Make

School of Computer
Wuhan University

1 make & Makefile
- 程序的编写步骤
- 依赖关系图
- 工程管理与 make
- make 的工作原理
- 规则与依赖关系
- 通配符的使用
- 变量的定义和引用
- 自动变量
- 模式规则
- Makefile 的结构
- make 命令参数与选项
- 常见的错误
- 依赖关系的自动生成
- Turbo C 的 make
- 与 make 相关的命令
- 参考文献

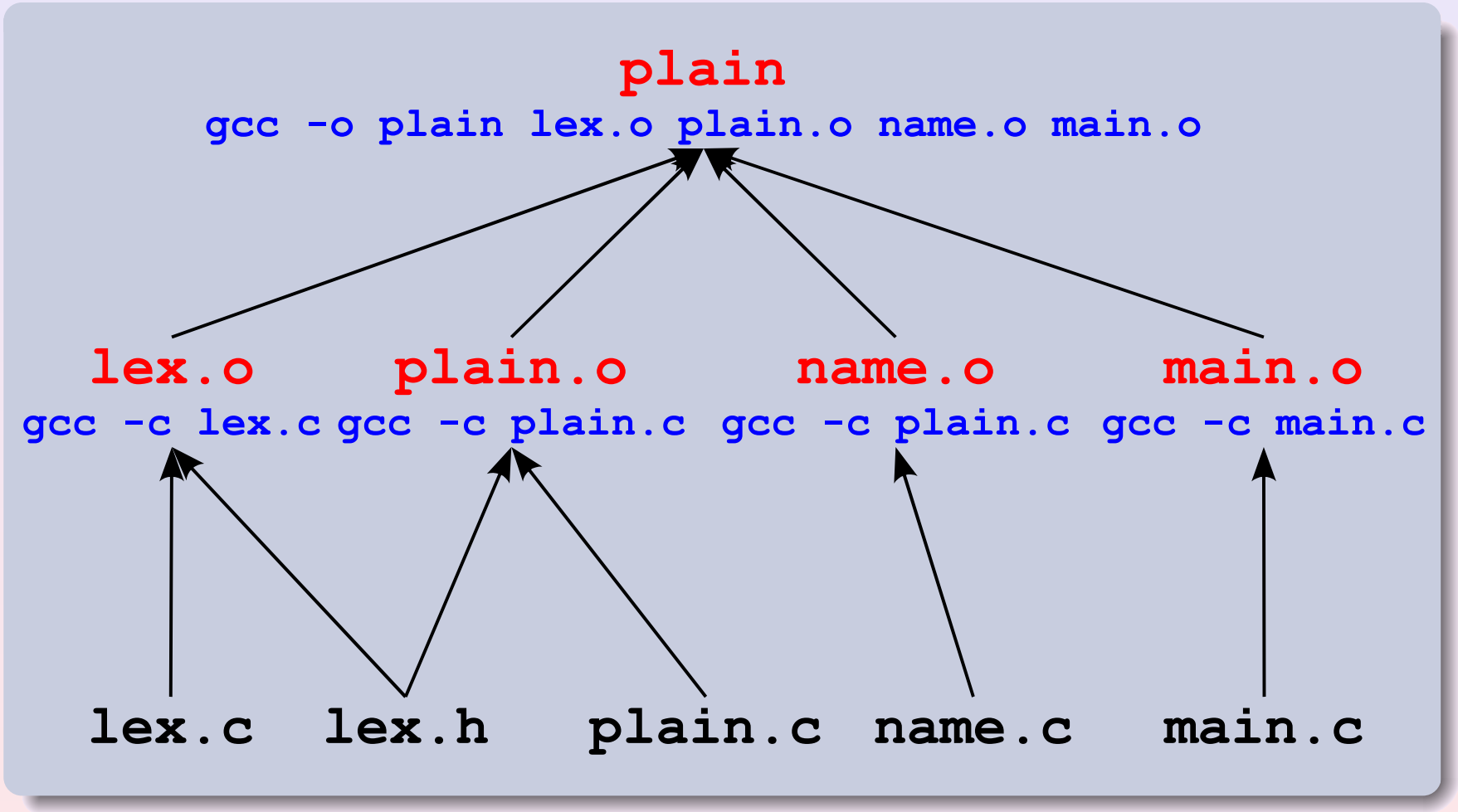
Free as in Freedom

GNU's Not UNIX

程序的编写步骤

1. 编写源程序，如：XL 语言的lex.h, lex.c, name.c, plain.c, main.c, 相关的文本编辑器有：vi, emacs等, 如：
   ```
   $ emacs lex.c
   ```
2. 编译生成目标文件，如：
   ```
   $ gcc -c lex.c
   $ gcc -c plain.c
   $ gcc -c name.c
   $ gcc -c main.c
   ```
   如果源程序有误，返回①;
3. 联结生成执行文件或库文件，如：
   ```
   $ gcc -o plain lex.o plain.o name.o main.o
   ```
   如果联结出错，如：出现未定义的函数, 则返回①;
4. 测试执行文件，如果有问题返回①;

