GNU LD(Linker)

中文翻译

version 2.30

2018.10.31

翻译：崔良金

微信号：leonkim999

邮箱：a2282@126.com

原文地址：https://sourceware.org/binutils/docs/ld/index.html

此翻译仅供参考，会包含一些错误，请谅解，也欢迎指正。

目录

1 概述

LD(GNU Linker)将一些 object 文件和 archive 文件（可以理解为多个 object 文件组合出来的文档文件）打包，重新布置这些文件中的数据，整理里面引用的符号。一般情况下，编译程序最后一步就是运行 LD。

LD 接受链接器命令语言文件来清晰完整地控制整个链接过程，这个文件使用的语言语法包含 AT&T 的链接编辑命令语言语法。

此版本的 LD 使用一般用途的 BFD 库来操作 object 文件。这允许 ld 以多种不同的格式读取、组合以及写入 object 文件。例如，COFF 或者 a.out。为了产生各种有效类型的 object 文件，可以将各种不同格式的文件链接到一起。

除了它的灵活性，GNU 链接器比其它链接器还能提供最能帮助诊断的信息。很多其它链接器遇到一个错误后会立即放弃执行；只要可能，LD 会继续执行，允许你识别出其它错误（或者，有些情况下，尽管有错误发生，仍然可以输出目标文件）。

2 调用

GNU 链接器 LD 意图处理非常复杂而广泛的情况，而且希望尽可能与其它链接器兼容。这样的结果就是，你可以有很多种选择来控制它的行为。

2.1命令行选项

链接器支持非常多的命令行选项，可事实上只有很少的一部分在实际应用中被用到。例如，LD 频繁用于在标准的支持 Unix 的系统中链接标准 Unix object 文件。在这种系统中，要链接 hello.o 文件：

ld -o *output* /lib/crt0.o hello.o -lc

这条命令告诉 LD 产生一个名叫 output 的文件，output 文件是由/lib/crt0.o、hello.o，以及 libc.a 链接而成，libc.a 在标准搜索目录中（可以参考下面的’ -l’选项的说明）。

给 ld 的一些命令行选项可能会放置在命令行中的任何位置。然而，指向文件的那些选项，例如’ -l’，’ -T’等，会在此选项出现的位置触发文件的读操作，涉及 object 文件和其它文件选项。用不同的参数重复那些非文件的选项没有任何影响，也不会覆盖命令行左侧出现过的选项参数。对于多次指定同一个参数却有意义的情况，我们下面介绍。

无选项的参数（没有‘-’开头）是 object 文件或者 archive 文件，这些文件是用来链接到一起的。这些可能在其它命令行选项的前边、中间或者后边，但是，它们不能放在选项和这个选项所对应的参数之间。

通常调用链接器时，至少要指定一个 object 文件，但是你可以使用’ -l’或’ -R’指定其它格式的二进制输入文件，以及指定脚本命令语言的文件。如果根本没有指定二进制的文件，链接器就不会产生输出文件，并且会提示‘No input files’的消息。

如果链接器无法识别一个 object 文件的格式，它会假定这个文件是链接脚本。以这种

方式指定的链接脚本会添加在链接时使用的主链接脚本后（主脚本或许是默认的链接脚本，又或者是通过’-T’用户指定的那个脚本）。这个特性允许链接器处理这样一些这样的文件，这些文件从命令行看起来是 object 文件或者是 archive 文件但却只是定义了一些符号的值，或者使用 INPUT、GROUP 加载其它 object 文件。通过这种方式指定的脚本只是主链接脚本的补充，这些脚本文件中的附加命令会放置在主脚本文件之后；使用’-T’选项才能完全的替换默认链接脚本，但需要注意 INSERT 命令的影响。可参考后边的 链接器脚本 章节。

对于一个字母的那些选项，参数必须要紧跟其后（可以有空格，也可以没有空格）。

对于多个字母的那些选项 ,前边一个横线和两个横线是一样的 ;例如 ,’-trace-symbol’和‘--trace-symbol’是等效的。注意，有一个例外：以小写的’o’开头的多字母选项前必须使用双横线。这是为了区分’-o’选项的。所以，例如’-omagic’是指设置输出文件名称为 magic，而’--omagic’是指在输出时设置 NMAGIC 标志。

多字母的选项要么通过=连接参数，要么通过空格连接紧跟它的参数。例如：‘--trace-symbol foo’ 和 ‘--trace-symbol=foo’是等效的。多字母选项名字的缩写若是唯一的，也是被接受的。

注意：如果链接器是被间接调用的，例如通过编译器调用时(gcc)，所有链接命令行选项都要带’-Wl’前缀，（或者其它的适合特定编译器的前缀），就像下面的例子：

```
gcc -Wl,--start-group foo.o bar.o -Wl,--end-group
```

这是很重要的，否则编译器程序就会默默地丢掉这些链接选项，然后导致错误的链接。由于选项与参数之间使用空格作为分隔符，让编译器传递带值的链接命令仍然会造成混乱，因为编译器会只把选项传递给链接器，而参数传递给编译器。这种情况下，最简单的方式就是使用单字母和多字母选项的连接格式（不用空格的格式），就像：

```
gcc foo.o bar.o -Wl,-eENTRY -Wl,-Map=a.map
```

以下是 GNU 链接器能够识别的命令行选项（开关）表：

@*file*

从 file 指定的文件中读取命令行选项。读出的选项被插入到@file 所在的位置。如果文件不存在或者不能读取，此选项也不会被移除，只是无效而已。

在 file 中的选项以空格分隔。在一个选项中若含有空格，必须要将整个选项包含在单引号或者双引号之间。任何字符（包括’\’）都能被包含进来，当然有些字符自被加上’\’的前缀。File 指定的文件中还可以包含另外的@file 选项；这些选项将会被递归式的处理。

-a *keyword*

这一选项用来支持对 HP/UX 的兼容。Keyword 参数必须是’archive’、’shared’、或者’default’之一。’-aarchive’从功能上等同于’-Bstatic’,另外的两个 keywords 从功能上等同于’-Bdynamic’.这个选项可以被多次用到。

--audit *AUDITLIB*

添加 AUDITLIB 到动态分区的 DT_AUDIT 条目。不用检查 AUDITLIB 是否存在，也不会使用在库中指定的 DT_SONAME。如果多次指定，DT_AUDIT 会包含要使用的

以冒号分隔的审计接口列表。如果链接器在搜索共享库时，发现一个 object 带有审计（audit）入口，它将在输出文件中添加一个合适的 DT_DEPAUDIT 条目。这个选项只在支持 rtld-audit 接口的 ELF 平台时有意义。

-A *architecture*

--architecture=*architecture*

在当前的 ld 发布版本中，这个选项只在 Intel 960 架构族时有用。Architecture 参数指定 960 家族中具体的架构名称，启用一起保护措施，修改 archive 库搜索路径。具体信息，可参考 ld and the Intel 960 family.

将来 ld 的发布版本中，可能会以类似的功能，支持其它的架构族。

-b *input-format*

--format=*input-format*

Ld 可以配置得支持一种以上 object 文件。如果你的 LD 这样配置了，你可以使用’-b’选项来指定紧跟其后的输入文件是二进制格式的。即使当 ld 需要支持另一种 object 格式时，你也通常不用指定，因为 ld 会被配置成在每种机器上将输入文件当作其默认的格式来处理。Input-format 是个字符串，是 BFD 库支持的某种特定的格式名。（你可以使用 objdump –i 来查看有效的二进制格式列表）。

你在链接一些不寻常的二进制格式文件时，往往希望使用这个选项。也可以通过在一组特别格式的 object 文件前面放置‘-b input-format’显式的使用’-b’来做格式的开关（当然是链接不同格式的 object 文件时）。

默认的格式是由环境变量 GNUTARGET 来定义的。可以参考环境 。你也可以使用在脚本文件中使用 TARGET 命令定义输入格式；参考格式命令。

-c *MRI-commandfile*

--mri-script=*MRI-commandfile*

为兼容 MRI 链接器，ld 允许脚本文件中交替出现一般格式和 MRI 格式的命令，但受一定限制。可参考 MRI 兼容脚本文件。使用’-c’选项引入 MRI 脚本文件；使用’-T’选项运行一般链接脚本。如果 MRI-cmdfile 不存在，ld 会从’-L’选项指定的目录中寻找。

-d

-dc

-dp

这三个选项是等效的；同时存在是为了与其它链接器兼容。即使指定了可重定位的输出文件（用’-r’选项），链接器也会为一般的符号分配空间。脚本命令 FORCE_COMMON_ALLOCATION 有同样的效果。参考杂项命令。

--depaudit *AUDITLIB*

-P *AUDITLIB*

添加 AUDITLIB 到动态分区的 DT_DEPAUDIT 条目。不用检查 AUDITLIB 是否存在，也不会使用在库中指定的 DT_SONAME。如果多次指定，DT_DEPAUDIT 会包含要使用的以冒号分隔的审计接口列表。这个选项只在支持 rtld-audit 接口的 ELF 平台

时有意义。-P 选项是用来兼容 Solaris 的。

-e *entry*

--entry=*entry*

使用 entry 来明显地指定你的应用程序的入口位置，要好过默认的入口点。如果链接器未找到名称为 entry 的符号，它会尝试将 entry 解析为一个数字，将这个数字作为入口地址（这个数字默认是 10 进制的，可以用 0x 作为开头表示 16 进制，以 0 开头表示 8 进制）。参考 入口点 ，会介绍默认的入口点，以及几种不同的指定入口点的方式。

--exclude-libs *lib*,*lib*,...

指定一个 archive 库列表，列出的库中的符号不会被自动导出。多个库名之间用逗号或冒号分隔。使用--exclude-libs ALL ,所有 archive 库中的符号都不会自动导出。这个选项只对以 i386 PE 格式和 ELF 格式为目标的链接器接口有效。对于 i386 PE，如果符号明确地列在.def 文件中，仍会忽略这个选项而被导出。对于 ELF，被此选项影响的符号将对外不可见。

--exclude-modules-for-implib *module*,*module*,...

指定模块列表（由 object 文件和 archive 成员组成），其中的符号不会被自动导出，但在链接时，列表中的模块会被批量复制到引入库中。多个模块名之间用逗号或冒号分隔，而且必须和 ld 使用的文件名完全相同。对于 archive 成员，这只是成员名，但对于 object 文件来说，这个名称必须和在命令行中输入的给 ld 使用的 object 文件名完全相同，既包括目录也包括文件名。这个选项只对以 i386 PE 格式和 ELF 格式为目标的链接器接口有效。对于 i386 PE，如果符号明确地列在.def 文件中，仍会忽略这个选项而被导出。

-E

--export-dynamic

--no-export-dynamic

当创建动态链接执行文件时，使用-E 选项或者 --export-dynamic 选项，链接器会将所有符号添加到动态符号表中。动态符号表包含在运行时对动态对象可见的符号。如果你不使用此选项，（或者使用--no-export-dynamic 选项来恢复链接器默认的行为），动态符号表中一般就只会包括那些在链接时被动态对象所使用到的符号。如果某个程序中使用 dlopen 加载一个动态对象，而这个对象又需要这个程序中的定义的符号，而不是使用另外的动态对象中的，那么你很可能需要在链接这个程序时使用此选项。

如果输出格式支持的话，你也可以使用动态列表来控制哪些符号应该添加到动态符号表。请参考 ‘—dynamic-list’ 的描述。

注意，这个选项是针对 ELF 目标接口的。PE 目标支持另外一个类似的功能来完成从 DLL 或者 EXE 中导出所有的符号；参考 ‘—export-all-symbols’ 的描述。

-EB

链接大端对象。这会影响默认的输出格式。

-EL

链接小端对象。这会影响默认的输出格式。

-f *name*

--auxiliary=*name*

当创建一个 ELF 共享对象时，给其内部 DT_AUXILIARY 域设置一个特定的名字。这就是告诉动态链接器，共享对象的符号表应该用作共享对象 name 的符号表的辅助筛选器。

如果你使用这个筛选器对象链接某个程序，那么，当你运行程序时，动态链接器会查看 DT_AUXILIARY 域。如果动态链接器要处理这个筛选器对象中的符号时，首先检查在共享对象 name 中有没有定义。如果有，找到的这个符号将替换掉筛选器对象中的那个。共享对象 name 可以不存在。因此共享对象 name 可以用来提供某些函数的替代实现，或许为了调试方便或者为了符合机器特定的性能等。

这一选项可以指定多次。多个 DT_AUXILIARY 条目会按在命令行中出现的顺序依次创建。

-F *name*

--filter=*name*

当创建一个 ELF 共享对象时，给其内部 DT_FILTER 域设置一个特定的名字。这就是告诉动态链接器，正在创建的共享对象的符号表应该用作共享对象 name 的符号表的筛选器。

如果你使用这个筛选器对象链接某个程序，那么，当你运行程序时，动态链接器会查看 DT_FILTER 域。动态链接器像平常一样依据此筛选器对象中的符号表来处理这些符号，但是事实上，链接器会从共享对象 name 中寻找那些符号的定义。因此筛选器对象可以被用来选择共享对象 name 所提供的符号表的一个子集。

一些旧的链接器为了指定输入输出 object 文件的格式，会在整个编译工具链中使用 -F 选项。GNU 链接器使用其它的办法来达到此目的 :-b,--format,--oformat 选项，在链接脚本中的 TARGET 命令，以及使用 GNUTARGET 环境变量。如果不是为了创建 ELF 共享对象，GNU 链接器会忽略-F 选项。

-fini=*name*

创建 ELF 可执行程序或共享对象时，通过设置 DT_FINI 域的值为 name 函数的地址，在执行时，当卸载可执行程序或共享对象后调用 name 函数。

-g

忽略。这是用来兼容其它工具的。

-G *value*

--gpsize=*value*

设置可以通过 GP 寄存器来优化的对象的最大尺寸。此选项只对类似于 MIPS ELF 的支持将大对象和小对象放在不同分区里的对象文件格式有意义。对其它格式的，此选项会忽略。

-h *name*

-soname=*name*

当创建 ELF 共享对象时，给内部的 DT_SONAME 域设置名称。当可执行程序和这样一个拥有 DT_SONAME 域的共享对象一同被链接后，执行程序运行时，动态链接器会尝试使用 DT_SONAME 指定的名称加载共享对象，而不是使用通过文件名加载。

-i

执行增量链接（与’ -r’ 选项相同）。

-init=*name*

当创建一个 ELF 可执行文件或共享对象时，当可执行文件或共享对象被加载时调用 name 符号，这是通过设置 DT_INIT 为 name 符号的地址实现的。默认情况下，链接器调用_init。

-l *namespec*

--library=*namespec*

添加由 namespec 指定的 archive 文件或 object 文件到链接文件列表中。这个选项可能被用到 N 多次。如果 namespec 是:filename 格式的，ld 会在库目录中寻找 filename 文件，否则，ld 会在库目录中寻找 libnamespec.a 文件。

在支持共享库的系统中，ld 也会查找一些别的文件。在 ELF 和 SunOS 系统中，ld 就会首先在库目录中寻找叫做 libnamespec.so 的文件，然后才找 libnamespec.a。(按惯例，以.so 为扩展名的文件是个共享库。)注意，以:filename 格式指定的不会这样，而是只寻找名为 filename 的文件。

链接器只会在 archive 在命令行中出现的位置搜索此 archive 文件一次。如果在 archive 中定义了一些符号，这些符号在一些 object 文件中使用了但并未定义，而这些 object 文件在命令行中先于 archive 出现的位置，链接器会从 archive 中引入适当的文件以满足对这些符号的使用。然而，如果使用了未定义符号的 object 文件在命令行中比 archive 出现的晚，链接器也不会重新从 archive 中寻找这些符号了。如果想通过一种方式让链接器从 archive 文件中多次搜索，请参考 –(选项。

你可以在命令行中多次指定同一个 archive 文件。

此种 archive 的搜索方式是 Unix 链接器的标准。然而，如果你在 AIX 上使用 ld，要注意 ld 的处理方式和 AIX 链接器是有区别的。

-L *searchdir*

--library-path=*searchdir*

添加 searchdir 到路径列表中 ,这个路径列表用于 ld 搜索 archive 库和 ld 控制脚本。可以使用多次此选项。搜索时按这些目录在命令行中出现的顺序进行。链接器会首先搜索命令行中指定的目录，然后才是默认目录。所有的-L 选项中指定的目录，都能用作-l 选项指定的 object 或 archive 文件的搜索，而不管在命令行中出现的先后次序。除非使用-T 选项指定链接脚本文件，否则-L 选项并不影响 ld 搜索链接脚本文件。

如果 searchdir 以=或者$SYSROOT 开头，这个前缀会被替换成 sysroot 前缀，

sysroot 前缀是由--sysroot 选项控制或者在配置链接器时指定的。

链接器所使用的默认路径（在没有指定-L 选项的情况下）依赖于 ld 正在使用的仿真模式，在某些情况下也要看如何配置的。参考 环境。

搜索路径也可以在链接脚本中用 SEARCH_DIR 命令指定。用这种方式指定的目录会被放置在命令行中脚本文件出现的位置处进行搜索。

-m *emulation*

仿真 emulation 链接器。你可以通过--verbose 或-V 选项查看有效的仿真器列表。如果没有使用-m 选项，如果在环境变量中定义的 LDEMULATION，那么将使用这个变量中指定的仿真器。

否则，链接器的配置将决定默认的仿真器。

-M

--print-map

在标准输出设备上打印出链接符号地址对应图（以下简称链接图）。链接图中提供链接的各种信息，包括以下：

- Object 文件被放置在内存中的位置。
- 一般符号是如何被指派和分配的。
- 在链接时，用到的所有 archive 的成员，以及由于使用哪些符号导致此 archive 成员被链接。
- 给符号所赋的值。

  注意：在表达式计算中需要使用某个符号的值，而表达式计算的结果再赋值给同名称的符号，那么在链接图中此符号显示的值有可能是错误的。这是因为，链接器会抛弃中间结果，而只保持表达式的最终结果。这种情况下，链接器会在显示时将最终值用方括号括起来。以下是个例子，链接器脚本如果包括以下脚本：

  ```
  foo = 1
  foo = foo * 4
  foo = foo + 8
  ```

  将会在链接图中输出以下内容（如果用-M 选项来显示）：

  ```
  0x00000001                foo = 0x1
  [0x0000000c]                foo = (foo * 0x4)
  [0x0000000c]                foo = (foo + 0x8)
  ```

  参考 表达式 ，获得关于在链接脚本中表达式的更多信息。

-n

--nmagic

关闭分区的页对齐功能，禁止对共享库的链接。如果输出格式支持 Unix 风格的 magic number，那么标记输出文件为 NMAGIC。

-N

--omagic

把 text 分区和 data 分区设置为可读可写。数据分区不进行页对齐，禁止对共享库的链接。如果输出格式支持 Unix 风格的 magic number，那么标记输出文件为 OMAGIC。注意：尽管在 PE-COFF 目标中 text 分区是允许写的，但这并不符合 Microsoft 发布的格式规范。

--no-omagic

此选项用来取消-N 选项的大部分作用。它会设置 text 分区为只读，强制 data 分区是页对齐的。注意：此选项不会启用对共享库的链接。若需要，使用-Bdynamic 选项。

-o *output*

--output=*output*

Ld 链接所产生的程序以 output 为名称。如果未指定此选项，默认输出为 a.out。脚本命令 OUTPUT 也可用来指定输出文件名。

-O *level*

如果 level 是个大于 0 的数据，ld 会优化输出。带有此选项会明显加长链接所需的时间，因此最好只在生成最终版二进制文件时启用。目前，此选项只会影响 ELF 共享库的生成。链接器将来的版本可能会用得更多一些。而且目前来说，此选项设置不同的非零值不会影响链接器的行为。当然，将来这一特性可能也会改观。

-plugin *name*

在链接的过程中引入一个插件。Name 参数就是插件的绝对文件地址。通常这个参数会在使用时间优化时被编译器自动添加。当然，用户也可以按意愿添加自己的插件。

注意：编译器与其插件的相对位置与 ar、nm、ranlib 等程序搜索它们插件的位置是不同的。要想使用这些命令中的基于编译器的插件，首先需要将插件复制到 ${libdir}/bfd-plugins 目录中。所有基于 gcc 的链接器插件都是向后兼容的，因此只复制最新版本的插件就 OK 了。

--push-state

可以使用--push-state 选项将当前控制输入文件的一些参数保存起来，而在合适的时候，使用--pop-state 选项将状态恢复。

包括的选项有：

-Bdynamic, -Bstatic, -dn, -dy, -call_shared, -non_shared, -static, -N, -n, --whole-archive, --no-whole-archive, -r, -Ur, --copy-dt-needed-entries, --no-copy-dt-needed-entries, --as-needed, --no-as-needed, 以及 -a.

由此选项所生成的一个目标正好为 pkg-config 工具提供了详细说明。当和--libs 选项一起使用时，所有被列出的库可能会一直被链接，所以，最好参考以下命令恢复原始状态：

-Wl,--push-state,--as-needed -libone -libtwo -Wl,--pop-state

--pop-state

--push-state 的逆操作，恢复此前输入文件参数的值。

-q

--emit-relocs

在生成的可执行文件中，保留重定位的分区和内容的完整信息。以后链接分析和优化工具要想正确执行，会需要这些信息。当然这会使可执行文件的尺寸大一些。

这一选项目前只支持 ELF 平台。

--force-dynamic

强制要求输出文件中包含动态分区。此选项专为 VxWorks 目标设置。

-r

--relocatable

产生可重定位的输出文件。例如，产生一个输出文件，可以反过来作为 ld 的输入文件进行链接。这常被称作部分链接。副作用就是，在支持标准 Unix magic number 的环境里，此选项也会设置输出文件的 magic number 为 OMAGIC。如果未指定此选项，会产生一个使用绝对地址的文件。在链接 C++程序时，这一选项不会解决关于构造函数的引用，若需要，可以使用‘-Ur’。

当输入文件和输出文件的格式不一致时，如果输入文件中不包括任何可重定位的部分，那么只支持部分链接。不同的输出格式会有更多的限制；例如，一些基于 a.out 的格式根本不支持对其它格式的输入文件进行链接。

此选项和’-i’干同样的事。

-R *filename*

--just-symbols=*filename*

从 filename 中读取符号名和地址，但并不在输出文件中分配和包含这些符号。这个特性允许你的输出文件以符号形式引用在其它程序中定义和分配地址的存储器。你可以多次使用此选项。

为兼容其它的 ELF 链接器，如果-R 选项后面跟着一个目录名，而不是一个文件名的话，会被看作是-rpath 选项。

-s

--strip-all

不保留输出文件中的所有符号信息。

-S

--strip-debug

从输出文件中去除关于调试器的符号信息（并不是所有符号信息都不保留）。

--strip-discarded

--no-strip-discarded

去除（或者保留）在被废弃的分区中定义的全局符号。默认是使能的。

-t

--trace

在 ld 处理输入文件时，打印出这些文件的名称。

-T *scriptfile*

--script=*scriptfile*

使用 scriptfile 作为链接脚本。此脚本会替换掉 Ld 的默认脚本（而不是添加在默认脚本之后），所以 commandfile 必须要指定所有必需的内容来描述输出文件。参考 链接器脚本。如果在当前目录下 scriptfile 不存在，ld 会在-L 指定的所有目录中寻找。多个‘-T’选项指定的脚本文件会连接在一起处理。

-dT *scriptfile*

--default-script=*scriptfile*

使用 scriptfile 作为默认的链接脚本。参考 链接器脚本。

此选项与--script 选项类似，只是在处理脚本文件时有些差别，默认的脚本文件会在处理完命令行中所有内容后才执行。这一特性允许放在--default-script 选项之后的那些命令行选项影响链接脚本中的行为，这对于链接命令行不能完全被用户直接控制的情况尤其重要。（例如，部分命令行被另一个就像 gcc 类的工具生成）。

-u *symbol*

--undefined=*symbol*

强制让 symbol 在输出文件中作为未定义的符号。这样做可能会触发链接标准库中的一些附加模块。’ -u ‘可以多次使用不同的参数添加未定义的符号。这一选项等价于 EXTERN 链接器脚本命令。

如果此选项用来使一些另外的模块被链接，而且由于该符号未定义引起了错误，那么应该使用--require-defined 选项替换之。

--require-defined=*symbol*

需要 symbol 在输出文件中定义。此选项几乎和--undefined 选项一模一样，但是当 symbol 在输出文件中未定义，而链接器会报告一个错误并且退出链接过程。这种效果可以在链接器脚本中一起使用 EXTERN, ASSERT 和 DEFINED 来实现。此选项可以多次使用，以要求另外的符号参与链接。

-Ur

除了 C++程序之外，此选项等效于’ -r ‘：它会产生可重定位的输出文件，也就是说 输出文件可以反过来作为 ld 的输入文件。当链接 C++程序时，’ -Ur ‘不像’ -r’ 选项，它解析对构造函数的引用。以’ -Ur ‘选项产生的文件，如果再用’ -Ur ‘选项处理，不会工作的；一旦构造函数表创建后，’ -Ur’ 就不能再次使用了。只在局部链接的最后使用’ -Ur’ 选项，其它时候应使用’ -r ‘。

--orphan-handling=*MODE*

控制如何处理孤立分区。孤立分区就是指在链接器脚本中未提及的分区。参考 孤立分区。

MODE 可能是如下几个值：

place

孤立分区会按 孤立分区中描述的策略被放置在一个合适的输出分区中。

‘--unique’ 选项也会影响这些分区的放置。

discard

所有孤立分区都会被丢弃，将这些分区放置在’/DISCARD/‘分区中。请参考 丢弃的输出分区

warn

链接器会像 place 一样放置孤立分区，但会抛出一个警告。

error

如果发现孤立分区，链接器会携带一个错误退出。

如果未被指定，‘--orphan-handling’的默认值是 place.

