GNU Make 简介

一个 GNU Make 快速导引

Mitch Davis
mjd@afork.com

for SZLUG, 14-Aug-2011

Make 是什么?

Make 是一个构建软件的系统，它知道：

- 构建什么
- 如何构建 (build)

简单规则 + 专家系统 = 强大的功能

如何使用 GNU make?

- 指令放在一个叫 `Makefile` 的文件里
- 只需执行 `make` 即可

Linux: `make`
其它系统：`gmake`（可能需要自己编译安装）

为何学习 / 使用 make?

- 重构建最小化⇒ 节省时间
- 并行编译 ⇒ 使用计算机所有的 CPU 内核
- 理解 / 维护其它的 makefile

哪个版本？

- 经典 make

  太古老 *!*

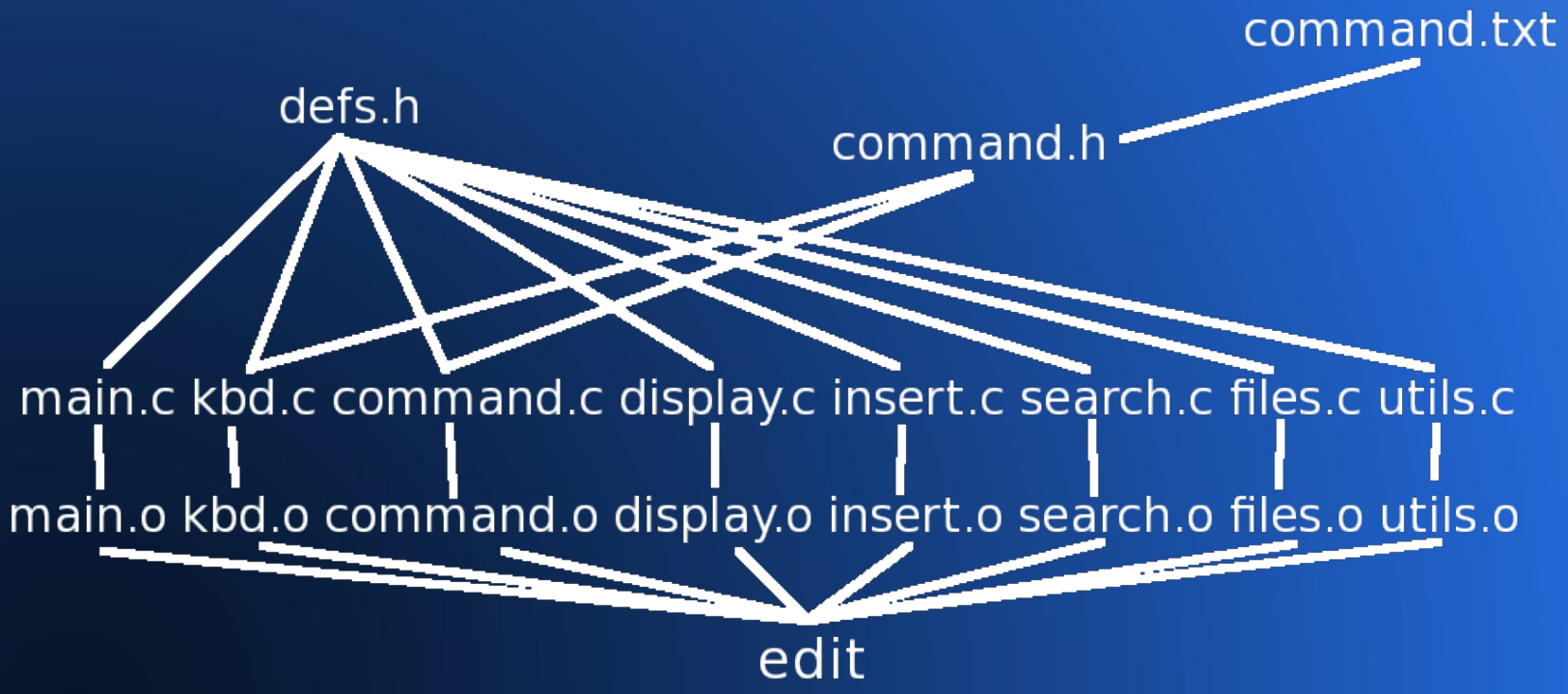
- cmake / scons

  更新，更好，但还未被广泛应用

- GNU make (gmake)

