

# 编译、构建和调试

孟宁

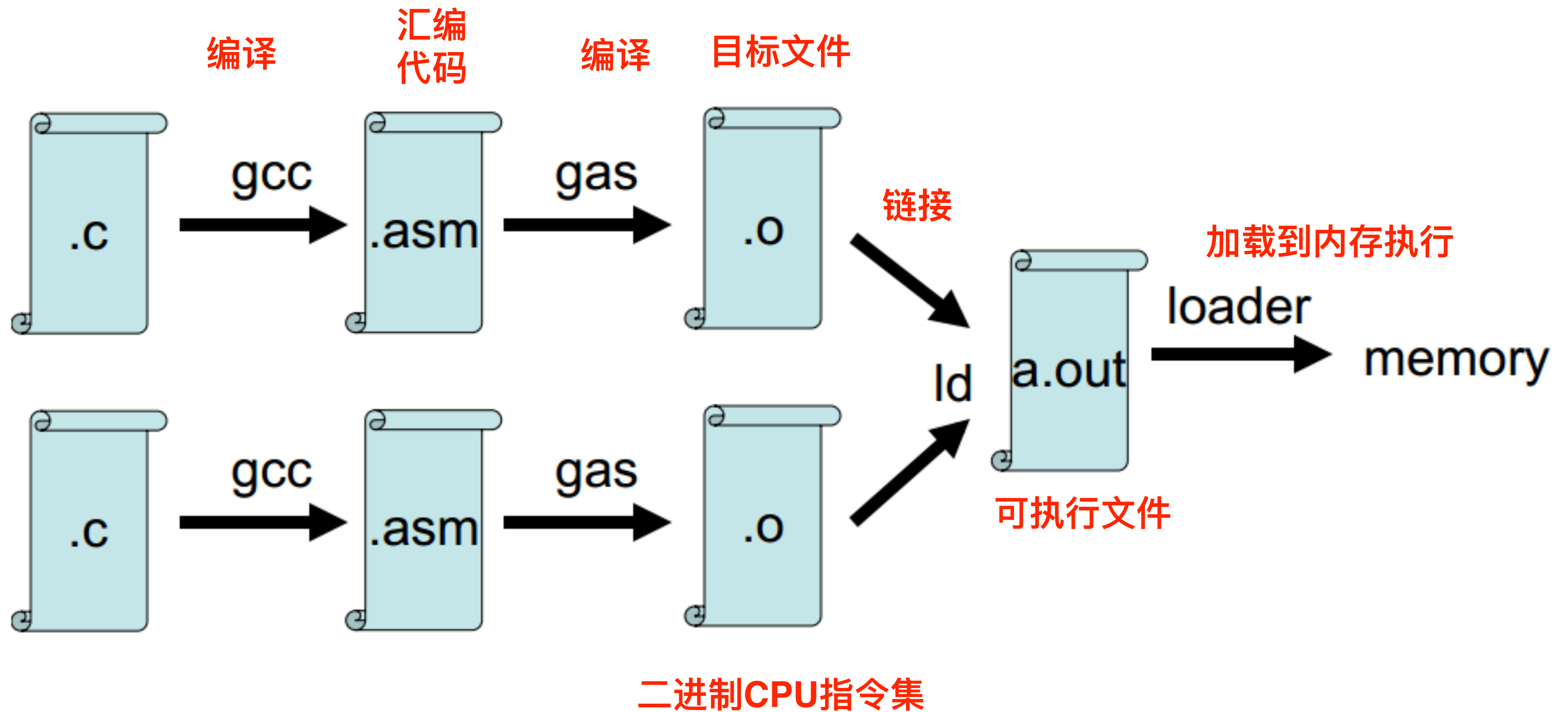


关注孟宁

# GCC

- 在默认情况下ubuntu没有提供c/c++编译环境, ubuntu提供了build-essential包让一次把相关软件安装好
- `$ sudo apt-get install build-essential`
- `$ apt depends build-essential` # 查看哪些包被build-essential依赖用命令

