

Python基础 C05 — 并发编程

信息科学技术学院

胡俊峰



内容提要

- 多进程、多线程运行与交互
- 协程与任务分时任务调度
- 事件循环机制
- Tkinter简介



Python中的并发处理

- 多进程
 - 进程通讯机制
- 多线程
 - 线程锁与同步
- 协程-事件轮转机制
 - 单线程分时任务处理



并发：

- 1、程序要同时处理多个任务
- 2、经常需要等待资源响应
- 3、多系统分时协作任务

- 游戏：同时显示场景、播放声音、相应用户输入
- 网络服务器：同时与多个客户端建立连接、处理请求
- 多个功能化例程协作完成任务
- 方法：并行/并发编程
 - 并行：如多核CPU不同核上跑的两个进程。两个计算流在时间上重叠。
 - 并发：如单核CPU上跑的两个进程。两个计算流在时间上交替执行。给我们带来宏观上两个进程同时运行的假象。



计算机如何执行程序？

- 内核在负责资源的调度
- 每个进程有独立的逻辑控制流、私有的虚拟地址空间
- 维护一个进程需要
 - 程序计数器、通用目的寄存器、浮点寄存器、状态寄存器
 - 用户栈、内核栈、内核数据结构（用来映射虚拟地址的页表、当前打开文件信息的文件表）——进程的上下文

具体执行中是通过进程头与CPU的寄存器组配合保留-回复进程的私有运行环境



程序执行时的私有环境:

- 程序计数器
 - 控制读到哪一句
- 通用目的寄存器、浮点寄存器、条件码寄存器
 - 存某些变量的值、中间计算结果、栈指针、子程序返回值……

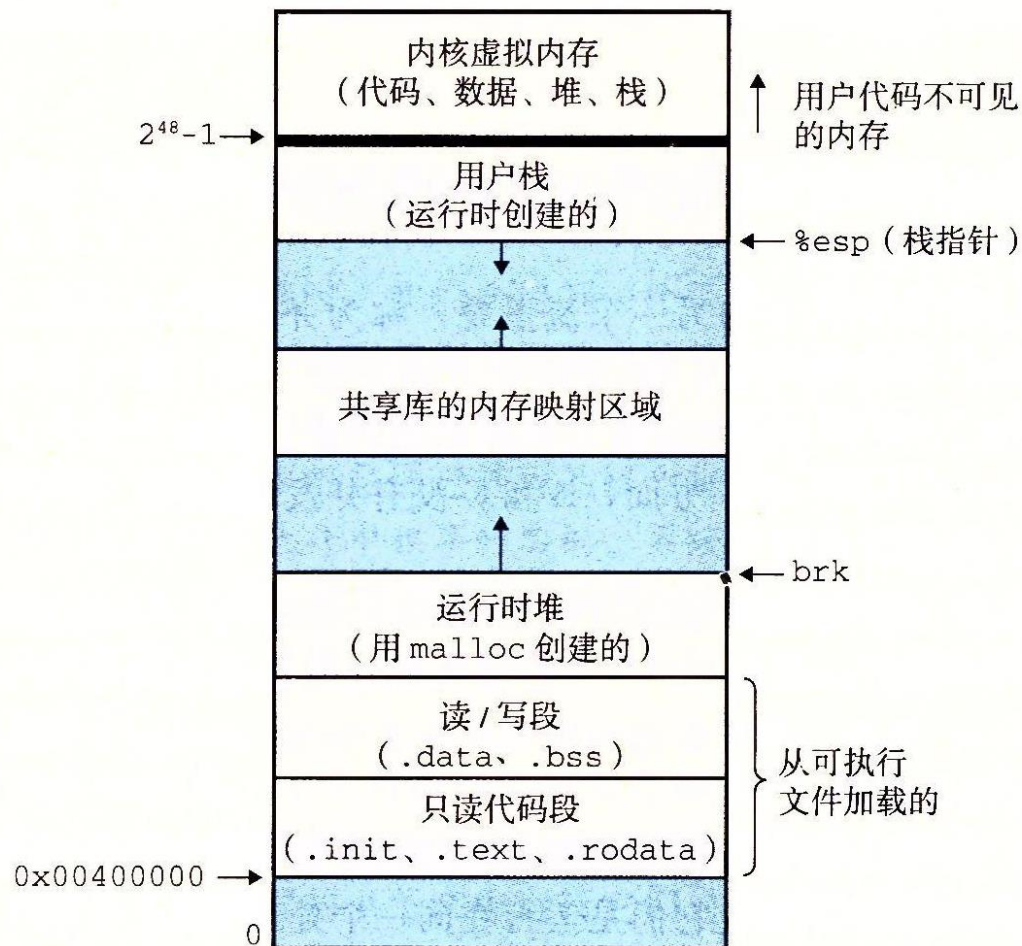


图 8-13 进程地址空间

内核：分时调度和共享资源

- 内核负责进程的挂起和唤醒，
 - 在进程执行期间进行上下文的切换
- 进程（process）：虚地址空间独立运行。其资源被内核保护起来。

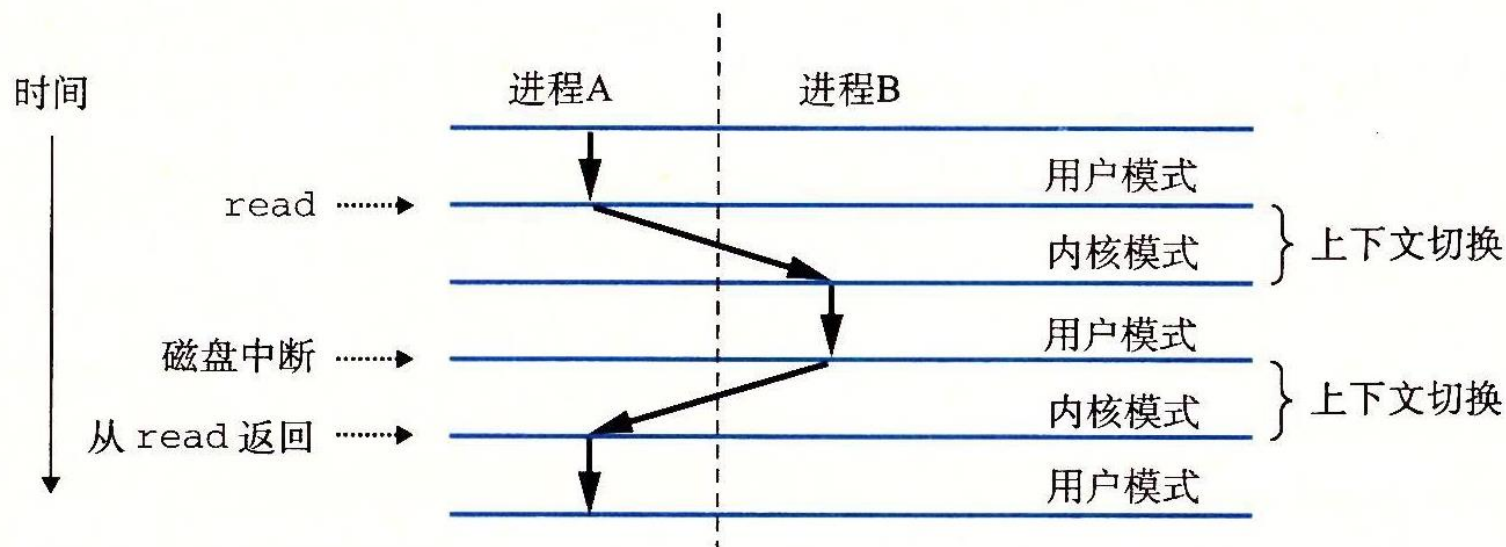


图 8-14 进程上下文切换的剖析

线程

- 由于进程之间不共享内存，进程的切换-协同效率低
- 线程（thread）：调度执行的最小单元
 - 每个线程运行在单一进程中
 - 一个进程中可以有多个线程
- 线程上下文
 - 程序计数器、通用目的寄存器、浮点寄存器、条件码
 - 线程ID，栈，栈指针
- 线程间可以共享进程空间的公有数据：进程打开的文件描述符、信号-锁处理器、进程的当前目录和进程用户ID与进程组ID。因此可以方便的进行通讯。

