

Week3 调试及性能分析、元编程演示实验、 PyTorch 编程

黄琬晴

September 2025

本次实验的代码与实验报告上传在 github 仓库中：

<https://github.com/uuukyoo/systools>

目录

1 调试和性能分析	1
1.1 查看日志	1
1.2 脚本检查	2
1.3 使用 python 的 time 模块计时来进行性能分析	4
1.4 对排序算法的性能分析	6
1.5 对排序算法占用内存的分析	8
1.6 斐波那契	10
1.7 找到正在监听端口的进程	15
1.8 cpu 可视化与限制进程资源	15
2 元编程	19
2.1 创建 makefile 并尝试 make	19
2.2 创建依赖文件再次 make	20
2.3 make clean	26
2.4 使用 git ls-files 子命令来清除未被 git 追踪的构建产物	27
2.5 hook	29

3 PyTorch 编程	30
3.1 张量创建	31
3.2 张量属性	34
3.3 张量操作	35
3.4 自动求导	38
3.5 神经网络	38
3.6 训练模型	40
3.7 图片识别	42
4 实验心得	44

1 调试和性能分析

1.1 查看日志

使用 Linux 上的 journalctl 或 macOS 上的 log show 命令来获取最近一天中超级用户的登录信息及其所执行的指令。如果找不到相关信息，您可以执行一些无害的命令，例如 sudo ls 然后再次查看

在 linux 上可以使用 journalctl 查看日志，并使用 --since "1 day ago" 筛选一天之内的条目

如果直接使用 grep sudo 来过滤，可能会得到虽然含 sudo 字段但是并不是 sudo 用户的信息

```
hwq@hwq:~$ journalctl --since "1 day ago" | grep sudo
9月 19 09:24:51 hwq systemd[1]: sssd-sudo.socket: Bound to unit sssd.service, bu
t unit isn't active.
9月 19 09:24:51 hwq systemd[1]: Dependency failed for sssd-sudo.socket - SSSD Su
do Service responder socket.
9月 19 09:24:51 hwq systemd[1]: sssd-sudo.socket: Job sssd-sudo.socket/start fai
led with result 'dependency'.
```

所以使用-t sudo 来筛选与 sudo 用户相关的条目，可以看到一开始没有相关信息，执行了 sudo ls 后，再次执行该命令，可以看见此时多出的条目

```
hwq@hwq:~$ journalctl -t sudo --since "1 day ago"
-- No entries --
hwq@hwq:~$ sudo ls
[sudo] hwq 的密码：
公共  buggy.sh          httpserver.py      out.log          unlock.c
模板  ca.crt            lab5_vir.py       received_file.txt vir.crt
视频  client.crt        last3.txt        server2.py      vir.key
图片  client.key         last-modified.txt server.py      virtual.py
文档  debug_for.sh      lock             snap             wk3_1_vir.py
下载  file_to_send.txt  lock.c           starttime.txt  wk3_2_vir.py
音乐  getlog.sh          marco_history.log test            wk4vir.py
桌面  getlot.sh          marco.sh         unlock
hwq@hwq:~$ journalctl -t sudo --since "1 day ago"
9月 19 09:33:16 hwq sudo[3725]:      hwq : TTY=pts/0 ; PWD=/home/hwq ; USER=root>
9月 19 09:33:16 hwq sudo[3725]: pam_unix(sudo:session): session opened for user>
9月 19 09:33:16 hwq sudo[3725]: pam_unix(sudo:session): session closed for user>
```

1.2 脚本检查

安装 shellcheck 并尝试对下面的脚本进行检查。这段代码有什么问题吗？请修复相关问题。在您的编辑器中安装一个 linter 插件，这样它就可以自动地显示相关警告信息。

首先安装 shellcheck

```
hwq@hwq:~$ sudo apt update
sudo apt install shellcheck
命中:1 http://mirrors.tuna.tsinghua.edu.cn/ubuntu noble InRelease
命中:3 https://mirrors.tuna.tsinghua.edu.cn/ubuntu noble InRelease
命中:2 http://mirrors.tuna.tsinghua.edu.cn/ubuntu noble-updates InRelease
命中:5 https://mirrors.tuna.tsinghua.edu.cn/ubuntu noble-updates InRelease
命中:4 http://mirrors.tuna.tsinghua.edu.cn/ubuntu noble-backports InRelease
命中:6 https://mirrors.tuna.tsinghua.edu.cn/ubuntu noble-backports InRelease
```

在`\~/.vimrc` 中添加如下信息：

```
call plug#begin()
Plug 'neonake/neomake'
call plug#end()
```

尝试打开 vim 时显示报错，因为未安装 plug，尝试在线安装 plug 但是无法成功

```
hwq@hwq:~$ vim
处理 /home/hwq/.vimrc 时发生错误：
第    4 行:
E117: 未知的函数: plug#begin
第    5 行:
E492: 不是编辑器的命令: Plug 'neomake/neomake'
第    6 行:
E117: 未知的函数: plug#end
请按 ENTER 或其它命令继续
hwq@hwq:~$ curl -fLo ~/.vim/autoload/plug.vim --create-dirs \
  https://raw.githubusercontent.com/junegunn/vim-plug/master/plug.vim
% Total    % Received % Xferd  Average Speed   Time     Time      Current
          Dload  Upload   Total   Spent    Left  Speed
  0       0       0       0       0       0       0  --::--  0:00:11  --::--       0
```

于是尝试离线安装 plug, 将下载好的 plug.vim 文件拷贝到`~/.vim/autoload`路径下

```
hwq@hwq:~$ mkdir -p ~/.vim/autoload/
hwq@hwq:~$ cp plug.vim ~/.vim/autoload/plug.vim
hwq@hwq:~$ vim
hwq@hwq:~$
```

安装好 plug 后，再在 vim 中执行`:PlugInstall` 安装插件

```
[Plugins] [未命名]
Updated. Elapsed time: 2.314262 sec.
[=]

- Finishing ... Done!
- neomake: 已经是最新的。
-
```

使用 vim 打开题目中给到的错误脚本，执行`:Neomake`，将光标移动到出错的位置，即可查看相应的报错

```
#!/bin/sh
## Example: a typical script with several problems
for f in $(ls *.m3u)
do
    grep -qi hq.*mp3 $f \
        && echo "Playlist $f contains a HQ file in mp3 format"
done
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
shellcheck: Iterating over ls output is fragile. Use globs. ... 3,6
```

1.3 使用 python 的 time 模块计时来进行性能分析

如下，使用 time 模块

```
hwq@hwq:~$ cat time.py
import time, random
n = random.randint(1, 10) * 100

# 获取当前时间
start = time.time()

# 执行一些操作
print("Sleeping for {} ms".format(n))
time.sleep(n/1000)

# 比较当前时间和起始时间
print(time.time() - start)
```

获取待分析操作执行前后的时间，由起止时间差可以得知执行完这些操作的系统耗时

```
hwq@hwq:~$ python3 time.py
Sleeping for 600 ms
0.6005480289459229
```

不过由于是测试起止时的时间差，所以可能还包括了对其他进程的等待的时间

```
hwq@hwq:~$ time curl https://missing.csail.mit.edu &> /dev/null

real    0m1.253s
user    0m0.019s
sys     0m0.016s
```

这涉及到真实时间，用户时间和系统时间的对比，如下真实时间为一秒多，但是用户时间和系统时间均只有 0.01 几秒

1.4 对排序算法的性能分析

这里有一些排序算法的实现。请使用 cProfile 和 line_profiler 来比较插入排序和快速排序的性能。两种算法的瓶颈分别在哪里？

首先使用 cProfile 来分析性能，按执行实践排序

```
hwq@hwq:~$ python3 -m cProfile -s time sorts.py | grep sorts.py
33826/1000    0.013    0.000    0.015    0.000 sorts.py:23(quicksort)
33820/1000    0.011    0.000    0.012    0.000 sorts.py:32(quicksort_inplace)
      3    0.009    0.003    0.114    0.038 sorts.py:4(test_sorted)
    1000    0.005    0.000    0.005    0.000 sorts.py:11(insertionsort)
      1    0.000    0.000    0.114    0.114 sorts.py:1(<module>)
```

