python_practice Documentation

发布 *v1.0*

yuanjh

2020 年 10 月 31 日

Contents:

python 研发过程中需要用到工具，注意的坑等等

CHAPTER 1

python 实战 01FFT 快速傅里叶变换

本科小作业，快速傅里叶变换

python3.2

```
# Copyright (C) 2011-2012 YuanJh
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see <http://www.gnu.org/licenses/>.

"""yuanJh's fft and ifft module.
the detail about fft you can see "http://en.wikipedia.org/wiki/Fast_Fourier_transform"
Main functions:
  fft(a)
  ifft(a)
```

(下页继续)

(续上页)

```
"""
from math import *
from cmath import *
import sys
from numpy import *
pi=3.1414926

#version infomation
__version__ = '1.0'

#################### function recursive FFT ####################
def RECURSIVE_FFT(a):
    "Use recursive method to realize FFT"
    n=len(a)
    if n==1:
        return a
    wn=complex(cos(-2*pi/n),sin(-2*pi/n))    #create the complex number wn=cos(-2pi/
↪n)+sin(-2pi/n)i
    w=1

    a0=[]
    a1=[]
    for i in range(0,n-1,2):
        a0.append(a[i])        #pick up the even number
        a1.append(a[i+1])    #pick up the odd number

    y0=RECURSIVE_FFT(a0)    #translate the even part
    y1=RECURSIVE_FFT(a1)    #translate the odd part

    y=[0 for i in range(n)]
    for k in range(0,int(n/2)):    #combine the even part and the odd part
        y[k]=y0[k]+w*y1[k]
        y[k+int(n/2)]=y0[k]-w*y1[k]
        w*=wn
    return y
#################### end function recursive FFT ####################

#################### function recursive IFFT ####################
def RECURSIVE_IFFT(a):
    "Use recursive method to realize IFFT"
```

(下页继续)

(续上页)

```
    n=len(a)
    if n==1:
        return a
    wn=complex(cos(2*pi/n),sin(2*pi/n))    #create the complex number wn=cos(2pi/
↪n)+sin(2pi/n)i
    w=1

    a0=[]
    a1=[]
    for i in range(0,n-1,2):
        a0.append(a[i])        #pick up the even number
        a1.append(a[i+1])    #pick up the odd number

    y0=RECURSIVE_IFFT(a0)
    y1=RECURSIVE_IFFT(a1)

    y=[0 for i in range(n)]
    for k in range(0,int(n/2)):    #combine the even part and the odd part
        y[k]=y0[k]+w*y1[k]
        y[k+int(n/2)]=y0[k]-w*y1[k]
        w*=wn
    return y
##################### end function recursive IFFT ####################

##################### got data and verify the program ####################
a=input("please input the number (use ',' split):")
a=a.split(',')
n=len(a)
#tempn=ceil(log2(n))
#tempn=tempn**2

for i in range(0,n):
    a[i]=int(a[i])

##################### verify the FFT ####################
print ("the results of my program fft")
b=RECURSIVE_FFT(a)
for i in range(0,n):
    print ("%4d: %4.4f+%4.4fi"%(a[i],b[i].real,b[i].imag))
```

(下页继续)

(续上页)

```
print ("the results of the numpy program fft")    #Use the function in the package numpy
b=fft.fft(a)
for i in range(0,n):
    print ("%4d: %4.4f+%4.4fi"%(a[i],b[i].real,b[i].imag))

#################### verify the IFFT ####################
#test the my ifft program is right or not by compare with the ifft function in numpy␣
↪package
print ("the results of my program ifft")
lenb=len(b)
a=RECURSIVE_IFFT(b)
for i in range(0,n):
    print("%4.4f+%4.4fi: %4.4f+%4.4fi"%(b[i].real,b[i].imag,a[i].real/lenb,a[i].imag/
↪lenb))

print ("the results of the numpy program ifft")
a=fft.ifft(b)
for i in range(0,n):
    print("%4.4f+%4.4fi: %4.4f+%4.4fi"%(b[i].real,b[i].imag,a[i].real,a[i].imag))

文件名: fft.py
实现语言: PYTHON
1, 系统概述
利用递归的方法实现快速傅里叶变化 (FFT) 和快速傅里叶逆变换 (IFFT)。
2, 模块层次
主要包含三部分:
    1, 递归实现快速傅里叶变换 (FFT)。
    2, 快速傅里叶逆变换 (IFFT)。
    3, 数据录入以及结果正确性验证。
3, 函数声明表
def RECURSIVE_FFT(a):
    功能说明: 递归实现快速傅里叶变换。
    参数说明:
        a: 需要进行快速傅里叶变换的数据, 以列表 (类似 C,JAVA 中数组) 形式存储。
def RECURSIVE_IFFT(a):
    功能说明: 递归实现快速傅里叶逆变换,
    参数说明:
        a: 需要进行快速傅里叶逆变换的数据, 以列表 (类似 C,JAVA 中数组) 形式存储。

filename:fft.py
```

(下页继续)

(续上页)

```
language:python
1,System overview
Using recursive method to realize Fast Fourier Transform (FFT) and inverse Fast Fourier␣
↪Transform (IFFT).
2,Module and level
Mainly includes three parts:
    1, The recursive Fast Fourier Transform(FFT).
    2, Inverse Fast Fourier Transform(IFFT).
    3, Got data and verify the results.
3,Function declaration table
def RECURSIVE_FFT (a):
    Function introduction:recursive realize Fast Fourier Transform.
    Parameters:
        a: the data need for Fast Fourier Transform store as a list (similar to C, JAVA␣
↪the array).
def RECURSIVE_IFFT (a):
    Function introduction: recursive inverse Fast Fourier Transform.
    Parameters:
        a: the data need for inverse Fast Fourier Transform store as a list.
```

CHAPTER 2

python 实战 02 异常报错

2.1 百度自然语言处理报错：UnicodeEncodeError: ‘gbk’ codec can’t encode character ‘\U0001f602’ in posit

原因：ubuntu 机器编码 utf8，接口尝试用 gbk 解析，在对 str 转换格式时报错

解决：比如 title 变量.str(title.encode(“GBK”,’ignore’))

2.2 解决方案 ObjectOfTypeXXisNotJSONSerializable

对象的部分字段不支持序列化.

01,

```
print(json.dumps(obj.__dict__, indent=4, sort_keys=True, default=str))
参考：https://stackoverflow.com/questions/57671281/python-json-dumps-unable-to-ignore-
↪non-serializable-objects#comment101791277_57671370
```

02,

```
def set_default(obj):
    if isinstance(obj, set):
        return list(obj)
```

(下页继续)

(续上页)

```
    raise TypeError

result = json.dumps(yourdata, default=set_default)
```

参考: http://www.ojit.com/article/95292

03,

```
result=eval(repr(results))
json = simplejson.dumps({'results':result,'retrieveStyle': 'distRetrieve', 'status': 'ok
↪'})
```

参考; https://blog.csdn.net/qq_20373723/article/details/68961099

04,

```
import json

class Object:
    def toJSON(self):
        return json.dumps(self, default=lambda o: o.__dict__,
            sort_keys=True, indent=4)
```

参考: https://www.codenong.com/3768895/

05,

```
import json

class FileItem:
    def __init__(self, fname):
        self.fname = fname

    def __repr__(self):
        return json.dumps(self.__dict__)
```

参考: https://www.codenong.com/3768895/

最终推荐方案:

```
json.dumps([x.__dict__ for x in all_results], default=str)
```

2.3 解决方案 UnicodeEncodeError

又遇到报错：

UnicodeEncodeError: 'ascii' codec can't encode character u'\uff08' in position 13: ordinal not in range(128)

这个问题遇到多次了了，但都是通过第一个方案解决了，但这次貌似不行了。最终采用了方案三，顺便整理下网上其他方案

第一种方案（90% 情况下，大部分帖子都是这个）

一般报错到代码都是自己写到代码，代码上添加

```
import sys
reload(sys)
sys.setdefaultencoding('utf-8')
```

第二种方案，引用到包出现错误（未解决我的问题，但有人提到过这种处理方案）

在 python 的 lib\site-packages 文件夹下新建一个 sitecustomize.py

cat sitecustomize.py # 添加如下内容，设置编码为 utf8

```
#encoding=utf8
import sys
reload(sys)
sys.setdefaultencoding('utf8')
```

参考：https://www.cnblogs.com/kevingrace/p/5893121.html

第三种方案进入 python 终端，执行如下命令

import sys, codecs, locale; print str(sys.stdout.encoding);

是否时 utf8（ubunut 系统）

如果不是，比如我的是这个

'ANSI_X3.4-1968'

则修改环境变量 PYTHONIOENCODING 为 utf8

执行：export PYTHONIOENCODING=utf-8

2.4 解决方案 ImportErrorlibta_lib

原始问题：

```
>>> import talib
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/root/anaconda3/envs/vnpy27/lib/python2.7/site-packages/talib/__init__.py", line␣
↪4, in <module>
    from . import common
ImportError: libta_lib.so.0: cannot open shared object file: No such file or directory
```

附录：官方的安装方法

```
pip install TA-Lib
参考：https://github.com/mrjbq7/ta-lib
```

2.4.1 方案 01,export

参考：https://github.com/mrjbq7/ta-lib/issues/6

```
export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH

等价做法：
$ LD_LIBRARY_PATH="/usr/local/lib:$LD_LIBRARY_PATH" python
>>> import talib
```

2.4.2 方案 02, 源代码安装

参考：https://stackoverflow.com/questions/45406361/importerror-libta-lib-so-0-cannot-open-shared-object-file-no-such-file-or-dir

https://ideaorchard.wordpress.com/2015/01/16/installing-ta-lib-ubuntu/

```
I had the same issue. See below for what I did to fix it.

installing

wget http://prdownloads.sourceforge.net/ta-lib/ta-lib-0.4.0-src.tar.gz
tar -xzf ta-lib-0.4.0-src.tar.gz
cd ta-lib/
./configure --prefix=/usr
make
Sudo make install
```

(下页继续)

(续上页)

```
pip install numpy
If you don't have it installed

pip install TA-Lib
if you do have it installed

pip install --upgrade --force-reinstall TA-Lib
```

2.4.3 方案 3，conda 安装（这个一般比较好使）

```
sudo chmod -R 777 anaconda3
conda install -chttps://conda.anaconda.org/quantopian ta-lib
```

2.4.4 方案 4，conda 安装 talib，冲突的 numpy 自动卸载

之后 pip 卸载 numpy，再 pip 自动安装 numpy，安装时指定版本 ==1.12.0

2.5 python 绘制 k 线图 (蜡烛图) 报错 No module named ‘matplotlib.finance

使用 python 绘制蜡烛图报错：No module named ‘matplotlib.finance

部分版本移除了 finance 模块，需要独立安装

安装命令：pip install git+https://github.com/matplotlib/mpl_finance.git

其他参考：https://blog.csdn.net/u014281392/article/details/73611624

https://www.jb51.net/article/140799.htm

2.6 坑 map 函数不执行

常用 series.map() 函数，回头使用 map(lamba x:func(x),listxx)，容易忘记 map 是生成式，如果不调用 next 不会触发实际运行，所以需要联用 list(map(xx,yy)) 才有意义。

2.7 坑文件读写方式和乱码问题

java 中文件读写打开为“rw”，但 python 中“rw”并非有效配置，会报错

```
ValueError: must have exactly one of create/read/write/append mode
```

使用’r+’代替’rw’读写模式.

