3 编译与调试

3.1 gcc/g++编译器

- 当我们进行编译的时候，要使用一系列的工具，我们称之为工具链。SDK就是编译工具链的简写，我们所使用的是gcc系列编译工具链
- 使用-v参数来查看gcc的版本，从而确定某些语法特性是否可用，比如是否允许使用时声明

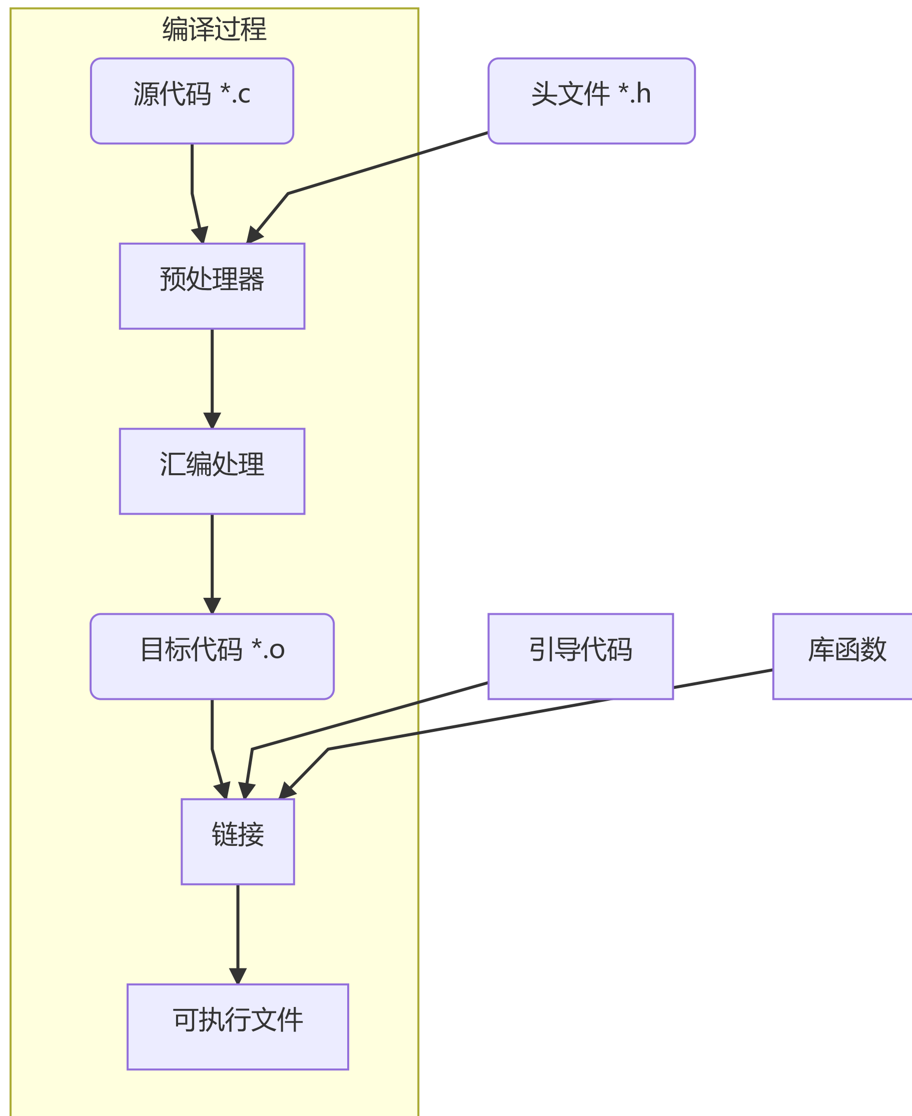
```
$gcc -v
```

- 对于.c格式的C文件，可以采用gcc或g++编译
- 对于 .cc、.cpp格式的C++文件，应该采用g++进行编译
- 常用的选项：

选项	效果
-c	表示编译源文件
-o	表示输出目标文件
-g	表示在目标文件中产生调试信息，用于gdb调试
-D	<宏定义> 编译时将宏定义传入进去
-Wall	打开所有类型的警告。

