本文乃转载，我在其基础上做了少量修改．原作者的 E-mail 是 zhanglei@sict.ac.cn．完成于 2005.11.5-2005.11.8

0. Contents

1．概论

每一个链接过程都由链接脚本(linker script，一般以 lds 作为文件的后缀名)控制．链接脚本主要用于规定如何把输入文件内的 section 放入输出文件内，并控制输出文件内各部分在程序地址空间内的布局．但你也可以用连接命令做一些其他事情．

连接器有个默认的内置连接脚本，可用 ld --verbose 查看．连接选项-r 和-N 可以影响默认的连接脚本(如何影响?)．

-T 选项用以指定自己的链接脚本，它将代替默认的连接脚本。你也可以使用<暗含的连接脚本>以增加自定义的链接命令．

以下没有特殊说明，连接器指的是静态连接器．

2．基本概念

链接器把一个或多个输入文件合成一个输出文件．

输入文件：目标文件或链接脚本文件．
输出文件：目标文件或可执行文件．

> 目标文件(包括可执行文件)具有固定的格式，在UNIX或GNU/Linux平台下，一般为ELF格式．若想了解更多，可参考 UNIX/Linux平台可执行文件格式分析

有时把输入文件内的 section 称为输入 section(input section)，把输出文件内的 section 称为输出 section(output sectin)．

目标文件的每个 section 至少包含两个信息：名字和大小．大部分 section 还包含与它相关联的一块数据，称为 section contents(section 内容)．一个 section 可被标记为“loadable(可加载的)”或“allocatable(可分配的)”．

loadable section：在输出文件运行时，相应的 section 内容将被载入进程地址空间中.
allocatable section：内容为空的 section 可被标记为“可分配的”. 在输出文件运行时，在进程地址空间中空出大小同 section 指定大小的部分. 某些情况下，这块内存必须被置零.

如果一个 section 不是“可加载的”或“可分配的”，那么该 section 通常包含了调试信息. 可用 objdump -h 命令查看相关信息.

每个“可加载的”或“可分配的”输出 section 通常包含两个地址：VMA(virtual memory address 虚拟内存地址或程序地址空间地址)和 LMA(load memory address 加载内存地址或进程地址空间地址). 通常 VMA 和 LMA 是相同的.

> 在目标文件中，loadable 或 allocatable 的输出 section 有两种地址：VMA(virtual Memory Address)和 LMA(Load Memory Address). VMA 是执行输出文件时 section 所在的地址，而 LMA 是加载输出文件时 section 所在的地址. 一般而言，某 section 的 VMA == LMA. 但在嵌入式系统中，经常存在加载地址和执行地址不同的情况：比如将输出文件加载到开发板的 flash 中(由 LMA 指定)，而在运行时将位于 flash 中的输出文件复制到 SDRAM 中(由 VMA 指定).

可这样来理解VMA和LMA，假设：
(1) .data section对应的VMA地址是 0x08050000，该section内包含了 3 个 32 位全局变量，i、j和k，分别为 1,2,3.
(2) .text section内包含由"printf("j=%d ", j);"程序片段产生的代码.

连接时指定.data section的VMA为 0x08050000，产生的printf指令是将地址为 0x08050004 处的 4 字节内容作为一个整数打印出来。

如果.data section的LMA为 0x08050000，显然结果是j=2
如果.data section的LMA为 0x08050004，显然结果是j=1

还可这样理解LMA:
.text section内容的开始处包含如下两条指令(intel i386 指令是 10 字节，每行对应 5 字节):

```
jmp 0x08048285
movl $0x1,%eax
```

如果.text section的LMA为 0x08048280，那么在进程地址空间内 0x08048280 处为“jmp 0x08048285”指令，0x08048285 处为 movl $0x1,%eax 指令. 假设某指令跳转到地址 0x08048280，显然它的执行将导致%eax寄存器被赋值为 1.

如果.text section的LMA为 0x08048285，那么在进程地址空间内 0x08048285 处为“jmp 0x08048285”指令，0x0804828a 处为 movl $0x1,%eax 指令. 假设某指令跳转到地址 0x08048285，显然它的执行又跳转到进程地址空间内 0x08048285 处，造成死循环.

符号(symbol)：每个目标文件都有符号表(SYMBOL TABLE)，包含已定义的符号(对应全局变量和static变量和定义的函数的名字)和未定义符号(未定义的函数的名字和引用但没定义的符s号)信息.

符号值：每个符号对应一个地址，即符号值(这与c程序内变量的值不一样，某种情况下可以把它看成变量的地址). 可用nm命令查看它们. (nm的使用方法可参考本blog的 GNU binutils笔记)

3. 脚本格式

链接脚本由一系列命令组成，每个命令由一个关键字(一般在其后紧跟相关参数)或一条对符号的赋值语句组成．命令由分号';'分隔开．

文件名或格式名内如果包含分号';'或其他分隔符，则要用引号'"'将名字全称引用起来．无法处理含引号的文件名．
/* */之间的是注释。

4. 简单例子

在介绍链接描述文件的命令之前，先看看下述的简单例子:

术语：把定位器符号
一般就是那个． s
以下脚本将输出文件的 text section 定位在 0x10000，data section 定位在 0x8000000:

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

解释一下上述的例子:
. = 0x10000 : 把定位器符号置为 0x10000 (若不指定，则该符号的初始值为 0)．

.text : { *(.text) } : 将所有(*符号代表任意输入文件)输入文件的.text section 合并成一个.text section，该 section 的地址由定位器符号的值指定，即 0x10000．

