


Python与数据科学导论-05

— 协程、网络通讯



胡俊峰 北京大学

2023-03-13




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
- 网络通讯与http协议

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



并发：

- 1、程序要同时处理多个任务
- 2、经常需要等待资源
- 3、黑板系统逻辑

- 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。

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.....

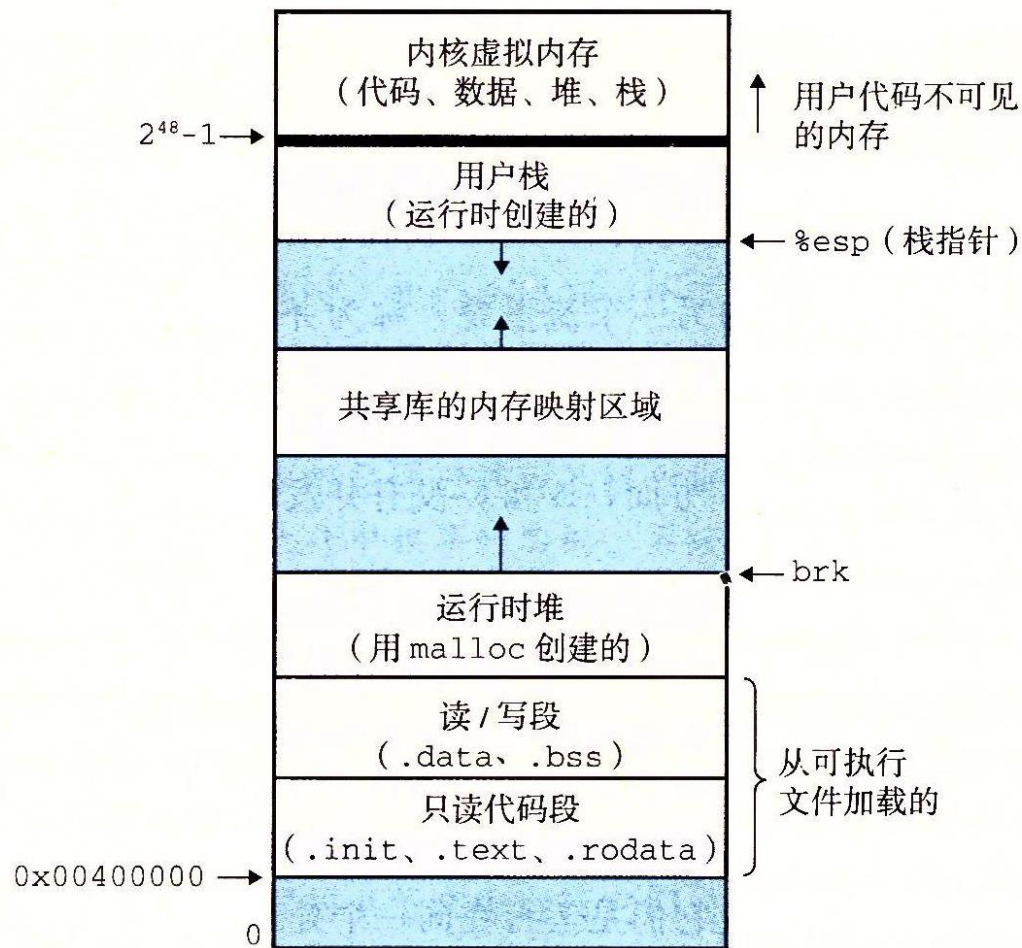


图 8-13 进程地址空间

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

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。

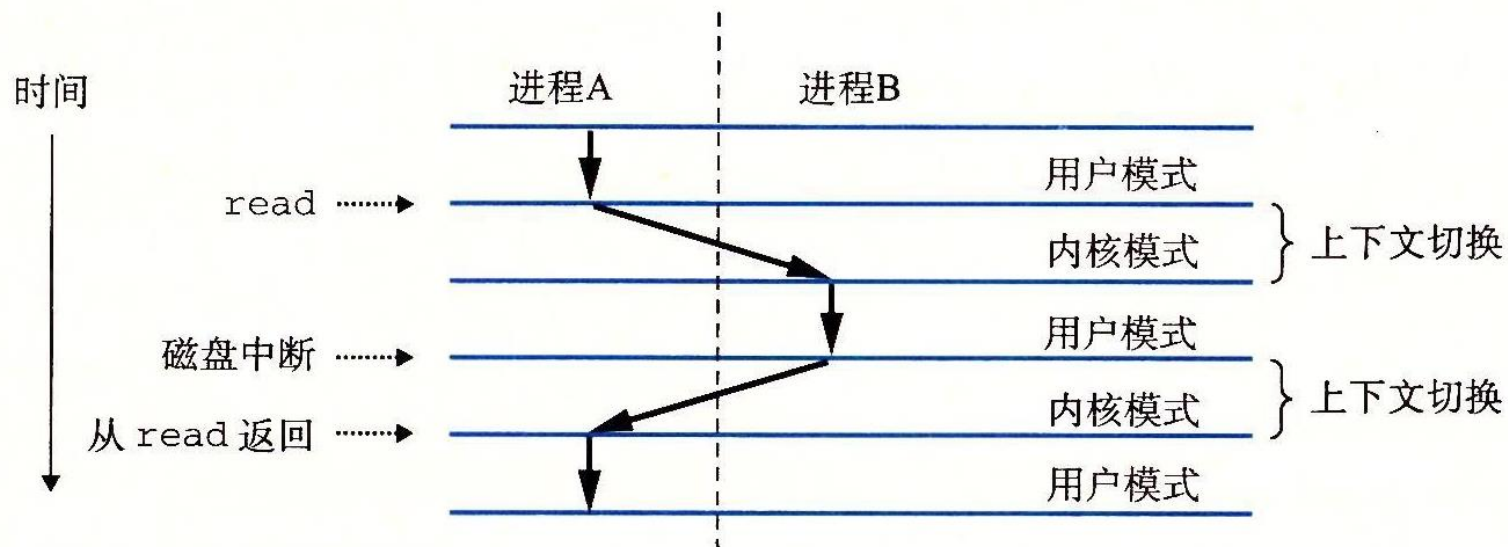


图 8-14 进程上下文切换的剖析

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。

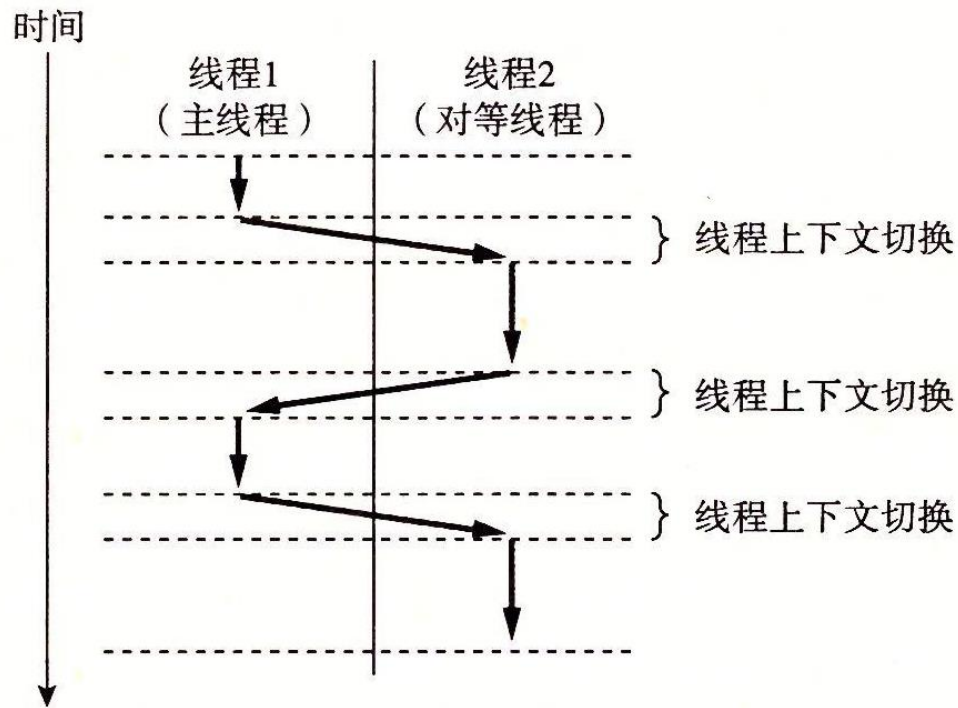


图 12-12 并发线程执行

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、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。

