


Python与数据科学导论-05

— 协程、网络通讯

胡俊峰 2022-03-14

北京大学计算机学院



内容提要

- 多进程运行与交互（补充）
- 协程与任务分时轮转调度
- 网络通讯与http协议
- TkInter交互式界面编程*

多进程在notebook环境下运行受限

```
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()      # 不阻塞主进程
    print("parent process done!")
```

parent process done!

```
C:\Users\hujf\2021Python\demo-code>python test4.py
child process done!
parent process done!
```

```
C:\Users\hujf\2021Python\demo-code>python test4-2.py
parent process done!
child process done!
```

在notebook环境守护进程派生子进程受限

利用队列进行进程同步

```
import time
from multiprocessing import Process, Queue

def func1(q):
    q.put([42, None, 'hello'])
    ➡ time.sleep(2)
    q.put("happy new Monday!")

if __name__ == '__main__': # 避免被自动包含到子进程里 Win
    q = Queue()
    p = Process(target=func1, args=(q,)) # 传入参数 iterable seq
    p.start()
    print(q.get())
    print(q.get()) # 这里
    print("the end.")
    p.join()
```

```
C:\Users\hujf\2021Python\demo-code>python test5-2.py
[42, None, 'hello']
happy new Monday!
the end.
```

```
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.
```

#使用Value, Array共享内存

```
from multiprocessing import Process, Value, Array
```

```
def f1(n, a):  
    n.value = 3  
    for i in range(len(a)):  
        a[i] = a[i]+1
```

```
def f2(n, a):  
    n.value = 1  
    for i in range(len(a)):  
        a[i] = a[i]-2
```

```
if __name__ == '__main__':  
    num = Value('d', 0.0)      #声明两个可共享的对象  
    arr = Array('i', range(10)) # 初始化一个数组
```

```
p1 = Process(target=f1, args=(num, arr)) # 作为参数传给进程  
p2 = Process(target=f2, args=(num, arr)) # 作为参数传给进程
```

```
p1.start()  
p2.start()  
p1.join()  
p2.join()
```

```
print(num.value)  
print(arr[:])
```

```
C:\Users\hujf\2021Python\demo-code>python test9-1.py
```

```
3.0  
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 8]
```

```
C:\Users\hujf\2021Python\demo-code>python test9-1.py
```

```
1.0  
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 8]
```

上下文和启动方法

根据不同的平台，`multiprocessing` 支持三种启动进程的方法。这些 *启动方法* 有

spawn

父进程会启动一个全新的 python 解释器进程。子进程将只继承那些运行进程对象的 `run()` 方法所必需的资源。特别地，来自父进程的非必需文件描述符和句柄将不会被继承。使用此方法启动进程相比使用 *fork* 或 *forkserver* 要慢上许多。

可在Unix和Windows上使用。Windows上的默认设置。

fork

父进程使用 `os.fork()` 来产生 Python 解释器分叉。子进程在开始时实际上与父进程相同。父进程的所有资源都由子进程继承。请注意，安全分叉多线程进程是棘手的。

只存在于Unix。Unix中的默认值。

forkserver

程序启动并选择* *forkserver* * 启动方法时，将启动服务器进程。从那时起，每当需要一个新进程时，父进程就会连接到服务器并请求它分叉一个新进程。分叉服务器进程是单线程的，因此使用 `os.fork()` 是安全的。没有不必要的资源被继承。

可在Unix平台上使用，支持通过Unix管道传递文件描述符。


```
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")
```

```
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]
```


协程（回顾和继续上次课的内容）

- 存在自阻塞操作（语句）的被动服务方
- 发出调用操作并等待调用返回才继续执行的主动客户方
- 线程本质上是多个协同操作的函数共处于一个程序运行环境下
- 其调度由用户程序（而非系统内核）自己来控制

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

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
```

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_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
```

Handler (句柄)

```
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
```



```
: 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
```

```
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
```

Python用async/await实现协程

- Python 3.5后 async/await 用于定义协程的关键字

Python 3.6

Generator-based coroutines should be decorated with `@asyncio.coroutine`, although this is not strictly enforced. The decorator enables compatibility with `async def` coroutines, and also serves as documentation. Generator-based coroutines use the `yield from` syntax introduced in PEP 380, instead of the original `yield` syntax.

