GNU Makeの使い方

2021/5/13

慶應義塾大学理工学部物理情報工学科
渡辺

ハンズオン用リポジトリ

https://github.com/kaityo256/make_tutorial

Makeとはなにか

ビルドツールの一つ

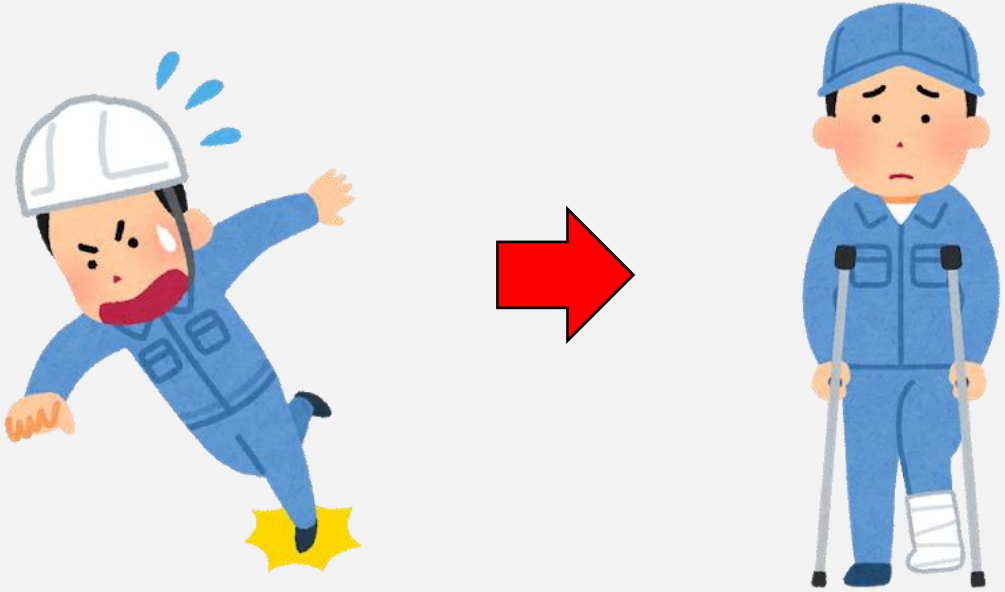
- プログラムのビルドを自動化してくれる
- 依存関係を認識してくれる
- インストールなどの作業も自動化できる

