1. NASM命令行语法(从百度摘抄而来)

要汇编一个文件，你可以以下面的格式执行一个命令：

nasm -f <format> <filename> [-o <output>]

比如,

nasm -f elf myfile.asm

会把文件'myfile.asm'汇编成'ELF'格式 的文件'myfile.o'.还有：

nasm -f bin myfile.asm -o myfile.com

会把文件'myfile.asm'汇编成纯二进制格式的文件'myfile.com'。

想要以十六进制代码的形式产生列表文件输出，并让代码显示在源代码的左侧，

使用'-l'选项并给出列表文件名，比如：

nasm -f coff myfile.asm -l myfile.lst

想要获取更多的关于NASM的使用信息，请输入：

nasm -h

它同时还会输出可以使用的输出文件格式，

如果你使用Linux并且不清楚你的系统是'a.out'还是'ELF'，请输入：

file nasm

(在nasm二进制文件的安装目录下使用),如果系统输出类似下面的信息：

nasm: ELF 32-bit LSB executable i386 (386 and up) Version 1

那么你的系统就是'ELF'格式的，然后你就应该在产生Linux目标文件时使用选

项'-f elf',如果系统输入类似下面的信息：

nasm: Linux/i386 demand-paged executable (QMAGIC)

或者与此相似的，你的系统是'a.out'的，那你应该使用'-f aout'(Linux的'a.out'

系统很久以前就过时了，现在已非常少见。)

就像其他的Unix编译器与汇编器，NASM在碰到错误以前是不输出任何信息的，所

以除了出错信息你看不到任何其他信息。

1.1 '-o'选项：指定输出文件的文件名。

NASM会为你的输出文件选择一个文件名；具体如何做取决于目标文件的格式，对

于微软的目标文件格式('obj'和'win32'),它会去掉你的源文件名的'.asm'扩展

名(或者其他任何你喜欢使用的扩展名，NASM并不关心具体是什么)，并替换上

'obj'。对于Unix的目标文件格式('aout','coff','elf'和'as86')它会替换成

'.o', 对于'rdf',它会使用'.rdf',还有为'bin'格式，它会简单地去掉扩展名，所以

'myfile.asm'会产生的一个输出文件'myfile'。

如果输出文件已经存在，NASM会覆盖它，除非它的文件名与输入文件同名，在这种

情况下，它会给出一个警告信息，并使用'nasm.out'作为输出文件的文件名。

在某些情况下，上述行为是不能接受的，所以，NASM提供了'-o'选项，它能让你指定

你的输出文件的文件名，你使用'-o'后面紧跟你为输出文件取的名字，中间可以加

空格也可以不加。比如：

nasm -f bin program.asm -o program.com

nasm -f bin driver.asm -odriver.sys

请注意这是一个小写的o，跟大写字母O是不同的，大写的是用来指定需要传递的选

项的数目，请参阅1.15

1.2 `-f'选项：指定输出文件的格式。

如果你没有对NASM使用'-f'选项，它会自己为你选择一个输出文件格式。在发布的

NASM版本中，缺省的输出格式总是'bin';如果你自己编译你的NASM,你可以在编译的

时候重定义'OF\_DEFAULT'来选择你需要的缺省格式。

就象'-o','-f'与输出文件格式之间的空格也是可选的，所以'-f elf'和'-felf'都是

合法的。

所有可使用的输出文件格式的列表可以通过运行命令'nasm -hf'得到。

1.3 `-l' 选项: 产生列表文件

如果你对NASM使用了'-l'选项，后面跟一个文件名，NASM会为你产生一个源文件的列表

文件,在里面，地址和产生的代码列在左边，实际的源代码(包括宏扩展，除了那些指定

不需要在列表中扩展的宏，参阅4.3.9)列在右边，比如：

nasm -f elf myfile.asm -l myfile.lst

1.4 `-M'选项: 产生Makefile依赖关系.