源文件、目标文件和执行文件的依赖关系图



问题及解决方法

1. 如果某个源文件 (source file)修改必须重新生成所有与之关联的目标文件 (targets), 如：对lex.h的改动, 必须重新手工编译lex.c, plain.c及最后生成plain, 这对管理工程造成了很大的不便;
2. IDE(Integrated Development Environment) 提供集成的工程管理，如: Turbo C 的Project, Visual Studio 的 Workspace 等, 工程管理简单, 但是需要平台的支持，离开的特定的 IDE，将不能重新生成目标文件;
3. make 通过Makefile中设定的源文件和目标文件依赖关系以及相应的生成操作，比较源文件和目标文件的时间，如果前者更新，表示源文件已经修改，make将按照Makefile对应的动作自动生成新的目标文件。

make 的优缺点

1. 不依赖于特定的开发平台，不需要知道Makefile的细节就可实现对源代码包的安装，如，开源软件的编译安装过程如下：

```
# wget http://caml.inria.fr/pub/distrib/ocaml-3.09/
ocaml-3.09.0.tar.bz2
# tar jxvf ocaml-3.09.0.tar.bz2
# ./configure
# make world
# make install
```

2. 不依赖于特定的程序设计语言，如：LaTeX 生成dvi、ps和pdf文件也可以用make进行管理；
3. 文法古怪，格式严格，初学者特别容易出错；
4. GNU make最早由 Richard Stallman (http://www.stallman.org/) 和 Roland McGrath (http://www.frob.com/~roland/) 编写；

Example of Makefile

Makefile for plain

```
$ cat Makefile
plain: lex.o plain.o name.o main.o
	gcc -o ./plain plain.o lex.o name.o main.o
lex.o: lex.c lex.h
	gcc -c lex.c
plain.o: plain.c lex.h
	gcc -c plain.c
name.o: name.c
	gcc -c name.c
main.o: main.c
	gcc -c main.c
$ make
gcc -c lex.c
gcc -c plain.c
gcc -c main.c
gcc -c name.c
gcc -o ./plain plain.o lex.o name.o main.o
```

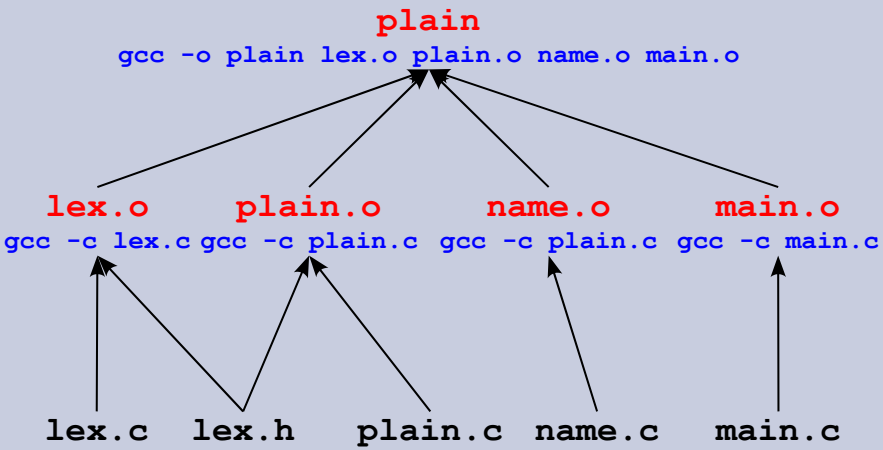
Example of Makefile

Makefile for plain

```
$ touch lex.h
$ make
gcc -c lex.c
gcc -c plain.c
gcc -o ./plain plain.o lex.o name.o main.o
$ touch main.c
$ make
gcc -c main.c
gcc -o ./plain plain.o lex.o name.o main.o
$ mv lex.h lex.h.bak
$ make
make: *** No rule to make target `lex.h', needed by `lex.o'.
Stop.
```

make 的工作原理

- make通过读取一个Makefile文件，首先对该文件进行语法分析，如果Makefile文件格式有误，则报错并退出执行；
- 如果输入的Makefile文件文法正确，make分析Makefile文件中描述的目标 (Targets)和前提 (Prerequisites)的关系，并建立一颗非环路的依赖关系图 (Dependency Graph)，如：



