符号未指定的基础版本上。含有该例子的源代码文件定义了 4 个 C 函数：original_foo，old_foo，old_foo1 和 new_foo。

当一个给定符号有多处定义时，就需要有某种方法来指定外部引用该符号时要被绑定的默认版本。你可以用 foo@@VERS_2.0 这一类的.symver 指令来这么做。用这种方式，你只可以指定一个符号的某个版本为默认版本，否则你要对同一个符号有多个定义。

如果你希望绑定一个引用到共享库中的符号的某个特定版本，你可以用方便起见而设的别名（例如象 old_foo），或者你可以用.symver 指令来明白地绑定到该函数的一个外部版本上。

你也可以在版本脚本中指定语言：

```
VERSION extern “lang” { version-script-commands }
```

支持的语言有C，C++和Java。链接器将在链接阶段遍历符号列表，并且在把符号与version-script-commands中指定的模式匹配以前，要根据lang来解码（demangle）它们（符号列表中的symbol）。

链接器脚本中的表达式

链接器脚本中的表达式语法同C语言中的是一样的。脚本中所有的表达式被评估

相同位数的整数，在主机(host)和目标机(target)都是 32 位时（这里原文中的host是指链接器所运行的平台，而target是指链接器生成的输出文件运行的平台。一般情况下，这两者是没有什么区别的。比如你在一台Linux机器上编译并链接源代码并得到一个可执行文件，你可以就在本地运行，或拷贝到另外的Linux机器上运行。前者host同target时同一台机器，而后者则是不同的机器，但host与target的类型是相同的，即你不可能让一个在Linux下编译的程序在Windows系统上运行---这里请高手们不要与我较真，我这里在说明一个问题，而不想牵扯进什么通过仿真环境是可以做到这类程序在不同系统间运行的。但在编译嵌入式系统的可执行文件时，一般host与target是不同的。你一般在host机上做交叉编译，得到输出文件后，再把该文件载入到嵌入式系统运行的机器，也就是target机上运行,而在这种情况下，host与target可能完全不一样。不知道我的理解对不对，请指教!），表达式得到的是 32 位值，反之则是 64 位值。

你可以在表达式中使用并设置符号的值。

链接器允许在表达式中使用一些它定义的特殊目的的内建函数。

常数

所以的常数都是整数。

像在 C 语言中一样，链接器认为以 0 开头的整数是 8 进制的，以 0x 或 0X 开头的是 16 进制的，对于其他整数认为是 10 进制的。

另外,你可以在一个常数后面跟 K 和 M 来表示该数要乘上 1024 或 1024 * 1024。例如，下面三行都表示相同的值：

```
_fourk_1 = 4K;
_fourk_2 = 4096;
_fourk_3 = 0x1000;
```

符号名

除非用引号引起来，否则符号名的首字符必须是字母，下划线或点(dot)，其后

可以包括字母，数字，下划线，点和连字符。不加引号的符号名不能与链接器脚本的关键字冲突。你可以指定包含有不允许的字符或与关键字同名的词的符号，只要把该符号名用双引号引起来就行：

```
“SECTION” = 9;
“with a space” = “also with a space” + 10;
```

由于符号可以包含许多非字母的字符，所以用空格来分隔符号是很安全的。比如，A-B 是符号，而 A – B 是包含减法的表达式。

位置指示器

特殊链接器变量 dot（.）总是指向当前的输出位置。由于.总是指的是输出节中的位置，所以它只可能出现在 SECTIONS 命令的表达式中。.符号能出现在表达式中的普通符号允许出现的任何地方。

对.的赋值将使得位置指示器被改变。这时常被用于在输出节中创建空洞。位置指示器不可以回退（即象中国象棋中的小卒子，只能前进不能后退。唉，有时候人生也是这样！）。

```
SECTIONS
{
   output:
   {
      file1(.text)
      . = . + 1000;
      file2(.text)
      . = . + 1000;
      file3(.text)
   } = 0x12345678;
}
```

在上例中，来自file1 的.text节被放置在output输出节的开头，接着是 1000

字节的空隙，然后是来自file2 的.text节，其后面也跟着 1000 字节的空隙，最后是来自file3 的.text节。“=0x12345678”指定将在上面的两处空隙中填充以 0x12345678(请参考 4.6.8.5 节输出节填充属性)。

注意：.一般指的是从当前包含的对象开始的字节偏移量，通常该对象就是 SECTIONS 声明，它从 0 地址开始。但.也能被用作绝对地址。如果.被用在节描述的内部，则它指的只能是从该节开始的字节偏移，而不是一个绝对地址。.指向绝对地址的例子见下面：

```
SECTIONS
{
   . = 0x100
   .text: {
      *(.text)
      . = 0x200
   }
   . = 0x500
   .data: {
      *（.data）
      . += 0x600
   }
}
```

.text节将被赋以 0x100 的开始地址和正好 0x200 字节的大小，即使没有足够空间把.text输入节放入该区域。（如果有太多数据而造成.有往回退的尝试，将报错）（比如所有.text输入节有 150 字节大小，则位置指示器将会从 0x100 向前移动到 0x250，当碰到. = 0x200 时，要求位置指示器回退，而这是不允许的，所以报错）。.data节开始于 0x500，它将在.data的所有输入节放置后的尾部及.data输出节的

尾部之间有额外的 600 字节空间。

运算符

链接器理解标准 C 集中的算术运算符，同样的形式和优先级：

precedence	associativity	operators	notes
(highest)			
1	left	! - ~	(1)
2	left	* / %	
3	left	+ -	
4	left	>> <<	
5	left	== != > < <= >=	
6	left	&	
7	left	|	
8	left	&&	
9	left	||	
10	right	? :	
11	right	&= += -= *= /=	(2)
(lowest)			

注意：（1）前缀运算符 （2）请参见 4.5 节为symbol赋值。

表达式求值

链接器用懒惰的方式来对表达式求值(所谓懒惰方式，即不到决定必要时，不会做)。

一个表达式的节

内部函数

链接器脚本语言包括一些可以用在脚本表达式中的内部函数。

ABSOLUTE(exp)