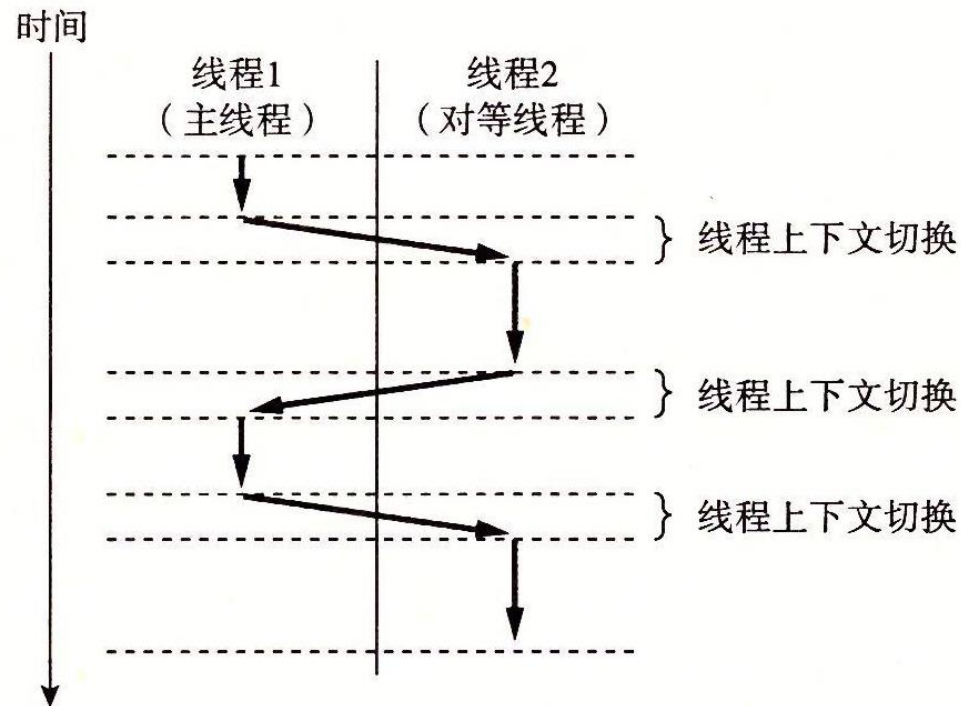


图 12-12 并发线程执行

多进程、多线程编程

- 多进程

程序之间不共享内存，使用虚拟地址映射保证资源独立

- 优点：一个进程挂了不影响别的进程
- 缺点：切换上下文效率低，通信和信息共享不太方便

- 多线程

- 优点：上下文切换效率比多进程高，线程之间信息共享和通信方便
- 缺点：一个线程挂了会使整个进程挂掉。操作全局变量需要锁机制

不同的进程、线程之间总是并发/并行的，

- 如果运行在多核CPU不同的核上，是可以并行的。



线程锁机制

- 多个逻辑控制流同时读写共享的资源时，需要加入锁机制
- 例：100个线程，每个都对一个全局变量cnt不加锁操作 $\text{cnt}+=1$
- 执行完后结果 ≤ 100

加锁

- 需要读写某个共享变量时，读写前上锁，读写后释放锁
- 如果要读写的共享变量上了锁，等待锁释放后再上锁读写。



多进程模式：Process对象

p.start ()	启动进程,并调用子进程中的p.run ()
p.run ()	进程启动时运行的方法，正是他去调用target指定的函数
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])	主进程等待p终止（是主进程处于等待的状态，而p是处于运行状态），timeout是可选的超时时间

- 对主进程来说，运行完毕指的是主进程代码运行完毕



多进程编程：

```
import time
from multiprocessing import Process

def func():
    time.sleep(3)
    print("child process done!")

if __name__ == '__main__':
    p = Process(target=func)
    p.start()
    time.sleep(2)
    #p.join()
    print("parent process done!")
```

```
(base) C:\Users\hjfat\test>python test4-2.py
parent process done!
child process done!
```



Python 多进程编程

```
from multiprocessing import Process
import os
import time
```

```
def hello(i):
    print('son process id {} - for task {}'.format(os.getpid(), i))
    time.sleep(2)
```

```
if __name__ == '__main__':
    print('current father process {}'.format(os.getpid()))
    start = time.time()
    p1 = Process(target=hello, args=(1,))
    p2 = Process(target=hello, args=(2,))
    p1.start() # start son process
    p2.start()
    p1.join() # wait until it finishes
    p2.join()
    end = time.time()
    print("Totally take {} seconds".format((end - start)))
```

```
current father process 9976
son process id 8508 - for task 1
son process id 5836 - for task 2
Totally take 2.420004367828369 seconds
```

← 包装一个进程



```
1 %cd c:\users\hujf\mp
```

c:\users\hujf\mp

```
1 %%writefile test2.py
2 from multiprocessing import Process
3 import os
4
5 def info(title):
6     print(title)
7     print('module name:', __name__)
8     print('parent process:', os.getppid())
9     print('process id:', os.getpid())
10
11 def f(name):
12     info('function f')
13     print('hello', name)
14
15 if __name__ == '__main__':
16     info('main line')
17     p = Process(target=f, args=('bob',))
18     p.start()
19     p.join()
```

Overwriting test2.py

```
1 %run test2  # 运行脚本
```

main line
module name: __main__
parent process: 11744
process id: 10168

Notebook下使用方案:

```
(base) C:\Users\hujf\mp>python test2.py
main line
module name: __main__
parent process: 18532
process id: 8652
function f
module name: __mp_main__
parent process: 8652
process id: 15120
hello bob

(base) C:\Users\hujf\mp>
```



多进程池（并发计算资源共享）

- 进程池

- 定义一个池子, 在里面放上固定数量的进程, 有任务要处理的时候就会拿一个池中的进程来处理任务, 等到处理完毕, 进程**并不关闭**而是放回进程池中继续等待任务。
- 如果很多任务需要执行, 池中的进程数不够, 任务会就要等待拿到空闲的进程才继续执行。

- multiprocessing.Pool模块

- Pool ([numprocess [,initializer [, initargs]]]) : 创建进程池

numprocess	要创建的进程数, 如果省略, 将默认使用os.cpu_count () 的值
initializer	是个工作进程启动时要执行的可调用对象, 默认为None
initargs	传给initializer的参数组



多进程任务调用方式:

- multiprocessing.Pool

p.apply (<u>func</u> [,args [,kwargs]])	在一个池工作进程中执行func(*args,**kwargs), 然后返回结果 (同步调用, 阻塞主进程)
p.apply_async(func [,args [,kwargs]])	在一个池工作进程中执行func(*args,**kwargs), 然后返回结果 (异步调用)
p.close()	关闭进程池, 防止进一步操作. 如果所有操作持续挂起, 他们将在工作进程终止前完成
p.join()	等待所有工作进程退出. 此方法只能在close () 或terminate () 之后调用