. = 0x8000000 : 把定位器符号置为 0x8000000
.data : { *(.data) } : 将所有输入文件的.text section 合并成一个.data section，该 section 的地址被置为 0x8000000．

.bss : { *(.bss) } : 将所有输入文件的.bss section 合并成一个.bss section，该 section 的地址被置为 0x8000000+.data section 的大小．

连接器每读完一个 section 描述后，将定位器符号的值*增加*该 section 的大小．注意：此处没有考虑对齐约束．

5. 简单脚本命令

- 1 -

ENTRY(SYMBOL) : 将符号 SYMBOL 的值设置成入口地址。

入口地址(entry point): 进程执行的第一条用户空间的指令在进程地址空间的地址)

ld 有多种方法设置进程入口地址，按一下顺序: (编号越前，优先级越高)
1, ld 命令行的-e 选项
2，连接脚本的 ENTRY(SYMBOL)命令
3，如果定义了 start 符号，使用 start 符号值
4，如果存在.text section，使用.text section 的第一字节的位置值
5，使用值 0

- 2 -

INCLUDE filename ： 包含其他名为 filename 的链接脚本

相当于 c 程序内的的#include 指令，用以包含另一个链接脚本.

脚本搜索路径由-L 选项指定. INCLUDE 指令可以嵌套使用，最大深度为 10. 即: 文件 1 内 INCLUDE 文件 2，文件 2 内 INCLUDE 文件 3... ，文件 10 内 INCLUDE 文件 11. 那么文件 11 内不能再出现 INCLUDE 指令了.

- 3 -

INPUT(files): 将括号内的文件做为链接过程的输入文件

ld 首先在当前目录下寻找该文件，如果没找到，则在由-L 指定的搜索路径下搜索. file 可以为 -lfile 形式，就象命令行的-l 选项一样. 如果该命令出现在暗含的脚本内，则该命令内的 file 在链接过程中的顺序由该暗含的脚本在命令行内的顺序决定.

- 4 -

GROUP(files) ： 指定需要重复搜索符号定义的多个输入文件

file 必须是库文件，且 file 文件作为一组被 ld 重复扫描，直到不在有新的未定义的引用出现。

- 5 -

OUTPUT(FILENAME) ： 定义输出文件的名字

同 ld 的-o 选项，不过-o 选项的优先级更高. 所以它可以用来定义默认的输出文件名. 如 a.out

- 6 -

SEARCH_DIR(PATH) ： 定义搜索路径，

同 ld 的-L 选项，不过由-L 指定的路径要比它定义的优先被搜索。

- 7 -

STARTUP(filename) ： 指定 filename 为第一个输入文件

在链接过程中，每个输入文件是有顺序的. 此命令设置文件 filename 为第一个输入文件。

- 8 -

OUTPUT_FORMAT(BFDNAME) ： 设置输出文件使用的 BFD 格式

同 ld 选项-o format BFDNAME，不过 ld 选项优先级更高.

OUTPUT_FORMAT(DEFAULT,BIG,LITTLE) ： 定义三种输出文件的格式(大小端)

若有命令行选项-EB，则使用第 2 个 BFD 格式；若有命令行选项-EL，则使用第 3 个 BFD 格式.否则默认选第一个 BFD 格式.

TARGET(BFDNAME)：设置输入文件的 BFD 格式

同 ld 选项-b BFDNAME. 若使用了 TARGET 命令，但未使用 OUTPUT_FORMAT 命令，则最用一个 TARGET 命令设置的 BFD 格式将被作为输出文件的 BFD 格式.

另外还有一些:
ASSERT(EXP, MESSAGE)：如果 EXP 不为真，终止连接过程

EXTERN(SYMBOL SYMBOL ...)：在输出文件中增加未定义的符号，如同连接器选项-u

FORCE_COMMON_ALLOCATION：为 common symbol(通用符号)分配空间，即使用了-r 连接选项也为其分配

NOCROSSREFS(SECTION SECTION ...)：检查列出的输出 section，如果发现他们之间有相互引用，则报错。对于某些系统，特别是内存较紧张的嵌入式系统，某些 section 是不能同时存在内存中的，所以他们之间不能相互引用。

OUTPUT_ARCH(BFDARCH)：设置输出文件的 machine architecture(体系结构)，BFDARCH 为被 BFD 库使用的名字之一。可以用命令 objdump -f 查看。

可通过 man -S 1 ld 查看 ld 的联机帮助，里面也包括了对这些命令的介绍.

6. 对符号的赋值

在目标文件内定义的符号可以在链接脚本内被赋值. (注意和 C 语言中赋值的不同!) 此时该符号被定义为全局的. 每个符号都对应了一个地址，此处的赋值是更改这个符号对应的地址.

e.g. 通过下面的程序查看变量 a 的地址:

```
/* a.c */
#include <stdio.h>
int a = 100;
int main(void)
{
   printf( "&a=0x%p ", &a );
   return 0;
}
```

```
/* a.lds */
a = 3;

$ gcc -Wall -o a-without-lds a.c
&a = 0x8049598

$ gcc -Wall -o a-with-lds a.c a.lds
&a = 0x3
```

> 注意：对符号的赋值只对全局变量起作用!