コード開発にはビルドツールは必須

他には、CMake、Rake (Ruby)、 SCons (Python)、Ant (Java)など多数

なぜビルドツールが必要か？

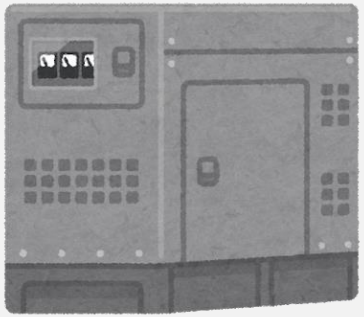
人間は間違える生き物だから



依存関係のあるタスク

装置AとBからなるシステムがあり
Aの電源を入れてからBの電源を
入れないとBが壊れてしまう

装置A



装置B

ありがちな解決策

テプラによる注意喚起

↓装置Bの電源確認！！！

←Bが上がるまで押さない！

ありがちな解決策

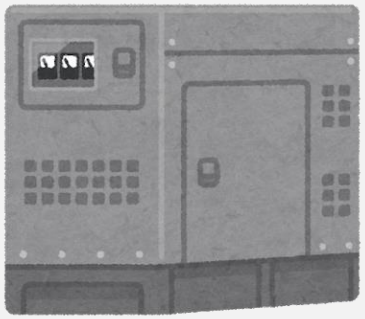
危機管理を人間の注意力に依存してはならない

↓装置Bの　　確認！！！

←Bが　　るまで押さない！

正しい解決策のひとつ

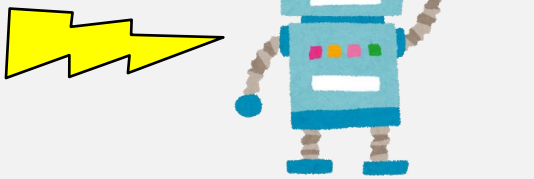
「これを押せば良い」というボタンを一つ作る



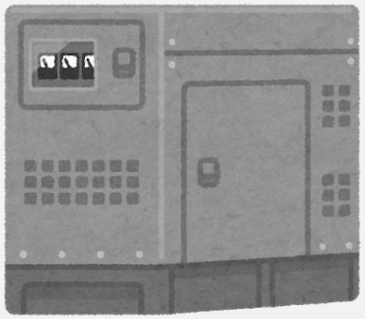
とにかくボタンを押せば
よしなに解決してくれる

正しい解決策のひとつ

「これを押せば良い」というボタンを一つ作る



装置Bをチェック
OFFなら電源ON



装置Bが立ち上がったら
装置Aの電源ON

内部では依存関係を認識し、正しく処理

ルール

Makeでは、条件とコマンドを「ルール」として記述する
ルールは、ターゲット、前提条件、コマンドから構成される

ターゲット： 前提条件
タブ → コマンド

ターゲット　実現したいこと、作りたいもの

前提条件　実現していなければならないこと

コマンド　前提条件が満たされているとき、
ターゲットを作るために必要なこと

C++の分割コンパイル

以下の3つのファイルを考える

param.hpp

```
const int N = 10;
```

main.cpp

```
#include "param.hpp"
#include <cstdio>

void show(void);

int main(void) {
  printf("main: N is %d¥n", N);
  show();
}
```

sub.cpp

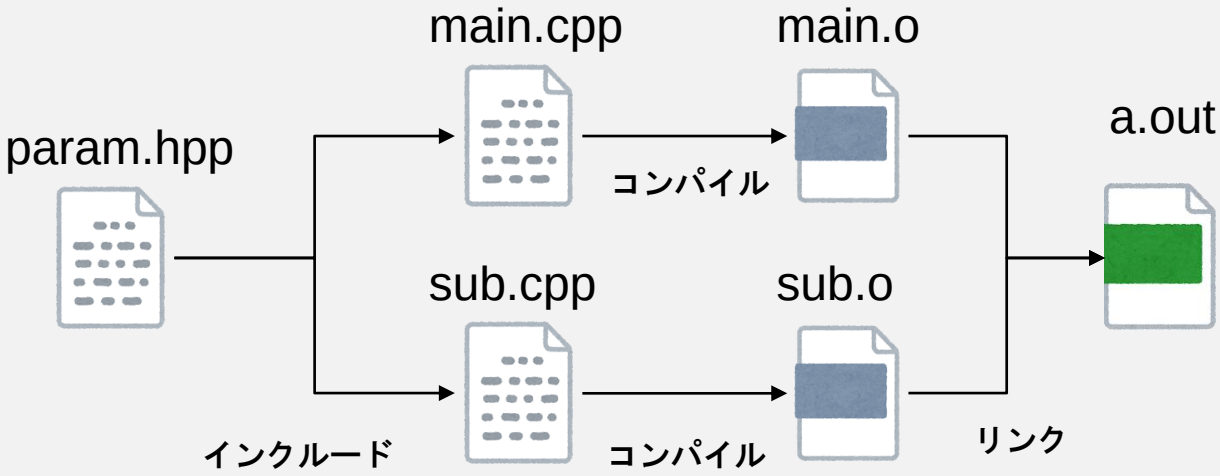
```
#include "param.hpp"
#include <cstdio>

void show(void){
  printf("sub: N is %d¥n",N);
}
```