使用 line_profiler 进行分析，首先安装 line_profiler

```
hwq@hwq:~$ sudo apt install python3-line-profiler
正在读取软件包列表... 完成
正在分析软件包的依赖关系树... 完成
正在读取状态信息... 完成
下列软件包是自动安装的并且现在不需要了：
  libgl1-amber-dri libglapi-mesa libllvm17t64 python3-netifaces
使用 'sudo apt autoremove' 来卸载它(它们)。
下列【新】软件包将被安装：
```

再在需要分析的函数前添加装饰器，执行 kernprof -l -v sorts.py 命令

```
hwq@hwq:~$ kernprof -l -v sorts.py
Wrote profile results to sorts.py.lprof
Timer unit: 1e-06 s

Total time: 0.057413 s
```

可以看见，插入排序耗时 0.057413s

```
Total time: 0.057413 s
File: sorts.py
Function: insertionsort at line 10

Line #      Hits         Time  Per Hit   % Time  Line Contents
=====
10                      @profile
11                      def insertionsort(array):
12
13      25473      2460.5     0.1      4.3
14      24473      1879.8     0.1      3.3
15      24473      2158.4     0.1      3.8
16    219701      19335.6    0.1     33.7
17    195228      14799.6    0.1     25.8
18    195228      14585.5    0.1     25.4
19      24473      2003.4     0.1      3.5
20      1000       190.2     0.2      0.3
                                return array

Total time: 0.0467027 s
```

快排耗时 0.0467027s，可以看出快排耗时更短，快速排序的瓶颈在于 left 和 right 的赋值，而插入排序的瓶颈在 while 循环。

```

Total time: 0.0467027 s
File: sorts.py
Function: quicksort at line 22

Line #      Hits         Time  Per Hit   % Time  Line Contents
=====
22                      @profile
23                      def quicksort(array):
24  33864        3926.1     0.1      8.4
25  17432        1592.1     0.1      3.4
26  16432        3399.7     0.2      7.3
27 125575        16370.8    0.1     35.1
if i < pivot]
28  125575        14001.9    0.1     30.0
] if i >= pivot]
29  16432        7412.1     0.5     15.9
vot] + quicksort(right)

```

1.5 对排序算法占用内存的分析

然后使用 memory_profiler 来检查内存消耗，为什么插入排序更好一些？

使用 memory_profiler 对两种排序的耗时进行分析

```

hwq@hwq:~$ pip install memory_profiler --break-system-packages
Defaulting to user installation because normal site-packages is not writeable
Collecting memory_profiler
  Downloading memory_profiler-0.61.0-py3-none-any.whl.metadata (20 kB)
Collecting psutil (from memory_profiler)
  Downloading psutil-7.1.0-cp36-abi3-manylinux_2_12_x86_64.manylinux2010_x86_64.manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (23 kB)
  Downloading memory_profiler-0.61.0-py3-none-any.whl (31 kB)
  Downloading psutil-7.1.0-cp36-abi3-manylinux_2_12_x86_64.manylinux2010_x86_64.manylinux_2_17_x86_64.manylinux2014_x86_64.whl (31 kB)

```

先在插入排序的函数前添加装饰器

```

hwq@hwq:~$ python3 -m memory_profiler sorts.py
Filename: sorts.py

Line #    Mem usage      Increment  Occurrences   Line Contents
=====
  10    23.117 MiB    23.117 MiB        1000  @profile
  11                               def insertionsort(array):
  12
  13    23.117 MiB    0.000 MiB     25939      for i in range(len(array)):
  14    23.117 MiB    0.000 MiB     24939          j = i-1
  15    23.117 MiB    0.000 MiB     24939          v = array[i]
  16    23.117 MiB    0.000 MiB    229659         while j >= 0 and v < arra
y[j]:
  17    23.117 MiB    0.000 MiB    204720             array[j+1] = array[j]
  18    23.117 MiB    0.000 MiB    204720             j -= 1
  19    23.117 MiB    0.000 MiB    24939             array[j+1] = v
  20    23.117 MiB    0.000 MiB     1000      return array

```

再将装饰器改为在快速排序的函数前，两者的内存消耗差不多，可能是因为测试用例规模太小

```

hwq@hwq:~$ python3 -m memory_profiler sorts.py
Filename: sorts.py

Line #    Mem usage      Increment  Occurrences   Line Contents
=====
  22    23.137 MiB    23.137 MiB     34444  @profile
  23                               def quicksort(array):
  24    23.137 MiB    0.000 MiB     34444      if len(array) <= 1:
  25    23.137 MiB    0.000 MiB     17722          return array
  26    23.137 MiB    0.000 MiB     16722      pivot = array[0]
  27    23.137 MiB    0.000 MiB    127777      left = [i for i in array[1:]]
if i < pivot]
  28    23.137 MiB    0.000 MiB    127777      right = [i for i in array[1:]]
if i >= pivot]
  29    23.137 MiB    0.000 MiB     16722      return quicksort(left) + [pivot] + quicksort(right)

```

1.6 斐波那契

这里有一些用于计算斐波那契数列 Python 代码，它为计算每个数字都定义了一个函数

```
#!/usr/bin/env python
def fib0(): return 0

def fib1(): return 1

s = """def fib{}(): return fib{}() + fib{}()"""

if __name__ == '__main__':
    for n in range(2, 10):
        exec(s.format(n, n-1, n-2))
    # from functools import lru_cache
    # for n in range(10):
    #     exec("fib{} = lru_cache(1)(fib{})".format(n, n))
    print(eval("fib9()"))
```

将代码拷贝到文件中使其变为一个可执行的程序。首先安装 pycallgraph 和 graphviz(如果您能够执行 dot, 则说明已经安装了 GraphViz.)。并使用 pycallgraph graphviz -./fib.py 来执行代码并查看 pycallgraph.png 这个文件。fib0 被调用了多少次？我们可以通过记忆法来对其进行优化。将注释掉的部分放开，然后重新生成图片。这回每个 fibN 函数被调用了多少次？

首先将题目所给代码写入 fib.py 中（把首行的 python 改为 python3）并添加执行权限

```
hwq@hwq:~$ vim fib.py
hwq@hwq:~$ chmod +x fib.py
```

安装 pycallgraph

```
hwq@hwq:~$ sudo apt install python3-pycallgraph
[sudo] hwq 的密码：
正在读取软件包列表... 完成
正在分析软件包的依赖关系树... 完成
正在读取状态信息... 完成
下列软件包是自动安装的并且现在不需要了：
liblouis-dev liblouis0 liblouis-data liblouis0.10.0


```

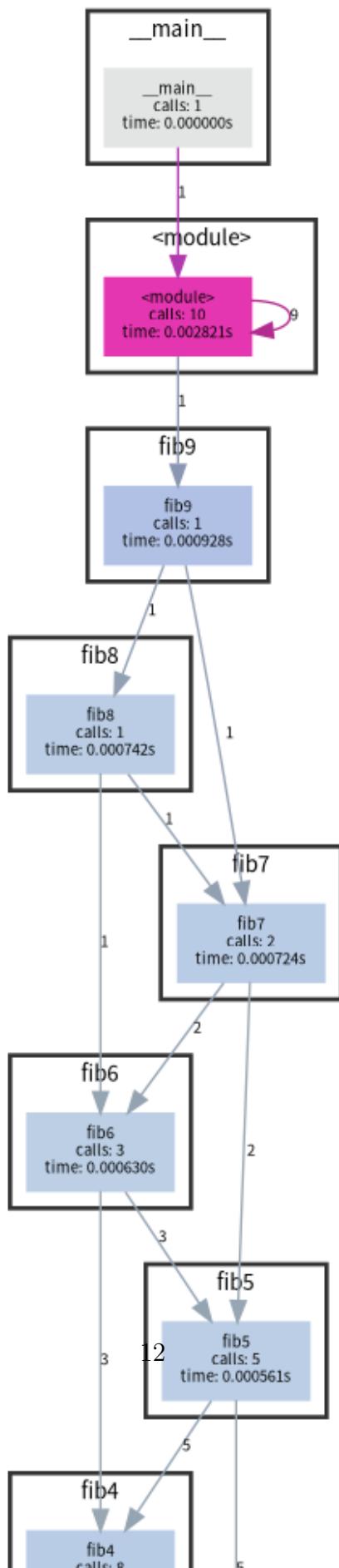
安装 graphviz

```
终端 hwq:~$ sudo apt-get install graphviz
正在读取软件包列表... 完成
正在分析软件包的依赖关系树... 完成
正在读取状态信息... 完成
graphviz 已经是最新版 (2.42.2-9ubuntu0.1)。
graphviz 已设置为手动安装
```

