python_study Documentation

发布 *v1.0*

yuanjh

2020 年 10 月 31 日

Contents:

个人学习 python 的一些资料整理，个人认为属于不怎么常用场景，所以归纳到进阶部分中了

CHAPTER 1

python 进阶 01 偏函数

定义：偏函数的第二个部分 (可变参数)，按原有函数的参数顺序进行补充，参数将作用在原函数上，最后偏函数返回一个新函数（类似于，装饰器 decorator，对于函数进行二次包装，产生特殊效果；但又不同于装饰器，偏函数产生了一个新函数，而装饰器，可改变被装饰函数的函数入口地址也可以不影响原函数）

效果：固定一部分参数，在后续调用时只需传递少量参数即可

个人倾向于按照重构函数行为来理解，比如需要 3 个函数，一个是 x 的平反，一个是 x 的 3 次方，一个是 x 的四次方，那么一个函数将 2,3,4 当做参数穿进去，生成一个新函数。这样的话可以把原函数看做函数集合，偏函数才是真正使用的函数具体对象。

举例 01：

```
from functools import partial

def mod( n, m ):
  return n % m

mod_by_100 = partial( mod, 100 )

print mod( 100, 7 )  # 2
print mod_by_100( 7 )  # 2
```

举例 02：

```
from functools import partial

bin2dec = partial( int, base=2 )
print bin2dec( '0b10001' )  # 17
print bin2dec( '10001' )  # 17

hex2dec = partial( int, base=16 )
print hex2dec( '0x67' )  # 103
print hex2dec( '67' )  # 103
```

CHAPTER 2

python 进阶 02yield

yield 关键字之前见过，也能读懂，但开发时也不大敢使用，感觉理解还是不够充分。刚好项目代码中有涉及，顺便再学习学习。

在理解 yield 之前，

2.1 可迭代，迭代器，生成器

```
可迭代对象，是其内部实现了，__iter__ 这个魔术方法。
=> 对比可迭代对象，迭代器其实就只是多了一个函数而已。就是__next__()，我们可以不再使用 for 循环来间断获取元素值。而可以直接使用 next() 方法来实现。
==> 生成器，则是在迭代器的基础上（可以用 for 循环，可以使用 next()），再实现了 yield。
```

可以看出三者存在明显递进关系，越往后要求越苛刻，需实现方法也越多.(如果分不清，查阅本文参考文献第一篇)

2.2 猜测代码结果

看如下代码

```
## 第一段代码
items=[i for i in range(10)]
```

(下页继续)

(续上页)

```
tmp=func(m)#func 是一函数
print(func02(tmp))# func02 是另一个函数
print(func02(tmp))

## 第二段代码
items=[i for i in range(10)]
tmp=func(m)#func 是一函数
print(func02(func(m)))# func02 是另一个函数
print(func02(func(m)))
```

请问这二者输出相同么？如果不考虑前文的铺垫，孤零零放出这么一段代码，可能 9 成都会不假思索的说相同。因为第二段代码，其实就是把一个变量多定义几遍而已。

由于 tmp=func(m)，所以 func02(tmp) 等价于 func02(func(m))，再结合软件开发中的业务逻辑确定性原则（同输入同输出），所以这两段代码输出必然相同。如果稍稍思考下就能发现，其实这个说法是有问题的，如果 func02 就是 next() 函数呢？显然输出会不同。

长久的编程习惯会让我们忽略一些东西，yield 就属于忽略点，其内部等价于内置了”状态机”的概念。

如果对上面代码持有异议，可运行如下代码

```
def flatten(items, ignore_types=(str, bytes)):
    for x in items:
        if isinstance(x, Iterable) and not isinstance(x, ignore_types):
            yield from flatten(x)
        else:
            yield x
以下不同
tmp=flatten(items)
print(next(flatten(items)))
print(next(flatten(items)))
print(next(flatten(items)))
print(next(flatten(items)))

tmp=flatten(items)
print(next(tmp))
print(next(tmp))
print(next(tmp))
print(next(tmp))
```

2.3 yield 简单介绍

yield 是什么东西呢，它相当于我们函数里的 return。在每次 next()，或者 for 遍历的时候，都会 yield 这里将新的值返回回去，并在这里阻塞，等待下一次的调用。记住要点，yield 相关的 **2 个动作和 2 个状态，return-自我阻塞-(别人)xx 动作-唤醒 (自己)**，完整就是,return 之后，自动，自我阻塞，然后，等待 xx 的动作，唤醒自己。这个要记牢，否则后面容易懵。

如何创建一个生成器，主要有如下两种方法

```
# 01, 使用列表生成式，注意不是 []，而是 ()
L = (x * x for x in range(10))
print(isinstance(L, Generator))  # True

# 02, 实现了 yield 的函数
def mygen(n):
    now = 0
    while now < n:
        yield now
        now += 1

if __name__ == '__main__':
    gen = mygen(10)
    print(isinstance(gen, Generator))  # True
```

如何运行/激活生成器

```
使用 next()
使用 generator.send(None)
```

这就是 yield 基础知识.

2.4 yield 个人理解 01，缓冲型 (延迟型)list

就是把 yield 的函数看做普通 list 列表，不过是他什么时候用到什么时候计算，不关心他怎么实现的 (或者什么时候真正执行的)。

这么理解，基本上大部分 yield 函数，看懂是没问题的，但是写呢？依然难以下手，因为不清楚每一行代码是如何运行的。我之前就是这么理解的，阅读足够，开发不足。

2.5 yield 个人理解 02，多 return 时序函数

首先以下定义结构

```
yield x=>等价于=>return x,receive y
```

举例: 执行到 yield 时

连线 1，返回。先 return index, 然后自我阻塞,

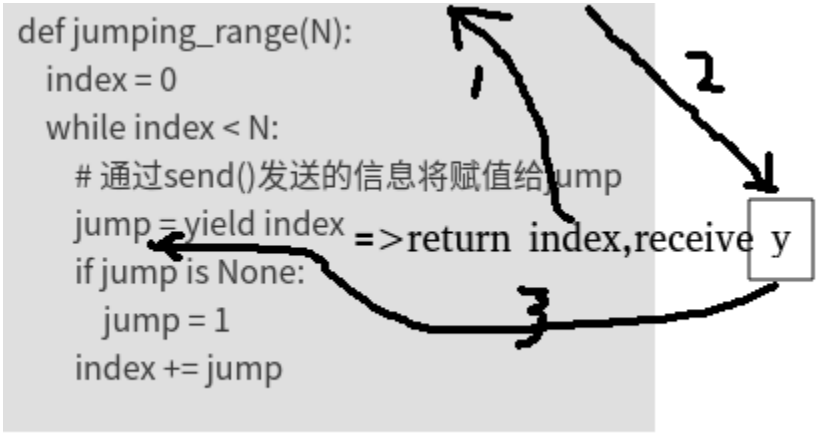
连线 2，等待填充。y 哪里其实是个空位，意味这等待外部向这里填充数据（如何填充? 前面介绍的 next or send）.

连线 3，填充后继续。填充数据之后，传递给 yield index, 中 yield 开始的位置，此处为 = 的右侧，所以 receive 后会赋值给 jump

所以说 yield 的执行其实是" 时序" 型的，一个 "时钟" 就是一个 next(or send), 每走一步，return 之后，等待, 等别人通过 next(or send)，叫醒自己继续走.

从时间角度看，yield 函数其实有" 一系列不同时间的返回值"。

所以其非常适合" 需要捕捉中间结果的迭代计算"



2.6 yield 实例和分析

```
def jumping_range(N):
    index = 0
    while index < N:
        # 通过 send() 发送的信息将赋值给 jump
        jump = yield index
        if jump is None:
            jump = 1
        index += jump

if __name__ == '__main__':
```

(下页继续)

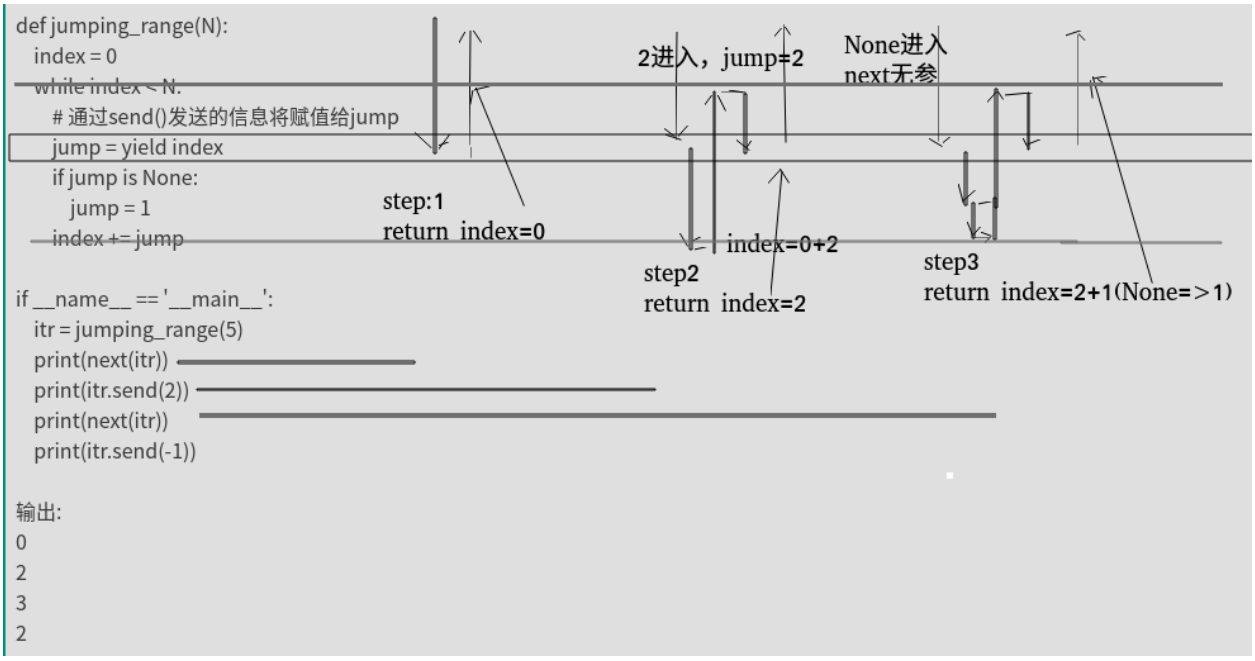
(续上页)

```
    itr = jumping_range(5)
    print(next(itr))
    print(itr.send(2))
    print(next(itr))
    print(itr.send(-1))

输出:
0
2
3
2
```

分析:

如下图，粗黑线部分就是程序代码执行流，可见 index=0 这部分代码其实只执行了一次, 后续每次 next 后程序起点都是 Jump=xx

这种角度看，yield 非常类似我们 debug 程序时加入的断点，断点处 return 当前取值, 然后 set 新的取值.



2.7 yield continue 和 break

```
def get_detection_result():
    tmpi = 0
```

(下页继续)

(续上页)

```
    ret = True
    while ret:
        if tmpi % 4 == 0:
            tmpi += 1
            yield #continue
        elif tmpi % 4 == 1:
            tmpi += 1
            ret = False
            yield #break ,will raise exception
        elif tmpi % 4 == 2:
            tmpi += 1
            m = yield tmpi
            print(m) # common multi return
        else:
            print('xxxxxxx') # dead loop
f = get_detection_result()
print(next(f))
print(f.send(101))
print(f.send(102))
```

2.8 胞兄 yield from

简单的理解

```
# 字符串
astr='ABC'
# 列表
alist=[1,2,3]
# 字典
adict={"name":"wangbm","age":18}
# 生成器
agen=(i for i in range(4,8))

def gen(*args, **kw):
    for item in args:
        # yield 方法
        for i in item:
            yield i
        # yield from 方法
```

(下页继续)

(续上页)

```
        #yield from item


new_list=gen(astr, alist, adict,  agen)
print(list(new_list))
# ['A', 'B', 'C', 1, 2, 3, 'name', 'age', 4, 5, 6, 7]
```

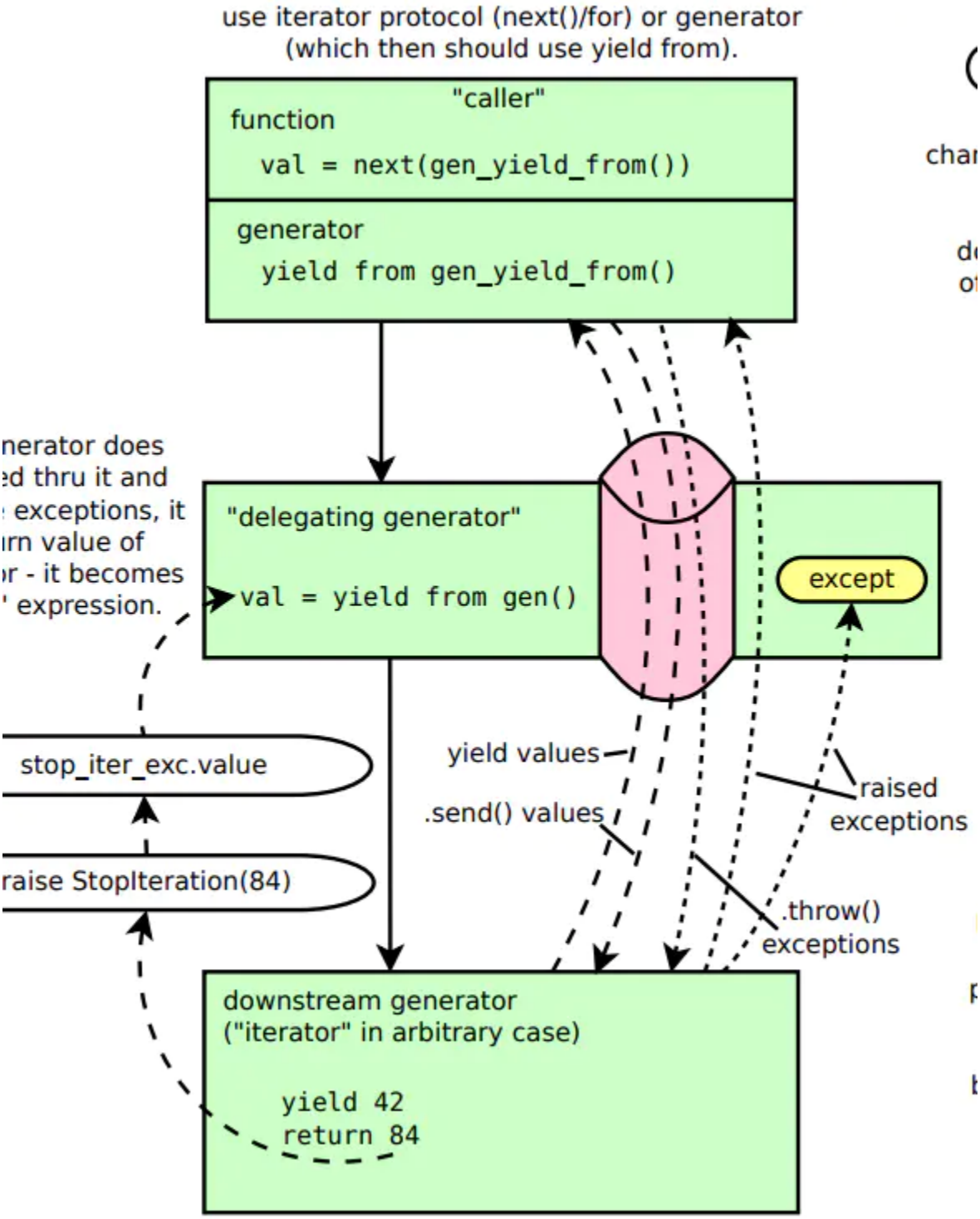
可以简单理解为

```
yield from item
等价于
for i in item:
    yield i
```

委托生成器的作用是：在调用方与子生成器之间建立一个双向通道。

所谓的双向通道是什么意思呢?

调用方可以通过 send() 直接发送消息给子生成器，而子生成器 yield 的值，也是直接返回给调用方。



2.9 参考文献

Python 并发编程之从生成器使用入门协程（七）

Python 并发编程之深入理解 yield from 语法（八）

Python yield from 用法详解

CHAPTER 3

python 进阶 03UnboundLocalError 和 NameError 错误

3.1 几个概念

code block：作为一个单元 (Unit) 被执行的一段 python 程序文本。例如一个模块、函数体和类的定义等。

scope：在一个 code block 中定义 name 的可见性；

block’s environment：对于一个 code block，其所有 scope 中可见的 name 的集合构成 block 的环境。

bind name：下面的操作均可视为绑定操作

```
函数的形参
import 声明
类和函数的定义
赋值操作
for 循环首标
异常捕获中相关的赋值变量
```

local variable：如果 name 在一个 block 中被绑定，该变量便是该 block 的一个 local variable。

global variable：如果 name 在一个 module 中被绑定，该变量便称为一个 global variable。

free variable: 如果一个 name 在一个 block 中被引用，但没有在该代码块中被定义，那么便称为该变量为一个 free variable。

关于变量作用域参考博文：python 进阶 14 变量作用域 LEGB

3.2 经典案例 1

```
def outer_func():
    loc_var = "local variable"
    def inner_func():
        loc_var += " in inner func"
        return loc_var
    return inner_func

clo_func = outer_func()
clo_func()
```

错误提示:

```
Traceback (most recent call last):
  File "G:\Project Files\Python Test\Main.py", line 238, in <module>
    clo_func()
  File "G:\Project Files\Python Test\Main.py", line 233, in inner_func
    loc_var += " in inner func"
UnboundLocalError: local variable 'loc_var' referenced before assignment
```

3.3 经典案例 2

```
def get_select_desc(name, flag, is_format = True):
    if flag:
        sel_res = 'Do select name = %s' % name
    return sel_res if is_format else name

get_select_desc('Error', False, True)
```

错误提示:

```
Traceback (most recent call last):
  File "G:\Project Files\Python Test\Main.py", line 247, in <module>
    get_select_desc('Error', False, True)
  File "G:\Project Files\Python Test\Main.py", line 245, in get_select_desc
    return sel_res if is_format else name
UnboundLocalError: local variable 'sel_res' referenced before assignment
```

3.4 经典案例 3

```
def outer_func(out_flag):
    if out_flag:
        loc_var1 = 'local variable with flag'
    else:
        loc_var2 = 'local variable without flag'
    def inner_func(in_flag):
        return loc_var1 if in_flag else loc_var2
    return inner_func

clo_func = outer_func(True)
print clo_func(False)
```

错误提示:

```
Traceback (most recent call last):
  File "G:\Project Files\Python Test\Main.py", line 260, in <module>
    print clo_func(False)
  File "G:\Project Files\Python Test\Main.py", line 256, in inner_func
    return loc_var1 if in_flag else loc_var2
NameError: free variable 'loc_var2' referenced before assignment in enclosing scope
```

3.5 分析案例

```
import sys
a = 1
for i in range(10):
    pass


def outer_func(g=3):
    loc_var = "local variable"

    def inner_func():
        print(a)
        print(g)

        try:
```

(下页继续)

(续上页)

```
            int("0.2")
        except ValueError as e:
            print(e)

        loc_var += " in inner func"
        return loc_var

    return inner_func


clo_func = outer_func()
clo_func()


class clss():
    def add(self):
        pass
```

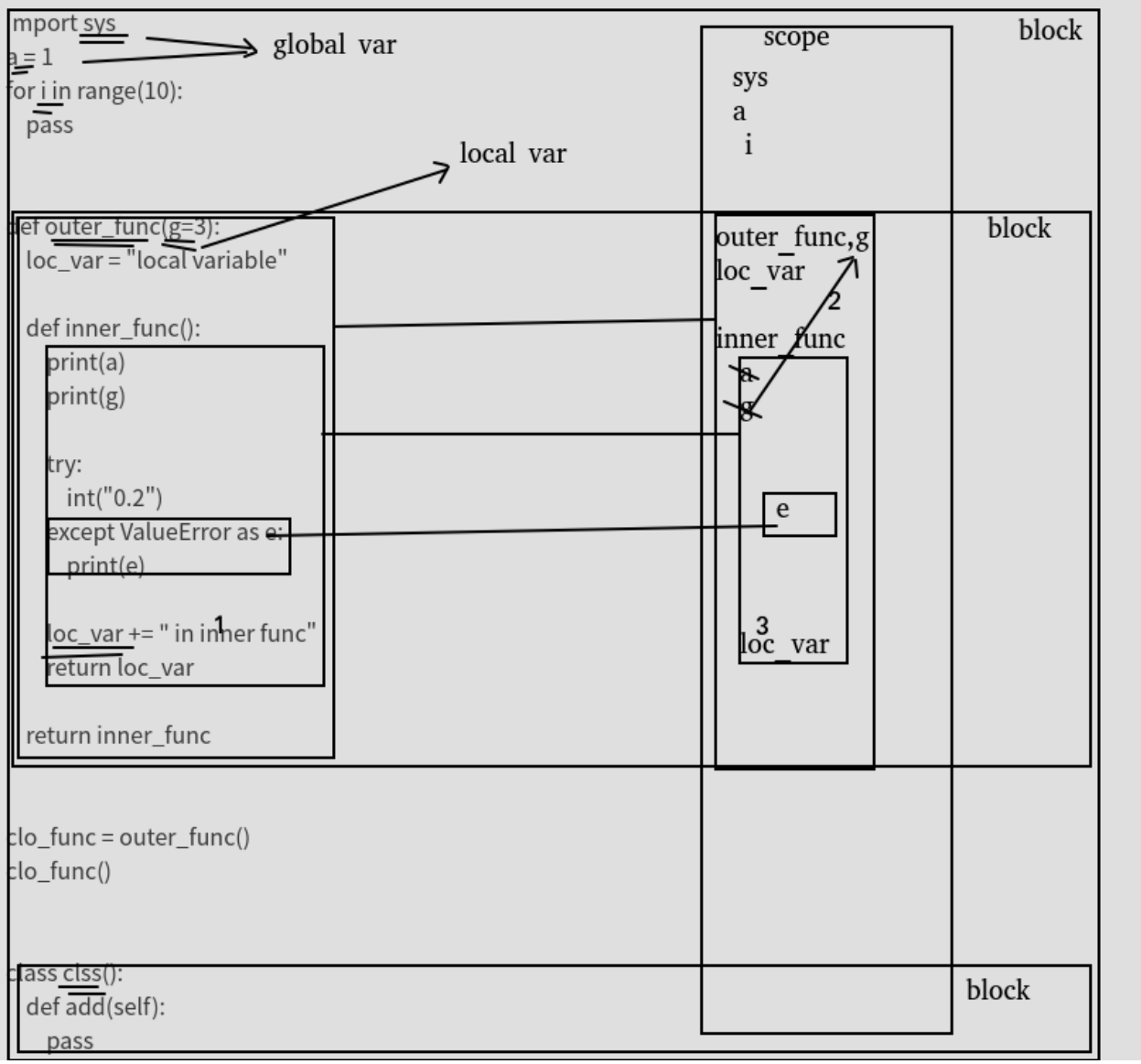
分析

1, 双下划线: 根据 bind 定义，识别为 scope 可见的变量

2,scope 可见是存在层次关系的, 越靠左侧说明包含范围越大（类似作用域的感觉), 越靠右侧，包含范围越小，（作用域越小）

3, 图中表明 1 和 3 的地方就是，标准案例 1 的问题点.loc_var 由于在赋值语句“=”的左边，所以别认为是 scope 内 (可见) 变量，所以执行 loc_var=loc_var+” xx” 时，会从最近的 scope 加载 loc_var, 最近的 loc_var 恰好指向自己，但是此时的” loc_var” 尚且未被初始化，所以报错。

4, 相比较，图中 2, 变量” g” 在局部也未被初始化，但是却未报错，应为 g 在 scope 中未找到，所以自动向上找 (往范围更大的找)，就找到入参处的变量” g” 了。



3.6 参考

UnboundLocalError，探讨 Python 中的绑定

Python UnboundLocalError 和 NameError 错误根源解析

CHAPTER 4

python 进阶 04IO 的同步异步，阻塞非阻塞

4.1 同步和异步

同步和异步关注的是**消息通信机制**。

所谓同步，就是在发出一个**调用**时，在没有得到结果之前，该调用就不返回。但是一旦调用返回，就得到返回值了。换句话说，就是由调用者主动等待这个调用的结果。

而异步则是相反，**调用**在发出之后，这个调用就直接返回了，所以没有返回结果。换句话说，当一个异步过程调用发出后，调用者不会立刻得到结果。而是在**调用**发出后，**被调用者**通过状态、通知来通知调用者，或通过回调函数处理这个调用。