C++の分割コンパイル

ビルド方法

```
g++ -c main.cpp
g++ -c sub.cpp
g++ main.o sub.o
```

依存関係



C++の分割コンパイル

まずは手順をそのままMakefileに書く

Makefile

```
all: a.out

a.out: main.o sub.o
	g++ main.o sub.o

main.o: main.cpp
	g++ -c main.cpp

sub.o: sub.cpp
	g++ -c sub.cpp
```

C++の分割コンパイル

カレントディレクトリにMakefile/makefileがある状態で makeを実行

```
$ make
g++ -c main.cpp
g++ -c sub.cpp
g++ main.o sub.o
```

ビルドが実行される

makeは、ファイルを指定しないと
Makefileもしくはmakefileを探しに行く

C++の分割コンパイル

```
all: a.out
```

ターゲット　引数なしで実行した場合、暗黙に「all」というターゲットを指定したことになる

前提条件　最終的に欲しい物をallの前提条件として書く

コマンド　なし

C++の分割コンパイル

```
a.out: main.o sub.o
        g++ main.o sub.o
```

ターゲット　a.outを作りたい

前提条件　そのためにはmain.oとsub.oが要る

コマンド　main.oとsub.oが用意できたら
リンクしてa.outを作る

C++の分割コンパイル

```
main.o: main.cpp
        g++ -c main.cpp
```

ターゲット　main.oを作りたい

前提条件　main.oが無いか、main.cppより古ければ作り直す

コマンド　main.cppからmain.oを作る方法

C++の分割コンパイル

ビルドをきれいにするルール「クリーン」を作る

```
clean:
        rm -f a.out *.o
```

ターゲット　ビルドをきれいにしたい(clean)

前提条件　なし

コマンド　中間ファイルや最終ターゲットを削除

```
make clean
make
```

これでクリーンビルドできる

C++の分割コンパイル

cleanも追加したMakefile

```
all: a.out

a.out: main.o sub.o
    g++ main.o sub.o

main.o: main.cpp
    g++ -c main.cpp

sub.o: sub.cpp
    g++ -c sub.cpp

clean:
    rm -f a.out *.o
```

似たような記述が繰り返されている

DRY原則

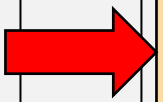
Don't Repeat Yourself

同じような記述を繰り返してはならない

※ 例えば一部を修正した場合、残りの修正忘れが発生するから

パターンルール

```
all: a.out

a.out: main.o sub.o
    g++ main.o sub.o

main.o: main.cpp
    g++ -c main.cpp

sub.o: sub.cpp
    g++ -c sub.cpp

clean:
    rm -f a.out *.o
```



まとめる

```
all: a.out

a.out: main.o sub.o
    g++ main.o sub.o

%.o: %.cpp
    g++ -c $<

clean:
    rm -f a.out *.o
```

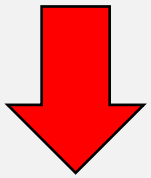
パターンルール

a.out: main.o sub.o

マッチ

a.outを作るにはmain.oが必要

%.o: %.cpp

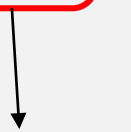


マッチにより%=mainと展開

main.o: main.cpp

自動変数(マクロ)

```
main.o: main.cpp
    g++ -c $<
```



依存関係の一番左に展開される

main.cpp

他には・・・

$@ **ターゲット名に展開** (main.o)

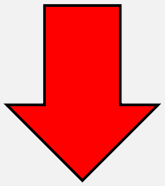
$* **パターンがマッチした部分** (main)

等多数

※ 気になったら「make 自動変数」で検索してください

パターンルール

```
%.o: %.cpp
    g++ -c $<
```



ターゲットとしてmain.oがマッチ

```
main.o: main.cpp
    g++ -c main.cpp
```

sub.oも同様

変数

```
all: a.out

a.out: main.o sub.o
    g++ main.o sub.o

%.o: %.cpp
    g++ -c $<

clean:
    rm -f a.out *.o
```

