GNU Make 数据生产线

续本达

清华大学 工程物理系

2024-07-14 清华

软件准备：带 Scheme 语言扩展的 GNU Make

- Debian

```
# Debian
apt install make-guile make-doc guile-3.0-doc
```

命令行

- 命令行操作系统外壳 shell 的一种，介于内核 kernel 与用户程序之间。
- 最佳工具：调度用户程序、管理文件。
- 研发门槛低：输入命令的过程中，就随手写了新程序。
- bash GNU Bourne-Again SHell 提供最流行的强大命令行环境
  - 五类命令：二进制程序、可执行脚本、函数、内建命令、别名
  - 标准输入输出、管道
  - 脚本 `#!/usr/bin/env bash` 开头，加上若干命令
  - 变量 `$A`, `${A}`
  - 引用

转译	单引号	双引号
空格	x	x
$	x	o

学堂路车神

- 判断题：学神还是菜鸡？
- 因害怕关闭机盖程序中断

当事人

- “骑车带电脑是因为程序没跑完，害怕关闭机盖后导致程序中断，三四个小时重新来。”
- “我没有也从未试图在骑车的过程中做出敲击键盘，编辑程序，阅读程序这类高难度行为，只是托着电脑避免机盖关闭。”
- “至于为什么没想到其他方法保持程序运行，是因为我确实不知道，实在是很菜了……”

学堂路车神

- 判断题：学神还是菜鸡？
- 因害怕关闭机盖程序中断

当事人

- “骑车带电脑是因为程序没跑完，害怕关闭机盖后导致程序中断，三四个小时重新来。”
- “我没有也从未试图在骑车的过程中做出敲击键盘，编辑程序，阅读程序这类高难度行为，只是托着电脑避免机盖关闭。”
- “至于为什么没想到其他方法保持程序运行，是因为我确实不知道，实在是很菜了……”

数据流水线的构造目标

复现 原则的要求

要记录下来以什么样的顺序和参数运行什么命令，执行什么程序。

思路和要点

- 把流程系统化成输入、输出与过程三要素。
  - 而向数据编程，data-driven programming
- 系统表达输入数据、输出数据和中间结果的依赖关系，
  - 成为“可执行的说明文档”
- 高效执行，包括并行处理和整合超级计算机等。
- 错误恢复
  - 修正错误后，可以从最后一步正确的数据开始继续执行。

数据流水线的构造目标

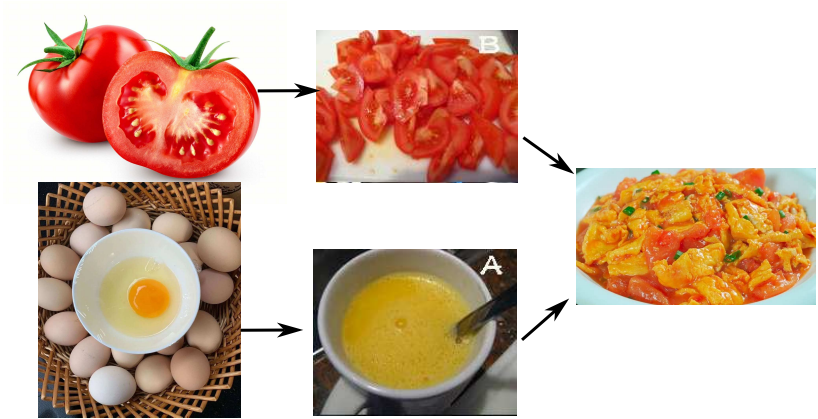
复现 原则的要求

要记录下来以什么样的顺序和参数运行什么命令，执行什么程序。

思路和要点

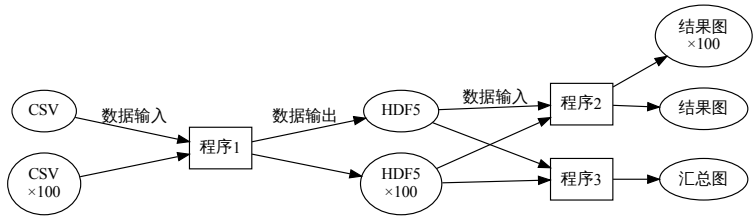
- 把流程系统化成输入、输出与过程三要素。
  - 而向数据编程，data-driven programming
- 系统表达输入数据、输出数据和中间结果的依赖关系，
  - 成为“可执行的说明文档”
- 高效执行，包括并行处理和整合超级计算机等。
- 错误恢复
  - 修正错误后，可以从最后一步正确的数据开始继续执行。