执行代码

```
hwq@hwq:~$ pycallgraph graphviz -- ./fib.py
34
```

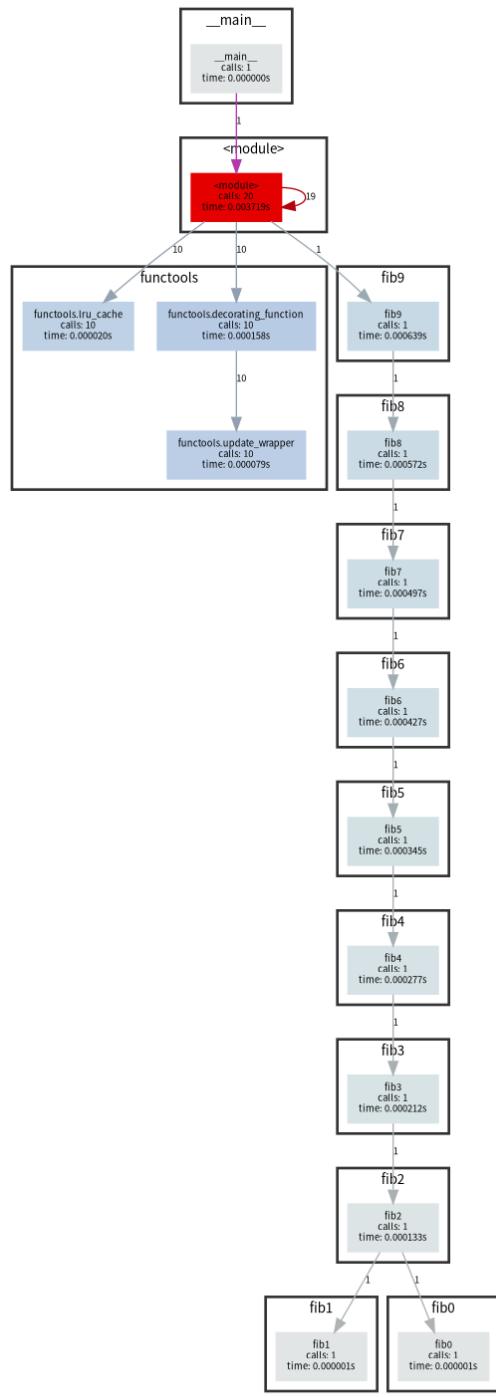
可以看见生成的 pycallgraph.png, fib0 被调用了 21 次



去掉注释内容再次执行

```
hwq@hwq:~$ vim fib.py
hwq@hwq:~$ pycallgraph graphviz -- ./fib.py
34
```

此时每个 fibN 都只调用了一次



Generated by Python Call Graph v1.1.3

1.7 找到正在监听端口的进程

我们经常会遇到的情况是某个我们希望去监听的端口已经被其他进程占用了。让我们通过进程的 PID 查找相应的进程。首先执行 `python -m http.server 4444` 启动一个最简单的 web 服务器来监听 4444 端口。在另外一个终端中，执行 `lsof | grep LISTEN` 打印出所有监听端口的进程及相应的端口。找到对应的 PID 然后使用 `kill <PID>` 停止该进程。

执行 `python3 -m http.server 4444` 来监听端口 4444

```
hwq@hwq:~$ python3 -m http.server 4444
Serving HTTP on 0.0.0.0 port 4444 (http://0.0.0.0:4444/) ...
```

打开另外一个终端，由于监听端口的进程过多，所以这里只过滤了含有 `python3` 字段的进程，找到相应的 pid，使用 `kill` 命令终止进程

```
hwq@hwq:~$ lsof | grep LISTEN | grep python3
python3 2921 hwq 3u IPv4 3
2378 0t0 TCP *:4444 (LISTEN)
hwq@hwq:~$ kill 2921
hwq@hwq:~$ lsof | grep LISTEN | grep python3
hwq@hwq:~$
```

回到原来的终端可以看见监听已停止

```
hwq@hwq:~$ python3 -m http.server 4444
Serving HTTP on 0.0.0.0 port 4444 (http://0.0.0.0:4444/) ...
已终止
hwq@hwq:~$
```

1.8 CPU 可视化与限制进程资源

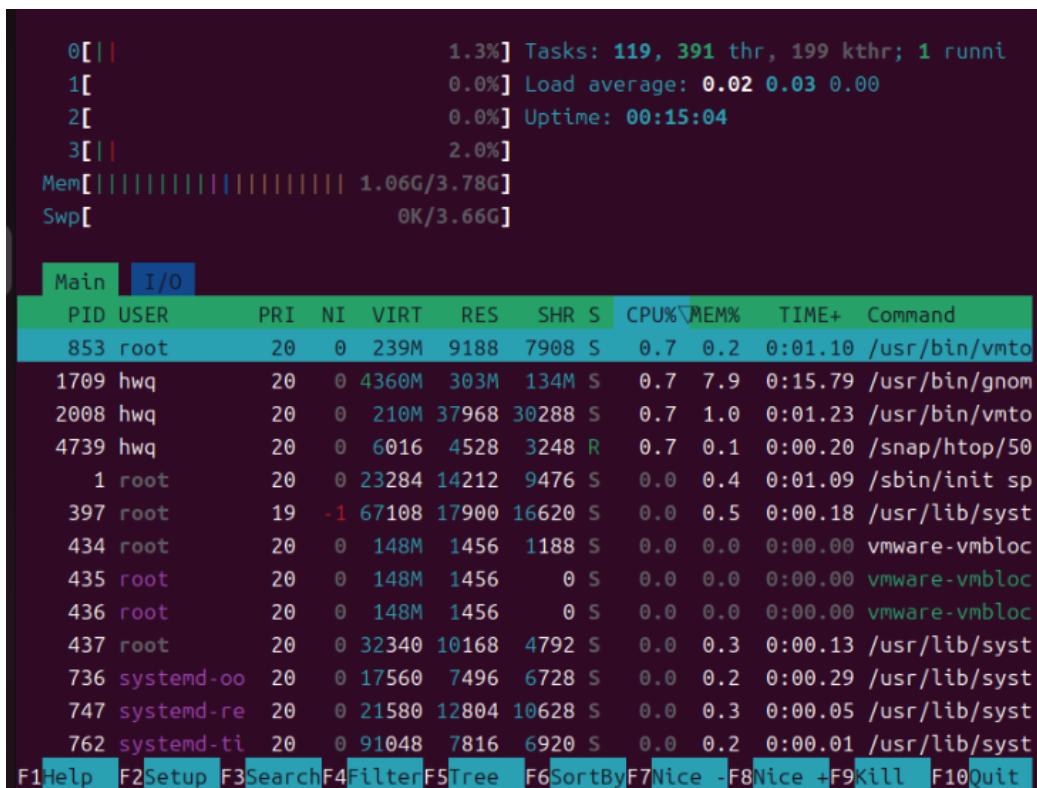
限制进程资源也是一个非常有用的技术。执行 `stress -c 3` 并使用 `htop` 对 CPU 消耗进行可视化。现在，执行 `taskset -cpu-list 0,2 stress -c 3`

并可视化。stress 占用了 3 个 CPU 吗？为什么没有？阅读 man taskset 来寻找答案

首先安装 htop

```
hwq@hwq:~$ sudo snap install htop # version 3.4.1
htop 3.4.1 from Maximiliano Bertacchini • (maxiberta) installed
hwq@hwq:~$
```

