编译工具链

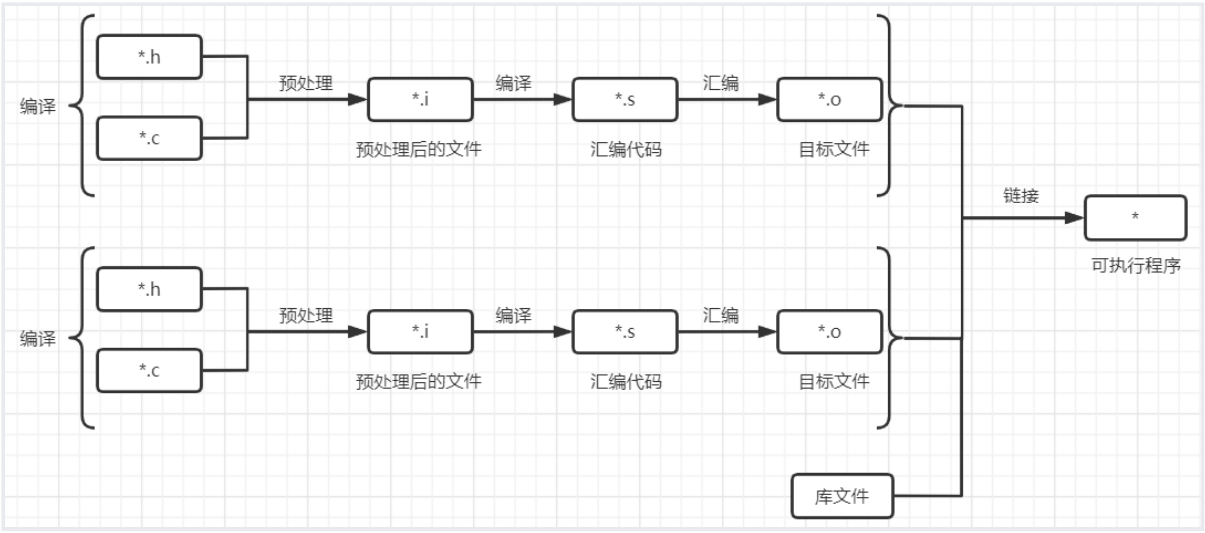
前面我们写程序的时候用的都是集成开发环境（IDE：Integrated Development Environment），集成开发环境可以极大地方便我们程序员编写程序，但是配置起来也相对麻烦。在 Linux 环境下，我们用的是编译工具链，又叫软件开发工具包(SDK：Software Development Kit)。Linux 环境下常见的编译工具链有：GCC 和 Clang，我们使用的是 GCC。

1 编译

gcc、g++分别是 gnu 下的 c 和 c++ 编译器。

```
$ sudo apt install gcc gdb          # 安装gcc和gdb
$ gcc -v                            # 查看gcc的版本
```

在讲如何编译之前，有必要给大家回顾一下生成可执行程序的整个过程：



对应的 gcc 命令如下：

```
gcc -E hello.c -o hello.i          # -E激活预处理，生成预处理后的文件
gcc -S hello.i -o hello.s          # -S激活预处理和编译，生成汇编代码
gcc -c hello.s -o hello.o          # -c激活预处理、编译和汇编，生成目标文件
gcc hello.o -o hello               # 执行所有阶段，生成可执行程序
```

其实没必要指定每一个步骤，我们常常会这样用：

```
gcc -c hello.c                     # 生成目标文件，gcc会根据文件名hello.c生成hello.o
gcc hello.o -o hello               # 生成可执行程序hello，这里我们需要指定可执行程序的名称，否则会默认生成a.out
```

甚至有时候，我们会一步到位：

```
gcc hello.c -o hello               # 编译链接，生成可执行程序hello
```

1.1 GCC其它选项

选项	含义
-Wall	生成所有警告信息
-O0,-O1,-O2,-O3	编译器的4个优化级别，-O0表示不优化，-O1为缺省值，-O3的优化级别最高
-g	指示编译器在编译的时候产生调试相关的信息。（调试程序必须加上这个选项）
-Dmacro	相当于在文件的开头加了#define macro
-Dmacro=value	相当于在文件的开头加了#define macro value
-Idir	对于 #include "file"，gcc/g++会先在当前目录查找你所指定的头文件，如果没有找到，他会到系统的 include 目录找。如果使用 -I 指定了目录，他会先在你所指定的目录查找，然后再按常规的顺序去找。对于#include <file>，gcc/g++会到 -I 指定的目录查找，查找不到，再到系统的 include 目录中查找。