Python 多进程编程

- 进程服务模式，使用进程池管理

```
from multiprocessing import Process, Pool
import os
import time
```

```
def hello(i):
    print('son process id {} - for task {}'.format(os.getpid(), i))
    time.sleep(1)
```

```
if __name__ == '__main__':
    print('current father process {}'.format(os.getpid()))
    start = time.time()
    p = Pool(4) # 4 kernel CPU.
    for i in range(5):
        p.apply_async(hello, args=(i,))
    p.close() # no longer receive new process
    p.join() # wait until all processes in the pool finishes
    end = time.time()
    print("Totally take {} seconds".format((end - start)))
```

```
current father process 6316
son process id 7632 - for task 0
son process id 9044 - for task 1
son process id 8376 - for task 2
son process id 7820 - for task 3
son process id 7632 - for task 4
Totally take 2.717404842376709 seconds
```



多进程异步模式:

- multiprocessing.Pool

```
((base) C:\Users>python tester.py
Parent process 8440.
Waiting for all subprocesses done...
Run task 0 (14028)...
Run task 1 (1640)...
Run task 2 (6380)...
Run task 3 (14212)...
Task 0 runs 0.35 seconds.
Run task 4 (14028)...
Task 3 runs 1.41 seconds.
Task 2 runs 1.71 seconds.
Task 4 runs 1.53 seconds.
Task 1 runs 2.39 seconds.
All subprocesses done.
```

In [*]:

```
from multiprocessing import Pool
import os, time, random

def long_time_task(name):
    print('Run task %s (%s)...' % (name, os.getpid()))
    start = time.time()
    time.sleep(random.random() * 3)
    end = time.time()
    print('Task %s runs %0.2f seconds.' % (name, (end - start)))

if __name__ == '__main__':
    print('Parent process %s.' % os.getpid())
    p = Pool(4)
    for i in range(5):
        p.apply_async(long_time_task, args=(i,))
    print('Waiting for all subprocesses done...')
    p.close()
    p.join()
    print('All subprocesses done.')
```

Parent process 24148.

Waiting for all subprocesses done...

- apply_async是异步的，子进程执行的同时，主进程继续向下执行。所以“Waiting for all subprocesses done...”先打印出来，“All subprocesses done.”最后打印。
- task 0, 1, 2, 3是立刻执行的，而task 4要等待前面某个task释放资源后才执行



进程间通讯

- 两个进程传递消息，可以使用**Pipe()**。多个进程间共享消息或数据可以采用**队列Queue**（允许多个生产者和消费者）。
 - 尽量避免自己设计资源锁模式进行同步
- multiprocessing使用**queue.Empty**和 **queue.Full**异常
- **Queue** 有两个方法，**get** 和 **put** （可以设定阻塞或非阻塞）



进程的主从通讯

- Pipe

- Pipe() 返回一个由管道连接的连接对象，默认情况下是双工（双向）

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    print(conn.recv())
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())
    parent_conn.send('Hi')
    p.join()
```

```
(base) C:\Users\hujf\mp>python test6-1.py
[42, None, 'hello']
Hi
```

多进程模式下的消息队列通讯

- Queue
 - Queue 是一个近似 queue.Queue 的克隆

q.put (item)	将item放入队列中, 如果当前队列已满, 就会阻塞, 直到有数据从管道中取出
q.put_nowait (item)	将item放入队列中, 如果当前队列已满, 不会阻塞, 但是会报错
q.get ()	返回放入队列中的一项数据, 取出的数据将是先放进去的数据, 若当前队列为空, 就会阻塞, 直到放入数据进来
q.get_nowait ()	返回放入队列中的一项数据, 同样是取先放进队列中的数据, 若当前队列为空, 不会阻塞, 但是会报错



```
from multiprocessing import Process, Queue
```

```
def f1(q):
```

```
    print('f1 start...')
```

```
    q.put([42, None, 'hello'])
```

```
    print('f1 end')
```

```
def f2(q):
```

```
    print('f2 start...')
```

```
    info = q.get()  # 从队列中取消息, 阻塞式
```

```
    print(info)
```

```
    print('f2 end')
```

```
if __name__ == '__main__':
```

```
    q = Queue()
```

```
    p1 = Process(target=f2, args=(q,))
```

```
    p1.start()
```

```
    p2 = Process(target=f1, args=(q,))
```

```
    p2.start()
```

```
    p1.join()
```

```
    p2.join()
```

```
(base) C:\Users\hujf\mp>python test5-1.py
f2 start...
f1 start...
f1 end
[42, None, 'hello']
f2 end
```

消息队列的常见应用：生产者-消费者

- 生产者产生数据
- 消费者读取数据
- 数据常常保存在一个消息队列中



Python 多进程编程

- Python中提供了队列进行数据共享
 - Multiprocessing.Queue

```
Produce 0
Consume 0
Produce 1
Consume 1
Produce 2
Consume 2
Produce 3
Consume 3
Produce 4
Consume 4
Take 5.543811798095703 s.
```

```
from multiprocessing import Process, Queue
import os, time, random

# 写数据进程执行的代码:
def producer(q):
    for value in range(5):
        print('Produce %d' % value)
        q.put(value)
        time.sleep(1)

# 读数据进程执行的代码:
def consumer(q):
    while True:
        value = q.get(True)
        print('Consume %d' % value)
        time.sleep(1)

if __name__ == '__main__':
    t0 = time.time()
    # 父进程创建Queue, 并传给各个子进程
    q = Queue()
    pw = Process(target=producer, args=(q,))
    pr = Process(target=consumer, args=(q,))
    # 启动子进程pw, 写入
    pw.start()
    # 启动子进程pr, 读取
    pr.start()
    # 等待pw结束:
    pw.join()
    # pr进程里是死循环, 无法等待其结束, 只能强行终止
    pr.terminate()
    print("Take %s s." % (time.time() - t0))
```



```
from multiprocessing import Process
```

```
class MyProcess(Process):  # 自定义的类要继承Process类
```

```
    pool = [0,0]  # 每个进程继承独立的副本
```

```
    def __init__(self, n, name):
```

```
        super().__init__()  # 如果自己想要传参name, 那么要首先用super()执行父类的init方法
```

```
        self.n = n
```

```
        self.name = name
```

```
➡ def run(self):  # 在start方法后运行该方法
```

```
    print("子进程的名字是>>>", self.name)
```

```
    if MyProcess.pool[0] == 0:
```

```
        MyProcess.pool[0] = self.n
```

```
        MyProcess.pool[1] += 1
```

```
    else:
```

```
        MyProcess.pool[1] = self.n
```

```
    print(MyProcess.pool)
```

```
if __name__ == '__main__':
```

```
    p1 = MyProcess(101, name="子进程01")
```

```
    p2 = MyProcess(102, name="子进程02")
```

```
    p1.start()  # 给操作系统发送创建进程的指令, 子进程创建好之后, 要被执行, 执行的时候就会执行run方法
```

```
    p2.start()
```

```
    p1.join()
```

```
    p2.join()
```

```
    print("主进程结束")
```

通过封装进程类来个性化进程的运行环境

```
C:\Users\hujf\2021Python\demo-code>python test10-1.py
子进程的名字是>>> 子进程01
[101, 1]
子进程的名字是>>> 子进程02
[102, 1]
```



Python 多线程编程


- 代码形式和多进程很类似，共享主线程环境。