举个通俗的例子：你打电话问书店老板有没有《分布式系统》这本书，如果是同步通信机制，书店老板会说，你稍等，”我查一下”，然后开始查啊查，等查好了（可能是 5 秒，也可能是一天）告诉你结果（返回结果）。而异步通信机制，书店老板直接告诉你我查一下啊，查好了打电话给你，然后直接挂电话了（不返回结果）。然后查好了，他会主动打电话给你。在这里老板通过“回电”这种方式来回调。

同步的整个流程就 1 部分,1 问答，异步模式 2 部分，1 问空答复 + 无问 1 答复。

4.2 阻塞和非阻塞

阻塞和非阻塞关注的是程序在**等待调用结果（消息，返回值）时的状态.** 阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。

非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

你打电话问书店老板有没有《分布式系统》这本书，你如果是阻塞式调用，你会一直把自己“挂起”，直到得到这本书有没有的结果，如果是非阻塞式调用，你不管老板有没有告诉你，你自己先一边去玩了，当然你也要偶尔过几分钟 check 一下老板有没有返回结果。在这里阻塞与非阻塞与是否同步异步无关。跟老板通过什么方式回答你结果无关。

还是上面的例子，你打电话问书店老板有没有《分布式系统》这本书，你如果是阻塞式调用，你会一直把自己“挂起”，直到得到这本书有没有的结果，如果是非阻塞式调用，你不管老板有没有告诉你，你自己先一边去玩了，当然你也要偶尔过几分钟 check 一下老板有没有返回结果。在这里阻塞与非阻塞与是否同步异步无关。跟老板通过什么方式回答你结果无关。如果是关心阻塞 IO/ 异步 IO, 参考 Unix Network Programming View Book–还是 2014 年写的以解释概念为主，主要是同步异步阻塞和非阻塞会被用在不同层面上，可能会有不准确的地方，并没有针对阻塞 IO/ 异步 IO 等进行讨论，大家可以后续看看这两个回答:

作者：Yi Lu

链接：https://www.zhihu.com/question/19732473/answer/20851256

4.3 例子: 老张水壶

老张爱喝茶，废话不说，煮开水。

出场人物：老张，水壶两把（普通水壶，简称水壶；会响的水壶，简称响水壶）。

1 老张把水壶放到火上，立等水开。(同步阻塞)

2 老张把水壶放到火上，去客厅看电视，时不时去厨房看看水开没有。(同步非阻塞)

老张还是觉得自己有点傻，于是变高端了，买了把会响笛的那种水壶。水开之后，能大声发出嘀 ~~~~ 的噪音。

3 老张把响水壶放到火上，立等水开。(异步阻塞)

4 老张把响水壶放到火上，去客厅看电视，水壶响之前不再去看它了，响了再去拿壶。(异步非阻塞)

所谓同步异步，只是对于水壶而言。

普通水壶，同步；响水壶，异步。虽然都能干活，但响水壶可以在自己完工之后，提示老张水开了。这是普通水壶所不能及的。同步只能让调用者去轮询自己（情况 2 中），造成老张效率的低下。

所谓阻塞非阻塞，仅仅对于老张而言。

立等的老张，阻塞；看电视的老张，非阻塞。情况 1 和情况 3 中老张就是阻塞的，媳妇喊他都不知道。虽然 3 中响水壶是异步的，可对于立等的老张没有太大的意义。所以一般异步是配合非阻塞使用的，这样才能发挥异步的效用。

作者：愚抄

链接：https://www.zhihu.com/question/19732473/answer/23434554

4.4 同步异步，阻塞非阻塞

二者其实是不同维度的东西，但是的确容易搞混，

代码特征：

做法 A(同步，阻塞)：开单独线程（进程）处理，线程内一般是循环接收消息。

做法 B(同步，非阻塞)：主线程循环处理，但一般搭配 sleep 函数 (释放时间片，自身状态运行转就绪再等待时间片)，避免完全的 cpu 空转

做法 C(异步，非阻塞): 函数参数包含函数 (回调函数)，或 url 地址，类似的东西，对方有消息就推送给你，所以还需要新开一个接口 (or 功能) 用来接受信息。

至于 A,B 实例可参考本博客的 “django 进阶系列 02websocket”，C 实例就不多说了，js 中非常常见。

4.5 参考

怎样理解阻塞非阻塞与同步异步的区别

NIO 详解（二）：BIO 浅谈同步异步与阻塞非阻塞

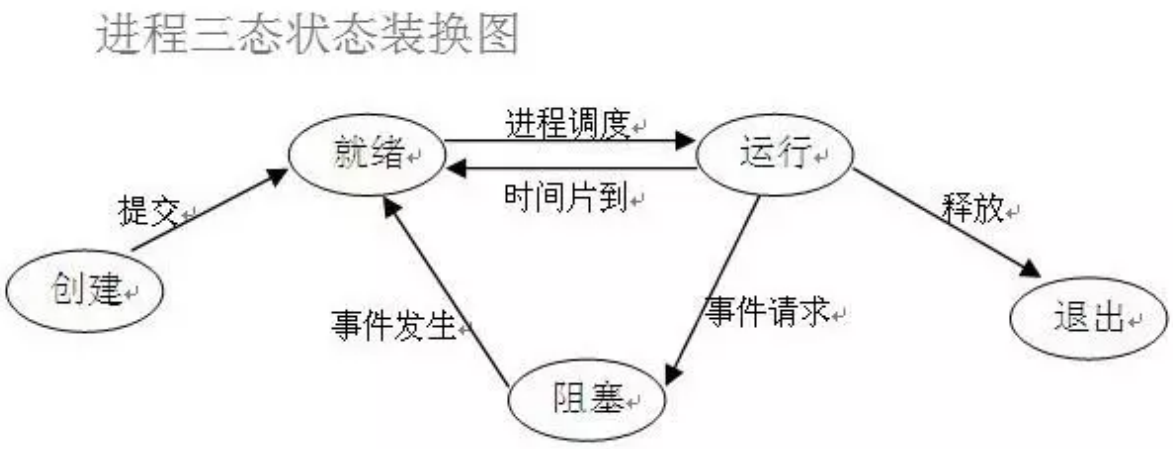
怎样理解同步异步和阻塞非阻塞

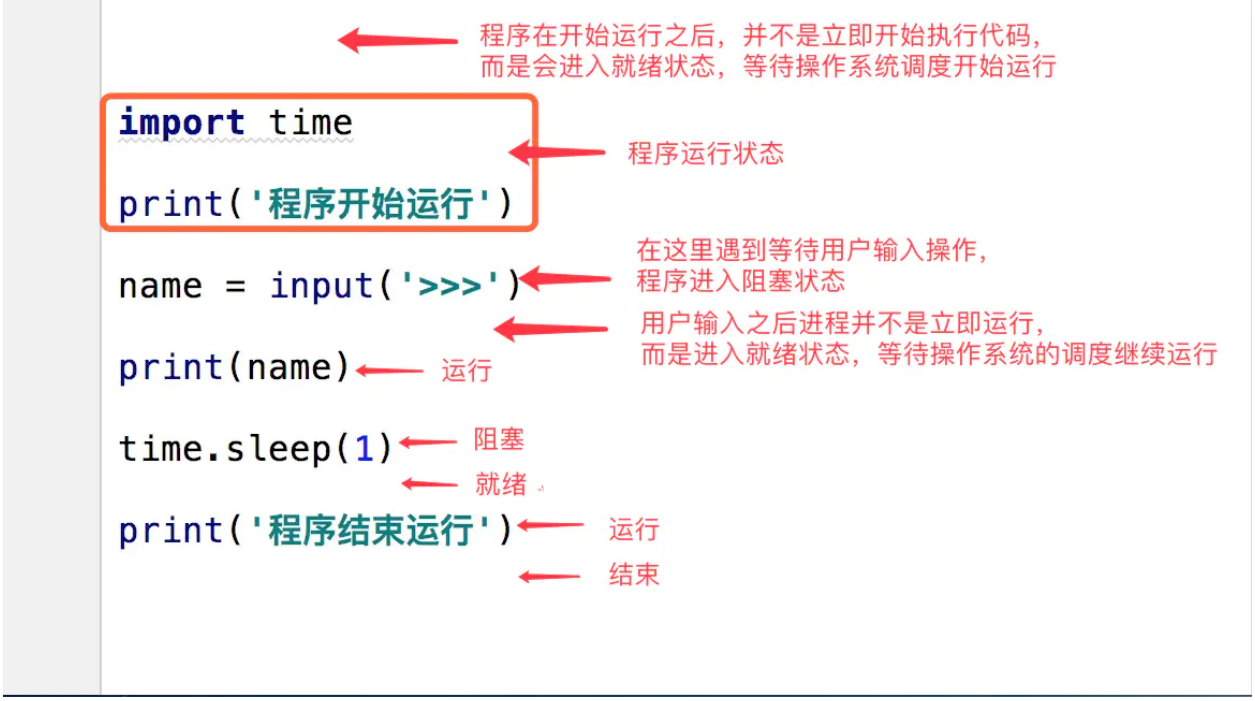
CHAPTER 5

python 进阶 05 并发之一基本概念

大多编程语言，一旦涉及并发，都会比较复杂，知识点也较多（大多为历史问题，很多技术点现在非常少使用了，但语言层面也提供支持，对于这些冷门点，只需要知道即可，使用时也**尽量避免使用这种冷门技术**，除非和应用场景非常匹配）。实际使用过程中，只需要知道各名词以及大概功能，大多现用现查，毕竟涉及点太多，而且使用频率也并非很高，一般也就新系统研发会使用，后续维护时基本不会涉及太多。

5.1 进程状态和调度





5.2 进程，线程，协程

1、进程

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，进程是系统进行**资源**分配和调度的一个独立单位。每个进程都有自己的独立内存空间，不同进程通过进程间通信来通信。由于进程比较重量，占据独立的内存，所以上下文进程间的切换开销（栈、寄存器、虚拟内存、文件句柄等）比较大，但相对比较稳定安全。

2、线程

线程是进程的一个实体，是 **CPU 调度**和分派的基本单位，它是比进程更小的能独立运行的基本单位. 线程自己基本上不拥有系统资源，只拥有一点在运行中必不可少的资源 (如程序计数器，一组寄存器和栈)，但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。线程间通信主要通过共享内存，上下文切换很快，资源开销较少，但相比进程不够稳定容易丢失数据。

3、协程

协程是一种**用户态的轻量级线程**，协程的调度完全由用户控制。协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈，直接操作栈则基本没有内核切换的开销，**可以不加锁的访问全局变量**，所以上下文的切换非常快。

5.3 多进程和多线程

在数据共享、同步方面，多进程是数据共享复杂，需要用 IPC，数据是分开的，同步简单。多线程因为共享进程数据，数据共享简单，但同步复杂；

在内存、CPU 方面，多进程占用内存多，切换复杂，CPU 利用率低。

多线程占用内存少，切换简单，CPU 利用率高；

在创建销毁、切换方面，多进程创建销毁、切换复杂，速度慢。多线程创建销毁、切换简单，速度很快；

在编程、调试方面，**多进程编程和调试都简单**。**多线程编程和调试都复杂**；

可靠性方面，多进程间不会互相影响。**多线程中的一个线程挂掉将导致整个进程挂掉；**

在分布式方面，多进程适应于多核、多机分布式。多线程适应于多核分布式。

多进程模式最大的优点就是稳定性高，因为一个子进程崩溃了，不会影响主进程和其他子进程。（当然主进程挂了所有进程就全挂了，但是 Master 进程只负责分配任务，挂掉的概率低）著名的 Apache 最早就是采用多进程模式。

多进程模式的缺点是创建进程的代价大，在 Unix/Linux 系统下，用 fork 调用还行，在 Windows 下创建进程开销巨大。另外，操作系统能同时运行的进程数也是有限的，在内存和 CPU 的限制下，如果有几千个进程同时运行，操作系统连调度都会成问题。

多线程模式通常比多进程快一点，但是也快不到哪去，而且，**多线程模式致命的缺点**就是任何一个线程挂掉都可能直接造成**整个进程崩溃**，因为所有线程共享进程的内存。在 Windows 上，如果一个线程执行的代码出了问题，你经常可以看到这样的提示：“该程序执行了非法操作，即将关闭”，其实往往是某个线程出了问题，但是操作系统会强制结束整个进程。

在 Windows 下，多线程的效率比多进程要高，所以微软的 IIS 服务器默认采用多线程模式。由于多线程存在稳定性的问题，IIS 的稳定性就不如 Apache。为了缓解这个问题，IIS 和 Apache 现在又有多进程 + 多线程的混合模式，真是把问题越搞越复杂。

5.4 线程和协程

协程: 又被称为用户级线程或绿色线程。

1. 一个线程可以多个协程，一个进程也可以单独拥有多个协程，这样 python 中则能使用多核 CPU。
2. 线程进程都是同步机制，而协程则是异步
3. 协程能保留上一次调用时的状态，每次过程重入时，就相当于进入上一次调用的状态

5.5 事件驱动 (协程依赖)

在 UI 编程中，常常要对鼠标点击进行相应，首先如何获得鼠标点击呢?

方式一：创建一个线程，该线程一直循环检测是否有鼠标点击，那么这个方式有以下几个缺点：

1. CPU 资源浪费，可能鼠标点击的频率非常小，但是扫描线程还是会一直循环检测，这会造成很多的 CPU 资源浪费；如果扫描鼠标点击的接口是阻塞的呢？
2. 如果是堵塞的，又会出现下面这样的问题，如果我们不但要扫描鼠标点击，还要扫描键盘是否按下，由于扫描鼠标时被堵塞了，那么可能永远不会去扫描键盘；
3. 如果一个循环需要扫描的设备非常多，这又会引来响应时间的问题；

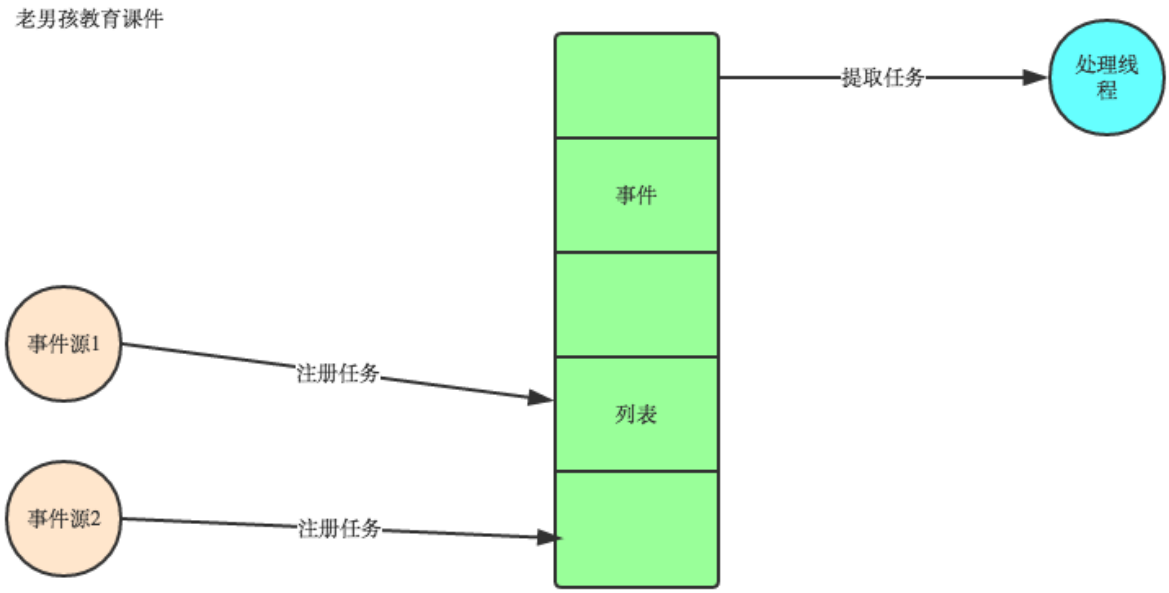
所以，该方式是非常不好的

方式二：就是事件驱动模型

目前大部分的 UI 编程都是事件驱动模型，如很多 UI 平台都会提供 onClick() 事件，这个事件就代表鼠标按下事件。事件驱动模型大体思路如下：

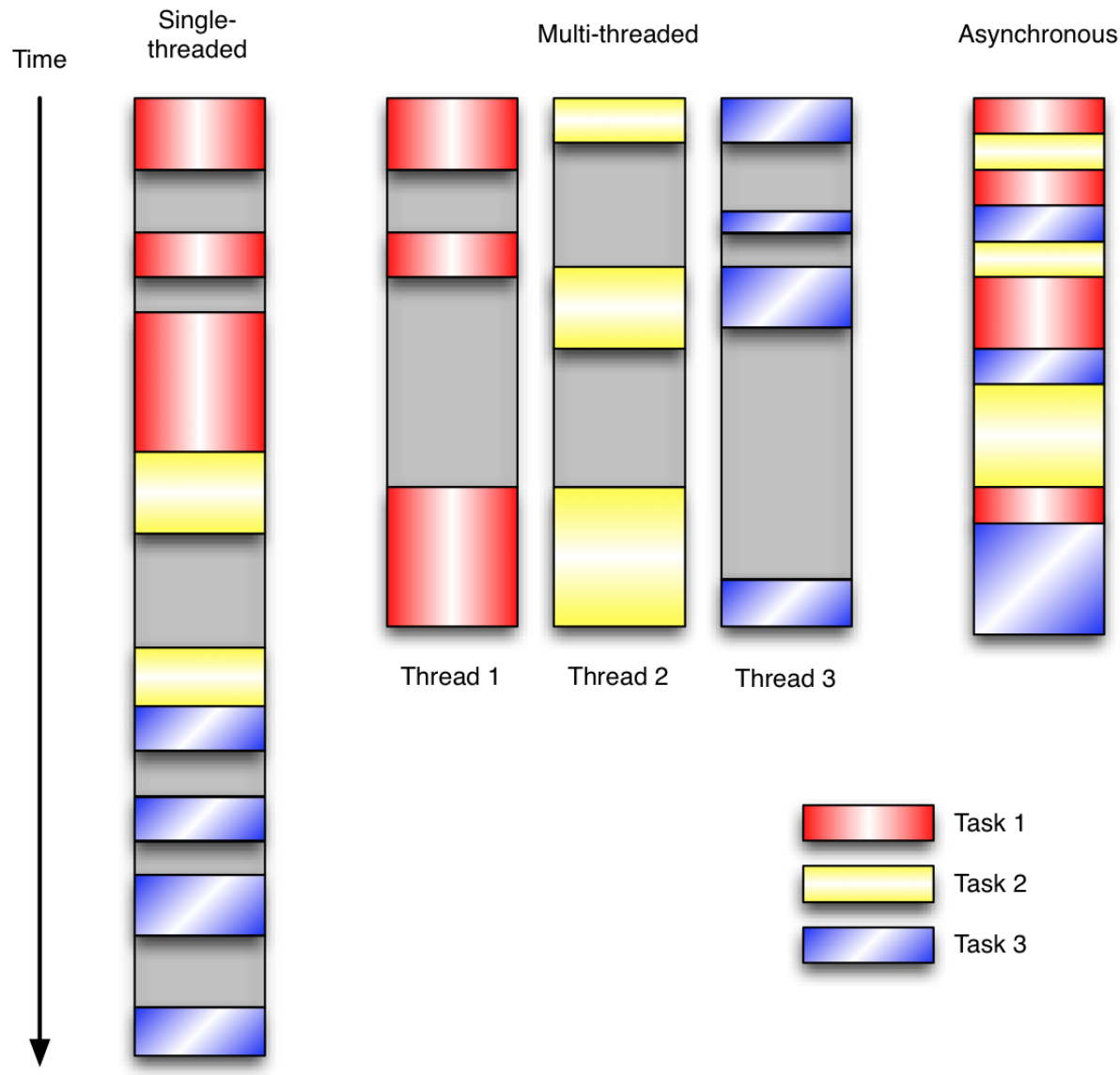
1. 有一个事件（消息）队列；
2. 鼠标按下时，往这个队列中增加一个点击事件（消息）；
3. 有个循环，不断从队列取出事件，根据不同的事件，调用不同的函数，如 onClick()、onKeyDown() 等；

4. 事件（消息）一般都各自保存各自的处理函数指针，这样，每个消息都有独立的处理函数；



事件驱动编程是一种编程范式，这里程序的执行流由外部事件来决定。它的特点是包含一个事件循环，当外部事件发生时使用回调机制来触发相应的处理。另外两种常见的编程范式是（单线程）同步以及多线程编程。

让我们用例子来比较和对比一下单线程、多线程以及事件驱动编程模型。下图展示了随着时间的推移，这三种模式下程序所做的工作。这个程序有 3 个任务需要完成，每个任务都在等待 I/O 操作时阻塞自身。阻塞在 I/O 操作上所花费的时间已经用灰色框标示出来了



当我们面对如下的环境时，事件驱动模型通常是一个好的选择：

程序中有许多任务，而且… 任务之间高度独立（因此它们不需要互相通信，或者等待彼此）而且… 在等待事件到来时，某些任务会阻塞。 当应用程序需要在任务间共享可变的数据时，这也是一个不错的选择，因为这里不需要采用同步处理。

网络应用程序通常都有上述这些特点，这使得它们能够很好的契合事件驱动编程模型。

总结：异步 IO 涉及到了事件驱动模型，进程中维护一个**消息队列**，当客户端又请求时，就会把请求添加到消息队列中，线程从消息队列中**轮询取要处理的请求**，遇到 I/O 阻塞时（操作系统处理调用 I/O 接口处理，与程序无关），则进行上下文切换，处理其他请求，当 I/O 操作完成时，调用回调函数，告诉线程处理完成，然后再切换回来，处理完成后返回给客户端 Nginx 能处理高并发就是用的这个原理

5.6 参考

进程和线程、协程的区别

进程 vs. 线程

以 Python 爬取数据为例，多线程和多进程的优劣

在多核 CPU 下，同一进程下的多个线程可以并行运行吗

python 并发编程之多进程 (实践篇)

python 多进程原理及其实现（1-6 总结，较好）

python 之路多进程和多线程总结 (四)

一文看懂 Python 多进程与多线程编程 (工作学习面试必读)

搞定 python 多线程和多进程（详细）

Python 的进程间通信

Python 进程间共享数据（三）（dict、list）

多进程, 多线程, 协程实现简单举例

异步 IO、多线程、多进程

Python 开发【第九章】：线程、进程和协程

multiprocess 模块使用进程池调用 apply_async() 提交的函数及回调函数不执行问题

协程/进程/线程资料收集

CHAPTER 6

python 进阶 06 并发之二技术点关键词

6.1 GIL, 线程锁

python 中存在 GIL 这个”线程锁”，

关键地方可以使用 c 语言解决 GIL 问题然后可以提高 cpu 占用效率

6.2 守护进程

主进程创建守护进程

1）守护进程会在主进程代码执行结束后就终止

2）守护进程内无法再开启子进程，否则抛出异常：AssertionError: daemonic processes are not allowed to have children

注意：进程之间是互相独立的，主进程代码运行结束，守护进程随即终止

```
# 主进程代码运行完毕，守护进程就会结束
from multiprocessing import Process
from threading import Thread
import time
def foo():
    print(123)
    time.sleep(1)
```

(下页继续)

(续上页)

```
    print("end123")

def bar():
    print(456)
    time.sleep(3)
    print("end456")


p1=Process(target=foo)
p2=Process(target=bar)

p1.daemon=True
p1.start()
p2.start()
print("main-------") # 打印该行则主进程代码结束，则守护进程 p1 应该被终止，可能会有 p1 任务执行的打印信息 123，因为主进程打印 main----时，p1 也执行了，但是随即被终止
```

6.3 互斥锁 (mutex)

为了方式上面情况的发生，就出现了互斥锁 (Lock)

```
import threading
import time


def run(n):
    lock.acquire()  # 获取锁
    global num
    num += 1
    lock.release()  # 释放锁

lock = threading.Lock()     # 实例化一个锁对象

num = 0
t_obj = []

for i in range(20000):
    t = threading.Thread(target=run, args=("t-%s" % i,))
    t.start()
```

(下页继续)

(续上页)

```
    t_obj.append(t)

for t in t_obj:
    t.join()

print "num:", num
```

6.4 RLock 递归锁 (了解)

6.5 队列 (推荐)