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]) | 主线程等待p终止 (是主线程处于等待的状态,而p是处于运行状态), timeout是可选的超时时间,需要强调的是, p.join只能join start开启的进程,而不能join run开启的进程 |

- ➡ 对主进程来说, 运行完毕指的是主进程代码运行完毕
- ➡ 对主线程来说, 运行完毕指的是主线程所在的进程内所有非守护线程统统运行完毕, 主线程才算运行完毕

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, 拿到空闲的进程才能继续执行。

➡ 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 <code>os.cpu_count ()</code> 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

| | |
|--|---|
| <code>p.apply(<u>func</u> [,args [,kwargs]])</code> | 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) |
| <code>p.apply_async(func [,args [,kwargs]])</code> | 在一个池工作进程中执行func(*args,**kwargs), 然后返回结果 (异步调用) |
| <code>p.close()</code> | 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 |
| <code>p.join()</code> | 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完成后才执行

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

| | |
|----------------------------------|---|
| <code>q.put (item)</code> | 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 |
| <code>q.put_nowait (item)</code> | 将item放入队列中, 如果当前队列已满, 不会阻塞, 但是会报错 |
| <code>q.get ()</code> | 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 |
| <code>q.get_nowait ()</code> | 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

```
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）

- 进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)
```

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 = 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

```
`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
```

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


```
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async/await实现协程

- Python 3.5后 async/await 用于定义协程的关键字
- async和await是针对coroutine设计的新语法：
 - @asyncio.coroutine替换为async（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awaitable object

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

```
    print(what)
```

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

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

```
    await say_after(1, 'hello') ←
```

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

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