该选项可以用来向标准输出产生makefile依赖关系，可以把这些信息重定向到一个文件

中以待进一步处理，比如：

NASM -M myfile.asm > myfile.dep

1.5 `-F'选项: 选择一个调试格式

该选项可以用来为输出文件选择一个调试格式，语法跟-f选项相册，唯一不同的是它产

生的输出文件是调试格式的。

一个具体文件格式的完整的可使用调试文件格式的列表可通过命令'nasm -f <format> -y'

来得到。

这个选项在缺省状态下没有被构建时NASM。如何使用该选项的信息请参阅6.10

1.6 `-g' 选项:使调试信息有效。

该选项可用来在指定格式的输出文件中产生调试信息。

更多的信息请参阅1.5

1.7 `-E' 选项: 把错误信息输入到文件。

在'MS-DOS'下，尽管有办法，但要把程序的标准错误输出重定向到一个文件还是非常困

难的。因为NASM常把它的警告和错误信息输出到标准错误设备，这将导致你在文本编

辑器里面很难捕捉到它们。

因此NASM提供了一个'-E'选项，带有一个文件名参数，它可以把错误信息输出到指定的

文件而不是标准错误设备。所以你可以输入下面这样的命令来把错误重定向到文件：

nasm -E myfile.err -f obj myfile.asm

1.8 `-s' 选项: 把错误信息输出到'stdout'

'-s'选项可以把错误信息重定向到'stdout'而不是'stderr'，它可以在'MS-DOS'下进行

重定向。想要在汇编文件'myfile.asm'时把它的输出用管道输出给'more'程序，可以这样：

nasm -s -f obj myfile.asm | more

请参考1.7的'-E'选项.

1.9 `-i'选项: 包含文件搜索路径

当NASM在源文件中看到'%include'操作符时(参阅4.6)，它不仅仅会在当前目录下搜索给

出的文件，还会搜索'-i'选项在命令行中指定的所有路径。所以你可以从宏定义库中

包含进一个文件，比如，输入：

nasm -ic:\macrolib\ -f obj myfile.asm

(通常，在 '-i'与路径名之间的空格是允许的，并且可选的。)

NASM更多的关注源代码级上的完全可移植性，所以并不理解正运行的操作系统对文件的

命名习惯；你提供给'-i'作为参数的的字符串会被一字不差地加在包含文件的文件名前。

所以，上例中最后面的一个反斜杠是必要的，在Unix下，一个尾部的正斜线也同样是必要的。

(当然，如果你确实需要，你也可以不正规地使用它，比如，选项'-ifoo'会导致

'%incldue "bar.i'去搜索文件'foobar.i'...)

如果你希望定义一个标准的搜索路径，比如像Unix系统下的'/usr/include'，你可以在环境

变量NASMENV中放置一个或多个'-i'(参阅1.19)

为了与绝大多数C编译器的Makefile保持兼容，该选项也可以被写成'-I'。

1.10 `-p' 选项: 预包含一个文件

NASM允许你通过'-p'选项来指定一个文件预包含进你的源文件。所以，如果运行：

nasm myfile.asm -p myinc.inc

跟在源文件开头写上'%include "myinc.inc"然后运行'nasm myfile.asm'是等效的。

为和'-I','-D','-U'选项操持一致性，该选项也可以被写成'-P'

