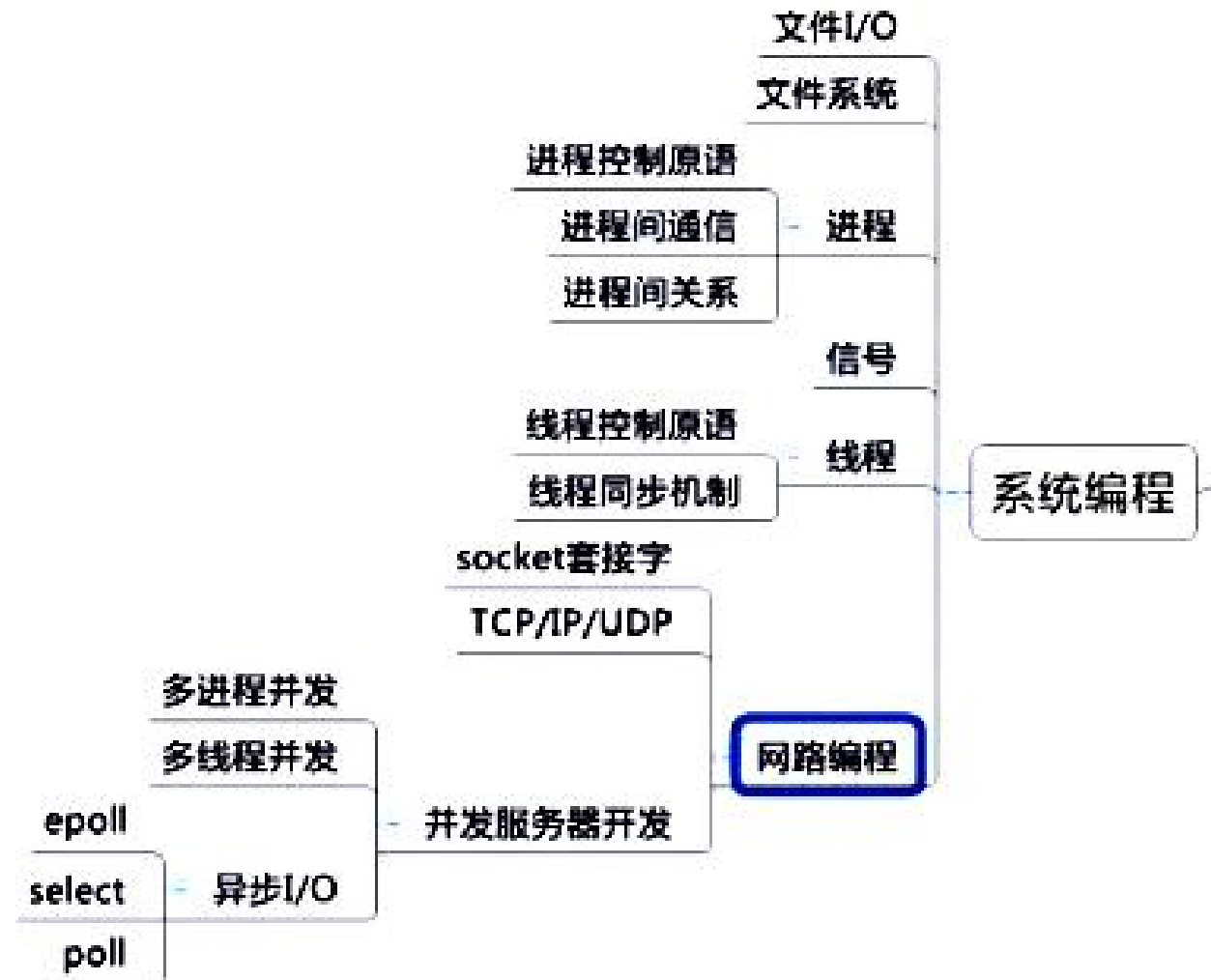




Linux C编程基础

信息工程学院



- ❑ 编辑器vim——编写代码
- ❑ 编译器gcc——编译运行代码
- ❑ 调试器gdb——调试代码
- ❑ 项目管理器Makefile——自动管理软件编译

❑ 离线安装

➤ 源码安装

从网上找到以 .tar.gz或 .tar.bz2的源码压缩文件（这两种只是压缩格式不一样，内容完全一致，下载其中一种即可）。使用# tar zxvf ****.tar.gz或# bzcat *****.tar.bz2 | tar xvf -命令解压后，用# \${srcdir}/configure --prefix=\${destdir}来配置将GCC编译器安装到什么地方，然后还要用make和make install编译安装。整个过程很麻烦也很费时。

➤ rpm包安装

从网上找到所有rpm安装包，使用#rpm -ivh *****.rpm命令安装一系列的rpm包，有些包间有依赖关系，比较麻烦。

❑ 在线安装

➤ 在root用户下使用\$ yum -y install gcc*命令即可自动安装，比较简便。

❑ \$ yum -y install gcc*

```
File Edit View Search Terminal Help
(140/148): python3-pygments-1.6-3.fc21.noarch.rpm | 956 kB 00:01
(141/148): python3-tkinter-3.4.1-16.fc21.i686.rpm | 303 kB 00:00
(142/148): tix-8.4.3-15.fc21.i686.rpm | 255 kB 00:00
(143/148): tcl-8.6.3-1.fc21.i686.rpm | 2.2 MB 00:02
(144/148): python3-test-3.4.1-16.fc21.i686.rpm | 5.9 MB 00:03
(145/148): tk-8.6.3-1.fc21.i686.rpm | 1.6 MB 00:01
(146/148): tkinter-2.7.8-9.fc21.i686.rpm | 375 kB 00:00
(147/148): python3-tools-3.4.1-16.fc21.i686.rpm | 419 kB 00:03
libstdc++-devel-4.9.2-6.fc21.i686.rpm FAILED
http://ftp.isu.edu.tw/pub/Linux/Fedora/linux/updates/21/i386/l/libstdc%2B%2B-dev
el-4.9.2-6.fc21.i686.rpm: [Errno 12] Timeout on http://ftp.isu.edu.tw/pub/Linux/
Fedora/linux/updates/21/i386/l/libstdc++-devel-4.9.2-6.fc21.i686.rpm: (28, 'Oper
ation too slow. Less than 1000 bytes/sec transferred the last 30 seconds')
Trying other mirror.
(148/148): libstdc++-devel-4.9.2-6.fc21.i686.rpm | 1.6 MB 00:02
-----
Total 803 kB/s | 574 MB 12:12
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction (shutdown inhibited)
Warning: RPMDB altered outside of yum.
Installing : cross-gcc-common-4.9.2-5.fc21.noarch 1/157
Installing : cross-binutils-common-2.25 [#####] 2/157
```

