Task Automation

Anthony Scemama <scemama@irsamc.ups-tlse.fr>

Labratoire de Chimie et Physique Quantiques
IRSAMC (Toulouse)

Introduction

- Many common tasks are repetitive
- Computers are better than humans at doing repetitive tasks
- When you have the feeling you are doing *mechanical* work, immediately stop and write a program to do it for you

Outline

1. Introduction to GNU make
2. Automating GIT
3. Metaprogramming

Introduction to GNU make

1. Introduction to GNU make
2. Automating GIT
3. Metaprogramming

- The make utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them.
- You need a file called a *makefile* to tell make what to do.
- Rules are given in the makefile
- The compilation is done by running `make`
- The `-j N` option will use N processes to compile in parallel
- Internally a dependency tree of the files is built, and the last modification dates are accessed to check if a file needs to be re-built.

Rules

Rules have the following shape

```
target : prerequisites
        recipe
...
```

- `target` : Name of the file that is generated
- `prerequisites` : File needed to create the target
- `recipe` : Commands to create the target

Simple example:

```
my_program: main.o func1.o func2.o
	gfortran -o my_program main.o func1.o func2.o

main.o: main.f
	gfortran -c main.f

func1.o: func1.f
	gfortran -c func1.f

func2.o: func2.f
	gfortran -c func2.f

clean:
	rm my_program *.o
```

Make has *implicit* rules. These are default rules to build `.o` files from source file name extensions.

This works:

```
my_program: main.o func1.o func2.o
        gfortran -o my_program main.o func1.o func2.o
main.o: main.f
func1.o: func1.f
func2.o: func2.f

.PHONY: clean
clean:
        rm my_program *.o
```

(A phony target is one that is not really the name of a file)

Pattern rules can be defined, and this is the most common way to use make.

The % symbol represents the pattern to match. For example:

```
%.o: %.f
	gfortran -c $< -o $@
```

defines a rule to compile all files ending with `.f`. In this example, the automatic variables $@ and $< are used to substitute the names of the target file and the source file in each case where the rule applies.

```
my_program: main.o func1.o func2.o
	gfortran -o my_program $^
%.o: %.f
	gfortran -c $< -o $@
.PHONY: clean
clean:
	rm my_program *.o
```

The automatic variable $^ corresponds to the names of all the prerequisites, with spaces between them.

Variables

Variables make Makefiles simpler. Machine-dependent data can be defined in variables:

```
TARGET=my_program
F90=gfortran
F90_FLAGS=-O2
OBJ=main.o func1.o func2.o
RM=rm -f

$(TARGET): $(OBJ)
        $(F90) -o $(TARGET) $(OBJ)

%.o: %.f
        $(F90) $(F90_FLAGS) -c $< -o $@

.PHONY: clean
clean:
        $(RM) $(TARGET) $(OBJ)                                          .
```

Built-in functions can simplify variable definitions:

```
SRC=$(wildcard *.f)
OBJ=$(patsubst %.f, %.o, $(SRC))
```

- `$(wildcard *.f)` is equivalent to `*.f` in the shell.
- `$(patsubst %.f, %.o, $(SRC))` will return `$(SRC)` with all filenames finishing with `.f` substituted by `.o`.

All machine-dependent data can be moved into another file which will be included in the makefile:

`make.inc`:

```
TARGET=my_program
F90=gfortran
F90_FLAGS=-O2
RM=rm -f
```

Makefile :

```
include make.inc
SRC=$(wildcard *.f)
OBJ=$(patsubst %.f, %.o, $(SRC))

$(TARGET): $(OBJ)
	$(F90) -o $(TARGET) $(OBJ)

%.o: %.f
	$(F90) $(F90_FLAGS) -c $< -o $@

.PHONY: clean
clean:
	$(RM) $(TARGET) $(OBJ)
```

Automating GIT

1. Introduction to GNU make
2. Automating GIT
3. Metaprogramming

Git hooks

- Git has a way to start scripts when important actions occur : hooks
- Client-side: actions to be performed when commiting and merging
- Server-side: actions to be performed when receiving pushed commits
- Hooks are located in `.git/hooks`
- Hooks can be written in any scripting language (bash, Python, Perl, ruby, etc)
- To install a hook, make your script executable and git it the proper name