返回表达式exp的绝对值（不能重定位，相对于非负的）。首要用处是在节定义的内部给符号赋以绝对值，而该符号的值通常是相对于节的。请参见4.10.6 节一个表达式的节。

ADDR(section)

返回名为被命名节的VMA绝对值(section为节名)。你的脚本必须在以前定义过该节的位置。在下面的例子中，symbol_1 和symbol_2 被赋以相同的值。

```
SECTIONS { ...
   .output1 :
   {
   start_of_output_1 = ABSOLUTE(.);
   ...
   }
   .output :
   {
   symbol_1 = ADDR(.output1);
   symbol_2 = start_of_output_1;
   }
...   }
```

ALIGN(exp)

把位置指示器（.）对齐到下一个exp的边界。ALIGN并不会改变位置指示器的值 --- 它只是对它（位置指示器）做算术运算。这里有个例子，在该例中把输出节.data的起始地址对齐到前面输出节结束后的下一个 0x2000 边界上，还有设置该节中的一个变量的地址为输入节后的下一个 0x8000

边界上。

```
SECTIONS { ...
   .data ALIGN(0x2000): {
      *(.data)
      variable = ALIGN(0x8000);
   }
... }
```

上例中第一个ALIGN指定节的位置，它被作为节定义（请参见 4.6.3 节输出节介绍）中的可选address属性。第二个ALIGN被用来定义符号的值。

内部函数 NEXT 与 ALIGN 紧密相关。

BLOCK(exp)

这是 ALIGN 的同义词。以便兼容老的链接器脚本。在设置输出节地址时时常能看到。

DATA_SEGMENT_ALIGN(maxpagesize, commonpagesize)

该函数等同于或者是
(ALIGN(maxpagesize) + (. & (maxpagesize - 1)))
或者是
(ALIGN(maxpagesize) + (. & (maxpagesize - commonpagesize)))
的表达式。到底是使用哪一个表达式取决于两者产生的数据段（该表达式的地址到DATA_SEGMENT_END函数定义的地址间的区域）哪一个更少。如果是后者，就表明有commonpagesize的运行期内存被节省但有至多commonpagesize字节的磁盘空间被浪费（既要省内存又要省磁盘空间这样的两全其美的事难找）。

该表达式仅能被用在 SECTIONS 命令中，但不能用在任何输出节描述中，并且在脚本中只能出现一次。commonpagesize 应当小于等于maxpagesize，并且应当是对象试图被优化的系统页面大小（while still working on system page sizes up to maxpagesize）。

例子如下：

```
. = DATA_SEGMENT_ALIGN(0x10000, 0x2000);
```

DATA_SEGMENT_END(exp)

配合上面 DATA_SEGMENT_ALIGN 计算目的，该函数设置数据段的结尾。

```
. = DATA_SEGMENT_END(.);
```

DEFINED(symbol)

如果 symbol 在链接器的全局符号表中及被定义，返回 1，否则返回 0。你可以用该函数来为符号提供默认值。比如，下面的脚本片段展示了怎样把全局符号 begin 设置成.text 节的首地址 --- 但如果名字为 begin 的符号已经存在，那该符号就保留原值：

```
SECTIONS { ...
   .text : {
      begin = DEFINED（begin） ? begin : . ;
      ...
   }
   ...
}
```

LOADADDR(section)

返回名为section的节的绝对LMA(Loadable Memory Address)，这通常同**ADDR**返回的值是一样的，但如果在输出节定义时指定了**AT**属性，这两条命令返回的值就不同了(请参见 4.6.8.2 节输出节属性LMA)。

MAX(exp1, exp2)

返回 exp1 与 exp2 中的大值。

MIN(exp1, exp2)

返回 exp1 与 exp2 中的小值。

NEXT(exp)

返回下一个是exp整数倍的未分配地址。该函数与**ALIGN**(exp)非常相似。

除非你用**MEMORY**命令为输出文件指定了不连续的内存，否则这两个函数是相同的。

SIZEOF(section)

如果节名为 section 的节已经被分配，则返回该节的大小（占多少字节）。如果当计算该表达式时，该节还没有被分配，链接器将报错。在下面的例子中，symbol_1 和 symbol_2 被赋以相同的值：

```
SECTIONS { ...
   .output {
      .start = . ;
      ...
      .end = . ;
      }
   symbol_1 = .end - .start ;
   symbol_2 = SIZEOF(.output);
... }
```

SIZEOF_HEADERS

sizeof_headers

返回输出文件头大小，这个信息将出现在输出文件的开始处。当设置输出文件的第一个节的起始地址时，你可能要用到该值。If you choose, to facilitate paging.

在生成ELF格式的输出文件时，如果链接器脚本用到了SIZEOF_HEADERS的内部函数，那么在定出所有节的地址和大小以前，链接器必须先计算得到program header的值。如果链接器后来才发觉它需要额外的program header，它将报“not enough room for program headers”的错。为了避免这种错，你要么不要用SIZEOF_HEADERS函数，要么你重写你的

链接器脚本以避免强迫链接器需要产生额外program header的情况，要么你用PHDRS命令(请参见4.8节PHDRS命令)自己定义program header。

隐含链接器脚本

如果你指定的输入文件，链接器发现既不是对象文件(obj文件)又不是库文件(很多翻译把archive file翻作归档文件，我看着莫名其妙)，那它将把该文件当作链接器脚本。如果该文件不能被当作链接器脚本解析，链接器将报错。

隐含的链接器脚本不会替代默认的链接器脚本。

典型的链接器脚本可能只包括符号的赋值，或只有**INPUT**，**GROUP**，**VERSION**命令(很显然，gnu的linker不是这么简单，非但不简单，而且蛮复杂的)。

Any input files read because of an implicit linker script will be read at the position in the command line where the implicit linker script was read.这可能影响到库文件的搜索。