```
import threading
import time

def hello(i):
    print('thread id: {} for task {}'.format(threading.current_thread().name, i))
    time.sleep(2)

if __name__ == '__main__':
    start = time.time()

    t1 = threading.Thread(target=hello, args=(1,))
    t2 = threading.Thread(target=hello, args=(2,))
    t1.start()
    t2.start()
    t1.join()
    t2.join()
    end = time.time()
    print("Take {} s".format((end - start)))
```



```
thread id: Thread-1 for task 1
thread id: Thread-2 for task 2
Take 2.0140459537506104 s
```

Python多线程编程（资源锁）

- 线程间可以共享全局变量
- 可以用 锁（Lock） 机制来协调多进程的运行
- 在CPython中由于全局解释器锁（GIL） 的存在
 - 全局解释器锁： 一个进程任一时刻仅有一个线程在执行
 - 多核CPU并不能为它显著提高效率
 - 可以考虑选择没有GIL的Python解释器（如JPython）



```
import time
from multiprocessing import Process, Lock

def f(i, l):

    #l.acquire() # 加锁

    try:
        print('hello world', i)
        time.sleep(1)
        print(i, "do something.")

    finally:
        pass
        #l.release() # 保证会释放

if __name__ == '__main__':
    lock = Lock()

    for num in range(5): # 派生5个进程
        Process(target=f, args=(num, lock)).start()
```

```
C:\Users\hujf\2021Python\demo-code>python test7-2.py
hello world 1
1 do something.
hello world 0
0 do something.
hello world 3
3 do something.
hello world 4
4 do something.
hello world 2
2 do something.
```

```
C:\Users\hujf\2021Python\demo-code>python test7-1.py
hello world 0
hello world 1
hello world 2
hello world 3
hello world 4
1 do something.
2 do something.
3 do something.
0 do something.
4 do something.
```

资源管理器：独占同时防止意外死锁：

```
from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()

if __name__ == '__main__':
    with Manager() as manager: # 类似上下文管理器，先加再解
        d = manager.dict()
        l = manager.list(range(10))

        p = Process(target=f, args=(d, l))
        p.start()
        p.join()

        print(d)
        print(l)
```



小结一下：

- 进程：完全独立。通讯要靠系统缓冲区或管道
- 线程：环境不独立、要加锁解决冲突问题。



协程

- 本质上是一个可被异步唤醒的函数
- 存在自阻塞操作（语句）的被动服务函数
- 存在发出调用操作并等待调用返回才继续执行的主动客户方
- 协程的调度由用户程序（而非系统内核）自己来控制



Python迭代器

```
def fib(n):  
    index = 0  
    a = 0  
    b = 1  
  
    while index < n:  
        receive = yield b  
        print('`fib` receive %d' % receive)  
        a, b = b, a+b  
        index += 1
```

```
`fib` yield 1  
`fib` receive 1  
`fib` yield 1  
`fib` receive 1  
`fib` yield 2  
`fib` receive 1  
`fib` yield 3  
`fib` receive 1  
`fib` yield 5
```

```
fib = fib(20)  
print('`fib` yield %d ' % fib.send(None)) # 效果等同于print(next(fib))  
print('`fib` yield %d ' % fib.send(1))  
print('`fib` yield %d ' % fib.send(1))  
print('`fib` yield %d ' % fib.send(1))  
print('`fib` yield %d ' % fib.send(1))
```

executed in 36ms, finished 10:32:16 2020-11-25



Python用迭代器实现协程

```
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()

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

```
[PRODUCER] Producing 1...
[CONSUMER] Consuming 1...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 2...
[CONSUMER] Consuming 2...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 3...
[CONSUMER] Consuming 3...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 4...
[CONSUMER] Consuming 4...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 5...
[CONSUMER] Consuming 5...
[PRODUCER] Consumer return: 200 OK
```

Python用迭代器实现协程

```
def gen_cal():  
    x = 1  
    y = 1  
    exp = None  
    while x < 256:  
        if exp == None:    # 这里接受发送的值  
            x, y = y, x+y  
            exp = yield y  
        else:  
            exp = yield (eval(exp))
```

一个有隐含功能的迭代器

```
gc = gen_cal()
```

```
print(next(gc))    # next其实会发送None  
print(next(gc))  
print(gc.send('23+9/3.0')) ←  
print(next(gc))
```

2

3

26.0

5



任务轮转调度：用协程实现

```
def task1():

    timeN = 12
    dur = 6

    while timeN > 0:
        timeN -= dur
        print('Task1 need:', timeN)
        ➡ yield timeN    # 中断

    print('Task 1 Finished')

def task2():

    timeN = 11
    dur = 3

    while timeN > 0:
        timeN -= dur
        print('Task2 need:', timeN)
        yield    # 只交出运行权 可以不返回值

    print('Task 2 Finished')
```

```
def RoundRobin(*task):

    ➡ t1s = list(task)

    while len(t1s) > 0:
        for p in t1s:
            try:
                next(p)
            except StopIteration:
                t1s.remove(p)

        print('All finished! 可以歇一会了')

t1 = task1()
t2 = task2()

RoundRobin(t1, t2)
```

这里可以输入控制参数

回调函数 (call back)

- 在函数中调用一个作为参数传入的函数

```
: def demo_callbcak(st):  
    print(st)  
  
    def caller(args, func):  
        print('Caller: Do something.')  
        func(args)  
  
caller(('I am callee'), demo_callbcak)
```

```
Caller: Do something.  
I am callee
```



回调函数方案实现函数的功能组合

```
def demo_handle(func, args, callback):  
    ➡ result = func(*args)  
    callback(result, func.__name__) # 参数: 计算结果 函数名  
  
def add(x, y):  
    return x + y  
  
def notify(result, frm):  
    print('Call fun {}() resule = {}'.format(frm, result))  
  
demo_handle(add, (3, 5), callback = notify)
```

— Call fun add() resule = 8

```
: def apply_async(func, args, *, callback): # 异步方式组装回调函数
    result = func(*args)
    callback.send(result) # 这里重新唤醒协程

def add(x, y):

    return x + y

def times(x, y):
    return x * y

def make_handler(): # 把函数包装为一个协程
    counter = 0
    while True:
        result = yield # 这里把自己阻塞，等待被调用才执行
        counter += 1
        print("counter = {} result: {}".format(counter, result)) # 可以假定这里是打印输出例程

handle = make_handler()
next(handle) # 这里初始化第一轮send, 到yield开始等待

apply_async(add, (3, 5), callback = handle)
apply_async(times, (3, 5), callback = handle)
```

```
counter = 1 result: 8
counter = 2 result: 15
```

```
def apply_handler(func, args, *, callback): #异步框架函数, 用来协调运行
    result = func(*args)
    callback(result) # 把要具体计算的任务交给一个回调函数

def add(x, y): # 计算函数
    return x + y

def times(x, y): # 也可以是一个IO之类的操作
    return x * y

def make_handler():
    counter = 0 # 计数器 记录总调用次数
    def handler(result): # 把要进行的任务包装一下, 加入日志、结果输出等
        nonlocal counter
        counter += 1
        print("counter = {} result: {}".format(counter, result))
    return handler

handler = make_handler() # 类似一个装饰器的功能, 可具备全局管理的功能
apply_handler(add, (2, 3), callback=handler)
apply_handler(times, (4, 6), callback=handler)
```

```
counter = 1 result: 5
counter = 2 result: 24
```



```
waiting_list = []
```

```
class Handle(object): # 这里可以对准备调度的任务进行包装, 如设置时间片大小、优先级、最大运行时间
```

```
    def __init__(self, gen):
```

```
        self.gen = gen
```

```
    def call(self):
```

```
        next(self.gen)
```

```
        # 被调用后做一些工作
```

```
        waiting_list.append(self)
```

```
        # 再把自己放回队列中
```

```
def RoundRobin(*tasks):
```

```
    waiting_list.extend(Handle(c) for c in tasks) # extend
```

```
    while waiting_list: # 如果队列不空则
```

```
        for p in waiting_list:
```

```
            try:
```

```
                p.call() # 启动一个任务
```

```
            except StopIteration:
```

```
                waiting_list.remove(p) # 从队列里删除任务
```

```
    print('All finished! 可以歇一会了')
```

```
if __name__ == "__main__":
```

```
    RoundRobin(task1(), task2())
```