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
    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!

async/await、事件循环event loop

```
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
```

用事件循环队列实现生产者-消费者协程

```
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 enqueued task is done
            break
        else:
            await asyncio.sleep(1) # take 1s to consume
            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 processed
    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

- 根据问题的特点选用合适的编程方式
 - 考虑通信方式
 - 计算密集/IO密集
- 计算密集型：往往用多进程（每个进程只完成比较独立的一部分）
- IO密集型：多线程、协程
- 协程及可等待对象（awaitable object）是目前python语言发展快速的一个技术分支



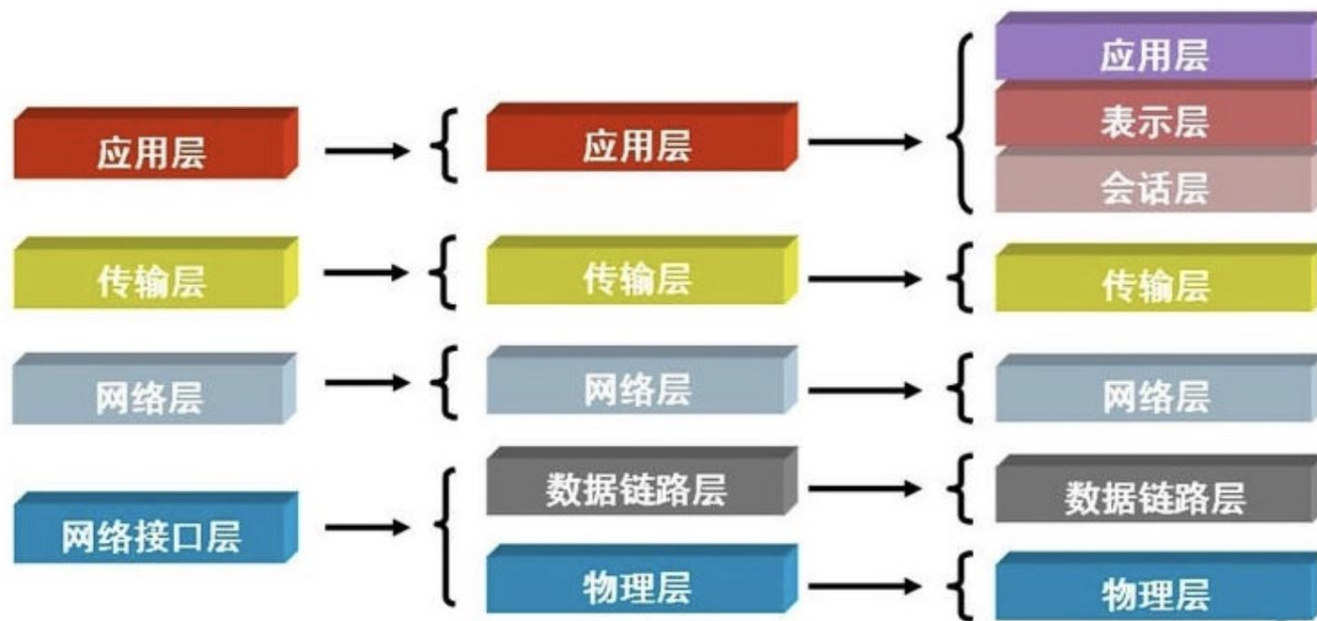
Python 网络编程介绍

