SECTIONS 命令

'SECTIONS'命令告诉连接器如何把输入节映射到输出节，并如何把输出节放入到内存中.

'SECTIONS'命令的格式如下:

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

每一个 SECTIONS-COMMAND 可能是如下的一种:

* 一个'ENTRY'命令.
* 一个符号赋值.
* 一个输出节描述.
* 一个重叠描述.

'ENTRY'命令和符号赋值在'SECTIONS'命令中是允许的，这是为了方便在这些命令中使用定位计数器. 这也可以让连接脚本更容易理解，因为你可以在更有意义的地方使用这些命令来控制输出文件的布局.

输出节描述在下面描述.

如果你在连接脚本中不使用'SECTIONS'命令，连接器会按在输入文件中遇到的节的顺序把每一个输入节放到同名的输出节中. 如果所有的输入节都在第一个文件中存在,那输出文件中的节的顺序会匹配第一个输入文件中的节的顺序. 第一个节会在地址零处.

输出节描述

一个完整的输出节的描述应该是这个样子的:

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

大多数输出节不使用这里的可选节属性.

SECTION 边上的空格是必须的, 所以节名是明确的. 冒号跟花括号也是必须的. 断行和其他的空格是可选的.

SECTION 输出节名.

输出节的名字是 SECTION. SECTION 必须满足你的输出格式的约束. 在一个只支持限制数量的节的格式中,比如'a.out',这个名字必须是格式支持的节名中的一个(比如, 'a.out'只允许'.text', '.data'或'.bss').如果输出格式支持任意数量的节, 但是只支持数字,而没有名字(就像 Oasys 中的情况), 名字应当以一个双引号中的数值串的形式提供.一个节名可以由任意数量的字符组成,但是一个含有任意非常用字符(比如逗号)的字句必须用双引号引起来.

ADDRESS

ADDRESS 是关于输出节中 VMS 的一个表达式.

如果你不提供 ADDRESS, 连接器会基于 REGION(如果存在)设置它,或者基于定位计数器的当前值.

如果你提供了 ADDRESS, 那输出节的地址会被精确地设为这个值.

如果你既不提供 ADDRESS 也不提供 REGION, 那输出节的地址会被设为当前的定位计数器向上对齐到输出节需要的对齐边界的值. 输出节的对齐要求是所有输入节中含有的对齐要求中最严格的一个.

比如:

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

和

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

有细微的不同. 第一个会把'.text'输出节的地址设为当前定位计数器的值.

第二个会把它设为定位计数器的当前值向上对齐到'.text'输入节中对齐要求最严格的一个边界.

ADDRESS 可以是任意表达式; 比如,如果你需要把节对齐对 0x10 字节边界,这样就可以让低四字节的节地址值为零, 你可以这样做:

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

这个语句可以正常工作,因为'ALIGN'返回当前的定位计数器,并向上对齐到指定的值.

指定一个节的地址会改变定位计数器的值.

输出节类型 TYPE

.....

每一个输出节可以有一个类型。类型是一个放在括号中的关键字，已定义的类型如下所示：
`NOLOAD'
这个节应当被标式讵不可载入，所以当程序运行时，它不会被载入到内存中。

`DSECT'
`COPY'
`INFO'
`OVERLAY'
支持这些类型名只是为了向下兼容，它们很少使用。它们都具有相同的效果：这个节应当被标式讵不可分配，所以当程序运行时，没有内存为这个节分配。

连接器通常基于映射到输出节的输入节来设置输出节的属性。你可以通过使用节类型来重设这个属性，比如，在下面的脚本例子中，‘ROM’节被定址在内存地址零处，并且在程序运行时不需要被载入。
‘ROM’节的内容会正常出现在连接输出文件中。

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

输出节 LMA

.....