コンパイルとリンクで
同じコマンドを使っている

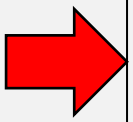
別のコンパイラを使う時、二か所を修正しなければならない

➡ DRY原則に反する

変数

CXXという変数を定義し、g++という値を代入

```
all: a.out

a.out: main.o sub.o
   g++ main.o sub.o

%.o: %.cpp
   g++ -c $<

clean:
   rm -f a.out *.o
```



```
CXX=g++
all: a.out

a.out: main.o sub.o
    $(CXX) main.o sub.o

%.o: %.cpp
    $(CXX) -c $<

clean:
    rm -f a.out *.o
```

使う時は$(変数名)とする

コンパイラを変更する場合は、一か所だけ修正すればよくなった

依存関係

このMakefileには、param.hppの依存関係が正しく入っていない

```
all: a.out
CXX=g++

a.out: main.o sub.o
	$(CXX) main.o sub.o

%.o: %.cpp
	$(CXX) -c $<

clean:
	rm -f a.out *.o
```

依存関係をmakeにどうやって教えるか？

依存関係

✕ がんばって人間が依存関係を書く

○ ツールに自動的に依存関係を抽出させる

人間のミスを防ぐための仕組みを
人間が作るのはナンセンス

依存関係

g++はMake用の依存関係を出力できる

```
$ g++ -MM *.cpp
main.o: main.cpp param.hpp
sub.o: sub.cpp param.hpp
```

ファイルにリダイレクトして

```
$ g++ -MM *.cpp > makefile.dep
```

Makefileにインクルードする

```
-include makefile.dep
```

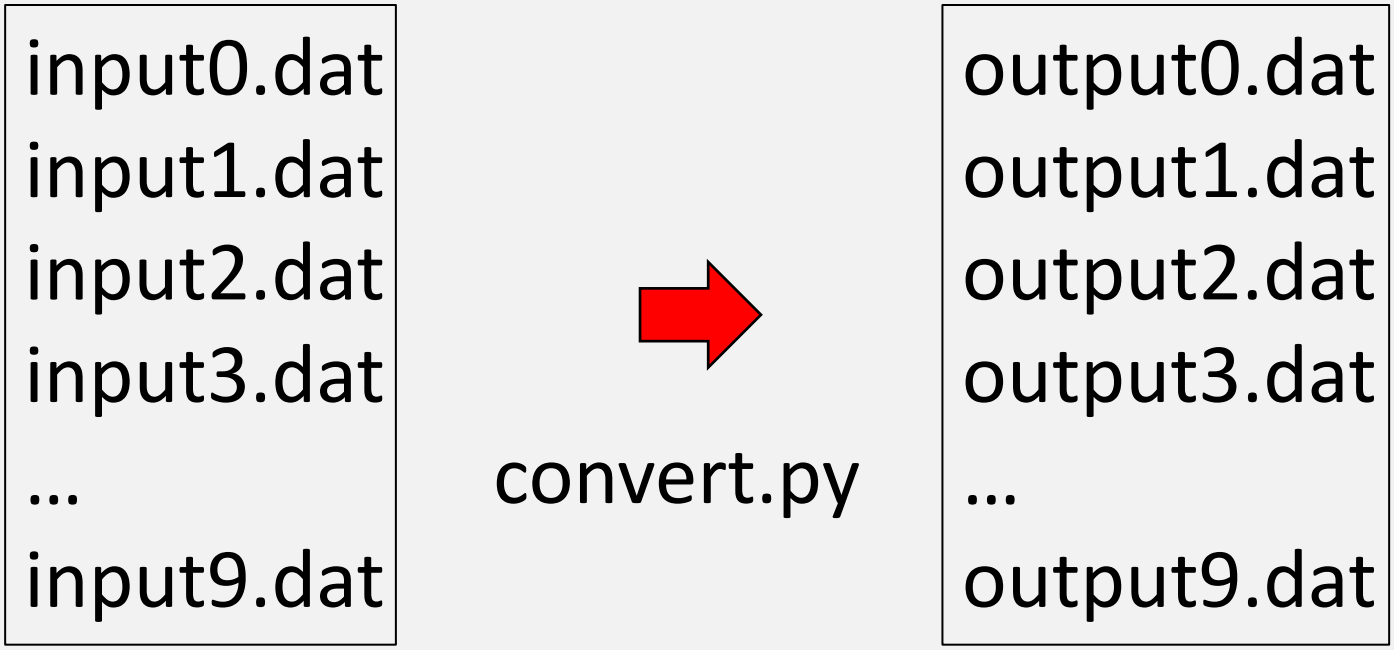
完成

```
all: a.out
CXX=g++

a.out: main.o sub.o
	$(CXX) main.o sub.o

%.o: %.cpp
	$(CXX) -c $<

clean:
	rm -f a.out *.o

-include makefile.dep
```