西红柿炒蛋如何做？



- 任务的执行有前后的依赖顺序。

批量处理海量数据



命令行探索之后，要将数据处理方法自动化

- 实验要调用很多命令和程序
  - 重复运行，控制变量：以不同条件多次测量，探索规律
- 处理很多数据，有很多中间结果，依赖关系复杂
- 程序有更新怎么办？数据有更新怎么办？

Make 是最佳工具

- make 工具已经有 40 多年的历史，最初用来管理 C 语言程序的编译。
  - 根据依赖关系决定命令执行顺序
- GNU make 是 GNU 运动中，对 make 进行的扩展，更适合管理数据
  - 与 BSD make 有区别，在 macOS 环境里请配置 GNU 环境，使用 gmake 。
  - 版本 ≥ 4.3 支持 多目标规则

```
make --version
```

```
GNU Make 4.3
Built for x86_64-pc-linux-gnu
Copyright (C) 1988-2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

作用

1. 实现 复现 要求
2. 管理程序运行，在超级计算机上运行
3. 从错误中恢复

Make 要素

- 文档

```
info make
man make
```

- 基本语法单元：清晰写明输入数据，输出数据和计算方法

```
target: source
        program source target # 必须 TAB 起始，记录如何做
```

- `target` 和 `source` 都应对应于当前路径下的文件名。
  - 当 `source` 存在时，可以通过运行 `program source target` 生成文件 `target`

参考书

- John Graham Cumming, The GNU Make Book
- 陈皓，跟我一起写 Makefile

变量取值

```
a:=1 # eager evaluation (= 是 lazy evaluation)
$(info $(a))
```

- 调用时使用 $( ) ， $(info TEXT) 输出 TEXT
- 一次 原则，避免重复
  - 特殊变量： $^ 输入数据、 $@ 输出数据。
  - 目标与源都以文件表示，引用 $^ $@ 可有效减少重复。

```
target: source
        program source target #如何做
        program $^ $@ #如何做
```

```
target: source1 source2 source3
        program $< --reference $(word 2,$^) \
                --location $(word 3,$^) -o $@
```

Make 的函数

- 函数调用 `$(func argument1,argument2,...)`
  - word 函数 `$(word N,TEXT)`

```
$(word 2,make.c use.c high.c) # => use.c
```

  - patsubst 函数 `$(patsubst PATTERN,REPLACEMENT,TEXT)`

```
$(patsubst %.c,%.f,make.c use.c high.c) # => make.f use.f high.f
```

  - 更多的函数数在 `info make` 指南中查询（Emacs 下为 `C-h i`）

生成含有 1 到 100 数字的文件

- 创建 Makefile 文件

```
numbers.csv:
	seq 100 > $@
```

- 到 Makefile 所在的文件夹中执行

```
make numbers.csv
```

```
seq 100 > numbers.csv
```

```
cat numbers.csv | wc -l
```

```
100
```

不做无谓重复

```
numbers.csv:
	seq 100 > $@
```

再执行时， make 会判断是否还有必要

```
make numbers.csv
```

```
make: 'numbers.csv' is up to date.
```

判断准则

1. numbers.csv 已经存在
2. numbers.csv 比它依赖的文件都新。此例中无依赖文件，存在即是最新。

加入求和规则

```
total: numbers.csv
	paste -s -d + < $^ | bc > $@
```

```
make total
```

```
paste -s -d + < numbers.csv | bc > total
```

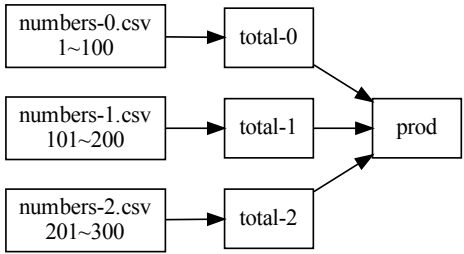
- 如果第一步 numbers.csv 不存在， make 会自动生成它