其次读写同一文件时，如果写入从文件开头覆盖式写入则需要按照如下方式

```
content = ''.join(f.readlines())
content.replace(' ','')
f.seek(0)
f.truncate()
f.seek(0)
f.write(content)
```

CHAPTER 3

python 实战 03 为 list 实现 find 方法

string 类型的话可用 find 方法去查找字符串位置：

```
a_list.find('a')
```

如果找到则返回第一个匹配的位置，如果没找到则返回-1，而如果通过 index 方法去查找的话，没找到的话会报错。

如果我们希望在 list 中也使用 find 呢?

3.1 方法 1，独立函数法

```
def list_find(item_list, find_item):
    if find_item in item_list:
        return item_list.index(find_item)
    return -1

item_list=[1,2,3]
print(list_find(item_list,1),list_find(item_list,4))
```

缺点：代码太多，麻烦

3.2 方法 2,if 三元表达式 (本质同上)

```
item_list.index(find_item) if find_item in item_list else -1
```

优点：简单，明了

缺点:item_list 在上面出现两次，想想一下，如果 item_list 是一个比较长表达式的结果 (或者函数结果)，则会导致代码过长，且会执行 2 次

3.3 方法 3,next(利用迭代器遍历的第二个参数)

```
next((item for item in item_list if item==find_item ),-1)
```

缺点：如果对迭代器不熟悉，不大好理解

优点：扩展性好，if 后面的条件可以不只是相等，可支持更为复杂的逻辑判断

3.4 方法 4,list 元素 bool 类型

```
''.join(map(str, map(int, item_list))).find(str(int(True)))
```

简单容易理解

3.5 参考

python 中 list 的五种查找方法：https://blog.csdn.net/qq_31747765/article/details/80944227

python list 查找与过滤方法整合（查找第一个匹配项:next，重复时想要所有的索引:enumerate）：https://blog.csdn.net/qq997843911/article/details/93855706

CHAPTER 4

python 实战 04 常见坑

何谓坑：凡是和**大多数人的直观理解不一致**的，都可称之为“坑”

一部分在其他博文中提到过，不再重复

比如：

```
默认参数最好不为可变对象
时有时无的切片异常
不执行的 del
return 不一定都是函数的终点
```

博文:python 进阶 17 炫技巧, 中提过，所以不再重复

博文：python 阅读 wtfbook 疑问和验证，也记录了一些坑，也不再重复

本文作为对以上的补充

4.1 括号和元组 ()

单个元素要被识别为元组，必须在括号后面加个逗号, 详见如下代码：

```
In [65]: a = (10)

In [66]: type(a)
Out[66]: int
```

(下页继续)

(续上页)

```
In [67]: b = (10,)

In [68]: type(b)
Out[68]: tuple
```

4.2 空集合空字典 {}

创建集合与字典，它们都用一对 {}，但是默认返回字典，而不是集合。要想创建空集合，可使用内置函数 set()

```
In [69]: d = {}

In [70]: type(d)
Out[70]: dict

In [71]: s = set()

In [72]: type(s)
Out[72]: set
```

4.3 默认参数最好不为可变对象

函数的参数分三种 - 可变参数 - 默认参数 - 关键字参数

当你在传递默认参数时，有新手很容易踩雷的一个坑。

先来看一个示例

```
def func(item, item_list=[]):
    item_list.append(item)
    print(item_list)

func('iphone')
func('xiaomi', item_list=['oppo','vivo'])
func('huawei')
```

在这里，你可以暂停一下，思考一下会输出什么?

思考过后，你的答案是否和下面的一致呢

```
['iphone']
['oppo', 'vivo', 'xiaomi']
['iphone', 'huawei']
```

如果是，那你可以跳过这部分内容，如果不是，请接着往下看，这里来分析一下。

Python 中的 def 语句在每次执行的时候都初始化一个函数对象，这个函数对象就是我们要调用的函数，可以把它当成一个一般的对象，只不过这个对象拥有一个可执行的方法和部分属性。

对于参数中提供了初始值的参数，由于 Python 中的函数参数传递的是对象，也可以认为是传地址，在第一次初始化 def 的时候，会先生成这个可变对象的内存地址，然后将这个默认参数 item_list 会与这个内存地址绑定。在后面的函数调用中，如果调用方指定了新的默认值，就会将原来的默认值覆盖。如果调用方没有指定新的默认值，那就会使用原来的默认值。

第一次调用时，会执行初始化，生成[] 的内存地址是：2830870084744
第二次调用时，指定了新的默认对象（2830874211912），将原来的覆盖，就像“压栈”一样，在函数结束后，会将这个“栈”弹出。
第三次调用时，item_list 的默认参数还是指向2830870084744（因为上一次调用已经将新对象的引用弹出了）

stackoverflow 上有一个更适当的例子来说明默认参数是在定义的时候求值，而不是调用的时候。

```
>>> import time
>>> def report(when=time.time()):
...     return when
...
>>> report()
1500113234.487932
>>> report()
1500113234.487932
```

4.4 时有时无的切片异常

alist 只有 5 个元素，当你取第 6 个元素时，会抛出索引异常。这与我们的认知一致。

```
>>> alist = [0, 1, 2, 3, 4]
>>> alist[5]
Traceback (most recent call last):
```

(下页继续)

(续上页)

```
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

但是当你使用 alist[5:] 取一个区间时，即使 alist 并没有第 6 个元素，也不抛出异常，而是会返回一个新的列表。

```
>>> alist = [0, 1, 2, 3, 4]
>>> alist[5:]
[]
>>> alist[100:]
[]
```

4.5 return 不一定都是函数的终点

那结论就出来了，如果 finally 里有显式的 return，那么这个 return 会直接覆盖 try 里的 return，而如果 finally 里没有显式的 return，那么 try 里的 return 仍然有效。

4.6 用户无感知的小整数池

Python 定义了一个小整数池 [-5, 256] 这些整数对象是提前建立好的，不会被垃圾回收。

4.7 循环中的局部变量泄露

4.8 python 版本升级

python3.x 并不向后兼容，所以如果从 2.x 升级到 3.x 的时候得小心了，下面列举两点：

在 python2.7 中，range 的返回值是一个**列表**；而在 python3.x 中，返回的是一个 **range 对象**。

map()、filter()、dict.items() 在 python2.7 返回列表，而在 3.x 中返回迭代器。当然**迭代器大多数都是比较好的选择**，更加 pythonic，但是也有缺点，就是**只能遍历一次**。在 instagram 的分享中，也提到因为这个导致的一个坑爹的 bug。

4.9 参考

Python 黑魔法指南 50 例:python.iswbm.com/en/latest/c01/c01_10.html

Python 有什么不为人知的坑?：https://www.zhihu.com/question/29823322?sort=created

这十多个 Python 中的坑，简直太恶心了，摔的我眼冒金星:https://www.pythonf.cn/read/42134

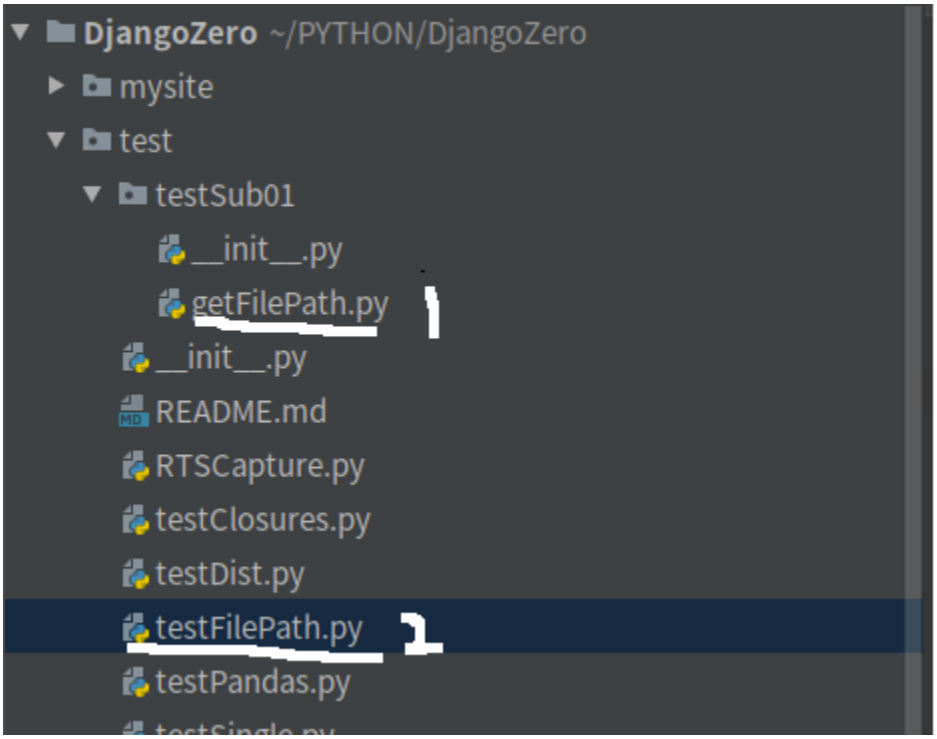
CHAPTER 5

python 实战 05 文件路径 (找不到文件)

开发时遇到问题，文件路径不正确，找不到文件等等，都是这一类问题.

5.1 curdir,argv,file

举例:



文件 1 代码:

```
def get_cur_path1():
    import os
    print(os.path.abspath(os.curdir))


def get_cur_path2():
    import sys
    print(sys.argv[0])


def get_cur_path3():
    import os
    print(os.path.abspath(__file__))
```

文件 2 代码:

```
from test.testSub01.getFilePath import get_cur_path3, get_cur_path2, get_cur_path1

get_cur_path1()
get_cur_path2()
get_cur_path3()
```

文件 2 执行输出结果和解析:

```
/home/john/PYTHON/DjangoZero/test                          #01, 项目 work dir
/home/john/PYTHON/DjangoZero/test/testFilePath.py     #02, 调用者的路径
/home/john/PYTHON/DjangoZero/test/testSub01/getFilePath.py     #03, 当前文件真实路径
```

如果在代码 01, 里面写了 open('../a/' ,' r') 类似的代码，那么其实是使用了 **workdir 为基准的路径**,

本例就是: /home/john/PYTHON/DjangoZero/test/../a=>/home/john/PYTHON/DjangoZero/a/

如果大家都是同一个项目，一般没问题，有偶尔会有别人开发的模块，自己去调用，发现别人可以正常跑的代码，自己确提示找不到文件

大概率就是 workdir 配置的不一致导致，(默认 work direction, 大部分是执行的 py 文件的父文件夹, 比如 aa/bb/c.py 就是 aa/bb/)

如何解决此类问题呢? 最好的做法是使用上面 "03, 当前文件真实路径为基准" 的方法（以 03 返回的路径为基准，../到需访问文件相应的文件夹)，这样的话，不论从哪里调用，只要 **py 和要访问的目标文件相对位置不变**就行了，而觉大多数情况下，二者是位于同一个文件夹中的。

举例: 如果在 getFilePath.py 中访问, 上层文件夹同级别的 README.md, 则使用

```
with open(os.path.join(os.path.dirname(__file__),'../README.md')) as f:
    print(f.read())
```

这样的话，不论从哪个路调用 getFilePath.py 里面的方法，都会找到文件的正确路径。

5.2 normpath

另外在做路径连接时，优先使用 os.path.normpath, 而非 os.path.join，虽然 join 比较常见，但是坑比较多

```
print('normpath')
print(os.path.normpath("%s/%s" % ("dirName1", "dirName2")))
print(os.path.normpath("%s/%s" % ("/dirName1", "dirName2")))
print(os.path.normpath("%s/%s" % ("/dirName1/", "dirName2")))
print(os.path.normpath("%s/%s" % ("/dirName1/", "/dirName2")))
print(os.path.normpath("%s/%s" % ("/dirName1/", "/dirName2/")))
print(os.path.normpath("%s/%s" % ("/dirName1/", "/dirName2/1.txt")))
print('join')
print(os.path.join("dirName1", "dirName2"))
print(os.path.join("/dirName1/", "/dirName2"))
print(os.path.join("/dirName1/", "/dirName2/"))
print(os.path.join("dirName1", "dirName2/1.txt"))
```

运行结果:

```
normpath# 结果均符合直观理解
dirName1/dirName2
/dirName1/dirName2
/dirName1/dirName2
/dirName1/dirName2
/dirName1/dirName2
/dirName1/dirName2/1.txt
join# 个别结果不符合直观理解
dirName1/dirName2# 符合直观
/dirName2# 不符合直观
/dirName2/# 不符合直观
dirName1/dirName2/1.txt# 符合直观
```

5.3 更多功能测试

```
(base) john@john-P95-HP:~/PYTHON/DjangoZero$ python test/testFilePath.py
argv:test/testFilePath.py
file:test/testFilePath.py
abs file:/home/john/PYTHON/DjangoZero/test/testFilePath.py
dir file:test
dir abs file :/home/john/PYTHON/DjangoZero/test
curdir:.
abs file:/home/john/PYTHON/DjangoZero
dir file:
dir abs file :/home/john/PYTHON
getpwd:/home/john/PYTHON/DjangoZero
abs file:/home/john/PYTHON/DjangoZero
dir file:
dir abs file :/home/john/PYTHON



cur:.
abs cur:/home/john/PYTHON/DjangoZero
argv:['test/testFilePath.py']
file:/home/john/PYTHON/DjangoZero/test/testSub01/getFilePath.py
abs file:/home/john/PYTHON/DjangoZero/test/testSub01/getFilePath.py
```

```
(base) john@john-P95-HP:~/PYTHON/DjangoZero$ cd test/
```

(下页继续)

(续上页)

```
(base) john@john-P95-HP:~/PYTHON/DjangoZero/test$ python testFilePath.py
argv:testFilePath.py
file:testFilePath.py
abs file:/home/john/PYTHON/DjangoZero/test/testFilePath.py
dir file:
dir abs file :/home/john/PYTHON/DjangoZero/test
curdir:.
abs file:/home/john/PYTHON/DjangoZero/test
dir file:
dir abs file :/home/john/PYTHON/DjangoZero
getpwd:/home/john/PYTHON/DjangoZero/test
abs file:/home/john/PYTHON/DjangoZero/test
dir file:
dir abs file :/home/john/PYTHON/DjangoZero



cur:.
abs cur:/home/john/PYTHON/DjangoZero/test
argv:['testFilePath.py']
file:/home/john/PYTHON/DjangoZero/test/testSub01/getFilePath.py
abs file:/home/john/PYTHON/DjangoZero/test/testSub01/getFilePath.py
```

```
(base) john@john-P95-HP:~/PYTHON/DjangoZero/test$  /home/john/anaconda3/envs/django_zero/
↪bin/python testFilePath.py
argv:testFilePath.py
file:testFilePath.py
abs file:/home/john/PYTHON/DjangoZero/test/testFilePath.py
dir file:
dir abs file :/home/john/PYTHON/DjangoZero/test
curdir:.
abs file:/home/john/PYTHON/DjangoZero/test
dir file:
dir abs file :/home/john/PYTHON/DjangoZero
getpwd:/home/john/PYTHON/DjangoZero/test
abs file:/home/john/PYTHON/DjangoZero/test
dir file:
dir abs file :/home/john/PYTHON/DjangoZero
```

(下页继续)

(续上页)

```
cur:.
abs cur:/home/john/PYTHON/DjangoZero/test
argv:['testFilePath.py']
file:/home/john/PYTHON/DjangoZero/test/testSub01/getFilePath.py
abs file:/home/john/PYTHON/DjangoZero/test/testSub01/getFilePath.py
```

父文件路径

argv=**FILE**

abs(),dir() 函数,

cur() 简写.

getpwd(),cur () 表现一致

子文件路径

argv!=**file**

agrv: 启动脚本的 argv(父程序)

File：子程序完整路径

abs(cur): 父程序启动位置路径

getcwd 和 abs(cur) 依然一致

使用 **file** 获取当前路径:**file** 表示显示文件当前的位置:

如果当前文件包含在 sys.path 里面，那么 **file** 返回一个相对路径

如果当前文件不包含在 sys.path 里面，那么 **file** 返回一个绝对路径

os.getcwd() 与 os.curdir 的使用 os.getcwd() 与 os.curdir 都是用于获取当前执行 python 文件的文件夹，不过当直接使用 os.curdir 时会返回‘.’(这个表示当前路径)，记住返回的是当前执行 python 文件的文件夹，而不是 python 文件所在的文件夹。PS：**os.getcwd() 与 os.path.abspath(os.curdir) 返回的结果是一样的**.

5.4 附录

Python - 编写模块时获取当前路径 **file** 与 getcwd():https://blog.csdn.net/sigmarising/article/details/83444463

python 函数深入浅出 12.os.getcwd() 函数详解:https://www.jianshu.com/p/77bf050ba274

CHAPTER 6

python 实战 06 多线程 bug 处理记录

多线程 bug 处理记录

```
Thread(target=func02)
while True:
    dataA=DataA()
    dataA.data=[[[xx,yy]]]
    xxx,
    yyy,
    zzz,
    dataA.data=[[[xx,yy]],[[ff,zz]]]
    assert len(dataA.data)>1,error data# 标记 01
    self.queue.put(dataA)


func02():
    dataA=self.queue.get()
    assert len(dataA.data)>1,error data# 标记 02
```

问题:func02 获取的 data 有时为一维的，而理论上应该是二维的

添加标记 01, 标记 02 后发现，标记 01 是 ok 的（说明存放时的确是二维的)，但标记 02 却偶然有报错 (偶尔，而非绝对)

修改测试 01,queue 不支持多维数据存放? 不大可能，经过独立测试，的确没有问题，可以存放

修改测试 02, 将多线程改为顺序执行，问题不存在 => 说明是多线程导致的问题

当然最终也发现问题了，这也是**多线程最常出现的问题，就是"以为没共享的数据，其实共享了**"在博客python 进阶 12 并发之八多线程与数据同步中也强调过，但不经意间依然会掉坑里。

其实对于此类问题，最好的调试方式是，出问题的是 queue，那么就**打印所有操作 quque 的地方**，就可以发现 queue 其实会在 2 个线程中被操作，即使 queue 本身的存取是原子的，由于存放的数据是引用类型，所以 queue 临近的代码 "上下文"，其实都属于 "临界区"。需要 "特殊防范"。

最简单的处理方式是 self.queue.put(copy.deepcopy(dataA))，这样就彻底隔离了不同线程的数据，避免了一些隐患。

CHAPTER 7

python 实战 07 调试 pdb

7.1 Python 多线程的时候调试的简单方法 (thread.run)

https://blog.csdn.net/york1996/article/details/89305847

```
for thread in threads:
    thread.run()# 原本是 thread.start ()
```

7.2 OpenStack 断点调试方法总结 (重定向 stdin,stdout 实现远程调试)

https://zhuanlan.zhihu.com/p/63898351

```
import sys
import socket

def pre_pdb(addr='127.0.0.1', port=1234):
    old_stdout = sys.stdout
    old_stdin = sys.stdin
    handle_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    handle_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    handle_socket.bind((addr, port))
    handle_socket.listen(1)
    print("pdb is running on %s:%d" % (addr, port))
    (client, address) = handle_socket.accept()
    handle = client.makefile('rw')
    sys.stdout = sys.stdin = handle
    return handle
```