附录

Linux 下普通应用程序的链接器脚本解析

```
GNU ld version 2.13.90.0.2 20020802
  Supported emulations:
   elf_i386
   i386linux
   elf_i386_glibc21
using internal linker script:
==================================================
/* Script for -z combreloc: combine and sort reloc sections */
OUTPUT_FORMAT("elf32-i386", "elf32-i386",      因为我的系统是 x86 的 Linux
          "elf32-i386")
```

```
OUTPUT_ARCH(i386)
ENTRY(_start)                所有用该链接器链接的应用程序都已_start 为入口点

SEARCH_DIR("/usr/i386-redhat-linux/lib");      SEARCH_DIR("/usr/lib");
SEARCH_DIR("/usr/local/lib"); SEARCH_DIR("/lib");

上面为设置链接时搜索库文件的目录，即当你在代码中调用了象 open()函数后，到哪儿去找到
该函数所在的库文件。

/* Do we need any of these for elf?
   __DYNAMIC = 0;    */
SECTIONS        下面就开始产生输出节了，也就是你通过 objdump –h 列出来的节
{
  /* Read-only sections, merged into text segment: */
  . = 0x08048000 + SIZEOF_HEADERS;

  0x08048000 ！这就是为什么 Linux 下的应用程序都被放置到同一个位置开始的原因。
  下面是把各个输入文件（即 obj 文件）中的同名节合并到输出文件中，并且节名不变。
  可以对照着 objdump –h 的输出来看一下下面的脚本。
```

```
[wzhou@DEBUG wzhou]$  objdump -h helloworld

helloworld:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .interp       00000013  080480f4  080480f4  000000f4  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000020  08048108  08048108  00000108  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
```

```
 2 .hash         00000028  08048128  08048128  00000128  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 3 .dynsym       00000050  08048150  08048150  00000150  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 4 .dynstr       0000004c  080481a0  080481a0  000001a0  2**0
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 5 .gnu.version  0000000a  080481ec  080481ec  000001ec  2**1
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 6 .gnu.version_r 00000020  080481f8  080481f8  000001f8  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 7 .rel.dyn      00000008  08048218  08048218  00000218  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 8 .rel.plt      00000010  08048220  08048220  00000220  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 9 .init         00000018  08048230  08048230  00000230  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
10 .plt          00000030  08048248  08048248  00000248  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
11 .text         000000fc  08048278  08048278  00000278  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
12 .fini         0000001c  08048374  08048374  00000374  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
13 .rodata       00000016  08048390  08048390  00000390  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
14 .data         0000000c  080493a8  080493a8  000003a8  2**2
                 CONTENTS, ALLOC, LOAD, DATA
15 .eh_frame     00000004  080493b4  080493b4  000003b4  2**2
                 CONTENTS, ALLOC, LOAD, DATA
16 .dynamic      000000c8  080493b8  080493b8  000003b8  2**2
                 CONTENTS, ALLOC, LOAD, DATA
```

```
 17 .ctors       00000008  08049480  08049480  00000480  2**2
                 CONTENTS, ALLOC, LOAD, DATA
 18 .dtors       00000008  08049488  08049488  00000488  2**2
                 CONTENTS, ALLOC, LOAD, DATA
 19 .jcr         00000004  08049490  08049490  00000490  2**2
                 CONTENTS, ALLOC, LOAD, DATA
 20 .got         00000018  08049494  08049494  00000494  2**2
                 CONTENTS, ALLOC, LOAD, DATA
 21 .bss         00000004  080494ac  080494ac  000004ac  2**2
                 ALLOC
 22 .comment     00000132  00000000  00000000  000004ac  2**0
                 CONTENTS, READONLY
 23 .debug_aranges 00000058  00000000  00000000  000005e0  2**3
                 CONTENTS, READONLY, DEBUGGING
 24 .debug_pubnames 00000025  00000000  00000000  00000638  2**0
                 CONTENTS, READONLY, DEBUGGING
 25 .debug_info   00000c85  00000000  00000000  0000065d  2**0
                 CONTENTS, READONLY, DEBUGGING
 26 .debug_abbrev 00000127  00000000  00000000  000012e2  2**0
                 CONTENTS, READONLY, DEBUGGING
 27 .debug_line   000001f2  00000000  00000000  00001409  2**0
                 CONTENTS, READONLY, DEBUGGING
 28 .debug_frame  00000014  00000000  00000000  000015fc  2**2
                 CONTENTS, READONLY, DEBUGGING
 29 .debug_str    0000098a  00000000  00000000  00001610  2**0
                 CONTENTS, READONLY, DEBUGGING
```

从上面的例子的输出可看到节的 VMA 地址是按脚本中定义的次序出现的（VMA 为 0 的是记录辅助信息的节，比如象记录了调试信息的节，在运行时是不用载入内存的，只是供 debugger 用）。这里最初始的节.interp 开始与 080480f4，在该节与输出文件的最初始位置

0x08048000 之间是 ELF 的文件头（具体信息请你参考 ELF Format）

```
  .interp         : { *(.interp) }
  .hash           : { *(.hash) }
  .dynsym         : { *(.dynsym) }
  .dynstr         : { *(.dynstr) }
  .gnu.version    : { *(.gnu.version) }
  .gnu.version_d  : { *(.gnu.version_d) }
  .gnu.version_r  : { *(.gnu.version_r) }
  .rel.dyn        :
    {
      *(.rel.init)
      *(.rel.text .rel.text.* .rel.gnu.linkonce.t.*)
      *(.rel.fini)
      *(.rel.rodata .rel.rodata.* .rel.gnu.linkonce.r.*)
      *(.rel.data .rel.data.* .rel.gnu.linkonce.d.*)
      *(.rel.tdata .rel.tdata.* .rel.gnu.linkonce.td.*)
      *(.rel.tbss .rel.tbss.* .rel.gnu.linkonce.tb.*)
      *(.rel.ctors)
      *(.rel.dtors)
      *(.rel.got)
      *(.rel.bss .rel.bss.* .rel.gnu.linkonce.b.*)
    }
  .rela.dyn       :
    {
      *(.rela.init)
      *(.rela.text .rela.text.* .rela.gnu.linkonce.t.*)
      *(.rela.fini)
      *(.rela.rodata .rela.rodata.* .rela.gnu.linkonce.r.*)
      *(.rela.data .rela.data.* .rela.gnu.linkonce.d.*)
      *(.rela.tdata .rela.tdata.* .rela.gnu.linkonce.td.*)
```

```
    *(.rela.tbss .rela.tbss.* .rela.gnu.linkonce.tb.*)
    *(.rela.ctors)
    *(.rela.dtors)
    *(.rela.got)
    *(.rela.bss .rela.bss.* .rela.gnu.linkonce.b.*)
  }
  .rel.plt       : { *(.rel.plt) }
  .rela.plt      : { *(.rela.plt) }
  .init          :
  {
    KEEP (*(.init))
  } =0x90909090
```