查看设备正常运行状态下的资源占用情况



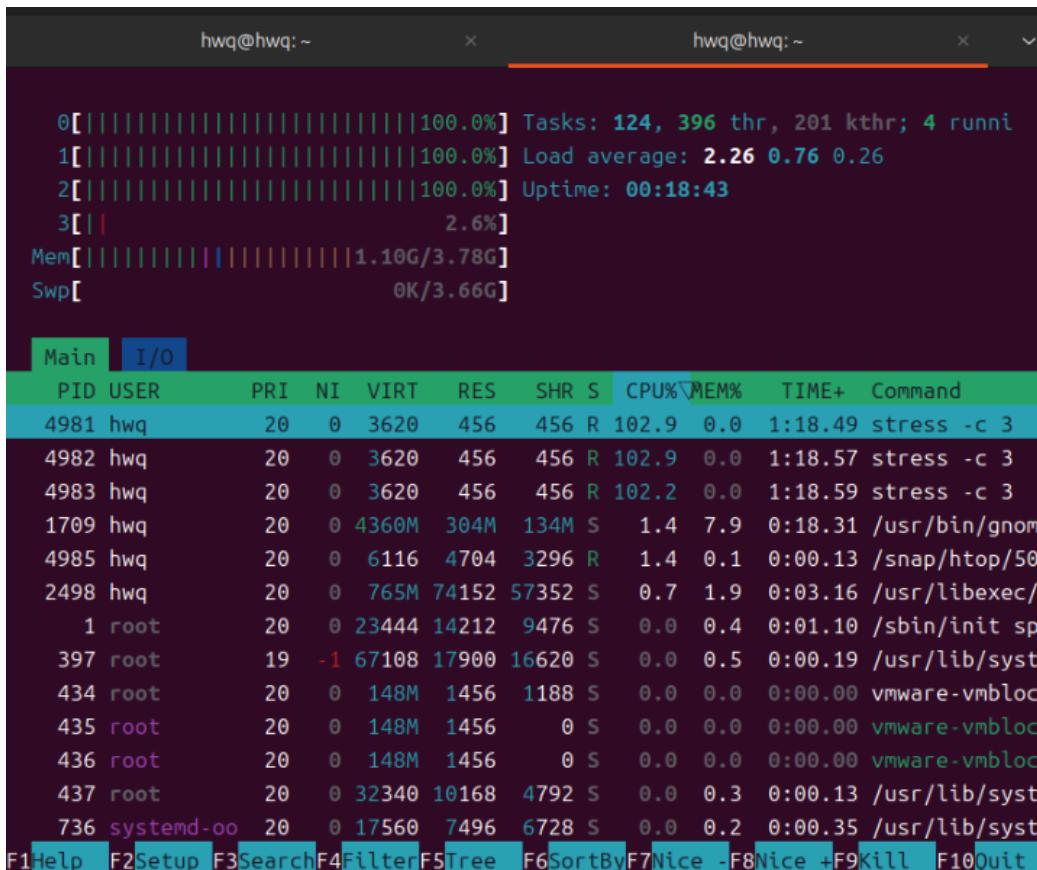
安装 stress

```
sudo apt install stress
hwq@hwq:~$ sudo apt install stress
正在读取软件包列表... 完成
正在分析软件包的依赖关系树... 完成
正在读取状态信息... 完成
下列软件包是自动安装的并且现在不需要了：
  libgl1-amber-dri libglapi-mesa liblllvm17t64 python3-netifaces
使用'sudo apt autoremove'来卸载它(它们)。
下列【新】软件包将被安装：
  stress
升级了 0 个软件包，新安装了 1 个软件包，要卸载 0 个软件包，有 18 个软件包
级。
需要下载 18.1 kB 的归档。
解压缩后会消耗 52.2 kB 的额外空间。
```

执行 stress -c 3 创建负载

```
hwq@hwq:~$ stress -c 3
stress: info: [4980] dispatching hogs: 3 cpu, 0 io, 0 vm, 0 hdd
```

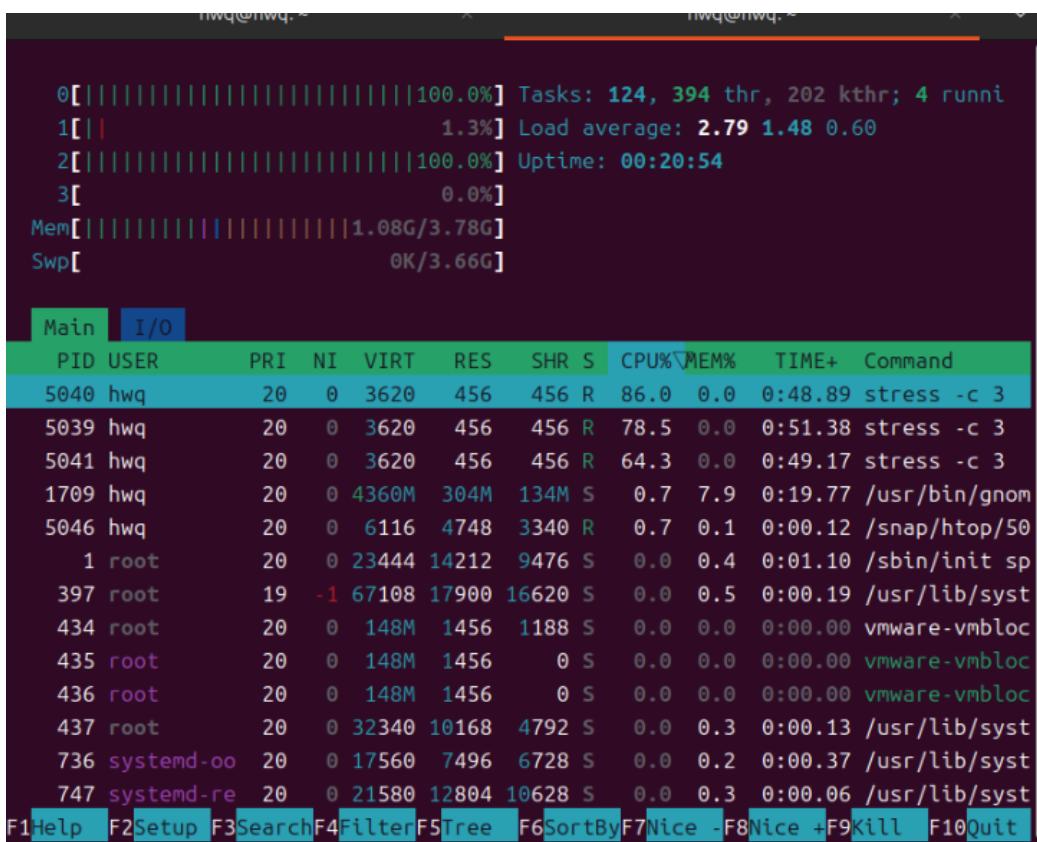
可以看见有三个 cpu 核心被完全占用



执行 taskset --cpu-list 0,2 stress -c 3

```
hwq@hwq:~$ taskset --cpu-list 0,2 stress -c 3
stress: info: [5038] dispatching hogs: 3 cpu, 0 io, 0 vm, 0 hdd
```

并没有占用三个 cpu 这是因为 taskset --cpu-list 0,2 指定了 stress 只能占用编号为 0 和 2 的 cpu 核心，所以即便有三个工作线程，也只能占用两个 cpu



2 元编程

2.1 创建 makefile 并尝试 make

make 是最常用的构建系统之一，您会发现它通常被安装到了几乎所有基于 UNIX 的系统中。make 并不完美，但是对于中小型项目来说，它已经足够好了。当您执行 make 时，它会去参考当前目录下名为 Makefile 的文件。所有构建目标、相关依赖和规则都需要在该文件中定义

在 paper 文件夹中创建 Makefile 文件

```
hwq@hwq:~/paper$ touch Makefile
hwq@hwq:~/paper$ vim Makefile
```

内容如下：

冒号左侧的 paper.pdf 和 plot-%.png 是构建目标，右侧的 paper.tex, plot-data.png 是构建 paper.pdf 的依赖，%.dat, plot.py 是 plot-%.png 的依赖，缩进部分是从依赖构建目标时所需的指令

```
paper.pdf: paper.tex plot-data.png  
        pdflatex paper.tex  
  
plot-%.png: %.dat plot.py  
        ./plot.py -i $*.dat -o $@  
~
```

