第六章 常用开发工具

主要内容

- GNU C编译器--gcc
- GNU调试工具--gdb
- 程序维护工具--make

gcc

文件名后缀

gcc简介

gcc编译过程

文件名后缀

文件名后缀	文件类型
.c	c源文件
.C .cpp .cc .c++ .cxx	c++源文件
.h	头文件
.i	预处理后的c源文件
.s	汇编程序文件
.o	目标文件
.a	静态链接库
.so	动态链接库

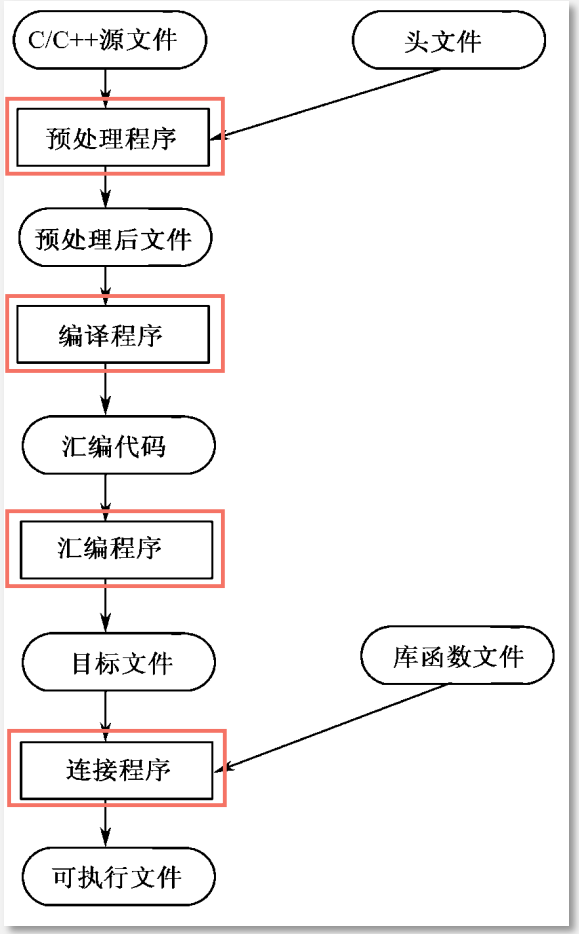
gcc简介

- gcc（GNU Compiler Collection）
  - 一个工具集合，包含预处理器、编译器、汇编器、链接器等组件
  - gcc的使用格式

    gcc [options] [filenames]
  - 说明

    当不用任何选项时，gcc将会生成一个名为a.out的可执行文件

gcc的编译过程



gcc的编译过程

预处理选项

选 项	说 明
-E	只对文件进行预处理，不进行编译，生成的结果送标准输出
-o file	将输出写到指定的文件file中
-I dir	指定头文件的路径dir。先在指定的路径中搜索要包含的头文件，若找不到，则在标准路径（/usr/include，/usr/lib及当前工作目录）上搜索
-D name	定义一个宏name，并可以指定值

gcc的编译过程

- 预处理选项
  - 例子

```
gcc -E test1.c                          //只进行预处理
gcc -E test1.c -o test1.i               //预处理并生成test1.i
gcc -I /usr/openwin/include file.c
```

gcc的编译过程

- 预处理选项
  - 例2:

```
#define DOPTION "testing"
main()
{
        printf("display -D variable %s\n", DOPTION);
        printf("hello,everybody..\n");
}
~
```