轮转调度队列中的协程

```
def task1(): # 可以设置一个任务对象方便设置参数
```

```
    timeN = 12
```

```
    dur = 6
```

```
    while timeN > 0:
```

```
        timeN -= dur
```

```
        # 运行了一个给定的时间片
```

```
        print('Task1 need:', timeN)
```

```
        yield timeN
```

```
        # 自阻塞
```

```
    print('Task 1 Finished')
```

```
def task2():
```

```
    timeN = 11
```

```
    dur = 3
```

```
    while timeN > 0:
```

```
        timeN -= dur
```

```
        print('Task2 need:', timeN)
```

```
        yield
```

```
        # 可以不返回值只交出运行权
```

```
    print('Task 2 Finished')
```



```
waiting_list = []
```

```
class Handle(object): # 这里可以对准备调度的任务进行包装，如设置时间片大小、优先级、最大运行时间等
```

```
def __init__(self, gen, pri = 0.5):
    self.gen = gen
    self.timeSlice = 0
    self.timeNeed = 0
    self.pr = pri # 可以看作是优先级，pr<1 越高运行时间片越大
def call(self):
    try:
        if self.timeSlice == 0:
            self.timeNeed = next(self.gen) # 首次调用接受timeNeed
            self.timeSlice = int(self.timeNeed * self.pr)
        else:
            self.gen.send(self.timeSlice) # 运行，并设置下一个时间片长度

        waiting_list.append(self) # 再把自己放回队列中

    except StopIteration:
        print(self.gen.__name__, 'finished')
```

```
def RoundRobin(*tasks):
```

```
    waiting_list.extend(Handle(c) for c in tasks) # 加入被handle过的例程items
```

```
    while waiting_list: # 如果队列不空则
```

```
        p = waiting_list.pop(0) # 从队头弹出一个Handle
        p.call() # 启动一个任务
```

```
    print('All finished! 可以歇一会了')
```

```
RoundRobin(task1(), task2())
```

```
Task1 need: 12 Time slice = 3
Task2 need: 6 Time slice = 3
Task1 need: 9 Time slice = 4
Task2 need: 3 Time slice = 1
Task1 need: 5 Time slice = 4
Task2 need: 2 Time slice = 1
Task1 need: 1 Time slice = 4
Task2 need: 1 Time slice = 1
task1 finished
task2 finished
All finished! 可以歇一会了
```

```
waiting_list = []
```

```
class Handle(object): # 这里可以对准备调度的任务进行包装，如设置时间片大小、优先级、最大运行时间等
```

```
def __init__(self, gen, pri = 0.5):
    self.gen = gen
    self.timeSlice = 0
    self.timeNeed = 0
    self.pr = pri # 可以看作是优先级，pr<1 越高运行时间片越大
def call(self):
    try:
        if self.timeSlice == 0:
            self.timeNeed = next(self.gen) # 首次调用接受timeNeed
            self.timeSlice = int(self.timeNeed * self.pr)
        else:
            self.gen.send(self.timeSlice) # 运行，并设置下一个时间片长度

            waiting_list.append(self) # 再把自己放回队列中

    except StopIteration:
        print(self.gen.__name__, 'finished')
```

```
def RoundRobin(*tasks):
```

```
    waiting_list.extend(Handle(c) for c in tasks) # 加入被handle过的例程items
```

```
    while waiting_list: # 如果队列不空则
```

```
        p = waiting_list.pop(0) # 从队头弹出一个Handle
        p.call() # 启动一个任务
```

```
    print('All finished! 可以歇一会了')
```

```
RoundRobin(task1(), task2())
```

```
Task1 need: 12 Time slice = 3
Task2 need: 6 Time slice = 3
Task1 need: 9 Time slice = 4
Task2 need: 3 Time slice = 1
Task1 need: 5 Time slice = 4
Task2 need: 2 Time slice = 1
Task1 need: 1 Time slice = 4
Task2 need: 1 Time slice = 1
task1 finished
task2 finished
All finished! 可以歇一会了
```

Python用asyncio/await实现协程

- Python 3.5后 async/await 用于定义协程的关键字
 - 不用先next, 直接可以send调用
 - yield from替换为await

For any asyncio functionality to run on Jupyter Notebook you cannot invoke a `run_until_complete()`, since the loop you will receive from `asyncio.get_event_loop()` will be active. Instead, you must add task to the current loop.



```
%%writefile co-routine2.py
```

```
import asyncio
```

```
import time
```

```
async def say_after(delay, what): # 接受参数
```

定义异步awaitable object

```
    await asyncio.sleep(delay) ←
```

```
    print(what)
```

```
async def main():
```

```
    print(f"started at {time.strftime('%X')}")
```

```
    await say_after(1, 'hello') ← 在一个已经运行的协程中等待响应
```

```
    await say_after(2, 'world')
```

```
    print(f"finished at {time.strftime('%X')}")
```

```
asyncio.run(main()) ← 执行协程（创建一个事件循环） v3.7+
```

Writing co-routine2.py

started at 09:25:56

hello

world

finished at 09:25:59

```
# The asyncio.create_task() function to run coroutines concurrently as asyncio Tasks.
# 进一步参考: https://docs.python.org/3.7/library/asyncio-task.html
async def main():
    task1 = asyncio.create_task(
        say_after(3, 'hello'))

    task2 = asyncio.create_task(
        say_after(1, 'world'))

    print(f"started at {time.strftime('%X')}")

    # Wait until both tasks are completed (should take
    # around 2 seconds.)
    await task1
    await task2

    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())
```

```
started at 09:35:16
world
hello
finished at 09:35:19
```

```
import asyncio
import time

async def eternity():
    # Sleep for one hour
    await asyncio.sleep(3600)
    print('yay!')

async def main():
    # Wait for at most 1 second
    try:
        await asyncio.wait_for(eternity(), timeout=1.0) # 设定超时
    except asyncio.TimeoutError:
        print('timeout!')

asyncio.run(main())
```

timeout!



```
import asyncio
import datetime
import time
```

事件循环队列

```
def function_1(end_time, loop):
    print("function_1 called")
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, function_2, end_time, loop)
    else:
        loop.stop()

def function_2(end_time, loop):
    print("function_2 called ")
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, function_3, end_time, loop)
    else:
        loop.stop()

def function_3(end_time, loop):
    print("function_3 called")
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, function_1, end_time, loop)
    else:
        loop.stop()
```

```
loop = asyncio.get_event_loop() # 创建一个事件循环
```

```
end_loop = loop.time() + 9.0
```

```
loop.call_soon(function_4, end_loop, loop)
```

```
loop.run_forever() # 启动循环
```

```
loop.close()
```

```
def function_4(end_time, loop):
    print("function_4 called")
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, function_1, end_time, loop)
    else:
        loop.stop()
```

```
(base) C:\Users\hjfat\test>python test_event.py
function_1 called
function_2 called
function_3 called
function_1 called
function_2 called
function_3 called
function_1 called
function_2 called
function_3 called
```

```
(base) C:\Users\hjfat\test>python test_event.py
function_4 called
function_1 called
function_2 called
function_3 called
function_1 called
function_2 called
function_3 called
function_1 called
function_2 called
```



