海纳百川
取则行远

中国海洋大学
OCEAN UNIVERSITY OF CHINA

Week4

左昊天

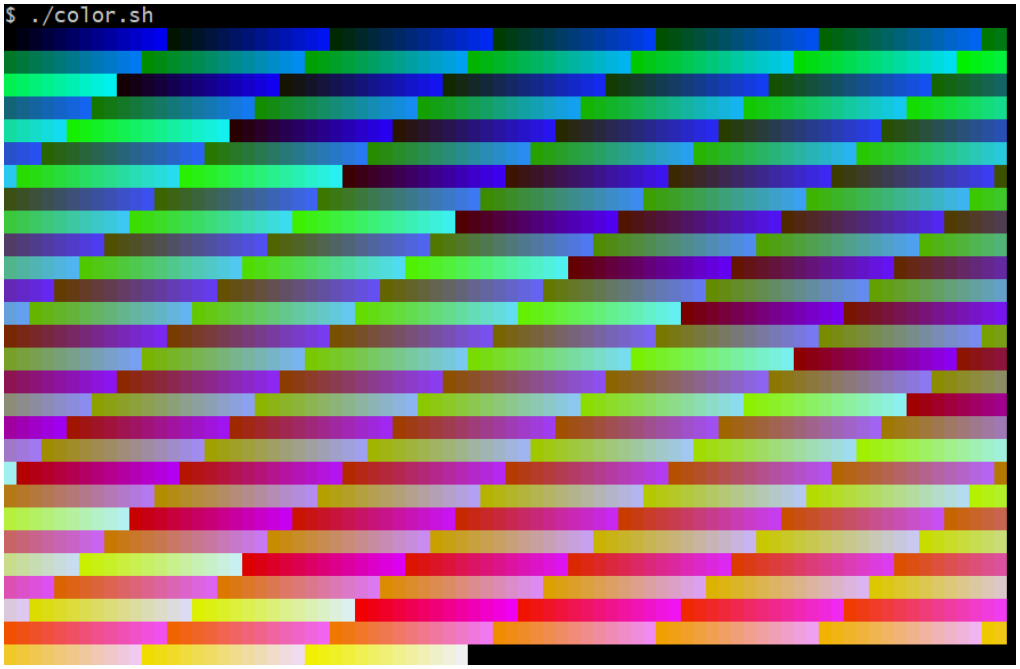
2024-09-13

Github 仓库地址

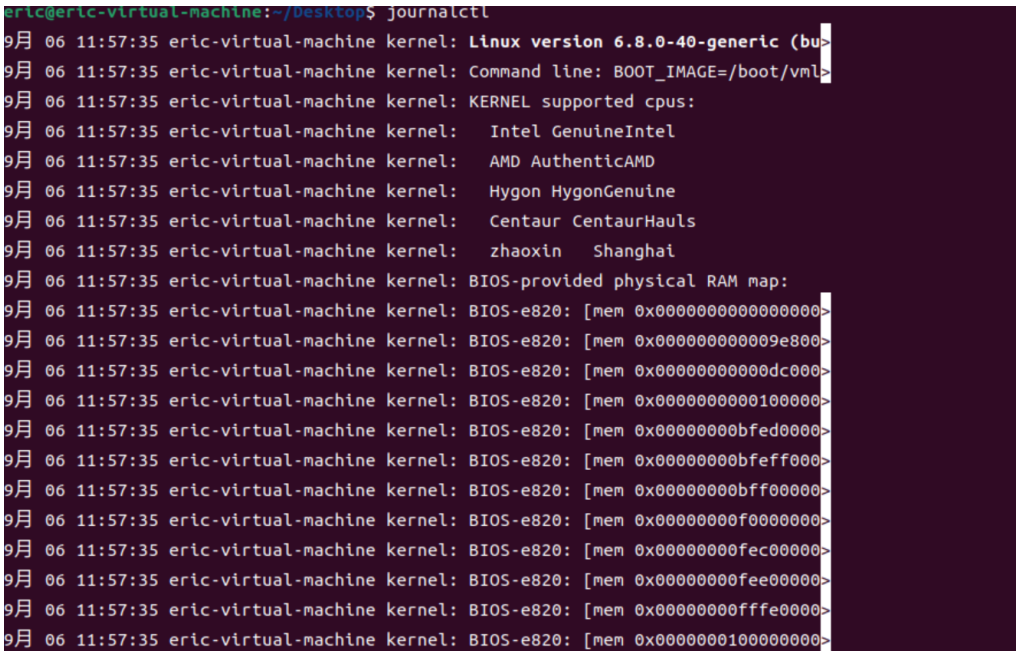
1 调试与性能分析

1.1 打印颜色

```
#!/usr/bin/env bash
for R in $(seq 0 20 255); do
    for G in $(seq 0 20 255); do
        for B in $(seq 0 20 255); do
            printf "\e[38;2;${R};${G};${B}m \e[0m";
        done
    done
done
```