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）

开放系统互连模型OSI

- 物理层：利用传输介质为数据链路层提供物理连接，实现比特流的透明传输。
- 数据链路层：将比特组成数据帧，临近网络部件间的数据传送
- 网络层：源到目的之间基于分组的数据交换
- 传输层：提供端对端的透明数据传输服务
- 会话层：不同主机的进程间会话的组织 and 同步
- 表示层：为上层用户提供共同需要的数据或信息语法表示及转换
- 应用层：为用户提供服务，提供网络应用



客户端-服务器模型

- 一个应用由一个服务器进程和一个或多个客户端进程组成。
- 服务器管理某种资源，并且通过操作这种资源来为它的客户端提供某种服务。

1. 客户端向服务器发送一个请求。

Web浏览器需要文件时，发送请求给Web服务器。

2. 服务器收到请求后，解释它，并以适当的方式操作资源。

Web服务器收到浏览器发出的请求后，读一个磁盘文件。

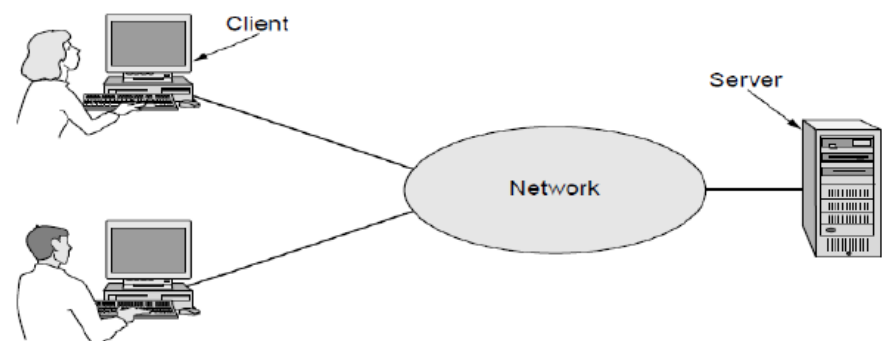
3. 服务器给客户端发送一个响应，并等待下一个请求。

Web服务器将文件发送回客户端。

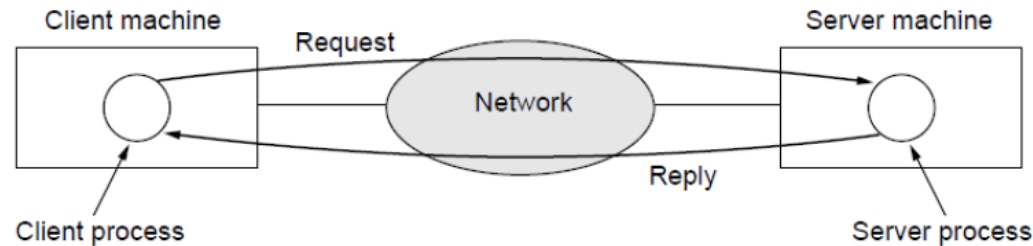
4. 客户端收到响应并处理它。

Web浏览器收到来自服务器的一页后，在屏幕上显示此页。

有两个客户机一个服务器的网络



在客户机服务器模式中包含请求和响应



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