※ 依存関係を自動で作ったり、ソースファイルを自動で取得したり、
まだ自動化できる部分はいろいろある

makeの応用例:データ処理

大量のデータをスクリプトで変換したい




```
python convert.py < input0.dat > output0.dat
python convert.py < input1.dat > output1.dat
python convert.py < input2.dat > output2.dat
…
```

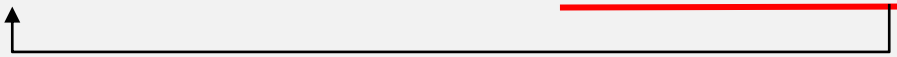
makeの応用例:データ処理

こんなMakefileを書けばmake一発で変換できる

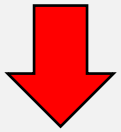
```
INPUTS=$(shell ls input*.dat)
OUTPUTS=$(INPUTS:input%=output%)

all: $(OUTPUTS)

output%: input%
    python convert.py < $< > $@

clean:
    rm -f $(OUTPUTS)
```

シェル関数

```
INPUTS=$(shell ls input*.dat)
```



実行結果を変数に代入する



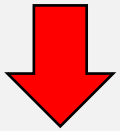
```
INPUTS=input0.dat input1.dat input2.dat … input9.dat
```

変数の置換

```
OUTPUTS=$(INPUTS:input%=output%)
```

別の変数を、パターンマッチにより置換する

```
INPUTS=input0.dat input1.dat … input9.dat
```

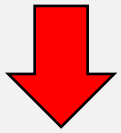


```
OUTPUTS=output0.dat output1.dat … output9.dat
```

パターンルール

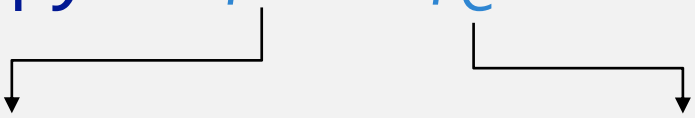
```
all: output0.dat output1.dat …
```


```
output%: input%
    python convert.py < $< > $@
```