3.1.1 gcc编译过程

- 使用gcc编译程序的过程是预处理-->编译-->汇编-->链接。期间所使用的工具依次是预处理器，编译器，汇编器as，链接器ld



- 编译过程的几个阶段具体如下:

```
    (1) 预处理：预处理器将对源文件中的宏进行展开
    (2) 编译：gcc将c文件编译成汇编文件
    (3) 汇编：as将汇编文件编译成机器码
    (4) 链接：ld将目标文件和外部符号进行连接，得到一个可执行二进制文件
```

- 下面以一个很简单的test.c文件来探讨这个过程

```
#include <stdio.h>
#define NUM 2+3
int main()
{
    int i=NUM*NUM;
    printf("i = %d\n", i);
}
```

```
如果直接使用
$ gcc test.c
那么本目录会出现一个a.out的可执行文件。输入命令执行文件
$ ./a.out
提问：结果是多少？
```

1. 预处理：

```
$gcc -E test.c -o test.i
```

- 使用cat查看test.i的内容如下：

```
    int x=2+3*2+3;
    我们可以看到，文件中宏定义NUMBER出现的位置被2+3替换掉了，其它的内容保持不变
```

> - 工作中经常通过test.i，来查看一些标准库定义的数据结构

```
$grep -n "typedef.*FILE" test.i
$grep -n "struct.*_IO_FILE" test.1
这样就可以得到FILE结构体的具体定义
```

2. 编译：

```
$gcc -S test.i -o test.s
```

- 使用cat查看test.s的内容为如下

```
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    #将函数压栈
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    #将rsp寄存器里面的内容减去16
    movl    $11, -4(%rbp)
    #将11赋值给内存地址为（rbp-4）的位置
    movl    -4(%rbp), %eax
    #内存地址为（rbp-4）的位置的内容赋值给eax寄存器
    movl    %eax, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    #调用printf
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

- 计算机的CPU是只能识别机器码，机器码是0和1组成序列，用来表示高低电平之间的变化信息。为了方便人类的阅读和理解，出现了汇编语言。汇编语言是机器码的助记符，每一条语句都和机器码是一一对应的
- 通常的PC处理器是x86架构（比如Intel和AMD公司出品的PC处理器）的，具体语言的含义可以查阅CPU厂商的帮助手册。rbp是基准帧指针，rsp是栈顶指针，其余的各种变量是通过“栈顶+偏移”的方式来确定它的数值。在编译以后，所有的变量名都会被等价的地址（偏移）所替代

```
提问：数组名，变量名在程序运行当中，放置在哪个区域；循环的过程怎么用汇编来描述；调用函数的过程呢？
试一试，检查下列程序的汇编代码，注意和内存模型对应
```

```
void print(int j)
{
    printf("I am print j = %d\n", j);
}
int main()
{
    int *p;
    int arr[3] = {1,2,3};
    int j = 10;
    arr[2] = 4;
    p = arr;
    print(j);
    for(int i=0; i<5; i++)
    {
        printf("Hello world!\n");
    }
    return 0;
}
```

3.汇编：

```
$ as test.s -o test.o
```

- 利用as命令（汇编器）可以将汇编文件编译成机器码，得到输出文件为test.o。test.o中为目标机器上的二进制文件。使用nm命令可以查看文件中的符号表：

```
                    U _GLOBAL_OFFSET_TABLE_
0000000000000024    T main
0000000000000000    T print
                    U printf
                    U puts
                    U __stack_chk_fail
```

4. 链接：

- 得到.o文件以后，如果直接执行这个文件，就会提示无法运行，这是因为前面的符号表当中还有很多U的部分，也就是地址未确定的部分，这种文件是无法执行的。在完成链接以后，各个部分的代码的地址都确定以后，文件才能执行。使用gcc命令就可以完成链接