--unique[=*SECTION*]

为每个匹配 SECTION 的输入分区创建一个单独的输出分区，或者如果可选的 SECTION 通配符未指定，为每个孤立输入分区创建单独的输出分区。孤立分区是指在链接器脚本中未提及的分区。你可以在命令行中多次使用此选项；它会阻止一般情况下同样名字的输入分区合并，以及在链接器脚本中对输出分区分配时的覆盖。

-v

--version

-V

显示 ld 的版本号。-V 选项也会列出支持的仿真器。

-x

--discard-all

删除所有局部符号。

-X

--discard-locals

删除所有临时局部符号。（这些符号以系统指定的局部标签前缀开始，典型的,ELF 系统是’.L’，而传统的 a.out 系统是以’L’开头。）

-y *symbol*

--trace-symbol=*symbol*

打印出 symbol 出现过的每个被链接的文件名称。此选项可以使用任何次数。在很多系统中，要想预先考虑如何标记这些 symbol 是必须的。

此选项在链接提示未定义符号但却不知道在哪里引用时特别有用。

-Y *path*

添加 path 到默认的库搜索路径中。此选项用来兼容 Solaris。

-z *keyword*

能识别的关键字有：

‘bndplt’

总在 PLT 条目中生成 BND 前缀。用于支持 Linux/x86_64。

‘call-nop=prefix-addr’

‘call-nop=suffix-nop’

‘call-nop=prefix-*byte*’

‘call-nop=suffix-*byte*’

当通过 GOT 槽把间接函数调用转换为局部函数时(例如函数 foo)，指定单字 NOP 作为填充。call-nop=prefix-addr 生成“0x67+foo 调用”. call-nop=suffix-nop 生成“foo 调用+0x90”. call-nop=prefix-byte 生成 “byte +foo 调用.” call-nop=suffix-byte 生成 “foo 调用+ byte”. 支持 i386 和 x86_64。

‘combreloc’

‘nocombreloc’

组合多个动态重定位分区，并重新排序，以提高动态符号查找缓存效率。而定义’ nocombreloc ‘时，不会这样做。

‘common’

‘nocommon’

在进行可重定位的链接时对一般符号采用 STT_COMMON 类型。’ nocommon’ 时使用 STT_OBJECT 类型。

‘common-page-size=*value*’

使用 value 值设置最常用的页大小。若系统使用这一页大小，链接器会在布局存储器镜像时优化到最小存储器页。

‘defs’

列出在常规 object 文件中引用但却未找到的符号。即使链接器正在创建非符号共享库。此选项是 ‘-z undefs’ 的逆操作。

‘dynamic-undefined-weak’

‘nodynamic-undefined-weak’

在创建动态对象时，如果一些未定义的弱符号被常规 object 文件引用，而且并未被符号的可见性或版本强制局部化，那就定义为动态符号。若使用 ‘nodynamic-undefined-weak’ 选项，则不会使这些符号动态化。如果两个选项都未指定，则目标可能默认为任一选项生效，或者使其他未定义的弱符号动态化。不是所有的目标都支持这些选项。

‘execstack’

将 object 标计为需要可执行栈。

‘global’

此选项只在创建共享对象时有意义。它会让这个共享对象定义的符号在将来被加载并对符号进行解析时有效。

‘globalaudit’

此选项只在创建一个动态可执行程序时有效。此选项设置 DT_FLAGS_1 动态标计中的 DF_1_GLOBAUDIT 位来标计可执行程序需要全局的审计。全局审计要求任何被 --depaudit 或-P 命令行选项定义的审计库在应用程序加载所有动态对象时必须运行。

‘ibtplt’

生成 Intel 间接分支跟踪(IBT)启用 PLT 条目。支持 Linux/i386 和 Linux/x86_64.

‘ibt’

在.note.gnu.property 分区中生成 GNU_PROPERTY_X86_FEATURE_1_IBT，用来指示兼容 IBT。这也意味着 ibtplt。支持 Linux/i386 和 Linux/x86_64.

‘initfirst’

此选项只在创建共享对象时有意义。这会标记此对象，在程序加载对象时，此对象的初始化会先于在同一时间加载的其它对象的初始化而首先执行。类似的，此对象的终止化操作会晚于其它对象。

‘interpose’

指示动态加载器要修改符号搜索顺序，使得在此共享库中的符号优先于没有如此标计的共享库。

‘lazy’

当生成可执行程序或共享库时，做标计告诉动态链接器推迟函数的解析时间到真正调用时（又叫懒捆绑模式），而不是加载时。懒捆绑模式是默认的。

‘loadfltr’

指定对象的过滤器在运行时立即执行。

‘max-page-size=*value*’

设置支持的最大存储页尺寸为 value.

‘muldefs’

允许多重定义。

‘nocopyreloc’

阻止链接器生成用于代替共享库中定义变量的.dynbss 变量。可能会导致动态代码的重定位。

‘nodefaultlib’

指定动态加载器在搜索本对象的依赖时，忽略所有默认库搜索路径。

‘nodelete’

指定对象在运行时不要被卸载。

‘nodlopen’

指定对象对 dlopen 是无效的。

‘nodump’

指定对象不能被 dldump 导出（数据整体打印）。

‘noexecstack’

标计该对象不需要可执行栈。

‘noextern-protected-data’

在创建共享库时，不要把被保护的数据符号当作对外的符号。此选项会覆盖链接器的默认值。这可以用来排除对由编译器生成的被保护数据符号的错误重定位。由其它模块更新被保护的数据符号进行的更改操作对其更改的共享库来说是不可见的。为支持 i386 和 x86-64。

‘noreloc-overflow’

禁能重定位越界检查。如果在运行时没有动态重定位溢出，可以将检查禁止。为支

持 i386 和 x86-64。

‘now’

当生成可执行程序或共享库时，做好标计用来告诉动态链接器在程序一开始就解析所有的符号，或者当共享库一被 dlopen 加载就解析所有的符号，而不是推迟到函数的首次调用。

‘origin’

指定此对象需要在目录中有’ $ORIGIN’ 的处理。

‘relro’

‘norelro’

在对象中创建 ELF PT_GNU_RELRO 段头。如果被支持的话，指定在重定位后会将一个存储段改为只读状态。若指定 ‘common-page-size’ 小于系统页大小将使这种保护措施失效。当’ norelro’ 时，不创建 ELF PT_GNU_RELRO 段。

‘separate-code’

‘noseparate-code’

在对象中创建单独的 PT_LOAD 代码段头。指定一个存储段只包含指令，而且与其它数据不能有任何页交叉。当 ‘noseparate-code’ 时，不创建 PT_LOAD 段。

‘shstk’

在.note.gnu.property 分区生成 GNU_PROPERTY_X86_FEATURE_1_SHSTK，兼容 Intel 影子栈。用于支持 Linux/i386 和 Linux/x86_64。

‘stack-size=*value*’

为 ELF PT_GNU_STACK 段指定栈大小。指定零将使所有非零大小的 PT_GNU_STACK 段覆盖。

‘text’

‘notext’

‘textoff’

如果 DT_TEXTREL 被设置，则报告一个错误，也就是说，如果二进制文件在只读分区里有个动态重定位的部分。如果’ notext’ 或’ textoff’ 将不会报告错误。

‘undefs’

在创建可执行程序或者共享库时，不要报告在常规 object 文件中引用却不能解析的符号。此选项是’ -z defs’ 的逆操作。

为了兼容 Solaris，其它关键字会被忽略。

-(*archives* -)

--start-group *archives* --end-group

Archives 参数应该是一个 archive 文件列表。要么是明确的文件名称，要么是-l 选项。

指定的这些 archive 会在没有新的未定义符号之前一直循环搜索。一般而言，archive 只在命令行中指定它的位置上被搜索一次。而如果 archive 需要为在命令行中它出现的位置之后出现的 object 文件中的未定义符号提供对象，就可以通过将这些

archive 分组 ,那么这些 archive 就会在被重复的搜索 ,直到所有可能的引用被解析。此选项还是相当浪费性能的。这种办法最好只在两个或多个 archive 之间有无法避免的循环引用时使用。

--accept-unknown-input-arch

--no-accept-unknown-input-arch

告诉链接器接受不认识的架构的输入文件。这就是假设用户知道自己在做什么，有意地链接这些不认识的输入文件。在 2.14 发布版本之前，这是链接器的默认行为。从 2.14 版本开始，默认行为变成了拒绝链接这些不认识的输入文件，若想继续保持之前的做法，请使用 '--accept-unknown-input-arch' 选项。

--as-needed

--no-as-needed

此选项会影响在命令行中后面出现的动态库中的 ELF DT_NEEDED 标记。一般情况下，无论动态库实际上是否真的被需要，链接器都会为命令行中提及的动态库添加一个 DT_NEEDED 标签。--as-needed 会使 DT_NEEDED 标记只被放置在以下动态库的相应位置：1.被常规 object 文件引用的非弱未定义符号，在此库中满足此符号；2.被另外所需库引用的非弱未定义符号，在此库中解决此符号（而在另外库的 DT_NEEDED 列表中没列出此库）。命令行中，在此问题的库之后出现的 Object 文件或者库文件都不会影响此库是否被看作需要。这和从 archive 中提取 object 文件很相似。--no-as-needed 恢复链接器的默认行为。

--add-needed

--no-add-needed

此两选项已经被废弃 ,这是因为名称与--as-needed/--no-as-needed 选项太相近。它们替换成了--copy-dt-needed-entries 和--no-copy-dt-needed-entries.

-assert *keyword*

此选项会被忽略，只是为了和 SumOS 的兼容而保留的。

-Bdynamic

-dy

-call_shared

对动态库进行链接。这只在支持共享库的平台下有意义。这一选项通常在这些平台下是默认的。此选项的这三种写法是为了兼容多种系统。可以多次在命令行中使用此选项：它会影响其后-l 选项指定库文件的搜索。

-Bgroup

设置动态分区的 DT_FLAGS_1 条目中的 DF_1_GROUP 标记。此选项会使运行时链接器控制此 object 以及它的依赖文件只能在组内执行。

--unresolved-symbols=report-all 被隐含定义。此选项只在支持共享库的 ELF 平台下有意义。

-Bstatic

-dn

-non_shared

-static

不要链接动态库。这只在支持共享库的平台下有意义。此选项的多种写法是为了兼容多种系统。可以多次在命令行中使用此选项：它会影响其后-l 选项指定库文件的搜索。此选项也会让--unresolved-symbols=report-all 被隐含的定义。此选项可以和-shared 一起使用，这样做就意味着，正在创建共享库，此共享库中所有的外部引用必须被静态库中的相应条目解决。

-Bsymbolic

当创建共享库时，将对全局符号的引用全部指向共享库中的定义（只要有）。通常，程序链接一个共享库来覆盖另一个共享库中的定义时使用。在创建一个位置独立的可执行程序时，将对全局符号的引用绑定到这个可执行程序中的定义时，此选项和--export-dynamic 选项一起使用。这一选项只在支持共享库和位置独立的可执行程序的 ELF 平台上有意义。

-Bsymbolic-functions

当创建共享库时，将对全局函数的引用全部指向共享库中的定义（只要有）。在创建一个位置独立的可执行程序时，将对全局函数的引用绑定到这个可执行程序中的定义时，此选项和--export-dynamic 选项一起使用。这一选项只在支持共享库和位置独立的可执行程序的 ELF 平台上有意义。

--dynamic-list=*dynamic-list-file*

为链接器指定一个动态列表文件名。典型的，当创建共享库时，指定一个全局符号列表，而这些符号的引用不该被绑定到这个共享库中的定义。或者创建动态链接的可执行程序时，指定一个全局符号列表，而这些符号需要被添加到此可执行文件的符号表中。此选项只在支持共享库的 ELF 平台上有意义。

动态列表的格式和版本节点是相同的，但没有节点名和作用范围。参考 Version。

--dynamic-list-data

将所有全局数据符号全部包含在动态列表中。

--dynamic-list-cpp-new

为 C++中的 new 和 delete 操作符提供内建的动态列表。主要用来创建共享的 libstdc++。

--dynamic-list-cpp-typeinfo

为 C++的运行时类型识别提供内建的动态列表。

--check-sections

--no-check-sections

请求链接器在完成分区地址的分配后进行检查，以查看这些分区是否会有重叠。默认情况下，链接器会进行检查，如果发现有重叠，也会产生合适的错误信息。链接器以自己知道的情况，尽量允许分区地址的重叠。可以通过命令行选项开关 --check-sections 来恢复链接器的默认行为。在可重定位的链接中，一般不会进行分区重叠检查。在那种情况下，你可以通过--check-sections 选项强制开启检查。

--copy-dt-needed-entries

--no-copy-dt-needed-entries

此选项会影响对待一些动态库的方式，这些动态库是被在命令行中涉及的另一些 ELF 动态库的 DT_NEEDED 标签中提到的。通常，链接器不会为输入的动态库的 DT_NEEDED 标签中提及的每个库在输出二进制文件中添加 DT_NEEDED 标签。使在命令行中使用--copy-dt-needed-entries，会让其后的动态库如论何时都将自己的 DT_NEEDED 条目添加。使用--no-copy-dt-needed-entries 恢复默认状态。

此选项也会影响对动态库中符号的解析。在命令行用--copy-dt-needed-entries 的动态库会在它们指向其它库的 DT_NEEDED 标签指引下被递归地搜索，为输出的二进制文件所需的符号提供解析。然而默认设置下，随后的动态库的符号搜索会与动态库自身一起停止。没有 DT_NEEDED 的链接不会解析符号。

--cref

输出一个交叉引用表。如果正在生成链接器图文件，这个交叉引用表会输出在图文件中。否则，会输出在标准输出设备上。

表格式有意做的非常简单，这样如果需要时，很容易被脚本所处理。打印出符号以名称排序。为每个符号都给出一个文件名列表。如果符号被定义了，列出的第一个文件就是此符号定义所在的位置。如果符号被定义为一个公共符号(common symbol, 是指未进行初始化的变量)，那么所有出现此符号的文件都会被列出。

--no-define-common

此选项阻止对公共符号进行地址分配。`INHIBIT_COMMON_ALLOCATION' 脚本命令具有同等的效果。参考 杂项命令。

`--no-define-common'选项允许从输出文件的类型选择中确定对公共符号的地址分 配; 否则, 一个非重定位输出类型强制为公共符号分配地址. 使用'--no-define-common'允许那些从共享库中引用的公共符号只在主程序中被分配地 址. 这会避免在共享库中无用的重复空间, 同时,也防止了在有多个指定了搜索路径的动态模块在进行运行时符号解析时引起的混乱.

--force-group-allocation

此选项会使链接器像一般的输入分区一样放置分区组成员，而且删除分区组。这是最终链接时的默认选项，但是你可以用它来更改可重定位链接的行为（ ‘-r’ ）。脚本文件 FORCE_GROUP_ALLOCATION 具有相同效果。参考 杂项命令。

--defsym=*symbol=expression*

在输出文件中建立一个全局符号,这个符号包含一个 expression 指定的绝对地址. 你可以在命令行中多次使用这个选项定义多个符号. expression 支持一个受限形式的算术运算: 你可以给出一个十六进制常数或者一个已存在符号的名字,或者使用'+'和'-'来加或减十六进制常数或符号。如果你需要更多的表达式,可以考虑在脚本中使用连接器命令语言。参考：赋值。注意在 symbol,=和 expression 之间不允许有空格.

--demangle[=*style*]

--no-demangle

这些选项控制是否在错误信息和其它的输出中重组符号名. 当连接器被告知要重组,它会试图把符号名以一种可读的形式展现: 如果符号被以目标文件格式使用,它剥去前导的下划线,并且把 C++形式的符号名转换成用户可读的名字. 不同的编译器有不同的重组形式. 可选的重组形式参数可以被用来为你的编译器选择一个相应的重组形式. 连接器会以缺省形式重组除非环境变量`COLLECT_NO_DEMANGLE'被设置. 这些选项可以被用来重载缺省的设置.

-I*file*

--dynamic-linker=*file*

设置动态链接器的名字. 只在产生动态链接的 ELF 可执行文件时有效. 缺省的动态链接器通常是正确的; 除非你知道你在干什么，否则不要使用这个选项.

--no-dynamic-linker

当产生可执行文件时，省略对加载时所需的动态链接器的需求。只在包含动态重定位的 ELF 可执行程序时有意义，而且通常需要能够处理这些重定位的入口代码。

--embedded-relocs

此选项和--emit-relocs 很类似，只不过这些重定位被保存在目标中指定的分区中。这个选项只被‘BFIN’，‘CR16’和 *M68K* 所支持。

--fatal-warnings

--no-fatal-warnings

把所有的警告都视为错误。使用--no-fatal-warnings 恢复默认的行为。

--force-exe-suffix

确保输出文件有一个.exe 后缀. 如果一个被成功完整连接的输出文件不带有一个'.exe'或'.dll'后缀, 这个选项会强制链接器把输出文件拷贝成带有'.exe'后缀的同名文件. 这个选项在微软系统上编译未经修改的 Unix 的 makefile 时很有用, 因为有些版本的 windows 不会运行一个不带有'.exe'后缀的映像.

--gc-sections

--no-gc-sections

允许对未使用的输入分区的垃圾收集. 在不支持这个选项的平台上,会被忽略. 缺省行为（不执行垃圾收集）可以用 `--no-gc-sections'进行恢复。注意：支持 COFF 和 PE 格式目标上的垃圾收集，只不过当前的实现仍在试验阶段。

‘--gc-sections’决定测试符号和重定位会使用哪个输入分区。包含入口符号以及包含在命令行中指定的未定义符号的分区都会被保留，包含被动态对象引用符号的分区同样被保留。注意：当创建共享库时，链接器必须假设所有可见的符号都被引用了。这些分区的集合只在初始时设置一次，链接器会递归地标记所有被重定位所引用的那些分区。参考 ‘--entry’和‘--undefined’。

此选项可以在做局部链接（用 ‘-r’开启）时被设置。在这种情况下，保持符号的根必须明确地用‘--entry’ 或者 ‘--undefined’选项提定，或者使用链接脚本的 ENTRY 命令。

--print-gc-sections

--no-print-gc-sections

列出被垃圾收集所移除的分区。列表会打印在 stderr 上。此选项只在通过’ -gc-sections’ 选项开启了垃圾收集后有效。可以通过命令行选项 ‘--no-print-gc-sections’ 恢复默认行为。

--gc-keep-exported

当 ‘--gc-sections’ 开启时，此选项阻止对未使用的包含拥有默认或受保护可见属性的全局符号的输入分区的垃圾收集。此选项用在包含未引用分区的可执行程序中，否则，这些分区中所包含的符号无论是否对外可见都会被垃圾收集。注意：此选项在链接共享对象时无效，因为这是那时的默认行为。此选项只支持 ELF 格式的目标。

--print-output-format

打印默认的输出格式名称（可能被其它命令行选项所影响）。这是个在链接器脚本命令的 OUTPUT_FORMAT 中出现的一个字符串。参考 文件相关命令。

--print-memory-usage

打印使用的大小、总大小以及由 MEMORY 命令创建的存储区域使用的大小。这在嵌入式目标中为了快速查看剩余存储是非常有用的。输出的格式中包含一个标题行和每多个区域行，每行表示一个区域。此格式又易读又易于工具解析。例子如下：

Memory region	Used Size	Region Size	%age Used
ROM:	256 KB	1 MB	25.00%
RAM:	32 B	2 GB	0.00%

--help

在标准输出设备上打印命令行选项的概述，然后退出。

--target-help

在标准输出设备上打印所有目标上专用的选项概述，然后退出。

-Map=*mapfile*

打印链接图到 mapfile 文件中。参考前面出现的 -M 选项。

--no-keep-memory

Ld 通常在优化时运行速度优先于内存使用，例如在内存中缓存输入文件的符号表。此选项告诉 ld 优化时节省内存优于速度，在需要访问符号表时重新读取。当链接一个巨大的可执行程序时，超出内存访问空间时，此选项是必需的。

--no-undefined

-z defs

从常规的 object 文件中报告出未解决的符号引用。即使正在链接无符号的共享库，也可以完成此操作。开关`--[no-]allow-shlib-undefined'控制对共享库中被链接的未解决的引用是否报告。

可以通过使用-z undefs 取消此选项带来的影响。

--allow-multiple-definition

-z muldefs

通常,当一个符号被多次定义时，链接器会报告一个致命错误. 这些选项允许重定义并且第一个定义被使用。

--allow-shlib-undefined

--no-allow-shlib-undefined

允许(缺省)或不允许无定义符号存在于共享对象中. 这个开关和--no-undefined 很类似，只不过此开关决定在共享库中未定义的符号的行为，而不是常规 object 文件中的。它不会影响常规 object 文件中未定义符号的处理。默认情况下，链接器在创建可执行程序时，若在共享库中发现任何未定义符号时会报告错误，不过如果是在创建共享库，这些未定义的符号是允许的。

为何在共享库中允许未定义符号呢：

- 在链接时被指定的共享库可能并不是加载时的那个,所以符号事实上可能会在加载时才解析。
- 有一些操作系统中,在共享库中有未定义符号是正常的。例如：BeOS、HPPA。举例说明，BeOS 内核在加载时给共享库打补丁，来选择哪个函数对当前的架构最适合。再举例，可用来动态地选择合适的 memset 函数。

--no-undefined-version

通常当一个符号有一个未定义的版本时,链接器会忽略它. 这个选项不允许符号有未定义的版本，碰到这种情况,会报告一个严重错误。

--default-symver

为没版本的对外导出符号创建并使用一个默认的符号版本(soname)。

--default-imported-symver

为没版本的导入符号创建并使用一个默认的符号版本(soname)。

--no-warn-mismatch

通常, 如果你因为一些原因,企图把一些不匹配的输入文件链接起来的时候,ld 会给出一个错误。可能这些文件是由不同的处理器编译或者大小端不同。 这个选项告诉'ld'应当对这样的错误默认允许。 这个选项必须小心使用，只有当你采用了一些特别的手段来确保链接器的错误可被忽略时才可使用。

--no-warn-search-mismatch

通常，ld 在搜索库时若发现了不兼容的库，会给出警告。此选项可关闭警告。

--no-whole-archive

为其后的 archive 文件关闭--whole-archive 选项所带来的影响。

--noinhibit-exec

无论是否可用，都保留输出可执行文件。 通常，链接器如果在链接过程中遇到了任何错误,都不会产生输出文件。

-nostdlib

仅搜索那些在命令行上显式指定的库路径. 在链接脚本中(包含在命令行上指定的链接脚本)指定的库路径都被忽略.