在把各个输入节合并成同名的输出节时，由于输入节被放置时还是有基本的地址对齐要求的，所以会造成期间有空隙。这里的空隙用 0x90 来填充，而该字节对应的指令码是 nop，即 no operation，一个执行的效果就是什么也没执行的指令。

比如，象下面的函数 call_gmon_start()与__do_global_dtors_aux()之间就有空隙。call_gmon_start()结束与 0x080482bd，而__do_global_dtors_aux()开始与 0x080482c0，期间有两个字节的空隙，就用 nop 指令来填充。

```
  0804829c <call_gmon_start>:
   804829c: 55                   push   %ebp
   804829d: 89 e5                mov    %esp,%ebp
   804829f: 53                   push   %ebx
   80482a0: 50                   push   %eax
   80482a1: e8 00 00 00 00           call   80482a6 <call_gmon_start+0xa>
   80482a6: 5b                   pop    %ebx
   80482a7: 81 c3 ee 11 00 00        add    $0x11ee,%ebx
   80482ad: 8b 83 14 00 00 00        mov    0x14(%ebx),%eax
   80482b3: 85 c0                test   %eax,%eax
   80482b5: 74 02                je     80482b9 <call_gmon_start+0x1d>
   80482b7: ff d0                call   *%eax
```

```
 80482b9: 8b 5d fc            mov    0xfffffffc(%ebp),%ebx
 80482bc: c9                  leave
 80482bd: c3                  ret
 80482be: 90                  nop
 80482bf: 90                  nop

080482c0 <__do_global_dtors_aux>:
 80482c0: 55                  push   %ebp
 80482c1: 89 e5               mov    %esp,%ebp
 80482c3: 83 ec 08            sub    $0x8,%esp
 80482c6: 80 3d ac 94 04 08 00    cmpb   $0x0,0x80494ac
 80482cd: 75 29               jne    80482f8 <__do_global_dtors_aux+0x38>
```

（脚本源码继续）

```
.plt           : { *(.plt) }
.text          :     这就是最正宗的代码段，有的文档称为正文段
{
  *(.text .stub .text.* .gnu.linkonce.t.*)
  /* .gnu.warning sections are handled specially by elf32.em.  */
  *(.gnu.warning)
} =0x90909090       空隙同样用 nop 指令填充
.fini          :
{
  KEEP (*(.fini))
} =0x90909090
PROVIDE (__etext = .);     到这里代码段就结束了，所以下面定义了 3 个指向代码段
PROVIDE (_etext = .);      结束的变量，__etext，_etext，etext。
PROVIDE (etext = .);       即下面就是数据段了。
.rodata        : { *(.rodata .rodata.* .gnu.linkonce.r.*) }
```

```
.rodata1       : { *(.rodata1) }
.eh_frame_hdr : { *(.eh_frame_hdr) }
/* Adjust the address for the data segment.  We want to adjust up to
   the same address within the page on the next page up.  */
. = DATA_SEGMENT_ALIGN(0x1000, 0x1000);
/* Ensure the __preinit_array_start label is properly aligned.  We
   could instead move the label definition inside the section, but
   the linker would then create the section even if it turns out to
   be empty, which isn't pretty.  */
. = ALIGN(32 / 8);
PROVIDE (__preinit_array_start = .);
.preinit_array    : { *(.preinit_array) }
PROVIDE (__preinit_array_end = .);
PROVIDE (__init_array_start = .);
.init_array    : { *(.init_array) }
PROVIDE (__init_array_end = .);
PROVIDE (__fini_array_start = .);
.fini_array    : { *(.fini_array) }
PROVIDE (__fini_array_end = .);
.data          :                    最正宗的数据段
{
  *(.data .data.* .gnu.linkonce.d.*)
  SORT(CONSTRUCTORS)
}
.data1         : { *(.data1) }
.tdata     : { *(.tdata .tdata.* .gnu.linkonce.td.*) }
.tbss      : { *(.tbss .tbss.* .gnu.linkonce.tb.*) *(.tcommon) }
.eh_frame      : { KEEP (*(.eh_frame)) }
.gcc_except_table   : { *(.gcc_except_table) }
```

```
  .dynamic       : { *(.dynamic) }
  .ctors         :
  {
    /* gcc uses crtbegin.o to find the start of
       the constructors, so we make sure it is
       first.  Because this is a wildcard, it
       doesn't matter if the user does not
       actually link against crtbegin.o; the
       linker won't look for a file to match a
       wildcard.  The wildcard also means that it
       doesn't matter which directory crtbegin.o
       is in.  */
    KEEP (*crtbegin.o(.ctors))
    /* We don't want to include the .ctor section from
       from the crtend.o file until after the sorted ctors.
       The .ctor section from the crtend file contains the
       end of ctors marker and it must be last */
    KEEP (*(EXCLUDE_FILE (*crtend.o ) .ctors))
    KEEP (*(SORT(.ctors.*)))
    KEEP (*(.ctors))
  }
  .dtors         :
  {
    KEEP (*crtbegin.o(.dtors))
    KEEP (*(EXCLUDE_FILE (*crtend.o ) .dtors))
    KEEP (*(SORT(.dtors.*)))
    KEEP (*(.dtors))
  }
  .jcr           : { KEEP (*(.jcr)) }
```

```
  .got           : { *(.got.plt) *(.got) }
  _edata = .;              数据段结束符号，_edata，edata
  PROVIDE (edata = .);
  __bss_start = .;         bss 段开始了，提供__bss_start 符号来标记
  .bss           :
  {
   *(.dynbss)
   *(.bss .bss.* .gnu.linkonce.b.*)
   *(COMMON)
   /* Align here to ensure that the .bss section occupies space up to
      _end.  Align after .bss to ensure correct alignment even if the
      .bss section disappears because there are no input sections.  */
   . = ALIGN(32 / 8);
  }
  . = ALIGN(32 / 8);
  _end = .;                                  整个可执行文件在内存中映像的结束标志
  PROVIDE (end = .);
  . = DATA_SEGMENT_END (.);
  /* Stabs debugging sections.  */      stab 格式的调试信息
  .stab          0 : { *(.stab) }          如果你在用 gcc 编译时，指定了-gstabs
  .stabstr       0 : { *(.stabstr) }       或-gstabs+，则生成的调试信息的格式是
  .stab.excl     0 : { *(.stab.excl) }  所谓 stab 格式的，一种比较老，但应用
  .stab.exclstr  0 : { *(.stab.exclstr) }  还蛮广泛的调试信息记录格式
  .stab.index    0 : { *(.stab.index) }
  .stab.indexstr 0 : { *(.stab.indexstr) }     由于调试信息只是被 debugger
  .comment       0 : { *(.comment) }           用的，即使要载入内存，那也是
debugger 负责载入并解释，与可执行文件载入器无关，所以这里把各调试信息的节的地址都
设为 0。
  /* DWARF debug sections.
```

```
    Symbols in the DWARF debugging sections are relative to the beginning
    of the section so we begin them at 0.  */
```