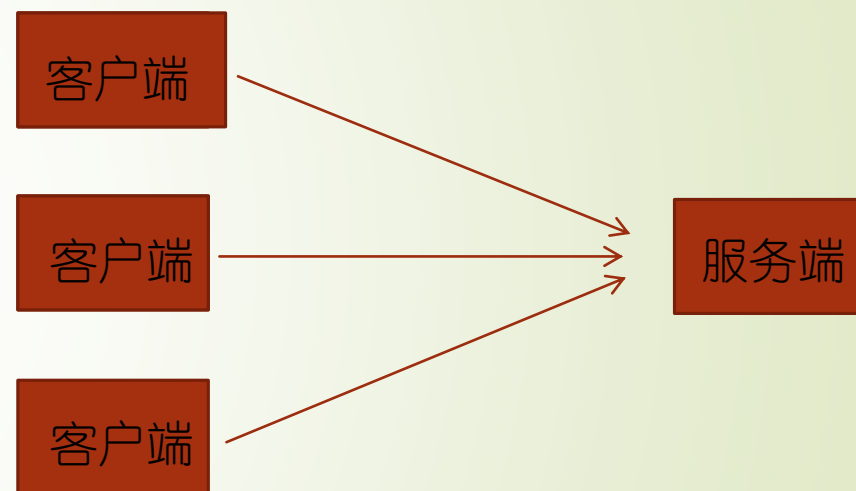
服务端：

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

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。

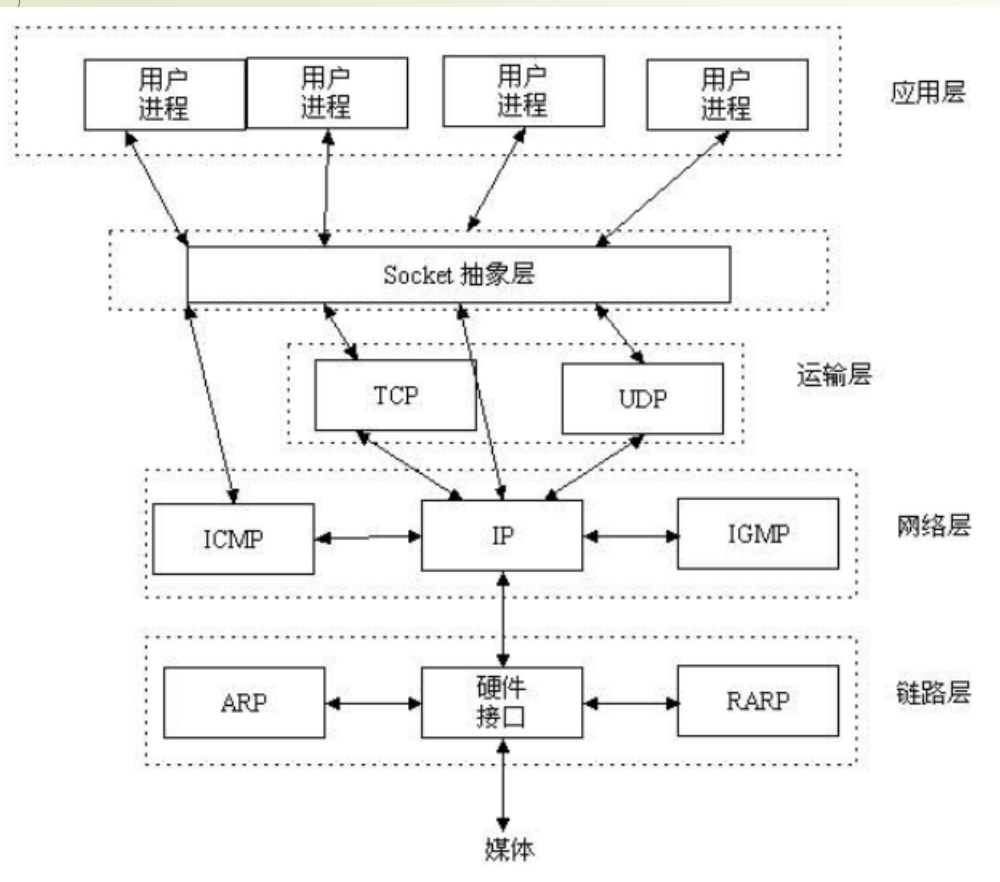
- UDP：用户数据报协议，面向非连接、不可靠。适用于实时应用。

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。

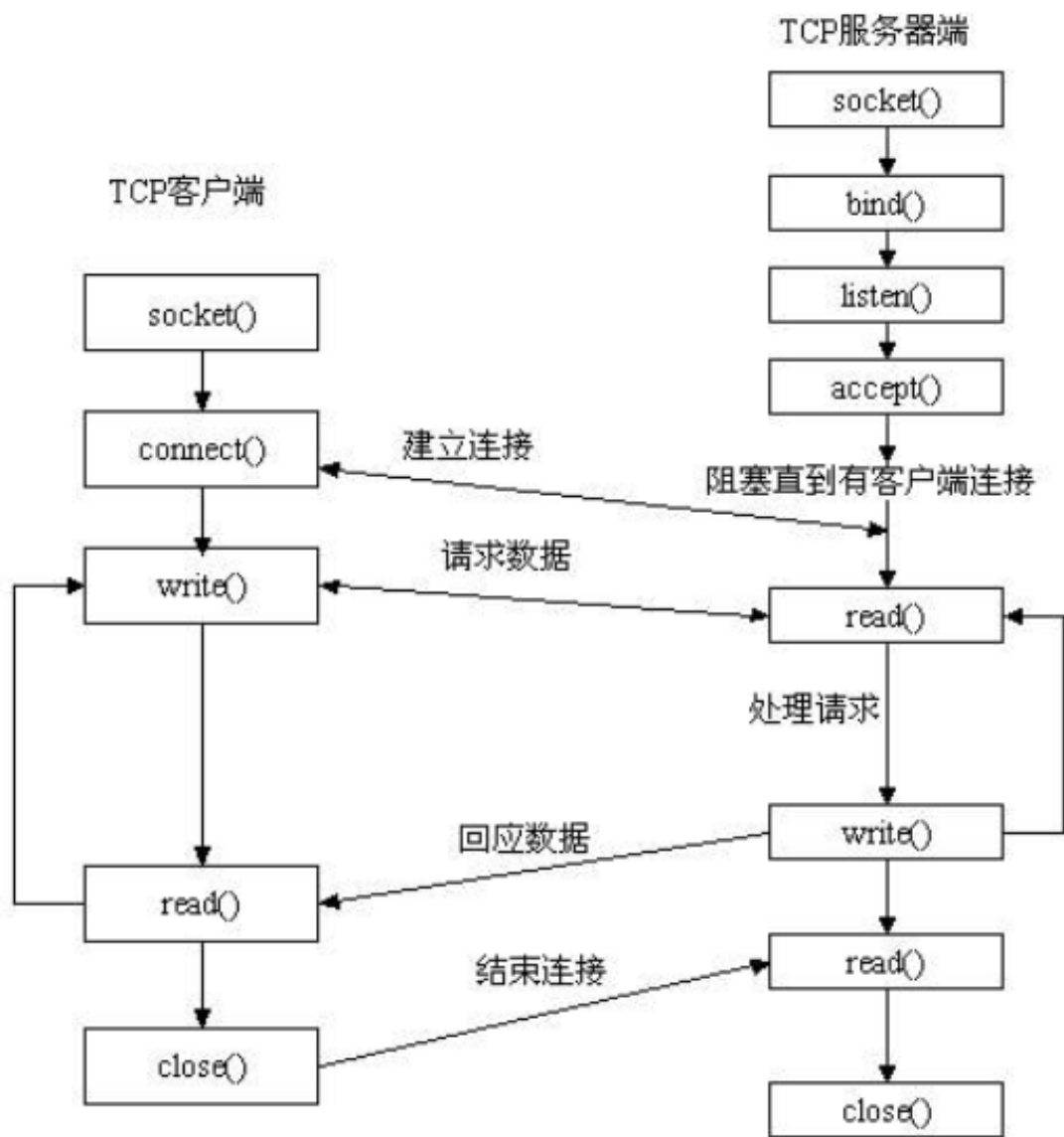
创建套接字: `socket.socket(family, type)`

参数说明:

family: 套接字家族, 可以使AF_UNIX或者AF_INET, 一般是AF_INET。

type: 套接字类型, 根据是面向连接的还是非连接分为SOCK_STREAM或SOCK_DGRAM, 也就是TCP和UDP的区别, 一般是SOCK_STREAM。

socket类型	描述
socket.AF_UNIX	只能够用于单一的Unix系统进程间通信
socket.AF_INET	IPv4
socket.AF_INET6	IPv6
socket.SOCK_STREAM	流式socket , for TCP
socket.SOCK_DGRAM	数据报式socket , for UDP
socket.SOCK_RAW	原始套接字, 普通的套接字无法处理ICMP、IGMP等网络报文, 而SOCK_RAW可以; 其次, SOCK_RAW也可以处理特殊的IPv4报文; 此外, 利用原始套接字, 可以通过IP_HDRINCL套接字选项由用户构造IP头。
