aws ~ ssh aws@free-tier

only_eVonne
他日若得凌云志，敢笑黄巢不丈夫

随笔 - 129 文章 - 0 评论 - 15

[转]Linux下的lds链接脚本详解

转载自：http://linux.chinaunix.net/techdoc/beginner/2009/08/12/1129972.shtml

一、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.

二、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

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0×08050000, 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0×08050000, 产生的printf指令是将地址为0×08050004处的4字节内容作为一个整数打印出来。

如果.data section的LMA为0×08050000, 显然结果是j=2

如果.data section的LMA为0×08050004, 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 0×08048285

movl $0×1,%eax

如果.text section的LMA为0×08048280, 那么在进程地址空间内0×08048280处为"jmp 0×08048285"指令, 0×08048285处为movl $0×1,%eax指令. 假设某指令跳转到地址0×08048280, 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0×08048285, 那么在进程地址空间内0×08048285处为"jmp 0×08048285"指令, 0×0804828a处为movl $0×1,%eax指令. 假设某指令跳转到地址0×08048285, 显然它的执行又跳转到进程地址空间内0×08048285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)

三、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。

四、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:

以下脚本将输出文件的text section定位在0×10000, data section定位在0×8000000:

```
SECTIONS
{
. = 0×10000;
.text : { *(.text) }
. = 0×8000000;
.data : { *(.data) }
.bss : { *(.bss) }
}
```

解释一下上述的例子:

. = 0×10000 : 把定位器符号置为0×10000 (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0×10000.

. = 0×8000000 : 把定位器符号置为0×8000000

公告

昵称: only_eVonne
园龄: 9年4个月
粉丝: 89
关注: 2
+加关注

< 2020年12月 >

日	一	二	三	四	五	六
29	30	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

搜索

找找看
谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

随笔分类

Android(1)
Linux编程(70)
Nginx源码(8)
QT编程(4)
STL学习(1)
程序人生(4)
多媒体音视频(1)
数据结构(11)
网络编程(12)
语言之美(12)

随笔档案

2014年11月(1)
2014年8月(1)
2014年7月(1)
2014年6月(1)
2013年11月(3)
2013年7月(7)
2013年5月(1)
2013年4月(17)
2013年3月(13)
2013年2月(7)
2012年6月(1)
2012年5月(4)
2012年4月(3)
2012年3月(4)
2012年2月(7)
更多

最新评论

1. Re:TCP协议通讯工作原理
赞
--烟花散尽13141
2. Re:[转]Linux下的lds链接脚本详解
我能说啥了
--进入代码内心的男人
3. Re:解决wireshark检测不到网卡的问题
解决问题
--joybear
4. Re:什么是A类、B类、C类地址？
b类的私有地址应该是 172.16.0.0～172.31.255.255吧
--萤火虫塔莉
5. Re:TCL脚本语言基础介绍
mark
--郝姬友

阅读排行榜

1. linux下使用tar命令(335424)
2. 什么是A类、B类、C类地址？(78414)
3. 解决wireshark检测不到网卡的问题(52098)

.data : { *(.data) } : 将所有输入文件的.data section合并成一个.data section, 该section的地址被置为0×8000000.
.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0×8000000+.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.

五、简单脚本命令

ENTRY(SYMBOL) : 将符号SYMBOL的值设置成入口地址。
入口地址(entry point)是指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
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.
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.
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。
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
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。
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。
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.
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.