--oformat=*output-format*

'ld'可以被配置为支持多种 object 文件。 如果'ld'以此方式配置，你可以使用'--oformat'选项来指定输出 object 文件的二进制格式。就算'ld'被配置为支持多种可选的 object 格式，你也不必指定此选项，因为'ld'应当被配置为产生在各种机器上最常用的输出格式的文件。Output-format 是个文本串,是被 BFD 库支持的一个特定格式的名字.(你可以使用 objdump -i 列出这些被支持的格式名。)脚本命令 OUTPUT_FORMAT 也可以指定输出格式，但此选项会覆盖它。参考 BFD。

--out-implib *file*

创建一个与正在链接的可执行程序（例如：DLL 或 ELF 程序）相匹配的输入库文件。这个输入库（一般称为 DLL 的*.dll.a 或者*.a）可用来与编译生成其它程序。使得跳过了生成单独的导入库的过程（例如：DLL 的 dlltool）。此选项仅仅对 i386 PE 和 ELF 目标的链接有效。

-pie

--pic-executable

创建位置独立的可执行程序。只在 ELF 平台上被支持。位置独立的可执行程序和共享库有些相像，它们会被动态链接器放在操作系统提供的虚拟地址上，因此地址在每次调用时可能会不同。就像普通的动态链接的可执行程序一样，它们可被执行，而且在其中定义的符号不能被共享库中的覆盖。

-qmagic

这个选项被忽略,只是为了跟 Linux 保持兼容.

-Qy

这个选项被忽略,只是为了跟 SVR4 保持兼容.

--relax

--no-relax

一个机器相关的选项. 只有在少数平台上,这个选项被支持. 参考 ld and the H8/300. 参考 ld and the Intel 960 family. 参考 机器的不同特性

在某些平台上,'--relax'选项使链接器在针对特定目标进行地址分配优化成为可能。链接器会在松散地址模式、合成新指令、选择当前指令的更短版本，以及合并常量时进行优化。

在某些平台上,链接时进行的全局时间优化会使最终程序无法进行符号调试。常见于 Matsushita MN10200 和 MN10300 处理器族。

在不支持这个选项的平台上,'--relax'能被接受,但会被忽略.

在支持这一选项的平台上，’ --no-relax’ 可用来关闭这一特性。

--retain-symbols-file=*filename*

只保留在 filename 文件中列出的符号，其它符号都丢弃。Filename 是个简单的文件，每行一个符号名。这一选项在某些环境（VxWorks）下特别有用，因为这种环境中有个逐渐累积起来的超大的全局符号表，为了节约运行时存储使用此选项。

‘--retain-symbols-file’ 不会丢弃未定义的符号，或者需要重新定位的符号。

只在命令行中指定‘--retain-symbols-file’一次即可。此选项会覆盖‘-s’和‘-S’

-rpath=*dir*

为运行时库的搜索路径增加一个目录. 在链接带有共享库的 ELF 可执行文件时有用。 '-rpath'的所有参数会被串连起来传递给运行时链接器, 链接器在运行时用它们定位共享对象. 通过使用'-rpath'选项，可以找到在链接时明确指定的共享对象所需的那些共享对象的位置。参阅关于'-rpath-link'选项的描述, 如果在链接一个 ELF 可执行文件时不使用'-rpath'选项,环境变量'LD_RUN_PATH'只要定义了，其中列出的那些位置就会被使用。

'-rptah'选项也可用在 SunOS 上. 缺省地,在 SunOS 上,连接器会从所有的'-L'选项中组成一个运行时搜索路径。如果使用了'-rpath'选项, 那运行时搜索路径就只从'-rpath'选项中得到, 忽略'-L'选项. 这在使用 gcc 时非常有用, 它会用上很多的'-L'选项，这些目录可能安装在 NFS 文件系统上。

为了兼容其它的 ELF 链接器，如果-R 选项后紧跟一个目录名而不是个文件，这就会被当成-rpath 选项。

-rpath-link=*dir*

当使用 ELF 或 SunOS 时,一个共享库可能会用到另一个共享库. 这会在 ld 把一个共享库作为输入文件进行共享链接时发生。

当链接器在进行非共享、不可重定位链接时，如果遇上这种依赖，如果没有被显式包含，它会自动尝试定位需要的共享库，把它包含在链接中。这这种情况下，'-rpath-link'选项指定优先搜索的一组路径名.'-rpath-link'选项可以指定一个目录名序列，要么使用由分号分隔的列表，要么多次指定 ‘-rpath-link’ 选项。

$ORIGIN 和$LIB 可以在这些目录名中出现。它们会被替换成为包含有该程序($ORIGIN)或共享目录的完整路径名。$LIB 要么是 32 位的 lib ,要么是 64 位的 lib64。*${ORIGIN}* 和 *${LIB}* 这两个变体也可使用。不支持 *$PLATFORM。*

要小心地使用此选项，因为它会覆盖那些可能已经被编译进共享库中的搜索路径。在这种情况下，就有可能无意间使用了一个运行时链接器并不想使用的搜索路径。

链接器使用下面的搜索路径来定位所需的共享库:

1.由-rpath-link 选项指定的所有目录。

2.由-rpath 选项指定的所有目录。在-rpath 与-rpath-link 之间的不同是 :-rpath 选项指定的目录会包含在可执行文件中，在运行时使用；而由-rpath-link 选项指定的目录只影响链接时。通过这种方式搜索-rpath 中的目录只被本地链接器以及使用--with-sysroot 选项配置过的交叉链接器支持。

3.在 ELF 系统中，本地链接器，如果未使用-rpath 和-rpath-link 选项，会搜索环境变量 LD_RUN_PATH 中指定的目录。

4.在 SunOS 中，如果未使用-rpath 选项，会搜索-L 选项指定的目录。

5.对于本地链接器，搜索环境变量 LD_LIBRARY_PATH 中指定的目录。

6.对于本地 ELF 链接器，共享库在 DT_RUNPATH 或者 DT_RPATH 中的目录搜索自己需要的其它共享库。DT_RUNPATH 条目如果存在，DT_RPATH 条目会被忽略。

7.一般默认的目录是/lib 和/usr/lib.

8.在 ELF 系统中的本地链接器，如果/etc/ld.so.conf 文件存在，目录会在该文件中找到。

假如所需的共享库未被找到，链接器会报告一个警告，然后继续链接。

-shared

-Bshareable

创建一个共享库. 这个选项只在 ELF, XCOFF 和 SunOS 平台上有用。 在 SunOS 上，如果'-e'选项没被使用，并在链接中有未定义符号，链接器会自动创建一个共享库。

--sort-common

--sort-common=ascending

--sort-common=descending

这个选项告诉'ld'当它把公共符号放到相应的输出分区中时升序还是降序。符号会按 16 字节或更大，8 字节，4 字节，2 字节，最后 1 字节的顺序安排。这是为了避免因为对齐而在符号间产生的空白的间隙。如果未指定，默认是降序排列。

--sort-section=name

此选项会在链接脚本中的通配符分区模式应用 SORT_BY_NAME。

--sort-section=alignment

此选项会在链接脚本中的通配符分区模式应用 SORT_BY_ALIGNMENT。

--spare-dynamic-tags=*count*

此选项指定为 ELF 共享对象的.dynamic 分区预留的空槽数量。空槽用于后期处理工具，例如预链接器。默认是 5。

--split-by-file[=*size*]

和--split-by-reloc 相似，当输入文件的尺寸达到指定值时，则为此文件创建一个新的输出分区。如果未指定，Size 默认是 1。

--split-by-reloc[=*count*]

尝试在输出文件中创建额外的区，这样每个区中都不会有超过 count 数量的可重定位的符号。这在生成巨大的重定位文件（用于下载到实时内核中的 COFF 文件格式的重定位文件）时非常有用。因为 COFF 不能在一个分区中表示超过 65536 个可重定位符号。注意：对不支持任意分区的 Object 文件格式执行此操作会失败。链接器不会分割独立的输入分区，所以如果一个单独的输入区包含超过 count 个数的可分配对象，那么在对应的输出区中也会包含这么多个。Count 默认值是 32768.

--stats

计算且显示链接操作的统计信息，假如执行时间以及内存使用情况。

--sysroot=*directory*

用 directory 作为 sysroot，覆盖掉默认的配置。此选项只对配置了--with-sysroot 的链接器有效。

--task-link

此选项是为基于 COFF/PE 的目标创建任务链接的文件时使用的，这种文件中所有的全局符号都已被转化为静态。

--traditional-format

对于一些目标，LD 的输出在某些方面与已经存在的链接器输出的不同。此开关选项命令 ld 用传统格式。

例如，在 SunOS 系统中，ld 将重复的条目组合到符号串表。这能将包含全部调试信息的输出文件尺寸减少 30%。可惜，SunOS dbx 程序不能读取这样的程序（gdb 就没问题）。 '--traditional-format' 开关告诉 ld 不要组合重复的符号。

--section-start=*sectionname*=*org*

将分区在输出文件中放在以 org 指定的绝对地址上。你可以在命令行上多次使用此选项，放置多个分区到绝对地址。Org 必须是一个 16 进制的整型数；为兼容其它链接器，你可以省略 16 进制值开头常见的 0x。注意：在 sectionname, 等号, org 之间不要有任何空格。

-Tbss=*org*

-Tdata=*org*

-Ttext=*org*

和--section-start 功能一样，放置.bss，.data，或.text 三个分区的位置。

-Ttext-segment=*org*

在创建 ELF 可执行程序时，这可以设置 text 段的首字节地址。

-Trodata-segment=*org*

在创建 ELF 可执行程序或者共享对象时，里面包含的只读数据，区别于执行的代码，这可以设置只读数据段首字节的地址。

-Tldata-segment=*org*

在为 x86-64 的中型内存模型创建 ELF 可执行程序或共享对象时 ,这可以设置 ldata 段首字节的地址。

--unresolved-symbols=*method*

决定如何处理未解析的符号。Method 有 4 种可能的值：

'ignore-all'

不可报告任何未解析的符号。

'report-all'

报告所有未解析的符号。这是默认值。

'ignore-in-object-files'

报告包含在共享库中未解析的符号，而忽略常规 object 文件中的。

'ignore-in-shared-libs'

报告常规 object 文件中未解析的符号，而忽略共享库中的。这在创建动态二进制文件时很有用，并且众所周知，它应该引用的所有共享库都包含在链接器的命令行中。共享库的行为也可以被--[no-]allow-shlib-undefined 选项所控制。

一般情况，链接器会为每个未解析的符号产生一个错误消息，但是

--warn-unresolved-symbols 选项可以将其改为一个警告。

--dll-verbose

--verbose[=*NUMBER*]

显示 ld 的版本号，列出链接器支持的仿真。显示哪些输入文件能和不能被打开。显示链接器当前使用的链接脚本。如果 NUMBER 参数>1，插件符号状态也会被显示。

--version-script=*version-scriptfile*

为链接器指定版本脚本的文件名。典型地，在创建共享库时，为其指定版本层级关系等附加信息就可使用此选项。此选项只在支持共享库的 ELF 平台上被完全的支持；参考 版本 。在 PE 平台上被部分支持，可用版本脚本在自动导出模式下过滤符号可见性：任何在版本脚本中标记为‘local’的符号将不会被导出。参考 机器的不同特性之 WIN32。

--warn-common

当一个公共符号跟另一个公共符号或符号定义合并起来时，警告。Unix 链接器允许这个稍显草率的选项，但是在其他的操作系统上的链接器不允许。这个选项可以让你在合并全局符号时发现某些潜在问题. 可惜，有些 C 库使用这项特性,所以你可能会像在你的程序中一样,在库中得到一些警告。

有三类全局符号，用 C 语言举例如下：

‘int i = 1;’

这是个符号定义，这会被放置在输出文件的有初始值的数据区中。

‘extern int i;’

这是一个未定义的符号引用，并不分配空间。在某处必须有此变量的一个定义或公共符号。

‘int i;’

这是一个公共符号。如果某个变量只有公共符号，那会被放置在输出文件的未初始化的数据区中。链接器会将相同变量的多个公共符号合并成一个符号。如果这些符号的大小不同，会选择最大的那个尺寸。如果对同一变量还有个符号定义，那链接器会将公共符号转变成定义。

‘--warn-common’选项会产生 5 种警告。每个警告由两行组成：第一行描述遭遇的符号，第二行描述同名的上一个符号。这两个符号中至少有一个是公共符号。

1.由于已经有符号的定义了，将一个公共符号变为一个引用。

file(*section*): warning: common of `*symbol*'
overridden by definition
file(*section*): warning: defined here

2.由于后面又出现的符号的定义，将这个公共符号变为一个引用。这和上种情况相同，只不过符号出现的顺序不同。

file(*section*): warning: definition of `*symbol*'
overriding common
file(*section*): warning: common is here

3.将两个同等大小的公共符号合并。

file(*section*): warning: multiple common
of `*symbol*'

file(*section*): warning: previous common is here

4.与上一个更大尺寸的符号合并。

file(*section*): warning: common of `*symbol*'
overridden by larger common

file(*section*): warning: larger common is here

5.与上一个稍小尺寸的符号合并。与上种情况相同 ,只不过符号出现的顺序不同。

file(*section*): warning: common of `*symbol*'
overriding smaller common

file(*section*): warning: smaller common is here

--warn-constructors

如果任何全局的构造函数被使用则警告。这只对一些 object 文件格式有用。对于像 COFF 或 ELF 的格式，链接器无法检测到全局构造函数的使用。

--warn-multiple-gp

如果在输出文件中,需要多个全局指针值,则警告。 这只对特定的处理器有意义，比如 Alpha. 特别的,有些处理器在特定的分区中放入很大的常数值. 一个特殊的寄存器(全局指针)指向这个分区的中间部分，所以 ,通过一个基地址寄存器相关的地址模式,这个常数可以被高效的载入。因为这个基寄存器相关模式的偏移值是固定的而且很小(比如,16 位)，这会限制常量池的最大尺寸。 因此,对于一个大程序,为了能够给所有可能的常数编址，经常需要使用多个全局指针值。这个选项在这种情况发生时产生一条警告。

--warn-once

对于每一个未定义符号只警告一次，而不是在每一个用到它的模块中警告一次.

--warn-section-align

如果输出分区的地址因为对齐被改变了,则警告。典型的，会在输入分区中设置对齐。只有分区没有明确指定地址时才会被改变；也就是，如果'SECTIONS'命令没有指定分区的起始地址。参考 SECTIONS。

--warn-shared-textrel

如果链接器增加一个 DT_TEXTREL 到共享对象中，则警告。

--warn-alternate-em

如果一个对象有交替的 ELF 机器码，则警告。

--warn-unresolved-symbols

如果链接器发现有未解析的符号时（参考--unresolved-symbols 选项）一般会产生一个错误。此选项会使这种情况产生一个警告而不是错误。

--error-unresolved-symbols

此选项可用来恢复链接器的默认行为。当发现未解析符号时，产生并报告错误。

--whole-archive

对于在命令行选项'--whole-archive'后提及的每个档案文件,在链接中包含档案文件中的所有目标文件，而不是在档案文件中搜索需要的目标文件。这通常用于将档案文件转化为共享库,把所有的对象强制放到最终的共享库中。此选项可多次使用。在 GCC 中使用这个选项需要注意两点: 首先,GCC 不知道这个选项，所以,你必须使用'-Wl,-whole-archive'。第二，不要忘了在你的档案文件列表后使用'-Wl,-no-whole-archive'，因为 GCC 会把它自己的档案列表加到你的链接后面,而你可能并不希望此选项也影响它们。

--wrap=*symbol*

对 symbol 符号使用包装函数. 任何未定义的对 symbol 符号的引用会被解析成'_wrap_symbol'。任何未定义的对'_real_symbol'的引用会被解析成 symbol。这可以用来为系统函数提供一个包装。包装函数应当以'__wrap_symbol'名称调用。如果需要调用对应的系统函数，那就应该调用'__real_symbol'。

下面是个简单的例子：

```
void * __wrap_malloc (size_t c)
{
  printf ("malloc called with %zu\n", c);
  return __real_malloc (c);
}
```

如果与其它代码一起链接此代码，并且指定--wrap malloc，那么所有对 malloc 的调用将会调用__wrap_malloc 函数。而在__wrap_malloc 中的__real_malloc 调用才是对真正的 malloc 函数的调用。

你可能也希望提供一个__real_malloc 函数，这样即使不用--wrap 选项也能成功链接。如果你这样做，那不要把__real_malloc 的定义和__wrap_malloc 放在一个文件中；如果你将其放在一个文件中，汇编器可能已经在链接器工作前就已经解析了对此符号的调用。

--eh-frame-hdr

--no-eh-frame-hdr

请求 (--eh-frame-hdr) 或者抑制 (--no-eh-frame-hdr)对.eh_frame_hdr 分区以及 ELF PT_GNU_EH_FRAME 段头的创建。

--no-ld-generated-unwind-info

请求创建链接器生成的代码分区(例如 PLT)的展开信息。此选项对支持生成展开信息的链接器是默认打开的。

--enable-new-dtags

--disable-new-dtags

链接器可以在 ELF 中创建一个新的动态标签。但是旧的 ELF 系统可能不理解。如果指定了'--enable-new-dtags'，新的动态标签会按需要被创建，而旧的动态标签会被忽略。如果指定了'--disable-new-dtags',那不会有新的动态标签被创建。缺省

地，新的动态标签不会被创建。注意这些选项只在 ELF 系统中有效。

--hash-size=*number*

设置链接器的哈希表的大小为接近 number 的一个素数。增大此值可以使链接器执行任务时所需的时间变短，代价就是，链接器所需的存储空间也会相应增加。相似地，减少此值会降低对存储的需求，但同样也减慢了工作的速度。

--hash-style=*style*

设置链接器哈希表类型。style 可以是 sysv(经典 ELF.hash 分区), gnu（新类型 GNU.gnu.hash 分区），也可以是 both(表示以上的两种哈希表)。默认值是 sysv。

--compress-debug-sections=none

--compress-debug-sections=zlib

--compress-debug-sections=zlib-gnu

--compress-debug-sections=zlib-gabi

在 ELF 平台，这些选项控制着 zlib 如何压缩 DWARF 调试分区。

--compress-debug-sections=none 不压缩 DWARF 调试分区。

--compress-debug-sections=zlib-gnu 压缩 DWARF 调试分区，并且重命名，以.zdebug 开头，而不是.debug。--compress-debug-sections=zlib-gabi 压缩 DWARF 调试分区，但不更改名称，而是在区头中设置 SHF_COMPRESSED 标签。

--compress...sections=zlib 选项是--compress...sections=zlib-gabi 的别名。

注意此选项会覆盖在输入调试分区中的任何压缩，举例说明，如果二进制文件使用 --compress-debug-sections=none 进行链接，那么输入文件中的所有压缩过的调试分区在复制到输出二进制文件前都会被解压。

默认的压缩行为多种多样，依赖目标内置属性，也依赖在生成工具链时所使用的配置选项。可以通过检查链接器的--help 选项来确定其默认行为。

--reduce-memory-overheads

此选项减少 ld 运行时所需的存储，代价是降低链接速度。这会使链接器选择旧算法(n^2)来产生链接图，而不使用比原来多出 40%的存储来保存符号的新算法(n)。

此开关的另一个影响是会设置哈希表的默认大小为 1021，这也会节约存储，但会加长链接器的运行时间。然而如果用了--hash-size 选项，哈希表的大小就不会更改。

--reduce-memory-overheads 开关也可用来开启链接器的未来版本中的其它折衷特性(tradeoff)。

--build-id

--build-id=*style*

申请创建一个.note.gnu.build-id ELF 注解分区或者一个.buildid COFF 分区。注解的内容是用来识别这个链接文件的唯一的一组二进制位。style 可以是 128 随机位的 uuid，也可以是按照规范用部分输出内容生成的 160 位 SHA1 哈希值，还可能是 128 位按规范用部分输出内容生成的 MD5 哈希值，又或者 0x 开头的 16 进制数值串来指定一个 16 进制的偶数。（在数字对之间的-和:字符会被忽略。）如果忽略 style，sha1 会被使用。

Md5 和 sha1 类型对于同一个输出文件总会产生相同的识别符，而对不同的输出文件产生的识别符总是不同。不要用来校验文件的内容。链接产生的文件可能会被后来的工具更改，可是用一开始链接产生的文件生成的 build-id 串并不会随之更改。为 style 参数传递 none 会关闭命令行中之前为--build-id 设置的所有值。

2.1.1 i386 PE 专用的选项

i386 PE 链接器支持'-shared'选项，它使链接器输出一个动态链接库(DLL),而不是普通的可执行文件。在使用这个选项的时候，你应当为输出文件取名'*.dll'。另外，链接器完全支持标准的'*.def'文件，这类文件可以在链接器命令行上象一个目标文件一样被指定(实际上，它应当被放在它从中导出符号的那些档案文件之前，以保证它们象普通目标文件一样被链接)。

除了对所有平台通用的那些选项外，i386 PE 链接器支持一些 i386 平台专有的命令行选项。带有值的选项应当用空格或等号把它跟值分开。

--add-stdcall-alias

如果给出这个选项，带有标准调用后缀(@NN)的符号会被剥掉后缀后导出。[此选项专门用于 i386 PE 目标接口]。

--base-file *file*

使用 file 作为文件名，该文件用于存放用'dlltool'产生 DLL 文件时所需的所有重定位符号基地址。[此选项 i386 PE 平台专用]

--dll

创建一个 DLL 文件而不是常规可执行文件。你可能使用'-shared'选项，或者在'.def'文件中指定'LIBRARY'。[此选项专门用于链接器的 i386 PE 目标接口]

--enable-long-section-names

--disable-long-section-names

为 COFF 对象格式的 PE 变量增加一个扩展，允许使用超过 8 个字符长的分区名(这是 COFF 的限制）。默认情况，这些名称只允许在对象文件中作为完整链接的可执行镜像，并不搬移用于支持长名称的 COFF 字符串表。作为 GNU 的一个扩展，可以使用这两个选项在可执行镜像时允许而在对象文件时禁止（可能根本无意义）的操作。使用这些长分区名生成的可执行镜像稍微有点不标准，就像对待一个字符串表一样，而且在使用非 GNU 的 PE 相关工具(例如文件查看器或者 dumper)检查此镜像时，有可能会产生错乱的输出。然而，GDB 依赖 PE 的长分区名在运行时在可执行镜像中找到 Dwarf-2 调试信息分区。如果在命令行中没有指定此选项，当 ld 在链接可执行镜像而不剥离符号，为符号寻找调试信息时，就会启用长分区名，覆盖默认的技术上正确的行为。[此选项对于所有链接器的 PE 目标接口有效]

--enable-stdcall-fixup

--disable-stdcall-fixup

如果链接时发现一个不能解析的符号，链接器会尝试进行'模糊链接',即寻找另一个定义的符号,它们只是在符号名的格式上不同(cdecl vs stdcall),并把符号解析为找

到的这个符号。比如，一个未定义的符号'_foo'可能被链接到函数'_foo@12'，或者未定义的符号'_bar@16'可能被链接到函数'_bar'。当链接器这样做时，它会打印出一条警告信息，因为在正常情况下,这会链接失败,但有时,由第三方库产生的导入库可能需要这个特性才可用。如果你指定'--enable-stdcall-fixup'，这个特性会被完全开启,警告信息也不会打印了。如果你指定'--disable-stdcall-fixup'，这个特性被关闭，这样的不匹配会被认为是错误。[此选项对于链接器的 i386 PE 目标接口有效]

--leading-underscore

--no-leading-underscore

对于大多种目标来说，默认的符号前辍是个下划线，是在目标描述中定义的。通过此选项，你可以开启/关闭这个默认的下划线前辍。

--export-all-symbols

如果给出这个选项，对象中用来建立 DLL 的所有全局符号都会被导出。注意这是缺省情况，否则没有任何符号会被导出。如果符号由 DEF 文件显式地导出，或由函数本身的属性隐式地导出，缺省情况是不导出任何其他的符号，除非指定此选项。注意符号`DllMain@12',`DllEntryPoint@0', `DllMainCRTStartup@12'和`impure_ptr'不会自动被导出。而且，由其他的 DLL 导入的符号也不会被再次导出,那些指定 DLL 内部布局的符号，比如以'_head_'开头,或者以'_iname'结尾的符号也不会被导出。另外,'libgcc','libstd++','libmingw32'或'crtX.o'中的符号也不会被导出。为帮助 C++DLL，以__rtti_或者__builtin_开头的符号不会被导出。最后，有个很大的不被导出的 cygwin 私有符号列表（显然，在为 cygwin 目标创建 DLL 时会应用）。有些要排除：_cygwin_dll_entry@12, _cygwin_crt0_common@8,_cygwin_noncygwin_dll_entry@12, _fmode, _impure_ptr, cygwin_attach_dll, cygwin_premain0, cygwin_premain1, cygwin_premain2, cygwin_premain3,以及 environ。[此选项对于链接器的 i386 PE 目标接口有效]

--exclude-symbols *symbol*,*symbol*,...