1.11 `-d'选项: 预定义一个宏。

就像'-p'选项给出了在文件头放置'%include'的另一种实现，'-d'选项给出了在文

件中写'%define'的另一种实现，你可以写：

nasm myfile.asm -dFOO=100

作为在文件中写下面一行语句的一种替代实现：

%define FOO 100

在文件的开始，你可以取消一个宏定义，同样，选项'-dFOO'等同于代码'%define FOO'。

这种形式的操作符在选择编译时操作中非常有用，它们可以用'%ifdef'来进行测试，

比如'-dDEBUG'。

为了与绝大多数C编译器的Makefile保持兼容，该选项也可以被写成'-D'。

1.12 `-u' 选项: 取消一个宏定义。

'-u'选项可以用来取消一个由'-p'或'-d'选项先前在命令行上定义的一个宏定义。

比如，下面的命令语句：

nasm myfile.asm -dFOO=100 -uFOO

会导致'FOO'不是一个在程序中预定义的宏。这在Makefile中不同位置重载一个操

作时很有用。

为了与绝大多数C编译器的Makefile保持兼容，该选项也可以被写成'-U'。

1.13 `-e'选项: 仅预处理。

NASM允许预处理器独立运行。使用'-e'选项(不需要参数)会导致NASM预处理输入

文件，展开所有的宏，去掉所有的注释和预处理操作符，然后把结果文件打印在标

准输出上(如果'-o'选项也被指定的话，会被存入一个文件)。

该选项不能被用在那些需要预处理器去计算与符号相关的表达式的程序中，所以

如下面的代码：

%assign tablesize ($-tablestart)

会在仅预处理模式中会出错。

1.14 `-a' 选项: 不需要预处理。

如果NASM被用作编译器的后台，那么假设编译器已经作完了预处理，并禁止NASM的预

处理功能显然是可以节约时间，加快编译速度。'-a'选项(不需要参数)，会让NASM把

它强大的预处理器换成另一个什么也不做的预处理器。

1.15 `-On'选项: 指定多遍优化。

NASM在缺省状态下是一个两遍的汇编器。这意味着如果你有一个复杂的源文件需要

多于两遍的汇编。你必须告诉它。

使用'-O'选项，你可以告诉NASM执行多遍汇编。语法如下：

(\*)'-O0'严格执行两遍优化，JMP和Jcc的处理和0.98版类似，除了向后跳的JMP是短跳

转，如果可能，立即数在它们的短格式没有被指定的情况下使用长格式。

(\*)'-O1'严格执行两遍优化，但前向分支被汇编成保证能够到达的代码；可能产生比

'-O0'更大的代码，但在分支中的偏移地址没有指定的情况下汇编成功的机率更大，

(\*)'-On' 多编优化，最小化分支的偏移，最小化带符号的立即数，当'strict'关键字

没有用的时候重载指定的大小(参阅3.7)，如果2<=n<=3，会有5\*n遍,而不是n遍。

注意这是一个大写的O，和小写的o是不同的，小写的o是指定输出文件的格式，可参阅

1.1

1.16 `-t'选项: 使用TASM兼容模式。

NASM有一个与Borlands的TASM之间的受限的兼容格式。如果使用了NASM的'-t'选项，

就会产生下列变化：

(\*)本地符号的前缀由'.'改为'@@'

(\*)TASM风格的以'@'开头的应答文件可以由命令行指定。这和NASM支持的'-@resp'

风格是不同的。

(\*)扩号中的尺寸替换被支持。在TASM兼容模式中，方括号中的尺寸替换改变了操作

数的尺寸大小，方括号不再支持NASM语法的操作数地址。比如，'mov eax,[DWORD VAL]'

在TASM兼容语法中是合法的。但注意你失去了为指令替换缺省地址类型的能力。

(\*)'%arg'预处理操作符被支持，它同TASM的ARG操作符相似。

(\*) `%local'预处理操作符。

(\*) `%stacksize'预处理操作符。

(\*) 某些操作符的无前缀形式被支持。 (`arg', `elif',`else', `endif', `if',

`ifdef', `ifdifi', `ifndef', `include',`local')

(\*) 还有很多...

需要更多的关于操作符的信息，请参阅4.9的TASM兼容预处理操作符指令。

1.17 `-w'选项: 使汇编警告信息有效或无效。

NASM可以在汇编过程中监视很多的情况，其中很多是值得反馈给用户的，但这些情况

还不足以构成严重错误以使NASM停止产生输出文件。这些情况被以类似错误的形式

报告给用户，但在报告信息的前面加上'warning'字样。警告信息不会阻止NASM产生

输出文件并向操作系统返回成功信息。

有些情况甚至还要宽松：他们仅仅是一些值得提供给用户的信息。所以，NASM支持'-w'