```
asyncio.run(main()) ←
```

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!

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")
```

```
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 网络编程简介

什么是计算机网络？

- 计算机网络：用通信设备将计算机连接起来，在计算机之间传输数据（信息）的系统。
- 连网的计算机根据其提供的功能将之区分为客户机或服务器（C/S）
- 通信协议：计算机之间以及计算机与设备之间进行数据交换而遵守的规则、标准或约定
 - 典型的协议：TCP/IP（在互联网上采用），IEEE802.3以太网协议（局域网），IEEE902.11（无线局域网，WIFI）

以买火车票为例：

客户端：

发出查询请求，如果有则购买一张
票

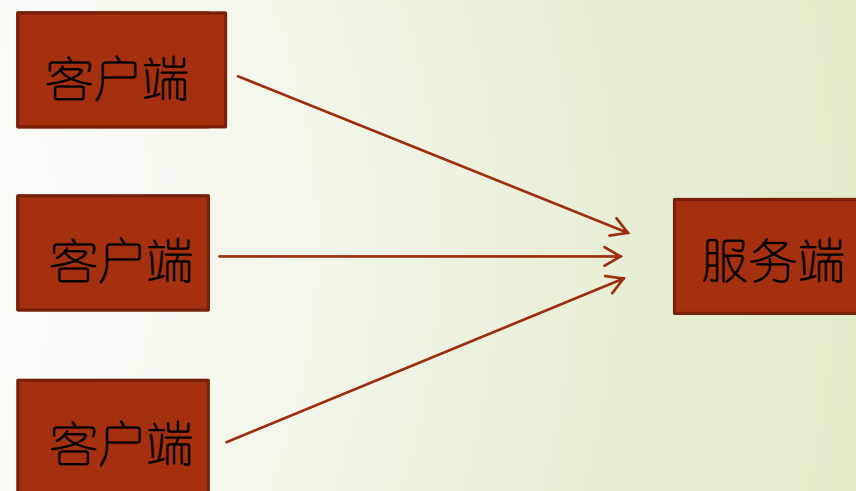
服务端：

维护余票情况，如果有余票则卖票
给客户端，余票数量减一；没有则
返回购买失败

B/S模式：Browser/Server，对C/S模式的改进。一部分事务逻辑在前端实现，但是主要事务逻辑在服务器端实现，和数据库端形成三层结构。

建立在广域网之上，只要有网络、浏览器，可以随时随地进行业务处理。

以12306 APP买票是C/S服务模式，用12306网页购票是B/S模式。



Socket通信

套接字socket：网络中不同主机上的应用进程之间进行双向通信的端点。


每台主机有一个唯一的主机地址标识（IP），同时主机内还有标识服务的序号id，称作端口（port）。

socket绑定了相应的**IP**和**port**，可以用（**IP : port**）的形式表示一个**socket**地址。

