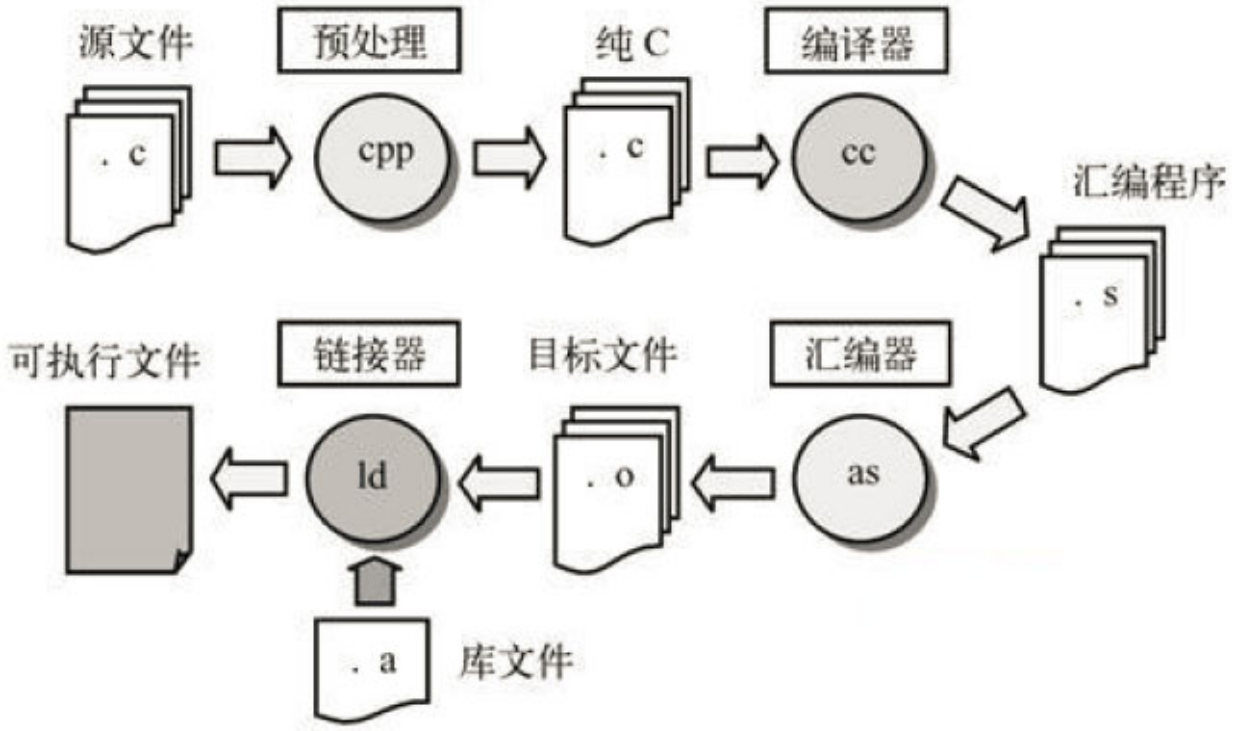
GCC+NASM联合编译

hanzhuo@smail.nju.edu.cn韩茁

基础知识-C语言的编译链接图



基础知识-编译链接

- 汇编/编译（生成语言无关中间代码）
  - C→中间代码，ASM→中间代码，DLX→中间代码
  - 会包含代码，函数定义
  - 可以是很多个独立文件
  - 参照上图，这个中间代码就是.o文件中的代码
- 链接（生成平台相关的可执行二进制代码）
  - 中间代码→linux（elf），mac（maco），windows（exe/dll）
  - 将多个中间文件的代码组合起来
  - 链接的多个中间文件的函数要声明齐全
- 动态链接和静态链接

  （检查作业考核点）

基础知识-C函数

- 声明和定义的区别
  - 声明只是告诉编译器有这么一个变量/函数
  - 定义是告诉编译器这个变量会占用多少内存，这个函数具体的代码是如何
- 先声明后调用
  - 标准ascii-c所有的函数都要在使用前先声明。
  - 提前分配地址和空间