> 补充：可以通过 cpp -v 命令查看系统的 include 目录。

1.2 条件编译

所谓**条件编译**，就是在**预处理**阶段决定包含还是排除某些程序片段。主要涉及以下预处理指令：

```
1) #if [#elif] [#else] #endif
2) #ifdef [#elif] [#else] #endif
3) #ifndef [#elif] [#else] #endif
```

1. #if 指令的格式如下：

```
#if 常量表达式
...
#endif
```

当预处理器遇到 #if 指令时，会计算后面常量表达式的值。如果表达式的值为 0，则 #if 与 #endif 之间的代码会在预处理阶段删除；否则，#if 与 #endif 之间的代码会被保留，交由编译器处理。

#if 指令常用于调试程序，如下所示：

```
#define DEBUG 1
...
#if DEBUG
    printf("i = %d\n", i);
    printf("j = %d\n", j);
#endif
```

2. defined 是预处理器的一个运算符，它后面接标识符。如果标识符是一个定义过的宏则值为 1，否则值为 0。defined 运算符常和 #if 指令一起使用，比如：

```
#if defined(DEBUG)
...
#endif
```

仅当 DEBUG 被定义成宏时，#if 和 #endif 之间的代码会保留到程序中。defined 后面的括号不是必须的，因此可以写成这样：

```
#if defined DEBUG
```

defined 运算符仅检测 DEBUG 是否有被定义成宏，所以我们不需要给 DEBUG 赋值：

```
#define DEBUG
```

3. #ifdef 的格式如下：

```
#ifdef 标识符
...
#endif
```

当标识符有被定义成宏时，保留 #ifdef 与 #endif 之间的代码；否则，在预处理阶段删除 #ifdef 与 #endif 之间的代码。等价于：

```
#if defined(标识符)
...
#endif
```

4. #ifndef 的格式如下：

```
#ifndef 标识符
...
#endif
```

它的作用恰恰与 #ifdef 相反：当标识符没有被定义成宏时，保留 #ifndef 与 #endif 之间的代码。

1.2.1 条件编译的作用

条件编译对于调试是非常方便的，但它的作用不仅限于此。下面是其它一些常见的应用：

编写可移植的程序

下面的例子会根据 WIN32、MAC_OS 或 LINUX 是否被定义为宏，而将对应的代码包含到程序中：

```
#if defined(WIN32)
...
#elif defined(MAC_OS)
...
#elif defined(LINUX)
...
#endif
```

我们可以在程序的开头，定义这三个宏中的一个，从而选择一个特定的操作系统~

为宏提供默认定义

我们可以检测一个宏是否被定义了，如果没有，则提供一个默认的定义：

```
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 1024
#endif
```

避免头文件重复包含

多次包含同一个头文件，可能会导致编译错误(比如，头文件中包含类型的定义)。因此，我们应该避免重复包含头文件。使用 #ifndef 和 #define 可以轻松实现这一点：

```
#ifndef __WD_FOO_H
#define __WD_FOO_H

typedef struct {
    int id;
    char name[25];
    char gender;
    int chinese;
    int math;
    int english;
} Student;

#endif
```

临时屏蔽包含注释的代码

我们不能用 /*...*/ "注释掉" 已经包含 /*...*/注释的代码。但是我们可以用 #if 指令来实现：

```
#if 0
包含/*...*/注释的代码
#endif
```

注：这种屏蔽方式，我们称之为"条件屏蔽"。

2 调试

写程序难免会遇到 Bug，这时我们就需要 GDB 来对程序进行调试了。调试需要在编译的时候，加上一些调试相关的信息，也就是说，需要指定 -g 选项。如：

```
$ gcc hello.c -o hello -g
```

Q：在具体讲如何使用 GDB 调试程序之前，我们回顾一下 Visual Studio 为调试程序提供了哪些功能？GDB 也提供了类似的功能~

2.1 进入GDB调试界面

我们可以用下面两种方式启动调试：

```
$ gdb executable_name                    # 不设置任何命令行参数
$ gdb --args executable_name [arg]...
```

比如：

```
$ gdb foo
$ gdb --args foo arg1 arg2 arg3
```

当然，我们也可以先进入调试界面，然后再设置命令行参数，如下所示：

```
$ gdb foo
(gdb) set args arg1 arg2 arg3
```