在线安装GCC

```
root@localhost:~  
File Edit View Search Terminal Help  
python3-lxml.i686 0:3.3.6-1.fc21  
python3-pygments.noarch 0:1.6-3.fc21  
python3-six.noarch 0:1.9.0-1.fc21  
python3-test.i686 0:3.4.1-16.fc21  
python3-tkinter.i686 0:3.4.1-16.fc21  
python3-tools.i686 0:3.4.1-16.fc21  
tcl.i686 1:8.6.3-1.fc21  
tix.i686 1:8.4.3-15.fc21  
tk.i686 1:8.6.3-1.fc21  
tkinter.i686 0:2.7.8-9.fc21  
  
Dependency Updated:  
glibc.i686 0:2.20-8.fc21  
gmp.i686 1:6.0.0-9.fc21  
libgomp.i686 0:4.9.2-6.fc21  
mpfr.i686 0:3.1.2-8.fc21  
python-libs.i686 0:2.7.8-9.fc21  
glibc-common.i686 0:2.20-8.fc21  
libgcc.i686 0:4.9.2-6.fc21  
libstdc++.i686 0:4.9.2-6.fc21  
python.i686 0:2.7.8-9.fc21  
  
Complete!  
[root@localhost ~]#
```



gcc编译器



- ❑ GCC全称GNU Compiler Collection，即GNU编译工具集合，是整个GNU项目中的一个重要组成部分，是GNU项目中符合ANSI C标准的编译系统，能够编译用C、C++和Object C等语言编写的程序。
- ❑ GCC逐步扩展到能支持多种语言，如Java、Fortran、Pascal、Modula-3和Ada等。
- ❑ GCC可以在多种处理器硬体平台上编译，其执行效率与一般的编译器相比，平均效率要高20%~30%。

第一个Linux c程序

例1：设计一个程序，要求在屏幕上输出“这是第一个Linux c程序!”。

步骤 1:设计编辑源程序代码

- 使用文本编辑器vi，在终端中输出：

[root@localhost root]#vi hello.c

```
#include <stdio.h>
int main()
{
    printf("hello world!\n");
}
```

- 输入完成后存盘：按ESC键→输入“:wq”回车

步骤 2:编译程序

```
[wlw@CM00 mycfile]$ ls  
hello.c  
[wlw@CM00 mycfile]$ gcc hello.c  
[wlw@CM00 mycfile]$ ls  
a.out hello.c
```

```
[wlw@CM00 mycfile]$ gcc hello.c -o hello  
[wlw@CM00 mycfile]$ ls  
a.out hello hello.c
```

步骤 3:运行程序

```
[wlw@CM00 mycfile]$ ./a.out  
hello world!
```

```
[wlw@CM00 mycfile]$ ./hello  
hello world!
```

- ❑ 在Linux系统中，可执行文件没有统一的后缀，系统从文件的属性来区分可执行文件和不可执行文件。
- ❑ gcc编译时如果没有给出可执行文件的名字，gcc将自动生成一个名为a.out的可执行文件。
- ❑ 执行文件时，在可执行文件前要加./，即运行当前目录下的可执行文件，否则报错。因为当前目录不在PATH系统变量中，所以执行程序必须加上路径（绝对路径或相对路径）。

□ gcc一般格式：

gcc [参数] 要编译的文件 [参数] [目标文件]

□ gcc常用格式：

gcc 源文件名.c //生成默认文件a.out

如： gcc hello.c

gcc 源文件名.c -o 可执行文件名 //生成指定文件

如： gcc hello.c -o hello

gcc -o 可执行文件名 源文件名.c //生成指定文件

如： gcc -o hello hello.c



linux下用gcc编译生成的执行文件前可否不加./ ?

如果不加“./”，有如下解决方法：

- ❑ 1) 把这个程序复制到PATH变量中已有的目录中去。

(查看PATH中有哪些目录用“echo \$PATH”)

```
[wlw@CM00 mycfile]$ echo $PATH  
/usr/lib64/qt-3.3/bin:/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/wlw/bin
```

- ❑ 2) 把路径“.”加入到PATH中去：执行**export PATH=\$PATH:.**

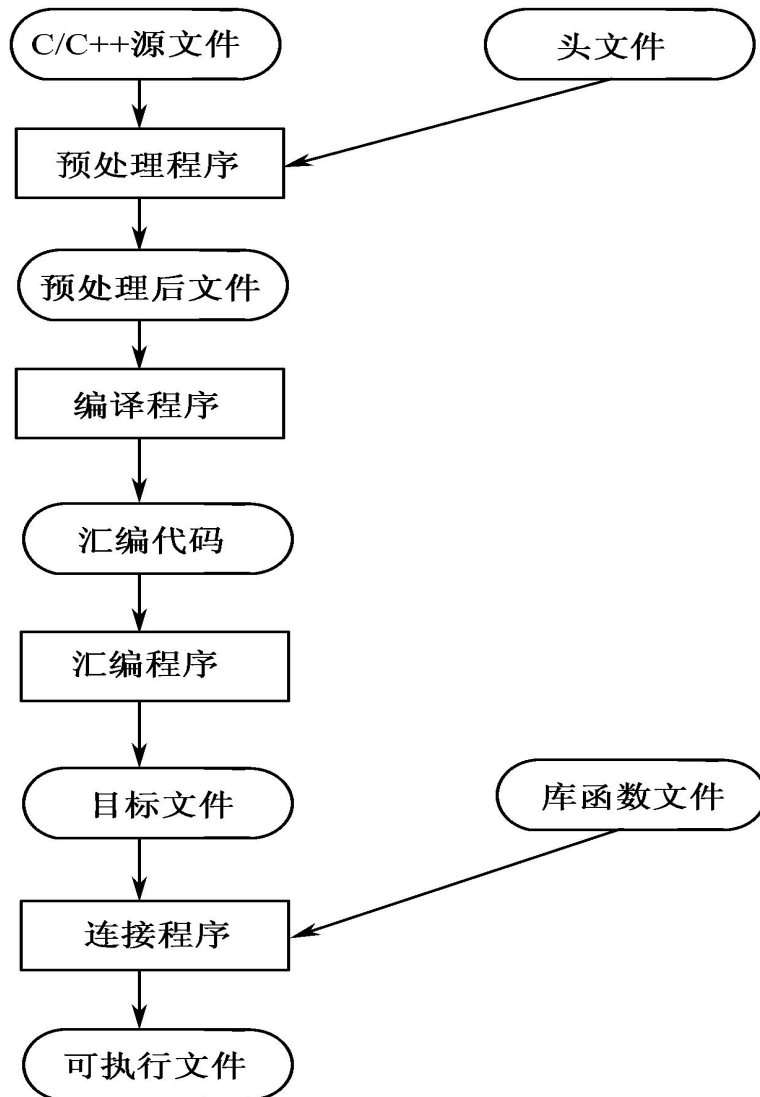
如果希望不用每次启动新BASH的时候都设置这个环境变量，则在~/.bash_profile中找到PATH变量并添加“.”路径。(在不同的系统中可能并不一样，在Ubuntu中默认是~/.profile)

值得注意：如果系统PATH中有与你程序同名的执行文件，那么执行的将不是你的程序，这是很危险的。（假如有人在某个目录下把一个木马起名ls，那么你在运行ls就会运行木马）。

总之，建议习惯使用“./”来执行当前目录的程序。

gcc则通过后缀来区别输入文件的类别：

后缀	文 件 类 型	后缀	文 件 类 型
.c	C源文件	.i	预处理后的C源文件
.C .cc .cp .cpp .c++ .cxx	C++源文件	.ii	预处理后的C++源文件
.o	目标文件	.h	头文件
.s	汇编源程序	.a	静态链接库
.S	必须预处理的汇编程序文件	.so	动态链接库