```
rm total numbers.csv
make total
```

```
seq 100 > numbers.csv
paste -s -d + < numbers.csv | bc > total
```

- 一次原则：只必要地写两处 numbers.csv 防止更新时造成不一致

进阶：暗含循环




```
numbers-0.csv:
        seq 100 > $@
numbers-1.csv:
        seq 101 200 > $@
numbers-2.csv:
        seq 201 300 > $@
total-0: numbers-0.csv
        paste -s -d + < $^ | bc > $@
total-1: numbers-1.csv
        paste -s -d + < $^ | bc > $@
total-2: numbers-2.csv
        paste -s -d + < $^ | bc > $@
prod: total-0 total-1 total-2
        paste -d '*' $^ | bc > $@
```

- 召唤一次原则！
- 循环程序结构是共通的。

一般匹配关系 pattern

- 处理 *.h5 文件，生成对应文件名的 s/.h5/.png 图形。

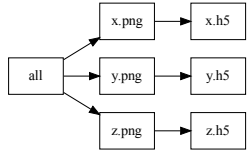
```
filelist:=x.h5 y.h5 z.h5
.PHONY: all
all: $(filelist:.h5=.png)

%.png: %.h5
	./plot-celestial.py $^ -o $@
```

说明

- % 是 GNU Make 的通配符，用于替代任意字符串
- $(filelist:%.h5=%.png) 是 $(patsubst P,R,TEXT) 的简写。
  - 意为在 $(filelist) 中把符合 P (%.h5) 的部分替换成 R (%.png)
  - 有共同前缀时，可进一步简写为
    $(filelist:.h5=.png)
- 在执行规则中， %.png: %.h5 定义任意 png 都由相应的 h5 文件生成。

all 是一个总括目标

```
filelist:=x.h5 y.h5 z.h5
.PHONY: all
all: $(filelist:%.h5=%.png)
```



特殊目标

- .PHONY 代表无对应文件的目标
  - 否则如果存在一个名为 all 那么 make all 就不会执行
  - GNU Make 默认执行第一个目标，可以把它定义为无对应文件的 all
- .DELETE_ON_ERROR 如果出错就把坏文件删掉
- .SECONDARY 保留中间结果

```
.DELETE_ON_ERROR:
.SECONDARY:
```

新工具：guile

- make-guile 在 GNU make 的基础上嵌入了 GNU guile [gʌil] 解释器支持 scheme 语言。
- scheme 语言是 LISP 语言的一支，是历史悠久的人工智能语言。
  - LISP 与 fortran 是最早的高级编程语言，都出现在 1960 年代。
- 算术练习

```
(+ 1 1)
```

2

```
(* 100 (+ 1 1))
```

200

- 手册 info guile

函数式编程：把目标描述清楚

- Makefile 是函数式编程 Functional Programming 的语言。
- 不再关注“执行什么操作”，而是关注“输入到输出的映射”。
  - 类比：Python 的 `map()`， 迭代器

```
# 生成 numbers-N.csv, 包含 N*100+1 到 (N+1)*100 的数字
numbers-%.csv:
        seq $(guile (+ 1 (* $* 100))) $(guile (* 100 (+ $* 1))) > $@
# 分别求和
total-%: numbers-%.csv
        paste -s -d + < $^ | bc > $@
# 求积
prod: total-0 total-1 total-2
        paste -d* $^ | bc > $@
```

- 循环从 `prod: total-0 total-1 total-2` 构造，由数据驱动

数据驱动暗含循环

```
make prod
```

```
seq 1 100 > numbers-0.csv
paste -s -d + < numbers-0.csv | bc > total-0
seq 101 200 > numbers-1.csv
paste -s -d + < numbers-1.csv | bc > total-1
seq 201 300 > numbers-2.csv
paste -s -d + < numbers-2.csv | bc > total-2
cat total-0 total-1 total-2 | paste -s -d '*' | bc > prod
rm numbers-0.csv numbers-2.csv numbers-1.csv
```

- 可以自动补上失去的文件，只执行需要的部分