六、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.
举例. 通过下面的程序查看变量a的地址:
a.c文件

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

a.lds文件

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

编译命令:

```
$ gcc -Wall -o a-without-lds.exe a.c
```

运行结果:

```
&a = 0×601020
```

编译命令:

```
$ gcc -Wall -o a-with-lds.exe a.c a.lds
```

运行结果:

```
&a = 0×3
```

注意: 对符号的赋值只对全局变量起作用!
对于一些简单的赋值语句, 我们可以使用任何c语言语法的赋值操作:

```
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已经被在某目标文件的源码中被定义。
. 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。
示例1:

```
floating_point = 0; /* 全局位置 */
SECTIONS
{
```


4. linux下core文件调试方法(50777)
5. IP头, TCP头, UDP头, MAC帧头定义(45957)

评论排行榜

1. TCL脚本语言基础介绍(3)
2. popen函数的实现(3)
3. 什么是A类、B类、C类地址？(2)
4. [转]Linux下的lds链接脚本详解(1)
5. 解决wireshark检测不到网卡的问题(1)

推荐排行榜

1. linux下使用tar命令(16)
2. fork与vfork的区别(6)
3. [转]Linux下的lds链接脚本详解(5)
4. IP头, TCP头, UDP头, MAC帧头定义(5)
5. TCL脚本语言基础介绍(5)

```
.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。

示例2：

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

这里，当目标文件内引用了etext符号，却没有定义它时，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。

七、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。

7.1、输出section描述（基本）

输出section描述具有如下格式：

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

[ ]内的内容为可选选项，一般不需要.

SECTION-NAME：section名字.SECTION-NAME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。

7.1.1、输出section名字

输出section名字SECTION-NAME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。

7.1.2、输出section地址

输出section地址[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，这时输出 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对齐要求。

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(0×10)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。

7.1.3、输出section描述

输出section描述OUTPUT-SECTION-COMMAND为以下四种之一：

(1).符号赋值语句

(2).输入section描述

(3).直接包含的数据值

(4).一些特殊的输出section关键字

7.1.3.1、符号赋值语

符号赋值语句已经在《Linux下的lds链接脚本基础(一)》前文介绍过，这里就不累述。

7.1.3.2、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))
最后我们看个简单的输入section相关例子：
```
SECTIONS {
outputa 0×10000 :
{
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outputasection内，该section的VMA是0×10000；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. input1section和.input2 section放入输出outputc section内。

7.1.3.3、直接包含数据值
可以显示地在输出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(0×9090)。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！！！

7.1.3.4、特殊的输出section关键字

在输出section描述OUTPUT-SECTION-COMMAND中还可以使用一些特殊的输出section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。
修改定位器
我们可以对定位器符合。进行赋值来修改定位器的值。
示例

```
SECTIONS
{
. = SIZEOF_HEADERS;
.text : { *(.text) }
. = 0×10000;
.data : { *(.data) }
. = 0×8000000;
.bss : { *(.bss) }
}
```

输出section的丢弃
对于.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。
另外，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。

7.2、输出section描述(进阶)

我们再回顾以下输出section描述的文法：

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

前面我们介绍了SECTION、ADDRESS、OUTPUT-SECTION-COMMAND相关信息，下面我们将介绍其他属性。

7.2.1、输出section的类型

可以通过[(TYPE)]设置输出section的类型。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。
默认值是多少呢？Puzzle!

7.2.2、输出section的LMA

默认情况下，LMA等于VMA，但可以通过[AT(LMA)]项，即关键字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 0×1000 : {_etext = . ;*(.text); }
.mdata 0×2000 :
AT ( ADDR (.text) + SIZEOF (.text) )
{ _data = . ; *(.data); _edata = . ; }
.bss 0×3000 :
{ _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 rom }
```

7.2.3、设置输出section所在的程序段

可以通过[:PHDR HDR ...]项将输出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

7.2.4、设置输出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。我们可以通过[=FILLEXP]项设置填充值。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) } =0×9090 }
```

7.3、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。这里VMA由[START] 决定。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关键字说明各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。

示例：

```
SECTIONS{
…
OVERLAY 0×1000 : AT (0×4000)
{
.text0 { o1/*.o(.text) }
.text1 { o2/*.o(.text) }
}
…
}
```

.text0 section和.text1 section的VMA地址是0×1000，.text0 section加载于地址0×4000，.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 *) 0×1000, &__load_start_text1,&__load_stop_text1 – &__load_start_text1);
```