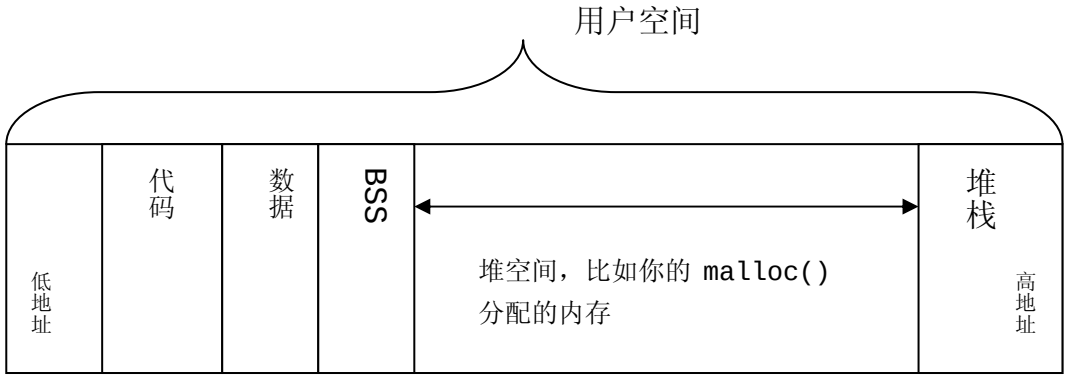
DWARF 是一种比较新的调试信息记录格式，也可能是以后类 Unix 平台上应用的调试信息记录标准吧。现在你常用的在 gcc 下编译时用的-g 选项生成的调试信息用的是 gdb 特有格式。DWARF 已经推出 3 版了，最新的是第 3 版，即 DWARF 3。

```
  /* DWARF 1 */
  .debug          0 : { *(.debug) }
  .line           0 : { *(.line) }
  /* GNU DWARF 1 extensions */
  .debug_srcinfo  0 : { *(.debug_srcinfo) }
  .debug_sfnames  0 : { *(.debug_sfnames) }
  /* DWARF 1.1 and DWARF 2 */
  .debug_aranges  0 : { *(.debug_aranges) }
  .debug_pubnames 0 : { *(.debug_pubnames) }
  /* DWARF 2 */
  .debug_info     0 : { *(.debug_info .gnu.linkonce.wi.*) }
  .debug_abbrev   0 : { *(.debug_abbrev) }
  .debug_line     0 : { *(.debug_line) }
  .debug_frame    0 : { *(.debug_frame) }
  .debug_str      0 : { *(.debug_str) }
  .debug_loc      0 : { *(.debug_loc) }
  .debug_macinfo  0 : { *(.debug_macinfo) }
  /* SGI/MIPS DWARF 2 extensions */
  .debug_weaknames 0 : { *(.debug_weaknames) }
  .debug_funcnames 0 : { *(.debug_funcnames) }
  .debug_typenames 0 : { *(.debug_typenames) }
  .debug_varnames  0 : { *(.debug_varnames) }
```

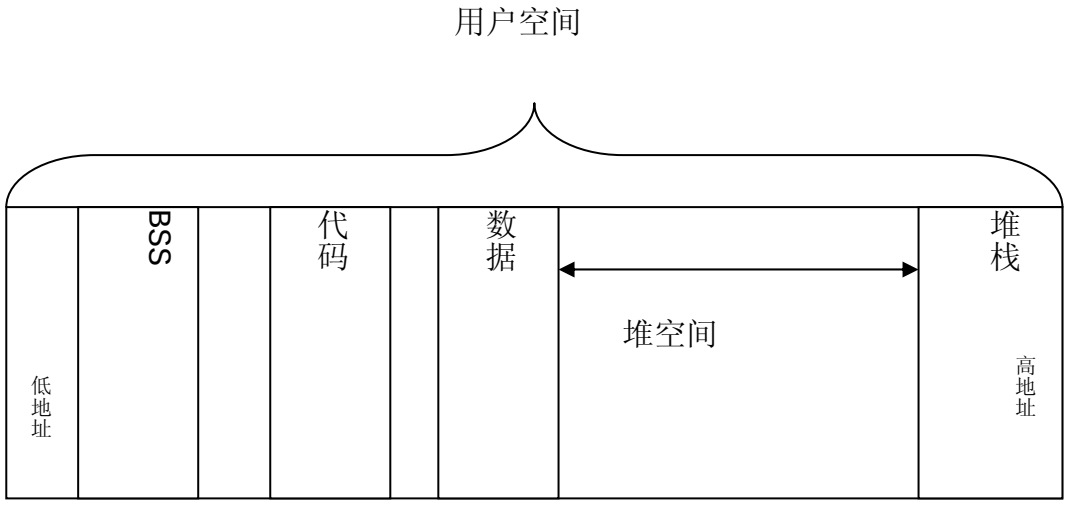
```
}
```

我对上面的脚本做了大致的注释，如果你看过《链接器脚本介绍》，则应该能明白绝大部分脚本行的含义。

我们程序员在很早（应该是你的第一本编译型语言程序开发教材）就被告诉，我们写的程序在电脑中运行时有下面类似的内存分布：



上图是一种很粗略的描述，但大致是对的（当然也只能说是“大致”，因为一细究，就有很多漏洞）。正因为它的“大致”性，所以它不但适用于几乎所有通用Unix 平台，连 Windows 也适用。基于上面脚本链接器的知识，我们知道，我们完全可以生成一个怪胎程序，比如象下图中那样：