如果不加最开始的两行函数声明的代码，无论两个函数哪个写在前面，编译都不能通过

```c
#include <stdio.h>
int do_something(int, char*);
int do_another(double, int);

int do_something(int n, char* name) {
    if(n==0) {
        return 0;
    } else {
        return do_another(1.0, n);
    }
}

int do_another(double score, int k) {
    return do_something(k - 1, "nju");
}

int main() {
    printf("%d", do_another(1.0, 10));
    return 0;
}
```

基础知识-C函数

- 如果函数的定义在其它文件，比如说你想要复用之前用汇编写的函数，那么要通过extern指定。
  - （gcc如果发现函数只有声明没有定义，默认是extern，不需要专门指定了）

```c
#include <stdio.h>
int do_something(int, char*);
extern int do_another(double, int);

int do_something(int n, char* name) {
    if(n==0) {
        return 0;
    } else {
        return do_another(1.0, n);
    }
}

int main() {
    printf("%d", do_another(1.0, 10));
    return 0;
}
```

这个代码，如果gcc main.c则会报错，而gcc -c main.c则不会报错。前者会编译同时链接，链接时找不到do_another的定义，而后者只编译，生成中间文件。链接时只要同时给出do_another的定义的代码就可以了，这个代码可以是c写的，也可以是任何语言写的

不链接，强行gcc就会这样

```
/tmp/cc8jMWe4.o: In function `main':
main.c:(.text+0x163): warning: the `gets' function is dangerous and should not be used.
main.c:(.text+0x109): undefined reference to `my_println'
main.c:(.text+0x118): undefined reference to `my_println'
main.c:(.text+0x140): undefined reference to `my_println'
main.c:(.text+0x14f): undefined reference to `my_print'
/tmp/cc8jMWe4.o: In function `splitter':
main.c:(.text+0x363): undefined reference to `my_colorPrint'
main.c:(.text+0x37f): undefined reference to `my_println'
main.c:(.text+0x39d): undefined reference to `my_colorPrint'
main.c:(.text+0x3ac): undefined reference to `my_print'
/tmp/cc8jMWe4.o: In function `fillBPB':
main.c:(.text+0x401): undefined reference to `my_println'
main.c:(.text+0x436): undefined reference to `my_println'
/tmp/cc8jMWe4.o: In function `printFiles':
main.c:(.text+0x4bf): undefined reference to `my_println'
main.c:(.text+0x4f4): undefined reference to `my_println'
/tmp/cc8jMWe4.o: In function `findChildren':
main.c:(.text+0xa1c): undefined reference to `my_println'
/tmp/cc8jMWe4.o:main.c:(.text+0xb65): more undefined references to `my_println' follow
/tmp/cc8jMWe4.o: In function `printContent':
main.c:(.text+0x1323): undefined reference to `my_print'
/tmp/cc8jMWe4.o: In function `printChildren':
main.c:(.text+0x1489): undefined reference to `my_println'
main.c:(.text+0x151a): undefined reference to `my_println'
main.c:(.text+0x1570): undefined reference to `my_println'
main.c:(.text+0x16f6): undefined reference to `my_println'
main.c:(.text+0x1737): undefined reference to `my_println'
```

示例

- 主程序为c语言，子程序为一个函数，返回3个参数的最大值。
- 主程序为main.c文件
  - 声明maxofthree函数，但不给出定义。（gcc会默认为extern）
- 子程序为func.asm文件
  - 通过global关键字，表明这个标签是对外的函数标签。

func.asm

maxofthree可以被其它模块使用，但label_2不可以。通过global关键字，nasm在汇编时会把maxofthree这个函数的信息写到中间文件里，链接器链接时就可以看到。

```
1  global  maxofthree
2
3      section .text
4  maxofthree:
5          mov     eax, [esp+4]
6          mov     ecx, [esp+8]
7          mov     edx, [esp+12]
8  label_2:
9          cmp     eax, ecx
10         cmovl   eax, ecx
11         cmp     eax, edx
12         cmovl   eax, edx
13         ret
```

cmovl：根据该指令上一句cmp指令比较的结果判定，例如cmp eax,ebx，此时，若eax<ebx，则eax=ebx；否则什么也不做。

main.c

- maxofthree只有声明没有定义，gcc只能编译不能链接。

```c
#include <stdio.h>

int maxofthree(int, int, int);

int main() {
    printf("%d\n", maxofthree(1, -4, -7));
    printf("%d\n", maxofthree(2, -6, 1));
    printf("%d\n", maxofthree(2, 3, 1));
    printf("%d\n", maxofthree(-2, 4, 3));
    printf("%d\n", maxofthree(2, -6, 5));
    printf("%d\n", maxofthree(2, 4, 6));
    return 0;
}