Queue 是多进程的安全队列，可以使用 Queue 实现多进程之间的数据传递。

```
Queue.qsize(): 返回当前队列包含的消息数量;
Queue.empty(): 如果队列为空，返回 True，反之 False ;
Queue.full(): 如果队列满了，返回 True，反之 False;
Queue.get(): 获取队列中的一条消息，然后将其从列队中移除，可传参超时时长。
Queue.get_nowait(): 相当 Queue.get(False)，取不到值时触发异常: Empty;
Queue.put(): 将一个值添加进数列，可传参超时时长。
Queue.put_nowait(): 相当于 Queue.get(False)，当队列满了时报错: Full。

from multiprocessing import Process, Queue
import time

def write(q):
   for i in ['A', 'B', 'C', 'D', 'E']:
      print('Put %s to queue' % i)
      q.put(i)
      time.sleep(0.5)

def read(q):
   while True:
      v = q.get(True)
      print('get %s from queue' % v)

if __name__ == '__main__':
   q = Queue()
   pw = Process(target=write, args=(q,))
```

(下页继续)

(续上页)

```
    pr = Process(target=read, args=(q,))
    print('write process = ', pw)
    print('read  process = ', pr)
    pw.start()
    pr.start()
    pw.join()
    pr.join()
    pr.terminate()
    pw.terminate()
```

6.6 管道 (了解)

6.7 共享数据 (Manager)

```
if __name__ == '__main__':
    with multiprocessing.Manager() as MG: # 重命名
        mydict=MG.dict()# 主进程与子进程共享这个字典
        mylist=MG.list(range(5))# 主进程与子进程共享这个 LIST

        p=multiprocessing.Process(target=func,args=(mydict,mylist))

        p.start()
        p.join()

        print(mylist)
        print(mydict)
```

6.8 信号量 (了解)

6.9 事件

事件（Event 类）python 线程的事件用于主线程控制其他线程的执行，事件是一个简单的线程同步对象，其主要提供以下几个方法：

事件处理的机制：全局定义了一个 “Flag”，当 flag 值为 “False”，那么 event.wait() 就会阻塞，当 flag 值为 “True”，那么 event.wait() 便不再阻塞。

```
# 利用 Event 类模拟红绿灯
import threading
import time

event = threading.Event()

def lighter():
    count = 0
    event.set()     # 初始值为绿灯
    while True:
        if 5 < count <=10 :
            event.clear()  # 红灯，清除标志位
            print("\33[41;1mred light is on...\033[0m")
        elif count > 10:
            event.set()  # 绿灯，设置标志位
            count = 0
        else:
            print("\33[42;1mgreen light is on...\033[0m")

        time.sleep(1)
        count += 1

def car(name):
    while True:
        if event.is_set():      # 判断是否设置了标志位
            print("[%s] running..."%name)
            time.sleep(1)
        else:
            print("[%s] sees red light,waiting..."%name)
            event.wait()
            print("[%s] green light is on,start going..."%name)

light = threading.Thread(target=lighter,)
light.start()

car = threading.Thread(target=car,args=("MINI",))
car.start()
```

6.10 fork

Unix/Linux 操作系统提供了一个 fork() 系统调用，它非常特殊。普通的函数调用，调用一次，返回一次，但是 fork() 调用一次，返回两次，因为操作系统自动把当前进程（称为父进程）复制了一份（称为子进程），然后，分别在父进程和子进程内返回。

子进程永远返回 0，而父进程返回子进程的 ID。这样做的理由是，一个父进程可以 fork 出很多子进程，所以，父进程要记下每个子进程的 ID，而子进程只需要调用 getppid() 就可以拿到父进程的 ID。

Python 的 os 模块封装了常见的系统调用，其中就包括 fork，可以在 Python 程序中轻松创建子进程：

```
import os

print('Process (%s) start...' % os.getpid())
# Only works on Unix/Linux/Mac:
pid = os.fork()
if pid == 0:
    print('I am child process (%s) and my parent is %s.' % (os.getpid(), os.getppid()))
else:
    print('I (%s) just created a child process (%s).' % (os.getpid(), pid))
```

运行结果如下：

```
Process (876) start...
I (876) just created a child process (877).
I am child process (877) and my parent is 876.
```

由于 Windows 没有 fork 调用，上面的代码在 Windows 上无法运行。而 Mac 系统是基于 BSD（Unix 的一种）内核，所以，在 Mac 下运行是没有问题的，

6.11 Process 模块

1. 注意：Process 对象可以创建进程，**但 Process 对象不是进程**，其**删除与否与系统资源是否被回收没有直接的关系**。

2. 主进程执行完毕后会默认等待子进程结束后回收资源，不需要手动回收资源；join() 函数用来控制子进程结束的顺序，其**内部也有一个清除僵尸进程的函数，可以回收资源**；

3.Process 进程创建时，子进程会将主进程的 Process 对象完全复制一份，这样在主进程和子进程各有一个 Process 对象，**但是 p.start() 启动的是子进程，主进程中的 Process 对象作为一个静态对象存在，不执行**。

4. 当**子进程执行完毕后，会产生一个僵尸进程，其会被 join 函数回收**，或者再有一条进程开启，**start 函数也会回收僵尸进程**，所以不一定需要写 join 函数。

5.windows 系统在子进程结束后会立即自动清除子进程的 Process 对象，而 **linux 系统子进程的 Process 对象如果没有 join 函数和 start 函数的话会在主进程结束后统一清除**。

进程直接的内存空间是隔离的

```
from multiprocessing import Process
n=100 # 在 windows 系统中应该把全局变量定义在 if __name__ == '__main__' 之上就可以了
def work():
    global n
    n=0
    print('子进程内: ',n)


if __name__ == '__main__':
    p=Process(target=work)
    p.start()
    print('主进程内: ',n)
```

6.12 multiprocessing 模块

Process 模块是一个创建进程的模块, 借助这个模块可以创建进程

```
Process([group [, target [, name [, args [, kwargs]]]]]), 由该类实例化得到的对象，表示一个子进程中的任务（尚未启动）
```

强调:

1. 需要使用关键字的方式来指定参数
2. args 指定的为传给 target 函数的位置参数，是一个元组形式，必须有逗号

参数介绍:

group 参数未使用，值始终为 None

target 表示调用对象，即子进程要执行的任务

args 表示调用对象的位置参数元组，args=(1,2,’egon’,)

kwargs 表示调用对象的字典,kwargs={‘name’:’egon’,’age’:18}

name 为子进程的名称

方法介绍

p.start()：启动进程，并调用该子进程中的 p.run()

p.run(): 进程启动时运行的方法，正是它去调用 target 指定的函数，我们自定义类的类中一定要实现该方法

p.terminate(): 强制终止进程 p，不会进行任何清理操作，如果 p 创建了子进程，该子进程就成了僵尸进程，使用该方法需要特别小心这种情况。如果 p 还保存了一个锁那么也将不会被释放，进而导致死锁

p.is_alive(): 如果 p 仍然运行，返回 True

p.join([timeout]): 主线程等待 p 终止（强调：是主线程处于等的状态，而 p 是处于运行的状态）。

timeout 是可选的超时时间，需要强调的是，p.join 只能 join 住 start 开启的进程，而不能 join 住 run 开启的进程

属性介绍

p.daemon：默认值为 False，如果设为 True，代表 p 为后台运行的守护进程，当 p 的父进程终止时，p 也随之终止，

并且设定为 True 后，p 不能创建自己的新进程，必须在 p.start() 之前设置

p.name: 进程的名称

p.pid：进程的 pid

p.exitcode: 进程在运行时为 None、如果为–N，表示被信号 N 结束 (了解即可)

p.authkey: 进程的身份验证键, 默认是由 os.urandom() 随机生成的 32 字符的字符串。这个键的用途是

为涉及网络连接的底层进程间通信提供安全性，这类连接只有在具有相同的身份验证键时才能成功（了解即可）

window 中使用 Process 注意事项:

在 Windows 操作系统中由于没有 fork(linux 操作系统中创建进程的机制)，在创建子进程的时候会自动 import 启动它的这个文件，而在 import 的时候又执行了整个文件。因此如果将 process() 直接写在文件中就会无限递归创建子进程报错。所以必须把创建子进程的部分使用 if **name** == '**main**' 判断保护起来，import 的时候，就不会递归运行了。

6.13 进程池

由于进程启动的开销比较大，使用**多进程的时候会导致大量内存空间被消耗**。为了防止这种情况发生可以使用进程池，（由于启动线程的开销比较小，所以不需要线程池这种概念，多线程只会频繁得切换 cpu 导致系统变慢，并不会占用过多的内存空间）

进程池中常用方法：

```
1 p.apply(func [, args [, kwargs]])
在一个池工作进程中执行 func(*args,**kwargs)，然后返回结果。
需要强调的是：此操作并不会在所有池工作进程中并执行 func 函数。如果要通过不同参数并发地执行
func 函数，必须从不同线程调用 p.apply() 函数或者使用 p.apply_async()
2 p.apply_async(func [, args [, kwargs]]):
```

(下页继续)

(续上页)

```
在一个池工作进程中执行 func(*args,**kwargs)，然后返回结果。
此方法的结果是 AsyncResult 类的实例，callback 是可调用对象，接收输入参数。当 func 的结果变为可用时，
将理解传递给 callback。callback 禁止执行任何阻塞操作，否则将接收其他异步操作中的结果。
如果传递给 apply_async() 的函数如果有参数，需要以元组的形式传递 并在最后一个参数后面加上 ,␣
↪号，如果没有加，号，提交到进程池的任务也是不会执行的
3 p.close(): 关闭进程池，防止进一步操作。如果所有操作持续挂起，它们将在工作进程终止前完成
4 P.jion(): 等待所有工作进程退出。此方法只能在 close () 或 teminate() 之后调用

from  multiprocessing import Process,Pool
import time

def Foo(i):
    time.sleep(2)
    return i+100

def Bar(arg):
    print('-->exec done:',arg)

pool = Pool(5)  # 允许进程池同时放入 5 个进程

for i in range(10):
    pool.apply_async(func=Foo, args=(i,),callback=Bar)  #func 子进程执行完后，才会执行
callback，否则 callback 不执行（而且 callback 是由父进程来执行了）
    #pool.apply(func=Foo, args=(i,))

print('end')
pool.close()
pool.join() # 主进程等待所有子进程执行完毕。必须在 close() 或 terminate() 之后。
```

进程池内部维护一个进程序列，当使用时，去进程池中获取一个进程，如果进程池序列中没有可供使用的进程，那么程序就会等待，直到进程池中有可用进程为止。在上面的程序中产生了 10 个进程，但是只能有 5 同时被放入进程池，剩下的都被暂时挂起，并不占用内存空间，等前面的五个进程执行完后，再执行剩下 5 个进程。

回调函数：进程池支持回调函数

6.14 协程 (gevent)

协程:

能够在一个线程中实现并发效果的效果, 提高 cpu 的利用率

无需原子操作锁定及同步的开销

能够规避一些任务中的 IO 操作

方便切换控制流，简化编程模型

协程相比于多线程的优势切换的效率更快了

缺点:

无法利用多核资源：协程的本质是个单线程, 它不能同时将单个 CPU 的多个核用上, 协程需要和进程配合才能运行在多 CPU 上. 当然我们日常所编写的绝大部分应用都没有这个必要，除非是 cpu 密集型应用。

线程和进程的操作是由程序触发系统接口，最后的执行者是系统，它本质上是操作系统提供的功能。而协程的操作则是程序员指定的，在 python 中通过 yield，人为的实现并发处理。

协程存在的意义：对于多线程应用，CPU 通过切片的方式来切换线程间的执行，线程切换时需要耗时。协程，则只使用一个线程，分解一个线程成为多个“微线程”，在一个线程中规定某个代码块的执行顺序。

协程的适用场景：当程序中存在大量不需要 CPU 的操作时（IO）。

常用第三方模块 gevent 和 greenlet。(本质上，gevent 是对 greenlet 的高级封装，因此一般用它就行，这是一个相当高效的模块。)

greenlet

```
from greenlet import greenlet

def test1():
    print(12)
    gr2.switch()
    print(34)
    gr2.switch()

def test2():
    print(56)
    gr1.switch()
    print(78)

gr1 = greenlet(test1)
gr2 = greenlet(test2)
gr1.switch()
```

实际上，greenlet 就是通过 switch 方法在不同的任务之间进行切换。

gevent

```
from gevent import monkey; monkey.patch_all()
import gevent
import requests

def f(url):
    print('GET: %s' % url)
    resp = requests.get(url)
    data = resp.text
    print('%d bytes received from %s.' % (len(data), url))

gevent.joinall([
        gevent.spawn(f, 'https://www.python.org/'),
        gevent.spawn(f, 'https://www.yahoo.com/'),
        gevent.spawn(f, 'https://github.com/'),
])
```

通过 joinall 将任务 f 和它的参数进行统一调度，实现单线程中的协程。代码封装层次很高，实际使用只需要了解它的几个主要方法即可。

6.15 ThreadLocal

创建一个全局的 ThreadLocal 对象, 每个线程有独立的存储空间, 每个线程对 ThreadLocal 对象都可以读写, 但是互不影响.

```
import threading

# 创建全局 ThreadLocal 对象:
local = threading.local()

def process_student():
    # 获取当前线程关联的 student:
    print('local.student: %s , current_thread : %s' % (local.student, threading.current_
↪thread().name))

def process_thread(stu_name):
    # 绑定 ThreadLocal 的 student:
    local.student = stu_name
    process_student()

t1 = threading.Thread(target= process_thread, args=('Alice',), name='Thread-A')
```

(下页继续)

(续上页)

```
t2 = threading.Thread(target= process_thread, args=('Bob',), name='Thread-B')
t1.start()
t2.start()
t1.join()
t2.join()
# local.student: Alice , current_thread : Thread-A
# local.student: Bob , current_thread : Thread-B
```

全局变量 local 就是一个 ThreadLocal 对象，每个 Thread 对它都可以读写 student 属性，但互不影响。你可以把 local 看成全局变量，但每个属性如 local.student 都是线程的局部变量，

可以任意读写而互不干扰，也不用管理锁的问题，ThreadLocal 内部会处理。

可以理解为全局变量 local 是一个 dict，不但可以用 local.student，还可以绑定其他变量，

如 local.teacher 等等。

应用:ThreadLocal 最常用的地方就是为每个线程绑定一个数据库连接，HTTP 请求，用户身份信息等，

这样一个线程的所有调用到的处理函数都可以非常方便地访问这些资源。

小结

一个 ThreadLocal 变量虽然是全局变量，但每个线程都只能读写自己线程的独立副本，互不干扰。

ThreadLocal 解决了参数在一个线程中各个函数之间互相传递的问题。

```
import threading, time
local = threading.local()  # 创建一个全局的 ThreadLocal 对象
num = 0  # 将线程中需要访问的变量绑定到全局 ThreadLocal 对象上

def run(x, n):
    x = x + n
    x = x - n
    return  x

def func(n):
    # 每个线程都有 local.x, 就是线程的局部变量
    local.x = num                                                    # 在线程调用的函
数中, 将访问的变量和 ThreadLock 绑定
    for i in range(1000000):
        run(local.x, n)
    print("%s-  local.x =%d"%(threading.current_thread().name, local.x))

if __name__ == "__main__":
```

(下页继续)

(续上页)

```
    iTimeStart = time.time()
    t1 = threading.Thread(target=func, args=(6,))
    t2 = threading.Thread(target=func, args=(9,))
    t1.start()
    t2.start()
    t1.join()
    t2.join()

    print("num =",num)
    iTimeEnd = time.time()
    print(iTimeEnd - iTimeStart)   # 1.6630573272705078
# 不仅不会导致数据混乱，而且所用时间已经接近不加锁的时间.
```

6.16 参考

进程和线程、协程的区别

进程 vs. 线程

以 Python 爬取数据为例，多线程和多进程的优劣

在多核 CPU 下，同一进程下的多个线程可以并行运行吗

python 并发编程之多进程 (实践篇)

python 多进程原理及其实现（1-6 总结，较好）

python 之路多进程和多线程总结 (四)

一文看懂 Python 多进程与多线程编程 (工作学习面试必读)

搞定 python 多线程和多进程（详细）

Python 的进程间通信

Python 进程间共享数据（三）（dict、list）

多进程, 多线程, 协程实现简单举例

异步 IO、多线程、多进程

Python 开发【第九章】：线程、进程和协程

multiprocess 模块使用进程池调用 apply_async() 提交的函数及回调函数不执行问题

CHAPTER 7

python 进阶 07 并发之三其他问题

7.1 何时使用多进程 (线程)

使用多进程 or 线程, 对于 python，考虑到 GIL 锁，基本上**默认使用多进程**就对了。

除此之外，线程**共享全局变量**，进程**全局变量则是隔离的**，实际进程大多数情况需要通信的，所以也需要考虑共享数据读写问题处理。决定因素稳定性和数据共享要求上（操作系统差异性，win 偏好线程，linux 偏好进程）

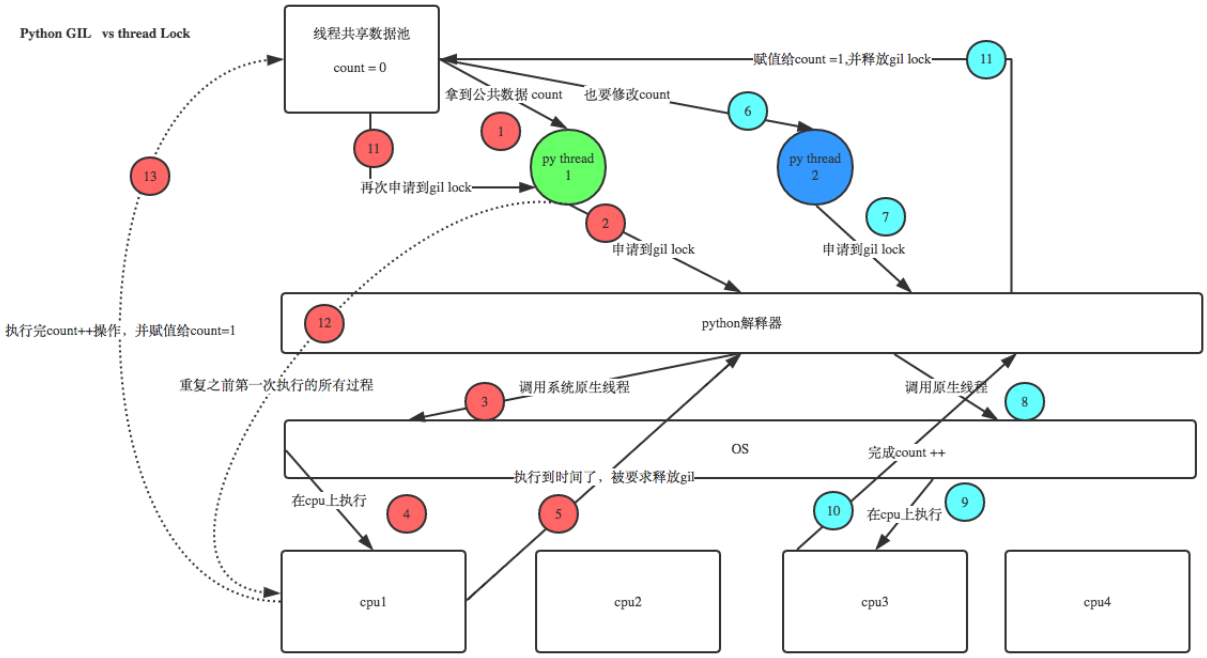
性能上虽然线程较好，但实际除了大型互联网公司和部分专业性质软件，大多数中小型公司的并发量，并不会带来很大影响，况且目前服务器领域，基本上 LInux 和 Unix 占比较高，线程相比进程在性能上优势并不十分突出。所以这方面考量不会太大的。

以下几种情况可考虑多进程 (线程)

```
循环（互相独立，无内部依赖）
耗时操作，批量下载 IO 等操作，典型的是网络爬虫
分叉型计算，典型的是分治算法，或 mapreduce 的 map 阶段
```

7.2 python 多线程既然有 GIL 锁为何还需要加锁

Python 已经有一个 GIL 来保证同一时间只能有一个线程来执行了，为什么这里还需要 lock? 注意啦，这里的 lock 是用户级的 lock, 跟那个 GIL 没关系，具体我们通过下图来看一下



简单来说就是，**“GIL 同一时间只有一个线程执行”是微观层面锁**，但我们需要加锁的是**宏观层面锁**。

比如锁 (process_a1,process_a2,process_a3), 和锁 (process_b1,process_b2,process_b3)，如果去掉锁，仅仅依赖 GIL 的机制, 则可能出现：process_a1,process_b1,process_a2.process_b2,process_a3,process_b3，可能导致错误数据出现.

相比之下，**协程里的多线程（的共享数据访问）则是不需要加锁的**！因为从**用户层面看，协程是多线程的**。但**虚拟机层面，其实单线程的**，协程内部进行 cpu 执行权利的自行控制。关于单线程实现多线程并发执行，可参考博文”并发之一基本概念的事件模型部分”。

7.3 同进程不同线程可运行在不同核心上？

这个还要看编程语言的线程模型。**某些语言（如 python）的线程模型不支持并行运行在多个核上**

实际上有些操作系统在内部并不分进程和线程，调度方式是一致的。比如 **Linux，共享内存的就是线程，不共享内存的就是进程**，然后把包装好的创建函数暴露给 POSIX API。

现在同一进程下的多个线程是可以在多核 CPU 下并行运行的。但 2.4 内核及以前的系统实现的线程没有内核支持，无法在多核的情况下并行运行。

多线程的概念主要有两种：一种是**用户态多线程**；一种是**内核态多线程**

内核态多线程，如楼上所言，在操作系统内核的支持下可以在多核下并行运行；

对于用户态多线程，尽管没有内核的直接支持，但若一个用户态线程对应于内核的一个进程的话（从这个角度，内核还是间接支持的），仍然是可以在多核上并行运行的。

因此，这归结为，用户态多线程的实现技术。

似乎目前 Linux 上的用户态多线程，就是利用了内核的进程来实现的。

如果是内核线程（就是 fork 出来的，pthread_create 在 2.4 后最终也用 fork，具体参看其实现），那么可以调度到多 cpu，内核支持线程的诱导因素之一就是可以利用多 cpu 资源进行并行计算；如果是用户线程，那么就不能在多 cpu 上并行计算了，用户库线程的弊端之一就是不能利用多 cpu 资源；

7.4 线程是并发还是并行，进程是并发还是并行？

线程是并发，进程是并行；

现代 os 都将**线程作为最小调度单位**，进程作为**资源分配的最小单位**。

7.5 父子进程如何区分?

子进程是父进程通过 fork() 产生出来的，pid = os.fork()

通过**返回值 pid 是否为 0**，判断是否为子进程，如果是 0，则表示是子进程

由于 fork() 是 Linux 上的概念，所以如果要跨平台，最好还是使用 subprocess 模块来创建子进程。

7.6 子进程如何回收？

python 中采用 os.wait() 方法用来回收子进程占用的资源

pid, result = os.wait() # 回收子进程资源　阻塞，等待子进程执行完成回收

如果有子进程没有被回收的，但是父进程已经死掉了，这个子进程就是僵尸进程。孤儿进程, 父类进程 over., 子进程未结束

7.7 使用多处理池的 apply_async 方法时，谁运行回调

回调在主进程中处理，但单独线程 (循环方式依次处理)

参考：使用多处理池的 apply_async 方法时，谁运行回调?