当客户端发起一个连接请求时，客户端socket地址中的端口由系统自动分配，服务器端套接字地址中的端口通常是某个和服务相对应的知名端口。（例如Web服务器常使用端口80，电子邮件服务器使用端口25）

一个连接由它两端的socket地址唯一确定：

（ClientIP : ClientPort, ServerIP : ServerPort）



信息：需要寄的快递

IP：小区地址

Port：门牌号,共有65536个端口

Socket：快递地址（小区+门牌号）

TCP，UCP等协议：快递公司

利用socket发送消息：把快递（消息）放到门口（socket），由快递公司（TCP等协议）负责送到对应的地址（对方socket）

传输层控制协议

- TCP：传输控制协议，面向连接、可靠。适用于要求可靠传输的应用。

面向连接：发送数据之前必须在两端建立连接。

仅支持单播传输：只能进行点对点数据传输。

面向字节流：在不保留报文边界的情况下以字节流的方式进行传输。

可靠：对每个包赋予序号，来判断是否出现丢包、误码。

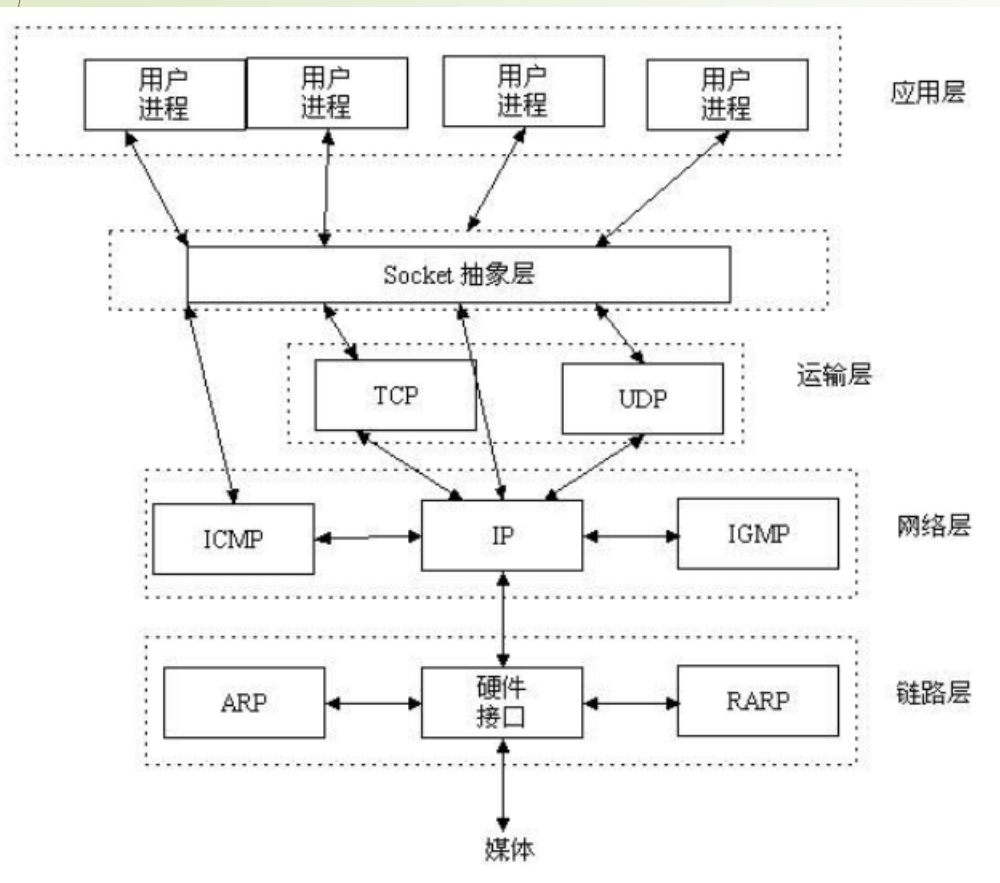
- UDP：用户数据报协议，面向非连接、不可靠。适用于实时应用。

面向非连接：发送数据不需要建立连接。

支持单播、多播、广播

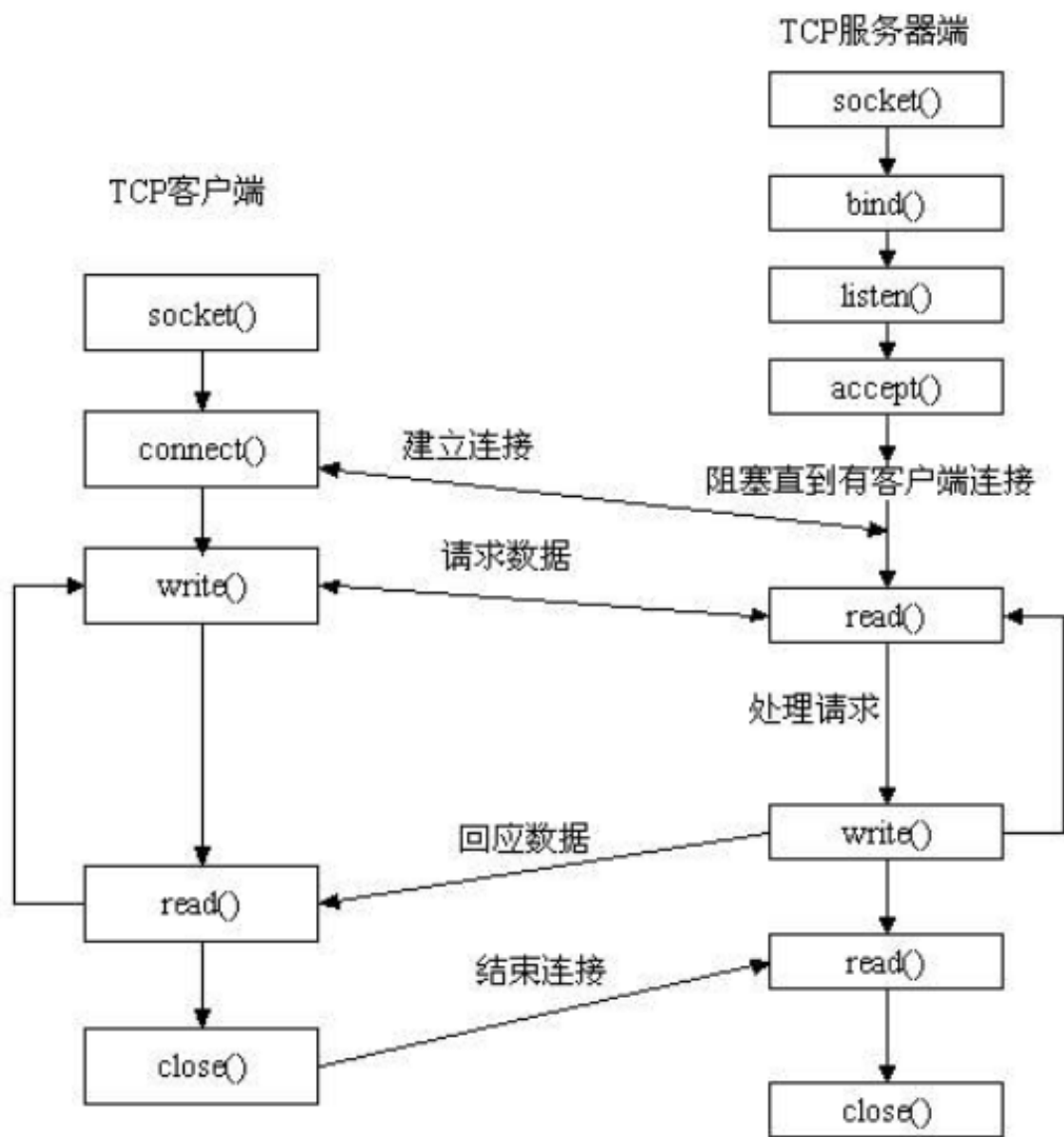
面向报文：对应用层的报文添加首部后直接向下层交付。

不可靠：没有拥塞控制，不会调整发送速率。



Socket是传输层和应用层之间的软件抽象层，是一组接口。

对于用户来说，socket把复杂的TCP/IP协议族隐藏在接口后，只需要遵循socket的规范，就能得到遵循TCP/UDP标准的程序。



服务器端：