每一个节有一个虚地址（VMA）和一个载入地址（LMA）；出现在输出节描述中的地址表达式设置 VMS

连接器通常把 LMA 跟 VMA 设成相等。你可以通过使用‘AT’关键字改变这个。跟在关键字‘AT’后面的表达式 LMA 指定节的载入地址。或者，通过`AT>LMA_REGION'表达式， 你可以为节的载入地址指定一个内存区域。

这个特性是为了便于建立 ROM 映像而设计的。比如，下面的连接脚本创建了三个输出节：一个叫做‘.text’从地址‘0x1000’处开始，一个叫‘.mdata’，尽管它的 VMA 是'0x2000'，它会被载入到'.text'节的后面，最后一个叫做‘.bss’是用来放置未初始化的数据的，其地址从'0x3000'处开始。符号'_data'被定义为值
'0x2000'，它表示定位计数器的值是 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 = . ;}
}
```

这个连接脚本产生的程序使用的运行时初始化代码会包含象下面所示的一些东西，以把初始化后的数据从 ROM
映像中拷贝到它的运行时地址中去。注意这节代码是如何利用好连接脚本定义的符号的。

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

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

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

OUTPUT-SECTION-COMMAND

每一个 OUTPUT-SECTION-COMMAND 可能是如下的情况:

* 一个符号赋值.
* 一个输入节描述.
* 直接包含的数据值.
* 一个特定的输出节关键字.

符号赋值

见链接脚本.pdf

输入节描述

最常用的输出节命令是输入节描述.

输入节描述是最基本的连接脚本操作. 你使用输出节来告诉连接器在内存中如何布局你的程序. 你使用输入节来告诉连接器如何把输入文件映射到你的内存中.

输入节基础

一个输入节描述由一个文件名后跟有可选的括号中的节名列表组成.

文件名和节名可以通配符形式出现, 这个我们以后再介绍.

最常用的输入节描述是包含在输出节中的所有具有特定名字的输入节. 比如, 包含所有输入'.text'节,你可以
这样写:

```
*(.text)
```

这里,'*'是一个通配符,匹配所有的文件名. 为把一部分文件排除在匹配的名字通配符之外, EXCLUDE_FILE 可以用来匹配所有的除了在 EXCLUDE_FILE 列表中指定的文件.比如:

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

会让除了`crtend.o'文件和`otherfile.o'文件之外的所有的文件中的所有的.ctors 节被包含进来.

有两种方法包含多于一个的节:

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

上面两句的区别在于'.text'和'.rdata'输入节的输出节中出现的顺序不同. 在第一个例子中, 两种节会交替出现,并以连接器的输入顺序排布. 在第二个例子中,所有的'.text'输入节会先出现,然后是所有的'.rdata'节.

你可以指定文件名,以从一个特定的文件中包含节. 如果一个或多个你的文件含有特殊的数据在内存中需要特殊的定位,你可以这样做. 比如:

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

如果你使用一个不带有节列表的文件名, 那输入文件中的所有的节会被包含到输出节中. 通常不会这样做, 但是在某些场合下这个可能非常有用. 比如:

```
data.o
```

当你使用一个不含有任何通配符的文件名时, 连接器首先会查看你是否在连接命令行上指定了文件名或者在'INPUT'命令中. 如果你没有, 连接器会试图把这个文件作为一个输入文件打开, 就像它在命令行上出现一样.
注意这跟'INPUT'命令不一样, 因为连接器会在档案搜索路径中搜索文件.

输入节通配符

在一个输入节描述中, 文件名或者节名,或者两者同时都可以是通配符形式.

文件名通配符'*'在很多例子中都可以看到,这是一个简单的文件名通配符形式.

通配符形式跟 Unix Shell 中使用的一样.

`*'
匹配任意数量的字符.

`?'
匹配单个字符.

`[CHARS]'
匹配 CHARS 中的任意单个字符; 字符'-'可以被用来指定字符的方讧, 比如[a-z]匹配任意小字字符.

`\'
转义其后的字符.

当一个文件名跟一个通配符匹配时, 通配符字符不会匹配一个'/'字符(在 UNIX 系统中用来分隔目录名), 一个含有单个'*'字符的形式是个例外; 它总是匹配任意文件名, 不管它是

否含有'/'. 在一个节名中, 通配符字符会匹配'/'字符.文件名通配符只匹配那些在命令行或在'INPUT'命令上显式指定的文件. 连接器不会通过搜索目录来展开通配
符.

如果一个文件名匹配多于一个通配符, 或者如果一个文件名显式出现同时又匹配了一个通配符, 连接器会使用第一次匹配到的连接脚本. 比如, 下面的输入节描述序列很可能就是错误的,因为'data.o'规则没有被使用:

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

通常, 连接器会把匹配通配符的文件和节按在连接中被看到的顺序放置. 你可以通过'SORT'关键字改变它, 它
出现在括号中的通配符之前(比如, 'SORT(.text*)'). 当'SORT'关键字被使用时, 连接器会在把文件和节放到输出文件中之前按名字顺序重新排列它们.

如果你对于输入节被放置到哪里去了感到很困惑, 那可以使用'-M'连接选项来产生一个位图文件. 位图文件会精确显示输入节是如何被映射到输出节中的.

这 个例子显示了通配符是如何被用来区分文件的. 这个连接脚本指示连接器把所有的'.text'节放到'.text'中, 把所有的'.bss'节放到'.bss'. 连接器会把所有的来自文件名以一个大写字母开始的文件中的'.data'节放进'.DATA'节中; 对于所有其他文件, 连接器会把'.data'节放进'.data'节中.

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

输入节中的普通符号.

对于普通符号,需要一个特殊的标识, 因为在很多目标格式中, 普通符号没有一个特定的输入节. 连接器会把普通符号处理成好像它们在一个叫做'COMMON'的节中.

你可能像使用带有其他输入节的文件名一样使用带有'COMMON'节的文件名。你可以通过这个把来自一个特定输入文件的普通符号放入一个节中，同时把来自其它输入文件的普通符号放入另一个节中。

在大多数情况下，输入文件中的普通符号会被放到输出文件的'.bss'节中。比如：

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

有些目标文件格式具有多于一个的普通符号。比如，MIPS ELF 目标文件格式区分标准普通符号和小普通符号。在这种情况下，连接器会为其他类型的普通符号使用一个不同的特殊节名。 在 MIPS ELF 的情况中， 连接器为标准普通符号使用'COMMON'，并且为小普通符号使用'.common'。这就允许你把不同类型的普通符号映射到内存的不同位置。

在一些老的连接脚本上，你有时会看到'[COMMON]'。这个符号现在已经过时了， 它等效于'*(COMMON)'。

输入节和垃圾收集

当连接时垃圾收集正在使用中时（'--gc-sections'），这在标识那些不应该被排除在外的节时非常有用。这是通过在输入节的通配符入口外面加上'KEEP()'实现的，比如'KEEP(*(.init))'或者'KEEP(SORT(*)(.sorts))
'。

输入节示例

接下来的例子是一个完整的连接脚本。它告诉连接器去读取文件'all.o'中的所有节，并把它们放到输出节'outputa'的开始位置处， 该输出节是从位置'0x10000'处开始的。 从文件'foo.o'中来的所有节'.input1'在同一个输出节中紧密排列。 从文件'foo.o'中来的所有节'.input2'全部放入到输出节'outputb'中，后面跟上从'foo1.o'中来的节'.input1'。来自所有文件的所有余下的'.input1'和'.input2'节被写入到输出节'outputc'中。

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

输出节数据

你可以通过使用输出节命令'BYTE','SHORT','LONG','QUAD','SQUAD'在输出节中显式包含几个字节的数据每一个关键字后面都跟上一个圆括号中的要存入的值。表达式的值被存在当前的定位计数器的值处。

‘BYTE’，‘SHORT’，‘LONG’‘QUAD’命令分别存储一个，两个，四个，八个字节。存入字节后，定位计数器的值加上被存入的字节数。

比如，下面的命令会存入一字节的内容 1,后面跟上四字节，其内容是符号'addr'的值。

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

当使用 64 位系统时，‘QUAD’和‘SQUAD’是相同的；它们都会存储 8 字节，或者说是 64 位的值。而如果软硬件系统都是 32 位的，一个表达式就会被作为 32 位计算。在这种情况下，‘QUAD’存储一个 32 位值，并把它零扩展到 64 位， 而‘SQUAD’会把 32 位值符号扩展到 64 位。

如果输出文件的目标文件格式有一个显式的 endianness，它在正常的情况下，值就会被以这种 endianness 存储，当一个目标文件格式没有一个显式的 endianness 时，值就会被以第一个输入目标文件的 endianness 存储。

注意， 这些命令只在一个节描述内部才有效，而不是在它们之间， 所以，下面的代码会使连接器产生一个错误信息：

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

而这个才是有效的：

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

你可能使用‘FILL’命令来为当前节设置填充样式。它后面跟有一个括号中的表达式。任何未指定的节内内存区域（比如，因为输入节的对齐要求而造成的裂缝）会以这个表达式的值进行填充。一个'FILL'语句会覆盖到它本身在节定义中出现的位置后面的所有内存区域；通过引入多个‘FILL’语句，你可以在输出节的不同位置拥有不同的填充样式。

这个例子显示如何在未被指定的内存区域填充'0x90'：