socket.SOCK_SEQPACKET	可靠的连续数据包服务
创建TCP Socket:	<code>s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)</code>
创建UDP Socket:	<code>s=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)</code>



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

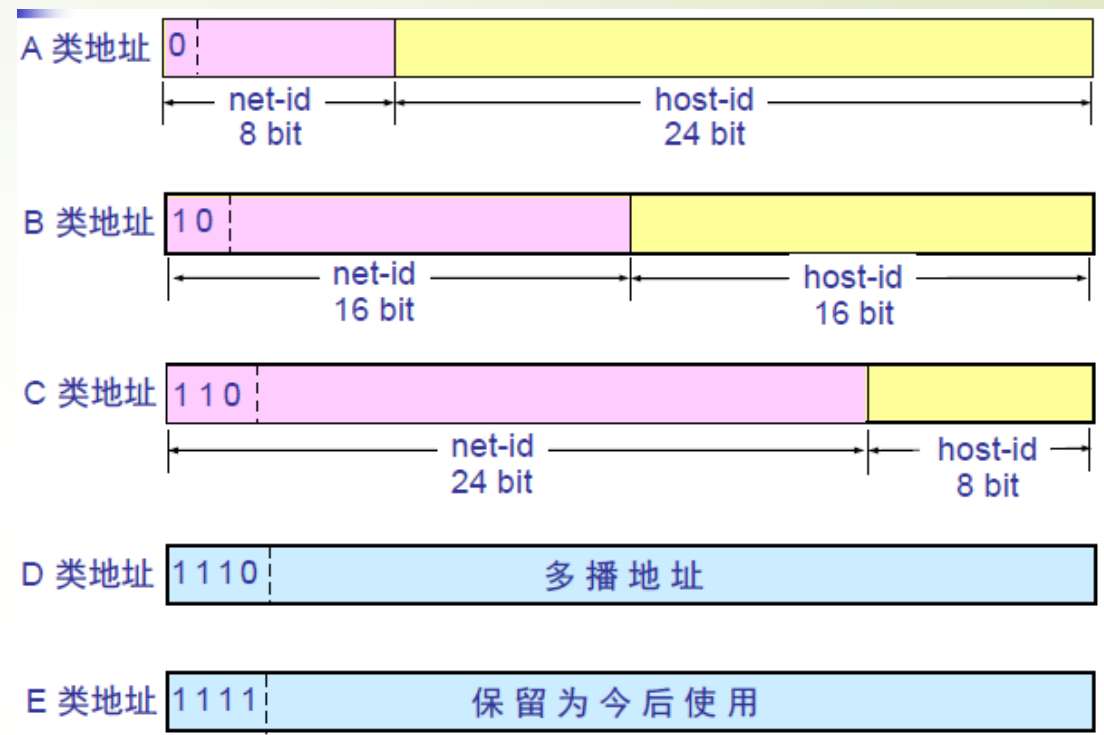
s.recv(bufsize)	接收数据，数据以bytes类型返回，bufsize指定要接收的最大数据量。
s.send()	发送数据。返回值是要发送的字节数量。
s.sendall()	完整发送数据。将数据发送到连接的套接字，但在返回之前会尝试发送所有数据。成功返回None，失败则抛出异常。
s.recvfrom() s.recvfrom()	接收UDP数据，与recv()类似，但返回值是（data,address）。其中data是包含接收的数据，address是发送数据的套接字地址。
s.sendto(data,address)	发送UDP数据，将数据data发送到套接字，address是形式为（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()
```



