実践 make

総合演習 B

神戸大学

陰山

データの依存関係

- ファイルAを更新する
- ファイルAを入力として処理B、処理Cをする
- ファイルAが新たに変更されたら、また処理B、処理Cをする
- ただし、ファイルAが変更されない限り、処理B、処理Cの結果は変わらない
  - 例：プログラムのコンパイル
    ソースコード　→　実行ファイル

データの依存関係

- 現実のプロジェクトは多数のファイルが複雑な依存関係をもつ
- 例：ZがYに依存し、YがXに依存している（X → Y → Z）とき
  - Xが更新されたらYとZを更新すべき
  - Yが更新されたらZを更新するべき
  - Yは常にXよりも新しく、Zは常にYよりも新しい
- ファイルの更新時刻を比較することで自動化できるはず
- その自動化をするのがmakeコマンドである。依存関係はMakefileというファイルに記述する
- Makefileには依存関係だけでなく、「Xから如何にしてYを作るか」というルールも記述する

makeコマンドとは

- makeコマンドはMakefileというファイルを読む
  - makefile という名前でもよい
  - make -f 任意ファイル名　としてもよい
- Makefileはmakeのルールを記述
  - ルールは複数可
- 最初のルールがデフォルトルール

Makefileの基本ルール

```
target: prereq1 prereq2
	command
```

Makefile

```
target: prereq1 prereq2
	command
```

ここは TAB （スペースではない）

Makefileの解説

- Makefileのルールは以下の三つからなる
  - ターゲット target
  - 依存項目（必須項目） prereq
  - 実行コマンド command

```
target: prereq1 prereq2
        command
```

ここは TAB （スペースではない）

Makefileの解説

- 依存項目はターゲットが作成される前に存在するべきもの
- 実行コマンドは依存項目からターゲットを作るためのもの
- ターゲットと依存項目はコロンで区切る

Makefileの文法

```
target1 target2 target3: prereq1 prereq2 prereq3
	command1
	command2
	command3
```

擬似ターゲット

- make clean とすれば不要なファイルを削除
- だがcleanというファイルがあるとおかしなことになる

```
collatz: collatz.c
	gcc collatz.c -o collatz

clean:
	rm -f collatz collatz.o
```

擬似ターゲット

- Makefileに.PHONY: clean とかく
- 擬似ターゲット
- make cleanとすれば必ず実行

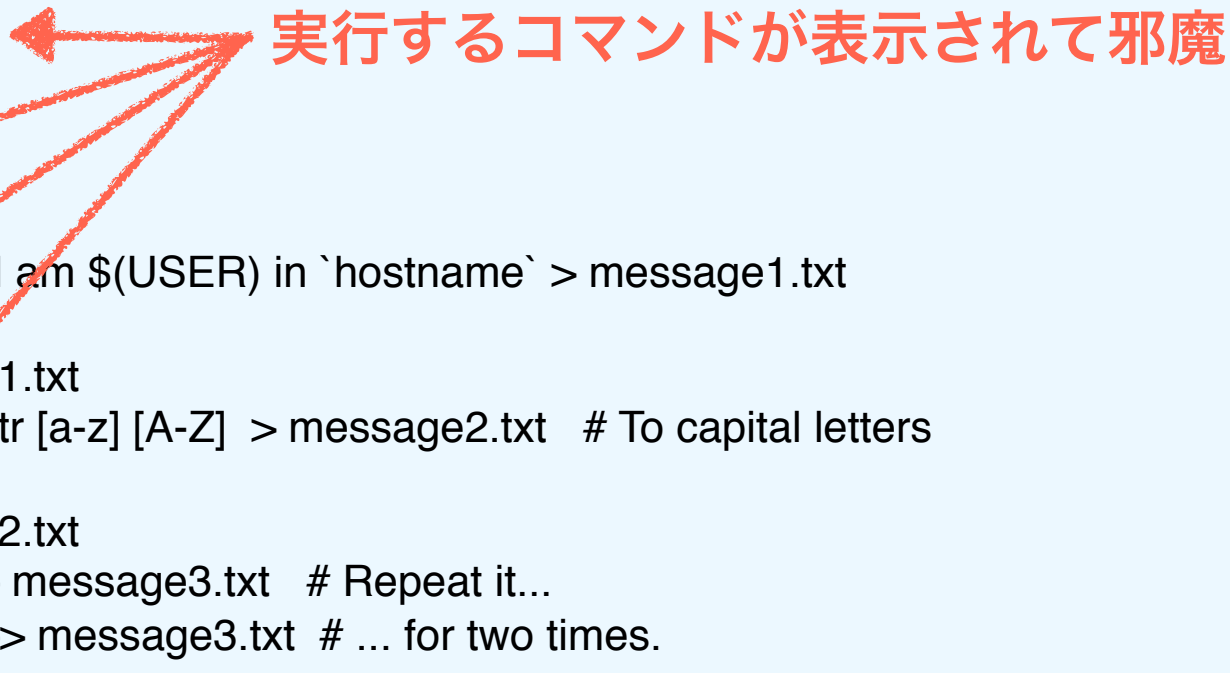
```
.PHONY: clean

collatz: collatz.c
	gcc collatz.c -o collatz

clean:
	rm -f collatz collatz.o
```