- 关系图中入度为零的结点不依赖任何前提，一般是源文件；
- make比较依赖关系图中的前题和目标文件的时间，如果前提时间更新，或目标文件不存在，则利用Makfile中相关目标的生成操作命令生成新目标，如此反复直到最后目标是最新的为止；

依赖关系的描述

- Makefile前题和目标的依赖关系及相关的生成操作称为规则(Rules)，其文法如下：

  target1 target2 ： prerequisite1 prerequisite2 ...

  [redacted]*action1*

  [redacted]*action2*

  [redacted]...

- action绝对以[redacted](横向跳格符 TABLATION) 为行首字符;
- action可以是任意的命令, make以当前用户缺省的shell执行该命令：

```
$ echo $SHELL
/bin/bash
```

- 如：

```
lex.o : lex.h lex.c
[redacted]@echo Rebuild lex.o ...
[redacted]gcc -c lex.c
```

一条规则中可以出现多重目标

- 如下述修改后的Makefile与原Makefile等价：

```
lex.o : lex.c
	gcc -c lex.c
plain.o : plain.c
	gcc -c lex.c
lex.o plain.o : lex.h
```

- 如果一个目标被多条规则描述，应只有一条有action成分，如果出现有多个action，则make仅执行最后一条规则对应的action，如，上例中如果对多重目标加上action

```
lex.o plain.o : lex.h
	gcc -c lex.c
$ make
Makefile:16: warning: overriding commands for target `lex.o'
Makefile:7: warning: ignoring old commands for target `lex.o'
```

最终目标

- make执行时，读取当前目录下的名称为Makefile的文件，将第一条规则中出现的目标作为最后要实现的目标，按照依赖关系图，从叶到该目标逐一构造相关目标，如，将上例Makefile的规则次序做一下的修改:

```
$ cat Makefile
......
lex.o: lex.h lex.c
        gcc -c lex.c
plain: lex.o plain.o name.o main.o
        gcc -o ./plain plain: lex.o plain.o name.o main.o
......
$ make
make: `lex.o' is up to date.
```

- make可以通过命令行的目标选项设定最终目标，其格式为：make *target1 target2 ...*，如：

```
$ make plain
$ make notarget
make: *** No rule to make target `notarget'. Stop.
```

虚拟目标 1/2

- 一般规则中的目标名与要生成文件名对应，如：plain和lex.o, 但有些特殊情况目标名并不生成任何文件，这样的目标称为虚拟目标 (Phony Targets), 如:

```
del:
	rm -f *.o plain
all: plain retval
```

- make在处理虚拟目标时，由于对应的文件不存在，因此在任何情况下都需要更新该虚拟目标，如“make del”将删除当前目录下的所有的.o文件和plain文件：

虚拟目标 2/2

- 如果碰巧当前目录下存在一个文件名与虚拟目标对应，这时虚拟目标则变成了实目标，如：

```
$ touch del↵
$ make del↵
make: `del' is up to date.
```

- make为了保证虚拟目标不会因为对应文件的干扰而变为实目标，特提供下述文法申明虚拟目标，并且缺省设置一些虚拟目标，如: all, clean和install等：

```
.PHONY: del
```

这样即使del存在，系统也会认为del是虚拟目标，“make del↵”将无条件地删除相关文件，习惯上可以用系统默认的虚拟目标“clean”来做相关删除操作, 如：

```
clean:
	rm -f *.o plain
```

修改后的 Makefile

Makefile for all

```
$ cat Makefile
all: plain retval

plain: lex.o plain.o name.o main.o
	gcc -o ./plain plain.o lex.o name.o main.o
retval: lex.o retval.o name.o main.o
	gcc -o ./retval retval.o lex.o name.o main.o
lex.o: lex.c lex.h
	gcc -c lex.c
plain.o: plain.c lex.h
	gcc -c plain.c
retval.o: retval.c lex.h
	gcc -c retval.c
name.o: name.c
	gcc -c name.c
main.o: main.c
	gcc -c main.c
clean:
	rm -f *.o plain retval
```