socket是python用于网络编程的基础

利用socket为基础在各个子领域构建起更灵活的框架。

以web服务为例：

Flask:web开发框架

Gunicorn:多进程管理

nginx:负载均衡

正则表达式

- 可以用来判断某个字符串是否符合某种模式，比如判断某个字符串是不是邮箱地址，网址，电话号码，身份证号
- 可以用来在文本中寻找并抽取符合某种模式的字符串，比如电子邮件地址，网址，身份证号
- 正则表达式规定了一种模式，是一个字符串，如“abc”，“a.?p.*c”

正则表达式字符

字符			
一般字符	代表自身。	abc	abc
.	匹配任意除换行符\n外的字符。	a.c	abc
\	转义字符，使后一个字符改变原来的意思。	a\.c a\\c	a.c a\c
[...]	字符集。对应的位置可以是字符集中任意字符。	a[bcd]e a[b-d]e	abe ace ade
[^abc]	第一个字符是^表示取反。		匹配abc之外的任意一个字符

预定义字符，可以在字符集中使用

字符			
\d	数字[0-9]	a\dc	a2c
\D	非数字[^\d]	a\Dc	abc
\s	空白字符[<空格>\n\t\r]	a\sc	a c
\S	非空白字符	a\Sc	abc
\w	单词字符[A-Za-z0-9]	a\wc	abc
\W	非单词字符[^\w]	a\Wc	a c

数量词字符

字符			
*	匹配前一个字符0或无数次	abc*	ab abccccc
+	匹配前一个字符1次或无数次	abc+	abc abccccc
?	匹配前一个字符0次或1次	abc?	ab abc
{m}	匹配前一个字符m次	ab{3}c	abbbbc
{m, n}	匹配前一个字符m至n次, 可省略	ab{2, 3}c	abbc abbbbc
[\\dabc]+	长度不为0的由数字或a、b、c构成的字符串		123abc 123abc123abc