初始化socket，与IP端口绑定，对IP端口进行监听，调用accept()阻塞，等待客户端连接。

客户端：

初始化socket，连接服务器。

连接成功后客户端发送数据请求，服务器端接收并处理请求、回应数据，客户端读取数据。

最后关闭连接，一次交互结束。

服务器端方法

s.bind()

绑定地址 (host,port) 到套接字, 在AF_INET下,以元组 (host,port) 的形式表示地址。

s.listen(backlog)

开始监听。backlog指定在拒绝连接之前, 操作系统可以挂起的最大连接数量。该值至少为1, 大部分应用程序设为5就可以了。

s.accept()

被动接受客户端连接,(阻塞式)等待连接的到来, 并返回 (conn,address) 二元元组,其中conn是一个通信对象, 可以用来接收和发送数据。address是连接客户端的地址。

客户端方法

s.connect(address)

客户端向服务端发起连接。一般address的格式为元组 (hostname,port) , 如果连接出错, 返回socket.error错误。

s.connect_ex()

connect()函数的扩展版本,出错时返回出错码,而不是抛出异常

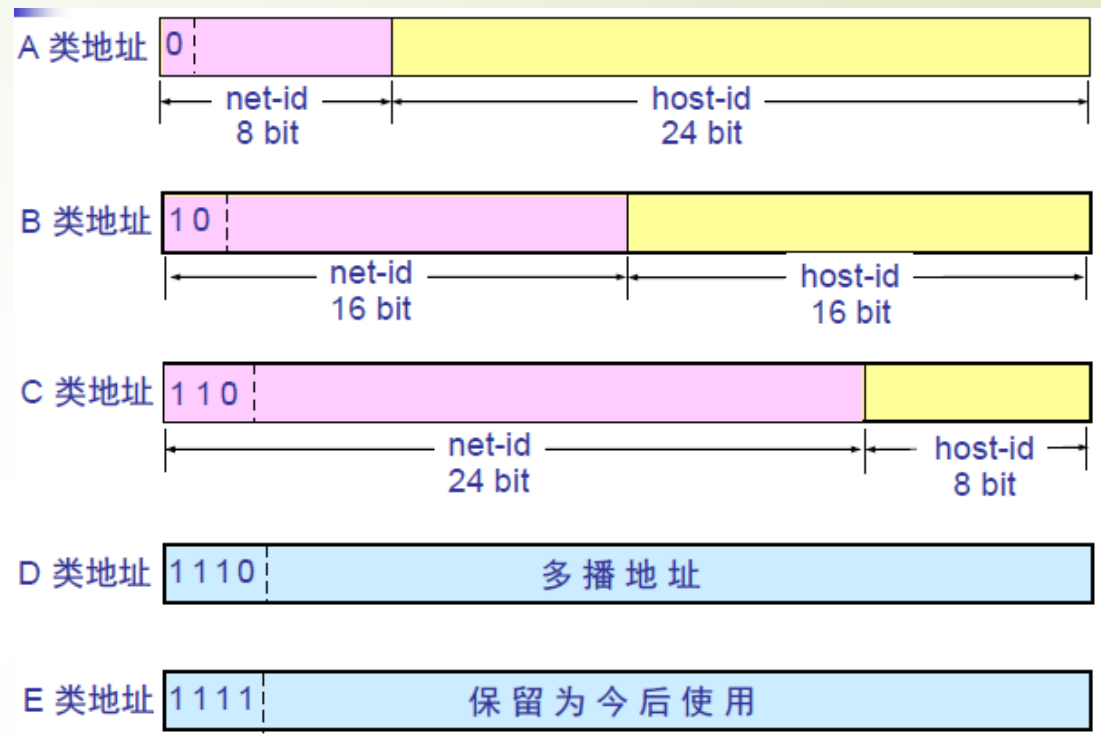
| | |
|--------------------------------------|---|
| s.recv(bufsize) | 接收数据，数据以bytes类型返回，bufsize指定要接收的最大数据量。 |
| s.send() | 发送数据。返回值是要发送的字节数量。 |
| s.sendall() | 完整发送数据。将数据发送到连接的套接字，但在返回之前会尝试发送所有数据。成功返回None，失败则抛出异常。 |