```
$ gcc test.o -o test
使用ld命令可以调用静态链接器
$ ld test.o [其他系统库文件] -o test
```

- gcc编译过程的参数汇总：

选项	含义
-c	只编译不链接，如果不指定输出文件，将自动生成后缀为.o的目标文件
-S	只编译不汇编，生成汇编代码
-E	只进行预处理
-o file	将file文件作为输出文件
-v	打印编译器的版本信息
-I dir	在头文件的搜索路径列表当中添加dir目录

3.1.2 其他编译选项

- 在gcc命令之后可以添加一些参数来实现不同的需求：

> -I 目录名：如果代码里面包含的头文件不是位于代码所在的目录之下，那么可以在编译时指定头文件所在的目录，或者在头文件处加入路径（这种方法不常使用，一旦程序的代码文件发生位置调整，代码内容也要随即调整，牵扯较大）

> -D 宏：通常测试版本会多一些测试语句，例如调试、报错信息打印等等。程序员可以采用测试开关的形式来打开或者关闭测试语句

```
//示例1 #ifdef/#endif
#include<stdio.h>
int main()
{
#ifdef DEBUG
    printf("This is main!\n");//如果有DEBUG的定义，那么久执行这个语句
#endif
}
//示例2 #ifdef/#else/#endif
#include <stdio.h>
int main()
{
#ifdef ON     //表示如果定义了ON，即命令行参数传了ON，就执行下面的输出
    printf("cjy is defined!\n");
#else
    printf("cjy is not defined!\n");
#endif
    printf("main exit\n");
}
```

```
$ gcc -E project2.c -o project2.i -D ON  //条件编译，用-D传递，如果没有传cjy则执行#else
$ gcc -S project2.i -o project2.s
$ gcc -o project2 project2.s

$ gcc -o project2 project2.c -D ON //等价于上面命令
```

> 在编译的时候，如果不添加-D选项，那么编译出来的文件就不会执行测试语句，在例子1当中就不会打印任何信息；但是如果编译时添加-D DEBUG，那么控制台就会出现This is main！字样

> 关于-Wall参数：通常来说，在书写程序的时候，一些不规范的写法是不会违背C语言的语法规则的，但是却会很有可能在运行的时候带来意想不到的问题。解决这个问题的方法自然就是添加警告信息。

```
int main()
{
    int a;//虽然定义了变量，但是没有使用它
}
```

- 编译时添加-Wall以后，会出现警告warning: unused variable 'a' [-Wunused-variable]
- 另一个例子：

```
#include <stdio.h>
int main()
{
    long long temp = 1;
    printf("This is a bad code!\n");
    return 0;
}
```

- -ansi : 生成标准语法（ANSI C标准）所要求的警告信息（并不列出所有警告）

```
$ gcc -ansi warning.c -o warning
```

```
warning.c: 在函数"main"中:
warning.c:7 警告: 在无返回值的函数中，"return"带返回值
warning.c:4 警告: "main"的返回类型不是"int"
可以看出，该选项并没有发现"long long"这个无效数据类型的错误
```

- -pedantic : 列出ANSI C标准的全部警告信息。

```
$ gcc -pedantic warning.c -o warning
```

```
warning.c: 在函数"main"中:
warning.c:5 警告: ISO C89不支持"long long"
warning.c:7 警告: 在无返回值的函数中，"return"带返回值
warning.c:4 警告: "main"的返回类型不是"int"
```

- -Wall : 列出所有的警告信息（常用）

```
$ gcc -Wall warning.c -o warning
```

```
warning.c:4 警告: "main"的返回类型不是"int"
warning.c: 在函数"main"中:
warning.c:7 警告: 在无返回值的函数中，"return"带返回值
warning.c:5 警告: 未使用的变量"tmp"
```

```
$ gcc -Werror warning.c -o warming
```

- 通常用的是-Wall显示所有有用的报警信息。
- 常用的警告信息汇总

选项	含义
-ansi	支持符合ANSI标准的C程序

3.1.3 编译过程的文件和生成方法

- 编译过程中各个文件的后缀使用约定

后缀名	所对应的语言
.c	C原始程序
.s/.S	汇编语言原始程序
.C/.cc/.cxx	C++原始程序
.m	Objective-C原始程序
.h	预处理文件（头文件）
.o	目标文件
.i	已经过预处理的C原始程序
.ii	已经过预处理的C++原始程序
.a/.so	编译后的库文件

- 预处理阶段：对包含的头文件（#include）和宏定义（#define、#ifdef等）进行处理