7.8 异常处理，异常消失问题

7.9 参考

进程和线程、协程的区别

进程 vs. 线程

以 Python 爬取数据为例，多线程和多进程的优劣

在多核 CPU 下，同一进程下的多个线程可以并行运行吗

python 并发编程之多进程 (实践篇)

python 多进程原理及其实现（1-6 总结，较好）

python 之路多进程和多线程总结 (四)

一文看懂 Python 多进程与多线程编程 (工作学习面试必读)

搞定 python 多线程和多进程（详细）

Python 的进程间通信

Python 进程间共享数据（三）（dict、list）

多进程, 多线程, 协程实现简单举例

异步 IO、多线程、多进程

Python 开发【第九章】：线程、进程和协程

multiprocess 模块使用进程池调用 apply_async() 提交的函数及回调函数不执行问题

CHAPTER 8

python 进阶 08 并发之四 map, apply, map_async, apply_async 差异

8.1 差异矩阵

python 封装了 4 种常用方法，用于实现并发

其差异如下

需要注意：map 和 map_async 入参为迭代器类型，可以批量调用。而 apply 和 apply_async 只能一个个调用。

```
# map
results = pool.map(worker, [1, 2, 3])

# apply
for x, y in [[1, 1], [2, 2]]:
    results.append(pool.apply(worker, (x, y)))

def collect_result(result):
    results.append(result)

# map_async
pool.map_async(worker, jobs, callback=collect_result)

# apply_async
```

(下页继续)

(续上页)

```
for x, y in [[1, 1], [2, 2]]:
    pool.apply_async(worker, (x, y), callback=collect_result)
```

8.2 apply 和 apply_async

Pool.apply_async：调用立即返回而不是等待结果。AsyncResult 返回一个对象。你调用其 get() 方法以检索函数调用的结果。该 get() 方法将阻塞直到功能完成。

因此，pool.apply(func, args, kwargs) 等效于 pool.apply_async(func, args, kwargs).get()。

相比 Pool.apply，该 Pool.apply_async 方法还具有一个回调，则在函数完成时调用该回调。可以使用它来代替 get()。

```
import multiprocessing as mp
import time

def foo_pool(x):
    time.sleep(2)
    return x*x

result_list = []
def log_result(result):
    # This is called whenever foo_pool(i) returns a result.
    # result_list is modified only by the main process, not the pool workers.
    result_list.append(result)

def apply_async_with_callback():
    pool = mp.Pool()
    for i in range(10):
        pool.apply_async(foo_pool, args = (i, ), callback = log_result)
    pool.close()
    pool.join()
    print(result_list)

if __name__ == '__main__':
    apply_async_with_callback()
可能会产生如下结果

[1, 0, 4, 9, 25, 16, 49, 36, 81, 64]
```

还要注意，可使用调用许多不同的函数 Pool.apply_async（并非所有调用都需要使用同一函数）。

相反，Pool.map 将相同的函数应用于许多参数。但是，与不同 Pool.apply_async，返回结果的顺序与参数的顺序相对应。

8.3 参考

Python multiprocessing.Pool: Difference between map, apply, map_async, apply_async

Python-multiprocessing.Pool：何时使用 apply，apply_async 或 map?

CHAPTER 9

python 进阶 09 并发之五生产者消费者

这也是实际项目中使用较多的一种并发模式，用 Queue(JoinableQueue) 实现，是 Python 中最常用的方式 (这里的 queue 特指 multiprocess 包下的 queue，非 queue.Queue)。

9.1 Queue

```
# encoding:utf-8
__author__ = 'Fioman'
__time__ = '2019/3/7 14:06'
from multiprocessing import Process,Queue
import time,random

def consumer(q,name):
    while True:
        food = q.get()
        if food is None:
            print('接收到了一个空，生产者已经完事了')
            break

        print('\033[31m{}消费了{}\033[0m'.format(name,food))
        time.sleep(random.random())
```

(下页继续)

(续上页)

```
def producer(name,food,q):
    for i in range(10):
        time.sleep(random.random())
        f = '{}生产了{}{}'.format(name,food,i)
        print(f)
        q.put(f)



if __name__ == '__main__':
    q = Queue(20)
    p1 = Process(target=producer,args=('fioman','包子',q))
    p2 = Process(target=producer,args=('jingjing','馒头',q))
    p1.start()
    p2.start()

    c1 = Process(target=consumer,args=(q,'mengmeng'))
    c2 = Process(target=consumer,args=(q,'xiaoxiao'))
    c1.start()
    c2.start()

    # 让主程序可以等待子进程的结束.
    p1.join()
    p2.join()
    # 生产者的进程结束，这里需要放置两个空值，供消费者获取，用来判断已经没有存货了
    q.put(None)
    q.put(None)

    print('主程序结束..........')
```

9.2 JoinableQueue

创建可连接的共享进程队列, 它们也是队列, 但是这些队列比较特殊. 它们可以允许消费者通知生产者项目已经被成功处理. 注意, 这里必须是生产者生产完了, 生产者的进程被挂起, 等到消费者完全消费的时候, 生产者进程就结束, 然后主程序结束. 将消费者进程设置为守护进程, 这样的话, 主进程结束的时候, 消费进程也就结束了.

JoinableQueue() 比普通的 Queue() 多了两个方法:

q.task_done()
使用者使用此方法发出信号，表示 q.get() 返回的项目已经被处理。如果调用此方法的次数大于从队列中删除的项目数量，将引发 ValueError 异常。

q.join()
生产者将使用此方法进行阻塞，直到队列中所有项目均被处理。阻塞将持续到为队列中的每个项目均调用 q.task_done() 方法为止。

```
# encoding:utf-8
__author__ = 'Fioman'
__time__ = '2019/3/7 14:06'
from multiprocessing import Process,JoinableQueue
import time,random

def consumer(q,name):
    while True:
        food = q.get()
        if food is None:
            print(' 接收到了一个空，生产者已经完事了')
            break

        print('\033[31m{} 消费了{}\033[0m'.format(name,food))
        time.sleep(random.random())
        q.task_done()  # 向生产者发送信号，表示消费了一个

def producer(name,food,q):
    for i in range(10):
        time.sleep(random.random())
        f = '{} 生产了{}{}'.format(name,food,i)
        print(f)
        q.put(f)
    q.join() # 当生产者生产完毕之后，会在这里阻塞. 等待消费者的消费.



if __name__ == '__main__':
    q = JoinableQueue(20)
    p1 = Process(target=producer,args=('fioman',' 包子',q))
    p2 = Process(target=producer,args=('jingjing',' 馒头',q))
    p1.start()
    p2.start()
```

(下页继续)

(续上页)

```
c1 = Process(target=consumer,args=(q,'mengmeng'))
c2 = Process(target=consumer,args=(q,'xiaoxiao'))
c1.daemon = True # 将消费者设置为守护进程
c2.daemon = True # 将消费者设置为守护进程
c1.start()
c2.start()

# 让主程序可以等待生产子进程的结束.
p1.join()
p2.join()



print(' 主程序结束.....')
```

个人不习惯使用 JoinableQueue，为什么呢？因为他是通过生产者来“得知”，整个生产消费流程的终结.

在消费者调用 q.task_done() 时，会触发一次 q.join() 的检查 (q.join() 是用来阻塞进程的，最后一个任务完成后，q.task_done() =》 q.join() =》 阻塞解除)，之后生产者进程退出。而消费者呢？业务逻辑层面上是没有退出的（本例)。比如，本例中通过**设置为守护进程的方式进行退出**。但如果后续主进程还有其他任务，而没有退出呢？那么这些子进程则沦为僵尸进程，虽然对系统资源消耗很少 (消费者的 queue.get() 也是阻塞的，所以不会执行循环，仅仅会“卡”在那里，但也不会自动消亡)，但感觉非常别扭的。所以个人还是倾向于用”生产者 queue.put(None) , 消费者见到 None 则 break(退出循环)”的传统方式进行消费者进程触发退出。如果采用这种方式那么 JoinableQueue 相比 Queue 就没有优势了。

9.3 一点思考

关于生产者和消费者，曾经思考过这么一种实现方式。

假如有一种队列，内置了**状态信息（存活生产者个数)**，设置目前存活的生产者个数

StatusableQueue(product_n=2,size=20)　　#product_n=2 含义：存活的生产者个数,size=20, 队列长度

生产者：生产结束，q.product_n - 1(存活生产者个数-1)

消费者：存活生产者个数 =0(生产者均已经完成生成) 且队列长度 =0(队列也已经消费结束) 则退出消费者进程.

这种情况下，只需要　消费者.join()　就可以保证整个生产消费进程的执行结束 (这一点和 JoinableQueue 很像，不过 JoinableQueue 是生产者.join())

一共只改动 3 处，就可以完成生产者消费者的并行化控制。而且更符合逻辑，因为生产者是明确知道自己的退出条件的，而消费者依赖生产者，所以只需要观察消费者就可以知道（生成者是否结束）整个 - 生成消费链是否完成。

```
def consumer(q,name):
    while not (q.product_n==0 and q.size==0):# 存活生产者 =0，意味着全部结束生产，队列不会新增数据,queue.size=0 说明消费完毕
        food = q.get()
        print('\033[31m{}消费了{}\033[0m'.format(name,food))
        time.sleep(random.random())

def producer(name,food,q):
    for i in range(10):
        time.sleep(random.random())
        f = '{}生产了{}{}'.format(name,food,i)
        print(f)
        q.put(f)
    q.product_n -= 1 # 当生产者生产完毕之后,q.product_n - 1(存活生产者个数-1)



if __name__ == '__main__':
    q = StatusableQueue(product_n=2,size=20)# 默认状态 = 正常,n=2 含义：生产者个数,
↪size=20, 对列长度
    p1 = Process(target=producer,args=('fioman','包子',q))
    p2 = Process(target=producer,args=('jingjing','馒头',q))
    p1.start()
    p2.start()

    c1 = Process(target=consumer,args=(q,'mengmeng'))
    c2 = Process(target=consumer,args=(q,'xiaoxiao'))
    c1.start()
    c2.start()

    # 消费者消费结束（说明生产也一定结束了），则说明整个生产 - 消费逻辑完成
    c1.join()
    c2.join()
```

文中加注释地方为修改点，这样代码最简单，调用方面，含义清晰。

缺点：**必须知道生产者个数**，这个数据应该不难获取，毕竟后面在创建生产者时也需要使用这个变量控制。

9.4 参考

Python 的进程间通信

CHAPTER 10

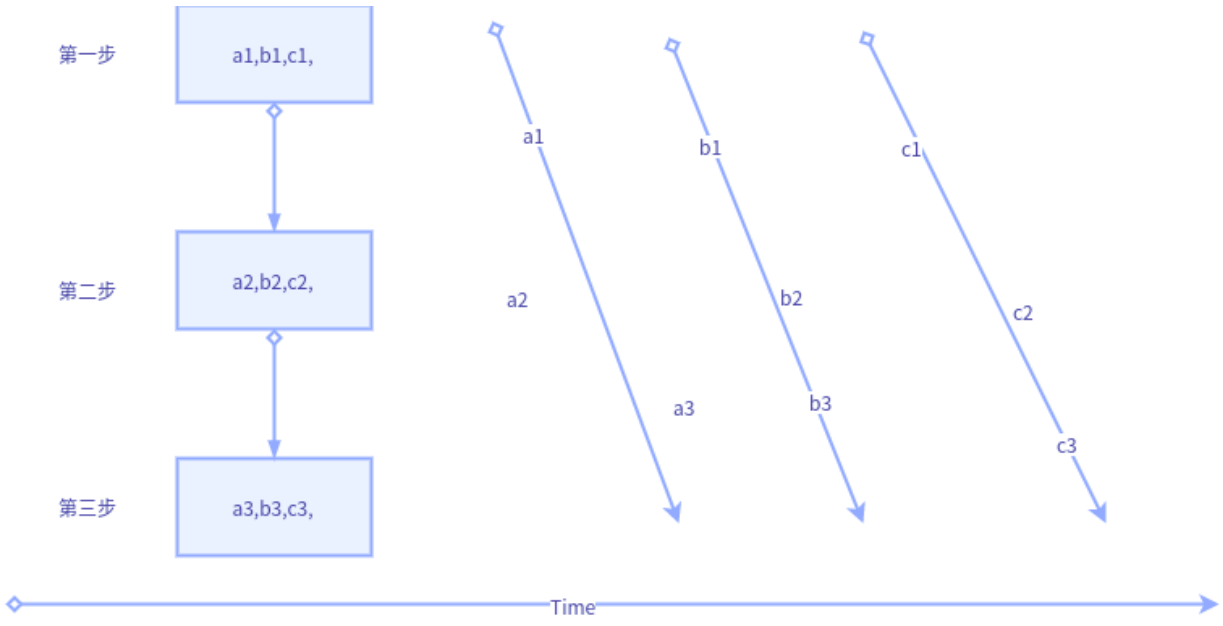
python 进阶 10 并发之六并行化改造

图示变量含义说明:

1 个大 step 中包含 3 个小 step, 大 step 内部的第一步，二步，三步存在依赖关系（就是内部保持顺序执行）

a1,b1,c1, 表示子任务 a 的第一步，b 的第一步，c 的第一步. 同理 a2, 表示子任务 a 的第二步。

10.1 无并行

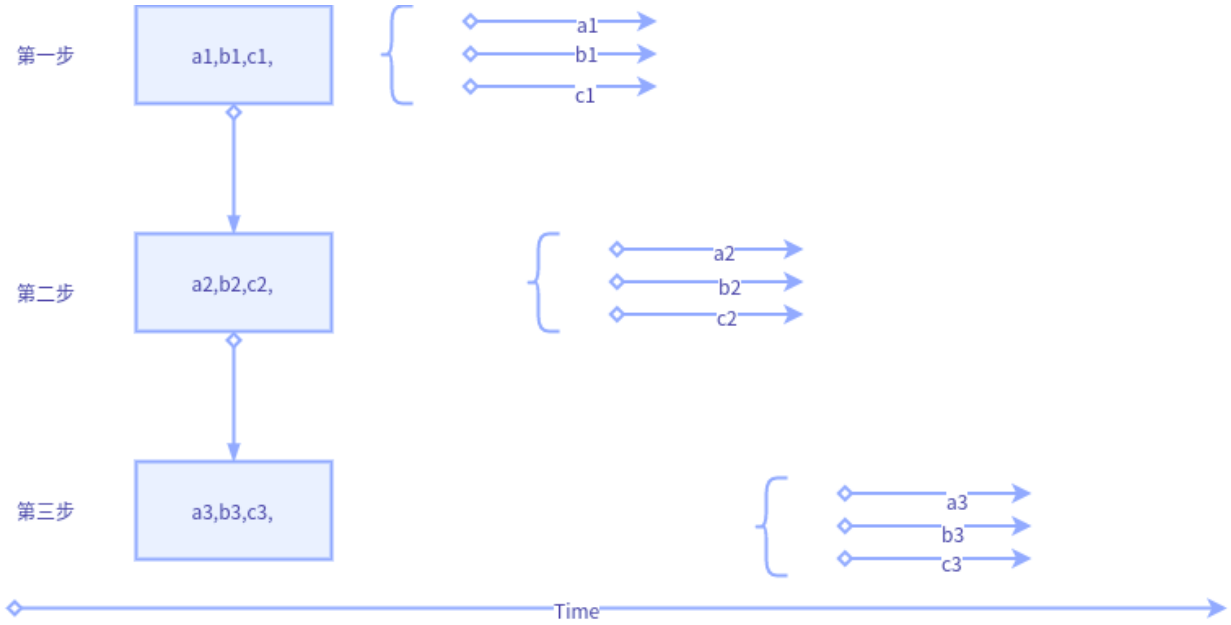


10.2 水平并行

优点：修改简单，容易排错

缺点：使用场景有限，适合批量数据，不适合流式数据

实现：pool().map(step1);pool().map(step2);pool().map(step3)

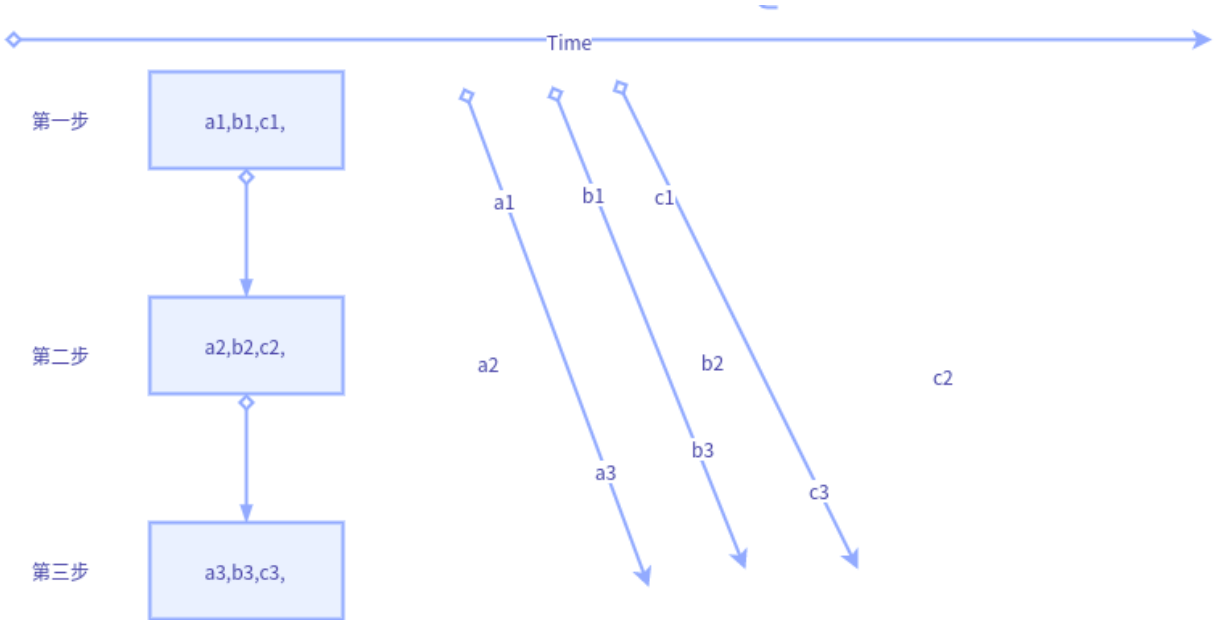


10.3 垂直并行

优点：修改简单，容易排错

缺点：如果中间步骤耗时过多，上游依然处于限制状态

实现：pool().apply_sync(func(step1,step2,step3))

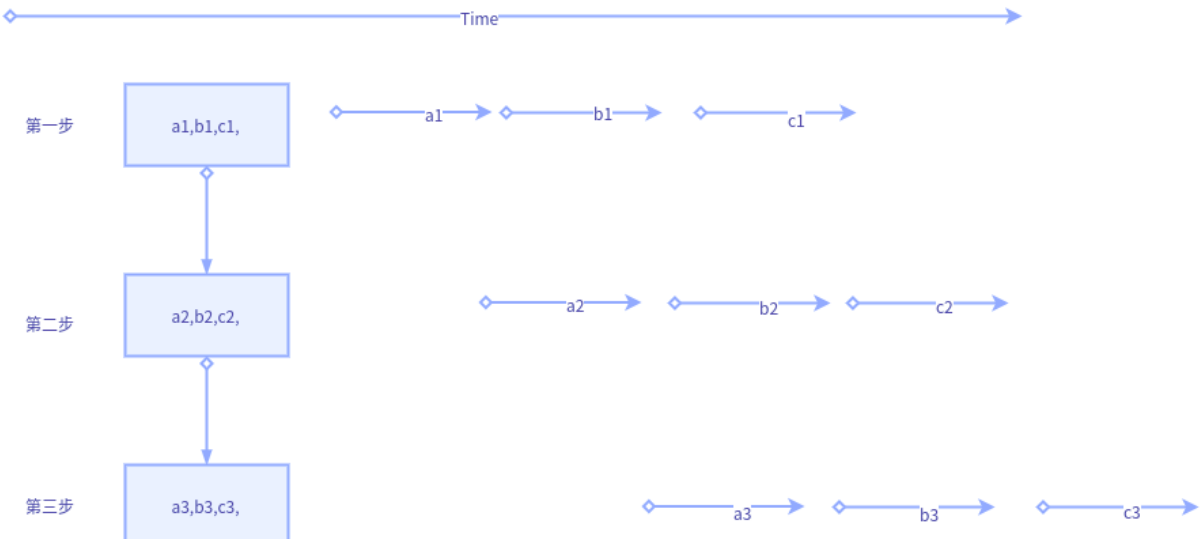


10.4 生产者消费者

优点：上下游耦合小，任务粒度更细

缺点：改造稍微复杂，需考虑生产者生成结束等特殊情况的兼容，并且调试也较麻烦

实现：Queue(多进程 multiprocessing,Queue, 多线程 queue.Queue,python 大多多进程)



10.5 协程

协成可看做特殊单线程（意味着本质是单线程，多线程是表象），任务角度多个线程同时执行，实时角度看只有一个线程真正执行，好处是无需处理线程共享数据的加锁等情况（因为只有一个线程会执行，不存在同时修改的情况）。还有就是其**进程内部不需要操作系统调度**（进程整体肯定是操作系统调度，否则就凌驾与操作系统了），**会自行调度**，释放时间片给其他内部线程。

常规的线程一旦得到 cpu 时间片，会毫不犹豫执行，哪怕处于 sleep 状态也会占用资源。而协程则不会，其会把 cpu 主动出让 (给自己其他线程)，等到别人" 呼唤" 自己时才会真正执行（比如 next（自己）,gevent.sleep 时间到了也算唤醒）。目前对协程的体会并不深，基本上就是循环改 yield，然后外层通过 next(send) 触发不断的类似时序的执行 (本博客前面有写过 yield 的专题, 自行翻阅)。复杂协程也未写过，所以不做过多描述了，免得误人子弟。

关键词:yield,gevent

10.6 事件

优点：如果可以不同事件自动并行化 (不确定 python 是否已实现)，基本上生产者消费者所有优点都具备，并且，额外还有容易追溯，调试的好处.

缺点：程序架构需调整。改动最大.

实现：signal(python,django)

这个严格来说，并不属于并行范畴，但是将其放到这里，因为其和生产者消费者有共通之处，而且协程的实现底层也是基于事件模型。

生产者消费者存在很大问题，就是难以调试以及流程难以控制，由于切分粒度很细，并且不同步骤之间几乎独立，虽然可以保证整体的顺序执行以及最终任务可完成，但其上下游关系难以追溯，一旦出错也难以复现。所以个人很排斥生产者消费者这种方式。

而事件则不同，生产完成后将信息注册到事件链中，不但可以保存任务结束的 result, 还可以传递任务本身初始参数信息。每个事件都可以看做独立函数，即使某一个出错，也可以将事件参数作为 debug 锚点进行追踪。

django 的事件模型用信号实现的，尚不确定是否是并发的。理论上来说，同类型事件应该可以串行，非同类事件并行，是比较稳妥的处理方式。但保不齐 Python 事件共享了同一事件通道。同时只执行一个事件，那样的话效率就未见得高了。

用事件的方式实现生产消费和协程效率应该类似, 如果事件引擎支持多进程的话 (不考虑 GIL, 多线程也行), 那么效率会更高，毕竟协程只是单线程的。

CHAPTER 11

python 进阶 11 并发之七多种并发方式的效率测试

测试 map,apply_async,gevent 协程爬虫

测试代码: 网页爬虫

11.1 函数代码

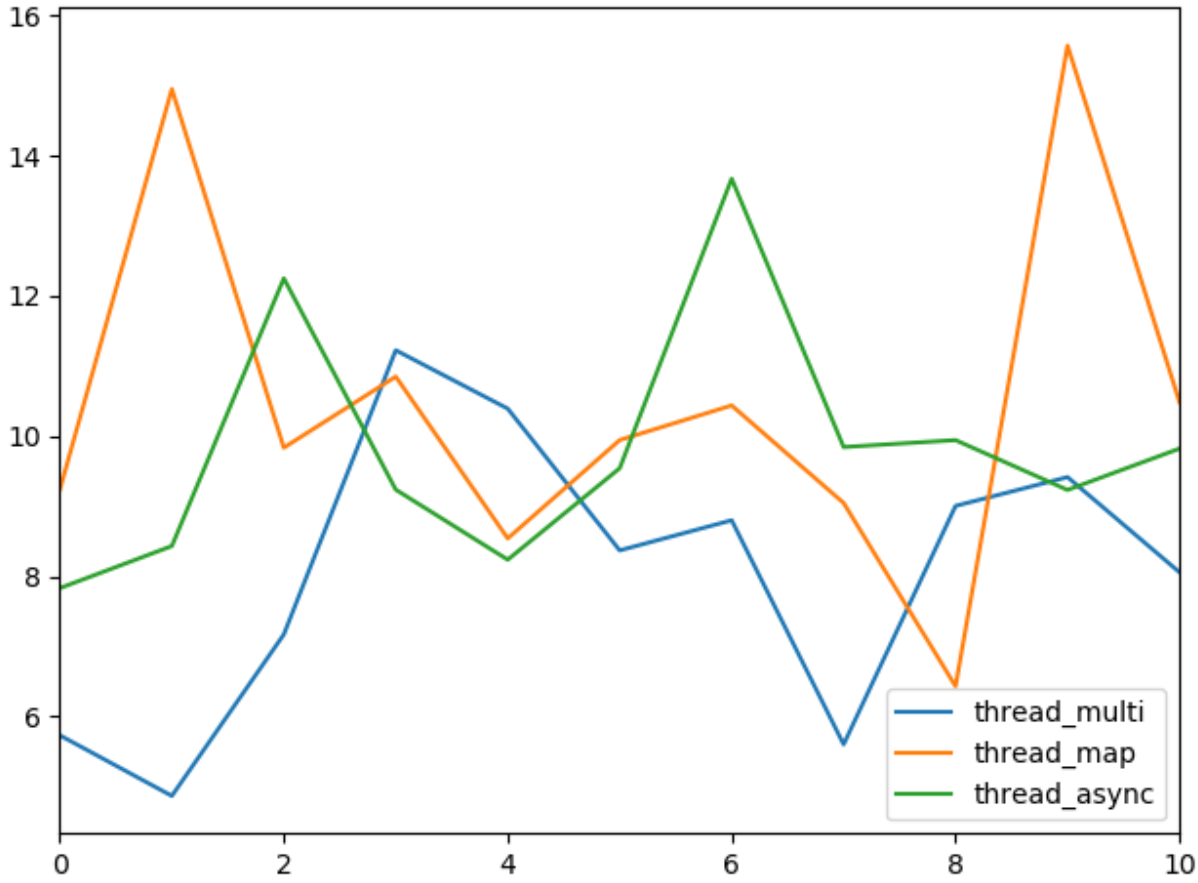
```
def thread_multi():
    threads = list()
    for url in urls:
        threads.append(threading.Thread(target=process, args=(url,)))
    [t.start() for t in threads]
    [t.join() for t in threads]


def thread_map():
    pool = ThreadPool(max(1, cpu_count() - 1))
    results = pool.map(process, urls)
    pool.close()
    pool.join()
    print(results)
```