指定一个不要自动导出的符号列表。符号名之间用逗号或冒号分隔。[此选项对于链接器的 i386 PE 目标接口有效]

--exclude-all-symbols

指定所有符号都不要被自动导出。[此选项对于链接器的 i386 PE 目标接口有效]

--file-alignment

指定文件对齐。文件中的分区总是以文件偏移开始，这个偏移是此数字的倍数。数字默认是 512。[此选项对于链接器的 i386 PE 目标接口有效]

--heap *reserve*

--heap *reserve*,*commit*

指定为此程序预留的（commit 可选，是指操作系统每次分配内存的最小单元）用作堆空间的以字节为单位的内存数量。默认预留 1MB，commit 4KB。[此选项对于链接器的 i386 PE 目标接口有效]

--image-base *value*

以 value 作为程序或 dll 的基地址。这是你的程序或 dll 加载时会用到的最低存储位置。为减少重定位的需求，提高 dll 的性能，每个 dll 都要有个唯一的基地址，不要和其它 dll 有重叠。默认的，可执行文件是 0x400000，Dll 是 0x10000000。[此选项对于链接器的 i386 PE 目标接口有效]

--kill-at

如果使用此选项，stdcall 后辍(@nn)在被导出前会被去掉。[此选项对于链接器的 i386 PE 目标接口有效]

--large-address-aware

如果使用此选项，COFF 头的 Characteristics 区域中合适的位会被设置，这样就使此执行程序能够支持超过 2GB 的虚拟地址。此选项应该和在 BOOT.INI 中的[operating systems]节中的/3GB、/USERVA=valuemegabytes 开关一起使用。否则，此开关不生效。[此选项对于链接器的 PE 目标接口有效]

--disable-large-address-aware

与上一个选项相反。如果编译器总是设置--large-address-aware 开启(例如：Cygwin gcc)，可是执行程序并不支持超过 2GB 的虚拟地址时，此选项就很有用了。[此选项对于链接器的 PE 目标接口有效]

--major-image-version *value*

设置”image version”的主版本号。默认是 1。[此选项对于链接器的 i386 PE 目标接口有效]

--major-os-version *value*

设置“os version”的主版本号。默认是 4。[此选项对于链接器的 i386 PE 目标接口有效]

--major-subsystem-version *value*

设置“subsystem version”的主版本号。默认是 4。[此选项对于链接器的 i386 PE 目标接口有效]

--minor-image-version *value*

设置”image version”的副版本号。默认是 0。[此选项对于链接器的 i386 PE 目标接口有效]

--minor-os-version *value*

设置“os version”的副版本号。默认是 0。[此选项对于链接器的 i386 PE 目标接口有效]

--minor-subsystem-version *value*

设置“subsystem version”的副版本号。默认是 0。[此选项对于链接器的 i386 PE 目标接口有效]

--output-def *file*

链接器会在生成 DLL 文件的同时创建与 DLL 相匹配的 DEF 文件。此 DEF 文件（应该被叫做*.def）可以被 dlltool 使用创建一个导入库，或者被用作自动隐式导出符

号的引用。[此选项对于链接器的 i386 PE 目标接口有效]

--enable-auto-image-base

--enable-auto-image-base=*value*

除非已经使用--image-base 指定过，否则自动地为 DLL 选择镜像基地址，或者以 value 开始。通过使用一个从 dllname 产生的哈希值，为每个 DLL 创建唯一的镜像基地址，就可以避免可能延迟程序执行的存储内冲突以及重定位。[此选项对于链接器的 i386 PE 目标接口有效]

--disable-auto-image-base

不自动产生唯一的镜像基地址。如果未指定--image-base ,那会使用平台的默认值。[此选项对于链接器的 i386 PE 目标接口有效]

--dll-search-prefix *string*

在不使用导入库动态链接一个 dll 时，引用<basename>.dll 库会搜索 <string><basename>.dll。此行为允许一种简单的方式来区分为多种平台创建的 DLL:本地的,cygwin,uwin,pw,等等。例如，cygwin 中的 DLL 典型的使用 --dll-search-prefix=cyg。[此选项对于链接器的 i386 PE 目标接口有效]

--enable-auto-import

把从 DLL 中导入的 DATA 符号_symbol 与__imp__symbol 进行复杂的链接。在处理带有 DATA 的导入库时进行必需的形实转换。注意：对’auto-import‘的使用将会导致镜像文件中的代码分区变为可写。这和微软发布的 PE-COFF 格式说明不符。

注意：合适’auto-import’也会导致通常放在.rdata 区中的只读数据放到.data 区中。这是为了解决下面描述的这个关于 const 的问题的：

http://www.cygwin.com/ml/cygwin/2004-09/msg01101.html

使用’auto-import’一般会立即工作，但有时可能会看到以下信息：

"variable ’<var>’ can’t be auto-imported. Please read the documentation for ld’s --enable-auto-import for details."

此消息会在一些(子)表达式访问一个由两个常量的和表示的地址（Win32 的导入表只允许一个）时发生。会发生此情况的例子中包括访问从 DLL 中导入的结构体变量的成员域，就像访问从 DLL 中导入数组的一个常量索引一样。任何多字变量（数组、结构体、long long 型，等）都会触发这种错误。然而，不管这些违规的导出变量到底是什么数据类型，ld 总是会对其进行检测，发出警告，并且退出。

有几种方法来解决这个难题，而不管导出变量的数据类型：

一种方式是使用–enable-runtime-pseudo-reloc 开关。这会将判断引用的任务留到你的运行时环境的客户代码中，因此这种方法只在运行时环境支持此特征时工作。

第二种方案是强制让其中一个常量变成变量，也就是，在编译时不可知且不可优化。对于数组，有两种可能性：a)将数组的地址保存在一个变量中，b)将常量的索引用变量来表示。因此：

extern type extern_array[];

extern_array[1] --> { volatile type *t=extern_array; t[1] }
或者
extern type extern_array[];
extern_array[1] --> { volatile int t=1; extern_array[t] }
对结构等多字数据类型，唯一的选择就是让结构体等数据自身变成变量：
extern struct s extern_struct;
extern_struct.field --> { volatile struct s *t=&extern_struct; t->field }
或者
extern long long extern_ll;
extern_ll --> { volatile long long * local_ll=&extern_ll; *local_ll }
第三种方法是为违规符号放弃’auto-import’，标计为__declspec(dllimport)。然而，在实践中需要使用编译时的#defines 指定你是在生成 DLL，生成将要链接到 DLL 的客户代码，或者只是创建/链接一个静态库。在解决’常量索引的直接地址’问题的方案选择中，你可以考虑典型的现实用法：
原型:
--foo.h
extern int arr[];
--foo.c
#include "foo.h"
void main(int argc, char **argv){
  printf("%d\n",arr[1]);
}
方案 1:
--foo.h
extern int arr[];
--foo.c
#include "foo.h"
void main(int argc, char **argv){
  /* 此方案是为了 win32 和 cygwin; 不进行优化 */
  volatile int *parr = arr;
  printf("%d\n",parr[1]);
}
方案 2:
--foo.h
/* 注意: auto-export 假设没有 __declspec(dllexport) 宏定义 */
#if (defined(_WIN32) || defined(__CYGWIN__)) && \
  !(defined(FOO_BUILD_DLL) || defined(FOO_STATIC))
#define FOO_IMPORT __declspec(dllimport)

```
#else
#define FOO_IMPORT
#endif
extern FOO_IMPORT int arr[];
--foo.c
#include "foo.h"
void main(int argc, char **argv){
  printf("%d\n",arr[1]);
}
```

第 4 种避免此问题的方法是使用功能接口的方法重新编写你的库代码，而不是使用这种违规变量的访问接口。（例如：set_foo()和 get_foo()访问函数）[此选项对于链接器的 i386 PE 目标接口有效]

--disable-auto-import

不要尝试为从 DLL 导入的 DATA 进行复杂的形实转换链接。[此选项对于链接器的 i386 PE 目标接口有效]

--enable-runtime-pseudo-reloc

如果你的代码包含在–enable-auto-import 部分描述过的表达式，那么从 DLL 中导入的 DATA 带有非 0 的偏移位置，那么此开关将创建一个‘运行时伪重分配’向量表，此表可帮助运行时环境定位客户代码中对这些数据的引用。[此选项对于链接器的 i386 PE 目标接口有效]

--disable-runtime-pseudo-reloc

不要为 DLL 中导入的非 0 偏移的 DATA 创建伪重分配表。[此选项对于链接器的 i386 PE 目标接口有效]

--enable-extra-pe-debug

显示与 auto-import 形实转换相关的附加调试信息。[此选项对于链接器的 i386 PE 目标接口有效]

--section-alignment

设置分区对齐。在存储器中的分区将总是被开始于这个数字的倍数地址。默认的数字是 0x1000。[此选项对于链接器的 i386 PE 目标接口有效]

--stack *reserve*

--stack *reserve*,*commit*

指定为此程序的栈预留的字节数（commit 可选）。默认预留-2MB，commit-4KB。[此选项对于链接器的 i386 PE 目标接口有效]

--subsystem *which*

--subsystem *which*:*major*

--subsystem *which*:*major*.*minor*

指定你的程序将会在哪个子系统下执行。合法的 which 有 native, windows, console, posix, xbox。你还可以设置子系统的版本。which 中也接受数值。[此选

项对于链接器的 i386 PE 目标接口有效]

接下来的选项会在 PE 文件头的 DLLCharacteristics 信息域设置标志：[此选项对于链接器的 PE 目标接口有效]

--high-entropy-va

镜像与 64 位地址空间布局随机化兼容(ASLR)。

--dynamicbase

镜像的基地址会使用地址空间布局随机化(ASLR)而重新放置。此特性是与 i386PE 目标的微软 Windows Vista 一同引进的。

--forceinteg

强制进行代码完整性检查。

--nxcompat

镜像与数据执行保护技术兼容。此特性是与 i386PE 目标的微软 Windows XP SP2 一同引进的。

--no-isolation

虽然镜像理解隔离，但不要隔离镜像。

--no-seh

镜像不使用 SEH。此镜像不会调用 SEH(结构化异常处理函数)。

--no-bind

不要绑定此镜像。

--wdmdriver

驱动程序使用微软 Windows 驱动模型。

--tsaware

镜像是对终端服务器可感知的。

--insert-timestamp

--no-insert-timestamp

在镜像中添加时间戳。为了匹配以前的代码，这是默认的行为，这也意味着，镜像可以和其它具有所有权的工具一起工作。这种默认行为的问题是会导致同一份代码在不同时刻编译生成时会有些微的不同。--no-insert-timestamp 选项可以用于在时间戳的位置插入 0 值，这就能保证相同的源代码可以生成出唯一的二进制文件。

2.1.2 C6X uClinux 专用的选项

C6X uClinux 目标使用叫做 DSBT 的二进制格式来支持共享库。在系统中每个共享库都需要一个唯一的索引；所有可执行程序都使用 0 索引号。

--dsbt-size *size*

此选项设置当前的可执行程序或共享库的 DSBT 中条目的数量。默认使用 64 个条目的表格。

--dsbt-index *index*

此选项设置当前程序或共享库的 DSBT 索引为 index。默认为 0，这是用来生成可执行程序的。如果共享库生成时被设置成 0 索引，R_C6000_DSBT_INDEX 会被复制到输出文件中。

‘--no-merge-exidx-entries’开关禁止对帧展开信息中的相邻 exidx 条目的合并。

2.1.3 Motorola 68HC11 和 68HC12 专用的选项

68HC11 和 68HC12 链接器支持专门的选项来控制存储体交换映射和”蹦床代码”的生成。

--no-trampoline

此选项禁止“蹦床代码”的生成。默认地，使用 jsr 指令调用每个远端函数（当使用远端指针时会发生）时，会产生一个“蹦床”。

--bank-window *name*

此选项告诉链接器在’ MEMORY’规格说明中描述存储体窗口(memory bank window)的存储区域的名称。这个区域的定义会被链接器用来和存储窗口一起进行页计算和地址计算。

2.1.4 Motorola 68K 专用的选项

以下选项是用来在链接 68K 目标时控制 GOT 生成的。

--got=*type*

此选项告诉链接器使用哪种 GOT 生成策略。Type 应该是 single、negative、multigot 或 target 中的一个。更多信息参考 ld 的 Info 条目。

2.1.5 MIPS 专用的选项

以下选项用来在链接 MIPS 目标时，控制 microMIPS 指令的生成，以及 ISA 模式转换的分支重定位检查。

--insn32

--no-insn32

这些选项控制由链接器在代码生成时所选择的 microMIPS 指令，例如在 PLT、延时绑定端(lazy binding stubs)以及 relaxation 中。如果使用’--insn32’,链接器只使用 32 位指令编码。默认的或者使用’--no-insn32’，所有的指令编码（包括 16 位的指令）都会被使用。

--ignore-branch-isa

--no-ignore-branch-isa

这些选项控制无效的 ISA 模式转换的分支重定位检查。如果--ignore-branch-isa 被使用，那么链接器允许任何的分支重定位，而且在重定位计算中的所有 ISA 模式的转换请求都会丢弃 ,除非有时所使用的 BAL 指令满足了 relaxation 条件且由计算出的对应的重定位转换成了等效的 JALX 指令。默认情况下或者使用了 --no-ignore-branch-isa 选项 ,当 ISA 模式转换发生错误时被检查且产生一个错误。

2.2环境

你可以使用环境变量 GNUTARGET, LDEMULATION 和 COLLECT_NO_DEMANGLE 来改变 ld 的行为。

如果你不使用-b(或者它的同义选项--format)，GNUTARGET 决定输入文件的格式。它的值应当是 BFD 中关于输入格式的一个名字。如果环境中没有 GNUTARGET 变量，ld 使用目标平台的缺省格式。如果 GNUTARGET 被设为 default，那 BFD 就会通过检查二进制的输入文件来寻找输入格式；这个方法通常会成功，但会有潜在的分歧，因为没有办法保证指定目标文件格式的魔数总是唯一的。然而，在每个系统上的 BFD 配置程序会把该系统的常规格式放在系统搜索列表的首位，所以分歧可以通过这种方式来解决。

如果你没有使用-m 选择，LDEMULATION 决定默认的仿真器。仿真器会影响链接器行为的很多方面，特别是默认的链接器脚本。你可以通过--verbose 或-V 选项列出所有有效的仿真器。如果-m 选项未使用，LDEMULATION 环境变量也没定义，那默认的仿真器就要依赖链接器在生成时是如何配置的了。

通常，链接器默认会进行符号重组。然而如果环境变量 COLLECT_NO_DEMANGLE 被设置，链接器就不再进行符号重组了。此环境变量以类似的方式被 gcc 链接器包装程序所使用。默认值可被--demangle 和--no-demangle 选项覆盖。

3 链接器脚本

所有的链接过程都是由链接器脚本控制的。链接器脚本是用链接器命令语言编写的。

链接器脚本的主要目的是用于描述输入文件中的分区（有的也叫段）如何映射到输出文件中，控制输出文件中代码及内存在存储器里的位置。大多数链接器脚本只做这些工作。然而，在必要时，链接器脚本也可以通过其中描述的命令来指挥链接器执行更多其它的操作。

链接器总需要使用链接器脚本。如果你没有为其准备一个，链接器会使用其内置的默认脚本。你可以使用--verbose 命令行选项来显示默认的链接器脚本。一些像-r 或者-N 的命令行选项，将会影响默认的脚本。

你可以使用-T 选项指定你自己的链接器脚本。当你这样做时，你指定的链接器脚本将会替换掉默认脚本而生效。

你也可以像其它被链接的文件一样，通过文件名的方法将包含有链接器脚本的脚本文件传递给链接器. 参考：隐含式的链接器脚本

3.1基本的脚本概念

为了描述链接器脚本语言，我们需要先定义一些基本的概念和名词。

链接器将所有的输入文件合成一个输出文件。输出文件和每个输入文件都是 object 文件格式的。每个文件都叫做 object 文件。输出文件经常被叫做执行文件，我们为了介绍清楚，也称之为 object 文件。每个 object 文件当中，除了其它部分之外，总有一个分区列表。

我们有时在输入文件中会提及的分区被称作输入分区；类似的，在输出文件中的分区称为输出分区.

Object 文件中的每个分区都有一个名字和大小。大多数分区同样包含相关联的数据 block(块)，一般称之为 section content(分区内容)。分区被标记为 loadable,就意味着输出文件在运行时，此分区中的内容（section content）应该被加载到存储器中。如果分区中没有内容可以标记为 allocatable,这意味着在存储器中的这个区域应该被留出来，什么都不要加载（有些情况下，这块存储器必须清成 0）。如果分区既不是 loadable 又不是 allocatable，一般情况下，这个分区中包含一些 debugging 信息。

每个 loadable 或者 allocatable 输出分区拥有两个地址。第一个地址是 VMA，或者称之为虚拟存储地址。当输出文件在运行时，需要使用这个地址。第二个是 LMA，也称之为加载存储地址。这是该分区将被加载到的地址。大多数情况下，这两个地址是相同的。两地址不同的一个例子是 :当一个数据分区被加载到 ROM 中 ,然后在程序运行时被复制到 RAM 中（这项技术经常用来在基于 ROM 的系统中初始化全局变量）。这种情况下，ROM 中的地址就是 LMA，而 RAM 中的地址就是 VMA.

你可以通过 objdump 命令的-h 选项来查看 object 文件中的分区信息。

每个 object 文件还包含一个 symbol(符号)列表 ,一般叫做符号表。符号可能是定义的，也可能是未定义的。每个符号都拥有一个名称，每个定义了的符号除了其它信息之外，都拥有一个地址。如果你编译 C 或 C++程序到 object 文件中，你将得到每一个定义的函数、全局变量或者静态变量所对应的符号。每个被引用的，但没在输入文件中定义的函数或全局变量会成为一个未定义的符号。

使用 nm 命令你可以查看 object 文件中的符号信息，或者使用 objdump 命令的-t 选项。

3.2脚本格式

链接器脚本是文本文件格式的。

就像编写一系列命令一样编写链接器脚本。每条命令要么是个跟着参数的关键字，要么是对符号的赋值语句。可以使用分号（；）将多条命令分隔开。空格或回车等一般会被忽略。

可以直接输入字符串，例如文件名、格式名等。如果文件名中包含逗号（，）或者其它用于分隔文件名的符号时，你可以将文件名放在双引号（”“）中间。在文件名中不可以有双引号。

你可以在链接器脚本中加入注释，就像 C 语言中一样，用’/*’和‘*/’符号包含要注释的内容。注释也会在脚本解释时忽略。

3.3简单例子

很多链接器脚本都很简单。

最简单的链接器脚本只有一个命令’SECTIONS’。你使用’SECTIONS’命令描述输出文件中存储器放置情况。

‘SECTIONS’命令是个强大的命令。在这里，我们将描述一个使用此命令的简单例子。我们假设你的程序中只包含 code(代码)、data(有初始化值的数据)和没有初始化值的数据。这些将被分别放置在’.text’，’.data’，以及’.bss’ 分区中。我们可以进一步想象，这些就是你的输入文件中出现的所有的分区。

以下这个例子，代码会被加载到 0x10000 开始的地址，data 会从 0x8000000 地址开始，没有初始值的数据(.bss)会紧跟在.data 分区之后。

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

写 SECTIONS 命令时，使用’SECTIONS’关键字，后边跟着一系列符号赋值及输出分区描述，这些输出分区的描述使用大括号（{}）包围。

在大括号中的第一行，为特殊的符号’.’赋值，这是指当前位置计数器。如果不通过其他方式(一会儿介绍其他的方式)指定输出分区的地址，那么地址将被设置成当前位置计数器的值。然后这个计数器会按输出分区的大小自动增加。在 SECTIONS 命令的开始，计数器的值为 0.