❑ 预处理阶段

❑ 编译阶段

❑ 汇编阶段

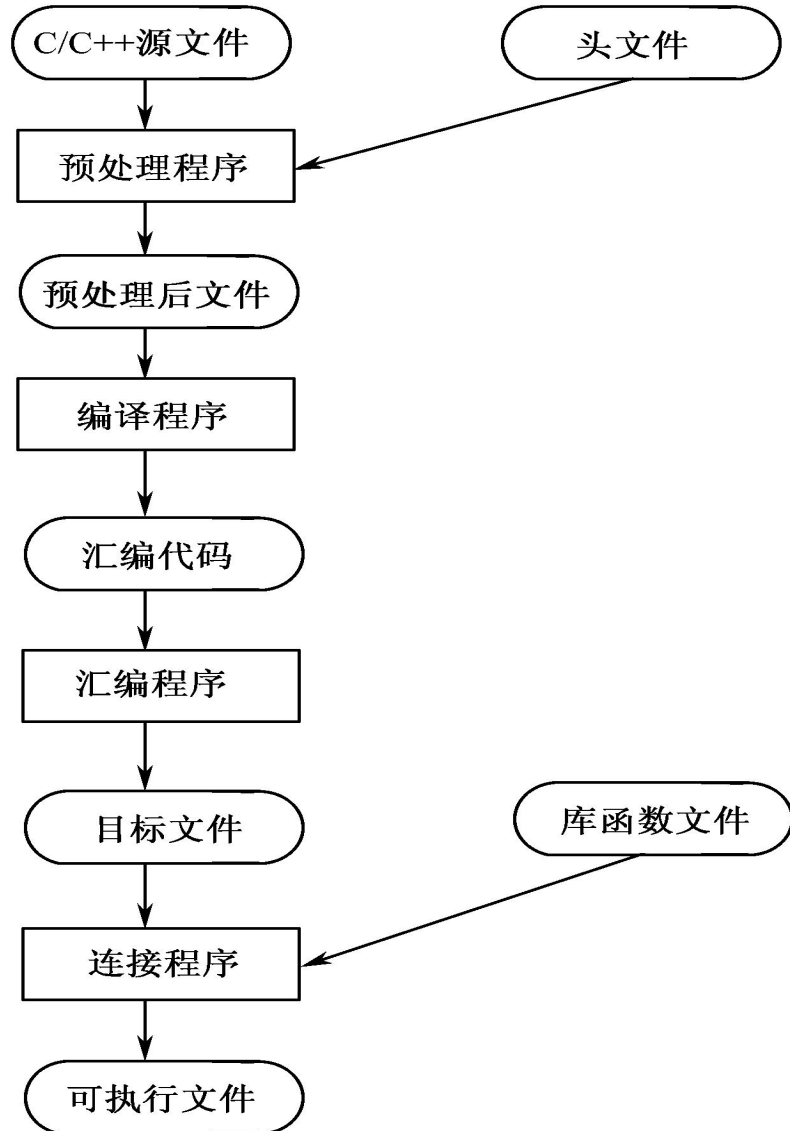
❑ 连接阶段

以下通过实例讲解

通过使用gcc的参数，控制gcc的编译过程，了解gcc的编译过程，进一步认识gcc的灵活性。

□例：编程要求输入两个整数，求和输出。

```
[root@localhost root]#vi sum.c
#include <stdio.h>
main()
{
    int a,b,sum;
    printf("please input no.1 num:");
    scanf("%d",&a);
    printf("please input no.2 num:");
    scanf("%d",&b);
    sum=a+b;
    printf("the total is %d\n",sum);
}
```



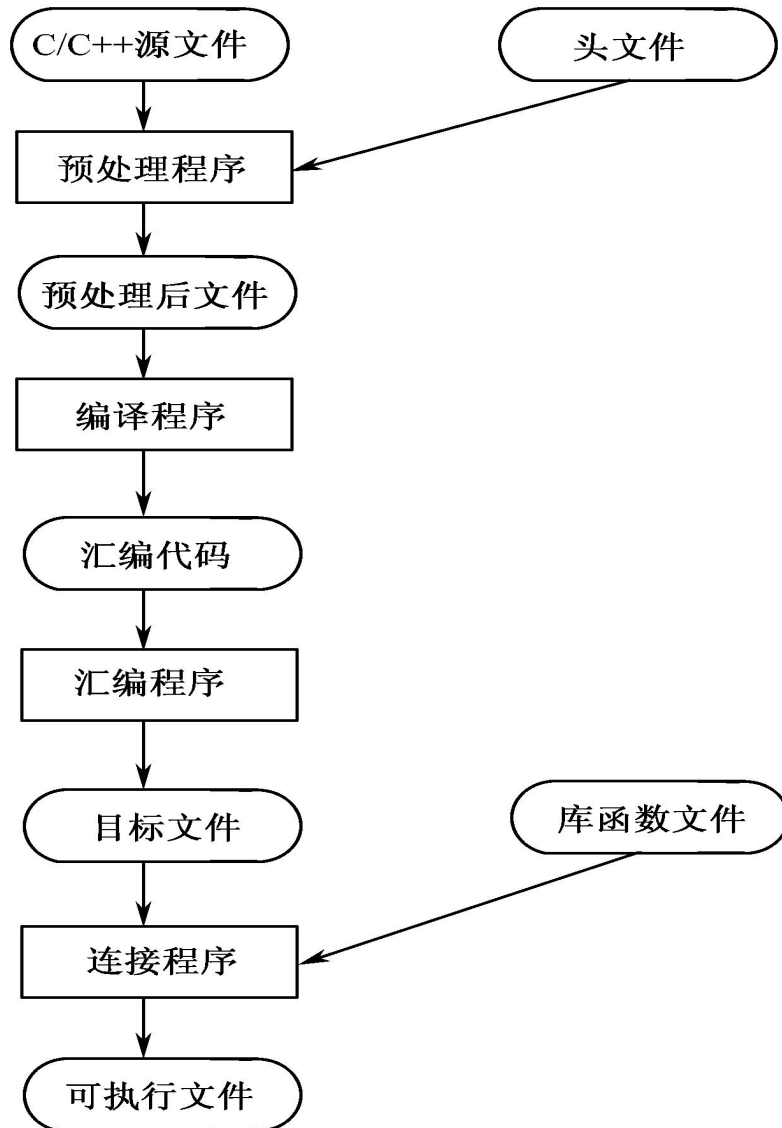
生成预处理后的文件

```
# gcc sum.c -o sum.i -E
```

□ 预处理阶段

主要包括:

语法检查、相应头文件加载、宏展开等工作。

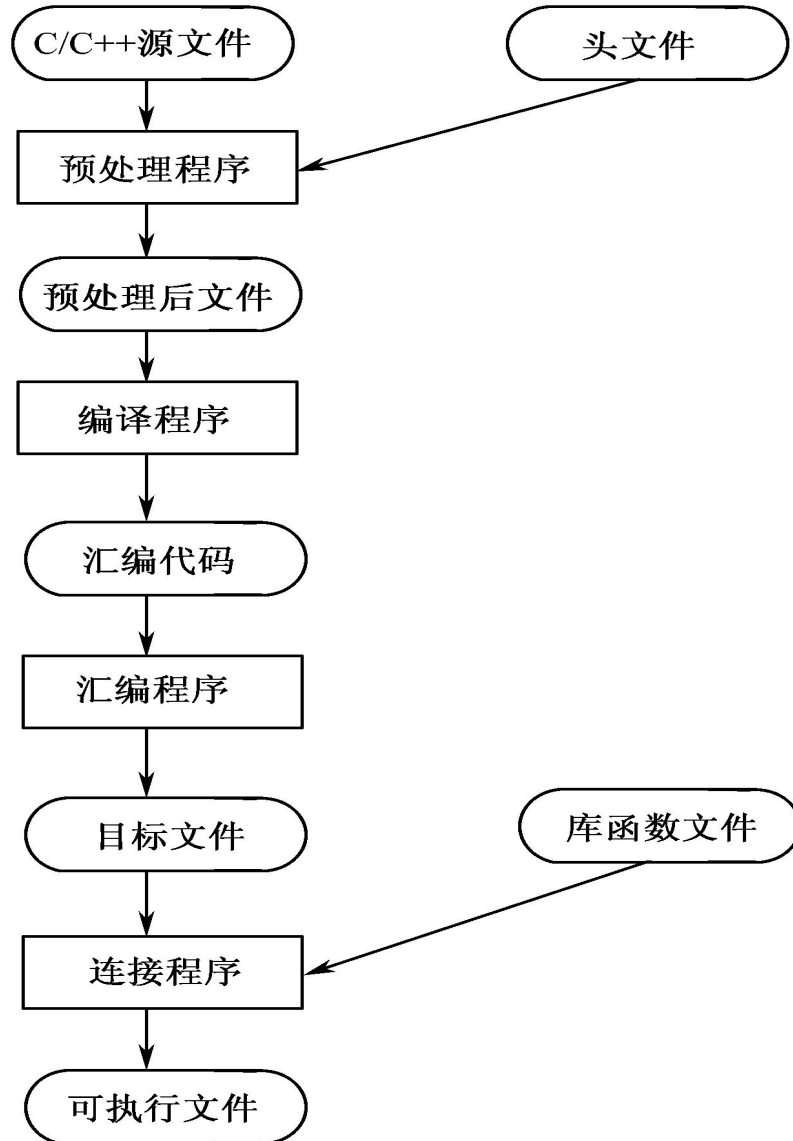


生成汇编代码

```
#gcc sum.i -o sum.s -S
```

□编译阶段

编译器对预处理之后的文件进行词法分析、语法分析和代码优化，生成汇编代码。

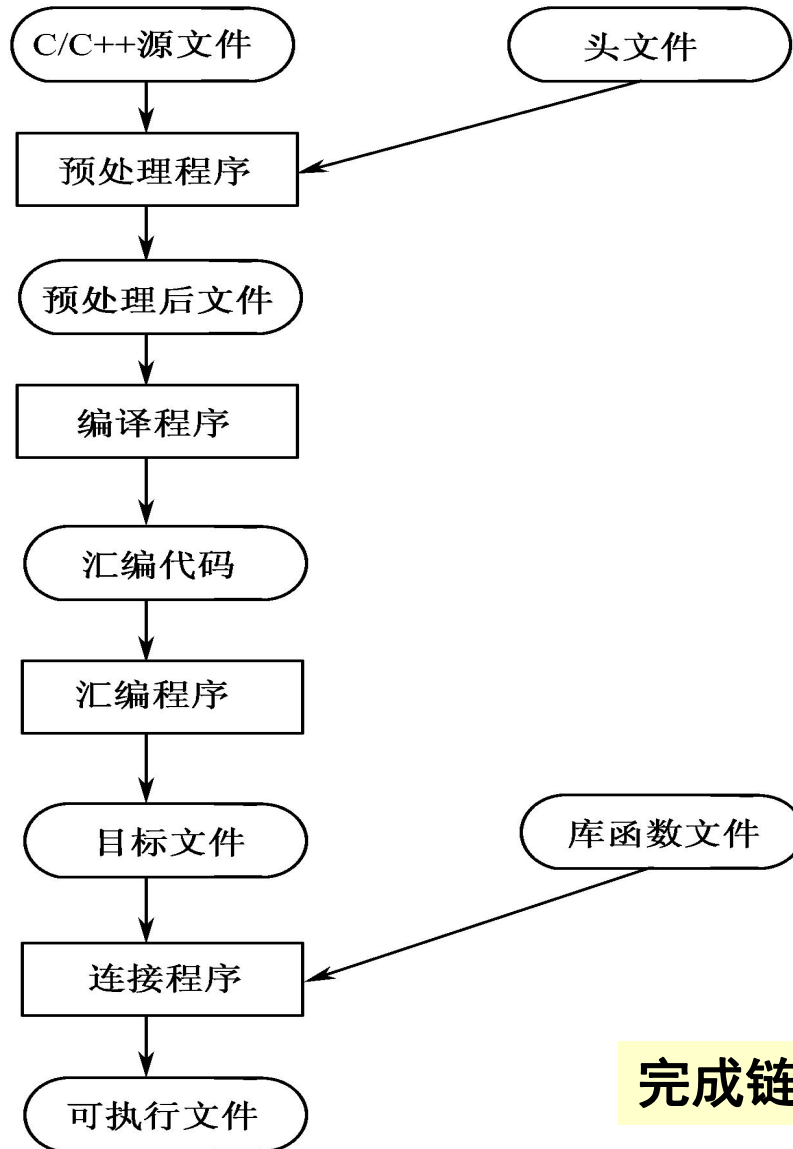


生成机器代码

```
#gcc sum.s -o sum.o -c
```

□ 汇编阶段

汇编器把汇编语言代码翻译成目标机器代码。



□ 链接阶段

把所有的.o文件和需要的库文件链接成一个完整的可执行文件。（链接模式分为静态链接和动态链接）

完成链接后，gcc就生成可执行文件。

说明

注意：gcc在编译的时候默认使用动态链接库，编译链接时并不把库文件的代码加入到可执行文件中，而是在程序执行的时候动态加载链接库，这样可以节省系统开销。

Linux下库文件的命名约定：所有的库名都以lib开头。

如：libx.a （其中，x是指定的库名）

- 以.so（共享目标，shared object）结尾的库是动态库
- 以.a（归档，archive）结尾的库是静态库
- ◆ 生成静态库的方法实际上可分为两步：
 - ① 将各函数的源文件编译成目标文件
 - ② 使用ar工具将目标文件收集放到一个归档文件中

说明：ar命令可以用来创建、修改库，也可以从库中提出单个模块。

gcc编译器的主要参数

1. 总体参数

选项	功 能
-E	只预编译不连接，得到.i文件
-S	只编译不汇编，得到汇编.s文件
-c	只编译不连接，得到.o文件，用于对源文件的分别编译。
-o file	生成指定可执行文件file。如果未使用file，则可执行文件为a.out。
-g	生成可执行程序中包含调试程序gdb使用的信息
-v	在标准出错输出上显示编译驱动程序及预处理程序的版本号

连接程序选项

连接程序常用的选项及其功能

选 项	功 能
-l <u>library</u>	连接时搜索由liblibrary命名的库，按默认规则，将开头的lib和后缀.a或.so省略后与l组合
-static	在支持动态链接的系统中，强制使用静态链接库阻止链接动态库
-L <u>dir</u>	在库文件搜索路径表中添加指定目录dir。即在搜索-l后面列举的库文件时，首先到dir下搜索，找不到再到标准位置下搜索
-I <u>dir</u>	在头文件搜索路径表中添加指定目录dir

❑ 当头文件与gcc不在同一目录下要用-I dir编译，而添加库文件时需用-L dir参数。

gcc编译器的主要参数

例：头文件为“mystd.h”是自定义的。

```
#include <mystd.h>
```

```
main()
{
    int a,b,sum;
    printf("please input no.1 num:");
    scanf("%d",&a);
    printf("please input no.2 num:");
    scanf("%d",&b);
    sum=a+b;
    printf("the total is %d\n",sum);
}
```

❑ 正常编译 # gcc sum.c -o sum //编译器提示致命错误

加-I参数 # gcc sum.c -o sum -I 当前目录 //成功编译

最佳解决方法：#include <mystd.h>——> #include “mystd.h”