  极好！

一个实际案例


command.txt
defs.h
command.h
main.c kbd.c command.c display.c insert.c search.c files.c utils.c
main.o kbd.o command.o display.o insert.o search.o files.o utils.o
edit

基本的 **makefile** 组件

```
target: prerequisites
   commands
```

target: prerequisites 为规则

commands *是动作*

Target：目标

Prerequisites: 前件

Makefile 规则

- 何时需要重构软件的规则
- 如果必要前件变更，目标必须重构建
- Make 使用时间戳
- 重构的条件：
  - 必要前件比目标更新，或者
  - 目标缺失

Makefile 动作

- 动作是指如何构建某个目标。
- 动作行必须以 Tab 字符开头
- 第一条规则是默认规则
- 通常要有一个叫 `all` 的默认规则

一个简单的 makefile

```
edit : main.o kbd.o command.o display.o \
            insert.o search.o files.o utils.o
   cc -o edit main.o kbd.o command.o display.o \
                  insert.o search.o files.o utils.o

main.o : main.c defs.h
   cc -c main.c
kbd.o : kbd.c defs.h command.h
   cc -c kbd.c
command.o : command.c defs.h command.h
   cc -c command.c
display.o : display.c defs.h buffer.h
   cc -c display.c
insert.o : insert.c defs.h buffer.h
   cc -c insert.c
search.o : search.c defs.h buffer.h
   cc -c search.c
files.o : files.c defs.h buffer.h command.h
   cc -c files.c
utils.o : utils.c defs.h
   cc -c utils.c
```

这是一个非常基本的 makefile，还可进一步改进！

变量

- Simply expanded variables: `VAR:=`
  Evaluated when defined
  ```
  DATE:=$(shell date); cc -DDATE="$(DATE)"
  ```
- Recursively expanded variables: `VAR=`
  Evaluated when used
  ```
  CFLAGS=-O2 -g $(OTHER)
  OTHER += -fpic
  ```
- 附加：`VAR+=`
- 引用变量：`$(VAR)`
- 替换：`SRCS=foo.c bar.c; $(SRCS:.c=.o)`

变量实例

比如说变量 OBJECTS 包含一个目标文件列表：

```
objects = program.o foo.o utils.o
program: $(objects)
  cc -o program $(objects)

$(objects): defs.h
```

模式规则

我们已见过显式规则：

```
insert.o : insert.c defs.h buffer.h
```

*模式*规则指明如何构建一类文件 .

以一个 % (“ 主干”) 作为公共部分

实例：

```
%.o: %.c
```

⇒ 无需列出每个规则的动作

动作再谈

动作是为了构建一些东西所需的一个或多个欲运行的命令。

传递到 shell.

可以使用 make 变量，如 `mkdir -p $(BINDIR)`

常见陷阱：

- Shell 的状态（例如环境变量）在行间不共用
- 使用 `$$` 代替 `$`

自动变量

模式规则中常常用到自动变量

```
%.o: %.c
   cc -o $@ -c $<
```

下面是一些自动变量

$@	目标的名字
$<	第一个变更的前件的名字
*$^*	所有前件的名字

更好的 makefile

```
OBJECTS = main.o kbd.o command.o display.o insert.o search.o files.o utils.o

all: edit

edit: $(OBJECTS)
   cc -o $@ $^

%.o: %.c
   cc -o $@ -c $<

$(OBJECTS): defs.h
kbd.o: command.h
command.o: command.h
display.o: buffer.h
insert.o: buffer.h
search.o: buffer.h
files.o: buffer.h command.h
```