```
import pdb
a = 1
b = 2
handle = pre_pdb() # 重定向stdin、stdout
pdb.Pdb(stdin=handle, stdout=handle).set_trace()
c = a + b
```


通过这种方式可以实现远程调试，不过我们不用每次都写那么长一段代码，社区已经有实现了，只需要使用 rpdb 替换 pdb 即可进行远程调试，原理与之类似，

7.3 python 多线程断点调试 (管道方法)

https://blog.csdn.net/DeathlessDogface/article/details/84074461

插入断点

```
import pdb
pdb.Pdb(stdin=open('/root/p_in','r+'),stdout=open('/root/p_out','w+')).set_trace()
```

创建管道

```
mkfifo /root/p_in /root/p_out
```

7.4 gdb 调试多进程和多线程命令 (设置 follow-fork-mode)

https://blog.csdn.net/pbymw8iwm/article/details/7876797

设置 follow-fork-mode(默认值：parent) 和 detach-on-fork（默认值：on）即可。

follow-fork-mode detach-on-fork 说明

```
parent                  on              只调试主进程 (GDB 默认)
child                    on              只调试子进程
parent                  off             同时调试两个进程，gdb 跟主进程，子进程 block 在
fork 位置
child                    off             同时调试两个进程，gdb 跟子进程，主进程 block 在
fork 位置
```

设置方法：set follow-fork-mode [parent|child] set detach-on-fork [on|off]

查询正在调试的进程：info inferiors

切换调试的进程：inferior

查询线程：info threads

切换调试线程：thread

7.5 Linux 多进程和多线程的一次 gdb 调试实例 (参考上面的)

https://typecodes.com/cseries/multilprocessthreadgdb.html

follow-fork-mode detach-on-fork 说明

```
parent              on          GDB 默认的调试模式：只调试主进程
child               on          只调试子进程
parent              off         同时调试两个进程，gdb 跟主进程，子进程 block 在 fork 位置
child               off         同时调试两个进程，gdb 跟子进程，主进程 block 在 fork 位置
```

查看 gdb 默认的参数设置：

```
(gdb) show follow-fork-mode
Debugger response to a program call of fork or vfork is "parent".
(gdb) show detach-on-fork
Whether gdb will detach the child of a fork is on.


catch fork
```

(下页继续)

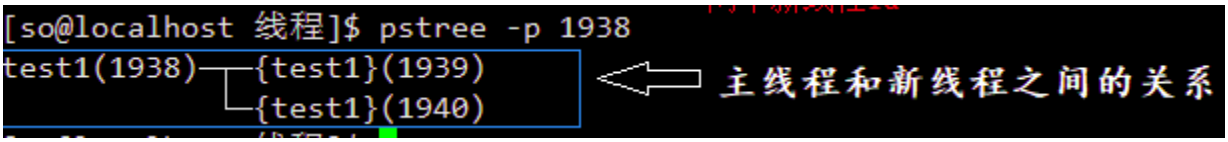
(续上页)

```
info b
bt
```

7.6 线程的查看以及利用 gdb 调试多线程 (详细，截图中标示含义)

https://blog.csdn.net/zhangye3017/article/details/80382496

pstree -p xx

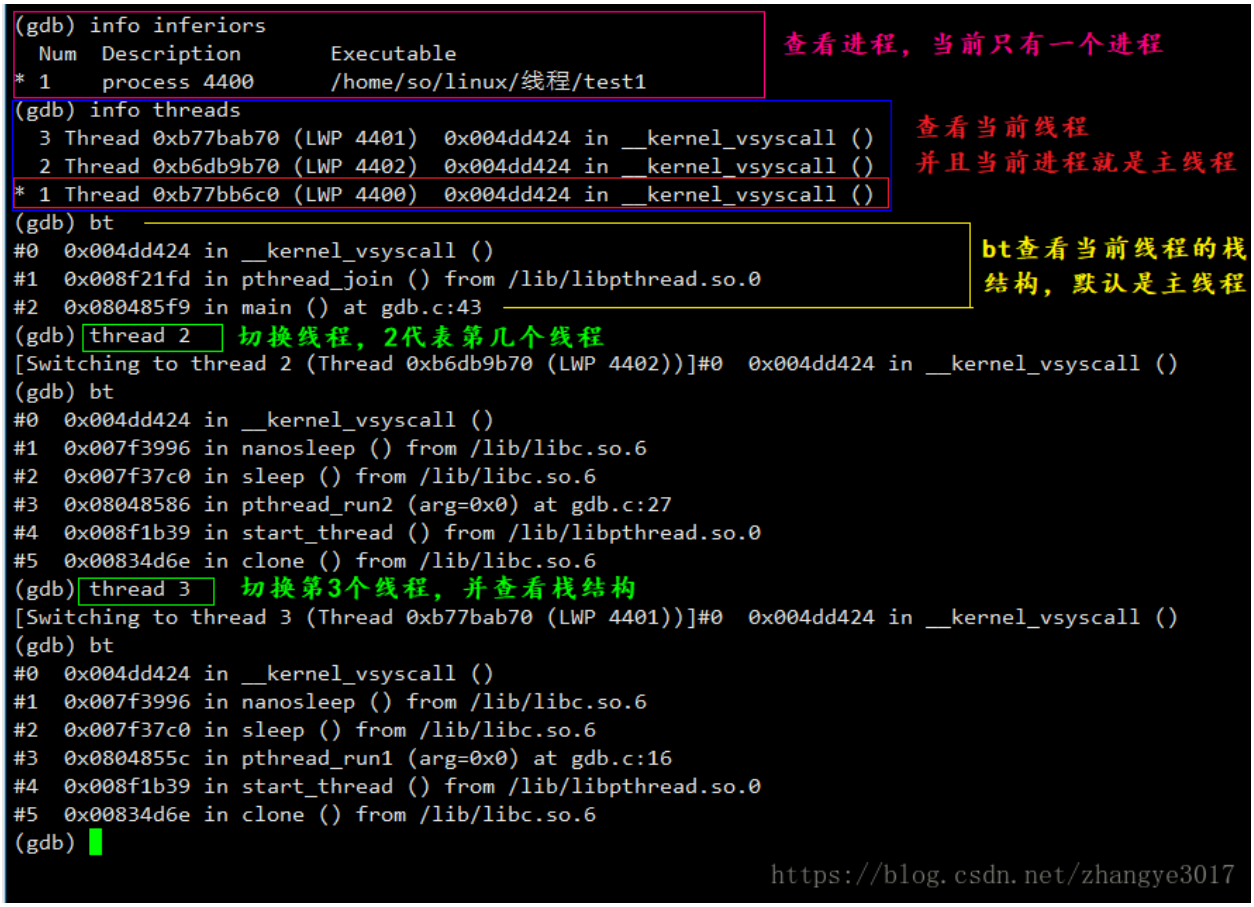


pstack xx

gdb attach 主线程 ID

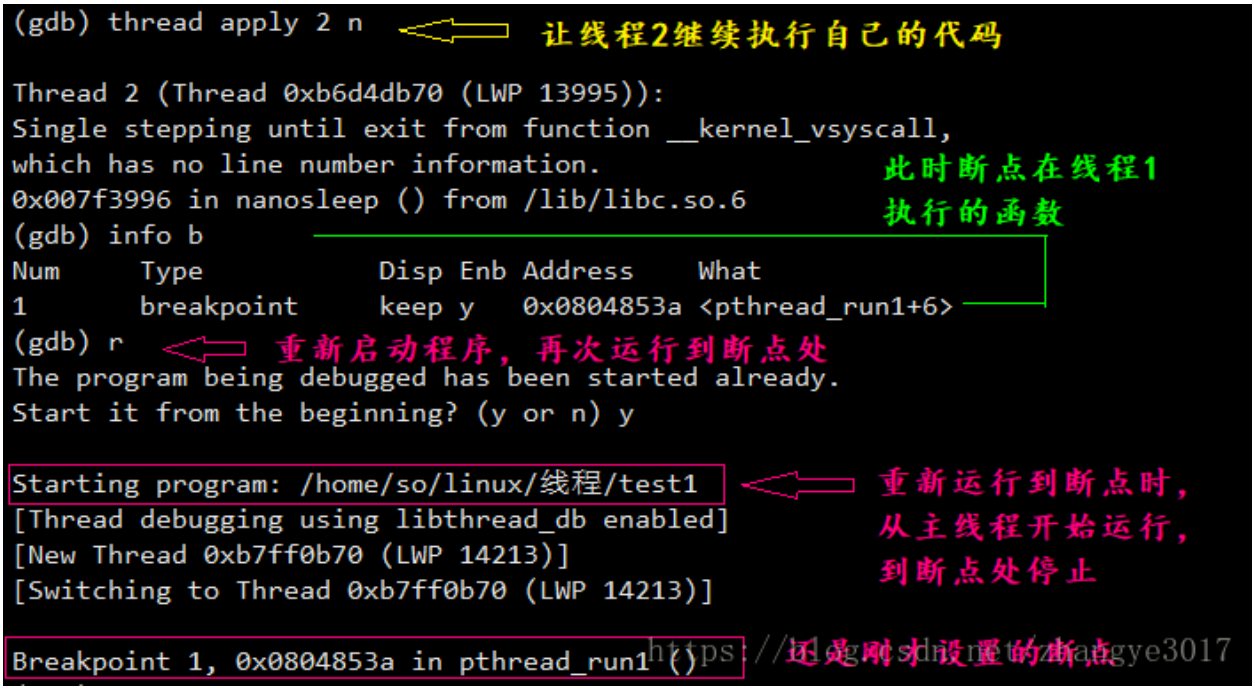
常用命令

1. 查看进程：info inferiors
2. 查看线程：info threads
3. 查看线程栈结构：bt
4. 切换线程：thread n（n 代表第几个线程）



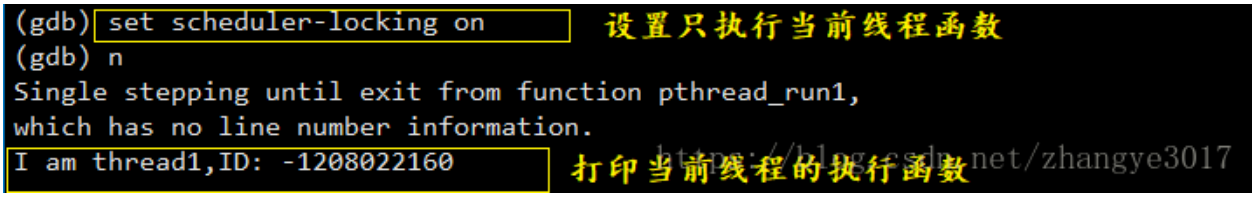
执行线程 2 的函数，指行完毕继续运行到断点处

1. 继续使某一线程运行：thread apply 1-n（第几个线程） n
2. 重新启动程序运行到断点处：r



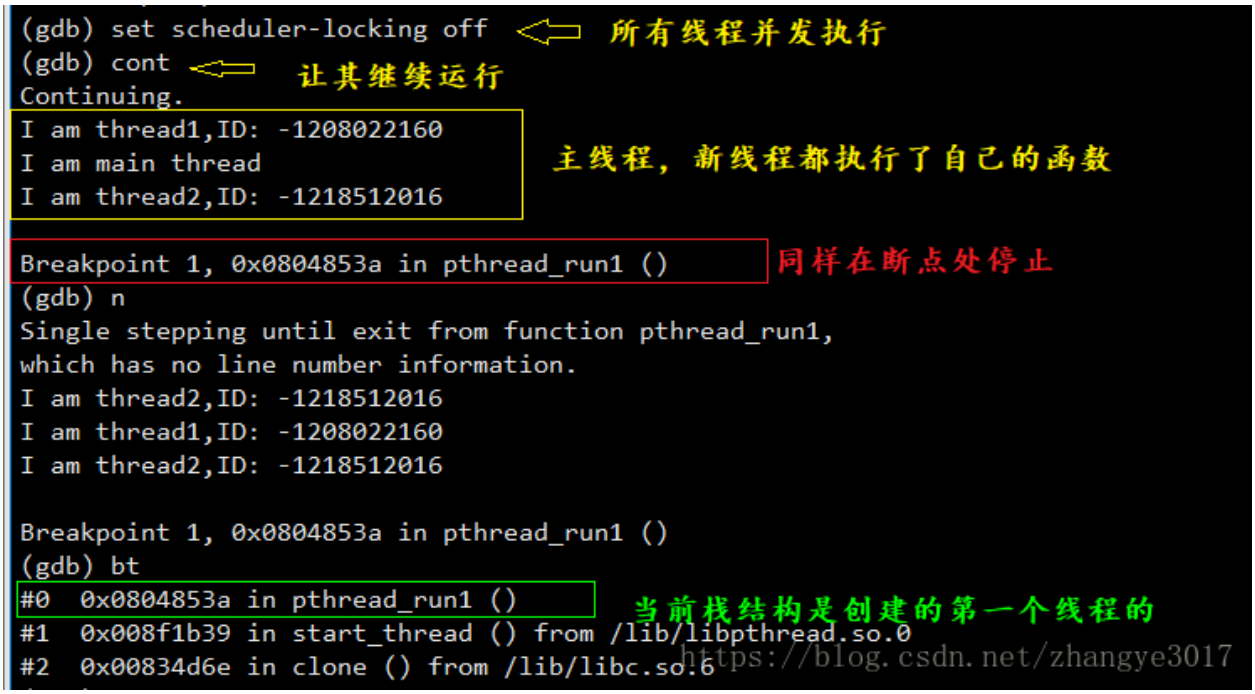
只运行当前线程

1. 设置：set scheduler-locking on
2. 运行：n



所有线程并发执行

1. 设置：set scheduler-locking off
2. 运行：n



7.7 GDB 多线程多进程调试 (操作和命令对应)

https://cloud.tencent.com/developer/article/1142947

thread thread-id 实现不同线程之间的切换

info threads 查询存在的线程

thread apply [thread-id-list] [all] args 在一系列线程上执行命令

线程中设置指定的断点

set print thread-events 控制打印线程启动或结束是的信息

set scheduler-locking off|on|step 在使用 step 或是 continue 进行调试的时候，其他可能也会并行的执行，如何才能够只让被调试的线程执行呢？该命令工具可以达到这个效果。

off：不锁定任何线程，也就是所有的线程都执行，这是默认值。

on：只有当前被调试的线程能够执行。

step：阻止其他线程在当前线程单步调试时，抢占当前线程。只有当 next、continue、util 以及 finish 的时候，其他线程才会获得重新运行的机会。

使用 thread apply 来让一个或是多个线程执行指定的命令。例如让所有的线程打印调用栈信息。