大括号中的第二行定义了输出分区，’.text‘。后边的冒号（：）是必须的。在输出分区名称后的大括号中，列出想要放在这个输出分区中的输入分区的名字列表。’*‘是个通配符，可匹配所有文件名。表达式’*(.text)‘意思是所有输入文件中的所有’.text’输入分区。

由于输出分区’.text’在定义的时候，位置计数器的值是 0x10000，链接器将把输出文件中’.text’ 分区的起始地址设置为 0x10000。

剩下的两行定义了在输出文件中’.data’和’.bss’ 分区。链接器将’.data‘输出分区放置在 0x8000000 位置。在放置完.data 分区之后，位置计数器的值是 0x8000000 再加上.data 的大小。最终效果就是链接器会在输出文件中紧跟着.data 分区后放置.bss 分区。

若有必要，链接器将通过增加位置计数器的值来确保每个输出分区是按要求被对齐的。在这个例子中，’.text’和’.data’ 分区指定的地址很可能能够满足任何对齐的约束，但链接器不得不在’.data’和’.bss’两个分区之间留个小空。

这就是个简单但完整的链接器脚本了。

3.4简单的命令

这一节，我们介绍简单的链接器脚本命令。

3.4.1 设置入口点

程序中第一条要执行的指令被叫做入口点(Entry Point)。可以使用 ENTRY 脚本命令来设置入口点，参数是一个符号名称：

ENTRY(*symbol*)

设置入口点有几种方式。链接器会按以下几种方式按次序尝试设置入口点，一旦成功则停止：

- ’-e‘ entry 命令行选项；
- 在链接器脚本中的 ENTRY(symbol)命令；
- 目标指定符号的值；好多目标都是指定 start 符号，但基于 PE 和 BeOS 的系统会检查一系列可能的入口符号，找到第一个即可。
- ‘.text’ 分区的第一个字节的地址；
- 0 地址。

3.4.2 处理文件的命令

一些链接器脚本命令用于处理文件。

INCLUDE *filename*

将链接器脚本文件包含在文件中的这个位置(类似于 C 语言中的#include filename)。被 include 的文件将会在当前目录中查找，也会在任何使用-L 选项指定的目录中寻找。可以嵌套的使用 include 文件最深 10 层。

你可以在最上层使用 INCLUDE 指令，例如在 MEMORY 或者 SECTIONS 命令中，也可以在输出分区的描述中使用。

INPUT(*file*, *file*, ...)

INPUT(*file file* ...)

INPUT 命令指示链接器将指定的这些文件包含(此包含不是 INCLUDE 命令中的包含，INCLUDE 命令是将链接器脚本的文本内容插入到当前的脚本文件中，而此处的包含是指作为输入文件)进来，用于链接，就像在命令行中指定一样。

举个例子，如果你想在链接时每次都包含 subr.o 文件，你又不想每次都在命令行中指定它，那么你就可以在链接器脚本中放入 ‘INPUT(subr.o)’ 。

事实上，如果你喜欢，你可以将所有的输入文件名列在链接器脚本文件中，然后只使用链接器的’-T’选项指定脚本文件即可。

在配置了 sysroot 前缀的情况下，并且文件名以’/’开头，被处理的脚本文件将在 sysroot 前缀指定的目录中寻找。否则，链接器将会尝试从当前目录中打开文件。如果没找到，链接器会在整个存档库查询路径中查找。Sysroot 前缀还可以通过在文件路径的第一个字符指定为‘=’来实现，或者在文件路径名前放置$SYSROOT。也可参考命令行选项中‘-L’命令选项的说明。

如果使用 ‘INPUT (-l*file*)’ ，ld 会将名称转换为 libfile.a，就像用命令行参数-l 一样。

当你在隐式链接器脚本中使用 INPUT 命令时，那些被包含的文件名会被放置在包含这个链接器脚本文件的位置上。这会影响在档案文件(archive)搜索。

GROUP(*file*, *file*, ...)

GROUP(*file file* ...)

GROUP 命令就像 INPUT，只不过被指定的文件必须都是档案文件，而且会被不断重复搜索，直到没有新的未定义引用了。参考命令行选项中'-('的描述。

AS_NEEDED(*file*, *file*, ...)

AS_NEEDED(*file file* ...)

此结构只能出现在 INPUT 或者 GROUP 命令内部，在其它的文件名之中。除了 ELF 共享库之外，所列出的文件将被当作直接出现在 INPUT 或 GROUP 命令中那样处理，只有当实际需要时才会添加这些文件。这一结构相当于开启了其中的所有文件的 --as-needed 选项，则括号结束后又恢复了之前的值。

OUTPUT(*filename*)

OUTPUT 命令指定输出文件的名称。在链接器脚本中使用 OUTPUT(*filename*)与在命令行上使用 '-o *filename*' 非常相似（参考命令行选项）。如果都有，命令行优先。可以使用 OUTPUT 命令来定义输出文件的默认名，替换掉常用的默认名 a.out。

SEARCH_DIR(*path*)

SEARCH_DIR 命令为 ld 寻找档案库的路径列表中增加路径。使用 SEARCH_DIR(*path*)就像在命令行中使用-L *path 一样*（参考命令行选项）。如果都有，链接器都会搜索。使用命令行指定的目录会先搜索。

STARTUP(*filename*)

STARTUP 命令和 INPUT 命令相像，只是 *filename* 会作为第一个输入文件被链接，就好像在命令行上这个文件第一个被指定。对于入口地址总是第一个文件的开始位置的某些系统中，这可能很有用。

3.4.3 处理对象(object)文件格式的命令

有几个链接器脚本命令用来处理对象文件格式。

OUTPUT_FORMAT(*bfdname*)

OUTPUT_FORMAT(*default*, *big*, *little*)

OUTPUT_FORMAT 命令选择以哪种 BFD 格式来输出文件（参考 BFD）。使用 OUTPUT_FORMAT(*bfdname*)和在命令行上使用 '--oformat *bfdname*' 完全一致（参考命令行选项）。若都有，命令行选项优先。

你可以带有 3 个参数的使用 OUTPUT_FORMAT，来基于-EB 和-EL 命令行选项使用不同的格式。这允许链接器脚本根据需要的大小端设置输出格式。

如果'-EB'和'-EL'都没有使用，那输出格式会是第一个参数 *default*。如果使用了'-EB'，输出格式会是第二个参数 *big*，如果使用了'-EL'，输出格式会是第三个参数,*little*。

例如, MIPS ELF 平台默认的链接器脚本使用如下命令:

OUTPUT_FORMAT(elf32-bigmips, elf32-bigmips, elf32-littlemips)

这表示缺省的输出文件格式是'elf32-bigmips'，但是当用户使用'-EL'命令行选项，输出文件就会被以`elf32-littlemips'格式创建。

TARGET(*bfdname*)

TARGET 命令设定在读取输入文件时所选择的 BFD 格式名。它会影响到后来的'INPUT'和'GROUP'命令。这个命令和在命令行上的`-b BFDNAME'相似（参考命令行选项）。 如果使用了'TARGET'命令但没指定`OUTPUT_FORMAT'，那么最后的'TARGET'命令也被用来设置输出文件的格式。参考 BFD。

3.4.4 为存储区指定别名

可以为 MEMORY 命令创建的存储区添加一个别名。每个名称最多对应一个区域。

REGION_ALIAS(*alias*, *region*)

REGION_ALIAS 函数为 *region* 存储区起了 *alias* 的别名。这允许对输出分区进行复杂的存储区域映射。下边是个例子：

设想我们有个嵌入式应用程序，运行在三种不同的存储器配置的系统上。这三种系统都具有通用易失的 RAM 存储器，可用来执行代码及保存数据。有些系统中还有只读的非易失的 ROM 存储器，可用于执行代码和只读数据的访问。其中一个系统中还有 ROM2 存储器，是只读的非易失的，只能作为只读数据的访问，而不具备代码执行的能力。我们有 4 个输出分区：

- .text 程序代码；
- .rodata 只读数据；
- .data 可读可写有初始值的数据；
- .bss 可读可写初始化为 0 的数据。

现在，我们要提供一个链接命令文件，包括两部分：1.独立于系统的部分，定义了输出分区；2.系统相关部分，将输出分区映射到系统有效的存储区域。这三种不同存储器布置的嵌入式系统是 A、B 和 C：

分区	A	B	C
.text	RAM	ROM	ROM
.rodata	RAM	ROM	ROM2
.data	RAM	RAM/ROM	RAM/ROM2
.bss	RAM	RAM	RAM

RAM/ROM 或 RAM/ROM2 的表示形式是指此分区被分别加载到 ROM 或 ROM2 中。要注意.data 分区的加载地址在这三种系统中都开始于.rodata 分区的尾部。

The base linker script that deals with the output sections follows. It includes the system dependent linkcmds.memory file that describes the memory layout:

基础的链接器脚本处理下面的输出分区。脚本中包含依赖系统的 linkcmds.memory。

```
INCLUDE linkcmds.memory

SECTIONS
  {
```

```
    .text :
      {
        *(.text)
      } > REGION_TEXT
    .rodata :
      {
        *(.rodata)
        rodata_end = .;
      } > REGION_RODATA
    .data : AT (rodata_end)
      {
        data_start = .;
        *(.data)
      } > REGION_DATA
    data_size = SIZEOF(.data);
    data_load_start = LOADADDR(.data);
    .bss :
      {
        *(.bss)
      } > REGION_BSS
  }
```

现在我们需要三种不同的 linkcmds.memory 文件来定义存储区域和别名。以下是 A，B，C 三种系统的 linkcmds.memory 文件中的内容：

A

在此系统中，一切都在 RAM 中。

```
MEMORY
  {
    RAM : ORIGIN = 0, LENGTH = 4M
  }

REGION_ALIAS("REGION_TEXT", RAM);
REGION_ALIAS("REGION_RODATA", RAM);
REGION_ALIAS("REGION_DATA", RAM);
REGION_ALIAS("REGION_BSS", RAM);
```

B

程序代码和只读数据在 ROM 中。可读写数据放在 RAM 中。有初始值的数据镜像被加载到 ROM 中，系统开始运行时复制到 RAM 中。

```
MEMORY
```

```
    {
      ROM : ORIGIN = 0, LENGTH = 3M
      RAM : ORIGIN = 0x10000000, LENGTH = 1M
    }

  REGION_ALIAS("REGION_TEXT", ROM);
  REGION_ALIAS("REGION_RODATA", ROM);
  REGION_ALIAS("REGION_DATA", RAM);
  REGION_ALIAS("REGION_BSS", RAM);
```

C

程序代码在 ROM 中。只读数据在 ROM2 中。可读写数据在 RAM 中。有初始值的数据镜像被加载到 ROM2 中，系统开始运行时复制到 RAM 中。

```
  MEMORY
    {
      ROM : ORIGIN = 0, LENGTH = 2M
      ROM2 : ORIGIN = 0x10000000, LENGTH = 1M
      RAM : ORIGIN = 0x20000000, LENGTH = 1M
    }

  REGION_ALIAS("REGION_TEXT", ROM);
  REGION_ALIAS("REGION_RODATA", ROM2);
  REGION_ALIAS("REGION_DATA", RAM);
  REGION_ALIAS("REGION_BSS", RAM);
```

如果需要，可以写个通用的系统初始化程序来完成从 ROM 或 ROM2 中复制.data 分区中的数据到 RAM 中：

```
#include <string.h>

extern char data_start [];
extern char data_size [];
extern char data_load_start [];

void copy_data(void)
{
  if (data_start != data_load_start)
    {
      memcpy(data_start, data_load_start, (size_t) data_size);
    }
}
```

3.4.5 杂项命令

还有几个其它的链接器脚本命令。

ASSERT(*exp*, *message*)

确保 *exp* 非 0。如果是 0，那么退出链接，返回一个错误码，打印 *message*。

注意，断言会在链接最后阶段发生前检查。这意味着如果用户没为符号赋值，而在分区定义中这些符号被定义在 PROVIDE 中，含有这些符号的表达式就会失败。这一规则唯一的例外是 PROVIDE 的符号中正好引用了点符号(.)。因此这样的断言：

```
.stack :
{
   PROVIDE (__stack = .);
   PROVIDE (__stack_size = 0x100);
   ASSERT ((__stack > (_end + __stack_size)), "错误：给栈留下的空间没有了");
}
```

如果__stack_size 在任何其它地方都没有定义，就会失败。在分区定义之外的包含在 PROVIDE 中的符号会更早地进行计算，所以它们可以在 ASSERT 中使用。因此：

```
PROVIDE (__stack_size = 0x100);
.stack :
{
   PROVIDE (__stack = .);
   ASSERT ((__stack > (_end + __stack_size)), "错误：给栈留下的空间没有了");
}
```

就能正常工作。

EXTERN(*symbol symbol* ...)

强制使 *symbol* 进入输出文件后成为未定义的符号。这样，可能会将标准库中的模块链接进来。你可以在一个 EXTERN 中指定若干个 *symbol*，你也可以多次使用 EXTERN。此命令与-u 命令行选项有相同功效。

FORCE_COMMON_ALLOCATION

此命令与-d 命令行选项有相同功效：会使 ld 为公共符号分配空间，即使使用-r 指定了输出文件为可重定位的。

INHIBIT_COMMON_ALLOCATION

此命令与--no-define-common 命令行选项有相同功效：使 ld 忽略为公共符号所指定的地址，即使是个不可重定位的输出文件。

FORCE_GROUP_ALLOCATION

此命令与--force-group-allocation 命令行选项有相同功效：使 ld 像普通的输入分区一样来放置分区组成员，删除分区组，即使用-r 指定为可重定位的输出文件。

INSERT [AFTER | BEFORE] *output_section*

此命令通常用在'-T'指定的脚本中，以便用例如重叠的方法，扩展默认的分区。它会

把所有的之前出现的链接器脚本语句全部插入到 *output_section* 之前或之后 ,而且也使-T 选项不要覆盖默认的链接脚本。确切的插入位置是和孤立分区有关。参考孤立分区。插入操作会发生在链接器完成了由输入分区到输出分区的映射之后。在插入之前，由于-T 指定的脚本会在默认链接器脚本之前解析，在-T 脚本中的语句会在链接器处理默认脚本之前发生。尤其是在处理输入分区向输出分区对应时，-T 脚本会先于默认脚本。下面是个在-T 脚本中如何使用 INSERT 的可参考的例子：

```
SECTIONS
{
  OVERLAY :
  {
    .ov1 { ov1*(.text) }
    .ov2 { ov2*(.text) }
  }
}
INSERT AFTER .text;
```

NOCROSSREFS(*section section* ...)

此命令可用来告诉 ld 当在指定的输出分区中有任何交叉引用时，就引起一个错误。在一些类型的程序中，尤其是嵌入式系统，当使用重叠，当一个分区被加载到存储中，而另一个分区不会被加载。任何在两个分区间的直接引用都会发生错误（例如，如果在一个分区中的代码调用另一个分区的函数）。

NOCROSSREFS 命令包含一个输出分区名称列表。如果 ld 检测到之间的分区有交叉引用，会报告一个错误，并且返回非 0 的退出状态。注意，NOCROSSREFS 命令使用输出分区名，而不是输入分区名。

NOCROSSREFS_TO(*tosection fromsection* ...)

此命令用来告诉 ld 当分区列表中任一个引用了指定分区中符号时，就发起错误。

NOCROSSREFS 命令当确保两个或更多输出分区完全独立时很有用 ,而有时单向依赖还是需要的。例如：在一个多核应用中，可能有需要被每个核都调用的共享代码，但为了安全，决不允许回调。

NOCROSSREFS_TO 命令维护一个输出分区名列表。第一个分区不能被其它分区引用。如果 ld 检测到任何其它分区向第一个分区的引用时，会报告错误，并且返回非 0 的退出值。注意，NOCROSSREFS_TO 命令使用输出分区名，不是输入分区。

OUTPUT_ARCH(*bfdarch*)

指定一个特定的输出机器架构。参数是 BFD 库中的一个名字（参考 BFD）。你可以使用 objdump 程序带上-f 选项查看一个 object 文件的机器架构。

LD_FEATURE(*string*)

此命令是用来修改 ld 行为的。如果 *string* 是"SANE_EXPR" ，那么在脚本中的绝对符号和数字在所有地方都将会简单地当作数字。参考表达式分区。

3.5 符号赋值

你可以为链接器脚本中的符号赋值。这会定义此符号并且将其放进全局域的符号表。

3.5.1 简单赋值

你可以使用任何一种 C 赋值符给符号赋值：

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

第一种情况会为 *symbol* 赋值为 *expression*。而其它情况，*symbol* 必须已经定义了，它最终的值会依据原值产生。

‘.’这个特别的符号指示当前的位置计数器。你可以在 SECTIONS 命令中只使用这个。参考位置计数器。

在 *expression* 后面的分号是必需的。

后面会定义表达式；参考表达式。

你可以只写符号赋值命令，或者在 SECTIONS 命令中作为语句，或者在 SECTIONS 命令中作为输出分区描述的一部分。

符号的分区会被设置成表达式的分区；更多信息，请参考表达式分区。

下面是个例子，展示符号赋值三种不同位置：

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

在这个例子中，‘floating_point’将被定义成 0.符号’_etext’将被定义成最后的’.text’输入分区随后的地址。’ _bdata’符号将被定义成‘.text’输出分区随后的地

址向上 4 字节对齐后的值。

3.5.2 HIDDEN

对于 ELF 目标接口，定义一个隐藏的符号而不被导出。语法如下：
HIDDEN(*symbol* = *expression*).
下面是个使用 HIDDEN 改写的简单赋值中的例子：

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

这三个符号在此模块之外都不可见。

3.5.3 PROVIDE

某些情况下，链接器脚本很想实现这样的功能，只有没被链接中引入的任何 object 文件定义且符号被引用时，符号定义才生效。例如，传统链接器定义了 etext 符号，而 ANSI C 需要让用户可以把 etext 当作一个函数名使用而不报错。PROVIDE 关键字就可以用来定义符号，就像 etext，只有它被引用且未定义才生效。语法：PROVIDE(*symbol* = *expression*)。

这里是一个使用 PROVIDE 定义 etext 的例子：

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

在这个例子中，如果程序定义‘_etext’（以下划线开头），链接器会提示多次定义的错误。而如果程序定义’etext’(不以下划线开头)，链接器会默默地使用程序中的定义。如果引用了’etext’，但并未定义它，链接器就会使用在链接器脚本中的定义。

3.5.4 PROVIDE_HIDDEN

与 PROVIDE 类似。对于 ELF 目标接口，符号会被隐藏，不会被导出。

3.5.5 源代码引用

从源代码访问链接器脚本定义变量不直观。特别是链接器脚本符号与高级语言的变量声明并不等价，而只是一个没有值的符号。

在继续介绍之前，有个重要的东西要注意，编译器经常在保存符号表时将源代码中的名称转换成不同的名称。例如，Fortran 编译器一般会添加下划线，而 C++会进行另外的名称重组。因此在代码中的变量和在链接器脚本中定义的同名符号之间可能会出现不符。例如在 C 语言中，链接器脚本可能引用的变量是这样的：

```
extern int foo;
```

但在链接器脚本中，它会是这样定义的：

```
_foo = 1000;
```

然而，在接下来的例子中，我们假设不发生名称转换。

当在高级语言（例如 C）中声明一个符号时，会发生两件事。第一，编译器会在程序存储器中预留足够的空间来保持符号的值。第二，编译器会在程序的符号表中创建一个条目，条目中要保持符号的地址。也就是说，符号表包含符号的地址，而存储器保持着符号的值。所以，举例说明，下面的 C 语言声明，在此文件作用域内：

```
int foo = 1000;
```

在符号表中创建一个叫 foo 的条目。这个条目保存着一个 int 大小的一块存储的地址，而其中放着一个初始化的数值 1000。

当程序引用符号时，编译器生成的代码首先访问符号表找到符号的存储块地址，然后代码从存储器块中读出值。所以：

```
foo = 1;
```

在符号表中找到 foo，获得它相关的地址，然后写 1 到这个地址里。然而：

```
int * a = & foo;
```

在符号表中找到 foo，获得它的地址，然后将此地址复制到变量 a 相关的存储块中。

相对的，链接器脚本符号声明在符号表中创建一条，但并不为它分配任何存储。因此这些符号只是个地址而没有值。所以举个例子，链接器脚本定义：

```
foo = 1000;
```

在符号表中创建一个叫’foo’的条目，它保存着一个 1000 的地址，但是在 1000 个这地址中什么都没保存。这就意味着你不能访问链接器脚本定义的符号的值，因为它根本没有值，你唯一能做的就是访问链接器脚本定义的符号的地址。

因此当你在源代码中使用链接器脚本定义的符号时，你应该总是使用符号的地址，而不要试图使用其值。例如，设想你想要复制名叫.ROM 的内存区域的内容到.FLASH 分区中，并且链接器脚本包含了这样的声明：

```
start_of_ROM     = .ROM;
end_of_ROM       = .ROM + sizeof (.ROM);
start_of_FLASH   = .FLASH;
```

然后，C 源码这样执行复制操作：

```
extern char start_of_ROM, end_of_ROM, start_of_FLASH;
memcpy (& start_of_FLASH, & start_of_ROM, & end_of_ROM - & start_of_ROM);
```

注意这里所使用的‘&’操作符。这是正确的。也可以将符号看成是向量或数组的名称，代码也能正确工作：

```
extern char start_of_ROM[], end_of_ROM[], start_of_FLASH[];
memcpy (start_of_FLASH, start_of_ROM, end_of_ROM - start_of_ROM);
```

注意，使用此方法不需要使用‘&’操作符。

3.6分区命令

SECTIONS 命令告诉链接器如何将输入分区映射到输出分区，如何在存储器中布局。

SECTIONS 命令的格式如下：

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

每个 *sections-command* 可以是以下形式的一种：

- 一个 ENTRY 命令（参考设置入口点）
- 一个符号赋值（参考为符号赋值）
- 一个输出分区描述
- 一个重叠描述

ENTRY 命令和符号赋值在 SECTIONS 命令中是允许的，这是由于在这里使用位置计数器比较方便。这也同样可以使链接器脚本更容易理解，因为你可以在输出文件布局的有意义的点上使用这样命令。

输出分区描述和重叠描述会在下面进行说明。

如果你不在链接器脚本中使用 SECTIONS 命令，链接器会按出现的顺序为每个输入分区创建一个不同名称的输出分区。例如：如果所有输入分区都在第一个文件中，那么输出文件中分区的顺序将会匹配首个输入文件中的顺序。第一个分区的地址是 0。

3.6.1 输出分区的描述

对输出分区的完整描述会像这样：

section [*address*] [(*type*)]：

[AT(*lma*)]
[ALIGN(*section_align*) | ALIGN_WITH_INPUT]
[SUBALIGN(*subsection_align*)]
[*constraint*]
{
output-section-command
output-section-command
...
} [> *region*] [AT> *lma_region*] [:*phdr* :*phdr* ...] [= *fillexp*] [,]

大多数输出分区不会使用到这么多的可选属性。

在 *section* 周围的空格是必须的，所以分区名称是清晰的。冒号和大括号也是必需的。如果使用了 fillexp 而且下一条分区命令看起来像一个表达式的附加部分，那么结尾的逗号也是必需的。换行以及其它的空格是可选的。

每条 *output-section-command* 会是以下命令中的一个：

- 一条符号赋值（参考符号赋值）
- 一条输入分区描述（参考输入分区）
- 直接包含的数据值（参考输出分区数据）
- 一个特殊的输出分区关键字（参考输出分区关键字）