# From C to running program



# gcc用法参考

文本

- `gcc -E -o *.cpp *.c`
- `gcc -x cpp-output -S -o *.s *.cpp`
  - `gcc -S -o *.s *.c`
- `gcc -x assembler -c *.s -o *.o`
  - `gcc -c *.c -o *.o`
  - `as -o *.o *.s`
- `gcc -o * *.o`
  - `gcc -o * *.c`

gcc把上面图的过程全部做了  
.c -> .out

# make & Makefile

当项目很多的时候，gcc就很不方便了，需要makefile

- make是一个命令工具，是一个解释Makefile中指令的命令工具，一般来说，大多数的IDE都有这个命令，比如：Delphi的make，Visual C++的nmake，Linux下GNU的make。
- make命令执行时，需要一个 Makefile文件，以告诉make命令需要怎么样的去编译和链接程序。

# 工程文件的作用

- 如果这个工程没有编译过，那么我们的所有C文件都要编译并被链接。
- 如果这个工程的某几个C文件被修改，那么我们只编译被修改的C文件，并链接目标程序。
- 如果这个工程的头文件被改变了，那么我们需要编译引用了这几个头文件的C文件，并链接目标程序。

# Makefile的规则

- <target> : <prerequisites>
- [tab] <commands> (任意的Shell命令)
- ...
- ...

# make如何工作？

- make会在当前目录下找名字叫“Makefile”或“makefile”的文件
- 如果make命令运行时没有指定目标，默认会执行Makefile文件的第一个目标，第一个目标习惯写为all
- make会一层又一层地去找目标的依赖关系
- 目标的前置依赖都执行完了，执行该目标下的<commands>



# 目标 (target)

- 一个目标 (target) 就构成一条规则。目标通常是文件名，指明make命令所要构建的对象。目标可以是一个文件名，也可以是多个文件名，之间用空格分隔。
- 目标还可以是某个操作的名字，这称为"伪目标" (phony target)，比如clean，如果当前目录中，正好有一个文件叫做clean，那么这个目标不会执行。因为make发现clean文件已经存在，就认为没有必要重新构建了，就不会执行clean"伪目标"。为了避免这种情况，可以明确声明clean是"伪目标"：".PHONY: clean"

# 前置条件 (prerequisites)

- 前置条件通常是一组文件名，之间用空格分隔。它指定了"目标"是否重新构建的判断标准：只要有一个前置文件不存在，或者有过更新（前置文件的last-modification时间戳比目标的时间戳新），"目标"就需要重新构建。

# 命令 (commands)

- 命令 (commands) 表示如何更新目标文件，由一行或多行的Shell命令组成。它是构建“目标”的具体指令，它的运行结果通常就是生成目标文件。
- 每行命令之前必须有一个tab键。如果想用其他键，可以用内置变量.RECIPEPREFIX声明。用.RECIPEPREFIX指定大于号 (>) 替代tab键：“.RECIPEPREFIX = >”
- 需要注意的是，每行命令在一个单独的shell中执行。这些Shell之间没有继承关系。

# Makefile文件的语法

- 井号（#）在Makefile中表示注释。
- 正常情况下，make会打印每条命令，然后再执行，这就叫做回声（echoing）。在命令的前面加上@，就可以关闭回声。
- 通配符（wildcard）用来指定一组符合条件的文件名。Makefile的通配符与 Bash 一致，主要有星号（\*）、问号（?）等。比如，\*.o 表示所有后缀名为o的文件。

# 模式匹配

- make命令允许对文件名，进行类似正则运算的匹配，主要用到的匹配符是%。比如，假定当前目录下有 f1.c 和 f2.c 两个源码文件，需要将它们编译为对应的对象文件：“%.o: %.c” 简写为“.c.o”目标
- 使用匹配符%，可以将大量同类型的文件，只用一条规则就完成构建。

# 变量和赋值

- 使用等号自定义变量，调用时变量需要放在 `$( )` 之中
- `txt = Hello World; echo $(txt)`
- 调用Shell变量，需要在美元符号前，再加一个美元符号，这是因为make命令会对美元符号转义。
- `echo $$HOME`
- 变量的值可能指向另一个变量，如 `v1 = $(v2)`

# 四个赋值运算符（=、:=、?=、+=）

- VARIABLE = value # 在执行时扩展，允许递归扩展。
- VARIABLE := value # 在定义时扩展。
- VARIABLE ?= value # 只有在该变量为空时才设置值。
- VARIABLE += value # 将值追加到变量的尾端。