```
rm total-2
make prod
```

```
seq 201 300 > numbers-2.csv
paste -s -d + < numbers-2.csv | bc > total-2
cat total-0 total-1 total-2 | paste -s -d '*' | bc > prod
rm numbers-2.csv
```

可以用 shell 命令替换 guile

```
# 调用 shell 命令方案
numbers-%.csv:
	seq $(shell echo "$* * 100 + 1" | bc ) $(shell echo "($* + 1) * 100" | bc ) > $@
```

但是不如 guile 简明

```
numbers-%.csv:
	seq $(guile (+ 1 (* $* 100))) $(guile (* 100 (+ $* 1))) > $@
```

调试

- Makefile 的调试器？ `remake` 可以试验。
- 一般调试使用 `$(info )` 查看中间变量和结果
- 不要把文件写得太长，太长时分成小文件，分成小单元调试
- 多写多 bug 少写少 bug，不写无 bug
- 推论：以最简洁的代码实现目的和四个原则
- 把不同功能分放在不同文件中

```
include names.mk
include process.mk

.PHONY: all
all: step1 step2
```

深入理解 make 的执行机理：两个阶段

```
file=numbers
$(file)-%.csv:
        seq $(shell echo "$* * 100 + 1" | bc ) $(guile (* 100 (+ $* 1))) > $@
```

替换阶段

把所有的 $(...) 都进行替换：变量换成它的值，函数执行后换成返回值。
当执行 ‘make numbers-1 numbers-2‘ 时，

```
file=numbers
numbers-1.csv: # %=1, $*=1, $@=numbers-1.csv
        seq 101 200 > numbers-1.csv
numbers-2.csv: # %=2, $*=2, $@=numbers-2.csv
        seq 201 300 > numbers-2.csv
```

深入理解 make 的执行机理：第二阶段

```
file=numbers
$(file)-%.csv:
        seq $(shell echo "$* * 100 + 1" | bc ) $(guile (* 100 (+ $* 1))) > $@
```

执行阶段

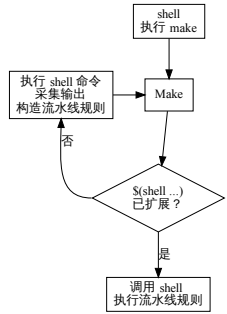
根据每一个要生成的目标， numbers-1.csv numbers-2.csv ，先生成它的依赖，再执行对应的命令

```
seq 101 200 > numbers-1.csv
seq 201 300 > numbers-2.csv
```

Make 与 shell 的调用关系

```
numbers-%.csv:
        seq $(shell echo "$* * 100 + 1" | bc ) $(shell echo "($* + 1) * 100" | bc ) > $@
```

- make 在 shell 中执行，make 通过 shell 执行流水线规则和取值



赋值

- “:=”是 eager evaluation，在被调用时，就把运算做完。
- “=”是 lazy evaluation，在被使用时，才做运算。
  - 不同时节运算，可能有不同变量环境，可到不同的效果

继续深化落实一次原则

- Makefile 里可以调用 shell 命令，将其标准输出作为值。

```
sequence:=$(shell seq -w 00 99)
file_list:=$(shell find . -name "*.h5")
```

- 构造 `total-0 total-1 total-2`

```
seqs:=$(shell seq 0 2)
totals:=$(seqs:%=total-%)
# 另一种方法
totals:=$(addprefix total-,$(seqs))
```

内嵌 scheme

- 最佳工具原则： `$(shell )` 和 `$(guile )` 哪个简洁用哪个。 `iota` [ʌɪˈəʊtə]

```
seqs:=$(guile (iota 3))
```

再次尝试

- 直接来 10 个

```
seqs:=$(guile (iota 10))
totals:=$(seqs:%=total-%)

# 生成 numbers-N.csv, 包含 N*100+1 到 (N+1)*100 的数字
numbers-%.csv:
        seq $(guile (+ 1 (* $* 100))) $(guile (* 100 (+ $* 1))) > $@
# 分别求和
total-%: numbers-%.csv
        paste -s -d + < $^ | bc > $@
# 求积
prod: $(totals)
        paste -d '*' $^ | bc > $@
```

运行