在include语句中，<>表示在默认路径/usr/include中搜索头文件，
“ ”表示在本目录中搜索。

gcc编译器的主要参数

□ 例：编译含有数学函数的程序。

```
#include <stdio.h>
#include <math.h>
int main()
{
    float x,y;
    printf("input a num of sinx:");
    scanf("%f",&x);
    y=sin(x);
    printf("sin(%f)=%f\n",x,y);
}
```

```
[wlw@CM00 mycfile]$ gcc sin.c -o sin
/tmp/ccgBrIda.o: In function `main':
sin.c:(.text+0x3f): undefined reference to `sin'
collect2: ld 返回 1
```

报错原因：未定义sin，GCC默认不链接数学函数库

解决方法：# gcc sin.c -o sin -lm

要找函数sin所在库，使用命令nm

```
# nm -o /usr/lib64/*.so | grep sin --color
```

```
.....  
/usr/lib64/libm.so:0000003917a33910 t isinff  
/usr/lib64/libm.so:0000003917a3bf80 t isinfl  
/usr/lib64/libm.so:0000003917a1c750 W sin  
/usr/lib64/libm.so:0000003917a72b00 r sincos  
.....
```

❑在/usr/lib64/libm.so中除去函数库头lib余下的符号为“m”，在编译时用字符“l”与余下的符号“m”相连接成“lm”。

命令：nm [参数] File ...

- 显示关于指定 File 中符号的信息，文件是对象文件、可执行文件或对象文件库，如果指出目标文件，则是a.out。
- 参数中有-A、-o或--print-file-name：在找到的各个符号的名字前加上文件名，而不是在此文件的所有符号前只出现文件名一次。

gcc编译器的主要参数

□ gcc的常用告警和出错参数

参 数	含 义
-ansi	支持符合 ANSI 的 c 程序
-pedantic	允许发出 ANSI c 标准所列的全部警告信息
-pedantic-error	允许发出 ANSI c 标准所列的全部错误信息
-w	关闭所有告警
-Wall	允许发出 gcc 提供的所有有用的告警信息
-Werror	把所有的告警信息转化为错误信息，并在告警发生时终止编译

gcc编译器的主要参数

例：设计包含一些非标准语法的程序。

```
#include <stdio.h>
void main()
{
    long long tmp=1;
    printf("This is a test program file.\n");
    return 0;
}
```

```
[wlw@CM00 mycfile]$ gcc erralm.c -o erralm
```

erralm.c: 在函数 'main' 中:

erralm.c:6: 警告: 在无返回值的函数中, 'return' 带返回值

```
[wlw@CM00 mycfile]$
```

```
[wlw@CM00 mycfile]$ gcc erralm.c -o erralm -w
```

```
[wlw@CM00 mycfile]$
```

#gcc 源码文件.c -o 可执行文件 -w //关闭所有告警

gcc编译器的主要参数

❑ 显示不符合ANSI c标准语法的告警信息

#gcc 源码文件.c -o 可执行文件 -ansi

```
[wlw@CM00 mycfile]$ gcc erralm.c -o erralm -ansi  
erralm.c: 在函数 'main' 中:  
erralm.c:6: 警告: 在无返回值的函数中, 'return'带返回值
```

❑ 允许发出ANSI c标准所列的全部警告信息

#gcc 源码文件.c -o 可执行文件 -pedantic

```
[wlw@CM00 mycfile]$ gcc erralm.c -o erralm -pedantic  
erralm.c:2: 警告: 'main'的返回类型不是 'int'  
erralm.c: 在函数 'main' 中:  
erralm.c:4: 警告: ISO C90 不支持 'long long'  
erralm.c:6: 警告: 在无返回值的函数中, 'return'带返回值
```

❑ 允许发出gcc提供的所有有用的告警信息

#gcc 源码文件.c -o 可执行文件 -Wall

```
[wlw@CM00 mycfile]$ gcc erralm.c -o erralm -Wall  
erralm.c:2: 警告: 'main'的返回类型不是 'int'  
erralm.c: 在函数 'main' 中:  
erralm.c:6: 警告: 在无返回值的函数中, 'return'带返回值  
erralm.c:4: 警告: 未使用的变量 'tmp'
```

gcc编译器的主要参数

选 项	功 能
-O -O1	试图减少代码大小和执行时间，但并不执行需要花费大量编译时间的任何优化
-O2	在-O1级别的优化之上，还进行一些额外调整工作——除不做循环展开、函数内联和寄存器重新命名外，几乎进行所有可选优化
-O3	除了完成所有-O2级别的优化之外，还进行包括循环展开和其他一些与处理器特性相关的优化工作
-O0	不执行优化
-Os	具有-O2级别的优化，同时并不特别增加代码大小

优化参数

1. 代码优化指的是编译器通过分析源代码，找出其中尚未达到最优的部分，然后对其重新进行组合，改善程序的执行性能。
2. gcc提供的代码优化功能非常强大，通过编译参数“-On”来控制优化代码的生成，通常来说，数字n越大优化的等级越高，同时也就意味着程序的运行速度越快。

gcc编译器的主要参数

□ 例：设计一个优化前后变化很大的程序。

```
#include <stdio.h>
int main(void)
{
    double con, res, tmp;
    for(con=0; con<4000.0*4000.0*4000.0/20.0+2030; con+=(5-3+2+1)/4)
    {
        tmp=con/1239;
        res=con;
    }
    printf("The result is :%lf\n", res);
}
```

```
[wlw@CM00 mycfile]$ gcc opton.c -o opton
[wlw@CM00 mycfile]$ ./opton
The result is :3200002029.000000
```

gcc编译器的主要参数

- 分别在不加任何优化参数和加“-O2”优化参数后进行编译运行。
分别用time命令大致统计出该程序在运行时所需时间对比。

```
[wlw@CM00 mycfile]$ gcc opton.c -o opton
[wlw@CM00 mycfile]$ time ./opton
The result is :3200002029.000000
```

```
real    0m14.223s
user    0m14.216s
sys     0m0.006s
```

```
[wlw@CM00 mycfile]$ gcc opton.c -o opton -O2
[wlw@CM00 mycfile]$ time ./opton
The result is :3200002029.000000
```

```
real    0m7.712s
user    0m7.709s
sys     0m0.002s
```

- 优化虽然能够给程序带来更好的执行性能，但在一些场合中应该避免优化代码。
程序开发时、资源受限时、跟踪调试时

程序调试——GDB调试器简介

GNU组织开发并发布的UNIX/Linux下的程序调试工具。

- ❑ 没有图形化界面
- ❑ 调试的对象是包含调试信息的可执行文件，而不是源文件。
- ❑ 必须在调试前用gcc编译时加上-g参数，使程序在编译时包含调试信息，否则生成的可执行文件不能用gdb调试。

```
[wlw@CM00 mycfile]$ gcc -g hello.c -o hello
[wlw@CM00 mycfile]$ gdb hello
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-92.el6)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/wlw/mycfile/hello...done.
(gdb) █
```

gdb基本命令

- 1) file命令: 装入想要调试的可执行文件。
- 2) cd命令: 改变工作目录。
- 3) pwd命令: 返回当前工作目录。
- 4) run命令: 执行当前被调试的程序。
- 5) kill命令: 停止正在调试的应用程序。
- 6) list命令: 列出正在调试的应用程序的源代码。
- 7) break命令: 设置断点。
- 8) Tbreak命令: 设置临时断点。它的语法与break相同。
区别在于用tbreak设置的断点执行一次之后立即消失。
- 9) watch命令: 设置监视点, 监视表达式的变化。