使用 make 命令尝试构建，提示需要 paper.tex，没有规则可构建 paper.tex

```
hwq@hwq:~/paper$ make  
make: *** 没有规则可制作目标“paper.tex”，由“paper.pdf”需求。 停止。
```

创建 paper.tex 文件，再次尝试 make，此时的提示是因为虽然有构建 plot-data.png 的规则，但是构建其的 data.dat 并不存在，所以无法构建 plot-data.png

```
hwq@hwq:~/paper$ touch paper.tex  
hwq@hwq:~/paper$ make  
make: *** 没有规则可制作目标“plot-data.png”，由“paper.pdf”需求。 停止。
```

2.2 创建依赖文件再次 make

创建 makefile 中所需的依赖文件

```
hwq@hwq:~/paper$ vim paper.tex
hwq@hwq:~/paper$ vim plot.py
hwq@hwq:~/paper$ vim plot.py
hwq@hwq:~/paper$ vim data.dat
```

内容如下

```
hwq@hwq:~/paper$ cat paper.tex
\documentclass{article}
\usepackage{graphicx}
\begin{document}
\includegraphics[scale=0.65]{plot-data.png}
\end{document}
hwq@hwq:~/paper$ cat plot.py
#!/usr/bin/env python
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('-i', type=argparse.FileType('r'))
parser.add_argument('-o')
args = parser.parse_args()

data = np.loadtxt(args.i)
plt.plot(data[:, 0], data[:, 1])
plt.savefig(args.o)
```

```
hwq@hwq:~/paper$ cat data.dat
1 1
2 2
3 3
4 4
5 8
```

再次 make, 提示权限不够, 为 plot.py 加上执行权限

```
hwq@hwq:~/paper$ make
./plot.py -i data.dat -o plot-data.png
make: ./plot.py: 权限不够
make: *** [Makefile:5: plot-data.png] 错误 127
hwq@hwq:~/paper$ chmod +x plot.py
```

此时提示”python”: 没有那个文件或目录

将 plot.py 第一行#!/usr/bin/env python 中的 python 改为 python3

```
hwq@hwq:~/paper$ make
./plot.py -i data.dat -o plot-data.png
/usr/bin/env: "python": 没有那个文件或目录
make: *** [Makefile:5: plot-data.png] 错误 127
hwq@hwq:~/paper$ vim Makefile
hwq@hwq:~/paper$ vim plot.py
```

紧接着又因为无 matplotlib 和无 pdflatex 报错

```
hwq@hwq:~/paper$ make
./plot.py -i data.dat -o plot-data.png
Traceback (most recent call last):
  File "/home/hwq/paper./plot.py", line 2, in <module>
    import matplotlib
ModuleNotFoundError: No module named 'matplotlib'
make: *** [Makefile:5: plot-data.png] 错误 1
hwq@hwq:~/paper$ sudo apt install python3-matplotlib
```

```
hwq@hwq:~/paper$ make
./plot.py -i data.dat -o plot-data.png
pdflatex paper.tex
make: pdflatex: 没有那个文件或目录
```

安装 matplotlib 和 pdflatex 后

```
终端 hwq:~/paper$ sudo apt install python3-matplotlib
[sudo] hwq 的密码:
正在读取软件包列表... 完成
正在分析软件包的依赖关系树... 完成
正在读取状态信息... 完成
下列软件包是自动安装的并且现在不需要了：
 libgl1-amber-dri libglapi-mesa libllvm17t64 python3-netifaces
使用'sudo apt autoremove'来卸载它(它们)。
```

```
市场 hwq:~/paper$ sudo apt update
[sudo] hwq 的密码:
sudo apt install texlive-latex-base texlive-latex-extra texlive-fonts-recommended
命中:3 https://mirrors.tuna.tsinghua.edu.cn/ubuntu noble InRelease
获取:5 https://mirrors.tuna.tsinghua.edu.cn/ubuntu noble-updates InRelease [126 kB]
命中:1 http://mirrors.tuna.tsinghua.edu.cn/ubuntu noble InRelease
获取:6 https://mirrors.tuna.tsinghua.edu.cn/ubuntu noble-backports InRelease [16 kB]
获取:2 http://mirrors.tuna.tsinghua.edu.cn/ubuntu noble-updates InRelease [126 kB]
获取:7 https://mirrors.tuna.tsinghua.edu.cn/ubuntu noble-updates/main amd64 Pac
```