```
$ gcc -E hello.c -o hello.i   //-o表示输出为指定文件类型 -E将源文件.c转换为.i
```

- 编译阶段：检查代码规范性、语法错误等，在检查无误后把代码翻译成汇编语言

```
$ gcc -S hello.i -o hello.s   //-S将已预处理的C原始程序.i转换为.s
```

- 链接阶段：将.s的文件以及库文件整合起来链接为可执行程序

```
$ gcc -o hello hello.s   //最后将汇编语言原始程序(*.s)和一些库函数整合成可执行程序
```

- 例子：

```
#include <stdio.h>
#define MAX 100
#define max(a,b) ((a)>(b)?(a):(b))   //宏定义，执行-E之后被替换
main()
{
    printf("MAX=%d\n",MAX);
    printf("max(3,4)=%d\n",max(3,4));
}
```

```
//方法一：
$ gcc -E project1.c -o project1.i       //预编译，生成已预编译过的C原始程序*.i
$ gcc -S project1.i -o project1.s       //编译，生成汇编语言原始程序*.s
$ gcc -o project1.exe  project1.s       //链接，生成可执行程序
//方法二：
$ gcc -c project1.c -o project1.o       //编译
$ gcc -o project1.exe project1.o        //链接
//方法三：
$ gcc -o project1.exe project1.c    //编译并链接
```

3.1.4 静态库和动态库

选项	含义
-static	进行静态编译，即链接静态库，禁止使用动态库
-shared	1.可以生成动态库文件
-shared	2.进行动态编译，尽可能地链接动态库，只有没有动态库时才会链接同名的静态库
-L dir	在库文件的搜索路径列表中添加dir目录
-l[name]	链接称为libname.a或者libname.so的库文件。若两个库都存在，则根据编译方式来进行链接
-fpic	生成位置无关的目标代码(Position Independent Code)

- 静态库是目标文件.a的归档文件（格式为libname.a）。如果在编译某个程序时链接静态库，则链接器将会搜索静态库并直接拷贝到该程序的可执行二进制文件到当前文件中；
- 动态库（格式为libname.so[.主版本号.次版本号.发行号]）。在程序编译时并不会被链接到目标代码中，而是在程序运行时才被载入。
- 创建静态库

```
$ gcc  -c  add.c    //编译add.c源文件生成add.o目标文件
$ ar  crsv  libadd.a  add.o  //对目标文件*.o进行归档，生成lib*.a，Linux动态库的命名规范
```

- 将库文件libadd.a拷贝到/lib或者/usr/lib下（系统默认搜索库路径）

```
$ gcc  -o  main  main.c   -ladd  （-ladd表示链接库文件libadd.a/.so ）
$./main
```

- 创建动态库

```
$ gcc  -fPIC  -Wall  -c  add.c （这里可以省略-o 目标文件）
$ gcc  -shared  -o  libadd.so  add.o
$ gcc  -o  main  main.c  -ladd
```

- 在运行main前，需要注册动态库的路径。将库文件拷贝到/lib或者/usr/lib下（系统默认搜索库路径）。

```
$ cp libadd.so /lib    //通常采用的方法，cp  lib*.so  /lib
$ ./main
```

- 静态库与动态库的比较：

> - 动态库只在执行时才被链接使用，不是直接编译为可执行文件，并且一个动态库可以被多个程序使用，故可称为共享库
> - 静态库将会整合到程序中，在程序执行时不用加载静态库。
> - 因此，静态库会使你的程序臃肿并且难以升级，但比较容易部署。而动态库会使你的程序轻便易于升级但难以部署

- 符号链接生成

```
$ ln -s 源文件 软链接名
```

- 查看库的依赖的关系

```
$ which ls
检查ls程序的位置，发现在/bin/ls
$ ldd /bin/ls
查看ls程序的依赖关系（ldd只能检查动态依赖）
```

- 升级版本