通配符的使用

- 在Makefile中可以像shell一样使用通配符，如: *, ?, ~, [...]和[^...]等，如：“*.c” 将替换成当前目录下所有的以“.c” 结尾的文件名中间以空格相间的字符串, 如: “lex.c main.c name.c plain.c retval.c”
- 利用通配符可以减少Makefile中的字符输入，如：

  ```
  *.o: lex.h
  ```

  在系统初次执行make时，由于当前目录下没有“.o” 文件，因此“*.o” 所替换的字符串是空串，上规则将转换为：

  ```
  : lex.h
  ```

  由于make接受目标为空的规则，并不向用户提示任何错误信息，但是由于该规则的目标为空，没有任何的作用，而不是用户期待的所有的*.o目标与lex.h相关。要实现这样的关联可以通过Makefile更强大的内置函数功能实现：

  ```
  $(subst .c,.o,*.c): lex.h
  ```

  注意：.c,.o中间没有空格！

变量的定义和引用

- Makefile支持强大的变量定义，并且提供许多内置变量，同时还具有宏定义和函数定义等功能，这些使得make能产生奇效；
- 变量的定义文法如下：

  var_name = *value*

  var_name := *value*

  前者定义了延迟展开变量 (Recursively Expanded Variable)；后者定义了立刻展开变量 (Simply Expanded Variable)，两者的区别：如果*value*中如果出现变量的引用，该引用是在*var_name*定义时就展开 (simply)，还是在下文引用*var_name*时才展开 (recursively)，因此如果*value*中没有变量引用，两者效果一样；
- 变量的引用格式如下：

  `${var_name}` 或 `$(var_name)`

  注意：这是为了区别Makefile中其他单词强制规定，否则就出错，如：如果CC = gcc，在 “$CC -c lex.c” 中引用时，make会将 “$CC” 理解为 “$(C)C”，而由于变量 “$(C)” 没有定义，系统用空串替换，最后的展开结果为 “C”，从而在调用该命令时由于 “C” 命令不存在而出错。

= 与:= 的区别

```
CC = gcc $(CFLAGS)
CFLAGS = -c -Os
lex.o: lex.h lex.c
	$(CC) lex.c
CFLAGS = -Os
plain: lex.o plain.o name.o main.o
	$(CC) -o ./plain plain.o lex.o name.o main.o
```

如果将第一行改为“CC := gcc $(CFLAGS)”, 则make在扫描到这行时，由于$(CC)是立刻展开的简单变量，而由于“$(CFLAGS)”此时还没有定义, 因此$(CC)的值是“gcc”，但是上述使用的是延迟展开的变量类型，因此在引用时才展开其值，第一次引用时“$(CFLAGS)”的值为“-c -Os”, 此时$(CC) 的值展开后为“gcc -c -Os”; 而第二次引用时“$(CFLAGS)”的值为“-Os”, 此时$(CC)的值展开后为“gcc -Os”;

注意：make能够检测到循环定义并报错，如：

```
CFLAGS = -c -Os $(CFLAGS)
```

make将报错：Makefile:3: *** Recursive variable `CFLAGS' references itself (eventually). Stop.; 对“$(CFLAGS)”追加选项可用“CFLAGS += -c -Os $(CFLAGS)”实现。

shell 环境变量和 shell 命令在变量定义中的使用

- Makefile可以通过$$引用shell的环境变量，其格式为：$$NAME或$${NAME}，其中NAME是shell环境变量，如：

```
install : plain
	test -d ${HOME}/bin && cp plain ${HOME}/bin/
```

- 在变量定义中其所定义的值还可以是shell命令输出的结果，其格式为：

  VAR = $(shell *command*) 或 *VAR* := $(shell *command*)

  其中“$(shell *command*)”是Makefile的函数调用，shell是内建函数名，*command* 是函数参数，如：

```
START_TIME := $(shell date)
CURRENT_TIME = $(shell date)
```

  注意：=与:=的区别;

Automatic Variables

- 为了方便用户书写Makefile，系统提供一组形式固定，但其值随上下文环境变化的变量，称为Automatic Variable;
- 自动的值来自当前规则的前提和目标，其值随规则的改变而改变，一般在当前规则的action中引用。
- 常用的自动变量有：
  - $@ 当前规则的目标名；如果当前规则是多目标，则$@是make正在激活的目标名；
  - $^ 当前规则的所有前提名列表，中间以空格间隔；
  - $< 当前规则中第一个前提名；
  - $? 当前规则所有的其对应文件的修改日期比目标对应文件修改日期更新的前提名列表；
  - $* 当前规则中目标名的前缀，要求规则的前提和目标的前缀是相同的，用在模式规则中；