函数 (1)

Make 有很多函数。下面是一些有用的函数：

- info/error
```
$(info CFLAGS=$(CFLAGS))
```
- patsubst
```
$(patsubst $(SRCDIR)/%.c,$(BINDIR)/%.o,$(SRCS))
```
- addprefix/addsuffix
```
CFLAGS += $(addprefix -L,$(INCDIRS))
```
- filter/sort
```
SRCS_c := $(sort $(filter %.c,$(SRCS)))
```

函数 (2)

更多的常用函数：

- dir / notdir / basename

```
RESOLV=/etc/resolv.conf
$(info $(dir $(RESOLV)) ⇒ /etc/
$(info $(notdir $(RESOLV)) ⇒ resolv.conf
$(info $(basename $(RESOLV)) ⇒ resolv
```

- shell
- Wildcard

```
SRCS_c := $(wildcard *.c)
```

```
foreach
  dirs := a b c d
  files := $(foreach dir,$(dirs),$(wildcard $(dir)/*))
```

make 不能干什么

Make 的计算单元是文件，因此对于非文件目标，make 将无效：

- 非本地目标（例如远程网站上的二进制固件）

Make 的规则是一次执行一个命令重构一个目标：

- 这个对 Java 有问题，因为 Java 是针对某个目录来构建的，而 make 是针对每个文件。
- Make cannot make coffee :-(

高阶 make:
自动化的 C/C++ 依赖生成

即使有模式规则，仍然需要表达 .o 和 .h 文件之间的关系。

一条使编译器为目标自动生成规则的规则。

```
cc -M main.c
```

⇒ main.o : main.c defs.h ⇒ .d 临时文件。

修改 makefile 以包含 .d 文件

然后就不必指明 .c 文件包含了哪些 .h 文件！

高阶 **Make**：并行编译

并行构建：`-j`*n*

推荐：使用 n = CPU 个数 + 1

计算变量 (Computed variables)

```
CONFIG_foo=y
 ⋮
OBJECTS_$(CONFIG_FOO) += foo.o
 ⋮
target: $(OBJECTS_y)
```

在 Linux kernel 中很常见。

高阶 make: 库与抽象

- GNU make 可包含别的文件

```
PROGRAMS    = server client

server_OBJS = server.o server_priv.o server_access.o
server_LIBS = priv protocol
client_OBJS = client.o client_api.o client_mem.o
client_LIBS = protocol

include my_make_library
```

高阶 make：递归 make

对于那些有多个目录的项目，make 能递归遍历目录树：

```
for dir in lib webclient cmdline; do $(MAKE) -C $$dir; done
```

高阶 make: 非递归 make!

- 递归 make 效率很低，即使什么都不用干它也必须遍历整个目录树。
- 可以用包含的方法来达到递归 make 效果，而不用 submakes.
- 如 Linux kernel 中使用的方法
- Recursive Make Considered Harmful
  http://miller.emu.id.au/pmiller/books/rmch/

高阶 make: GNU autotools

- `configure.ac` 生成:
- `configure.in` 又生成:
- `configure` 使用:
- `Makefile.in` 来生成
- `Makefile`

高阶 **make**：非程序目标

- 当你需要对多个文件执行某个操作的时候这将非常有用，例如压缩文件。
- Make a target which is the list of processed files. `($wildcard)` 函数在这里将很有用。
- 执行 `make -j`*`n`*
- 去喝杯茶，回来时任务就完成了！

更多信息

- Make 的 info page: `info make`
- GNU Make 标准库

  http://gmsl.sourceforge.net/
- Google!

谢谢

- 谢谢社区给我演讲的机会。
- 谢谢 Atommann 的翻译
- 谢谢大家听我的演讲！