- 10) **awatch**命令：设置读写监视点。当要监视的表达式被读或写时将应用程序挂起。它的语法与**watch**命令相同。
- 11) **rwatch**命令：设置读监视点，当监视表达式被读时将程序挂起，等待调试。此命令的语法与**watch**相同。
- 12) **next**命令：执行下一条源代码，**但是不进入函数内部**。也就是说，将一条函数调用作为一条语句执行。执行这个命令的前提是已经**run**，开始了代码的执行。
- 13) **step**命令：执行下一条源代码，**进入函数内部**。如果调用了某个函数，会跳到函数所在的代码中等候一步步执行。执行这个命令的前提是已经用**run**开始执行代码。
- 14) **display**命令：在应用程序每次停止运行时显示表达式的值。

- 15) **info break**命令：显示当前断点列表，包括每个断点到达的次数。
- 16) **info files**命令：显示调试文件的信息。
- 17) **info func**命令：显示所有的函数名。
- 18) **info local**命令：显示当前函数的所有局部变量的信息。
- 19) **info prog**命令：显示调试程序的执行状态。
- 20) **print**命令：显示表达式的值。
- 21) **delete**命令：删除断点。指定一个断点号码，则删除指定断点。不指定参数则删除所有的断点。
- 22) **Shell**命令：执行Linux Shell命令。
- 23) **make**命令：不退出gdb而重新编译生成可执行文件。
- 24) **Quit**命令：退出gdb。

常用的gdb命令格式

- 显示源文件中指定的函数或代码行

```
list                                list -  
list [file:] num                   list start , end  
list [file:]function
```

- print命令 一般格式 : print [/fmt] exp

当被调试的程序停止时，可以用print命令（简写为p）或同义命令inspect来查看当前程序中运行的数据。

在print /fmt exp命令中，“/”之后的fmt是表示输出格式的字母，它由表示格式的字母和表示数据长度的字母组成。

表示格式的字母: o x d u t f a i c s

表示长度的字母: b w h g

控制程序的执行

□ 设置和显示断点

(1) 设置断点：用break命令（其缩写形式为b）设置断点：

```
break linenum
```

```
break linenum if condition
```

```
break function
```

```
break file:linenum
```

```
break file:function
```

(2) 显示断点信息

```
info breakpoints [num]
```

```
info break [num]
```

□ 运行程序

run命令的格式: `run [args]`

□ 程序的单步跟踪和连续执行

(1) 单步跟踪

实行单步跟踪的命令是step和next, 其格式是:

`step [N]`

`next [N]`

(2) 连续执行

`continue`, `c`或`fg`命令

程序维护工具make

- ❑ make是一个解释makefile中指令的命令工具，一般大多数的IDE都有这个命令，比如：Delphi的make，Visual C++的make，Linux下GNU的make。
- ❑ 一个工程中的源文件其按类型、功能、模块分别放在若干个目录中，makefile定义了一系列的规则来指定，哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作。
- ❑ make采用的是增量编译，根据文件修改的时间自动判断哪些需要重新编译，哪些可利用上一次的结果，从而避免了大量的重复计算，提高编译效率。
- ❑ 只需要一个make命令，整个工程完全自动编译，极大的提高了软件开发的效率。

程序维护工具make

一个工程中的源文件其按类型、功能、模块分别放在若干个目录中。
如: 一位程序员完成加法功能, 编写了**add.h**和**add.c**, 生成了**add.o**
一位程序员完成减法功能, 编写了**sub.h**和**sub.c**, 生成了**sub.o**
因此, 只需要交付**.o**文件, 不需要提交**.c**文件, 即可使用。

```
//-----add.h-----  
int add(int,int);
```

```
//-----add.c-----  
int add(int a,int b)  
{  
    return a+b;  
}
```

```
[wlw@CM00 mymkfile]$ gcc -c add.c -o add.o  
[wlw@CM00 mymkfile]$
```

```
//-----sub.h-----  
int sub(int,int);
```

```
//-----sub.c-----  
int sub(int a,int b)  
{  
    return (a-b);  
}
```

```
[wlw@CM00 mymkfile]$ gcc -c sub.c -o sub.o  
[wlw@CM00 mymkfile]$
```

现一项目需要使用加法和减法功能，则另一程序员编写了主程序**mymain.c**，**gcc**编译时使用**add.o**和**sub.o**即可。

```
#include <stdio.h>
#include "add.h"
#include "sub.h"
int main()
{
    printf("%d+%d=%d\n", 1, 2, add(1, 2));
    printf("%d-%d=%d\n", 3, 4, sub(3, 4));
    return 0;
}
```

```
[wlw@CM00 mymkfile]$ gcc -c mymain.c -o mymain.o
[wlw@CM00 mymkfile]$
[wlw@CM00 mymkfile]$ gcc mymain.o add.o sub.o -o mymain
[wlw@CM00 mymkfile]$

[wlw@CM00 mymkfile]$ ./mymain
1+2=3
3-4=-1
```

Makefile介绍

如果项目中要增加功能或修改已有功能，则整个项目的各功能都需要重新编译，工作量大，故使用Makefile可以提高效率。

比如：

- 如果工程没有编译过，所有C文件都必须要编译并被链接
- 如果工程中的某几个C文件被修改，只需编译被修改的C文件，并链接目标程序。
- 如果工程的头文件改变了，需编译引用这个头文件的C文件，并链接目标程序。

- ❑ makefile关系到了整个工程的编译规则。
- ❑ makefile就像一个Shell脚本一样，其中也可以执行操作系统的命令。一旦写好，只需要一个make命令，整个工程完全自动编译，极大的提高了软件开发的效率。
- ❑ makefile带来的好处就是——“**自动化编译**”。



Makefile里有什么



Makefile里主要包含了五种类型的语句：**显式规则、隐晦规则、变量定义、文件指示和注释。**

1、**显式规则**。显式规则在Makefile中写出如何生成一个或多的的目标文件，即要明显指出要生成的文件，文件的依赖文件，生成的命令。

2、**隐晦规则**。make有自动推导的功能，隐晦规则可以使程序员简略地书写Makefile。

3、**变量的定义**。在Makefile中定义的变量一般都是字符串，当Makefile被执行时，其中的变量都会被替换到相应的引用位置上。

4、**文件指示**。其包括了三个部分，一个是在一个Makefile中引用另一个Makefile，就像C语言中的include一样；另一个是指根据某些情况指定Makefile中的有效部分，就像C语言中的预编译#if一样；还有就是定义一个多行的命令。

5、**注释**。Makefile中用“#”字符只有行注释，和Shell脚本一样。如果在Makefile中要使用“#”字符，则用反斜框进行转义，如：“\#”。

makefile基本格式:

目标文件: [依赖文件...]

<tab>命令1 [#注释]

...