一些简单的赋值语句
能使用任何 c 语言内的赋值操作:

```
SYMBOL = EXPRESSION ;
SYMBOL += EXPRESSION ;
SYMBOL -= EXPRESSION ;
SYMBOL *= EXPRESSION ;
SYMBOL /= EXPRESSION ;
SYMBOL <<= EXPRESSION ;
SYMBOL >>= EXPRESSION ;
SYMBOL &= EXPRESSION ;
SYMBOL |= EXPRESSION ;
```

除了第一类表达式外，使用其他表达式需要 SYMBOL 被定义于某目标文件。
. 是一个特殊的符号，它是定位器，一个位置指针，指向程序地址空间内的某位置(或某 section 内的偏移，如果它在 SECTIONS 命令内的某 section 描述内)，该符号只能在 SECTIONS 命令内使用。
注意：赋值语句包含 4 个语法元素：符号名、操作符、表达式、分号；一个也不能少。
被赋值后，符号所属的 section 被设值为表达式 EXPRESSION 所属的 SECTION(参看 11. 脚本内的表达式)
赋值语句可以出现在连接脚本的三处地方：SECTIONS 命令内，SECTIONS 命令内的 section 描述内和全局位置；如下，

```
floating_point = 0; /* 全局位置 */
SECTIONS
{
.text :
{
*(.text)
_etext = .; /* section 描述内 */
}
_bdata = (. + 3) & ~ 4; /* SECTIONS 命令内 */
.data : { *(.data) }
}
```

PROVIDE 关键字
该关键字用于定义这类符号：在目标文件内被引用，但没有在任何目标文件内被定义的符号。
例子：

```
SECTIONS
{
.text :
```

```
{
*(.text)
_etext = .;
PROVIDE(etext = .);
}
}
```
当目标文件内引用了 etext 符号，确没有定义它时，etext 符号对应的地址被定义为.text section 之后的第一个字节的地址。

7. SECTIONS 命令

SECTIONS 命令告诉 ld 如何把输入文件的 sections 映射到输出文件的各个 section：如何将输入 section 合为输出 section；如何把输出 section 放入程序地址空间(VMA)和进程地址空间(LMA).该命令格式如下：

```
SECTIONS
{
SECTIONS-COMMAND
SECTIONS-COMMAND
...
}
```

SECTION-COMMAND 有四种：
(1) ENTRY 命令
(2) 符号赋值语句
(3) 一个输出 section 的描述(output section description)
(4) 一个 section 叠加描述(overlay description)

如果整个连接脚本内没有 SECTIONS 命令，那么 ld 将所有同名输入 section 合成为一个输出 section 内，各输入 section 的顺序为它们被连接器发现的顺序.

如果某输入 section 没有在 SECTIONS 命令中提到，那么该 section 将被直接拷贝成输出 section。

输出 section 描述
输出 section 描述具有如下格式：

```
SECTION [ADDRESS] [(TYPE)] : [AT(LMA)]
{
OUTPUT-SECTION-COMMAND
OUTPUT-SECTION-COMMAND
...
} [>REGION] [AT>LMA_REGION] [:PHDR :PHDR ...] [=FILLEXP]
```

[ ]内的内容为可选选项，一般不需要.
SECTION：section 名字
SECTION 左右的空白、圆括号、冒号是必须的，换行符和其他空格是可选的。
每个 OUTPUT-SECTION-COMMAND 为以下四种之一，
符号赋值语句
一个输入 section 描述

直接包含的数据值
一个特殊的输出 section 关键字

输出 section 名字(SECTION)：
输出 section 名字必须符合输出文件格式要求，比如：a.out 格式的文件只允许存在.text、.data 和.bss section 名。而有的格式只允许存在数字名字，那么此时应该用引号将所有名字内的数字组合在一起；另外，还有一些格式允许任何序列的字符存在于 section 名字内，此时如果名字内包含特殊字符(比如空格、逗号等)，那么需要用引号将其组合在一起。

输出 section 地址(ADDRESS)：
ADDRESS 是一个表达式，它的值用于设置 VMA。如果没有该选项且有 REGION 选项，那么连接器将根据 REGION 设置 VMA；如果也没有 REGION 选项，那么连接器将根据定位符号'.'的值设置该 section 的 VMA，将定位符号的值调整到满足输出 section 对齐要求后的值，输出 section 的对齐要求为：该输出 section 描述内用到的所有输入 section 的对齐要求中最严格的。
例子：
.text . : { *(.text) }
和
.text : { *(.text) }
这两个描述是截然不同的，第一个将.text section 的 VMA 设置为定位符号的值，而第二个则是设置成定位符号的修调值，满足对齐要求后的。
ADDRESS 可以是一个任意表达式，比如 ALIGN(0x10)这将把该 section 的 VMA 设置成定位符号的修调值，满足 16 字节对齐后的。
注意：设置 ADDRESS 值，将更改定位符号的值。