```
(gdb) thread apply all bt

Thread 3 (Thread 0x7ffff688a700 (LWP 20568)):
```

(下页继续)

(续上页)

```
#0  0x00007ffff7138a3d in nanosleep () from /lib64/libc.so.6
#1  0x00007ffff71388b0 in sleep () from /lib64/libc.so.6
#2  0x000000000040091e in threadPrintWorld (arg=0x0) at multithreads.cpp:18
#3  0x00007ffff74279d1 in start_thread () from /lib64/libpthread.so.0
#4  0x00007ffff71748fd in clone () from /lib64/libc.so.6

Thread 2 (Thread 0x7ffff708b700 (LWP 20567)):
#0  threadPrintHello (arg=0x0) at multithreads.cpp:10
#1  0x00007ffff74279d1 in start_thread () from /lib64/libpthread.so.0
#2  0x00007ffff71748fd in clone () from /lib64/libc.so.6

Thread 1 (Thread 0x7ffff7fe5720 (LWP 20564)):
#0  0x00007ffff7138a3d in nanosleep () from /lib64/libc.so.6
#1  0x00007ffff71388b0 in sleep () from /lib64/libc.so.6
#2  0x00000000004009ea in main (argc=1, argv=0x7fffffffe628) at multithreads.cpp:47
```

7.8 参考

pyrasite(4 年前的老项目，久不更新)

线程的查看以及利用 gdb 调试多线程（同上，但多了“安装的 pstack 有问题，自实现”，但 pstack 依然乱码)：https://www.cnblogs.com/fengtai/p/12181907.html

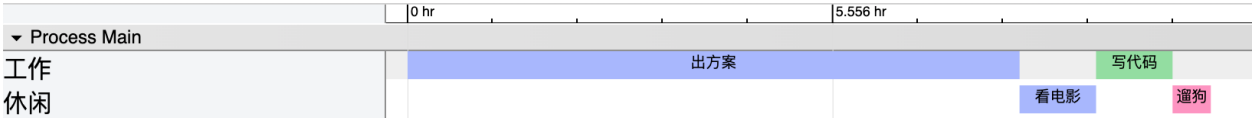
CHAPTER 8

python 实战 08 多线程性能分析 (装饰器和 chromeTrace)

8.1 需求

多线程开发时，需要进行性能分析时，希望查看各线程（进程的执行时序图）

类似如下效果



8.2 转化脚本和使用方法

如下代码将在 error.log（添加到 debug.log 可能更合理）中添加部分日志，当然，生成的日志无法直接用 chrome tracing 绘制时序图，但经过转化脚本后则可以

```
def log_time(name, start_time, end_time, pid, tid):
    logger.error({"name": name, "ph": "B", "pid": pid, "tid": tid, "ts": start_time * 1000 * 1000})
    logger.error({"name": name, "ph": "E", "pid": pid, "tid": tid, "ts": end_time * 1000 * 1000})


def cal_time(name=None):
    def logger_time(func):
        def make_decorater(*args, **kwargs):
            start_time = time.time()
            func(*args, **kwargs)
            end_time = time.time()
            log_time(name, start_time, end_time, pid=os.getpid(), tid=threading.current_thread().ident)

        return make_decorater

    return logger_time
```

使用方式 01，装饰器

```
@cal_time(name=xx)
def func_run(txt):
    print('txt:%s' % txt)
```

使用方式 02，直接使用

```
while self.is_start:
    start_time = time.time()
    
    需统计耗费时间的代码
    
    end_time = time.time()
    OPEN_LOG and log_time('[redacted]read', start_time, end_time, pid=os.getpid(), tid=threading.current_thread().ident)
```

仅仅使用如上方式，可以在日志中得到 func 的开始和结束时间

需要转为 chrome trace 支持的数据格式，进行进一步分析

8.3 转化脚本

借助如下 sed 脚本可以将数据转为 chrome trace 支持的格式

```
grep -P '\{.*?\}' [redacted] err.log -o > chrome_tracing.log
sed -i 's/$/,/g' chrome_tracing.log && sed -i 1i[ chrome_tracing.log  && sed -i '$s/,/]/5' chrome_tracing.log
sed -i s/\'/\"/g chrome_tracing.log
```

转换格式后如下

```
[
    {"name": "出方案", "ph": "B", "pid": "Main", "tid": "工作", "ts": 0},
    {"name": "出方案", "ph": "E", "pid": "Main", "tid": "工作", "ts": 28800000000},
    {"name": "看电影", "ph": "B", "pid": "Main", "tid": "休闲", "ts": 28800000000},
    {"name": "看电影", "ph": "E", "pid": "Main", "tid": "休闲", "ts": 32400000000},
```

(下页继续)

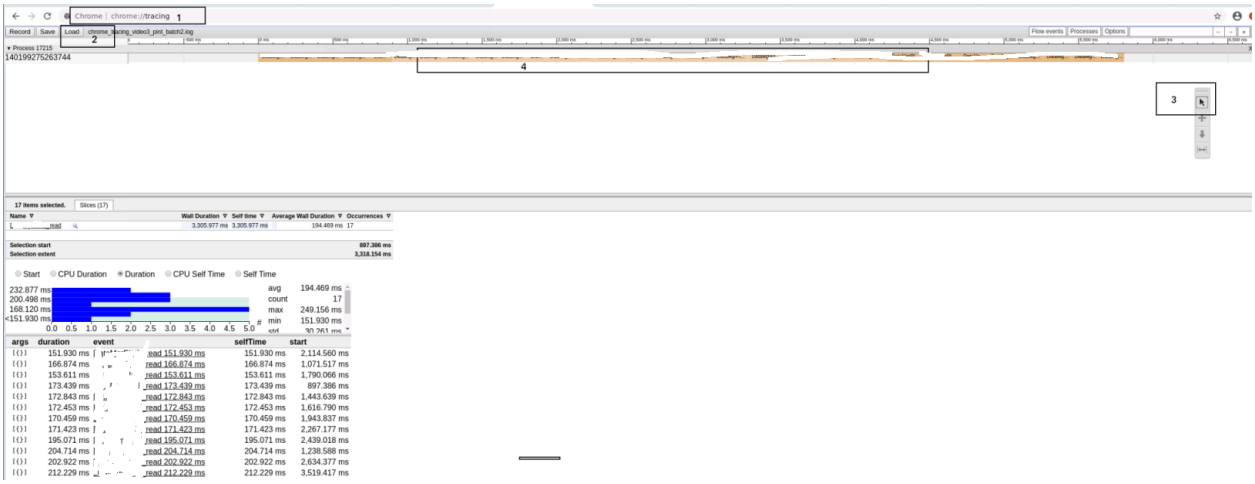
(续上页)