```
make prod
```

```
seq 301 400 > numbers-3.csv
paste -s -d + < numbers-3.csv | bc > total-3
seq 401 500 > numbers-4.csv
paste -s -d + < numbers-4.csv | bc > total-4
seq 501 600 > numbers-5.csv
paste -s -d + < numbers-5.csv | bc > total-5
seq 601 700 > numbers-6.csv
paste -s -d + < numbers-6.csv | bc > total-6
seq 701 800 > numbers-7.csv
paste -s -d + < numbers-7.csv | bc > total-7
seq 801 900 > numbers-8.csv
paste -s -d + < numbers-8.csv | bc > total-8
seq 901 1000 > numbers-9.csv
paste -s -d + < numbers-9.csv | bc > total-9
cat total-0 total-1 total-2 total-3 total-4 total-5 total-6 total-7 total-8 total-9 | paste -
s -d '*' | bc > prod
rm numbers-8.csv numbers-3.csv numbers-7.csv numbers-5.csv numbers-6.csv numbers-9.csv number
4.csv
```

选择结构

```
a:=$(if CONDITION,THEN-PART[,ELSE-PART])
```

- 如果 CONDITION 成立，则取 THEN-PART。类比 Python

```
a = THEN-PART if CONDITION else ELSE-PART
```

- `$(or COND1,COND2)` `$(and COND1,COND2)` 用于逻辑运算

循环结构与自定义函数

- 数据本身可以构造一重循环，当循环多于一重时，使用 `$(foreach )`

```
# radius list
rl:=$(shell seq -w 0.10 0.01 1.00)
profile:=$(foreach d,x y z,$(rl:%=1t_+%_$(d).h5))
$(info $(profile))
```

```
1t_+0.10_x.h5 1t_+0.11_x.h5 1t_+0.12_x.h5 1t_+0.13_x.h5 ...
```

- Makefile 模版（起到“函数”复用的作用）由 `define … endef` 定义，使用 `$(eval )` 执行。

```
define video
$(1)/%.avi: $(1)/%.mp4
        ffmpeg -i $$^ $$@

endef

$(eval $(foreach d,up down transverse,$(call video,$(d))))
```

更简洁的笛卡尔积：bash

```
echo {w..z}{1,2,5,9}
```

```
w1 w2 w5 w9 x1 x2 x5 x9 y1 y2 y5 y9 z1 z2 z5 z9
```

lecture 仓库的 Makefile

- `https://git.tsinghua.edu.cn/physics-data/lecture/`
  - 仓库使用的 `Makefile` 描述如何生成课件和讲义。
  - 尝试阅读 `Makefile` 给出文件的依赖关系。

JUNO 真实世界例子

- 用于数百 TB 蒙特卡罗数据生成的例子

```
# 1=model, 2=imh, 3=dist, 4=iter
define SN-tpl

output+=data/det/$(1)/$(2)/$(3)/ith/$(4).root

data/$(1)/$(2)/$(3)/%.root:
        ./genSN.sh $(1) $(2) $(3) $$@

data/det/%/$(4).root: data/%.root
        ./exeDet.sh $$^ $$@ $(4)
endef

$(eval $(foreach i,$(imod),$(foreach j,$(imh),\
        $(foreach k,$(dist),$(call SN-tpl,$(i),$(j),$(k),0))\
        $(foreach l,$(ip),$(call SN-tpl,$(i),$(j),10,$(l))))))

all: $(foreach i,$(ith),$(subst ith,$(i),$(output)))

# Delete partial files when the processes are killed.
.DELETE_ON_ERROR:
# Keep intermediate files around
```

并行计算

- `make -j`
- 把 SHELL 换成超算任务调度器 → 超算并行计算

命令行指定变量

```
make -j SHELL=/bin/bash
```

联合目标

When 'make' builds any one of the grouped targets, it understands that all the other targets in the group are also created as a result of the invocation of the recipe. Furthermore, if only some of the grouped targets are out of date or missing 'make' will realize that running the recipe will update all of the targets.
As an example, this rule defines a grouped target:

```
foo bar biz &: baz boz
        echo $^ > foo
        echo $^ > bar
        echo $^ > biz
```