输入 section 描述：
最常见的输出 section 描述命令是输入 section 描述。
输入 section 描述是最基本的连接脚本描述。
输入 section 描述基础：
基本语法：FILENAME([EXCLUDE_FILE (FILENAME1 FILENAME2 ...) SECTION1 SECTION2 ...)
FILENAME 文件名，可以是一个特定的文件的名字，也可以是一个字符串模式。
SECTION 名字，可以是一个特定的 section 名字，也可以是一个字符串模式
例子是最能说明问题的，
*(.text) ：表示所有输入文件的.text section
(*(EXCLUDE_FILE (*crtend.o *otherfile.o) .ctors)) ：表示除 crtend.o、otherfile.o 文件外的所有输入文件的.ctors section。
data.o(.data) ：表示 data.o 文件的.data section
data.o ：表示 data.o 文件的所有 section
*(.text .data) ：表示所有文件的.text section 和.data section，顺序是：第一个文件的.text section，第一个文件的.data section，第二个文件的.text section，第二个文件的.data section，...
*(.text) *(.data) ：表示所有文件的.text section 和.data section，顺序是：第一个文件的.text section，第二个文件的.text section，...，最后一个文件的.text section，第一个文件的.data section，第二个文件的.data section，...，最后一个文件的.data section
下面看连接器是如何找到对应的文件的。
当 FILENAME 是一个特定的文件名时，连接器会查看它是否在连接命令行内出现或在 INPUT 命令中出现。
当 FILENAME 是一个字符串模式时，连接器仅仅只查看它是否在连接命令行内出现。
注意：如果连接器发现某文件在 INPUT 命令内出现，那么它会在-L 指定的路径内搜寻该文件。

字符串模式内可存在以下通配符：
* ：表示任意多个字符

? ：表示任意一个字符
[CHARS] ：表示任意一个 CHARS 内的字符，可用-号表示范围，如：a-z
：表示引用下一个紧跟的字符

在文件名内，通配符不匹配文件夹分隔符/，但当字符串模式仅包含通配符*时除外。
任何一个文件的任意 section 只能在 SECTIONS 命令内出现一次。看如下例子，
SECTIONS {
.data：{ *(.data) }
.data1：{ data.o(.data) }
}
data.o 文件的.data section 在第一个 OUTPUT-SECTION-COMMAND 命令内被使用了，那么在第二个 OUTPUT-SECTION-COMMAND 命令内将不会再被使用，也就是说即使连接器不报错，输出文件的.data1 section 的内容也是空的。
再次强调：连接器依次扫描每个 OUTPUT-SECTION-COMMAND 命令内的文件名，任何一个文件的任何一个 section 都只能使用一次。
读者可以用-M 连接命令选项来产生一个 map 文件，它包含了所有输入 section 到输出 section 的组合信息。
再看个例子，
SECTIONS {
.text：{ *(.text) }
.DATA：{ [A-Z]*(.data) }
.data：{ *(.data) }
.bss：{ *(.bss) }
}
这个例子中说明，所有文件的输入.text section 组成输出.text section；所有以大写字母开头的文件的.data section 组成输出.DATA section，其他文件的.data section 组成输出.data section；所有文件的输入.bss section 组成输出.bss section。
可以用 SORT()关键字对满足字符串模式的所有名字进行递增排序，如 SORT(.text*)。
通用符号(common symbol)的输入 section：
在许多目标文件格式中，通用符号并没有占用一个 section。连接器认为：输入文件的所有通用符号在名为 COMMON 的 section 内。
例子，
.bss { *(.bss) *(COMMON) }
这个例子中将所有输入文件的所有通用符号放入输出.bss section 内。可以看到 COMMOM section 的使用方法跟其他 section 的使用方法是一样的。
有些目标文件格式把通用符号分成几类。例如，在 MIPS elf 目标文件格式中，把通用符号分成 standard common symbols(标准通用符号)和 small common symbols(微通用符号，不知道这么译对不对？)，此时连接器认为所有 standard common symbols 在 COMMON section 内，而 small common symbols 在.scommon section 内。
在一些以前的连接脚本内可以看见[COMMON]，相当于*(COMMON)，不建议继续使用这种陈旧的方式。
输入 section 和垃圾回收：
在连接命令行内使用了选项--gc-sections 后，连接器可能将某些它认为没用的 section 过滤掉，此时就有必要强制连接器保留一些特定的 section，可用 KEEP()关键字达此目的。如 KEEP(*(.text))或 KEEP(SORT(*)(.text))
最后看个简单的输入 section 相关例子：
SECTIONS {
outputa 0x10000 :
{
all.o

foo.o (.input1)
}
outputb：
{
foo.o (.input2)
foo1.o (.input1)
}
outputc：
{
*(.input1)
*(.input2)
}
}
本例中，将 all.o 文件的所有 section 和 foo.o 文件的所有(一个文件内可以有多个同名 section).input1 section 依次放入输出 outputa section 内，该 section 的 VMA 是 0x10000；将 foo.o 文件的所有.input2 section 和 foo1.o 文件的所有.input1 section 依次放入输出 outputb section 内，该 section 的 VMA 是当前定位器符号的修调值(对齐后)；将其他文件(非 all.o、foo.o、foo1.o)文件的.input1 section 和.input2 section 放入输出 outputc section 内。

在输出 section 存放数据命令：
能够显示地在输出 section 内填入你想要填入的信息(这样是不是可以自己通过连接脚本写程序？当然是简单的程序)。
BYTE(EXPRESSION) 1 字节
SHORT(EXPRESSION) 2 字节
LOGN(EXPRESSION) 4 字节
QUAD(EXPRESSION) 8 字节
SQUAD(EXPRESSION) 64 位处理器的代码时，8 字节
输出文件的字节顺序 big endianness 或 little endianness，可以由输出目标文件的格式决定；如果输出目标文件的格式不能决定字节顺序，那么字节顺序与第一个输入文件的字节顺序相同。
如：BYTE(1)、LANG(addr)。
注意，这些命令只能放在输出 section 描述内，其他地方不行。
错误：SECTIONS { .text：{ *(.text) } LONG(1) .data：{ *(.data) } }
正确：SECTIONS { .text：{ *(.text) LONG(1) } .data：{ *(.data) } }
在当前输出 section 内可能存在未描述的存储区域(比如由于对齐造成的空隙)，可以用 FILL(EXPRESSION)命令决定这些存储区域的内容， EXPRESSION 的前两字节有效，这两字节在必要时可以重复被使用以填充这类存储区域。如 FILE(0x9090)。在输出 section 描述中可以有=FILEEXP 属性，它的作用如同 FILE()命令，但是 FILE 命令只作用于该 FILE 指令之后的 section 区域，而=FILEEXP 属性作用于整个输出 section 区域，且 FILE 命令的优先级更高!!!