八、内存区域命令

在默认情形下，连接器可以为section在程序地址空间内分配任意位置的存储区域。并通过输出section描述的 > REGION属性显示地将该输出section限定于在程序地址空间内的某块存储区域，当存储区域大小不能满足要求时，连接器会报告该错误。

你也可以用MEMORY命令让在SECTIONS命令内*未*引用的selection分配在程序地址空间内的某个存储区域内。

注意：以下存储区域指的是在程序地址空间内的。

MEMORY命令的文法如下，

```
MEMORY {
NAME1 [(ATTR)] : ORIGIN = ORIGIN1, LENGTH = LEN1
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

示例

```
MEMORY
{
rom (rx) : ORIGIN = 0, LENGTH = 256K
ram (!rx) : org = 0×40000000, 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。

九、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。

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

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。

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。

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

示例

```
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 */
…
. = . + 0×1000; /* move to a new page in memory */
.data : { *(.data) } :data
.dynamic : { *(.dynamic) } :data :dynamic
…
}
```

十、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。版本号也只限于ELF文件格式。

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 }

以下讨论用gcc

10.1. 带版本号的符号的定义(共享库内)

文件b.c内容如下，

```
int getVersion()
{
```

```
return 1;
}
```
写连接器的版本控制脚本，本例中为b.lds，内容如下
```
VER1.0{
getVersion;
};
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getVersion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getVersion符号，那么它连接的就是VER1.0版本的getVersion符号
如果我们对b.c文件进行了升级，更改如下：
```
int getVersion()
{
return 101;
}
```
这里我对getVersion()进行了更改，其返回值的意义也进行改变，也就是它和前不兼容：
为了程序的安全，我们把b.lds更改为，
```
VER1.0{
};
VER2.0{
getVersion;
};
```
然后生成新的libb.so文件。
这时如果我们运行app.exe(它已经连接到VER1.0版本的getVersion())，就会发现该应用程序不能运行了。
提示信息如下：
```
./app.exe: relocation error: ./app.exe: symbol getVersion, version VER1.0 not defined in file libb.so with link time reference
```
因为库内没有VER1.0版本的getVersion()，只有VER2.0版本的getVersion()。

10.2、参看连接的符号的版本
对上面生成的app.exe执行以下命令：
```
nm app.exe | grep getVersion
```
结果
```
U new_true@@VER1.0
```
用nm命令发现app连接到VER1.0版本的getVersion

10.3、GNU的扩充
在GNU中，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_getVersion()
{
return 1;
}
int new_getVersion()
{
return 101;
}
__asm__(".symver old_getVersion,getVersion@VER1.0");
__asm__(".symver new_getVersion,getVersion@@VER2.0");
```
其中，对于VER1.0版本号的getVersion别名符号是old_getVersion；
对于VER2.0版本号的getVersion别名符号是new_getVersion，
在连接时，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
再次执行以下命令编译连接b.c和app.c
```
gcc -c src/b.c
gcc -shared -Wl,--version-script=./lds/b.lds -o libb.so b.o
gcc -o app.exe ./src/app.c libb.so
```
运行：
```
./app.exe
```
结果：
```
Version=0x65
```
说明app.exe的确是连接的VER2.0的getVersion，即new_getVersion()

我们再对app.c进行修改，以使它连接的VER1.0的getVersion，即old_getVersion()
app.c文件：
```
#include <stdio.h>
__asm__(".symver getVersion,getVersion@VER1.0");
extern int getVersion();
int main()
{
printf("Version=%p\n", getVersion());
return 0;
}
```
再次编译连接b.c和app.c
运行：
```
./app.exe
```
结果：
```
Version=0x1
```
说明此次app.exe的确是连接的VER1.0的getVersion，即old_getVersion()

十一、表达式
lds中表达式的文法与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。
以下是一些常用的表达式：
```
_fourk_1 = 4K; /* K、M单位 */
```

```
_fourk_2 = 4096; /* 整数 */
_fourk_3 = 0×1000; /* 16 进位 */
_fourk_4 = 01000; /* 8 进位 */
```
注意:1K=1024 1M=1024*1024