```
FILL(0x90909090)
```

‘FILL’命令跟输出节的‘=FILLEXP’属性相似，但它只影响到节内跟在‘FILL’命令后面的部分，而不是整个节。如果两个都用到了，那‘FILL’命令优先。

输出节关键字

有两个关键字作为输出节命令的形式出现。

`CREATE_OBJECT_SYMBOLS'
这个命令告诉连接器为每一个输入文件创建一个符号。而符号的名字正好就是相关输入文件的名字。
而每一个符号的节就是`CREATE_OBJECT_SYMBOLS'命令出现的那个节。

这个命令一直是 a.out 目标文件格式特有的。 它一般不为其它的目标文件格式所使用。

`CONSTRUCTORS'
当使用 a.out 目标文件格式进行连接的时候， 连接器使用一组不常用的结构以支持 C++的全局构造函数和析构函数。当连接不支持专有节的目标文件格式时， 比如 ECOFF 和 XCOFF，连接器会自动辨识 C++全局构造函数和析构函数的名字。对于这些目标文件格式，‘CONSTRUCTORS’命令告诉连接器把构造函数信息放到‘CONSTRUCTORS’命令出现的那个输出节中。对于其它目标文件格式，‘CONSTRUCTORS’命令被忽略。

符号`__CTOR_LIST__'标识全局构造函数的开始，而符号`__DTOR_LIST'标识结束。这个列表的第一个 WORD 是入口的数量，紧跟在后面的是每一个构造函数和析构函数的地址，再然后是一个零 WORD。编译器必须安排如何实际运行代码。对于这些目标文件格式，GNU C++通常从一个`__main'子程序中调用构造函数，而对`__main'的调用自动被插入到`main'的启动代码中。GNU C++通常使用'atexit'运行析构函数，或者直接从函数'exit'中运行。

对于像‘COFF’或‘ELF’这样支持专有节名的目标文件格式，GNU C++通常会把全局构造函数与析构函数的地址值放到'.ctors'和'.dtors'节中。把下面的代码序列放到你的连接脚本中去，这样会构建出 GNU C++运行时代码希望见到的表类型。

```
__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__ = .;
```

如果你正使用 GNU C++支持来进行优先初始化，那它提供一些可以控制全局构造函数运行顺序的功能，你必须在连接时给构造函数排好序以保证它们以正确的顺序被执行。当使用'CONSTRUCTORS'命令时，替代为`SORT(CONSTRUCTORS)'。当使用'.ctors'和'dtors'节时，使用`*(SORT(.ctors))'和`*(SORT(.dtors))' 而不是`*(.ctors)'和`*(.dtors)'。

通常，编译器和连接器会自动处理这些事情，并且你不必亲自关心这些事情。但是，当你正在使用 C++，并自己编写连接脚本时，你可能就要考虑这些事情了。

>REGION

输出节区域

.....

你可以通过使用`>REGION'把一个节赋给前面已经定义的一个内存区域。

这里有一个简单的例子:

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

PHDR

.....

你可以通过使用`:PHDR'把一个节赋给前面已定义的一个程序段。如果一个节被赋给一个或多个段，那后来分配的节都会被赋给这些段，除非它们显式使用了':PHDR'修饰符。你可以使用':NONE'来告诉连接器不要把节放到任何一个段中。

这儿有一个简单的例子：

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

FILLEXP

输出段填充

.....

你可以通过使用'=FILLEXP'为整个节设置填充样式。FILLEXP 是一个表达式。任何没有指定的输出段内的内存区域（比如，因为输入段的对齐要求而产生的裂缝）会被填入这个值。如果填充表达式是一个简单的十六进制值，比如，一个以'0x'开始的十六进制数字组成的字符串，并且尾部不是'k'或'M'，那一个任意的十六进制数字长序列可以被用来指定填充样式；前导零也变为样式的一部分。对于所有其他的情况，包含一个附加的括号或一元操作符'+'，那填充样式是表达式的最低四字节的值。在所有的情况下，数值是 big-endian.

你还可以通过在输出节命令中使用'FILL'命令来改变填充值。

这里是一个简单的例子:

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

输出节的丢弃

连接器不会创建那些不含有任何内容的输出节。这是为了引用那些可能出现或不出现在任何输入文件中的输入节时方便。比如：