用事件循环队列实现生产者-消费者异步流程

```
import asyncio, time
```

```
async def consumer(q):
```

```
    print('consumer starts.')
```

```
    while True:
```

```
        → item = await q.get()
```

```
        if item is None:
```

```
            q.task_done() # Indicate that a formerly  
            break
```

```
        else:
```

```
            await asyncio.sleep(1) # take 1s to cons  
            print('consume %d' % item)  
            q.task_done() ←
```

```
    print('consumer ends.')
```

```
async def producer(q):
```

```
    print('producer starts.')
```

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

```
        await asyncio.sleep(1) # take 1s to produce
```

```
        print('produce %d' % i)
```

```
        → await q.put(i)
```

```
    await q.put(None)
```

```
    await q.join() # Block until all items in the queue have been gotten and
```

```
    print('producer ends.')
```

```
q = asyncio.Queue(maxsize=10)
```

```
t0 = time.time()
```

```
loop = asyncio.get_event_loop() ←
```

```
tasks = [producer(q), consumer(q)]
```

```
loop.run_until_complete(asyncio.wait(tasks)) ←
```

```
loop.close()
```

```
print(time.time() - t0, " s")
```

<https://zhuanlan.zhihu.com/p/476888052>

```
producer starts.
```

```
consumer starts.
```

```
produce 0
```

```
produce 1
```

```
consume 0
```

```
produce 2
```

```
consume 1
```

```
produce 3
```

```
consume 2
```

```
produce 4
```

```
consume 3
```

```
consume 4
```

```
consumer ends.
```

```
producer ends.
```

```
6.084010601043701 s
```


协程（routine）的使用场景

- 协程之间不是并发/并行的关系
- 协程在逻辑上倾向于一个功能独立的例程
- 可以被反复调用并在被调用过程中保持内部状态，直到异常中断或自行退出
- 常用于I/O通讯，资源管理与操作响应等



进程、线程、协程小结

- 进程是程序
- 线程是同环境下可并发过程
- 协程是带自休眠机制的可唤醒的伺服函数



Python GUI编程包Tkinter简介

Tkinter

Tkinter模块（接口）是Python的标准Tk GUI工具包的接口，速度较快，操作比较简单，可以在大多数平台下使用

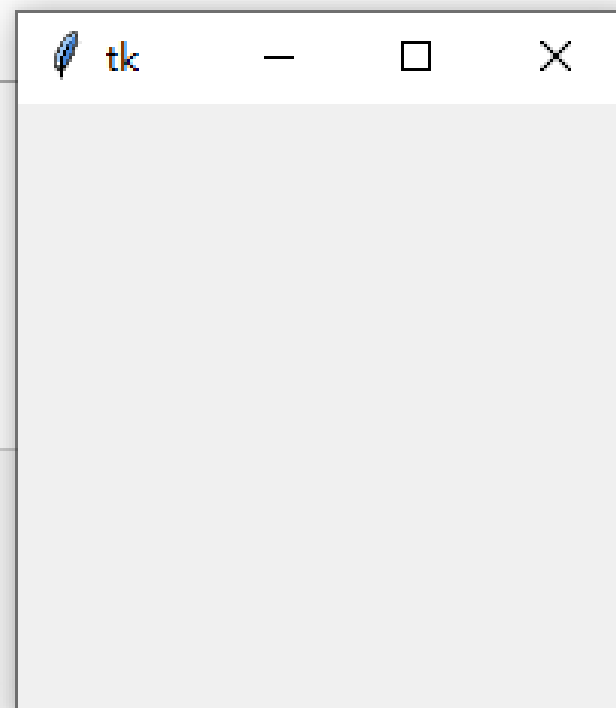
Tkinter Doc

[Python Tkinter Resources](#) [TkDocs](#) [Python Module](#)

简单例子

1.空窗口

```
#导入模块  
# import tkinter  
  
from tkinter import *  
top = Tk()  
top.mainloop()
```



基础控件（预定义GUI类）：

Basic Widgets

窗口，可以设置一下相关的参数，比如颜色位置以及一下动作，比如鼠标点击等

1.Frame

一般作为其他widgets的容器，长方形，一般使用几何布局

2.Label

可以现实图片文字等



一个Frame 和 Label结合的例子

```
import tkinter as tk
top = Tk()

#定义一个`label`显示`on the mainwindow`
tk.Label(top, text='on the window').pack()

###在`top`上创建一个`frame`
frm = tk.Frame(top)
frm.pack()
```

#在刚刚创建的`frame`上创建两个`frame`，可以理解成一个大容器里套了小容器，即`frm`上有两个`frame`，`frm_l`和`frm_r`

```
frm_l = tk.Frame(frm)
frm_r = tk.Frame(frm)
```

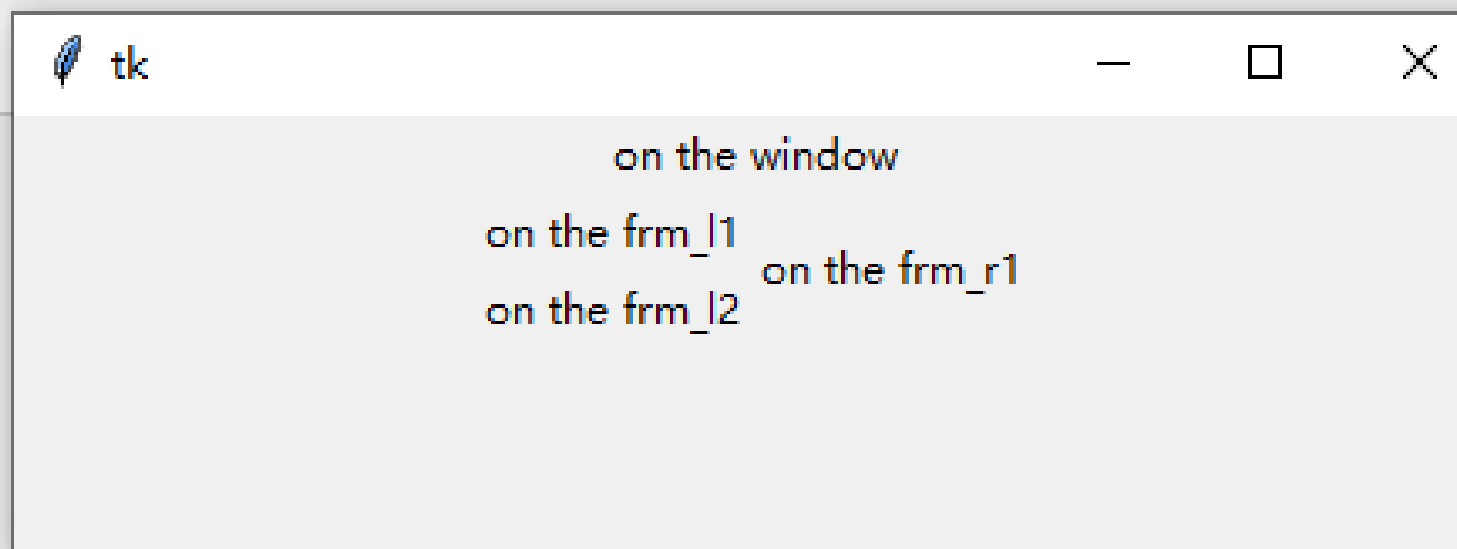
#这里是控制小的`frm`部件在大的`frm`的相对位置，此处`frm_l`就是在`frm`的左边，`frm_r`在`frm`的右边

```
frm_l.pack(side='left')
frm_r.pack(side='right')
```