```

gcc+nasm

- nasm -f elf -o func.o func.asm
- gcc -c -o main.o main.c
- ld -o hello main.o func.o
- ./hello 运行

- 如果报printf函数找不到，这是因为gcc –c指令生成的obj文件是只与源代码对应的，printf函数在源代码中是通过#include<stdio.h>引入的，相当于只有声明没有定义，而-c选项指定了不进行链接过程，所以找不到printf函数
- 需要ld添加参数来引用标准c库。不过简单的作法是把ld的过程交给gcc命令，如下
  - nasm -f elf -o func.o func.asm
  - gcc -o hello main.c func.o
  - ./hello 运行

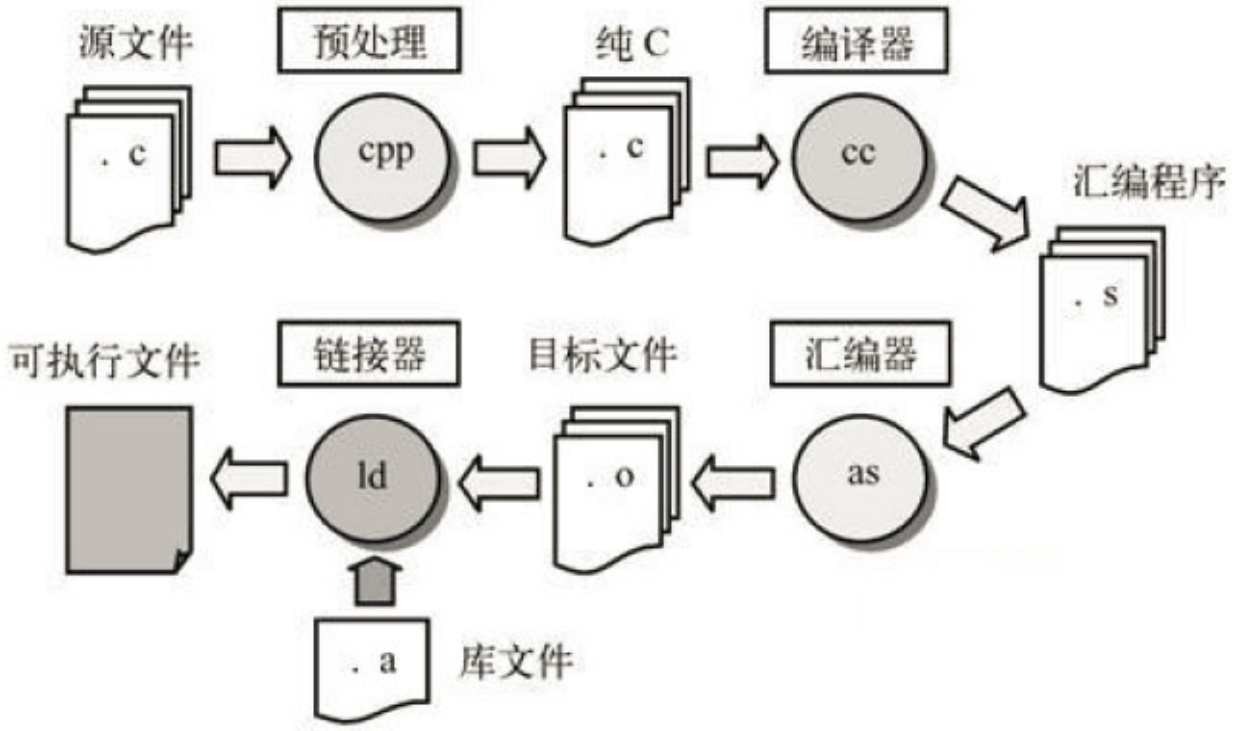
编译，汇编，链接

- gcc -c (编译)
- gcc (链接或者编译+ 链接)