2.2 调试程序

查看源代码

我们可以用 list/l 命令查看源代码：

```
格式：
    list/l [文件名:][行号|函数名]
```

常见用法：

```
(gdb) list                               # 下翻源代码
(gdb) list -                             # 上翻源代码
(gdb) list 20                            # 查看20行附近的源代码
(gdb) list main                          # 查看main函数附近的源代码
(gdb) list scanner.c:20                  # 查看scanner.c文件第20行附近的源代码
(gdb) list scanner.c:scanToken           # 查看scanner.c文件scanToken函数附近的源代码
```

设置断点

我们可以用 break/b 命令设置断点：

```
格式：
    break/b [文件名:][行号|函数名]
```

常见用法：

```
(gdb) break 20                           # 在第20行设置断点
(gdb) break main                         # 在main函数的开头设置断点
(gdb) break scanner.c:20                 # 在scanner.c文件的第20行设置断点
(gdb) break scanner.c:scanToken          # 在scanner.c文件的scanToken函数开头设置断点
```

查看断点

我们可以用 info break/i b 命令查看断点信息：

格式：

```
    info break/i b
```

常见用法：

```
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000555555554e1d in main at main.c:79
2       breakpoint     keep y   0x0000555555555a99 in scanToken at
scanner.c:282
...
```

其中 Num 为断点的编号，Enb(enable)表示断点是否有效，What表示断点在源代码的哪个位置。

删除断点

我们可以用 delete/d 命令删除断点：

格式：

```
    delete/d [n] -- 删除所有断点或n号断点
```

常见用法：

```
(gdb) delete 2          # 删除2号断点
(gdb) d                 # 删除所有断点
```

启动调试

我们可以用 run/r 命令启动调试：

```
(gdb) r
```

继续

continue/c 命令可以运行到逻辑上的下一个断点处：

```
(gdb) c
```

忽略断点n次

我们可以用 ignore 命令来指定忽略某个断点多少次，这在调试循环的时候非常有用：

格式：

```
    ignore N COUNT
```

常见用法：

```
(gdb) ignore 1 10           # 忽略1号断点10次
```

单步调试

step/s 命令可以用来进行单步调试，即遇到函数调用会进入函数。

```
(gdb) step
```

跳出函数

我们可以使用 finish 命令执行完整个函数：

```
(gdb) finish
```

逐过程

next/n 命令表示逐过程，也就是说遇到函数调用，它不会进入函数，而是把函数调用看作一条语句。

```
(gdb) n
```

监视

print/p 命令可以打印表达式的值：

```
格式：
    print/p EXP
```

如：

```
(gdb) print PI*r*r
```

print/p 命令还可以改变变量的值：

```
格式：
    print/p EXP=VAL
```

比如：

```
(gdb) print r=2.0
```

我们可以用 display 命令自动展示表达式的值：

```
格式：
    display EXP             # 自动展示EXP
    info display            # 显示所有自动展示的表达式信息
    undisplay [n]           # 删除所有或[n]号自动展示的表达式
```

常见用法：

```
(gdb) display r
(gdb) display PI*r*r
(gdb) info display
Auto-display expressions now in effect:
Num Enb Expression
1:   y  r
2:   y  PI*r*r
(gdb) undisplay 2
(gdb) undisplay
Delete all auto-display expressions? (y or n)
```

我们还可以通过命令查看参数和局部变量的值：

```
(gdb) info args              # 查看函数的参数
(gdb) info locals            # 查看函数所有局部变量的值
```

查看内存（了解）

我们可以用 x 命令查看内存的值(一般用得很少，了解即可)：

```
格式:
    x/nFU
    其中，n为一个整数，表示查看n个单元的内存
    F表示输出格式:
        常用的输出格式有:
            o(octal),
            x(hex),
            d(decimal),
            u(unsigned decimal),
            t(binary),
            f(float),
            c(char),
            ...
        默认输出格式为x(hex)。
    U表示内存单元:
        b(byte), h(halfword, 2 bytes), w(word, 4 bytes), g(giant, 8
bytes)
        默认单位为w(word)
```

常见用法：

```
(gdb) x/4dw arr
0x7fffffffe3a0: 0   1   2   3
(gdb) x/4xb &i
0x7fffffffe38c: 0x37    0x25    0x00    0x00        # 其中i=9527
```