Client-side hooks

`pre-commit`

- It is run before you type the commit message
- Its is intended to check your snapshot (make tests for for example), check code style, up-to-date documentation, etc
- A non-zero exit code aborts the commit
- It can be bypassed by `git commit --no-verify`

`prepare-commit-msg`

- Lets you edit the default message before the editor is opened

`commit-msg`

- Used to check if the commit message is valid
- If the exit code is non-zero, the commit is aborted

`post-commit`

- Run when the commit is completed

`post-checkout`

- Run after a succesful check out

`post-merge`

- Run after a succesful merge
- Can validate that files not tracked by git are present

Server-side hooks

`pre-receive`

- Runs on the server when a push occurs
- If the exit code is non-zero, the push is aborted

`post-receive`

- Runs on the server when a push is completed
- Used to notify users by email, update a ticket tracking system, etc

Hook example

Goal: Refuse to add the `make.inc` file to the repository.

Add to your `.git/config`

```
[hooks]
    refused = make.inc
```

Then create the `.git/hooks/pre-commit` file and change the permissions to make it executable.

```
#!/bin/bash
ADDED_FILES=$(git diff-index --name-status HEAD -- | cut -c3-)
for FILE in $ADDED_FILES
do
    for REFUSED in $(git config hooks.refused)
    do
      if [[ $FILE = $REFUSED ]]
      then
          echo $FILE should not be added
          exit 1
      fi
    done
done
exit 0
```

Git bisect

- You are discovering a bug that has been added in the past
- You have no idea when the bug was added
- You are sure that two years ago, the bug wasn't there
- You can use a binary search to find the bug in $\mathcal{O}(\log(N))$
- The machine is better than you for this job

```
$ ./my_program
 One plus one equals    1.0000000
```

On your current revision, start git bisect:

```
$ git bisect start
```

Tell git-bisect that the current revision is bad:

```
$ git bisect bad
```

Tell git-bisect that the revision two years ago was good

```
$ git bisect good 938ca4781610eeee9bf18abefc4c0da3229776bd
Bisecting: 161 revisions left to test after this (roughly 7 steps)
[35bd4cfd09390790c6c656813038f2b082243504] Random commit number 19172
```

```
$ make >/dev/null ; ./my_program
 One plus one equals    1.0000000
$ git bisect bad
Bisecting: 80 revisions left to test after this (roughly 6 steps)
[956cdd7991036540f250cdf075683502ef9a66b3] Random commit number 9520
$
$ make >/dev/null ; ./my_program
 One plus one equals    2.0000000
$ git bisect good
Bisecting: 40 revisions left to test after this (roughly 5 steps)
[17c741ffbf2f850aa319a46d02d8ec8471e30bf5] Random commit number 14280
...
$
```

```
$ make >/dev/null ; ./my_program
 One plus one equals    1.0000000
$
$ git bisect bad
61bc7a8a58684260ccf276dce4ed81207ffee409 is the first bad commit
commit 61bc7a8a58684260ccf276dce4ed81207ffee409
Author: Anthony Scemama <scemama@irsamc.ups-tlse.fr>
Date:   Tue Apr 9 12:07:00 2013 +0200

    Surprise...

:100644 100644 5334b6165ca4e33192717b47f48cc00eeebd76ac
 18f15065c486d68e456fafb6bd43c5c4b3c043ac M      main.f
```

- If compiling and testing takes 10 minutes, the bisection can take a few hours!
- Git bisect can be automated using a script
- If the exit code is zero, the test is good
- If the exit code is non-zero, the test is bad

```
#!/bin/bash
make >/dev/null
if [[ $? -ne 0 ]] ; then
        echo "WARNING : make failed"
        echo "Choosing randomly"
        exit $(( $RANDOM / 16384 ))
fi
result=$(./my_program | cut -b22-)
the_test=$(python -c "print $result == 2.")
if [[ $the_test = False ]] ; then
        exit 1
elif [[ $the_test = True ]] ; then
        exit 0
else
        echo "Python returned $the_test"
        exit $(( $RANDOM / 16384 ))
fi
```

```
$ git bisect start
$ git bisect bad
$ git bisect good 938ca4781610eeee9bf18abefc4c0da3229776bd
Bisecting: 161 revisions left to test after this (roughly 7 steps)
[35bd4cfd09390790c6c656813038f2b082243504] Random commit number 19172
$
$ time git bisect run ./bisect_run.sh
running ./bisect_run.sh
Bisecting: 80 revisions left to test after this (roughly 6 steps)
[956cdd7991036540f250cdf075683502ef9a66b3] Random commit number 9520
running ./bisect_run.sh
Bisecting: 40 revisions left to test after this (roughly 5 steps)
[17c741ffbf2f850aa319a46d02d8ec8471e30bf5] Random commit number 14280
...
running ./bisect_run.sh
61bc7a8a58684260ccf276dce4ed81207ffee409 is the first bad commit
commit 61bc7a8a58684260ccf276dce4ed81207ffee409
Author: Anthony Scemama <scemama@irsamc.ups-tlse.fr>
```

```
Date:   Tue Apr 9 12:07:00 2013 +0200
```

```
    Surprise...

:100644 100644 5334b6165ca4e33192717b47f48cc00eeebd76ac
18f15065c486d68e456fafb6bd43c5c4b3c043ac M      main.f
bisect run success

real    0m4.000s
user    0m2.910s
sys     0m0.210s
```