```
    {"name": "写代码", "ph": "B", "pid": "Main", "tid": "工作", "ts": 32400000000},
    {"name": "写代码", "ph": "E", "pid": "Main", "tid": "工作", "ts": 36000000000},
    {"name": "遛狗", "ph": "B", "pid": "Main", "tid": "休闲", "ts": 36000000000},
    {"name": "遛狗", "ph": "E", "pid": "Main", "tid": "休闲", "ts": 37800000000}
]
```

8.4 使用 chrome tracing 分析

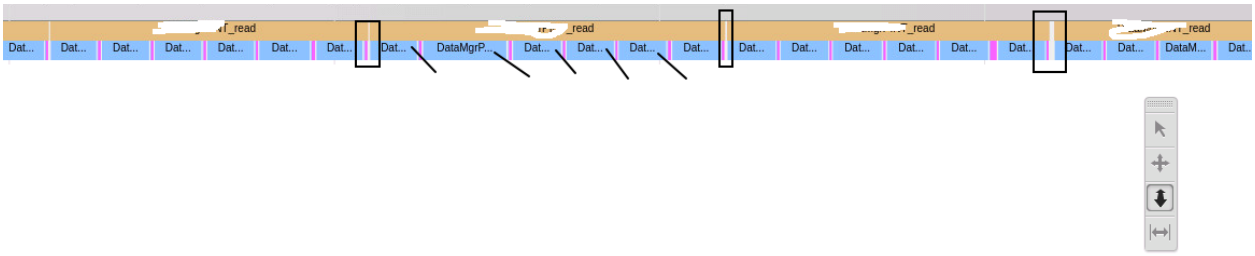
使用 chrome tracing 分析

效果示意图:



可以统计出总时间区间个数，最大值，最小值和平均时间等信息

另一个例子: 一个大方法内部各子步骤时序图样例



8.5 参考

多线程程序性能分析：none

多线程程序性能分析工具：none

vtune 性能分析工具–找出程序性能瓶颈：https://blog.csdn.net/pzhw520hchy/article/details/79823038

valgrind 的 callgrind 工具进行多线程性能分析：https://www.cnblogs.com/zengkefu/p/5642991.html

python 多线程程序性能分析工具：none

python 多线程性能分析：none

python 中如何理解装饰器代码?：https://www.zhihu.com/question/65229244

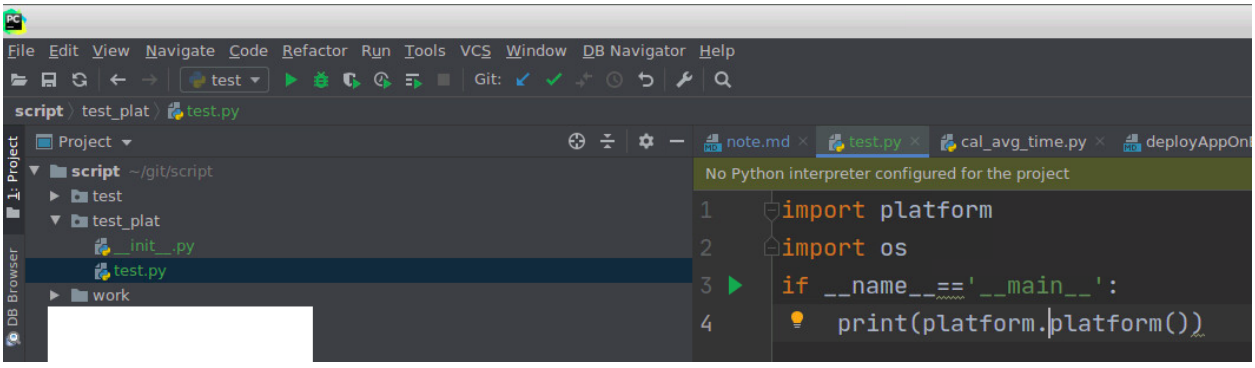
强大的可视化利器 Chrome Trace Viewer 使用详解:https://limboy.me/2020/03/21/chrome-trace-viewer/

CHAPTER 9

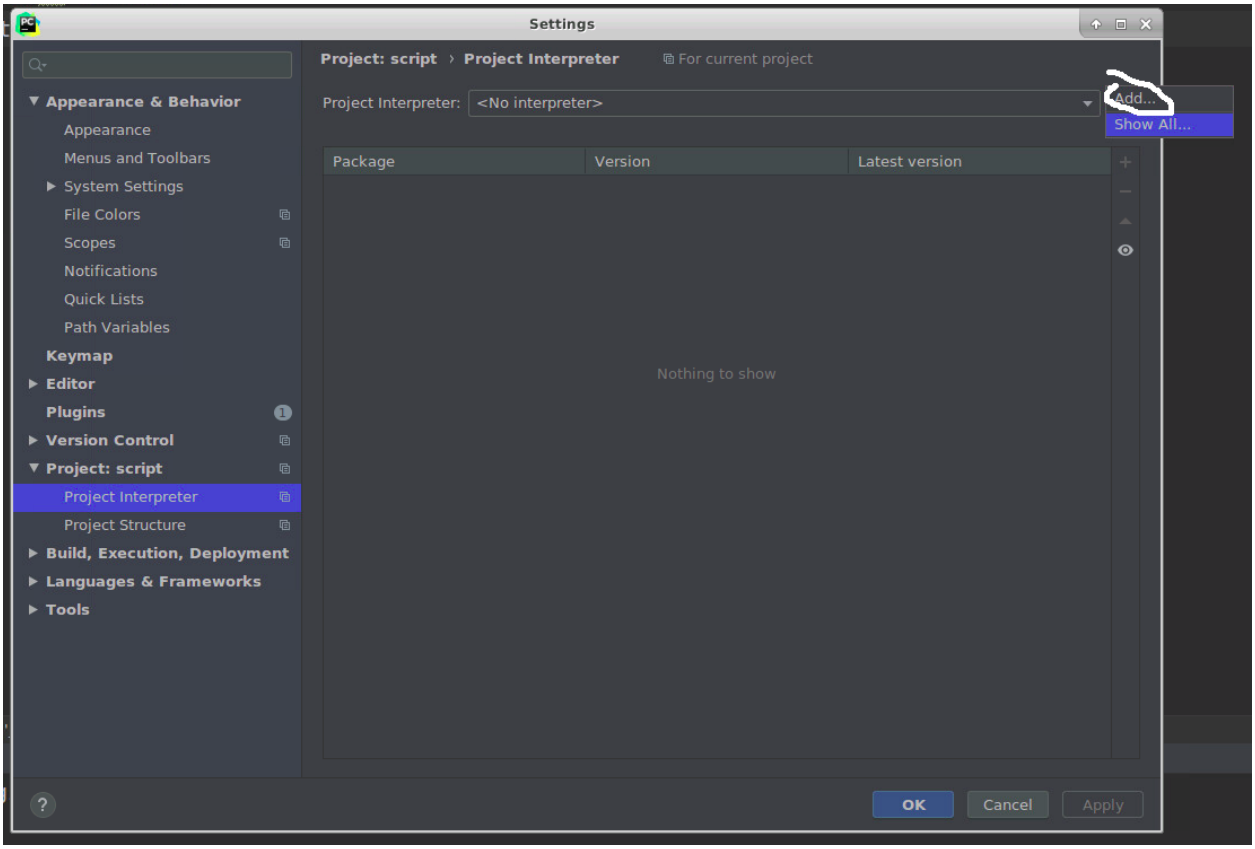
python 实战 09 远程接口调试

需要将项目部署到 armbox 上，但是发现 arm 上调试非常繁琐。由于没有图形界面，所以无法使用 pycharm 工具，只能在终端中执行 python 命令，然后通过 pdb 进行调试。对于单线程尚无问题，但是对于 web 等多线程项目，pdb 就无法进行调试了。之前做 java 时曾用过远程调试，查了下，pycharm 其实也有这个功能。当然必须是专业版，免费版是不行的。

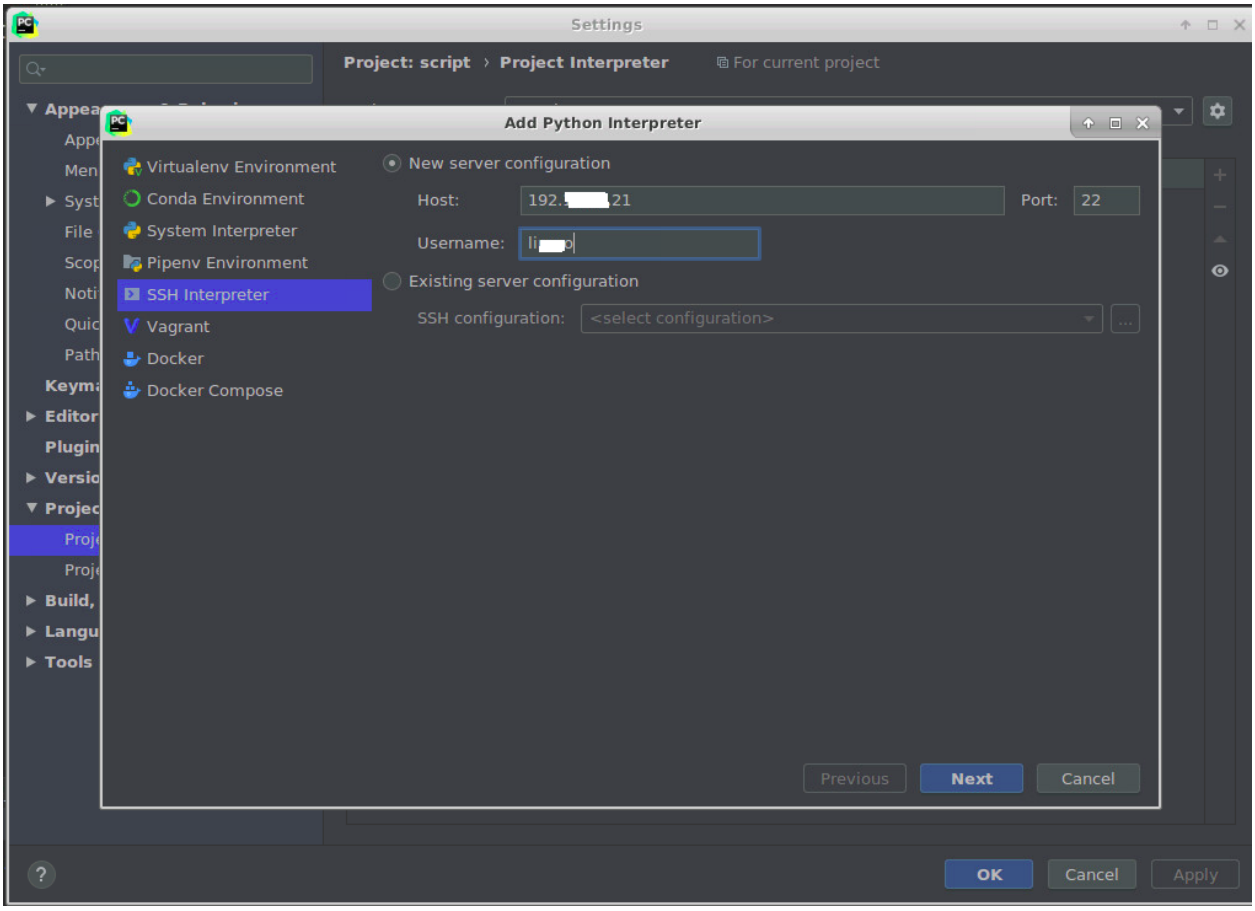
9.1 原始代码：打印所属平台信息



9.2 添加 Python Interpreter

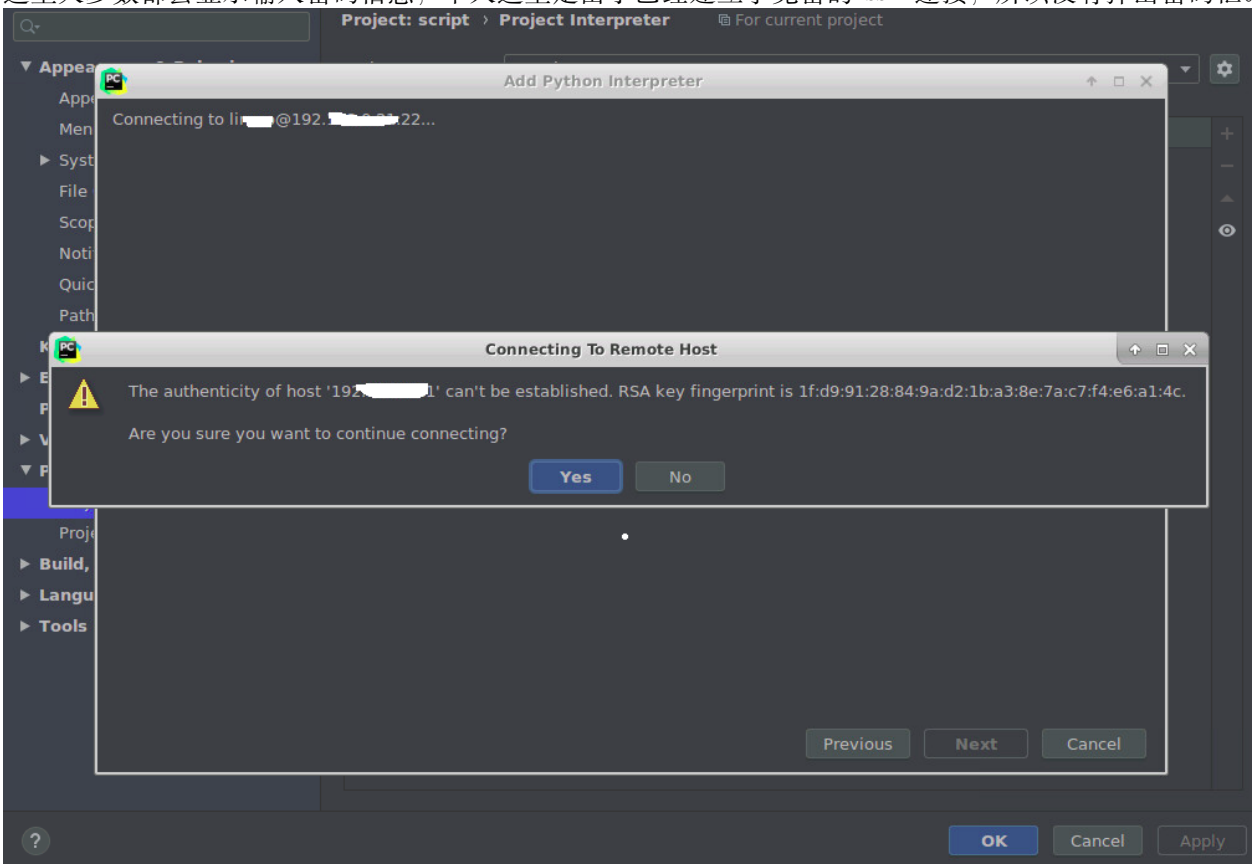


9.3 添加 ssh interpreter 连接信息



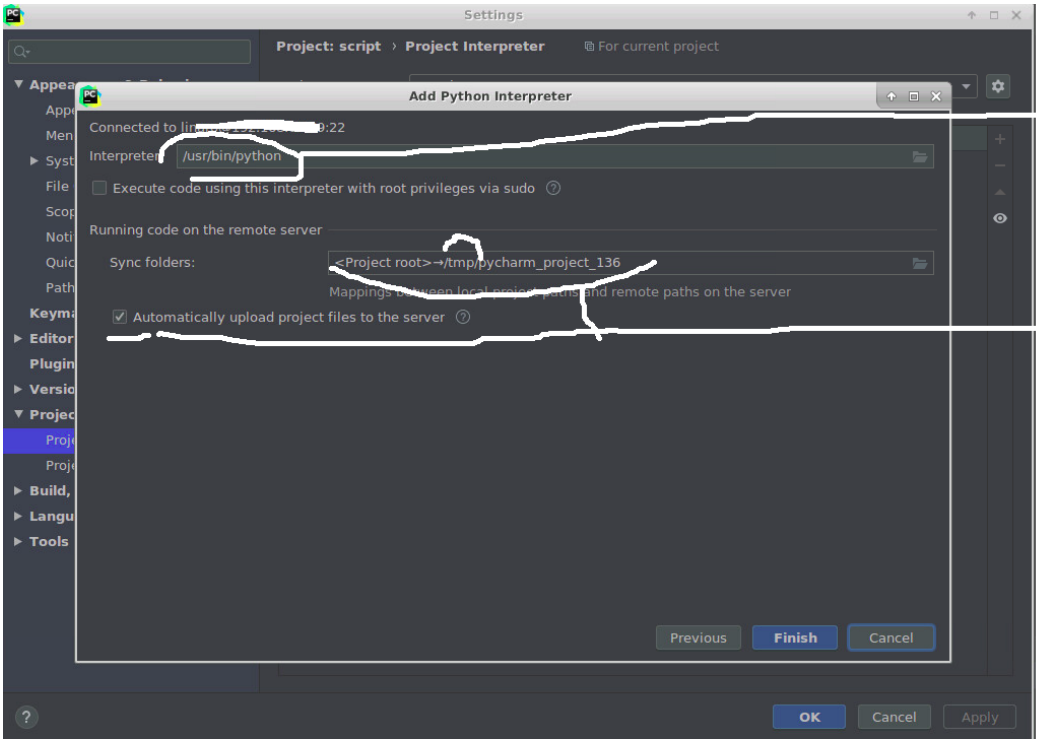
9.4 ssh interpreter 密码信息