输出 section 内命令的关键字：
CREATE_OBJECT_SYMBOLS ：为每个输入文件建立一个符号，符号名为输入文件的名字。每个符号所在的 section 是出现该关键字的 section。
CONSTRUCTORS ：与 c++内的(全局对象的)构造函数和(全局对像的)析构函数相关，下面将它们简称为全局构造和全局析构。
对于 a.out 目标文件格式，连接器用一些不寻常的方法实现 c++的全局构造和全局析构。当连接器生成的目标文件格式不支持任意 section 名字时，比如说 ECOFF、XCOFF 格式，连接器将通过名字来识别全局构造和全局析构，对于这些文件格式，连接器把与全局构造和全局析构的相关信息放入出现 CONSTRUCTORS 关键字的输出 section 内。
符号__CTORS_LIST__表示全局构造信息的的开始处，__CTORS_END__表示全局构造信息的结束处。

符号__DTORS_LIST__表示全局构造信息的的开始处，__DTORS_END__表示全局构造信息的结束处。
这两块信息的开始处是一字长的信息，表示该块信息有多少项数据，然后以值为零的一字长数据结束。
一般来说，GNU C++在函数__main 内安排全局构造代码的运行，而__main 函数被初始化代码(在 main 函数调用之前执行)调用。是不是对于某些目标文件格式才这样？？？
对于支持任意 section 名的目标文件格式，比如 COFF、ELF 格式，GNU C++将全局构造和全局析构信息分别放入.ctors section 和.dtors section 内，然后在连接脚本内加入如下，

```
__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++提供的初始化优先级支持(它能控制每个全局构造函数调用的先后顺序)，那么请在连接脚本内把 CONSTRUCTORS 替换成 SORT (CONSTRUCTS)，把 *(.ctors) 换成 *(SORT(.ctors))，把*(.dtors)换成*(SORT(.dtors))。一般来说，默认的连接脚本已作好的这些工作。

输出 section 的丢弃：
例子，.foo { *(.foo) }，如果没有任何一个输入文件包含.foo section，那么连接器将不会创建.foo 输出 section。但是如果在这些输出 section 描述内包含了非输入 section 描述命令(如符号赋值语句)，那么连接器将总是创建该输出 section。
有一个特殊的输出 section，名为/DISCARD/，被该 section 引用的任何输入 section 将不会出现在输出文件内，这就是 DISCARD 的意思吧。如果/DISCARD/ section 被它自己引用呢？想想看。

输出 section 属性：
终于讲到这里了，呵呵。
我们再回顾以下输出 section 描述的文法：

```
SECTION [ADDRESS] [(TYPE)] : [AT(LMA)]
{
OUTPUT-SECTION-COMMAND
OUTPUT-SECTION-COMMAND
...
} [>REGION] [AT>LMA_REGION] [:PHDR :PHDR ...] [=FILLEXP]
```

前面我们浏览了 SECTION、ADDRESS、OUTPUT-SECTION-COMMAND 相关信息，下面我们将浏览其他属性。

TYPE ：每个输出 section 都有一个类型，如果没有指定 TYPE 类型，那么连接器根据输出 section 引用的输入 section 的类型设置该输出 section 的类型。它可以为以下五种值，
NOLOAD ：该 section 在程序运行时，不被载入内存。
DSECT,COPY,INFO,OVERLAY ：这些类型很少被使用，为了向后兼容才被保留下来。这种类型的 section 必须被标记为“不可加载的”，以便在程序运行不为它们分配内存。

输出 section 的 LMA ：默认情况下，LMA 等于 VMA，但可以通过关键字 AT()指定 LMA。
用关键字 AT()指定，括号内包含表达式，表达式的值用于设置 LMA。如果不用 AT()关键字，那么可用 AT>LMA_REGION 表达式设置指定该 section 加载地址的范围。

这个属性主要用于构件 ROM 境象。
例子，

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

程序如下，

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

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

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

此程序将处于 ROM 内的已初始化数据拷贝到该数据应在的位置(VMA 地址)，并将为初始化数据置零。读者应该认真的自己分析以上连接脚本和程序的作用。

输出 section 区域：可以将输出 section 放入预先定义的内存区域内，例子，

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

输出 section 所在的程序段：可以将输出 section 放入预先定义的程序段(program segment)内。如果某个输出 section 设置了它所在的一个或多个程序段，那么接下来定义的输出 section 的默认程序段与该输出 section 的相同。除非再次显示地指定。例子，

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

可以通过:NONE 指定连接器不把该 section 放入任何程序段内。详情请查看 PHDRS 命令

输出 section 的填充模版：这个在前面提到过，任何输出 section 描述内的未指定的内存区域，连接器用该模版填充该区域。用法：=FILEEXP，前两字节有效，当区域大于两字节时，重复使用这两字节以将其填满。例子，