よく使う擬似ターゲット

all	アプリケーションを構築する全ての作業を行う
install	コンパイル済のバイナリをインストールする
clean	ソースから作られたバイナリを削除する
check	このアプリケーションに関連する全てのテストを実行する
info	GNU infoファイルを作成する

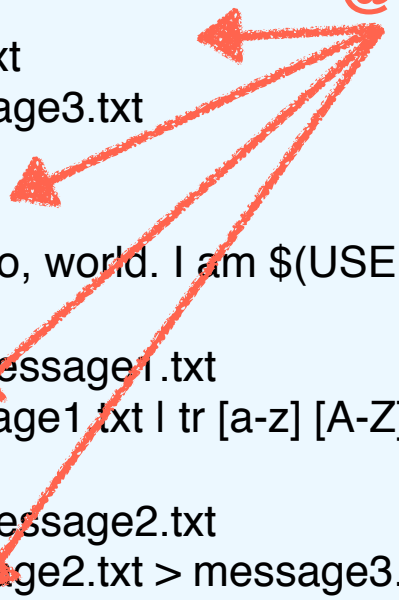
実行コマンドの非表示

```
.PHONY: run clean

run: message3.txt
	cat message3.txt

message1.txt:
	echo Hello, world. I am $(USER) in `hostname` > message1.txt

message2.txt: message1.txt
	cat message1.txt | tr [a-z] [A-Z]  > message2.txt   # To capital letters

message3.txt: message2.txt
	cat message2.txt > message3.txt   # Repeat it...
	cat message2.txt >> message3.txt  # ... for two times.

clean:
	rm -f message1.txt message2.txt message3.txt
	rm -f message1.txt~ message2.txt~ message3.txt~
```



実行コマンドの非表示

```
.PHONY: run clean

run: message3.txt
	@cat message3.txt

message1.txt:
	@echo Hello, world. I am $(USER) in `hostname` > message1.txt

message2.txt: message1.txt
	@cat message1.txt | tr [a-z] [A-Z]  > message2.txt   # To capital letters

message3.txt: message2.txt
	@cat message2.txt > message3.txt   # Repeat it...
	@cat message2.txt >> message3.txt  # ... for two times.

clean:
	@rm -f message1.txt message2.txt message3.txt
	@rm -f message1.txt~ message2.txt~ message3.txt~
```



@をつけると実行コマンドが表示されなくなる

変数

- 変数定義の仕方は2種類

  変数 = 値

  変数 := 値

- 例

  CC := gc

  source = *.c

変数

- 違い

  変数 = 値

  - 遅延評価（変数を使用するときに右辺を評価）

  変数 := 値

  - 即時評価（この行が読み込まれたときに右辺評価）

```
COMPILE = $(CC) -g
  .
  .
  .
# 後で
CC = gcc
  .
  .
  .
# gcc -g と評価される
$(COMPILE) test.c
```

変数名の慣習

- 環境変数やmakeの引数で値を変更（上書き）する可能性のあるものは大文字

  例: CC

- そのMakefileだけで使われるものは小文字

  例: source = *.c

変数の展開（参照）

- $(変数) または ${変数}
- 一文字の変数なら()や{}は不要　⇒　自動変数も
- ${}は古い。$()の方が新しい

自動変数

- $@

  ターゲットのファイル名

- $<

  依存項目の最初のファイル名

- $^

  全ての依存項目をスペースで区切ったリスト。重複は除かれる

- $+

  全ての依存項目をスペースで区切ったリスト。重複は除かれない

```
collatz: collatz.c collatz.h
```

のとき、

$@ = collatz
$< = collatz.c
$^ = collatz.c collatz.h

パターンルール

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

- %はワイルドカード（UNIXシェルの * と同じ）
- any.c というファイルがあればコンパイルして any.o を作る

パターンルール

```
.PHONY: clean all

all: collatz01 collatz02 collatz03 collatz04 collatz05

collatz01: collatz01.c collatz.h
  gcc collatz01.c -o collatz01

collatz02: collatz02.c collatz.h
  gcc collatz02.c -o collatz02

collatz03: collatz03.c collatz.h
  gcc collatz03.c -o collatz03

collatz04: collatz04.c collatz.h
  gcc collatz04.c -o collatz04

collatz05: collatz05.c collatz.h
  gcc collatz05.c -o collatz05

clean:
  rm -f collatz0? collatz0?.o *~
```

- ソースコードがたくさんあるとルールを記述するのが大変
  - バグも入りやすくなる
- ⇒ パターンルール

パターンルール

```
.PHONY: clean all

all: collatz01 collatz02 collatz03 collatz04 collatz05


%: %.c collatz.h
  gcc $< -o $@


clean:
  rm -f collatz0? collatz0?.o *~
```

14行が6行になった

暗黙のルール

実はこれだけでも十分

```
.PHONY: clean all

all: collatz01 collatz02 collatz03 collatz04 collatz05

clean:
  rm -f collatz0? collatz0?.o *~
```