这里大多数都会显示输入密码信息，本人这里是由于已经建立了免密的 ssh 连接，所以没有弹出密码框。

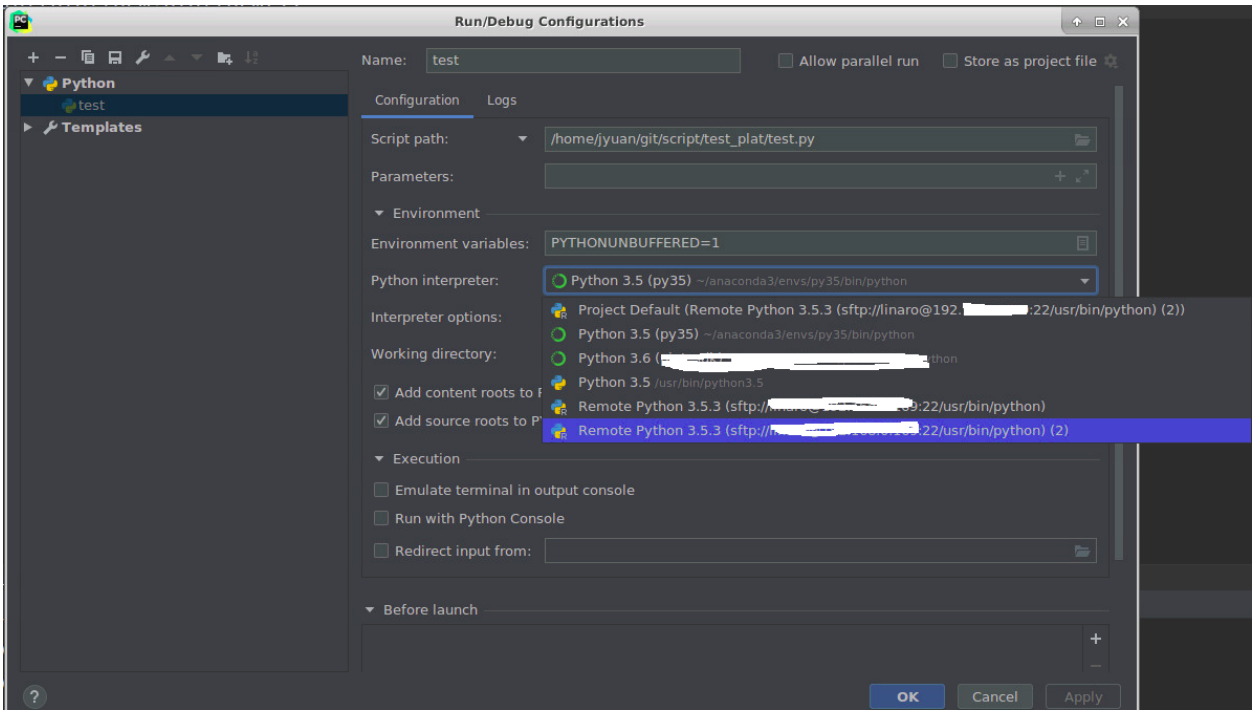


弹出密码样式

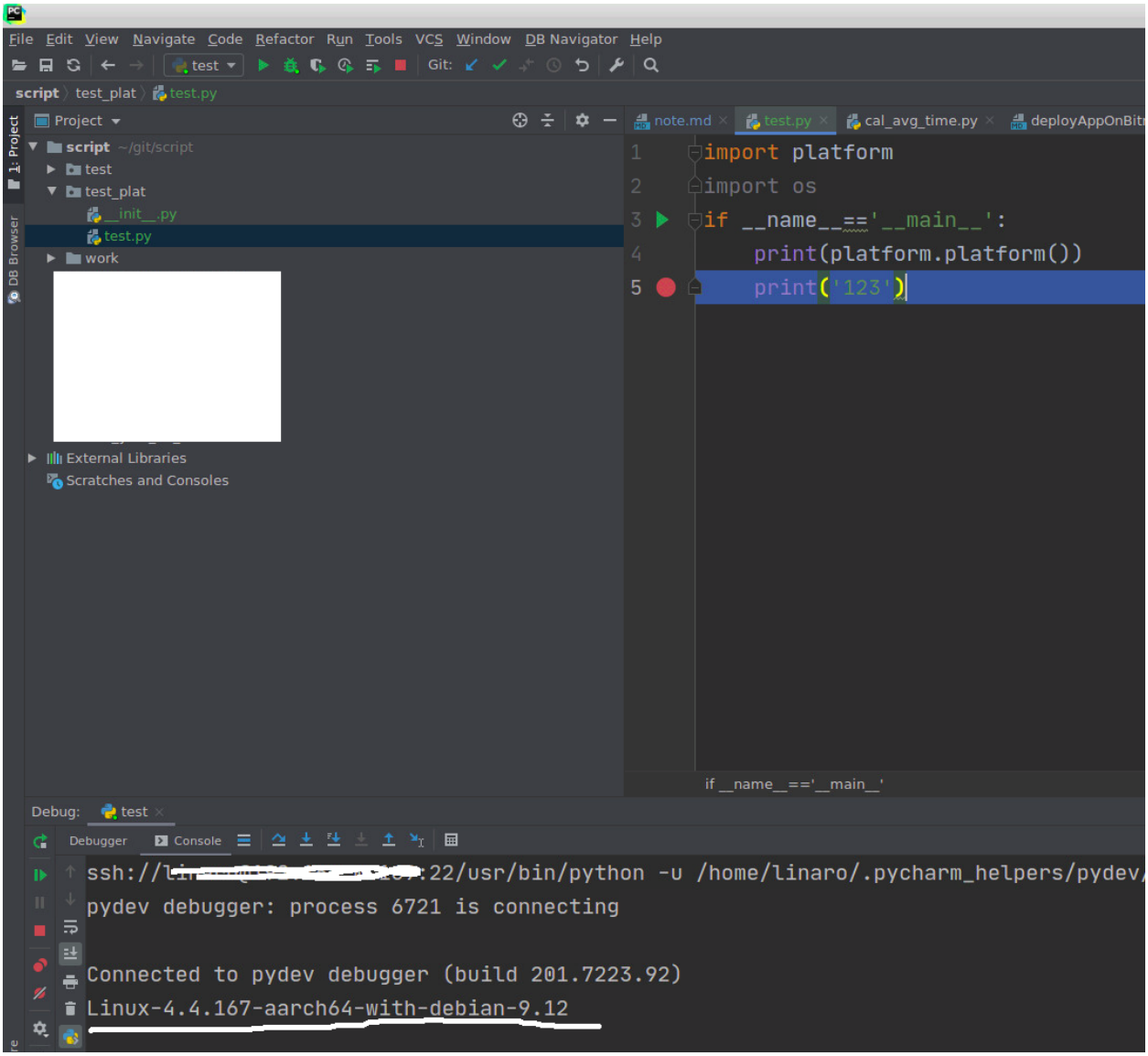
9.5 interpreter 配置信息



9.6 修改执行代码的 interpreter



9.7 验证结果正确



9.8 参考

PyCharm 远程开发调试:https://blog.csdn.net/yejingtao703/article/details/80292486SSH 三步解决免密登录：https://blog.csdn.net/jeikerxiao/article/details/84105529ssh 隧道解决 pycharm 跨过跳板机连接服务器问题：https://blog.csdn.net/huangbx_tx/article/details/93339715

CHAPTER 10

python 实战 10pytest 测试和覆盖率

插件安装：pip install pytest-cov

命令：pytest –cov=src –cov-report=html

src：python 源代码路径（文件夹形式，不支持模块 or 模块.py 等形式）

注意：文件夹下所有**符合文件名:test_.__test.py 都必须能跑通，否则 html 报表中只有函数定义**，没有函数内的代码执行情况。

其他插件:

1. 多重校验 pytest-assume

2. 设定执行顺序 pytest-ordering

3. 失败重跑 pytest-rerunfailures

4. 显示进度条 pytest-sugar

5.pytest-pep8，就是在做 pytest 测试时，自动检测代码是否符合 PEP 8 规范的插件。

6.pytest-mock 是一个 pytest 的插件，安装即可使用。它提供了一个名为 mocker 的 fixture，仅在当前测试 function 或 method 生效，而不用自行包装。

10.1 参考

[转]Pytest 基础教程：https://blog.csdn.net/u011331731/article/details/108189950

pytest 的一些实用插件实践：https://www.jianshu.com/p/7df6d781f100

【pytest】pytest-cov ： 统计代码测试覆盖率:https://blog.csdn.net/waitan2018/article/details/104400749

pytest-cov 插件计算单元测试代码覆盖率:https://blog.csdn.net/u011519550/article/details/86367137

CHAPTER 11

python 实战 11 代码洁癖

11.1 缘由

先说垃圾代码：表面是代码垃圾，实际是逻辑垃圾，思路不清或不够简单。简单代码需要更深入思考，建立更简单，简洁，明了的思路。python 具有很高的灵活性，所以这一点尤其重要。

11.2 类型注解

大型程序中非常必要，弥补 python 弱类型在开发中的沟通成本（接口调用方需阅读接口实现或代码注释才知传参类型）。