<tab>命令n [#注释]

注意: 每个命令行的开头必须为[TAB]字符!

```
#目标: 依赖
mymain: mymain.o add.o sub.o
        gcc mymain.o add.o sub.o -o mymain
mymain.o: mymain.c
        gcc -c mymain.c -o mymain.o
add.o: add.c
        gcc -c add.c -o add.o
sub.o: sub.c
        gcc -c sub.c -o sub.o
```

```
#目标： 依赖
mymain: mymain.o add.o sub.o
        gcc mymain.o add.o sub.o -o mymain
mymain.o: mymain.c
        gcc -c mymain.c -o mymain.o
add.o: add.c
        gcc -c add.c -o add.o
sub.o: sub.c
        gcc -c sub.c -o sub.o
```

在格式上应注意：

- ❑ **目标文件**：以空格分开，可以使用通配符。一般目标基本上是一个文件，但也有可能是多个文件。
- ❑ **依赖文件**：目标文件和依赖文件之间要用一个或两个冒号分开。
- ❑ **命令行**：各命令行单独占一行，必须以[Tab键]开头，而不能使用8个空格。如果命令太长，可以使用反斜框‘\’作为换行符。make对一行上字符个数没有限制。如果命令行与“目标文件：[依赖文件…]”在一行，那么可以用分号做为分隔。

```
#目标： 依赖
mymain: mymain.o add.o sub.o
        gcc mymain.o add.o sub.o -o mymain
mymain.o: mymain.c
        gcc -c mymain.c -o mymain.o
add.o: add.c
        gcc -c add.c -o add.o
sub.o: sub.c
        gcc -c sub.c -o sub.o
```

写好makefile文件，在命令行中直接键入**make**命令执行。

```
[wlw@CM00 demo1]$ ls
add.c add.h makefile mymain.c sub.c sub.h
[wlw@CM00 demo1]$ make
gcc -c mymain.c -o mymain.o
gcc -c add.c -o add.o
gcc -c sub.c -o sub.o
gcc mymain.o add.o sub.o -o mymain
[wlw@CM00 demo1]$ ls
add.c add.h add.o makefile mymain mymain.c mymain.o sub.c sub.h sub.o
[wlw@CM00 demo1]$ ./mymain
1+2=3
3-4=-1
```

注意：如果报makefile: 2: *** missing separator. Stop. 提示信息时，说明makefile文件中，命令行没有以<tab>键开始，而是以空格开始。

```
#目标： 依赖
mymain: mymain.o add.o sub.o
        gcc mymain.o add.o sub.o -o mymain
mymain.o: mymain.c
        gcc -c mymain.c -o mymain.o
add.o: add.c
        gcc -c add.c -o add.o
sub.o: sub.c
        gcc -c sub.c -o sub.o
```

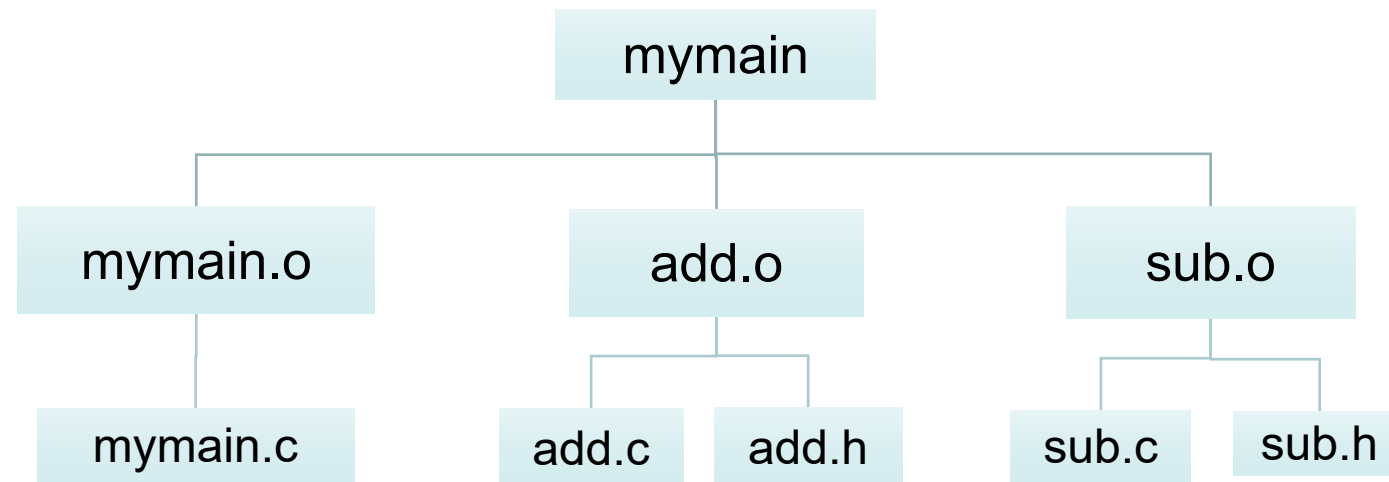
默认方式下，make工作过程：

- ❑ make在当前目录下找名字叫Makefile或makefile的文件。
- ❑ 在makefile文件中的第一个目标文件，作为最终的目标文件。
- ❑ 若目标文件不存在、或有任意一个依赖文件不存在、或有任意一个依赖文件比目标文件新，则会找到相应的依赖关系生成后再执行。

make的依赖性，make会一层又一层地去找文件的依赖关系，直到最终编译出第一个目标文件。如果出现比如被依赖的文件找不到，那么make直接退出并报错，而对于所定义的错误或编译不成功，make根本不理睬。

使用make的一个核心问题是确定好各文件之间的依赖关系。

- 生成一个目标文件可能有多种不同途径，能够指定不同的依赖关系。
- 适当地引入中间结果，合理地构造依赖关系，可以省去一部分编译工作量。
- make是依据“关系图深度优先搜索”算法来核查目标文件及相依文件的修改时间。并非层次越多越好，要考虑目标文件的生成过程及其所起的作用。



```
#目标： 依赖
mymain:mymain.o add.o sub.o
        gcc mymain.o add.o sub.o -o mymain
mymain.o:mymain.c
        gcc -c mymain.c -o mymain.o
add.o:add.c
        gcc -c add.c -o add.o
sub.o:sub.c
        gcc -c sub.c -o sub.o
clean:
        rm -rf *.o mymain
```

作用：执行“**make clean**”可以删除执行文件和所有中间目标文件。

```
[wlw@CM00 demo1]$ ls
add.c add.h add.o makefile mymain mymain.c mymain.o sub.c sub.h sub.o
[wlw@CM00 demo1]$
[wlw@CM00 demo1]$ make clean
rm -rf *.o mymain
[wlw@CM00 demo1]$ ls
add.c add.h makefile mymain.c sub.c sub.h
```

自动变量

make中定义的一些因环境的不同而发生改变的变量

- ❑ `$@` 规则中的目标文件集合
- ❑ `$$` 规则中的依赖文件的集合
- ❑ `$?` 所有比目标文件还新的那些依赖文件的集合
- ❑ `$<` 规则中的第一个依赖文件
- ❑ `$%` 仅当目标文件是一个静态库成员时，表示规则中的目标成员名，而此时`$@`表示相应库文件的名称
- ❑ `$*` 如果目标文件的后缀是make所识别的，则`$*`就是去掉后缀的目标文件名，但该引用只有用在隐含规则中才有意义

- ❑ `%`: 与任何非空字符串匹配

```
#目标： 依赖
mymain:mymain.o add.o sub.o
        gcc mymain.o add.o sub.o -o mymain
mymain.o:mymain.c
        gcc -c mymain.c -o mymain.o
add.o:add.c
        gcc -c add.c -o add.o
sub.o:sub.c
        gcc -c sub.c -o sub.o
clean:
        rm -rf *.o mymain
```

```
#目标： 依赖
mymain:mymain.o add.o sub.o
        gcc *.o -o $@
%.o:%.c
        gcc -c $^ -o $@
clean:
        rm -rf *.o mymain
```