```
SECTIONS { .text : { *(.text) } =0x9090 }
```

覆盖图(overlay)描述：
覆盖图描述使两个或多个不同的 section 占用同一块程序地址空间。覆盖图管理代码负责将 section 的拷入和拷出。考虑这种情况，当某存储块的访问速度比其他存储块要快时，那么如果将 section 拷到该存储块来执行或访问，那么速度将会有所提高，覆盖图描述就很适合这种情形。文法如下，

```
SECTIONS {
...
```

```
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]

...
}
```

由以上文法可以看出，同一覆盖图内的 section 具有相同的 VMA。SECNAME2 的 LMA 为 SECTNAME1 的 LMA 加上 SECNAME1 的大小，同理计算 SECNAME2,3,4...的 LMA。SECNAME1 的 LMA 由 LDADDR 决定，如果它没有被指定，那么由 START 决定，如果它也没有被指定，那么由当前定位符号的值决定。

NOCROSSREFS 关键字指定各 section 之间不能交叉引用，否则报错。

对于 OVERLAY 描述的每个 section，连接器将定义两个符号 __load_start_SECNAME 和 __load_stop_SECNAME，这两个符号的值分别代表 SECNAME section 的 LMA 地址的开始和结束。

连接器处理完 OVERLAY 描述语句后，将定位符号的值加上所有覆盖图内 section 大小的最大值。

看个例子吧，

```
SECTIONS{
...

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

.text0 section 和 .text1 section 的 VMA 地址是 0x1000，.text0 section 加载于地址 0x4000，.text1 section 紧跟在其后。

程序代码，拷贝.text1 section 代码，

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

8. 内存区域命令

注意：以下存储区域指的是在程序地址空间内的。
在默认情形下，连接器可以为 section 分配任意位置的存储区域。你也可以用 MEMORY 命令定义存储区域，并通过输出 section 描述的> REGION 属性显示地将该输出 section 限定于某块存储区域，当存储区域大小不能满足要求时，连接器会报告该错误。
MEMORY 命令的文法如下，
MEMORY {
NAME1 [(ATTR)] : ORIGIN = ORIGIN1, LENGTH = LEN2
NAME2 [(ATTR)] : ORIGIN = ORIGIN2, LENGTH = LEN2
...
}
NAME ：存储区域的名字，这个名字可以与符号名、文件名、section 名重复，因为它处于一个独立的名字空间。
ATTR ：定义该存储区域的属性，在讲述 SECTIONS 命令时提到，当某输入 section 没有在 SECTIONS 命令内引用时，连接器会把该输入 section 直接拷贝成输出 section，然后将该输出 section 放入内存区域内。如果设置了内存区域设置了 ATTR 属性，那么该区域只接受满足该属性的 section(怎么判断该 section 是否满足？输出 section 描述内好象没有记录该 section 的读写执行属性)。ATTR 属性内可以出现以下 7 个字符，
R 只读 section
W 读/写 section
X 可执行 section
A ‘可分配的’section
I 初始化了的 section
L 同 I
! 不满足该字符之后的任何一个属性的 section
ORIGIN ： 关键字，区域的开始地址，可简写成 org 或 o
LENGTH ： 关键字，区域的大小，可简写成 len 或 l

例子，
MEMORY
{
rom (rx) : ORIGIN = 0, LENGTH = 256K
ram (!rx) : org = 0x40000000, l = 4M
}
此例中，把在 SECTIONS 命令内*未*引用的且具有读属性或写属性的输入 section 放入 rom 区域内，把其他未引用的输入 section 放入 ram。如果某输出 section 要被放入某内存区域内，而该输出 section 又没有指明 ADDRESS 属性，那么连接器将该输出 section 放在该区域内下一个能使用位置。

9. PHDRS 命令

该命令仅在产生 ELF 目标文件时有效。
ELF 目标文件格式用 program headers 程序头(程序头内包含一个或多个 segment 程序段描述)来描述程序如何被载入内存。可以用 objdump -p 命令查看。
当在本地 ELF 系统运行 ELF 目标文件格式的程序时，系统加载器通过读取程序头信息以知道如何将程序加载到内存。要了解系统加载器如何解析程序头，请参考 ELF ABI 文档。
在连接脚本内不指定 PHDRS 命令时，连接器能够很好的创建程序头，但是有时需要更精确的描述程序头，那么 PAHDRS 命令就派上用场了。
注意：一旦在连接脚本内使用了 PHDRS 命令，那么连接器**仅会**创建 PHDRS 命令指定的信息，所以使用时须谨慎。