よくある言語には「暗黙のルール」が決められている。
今の場合は以下のルール

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

暗黙のルールを調べるには `make --print-data-base`

コマンドライン変数

```
make CFLAGS=-g CPPFLAGS='-DDEBUG'
```

- Makefile中の二つの変数CFLAGSとCPPFLAGSは上書きされる
- Makefile中での定義を優先させるにはoverride命名を使う

```
override CFLAGS=-g
```

アペンド代入

- 意味は明らかであろう

```
FLAGS += -DDEBUG
```

- 単純代入変数なら自明

  FLAGS := 追加

  - FFLAGS := $(FLAGS) + 追加　　ということ

- 遅延評価変数の場合は？

  FLAGS = 追加

  - FLAGS = $(FLAGS) + 追加
    無限ループ？　　⇒　うまく処理してくれる

条件判断

if条件分の種類

```
ifdef variable-name
ifndef variable-name
ifeq test
ifneq test
```

```
# 例: COMSPECはWindowsにおいてのみ設定されている
ifdef COMSPEC
  PATH_SEP := ;
  EXE_EXT := .exe
else
  PATH_SEP := :
  EXE_EXT :=
endif
```

組み込み関数　filter

関数の一般形　$(function-name arg1[, argn])

文字列関数　$(filter pattern…, text)

Makefileのサンプル

```
words := he the hen other
get-the:
  @echo he matches: $(filter he, $(words))
  @echo %he matches: $(filter %he, $(words))
  @echo he% matches: $(filter he%, $(words))
  @echo %he% matches: $(filter %he%, $(words))
```

%はワイルドカード

実行結果

```
he matches: he
%he matches: he the
he% matches: he hen
%he% matches:
```

ワイルドカードの%は最初の一つだけ（以降は%文字そのもの）

組み込み関数 filter-out

```
filter-out関数  $(filter-out pattern…, text)
```

パターンに一致しなかった単語を取り出す

例: Cのヘッダファイル以外を取り出す

Makefile

```
all_source := count_words.c counter.c lexer.l counter.h lexer.h
to_compile := $(filter-out %.h, $(all_source))

test:
  @echo all_source = $(all_source)
  @echo to_compile = $(to_compile)
```

実行結果

```
all_source = count_words.c counter.c lexer.l counter.h lexer.h
to_compile = count_words.c counter.c lexer.l
```

組み込み関数 sort

sort関数　$(sort list)　　**list の文字列を sortする**

Makefile

```
words_before := cdef defgh bcd zzz efghi abc

words_after := $(sort $(words_before))

test:
  @echo words_before = $(words_before)
  @echo words_after = $(words_after)
```

実行結果

```
words_before = cdef defgh bcd zzz efghi abc
words_after = abc bcd cdef defgh efghi zzz
```

組み込み関数 shell

sort関数　$(shell command)

サブシェルでcommandを実行する

Makefile

```
c_source_files := $(shell ls -1 *.c)
today := $(shell date "+%Y_%m_%d")

test:
  @echo c_source_files = $(c_source_files)
  @echo today = $(today)
```

実行結果

```
c_source_files = collatz01.c collatz02.c collatz03.c collatz04.c
collatz05.c
today = 2018_11_04
```

組み込み関数 dirとnotdir

`$(dir list)`	listにある各単語のディレクトリ部分を返す
`$(notdir list)`	listにある各単語のファイル名部分を返す

Makefile

```
c_source_files := $(shell ls -1 *.c)
today := $(shell date "+%Y_%m_%d")

test:
  @echo c_source_files = $(c_source_files)
  @echo today = $(today)
```

実行結果

```
c_source_files = collatz01.c collatz02.c collatz03.c collatz04.c
collatz05.c
today = 2018_11_04
```

サフィックスルール

```
.c.o:
	$(COMPILE) $(OPTION) $<
```

- 古いやり方
- パタールルート同じ
- 今ならパタールルールを使うことを推奨
- any.c というファイルがあればコンパイルして any.o を作る
- 既定のサフィックスルールを無効にする方法

```
.SUFFIXES:
```

VPATHとvpath

- これまではMakefileと全てのソースが同じディリクトリにある場合
- VPATH変数はソースコードを探すディレクトリのリスト

  例: VPATH = src

- vpath 命令でもOK（こちらの方が便利）

  例:

  vpath %.c src

  vpath %.l src

  vpath %h include

Makefileの注意

- コマンド行はサブシェルで実行される
- サブシェルの例:

```
#!/bin/sh

echo My process id and shell level = $$ $SHLVL

if [ $SHLVL -gt 3  ]
then
  exit
fi

sh ./sub_shell_test.sh
```

並列make

- 複数プロセスで処理
- make --jobs=2 （またはmake -j 2）とすると可能なら2個のターゲットを同時に更新する
- 注意: 2プロセスというわけではない（実際には2プロセス以上が立ち上がる）
- make -j とするとできるだけ多くの更新作業をするが、むしろ遅くなる場合があるので注意

makeのデバッグ

- make --just-print

  実行するコマンドを表示するが実行はしない

- make --print-data-base （またはmake -p）

  内部データベースを全て表示

- make --debug

  デバッグ用情報の表示