The bad commit was found in 4 seconds.

Metaprogramming

1. Introduction to GNU make
2. Automating GIT
3. Metaprogramming

Metaprogramming

- Metaprogramming : Writing programs that write programs
- Example: Compiler
- Writing code can be very boring. Let the machine do it for you
- **Advantages:**
  - Computers don't make typos
  - The names of the subroutines are *computable*, so easy to guess by a human

In Bash

A function can be defined at runtime, depending on the context

```
#!/bin/bash

debug()
{
  if [[ -n $DEBUG ]]
  then
    debug()
    {
      echo "DEBUG : $@"
      eval "$@"
    }
  else
    debug()
    {
```

```
        eval "$@"
    }
  fi
 debug $@
}


debug ls -l
debug pwd
echo HELLO
ls
debug sleep 3
```

```
$ ./test.sh
total 4
-rwxrwxr-x 1 scemama scemama 233 Apr 16 15:09 test.sh
/home/scemama/Dropbox/GDRCorrel/Cours_TaskAutomation/Bash
HELLO
```

```
test.sh
$
$ DEBUG=1 ./test.sh
DEBUG : ls -l
total 4
-rwxrwxr-x 1 scemama scemama 233 Apr 16 15:09 test.sh
DEBUG : pwd
/home/scemama/Dropbox/GDRCorrel/Cours_TaskAutomation/Bash
HELLO
test.sh
DEBUG : sleep 3
```

In Python

`exec` : execute a bit of code expressed in a string

```
#!/usr/bin/python
my_string = "print 'Hello world!'"
exec my_string
```

```
$ ./test.py
Hello world!
```

Example: Writing many similar functions

```
#!/usr/bin/python
text = """
def multiply_by_%(number)d (input):
   return input * %(number)d
"""
for number in range(1000):
  exec text%locals()
print multiply_by_69(3)
```

`locals()` returns a dictionary containing all the local variables of the current context.

`%(number)d` will be substituted by the integer (d) associated with the key `number`.

```
$ ./test.py
207
```

Example: Writing a function factory

```
def create_useful_function(number):
  text = """def f(input):
   return input * %(number)d
  """
  exec text%locals()
  return f

multiply_by_69 = create_useful_function(69)
print multiply_by_69(3)
```

```
$ ./test.py
207
```

Example: Writing a decorator

You have a class that looks like this:

```
class MyClass(object):

  def __init__(self):
    self.one = 1

  def test1(self):
    return self.one

  def test2(self):
    tmp = self.test1()
    return tmp
```

and you want to print debug statements when you enter and exit the functions.

```
>>> # Contents of the Class:
>>> print MyClass.__dict__
{'test1': <function test1 at 0x2ae350a255f0>, '__module__':  [...]
,'__main__',  '__init__': <function __init__ at 0x2ae350a25578>}
>>>
>>> # Filter out everything but functions
>>> functions = [ f for f in MyClass.__dict__.items() \
...  if str(f[1]).startswith('<function') ]
>>>
>>> for f in functions: print f
('test1', <function test1 at 0x2abd0102d5f0>)
('test2', <function test2 at 0x2abd0102d668>)
('__init__', <function __init__ at 0x2abd0102d578>)
```