PHDRS 命令文法如下，
PHDRS
{
NAME TYPE [FILEHDR] [PHDRS] [AT (ADDRESS)]
[FLAGS (FLAGS)] ;
}
其中 FILEHDR、PHDRS、AT、FLAGS 为关键字。
NAME ：为程序段名，此名字可以与符号名、section 名、文件名重复，因为它在一个独立的名字空间内。此名字只能在 SECTIONS 命令内使用。
一个程序段可以由多个‘可加载’的 section 组成。通过输出 section 描述的属性:PHDRS 可以将输出 section 加入一个程序段，：PHDRS 中的 PHDRS 为程序段名。在一个输出 section 描述内可以多次使用:PHDRS 命令，也即可以将一个 section 加入多个程序段。
如果在一个输出 section 描述内指定了:PHDRS 属性，那么其后的输出 section 描述将默认使用该属性，除非它也定义了:PHDRS 属性。显然当多个输出 section 属于同一程序段时可简化书写。
在 TYPE 属性后存在 FILEHDR 关键字，表示该段包含 ELF 文件头信息；存在 PHDRS 关键字，表示该段包含 ELF 程序头信息。
TYPE 可以是以下八种形式，
PT_NULL 0
表示未被使用的程序段
PT_LOAD 1
表示该程序段在程序运行时应该被加载
PT_DYNAMIC 2
表示该程序段包含动态连接信息
PT_INTERP 3
表示该程序段内包含程序加载器的名字，在 linux 下常见的程序加载器是 ld-linux.so.2
PT_NOTE 4
表示该程序段内包含程序的说明信息
PT_SHLIB 5
一个保留的程序头类型，没有在 ELF ABI 文档内定义
PT_PHDR 6
表示该程序段包含程序头信息。
EXPRESSION 表达式值
以上每个类型都对应一个数字，该表达式定义一个用户自定的程序头。
AT(ADDRESS)属性定义该程序段的加载位置(LMA)，该属性将**覆盖**该程序段内的 section 的 AT()属性。
默认情况下，连接器会根据该程序段包含的 section 的属性(什么属性？好象在输出 section 描述内没有看到)设置 FLAGS 标志，该标志用于设置程序段描述的 p_flags 域。
下面看一个典型的 PHDRS 设置，
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) } /* defaults to :text */
...
. = . + 0x1000; /* move to a new page in memory */
.data : { *(.data) } :data
.dynamic : { *(.dynamic) } :data :dynamic
...
}
```

10. 版本号命令

当使用 ELF 目标文件格式时，连接器支持带版本号的符号。
读者可以发现仅仅在共享库中，符号的版本号属性才有意义。
动态加载器使用符号的版本号为应用程序选择共享库内的一个函数的特定实现版本。
可以在连接脚本内直接使用版本号命令，也可以将版本号命令实现于一个特定版本号描述文件(用连接选项--version-script 指定该文件)。
该命令的文法如下，
VERSION { version-script-commands }
以下内容直接拷贝于以前的文档，
========== 开 始 ==========
内容简介
---
0 前提
1 带版本号的符号的定义
2 连接到带版本的符号
3 GNU 扩充
4 我的疑问
5 英文搜索关键字
6 我的参考

0. 前提

-- 只限于 ELF 文件格式
-- 以下讨论用 gcc

1. 带版本号的符号的定义(共享库内)

文件 b.c 内容如下，
```
int old_true()
{
return 1;
}

int new_true()
{
return 2;
```

```
}
```

写连接器的版本控制脚本，本例中为 b.lds，内容如下

```
VER1.0{
new_true;
};
VER2.0{
};
```

```
$gcc -c b.c
$gcc -shared -Wl,--version-script=b.lds -o libb.so b.o
```

可以在{}内填入要绑定的符号，本例中 new_true 符号就与 VER1.0 绑定了。
那么如果有一个应用程序连接到该库的 new_true 符号，那么它连接的就是 VER1.0 版本的 new_true 符号

如果把 b.lds 更改为，

```
VER1.0{
};
VER2.0{
new_true;
};
```

然后在生成 libb.so 文件，在运行那个连接到 VER1.0 版本的 new_true 符号的应用程序，可以发现该应用程序不能运行了，
因为库内没有 VER1.0 版本的 new_true，只有 VER2.0 版本的 new_true。

2. 连接到带版本的符号
写一个简单的应用(名为 app)连接到 libb.so，应用符号 new_true
假设 libb.so 的版本控制文件为，

```
VER1.0{
};
VER2.0{
new_true;
};
```

```
$ nm app | grep new_true
U new_true@@VER1.0
$
```

用 nm 命令发现 app 连接到 VER1.0 版本的 new_true

3. GNU 的扩充
它允许在程序文件内绑定 *符号* 到 *带版本号的别名符号*

文件 b.c 内容如下，

```
int old_true()
{
return 1;
}
```

```
int new_true()
{
return 2;
}
__asm__( ".symver old_true,true@VER1.0" );
__asm__( ".symver new_true,true@@VER2.0" );
```

其中，带版本号的别名符号是 true，其默认的版本号为 VER2.0

供连接器用的版本控制脚本 b.lds 内容如下，

```
VER1.0{
};
VER2.0{
};
```

版本控制文件内必须包含版本 VER1.0 和版本 VER2.0 的定义，因为在 b.c 文件内有对他们的引用

****** 假定 libb.so 与 app.c 在同一目录下 ********

以下应用程序 app.c 连接到该库，

```
int true();
int main()
{
printf( "%d ", true );
}
```

```
$ gcc app.c libb.so
$ LD_LIBRARY_PATH=. ./app
2
$ nm app | grep true
U true@@VER2.0
$
```

很明显，程序 app 使用的是 VER2.0 版本的别名符号 true，如果在 b.c 内没有指明别名符号 true 的默认版本，
那么 gcc app.c libb.so 将出现连接错误，提示 true 没有定义。

也可以在程序内指定特定版本的别名符号 true，程序如下，

```
__asm__( ".symver true,true@VER1.0" );
int true();
int main()
{
printf( "%d ", true );
}
```