```
    首先，生成新的库文件
    然后，将原来的软链接指向新的库文件
    最后，删除旧的库文件
```

- 不要混用动态库和静态库（测试删除的时候也要小心，不要影响到其他库）

3.1.5 gcc优化选项

- gcc对代码进行优化通过选项"-On"来控制优化级别（n是整数）。不同的优化级别对应不同的优化处理工作。如使用优化选项"-O1"主要进行线程跳转和延迟退栈两种优化。使用优化选项"-O2"除了完成所有"-O1"级别的优化之外，还要进行一些额外的调整工作，如处理其指令调度等。选项"-O3"则还包括循环展开或其他一些与处理器特性相关的优化工作
- 虽然优化选项可以加速代码的运行速度，但对于调试而言将是一个很大的挑战。因为代码在经过优化之后，原先在源程序中声明和使用的变量很可能不再使用，控制流也可能会突然跳转到意外的地方，循环语句也有可能因为循环展开而变得到处都有，所有这些对调试来讲都是不好的。所以在调试的时候最好不要使用任何的优化选项，只有当程序在最终发行的时候才考虑对其进行优化。通常用的是-O2

3.2 程序调试gdb

3.2.1 gdb常用命令

- Linux 包含了一个叫gdb的调试程序。gdb可以用来调试C和C++ 程序。在程序编译时用 -g 选项可打开调试选项
- 常见的调试程序的步骤如下：

```
$ gcc -o filename  -Wall filename.c -g //编译一定要加-g
```

```
gdb filename    //进入调试
l               //显示代码(list)
b 4             //在第四行设置断点，相当于Windows的F9(break)
r               //运行(run)
n               //下一步不进入函数，相当于Windows的F10(next)
s               //表示单步进入函数， 相当于Windows的F11(step)
p I             //打印变量I，相当于Windows的Watch窗口  (print)
c               //运行到最后，相当于Windows的F5(continue)
q               //退出，相当于Windows的Shift+F5    (quit)
```

3.2.2 gdb调试命令列表

- 按 Tab 键补齐命令,用光标键上下翻动历史命令。用help up看帮助

命令格式	含义
set args 运行时的参数	指定运行时的参数
show args	查看设置好的参数
info b	查看断点信息
break [文件名:] 行号或者函数名 [if <条件表达式>]	设置断点 示例：b 23 if i==2 当i==2时，在23行触发断点
tbreak [文件名:] 行号或者函数名 [if <条件表达式>]	设置临时断点，触发断点以后会被自动删除
delete [断点号]	删除指定的断点（如果没有断点号就是所有断点）
disable [断点号]	停止指定的断点（如果没有断点号就是所有断点）
enable [断点号]	激活指定的断点
condition [断点号] <条件表达式>	修改对应断点的条件
ignore [断点号] <忽略次数>	忽略断点num次
step	单步调试，进入函数调用
next	单步调试，不进入函数调用
finish	跳出当前函数
continue	继续执行，直到遇到下个断点
list [文件名:] 行号或者函数名	显示程序文本10行
print 表达式或变量	监视表达式或者变量的值
x <n/f/u>	查看内存内容 n表示内存的长度 f表示内存的格式 u表示内存的单位
display 表达式	单步调试的时候，设置自动显示的表达式内容
backtrace	查看调用堆栈

3.2.3 gdb调试段错误

- 当程序运行的时候出现了segmentation fault（即段错误）之类的错误以后，使用gdb可以进行调试
- 首先使用ulimit -a 来查看当前系统的各项属性的大小限制

```
$ ulimit -a
```

- 再使用ulimit -c unlimited 设置core file size为不限制大小

```
$ ulimit -c unlimited
```

- 设置完毕后，可以通过ulimit -a来检查是否成功设置

```
$ ulimit -a
```

- 再次运行程序，会产生core文件，通过gdb 可执行程序 core文件，进行调试。直接通过bt可以看到程序段错误时的现场