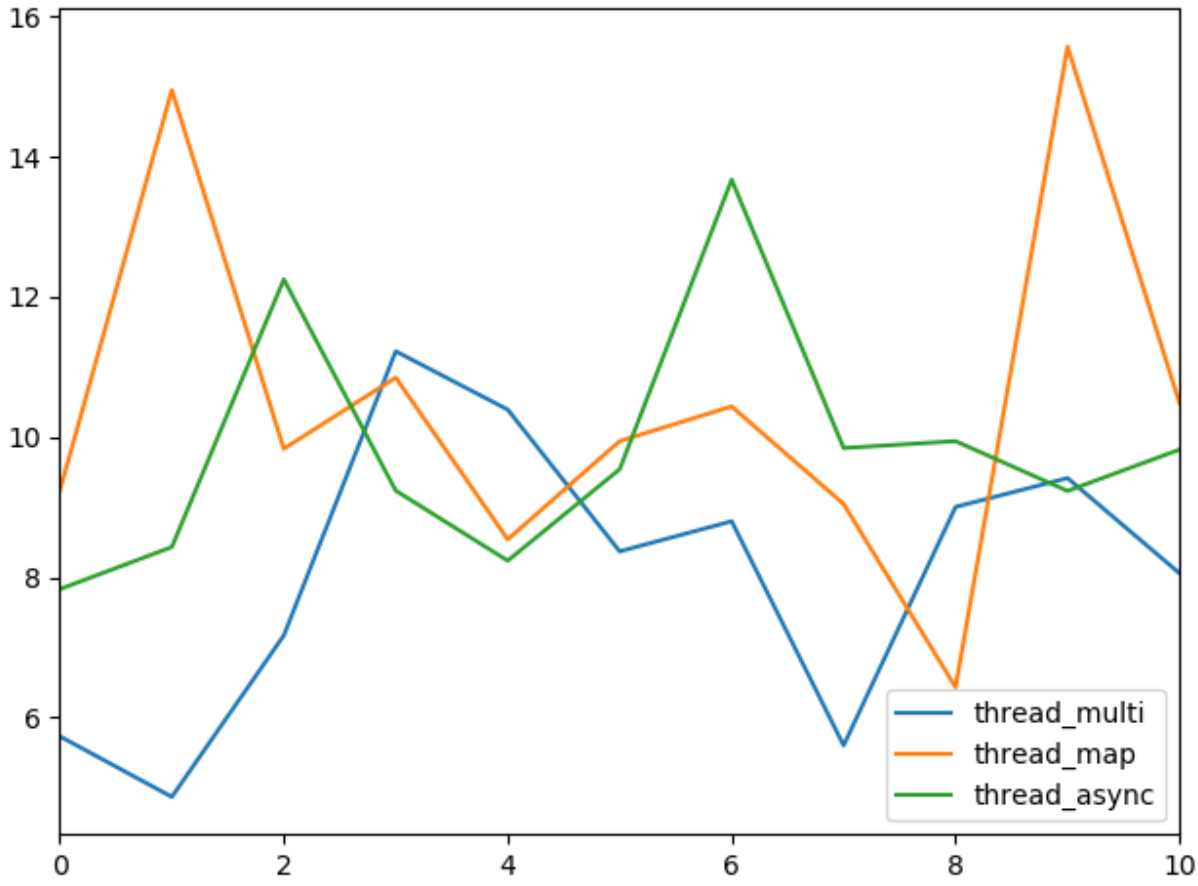
(下页继续)

(续上页)

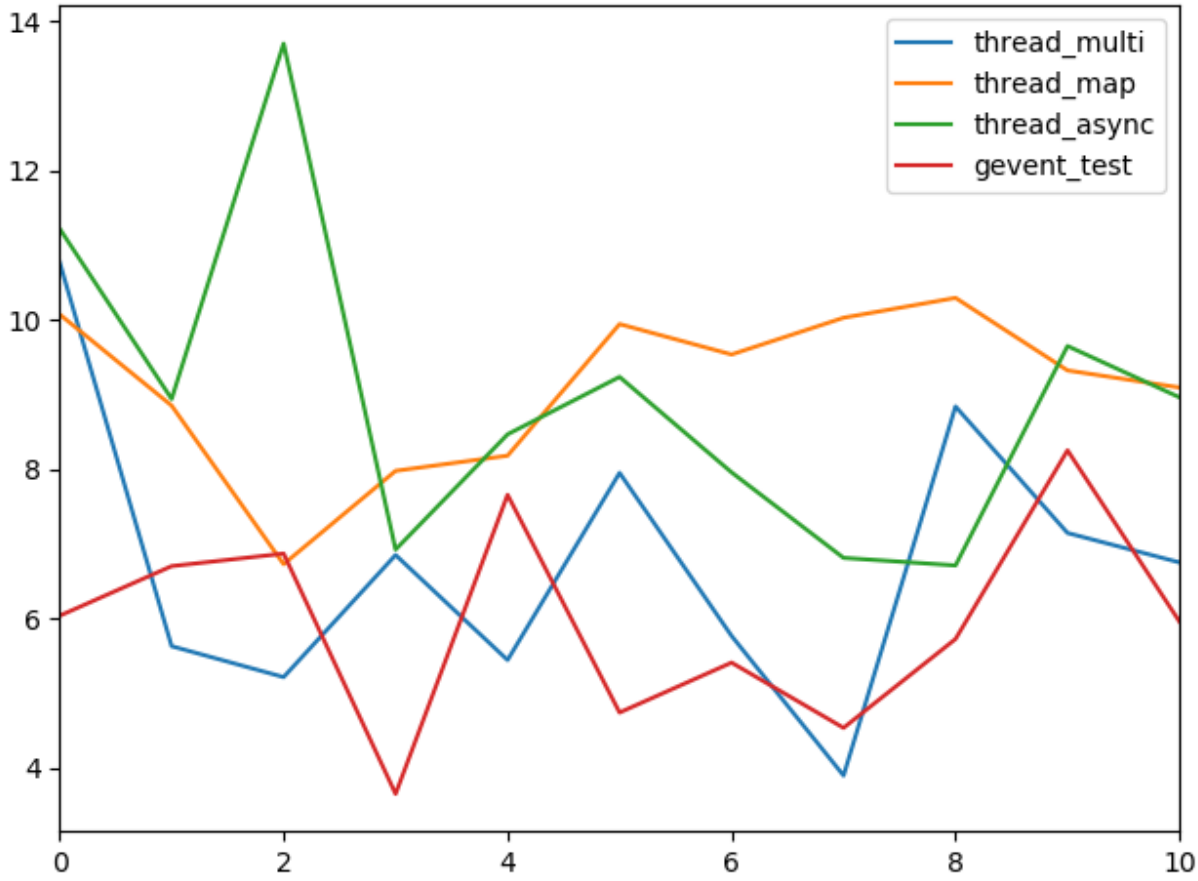
```
def thread_async():
    pool = ThreadPool(max(1, cpu_count() - 1))
    results = list()
    for url in urls:
        results.append(pool.apply_async(process, args=(url,)))
    pool.close()
    pool.join()
    print([result.get() for result in results])


def process_multi():
    processes = list()
    for url in urls:
        processes.append(Process(target=process, args=(url,)))
    [t.start() for t in processes]
    [t.join() for t in processes]


def process_map():
    pool = Pool(processes=max(1, cpu_count() - 1))
    results = pool.map(process, urls)
    pool.close()
    pool.join()
    print(results)


def process_async():
    pool = Pool(processes=max(1, cpu_count() - 1))
    results = list()
    for url in urls:
        results.append(pool.apply_async(process, (url,)))
    pool.close()
    pool.join()
    print([result.get() for result in results])
```

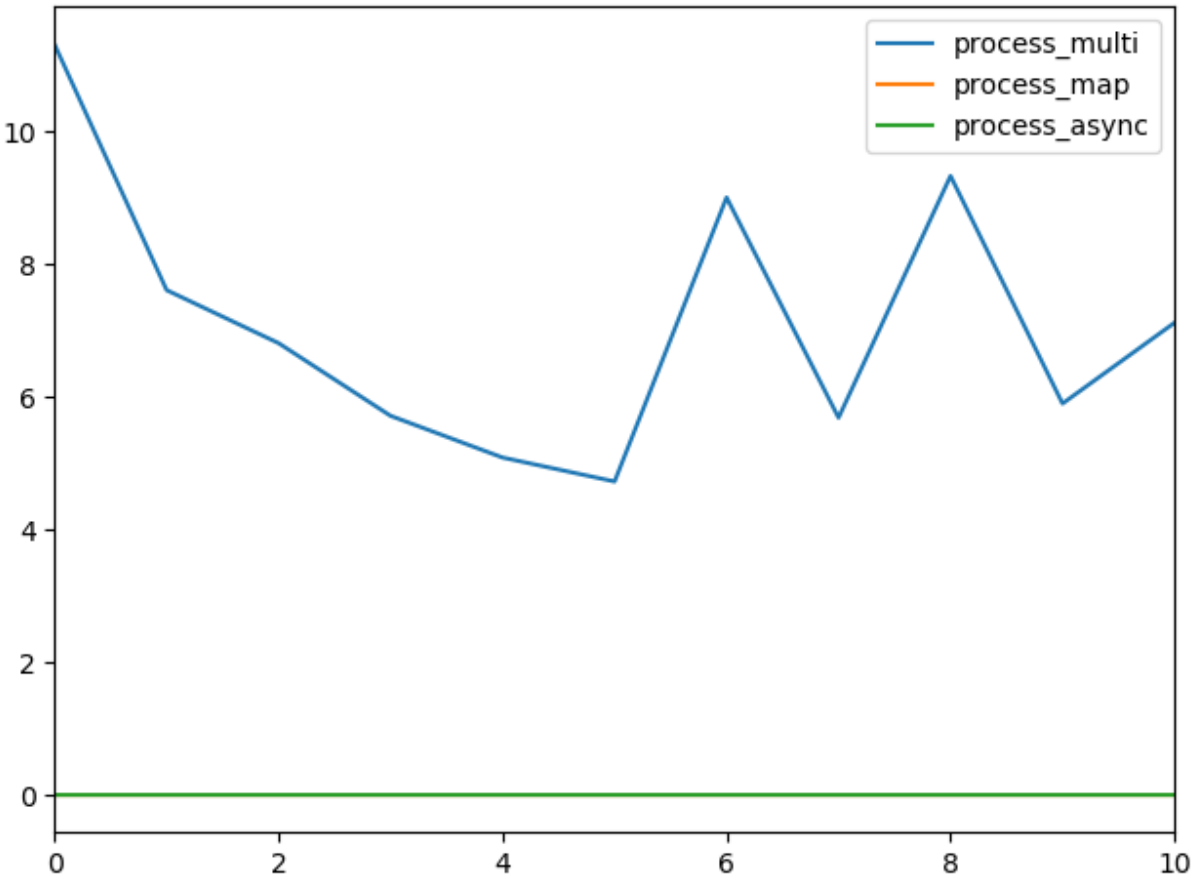
11.2 测试结果 concurrentOpt




16
14
12
10
8
6
0
2
4
6
8
10
thread_multi
thread_map
thread_async

11.3 测试结果 concurrentOptGevent





11.4 总结

结论:

01，启用 gevent 后，除了卡住的，线程和进程均加快 1s 左右时间

02，协程在线程程序中是最快的

03，多线程程序下载速度弱优于多进程

04，不论是进程还是线程，使用 thread_async 都快于 map

05，不考虑协程时，多线程较线程池速度更快，多进程较进程池速度更快，这一点不大符合理论，个人感觉和 url 数量少有关.

至于进程池在启用 gevent 后卡住的问题，网上也没查到相关的靠谱资料，哪位大牛晓得的话，求解释 ~

测试代码:github的 concurrentOpt.py 和 concurrentOptGevent.py

CHAPTER 12

python 进阶 12 并发之八多线程与数据同步

python 并发**首选进程**，但偶尔有场景进程无法搞定，比如有些**变量是无法序列化的**，就无法使用工具包 **manager() 的工具类进行共享**。如果自己实现新的共享方法，可能开发量较大，且质量难以保证。此时可考虑用线程处理，规避进程的变量共享难题，而且实际场景中,**IO 大概率都是瓶颈**，所以使用线程其实也的确有些优势。个人而言，选择进程和线程较为重视的**安全性**，进程数据隔离较好，互不干扰。其次就是**公用数据占比**，如果大多数数据都需公用，那么线程也会比进程更佳，避免了进程较多的数据共享问题。

线程而言，难点数据一致性，

12.1 哪些共享，哪些不共享

使用线程，大概率出现的情况，本以为没共享，实际共享了。由于 (以为) 没共享，所以没做同步处理，导致最后数据一团糟。

参考代码:

```
# coding=utf-8
############## 共享变量均未加锁，仅用来演示共享问题，未考虑同步问题 ############
############## 线程的变量共享  ##############
import threading
import time

gnum = 1
```

(下页继续)

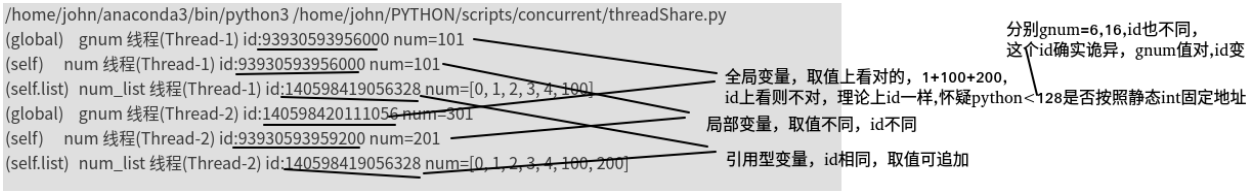
(续上页)

```
class MyThread(threading.Thread):
    # 重写 构造方法
    def __init__(self, num, num_list, sleepTime):
        threading.Thread.__init__(self)
        self.num = num
        self.sleepTime = sleepTime
        self.num_list = num_list

    def run(self):
        time.sleep(self.sleepTime)
        global gnum
        gnum += self.num
        self.num_list.append(self.num)
        self.num += 1
        print('(global)\tgnum 线程 (%s) id:%s num=%d' % (self.name, id(gnum), gnum))
        print('(self)\t\tnum 线程 (%s) id:%s num=%d' % (self.name, id(self.num), self.
↪num))
        print('(self.list)\tnum_list 线程 (%s) id:%s num=%s' % (self.name, id(self.num_
↪list), self.num_list))


if __name__ == '__main__':
    mutex = threading.Lock()
    num_list = list(range(5))
    t1 = MyThread(100, num_list, 1)
    t1.start()
    t2 = MyThread(200, num_list, 5)
    t2.start()
```

执行结果:

```
/home/john/anaconda3/bin/python3 /home/john/PYTHON/scripts/concurrent/threadShare.py
(global)        gnum 线程 (Thread-1) id:93930593956000 num=101
(self)              num 线程 (Thread-1) id:93930593956000 num=101
(self.list)       num_list 线程 (Thread-1) id:140598419056328 num=[0, 1, 2, 3, 4, 100]
(global)        gnum 线程 (Thread-2) id:140598420111056 num=301
(self)              num 线程 (Thread-2) id:93930593959200 num=201
(self.list)       num_list 线程 (Thread-2) id:140598419056328 num=[0, 1, 2, 3, 4, 100,␣
↪200]
```

结果解析:



12.2 共享数据的同步 (参考博文:python 进阶 06 并发之二技术点关键词)

最简单做法，凡是会在多个线程中修改的共享对象 (变量)，都加锁。这样可能会有部分锁多加了，但绝对好过不加，毕竟多加锁无非导致效率低下 (也可能导致死锁)，而一旦该加的没有加，则会导致数据错误，二者孰轻孰重很清楚。建议多了解下”**原子操作**”，如果不熟悉，可以按照**先加锁，再删锁**的思路，将原子操作的锁删掉即可（业务逻辑开发阶段，哪些会在多个线程被修改，是很难想全面的。所以一般是先开发，实现业务逻辑思路，再找共享变量，尽可能缩小临界区间，最后再上锁)。这样一方面保险，另一方面也避免了过多锁带来的低效问题。

12.3 thread 完整版和简单版的关系

```
class Thread:
    def __init__(self, group=None, target=None, name=None,
                 args=(), kwargs=None, *, daemon=None):
        if kwargs is None:
            kwargs = {}
        self._target = target
        self._name = str(name or _newname())
        self._args = args
        self._kwargs = kwargs

    def run(self):
        try:
            if self._target:
                self._target(*self._args, **self._kwargs)
        finally:
            del self._target, self._args, self._kwargs
```

12.4 线程本身就有局部变量，为何还需要 ThreadLocal？

ThreadLocal 例子

```
import threading

# 创建全局 ThreadLocal 对象:
local_school = threading.local()

def process_student():
  print 'Hello, %s (in %s)' % (local_school.student, threading.current_thread().name)

def process_thread(name):
  # 绑定 ThreadLocal 的 student:
  local_school.student = name
  process_student()

t1 = threading.Thread(target= process_thread, args=('Alice',), name='Thread-A')
t2 = threading.Thread(target= process_thread, args=('Bob',), name='Thread-B')
t1.start()
t2.start()
t1.join()
t2.join()
```

网上没有查询到有效资料，说明个人理解吧，

首先，如果你的线程采用了完整模式书写 (定义 class 继承 thread)，则的确不需要使用 ThreadLocal，其 ___init___ 内完全可以定义对象自身的私有变量 (list 等引用型入参，可通过 deepcopy 复制出私有的一份).

如果你想采用简洁模式, threading.Thread(target= process_thread, args=xx), 那么其实是没有定义私有变量的地方的 (也不是完全没有，如果是 int,str 等，本来就是形参，如果是 list() 则会共享)

举例:

```
############## 线程的变量共享 (short mode)   ##############
gnum = 1


def process(num, num_list, sleepTime):
    time.sleep(sleepTime)
    global gnum
    gnum += num
    num_list.append(num)
    num += 1
    print('(global)\tgnum 线程 (%s) id:%s num=%d' % (threading.currentThread().name,␣
↪id(gnum), gnum))
    print('(self)\t\tnum 线程 (%s) id:%s num=%d' % (threading.currentThread().name,␣
↪id(num), num))
```

(下页继续)

(续上页)

```
    print('(self.list)\tnum_list 线程 (%s) id:%s num=%s' % (threading.currentThread().
↪name, id(num_list), num_list))


if __name__ == '__main__':
    mutex = threading.Lock()
    num_list = list(range(5))
    t1 = threading.Thread(target=process, args=(100, num_list, 1,))
    t1.start()
    t2 = threading.Thread(target=process, args=(200, num_list, 5,))
    t2.start()
```

结果:(和前面相同)

```
(global)        gnum 线程 (Thread-1) id:94051294298272 num=101
(self)               num 线程 (Thread-1) id:94051294298272 num=101
(self.list)        num_list 线程 (Thread-1) id:140412783240456 num=[0, 1, 2, 3, 4, 100]
(global)        gnum 线程 (Thread-2) id:140412784295536 num=301
(self)               num 线程 (Thread-2) id:94051294301472 num=201
(self.list)        num_list 线程 (Thread-2) id:140412783240456 num=[0, 1, 2, 3, 4, 100,␣
↪200]
```

可见，对于单个函数的线程，其实没必要使用 threadLocal

那么那种情况需要使用呢?

```
global_dict = {}

def std_thread(name):
    std = Student(name)
    # 把 std 放到全局变量 global_dict 中:
    global_dict[threading.current_thread()] = std
    do_task_1()
    do_task_2()

def do_task_1():
    # 不传入 std, 而是根据当前线程查找:
    std = global_dict[threading.current_thread()]
    ...

def do_task_2():
```

(下页继续)

(续上页)

```
    # 任何函数都可以查找出当前线程的 std 变量:
    std = global_dict[threading.current_thread()]
    ...
```

对于存在调用子函数，且函数之间存在参数传递的情况才需要使用 threadLocal

同时，如果本身 thread 使用的就是完整模式的 thread 了，那么由于本身的 self.xx 已经是局部变量了，所以也不需要使用 threadLocal 进行中转保存.

综上所述，其实 threadLocal 的使用场景是比较有限的，必须是 thread 简洁模式下，存在函数调用和传参的情况下在有必要使用。

12.5 类锁还是实例锁?

由于锁和临界区是对应的（作为临界变量，临界区的保镖），如果临界变量（区）是类级别信息（比如统计类实例个数），就用类锁，否则就是实例锁。

12.6 阻塞式 io 中，cpu 分配时间片给阻塞线程么

运行态—wait/阻塞 io-→ 阻塞态

运行态——-调度———→ 就绪态

就绪态——-调度———→ 运行态

阻塞态—信号/io 返回-→ 就绪态

所以不占用时间片。

sleep () 和 wait() 这两个函数被调用之后线程都应该放弃执行权，不同的是 sleep () 不释放锁而 wait () 的话是释放锁。直白的意思是一个线程调用 Sleep () 之后进入了阻塞状态中的其他阻塞，它的意思就是当 sleep() 状态超时、join() 等待线程终止或者超时，线程重新转入可运行 (runnable) 状态。而 Wait () 是不同的在释放执行权之后 wait 也把锁释放了进入了线程等待阻塞，它要运行的话还是要和其他的线程去竞争锁，之后才可以获得执行权。

12.7 多线程中,target 为实例方法，可访问哪些变量的测试

举例:

get_face_result 中可以访问 self 里面的哪些资源? 在 1 处（thread 声明后，start 前）和 2 处（start 后）定义的变量可以访问么?