退出GDB

quit/q 命令可以退出 GDB。

```
(gdb) q
```

2.3 调试coredump文件

通常情况下，程序异常终止时，会产生 Coredump 文件。Coredump 文件类似飞机上的"黑匣子"，它会保留程序"失事"瞬间的一些信息，通常包含寄存器的状态、栈调用情况等。Coredump 文件常用于辅助分析和 Debug。

查看系统是否允许生成Coredump文件

```
$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
...
$ ulimit -c unlimited            # 将core文件的大小临时设置为不受限制
```

设置Coredump文件的格式

```
$ sudo vim /etc/sysctl.conf
  kernel.core_pattern = %e_core_%s_%t           # %e:executable-name,
%s:signal, %t:time
$ sudo sysctl -p                                # 让配置生效
```

使用Coredump文件调试程序

```
// test.c
int div(int div_i, int div_j) {
    int a4, b4;
    char *c4;

    a4 = div_i + 3;
    b4 = div_j + 3;
    c4 = "div function";

    return (div_i / div_j);
}

int sub(int sub_i, int sub_j) {
    int a3, b3;
    char *c3;
```

```
    a3 = sub_i + 2;
    b3 = sub_j + 2;
    c3 = "sub function";
    div(a3, 0);         // Error: divided by 0!

    return (sub_i - sub_j);
}

int add(int add_i, int add_j) {
    int a2, b2;
    char *c2;

    a2 = add_i + 1;
    b2 = add_j + 1;
    c2 = "add function";
    sub(a2, b2);

    return (add_i + add_j);
}

int main(int argc, char *argv[]) {
    int a1, b1;
    char *c1;

    a1 = 1;
    b1 = 0;
    c1 = "main function";

    add(a1, b1);

    return 0;
}
```

```
$ gcc test.c -o test -g             # 生成可执行程序test
$ ./test                            # 运行test
$ gdb test test_core_8_1679196427   # 使用Coredump文件调试程序
...
[New LWP 5036]
Core was generated by `./test'.
Program terminated with signal SIGFPE, Arithmetic exception.
#0  0x0000564d0188a645 in div (div_i=4, div_j=0) at test.c:9
9       return (div_i / div_j);
```

```
(gdb) bt                            # backtrace,查看栈调用情况
#0  0x0000564d0188a645 in div (div_i=4, div_j=0) at test.c:9
#1  0x0000564d0188a684 in sub (sub_i=2, sub_j=1) at test.c:19
#2  0x0000564d0188a6c6 in add (add_i=1, add_j=0) at test.c:31
#3  0x0000564d0188a707 in main (argc=1, argv=0x7ffc5cf053d8) at
test.c:44
(gdb) frame 3                       # 查看#3栈帧的情况
#3  0x0000564d0188a707 in main (argc=1, argv=0x7ffc5cf053d8) at
test.c:44
44      add(a1, b1);
(gdb) info args                     # 查看参数的值
argc = 1
argv = 0x7ffc5cf053d8
(gdb) info locals                   # 查看局部变量的值
a1 = 1
b1 = 0
c1 = 0x564d0188a7bb "main function"
(gdb) info registers                # 查看寄存器的值
rax            0x4  4
rbx            0x0  0
rcx            0x564d0188a710   94888738203408
rdx            0x0  0
rsi            0x0  0
rdi            0x4  4
rbp            0x7ffc5cf052f0   0x7ffc5cf052f0
rsp            0x7ffc5cf052d0   0x7ffc5cf052d0
...
```

注：GDB 的功能非常强大，这里我们只是介绍了它的一些基本用法~

3 静态库与动态库(了解)

这节的主题是——让大家学会创建与使用静态库、动态库，知道静态库与动态库的区别，知道使用的时候如何选择。这里不深入介绍静态库、动态库的底层格式，内存布局等，有兴趣的同学，可以阅读《程序员的自我修养——链接、装载与库》这本书。

什么是库?

库是写好的现有的，成熟的，可以复用的代码（库有时候也被称为轮子，不要重复造轮子~)。现实中每个程序都要依赖很多基础的底层库，不可能所有的代码都从零开始编写，因此库的存在意义非同寻常。

本质上来说库是一种可执行代码的二进制形式，可以被操作系统载入内存执行（说得更通俗易懂点，就是目标文件的集合)。

库有两种：静态库和动态库。在类 Unix 系统上，静态库一般是以 .a 结尾，Windows 上一般是以 .lib 结尾；在类 Unix 系统上，动态库一般是以 .so 结尾，Windows 上一般是以 .dll 结尾。