11.1、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;
```

11.2、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:在连接时，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。当然当程序载入后，符号最后的地址还是程序地址空间的绝对地址。

示例11.2_1：

```
SECTIONS
{
output :
{
file1(.text)
. = . + 1000;
file2(.text)
. += 1000;
file3(.text)
} = 0×1234;
}
```

其中由于对定位符的赋值而产生的空隙由0×1234填充。其他的内容应该容易理解吧。

示例11.2_2：

```
SECTIONS
{
. = 0×100
.text: {
*(.text)
. = 0×200
}
. = 0×500
.data: {
*(.data)
. += 0×600
}
}
```

.text section在程序地址空间的开始位置是0x100

示例11.2_3

文件src\a.c

```
#include <stdio.h>
int a = 100;
int b=0;
int c=0;
int d=1;
int main()
{
printf( "&a=%p\n", &a );
printf( "&b=%p\n", &b );
printf( "&c=%p\n", &c );
printf( "&d=%p\n", &d );
return 0;
}
```

文件lds\a.lds

```
a = 10; /* 全局位置 */
SECTIONS
{
b = 11;
.text :
{
*(.text)
c = .; /* section描述内 */
. = 10000;
d = .;
}
_bdata = (. + 3) & ~ 4; /* SECTIONS命令内 */
.data : { *(.data) }
}
```

在没有使用a.lds情况下编译

```
gcc -Wall -o a-without-lds.exe ./src/a.c
```

运行./a-without-lds.exe

结果:

```
&a=0x601020
&b=0x601038
&c=0x60103c
&d=0x601024
```

在使用a.lds情况下编译

```
gcc -Wall -o a-with-lds.exe ./src/a.c ./lds/a.lds
```

运行./a-with-lds.exe

结果:

```
&a=0xa
&b=0xb
&c=0x400638
&d=0x402b20
```

10.3、表达式的操作符

在lds中，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 right ? :
10 right &= += -= *= /= (2)
(1)表示前缀符，(2)表示赋值符。

10.4、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。

10.5、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。
示例
```
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)。

10.6、内建函数
lds中有以下一些内建函数：
ABSOLUTE(EXP)：转换成绝对值
ADDR(SECTION)：返回某section的VMA值。
ALIGN(EXP)：返回定位符'.'的按照EXP进行对齐后的修调值，对齐后的修调值算法为:(. + 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。
示例：
```
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：返回输出文件头部的字节数。这些信息出现在输出文件的开始处。当设置第一个段的开始地址时，你可以使用这个数字。如果你选择了加速分页，当产生一个ELF输出文件时，如果链接器脚本使用SIZEOF_HEADERS内建函数，连接器必须在它
算出所有段地址和长度之前计算程序头部的数值。如果连接器后来发现它需要附加程序头，它将报告一个“not enough room for
program headers”错误。为了避免这样的错误，你必须避免使用SIZEOF_HEADERS函数，或者你必须修改你的连接器脚本去避免强制
连接器去使用附加程序头，或者你必须使用PHDRS命令去定义你自己的程序头

十二、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。

分类: Linux编程

好文要顶 关注我 收藏该文


only_eVonne
关注 - 2
粉丝 - 89
+加关注

5 推荐 0 反对

« 上一篇: Intent用法简介
posted @ 2014-11-19 12:34 only_eVonne 阅读(38363) 评论(1) 编辑 收藏

评论列表

#1楼 2019-09-02 16:00 进入代码内心的男人

我能说啥了

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

登录后才能发表评论，立即 登录 或 注册，访问 网站首页

写给园友们的一封求助信

最新 IT 新闻:

- Rust 语言正吸引更多项目使用
- Google 加强对科学家发表敏感主题论文的审查
- 纽约州暂停在学校使用面部识别
- Telegram 用户数接近五亿，准备引入广告
- 宝马将用智能广告牌公开羞辱过保的司机

» 更多新闻...

Copyright © 2020 only_eVonne
Powered by .NET 5.0.1-servicing.20575.16 on Kubernetes