

Ld 链接器与脚本语法介绍

概述

'ld' 把一定量的目标文件跟档案文件链接起来, 并重定位它们的数据, 链接符号引用. 一般, 在编译一个程序时, 最后一步就是运行'ld'.

'ld' 能接受链接命令语言文件, 这是一种用 AT&T 的链接编辑命令语言的超集写成的文件, 用来在链接的整个过程中提供显式的, 全局的控制.

命令行选项

链接器提供大量的命令行选项, 但是, 在实际使用中, 只有少数被经常使用. 比如, 'ld' 的一个经常的使用场合是在一个标准的系统上链接标准的目标文件. 在这样的一个系统上, 链接文件'hello.o' 如下:

```
ld -o OUTPUT /lib/crt0.o hello.o -lc
```

这告诉'ld' 产生一个叫 OUTPUT 的文件, 作为链接文件'/lib/crt0.o' 和'hello.o' 和库'libc.a' 的结果.'libc.a' 来自标准的搜索路径. (如果库在其他路径, 使用'-L' 传递库的路径).

有些命令行选项可以在命令行的任何位置出现. 但是, 那些带有文件名的选项, 比如'-l' 或者'-T', 会让文件在选项出现的位置上被读取. 对于非文件选项, 以带不同的参数重复它, 不会有进一步的效果, 或者覆盖掉前面的相同项.

无参数选项是那些被链接的目标文件和库文件. 它们可能紧随命令行选项, 或在它们前面, 或者跟它们夹杂在一起, 但是一个目标文件参数是不会出现在一个选项跟它的参数之间.

通常, 链接器至少引用一个目标文件, 但是你可指定其它形式的二进制输入文件, 这可以通过'-l', '-R' 或者脚本命令语言来实现. 如果没有任何二进制文件被指定, 链接器不会产生任何输出, 并给出信息: "缺少输入文件."

如果链接器不能识别目标文件的格式, 它会假设这些只是链接脚本. 以这种方式指定的脚本增加了链接用的主链接脚本的内容(主链接脚本即缺省链接脚本或使用'-T' 指定的脚本). 这个特性可以允许链接器链接一些文件, 它们看上去既像目标文件, 又像档案文件, 但实际上只是定义了一些符号值, 或者使用'INPUT' 或'

GROUP' 来载入其它的目标文件. 需要注意的是, 用这种方式指定一个脚本只是增加了主链接脚本的内容; 要完全替换掉主链接脚本, 需要使用' -T'.

集成开发环境已经集成了链接器的命令行所需信息, 这里常见的 -o 指定输出文件名, -L 添加库路径, -l 添加库文件, --gc-sections 执行死段优化, -Map XX.map 输出信息内容, -T"filename" 指定脚本文件。

链接脚本介绍

.....

集成开发环境集成了默认芯片型号的链接脚本, 该脚本存在在安装路径下的工具链的脚本路径中, 如: "C:\Program Files (x86)\ChipON IDE\KungFu32\ChiponCC32\scripting\ccr1_issue\KF32F350MQV.ld", 如果需要编写调整项目需要的脚本, 可复制该文件到项目中, 并修改项目的属性更换目标链接脚本, 即在项目属性下, 选择 C/C++ 构建下面的设置, 并分别选择配置的 Release 和 Debug, 在工具设置下面的链接器设置的通用设定对话框页面在芯片脚本文件栏进行脚本替换, 默认为 -T"\${CHIP_NAME_SCRIPT}", 即为内部集成变量的控制传递。这里可以更改为 -T".. /MyKF32F350MQV.ld", 即使用项目根目录下的脚本文件。

任何脚本的控制均应循序芯片的资源空间进行编写, 可以进行拆分但不能超过空间限定, 否则仅仅为工具的链接通过, 在实际的烧录下芯片中运行会不正确。

每个链接都被一个' 链接脚本' 所控制. 这个脚本是用链接命令语言书写的. 链接脚本的一个主要目的是描述输入文件中的节如何被映射到输出文件中, 并控制输出文件的内存排布. 几乎所有的链接脚本只做这两件事情. 但是, 在需要的时候, 链接器脚本还可以指示链接器执行很多其他的操作. 这通过下面描述的命令实现.