静态库VS动态库

静态库和动态库的区别，类似于家用汽车和F1赛车之间的区别。

静态库之所以称为静态，是因为它会在链接阶段打包到可执行程序中。静态库有如下特点：

1. 静态库对函数的链接是在链接阶段完成的。
2. 程序在运行时，与静态库再无瓜葛。移植方便。
3. 浪费空间，每一个进程中都有静态库的一个副本。
4. 对程序的更新，部署，发布不友好(需要所有用户重新下载安装新的可执行程序)。

动态库之所以称为动态，是因为它在链接阶段并不会打包到可执行程序中，而是在程序运行的时候才加载的。动态库有如下特点：

1. 动态库对函数的链接是在运行时完成的。
2. 动态库可以在进程之间共享(所以，动态库又被称为共享库)。
3. 对程序的更新，部署，发布友好(因为，我们只要更新动态库就好了)。
4. 程序在运行时，依赖动态库。不方便移植。

生成静态库

假如我们写了一个很好用的算法库，里面包含了加、减、乘、除算法(lll¬ω¬)，如下所示：

```
// algs.h
#ifndef __WD_ALGS_H
                                        #define __WD_ALGS_H

int add(int a, int b);
int sub(int a, int b);
int mul(int a, int b);
int div(int a, int b);

#endif
```

```
// add.c
#include "algs.h"

int add(int a, int b) {
    return a + b;
}
```

```
// sub.c
#include "algs.h"

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

```c
// mul.c
#include "algs.h"

int mul(int a, int b) {
    return a * b;
}
```

```c
// div.c
#include "algs.h"

int div(int a, int b) {
    return a / b;
}
```

```
1. 生成目标文件
$ gcc -c add.c
$ gcc -c sub.c
$ gcc -c mul.c
$ gcc -c div.c
2. 把目标文件打包成静态库
$ ar crsv libalgs.a add.o sub.o mul.o div.o          # 静态库一般以.a结尾，
库的名字为algs
3. 将生成的静态库移动到/usr/lib目录下
$ mv libalgs.a /usr/lib
```

编写 main.c，引用 algs 库中的函数~

```c
#include <stdio.h>
#include "algs.h"

int main(void) {
    printf("add(7, 3)=%d\n", add(7,3));
    printf("sub(7, 3)=%d\n", sub(7,3));
    printf("mul(7, 3)=%d\n", mul(7,3));
    printf("div(7, 3)=%d\n", div(7,3));

    return 0;
}
```

```
$ gcc main.c -o main -lalgs                    # 将静态库链接到程序
$ ./main                                       # 执行程序
add(7, 3)=10
sub(7, 3)=4
mul(7, 3)=21
div(7, 3)=2
$ sudo rm /usr/lib/libalgs.a                   # 删除静态库
(在/usr/lib目录下删除内容，请额外小心!)
$ ./main                                       # 程序依然能够运行
add(7, 3)=10
sub(7, 3)=4
mul(7, 3)=21
div(7, 3)=2
```

生成动态库

```
1. 生成目标文件，需要加上-fpic选项
$ gcc -c add.c -fpic
$ gcc -c sub.c -fpic
$ gcc -c mul.c -fpic
$ gcc -c div.c -fpic
2. 把目标文件打包成动态库
$ gcc -shared add.o sub.o mul.o div.o -o libalgs.so
3. 将生成的动态库移动到/usr/lib目录下
$ mv libalgs.so /usr/lib
```

编写程序，引用 algs 库中的函数 (这里我们沿用上面的 main.c)。

```
$ gcc main.c -o main -lalgs                # 将动态库链接到程序
$ ./main                                   # 执行程序
add(7, 3)=10
sub(7, 3)=4
mul(7, 3)=21
div(7, 3)=2
$ sudo rm /usr/lib/libalgs.so              # 删除动态库
$ ./main                                   # 程序不再能运行
./main: error while loading shared libraries: libalgs.so: cannot open
shared object file: No such file or directory
```

使用动态库，更新是非常容易的。我们只需要更新动态库即可，而不需要重新生成可执行程序。我们将 add.c，sub.c，mul.c，div.c 修改如下：