1.2 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。



1.3 pdb 调试器

l(ist) - 显示当前行附近的 11 行或继续执行之前的显示；

s(tep) - 执行当前行，并在第一个可能的地方停止；

n(ext) - 继续执行直到当前函数的下一条语句或者 return 语句；

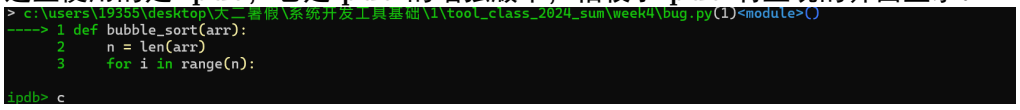
b(reak) - 设置断点（基于传入的参数）；

p(rint) - 在当前上下文对表达式求值并打印结果。还有一个命令是 pp ，它使用 pprint 打印；

r(eturn) - 继续执行直到当前函数返回；

q(uit) - 退出调试器。

这里使用的是 ipdb，它是 pdb 的增强版本，相较于 pdb 有直观的界面显示。



c 命令代表继续执行，直到出现错误。

```
Traceback (most recent call last):
  File "C:\Users\19355\AppData\Local\Programs\Python\Python310\lib\site-packages\ipdb\__main__.py", line 323, in main
    pdb._runscript(mainpyfile)
  File "C:\Users\19355\AppData\Local\Programs\Python\Python310\lib\pdb.py", line 1592, in _runscript
    self.run(statement)
  File "C:\Users\19355\AppData\Local\Programs\Python\Python310\lib\bdb.py", line 597, in run
    exec(cmd, globals, locals)
  File "<string>", line 1, in <module>
  File "c:\users\19355\desktop\大二暑假\系统开发工具基础\1\tool_class_2024_sum\week4\bug.py", line 10, in <module>
    print(bubble_sort([4, 2, 1, 8, 7, 6]))
  File "c:\users\19355\desktop\大二暑假\系统开发工具基础\1\tool_class_2024_sum\week4\bug.py", line 5, in bubble_sort
    if arr[j] > arr[j+1]:
IndexError: list index out of range
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
> c:\users\19355\desktop\大二暑假\系统开发工具基础\1\tool_class_2024_sum\week4\bug.py(5)bubble_sort()
      4         for j in range(n):
----> 5             if arr[j] > arr[j+1]:
      6                 arr[j] = arr[j+1]
```

使用 p locals() 命令，可以查看此时所有变量的值，用此进行错误分析。

```
ipdb> p locals()
{'arr': [2, 1, 1, 7, 6, 6], 'n': 6, 'i': 0, 'j': 5}
ipdb>
```

1.4 计时

使用 time 库中的 time 函数

```python
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)
```

```
Sleeping for 800 ms
0.813072919845581
```

1.5 memory-profiler

使用 memory-profiler 模块对内存进行监控。

实例代码：

```python
@profile
def my_func():
    a = [1] * (10 ** 6)
```

```
4       b = [2] * (2 * 10 ** 7)
5       del b
6       return a
7
8   if __name__ == '__main__':
9       my_func()
```

```
PS C:\Users\19355\Desktop\大二暑假\tools\1\tool_class_2024_sum\week4\python> python -m memory_profiler 内存.py
Filename: 内存.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     1   45.453 MiB   45.453 MiB           1   @profile
     2                                         def my_func():
     3   53.090 MiB    7.637 MiB           1       a = [1] * (10 ** 6)
     4  205.680 MiB  152.590 MiB           1       b = [2] * (2 * 10 ** 7)
     5   53.090 MiB -152.590 MiB           1       del b
     6   53.090 MiB    0.000 MiB           1       return a
```

1.6 shellcheck

shellcheck 是一个静态的脚本分析工具，它可以得到程序错误的原因。

```
1       shellcheck 文件名
```

```
$ shellcheck test.sh

In test.sh line 1:
echo "Hello World"
^-- SC2148 (error): Tips depend on target shell and yours is unknown. Add a sheb
ang or a 'shell' directive.
```