- gcc hello.c –o hello.bin其实是先编译成obj文件hello.out，再连接成hello.bin
- gcc编译生成的obj文件和nasm汇编生成的obj文件是统一的格式。

- 所以，可以使用gcc编译c语言，nasm汇编汇编语言，最后使用链接器（通常是ld命令）将多个obj文件链接成可执行文件。
- 但是，obj文件和可执行文件都是平台相关的，比如linux下面是ELF格式，mac下面是maco格式。

复习-C语言的编译链接图



其它说明

- 64位ubuntu
  - 首先安装32位库
    - sudo apt-get install gcc-multilib
  - nasm -f elf32 func.asm
  - gcc -m32 main.c func.o
- mac系统
  - nasm的参数需要修改为maco
  - nasm -hf 可以查看各个平台的参数。windows为win32

其它说明

- nasm 命令
  - 参见nasmdoc第二章
- gcc 命令
  - http://man.linuxde.net/gcc

Chapter 2: Running NASM

2.1 NASM Command-Line Syntax

To assemble a file, you issue a command of the form

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

For example,

```
nasm -f elf myfile.asm
```

will assemble `myfile.asm` into an `ELF` object file `myfile.o`. And

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

will assemble `myfile.asm` into a raw binary file `myfile.com`.

To produce a listing file, with the hex codes output from NASM displayed on the left of the original sources, use the `-l` option to give a listing file name, for example:

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

To get further usage instructions from NASM, try typing

```
nasm -h
```

As `-hf`, this will also list the available output file formats, and what they are.

If you use Linux but aren't sure whether your system is `a.out` or `ELF`, type

```
file nasm
```

(in the directory in which you put the NASM binary when you installed it). If it says something like

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

then your system is `ELF`, and you should use the option `-f elf` when you want NASM to produce Linux object files. If it says

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

or something similar, your system is `a.out`, and you should use `-f aout` instead (Linux `a.out` systems have long been obsolete, and are rare these days.)

Like Unix compilers and assemblers, NASM is silent unless it goes wrong: you won't see any output at all, unless it gives error messages.

2.1.1 The -o Option: Specifying the Output File Name

NASM will normally choose the name of your output file for you; precisely how it does this is dependent on the object file format. For Microsoft object file formats (`obj`, `win32` and `win64`), it will remove the `.asm` extension (or whatever extension you like to use – NASM doesn't care) from your source file name and substitute `.obj`. For Unix object file formats (`aout`, `as86`, `coff`, `elf32`, `elf64`, `elfx32`, `ieee`, `macho32` and `macho64`) it will substitute `.o`. For `dbg`, `rdf`, `ith` and `srec`, it will use `.dbg`, `.rdf`,

静态链接与动态链接

- 静态链接：静态链接就是在编译链接时直接将需要的执行代码拷贝到调用处

  使用静态链接生成的可执行文件体积较大，包含相同的公共代码，造成浪费

- 动态链接：使用这种方式的程序并不在一开始就完成动态链接，而是直到真正调用动态库代码(调用未被本文件实现的函数代码)时，载入程序才计算(被调用的那部分)动态代码的逻辑地址。

  这种方式使程序初始化时间较短，但运行期间的性能比不上静态链接的程序

动态链接两种方法

- 装载时动态链接：这种用法的前提是在编译之前已经明确知道要调用DLL中的哪几个函数，编译时在目标文件中只保留必要的链接信息，而不含DLL函数的代码；当程序执行时，调用函数的时候利用链接信息加载DLL函数代码并在内存中将其链接入调用程序的执行空间中(全部函数加载进内存），其主要目的是便于代码共享。（动态加载程序，处在加载阶段，主要为了共享代码，共享代码内存）

- 运行时动态链接(Run-time Dynamic Linking)：这种方式是指在编译之前并不知道将会调用哪些DLL函数，完全是在运行过程中根据需要决定应调用哪个函数，将其加载到内存中（只加载调用的函数进内存），并标识内存地址，其他程序也可以使用该程序，并用LoadLibrary和GetProcAddress动态获得DLL函数的入口地址。（dll在内存中只存在一份，处在运行阶段）

重点

- 动态链接和静态链接的理解
- 汇编语言和C语言函数调用时的参数传递规则的理解

THANKS!