上面的“BSS”，“代码”和“数据”你可以按你自己的喜好排列，但堆栈和堆空间不是链接器控制的范围，所以你就没办法了。要这么做，对你而言，易如反掌，你写的代码一行都不用变，唯一要变的就是你需要指定自己的链接器脚本。在该脚本中你可以按你的喜好来对各个节进行排序。但关键是这样的程序还能运行吗？答案是，“否”！因为现在Linux中的可执行文件载入器实际上认为程序的layout是先代码后数据最后是BSS。所以你的怪胎程序从ELF Format的角度

而言，是完全合法的，但就是没法运行。当然要运行也不是不可能，但你得写自己的“载入器”了。当然这不在本文讨论之列，请参考Linux源码。我有一篇文章[12]详细解读了Linux下载入ELF可执行文件的内核源码，可参考。

Linux 内核链接器脚本解析

链接当然是因平台而异的，这里看一下基于 i386 的 Linux 内核链接器脚本。

2.4.18 内核

```
/* ld script to make i386 Linux kernel
 * Written by Martin Mares <mj@atrey.karlin.mff.cuni.cz>;
 */
OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)
SECTIONS
{
  . = 0xC0000000 + 0x100000;     Linux 内核被定址在 3G+1M 之上。3G 是因为 Linux 系统的内核空间被分配在 32 位 CPU 最大可寻址范围 4G 的最后 1G，而 1M 显然是为了避开 x86 上的元勋系统 DOS 的地盘。
  _text = .;          /* Text and read-only data */
                      符号_text 指向内核代码的开始，即 3G+1M
  .text : {           合并各个 obj 文件中的相应节到代码段，空隙用 nop 指令填充
    *(.text)          .fixup 节中包含内核中的异常处理器
    *(.fixup)
    *(.gnu.warning)
    } = 0x9090
```

[12] 《Linux下ELF可执行文件载入源码分析》

```
  _etext = .;         /* End of text section */
```

符号_etext 指向代码段的尾部，这样整个内核代码的范围可由[.text, _etext)表示

```
  .rodata : { *(.rodata) *(.rodata.*) }
  .kstrtab : { *(.kstrtab) }

  . = ALIGN(16);      /* Exception table */
  __start___ex_table = .;
  __ex_table : { *(__ex_table) }
  __stop___ex_table = .;
```

异常处理表在__ex_table 节中，这里把所有 obj 文件中的表归类，并由 __start___ex_table 来指向表头，__stop___ex_table 来指向表尾。

这两个符号在内核源代码中是可以访问的。所以在阅读内核代码是看到引用这两个变量，但又没找到定义的地方(在内核代码中只有对这两个变量的 extern 定义)，就不用奇怪了。

```
  __start___ksymtab = .; /* Kernel symbol table */
  __ksymtab : { *(__ksymtab) }
  __stop___ksymtab = .;
```

这是内核符号表的处理。你在内核中输出的符号就被安排在该节。同样用两个符号来指示该表的头尾，__start___ksymtab，__stop___ksymtab。在源代码中载入内核模块时要 fix 被载入模块对其他内核模块中函数的引用时就会查询该表。

```
  .data : {           /* Data */
    *(.data)
    CONSTRUCTORS
    }

  _edata = .;         /* End of data section */
```

_edata 符号指向内核数据段的结束

```
  . = ALIGN(8192);        /* init_task */
  .data.init_task : { *(.data.init_task) }
```

Linux 上 0 号进程(内核进程)的堆栈是固定的，就在这里指定。每个进程有 2 个页面，8K 空间的内核堆栈，同时该堆栈必须对齐在 8K 边界上。

```
  . = ALIGN(4096);        /* Init code and data */
  __init_begin = .;       init 节的代码和数据是在系统初始化好以后，可以丢弃的，
```

从而节省出一部分物理内存，毕竟内核省出来的是货真价实的物理内存啊！这个应该是向 Windows 学的吧。我记得 Windows 9.X 内核就有这逻辑，Windows NT 当然更有了。这么说不会招 Linux 爱好者骂吧？但事实是，该逻辑在 Linux 上的实现肯定晚于 Windows。这是我 Windows 9.X 驱动开发与 Linux 读码经验告诉我的，总不能让我不评良心说话吧！^_^

```
  .text.init : { *(.text.init) }
  .data.init : { *(.data.init) }
  . = ALIGN(16);
  __setup_start = .;
  .setup.init : { *(.setup.init) }
  __setup_end = .;
  __initcall_start = .;
  .initcall.init : { *(.initcall.init) }
  __initcall_end = .;
  . = ALIGN(4096);
  __init_end = .;

  . = ALIGN(4096);
  .data.page_aligned : { *(.data.idt) }

  . = ALIGN(32);
```

```
  .data.cacheline_aligned : { *(.data.cacheline_aligned) }

  __bss_start = .;       /* BSS */      符号__bss_start 指向 BSS 的开始
  .bss : {
    *(.bss)
    }
  _end = . ;                          _end 指向整个内核空间的结尾

  /* Sections to be discarded */
  /DISCARD/ : {       要丢弃的节，即不放入 Linux 内核文件的节
    *(.text.exit)
    *(.data.exit)
    *(.exitcall.exit)
    }

  /* Stabs debugging sections.  */      在脚本中只特别关照了 stab 调试格式
  .stab 0 : { *(.stab) }
  .stabstr 0 : { *(.stabstr) }
  .stab.excl 0 : { *(.stab.excl) }
  .stab.exclstr 0 : { *(.stab.exclstr) }
  .stab.index 0 : { *(.stab.index) }
  .stab.indexstr 0 : { *(.stab.indexstr) }
  .comment 0 : { *(.comment) }
}
```