正则表达式示例

`[1 - 9]\d *`

正整数

`-[1 - 9]\d *`

负整数

`-?[1 - 9]\d *|0`

整数

`-?([1 - 9]\d *.\d *|0.\d *|0?\d *|0?\d *|0?)`

小数

注: | 表示“或”，短路匹配

Python与正则表达式——import re

`re.match(pattern, string, flags=0)`

从字符串的起始位置一个模式pattern

flags是标志位，用于控制模式串的匹配方式，

如是否区分大小写，多行匹配等，匹配成功则返回一个匹配对象，失败返回None。

re.I 使匹配不区分大小写。

re.M 跨多行匹配，影响^和\$。

re.S 使.匹配所有字符。

```
# re.match(pattern, string, flags)
```

```
import re
```

```
m1 = re.match("ab*c", "abbcd")
```

```
print('m1: ', m1 != None)
```

```
m2 = re.match("ab+c", "ac")
```

```
print('m2: ', m2 != None)
```

```
m3 = re.match("a.?bc.*", "abcd")
```

```
print('m3: ', m3 != None)
```

```
m4 = re.match("a.?bc.*", "aBcd")
```

```
print('m4: ', m4 != None)
```

```
m5 = re.match("a.?bc.*", "aBcd", re.I) # 修饰符re.I不区分大小写
```

```
print('m5: ', m5 != None)
```

```
m1: True
```

```
m2: False
```

```
m3: True
```

```
m4: False
```

```
m5: True
```

re.search(pattern, string[, flags])

查找字符串中可以匹配成功的子串

匹配成功则返回一个匹配对象，失败返回None

```
# re.search(pattern, string[, flags])
import re
print('1: ', re.search("a.+bc*$", "mnadewbc")) # '$' 表示处于字符串的结尾
print('2: ', re.search("a.+bc*$", "mnadewbcd"))
print('3: ', re.search("^a.+bc*", "mnadewbcd")) # '^' 表示与字符开始处进行匹配
print('4: ', re.search("^a.+bc*", "adewbcd"))
```

```
1:  <re.Match object; span=(2, 8), match='adewbc'>
2:  None
3:  None
4:  <re.Match object; span=(0, 6), match='adewbc'>
```

re.findall(pattern, string[, flags])

查找字符串中所有和模式匹配的子串放入列表，一个子串都找不到则返回[]

```
# re.findall(pattern, string[, flags])
import re
sentence = '2020 is a leap year with 366 days.'
print('number: ', re.findall('\d+', sentence))
print('word: ', re.findall('[A-Za-z]+', sentence))
```

```
number:  ['2020', '366']
word:    ['is', 'a', 'leap', 'year', 'with', 'days']
```

字符边界

\A 与字符串开始的位置匹配，不消耗任何字符。

\Z 与字符串结束的位置匹配，不消耗任何字符。

^ 与字符串开始的位置匹配，不消耗任何字符。在多行模式中，匹配每一行开头。

\$ 与字符串结束的位置匹配，不消耗任何字符。在多行模式中，匹配每一行末尾。

\b 匹配一个单词的开始处和结束处，不消耗任何字符。

\B 和\b相反，不允许是单词的开始处和结束处。

字符边界

```
import re
test1 = "\Ahow are"
print('test1-1: ', re.search(test1, "hhow are you"))
print('test1-2: ', re.search(test1, "how are you").group())
test2 = "are you\Z"
print('test2-1: ', re.search(test2, "how are you?"))
print('test2-2: ', re.search(test2, "how are you").group())
test3 = "^how are"
print('test3-1: ', re.findall(test3, "how are you\nhow are you", re.M))
test4 = "are you$"
print('test4-1: ', re.search(test4, "how are you\nhow are you?", re.M).group())
test5 = r"\bA.*B\b C"
print('test5-1: ', re.search(test5, "Abb$BD CD"))
print('test5-2: ', re.search(test5, "test Abb$B CD").group())
test6 = r"\BA.*B\B\w C"
print('test6-1: ', re.search(test6, "test Aab$B CD"))
print('test6-2: ', re.search(test6, "testAab$BE CD").group())
```

```
test1-1: None
test1-2: how are
test2-1: None
test2-2: are you
test3-1: ['how are', 'how are']
test4-1: are you
test5-1: None
test5-2: Abb$B C
test6-1: None
test6-2: Aab$BE C
```