命令行选项。它以使特定类型的汇编警告信息输出有效或无效。这样的警告类型是

以命名来描述的，比如，'orphan-labels'，你可以以下列的命令行选项让此类警告信息

得以输出：'-w+orphan-labels'，或者以'-w-orphan-labels'让此类信息不能输出。

可禁止的警告信息类型有下列一些：

(\*)`macro-params'包括以错误的参数个数调用多行的宏定义的警告。这类警告信息

缺省情况下是输出的，至于为什么你可能需要禁止它，请参阅4.3.1。

(\*)`orphan-labels'包含源文件行中没有指令却定义了一个没有结尾分号的label的

警告。缺省状况下，NASM不输出此类警告。如果你需要它，请参阅3.1的例子。

(\*) 'number-overflow'包含那些数值常数不符合32位格式警告信息（比如，你很容易打

了很多的F，错误产生了'0x7fffffffffff'）。这种警告信息缺省状况下是打开的。

1.18 `-v'选项: 打印版本信息。

输入'NASM -v'会显示你正使用的NASM的版本号，还有它被编译的时间。

如果你要提交bug报告，你可能需要版本号。

1.19 `NASMENV'环境变量。

如果你定义了一个叫'NASMENV'的环境变量，程序会被把它认作是命令行选项附加的一

部分，它会在真正的命令行之前被处理。你可以通过在'NASMENV'中使用'-i'选项来定

义包含文件的标准搜索路径。

环境变量的值是通过空格符分隔的，所以值'-s ic:\nasmlib'会被看作两个单独的操

作。也正因为如此，意味着值'-dNAME='my name'不会象你预期的那样被处理, 因为它

会在空格符处被分开，NASM的命令行处理会被两个没有意义的字符串'-dNAME="my'和

'name"'给弄混。

为了解决这个问题，NASM为此提供了一个特性，如果你在'NASMENV'环境变量的第一个

字符处写上一个非减号字符，NASM就会把这个字符当作是选项的分隔符。所以把环

境变量设成'!-s!-ic:\nasmlib'跟'-s -ic:\nasmlib'没什么两样，但是

'!-dNAME="my name"就会正常工作了。

这个环境变量以前叫做'NASM',从版本0.98.32以后开始叫这个名字。

1. C元调用约定：

在参数传递中，有两个很重要的问题必须得到明确说明：

1) 当参数个数多于一个时，按照什么顺序把参数压入堆栈；

2) 函数调用后，由谁来把堆栈恢复原装。

在高级语言中，通过函数调用约定来说明这两个问题。常见的调用约定有：

Stdcall cdecl fastcall thiscall nakedcall

**\_\_cdecl调用约定**

\_\_cdecl调用约定又称为 C 调用约定，是 C/C++ 语言缺省的调用约定。参数按照从右至左的方式入栈，函数本身不清理栈，此工作由调用者负责，返回值在EAX中。由于由调用者清理栈，所以允许可变参数函数存在，如int sprintf(char\* buffer,const char\* format,...);。

**cdecl调用约定**

cdecl调用约定又称为C调用约定，是C语言缺省的调用约定，它的定义语法是：

int function (int a ,int b) // 不加修饰就是C调用约定

int \_\_cdecl function(int a,int b) // 明确指出C调用约定

在写本文时，出乎我的意料，发现cdecl调用约定的参数压栈顺序是和stdcall是一样的，参数首先由有向左压入堆栈。所不同的是，函数本身不清理堆栈，调用者负责清理堆栈。由于这种变化，C调用约定允许函数的参数的个数是不固定的，这也是C语言的一大特色。对于前面的function函数，使用cdecl后的汇编码变成：