基本的链接脚本的概念

=====

我们需要定义一些基本的概念与词汇以描述链接脚本语言.

链接器把多个输入文件合并成单个输出文件. 输出文件和输入文件都以一种叫做' 目标文件格式' 的数据格式形式存在. 每一个文件被叫做' 目标文件'. 输出文件经常被叫做' 可执行文件', 但是由于需要, 我们也把它叫做目标文件. 每一个目标文件中, 在其它东西之间, 有一个节列表. 我们有时把输入文件的节叫做输入节; 相似的, 输出文件中的一个节经常被叫做输出节.

一个目标文件中的每一个节都有一个名字和一个大小尺寸。大多数节还有一个相关的数据块，称为节内容。

链接脚本的格式

链接脚本是文本文件。

你写了一系列的命令作为一个链接脚本。每一个命令是一个带有参数的关键字，或者是一个对符号的赋值。你可以用分号分隔命令。空格一般被忽略。

文件名或格式名之类的字符串一般可以被直接键入。如果文件名含有特殊字符，比如一般作为分隔文件名用的逗号，你可以把文件名放到双引号中。文件名中间无法使用双引号。

你可以象在 C 语言中一样，在链接脚本中使用注释，用 `/*` 和 `*/` 隔开。就像在 C 中，注释在语法上等同于空格。

可能的最简单的脚本只含有一个命令：`SECTIONS`。你可以使用 `SECTIONS` 来描述输出文件的内存布局。

`SECTIONS` 是一个功能很强大的命令。这里我们会描述一个很简单的使用。让我们假设你的程序只有代码节，初始化过的数据节，和未初始化过的数据节。这些会存在于 `.text`，`.data` 和 `.bss` 节，另外，让我们进一步假设在你的输入文件中只有这些节。

对于这个例子，我们说代码应当被载入到地址 `0x8000` 处，而数据应当从 `0x1000000` 处开始。下面是一个实现这个功能的脚本：

```
SECTIONS
{
    . = 0x8000;
    .text : { *(.text) }
    . = 0x10000000;
    .data : { *(.data) }
    .bss : { *(.bss) }
}
```

使用关键字 `SECTIONS` 写了这个 `SECTIONS` 命令，后面跟有一串放在花括号中的符号赋值和输出节描述的内容。

上例中，在 `SECTIONS` 命令中的第一行是对一个特殊的符号 `.` 赋值，这是一个定位计数器。如果你没有以其它的方式指定输出节的地址（其他方式在后面会描述），那地址值就会被设为定位计数器的现有值。定位计数器然后被加上输出节的尺寸。在 `SECTIONS` 命令的开始处，定位计数器拥有值 `0`。

第二行定义一个输出节, '.text'. 冒号是语法需要, 现在可以被忽略. 节名后面的花括号中, 你列出所有应当被放入到这个输出节中的输入节的名字. '*' 是一个通配符, 匹配任何文件名. 表达式 '*'(.text)' 意思是所有的输入文件中的 '.text' 输入节.

因为当输出节 '.text' 定义的时候, 定位计数器的值是 '0x8000', 链接器会把输出文件中的 '.text' 节的地址设为 '0x8000'.

余下的内容定义了输出文件中的 '.data' 节和 '.bss' 节. 链接器会把 '.data' 输出节放到地址 '0x10000000' 处. 链接器放好 '.data' 输出节之后, 定位计数器的值是 '0x10000000' 加上 '.data' 输出节的长度. 得到的结果是链接器会把 '.bss' 输出节放到紧接 '.data' 节后面的位置.

链接器会通过必要时增加定位计数器的值来保证每一个输出节具有它所需的对齐. 在这个例子中, 为 '.text' 和 '.data' 节指定的地址会满足对齐约束, 但是链接器可能会需要在 '.data' 和 '.bss' 节之间创建一个小的缺口.