2.6.20 内核

```
/* ld script to make i386 Linux kernel
 * Written by Martin Mares <mj@atrey.karlin.mff.cuni.cz>;
 *
```

```
 * Don't define absolute symbols until and unless you know that symbol
 * value is should remain constant even if kernel image is relocated
 * at run time. Absolute symbols are not relocated. If symbol value should
 * change if kernel is relocated, make the symbol section relative and
 * put it inside the section definition.
 */

/* Don't define absolute symbols until and unless you know that symbol
 * value is should remain constant even if kernel image is relocated
 * at run time. Absolute symbols are not relocated. If symbol value should
 * change if kernel is relocated, make the symbol section relative and
 * put it inside the section definition.
 */
#define LOAD_OFFSET __PAGE_OFFSET

#include <asm-generic/vmlinux.lds.h>
#include <asm/thread_info.h>
#include <asm/page.h>
#include <asm/cache.h>
#include <asm/boot.h>

OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(phys_startup_32)
jiffies = jiffies_64;
_proxy_pda = 0;

PHDRS {
    text PT_LOAD FLAGS(5);   /* R_E */
```

```
    data PT_LOAD FLAGS(7);   /* RWE */
    note PT_NOTE FLAGS(4);   /* R__ */
}
SECTIONS
{
  . = LOAD_OFFSET + LOAD_PHYSICAL_ADDR;
  phys_startup_32 = startup_32 - LOAD_OFFSET;
  /* read-only */
  .text : AT(ADDR(.text) - LOAD_OFFSET) {
    _text = .;        /* Text and read-only data */
    *(.text)
    SCHED_TEXT
    LOCK_TEXT
    KPROBES_TEXT
    *(.fixup)
    *(.gnu.warning)
    _etext = .;          /* End of text section */
  } :text = 0x9090

  . = ALIGN(16);     /* Exception table */
  __ex_table : AT(ADDR(__ex_table) - LOAD_OFFSET) {
    __start___ex_table = .;
     *(__ex_table)
    __stop___ex_table = .;
  }

  RODATA

  BUG_TABLE
```

```
. = ALIGN(4);
.tracedata : AT(ADDR(.tracedata) - LOAD_OFFSET) {
  __tracedata_start = .;
  *(.tracedata)
  __tracedata_end = .;
}

/* writeable */
. = ALIGN(4096);
.data : AT(ADDR(.data) - LOAD_OFFSET) {  /* Data */
  *(.data)
  CONSTRUCTORS
  } :data

.paravirtprobe : AT(ADDR(.paravirtprobe) - LOAD_OFFSET) {
  __start_paravirtprobe = .;
  *(.paravirtprobe)
  __stop_paravirtprobe = .;
}

. = ALIGN(4096);
.data_nosave : AT(ADDR(.data_nosave) - LOAD_OFFSET) {
  __nosave_begin = .;
  *(.data.nosave)
  . = ALIGN(4096);
  __nosave_end = .;
}
```

```
  . = ALIGN(4096);
  .data.page_aligned : AT(ADDR(.data.page_aligned) - LOAD_OFFSET) {
    *(.data.idt)
  }

  . = ALIGN(32);
  .data.cacheline_aligned    :    AT(ADDR(.data.cacheline_aligned)    -
LOAD_OFFSET) {
    *(.data.cacheline_aligned)
  }

  /* rarely changed data like cpu maps */
  . = ALIGN(32);
  .data.read_mostly : AT(ADDR(.data.read_mostly) - LOAD_OFFSET) {
    *(.data.read_mostly)
    _edata = .;       /* End of data section */
  }

  . = ALIGN(THREAD_SIZE);   /* init_task */
  .data.init_task : AT(ADDR(.data.init_task) - LOAD_OFFSET) {
    *(.data.init_task)
  }

  /* might get freed after init */
  . = ALIGN(4096);
  .smp_altinstructions : AT(ADDR(.smp_altinstructions) - LOAD_OFFSET) {
    __smp_alt_begin = .;
    __smp_alt_instructions = .;
    *(.smp_altinstructions)
```

```
    __smp_alt_instructions_end = .;
  }
  . = ALIGN(4);
  .smp_locks : AT(ADDR(.smp_locks) - LOAD_OFFSET) {
    __smp_locks = .;
    *(.smp_locks)
    __smp_locks_end = .;
  }
  .smp_altinstr_replacement  :  AT(ADDR(.smp_altinstr_replacement)  -
LOAD_OFFSET) {
    *(.smp_altinstr_replacement)
    __smp_alt_end = .;
  }
  /* will be freed after init
   * Following ALIGN() is required to make sure no other data falls on
the
   * same page where __smp_alt_end is pointing as that page might be freed
   * after boot. Always make sure that ALIGN() directive is present after
   * the section which contains __smp_alt_end.
   */
  . = ALIGN(4096);

  /* will be freed after init */
  . = ALIGN(4096);       /* Init code and data */
  .init.text : AT(ADDR(.init.text) - LOAD_OFFSET) {
    __init_begin = .;
    _sinittext = .;
    *(.init.text)
    _einittext = .;
```

```
  }
  .init.data : AT(ADDR(.init.data) - LOAD_OFFSET) { *(.init.data) }
  . = ALIGN(16);
  .init.setup : AT(ADDR(.init.setup) - LOAD_OFFSET) {
    __setup_start = .;
    *(.init.setup)
    __setup_end = .;
   }
  .initcall.init : AT(ADDR(.initcall.init) - LOAD_OFFSET) {
    __initcall_start = .;
    INITCALLS
    __initcall_end = .;
  }
  .con_initcall.init : AT(ADDR(.con_initcall.init) - LOAD_OFFSET) {
    __con_initcall_start = .;
    *(.con_initcall.init)
    __con_initcall_end = .;
  }
  SECURITY_INIT
  . = ALIGN(4);
  .altinstructions : AT(ADDR(.altinstructions) - LOAD_OFFSET) {
    __alt_instructions = .;
    *(.altinstructions)
    __alt_instructions_end = .;
  }
  .altinstr_replacement : AT(ADDR(.altinstr_replacement) - LOAD_OFFSET)
{
    *(.altinstr_replacement)
  }
```

```
  . = ALIGN(4);
  .parainstructions : AT(ADDR(.parainstructions) - LOAD_OFFSET) {
    __start_parainstructions = .;
    *(.parainstructions)
    __stop_parainstructions = .;
  }
  /* .exit.text is discard at runtime, not link time, to deal with
references
     from .altinstructions and .eh_frame */
  .exit.text : AT(ADDR(.exit.text) - LOAD_OFFSET) { *(.exit.text) }
  .exit.data : AT(ADDR(.exit.data) - LOAD_OFFSET) { *(.exit.data) }
  . = ALIGN(4096);
  .init.ramfs : AT(ADDR(.init.ramfs) - LOAD_OFFSET) {
    __initramfs_start = .;
    *(.init.ramfs)
    __initramfs_end = .;
  }
  . = ALIGN(L1_CACHE_BYTES);
  .data.percpu  : AT(ADDR(.data.percpu) - LOAD_OFFSET) {
    __per_cpu_start = .;
    *(.data.percpu)
    __per_cpu_end = .;
  }
  . = ALIGN(4096);
  /* freed after init ends here */

  .bss : AT(ADDR(.bss) - LOAD_OFFSET) {
    __init_end = .;
    __bss_start = .;     /* BSS */
```

```
    *(.bss.page_aligned)
    *(.bss)
    . = ALIGN(4);
    __bss_stop = .;
    _end = . ;
    /* This is where the kernel creates the early boot page tables */
    . = ALIGN(4096);
    pg0 = . ;
  }

  /* Sections to be discarded */
  /DISCARD/ : {
    *(.exitcall.exit)
    }

  STABS_DEBUG

  DWARF_DEBUG

  NOTES
}
```