正则表达式中的“分组”

括号中的表达式是一个分组，多个分组按左括号从左到右从1开始编号。

```
# group
import re
pattern = "((ab*)c)d)e"
r = re.match(pattern, "abbcddefg")
print('group: ', r.groups())
print('index: ', r.lastindex)
print('group 0:', r.group(0))
print('group 1:', r.group(1))
print('group 2:', r.group(2))
print('group 3:', r.group(3))
```

```
group:      ('abbcd', 'abbc', 'abb')
index:      1
group 0:    abbcde
group 1:    abbcd
group 2:    abbc
group 3:    abb
```


re.finditer(pattern, string[, flags])

在字符串中找到正则表达式所匹配的所有子串，并作为一个迭代器返回

```
import re
s = '123[45]67<890>a[bc]ba<098>'
m = '\\[(\\d+)\\]|<(\\d+)>'
for x in re.finditer(m,s):
    print(x.group())
```

```
[45]
<890>
<098>
```

```
import re
p = r"(((ab*)+c|12+3)d)e"
for x in re.finditer(p, 'ababcdef12def1223def'):
    print(x.group())      # group 等价于 group(0)
    print(x.groups())     # groups 返回一个元祖，元素依次是1-3号分组所匹配的字符串
```

```
ababcde
('ababcd', 'ababc', 'ab')
1223de
('1223d', '1223', None)
```

用于替换匹配的子串，
repl可以是替换串，
也可以是函数。

re.sub(pattern, repl, string)

```
import re
phone = '010-1234-5678'
num = re.sub(r'\D', '', phone)
print("Phone Number : ", num)
```

Phone Number : 01012345678

```
import urllib.request
url = "https://its.pku.edu.cn"
file = urllib.request.urlopen(url)
info = file.info()
print(info)
```

Date: Sun, 01 Mar 2020 06:53:12 GMT
Server: Apache
Set-Cookie: JSESSIONID=57442C88BECAFCBF64BDD23B2C7E1F20; Path=/; Secure; HttpOnly
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8

```
new_info = re.sub('(.+): (.+)',
                  lambda x: '{}: {}'.format(x.group(1), x.group(2)),
                  str(info))
print(new_info)
```

'Date': 'Sun, 01 Mar 2020 06:53:12 GMT'.
'Server': 'Apache'.
'Set-Cookie': 'JSESSIONID=57442C88BECAFCBF64BDD23B2C7E1F20; Path=/; Secure; HttpOnly'.
'Connection': 'close'.
'Transfer-Encoding': 'chunked'.
'Content-Type': 'text/html; charset=utf-8'.

数量词的贪婪模式与非贪婪模式

```
import re
# 量词+,*,?,{m,n}默认匹配尽可能长的字符串 贪婪模式
pattern = "<t>.*</t>"
string = "<t>abcd</t><t>abcde</t>"
test1 = re.match(pattern, string)
print('test1: ', test1.group())

# 在量词+,*,?,{m,n}后面加'?'匹配尽可能短的字符串 非贪婪模式
pattern1 = "<t>.*?</t>"
test2 = re.match(pattern1, string)
print('test2: ', test2.group())
```

```
test1:  <t>abcd</t><t>abcde</t>
test2:  <t>abcd</t>
```

正则表达式自定义命名分组(?P)

一个正则表达式可以有多个自定义名称的分组，可以通过分组名称提取匹配的字符串，每一个分组的格式是：(?P<自定义分组名称>正则字符串)

```
def addOne(matched):  
    value = int(matched.group('int'))  
    return str(value + 1)  
  
string = 'A123B4CD56EF789G'  
print(re.sub('(?P<int>\d+)', addOne, string))
```

A124B5CD57EF790G

参考资料

Python多进程、多线程

<https://zhuanlan.zhihu.com/p/46368084>

<https://www.liaoxuefeng.com/wiki/1016959663602400/1017968846697824>

正则表达式:

<https://docs.python.org/zh-cn/3.6/library/re.html>

<https://www.runoob.com/python/python-reg-expressions.html>