链接脚本其他命令.

设置入口点.

在运行一个程序时第一个被执行到的指令称为“入口点”. 你可以使用 'ENTRY' 链接脚本命令来设置入口点. 参数是一个符号名: ENTRY (SYMBOL)

有多种不同的方法来设置入口点. 链接器会通过按顺序尝试以下的方法来设置入口点, 如果成功了, 就会停止.

- * '-e' 入口命令行选项;
- * 链接脚本中的 'ENTRY (SYMBOL)' 命令;
- * 如果定义了 start, 就使用 start 的值;
- * 如果存在, 就使用 '.text' 节的首地址;
- * 地址 '0'.

当前环境下的 .text :

```
{
    . = 0x0000;
    KEEP (*vector.o(.text)) /* 中断向量表 */实现向量表的存放在 flash 空间的
    起始, 并使用 KEEP 关键字定义内容不参与死段优化, 即实现了程序的入口。
```

处理文件的命令.

``INCLUDE FILENAME'` 在当前点包含链接脚本文件 FILENAME. 在当前路径下或用 `'-L'` 选项指定的所有路径下搜索这个文件, 你可以嵌套使用 `'INCLUDE'` 达 10 层.

``INPUT(FILE, FILE, ...)`` ``INPUT(FILE FILE ...)`` `INPUT` 命令指示链接器在链接时包含文件, 就像它们是在命令行上指定的一样.

如, 如果你在链接的时候总是要包含文件 `'subr.o'`, 但是你对每次链接时要在命令行上输入感到厌烦, 你就可以在你的链接脚本中输入 `'INPUT (subr.o).'`

事实上, 如果你喜欢, 你可以把你所有的输入文件列在链接脚本中, 然后在链接的时候什么也不需要, 只要一个 `'-T'` 选项就够了.

在一个 `'系统根前缀'` 被配置的情况下, 一个文件名如果以 `'/'` 字符打头, 并且脚本也存放在系统根前缀的某个子目录下, 文件名就会被在系统根前缀下搜索. 否则链接器就会企图打开当前目录下的文件. 如果没有发现, 链接器会通过档案库搜索路径进行搜索.

如果你使用了 `'INPUT (-lFILE)'`, `'ld'` 会把文件名转换为 `'libFILE.a'`, 就象命令行参数 `'-l'` 一样.

``GROUP(FILE, FILE, ...)`` ``GROUP(FILE FILE ...)``

除了文件必须全是档案文件之外, `'GROUP'` 命令跟 `'INPUT'` 相似, 它们会被反复搜索, 直至没有未定义的引用被创建.

``OUTPUT(FILENAME)'` `'OUTPUT'` 命令命名输出文件. 在链接脚本中使用 `'OUTPUT(FILENAME)'` 命令跟在命令行中使用 `'-o FILENAME'` 命令是完全等效的. 如果两个都使用了, 那命令行选项优先.

你可以使用 `'OUTPUT'` 命令为输出文件创建一个缺省的文件名, 而不是常用的 `'a.out'`.

``SEARCH_DIR(PATH)'` `'SEARCH_DIR'` 命令给 `'ld'` 用于搜索档案文件的路径中再增加新的路径. 使用 ``SEARCH_DIR(PATH)'` 跟在命令行上使用 `'-L PATH'` 选项是完全等效的. 如果两个都使用了, 那链接器会两个路径都搜索. 用命令行选项指定的路径首先被搜索.

``STARTUP(FILENAME)'` 除了 FILENAME 会成为第一个被链接的输入文件, `'STARTUP'` 命令跟 `'INPUT'` 命令完全相似, 就象这个文件是在命令行上第一个被指定的文件一样. 如果在一个系统中, 入口点总是存在于第一个文件中, 那这个就很有用.

SECTIONS 命令

'SECTIONS' 命令告诉链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。这也可以让链接脚本更容易理解，因为你可以更有意义的地方使用这些命令来控制输出文件的布局。

如果你在链接脚本中不使用 'SECTIONS' 命令，链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.