3.6.2 输出分区名称

输出分区的名称是 *section*。*section* 必须满足你的输出格式的限制。在一些格式中只支持有限数量的分区，例如 a.out，分区名必须是被支持的名称中的一个（例如 a.out，只允许 ‘.text’， ‘.data’ 或者 ‘.bss’ ）。如果输出格式支持任意数量的分区名，使用数字而不是名字（例如 Oasys），那分区名就应该是双引号将数字包含起来的字符串。分区名可以由任意字符序列组成，但是如果名字中包含特殊字符（例如逗号等）必须用双引号。

输出分区名 ‘/DISCARD/’ 比较特殊；参考输出分区丢弃。

3.6.3 输出分区地址

Address 是个输出分区的 VMA（虚拟存储地址）表达式。此地址可选，但如果提供的话，那输出地址将会按指定的位置准确放置。

如果输出地址没有指定，那会基于以下推算方法为其选择一个。此地址会被调整至满足此输出分区的对齐需求。对齐需求是输出分区中包含的输入分区的最严格的对齐。

输出分区地址的推算方法如下：

- 如果为此分区设置了输出存储 *region*，那么此区会被添加进这个存储 region，而且此分区的起始地址就是 region 中下一个空闲的地址。
- 如果使用了 MEMORY 命令来创建一个存储区域列表，那么首个区域具有与此分区兼容的属性的就会被选择，且包含此分区。分区的输出地址就是这个 region 中下一

个空闲地址；参考 存储命令。

- 如果没有指定存储区域，而且也没有匹配分区的存储区域，那么输出地址会基于位置计数器的当前值。

例如：

.text . : { *(.text) } 和 .text : { *(.text) } 是有细微差别的。第一个会设置’.text’输出分区的地址为位置计数器的当前值。第二个会设置成位置计数器当前值对齐后的值，对齐成所有’.text’输出分区中最严格的对齐。

Address 也可以是个任意的表达式；参考 表达式。例如，如果你想将此分区 0x10 字节对齐，也就是此分区地址的低 4 位是 0，你可以这样定义：

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

这能达到效果，因为 ALIGN 会返回位置计数器当前值向上对齐指定值后的地址。

为分区指定 *address* 将会改变位置计数器的值，当然分区是非空才行。空区被忽略。

3.6.4 输入分区的描述

最常见的输出分区命令就是一个输入分区描述。

输入分区描述是最基本的链接器脚本操作。使用输出分区的描述来告诉链接器如何在存储器中布局。使用输入分区的描述来告诉链接器如何映射输入文件到存储器布局中。

3.6.4.1 输入分区基础

一个输入分区描述由一个文件名和可选地跟随一个分区名列表（可有通配符）组成。

文件名和分区名都可以是个含通配符的模板，会在后边介绍（输入分区通配符）。

最常见的输入分区描述就是在输出分区中使用特定的名称包含所有输入分区。例如，想要包含所有‘.text’输入分区，你要这样写：

*(.text)

这里的’*’是个通配符，它会匹配所有文件名。要从通配符匹配的文件中排除某此文件，你可以使用 EXCLUDE_FILE。例子：

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

会导致除 crtend.o 和 otherfile.o 之外所有文件中的.ctors 分区补包含。EXCLUDE_FILE 也可放置在分区列表中，例如：

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

结果和上个例子完全一样。支持 EXCLUDE_FILE 的两种语法，在分区列表包含超过一个分区时很有用，参考下面介绍的情况。

有两种方法引入两个以上的分区：

*(.text .rdata)

*(.text) *(.rdata)

这两个方法的差别是在输出分区中‘.text’和‘.rdata’出现的顺序。第一种方法，它们会混合在一起，和链接器在输入时发现它们的顺序一样。第二种方法，所有.’text’输

入分区会先出现，然后才是’.rdata’分区。

当使用 EXCLUDE_FILE 和两个以上分区时，如果在分区列表中出现，那只应用到紧跟其后的分区。例如：

*(EXCLUDE_FILE (*somefile.o) .text .rdata)

将会使除 somefile.o 之外的所有文件中的‘.text’分区包含进来，而所有文件中的‘.rdata’分区，包括 somefile.o 中的，都会被包含。要排除 somefile.o 中的‘.rdata’分区，那么例子要被修改成：

*(EXCLUDE_FILE (*somefile.o) .text EXCLUDE_FILE (*somefile.o) .rdata)

作为选择，将 EXCLUDE_FILE 放在分区列表之外，在输入文件选择之前，就会使这个排除应用在所有的分区上。因此前一个例子可以这样写：

EXCLUDE_FILE (*somefile.o) *(.text .rdata)

你可以通过指定一个文件名，来导入其中的分区。当你需要将一个或多个包含特殊数据的文件放在存储器中特别的位置时，可以用此方法。例如：

data.o(.data)

根据输入分区的标志，将精选后的分区包含，可以使用 INPUT_SECTION_FLAGS。

这里有个简单的例子，使用 ELF 分区的分区头标志：

```
SECTIONS {
  .text : { INPUT_SECTION_FLAGS (SHF_MERGE & SHF_STRINGS) *(.text) }
  .text2 :  { INPUT_SECTION_FLAGS (!SHF_WRITE) *(.text) }
}
```

这个例子中，输出分区‘.text’会由匹配*(.text)且分区头标志中 SHF_MERGE 和 SHF_STRINGS 被设置的所有输入分区组成。输出分区‘.text2’会由所有匹配*(.text)且分区头标志中 SHF_WRITE 没被设置的所有输入分区组成。

你也可以对档案文件使用一个模板匹配其中的文件，用一个冒号，冒号两侧没有空格。

‘archive:file’

在 archive 中匹配 file。

‘archive:’

匹配整个 archive。

‘:file’

匹配一个不是档案文件中的文件。

‘archive’和‘file’都可以包含 shell 通配符。在基于 DOS 的文件系统中，链接器会假设一个字母后边有个冒号是个驱动器的说明符，所以‘c:myfile.o’是个简单的文件标识，而不是在 c 档案文件中的‘myfile.o’文件。‘archive:file’这种文件标识也可以在 EXCLUDE_FILE 列表中使用，但是不要在其它的链接器脚本上下文中出现。例如，你不能在 INPUT 命令中通过使用‘archive:file’从一个档案文件中提取文件。

如果你使用一个文件名而没有分区列表，那么在这个输入文件中的所有分区都将被包含进当前的输出分区中。这不常用的，但有时很有用。例如：

data.o

当你使用一个文件名时，不是像‘archive:file’形式的，而且不包含任何通配符，链接器会首先判断你是否也在链接器命令行指定了此文件名，或者在 INPUT 命令中指定了。如果你没指定 ,链接器会尝试将此文件当成一个输入文件打开 ,好像它出现在了命令行一样。注意：这个与 INPUT 命令不同，因为链接器不会从档案搜索路径中搜索此文件。

3.6.4.2 输入分区通配符模板

在一个输入分区描述中，文件名和分区名中的一个或者两者都是通配符模板。

在很多例子中出现的‘*’是个简单的文件名通配符模板。

通配符模板就像在 Unix Shell 中使用的一样。

‘*’

匹配任何个数的字符。

‘?’

匹配任何单个字符。

‘[*chars*]’

匹配 *chars* 中的任何一个字符，‘-’可用来表示一个范围的字符，例如‘[a-z]’匹配所有小写字母。

‘\’

引用后边的那个字符。

当文件名与通配符匹配时，通配符字符将不匹配“/”字符（在 Unix 系统中，用来分隔目录名）。模板只包含一个‘*’时例外，这会匹配所有文件名，不管文件名中是否包括‘/’字符。在分区名， 通配字符会匹配‘/’字符。

文件名的通配模板只匹配那些在命令行或使用 INPUT 命令指定的文件。链接器不会在目录中去匹配。

如果一个文件匹配两个以上通配模板 ,或者如果文件名明确给出后又匹配了另一个通配模板，链接器会在链接器脚本中使用第一个匹配的。例如，这个输入分区描述序列可能包含错误，因为 data.o 这个规则根本不会被使用：

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

一般来说，链接器会以链接器匹配的顺序来放置文件和分区。将 SORT_BY_NAME 关键字放在通配模板前，再加括号，就能更改默认顺序（例如：SORT_BY_NAME(.text*)）。当使用了 SORT_BY_NAME 关键字，链接器会在放置文件和分区到输出文件前，按名称进行升序排列。

SORT_BY_ALIGNMENT 很像 SORT_BY_NAME。区别是 SORT_BY_ALIGNMENT 会按对齐的降序放置分区到输出文件中。较大对齐的会放在较小对齐的之前，这样是为了减少需要填充的存储空间的数量。

SORT_BY_INIT_PRIORITY 很像 SORT_BY_NAME。区别是 SORT_BY_INIT_PRIORITY 会按照在分区名中编码的 GCC init_priority 数值属性的升序排序，然后放到输出文件中。

SORT 是 SORT_BY_NAME 的一个别名。

当在链接器脚本中有嵌套的分区排序命令时，只能最多嵌套一层。

1. SORT_BY_NAME (SORT_BY_ALIGNMENT (通配符分区模板))。先将输入分区按名称排序，如果两个分区有同样的名称，再按对齐排序。
2. SORT_BY_ALIGNMENT (SORT_BY_NAME (通配符分区模板))。先将输入分区按对齐排序，如果两个分区有相同的对齐，再按名称排序。
3. SORT_BY_NAME (SORT_BY_NAME (通配符分区模板))和 SORT_BY_NAME (通配符分区模板)是一样的。
4. SORT_BY_ALIGNMENT (SORT_BY_ALIGNMENT (通配符分区模板))和 SORT_BY_ALIGNMENT (通配符分区模板)是一样的。
5. 其它所有的嵌套的排序命令都是无效的。

当同时使用了命令行分区排序选项和链接器脚本分区排序命令，分区排序命令会优先于命令行选项。

如果链接器脚本中分区排序命令未嵌套，命令行选项会和它结合形成嵌套排序命令。

1. SORT_BY_NAME (通配符分区模板) 结合 --sort-sections alignment 与 SORT_BY_NAME (SORT_BY_ALIGNMENT (通配符分区模板))等效。
2. SORT_BY_ALIGNMENT (通配符分区模板) 结合 --sort-sections name 与 SORT_BY_ALIGNMENT (SORT_BY_NAME (通配符分区模板))等效。

如果在链接器脚本中分区排序命令有嵌套，那么命令行排序选项会被忽略。

SORT_NONE 用于忽略命令行分区排序选项，以禁止分区排序。

如果你对输入分区会放到哪儿有困惑，使用‘-M’链接器选项生成链接图文件。此文件会精确地展示出输入分区是如何放置到输出分区中的。

下面的例子说明通配符模板在匹配文件时应如何使用。此脚本指挥链接器将所有‘.text’分区放进‘.text’，所有‘.bss’放进‘.bss’。链接器会将所有以大写字母开头的文件中的‘.data’分区放进‘.DATA’;而将其它所有文件中的‘.data’分区放入‘.data’。

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

3.6.4.3 公共符号的输入分区

对于公共符号有个特殊的记法，因为在很多 object 文件格式中，没有为公共符号指定专门的输入分区。链接器就好像公共符号在名叫‘COMMON’的输入分区中一样处理它们。

你可以使用带有‘COMMON’分区的文件，就像带有其它输入分区的文件一样。你可以使用这种办法将特定输入文件中的公共符号放到一个分区中,而将其它输入文件中的公共符号放置在另一个分区中。

在大多数情况下，输入文件中的公共符号会被放在输出文件中的‘.bss’分区。例如：.bss { *(.bss) *(COMMON) }

有些 object 文件格式有两个以上种类的公共符号。例如，MIPS ELF object 文件格式区分标准公共符号和小公共符号。这种情况下，链接器会为特殊的公共符号使用特殊的分区名称。在 MIPS ELF 中，链接器对标准公共符号使用‘COMMON’，而小公共分区使用’.scommon’。利用此特性，我们可以将不同类型的公共符号放到不同的位置。

你有时会在旧版本的链接器脚本中看到‘[COMMON]’。这种记法现在已经废弃了。它等效于‘*(COMMON)’。

3.6.4.4 输入分区和垃圾收集

当使用了命令行选项‘--gc-sections’，链接时会进行垃圾收集，经常需要给不要被淘汰的分区做标记。使用 KEEP()将输入分区的通配条目包括起来，例如 KEEP(*(.init)) 或者 KEEP(SORT_BY_NAME(*)(.ctors))。

3.6.4.5 输入分区的例子

下面的例子是个完整的链接器脚本。它告诉链接器从 all.o 文件中读取所有分区然后放置到输出分区’outputa’的起始位置，而此分区在’0x10000’地址。Foo.o 文件中’.input1’分区中的所有内容紧跟其后，放在同一个分区中。Foo.o 文件中’.input2’分区中的所有内容放到’outputb’分区中，此分区紧跟在 foo1.o 文件的’.input1’分区之后。’.input1’和’.input2’在其它文件中的剩余部分写入’outputc’分区中。

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

如果输入分区与输出分区有相同的名字，而且这个名字又是 C 语言中某个符号的名字，

链接器会自动生成两个符号__start_SECNAME 和 __stop_SECNAME，其中的 SECNAME 就是分区的名称。这两个符号分别指示着输出分区的开始和结束地址。注意：大部分的分区名都不会和 C 语言中的符号相同，因为分区名包含一个 '.' 字符。

3.6.5 输出分区数据

可以使用 BYTE, SHORT, LONG, QUAD, SQUAD 就像输出分区命令一样在输出分区中明确地插入一些字节的数据。每个关键字后面都有一个用小括号括起来的表达式，指定要插入的值（参考 表达式）。表达式的值会被放在位置计数器当前值的位置上。

BYTE, SHORT, LONG, QUAD, SQUAD 命令分别保存 1，2，4，8 个字节。保存这些字节后，位置计数器增加相应的字节值。

例如：以下例子中会保存一个字节，值为 1，然后保存 4 字节，值为符号 addr。

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

当在 64 位主机或目标上时，QUAD 和 SQUAD 是相同的；都会保存 8 字节，或者说 64 位的值。当主机和目标都是 32 位时，表达式会以 32 位计算。这种情况下，QUAD 保存一个用 0 扩展 32 位数的 64 位值，而 SQUAD 保存一个用符号扩展 32 位数的 64 位值。

如果输出文件的 object 文件格式中有显性的大小端定义，这是一般情况，数值就会按此定义进行保存。当 object 文件格式没有定义大小端，例如 S-records，这些数值会以首个输入 object 文件的大小端格式存储。

注意：这些命令只在分区描述内工作，而不是在分区之间。因此以下定义会导致错误：

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

然而，下面的能工作：

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

你可以使用 FILL 命令为当前分区设置设置填充。此命令后跟随一个小括号括起来的表达式。在此分区中任何未指定的存储区域（例如，为满足输入分区所需的对齐而产生的间隔）都被此表达式的值填充。FILL 语句会影响在分区定义中它出现的点之后的存储区域；通过使用两个以上 FILL 语句，可以为一个输出分区的不同部分指定不同的填充内容。

以下例子展示了如何将未指定存储区域填充为 '0x90' ：

```
FILL(0x90909090)
```

FILL 命令和 '=*fillexp*' 输出分区属性很类似，但是 FILL 命令只影响分区中它后的部分，而不是整个分区。如果两者都使用了，FILL 命令优先。参考 输出分区的填充。

3.6.6 输出分区关键字

这里是几个可以作为输出分区命令的关键字。

CREATE_OBJECT_SYMBOLS

此命令告诉链接器为每个输入文件创建符号。每个符号的名称对应输入文件名。在 CREATE_OBJECT_SYMBOLS 命令出现的地方 ,输出分区就是每个符号对应的分区。

这在 a.out object 文件格式中是惯例。而在其它 object 文件格式中并不常见。

CONSTRUCTORS

当使用 a.out object 文件格式链接时，链接器使用一个不寻常的集合结构来支持 C++全局构造函数和析构函数。当对不支持随机分区的 object 文件格式进行链接时，例如 ECOFF 和 XCOFF，链接器会通过名称自动识别 C++的全局构造函数和析构函数。对于这些 object 文件格式，CONSTRUCTORS 命令告诉链接器在输出分区中 CONSTRUCTORS 命令出现的位置放置构造函数信息。CONSTRUCTORS 命令对于其它 object 文件格式会被忽略。

__CTOR_LIST__符号标记全局构造函数的起始，__CTOR_END__标记结束。同样地，__DTOR_LIST__和 __DTOR_END__标记全局析构函数的起始和结束。列表中的首个 word 是条目的个数，后面是每个构造函数或析构函数的地址，以 0 值 word 结尾。编译器必须安排实际地运行这些代码。对于这些 object 文件格式，GNU C++一般会从子程序__main 中调用这些构造函数；对__main 的调用会自动在启动代码中插入。GNU C++一般会在 atexit 或 exit 函数中调用析构函数。

对于 COFF 或 ELF 这样的支持随机分区名称的 object 文件格式，GNU C++一般会将全局构造函数和析构函数的地址放进.ctors 和.dtors 分区中。将以下命令序列放到你的链接器脚本中，将会创建出 GNU C++ 运行时代码期待看到的那种表。

```
__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__ = .;
```

如果使用 GNU C++中对初始化优先级的支持，它提供了对全局构造函数运行顺序的一些控制，必须在链接时对构造函数进行排序，以确保它们按正确的顺序执行。当使用 CONSTRUCTORS 命令时，使用‘SORT_BY_NAME(CONSTRUCTORS)’来代替。当使用.ctors 和.dtors 分区时，使用‘*(SORT_BY_NAME(.ctors))’和‘*(SORT_BY_NAME(.dtors))’来代替‘*(.ctors)’ 和‘*(.dtors)’。

一般编译器和链接器会自动处理这些问题，也无需考虑。然而，如果你使用 C++而且编写自己的链接器脚本时，可能需要考虑这些。

3.6.7 输出分区的丢弃

链接器一般不会创建没有内容的输出分区。链接时会根据输入分区是否出现在输入文件中以及输入分区中是否有内容决定是否创建输出分区。例如：

.foo : { *(.foo) }

只有当至少有一个输入文件中包含‘.foo’分区，而且该输入分区中非空时，才会在输出文件中创建‘.foo’分区。其它的在输出分区中分配空间的链接脚本指令也会创建输出分区。尽管为点赋值并不分配空间，但是也能实现创建输出分区，不过以下这些赋值并不会创建分区‘. = 0’、‘.= .+0’、‘.=sym’,’ .=.+sym‘、’ .=ALIGN(.!=0,expr, 1)’(其中的 sym 是在脚本中定义的值为 0 的符号)。你可以使用‘. = .’强制创建空的输出分区。

链接器会忽略被丢弃分区的地址赋值（参考输出分区地址），除非链接器脚本在此输出分区中定义了符号。那种情况下，链接器会听从为分区进行的地址赋值，即使分区被丢弃了，位置计数器的值也很可能会推进一些。

‘/DISCARD/’这个特别的输出分区名称可用来丢弃输入分区。任何被指定到名叫‘/DISCARD/’的输出分区的输入分区都不会包含到输出文件中。

3.6.8 输出分区属性

对于输出分区的完整描述是这样的：

section [*address*] [(*type*)] :
  [AT(*lma*)]
  [ALIGN(*section_align*)]
  [SUBALIGN(*subsection_align*)]
  [*constraint*]
  {
    output-section-command
    output-section-command
    ...
  } [> *region*] [AT> *lma_region*] [:*phdr* :*phdr* ...] [= *fillexp*]

我们已经介绍了 *section*, *address*, 和 *output-section-command*. 在这一节，我们会介绍剩余的分区属性。

3.6.8.1 输出分区类型

每个输出分区都可以有个类型。类型是包含在小括号中的一个关键字。可有下面选择：

NOLOAD

分区会被标记为不加载，因此在程序运行时，不会被加载到存储器中。

DSECT

COPY

INFO

OVERLAY

这些类型名是为了向前兼容的，很少使用了。它们都有同样的效果：分区会标记为不可分配的，所以当程序运行时，不会为此分区分配存储空间。

链接器一般会根据被包含的输入分区设置输出分区的属性。你可以使用分区类型更改这

些属性。例如：在以下的脚本中，’ ROM’ 分区被放置在 0 地址的存储位置，而且程序运行时无需被加载。

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

3.6.8.2 输出分区的 LMA

每个分区都有虚拟地址(VMA)和加载地址(LMA)，参考基本脚本概念。虚拟地址在之前介绍的部分（输出分区地址）指定。加载地址可选，且由 *AT*或者 *AT>关键字* 来指定的。

AT 关键字以一个表达式作为参数，指定分区的准确加载地址。AT>关键字以一个存储区域名作为参数。参考 存储命令 。分区的加载地址会被设置成该区域的下个空闲的地址，当然此地址会按对齐要求进行对齐。

如果 AT 和 AT>都没被指定给可分配分区，链接器将使用以下推算方法决定加载地址：

- 如果为分区指定了 VMA 地址，那么使用 VMA 作为 LMA。
- 如果分区是不可分配的，那么它的 LMA 被设置成它的 VMA。
- 否则，如果能找到一个与当前分区兼容的存储区域，而且此区域中至少包含一个分区，那么 LMA 会被设置成这样：VMA 和 LMA 之间的差别与此区域中上个分区的 VMA 和 LMA 之间的差别完全相同。
- 如果没有声明存储区域，那么首先会使用一个默认的全地址覆盖的存储区域。
- 如果找不到合适的区域，或者没有上一个分区，那么 LMA 会被设置成 VMA。

此特性是为了方便创建 ROM 镜像而设计的。例如，下面的链接器脚本创建三个输出分区：一个是 ‘.text’ 从 0x1000 开始；另一个’ .mdata’ ，尽管它的 VMA 是 0x2000，也会加载到’ .text’ 的尾部，第三个’ .bss’ ，其中保存未初始化的数据，地址是 0x3000。_data 符号定义为 0x2000，这展示出位置计数器中是 VMA 的值，而不是 LMA。

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

与链接器脚本一起工作的运行时初始代码会包含以下代码，用来从 ROM 镜像中复制初始的值到运行时地址。注意这段代码如何利用链接器脚本定义的符号。

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

```
char *dst = &_data;

/* ROM 中在 text 后面放置着数据的初始值；复制 */
while (dst < &_edata)
  *dst++ = *src++;

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

3.6.8.3 强制输出对齐

你可以使用 ALIGN 来增加输出分区的对齐。可选的 ,你可以使用 ALIGN_WITH_INPUT 属性，强制使 VMA 和 LMA 之间的差距在整个输出分区中保持不变。

3.6.8.4 强制输入对齐

使用 SUBALIGN，可以在一个输出分区中强制输入分区的对齐。指定的值会覆盖输入分区中的对齐属性，无论大或者小。

3.6.8.5 输出分区约束

分别使用 ONLY_IF_RO 和 ONLY_IF_RW 关键字，你可以指定只有其中的所有输入分区都是只读或者可读写的才创建此输出分区。

3.6.8.6 为输出分区指定存储区域

使用’ *>region*’ 可以将一个分区指定到已经定义过的存储区域中。参考存储命令。这里是个简单的例子：