```
$ gcc app.c libb.so
$ LD_LIBRARY_PATH=. ./app
1
```

```
$ nm app | grep true
U true@VER1.0
$
```

显然，连接到了版本号为 VER1.0 的别名符号 true。其中只有一个@表示，该版本不是默认的版本

我的疑问：
版本控制脚本文件中，各版本号节点之间的依赖关系

英文搜索关键字：
.symver
versioned symbol
version a shared library

参考：
info ld, Scripts node
========== 结 束 ==========

11. 表达式

表达式的文法与 C 语言的表达式文法一致，表达式的值都是整型，如果 ld 的运行主机和生成文件的目标机都是 32 位，则表达式是 32 位数据，否则是 64 位数据。
能够在表达式内使用符号的值，设置符号的值。
下面看六项表达式相关内容，

常表达式：
```
_fourk_1 = 4K; /* K、M 单位 */
_fourk_2 = 4096; /* 整数 */
_fourk_3 = 0x1000; /* 16 进位 */
_fourk_4 = 01000; /* 8 进位 */
```
1K=1024 1M=1024*1024
符号名：
没有被引号""包围的符号，以字母、下划线或'.'开头，可包含字母、下划线、'.'和'-'。当符号名被引号包围时，符号名可以与关键字相同。如，
```
"SECTION"=9
"with a space" = "also with a space" + 10;
```
定位符号'.'：
只在 SECTIONS 命令内有效，代表一个程序地址空间内的地址。
注意：当定位符用在 SECTIONS 命令的输出 section 描述内时，它代表的是该 section 的当前**偏移**，而不是程序地址空间的绝对地址。
先看个例子，
```
SECTIONS
{
```

```
output :
{
file1(.text)
. = . + 1000;
file2(.text)
. += 1000;
file3(.text)
} = 0x1234;
}
```
其中由于对定位符的赋值而产生的空隙由 0x1234 填充。其他的内容应该容易理解吧。
再看个例子，
```
SECTIONS
{
. = 0x100
.text: {
*(.text)
. = 0x200
}
. = 0x500
.data: {
*(.data)
. += 0x600
}
} .text section 在程序地址空间的开始位置是 0x
```
表达式的操作符：
与 C 语言一致。
优先级 结合顺序 操作符
```
1 left ! - ~ (1)
2 left * / %
3 left + -
4 left >> <<
5 left == != > < <= >=
6 left &
7 left |
8 left &&
9 left ||
10 right ? :
11 right &= += -= *= /= (2)
```
(1)表示前缀符，(2)表示赋值符。
表达式的计算：
连接器延迟计算大部分表达式的值。
但是，对待与连接过程紧密相关的表达式，连接器会立即计算表达式，如果不能计算则报错。比如，对于 section 的 VMA 地址、内存区域块的开始地址和大小，与其相关的表达式应该立即被计算。
例子，
```
SECTIONS
{
.text 9+this_isnt_constant :
{ *(.text) }
}
```
这个例子中，9+this_isnt_constant 表达式的值用于设置.text section 的 VMA 地址，因此需要立即

运算，但是由于 this_isnt_constant 变量的值不确定，所以此时连接器无法确立表达式的值，此时连接器会报错。
相对值与绝对值：
在输出 section 描述内的表达式，连接器取其相对值，相对与该 section 的开始位置的偏移
在 SECTIONS 命令内且非输出 section 描述内的表达式，连接器取其绝对值
通过 ABSOLUTE 关键字可以将相对值转化成绝对值，即在原来值的基础上加上表达式所在 section 的 VMA 值。
例子，

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

该例子中，_edata 符号的值是.data section 的末尾位置(绝对值，在程序地址空间内)。
内建函数：
ABSOLUTE(EXP) ：转换成绝对值
ADDR(SECTION) ：返回某 section 的 VMA 值。
ALIGN(EXP) ：返回定位符'.'的修调值，对齐后的值，(. + EXP - 1) & ~(EXP - 1)
BLOCK(EXP) ：如同 ALIGN(EXP)，为了向前兼容。
DEFINED(SYMBOL) ：如果符号 SYMBOL 在全局符号表内，且被定义了，那么返回 1，否则返回 0。
例子，

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

LOADADDR(SECTION) ：返回三 SECTION 的 LMA
MAX(EXP1,EXP2) ：返回大者
MIN(EXP1,EXP2) ：返回小者
NEXT(EXP) ：返回下一个能被使用的地址，该地址是 EXP 的倍数，类似于 ALIGN(EXP)。除非使用了 MEMORY 命令定义了一些非连续的内存块，否则 NEXT(EXP)与 ALIGH(EXP)一定相同。
SIZEOF(SECTION) ：返回 SECTION 的大小。当 SECTION 没有被分配时，即此时 SECTION 的大小还不能确定时，连接器会报错。
SIZEOF_HEADERS ：
sizeof_headers ：返回输出文件的文件头大小(还是程序头大小)，用以确定第一个 section 的开始地址(在文件内)。？？？

12. 暗含的连接脚本

输入文件可以是目标文件，也可以是连接脚本，此时的连接脚本被称为 暗含的连接脚本
如果连接器不认识某个输入文件，那么该文件被当作连接脚本被解析。更进一步，如果发现它的格式又不是连接脚本的格式，那么连接器报错。
一个暗含的连接脚本不会替换默认的连接脚本，仅仅是增加新的连接而已。
一般来说，暗含的连接脚本符号分配命令，或 INPUT、GROUP、VERSION 命令。
在连接命令行中，每个输入文件的顺序都被固定好了，暗含的连接脚本在连接命令行内占住一个位置，这个位置决定了由该连接脚本指定的输入文件在连接过程中的顺序。
典型的暗含的连接脚本是 libc.so 文件，在 GNU/linux 内一般存在/usr/lib 目录下。

References

1, gnu ld在线手册

2, 程序的链接和装入及Linux下动态链接的实现

3, UNIX/Linux平台可执行文件格式分析

4, John R. Levine.《Linkers & Loaders》