```
$ gdb ./test2 core
```

3.3 Makefile工程项目管理器

3.3.1 Makefile简述

- 一个工程中的源文件不计数，其按类型、功能、模块分别放在若干个目录中。由于文件非常多，分布比较广，编译这些源文件的命令非常的复杂，此外，为了减少不必要的编译时间，工程中主要采用增量编译的模式，这也对编译命令脚本的设计带来了风险。Makefile是一种按照增量编译模式设计的命令脚本。它建立了各个文件（可执行程序-目标文件-库文件-源代码文件等等）之间的依赖关系，根据依赖关系和修改时间，来决定哪些命令需要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，因为Makefile就像一个Shell脚本一样，其中也可以执行操作系统的命令
- 使用Makefile的步骤非常简单，先建立一个名为makefile或者是Makefile的文件，然后在里面写入符合语法规则的编译命令，完成以后只需要在文件所在目录使用make命令就能运行编译命令

```
$ make
```

3.3.2 规则、目标文件和依赖文件

- Makefile文件的书写逻辑是这样的：首先，先确定需要生成的目标文件，然后，根据目标文件确定它所需的依赖文件，此后，递归地找到依赖文件的依赖文件，直到依赖文件是没有子依赖文件（例如，.c文件，.h文件等等）
- 以上从目标文件来找到依赖文件的就是makefile当中的**规则**
- 表述目标文件和依赖文件的规则需要采用如下的语法结构

```
[target]:[prerequisites]
<tab>[command]
```

- 下面是一个简单的Makefile文件。可以看出，初始的目标文件是main，首先需要得到依赖文件main.o 和func.o，依赖文件又分别依赖于.c的代码文件，然后利用gcc -c命令得到.o的依赖文件，最后再执行gcc -o main main.o func.o得到main的可执行文件

```
main:main.o func.o
    gcc -o main main.o func.o
main.o:main.c
    gcc -c main.c
func.o:func.c
    gcc -c func.c
```

- Makefile会自动根据文件的修改时间来判断是否执行指令。如果目标比所有的依赖文件都要“新”，那么就不会执行有关这个目标的所有指令，这个规则对于依赖文件也生效，如果修改了某个原始代码文件，make命令只会根据修改时间，来调整有影响文件

3.3.3 伪目标

- 有些时候，使用make时并不希望得到最开始的目标文件，而是中间的目标文件。在make命令以后添加目标文件的名字就能完成需求

```
$ make [target]
```

- 例如使用make main.o可以只生成main.o这个目标文件，而不会执行前面的命令
- 利用上述特点，可以专门设置一些伪目标（.PHONY），伪目标并不是生成程序所必须的可执行文件或者依赖文件，它们更加类似于实现其他功能的命令，例如清理二进制文件，重新生成代码等等

```
.PHONY:clean rebuild
rebuild:clean main
clean:
    rm -rf main.o file.o main
```

- 伪目标设计的主要是为了避免中间依赖文件和clean、rebuild重名的情况（这种情况，make命令会认为clean已经存在，就不再需要修改的情况），执行伪目标的用法和一般目标一样

```
$ make clean
$ make rebuild
```

3.3.4 变量

- Makefile可以定义变量，在调用的时候，需要使用$()来引用变量（实际上就是字符串替换）

```
out = main #out代表了main，在运行的时候会进行字符串替代
$(out):main.o func.o
    gcc -o $(out) main.o func.o
```

- 因为 = 定义变量会在执行的时候出现字符串替代，所以出现递归定义的时候，会进行递归展开。但是有些情况，我们不希望递归展开，只希望进行一次字符串替换，这种情况可以采用 := 来定义变量，这也是工作当中的主流用法

```
out := main #out代表了main，在定义完成的时候会进行字符串替代
$(out):main.o func.o
    gcc -o $(out) main.o func.o
```

- = 和 := 的区别可以从下面的例子当中区别，两次执行的结果会有区别