#这里的三个label就是在我们创建的frame上定义的label部件，还是以容器理解，就是容器上贴了标签，来解释这个容器。

```
tk.Label(frm_l, text='on the frm_l1').pack()#这个`label`长在`frm_l`上，显示为`on the frm_l1`
tk.Label(frm_l, text='on the frm_l2').pack()#这个`label`长在`frm_l`上，显示为`on the frm_l2`
tk.Label(frm_r, text='on the frm_r1').pack()#这个`label`长在`frm_r`上，显示为`on the frm_r1`
```

```
top.mainloop()
```



Entry 控件（输入框）：

—— 用来接收字符串等输入的控件

```
from tkinter import *  
  
def show_entry_fields():  
    print("First Name: %s\nLast Name: %s" % (e1.get(), e2.get()))
```

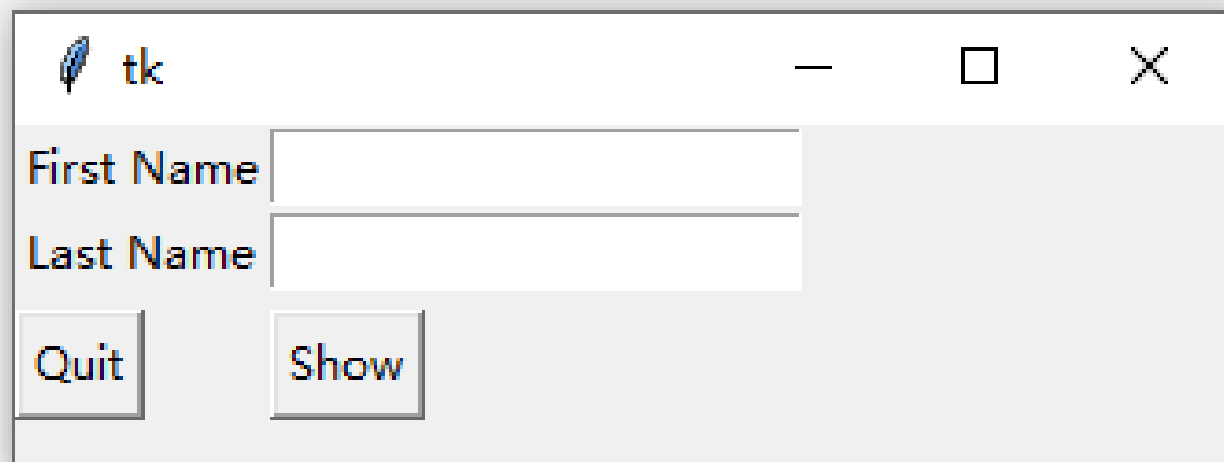
```
master = Tk()  
Label(master, text="First Name").grid(row=0)  
Label(master, text="Last Name").grid(row=1)
```

```
e1 = Entry(master)  
e2 = Entry(master)
```

```
e1.grid(row=0, column=1)  
e2.grid(row=1, column=1)
```

```
Button(master, text='Quit', command=master.quit).grid(row=3, column=0, sticky=W, pady=4)  
Button(master, text='Show', command=show_entry_fields).grid(row=3, column=1, sticky=W, pady=4)
```

```
mainloop( )
```

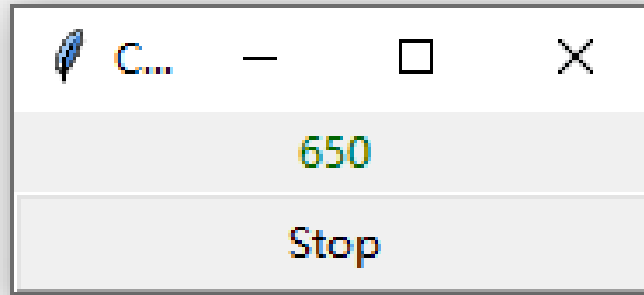


Button控件：

——用来展现不同样式的按钮，一般被用以和用户交互

```
counter = 0
def counter_label(label):
    counter = 0
    def count():
        global counter
        counter += 1
        label.config(text=str(counter))
        #after method, delay the time to call back the function
        label.after(100, count)
    count()

root = tk.Tk()
root.title("Counting Seconds")
label = tk.Label(root, fg="dark green")
label.pack()
counter_label(label)
button = tk.Button(root, text='Stop', width=25, command=root.destroy)
button.pack()
root.mainloop()
```



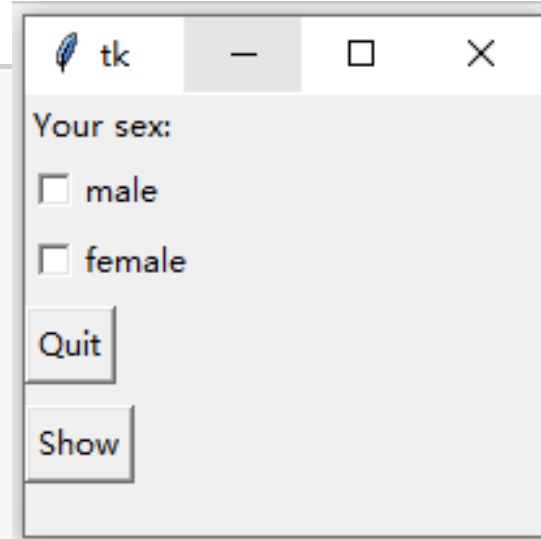
radiobutton & checkbox:

—— 单选按钮、复选框

```
from tkinter import *
master = Tk()

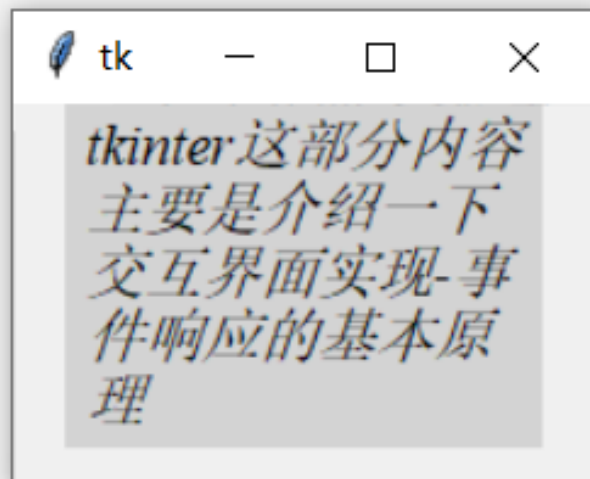
def var_states():
    print("male: %d, \nfemale: %d" % (var1.get(), var2.get()))

Label(master, text="Your sex:").grid(row=0, sticky=W)
var1 = IntVar()
Checkbutton(master, text="male", variable=var1).grid(row=1, sticky=W)
var2 = IntVar()
Checkbutton(master, text="female", variable=var2).grid(row=2, sticky=W)
Button(master, text='Quit', command=master.quit).grid(row=3, sticky=W, pady=4)
Button(master, text='Show', command=var_states).grid(row=4, sticky=W, pady=4)
mainloop()
```