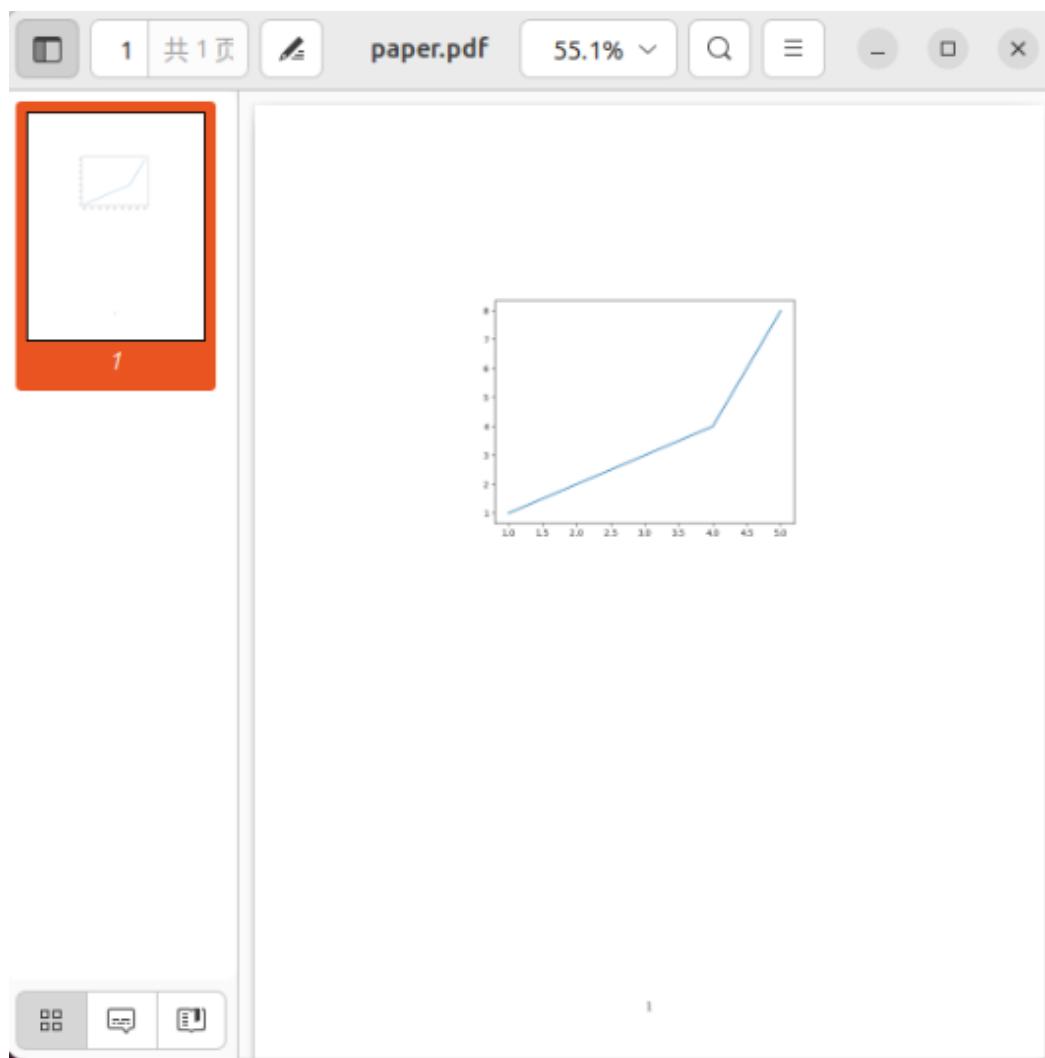
再次 make，此时构建成功

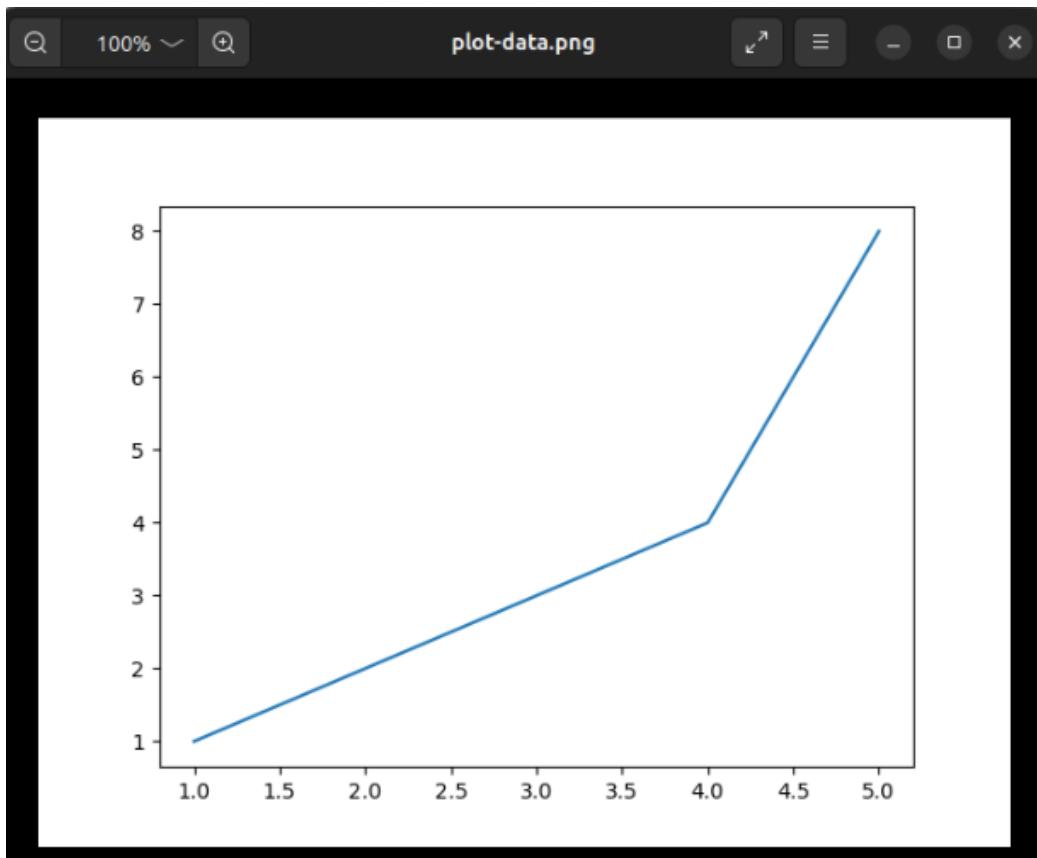
```
hwq@hwq:~/paper$ make
pdflatex paper.tex
This is pdfTeX, Version 3.141592653-2.6-1.40.25 (TeX Live 2023/Debian) (preloaded format=pdflatex)
  restricted \write18 enabled.
entering extended mode
./paper.tex
LaTeX2e <2023-11-01> patch level 1
L3 programming layer <2024-01-22>
(/usr/share/texlive/texmf-dist/tex/latex/base/article.cls
 Document Class: article 2023/05/17 v1.4n Standard LaTeX document class
(/usr/share/texlive/texmf-dist/tex/latex/base/size10.clo))
(/usr/share/texlive/texmf-dist/tex/latex/graphics/graphicx.sty
(/usr/share/texlive/texmf-dist/tex/latex/graphics/keyval.sty)
(/usr/share/texlive/texmf-dist/tex/latex/graphics/graphics.sty
(/usr/share/texlive/texmf-dist/tex/latex/graphics/trig.sty)
(/usr/share/texlive/texmf-dist/tex/latex/graphics-cfg/graphics.cfg)
(/usr/share/texlive/texmf-dist/tex/latex/graphics-def/pdftex.def)))
(/usr/share/texlive/texmf-dist/tex/latex/l3backend/l3backend-pdftex.def)
No file paper.aux.
```

ls 命令可以查看到生成的 paper.pdf 和 plot-data.png

```
回收站 :~/paper$ ls
data.dat  paper.aux  paper.pdf  plot-data.png
Makefile  paper.log  paper.tex  plot.py
```

可以在文件管理器中点击查看生成的 paper.pdf 和 plot-data.png 分别
如下





如果再次尝试 make，此时会提示 paper.pdf 已是最新

```
hwq@hwq:~/paper$ make
make: "paper.pdf" 已是最新。
```

2.3 make clean

大多数的 makefiles 都提供了一个名为 clean 的构建目标，这并不是说我们会生成一个名为 clean 的文件，而是我们可以使用它清理文件，让 make 重新构建。您可以理解为它的作用是“撤销”所有构建步骤。在上面的 makefile 中为 paper.pdf 实现一个 clean 目标。您需要构建 phony。

再次编写 Makefile 文件，在原有的文件基础上在末尾添加

```
.PHONY: clean  
clean:  
    rm -f *.pdf *.aux *.log *.png  
    # git ls-files -o | xargs rm -f
```

此时执行 make clean, 此时执行的是 rm -f *.pdf *.aux *.log *.png, 删 除存在的后缀名为.pdf .aux .log .png 的文件

```
hwq@hwq:~/paper$ make  
make: “paper.pdf”已是最新。  
hwq@hwq:~/paper$ vim Makefile  
hwq@hwq:~/paper$ make clean  
rm -f *.pdf *.aux *.log *.png  
# git ls-files -o | xargs rm -f
```

再次使用 ls 查看路径下文件, 发现所有的构建产物都已被删除

```
hwq@hwq:~/paper$ ls  
data.dat  Makefile  paper.tex  plot.py
```

2.4 使用 git ls-files 子命令来清除未被 git 追踪的构建产物

您也许会发现 git ls-files 子命令很有用

接下来在文件夹中初始化 git 仓库, 并将当前所有文件提交

```
hwq@hwq:~/paper$ git init
提示：使用 'master' 作为初始分支的名称。这个默认分支名称可能会更改。要在新仓库中
提示：配置使用初始分支名，并消除这条警告，请执行：
提示：
提示： git config --global init.defaultBranch <名称>
提示：
提示：除了 'master' 之外，通常选定的名字有 'main'、'trunk' 和 'development'。
提示：可以通过以下命令重命名刚创建的分支：
提示：
提示： git branch -m <name>
已初始化空的 Git 仓库于 /home/hwq/paper/.git/
hwq@hwq:~/paper$ git add .
hwq@hwq:~/paper$ git commit -m "paper code"
[master (根提交) 337882a] paper code
 4 files changed, 35 insertions(+)
 create mode 100644 Makefile
 create mode 100644 data.dat
 create mode 100644 paper.tex
 create mode 100755 plot.py
```

再次 make

```
hwq@hwq:~/paper$ make
./plot.py -i data.dat -o plot-data.png
pdflatex paper.tex
This is pdfTeX, Version 3.141592653-2.6-1.40.25 (TeX Live 2023/Debian) (preloaded
d format=pdflatex)
 restricted \write18 enabled.
entering extended mode
./paper.tex
LaTeX2e <2023-11-01> patch level 1
L3 programming layer <2024-01-22>
(/usr/share/texlive/texmf-dist/tex/latex/base/article.cls
 Document Class: article 2023/05/17 v1.4n Standard LaTeX document class
(/usr/share/texlive/texmf-dist/tex/latex/base/size10.clo))
(/usr/share/texlive/texmf-dist/tex/latex/graphics/graphicx.sty
(/usr/share/texlive/texmf-dist/tex/latex/graphics/keyval.sty)
( Floppy Disk e/texlive/texmf-dist/tex/latex/graphics/graphics.sty
(/usr/share/texlive/texmf-dist/tex/latex/graphics/trig.sty)
```