Your decorator code is :

```
functions = [ f for f in MyClass.__dict__.items() \
   if str(f[1]).startswith('<function') ]

text = """
MyClass.%(name)s_original = MyClass.%(name)s
def %(name)s(self, *args, **kw):
   print "Enter %(name)s"
   result = self.%(name)s_original(*args,**kw)
   print "Leave %(name)s"
   return result
MyClass.%(name)s = %(name)s
"""

for name, f in functions:
  exec text%locals()
```

```
$ ./test.py
Enter __init__
Leave __init__
Enter test2
Enter test1
Leave test1
Leave test2
1
```

In Fortran

- Fortran does not allow do write code that is compiled upon execution
- However, it is still possible to write scripts that write Fortran code
- Often, adding a new keyword in a large code is very painful : Metaprogramming for input data

Step 1

Create a file `keywords.data` that contains a simple definition of your input keywords:

```
integer          : Natoms
double precision : time_step
logical          : optimize
double precision : integrals_threshold
```

This file should be easy to parse with a script.

Step 2

Create a script `keywords.py` that reads `keywords.data`

```
keywords_data="keywords.data"
def extract_keywords():
  file=open(keywords_data,'r')
  lines = file.readlines()
  file.close()
  result = []
  for line in lines:
    buffer = [ i.strip() for i in line.split(':') ]
    result.append(buffer)
  return result

print extract_keywords()
# [['integer', 'Natoms'],['double precision', 'time_step']
# ['logical', 'optimize'],['double precision',
# 'integrals_threshold']]
```

Step 3

Add a function to create a Fortran module containing the input data

```
keywords_module="keywords_mod.f90"

def create_module(keywords):
  text=["module keywords"]
  for typ,name in keywords:
     text.append("  %s :: %s"%(typ,name))
  text+=["end module keywords"]
  result = '\n'.join(text)
  return result
```

Generated code from our `keywords.data` file

```
module keywords
  integer :: Natoms
  double precision :: time_step
  logical :: optimize
  double precision :: integrals_threshold
end module keywords
```

Step 4

Add a function to create a Fortran subroutine to read the input

```
keywords_subroutine="keywords_sub.f90"

def create_input_reader(keywords):
  text ="""
subroutine read_input
  use keywords
  character*80     :: keyword, line
  do while (.True.)
    read(*,'(A)',end=90) line
    read(line,*) keyword
    select case(keyword)""".splitlines()
  for typ,name in keywords:
    text += [ "       case('%s')"%(name) ]
    text += [ "         read(line,*) keyword, %s"%(name) ]
  text += """       case default
         print *, trim(keyword), ': Unknown keyword'
         stop 1
    end select
  end do
  90 continue
end subroutine read_input""".splitlines()
  return '\n'.join(text)
```

Generated code from our `keywords.data` file

```
subroutine read_input                                                    .
 use keywords
 character*80     :: keyword, line
 do while (.True.)
  read(*,'(A)',end=90) line
  read(line,*) keyword
  select case(keyword)
    case('Natoms')
      read(line,*) keyword, Natoms
    case('time_step')
      read(line,*) keyword, time_step
    case('optimize')
      read(line,*) keyword, optimize
    case('integrals_threshold')
      read(line,*) keyword, integrals_threshold
    case default
      print *, trim(keyword), ': Unknown keyword'
      stop 1
  end select
 end do
 90 continue
end subroutine read_input
```

Step 5

Now, mix everything to produce the Fortran files

```
def main():
  kw = extract_keywords()
  file = open(keywords_module,'w')
  file.write(create_module(kw))
  file.close()
  file = open(keywords_subroutine,'w')
  file.write(create_input_reader(kw))
  file.close()

if __name__ == '__main__':
   main()
```

Step 6

Modify your makefile:

```
KEYWORDS=keywords_mod.f90 keywords_sub.f90
SRC=$(wildcard *.f90) $(KEYWORDS)
OBJ=$(patsubst %.f90, %.o, $(SRC))
MOD=$(wildcard *_mod.f90) $(filter %_mod.f90, $(KEYWORDS))
MOD_OBJ=$(patsubst %.f90, %.o, $(MOD))

$(KEYWORDS): keywords.py keywords.data
	./keywords.py

%_mod.o: %_mod.f90 $(KEYWORDS)
	$(F90) $(F90_FLAGS) -c $< -o $@

%.o: %.f90 $(MOD_OBJ) $(KEYWORDS)
	$(F90) $(F90_FLAGS) -c $< -o $@
```

Index