```c
// add.c
#include <stdio.h>
#include "algs.h"

int add(int a, int b) {
    printf("I love xixi\n");
    return a + b;
}
```

```c
// sub.c
#include <stdio.h>
#include "algs.h"

int sub(int a, int b) {
    printf("I love xixi\n");
    return a - b;
}
```

```c
// mul.c
#include <stdio.h>
#include "algs.h"

int mul(int a, int b) {
    printf("I love xixi\n");
    return a * b;
}
```

```c
// div.c
#include <stdio.h>
#include "algs.h"

int div(int a, int b) {
    printf("I love xixi\n");
    return a / b;
}
```

```
1. 生成目标文件，需要加上-fpic选项
$ gcc -c add.c -fpic
$ gcc -c sub.c -fpic
$ gcc -c mul.c -fpic
$ gcc -c div.c -fpic
2. 把目标文件打包成动态库
$ gcc -shared add.o sub.o mul.o div.o -o libalgs.so.0.0.2
```

```
3. 将生成的动态库移动到/usr/lib目录下
$ sudo mv libalgs.so.0.0.2 /usr/lib
4. cd 到/usr/lib目录
$ cd /usr/lib
5. 创建符号链接，将其指向新的动态库                    # 这样就不用覆盖旧的库了，方便以后回退~
$ sudo ln -s libalgs.so.0.0.2 libalgs.so
6. cd回来
$ cd -
7. 执行程序
$ ./main
I love xixi
add(7, 3)=10
I love xixi
sub(7, 3)=4
I love xixi
mul(7, 3)=21
I love xixi
div(7, 3)=2
```

4 Makefile

什么是 Makefile？很多 Windows 程序员都不知道这个东西，因为 Windows 的集成开发环境帮你做了相关的工作，但要成为一个好的和专业的程序员，Makefile 还是很有必要学习的。

Makefile 定义了整个工程的编译规则。一个工程中的源文件不计其数，哪些文件需要先编译，哪些文件要后编译，哪些文件需要重新编译...，这些规则我们都可以在 Makefile 中定义。Makefile 带来的好处就是——“自动化编译”，一旦写好，只需要一个 make 命令，就可以构建整个工程，极大的提高了软件开发的效率。make 是一个解释执行 Makefile 文件的工具。

而且 Makefile 采用的是"增量编译"，也就是说，我们只编译那些更新过的和新增的源文件；那些没修改过的源文件，是不会重新编译的。这极大的节省了编译的时间，也节省了程序员摸鱼的时间(ノ□ヽ)...

> 注：相对来说，编译过程是很耗时的，编译 Linux 内核往往需要好几个小时。而链接过程则非常迅速。

接下来，我们就一起来学习如何书写 Makefile~

4.1 一个简单的示例

首先，我们用一个示例来说明 Makefile 的书写规则，以便给大家一个感性认识。

```
main: main.o add.o sub.o mul.o div.o
    gcc main.o add.o sub.o mul.o div.o -o main
main.o: main.c algs.h
    gcc -c main.c -Wall -g
add.o: add.c algs.h
    gcc -c add.c -Wall -g
sub.o: sub.c algs.h
    gcc -c sub.c -Wall -g
mul.o: mul.c algs.h
    gcc -c mul.c -Wall -g
div.o: div.c algs.h
    gcc -c div.c -Wall -g
```

写好 Makefile 之后，一个 make 命令就可以构建整个项目了。

```
$ make
```

接下来，我们详细解释一下上面的示例~

规则

Makefile 的核心就是规则，一个规则是由三部分组成的：目标(target)，依赖(prerequisites)以及命令(commands)。其格式如下：

```
target: prerequisites
    commands
# target: 即为要生成的目标。
# prerequisites: 生成目标所依赖的其它文件。
# commands: 一般情况下为生成该目标所需执行的命令(可以是任意的shell命令)
```

规则定义了文件之间的依赖关系。**说得更直白一点，make 其实就是一个管理文件之间依赖关系的工具**。

```
1. 如果target文件不存在，执行commands.
2. 如果prerequisites中有一个文件比target文件更新，也要执行commands.
```

4.2 make是如何工作的

一般情况下，我们只需输入 make 命令。

```
1. make会在当前目录下找名字叫"Makefile"或"makefile"的文件。                    # 推荐使用Makefile, Makefile更显眼
2. 如果找到，make会把文件中的第一个target，作为要执行的target。                  # 在上面的例子中，即为main
3. 如果main文件不存在，或是后面的.o文件比main更新，则会执行下面定义的命令。
4. 如果main依赖的.o文件也不存在，那么make会递归地去找以.o文件为目标的规则，然后根据那一个规则生成.o文件。
```

拓展：想一想你会用什么样的数据结构管理文件之间依赖关系？（有向无环图+拓扑排序）

这就是 make 的执行原理，make 会递归地去查找文件之间的依赖关系，直到最终生成要执行的目标。在查找的过程中，如果出现错误，比如最后依赖的文件不存在，那么 make 就会直接退出，并报错。

如果我们修改了某个文件，比如 add.c，然后重新执行 make 命令：