gcc -D DOPTION=' "testing" ' test2.c

gcc的编译过程

- 编译成汇编代码
  - 预处理文件→汇编代码
  - 源程序→汇编代码
- 编译选项

选 项	说 明
-S	只进行编译，不进行汇编，生成汇编代码文件.s

- 例子

```
gcc -S test1.c          //生成test1.s
gcc -S test1.i
```

gcc的编译过程

- 编译成目标代码
  - 汇编代码→目标代码
  - 源程序→目标代码
- 编译选项

选 项	说 明
-c	只把源文件编译成目标代码（.o），不进行链接

  - 例子

```
gcc -c test1.c          //生成test1.o
gcc -c test1.s
```

gcc的编译过程

- 编译成执行代码
  - 目标代码→执行代码
  - 源文件→执行代码
- 编译选项

选 项	说 明
-o file1 file1.c	将文件file1.c编译成可执行文件file1； 如果未使用该选项，则可执行文件放在a.out中

  - 例子

```
gcc -o test1 test1.c
gcc -o test1 test1.o
```

gcc的编译过程

- 多文件的编译

```c
//meng1.c
#include <stdio.h>
int main()
{
        int r;
        printf("enter a number.\n");
        scanf("%d",&r);
        square(r);
        return 0;
}
```

```c
//meng2.c
#include <stdio.h>
int square(int x)
{
        printf("The square=%d\n",x*x);
        return(x*x);
}
```

gcc的编译过程

- 多文件的编译
  - 编译方法1

    gcc -o meng meng1.c meng2.c
  - 编译方法2

    gcc -c meng1.c meng2.c

    gcc -o meng meng1.o meng2.o

gcc的编译过程

gcc的其它选项

选 项	说 明
-g	在目标代码中加入供调试程序gdb使用的附加信息
-v	显示gcc版本
-Wall	显示警告信息

- 例子

```
gcc -Wall -o test1 test1.c
gcc -g -o test1 test1.c
```

gcc的编译过程

优化程序选项

选 项	说 明
-O0	缺省情况，不优化
-O1(-O)	一级优化
-O2	进行比O1高一级的优化
-O3	产生更高级别的优化

gcc的编译过程

连接选项

选 项	说 明
-Ldir	将dir所指出的目录加到“函数库搜索列表”中
-lname	连接时，加载名字为name的函数库（实际的库名是libname）

说明

标准库文件一般存储在/lib和/usr/lib目录中

所有的库名都以lib开头

gcc的编译过程

- 库（Library）
  - 静态库(.a文件)

    在程序的链接阶段被复制到了程序中
  - 动态库/共享库 (.so文件)

    程序在运行时由系统动态加载到内存中供程序调用

gcc的编译过程

- 例4:

```c
//test.c
#include <stdio.h>

int main()
{
        int sum,cha;
        int a=8;
        int b=3;
        printf("a=%d\tb=%d\n", a, b);
        sum = add(a,b);
        cha = minus(a,b);
        printf("a+b=%d\n",sum);
        printf("a-b=%d\n",cha);
        return 0;
}
```

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

```c
//minus.c
int minus(int a, int b)
{
        int result;
        result = a - b;
        return result;
}
```

gcc的编译过程

- 静态库的生成
  - 将源文件编译成目标文件

    gcc -c add.c minus.c
  - 用ar命令把多个目标文件集合起来

    ar rcv libtest1.a add.o minus.o

gcc的编译过程

- 连接静态库
  - 方法1

    gcc -o test test.c libtest1.a
  - 方法2

    gcc -o test test.c -L. –ltest1

gcc的编译过程

- 动态库的生成
  - 将源文件编译成目标文件

    gcc -c add.c minus.c
  - 用-shared选项生成动态库

    gcc -shared -o libtest2.so add.o minus.o

gcc的编译过程

- 链接动态库
  - 方法一

    cp libtest2.so /usr/lib

    gcc -o test2 test2.c libtest2.so
  - 方法二

    cp libtest2.so /usr/lib

    gcc -o test2 test2.c -ltest2

gdb

gdb简介
gdb的基本命令
应用实例

gdb简介

- gdb（GNU Debugger）
  - 设置断点
  - 监视、修改变量
  - 单步执行
  - 查看堆栈

gdb简介

- 利用gdb调试的前提
  - 在程序编译过程中，使用-g选项

    gcc -g -o test test.c
  - 启动gdb

    gdb

    gdb filename
  - 说明

    filename是可执行文件，不是源代码文件

gdb的基本命令

- 文件操作
  - file命令：装入调试程序

    file filename
  - quit命令：退出gdb
  - shell命令：进入shell环境，执行shell命令

    shell

    shell date

    退出shell：exit

gdb的基本命令

- 显示源程序
  - list命令：列出调试程序的源代码

格式	说 明
list	没有参数，显示当前行开始10行
list n	显示从n行开始10行
list start,end	显示从start到end行的代码

gdb的基本命令

- 显示源程序
  - list命令：列出调试程序的源代码
  - 例子

```
list -                  //显示当前行之前的10行
list +                  //显示当前行之后的10行
list 2,10               //显示2到10行
list main               //显示函数main
```