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.

输出节名.

输出节的名称是 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.

输出节描述

ADDRESS 是关于输出节中 VMS 的一个表达式. 如果你不提供 ADDRESS, 链接器会基于 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到 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链接脚本操作。你使用输出节来告诉链接器在内存中如何布局你的程序。你使用输入节来告诉链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，并以链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，链接器首先会查看你是否在链接命令行上指定了文件名或者在`INPUT`命令中。如果你没有，链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，因为链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范围，比如`[a-z]`匹配任意小写字母。

`\' 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. 链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, 链接器会使用第一次匹配到的链接脚本. 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, 链接器会把匹配通配符的文件和节按在链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, 链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` 链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. 这个链接脚本指示链接器把所有的`.text` 节放到`.text` 中, 把所有的`.bss` 节放到`.bss`. 链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, 链接器会把`.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. 链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。在这种情况下,链接器会为其类型的普通符号使用一个不同的特殊节名。链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。

输入节和垃圾收集

当链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链接脚本。它告诉链接器去读取文件'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’，或者‘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 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，下面的代码会使链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’这个命令告诉链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 目标文件格式进行链接的时候，链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。当链接不支持专有节的目标文件格式时，比如 ECOFF 和 XCOFF，链接器会自动辨识 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’命令告诉链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‘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链接脚本中去，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，你必须在链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，编译器和链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++，并自己编写链接脚本时，你可能就要考虑这些事情了。

输出节的丢弃。

链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。

输出节属性

上面，我们已经展示了一个完整的输出节描述，看下去就象这样：

```
SECTION [ADDRESS] [(TYPE)] : [AT(LMA)]
{
```

```

        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. 在这一节中, 我们将介绍余下的节属性。

输出节类型

.....

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’ 这个节应当被标式诃不可载入, 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: 这个节应当被标记不可分配, 所以当程序运行时, 没有内存为这个节分配。

链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链接输出文件中。

```

    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

链接器通常把 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，下面的链接脚本创建了一个输出节：一个叫做 ‘.text’ 从地址 ‘0x1000’ 处开始，一个叫做 ‘.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 = . ; }
}
```

这个链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链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;
```

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 }
```


输出节 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` 来告诉链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 }
```

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 }
```

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，链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，链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));
```

MEMORY 命令

链接器在缺省状态下被配置为允许分配所有可用的内存块。你可以使用'MEMORY'命令重新配置这个设置。

'MEMORY'命令描述目标平台上内存块的位置与长度。你可以用它来描述哪些内存区域可以被链接器使用，哪些内存区域是要避免使用的。然后你就可以把节分配到特定的内存区域中。链接器会基于内存区域设置节的地址，对于太满的区域，会提示警告信息。链接器不会为了适应可用的区域而搅乱节。

一个链接脚本最多可以包含一次'MEMORY'命令。但是，你可以在命令中随心所欲定义任意多的内存块，语法如下：

```
MEMORY
{
    NAME [(ATTR)] : ORIGIN = ORIGIN, LENGTH = LEN
    ...
}
```

NAME 是用于在链接脚本中引用内存区域的名字。出了链接脚本，区域名就没有任何实际意义。区域名存储在一个单独的名字空间中，它不会和符号名，文件名，节名产生冲突，每一块内存区域必须有一个唯一的名字。

ATTR 字符串是一个可选的属性列表，它指出是否为一个没有在链接脚本中进行显式映射地输入段使用一个特定的内存区域。如果你没有为某些输入段指定一个输出段，链接器会创建一个跟输入段同名的输出段。如果你定义了区域属性，链接器会使用它们来为它创建的输出段选择内存区域。

ATTR 字符串必须包含下面字符中的一个，且必须只包含一个：

- `R' 只读节。
- `W' 可读写节。
- `X' 可执行节。
- `A' 可分配节。
- `I' 已初始化节。
- `L' 同 'I'
- `!' 对前一个属性值取反。