调用处

push 1

push 2

call function

add esp,8 注意：这里调用者在恢复堆栈  被调用函数

\_function处

push ebp 保存ebp寄存器，该寄存器将用来保存堆栈的栈顶指针，可以在函数退出时恢复

mov ebp,esp 保存堆栈指针

mov eax,[ebp + 8H] 堆栈中ebp指向位置之前依次保存有ebp,cs:eip,a,b,ebp +8指向a

add eax,[ebp + 0CH] 堆栈中ebp + 12处保存了b

mov esp,ebp 恢复esp

pop ebp

ret 注意，这里没有修改堆栈

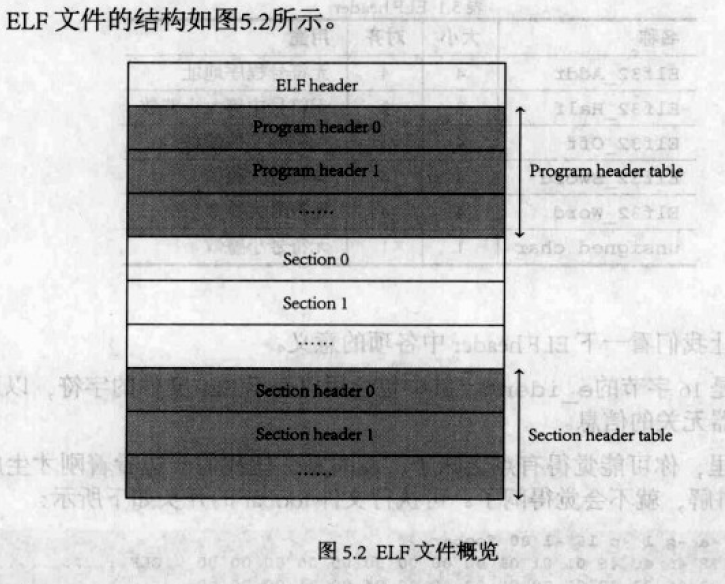
MSDN中说，该修饰自动在函数名前加前导的下划线，因此函数名在符号表中被记录为\_function，但是我在编译时似乎没有看到这种变化。

由于参数按照从右向左顺序压栈，因此最开始的参数在最接近栈顶的位置，因此当采用不定个数参数时，第一个参数在栈中的位置肯定能知道，只要不定的参数个数能够根据第一个后者后续的明确的参数确定下来，就可以使用不定参数，例如对于CRT中的sprintf函数，定义为：

int sprintf(char\* buffer,constchar\* format,...)

由于所有的不定参数都可以通过format确定，因此使用不定个数的参数是没有问题的。

1. ELF文件解析



ELF文件有4部分组成，分别是ELF头（ELF header），程序头表（Program header table），节（section）和节头表（section header）。

关于这些部分的具体结构，我觉得看具体的结构体定义，给枯燥的说这个东西要好的多。

所以我将GCC头文件elf.h拷贝了过来给大家作参考。

typedef struct

{

unsigned char e\_ident[EI\_NIDENT]; /\* Magic number and other info \*/

Elf32\_Half e\_type; /\* Object file type \*/

Elf32\_Half e\_machine; /\* Architecture \*/

Elf32\_Word e\_version; /\* Object file version \*/

Elf32\_Addr e\_entry; /\* Entry point virtual address \*/

Elf32\_Off e\_phoff; /\* Program header table file offset \*/

Elf32\_Off e\_shoff; /\* Section header table file offset \*/

Elf32\_Word e\_flags; /\* Processor-specific flags \*/

Elf32\_Half e\_ehsize; /\* ELF header size in bytes \*/

Elf32\_Half e\_phentsize; /\* Program header table entry size \*/

Elf32\_Half e\_phnum; /\* Program header table entry count \*/

Elf32\_Half e\_shentsize; /\* Section header table entry size \*/

Elf32\_Half e\_shnum; /\* Section header table entry count \*/

Elf32\_Half e\_shstrndx; /\* Section header string table index \*/

} Elf32\_Ehdr;

typedef struct

{

Elf32\_Word p\_type; /\* Segment type \*/

Elf32\_Off p\_offset; /\* Segment file offset \*/

Elf32\_Addr p\_vaddr; /\* Segment virtual address \*/

Elf32\_Addr p\_paddr; /\* Segment physical address \*/

Elf32\_Word p\_filesz; /\* Segment size in file \*/

Elf32\_Word p\_memsz; /\* Segment size in memory \*/

Elf32\_Word p\_flags; /\* Segment flags \*/

Elf32\_Word p\_align; /\* Segment alignment \*/

} Elf32\_Phdr;