此代码的问题是开头没有声明使用的 shell 类型。需加上`#!/bin/bash`

1.7 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。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。fibN 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？

```
1   #!/usr/bin/env python
2   def fib0(): return 0
3
4   def fib1(): return 1
```

```
5
6  s = """def fib{}(): return fib{}() + fib{}()"""
7
8  if __name__ == '__main__':
9
10     for n in range(2, 10):
11         exec(s.format(n, n-1, n-2))
12     # from functools import lru_cache
13     # for n in range(10):
14     #     exec("fib{} = lru_cache(1)(fib{})".format(n, n
           ))
15     print(eval("fib9()"))
```

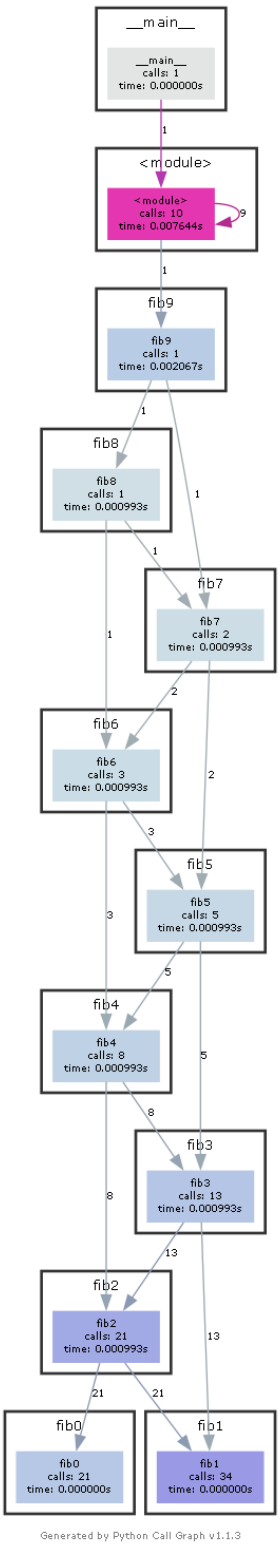


图 1: 放开注释前

被调用 101 次

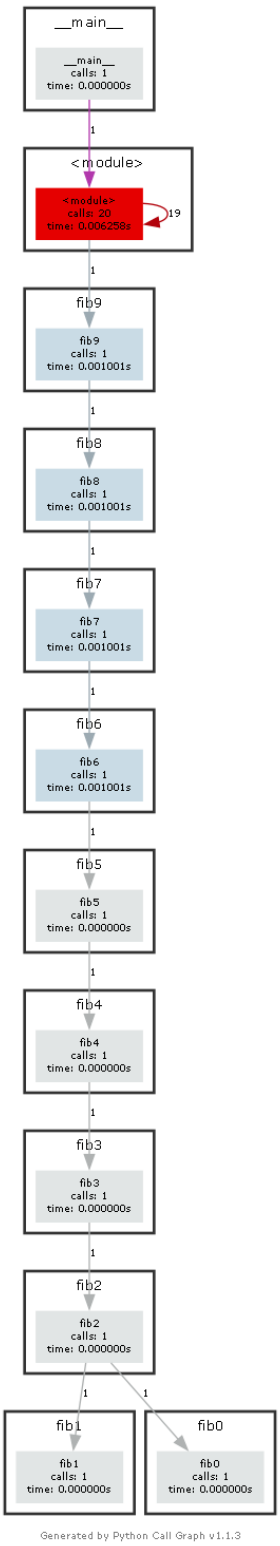


图 2: 放开注释后

被调用 10 次

1.8 遇到的问题：无法安装 pycallgraph

一开始根据提示，尝试回退 setuptools 版本，发现没有用。在网上不断搜索，找到解决方法，安装时用 pip install pycallgraph2 命令就可以安装了。

2 元编程

3 pytorch

3.1 引入库

```
from __future__ import print_function
import torch
```

注：以下代码均省略引入库的代码。

3.2 随机生成 a × b 的矩阵

```
rand_x = torch.rand(a, b)
print(rand_x)
```

```
tensor([[0.6003, 0.8039, 0.1552],
        [0.3492, 0.8825, 0.4327],
        [0.4235, 0.1198, 0.3892],
        [0.0445, 0.6159, 0.7562],
        [0.2491, 0.7200, 0.9900]])
```

（此为 5×3 的矩阵）

3.3 创建数值都是 0, 类型为 long 的矩阵

```
zero_x = torch.zeros(a, b, dtype=torch.long)
print(zero_x)
```

```
tensor([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]])
```