%	与任何非空字符串匹配；
\$@	规则中目标文件集合；
\$^	规则中依赖文件集合；

make变量，又称宏定义，**像一个环境变量**，对大小写敏感，一般由大写字母和数字组成。

□ 定义make变量的一般格式：<变量名>=<字符串>

例如： **OBJECT=x.o y.o z.o**

LIBES=-lm

□ 引用make变量的方式：把变量用圆括号括起来，并在前面加上“\$”符号。几乎可以从任何地方引用定义的变量。

例如： **\$(OBJECT)**

\$(LIBES)



变量的主要作用如下：

(1) 保存文件名列表。

作为依赖文件的一些目标文件名出现在可执行文件的规则中，而在这个规则的命令里同样包含这些文件并传递给gcc做为命令参数。

如果使用一个变量来保存所有的目标文件名，则可以方便地加入新的目标文件而且不易出错。

(2) 保存可执行命令名，如编译器。

在不同的Linux系统中存在着很多相似的编译器系统，这些系统在有些地方会有细微的差别，如果项目被用在一个非gcc的系统里，则必须将所有出现编译器名的地方改成用新的编译器名。

如果使用一个变量来代替编译器名，那么只需要改变该变量的值。

一个简单的Makefile的例子

```
test:  prog.o code.o
        gcc -o test prog.o code.o
prog.o: prog.c prog.h code.h
        gcc -c prog.c -o prog.o
code.o: code.c code.h
        gcc -c code.c -o code.o
clean:
        rm -f *.o
```

```
OBJS = prog.o  code.o
CC=gcc
test:  ${OBJS}
        ${CC} -o test ${OBJS}
prog.o: prog.c  prog.h  code.h
        ${CC} -c prog.c -o prog.o
code.o: code.c code.h
        ${CC} -c code.c -o code.o
clean:
        rm -f *.o
```

(3) 保存编译器的参数。

在很多源代码编译时，gcc需要的参数很长。

如果所有的编译命令使用一组相同的选项，则把这组选项使用一个变量代表，那么可以把这个变量放在所有引用编译器的地方。

```
OBJS = file1.o file2.o
```

```
CC = gcc
```

```
CFLAGS = -Wall -O -g
```

#-Wall: 输出所有的警告信息；-O: 在编译时进行优化；-g: 表示编译debug版本。

```
helloworld : $(OBJS)
```

```
$(CC) $(OBJS) -o helloworld
```

```
file1.o : file1.c file2.h
```

```
$(CC) $(CFLAGS) -c file1.c -o file1.o
```

```
file2.o : file2.c file2.h
```

```
$(CC) $(CFLAGS) -c file2.c -o file2.o
```

```
clean:
```

```
rm -rf *.o helloworld
```

□ make命令格式: make [参数]...

参数说明:

- f FILE** 指定**Makefile**文件，缺省设置为当前目录下的**GNUmakefile**、**Makefile**、**makefile**。
- C DIR** 制定**Makefile**文件的目录。
- n** 只检查**Makefile**的运行流程，不进行真正的操作。
- t** 只更新目标文件的修改时间，不进行真正的操作。
- B** 重新编译所有的目标文件。
- p** 输出**Makefile**文件中所有的信息。
- s** 运行时不显示任何输出。
- r** 禁止使用隐含规则。
- d** 输出所有的调试信息
- e** 指明环境变量优先于**makefile**文件中的变量
- l** 忽略在执行重新生成文件的命令的过程中出现的所有错误
- l dir 或 -ldir** 指定一个包含**makefile**文件的搜索目录



makefile文件中的隐式规则（补充）



- ❑ 在makefile文件中显式地指定了一些规则，称为**显式规则**。
- ❑ **隐式规则**就是一种惯例，即预先约定好了。
- ❑ 如果要使用隐含规则生成所需目标，所要做的就是不要写出这个目标的规则。
- ❑ make会试图去自动推导产生这个目标的规则和命令，如果make可以自动推导生成这个目标的规则和命令，那么这个行为就是**隐含规则的自动推导**。

例如，下面的一个Makefile：

```
foo : foo.o bar.o
```

```
cc -o foo foo.o bar.o $(CFLAGS) $(LDFLAGS)
```

□ 这个Makefile中并没有写下如何生成foo.o和bar.o这两目标的规则和命令。因为make的“隐含规则”功能会自动去推导这两个目标的依赖目标和生成命令。

◆ make 会在自己的“隐含规则”库中寻找可以用的规则，如果找到，那么就会使用。如果找不到，那么就会报错。在上面的那个例子中，make调用的隐含规则是，把[.o]的目标的依赖文件置成[.c]，并使用C的编译命令“cc -c \$(CFLAGS) [.c]”来生成[.o]的目标。也就是说，完全没有必要写下面的两条规则：

```
foo.o : foo.c
```

```
cc -c foo.c $(CFLAGS)
```

```
bar.o : bar.c
```

```
cc -c bar.c $(CFLAGS)
```

□ 因为，这已经是“约定”好了的事了，make和我们约定好了用C编译器“cc”生成[.o]文件的规则，这就是隐含规则。

□ 当然，如果我们为[.o]文件书写了自己的规则，那么make就不会自动推导并调用隐含规则，它会按照我们写好的规则执行。

几个常用的隐式规则：

1、编译C语言程序的隐式规则

“<n>.o” 的目标的依赖目标会自动推导为 “<n>.c” ，并且其生成命令是 “**`$(CC) -c $(CPPFLAGS) $(CFLAGS)`**”

2、编译C++程序的隐含规则

“<n>.o” 的目标的依赖目标会自动推导为 “<n>.cc” 或是 “<n>.C” ，并且其生成命令是 “**`$(CXX) -c $(CPPFLAGS) $(CFLAGS)`**” 。（建议使用 “.cc” 作为C++源文件的后缀，而不是 “.C” ）

3、汇编和汇编预处理的隐式规则

“<n>.o” 的目标的依赖目标会自动推导为 “<n>.s” ，默认使用编译品 “as” ，并且其生成命令是： “**`$(AS) $(ASFLAGS)`**” 。“<n>.s” 的目标的依赖目标会自动推导为 “<n>.S” ，默认使用C预编译器 “cpp” ，并且其生成命令是： “**`$(AS) $(ASFLAGS)`**” 。

4、链接Object文件的隐含规则

“<n>” 目标依赖于 “<n>.o” ，通过运行C的编译器来运行链接程序生成（一般是 “ld” ），其生成命令是： “**`$(CC) $(LDFLAGS) <n>.o $(LOADLIBES) $(LDLIBS)`**” 。这个规则对于只有一个源文件的工程有效，同时也对多个Object文件（由不同的源文件生成）的也有效。

❑ 在make的“隐含规则库”中，每一条隐含规则都在库中有其顺序，越靠前的则是越先被使用的，所以，**这会导致有时即使显示地指定了目标依赖，make也不会管用。**

❑ 如下面这条规则（没有命令）：

```
foo.o : foo.p
```

依赖文件“foo.p”（Pascal程序的源文件）有可能变得没有意义。如果目录下存在了“foo.c”文件，那么隐含规则一样会生效，并会通过“foo.c”调用C的编译器生成foo.o文件。因为，在隐含规则中，Pascal的规则出现在C的规则之后，所以，make找到可以生成foo.o的C的规则就不再寻找下一条规则了。如果你确实不希望任何隐含规则推导，那么，就不要只写出“依赖规则”，而不写命令。