| s.recvfrom() s.recvfrom() | 接收UDP数据，与recv()类似，但返回值是（data,address）。其中data是包含接收的数据，address是发送数据的套接字地址。 |
| s.sendto(data,address) | 发送UDP数据，将数据data发送到套接字，address是形式为（ipaddr, port）的元组，指定远程地址。返回值是发送的字节数。 |
| s.close() | 关闭套接字，必须执行。 |
| s.getpeername() | 返回连接套接字的远程地址。返回值通常是元组（ipaddr,port）。 |
| s.getsockname() | 返回套接字自己的地址。通常是一个元组(ipaddr,port) |
| s.setsockopt(level,optname,value) | 设置给定套接字选项的值。 |
| s.getsockopt(level,optname[.buflen]) | 返回套接字选项的值。 |
| s.settimeout(timeout) | 设置套接字操作的超时期，timeout是一个浮点数，单位是秒。值为None表示没有超时期。一般，超时期应该在刚创建套接字时设置，因为它们可能用于连接的操作（如connect()） |

- IP地址：IPv4 – 32位，IPv6 – 128位
- IP地址分类：每个地址由两个固定长度的字段组成，网络号net-id标志主机所连接到的网络，主机号host-id标志该主机。

127.0.0.1和0.0.0.0的区别：

回环地址127.x.x.x：该范围内的任何地址都将环回到本地主机中，不会出现在任何网络中。主要用来做回环测试。

0.0.0.0：任何地址，包括了环回地址。不管主机有多少个网口，多少个IP，如果监听本机的0.0.0.0上的端口，就等于监听机器上的所有IP端口。数据报的目的地址只要是机器上的一个IP地址，就能被接受。



单线程服务端