```
$ make
gcc -c add.c -Wall -g
gcc main.o add.o sub.o mul.o div.o -o main
```

从上面我们可以看到，我们只是重新编译了 add.c 文件，并重新链接生成可执行程序 main，这就是所谓的"增量编译"。

4.3 伪目标

make 只管理文件之间的依赖关系，如果目标不存在，则执行后面定义的命令。利用这个特性，我们可以定义一些伪目标：

```
main: main.o add.o sub.o mul.o div.o
    gcc main.o add.o sub.o mul.o div.o -o main
main.o: main.c algs.h
    gcc -c main.c -Wall -g
add.o: add.c algs.h
    gcc -c add.c -Wall -g
sub.o: sub.c algs.h
    gcc -c sub.c -Wall -g
```

```
mul.o: mul.c algs.h
    gcc -c mul.c -Wall -g
div.o: div.c algs.h
    gcc -c div.c -Wall -g

clean:
    rm -f main main.o add.o sub.o mul.o div.o
rebuild: clean main
```

我们可以这样执行目标 clean 和 rebuild。

```
$ make clean            # 清除可执行程序和所有的目标文件，make可以指定要执行的目标。
$ make rebuild          # 先清除可执行程序和所有的目标文件，然后再构建main。
```

但是这样写 Makefile，有一个弊端：如果存在名字为 clean 和 rebuild 的文件，那么 make clean 和 make rebuild 就不起作用了。将 clean 和 rebuild 添加到 .PHONY 的序列中，可以避免这种情况发生。

```
...
.PHONY: clean rebuild
clean:
    rm -f main main.o add.o sub.o mul.o div.o
rebuild: clean main
```

4.4 变量(了解)

在 Makefile 中也可以定义的变量，这对于编写通用的 Makefile 非常有帮助。Makefile 中的变量类似于 C/C++ 中的宏，代表一个文本字符串，在执行的时候会原模原样地展开在所使用的地方。与 C/C++ 的宏不同的是，我们可以修改 Makefile 中定义的变量的值。

变量的名称可以包含字母、数字和下划线，而且是大小写敏感的。也就是说："foo"，"Foo"和"FOO"是三个不同的变量名。

变量在声明时需要赋初始值；使用时，需要给在变量名前加上$符号，如果变量名包含多个字符，我们应该用小括号 () 或大括号 {} 把变量括起来。

自定义变量

自定义变量，顾名思义，就是程序员自己定义的变量。引入自定义变量后，我们可以将上面的 Makefile 改写成：

```
Objs := main.o add.o sub.o mul.o div.o
Out  := main

$(Out): $(Objs)
    gcc $(Objs) -o $(Out)
main.o: main.c algs.h
    gcc -c main.c -Wall -g
add.o: add.c algs.h
    gcc -c add.c -Wall -g
sub.o: sub.c algs.h
    gcc -c sub.c -Wall -g
mul.o: mul.c algs.h
    gcc -c mul.c -Wall -g
div.o: div.c algs.h
    gcc -c div.c -Wall -g

.PHONY: clean rebuild
clean:
    rm -f $(Out) $(Objs)
rebuild: clean $(Out)
```

预定义变量

预定义变量，即预先定义好的变量，这些变量的含义是事先确定的。

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

引入预定义变量后，我们可以将上面的 Makefile 改写成：

```
Objs := main.o add.o sub.o mul.o div.o
Out  := main
CC   := gcc
CFLAGS := -Wall -g

$(Out): $(Objs)
    $(CC) $(Objs) -o $(Out)
main.o: main.c algs.h
    $(CC) -c main.c $(CFLAGS)
add.o: add.c algs.h
    $(CC) -c add.c $(CFLAGS)
sub.o: sub.c algs.h
    $(CC) -c sub.c $(CFLAGS)
mul.o: mul.c algs.h
    $(CC) -c mul.c $(CFLAGS)
div.o: div.c algs.h
    $(CC) -c div.c $(CFLAGS)

.PHONY: clean rebuild
clean:
    $(RM) $(Out) $(Objs)
rebuild: clean $(Out)
```

规则中的特殊变量

规则中的特殊变量就是某些具有特殊含义的变量，它的含义和当前规则有关。

变量名	含义
$@	目标
$<	第一个依赖文件
$^	所有依赖文件，以空格分隔
$?	所有日期新于target的依赖文件

引入自动变量后，我们可以将上面的 Makefile 改写成：

