

1 makefile文件

makefile:

makefile文件是用来管理项目工程文件，通过执行make命令，make就会解析并执行makefile文件。

makefile命名：makefile或者Makefile

makefile的编写：

规则：

目标：依赖
(tab)命令

第一个版本：

```
main: main.c fun1.c fun2.c sum.c
    gcc -o main main.c fun1.c fun2.c sum.c
```

第二个版本

```
1 main: main.o fun1.o fun2.o sum.o
2     gcc -o main main.o fun1.o fun2.o sum.o
3
4 main.o: main.c
5     gcc -c main.c -I./
6
7 fun1.o: fun1.c
8     gcc -c fun1.c
9
10 fun2.o: fun2.c
11     gcc -c fun2.c
12
13 sum.o: sum.c
14     gcc -c sum.c
```

版本三，使用自定义变量和自动变量

\$< 表示使用目标中的第一个文件

\$@表示使用目标中的所有文件

类似于C语言中的宏定义

```
1 target=main #生成目标文件的别名
2 CC=g++ #指定编译器
3 CPPFLAGS=-I./
4
5 objects=main.o func1.o func2.o sum.o #生成target所需要的依赖文件
6
7 $(target): $(objects)
8     $(CC) -o main $(objects)
9
10 main.o: main.c #编译main.c
11     $(CC) -c $< $(CPPFLAGS)
12
13 func1.o: func1.c #编译func1.c
14     $(CC) -c $<
15
16 func2.o: func2.c #编译func2.c
17     $(CC) -c $<
18
19 sum.o: sum.c #编译sum.c
20     $(CC) -c $<
```

版本四

以后只需要修改第二、第三行即可

模式规则

至少在规则的目标定义中要包含‘%’, ‘%’表示一个或多个, 在依赖条件中同样可以使用‘%’, 依赖条件中的‘%’的取值取决于其目标:

比如: main.o:main.c fun1.o:fun1.c fun2.o:fun2.c, 说的简单点就是: xxx.o:xxx.c

makefile 的第三个版本:

```
target=main
object=main.o fun1.o fun2.o sum.o
CC=gcc
CPPFLAGS=-I./

$(target):$(object)
    $(CC) -o $@ $^

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

使用函数, 不用手动输入目标文件

```
1 target=main #生成目标文件的别名
2 CC=g++      #指定编译器
3 CPPFLAGS=-I./
4 src=$(wildcard *.c)
5 objects=$(patsubst %.c, %.o, $(src)) #生成target所需要的依赖文件
6
7 $(target): $(objects)
8     $(CC) -o $@ $^
9
10 %.o:%.c
11     $(CC) -c $< $(CPPFLAGS)
```

终极版本, 带clean

```
1 1 target=main #生成目标文件的别名
2 2 CC=g++      #指定编译器
3 3 CPPFLAGS=-I./
4 4 src=$(wildcard *.c)
5 5 objects=$(patsubst %.c, %.o, $(src)) #生成target所需要的依赖文件
6 6
7 7 $(target): $(objects)
8 8     $(CC) -o $@ $^
9 9
10 10 %.o:%.c
11 11     $(CC) -c $< $(CPPFLAGS)
12 12 .PHONY:clean
13 13 clean:
14 14     rm -f $(objects) $(target)
15 ~
```

```
1 target=main #生成目标文件的别名
2 CC=g++      #指定编译器
3 CPPFLAGS=-I./
4 src=$(wildcard *.c)
5 objects=$(patsubst %.c, %.o, $(src)) #生成target所需要的依赖文件
6
7 $(target): $(objects)
8     $(CC) -o $@ $^
9
10 %.o:%.c
11     $(CC) -c $< $(CPPFLAGS)
12 .PHONY:clean
13 clean:
14     rm -f $(objects) $(target)
```

2 GDB调试

gdb调试：

gdb是在程序运行的结果与预期不符合的时候，可以使用gdb进行调试，特别注意的是：使用gdb调试需要在编译的时候加-g参数。

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

启动gdb：

```
gdb program
```

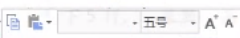
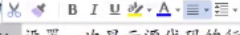
执行程序：