```
import socket
import time
# 定义服务器信息
print('初始化服务器主机信息')
port = 5002 #端口 0-1024 为系统保留
host = '0.0.0.0'
address = (host, port)
# 创建TCP服务socket对象
print("初始化服务器主机套接字对象.....")
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# 关掉连接释放掉相应的端口
# server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
# 绑定主机信息
print('绑定的主机信息.....')
server.bind(address)
# 启动服务器 一个只能接受一个客户端请求, 可以有1个请求排队
print("开始启动服务器.....")
server.listen(5)
#等待连接
while True:
    # 等待来自客户端的连接
    print('等待客户端连接')
    conn, addr = server.accept() # 等电话
    print('连接的客服端套接字对象为: {}\n客服端的IP地址 (拨进电话号码): {}'.format(conn, addr))
    #发送给客户端的数据
    conn.send("欢迎访问服务器".encode('utf-8'))
    time.sleep(100)
    conn.close()
```



```
# -*- coding: utf-8 -*-
import socket # 导入 socket 模块

port = 5002
hostname = '127.0.0.1'

client = socket.socket() # 创建 socket 对象
client.connect((hostname, port))
data = client.recv(100).decode('utf-8')
print(data)

client.close()
```

服务端输出:

初始化服务器主机信息

初始化服务器主机套接字对象.....

绑定的主机信息.....

开始启动服务器.....

等待客户端连接

连接的客户端套接字对象为: <socket.socket fd=1092, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 5002), raddr=('127.0.0.1', 60984)>

客服端的IP地址 (拨进电话号码) : ('127.0.0.1', 60984)

客户端输出:

```
$ python client.py
欢迎访问服务器
```


多线程服务端

```
import socket # 导入 socket 模块
from threading import Thread
import time

def link_handler(link, client):
    link.send("欢迎访问服务器".encode('utf-8'))
    time.sleep(10)
    print('关闭客服端')
    link.close()

print('初始化服务器主机信息')
port = 5002
host = '0.0.0.0'
address = (host, port)
print("初始化服务器主机套接字对象.....")
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
print('绑定的主机信息.....')
server.bind(address)
print("开始启动服务器.....")
server.listen(1)
while True:
    print('等待客户端连接')
    conn, addr = server.accept() # 等电话
    print('连接的客服端套接字对象为: {}\n客服端的IP地址 (拨进电话号码) : {}'.format(conn, addr))
    t = Thread(target=link_handler, args=(conn, address))
    t.start()
```

单进程服务端模拟购票