```
MEMORY { rom : ORIGIN = 0x1000, LENGTH = 0x1000 }
SECTIONS { ROM : { *(.text) } >rom }
```

3.6.8.7 输出分区 PHDR

PHDR 是 ELF 的 Program header 的缩写，又称为 Segment。通过使用’:phdr’你可以将一个分区指定到已经定义过的程序段中。参考 PHDRS。如果一个分区被指向了一个或多个段，那后面的所有可重分配分区也都会指定到这些段中，除非显性的使用了:phdr。你可以使用:NONE 告诉链接器不要将分区放到任何段中。这里是个简单的例子：

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

3.6.8.8 输出分区的填充

使用’ *=fillexp*’ 可以为整个分区设置填充的值。*Fillexp* 是个表达式（参考表达式）。在输出分区中任何未指定的存储区域（例如，由于输入分区对齐造成的空隙）都将被此值填充。如果填充表达式是个简单的 hex 数字，也就是一个以’ 0x’ 开头的 16 进制的字符串，而且尾部不是 k 或者 M，那么这个数字就被用来作为填充值；开头的 0 也是填充值的一部分。其它的所有情况，包括额外的括号以及一元+，填充模板都是表达式值最低 4 字节的值。任何时候，数字都是大端对齐的。

你也可在输出分区命令中使用 FILL 命令更改填充值。参考输出分区数据。简单例子：
SECTIONS { .text : { *(.text) } =0x90909090 }

3.6.9 重叠描述

重叠描述提供了一种简便方式，用来描述加载到镜像的不同部分但又在相同地址运行的那些分区。在运行时，某种重叠管理器会在需要时，从运行时存储器地址复制进或复制出，或许只是简单的操作寻址位。当存储器中某些区域比其它部分速度快时，此方法很有用。

使用 OVERLAY 命令描述重叠。在 SECTIONS 命令中使用 OVERLAY 命令，就像一个输出分区描述一样。OVERLAY 命令的完整语法：

```
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 OVERLAY 关键字外，其它都是可选的，每个分区必须有个名称（上例中是 secname1 和 secname2）。OVERLAY 结构中的分区定义和一般的 SECTIONS 结构（参考 分区命令 ）中的一样，只不过没有地址属性，也没有存储区域属性。

如果 *fill* 被使用，而且下一个 *sections-command* 看起来像是表达式的后续部分的话，末尾的逗号必须要有。

这些分区都被定义在了相同的起始地址。而这些分区的加载地址是不同的，这些分区中的内容在存储器上是连续的，以加载地址作为开始的一个整体提供给 OVERLAY。而一般分

区定义中，加载地址是可选的，默认就是开始的地址；开始地址也是可选的，默认是位置计数器的当前值。

当使用 NOCROSSREFS 关键字时，如果在这几个分区中有任何交叉引用，链接器会报告错误。这是因为这些分区都是在同一地址运行的，在运行时当前分区根本无法引用另一个分区。参考 NOCROSSREFS。

在 OVERLAY 中的每个分区，链接器都自动提供两个符号。 __load_start_*secname* 用来表示分区加载地址的开始。 __load_stop_*secname* 表示分区加载地址的结束。在 *secname* 中的不满足 C 语言名称的字符都会被去除。C 代码（或汇编）会使用这些符号，在需要时切换重叠的这些分区。

在重叠区域的末尾，位置计数器的值会变成重叠区开始地址加上最大分区的尺寸。

以下是个例子。请记住，这应该出现在 SECTIONS 结构中。

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

这会将‘.text0’和‘.text1’的起始地址都定义为 0x1000。‘.text0’会被加载在 0x4000 地址，‘.text1’会被加载在‘.text0’之后。以下的符号可以被引用：__load_start_text0，__load_stop_text0， __load_start_text1， __load_stop_text1。

在 C 语言中复制.text1 到重叠区域的代码如下：

```
extern char __load_start_text1, __load_stop_text1;
memcpy ((char *) 0x1000, &__load_start_text1,
            &__load_stop_text1 - &__load_start_text1);
```

请注意，OVERLAY 命令其实是个糖衣语法，因为它所做的都可以通过更基本的命令完成。上面的例子也可写成：

```
.text0 0x1000 : AT (0x4000) { o1/*.o(.text) }
PROVIDE (__load_start_text0 = LOADADDR (.text0));
PROVIDE (__load_stop_text0 = LOADADDR (.text0) + SIZEOF (.text0));
.text1 0x1000 : AT (0x4000 + SIZEOF (.text0)) { o2/*.o(.text) }
PROVIDE (__load_start_text1 = LOADADDR (.text1));
PROVIDE (__load_stop_text1 = LOADADDR (.text1) + SIZEOF (.text1));
. = 0x1000 + MAX (SIZEOF (.text0), SIZEOF (.text1));
```

3.7存储命令

链接器默认配置成允许对所有有效存储进行分配，MEMORY 命令可以改变这一行为。

MEMORY 命令描述在目标中的存储器的定位及大小。你可以用它来描述有哪些存储区域可以由链接器使用，以及哪此存储区域要避免使用。然后就可以将分区指定到特定的存储区域中。链接器会基于存储区域为分区设置地址，如果存储区域太满时会警告。链接器不会

因合适的大小放到有效存储区域而打乱分区的顺序或裁剪分区等。

链接器脚本中可能包含多个 MEMORY 命令，然而，定义的所有存储块都会像在一个 MEMORY 命令中指定的一样。MEMORY 语法如下：

MEMORY

{

name [(*attr*)] : ORIGIN = *origin*, LENGTH = *len*

…

}

Name 是链接器用来引用这个区域的名称。区域名在链接器脚本之外无意义。区域名保存在一个隔离的名称空间中，不会和符号名、文件名以及分区名混淆。每个存储区域都要在 MEMOEY 命令中有唯一的名称。然而，你可以使用 REGION_ALIAS 命令为已经存在的存储区域指定别名。

Attr 用来指定是否为没有在链接器脚本中明确映射的输入分区使用一个特别的存储区域。就像在 SECTIONS 中描述的，如果你不为一些输入分区指定输出分区的话，链接器会创建一个和输入分区相同名称的输出分区。如果你定义了区域属性，链接器会用这些区域来为它创建的输出分区选择存储区域。

Attr 字符串只能由以下字符组成：

‘R’

只读分区

‘W’

可读可写分区

‘X’

可执行分区

‘A’

可分配分区

‘I’

初始化的分区

‘L’

和 ‘I’ 一样

‘!’

对后边属性的含义取反。

如果未指定存储区域的分区匹配任何一条列出的属性，而不是’ !’ ,它会被放在此存储区域内。 ‘!’ 属性的含义正好相反，因此未映射的分区只有在不满足所有列出的属性时才会被纳入此存储区域。

Origin 是个数字表达式，表示存储区域的首地址。此表达式必须是个常量，不能包含任何符号。ORIGIN 关键字可被缩写成 org 或者 o(但不是 ORG)。

len 是个表示存储区域以字节为单位的大小的表达式。和 *origin* 表达式一样，此表达式必须是数字常量。LENGTH 关键字可以被简写成 len 或者 l。

在下面的例子中，我们指定了两个有效的存储区域：一个以 0 开始 256KB，另一个以 0x40000000 开始 4MB。链接器会将每个没有指定存储区域的只读或可执行的分区放到’rom’中。链接器会将没指定存储区域的其它分区放进’ram’存储区域中。

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

一旦定义一个存储区域，你可以通过使用‘>*region*’输出分区属性告诉链接器将指定的输出分区放进此存储区域。例如，如果你有个名叫’mem’的存储区域，你在输出分区定义中使用’>mem’即可。参考 输出分区区域。如果没为输出分区指定地址，链接器会为其设置成此存储区域中下一个有效地址。如果指定到一个存储区域中的组合输出分区太大了，链接器会抛出错误。

通过 ORIGIN(*memory*)和 LENGTH(*memory*)函数可以在表达式中访问存储区域的位置及大小：

```
  _fstack = ORIGIN(ram) + LENGTH(ram) - 4;
```

3.8 PHDRS 命令

ELF object 文件格式使用 *program headers*，也称作 *segments*。这些程序报头描述了程序如何被加载到存储器中。你可以使用 objdump 程序加上-p 选项打印出程序报头信息。

在原生的 ELF 系统中运行 ELF 程序，系统加载器为了搞清如何加载程序会读取程序报头。只有程序报头被正确的设置了才会如此工作。此手册不描述系统加载器如何解析程序报头的细节；更多信息，请参考 ELF ABI。

链接器默认会创建合理的程序报头。然而，有些情况下，你可能需要生成的程序报头更加精确些，这时可以使用 PHDRS 命令。当链接器在链接器脚本中看到 PHDRS 命令时，只会为指定的创建程序报头。

链接器只在生成 ELF 输出文件时才会关心 PHDRS 命令。其它时候，直接忽略。

这是 PHDRS 命令的语法。PHDRS FILEHDR AT FLAGS 是关键字。

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

name 只用来给链接器脚本的 SECTIONS 命令引用，并不会放到输出文件中。程序报头名称会被保存在隔离的名称空间里，不会和符号名、文件名以及分区名混淆。每个程序报头都要有唯一的名字。报头按顺序处理，通常按照加载地址升序映射到各分区。

某些程序报头类型描述系统加载器将会从文件中加载的存储器段。在链接脚本中，通过

将可分配的输出分区放到段中，指定这些段中的内容。你使用’ *:phdr*‘ 输出分区属性将分区放到特定的段中。参考 输出分区 PHDR。

将某些分区放到一个以上的段中是正常的。这仅仅意味着一个存储器段包含另一个。可以重复’ *:phdr* ‘，对应该包含此分区的每个段使用一次。

如果你使用’ *:phdr* ‘将一个分区放到一个或多个段上，链接器会将随后的未指定’ *:phdr*‘ 的可分配分区放到相同的段中。这是为了方便，因为一般相邻的分区集会被放在一个段中。你可以使用:NONE 来重新定义默认的段，告诉链接器不要将分区放进段中。

你可以在程序报文类型后使用 FILEHDR 和 PHDRS 关键字来更多地描述段中的内容。FILEHDR 关键字意思是段中要包含 ELF 文件报头。PHDRS 关键字意思是段中要包含它自己的 ELF 程序报头。如果应用在一个可加载的段（PT_LOAD），所有前面的可加载段都必须包含有其中一个关键字。

type 必须是以下中的一个。数字表示关键字的值。

PT_NULL (0)

一个未使用的程序报头。

PT_LOAD (1)

此程序报头描述一个从文件中加载的段。

PT_DYNAMIC (2)

包含动态链接信息的段。

PT_INTERP (3)

包含程序解释器名称的段。

PT_NOTE (4)

保存辅助信息的段。

PT_SHLIB (5)

保留的程序报头类型，定义了，但未被 ELF ABI 指定。

PT_PHDR (6)

保存程序报头的段。

PT_TLS (7)

保存线程本地存储的段。

expression

一个数字的表达式，说明程序报头的类型。用于上面未定义的类型。

使用 AT 表达式，可以加载段到存储器中特定的地址。这和在输出分区属性中的 AT 命令（参考 输出分区 LMA）相同。程序报头的 AT 命令会覆盖输出分区属性。

链接器通常会根据组成段的分区来设置段标志。你可以使用 FLAGS 关键字来显性指定段标志。*flags* 的值必须是整型的。这用来设置程序报头中的 p_flags 域。

这里是个 PHDRS 的例子。展示了一个典型的在原生 ELF 系统中使用的程序报头集。

```
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) } /* 默认会到 :text 中 */
  …
  . = . + 0x1000; /* 移动到存储器的一个新页 */
  .data : { *(.data) } :data
  .dynamic : { *(.dynamic) } :data :dynamic
  …
}
```

3.9Version 命令

链接器在使用 ELF 时支持符号版本。符号版本只在使用共享库时有用。动态链接器在运行一个可能已经链接到旧版本共享库的程序时，使用符号版本选择函数的特定版本。

你可以要主链接器脚本中直接包含一个版本脚本，或者也可以用隐式链接器脚本的方式提供。你还可以使用‘--version-script’链接器选项。

VERSION 命令的语法非常简单：

VERSION { version-script-commands }

version-script-commands 的格式和在 Solaris2.5 中使用的 Sun 链接器中的一样。版本脚本定义了一个版本节点树。你在版本脚本中指定节点名和依赖关系。你可以指定哪些符号绑定到哪个版本节点，也可以缩减指定的符号集到局部域中，这样它们就在共享库之外不再全局可见了。

要演示版本脚本的最简单的方式要用到一些例子：

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

```
};

VERS_1.2 {
        foo2;
} VERS_1.1;

VERS_2.0 {
        bar1; bar2;
    extern "C++" {
        ns::*;
        "f(int, double)";
    };
} VERS_1.2;
```

例子中版本脚本定义了三个版本节点。第一个是‘VERS_1.1’，不依赖其它节点。脚本绑定了 foo1 符号到‘VERS_1.1’，它还缩减了一系列符号到局部域，这些符号从共享库之外不再可见，使用了通配符模板，所以只要名称以’old‘，’original‘或者’new’开头的符号被匹配。通配符模板的方法就像在 shell 中使用一样同样适用于文件名。然而，如果你使用双引号来指定符号名，那么名称会以文字方式对待，而不是通配模板。

第二个，版本脚本定义了‘VERS_1.2’节点。此节点依赖‘VERS_1.1’。脚本绑定符号 foo2 到‘VERS_1.2’版本节点。

最后，版本脚本定义‘VERS_2.0’节点。此节点依赖‘VERS_1.2’。又将 bar1 和 bar2 绑定到了‘VERS_2.0’节点。

当链接器在库中寻找一个没在版本节点中指定的符号时，它会将其绑定到此库的未指定基版本的版本节点上。你可以在版本脚本中使用‘global: *;’将未指定的符号都绑定到某个版本节点上。注意，除非在最后的版本节点上，否则在全局域中使用通配符稍显疯狂。如果不是最后的版本节点的话，全局匹配符有从旧版本添加符号到符号导出集里的风险。由于旧版本应该被一个修正的符号集，所以这样是错误的。

版本节点最好用人能读懂的方式命名。‘2.0’版本确实可以写在‘1.1’和‘1.2’之间，然而，这会为写版本脚本的人带来困惑。

倘若节点是版本脚本中唯一的版本节，那么节点名可被省略。这种版本脚本不会将任何符号指定到任何版本中，只是选择出哪个符号是全局可见的，哪些不是。

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

当与一个包含版本符号的共享库一起链接程序时，应用程序自己知道需要哪个版本的符号，而且也知道所有共享库中自己所需的版本节点。因此，在运行时，动态加载器可以进行快速检查来确保你正链接的那些库事实上能够支持应用程序所需的版本节点中的动态符号。以此方式，动态链接器就能很确定的知道所有外部符号都会被解析(找到符号的定义，并将定义与引用之间建立关联)，而且无需为每个符号引用进行搜索。

符号版本化事实上比 SunOS 在小版本的检查方式上更为复杂。这里要解决的基本问题

是，通常对外部函数的引用是根据需要绑定的，并且在应用程序启动时不是全部绑定的。如果共享库过时，则可能缺少所需的接口；当应用程序尝试使用该接口时，可能会突然出乎意料地失败。使用符号版本化，如果与应用程序一起使用的库太旧，则用户在启动程序时将收到警告。

有几种 GNU 扩展到 Sun 的版本控制方法。第一种是将符号绑定到源文件中的版本节点的能力，其中符号是在源文件中定义的，而不是在版本控制脚本中。这主要是为了减轻库维护器的负担。你可以这样做：

__asm__(".symver original_foo,foo@VERS_1.1");

在 C 源文件中，这会将函数“original_foo”重命名为绑定到版本节点“VERS_1.1”的“foo”的一个别名。可以使用“local:”指令来防止符号“original_foo”被导出。一个“.symver”指令优先于版本脚本。

第二个 GNU 扩展允许同一函数的多个版本出现在给定的共享库中。通过这种方式，可以在不增加共享库的主版本号的情况下对接口进行一次不兼容的更改，同时仍然允许链接到旧接口的应用程序继续工作。

要做到这一点，必须在源文件中使用多个'.symver '指令。下面是一个例子：

```
__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");
```

在本例中，'foo@'表示绑定到该符号的未指定基础版本的符号'foo'。包含这个示例的源文件将定义 4 个 C 函数：'original_foo'、'old_foo'、'old_foo1'和'new_foo'。

当给定符号有多个定义时，需要有某种方式指定该符号的外部引用要绑定的默认版本。你可以用“foo@VERS_2.0”类型的“.symver”指令来执行这个操作。以这种方式，只能将符号的一个版本声明为默认版本；否则，你将会一下了拥有一个符号的多个定义。

如果希望将引用绑定到共享库中符号的特定版本，可以使用方便的别名（即“old_foo”），或者可以使用“.symver”指令来具体绑定所讨论的函数的一个外部版本。

还可以在版本脚本中指定编程语言：

VERSION extern "lang" { version-script-commands }

支持的“lang”是“C”、“C++”和“Java”。链接器将在链接时迭代符号列表，并在将它们与“version-script-command”中指定的模式匹配之前根据“lang”对它们进行重组。默认的“lang”是“C”。

重组的名字可以包含空格和其他特殊字符。如上所述，可以使用 glob 模式来匹配重组后的名称，也可以使用双引号字符串来精确匹配字符串。在后一种情况下，请注意版本脚本和重组器的输出之间的细微差异（例如不同的空格）将导致不匹配。由于重组器生成的确切字符串将来可能会改变，即使重组的名称没发生不匹配，你也应该检查升级时所有版本指令是否都按预期运行。

3.10 表达式

链接器脚本语言中表达式的语法与 C 表达式的语法相同。所有表达式都被计算为整数。所有表达式都以相同的大小计算，如果主机和目标都是 32 位，则该大小为 32 位，否则为 64 位。

可以在表达式中使用和设置符号值。

链接器定义了几个用于表达式的特殊的内置函数。

3.10.1 常量

所有常量都是整数。

与 C 中一样，链接器认为以“0”开头的整数是八进制，以“0x”或“0X”开头的整数是十六进制。或者，链接器接受十六进制的“h”或“H”，八进制的“o”或“O”，二进制的“b”或“B”，十进制的“d”或“D”的后缀。没有前缀或后缀的任何整数值都被认为是十进制的。

此外，可以使用后缀 K 和 M 分别缩放常数为 1024 倍或 1024*1024 倍。例如，以下都指相同的数量：

```
_fourk_1 = 4K;
_fourk_2 = 4096;
_fourk_3 = 0x1000;
_fourk_4 = 10000o;
```

注意，K 和 M 后缀不能与上述基本后缀一起使用。

3.10.2 符号常量

可以通过使用 CONSTANT(name)运算符来引用目标特定的常数，其中 *name* 是：

MAXPAGESIZE

目标的最大的页大小。

COMMONPAGESIZE

目标默认的页大小。

所以，以下例子：

```
.text ALIGN (CONSTANT (MAXPAGESIZE)) : { *(.text) }
```

将创建与目标支持的最大页面边界对齐的 text 分区。

3.10.3 符号名称

除非有引号，否则符号名称以字母、下划线或句点开头，可以包括字母、数字、下划线、句点和连字符。未使用引号的符号名称不能与任何关键字冲突。可以通过双引号包围符号名称来指定包含特殊字符或与关键字具有相同名称的符号：

```
"SECTION" = 9;
"with a space" = "also with a space" + 10;
```

由于符号可以包含许多非字母字符，所以用空格分隔符号是最安全的。例如，“A-B”

是一个符号，而“A - B”是一个涉及减法的表达式。

3.10.4 孤立分区

孤立分区是出现在输入文件中的分区，这些分区没有被链接器脚本显式地放入输出文件中。链接器仍然会通过查找或创建适当的输出分区来将这些区段复制到输出文件中，而这个输出分区中放置孤立分区。

如果孤立输入分区的名称与现有输出分区的名称完全相同，则孤立输入分区将放在该输出分区的末尾。

如果没有具有相同名称的输出分区，那么将创建新的输出分区。每个新的输出分区将具有与放在其中的孤立分区相同的名称。如果存在多个具有相同名称的孤立分区，则所有这些都将组合到一个新的输出分区中。

如果创建新的输出分区来保存孤立的输入分区，那么链接器必须决定将这些新的输出分区放在现有输出分区的哪儿。在大多数现代目标上，链接器试图将孤立分区放在同一属性的分区之后，例如代码与数据、可加载与不可加载等。如果没有找到具有匹配属性的分区，或者目标缺少这种支持，则孤立分区被放置在文件的末尾。

命令行选项'--orphan-handling'和'--unique'（参见命令行选项）可用于控制孤立分区被放置在哪个输出分区中。

3.10.5 位置计数器

特殊链接器变量 '.' 总是包含当前输出位置的计数值。由于 '.' 总是指输出分区中的一个位置，所以它只能出现在 SECTIONS 命令内的表达式中。'.' 符号可以出现在表达式中允许普通符号的任何地方。

为'.'赋值，会导致位置计数器被移动。这可以用来在输出分区中创建空白。位置计数器不能在输出分区内向后移动，而且也不要在输出分区外向后移动，否则会创建具有重叠 LMA 的区域。

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

在前面的示例中，file1 中的“.text”分区位于输出分区“output”的开头。接着是一

个 1000 字节的间隙。然后出现 file2 的“.text”分区，在 file3 的“.text”分区之前还有 1000 字节的空隙。“=0x12345678”指定在间隙中填充的数据。（参见输出分区填充）。

注意：'.' 实际上指的是从当前包含对象的开始字节偏移。通常包含对象是 SECTIONS 语句，它的起始地址是 0，因此 '.' 可以用作绝对地址。但是，如果 '.' 用在分区描述中，则它指的是从该分区开始的字节偏移量，而不是绝对地址。因此，在这样的脚本中：

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

即使“.text”输入分区中没有足够的数据来填充该区域，也会为“.text”分区分配 0x100 的起始地址和恰好 0x200 字节的大小。(如果数据太多，将产生错误，因为这就相当于试图向后移动。)“.data”部分将从 0x500 开始，并且在“.data”输入分区结束之后和“.data”输出分区结束之前，还具有额外的 0x600 字节的空间。

如果链接器需要放置孤立分区，则在输出分区语句外部将符号设置为位置计数器的值可能会导致意外的值。例如，如下例子：

```
SECTIONS
{
    start_of_text = . ;
    .text: { *(.text) }
    end_of_text = . ;

    start_of_data = . ;
    .data: { *(.data) }
    end_of_data = . ;
}
```

如果链接器需要放置脚本中未提及的一些输入分区，例如 .rodata，它可能选择在.text 和.data 之间放置。您可能认为链接器应该在上面脚本中的空白行上放置.rodata，但是空白行对链接器没有特别的意义。同样，链接器不会将上面的符号名称与其分区相关联。相反，它假定所有赋值或其他语句都属于前一输出分区，但为 '.' 赋值这种情况除外。也就是说，

链接器将放置孤立的.rodata 分区，就好像脚本是按照以下方式编写的：

```
SECTIONS
{
    start_of_text = . ;
    .text: { *(.text) }
    end_of_text = . ;

    start_of_data = . ;
    .rodata: { *(.rodata) }
    .data: { *(.data) }
    end_of_data = . ;
}
```