3.4 保留相同的尺寸大小，并修改数据类型

```
tensor1 = torch.zeros(3, 4, dtype=torch.long)
tensor2 = torch.randn_like(tensor1, dtype=torch.float)
```

```
print('tensor1: ', tensor1)
print('tensor2: ', tensor2)
```

```
tensor1:  tensor([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]])
tensor2:  tensor([[-0.6876, -0.6814,  0.5686, -0.2144],
        [-1.5013,  0.3748,  2.0203,  0.2432],
        [-0.1230,  0.6541,  0.6294, -0.1987]])
```

3.5 修改 tensor 的尺寸

```
x = torch.randn(3, 6)
y = x.view(18)
# -1 表示除给定维度外的其余维度的乘积
z = x.view(-1, 9)
print(x.size(), y.size(), z.size())
```

解释：z 矩阵的列计算方式：

x 是 4×4 的，所以总共有 18 个。z 的列乘以行数（9）等于 18，所以 z 的列数为 2。

```
torch.Size([3, 6]) torch.Size([18]) torch.Size([2, 9])
```

3.6 Tensor 的加法

三种方式：

1、+

2、`torch.add(tensor1, tensor2)`

3、`tensor1.add_(tensor2)`，该方法会直接修改 tensor1 变量的值

```
tensor1=  tensor([[-1.9233,  0.6145, -0.4592,  0.3885],
        [ 1.0400, -0.0471,  1.8407, -0.8701],
        [-0.5764,  0.2761,  0.5713,  1.2667]])
tensor2=  tensor([[0.7943, 0.2186, 0.0189, 0.5150],
        [0.8489, 0.2613, 0.5728, 0.9321],
        [0.7139, 0.9534, 0.1504, 0.1818]])
tensor1 + tensor2=  tensor([[-1.1290,  0.8331, -0.4402,  0.9035],
        [ 1.8889,  0.2141,  2.4136,  0.0619],
        [ 0.1375,  1.2295,  0.7218,  1.4485]])
tensor1 + tensor2=  tensor([[-1.1290,  0.8331, -0.4402,  0.9035],
        [ 1.8889,  0.2141,  2.4136,  0.0619],
        [ 0.1375,  1.2295,  0.7218,  1.4485]])
add result=  tensor([[-1.1290,  0.8331, -0.4402,  0.9035],
        [ 1.8889,  0.2141,  2.4136,  0.0619],
        [ 0.1375,  1.2295,  0.7218,  1.4485]])
tensor1=  tensor([[-1.1290,  0.8331, -0.4402,  0.9035],
        [ 1.8889,  0.2141,  2.4136,  0.0619],
        [ 0.1375,  1.2295,  0.7218,  1.4485]])
tensor2=  tensor([[0.7943, 0.2186, 0.0189, 0.5150],
        [0.8489, 0.2613, 0.5728, 0.9321],
        [0.7139, 0.9534, 0.1504, 0.1818]])
```

3.7 Tensor 转换为 Numpy 数组

```python
a = torch.ones(5)
print(a)
b = a.numpy()
print(b)
```

```
tensor([1., 1., 1., 1., 1., 1.])
[1. 1. 1. 1. 1. 1.]
```

3.8 Numpy 数组转换为 Tensor

```python
import numpy as np
a = np.ones(5)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)
```

```
[2. 2. 2. 2. 2. 2.]
tensor([2., 2., 2., 2., 2., 2.], dtype=torch.float64)
```

3.9 CUDA 张量

```python
x = torch.randn(4, 4)

# 当 CUDA 可用的时候，可以使用下方这段代码，采用 torch.device() 方法来改变 tensors 是否在 GPU 上进行计算操作
if torch.cuda.is_available():
    device = torch.device("cuda")          # 定义一个 CUDA 设备对象
    y = torch.ones_like(x, device=device)  # 创建一个在 GPU 上的 tensor
    x = x.to(device)                       # 将 x 移动到 GPU
    z = x + y
    print(z)
    print(z.to("cpu", torch.double))       # 将 z 移动到 CPU 并改变数值类型
else:
    print("CUDA is not available")
```

```
tensor([[ 1.1667, -0.2021,  2.3083,  1.7255],
        [ 0.7949,  0.0281, -0.4448, -0.2990],
        [ 2.3527,  0.0554,  1.7378,  2.1257],
        [ 0.5861,  0.4349,  0.6697,  0.3967]], device='cuda:0')
tensor([[ 1.1667, -0.2021,  2.3083,  1.7255],
        [ 0.7949,  0.0281, -0.4448, -0.2990],
        [ 2.3527,  0.0554,  1.7378,  2.1257],
        [ 0.5861,  0.4349,  0.6697,  0.3967]], dtype=torch.float64)
```

第一个结果是在 gpu 上运行的，第二个结果是在 cpu 上运行的。