```
.foo

{ *(.foo) }
```

如果至少在一个输入文件中有'.foo'节，它才会在输出文件中创建一个'.foo'节

如果你使用了其它的而不是一个输入节描述作为一个输出节命令，比如一个符号赋值，那这个输出节总是被创建，即使没有匹配的输入节也会被创建。

一个特殊的输出节名`/DISCARD/'可以被用来丢弃输入节。任何被分配到名为`/DISCARD/'的输出节中的输入节不包含在输出文件中。

覆盖描述

一个覆盖描述提供一个简单的描述办法，以描述那些要被作为一个单独内存映像的一部分载入内存，但是却要在同一内存地址运行的节。在运行时，一些覆盖管理机制会把要被覆盖的节按需要拷入或拷出运行时内存地址，并且多半是通过简单地处理内存位。 这个方法可能非常有用，比如在一个特定的内存区域比另一个快时。

覆盖是通过‘OVERLAY’命令进行描述。‘OVERLAY’命令在‘SECTIONS’命令中使用，就像输出段描述一样。
‘OVERLAY’命令的完整语法如下：

```
OVERLAY [START] : [NOCROSSREFS] [AT ( LDADDR )]
  {
    SECNAME1
      {
        OUTPUT-SECTION-COMMAND
        OUTPUT-SECTION-COMMAND
        ...
      } [:PHDR...] [=FILL]
    SECNAME2
      {
        OUTPUT-SECTION-COMMAND
        OUTPUT-SECTION-COMMAND
        ...
```

```
    } [:PHDR...] [=FILL]
  ...
} [>REGION] [:PHDR...] [=FILL]
```

除了‘OVERLAY’关键字，所有的都是可选的，每一个节必须有一个名字（上面的 SECNAME1 和 SECNAME2）。在
‘OVERLAY’结构中的节定义跟通常的‘SECTIONS’结构中的节定义是完全相同的，除了一点，就是在‘OVERLAY’
中没有地址跟内存区域的定义。

节都被定义为同一个开始地址。所有节的载入地址都被排布，使它们在内存中从整个 'OVERLAY'的载入地址开
始都是连续的（就像普通的节定义，载入地址是可选的，缺省的就是开始地址；开始地址也是可选的，缺省的
是当前的定位计数器的值。）

如果使用了关键字`NOCROSSREFS'， 并且在节之间存在引用，连接器就会报告一个错误。因为节都运行在同一
个地址上，所以一个节直接引用另一个节中的内容是错误的。

对于'OVERLAY'中的每一个节，连接器自动定义两个符号。符号`__load_start_SECNAME'被定义为节的开始载
入地址。符号`__load_stop_SECNAME'被定义为节的最后载入地址。SECNAME 中的不符合 C 规定的任何字符都将
被删除。C（或者汇编语言）代码可能使用这些符号在必要的时间搬移覆盖代码。

在覆盖区域的最后，定位计数器的值被设为覆盖区域的开始地址加上最大的节的长度。

这里是一个例子。记住这只会出现在‘SECTIONS’结构的内部。

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

这段代码会定义'.text0'和'.text1'，它们都从地址 0x1000 开始。 ‘.text0'会被载入到地址 0x4000 处，而
'.text1'会被载入到紧随'.text0'后的位置。下面的几个符号会被定义：
`__load_start_text0',
`__load_stop_text0', `__load_start_text1', `__load_stop_text1'.

拷贝'.text1'到覆盖区域的 C 代码看上去可能会像下面这样：

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

注意'OVERLAY'命令只是为了语法上的便利，因为它所做的所有事情都可以用更加基本的命令加以代替。上面
的例子可以用下面的完全特效的写法：

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