这可能是脚本作者对 start_of_data 值的意图，也可能不是。影响孤立分区布局的一种方法是将位置计数器赋值给自己，由于链接器假设赋值到 '.' 就相当于设置下一个输出分区的起始地址，因此应该与该分区分在一组。所以你可以写：

```
SECTIONS
{
    start_of_text = . ;
    .text: { *(.text) }
    end_of_text = . ;

    . = . ;
    start_of_data = . ;
    .data: { *(.data) }
    end_of_data = . ;
}
```

现在，孤立分区.rodata 就会放置在 end_of_text 和 start_of_data 之间了。

3.10.6 操作符

链接器能识别标准 C 的算术操作符集合，而且具有相同的结合性和优先级：

优先级	结合	操作符	备注
(最高)			
1	左	! - ~	(1)
2	左	* / %	
3	左	+ -	
4	左	>> <<	
5	左	== != > < <= >=	
6	左	&	

7	左	|
8	左	&&
9	左	||
10	右	? :
11	右	&= += -= *= /=

(2) (最低)

备注： (1) 前缀操作符 (2) 参考赋值。

3.10.7 求值

链接器懒惰地计算表达式。它只在非常必要的时候才计算表达式的值。

链接器在链接时需要一些信息，例如首个分区起始地址的值、存储区域的安排及大小等。当链接器从脚本文件中读到时，会尽快计算这些值。

然而，其它值(例如符号)直到分配完存储才需要使用。这些值就会晚一些计算，等到其它信息（例如输出分区的大小）可以为符号赋值表达式使用后再计算。

分区大小只有当分配后才可知，因此依赖此信息的符号赋值要等分配存储后才能进行。

一些表达式，例如依赖位置计数器的那些，必须在分区分配过程中计算。

如果某个表达式的结果是必需的，但值不可用，会导致一个错误。例如下面的脚本：

```
SECTIONS
  {
    .text 9+this_isnt_constant :
      { *(.text) }
  }
```

就会导致一条错误消息‘non constant expression for initial address’，是指“使用了非常量表达式作为初始地址”。

3.10.8 表达式的分区

地址和符号可以是分区相关的，或者是绝对值。一个分区相关的符号是可重定位的。如果你使用-r 选项申请可重定位的输出时，将来链接时会改变分区相关符号的值。另一方面，绝对值符号在将来链接时也会保持同样的值。

链接表达式中有些术语是地址，包括分区相关的符号，以及返回值是地址的内置函数(例如：ADDR, LOADADDR, ORIGIN, SEGMENT_START）。其它术语只是简单的数字，或者是具有非地址返回值的内置函数（例如 LENGTH）。一个复杂的问题是，除非设置 LD_FEATURE（“SANE_EXPR”）(参考 杂项命令)，数字和绝对符号根据它们的位置被不同方式对待，以便与旧版本的 LD 兼容。出现在输出分区定义之外的表达式，将所有数字视为绝对地址。出现在输出分区定义内的表达式将绝对符号视为数字。如果给出了 LD_FEATURE（“SANE_EXPR”），那在任何地方的绝对符号和数字就都被简单当作数字。

在下面的例子中，

```
SECTIONS
  {
    . = 0x100;
    __executable_start = 0x100;
    .data :
    {
      . = 0x10;
      __data_start = 0x10;
      *(.data)
    }
    …
  }
```

在开始的两个赋值语句中，‘.’和__executable_start 都被设置成 0x100 绝对地址。然后两个赋值语句中，‘.’和__data_start 都被设成相对于.data 分区加 0x10 的地址。

对于包含数字的表达式、相对地址和绝对地址，ld 用下面的规则计算：

- 对绝对地址或数字的一元运算，对两个绝对地址或两个数字的二元运算，或在一个绝对地址和数字之间的二元运算，会用这些值参与运算。
- 相对地址的一元运算，两个在同一分区中的相对地址之间，或者在一个相对地址和一个数字之间，进行的二元运算，会用地址的偏移部分参与运算。
- 其它的二元运算，也就是，两个不在同一分区的相对地址间，或者一个相对地址和一个绝对地址之间，首先将所有非绝对值的部分转化为绝对值，然后再用来运算。

每个子表达式的结果分区如下：

- 只包含数字的操作，结果就是数字。
- 比较的结果，‘&&’和‘||’也是个数字。
- 当使用 LD_FEATURE(“SANE_EXPR”)或在输出分区定义内部，对同一分区中的两个相对地址或两个绝对地址的其他二进制算术和逻辑运算的结果也是一个数字。但是如果没使用 LD_FEATURE(“SANE_EXPR”)，而且不在输出分区定义内部时，结果是个绝对地址。
- 在相对地址间的其它操作，或者一个相对地址和一个数字运算，结果就是在此分区内部的一个相对地址。
- 对绝对地址进行的其它操作仍然是个绝对地址。

你可以使用内置函数 ABSOLUTE，当它可能是相对值时，强制使表达式成为绝对值。例如，创建一个绝对符号，使其值为输出分区’.data’的结束地址：

```
SECTIONS
  {
    .data : { *(.data) _edata = ABSOLUTE(.); }
  }
```

如果没使用‘ABSOLUTE’，‘_edata’会是个和‘.data’分区的相对地址。

使用 LOADADDR 也能强制表达式成为绝对值，这是由于此内置函数返回绝对地址。

3.10.9 内置函数

链接器脚本语言包含一些内置函数，可用于脚本表达式中。

ABSOLUTE(*exp*)

返回表达式 exp 的绝对值（不可重定位，而不是非负数）。主要是用在分区定义内部给一个符号赋值为绝对值；若不使用，符号将是分区内的相对地址。参考表达式的分区。

ADDR(*section*)

返回 *section* 的 VMA 地址。在脚本中前面的位置必须已经定定了分区的位置。在下面的例子中，start_of_output_1、symbol_1、symbol_2 被赋予相同的值，只不过 symbol_1 是相对于.output1 分区的地址，而另外两个是绝对地址。

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

ALIGN(*align*)

ALIGN(*exp*,*align*)

返回位置计数器(.)或者任意表达式对齐到下个以 *align* 对齐的边界值。单个操作数的 ALIGN 不改变位置计数器的值，它只对位置计数器进行算术运算。两个操作数的 ALIGN 允许对一个任意表达式进行向大对齐，ALIGN(*align*) 与 ALIGN(ABSOLUTE(.), *align*)等效。

下面的例子中，将.data 输出分区对齐到上个分区后的下个 0x2000 边界的位置上，设置分区中的一个变量为输入分区后的下一个 0x8000 边界的值：

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

在上例中第一个 ALIGN 指定了一个分区的位置，这是由于使用了分区定义中可选的 *address* 属性（参考 输出分区地址）。第二个 ALIGN 定义了一个符号的值。

内置函数 NEXT 与 ALIGN 特别相似。

ALIGNOF(*section*)

如果此分区已经分配完成，则以字节为单位返回 section 分区的对齐。在该分区未分配时如果计算，链接器会报错。下面例子中，.output 分区的对齐被保存成此分区的首个值。

```
SECTIONS{ …
  .output {
    LONG (ALIGNOF (.output))
    …
    }
… }
```

BLOCK(*exp*)

是 ALIGN 的同义词，是为兼容旧版链接器脚本的。经常在设置输出分区地址时见到。

DATA_SEGMENT_ALIGN(*maxpagesize*, *commonpagesize*)

这与以下两个表达式之一等效：

1. (ALIGN(*maxpagesize*) + (. & (*maxpagesize* - 1)))
2. (ALIGN(*maxpagesize*)
   + ((. + *commonpagesize* - 1) & (*maxpagesize* - *commonpagesize*)))

取决于后者是否使用比前者较少的 *commonpagesize* 作为数据段的页大小(此表达式的值与 DATA_SEGMENT_END 之间的区域)。如果使用了后者，就意味着在运行时，*commonpagesize* 个字节的运行时内存会被节省，而以最多 *commonpagesize* 个字节的磁盘文件空间为代价。

此表达式只能直接在 SECTIONS 命令中使用，不能在任何输出分区描述中使用，而且只能在脚本中使用一次。*commonpagesize* 应该小于或等于 *maxpagesize*，并且应该是对象希望优化的系统页面大小，同时仍然在最大 *maxpagesize* 的系统页面大小上运行。注意，如果系统页面大小大于 *commonpagesize*，‘-z relro’保护则不会生效。例子：

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

DATA_SEGMENT_END(*exp*)

这定义了数据段的结尾，用于计算 DATA_SEGMENT_ALIGN。例子如下：

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

DATA_SEGMENT_RELRO_END(*offset*, *exp*)

当使用‘-z relro’时，定义 PT_GNU_RELRO 段的结尾。当未使用‘-z relro’选项时，DATA_SEGMENT_RELRO_END 什么也不做，如果用了‘-z relro’选项，DATA_SEGMENT_ALIGN 会被填充，这样 exp+offset 会被对齐到 DATA_SEGMENT_ALIGN 的 commonpagesize 属性。如果以上命令在链接脚本中出现，必须放在 DATA_SEGMENT_ALIGN 和 DATA_SEGMENT_END 之间。对第二个参数求值，加上由于区段对齐而在 PT_GNU_RELRO 区段末尾所需的所有填充。例子：

```
. = DATA_SEGMENT_RELRO_END(24, .);
```

DEFINED(*symbol*)

如果 *symbol* 在链接器全局符号表中，并且在使用 DEFINED 的脚本位置之前定义了，就会返回 1，否则返回 0。可以使用此函数来为符号提供默认值。例如，下面的脚本片段展示了如何设置一个全局符号‘begin’指向’.text’分区的首地址，但是如果已经有一个‘begin’符号，就保留原值：

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

LENGTH(*memory*)

返回名叫 *memory* 的存储区域长度。

LOADADDR(*section*)

返回 *section* 分区的 LMA 绝对地址。参考 输出分区 LMA。

LOG2CEIL(*exp*)

返回 $\log_2$(exp)向大取整的结果。LOG2CEL(0)返回 0。

MAX(*exp1*, *exp2*)

返回两者中较大的那个。

MIN(*exp1*, *exp2*)

返回两者中较小的那个。

NEXT(*exp*)

返回下一个是 exp 倍数的未分配的地址。此函数与 ALIGN(exp)特别相似；除非你使用 MEMORY 命令为输出文件定义不连续的存储区域，否则两个函数功能是相同的。

ORIGIN(*memory*)

返回 *memory* 存储区域的起点地址。

SEGMENT_START(*segment*, *default*)

返回 *segment* 段的基地址。如果已经为这个段明确指定了这个值（使用’-T’命令行选项），那么将会返回这个值，否则返回 *default*。目前，‘-T’命令行选项只能用来设置”text”“data”和”bss”分区的基地址，但是你可以在任何区段上使用 SEGMENT_START 函数。

SIZEOF(*section*)

如果已分配了该分区，则返回名为 *section* 的字节的大小。在计算未分配的分区时，链接器将报错。在下面的例子中，symbol_1 和 symbol_2 被设置了相同的值：

```
SECTIONS{ …
  .output {
    .start = . ;
    …
```

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

SIZEOF_HEADERS

sizeof_headers

返回输出文件报头的大小。此信息会出现在输出文件的开头。如果愿意，你可以在设置首个分区的开始地址时使用这个数字，以便于分页。

当生成 ELF 输出文件时，如果链接器脚本使用 SIZEOF_HEADERS 内置函数，则链接器必须在确定所有区段地址和大小之前计算程序头的数量。如果链接器后来发现它需要额外的程序头，它将报告一个错误”not enough room for program headers”（“没有足够的空间容纳程序头”）。为避免这个错误，您必须避免使用 SIZEOF_HEADERS 函数，或者必须重新编写链接器脚本，以避免强制链接器使用额外的程序头，或者您必须自己使用 PHDRS 命令定义程序头（参见 PHDRS）。

3.11 隐式的链接器脚本

如果为链接器指定一个它不能识别的输入文件，即不是 object 文件也不是 archive 文件时，链接器会试着当成链接器脚本文件进行解析。如果这个文件不能被解析成链接器脚本，那链接器将会报错。

隐含式的链接器脚本不会替换默认的链接器脚本。典型的隐含脚本只包含为符号赋值，或者 INPUT，GROUP，VERSION 命令。

正是由于链接器的这个特点，在命令行中作为输入文件指定的所有文件都将被链接器读取，注意，这会影响文件的查找。

4 机器的不同特性

LD 在一些平台上有附加特性；下面部分描述它们。LD 没有附加功能的机器没有列出。下面只保留英文的链接，并未翻译，请自行查看：

- H8/300: H8/300
- i960: Intel 960 家族
- M68HC11/68HC12: Motorola 68HC11 和 68HC12 家族
- ARM: ARM 家族
- HPPA ELF32: HPPA 32-bit ELF
- M68K: Motorola 68K 家族
- MIPS: MIPS 家族
- MMIX: MMIX

- MSP430: MSP430
- NDS32: NDS32
- Nios II: Altera Nios II
- PowerPC ELF32: PowerPC 32-bit ELF 支持
- PowerPC64 ELF64: PowerPC64 64-bit ELF 支持
- S/390 ELF: S/390 ELF 支持
- SPU ELF: SPU ELF 支持
- TI COFF: TI COFF
- WIN32: WIN32 (cygwin/mingw)
- Xtensa: Xtensa 处理器

5 BFD

链接器使用 BFD 库访问 object 和 archive 文件。这些库允许链接器使用相同的流程来操作 object 文件，而无论些 object 文件是什么格式。只是通过扩充 BFD 库将新的格式添加进去，就能支持新的 object 文件格式。然而，为了维护运行时内存，链接器和相关工具经常被配置成只支持有效 object 文件格式中的一个子集。可以使用 objdump -i 查看当前链接器的配置。

与大多数实现一样，BFD 是几个相互冲突的需求之间的折衷。影响 BFD 设计的主要因素是效率：如果不涉及 BFD，原本不需要任何时间进行格式转换。好在此代价所换来的是更加抽象化；由于 BFD 简化了应用程序和后端，所以可能要花费更多的时间和精力来优化算法以实现更高的速度。

您应该记住的 BFD 解决方案的一个小麻烦是信息丢失的可能性。使用 BFD 机制可能丢失有用信息的地方有两个：转换期间和输出期间。参见 BFD 信息丢失。

5.1如何工作的: BFD 概述

当打开目标文件时，BFD 子例程自动确定输入目标文件的格式。然后，在内存中构建一个描述符 ,该描述符带有指向程序的指针 ,这些程序将用于访问对象文件数据结构的元素。

由于需要来自目标文件中的各种信息，BFD 从不同分区读取并处理它们。例如，链接器非常常见的操作是处理符号表。每个 BFD 后端提供一个例程，用于在对象文件的符号表示和内部规范格式之间进行转换。当链接器请求对象文件的符号表时，它通过一个指向相关 BFD 后端例程的内存指针调用程序，该程序读取该表并将其转换为规范形式。链接器然后以规范的形式操作。当链接完成并且链接器写入输出文件的符号表时，调用另一个 BFD 后端例程来获取新创建的符号表并将其转换为所选的输出格式。

5.1.1 BFD 信息丢失

*在输出期间可能会丢失信息。*BFD 支持的输出格式不提供重复的信息，用一种形式描述的信息不会以另一种形式在其它地方出现。一个例子是在 b.out 中的对齐信息。在 a.out 格式的文件中没有地方存储关于所包含数据的对齐信息，因此当从 b.out 链接文件并生成 a.out 镜像时，对齐信息将不会传递到输出文件。（链接器仍然在内部使用对齐信息，所以能正确地执行）。

另一个例子是 COFF 分区名。COFF 文件可以包含无限数量的分区，每个分区具有文本名称。如果链接的目标是一种没有很多分区的格式（例如，a.out）或具有没名称的分区（例如，Oasys 格式），则链接不能简单地完成。可以通过使用链接器命令语言描述所需的输入到输出部分映射来避免这个问题。

*在规范化过程中，信息可能会丢失。*外部格式到 BFD 内部规范形式的转换并不彻底；在输入格式中存在内部没有直接表示的结构。这意味着，BFD 后端不能通过从外到内再到外部的格式转换来维持所有数据的丰富度。

只有当应用程序从一个格式读入，然后以另一种格式写出时，这个缺陷才是个问题。每个 BFD 后端都有责任维护尽可能多的数据，内部 BFD 规范形式具有对 BFD 核心不透明的结构，而且只导出到后端。当以一种格式读取文件时，为 BFD 和应用生成标准形式。同时，后端可以保存任何可能丢失的信息。如果数据随后以相同的格式被写回，则后端程序将能够像使用规范形式一样使用它先前准备的不规范的信息。由于后端之间有很多共性，所以在链接或复制大端 COFF 到小端 COFF 或 a.out 到 b.out 时，不会丢失任何信息。当链接混合格式时，只有输入格式与目标格式不同的文件中的信息才会丢失。

5.1.2 BFD 规范对象文件格式

当源格式提供的信息、标准格式存储的信息和目标格式需要的信息之间重叠最小时，信息丢失的最大可能性就出现了。对规范形式的简要描述可以帮助你理解哪些类型的数据可以跨转换进行保存。

files

基于每个文件存储的信息包括目标机器体系结构、特定实现格式类型、可分页需求位和写保护位。这里不存储像 Unix 魔术数字之类的信息，而只存储魔术数字的含义，因此 ZMAGIC 文件将同时设置可分页请求位和写保护文本位。目标的字节顺序是按每个文件存储的，因此大端对象文件和小端对象文件可以彼此一起使用。

sections

输入文件中的每个分区都包含该分区的名称、目标文件中该分区的起始地址、大小和对齐信息、各种标志以及指向其他 BFD 数据结构的指针。

symbols

每个符号都包含一个指针，指向最初定义它的目标文件的信息、它的名称、它的值以及各种标记位。当 BFD 后端读取符号表时，它将重新定位所有符号，使它们相对于定义它们的分区基地址。这样做可以确保每个符号指向其包含的分区。每个符号也有给 BFD 后端准备的不同数量的隐藏私有数据。由于符号指向原始文件，可以访问该符号的私有数据格式。ld 可以毫无问题地操作具有完全不同格式的符号集合。

在输出文件中，维护着一般全局符号和简单局部符号，因此输出文件（无论其格式如何）将保留指向函数和全局、静态和公共变量的符号。有些符号信息不值得保留；在 a.out 中，类型信息作为长符号名存储在符号表中。这些信息对大多数 COFF 调试器都是无用的；链接器具有命令行开关，允许用户丢弃它们。

在符号内有一个类型信息的字，所以如果格式支持符号内的符号类型信息（例如，COFF、IEEE、Oasys），并且类型足够简单以适合于一个字（几乎除了聚合之外的所有内容），那么信息将被保留。

relocation level

每个规范的 BFD 重定位记录都包含指向要重定位符号的指针、要重定位的数据偏移量、数据所在的分区以及指向重定位类型描述符的指针。通过将消息传递给重定位类型描述符和符号指针来执行重定位。因此，可以使用仅在输入格式之一中可用的重定位方法对输出数据进行重定位。例如，Oasys 提供了字节重定位格式。请求这种重定位类型的重定位记录将间接指向执行该重定位的例程，因此可以在写入 68k COFF 文件的字节上执行重定位，即使 68k COFF 没有这种重定位类型。

line numbers

为了调试的目的，对象格式可以包含符号、源代码行号和输出文件中的地址之间的某种形式的映射。这些地址必须与符号信息一起重新定位。每个具有相关联的行号列表的符号指向列表的第一个记录。行号列表的头部由指向符号的指针组成，该指针允许找出描述行号的函数的地址。列表的其余部分是由信息对组成的：分区的偏移和行号。任何能够简单获得此信息的格式都可以在格式（COFF、IEEE 和 Oasys）之间成功地传递它。

6 附录 A：MRI 兼容脚本文件

为了帮助用户从 MRI 链接器过渡到 GNU ld，ld 可以使用 MRI 兼容的链接器脚本作为脚本中描述的更通用的链接器脚本语言的替代。与 MRI 兼容的链接器脚本的命令集比与 ld 一起使用的脚本语言简单得多。GNU ld 支持最常用的 MRI 链接器命令；这里描述这些命令。

一般来说，MRI 脚本对于 a.out 对象文件格式没有多大用处，因为它只有三个分区，并且 MRI 脚本缺乏利用它们的一些特征。

可以使用“-C”命令行选项指定包含 MRI 兼容脚本的文件。

在 MRI 兼容脚本中的每个命令占一行；每个命令行都以标识该命令的关键字开头（尽管空白行也允许标点符号）。如果一行 MRI 兼容脚本以未识别的关键字开始，ld 发出警告消息，但是继续处理该脚本。

以“*”开头的行是注释。

可以用全部大写字母或者小写字母书写命令；例如，“chip”与”CHIP”是相同的。下面只用大写字母的形式来说明每个命令。

ABSOLUTE *secname*

ABSOLUTE *secname*, *secname*, … *secname*

通常，ld 在输出文件中包含来自所有输入文件的所有分区。但是，在 MRI 兼容脚本中，可以使用 ABSOLUTE 命令来限制输出程序中出现的分区。如果在脚本中使用了 ABSOLUTE 命令，那么只有 ABSOLUTE 命令中明确指定的分区才会出现在链接器输出中。您仍然可以使用其他输入分区(无论在命令行上选择什么 ,或者使用 LOAD)来解析输出文件中的地址。

ALIAS *out-secname*, *in-secname*

使用此命令将名为 *in-secname* 分区中的数据放到链接器输出文件中的名为 *out-secname* 的分区中。*In-secname* 可以是个整数。

ALIGN *secname* = *expression*

将 *secname* 对齐到 *expression*。*Expression* 应该是 2 的 n 次方。

BASE *expression*

使用 *expression* 的值作为输出文件中的最低地址（除了绝对地址）。

CHIP *expression*

CHIP *expression*, *expression*

此命令无动作；只为兼容。

END

此命令无动作；只为兼容。

FORMAT *output-format*

与更通用的链接器语言中的 OUTPUT_FORMAT 命令相似，但仅限于以下格式中的：

- S-records，如果 *output-format* 是‘S’。
- IEEE，如果 *output-format* 是‘IEEE’。
- COFF (BFD 中的‘coff-m68k’), 如果 *output-format* 是‘COFF’。

LIST *anything*...

打印(到标准输出文件)由 ld 命令行选项'-M'生成的链接映射。

关键字 LIST 同一行的后面可以跟随任何命令，都不会影响它的效果。

LOAD *filename*

LOAD *filename*, *filename*, ... *filename*

链接一个或多个目标文件；这与在 ld 命令行上直接指定文件名具有相同的效果。

NAME *output-name*

output-name 是 ld 生成的程序的名称；MRI 兼容命令 NAME 等效于命令行选项'-o'或一般脚本语言命令 OUTPUT。

ORDER *secname*, *secname*, ... *secname*

ORDER *secname secname secname*

通常情况下，ld 按照分区在输入文件中出现的顺序对输出文件中的分区进行排序。在 MRI 兼容脚本中，您可以用 ORDER 命令改写这一顺序。你使用 ORDER 列出的分区会以指定的顺序首先出现在输出文件中。

PUBLIC *name*= *expression*

PUBLIC *name*,*expression*

PUBLIC *name expression*

为链接器输入文件中使用的外部符号 *name* 提供一个值（*expression*）。

SECT *secname*, *expression*

SECT *secname*= *expression*

SECT *secname expression*

可以使用这三种形式的 SECT 命令中的任何一种来指定 *secname* 分区的起始地址（*expression*）。如果对于相同的 *secname* 有多个 SECT 语句，则只有第一个生效。