主要疑惑：target=get_face_result，如果 get_face_result 看作普通函数，那么由于不存在全局变量，所以所有参数都应该从 args 传入，

问题就是在 get_face_result 不是普通函数，不确定 python 是否会把 self. 里的变量传递到 get_face_result 内部（也就是 self. 里的实例变量看作 self.get_face_result 的全局变量）

结论：**以 thraed.start 为界，start 之前一定可以访问**，start 之后是否可以访问，视主线程和子线程执行速度，可能可以，可能不行

所以：**1 处的代码，子线程可以访问**，**2 处的代码**，由子线程执行速度（访问 2 处赋值的变量的时间），和父线程**执行速度决定**，如果父线程速度快，那么 2 处的（子线程）也可以访问，否则，（子线程）无法访问。

测试步骤

测试 01: 如上图

结果：

```
1
2
```

结论：start 后变量，如果有足够时间差，子线程就可以访问

测试 02：

```python
class PintTest(object):
    def __init__(self):
        self.num01 = 1
        self.thread = threading.Thread(target=self.get_face_result)
        self.thread.start()
        time.sleep(1.0)
        self.num02 = 2


    def get_face_result(self):
        print(self.num01)
        print(self.num02)


pintTest = PintTest()
```

```
Exception in thread Thread-1:
Traceback (most recent call last):
  File "/usr/lib/python3.5/threading.py", line 914, in _bootstrap_inner
    self.run()
  File "/usr/lib/python3.5/threading.py", line 862, in run
    self._target(*self._args, **self._kwargs)
  File "/home/jyuan/git/pint_app/web/object_detection/tests.py", line 19, in get_face_result
    print(self.num02)
1
AttributeError: 'PintTest' object has no attribute 'num02'


Process finished with exit code 0
```

结论：start 后变量，没有足够时间差，子线程无法访问

测试 04：引用型变量会如何

```python
class PintTest(object):
    def __init__(self):
        self.num01_list = list('a')
        self.thread = threading.Thread(target=self.get_face_result)
        time.sleep(1.0)
        self.num02_list = list('a')
        self.thread.start()



    def get_face_result(self):
        time.sleep(1.0)
        print('self.num01_list before:%s' % self.num01_list)
        print('self.num02_list before:%s' % self.num02_list)
        self.num01_list.append('b')
        self.num02_list.append('b')
        print('self.num01_list after:%s' % self.num01_list)
        print('self.num02_list after:%s' % self.num02_list)


pintTest = PintTest()
print('pintTest.num01_list:%s'%pintTest.num01_list)
print('pintTest.num02_list:%s'%pintTest.num02_list)
```

```
pintTest.num01_list:['a']
pintTest.num02_list:['a']
self.num01_list before:['a']
self.num02_list before:['a']
self.num01_list after:['a', 'b']
self.num02_list after:['a', 'b']

Process finished with exit code 0
```

```
class PintTest(object):
    def __init__(self):
        self.num01_list = list('a')
        self.thread = threading.Thread(target=self.get_face_result)
        time.sleep(1.0)
        self.num02_list = list('a')
        self.thread.start()



    def get_face_result(self):
        print('self.num01_list before:%s' % self.num01_list)
        print('self.num02_list before:%s' % self.num02_list)
        self.num01_list.append('b')
        self.num02_list.append('b')
        print('self.num01_list after:%s' % self.num01_list)
        print('self.num02_list after:%s' % self.num02_list)


pintTest = PintTest()
time.sleep(1.0)
print('pintTest.num01_list:%s'%pintTest.num01_list)
print('pintTest.num02_list:%s'%pintTest.num02_list)
```

```
self.num01_list before:['a']
self.num02_list before:['a']
self.num01_list after:['a', 'b']
self.num02_list after:['a', 'b']
pintTest.num01_list:['a', 'b']
pintTest.num02_list:['a', 'b']

Process finished with exit code 0
```

结论不变，start 后，父线程子线程先后顺序影响了变量赋值

12.8 Condition 和 Event

参考：

Python 多线程—线程同步（**Lock/RLock**、**Condition**、**queue**、**Event**）：
https://blog.csdn.net/Lesour/article/details/88808139

12.9 参考

python ThreadLocal

深入理解 Python 中的 ThreadLocal 变量（上）

Python 中 ThreadLocal 的理解与使用

在阻塞式 io 中，如果一个线程在等待 io 操作，那么 cpu 还会分配时间片给该线程吗?

Python 多线程—线程同步（**Lock/RLock、Condition、queue、Event**）：https://blog.csdn.net/Lesour/article/details/88808139

python 笔记 11-多线程之 Condition（条件变量）(**notify,wait 锁传递，细节控制**):https://www.cnblogs.com/yoyoketang/p/8337118.html

python3 是如何使用线程的（Event 与 Condition）(**在 python cookbook 中建议 event 作为一次性事件使用,Condition 可以进行多次通知，并通知不同数量的线程**):https://blog.csdn.net/qq_34392457/article/details/108319357

CHAPTER 13

python 进阶 13 并发之九多进程和数据共享

使用进程，大概率出现情况是，想当然以为共享了，实际没共享。所以最终程序大概率卡死（部分逻辑没有数据进来，导致的业务逻辑性卡住，并非程序死锁）

13.1 哪些共享，哪些不共享

默认进程是**都不共享，包括全局变量**。

父子进程其实处于**不同的资源空间**（进程是系统分配资源的最小单位），所以 2 进程其实是完全独立的资源空间，数据自然无法直接交互。如果要交互，必须**超越进程内部**，进入到操作系统层面，比如文件方式，等进行交互。

其实父子进程是一种非常**松散的关系**，在一个 app 中启动另一个 app 的关系就可以看做父子进程。只是开发过程中，大多数父子进程都是强业务关联的，所以自然会有一种**虚幻的“亲密关系”**，实际上这只是想当然的，子进程一旦启动，基本就和父进程没什么关系了，虽然不是完全没有，比如守护进程需要在父进程退出时也退出，但这个是操作系统的权利，而非父进程的权利。

所以，可以认为进程内变量都不共享!

13.2 如何实现共享 (参考博文:python 进阶 06 并发之二技术点关键词)

13.3 共享数据的同步

这个问题其实线程也存在，所以解决方法其实和线程类似的。

优点是进程数据默认独立的，凡是共享数据，其实都是非常“突出”的，容易定位，找到多个进程的共享变量，相关临界区加上锁即可 (视情况，未必一定是“加锁的方式”，或一定需要加锁，原子操作依然不需加锁)

另外需要留意以下几点:

01, 虽然使用了 manager() 封装好的 dict() 或 list() 对象，这些对象只能保证数据的共享，不能保证数据的同步，也就是说也需要自己考虑加锁问题。

02,dict=manager().dict(),dict[‘abc’]=list(‘asdf’), 这个赋值操作是进程安全的，而 dict[‘abc’][2]=’ f’ , 就未必了，应当避免使用后者的赋值方式。(不要尝试绕过 manager() 的操作，最好采用最为稳妥的赋值方式)

03, 创建进程时，参数必须能够被 pickle，所以有些自定义的类对象实例是不能被作为参数的。和 threading 不同，multiprocessing Process 参数必须能够被 pickle 进行序列化

13.4 使用了 multiprocessing.Value 为何还需要加锁

本质上而言，**共享和锁没有必然关系!**，只是**不共享不需加锁**，**共享协程不加锁**，**共享普通进程（线程）该加锁还是要加锁**

在 multiprocessing 库中的 Value 是细粒度的，Value 中有一个 ctypes 类型的对象，拥有一个 value 属性来表征内存中实际的对象。Value 可以保证同时只有一个单独的线程或进程在读或者写 value 值。这么看起来没有什么问题。

然而在第一个进程加载 value 值的时候，程序却不能阻止第二个进程加载旧的值。两个进程都会把 value 拷贝到自己的私有内存然后进行处理，并写回到共享值里。

共享变量虽然**做了一定锁封装**，但是其**锁粒度都是非常细**的，而程序中**涉及临界区可能比较大**，**仅依靠共享变量自身锁是无法限制住**的。

13.5 使用类变量可以实现跨进程共享?

否。类变量仅能实现同进程内，跨实例共享! (仔细想想很清楚，但开发过程中容易忽略，想当然以为是全局的，实际只是进程内部的全局，而非 web 应用层面的全局)

13.6 什么可以被 pickle

参见what is pickable。

```
None, True, False, 内建数字类型, 字符串
picklable 对象组成的 tuple、list、set、dict
在模块顶层中定义的函数、内建函数、类
其 __dict__ 或 __getstate__() 是可 pickle 的类实例
```

(下页继续)

(续上页)

```
特别地，numpy ndarray
注意 lambda 匿名函数不可被 pickle，除非使用 dill 之类的包，见此。
详见 http://luly.lamost.org/blog/python_multiprocessing.html
```

13.7 参考

Python 多进程相关的坑

Python 分布式进程中你会遇到的坑

Python 多进程开发中使用 Manager 进行数据共享的陷阱

python 多进程 Pool 使用遇到的坑

python multiprocessing 多进程变量共享与加锁

python 多进程共享对象 (bug 日记)

CHAPTER 14

python 进阶 14 变量作用域 LEGB

14.1 作用域

“作用域”定义了 Python 在哪一个层次上查找某个“变量名”对应的对象。接下来的问题就是：“Python 在查找‘名称-对象’映射时，是按照什么顺序对命名空间的不同层次进行查找的?”

答案就是：使用的是 LEGB 规则，表示的是 Local -> Enclosed -> Global -> Built-in，其中的箭头方向表示的是搜索顺序。

```
L: 先在局部变量中找，如果找不到
E: 则去闭包变量中找，如果找不到
G: 则去局部变量中找，如果找不到
B: 去内置变量中找，如果找不到，才报错 dir(__builtins__)
```

其中

```
Local 可能是在一个函数或者类方法内部。
Enclosed 可能是嵌套函数内，比如说 一个函数包裹在另一个函数内部。
Global 代表的是执行脚本自身的最高层次。
Built-in 是 Python 为自身保留的特殊名称。
```

python 作用域是以函数、类、模块来区分的，而不是块

也就是说 if、while，for 并不会影响变量的作用域!!! ,python 中没有块作用域。

这就能解释 python 的 if **name**==’**mian**’中声明的变量同样是全局变量

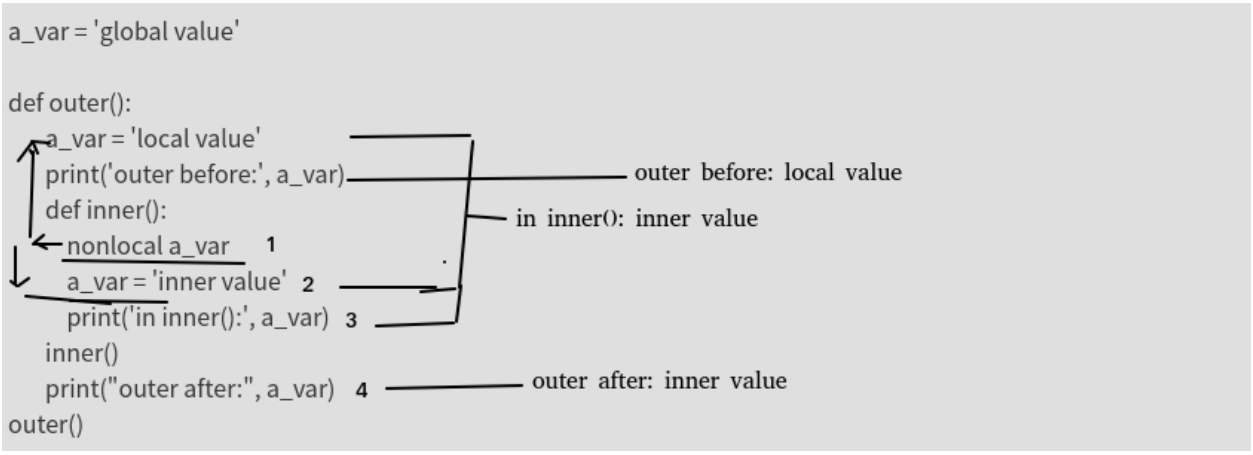
14.2 练习 01

```
a_var = 'global value'

def outer():
        a_var = 'local value'
        print('outer before:', a_var)
        def inner():
            nonlocal a_var
            a_var = 'inner value'
            print('in inner():', a_var)
        inner()
        print("outer after:", a_var)
outer()
```

结果:

```
outer before: local value
in inner(): inner value
outer after: inner value
```

分析:



14.3 练习 02

```
a = 'global'

def outer():
```

(下页继续)

(续上页)

```
    def len(in_var):
        print('called my len() function: ', end="")
        l = 0
        for i in in_var:
            l += 1
        return l

    a = 'local'

    def inner():
        global len
        nonlocal a
        a += ' variable'
    inner()
    print('a is', a)
    print(len(a))


outer()

print(len(a))
print('a is', a)
```

结果:

```
a is local variable
called my len() function: 14
6
a is global
```

可自行分析试试

14.4 注意点

01：在**函数作用域内修改全局变量通常是个坏主意**，因为这经常造成混乱或者很难调试的奇怪错误。如果你想要通过一个函数来修改一个全局变量，建议把它**作为一个变量传入，然后重新指定返回值**。

02：如果我们提前在全局命名空间中明确定义了 for 循环变量，也是同样的结果！在这种情况下，它会重新绑定已有的变量：**For 循环变量“泄漏”到全局命名空间**

```
b = 1
for b in range(5):
    if b == 4:
        print(b, '-> b in for-loop')
print(b, '-> b in global')

结果:
4 -> b in for-loop
4 -> b in global
```

在 Python 3.x 中，我们可以使用闭包来防止 for 循环变量进入全局命名空间。下面是一个例子（在 Python 3.4 中执行）:

```
i = 1
print([i for i in range(5)])
print(i, '-> i in global')

结果
[0, 1, 2, 3, 4]
1 -> i in global
```

为何 for 里面会有如此奇怪的规则？闭包本身具有独立作用域，所以这里的 i 对父域不会形成干扰。

还有另一个副作用就是

```
for i in range(5):
    print(i)
    i = 10
```

结果:

```
0
1
2
3
4
```

而不是直观理解的执行一次就退出

代码;

```
for i in range(5):
    i += 5
```

(下页继续)

(续上页)

```
    print(i)
print(i)
```

结果:

```
5
6
7
8
9
9
```

第一：成功污染外面的 i

第二：内部 i+5 只进行了 1 次，说明 i=i+5, 右侧的 i, 是真正的 for 里面的 i, 左侧的 i 是外部的 i，但是却未报错 unbounderror 的错误！(内部的 i 有赋值，所以理论上外部的 i 应该是被屏蔽的，应该报错 unbound 才对，但是没报。即使勉强接受这一点，最终外面的 i=9 而非 4，也很奇怪)

原因:**for 循环不会引入新的作用域**，所以，循环结束后，继续执行 print(i)，可以正常输出 i，原理上与情况 3 中的 if 相似。这一点 Python 就比较坑了，因此写代码时切忌 for 循环名字要与其他名字不重名才行。

上式中,for 里面 i+5, 到外面的 for 那里又重新赋值为原有的 i(无视了内部对 i 的修改)，所以**每次都 +5 了**，而最终结果依然 +5，是由于**最后一次的 i 并未被成功赋值**，所以最终结果看起来比较奇.

```
list_1 = [i for i in range(5)]
print(i)

结果:
NameError: name 'i' is not defined
```

情况 3 中说到过，for 循环不会引入新的作用域，那么为什么输出报错呢？真相只有一个：列表生成式会引入新的作用域，for 循环是在 Local 作用域里面的。事实上，lambda、生成器表达式、列表解析式也是函数，都会引入新作用域。

14.5 参考

Python 中的 LEGB 规则：https://www.cnblogs.com/GuoYaxiang/p/6405814.html

Python 中命名空间与作用域使用总结：https://www.cnblogs.com/chenhuabin/p/10123009.html

一道题看 Python 的 LEGB 规则:https://www.ucloud.cn/yun/45499.html

Python LEGB 规则:https://www.jianshu.com/p/3b72ba5a209c

python 中的 LEGB 规则:https://blog.csdn.net/xun527/article/details/76795328

CHAPTER 15

python 进阶 15 多继承与 Mixin

15.1 Mixin 解释

为了让大家，对这个 Mixin 有一个更直观的理解，摘录了网上一段说明。

民航飞机是一种交通工具，对于土豪们来说直升机也是一种交通工具。对于这两种交通工具，它们都有一个功能是飞行，但是轿车没有。所以，我们不可能将飞行功能写在交通工具这个父类中。但是如果民航飞机和直升机都各自写自己的飞行方法，又违背了代码尽可能重用的原则（如果以后飞行工具越来越多，那会出现许多重复代码）。

怎么办，那就只好让这两种飞机同时继承交通工具以及飞行器两个父类，这样就出现了多重继承。这时又违背了继承必须是”is-a”关系。这个难题该怎么破?

这时候 Mixin 就闪亮登场了。飞行只是飞机做为交通工具的一种（增强）属性，我们可以为这个飞行的功能单独定义一个（增强）类，称之为 Mixin 类。这个类，是做为增强功能，添加到子类中的。为了让其他开发者，一看就知道这是个 Mixin 类，一般都要求开发者遵循规范，在类名末尾加上 Mixin 。

举个例子

```
class Vehicle(object):
    pass

class PlaneMixin(object):
    def fly(self):
        print('I am flying')
```

(下页继续)

(续上页)

```
class Airplane(Vehicle, PlaneMixin):
    pass
```

使用 Mixin 类实现多重继承要遵循以下几个规范

责任明确：必须表示某一种功能，而不是某个物品；

功能单一：若有多个功能，那就写多个 Mixin 类；

绝对独立：不能依赖于子类的实现；子类即便没有继承这个 Mixin 类，也照样可以工作，就是缺少了某个功能。

15.2 一个 Mixin 类的实例 (这个例子并不符合前面的无依赖原则)

这里，我直接先上代码，有兴趣的同学，可以暂停到这里，看看这段代码中的 subclass.display() 这行代码，究竟是怎么执行的：

```
class Displayer():
    def display(self, message):
        print(message)


class LoggerMixin():
    def log(self, message, filename='logfile.txt'):
        with open(filename, 'a') as fh:
            fh.write(message)

    def display(self, message):
        super().display(message)
        self.log(message)


class MySubClass(LoggerMixin, Displayer):
    def log(self, message):
        super().log(message, filename='subclasslog.txt')


subclass = MySubClass()
subclass.display("This string will be shown and logged in subclasslog.txt")
```

代码不多，也就拢共 22 行，22 行的代码里面，定义了 3 个类。其中 MySubClass 多继承了 LoggerMixin 类

和 Displayer 类。看似并没有什么异常的代码里面，当你尝试去仔细推敲 subclass.display() 的调用逻辑之后，就变得异常的复杂。

问题: 我们的 LoggerMixin 类是怎么调用的 super().display() 方法的呢?

解答: 在多继承的环境下，super() 有相对来说更加复杂的含义。它会查看你的继承链，使用一种叫做 Methods Resolution Order（方法解析顺序）的方式，来决定调用最近的继承父类的方法。

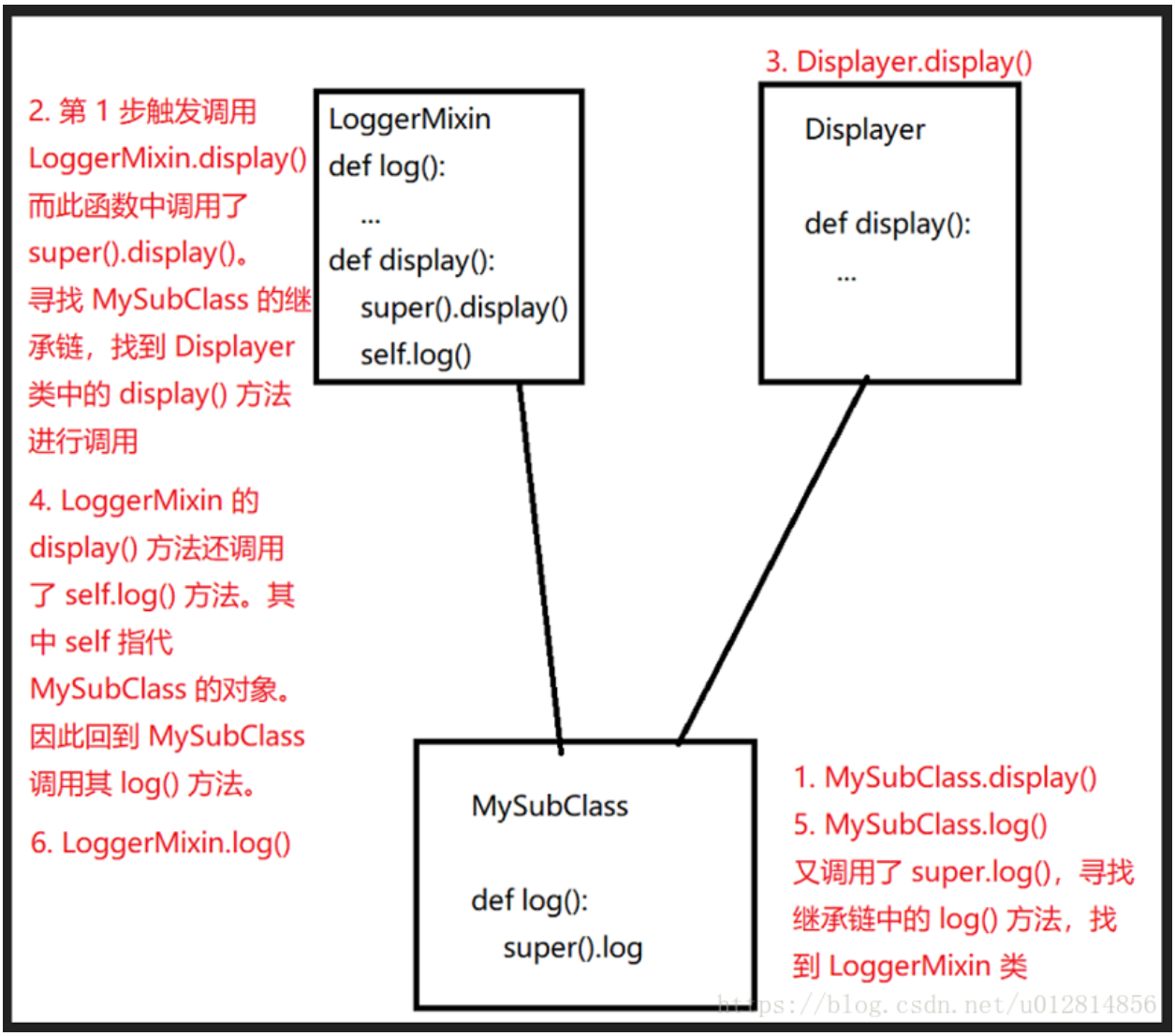
也就是说，我们的 MySubClass.display() 调用，触发了是这么一系列的行为:

```
1. MySubClass.display() is resolved to LoggerMixin.display().
MySubClass.display() 方法被解析为 LoggerMixin.display() 方法的调用。这应该还是比较好理解的。
因为对于 MySubClass 类来说，在继承链上的两个父类，LoggerMixin 和 Displayer 来说，
LoggerMixin 是最近的，因此调用它的 display() 方法。

2. LoggerMixin.display() calls super().display(), which is resolved to Displayer.
↪display().
LoggerMixin.display() 方法调用了 super().display()，这一行代码按照我们刚才的解释，查看␣
↪MySubClass 的继承链，是应该调用 Displayer 类的 display() 方法的。这一步是相对来说比较难以
理解的。

让我们这么来理解它，当 LoggerMixin.display() 中调用了 super().display() 的时候，它会尝试去寻
找属于当前类的继承链。而这个当前类是什么类呢？不是 LoggerMixin 类，而是 MySubClass 类。
MySubClass 类的继承连是 LoggerMixin，然后 Displayer。所以，我们就找到了 Displayer 的␣
↪display() 方法。

3. It alse calls self.log(). Since self, in this case, is a MySubClass instance, it␣
↪resolves to MySubClass.log(). MySubClass.log() calls super().log(), which is resolved␣
↪back to LoggerMixin.log().
别忘了，我们的 LoggerMixin 类还调用了 self.log() 方法。这个看似好像要直接调用 LoggerMixin␣
↪的 log 方法，其实不然。

LoggerMixin 的 display() 方法在当前语境中的 self，其实是 MySubClass 类的对象，因此对于␣
↪MySubClass 类的对象，想要调用 log 方法，是直接调用自己类中的 log 方法，也就是 MySubClass.
↪log() 方法，而不是 LoggerMixin.log() 方法的。

而又因为 MySubClass.log() 方法调用了 super().log() 方法，这才根据继承链寻找最近的父类，才找
到了 LoggerMixin 类中的 log() 方法进行调用。
```

15.3 画图总结

为了方便大家理解，我画了一个简略的图方便理解:

一句简单的 subclass.display() 的背后，究竟发生了什么?