如果一个未映射节匹配了上面除'!'之外的一个属性，它就会被放入该内存区域。'!' 属性对该测试取反，所以只有当它不匹配上面列出的任何属性时，一个未映射节才会被放入到内存区域。

ORIGIN 是一个关于内存区域起始地址的表达式。在内存分配执行之前，这个表达式必须被求值产生一个常数，这意味着你不可以使用任何节相关的符号。关键字 'ORIGIN' 可以被缩写为 'org' 或 'o' (但是，不可以写为，比如 'ORG')。

LEN 是一个关于内存区域长度（以字节为单位）的表达式。就像 ORIGIN 表达式，这个表达式在分配执行前也必须被求得为一个常数值。关键字 'LENGTH' 可以被简写为 'len' 或 'l'。

在下面的例子中，我们指定两个可用于分配的内存区域：一个从 0 开始，有 256 kb 长度，另一个从 0x4000000 开始，有 4mb 长度。链接器会把那些没有进行显式映射且是只读或可执行的节放到 'rom' 内存区域。并会把另外的没有被显式映射地节放入到 'ram' 内存区域。

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

一旦你定义了一个内存区域，你也可以指示链接器把指定的输出段放入到这个内存区域中，这可以通过使用 '>REGION' 输出段属性。比如，如果你有一个名为 'mem' 的内存区域，你可以在输出段定义中使用 '>mem'。如果没有为输出段指定地

址，链接器就会把地址设置为内存区域中的下一个可用的地址。如果总共的映射到一个内存区域的输出段对于区域来说太大了，链接器会提示一条错误信息。

链接脚本中的表达式

链接脚本语言中的表达式的语法跟 C 的表达式是完全是致的。所有的表达式都以整型值被求值。所有的表达式也被以相同的宽度求值。在 32 位系统是它是 32 位，否则是 64 位。你可以在表达式中使用和设置符号值。链接器为了使用表达式，定义了几个具有特殊途的内建函数。

常数

所有的常数都是整型值。

就像在 C 中，链接器把以 '0' 开头的整型数视为八进制数，把以 '0x' 或 '0X' 开头的视为十六进制。链接器把其它的整型数视为十进制。

另外，你可以使用 'K' 和 'M' 后缀作为常数的度量单位，分别为 '1024' 和 '1024*1024'。比如，下面的三个常数表示同一个值。

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

符号名

除了引用，符号名都是以一个字母，下划线或者句号开始，可以包含字母，数字，下划线，句点和链接号。

不是被引用的符号名必须不和任何关键字冲突。你可以指定一个含有不固定它符数或具有跟关键字相同名字但符号名必须在双引号内：

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

因为符号可以含有很多非文字字符，所以以空格分隔符号是很安全的。比如，'A-B' 是一个符号，而 'A - B' 是一个执行减法运算的表达式。

定位计数器

一个特殊的链接器变量 "dot" '.' 总是含有当前的输出定位计数器。因为 '.' 总引用输出段中的一个位置，它只可以出现在 'SECTIONS' 命令中的表达式中。 '.' 符号可以出现在表达式中一个普能符号允许出现的任何位置。

. = (. + 3) & (-4); 常常控制实现当前起始计算地址位四字节对齐。把一个值赋给 '.' 会让定位计数器产生移动。这会在输出段中产生空洞。定位计数器从不向前移动。

```

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

```

在前面的例子中，来自'file1'的'.text'节被定位在输出节'output'的起始位置。它后面跟有 1000byte 的空隙。然后是来自'file2'的'.text'节，同样是后面跟有 1000byte 的空隙，最后是来自'file3'的'.text'节。符号'=0x12345678'指定在空隙中填入什么样的数据。注意：'.'实际上引用的是当前包含目标的从开始处的字节偏移。通常，它就是'SECTIONS'语句，其起始地址是 0，因为'.'可以被用作绝对地址。但是如果'.'被用在一个节描述中，它引用的是从这个节起始处开始的偏移，而不是一个绝对地址。这样，在下面这样一个脚本中：

```

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