% = 0.dat

```
output0.dat: input0.dat
    python convert.py < $< > $@
```



input0.dat　　output0.dat

Makeによるデータ処理

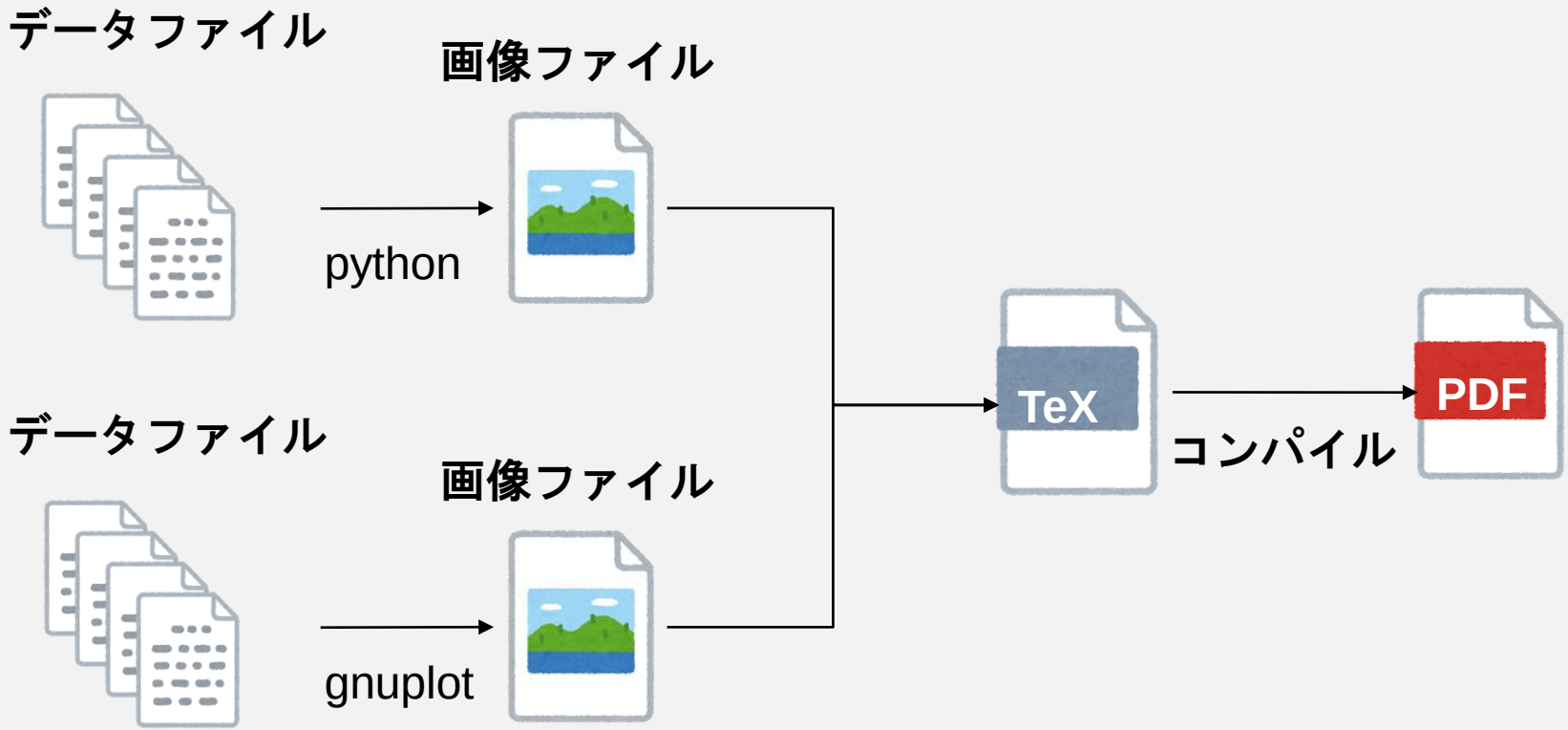
- 更新されたファイルのみ変換されて効率的
- make –j による並列ビルドができて便利
- **データの変換方法が記録として残る**

三日後の自分は他人
「データフォルダでmakeすればよい」とだけ
覚えておけば良いので、判断力を消費しない

※ makeではなくシェルスクリプトでも良いから、とにかく自動化＆保存

Makeによる論文ビルド

依存関係と処理をMakefileに記述しておけば、データの更新から論文PDFまでmake一発で行く



最初に精度の低いデータで図を作っておいて、後から本番の図に差し替える時等に便利

まとめ

依存関係のあるタスクは原則として自動化する

「○○したら○○しなければならない」や
「○○の前には○○すること」は危険信号

データ処理などは原則として自動化しておく

便利のためというより、後の記録のために