15.4 参考

一个例子走近 Python 的 Mixin 类：利用 Python 多继承的魔力：https://blog.csdn.net/u012814856/article/details/81355935

1.9 多继承与 Mixin 设计模式：python.iswbm.com/en/latest/c01/c01_09.html

CHAPTER 16

python 进阶 16 炫技巧

原则：**可读性第一**（效率固然重要，除非非常明显的效率差异，否则可读性优先）

学习炫技巧，更多为了读懂他人代码，自己开发过程中，相似代码量（可读性），建议使用通俗写法。反对为炫而炫。

16.1 可直接运行的 zip 包

有 Python 包，居然可以以 zip 包进行发布，并且可以不用解压直接使用。

这个 zip 是如何制作的呢，请看下面的示例。

```
[root@localhost ~]# ls -l demo
total 8
-rw-r--r-- 1 root root 30 May  8 19:27 calc.py
-rw-r--r-- 1 root root 35 May  8 19:33 __main__.py

[root@localhost ~]# cat demo/__main__.py
import calc
print(calc.add(2, 3))
[root@localhost ~]# cat demo/calc.py
def add(x, y):
    return x+y
[root@localhost ~]# python -m zipfile -c demo.zip demo/*
```

制作完成后，我们可以执行用 python 去执行它

```
[root@localhost ~]# python demo.zip
5
```

16.2 懒人必备技能：使用“_”

大家对于他的印象都是用于占位符，省得为一个不需要用到的变量，绞尽脑汁的想变量名。

今天要介绍的是他的第二种用法，就是在 console 模式下的应用。

示例如下：

```
>>> 3 + 4
7
>>> _
7
>>> name='公众号: Python 编程时光'
>>> name
' 公众号: Python 编程时光'
>>> _
' 公众号: Python 编程时光'
```

它可以返回上一次的运行结果。

但是，如果是 print 函数打印出来的就不行了。

```
>>> 3 + 4
7
>>> _
7
>>> print("公众号: Python 编程时光")
ming
>>> _
7
```

16.3 最快查看包搜索路径的方式

```
python3 -m site
sys.path = [
    '/home/wangbm',
```

(下页继续)

(续上页)

```
    '/usr/local/Python3.7/lib/python37.zip',
    '/usr/local/Python3.7/lib/python3.7',
    '/usr/local/Python3.7/lib/python3.7/lib-dynload',
    '/home/wangbm/.local/lib/python3.7/site-packages',
    '/usr/local/Python3.7/lib/python3.7/site-packages',
]
USER_BASE: '/home/wangbm/.local' (exists)
USER_SITE: '/home/wangbm/.local/lib/python3.7/site-packages' (exists)
ENABLE_USER_SITE: True
```

16.4 and 和 or 的取值顺序

当一个 or 表达式中所有值都为真，Python 会选择第一个值

当一个 and 表达式所有值都为真，Python 会选择第二个值。

示例如下：

```
>>>(2 or 3) * (5 and 7)
14  # 2*7
```

16.5 访问类中的私有方法

```
# 调用私有方法，以下两种等价
ins._Kls__private()
ins.call_private()
```

16.6 一行代码实现 FTP 服务器

python3 -m http.server 8888

16.7 for else 逻辑

for else 和 try else 相同，只要代码正常走下去不被 break，不抛出异常，就可以走 else。(所以和去掉 else，直接写后面没区别?)

优点在于，**可以识别正常退出还是 break 退出**（一般业务含义不同）

16.8 嵌套上下文管理的另类写法

with test_context(‘aaa’), test_context(‘bbb’): print(‘========== in main ==========’)

16.9 连接多个列表最极客的方式

```
>>> b = [3,4]
>>> c = [5,6]
>>>
>>> sum((a,b,c), [])
[1, 2, 3, 4, 5, 6]
```

另外几种连接列表的方式

```
>>> list01 + list02 + list03
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(chain(list01, list02, list03))
>>> [*list01, *list02]
>>> list01.extend(list02)
>>> [x for l in (list01, list02, list03) for x in l]
>>> from heapq import merge
>>> list(merge(list01, list02, list03))
sorted(itertools.chain(*iterables))
```

16.10 在程序退出前执行代码的技巧

使用 atexit 这个内置模块，可以很方便的注册退出函数。

如果 clean() 函数有参数，那么你可以不用装饰器，而是直接调用 atexit.register(clean_1, 参数 1, 参数 2, 参数 3=’ xxx’)。

16.11 合并字典的几种方法

```
profile.update(ext_info)
full_profile01 = {**profile, **ext_info}
dict(itertools.chain(profile.items(), ext_info.items()))
```

(下页继续)

(续上页)

```
dict(ChainMap(profile, ext_info))
full_profile = dict(profile.items() | ext_info.items())
```

16.12 条件语句的几种写法

```
<on_true> if <condition> else <on_false>
<condition> and <on_true> or <on_false>
(<on_true>, <on_false>)[condition]
(lambda: <on_false>, lambda:<on_true>)[<condition>]()
>>> msg2 = (lambda:"未成年", lambda:"已成年")[age2 > 18]()
>>> print(msg2)
未成年
 msg2 = {True: "已成年", False: "未成年"}[age2 > 18]
>>> print(msg2)
未成年
```

16.13 让我爱不释手的用户环境

当你在机器上并没有 root 权限时，如何安装 Python 的第三方包呢?

可以使用 pip install –user pkg 将你的包安装在你的用户环境中，该用户环境与全局环境并不冲突，并且多用户之间相互隔离，互不影响。

16.14 with 与上下文管理器

样例:

```
import contextlib

@contextlib.contextmanager
def open_func(file_name):
    # __enter__ 方法
    print('open file:', file_name, 'in __enter__')
    file_handler = open(file_name, 'r')

    try:
        yield file_handler
```

(下页继续)

(续上页)

```
    except Exception as exc:
        # deal with exception
        print('the exception was thrown')
    finally:
        print('close file:', file_name, 'in __exit__')
        file_handler.close()

        return

with open_func('/Users/MING/mytest.txt') as file_in:
    for line in file_in:
        1/0
        print(line)
```

16.15 在 linux 上看 json 文件

cat test.json | python -m json.tool

16.16 sh，最优雅的命令调用方式

```
 >>> sh.glob("/etc/*.conf")
['/etc/mke2fs.conf', '/etc/dnsmasq.conf', '/etc/asound.conf']
>>> r=sh.Command('/root/test.py')
>>> r()
hello,world
```

16.17 判断是否包含子串的七种方法

```
1. 使用 in 和 not in
2. 使用 find 方法
3. 使用 index 方法
4. 使用 count 方法
5, 借助 operator
```

operator 模块是 python 中内置的操作符函数接口，它定义了一些算术和比较内置操作的函数。operator 模块是用 c 实现的，所以执行速度比 python 代码快。

在 operator 中有一个方法 contains 可以很方便地判断子串是否在字符串中。

```
>>> import operator
>>>
>>> operator.contains("hello, python", "llo")
```

16.18 使用 json.dumps 打印字典

打印中文在 json.dumps 这里，却只要加个参数就好了

具体的代码示例如下：

```
>>> import json
>>> print json.dumps(info, indent=4, ensure_ascii=False)
```

16.19 slots

JIT 即时编译器，当虚拟机发现某个方法或代码块运行特别频繁时，就会把这些代码认定为热点代码，为了提高热点代码的运行效率，在运行时，虚拟机将会把这些代码编译成与本地平台的相关带代码，并进行各层次的优化。

正如上面所说的，默认情况下,Python 的新式类和经典类的实例都有一个 dict 来存储实例的属性。这在一般情况下还不错，而且非常灵活，

乃至在程序中可以随意设置新的属性。但是，对一些在”编译”前就知道有几个固定属性的小 class 来说，这个 dict 就有点浪费内存了。

当需要创建大量实例的时候，这个问题变得尤为突出。一种解决方法是在新式类中定义一个 ___slots___ 属性。

___slots___ 声明中包含若干实例变量，并为每个实例预留恰好足够的空间来保存每个变量；这样 Python 就不会再使用 dict，从而节省空间。

16.20 stateful service 排障

来个真的有点高（黑）级（暗）的技巧吧，开发一个 stateful service 的时候怎么排障？比如出现死锁了，或者某个线程意外退出。这个时候可以从

gc.get_objects()

掏出所有活着的对象，其中当然就包括活着的线程。

例如，打印出所有 Greenlet 的 stack：

```
import os

import gc

import greenlet

import traceback



greenlets = [

    o for o in gc.get_objects() if isinstance(o, greenlet.greenlet) if o]

stack = '\n\n'.join(

    ''.join(traceback.format_stack(o.gr_frame)) for o in greenlets)

open('/tmp/stack-%d.txt' % os.getpid(), 'w').write(stack)
```

这个方法结合 gevent.backdoor 堪称排障利器。如果愿意冒一定的风险，甚至可以使用调试器对事先没有埋过点的进程做活体检测——挂上 GIL 再打印一份线程 stack。详见 GitHub - wooparadog/pstack: Tool to dump python thread and greenlet stacks.

16.21 调试利器，显示调用栈

有时候 BUG 隐藏的太深，需要对上下文都有清晰的展示来帮助判断。用 pdb 调试不方便，用 print 不直观。可以使用如下函数获取当前调用栈：

```
import sys

def get_cur_info():

        print sys._getframe().f_code.co_filename # 当前文件名

        print sys._getframe(0).f_code.co_name # 当前函数名

        print sys._getframe(1).f_code.co_name # 调用该函数的函数的名字，如果没有被调用，则
返回 module
```

(下页继续)

(续上页)

```
        print sys._getframe().f_lineno # 当前行号
```

16.22 x 入参

```
def testa(*v):
    print(list(v))


testa(1, 2, 3)
testa(list('abc'))
testa('a', 'b', list('cde'))

[1, 2, 3]
[['a', 'b', 'c']]
['a', 'b', ['c', 'd', 'e']]
```

16.23 内存占用

下面的代码块可以检查变量 variable 所占用的内存。

```
import sys

variable = 30
print(sys.getsizeof(variable)) # 24
```

16.24 打印 N 次字符串

该代码块不需要循环语句就能打印 N 次字符串。

```
n = 2;

s ="Programming";
```

(下页继续)

(续上页)

```
print(s * n);

# ProgrammingProgramming
```

16.25 解包

如下代码段可以将打包好的成对列表解开成两组不同的元组。

```
array = [['a', 'b'], ['c', 'd'], ['e', 'f']]

transposed = zip(*array)

print(transposed)

# [('a', 'c', 'e'), ('b', 'd', 'f')]
```

16.26 链式函数调用

你可以在一行代码内调用多个函数。

```
def add(a, b):

    return a + b



def subtract(a, b):

    return a - b



a, b = 4, 5

print((subtract if a > b else add)(a, b)) # 9
```

16.27 回文序列

以下方法会检查给定的字符串是不是回文序列，它首先会把所有字母转化为小写，并移除非英文字母符号。最后，它会对比字符串与反向字符串是否相等，相等则表示为回文序列。

```
def palindrome(string):

    from re import sub

    s = sub('[\W_]', '', string.lower())

    return s == s[::-1]



palindrome('taco cat') # True
```

16.28 不使用 if-else 的计算子

这一段代码可以不使用条件语句就实现加减乘除、求幂操作，它通过字典这一数据结构实现：

```
import operator

action = {

    "+": operator.add,

    "-": operator.sub,

    "/": operator.truediv,

    "*": operator.mul,

    "**": pow

}

print(action['-'](50, 25)) # 25
```

16.29 参考

Python 黑魔法指南 50 例:python.iswbm.com/en/latest/c01/c01_10.htmlPython 炫技操作: 连接列表的八种方法: python.iswbm.com/en/latest/c01/c01_41.htmlPython 炫技操作: 判断是否包含子串的七种方法: python.iswbm.com/en/latest/c01/c01_40.htmlPython 炫技操作: 合并字典的七种方法: python.iswbm.com/en/latest/c01/c01_39.htmlPython 炫技操作: 条件语句的七种写法: python.iswbm.com/en/latest/c01/c01_37.html 每日一库: sh, 最优雅的命令调用方式: python.iswbm.com/en/latest/c01/c01_34.htmlPython 开发中有哪些高级技巧? :https://www.zhihu.com/question/23760468python **slots** 详解 (上篇) :https://blog.csdn.net/sxingming/article/details/5289264030 段极简 Python 代码: 这些小技巧你都 Get 了么:https://zhuanlan.zhihu.com/p/109016233

CHAPTER 17

python 进阶 17 正则表达式

17.1 正则基础知识

```
^`: 匹配行首
`$`: 匹配结尾
`*`: (** 贪婪 **) 前面字符匹配任意多次
`+`: (** 懒惰 **) 前面字符匹配 1 或者更多次
`?`: 前面字符匹配 0 或 1 次，还作为懒惰限定符使用，详看后面

`{m}`: 前面字符匹配 m 次
`{m,n}`: 前面字符匹配 m~n 次
`{m,}`: 前面字符匹配 m 或更多次
`{,n}`: 前面字符匹配 0~n 次

`|`: 或，必须加括号

`.`: 匹配除换行符以外的任意字符
`[1357]`: 匹配 1，3,5,7 中其中一个数字，当然也可以是字母
`[0-9]`: 匹配 0 到 9 的其中一个数字，类似用法还有：[a-zA-Z]
`[\u4E00-\u9FA5]`: 匹配中文
`[^012]`: 表示除 012 外的任意字符，包括 3-9,a-z,A-Z，等等
注意：[] 里面的. 和 * 等一些特殊字符都失去特殊意义，只表示本身。
```

语法	说明	例子	可匹配字符串
^	以什么字符串开始	^123	123abc、123321、123zxc
$	以什么字符串结尾	123$	abc123、321123、zxc123
\b	匹配单词边界，不匹配任何字符	\basd\b	asd
\d	匹配数字0-9	zx\dc	zx1c、zx2c、zx5c
\D	匹配非数字	zx\Dc	zxvc、zx$c、zx&c
\s	匹配空白符	zx\sc	zx c
\S	匹配非空白符	zx\Sc	zxac、zx1c、zxtc
\w	匹配字母、数字和下划线	zx\wc	zxdc、zx1c、zx_c

语法	说明	例子	可匹配字符串
.	匹配除了换行符 “\n” 以外的任意字符	a.b	acb、adb、a2b、a~b
\	转义，将转移字符后面的一个字符改变原来的意思	a[b\.\\]c	abc、a.c、a\c
[]	匹配括号内的任意字符	a[b,c,d,e]f	abd、acf、adf、aef

语法	说明
*?	匹配0次或多次，但要尽可能少重复
+?	匹配1次或多次，但要尽可能少重复
??	匹配0次或1次，但要尽可能少重复
{m,}?	匹配m次或多次，但要尽可能少重复

17.2 分组捕获

```
import re
str = 'booy123'
regex = '((boy|booy)123)'

# 如果有多个括号，则从最外面往里算，从 1 开始

re_match = re.match(regex, str)
re_match.group(1)
# 'booy123'
re_match.group(2)
# 'booy'
```

17.3 懒惰限定符

如果有多个贪婪，则第一个最贪婪

```
*? : 重复任意次，但尽可能少重复
+? : 重复 1 次或更多次，但尽可能少重复
?? : 重复 0 次或 1 次，但尽可能少重复
{n,m}? : 重复 n 到 m 次，但尽可能少重复
{n,}?:  重复 n 次以上，但尽可能少重复
str = 'abooabbapds abokskdap'
obj = re.compile('ab.*?ap') # 注意用非贪婪匹配，不然 list 里只有一个
```

17.4 匹配和搜索

```
match_list = obj.findall(str)  #match_list 是一个 list
# match_list -> ['abooabbap','abokskdap']

for match in match_list:
    print(match)
# 输出
# abooabbap
# abokskdap
import re
```

(下页继续)

(续上页)

```
str = 'abooabbapds abokslsldap'
obj = re.compile('ab.*?ap')
match_list = obj.finditer(str)
# match_list -> callable_iterator 对象, 需要用 group() 查询

for match in match_list:
    print(match.group())
# abooabbap
# abokslsldap
```

17.5 参考

Python 正则表达式急速入门:https://baijiahao.baidu.com/s?id=1652504385879645545&wfr=spider&for=pc

正则表达式必知必会：python.iswbm.com/en/latest/c01/c01_11.html

Python 正则表达式：https://www.runoob.com/python/python-reg-expressions.html

Python 正则表达式指南：https://www.cnblogs.com/huxi/archive/2010/07/04/1771073.html

CHAPTER 18

python 进阶 18 垃圾回收 GC

18.1 概述

python 采用的是**引用计数机制为主，标记-清除和分代收集两种机制为辅的策略**。

Python GC 主要使用**引用计数（reference counting）来跟踪和回收垃圾**。在引用计数的基础上，**通过“标记-清除”（mark and sweep）解决容器对象可能产生的循环引用问题，通过“分代回收”（generation collection）以空间换时间的方法提高垃圾回收效率**。

0 代触发将清理所有三代，1 代触发会清理 1,2 代，2 代触发后只会清理自己。

18.2 垃圾收集三大手段

18.2.1 一、引用计数（计数器）

Python 垃圾回收主要以引用计数为主，分代回收为辅。引用计数法的原理是每个对象维护一个 ob_ref，用来记录当前对象被引用的次数，也就是来追踪到底有多少引用指向了这个对象，当发生以下四种情况的时候，该对象的引用计数器 +1

```
对象被创建    a=14
对象被引用     b=a
对象被作为参数，传到函数中     func(a)
对象作为一个元素，存储在容器中     List={a,” a” ,” b” ,2}
```

与上述情况相对应，当发生以下四种情况时，该对象的引用计数器-1

当该对象的别名被显式销毁时　　del a
当该对象的引别名被赋予新的对象，　　a=26
一个对象离开它的作用域，例如 func 函数执行完毕时，函数里面的局部变量的引用计数器就会减一（但是全局变量不会）
将该元素从容器中删除时，或者容器被销毁时。

. 当指向该对象的内存的引用计数器为 0 的时候，该内存将会被 Python 虚拟机销毁

优点：

高效
运行期没有停顿 可以类比一下 Ruby 的垃圾回收机制，也就是 实时性：一旦没有引用，内存就直接释放了。不用像其他机制等到特定时机。实时性还带来一个好处：处理回收内存的时间分摊到了平时。
对象有确定的生命周期
易于实现

原始的引用计数法也有明显的缺点：

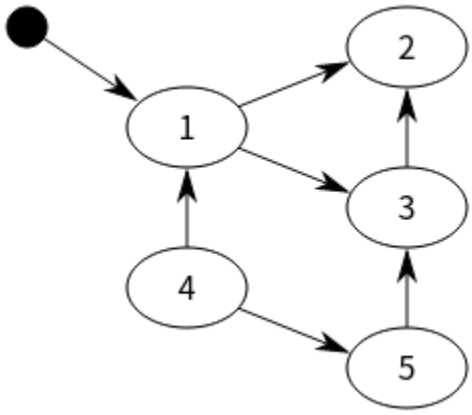
维护引用计数消耗资源，维护引用计数的次数和引用赋值成正比，而不像 mark and sweep 等基本与回收的内存数量有关。
无法解决循环引用的问题。A 和 B 相互引用而再没有外部引用 A 与 B 中的任何一个，它们的引用计数都为 1，但显然应该被回收。

18.2.2 二、标记-清除（双向链表）

『标记清除（Mark—Sweep）』算法是一种基于追踪回收（tracing GC）技术实现的垃圾回收算法。它分为两个阶段：第一阶段是标记阶段，GC 会把所有的『活动对象』打上标记，第二阶段是把那些没有标记的对象『非活动对象』进行回收。

那么 GC 又是如何判断哪些是活动对象哪些是非活动对象的呢?

对象之间通过引用（指针）连在一起，构成一个有向图，对象构成这个有向图的节点，而引用关系构成这个有向图的边。从根对象（root object）出发，沿着有向边遍历对象，可达的（reachable）对象标记为活动对象，不可达的对象就是要被清除的非活动对象。根对象就是全局变量、调用栈、寄存器。

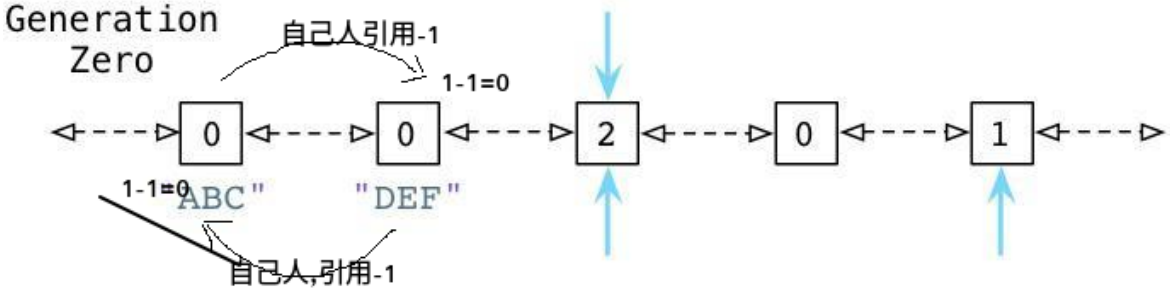


在上图中，我们把小黑圈视为全局变量，也就是把它作为 root object，从小黑圈出发，对象 1 可直达，那么它将被标记，对象 2、3 可间接到达也会被标记，而 4 和 5 不可达，那么 1、2、3 就是活动对象，4 和 5 是非活动对象会被 GC 回收。

> 标记清除算法作为 Python 的辅助垃圾收集技术主要处理的是一些容器对象，比如 list、dict、tuple，instance 等，因为对于字符串、数值对象是不可能造成循环引用问题。Python 使用一个双向链表将这些容器对象组织起来。不过，这种简单粗暴的标记清除算法也有明显的缺点：清除非活动的对象前它必须顺序扫描整个堆内存，哪怕只剩下小部分活动对象也要扫描所有对象。

检测循环引用

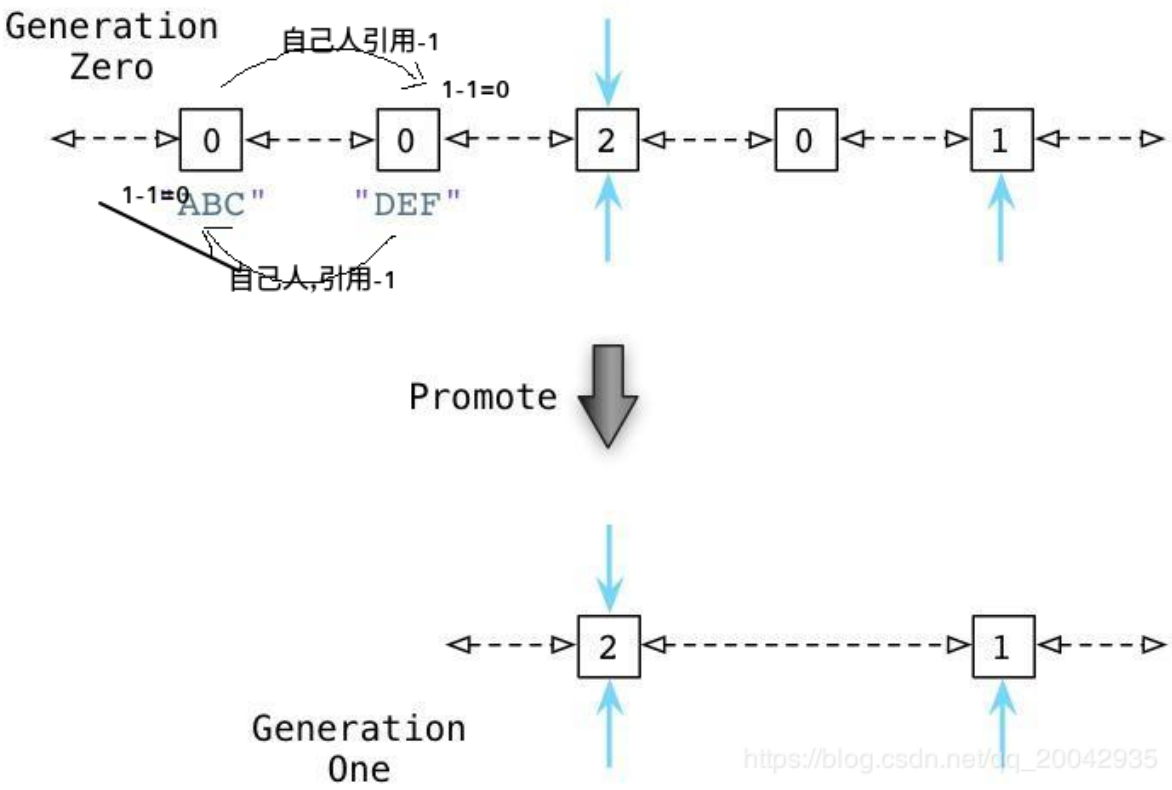
随后，Python 会循环遍历零代列表上的每个对象，检查列表中每个互相引用的对象，根据规则减掉其引用计数。在这个过程中，Python 会一个接一个的统计内部引用的数量以防过早地释放对象。



大多数情况下，循环引用计数其实都是 >1 的，所以-1 后其实也 >0，因为其大概率不止有一个指针，可能会指向多个地址，指向的多个地址中 ~~~~ 有一个地址被循环引用外的元素引用（**也即是说此元素是真正被需要的，非互相循环导致的假需求**），就不会导致计数器为 0

18.2.3 三、分代回收

为了便于理解，来看一个例子：



从而被分配对象的计数值与被释放对象的计数值之间的差异在逐渐增长。一旦这个差异累计超过某个阈值（**说白了就是 0 代留存量超过阈值，0 代链表长度超过阈值**），则 Python 的收集机制就启动了，并且触发上边所说到的零代算法，释放“浮动的垃圾”，并且将剩下的对象移动到一代列表。

而 Python 对于一代列表中对象的处理遵循同样的方法，一旦被分配计数值与被释放计数值累计到达一定阈值，Python 会将剩下的活跃对象移动到二代列表。

通过不同的阈值设置，Python 可以在不同的时间间隔处理这些对象。Python 处理零代最为频繁，其次是一代然后才是二代。

18.3 垃圾收集何时进行?