Examples

- 原Makefile中的下段：

```
plain: plain.o lex.o name.o main.o
	$(CC) -o ./plain plain.o lex.o name.o main.o
```

- 可简写为：

```
plain: plain.o lex.o name.o main.o
	$(CC) -o ./$@ $^
```

- 列表所有的比make最后一次执行还要新的源程序名：

```
new_files: *.c
	echo $? | tr ' ' '\n'
	touch news_files
```

Pattern Rules

- 在很多情况下，目标和前提的关系是固定的，如：.c文件生成.o文件，.l文件生成.c文件，.y文件生成.c文件等，为了避免在Makefile中对每个.o目标重复相同的规则，make提供了一个对一类源文件到目标的规则定义手段，称为Pattern Rule，如：

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

该模式规则定义了.c源文件到.o目标的成生规则，其中："%" 是模式规则特用的通配符，匹配任意的非空字符串，make生成一个目标时，如果需要一个前提，如:"foo.o"，此时 "foo.o" 匹配上述规则，make激活该规则，将规则中的 "%" 用 "foo" 替换，并查看目标 "foo.o" 和前提 "foo.c" 的时序关系，如果 "foo.c" 更新，则执行 "gcc -c foo.c" 生成新目标。由于该规则不能作为最后目标，并只有在需要时才能激活，因此称之为潜规则 (Implicit Rule)，make还有其他形式的潜规则，但模式规则最常用；

Suffix Rules 1/2

- 老式的make还支持一种称为Suffix Rule的潜规则，在新make下仍然可用，但是表达没有模式规则灵活，如上例可按后缀规则表达如下：

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

- 后缀规则不能表达所有的.o都与.c和lex.h有关，而模式规则可以表示如下:

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

Suffix Rules 2/2

- make缺省定义了一些常用的后缀及相应后缀规则，如上述.c.o规则：

  .SUFFIXES:

  删除所有的后缀及相关规则；

  ```
  .SUFFIXES: .tex .dvi .pdf
  .dvi.tex:
  	latex $<
  .pdf.dvi:
  	dvipdfm $<
  ```

  将追加新的后缀及相关规则；

更简洁的 Makefile

Makefile for all

```
$ cat Makefile
CC = gcc
OBJ = lex.o name.o main.o

%.o : %.c
	@echo Rebuild $*.o from $*.c
	$(CC) -c -o $@ $<

all: plain retval

plain: $(OBJ) plain.o
	$(CC) -o ./$@ $^

retval: $(OBJ) retval.o
	$(CC) -o ./$@ $^

lex.o plain.o retval.o : lex.h

clean:
	rm -f *.o plain retval
```

小节：Makefile 的结构

- 注释：和shell script一样以#开始直到行结束；
- 规则：由目标、前提和action三个部分组成；
  - 按能否成为最终目标分为：显规则 (Explicit Rules)和潜规则，模式规则和后缀规则是潜规则的一种；
  - 按规则的作用可分为实规则和虚拟规则：实规则产生和规则名对应的文件间，虚拟规则不产生对应文件；
- 变量定义和变量引用，如：用户定义的变量和系统内置变量，内置变量有具有恒定值的变量，如：CC = cc；还有随规则改变而改变的自动变量；此外还可引用shell的环境变量，其格式为$$NAME或$${NAME}，其中NAME为shell的环境变量；
- 高级功能：宏、函数定义和调用等；

make 命令格式 1/2

- Syntax of make:
  `make [options] [target1 target2 ...]`
- 常用的选项如下：
  - `-f file` 以*file*作为Makefile;
  - `-d` 输出 debug 信息，如：`make -d -f Myfile > log 2>&1`
  - `-p` 打印Makefile文件的所有的规则，包括系统内置的变量和潜规则;
  - `-k` 最大可能更新目标，而不是在遇见第一个错误后就停止工作;
  - `-n` 打印需要更新目标对应的action，而不是执行;
  - `-i` 忽略执行更新目标的返回码，即忽略所以的错误;
- 此外make还可以在命令行加上变量定义选项，对Makefile追加或修改变量定义，如：
  `make "CC = cc"`
  强制将Makefile的变量$(CC)设置为cc;

make 命令格式 2/2

- make在执行时如果没有-f *file*选项，首先以当前目录下的名为makefile的文件作为其分析处理的对象，其次是名为Makefile的文件；如果都不存在，make将报错如下：

  ```
  make: *** No targets specified and no makefile found.
  Stop.
  ```