```

' .text' 节被赋予起始地址 0x100，尽管在'.text'输入节中没有足够的数据来填充这个区域，但其长度还是 0x200bytes。（如果数据太多，那会产生一条错误信息，因为这会试图把'.'向前移）。'.data'节会从 0x500 处开始，并且它在结尾处还会有 0x600 的额外空间。

运算符

链接器可以识别标准的 C 的算术运算符集，以及它们的优先集。

优先集 (highest)	结合性	运算符	备注
1	left	! - ~	(1)

2	left	* / %	
3	left	+ -	
4	left	>> <<	
5	left	== != > < <= >=	
6	left	&	
7	left		
8	left	&&	
9	left		
10	right	? :	
11	right	&= += -= *= /=	(2)
(lowest)			

注：(1) 前缀运算符 (2) *Note Assignments::.

求值

链接器是懒惰求表达式的值。它只在确实需要的时候去求一个表达式的值。

链接器需要一些信息，比如第一个节的起始地址的值，还有内存区域的起点与长度，在做任何链接的时候这都需要。在链接器读取链接脚本的时候，这些值在可能的时候被计算出来。但是，其它的值（比如符号的值）直到内存被分配之后才会知道或需要。这样的值直到其它信息（比如输出节的长度）可以被用来进行符号赋值的时候才被计算出来。

直到内存分配之后，节的长度才会被知道，所以依赖于节长度的赋值只能到内存分配之后才会被执行。

有些表达式，比如那些依赖于定位计数器‘.’的表达式，必须在节分配的过程中被计算出来。如果一个表达式的结果现在被需要，但是目前得不到这个值，这样会导致一个错误。比如，象下面这样一个脚本：

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

会产生一个错误信息‘non constant expression for initial address’.

内建函数

为了使用链接脚本表达式，链接脚本语言含有一些内建函数。

‘ABSOLUTE(EXP)’ 返回表达式 EXP 的绝对值（不可重定位，而不是非负）。主要在把一个绝对值赋给一个节定义内的符号时有用。

‘ADDR(SECTION)’ 返回节 SECTION 的绝对地址（VMA）。你的脚本之前必须已经定义了这个节的地址。在接下来的例子中，‘symbol_1’ 和 ‘symbol_2’ 被赋以相同的值。

```
SECTIONS { ...
    .output1 :
```

```

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

```

‘ALIGN(EXP)’返回定位计数器‘.’对齐到下一个 EXP 指定的边界后的值。‘ALIGN’不改变定位计数器的值，它只是在定位计数器上面作了一个算术运算。这里有一个例子，它在前面的节之后，把输出节‘.data’对齐到下一个‘0x2000’字节的边界，并在输入节之后把节内的一个变量对齐到下一个‘0x8000’字节的边界。

```

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

```

这个例子中前一个‘ALIGN’指定一个节的位置，因为它是作为节定义的可选项 ADDRESS 属性出现的。第二个‘ALIGN’被用来定义一个符号的值。

内建函数‘NEXT’跟‘ALIGN’非常相似。

‘BLOCK(EXP)’这是‘ALIGN’的同义词，是为了与其它的链接器保持兼容。这在设置输出节的地址时非常有用。

‘DATA_SEGMENT_ALIGN(MAXPAGESIZE, COMMONPAGESIZE)’

这跟下面的两个表达同义：

```
(ALIGN(MAXPAGESIZE) + (. & (MAXPAGESIZE - 1)))
```

或者：

```
(ALIGN(MAXPAGESIZE) + (. & (MAXPAGESIZE - COMMONPAGESIZE)))
```

隐式链接脚本

如果你指定了一个链接器输出文件，而链接器不能识别它是一个目标文件还是档案文件，它会试图把它读作一个链接脚本。如果这个文件不能作为一个链接脚本被分析，链接器就会报告一个错误。一个隐式的链接器脚本不会替代缺省的链接器脚本。

一般，一个隐式的链接器脚本只包含符号赋值，或者‘INPUT’，‘GROUP’或‘VERSION’命令。