# 内置变量 (Implicit Variables)

- make命令提供一系列内置变量，主要是为了跨平台的兼容性，
- \$(CC) 指向当前使用的编译器，\$(MAKE) 指向当前使用的make工具



内置变量清单



# 自动变量 (Automatic Variables)

- `$@`指代当前目标，就是Make命令当前构建的那个目标。
- `$<` 指代第一个前置条件。
- `$?` 指代比目标更新的所有前置条件，之间以空格分隔。
- `$^` 指代所有前置条件，之间以空格分隔。
- `$(@D)` 和 `$(@F)` 分别指向 `$@` 的目录名和文件名。
- `$(<D)` 和 `$(<F)` 分别指向 `$<` 的目录名和文件名。



自动变量清单

# 判断和循环

- Makefile使用 Bash 语法，完成判断和循环。

```
ifeq ($(CC), gcc)
    libs=$(libs_for_gcc)
else
    libs=$(normal_libs)
endif
```

# 判断和循环

```
LIST = one two three
```

```
all:
```

```
    for i in $(LIST); do \
```

```
        echo $$i; \
```

```
    done
```

# 等同于

```
all:
```

```
    for i in one two three; do \
```

```
        echo $i; \
```

```
    done
```

# 函数

- Makefile 还可以使用函数，格式如
- `$(function arguments)` 或 `${function arguments}`
- shell函数用来执行 shell 命令，如`$(shell echo src/{00..99}.txt)`



内置函数列表

# Makefile实例

```
.PHONY: cleanall cleanobj cleandiff
```

```
cleanall : cleanobj cleandiff  
          rm program
```

```
cleanobj :  
          rm *.o
```

```
cleandiff :  
          rm *.diff
```

# Makefile实例

```
#
# Makefile for Menu Program
#

CC_PTHREAD_FLAGS      = -lpthread
CC_FLAGS               = -c
CC_OUTPUT_FLAGS        = -o
CC                     = gcc
RM                     = rm
RM_FLAGS               = -f
TARGET = test

OBJS = linktable.o menu.o test.o

all: $(OBJS)
    $(CC) $(CC_OUTPUT_FLAGS) $(TARGET) $(OBJS)

.c.o:
    $(CC) $(CC_FLAGS) $<

clean:
    $(RM) $(RM_FLAGS) $(OBJS) $(TARGET) *.bak
```

# GDB: The GNU Project Debugger

- 开始调试之前，必须用程序中的调试信息编译要调试的程序。这样，gdb 才能够调试所使用的变量、代码行和函数。如果要进行编译，请在 gcc（或 g++）下使用额外的 '-g' 选项来编译程序 `gcc -g hello.c -o hello`
- 在 shell 中，可以使用 'gdb' 命令并指定程序名作为参数来运行 gdb，例如 'gdb hello'；或者在 gdb 中，可以使用 file 命令来装入要调试的程序，例如 'file hello'。这两种方式都假设您是在包含程序的目录中执行命令。装入程序之后，可以用 gdb 命令 'run' 来启动程序。
- 如果一切正常，程序将执行到结束，此时 gdb 将重新获得控制。但如果有任何错误将会怎么样？这种情况下，gdb 会获得控制并中断程序，从而可以让您检查所有事物的状态，如果运气好的话，可以找出原因。

# 使用断点

- 可以在程序代码中的某一特定行或函数中设置断点，这样 gdb 会在遇到断点时中断执行
- `break main`
- `break 21 if value==div`
- `condition 1 value==div/condition 1`（取消条件）
- `info break`
- `next/step`



# 变量跟踪

- info locals
- print arg
- watch arg
- info watch
- set arg=1

# 堆栈跟踪

- 要打印堆栈，发出命令 'bt' ('backtrace' [回溯] 的缩写)
- 'frame' 命令中明确指定号码，或者使用 'up' 命令在堆栈中上移以及 'down' 命令在堆栈中下移来切换帧。要获取有关帧的进一步信息，如它的地址和程序语言，可以使用命令 'info frame'。

# 实验

- `wget https://github.com/mengning/menu/archive/b5b077336ad86ebdb5bf88012a1c44139eaaedc6.zip`
- `unzip b5b077336ad86ebdb5bf88012a1c44139eaaedc6.zip`
- `mv menu-b5b077336ad86ebdb5bf88012a1c44139eaaedc6/ menu`
- `cd menu`
- 以menu程序为例理解Makefile的执行过程以及gdb跟踪调试menu程序  
test