1. 被引用为 0 时，立即回收当前对象

2. 达到了垃圾回收的阈值，触发标记-清除

3. 手动调用 gc.collect()

4.Python 虚拟机退出的时候

18.4 为什么定义了 ___del___ 的循环引用对象在 Python 中无法收集

从 gc.garbage 的文档中：

Python 不会自动收集此类循环，因为通常来说，Python 不可能猜测出运行 ___del___() 方法的安全顺序。如果您知道安全订单，则可以通过检查垃圾清单并由于清单中的对象而明确中断周期来强制执行此问题。

简单来说，**gc 会计算出循环引用的计数器 =0，所以会尝试回收，但是由于自定义了 ___del___ 方法（重写了 obj 的 ___del___），所以就会懵逼，不知道从循环的哪里下口**。除非必要，**否则别重写 ___del___**，至于 ___del___ 中只有 pass 的就更没必要了.

18.5 代码测试

我们知道了 python 对于垃圾回收，采取的是引用计数为主，标记-清除 + 分代回收为辅的回收策略。对于循环引用的情况，一般的自动垃圾回收方式肯定是无效了，这时候就需要显式地调用一些操作来保证垃圾的回收和内存不泄露。这就要用到 python 内建的垃圾回收模块 gc 模块了。

```
import sys
import gc

a = [1]
b = [2]
a.append(b)
b.append(a)
#### 此时 a 和 b 之间存在循环引用####
sys.getrefcount(a)    # 结果应该是 3
sys.getrefcount(b)    # 结果应该是 3
del a
del b
#### 删除了变量名 a，b 到对象的引用，此时引用计数应该减为 1，即只剩下互相引用了####
try:
    sys.getrefcount(a)
except UnboundLocalError:
     print 'a is invalid'
#### 此时，原来 a 指向的那个对象引用不为 0，python 不会自动回收它的内存空间####
#### 但是我们又没办法通过变量名 a 来引用它了，这就导致了内存泄露####
unreachable_count = gc.collect()
####gc.collect() 专门用来处理这些循环引用，返回处理这些循环引用一共释放掉的对象个数。这里返回是 2####
```

可以看到，没有 gc 模块的时候，我们对循环引用是束手无策的，在调用了一些 gc 模块的方法之后，它会实现上面“垃圾回收机制”部分中提到的一些策略比如“标记-清除”来进行垃圾回收。因为有了这个模块的封装，我们就不用关心具体的实现了。

然而 collect 方法也不是万能的。有些时候它并不能有效地回收所有该回收的对象。比如下面这样一段代码：

```
class A():
  def __init__(self):
    pass
  def __del__(self):
    pass

class B():
  def __init__(self):
    pass
  def __del__(self):
    pass

a = A()
b = B()
a._b = b
b._a = a
del a
del b

print gc.collect()    # 结果是 4
print gc.garbage    # 结果是 [<__main__.A instance at 0x0000000002296448>, <__main__.B
↪instance at 0x0000000002296488>]
```

可以看到，对我们自定义类的对象而言，collect 方法并不能解决循环引用引起的内存泄露，即使在 collect 过后，解释器中仍然存在两个垃圾对象。

这里需要明确一下，之前对于“垃圾”二字的定义并不是很明确，在这里的这个语境下，垃圾是指在经过 collect 的垃圾回收之后仍然保持 unreachable 状态，即无法被回收，且无法被用户调用的对象应该叫做垃圾。gc 模块中有 garbage 这个属性，其为一个列表，每一项都是当前解释器中存在的垃圾对象。一般情况下，这个属性始终保持为空集。

那么为什么在这种场景下 collect 不起作用了呢? 这主要是因为我们在类中重载了 ___del___ 方法。del___ 方法指出了在用 del 语句删除对象时除了释放内存空间以外的操作。一般而言，在使用了 del 语句的时候解释器会首先看要删除对象的引用计数，如果为 0，那么就释放内存并执行 ___del___ 方法。在这里，首先 del 语句出现时本身引用计数就不为 0（因为有循环引用的存在），所以解释器不释放内存；再者，执行 collect 方法时照理由应该会清除循环引用所产生的无效引用计数从而达到 del 的目的，对于这两个对象而言，python 无法判断调用它们的 ___del___ 方法时会不会要用到对方那个对象，比如在进行 b. del () 时可能会用到 b._a 也就是 a，如果在那之前 a 已经被释放，那么就彻底 GG 了。为了避免这种情况，collect 方法默认不对重载了 ___del___ 方法的循环引用对象进行回收，而它们俩的状态也会从 unreachable 转变为 uncollectable。由于是 uncollectable 的，自然就不会被 collect 处理，所以就进入了 garbage 列表。

collect 返回 4 的原因是因为，在 A 和 B 类对象中还默认有一个 ___dict___ 属性，里面有所有属性的信息。比如对于 a，有 a. ___ dict ___ = { ‘_b’ :< **main** .B instance at xxxxxxxx>}。a 的 ___dict___ 和 b 的

___dict___ 也是循环引用的。但是字典类型不涉及自定义的 ___del___ 方法，所以可以被 collect 掉。所以 garbage 里只剩下两个了。

有时候 garbage 里也会出现那两个 ___dict___，这主要是因为在前面可能设置了 gc 模块的 debug 模式，比如 gc.set_debug(gc.DEBUG_LEAK)，会把所有已经回收掉的 unreachable 的对象也都加入到 garbage 里面。set_debug 还有很多参数诸如 gc.DEBUG_STAT|DEBUG_COLLECTABLE|DEBUG_UNCOLLECTABLE|DEBUG_SAVEALL 等等，设置了相关参数后 gc 模块会自动检测垃圾回收状况并给出实时地信息反映。

gc.get_threshold()

这个方法涉及到之前说过的分代回收的策略。python 中默认把所有对象分成三代。第 0 代包含了最新的对象，第 2 代则是最早的一些对象。在一次垃圾回收中，所有未被回收的对象会被移到高一代的地方。

这个方法返回的是 (700,10,10)，这也是 gc 的默认值。这个值的意思是说，在第 0 代对象数量达到 700 个之前，不把未被回收的对象放入第一代；而在第一代对象数量达到 10 个之前也不把未被回收的对象移到第二代。可以是使用 gc.set_threshold(threashold0,threshold1,threshold2) 来手动设置这组阈值。

18.6 参考

【Python】垃圾回收机制和 gc 模块:https://www.cnblogs.com/franknihao/p/7326849.html

Python 垃圾回收机制详解：https://blog.csdn.net/xiongchengluo1129/article/details/80462651

___del___ 的几个坑:https://blog.csdn.net/pirDOL/article/details/51586406

涉及循环引用时 ___del___ 方法未执行:https://www.pythonheidong.com/blog/article/398409/

为什么定义了 ___del___ 的循环引用对象在 Python 中无法收集?：https://www.pythonheidong.com/blog/article/141888/

CHAPTER 19

python 进阶 19 装饰器

19.1 Nested functions

Python 允许创建嵌套函数，这意味着我们可以在函数内声明函数并且所有的作用域和声明周期规则也同样适用。

```
>>> def outer():
...     x = 1
...     def inner():
...         print x # 1
...     inner() # 2
...
>>> outer()
```

这看起来稍显复杂，但其行为仍相当直接，易于理解。考虑一下在 #1 处发生了什么——Python 寻找一个名为 x 的 local 变量，失败了，然后在最邻近的外层作用域里搜寻，这个作用域是另一个函数！变量 x 是函数 outer 的 local 变量，但是和前文提到的一样，inner 函数拥有对外层作用域的访问权限（最起码有读和修改的权限）。在 #2 处我们调用了 inner 函数。请记住 inner 也只是一个变量名，它也遵从 Python 的变量查找规则——Python 首先在 outer 的作用域里查找之，找到了一个名为 inner 的 local 变量。

19.2 Closures

让我们不从定义而是从另一个代码示例开始。如果我们将上一个例子稍加修改会怎样呢?

```
>>> def outer():
...     x = 1
...     def inner():
...         print x # 1
...     return inner
>>> foo = outer()
>>> foo.func_closure
(\<cell at 0x.... int object at 0x...>,)
```

从上一个例子中我们看到 inner 是一个由 outer 返回的函数，存储于一个名为 foo 的变量，我们可以通过 foo() 调用它。但是它能运行吗？让我们先来考虑一下作用域规则。一切都依照 Python 的作用域规则而运行——x 是 outer 函数了一个 local 变量。当 inner 在 #1 处打印 x 时，Python 在 inner 中寻找一个 local 变量，没有找到；然后它在外层作用域即 outer 函数中寻找并找到了它。但是自此处从变量生命周期的角度来看又会如何呢？变量 x 是函数 outer 的 local 变量，这意味着只有当 outer 函数运行时它才存在。只有当 outer 返回后我们才能调用 inner，因此依照我们关于 Python 如何运作的模型来看，在我们调用 inner 的时候 x 已经不复存在了，那么某个运行时错误可能会出现。事实与我们的预想并不一致，返回的 inner 函数的确正常运行。Python 支持一种称为闭包 (function closures) 的特性，这意味着定义于非全局作用域的 inner 函数在定义时记得记得它们的外层作用域长什么样儿。这可以通过查看 inner 函数的 func_closure 属性来查看，它包含了外层作用域里的变量。请记住，每次当 outer 函数被调用时 inner 函数都被重新定义一次。目前 x 的值没有改变，因此我们得到的每个 inner 函数和其它的 inner 函数拥有相同的行为，但是如果我们将它做出一点改变呢？

```
>>> def outer(x):
...     def inner():
...         print x # 1
...     return inner
>>> print1 = outer(1)
>>> print2 = outer(2)
>>> print1()
1
>>> print2()
2
```

从这个例子中你可以看到 closures——函数记住他们的外层作用域的事实——可以用来构建本质上有一个硬编码参数的自定义函数。我们没有将数字 1 或者 2 传递给我们的 inner 函数但是构建了能”记住”其应该打印数字的自定义版本。closures 独自就是一个强有力的技术——你甚至想到在某些方面它有点类似于面向对象技术：outer 是 inner 的构造函数，x 扮演着一个类似私有成员变量的角色。它的作用有很多，如果你熟悉 Python 的 sorted 函数的 key 参数，你可能已经写过一个 lambda 函数通过第二项而不是第一项来排序一些列 list。也许你现在可以写一个 itemgetter 函数，它接收一个用于检索的索引并返回一个函数，这个函数适合传递给 key 参数。

但是让我们不要用闭包做任何噩梦般的事情！相反，让我们重新从头开始来写一个 decorator!

19.3 Decorators!

一个 decorator 只是一个带有一个函数作为参数并返回一个替换函数的闭包。我们将从简单的开始一直到写出有用的 decorators。

```
>>> def outer(some_func):
...     def inner():
...         print "before some_func"
...         ret = some_func() # 1
...         return ret + 1
...     return inner
>>> def foo():
...     return 1
>>> decorated = outer(foo) # 2
>>> decorated()
before some_func
2
```

请仔细看我们的 decorator 实例。我们定义了一个接受单个参数 some_func 的名为 outer 的函数。在 outer 内部我们定义了一个名为 inner 的嵌套函数。inner 函数打印一个字符串然后调用 some_func，在 #1 处缓存它的返回值。some_func 的值可能在每次 outer 被调用时不同，但是无论它是什么我们都将调用它。最终，inner 返回 some_func 的返回值加 1，并且我们可以看到，当我们调用存储于 #2 处 decorated 里的返回函数时我们得到了输出的文本和一个返回值 2 而不是我们期望的调用 foo 产生的原始值 1.

我们可以说“装饰”的变量是 foo 的一个装饰版本——由 foo 加上一些东西构成。实际上，如果我们写了一个有用的 decorator，我们可能想用装饰了的版本一起来替换 foo，从而我们可以总是得到 foo 的“增添某些东西”的版本。我们可以不用学习任何新语法而做到这一点——重新将包含我们函数的变量进行赋值：

```
>>> foo = outer(foo)
>>> foo # doctest: +ELLIPSIS
<function inner at 0x...>
```

现在任何对 foo() 的调用都不会得到原始的 foo，而是会得到我们经过装饰的版本！领悟到了一些 decorator 的思想吗？让我们写一个更加有用的装饰器。假设我们有一个提供坐标对象的库，它们可能只是由 x, y 两个坐标对组成。令人沮丧的是，这个坐标对象并不支持算术运算，并且我们无法修改这个库的源代码，因此我们不能添加这些对运算的支持。我们将做大量的运算，但是我们现在只想实现加、减函数，它们可以带两个坐标最想作为参数并做相应的算术运算。这些函数可能很容易写（为了描述我将提供一个简单的 Coordinate 类。

```
>>> class Coordinate(object):
...     def __init__(self, x, y):
...         self.x = x
```

（下页继续）

(续上页)

```
...         self.y = y
...     def __repr__(self)
...         return "Coord:" + str(self.__dict__)
>>> def add(a, b):
...     return Coordinate(a.x + b.x, a.y + b.y)
>>> def sub(a, b):
...      return Coordinate(a.x - b.x, a.y - b.y)
>>> one = Coordinate(100, 200)
>>> two = Coordinate(300, 200)
>>> add(one, two)
Coord:{'y': 400, 'x': 400}
```

但是，我们想当 one 和 two 都是 {x: 0, y: 0},one 和 three 的和为 {x: 100, y: 200}，在不修改 one, two, three 的前提下结果有所不同（实在没弄明白原作者此处是什么意思 ^ ^)。让我们写一个边界检查 decorator 而不用为每个函数添加一个对输入参数做边界检查然后返回函数值!

```
>>> def wrapper(func):
...     def checker(a, b): # 1
...         if a.x < 0 or a.y < 0:
...             a = Coordinate(a.x if a.x > 0 else 0, a.y if a.y > 0 else 0)
...         if b.x < 0 or b.y < 0:
...             b = Coordinate(b.x if b.x > 0 else 0, b.y if b.y > 0 else 0)
...         ret = func(a, b)
...         if ret.x < 0 or ret.y < 0:
...             ret = Coordinate(ret.x if ret.x > 0 else 0, ret.y if re> 0 else 0)
...         return ret
...     return checker
>>> add = wrapper(add)
>>> sub = wrapper(sub)
>>> sub(one, two)
Coord: {'y': 0, 'x': 0}
>>> add(one, three)
Coord: {'y': 200, 'x': 100}
```

这个装饰器的效果和前面实例的一样——返回一个修改过了的函数，只是在上例中对输入参数和返回值做了一些有用的检查和规范化，至于这样做是否让我们的代码变得更加简洁是一件可选择的事情：将边界检查隔绝在它自己的函数里，然后将其应用到通过用一个 decorator 包装将我们所关心的函数上。另一个可能的方法是每次调用算数函数时对每一个输入参数和输出结果前对参数或者结果做边界检查，毫无疑问的是使用 decorator 至少在对一个函数进行边界检查的代码量上重复更少。实际上，如果是装饰我们自己的函数，我们可以将装饰器应用程序写的更明显一点。

19.4 含参装饰器

可以在装饰器里传入一个参数，指明国籍，并在函数执行前，用自己国家的母语打一个招呼。

```
# 小明，中国人
@say_hello("china")
def xiaoming():
    pass

# jack，美国人
@say_hello("america")
def jack():
    pass
那我们如果实现这个装饰器，让其可以实现 传参 呢？

会比较复杂，需要两层嵌套。

def say_hello(contry):
    def wrapper(func):
        def deco(*args, **kwargs):
            if contry == "china":
                print(" 你好!")
            elif contry == "america":
                print('hello.')
            else:
                return

            # 真正执行函数的地方
            func(*args, **kwargs)
        return deco
    return wrapper

来执行一下
xiaoming()
print("------------")
jack()
```

看看输出结果。

```
你好!
------------
```

(下页继续)

(续上页)

```
hello.
```

19.5 用偏函数与类实现装饰器

绝大多数装饰器都是基于函数和闭包实现的，但这并非制造装饰器的唯一方式。事实上，Python 对某个对象是否能通过装饰器（@decorator）形式使用只有一个要求：decorator 必须是一个“可被调用（callable）的对象。对于这个 callable 对象，我们最熟悉的就是函数了。除函数之外，类也可以是 callable 对象，只要实现了 ___call___ 函数（上面几个例子已经接触过了）。还有容易被人忽略的偏函数其实也是 callable 对象。

```
import time
import functools

class DelayFunc:
    def __init__(self,  duration, func):
        self.duration = duration
        self.func = func

    def __call__(self, *args, **kwargs):
        print(f'Wait for {self.duration} seconds...')
        time.sleep(self.duration)
        return self.func(*args, **kwargs)

    def eager_call(self, *args, **kwargs):
        print('Call without delay')
        return self.func(*args, **kwargs)

def delay(duration):
    """
    装饰器：推迟某个函数的执行。
    同时提供 .eager_call 方法立即执行
    """
    # 此处为了避免定义额外函数，
    # 直接使用 functools.partial 帮助构造 DelayFunc 实例
    return functools.partial(DelayFunc, duration)
我们的业务函数很简单，就是相加

@delay(duration=2)
def add(a, b):
    return a+b
```

来看一下执行过程

```
>>> add    # 可见 add 变成了 Delay 的实例
<__main__.DelayFunc object at 0x107bd0be0>
>>>
>>> add(3,5)  # 直接调用实例，进入 __call__
Wait for 2 seconds...
8
>>>
>>> add.func # 实现实例方法
<function add at 0x107bef1e0>
```

19.6 参考

【翻译】12 步理解 Python Decorators：https://harveyqing.gitbooks.io/python-read-and-write/content/python_advance/python_decorator_in_12_steps.html 装饰器进阶用法详解：python.iswbm.com/en/latest/c03/c03_01.htmlpython 中的闭包:https://blog.csdn.net/weixin_44141532/article/details/87116038

CHAPTER 20

python 进阶 20 之 actor

actor 模型。actor 模式是一种最古老的也是最简单的并行和分布式计算解决方案。

优点：充分利用单线程＋事件机制，达到了多线程效果。

缺点，对 python 而言，由于 GIL 的存在，毕竟只是单线程，难以匹敌多进程，目前使用并不多。

20.1 简单任务调度器

```
class TaskScheduler:
    def __init__(self):
        self._task_queue = deque()

    def new_task(self, task):
        '''
        Admit a newly started task to the scheduler
        '''
        self._task_queue.append(task)

    def run(self):
        '''
        Run until there are no more tasks
        '''
```

(下页继续)

(续上页)

```
        while self._task_queue:
            task = self._task_queue.popleft()
            try:
                # Run until the next yield statement
                next(task)
                self._task_queue.append(task)
            except StopIteration:
                # Generator is no longer executing
                pass

# Example use
sched = TaskScheduler()
sched.new_task(countdown(10))
sched.new_task(countdown(5))
sched.new_task(countup(15))
sched.run()
```

20.2 协程生产者消费者

廖雪峰的 python 官网教程里面的协程生产者消费者

```
def consumer():
    r = ''
    while True:
        n = yield r
        if not n:
            return
        print('[CONSUMER] Consuming %s...' % n)
        r = '200 OK'

def produce(c):
    c.send(None)
    n = 0
    while n < 5:
        n = n + 1
        print('[PRODUCER] Producing %s...' % n)
        r = c.send(n)
        print('[PRODUCER] Consumer return: %s' % r)
    c.close()
```

(下页继续)

(续上页)

```
c = consumer()
produce(c)
```

20.3 并发网络应用程序

演示了使用生成器来实现一个并发网络应用程序:

```
class ActorScheduler:
    def __init__(self):
        self._actors = {}
        self._msg_queue = deque()

    def new_actor(self, name, actor):
        self._msg_queue.append((actor, None))
        self._actors[name] = actor

    def send(self, name, msg):
        actor = self._actors.get(name)
        if actor:
            self._msg_queue.append((actor, msg))

    def run(self):
        while self._msg_queue:
            # print(" 队列: ", self._msg_queue)
            actor, msg = self._msg_queue.popleft()
            # print("actor", actor)
            # print("msg", msg)
            try:
                actor.send(msg)
            except StopIteration:
                pass

if __name__ == '__main__':
    def say_hello():
        while True:
            msg = yield
            print("say hello", msg)
```

(下页继续)

(续上页)

```
    def say_hi():
        while True:
            msg = yield
            print("say hi", msg)

    def counter(sched):
        while True:
            n = yield
            print("counter:", n)
            if n == 0:
                break
            sched.send('say_hello', n)
            sched.send('say_hi', n)
            sched.send('counter', n-1)

    sched = ActorScheduler()
    # 创建初始化 actors
    sched.new_actor('say_hello', say_hello())
    sched.new_actor('say_hi', say_hi())
    sched.new_actor('counter', counter(sched))

    sched.send('counter', 10)
    sched.run()
```

20.4 参考

扩展 Python Gevent 的 Actor 模型：https://www.dazhuanlan.com/2020/02/29/5e5a7f241ed15/

终结 python 协程—-从 yield 到 actor 模型的实现：https://www.bbsmax.com/A/n2d9bQaYzD/

12.12 使用生成器代替线程:https://python3-cookbook.readthedocs.io/zh_CN/latest/c12/p12_using_generators_as_alterna

CHAPTER 21

python 进阶 21 再识单例模式

父子进程内部变量是否可以直接共享，当然不是，需要“特殊加工”下才行。

那么在 web 开发中的单例模式，是**真正的全局唯一的单例**么？自然也是**否**

惭愧，自己用单例还是比较多的，还真是第一次注意到这一点。之前使用时，**想当然的以为就是（应用程序级别）全局唯一的，譬如 java 的类里的 static，python 模块中的定义的对象**（只会加载一次），但严格说，**都是错误的用法**（侥幸的是，尚未出现由此导致的 Bug，大概率因为自己用单例大多是为了保存静态内容（只查，不改），加速查询而已。并未用来做全局性统计）。

21.1 如何理解单例模式中的唯一性？

根据定义，一个类只允许创建唯一的一个对象，对象的唯一作用范围是什么？是指**进程内只允许创建一个**，还是线程内只允许创建一个？答案是前者，也就是说单例创建的对象是进程内唯一的。怎么理解呢，我们编写的程序最终执行时，都是操作系统先启动一个进程，然后将程序（可执行文件）加载到内存地址空间，一条一条执行其中的指令，遇到类的实例化时就分配内存地址给新的对象，如果该进程 fork 了另外的新进程，操作系统会分配新的地址空间，并将原来的进程空间的所有内容全部复制到新的地址空间，包括已经实例化的对象，单例类在老进程内只有一个，在新进程内也只有一个，也就是说**进程内唯一，进程间不唯一**。

21.2 如何实现线程唯一的单例？

单例类默认是进程内只存在唯一的对象，进程又包含一个或多个线程，这也意味着在线程间也是唯一的，那么如何改进，实现线程内唯一，线程间不唯一的单例呢？其实也非常简单，我们只需借助一个字典，将线程 ID 作为键，单例类对象作为值进行存储，这样就可以做到相同的线程对应相同的单例对象。

21.3 如何在集群环境中实现单例？

刚才说了进程内唯一，线程内唯一，现在提到的集群环境中实现单例，就是集群内唯一，实质就是进程间唯一。进程之间是不共享内存的，那就需要借助外部存储来实现，比如文件或数据库，或像 redis 一样具有存储功能的中间件。我们把单例类对象通过序列化保存在外部存储，进程在使用这个单例类对象时先访问外部存储，然后反序列化成对象使用，使用完成后在序列化保存在外部存储。

为了保证任何时刻，在进程之间只有一份对象存在，一个进程在反序列化获取对象之后需要对对象加锁，防止其他进程获取该对象，使用完后序列化保存到外部存储，然后显式的从内存中删除对象 (instance = None)，并释放锁。

21.4 django 之全局变量

序列化存储:python 有 zodb django-solo helps working with singletons: things like global settings that you want to edit from the admin site.

python 的 web 应用都需要用 uwsgi 或 gunicorn 之类的多进程服务器，进程之间的全局变量实际上是相互隔离的。所有只能用 redis 或 django 的 cache 这种公共存储。

最省事是用 Django 的缓存

21.5 参考

你是否真的理解单例模式？:https://www.ershicimi.com/p/7cb8aab37c6653a09b8e5b187739af67

uwsgi 多进程配合 kafka-python 消息无法发送:https://www.cnblogs.com/MnCu8261/p/10482031.html

django 中如何维护一个全局变量:https://www.v2ex.com/t/504042

CHAPTER 22

Indices and tables

- genindex
- modindex
- search