Message控件： 用来展示一些文字短消息

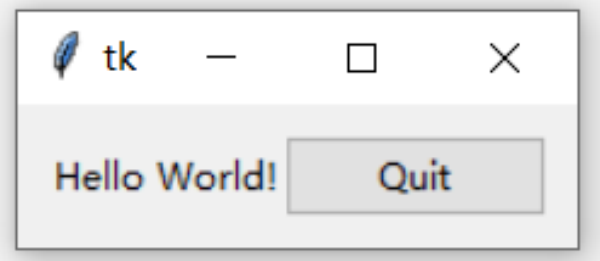
```
from tkinter import *
master = Tk()
whatever_you_do = "tkinter这部分内容主要是介绍一下交互界面实现-事件响应的基本原
msg = Message(master, text = whatever_you_do)
msg.config(bg='lightgrey', font=('times', 14, 'italic'))
msg.pack()
mainloop()
```



```
from tkinter import *  
top = Tk()  
top.mainloop()
```

<https://docs.python.org/zh-cn/3/library/tkinter.ttk.html>

```
from tkinter import *  
from tkinter import ttk # ttk支持样式分离  
root = Tk()  
frm = ttk.Frame(root, padding=10)  
frm.grid()  
ttk.Label(frm, text="Hello World!").grid(column=0, row=0)  
ttk.Button(frm, text="Quit", command=root.destroy).grid(column=1, row=0)  
root.mainloop()
```



```
from tkinter import *  
from tkinter.ttk import *
```

这段代码会让以下几个 tkinter.ttk 控件 (Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale 和 Scrollbar) 自动替换掉 Tk 的对应控件。

使用新控件的直接好处，是拥有更好的跨平台的外观，但新旧控件并不完全兼容。主要区别在于，Ttk 组件不再包含“fg”、“bg”等与样式相关的属性。而是用 ttk.Style 类来定义更美观的样式效果。



界面布局：

- **Grid布局：** 把整个窗口划分成格子
 - grid(row, column, rowspan, sticky.....)
- **Pack布局（流式布局）**
 - 将控件水平或垂直的逐个放在一起
- **Place 布局**
 - 指定控件的绝对位置或相对于其他控件的位置




Place布局:

```
import tkinter as tk
import random

root = tk.Tk()
# width x height + x_offset + y_offset:
root.geometry("170x200+30+30")

languages = ['Python', 'Perl', 'C++', 'Java', 'Tcl/Tk']
labels = range(5)
for i in range(5):
    ct = [random.randrange(256) for x in range(3)]
    brightness = int(round(0.299*ct[0] + 0.587*ct[1] + 0.114*ct[2]))
    ct_hex = "%02x%02x%02x" % tuple(ct)
    bg_colour = '#' + "".join(ct_hex)
    l = tk.Label(root,
                  text=languages[i],
                  fg='White' if brightness < 120 else 'Black',
                  bg=bg_colour)
    l.place(x = 20, y = 30 + i*30, width=120, height=25)

root.mainloop()
```



Event绑定机制

- 一个 Tkinter 应用生命周期中的大部分时间都处在一个消息循环 (event loop) 中. 它等待事件的发生: 事件可能是 按键按下, 鼠标点击, 鼠标移动 等
- Tkinter 提供了用以处理相关事件的机制. 处理函数可以被绑定给各个控件的各种事件. `widget.bind(event, handler)`
- 如果相关事件发生, `handler` 函数会被触发, 事件对象 `event` 会传递给 `handler` 函数. 点击/移动/按钮释放.....



```
from tkinter import *

def motion(event):
    print("Mouse position: (%s %s)" % (event.x, event.y))
    return

master = Tk()
whatever_you_do = "实现了有特色风格的作业可以跟助教打个招呼重点关注一下。"
msg = Message(master, text = whatever_you_do)
msg.config(bg='lightgreen', font=('times', 24, 'italic'))
msg.bind('<Motion>', motion)
msg.pack()
mainloop()
```

```
Mouse position: (286 170)
Mouse position: (268 152)
Mouse position: (262 147)
Mouse position: (248 136)
Mouse position: (245 132)
Mouse position: (242 130)
Mouse position: (238 125)
Mouse position: (238 123)
Mouse position: (237 123)
```



Python爬虫

4. 使用Cookie模拟登录

- Cookie

Cookie，可以简单认的为是在浏览器端记录包括登陆状态在内的各种属性值的容器名称，其实就是服务器为了保持浏览器与服务器之间连通状态，而在用户本地上创建的数据。只要用户再一次登陆，服务器会主动地寻找这些预存的数据，而无需再要求像第一次一样的操作。

- 使用Cookie的两种方式

- 将Cookie写在header头部

- 使用requests插入Cookie

- `cookie = {"Cookie": "xxxxxx"}`

- `html = requests.get(url, cookies=cookie)`



4. 使用Cookie模拟登录

- Cookie保持登录机制

前一次登录时，服务器发送了包含登录凭据(用户名和密码的某种加密形式)的Cookie到用户的硬盘中。再次登录时，如果Cookie尚未到期，则浏览器会发送该cookie，服务器验证凭据，于是不必输入用户名和密码就可以让用户登录



The screenshot shows the Weibo login page at login.weibo.cn/login/. It includes a navigation bar with links for [登录](#) (Login) and [注册](#) (Register). Below the navigation bar, there is a welcome message [欢迎访问微博](#) and a link [什么是微博?](#) (What is Weibo?). The main form area contains a label [手机号/电子邮箱/会员账号:](#) followed by a text input field. Below this is a label [密码:\(使用明文密码\)](#) followed by another text input field. A checkbox labeled ☒ [记住登录状态, 需支持并打开手机的cookie功能。](#) is present. At the bottom of the form, there are buttons for [登录](#) (Login) and [忘记密码](#) (Forgot Password). A section titled [小提示:](#) (Tips) contains two numbered items: 1、登录成功后保存任意页面为书签, 下次通过书签访问, 也可免去登录过程。 and 2、请不要直接通过手机浏览器的发送地址功能将登录后的页面地址发送给朋友, 以免泄露个人信息及密码。

图4-1. 微博登录实例

5. 使用多线程爬虫

- 使用multiprocessing模块

```
from multiprocessing.dummy import Pool  
results = pool.map(爬取函数, 网址列表)
```



6. 使用Selenium

- Selenium简介
 - Selenium可以用来模拟浏览器的运行，直接运行在浏览器中，可以让浏览器自动加载页面，获取需要的数据，甚至页面截图，或者判断页面上某些动作是否发生。



```
import requests
import time
import re
import json
import math

#目标网页地址, 如: raw_url = 'https://baijiahao.baidu.com/s?id=1746719319769828828&wfr=spider&for=pc'
raw_url = 'https://mbd.baidu.com/newspage/data/landingsuper?context=%7B%22nid%22%3A%22news_9513142528065658619%22%7D&n_type=-1&p_from=-1'

#设置协议头
headers = {
    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/106.0.0.0 Safari/537.36 Edg/106",
    "Connection": "close"
}

#提取前面的链接cookie值
res = requests.get(raw_url, headers = headers) # 按url获取网页
# print(list(dir(res)))
# print(res.text)

#提取cookie为字典形式
cookie = res.cookies.get_dict()

#for i in cookie.items():
#    print(i)

print('cookie=', cookie.values)

obj = re.compile(r'"tid\":"(?P<thread_id>.*?)\",""', re.S) # match='tid':"1034000052159141"', re.S 扩展到不含双引号的整个字符串
tidset = obj.finditer(res.text) # find tid 正则表达式返回迭代器

#thread id 后面情节评论的url中要用
for i in tidset:
    print(i)
    thread_id = i.group('thread_id')
res.close() # 关闭链接
```