```
##case 1 =
#out = hello
#rout = $(out)
#out = world
#$(rout):
#   @echo $(rout)
#case 2 :=
out := hello
rout := $(out)
out := world
$(rout):
    @echo $(rout)
```

- 除了自定义变量以外，还有预定义变量，自动变量和环境变量
- 预定义变量就是内部定义好的变量，这些变量的含义是固定的

变量名	功能	默认含义
AR	打包库文件	ar
AS	汇编程序	as
CC	C编译器	cc
CPP	C预编译器	$(CC) -E
CXX	C++编译器	g++
RM	删除	rm –f
ARFLAGS	库选项	无
ASFLAGS	汇编选项	无
CFLAGS	C编译器选项	无
CPPFLAGS	C预编译器选项	无
CXXFLAGS	C++编译器选项	无

- 自动变量就某些具有特殊含义的变量，它的含义和当前规则有关

变量	说明
$@	目标文件
$<	第一个依赖文件
$^	所有依赖文件，以空格分隔
$?	日期新于目标文件的所有相关文件列表，逗号分隔
$(@D)	目标文件的目录名部分
$(@F)	目标文件的文件名部分

- 因此makefile文件可以改写成如下：

```
OBJS:=main.o func.o
CC:=gcc
main:$(OBJS)
    $(CC) -o $@ $^
main.o:main.c
    $(CC) -c $^ -o $@
func.o:func.c
    $(CC) -c $^ -o $@
```

- 使用目录作为变量也是可行的，但是要注意变量的引用是简单的字符串替换，比如DIR = ./，那么(DIR)(OBJS) 就是 ./main.o func.o 。第二项的前面是不会添加目录的字符串的

3.3.5 通配符和模式匹配

- 因为makefile规则的命令部分是采用bash命令的，所以在这里就可以使用bash的规则来应用通配符

```
clean:
    rm -rf *.o
```

- makefile也允许对目标文件名和依赖文件名进行类似正则表达式运算的模式匹配，主要使用的是%匹配符（%表示在依赖文件列表当中匹配任意字符），例如将上述例子改写成

```
OBJS:=main.o func.o
CC:=gcc
main:$(OBJS)
    $(CC) -o $@ $^
%.o:%.c #先在依赖文件列表当中匹配得到后缀为.o的文件，再根据.o文件的文件名找到同名的.c文件
# 这里如果使用*.c，那么就会在当前目录所有文件里面进行匹配
    $(CC) -c $^ -o $@
```

- %也可以在变量内部进行查找替换

```
SRCS = test.c test1.c
OBJECTS = $(SRCS:%.c=%.o)
```

3.3.6 内置函数

- 为了满足一些特殊的需求，在makefile里面也可以使用函数使用格式如下

```
$([function] [arguments])
```

- 使用wildcard函数可以使用通配符，找到所有满足通配符的文件名

```
srcfiles := $(wildcard src/*.c)
```

- 使用subst函数来实现文本替换

```
$(subst from,to,text)
```

- 使用patsubst函数来实现模式文本替换

```
$(patsubst pattern,replacement,text)
$(patsubst %.c,%.o,func.c main.c)
```

3.3.7 循环

```
LIST = one two three
all:
	for i in $(LIST); do echo $$i; done
#等价于
all:
	for i in one two three; do echo $i; done
```

3.3.8 杂项

- 有些时候makefile文件的名字不希望以makefile或者Makefile来命名，此时可以使用make命令的-f参数来指定makefile文件

```
$ make -f newMake
```

- makefile默认会打印执行的命令，在命令前方添加@符号可以取消打印

```
#case 1
out1:
	echo "case 1"
#case 2
out2:
	@echo "case 2"
#分别使用make out1和make out2会有不一样的显示结果
```

- 有的时候如果某一行太长，可以使用 \+换行 来分开显示这一行

3.3.9 实例

- 以下是一个同时编译多个目标文件的示例

```
SRCS = $(wildcard *.c)
OBJECTS = $(patsubst %.c,%.o $(SRCS))
TARGETS = $(SRCS:%.c=%)
all:$(TARGETS)
	@for i in $(TARGETS);do gcc -o $${i} $${i}.c;done
.PHONY:clean
clean:
	rm $(TARGETS)
```

- 提问：如何不使用循环来完成需求？

```
CC:=gcc
SRCS := $(wildcard *.c)
BINS := $(SRCS:%.c=%)
all: $(BINS)
%: %.c
	$(CC) $< -o $@
.PHONY:clean
clean:
	rm $(TARGETS)
```