gdb的基本命令

- 显示源程序
  - show listsize命令：查看显示的行数
  - set listsize命令：设置显示的行数
  - 例子

    show listsize

    set listsize 5

gdb的基本命令

- 控制程序的执行
  - break命令：设置断点
  - 例子

```
break 10                    //在第10行设置断点
break main                  //在函数入口处设置断点
break main if i==10
```

gdb的基本命令

- 控制程序的执行
  - info breakpoints（info break）：查看断点
  - 例子

    info break

    info break 1
  - delete：删除断点
  - 例子

    delete　　//清除所有断点

    delete 1

gdb的基本命令

- 程序的运行
  - run命令：运行程序，执行到断点处，或者直到程序结束
  - next命令：单步执行，不进入函数内部
  - step命令：单步执行，进入函数内部
  - continue命令：程序从当前位置开始，执行到断点处，或者直到程序结束
  - kill命令：中止正在调试的程序

gdb的基本命令

- 查看运行时数据
  - print命令：显示变量/表达式的值
  - 例子

```
print i
print i*j
print array[3]@5                    //显示数组的值
```

gdb的基本命令

- 查看运行时数据
  - set命令：修改变量的值
  - 例子

    set variable i=9

    print i=9

gdb的基本命令

- 查看运行时数据
  - display命令：预先设置一些要显示的表达式
  - info display命令：显示当前设置的表达式的清单
  - delete display命令：取消对设置的表达式的自动显示功能

gdb的基本命令

- 显示函数调用堆栈的信息

格式	说 明
backtrace（bt） where	显示函数调用的层次关系
up [n]	向上移动n层栈桢
down[n]	向上移动n层栈桢

make

- 功能
  - 对大中型软件项目进行编译、链接、清除中间文件
  - 提供多种默认规则

make

make的工作机制

makefile的变量

隐式规则

make的工作机制

- 通过makefile文件来描述源程序之间的依赖关系并自动进行编译工作

make的工作机制

- 例子
  - 有三个源文件：program.c、pro1.c、pro2.c。program.c文件包含有自定义的头文件lib.h，要求生成可执行文件program

make的工作机制

```
//lib.h
void pro1(char *);
void pro2(char *);
```

```
//program.c
#include "lib.h"
int main()
{
        pro1("this is pro1.");
        pro2("this is pro2.");
        return 0;
}
```

```
//pro1.c
#include <stdio.h>
void pro1(char *arg)
{
        printf("hello:%s\n",arg);
}
```

```
//pro2.c
#include <stdio.h>
void pro2(char *arg)
{
        printf("welcome to:%s\n",arg);
}
```