2.6.X 内核引入了更多新特性，所以连链接器脚本的花头更多了。你要是看不懂有些行的具体含义，这很正常，因为这些知识超出了链接器脚本的范围，需要你了解 Linux 内核的知识。很遗憾，我也不是什么 Linux 内核高手，也无法说明脚本中的每一行的确切含义，只能了解各大概。一起看内核源代码去吧！^_^

某嵌入式系统链接器脚本解析

```
yuta2(fgu)% cat Linkcmd_core
```

```
PHDRS
{
        headers     PT_PHDR PHDRS;
        RAM         PT_LOAD;
        DATA        PT_LOAD;
        BSS         PT_NULL;
}
```

在《链接器脚本》一文中的“PHDRS 命令”一节告诉我们，PHDRS 命令只对 ELF 格式文件有效，显然该链接器脚本控制生成的输出文件还不是最终运行的形式，还需要后续处理，因为对没有运行时系统支持的嵌入式系统而言，ELF 格式对它是完全没有意义的，它只要一个可运行的二进制代码块就行。

在 PHDRS 命令中指定段(segment)。上面制定了 4 种。

headers，RAM，DATA 和 BSS，各个类型(type)在《链接器脚本》一文中的“PHDRS 命令”一节有详细介绍。

```
SECTIONS
{
        . = SIZEOF_HEADERS;

        .text       0x00010000 :       从 64K 开始
        {
                wrs_kernel_text_start = .; _wrs_kernel_text_start = .;
```

定义了两个指向 64K 地址的符号，wrs 是 Vxworks 的简称吗？我不知道，或许是吧。该系统的操作系统用的是 Vxworks。

这个脚本把真正的代码(code)与只读数据(rodata)都归为“代码”，反正都不能被写吧。

```
                core_*.o(.text)
                core_*.o(.text.*)
```

```
            core_*.o(.stub)
            core_*.o(.gnu.warning)
            core_*.o(.gnu.linkonce.t*)
            core_*.o(.rodata)
            core_*.o(.rodata.*)
            core_*.o(.gnu.linkonce.r*)
            core_*.o(.rodata1)
            core_*.o(.sdata2)
            core_*.o(.sbss2)
            romcontenttable_*.o(.text)
            romcontenttable_*.o(.rodata)
```

定义 4 个符号来标识“代码”段的结束

```
            wrs_kernel_text_end = .; _wrs_kernel_text_end = .;
            etext = .; _etext = .;
      } : RAM     上面的节将有“RAM”的属性

      代码段之后紧接着是 0x10000 空隙，然后就是数据段
      ADDR(.text)返回上面代码输出节的地址
      SIZEOF(.text)是整个代码输出节的大小
      0x10000 是空隙
      .data       ( ADDR(.text) + SIZEOF(.text) + 0x10000 ) :
      {
            wrs_kernel_data_start = .; _wrs_kernel_data_start = .;
            core_*.o(.data)
            core_*.o(.data.*)
            core_*.o(.gnu.linkonce.d*)
            core_*.o(.data1)
```

```
        core_*.o(.eh_frame)
        core_*.o(.gcc_except_table)
        core_*.o(.got.plt)
        core_*.o(.got)
        core_*.o(.dynamic)
        core_*.o(.got2)
        core_*.o(.dtors)
        core_*.o(.ctors)
        core_*.o(.sdata)
        core_*.o(.sdata.*)
        core_*.o(.lit8)
        core_*.o(.lit4)
        edata = .; _edata = .;
        wrs_kernel_data_end = .; _wrs_kernel_data_end = .;
    } : DATA
```

在数据段之后也是 0x10000 的空隙，接着是 BSS 段

```
    .bss ( ADDR(.data) + SIZEOF(.data) + 0x10000 ) :
    {
        core_*.o(.sbss)
        core_*.o(.scommon)
        core_*.o(.dynbss)
        core_*.o(.bss)
        core_*.o(.gnu.linkonce.b*)
        core_*.o(COMMON)
        end = .; _end = .;
        wrs_kernel_bss_end = .; _wrs_kernel_bss_end = .;
    } : BSS
```

```
}
```

译者

Walter Zhou

z-l-dragon@hotmail.com

2007-12-20，神奈川县海老名市国分北，第一稿

2007-12-25，上海浦东周浦，二稿