  习惯上用大写字母开始的Makefile作为缺省的make文件，这样在ls文件列表时总是在前面；
- 每个action是在新派生的shell环境下执行，如下面的两个actions 就不能得到期望的结果：

  ```
  cd dir
  command_concerned_dir
  ```

  但，如果将两个actions改为一个就能达到目的：

  ```
  cd dir && command_concerned_dir
  ```

常见的错误

- action不是以[redacted]开始;
- 多余的[redacted], 如:
  ```
  lex.o plain.o retval.o : lex.h
  [redacted]# a redundant tab
  ```
  make会将一个空action覆盖%.o: %.c的action;
- 变量的引用没有加上$( )或${ };
- shell环境变量的引用没有加上$$;
- 错误使用自动变量$?、$、$*和$等;
- 有关联的action没有写在一行;
- 依赖关系描述不完整;

依赖关系的自动生成 1/2

- “gcc -MM lex.c” 以Makefile的规则形式输出目标lex.o的前提：

```
lex.o: lex.c lex.h
```

- “gcc -MM *.c >> Makefile” 将当前目录下的所以的.c文件的依赖关系追加到Makefile中；
- “makedepend *.c” 和上命令有相同的效果；
- 还可以在Makefile中加上：

```
depend: lex.c plain.c retval.c name.c main.c
        $(CC) -MM $(CFLAGS) $^ > $@
include depend
```

在make all之前先make depend生成depend文件；但是每次对源程序的修改必须手工再次执行make depend，GNU Make Manual 提供提供了一个完全自动的解决方法，修改后的Makefile如下所示；

依赖关系的自动生成 2/2

New Makefile for all

```
$ cat Makefile
CC = gcc
OBJ = lex.o name.o main.o
SOURCES = name.c lex.c retval.c plain.c main.c
include $(subst .c,.d,$(SOURCES))

%.d: %.c
	$(CC) -MM $(CFLAGS) $< > $@.$$$$;                    \
	sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@;     \
	rm -f $@.$$$$
%.o : %.c
	@echo  Rebuild $*.o from $*.c
	$(CC) -c -o $@ $<
all: plain retval
plain: $(OBJ) plain.o
	$(CC) -o ./$@ $^
retval: $(OBJ) retval.o
	$(CC) -o ./$@ $^
clean:
	rm -f *.o plain retval
```

Turbo C 有自带的 make, 但仅部分支持 GNU make 的功能

Makefile for Turbo c

```
CFLAGS  = -v -O
INCLUDE = -I\tc\include;.
LIB = -L\tc\lib;\tc\tools
CC  = \tc\tcc
MODEL  = -mc
.c.obj:
        $(CC) $(INCLUDE) -c $(CFLAGS) $(MODEL) $*.c
OBJ = lex.obj name.obj main.obj

all: plain.exe retval.exe
plain.exe: $(OBJ) plain.obj
        $(CC) -eplain.exe $(LIB) $(MODEL) $(OBJ) plain.obj
retval.exe: $(OBJ) retval.obj
        $(CC) -eretval.exe $(LIB) $(MODEL) $(OBJ) retval.obj

lex.obj plain.obj retval.obj : lex.h
clean:
        del  *.obj
        del plain.exe
        del retval.exe
```

相关命令

- automake和autoconf: 自动配置可移植的源码包，使得源码安装成为: "./configure --> make --> make install" 三步曲;
- xmkmf和imake: 从第三方软件提供的Imakefile创建Makefile, 用于 X 窗口软件的源码安装;
- shell script: action的语言;
- sed和awk: 对字符流进行批处理的重要工具;

For Further Reading

R. Stallmen, R. McGrath, and P. Smith.
GNU Make Manual
http://www.gnu.org/software/make/manual/make.html, 2002.

R. Mecklenburg.
Managing Projects with GNU make, 3rd Edition (in My CD-ROM).
O'Reilly, 2004.
Code Examples: http://examples.oreilly.com/make3

C. Newham.
Learning the bash Shell, 3rd Edition.
O'Reilly, 2005.
Code Examples: http://examples.oreilly.com/bash2

本章小节

1 make & Makefile
- 程序的编写步骤
- 依赖关系图
- 工程管理与 make
- make 的工作原理
- 规则与依赖关系
- 通配符的使用
- 变量的定义和引用
- 自动变量
- 模式规则
- Makefile 的结构
- make 命令参数与选项
- 常见的错误
- 依赖关系的自动生成
- Turbo C 的 make
- 与 make 相关的命令
- 参考文献