make的工作机制

- 例子
  - 编译方式1

    gcc -o program program.c pro1.c pro2.c
  - 编译方式2

    gcc -c program.c pro1.c pro2.c

    gcc -o program program.o pro1.o pro2.o
  - 编译方式3

    编写makefile文件

make的工作机制

- makefile
  - 定义了多个文件之间的依赖关系，编译的规则和步骤
  - make根据makefile中的内容，完成整个软件项目的编译工作

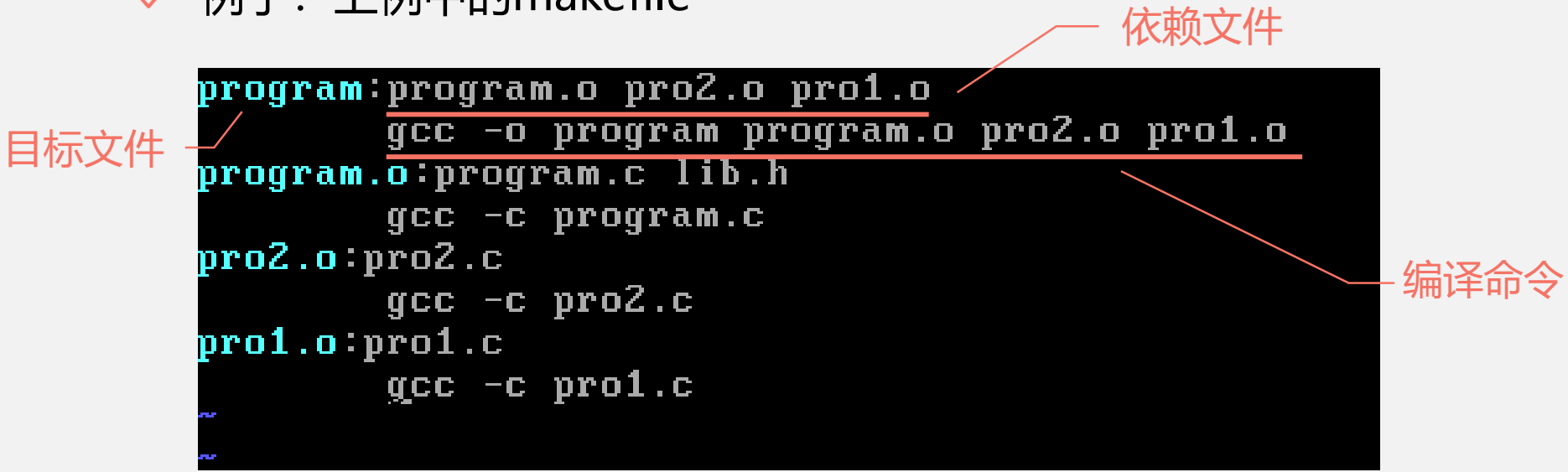
make的工作机制

- makefile的规则
  - 定义要创建的目标文件
  - 指出要生成目标文件的依赖文件
  - 表明通过依赖文件创建目标文件的编译命令
- 规则的格式

```
目标文件：依赖文件列表
<tab>编译命令
```

make的工作机制

- makefile
  - 例子：上例中的makefile



make的工作机制

- makefile的默认文件名
  - GNUmakefile
  - makefile
  - Makefile
- 说明
  - 如果要使用其它名称的文件作为makefile，需要在make时使用选项-f

make的工作机制

- make命令
  - 功能

    根据makefile文件完成程序的编译连接，生成可执行文件
  - 格式

    make [选项] [目标文件]

make的工作机制

- make命令
  - 例子

```
make                    //默认生成第一个目标文件
make prog1.o            //生成指定目标文件prog1.o
make -f makefile1       //读取指定的makefile文件
make clean              //用于清除编译过程中产生的二进制文件
```

  - 说明

clean目标不依赖于任何条件，并且也不会生成clean这个文件

make的工作机制

- 说明
  - make可识别makefile中的被修改文件，并在再次编译的时候只编译这些文件，从而提高编译效率

make的工作机制

- 原理
  - make命令会读取makefile文件的内容，比较目标文件和依赖文件的日期和时间，当依赖文件的日期比目标文件得时间新的时候，则根据命令重新生成目标文件