```
run
```

```
start
```

GDB 可以打印出所调试程序的源代码，当然，在程序编译时一定要加上-g的参数，把源程序信息编译到执行文件中。不然就看不到源程序了。当程序停下来以后，GDB 会报告程序停在了那个文件的第几行上。你可以用list命令来打印程序的源代码，默认打印10行，list命令的用法如下所示：

- list linenum：打印第 linenum 行的上下文内容。
- list function：显示函数名为 function 的函数的源程序。
- list：显示当前行后面的源程序。
- list-：显示当前文件开始处的源程序。
- list file:linenum：显示 file 文件下第 n 行
- list file: function：显示 file 文件的函数名为 function 的函数的源程序

一般是打印当前行的上， 数是上2行下8行，默认是10行，当然，你也可以定制显示的， 置一次显示源程序的行数。

- set listsize:count：设置一次显示源代码的行数。
- show listsize：查看当前 listsize 的设置。

断点操作：

```
b linenum
b func
b file:linenum
b file:func
```

```
info break
```

```
disable m n | m-n
enable  m n | m-n
```

```
delete m n | m-n
```

2. 文件读写（内置）

```

1 //IO函数测试
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <sys/types.h>
6 #include <unistd.h>
7 #include <sys/stat.h>
8 #include <fcntl.h>
9
10 int main(int argc, const char * argv[])
11 {
12     printf("argv[0] = %s\n", argv[0]);
13     //open file
14     int fd = open(argv[1], O_RDWR | O_CREAT, 0777);
15     if(fd == -1)
16     {
17         perror("open error");
18         return -1;
19     }
20
21     // write file
22     write(fd, "hello world", strlen("hello world"));
23     lseek(fd, 0, SEEK_SET);
24
25     //read file
26     char buf[1024];
27     memset(buf, 0, sizeof(buf));
28     int n = read(fd, buf, sizeof(buf));
29     printf("n = %d, buff = [%s]\n", n, buf);
30     close(fd);
31     return 0;
32 }

```

读取目录

```

1 //opendir 打开
2 //readdir 读取
3 //closedir 关闭

```

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <dirent.h>
6 #include <sys/types.h>
7
8 int main(int argc, const char* argv[])
9 {
10     //打开目录
11     DIR *pDir = opendir(argv[1]);
12     if(pDir == NULL)
13     {
14         perror("opendir error");
15         return -1;
16     }
17     //循环读取目录
18     struct dirent * pDent = NULL;
19     while((pDent=readdir(pDir)) != NULL)
20     {
21         //过滤掉.和..文件
22         if(strcmp(pDent->d_name, ".") == 0 || strcmp(pDent->d_name, "..") == 0)
23         {
24             continue;
25         }
26         printf("[%s]", pDent->d_name);
27         //判断文件类型
28         switch(pDent->d_type)
29         {
30             case DT_REG:
31                 printf("普通文件\n");
32                 break;
33             case DT_DIR:
34                 printf("目录文件\n");
35                 break;
36             case DT_LNK:
37                 printf("链接文件\n");
38                 break;
39             default:
40                 printf("未知文件\n");
41                 break;
42         }
43     }
44     //关闭目录
45     closedir(pDir);
46     return 0;
47 }

```

文件重定向

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <fcntl.h>
6 #include <sys/types.h>
7 #include <sys/stat.h>
8
9 //使用dup2函数实现重定向
10 int main(int argc, const char * argv[])
11 {
12     //打开文件
13     int fd = open(argv[1], O_RDWR | O_CREAT, 0777);
14     if(fd < 0)
15     {
16         perror("open error");
17         return -1;
18     }
19
20     //调用dup2函数，将标准输出重定向到文件中
21     dup2(fd, STDOUT_FILENO);
22     printf("ni hao hello world");
23     close(fd);
24     close(STDOUT_FILENO);
25     return 0;
26 }
~

```

linux fork 函数

fork函数的返回值

父进程返回的是子进程的PID，该PID是大于零的

子进程返回的是0

因此可以根据返回值来判断是子进程还是父进程

父子进程谁先抢到时间片谁先执行。

wait 和 waitpid函数

```

wait函数:
pid_t wait(int *status);
返回值:
    >0: 回收的子进程的PID
    -1: 没有子进程
参数:
    status: 子进程的退出状态
        if(WIFEXITED(status))
        {
            WEXITSTATUS(status)
        }
        else if(WIFSIGNALED(status))
        {
            WTERMSIG(status)
        }

```

waitpid函数:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

参数:

pid:

pid>0: 表示等待指定的子进程

pid=-1: 表示等待任意子进程

status:

同wait函数

options:

0: 表示阻塞

WNOHANG: 表示不阻塞

返回值:

>0: 回收的子进程的PID

=0: 若options取值为WNOHANG, 则表示子进程还或者

-1: 表示已经没有子进程了

注意: 调用一次waitpid或者wait函数只能回收一个子进程。