```
Objs := main.o add.o sub.o mul.o div.o
Out  := main
CC   := gcc
CFLAGS := -Wall -g

$(Out): $(Objs)
    $(CC) $^ -o $@
```

```
main.o: main.c algs.h
    $(CC) -c $< $(CFLAGS)
add.o: add.c algs.h
    $(CC) -c $< $(CFLAGS)
sub.o: sub.c algs.h
    $(CC) -c $< $(CFLAGS)
mul.o: mul.c algs.h
    $(CC) -c $< $(CFLAGS)
div.o: div.c algs.h
    $(CC) -c $< $(CFLAGS)

.PHONY: clean rebuild
clean:
    $(RM) $(Out) $(Objs)
rebuild: clean $(Out)
```

4.5 模式规则(了解)

模式规则类似于普通规则。只是在模式规则中，target 需要包含模式字符"%"，"%" 可以匹配任何非空字符串。规则的依赖中同样可以使用"%"，依赖中模式字符"%"的取值和目标中的"%"的取值一样。

(注：模式规则，就是说我们只要定义规则的模式即可。make 会根据规则的模式自动生成具体的规则)。

例如：模式规则"%.o：%.c"，表示的含义是：所有的.o文件依赖于对应的.c文件。

下面示例就是 Makefile 的一个模式规则，由所有的.c文件生成对应的.o文件：

```
%.o: %.c
    $(CC) -c $< -o $@
```

有了模式规则后，我们可以这样写 Makefile：

```
Objs := main.o add.o sub.o mul.o div.o
Out  := main
CC   := gcc
CFLAGS := -Wall -g

$(Out): $(Objs)                    # Objs := main.o add.o sub.o mul.o
div.o
    $(CC) $^ -o $@
```

```
%.o: %.c algs.h                          # 这里应用了Makefile的隐式推导，%.o是与上一个规则的依赖进行匹配，即$(Objs)
    $(CC) -c $< $(CFLAGS)

.PHONY: clean rebuild
clean:
    $(RM) $(Out) $(Objs)
rebuild: clean $(Out)
```

4.6 内置函数(了解)

Makefile 也支持函数调用，其语法和引用变量非常类似：

```
$(<function> <arguments>)
或
${<function> <arguments>}
```

<function>为函数名，<arguments>为参数列表。参数之间以逗号,分隔，而函数名和参数之间以"空格"分隔。

Makefile 内置的函数并不算多，这里我们介绍两个：

通配符函数

```
格式：
    $(wildcard <pattern>)
作用：
    查找符合<pattern>的所有文件列表
返回值：
    返回所有符合<pattern>的文件名，文件名之间以空格分隔。
```

示例：

```
Srcs := $(wildcard *.c)
# 查找当前目录下，所有以.c结尾的文件名。将文件名以空格分隔，并赋值给变量Srcs。
```

模式替换函数

格式：

```
$(patsubst <pattern>,<replacement>,<text>)
```

作用：

查找<text>中符合模式<pattern>的单词(单词以空白字符分隔)，将其替换为<replacement>。

注：<pattern>可以包括通配符%，表示任意长度的字符串。如果<replacement>中也含有%，那么，<replacement>中的%所代表的字符串和<pattern>中%所代表的字符串相同。

返回值：

返回替换后的字符串

示例：

```
$(patsubst %.c, %.o, foo.c.c bar.c)
# 将字符串 foo.c.c bar.c 中符合模式 %.c 的单词替换成 %.o，返回结果为 foo.c.o bar.o
```

引入内置函数后，我们可以将上面的 Makefile 改写成这样：

```
Srcs := $(wildcard *.c)
Objs := $(patsubst %.c, %.o, $(Srcs))
Out  := main
CC   := gcc
CFLAGS := -Wall -g

$(Out): $(Objs)
    $(CC) $^ -o $@
%.o: %.c algs.h
    $(CC) -c $< $(CFLAGS)

.PHONY: clean rebuild
clean:
    $(RM) $(Out) $(Objs)
rebuild: clean $(Out)
```

课堂小练习

请书写 Makefile 完成下面任务：在一个目录有多个.c文件，将每一个.c文件单独编译链接生成一个可执行程序 (注意，尽量提升通用性，而且要有 clean 和 rebuild 的功能)。