```
# -*- coding: utf-8 -*-
```

```
import socket
import time
```

```
port = 5002
host = '0.0.0.0'
```

```
ticket_num = 2
```

```
def buy_ticket(conn):
```

```
    if_bought = 0
```

```
    global ticket_num
```

```
    if ticket_num > 0:
```

```
        ticket_num -= 1
```

```
        if_bought = 1
```

```
    # 模拟信号传输时间
```

```
    time.sleep(5)
```

```
    conn.send((str(ticket_num) + str(if_bought)).encode('utf-8'))
```

```
    conn.close()
```

```
# 定义服务器信息
```

```
print('初始化服务器主机信息')
```

```
address = (host, port)
```

```
# 创建TCP服务socket对象
```

```
print("初始化服务器主机套接字对象.....")
```

```
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
# 关掉连接释放掉相应的端口
```

```
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

```
# 绑定主机信息
```

```
print('绑定的主机信息.....')
```

```
server.bind(address)
```

```
# 启动服务器 一个只能接受一个客户端请求，可以有1个请求排队
```

```
print("开始启动服务器.....")
```

```
server.listen(5)
```

```
#等待连接
```

```
while True:
```

```
    # 等待来自客户端的连接
```

```
    print('等待客户端连接')
```

```
    conn, addr = server.accept() # 等电话
```

```
    print('连接的客服端套接字对象为: {}\n客服端的IP地址 (拨进电话号码) : {}'.format(conn, addr))
```

```
    buy_ticket(conn)
```

弊端：顺序，一个客户端堵塞会影响其余客户端

客户端

```
▶ #-*- coding: utf-8 -*-  
import socket # 导入 socket 模块  
  
port = 5002  
hostname = '127.0.0.1'  
  
client = socket.socket() # 创建 socket 对象  
client.connect((hostname, port))  
data = client.recv(100).decode('utf-8')  
ticket_num, if_bought = int(data[:-1]), int(data[-1])  
if not if_bought:  
    print(f'现在还剩下{ticket_num}张票, 客户端1没有买到票')  
else:  
    print(f'现在还剩下{ticket_num}张票, 客户端1成功买到了一张票')  
client.close()
```

多进程服务端

```
# -*- coding: utf-8 -*-
```

```
import socket
import time
from multiprocessing import Lock, Process, Value
```

```
port = 5002
host = '0.0.0.0'
```

```
def buy_ticket(conn, ticket_num, lock):
    lock.acquire()
    if_bought = 0
    if ticket_num.value > 0:
        ticket_num.value -= 1
        if_bought = 1
    lock.release()
    # 模拟信号传输时间
    time.sleep(5)
    conn.send((str(ticket_num.value) + str(if_bought)).encode('utf-8'))
    conn.close()
```

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

```
    l = Lock() # 实例化一个锁对象
    ticket_num = Value("i", 2)
```

```
    # 定义服务器信息
```

```
    print('初始化服务器主机信息')
```

```
    address = (host, port)
```

```
    # 创建TCP服务socket对象
```

```
    print("初始化服务器主机套接字对象.....")
```

```
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
    # 关掉连接释放掉相应的端口
```

```
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

```
    # 绑定主机信息
```

```
    print('绑定的主机信息.....')
```

```
    server.bind(address)
```

```
    # 启动服务器 一个只能接受一个客户端请求, 可以有1个请求排队
```

```
    print("开始启动服务器.....")
```

```
    server.listen(5)
```

```
    #等待连接
```

```
    while True:
```

```
        # 等待来自客户端的连接
```

```
        print('等待客户端连接')
```

```
        conn, addr = server.accept()
```

```
        print('连接的客服端套接字对象为: {} \n客服端的IP地址 (拨进电话号码) : {}'.format(conn, addr))
```

```
        p = Process(target=buy_ticket, args=(conn, ticket_num, l))
```

```
        p.start()
```