将`rm -f *.pdf *.aux *.log *.png`注释掉，将`# git ls-files -o xargs rm -f`取消注释，再次 make clean，此时是清理掉所有的未被 git 追踪的文件，ls 查看文件，发现所有构建产物（未提交到仓库中的文件）都已被删除

```
hwq@hwq:~/paper$ vim Makefile
hwq@hwq:~/paper$ ls
data.dat  paper.aux  paper.pdf  plot-data.png
Makefile  paper.log  paper.tex  plot.py
hwq@hwq:~/paper$ make clean
# rm -f *.pdf *.aux *.log *.png
git ls-files -o | xargs rm -f
hwq@hwq:~/paper$ ls
data.dat  Makefile  paper.tex  plot.py
```

2.5 hook

在.git/hooks 目录下创建 pre-commit 文件内容如下, git 提交时执行 make 命令, 若 make 不成功, 则输出"build failed, commit rejected" 并退出

```
hwq@hwq:~/paper$ cat .git/hooks/pre-commit
if ! make ; then
    echo "build failed, commit rejected"
    exit 1
fi
hwq@hwq:~/paper$
```

为其添加执行权限, 将 make 的依赖 paper.tex 的文件名改为 paper.tax 这样在尝试构建时由于缺失必须的依赖, 就会构建不成功

```
hwq@hwq:~/paper$ chmod +x .git/hooks/pre-commit
hwq@hwq:~/paper$ mv paper.tex paper.tax
hwq@hwq:~/paper$ ls
data.dat  Makefile  paper.tax  plot.py
```

尝试提交, 如下, 在提交时会尝试执行 make 命令, 但是由于缺失依赖, 就会构建不成功

```
hwq@hwq:~/paper$ git add .
hwq@hwq:~/paper$ git commit -m 'test again'
./plot.py -i data.dat -o plot-data.png
pdflatex paper.tex
This is pdfTeX, Version 3.141592653-2.6-1.40.25 (TeX Live 2023/Debian) (preloaded format=pdflatex)
 restricted \write18 enabled.
entering extended mode
(./paper.tex
LaTeX2e <2023-11-01> patch level 1
L3 programming layer <2024-01-22>
(/usr/share/texlive/texmf-dist/tex/latex/base/article.cls
Document Class: article 2023/05/17 v1.4n Standard LaTeX document class
(/usr/share/texlive/texmf-dist/tex/latex/base/size10.clo))
(/usr/share/texlive/texmf-dist/tex/latex/base/article.sty)
```

```
hwq@hwq:~/paper$ git add .
hwq@hwq:~/paper$ git commit -m "test commit"
make: *** 没有规则可制作目标“paper.tex”，由“paper.pdf”需求。 停止。
build failed, commit rejected
```

此时再把 paper.tax 文件名改回 paper.tex

```
hwq@hwq:~/paper$ mv paper.tax paper.tex  
hwq@hwq:~/paper$ ls  
data.dat  Makefile  paper.tex  plot.py
```

再次尝试提交，此时构建成功 文件夹中也出现了构建产物

```
回收站: ~/paper$ ls  
data.dat paper.aux paper.pdf plot-data.png  
Makefile paper.log paper.tex plot.py
```

3 PyTorch 编程

参考网址：www.runoob.com

3.1 张量创建

可以使用 `torch.zeros()` 创建全 0 张量, `torch.ones()` 创建全 1 张量, `torch.randn()` 创建指定大小的元素值随机的张量, `torch.tensor()` 创建指定的张量, 除了二维张量也可以创建其他维度的张量, 也可以在指定设备上创建张量, 在 NumPy 数组上创建张量

```
import torch
# 创建二维的张量
a = torch.zeros(3,4)
print(a)

b = torch.ones(2, 3)
print(b)

c = torch.randn(1, 6)
print(c)

tensor_2d = torch.tensor([[1,2],[3, 4]])
print("2D Tensor (Matrix):\n", tensor_2d)
print("Shape:", tensor_2d.shape) # 形状
# 创建 3D 张量 (立方体)
tensor_3d = torch.stack([tensor_2d, tensor_2d + 10, tensor_2d - 5]) # 堆叠 3 个 2D 张量
print("3D Tensor (Cube):\n", tensor_3d)
print("Shape:", tensor_3d.shape) # 形状

# 创建 4D 张量 (向量的立方体)
tensor_4d = torch.stack([tensor_3d, tensor_3d + 100]) # 堆叠 2 个 3D 张量
print("4D Tensor (Vector of Cubes):\n", tensor_4d)
print("Shape:", tensor_4d.shape) # 形状

# 创建 5D 张量 (矩阵的立方体)
tensor_5d = torch.stack([tensor_4d, tensor_4d + 1000]) # 堆叠 2 个 4D 张量
print("5D Tensor (Matrix of Cubes):\n", tensor_5d)
print("Shape:", tensor_5d.shape) # 形状
# 从NumPy数组创建张量
import numpy as np
numpy_array = np.array([[1, 2], [3, 4], [5, 6]])
tensor_from_numpy = torch.from_numpy(numpy_array)
print(tensor_from_numpy)

# 在指定设备 (CPU/GPU) 上创建张量
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
d = torch.randn(2, 3, device=device)
print(d)
```

```

tensor([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
tensor([[1., 1., 1.],
       [1., 1., 1.]])
tensor([[0.1151, 0.7521, 0.2127, 0.4999, 0.3105, 1.7729]])
2D Tensor (Matrix):
tensor([[1, 2],
       [3, 4]])
Shape: torch.Size([2, 2])
3D Tensor (Cube):
tensor([[[1, 2],
         [3, 4],


         [[11, 12],
          [13, 14]],

         [[-4, -3],
          [-2, -1]]]])
Shape: torch.Size([3, 2, 2])
4D Tensor (Vector of Cubes):
tensor([[[[1, 2],
          [3, 4],


          [[11, 12],
           [13, 14]],

          [[-4, -3],
           [-2, -1]]],


          [[[101, 102],
            [103, 104]],

            [[111, 112],
             [113, 114]],

             [[96, 97],
              [98, 99]]]]])
Shape: torch.Size([2, 3, 2, 2])

```

```

5D Tensor (Matrix of Cubes):
tensor([[[[[
    [[ 1,    2],
     [ 3,    4]],

    [[ 11,   12],
     [ 13,   14]],

    [[ -4,   -3],
     [ -2,   -1]]],,

    [[[ 101,   102],
     [ 103,   104]],

    [[ 111,   112],
     [ 113,   114]],

    [[ 96,    97],
     [ 98,    99]]],,

    [[[1001,  1002],
     [1003,  1004]],

    [[1011,  1012],
     [1013,  1014]],

    [[ 996,   997],
     [ 998,   999]]],,

    [[[1101,  1102],
     [1103,  1104]],

    [[1111,  1112],
     [1113,  1114]],

    [[1096,  1097],
     [1098,  1099]]]]])
Shape: torch.Size([2, 2, 3, 2, 2])

```

```
tensor([[1, 2],
        [3, 4],
        [5, 6]], dtype=torch.int32)
tensor([[-0.2469, -0.8365,  0.9204],
       [ 1.4158, -0.9651,  0.7520]], device='cuda:0')
```

3.2 张量属性

如下可以显示一些张量的常见属性，可以由结果看出这是一个 2x3 共 6 个元素的二维浮点张量、支持自动求导、存在于 GPU 上、内存连续

```
import torch

# 创建一个示例张量
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

x = torch.tensor([[1, 2, 3], [4, 5, 6]], dtype=torch.float32,
                 requires_grad=True, device=device)

print(x)

# 打印常见属性
print("\n--- ATTRIBUTES ---")

print("Shape:", x.shape) # 获取形状
print("Size:", x.size()) # 获取尺寸
print("Data Type:", x.dtype) # 数据类型
print("Device:", x.device) # 设备
print("Dimensions:", x.dim()) # 维度数
print("Total Elements:", x.numel()) # 元素总数
print("Requires Grad:", x.requires_grad) # 是否启用梯度
print("Is CUDA:", x.is_cuda) # 是否在 GPU 上
print("Is Contiguous:", x.is_contiguous()) # 是否连续存储
```

```
tensor([[1., 2., 3.],
       [4., 5., 6.]], device='cuda:0', requires_grad=True)

--- ATTRIBUTES ---
Shape: torch.Size([2, 3])
Size: torch.Size([2, 3])
Data Type: torch.float32
Device: cuda:0
Dimensions: 2
Total Elements: 6
Requires Grad: True
Is CUDA: True
Is Contiguous: True
```