```
from typing import List, Sequence

def square(elems: Sequence[float]) -> List[float]:
    return [x**2 for x in elems]
```

11.3 列表生成式替代循环 ([i for xx])

大多数循环都可以用列表生成式替代，列表生成式另一个好处是很方便的就可以实现并行化.

11.4 拍平嵌套的多重循环 (product) 笛卡尔积

itertools.product()

```
In [68]: list(product('ABCD', 'xy'))
Out[68]:
[('A', 'x'),
 ('A', 'y'),
 ('B', 'x'),
 ('B', 'y'),
 ('C', 'x'),
 ('C', 'y'),
 ('D', 'x'),
 ('D', 'y')]
```

11.5 _ 替代无用临时变量 i

其他语言对于无意义的循环个数常用临时变量 i,j 等表示，python 中推荐使用 _(开源代码大多此规则，认为是一种约定习惯)

```
[x for x in range(3) for _ in range(2)]
[0,0,1,1,2,2]
```

11.6 批量化一致性操作优于依次 ifelse 判断

循环内部做 if-else 劣于先批量做处理再 filter 过滤。(不绝对) 对于统一性的批量处理，numpy 等本身会做一定优化，可以通过 Numba 进行进一步优化。而且一致性的批量处理符合 cpu 的 cache 的 LRU 规则，大概率命中 cache。所以不建议内部有 if-else 这样的判断逻辑（如果不同分支时间差异很大，另当别论）简单来说就是操作集中化，每个步骤都做简单的，多步骤组合。优于一个大循环内做不同的分支处理逻辑。

猜测如下代码那个速度最快

```
import time
a = time.time()
b, c = list(), list()
for i in range(10000):
    if not i % 7:
        b.append(i)
    else:
```

(下页继续)

(续上页)

```
        c.append(i)
print(time.time() - a)

a = time.time()
b = list()
c = list()
[b.append(i) if not i % 7 else c.append(i) for i in range(10000)]
print(time.time() - a)

a = time.time()
b = [i for i in range(10000) if not i % 7]
c = [i for i in range(10000) if i % 7]
print(time.time() - a)
```

结论: 最慢，中间，最快比较奇怪的是第一个和第二个，为何第二个比第一个快，测试多次都是同样结果，本人也不大理解!!

11.7 用 a and b(a or b) 替代 if a:b(a if a is not none else b) 的简单表达式

副作用 mypy 静态类型检查会报错 (在 b 没有返回值时)

CHAPTER 12

python 实战 12 神仙问题集锦

12.1 执行速度

问题：一个函数，姑且称之为函数 A，用 demo 单独测试，时间非常快 200ms，但是集成到大系统中，则非常慢，基本在 3s 左右。解析: 函数 A 内部有如下部分组成，预处理（cpu 密集），硬件处理（专用芯片），后处理（cpu 密集）猜测 1：cpu 运算，在 arm 上非常慢导致 (虽然单独测试的 demo 也在 arm 上，但 demo 是纯粹执行函数 A，而大系统中的函数 A 则和大系统本身分享 cpu 资源) 测试：分别统计各步骤测试，结论：全面落后，包括硬件处理（专用芯片）步骤，说明不是这个问题（如果预处理，后处理很慢，专用芯片正常，说明可能瓶颈在 cpu 上）

猜测 2：大系统中存在另一个解码线程，占用 cpu 过多，验证：跑程序时，ps -fe 找到进程 pid, 通过 top -Hp pid 查询进程下各线程 cpu 占用，发现一个线程基本占满 99% 左右。测试：解码一帧后就停止解码线程结论：函数执行时间正常，说明猜测正确，的确是由于进程内其他线程占用过多资源，导致函数 A 所在线程执行机会很少，所以各步骤都非常慢。这个角度看，猜测 1 也是正确的，不过自己忽略了在专用芯片的这个步骤上，如果 cpu 被其他任务占用过多，一样会拖慢时间。

CHAPTER 13

Indices and tables

- genindex
- modindex
- search