make的工作机制

- 依赖关系图
  - 体现各个文件之间的依赖关系
  - 生成一个目标文件可以有不同的依赖关系
  - 合理的构造依赖关系图，可以提高make的执行效率

make的工作机制

- 依赖关系图
  - 例子：一个程序包括以下内容

    三个C语言源文件：x.c y.c z.c

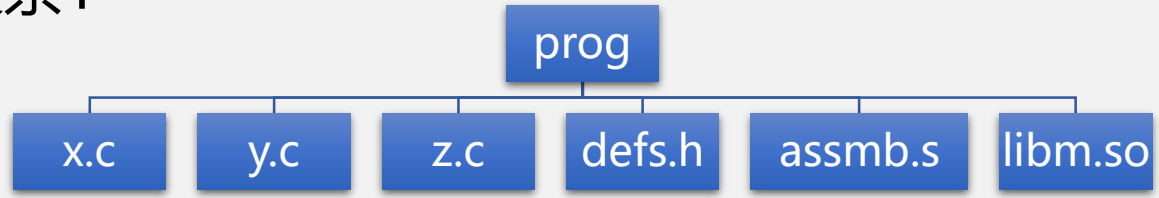
    一个头文件：defs.h

    汇编语言文件：assmb.s

    数学运算函数库：/home/mqc/lib/libm.so

make的工作机制

- 依赖关系1



- makefile文件

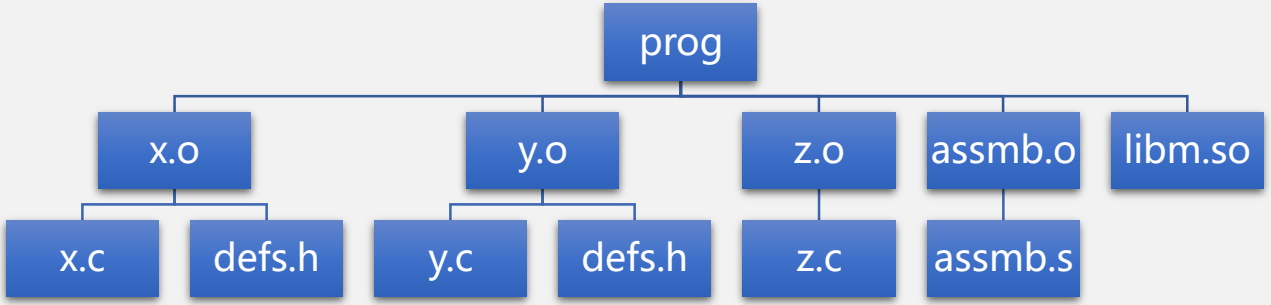
```
prog:x.c y.c z.c assmb.s
    gcc x.c y.c z.c assmb.s -L /home/mqc/libm -lm -o prog
```

  - 问题

    无论哪个文件被修改，所有文件都必须重新编译

make的工作机制

- 依赖关系2



- makefile文件

```
prog:x.o y.o z.o assmb.o
	gcc x.o y.o z.o assmb.o -L /home/mqc/libm -lm -o prog
x.o: x.c defs.h
	gcc -c x.c
y.o: y.c defs.h
	gcc -c y.c
z.o: z.c
	gcc -c z.c
assmb.o: assmb.s
	as -o assmb.o assmb.s
```

makefile的变量

- 变量的功能
  - 简化makefile

    贮存依赖文件名列表

    贮存编译器标志参数

makefile的变量

- 变量的命名规则
  - 包含数字、字符、下划线（可以是数字开头）
  - 不能含有“： # =”和空字符（空格、回车等）
  - 区分大小写

使用变量

- 定义变量
  - 格式

    变量名 = 字符串
- 引用变量
  - $(变量名)　或者　${变量名}
  - 例子：makefile1

```
object= program.o pro1.o pro2.o
program: $(object)
    gcc -o program $(object)
```

使用变量

内部变量

变量名	含 义
$@	当前目标文件的名称
$?	比当前目标文件新的依赖文件列表（当前目标文件所依赖的文件）
$<	规则中第一个依赖文件
$^	所有依赖文件

例子

```
object= program.o pro1.o pro2.o
program: $(object)
        gcc -o $@ $^
```

使用变量

预定义变量

类别	预定义变量	说明
C编译命令	CC	c编译器的名称，默认为cc
	CFLAGS	c编译器的选项，无默认值
C++编译命令	CXX	c++编译程序，默认为g++
	CXXFLAGS	c++编译程序的选项，无默认值
汇编命令	AS	汇编程序，默认是as
	ASFLAGS	汇编程序的选项，无默认值

使用变量

- 预定义变量
  - 例子：makefile2

```
object = program.o pro1.o pro2.o
CC = gcc
CFLAGS = -g
program: $(object)
	${CC} -o program $(object) $(CFLAGS)
```

隐式规则

- 含义
  - 不需要在makefile中写出来的规则
- 功能
  - 可以简化makefile文件的内容

隐式规则

- 常用的隐式规则
  - C语言程序

    .o文件会自动找到对应的.c文件，用cc命令进行编译
  - C++程序

    .o文件会自动找到对应的.cc文件，用g++进行编译
  - 汇编程序

    .o文件会自动找到对应的.s文件，并且用as命令进行汇编

隐式规则

- 例子：进一步简化。makefile4

```
object=program.o pro1.o pro2.o
program: $(object)
	$(CC) -o program $(object)
clean:
	rm *.o
```

练习

- 某个程序由下列源文件组成，并且其依赖关系图如图所示：
  - 写出makefile文件
  - 使用隐含规则对makefile文件进行简化
  - 写出编译命令