3.3 张量操作

如下对张量进行了一些常见操作，相加，逐元素乘，求张量所有元素的和，矩阵乘法，张量的转置，张量的展平，显示张量的形状

```
import torch
a = torch.randn(3, 4)
b = torch.randn(3, 4)
print('a:\n', a)
print('b:\n', b)
print('a + 5:\n', a+5)
print('b * 2:\n', b*2)
print('b元素的和:\n', b.sum())
print('a + b逐元素相加:\n', a+b)
print('a * b逐元素相乘:\n', a*b)
print('a, b矩阵乘法:\n', torch.matmul(a, b.T))
print('a转置:\n', a.t())
print('a展平:\n', a.flatten())
print('a:\n', a.shape)
```

```

a:
tensor([[-0.4372, -1.1626, -0.7771, -1.7675],
       [-0.5640, -0.0212,  2.0599,  1.8697],
       [ 0.8934, -1.4974, -0.6858,  1.2614]]))

b:
tensor([[ 0.7398,  1.3804, -0.1424, -0.0716],
       [-0.5329,  1.4955,  0.4823, -0.5611],
       [ 0.4296, -0.4913, -0.4531, -2.1708]]))

a + 5:
tensor([[4.5628, 3.8374, 4.2229, 3.2325],
       [4.4360, 4.9788, 7.0599, 6.8697],
       [5.8934, 3.5026, 4.3142, 6.2614]]))

b * 2:
tensor([[ 1.4796,  2.7608, -0.2848, -0.1432],
       [-1.0657,  2.9910,  0.9645, -1.1222],
       [ 0.8593, -0.9826, -0.9062, -4.3416]]))

b元素的和:
tensor(0.1044)

a + b逐元素相加:
tensor([[ 0.3025,  0.2178, -0.9195, -1.8391],
       [-1.0969,  1.4743,  2.5422,  1.3086],
       [ 1.3230, -1.9887, -1.1389, -0.9094]]))

a * b逐元素相乘:
tensor([[[-0.3235, -1.6049,  0.1107,  0.1266],
       [ 0.3005, -0.0317,  0.9934, -1.0491],
       [ 0.3838,  0.7357,  0.3107, -2.7382]]])

a, b矩阵乘法:
tensor([[[-1.6911, -0.8887,  4.5724],
       [-0.8737,  0.2131, -5.2239],
       [-1.3987, -3.7539, -1.3080]]])

a转置:
tensor([[-0.4372, -0.5640,  0.8934],
       [-1.1626, -0.0212, -1.4974],
       [-0.7771,  2.0599, -0.6858],
       [-1.7675,  1.8697,  1.2614]]))

a展平:
tensor([-0.4372, -1.1626, -0.7771, -1.7675, -0.5640, -0.0212,
        2.0599,  1.8697,
        0.8934, -1.4974, -0.6858,  1.2614])

a:
torch.Size([3, 4])
```

3.4 自动求导

pytorch 提供了自动求导功能，将 `requires_grad` 设置为 `True` 后，pytorch 就会自动跟踪该张量的所有操作，以便求导。通过反向传播`.backward()`之后就可以计算梯度了，如下

```
import torch
# 创建一个需要计算梯度的张量
x = torch.randn(2, 2, requires_grad=True)
print(x)

# 执行某些操作

y = x + 2
z = y * y * 3
out = z.mean()

print(out)

# 反向传播，计算梯度
out.backward()

# 查看 x 的梯度
print(x.grad)
```

3.5 神经网络

pytorch 提供了 `torch.nn.Module` 接口来构建神经网络模型
一下可以创建一个简单的神经网络，创建实例并打印模型结构

```
import torch
import torch.nn as nn
import torch.optim as optim

# 定义一个简单的全连接神经网络
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(2, 2) # 输入层到隐藏层
        self.fc2 = nn.Linear(2, 1) # 隐藏层到输出层

    def forward(self, x):
        x = torch.relu(self.fc1(x)) # ReLU 激活函数
        x = self.fc2(x)
        return x

# 创建网络实例
model = SimpleNN()

# 打印模型结构
print(model)
```

有两个全连接层，输入特征数都是 2，输出特征数都是 1，都设有偏置项

```
hwq@hwq:~$ time curl https://missing.csail.mit.edu &> /dev/null
real    0m1.253s
user    0m0.019s
sys     0m0.016s
```

3.6 训练模型

在刚刚的神经网络的基础上，加上损失函数和优化器，可以开始训练循环，每次轮换时都清空之前的梯度，将输入数据通过模型传递，计算预测输出，接着用损失函数评估预测输出与真实标签之间的差，利用自动求导计算损失相对于模型参数的梯度，根据计算出的梯度和优化器的策略更新模型参数

```

import torch
import torch.nn as nn
import torch.optim as optim

# 1. 定义一个简单的神经网络模型
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(2, 2) # 输入层到隐藏层
        self.fc2 = nn.Linear(2, 1) # 隐藏层到输出层

    def forward(self, x):
        x = torch.relu(self.fc1(x)) # ReLU 激活函数
        x = self.fc2(x)
        return x

# 2. 创建模型实例
model = SimpleNN()

# 3. 定义损失函数和优化器
criterion = nn.MSELoss() # 均方误差损失函数
optimizer = optim.Adam(model.parameters(), lr=0.001) # Adam 优化器

# 4. 假设我们有训练数据 X 和 Y
X = torch.randn(10, 2) # 10 个样本, 2 个特征
Y = torch.randn(10, 1) # 10 个目标值

# 5. 训练循环
for epoch in range(100): # 训练 100 轮
    optimizer.zero_grad() # 清空之前的梯度
    output = model(X) # 前向传播
    loss = criterion(output, Y) # 计算损失
    loss.backward() # 反向传播
    optimizer.step() # 更新参数

    # 每 10 轮输出一次损失
    if (epoch+1) % 10 == 0:
        print(f'Epoch [{epoch+1}/100], Loss: {loss.item():.4f}')

```

```
[Running] python -u "d:\Pythoncode\pytpratice\train.py"
Epoch [10/100], Loss: 0.7212
Epoch [20/100], Loss: 0.7134
Epoch [30/100], Loss: 0.7061
Epoch [40/100], Loss: 0.6997
Epoch [50/100], Loss: 0.6942
Epoch [60/100], Loss: 0.6894
Epoch [70/100], Loss: 0.6852
Epoch [80/100], Loss: 0.6814
Epoch [90/100], Loss: 0.6779
Epoch [100/100], Loss: 0.6746
```

3.7 图片识别

对前面学习的综合
代码过长仅展示代码运行结果
先运行 data.py 下载数据集

```
[Running] python -u "d:\Pythoncode\pyt\data.py"
Files already downloaded and verified
Files already downloaded and verified
```

运行 train.py，完成训练并进行评估

```
[--, 1000] loss: 0.828
[10, 6000] loss: 0.840
[10, 8000] loss: 0.853
[10, 10000] loss: 0.873
[10, 12000] loss: 0.879
Files already downloaded and verified
Finished Training
Files already downloaded and verified
Accuracy on test images: 61.13%
Files already downloaded and verified
Accuracy of plane: 63.40%
Accuracy of car : 77.90%
Accuracy of bird : 44.20%
Accuracy of cat : 31.30%
Accuracy of deer : 40.40%
Accuracy of dog : 68.20%
Accuracy of frog : 78.00%
Accuracy of horse: 71.70%
Accuracy of ship : 65.90%
Accuracy of truck: 70.30%
```

执行 predict.py 对 test 中的图片进行识别

```
Predicted:  dog    frog   dog    dog
```

4 实验心得

本次实验我学习了调试和性能分析的多种方式，已经如何使用 Makefile 文件和 make 命令来构建目标产物，并且接触了 pytorch 编程和神经网络。确实本课教给我了一些平时会用到但是并不很了解的东西，让我受益匪浅。