IP协议

```
ip地址 + mac地址可以标识全世界范围内的任何唯一一台计算机
ip地址 + mac地址 + port 地址可以标识全世界范围内的任何唯一一台计算机上的应用程序
```

TCP协议

```
tcp/udp
```

socket 套接字

```
"""
:param 服务端TCP套接字
创建套接字
绑定端口
监听服务
接受请求
接受数据
发送数据
关闭连接
关闭套接字  (可选)
"""

import socket

phone = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # 数据流 流式协议 --- tcp协议
phone.bind(('127.0.0.1', 12345))
phone.listen(5)  # 5指的是半连接池大小
print('监听中  等待客户端的连接。。。')
conn, client_addr = phone.accept()
print(conn)
print('客户端的ip和端口', client_addr)
data = conn.recv(1024)  # 最大数据接收量为1024个字节
print('客户端发送来的消息: ', data.decode('utf-8'))
conn.send(data.upper())
conn.close()
# phone.close() 断开服务连接

"""
:param 客户端TCP套接字
创建套接字
发起连接请求
发送数据
接受数据
关闭套接字  (必选)
"""
import socket
```

```
phone = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
phone.connect(('127.0.0.1', 12345))
phone.send('bytes类型的数据'.encode('utf-8'))
data = phone.recv(1024)
print('服务端发送来的数据: ', data.decode('utf-8'))
phone.close()

'''
无论是send 还是 recv实际上都是在和本机的操作系统做交互
'''

"""
:param 服务端UDP套接字
创建套接字
绑定端口
接收数据
发送数据
"""
import socket

server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)  # 数据报 报文协议 --- udp协议
server.bind(('127.0.0.1', 12345))
while True:
    data, client_addr = server.recvfrom(1024)
    data = data.decode('utf-8')
    print('来自客户端的消息>>>', data)
    server.sendto('已经完成消息接收'.encode('utf-8'), client_addr)
    if data == 'q':
        break
server.close()

"""
:param 客户端UDP套接字
创建套接字
发送数据
接受数据
"""
import socket

client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_addr = ('127.0.0.1', 12345)
while True:
    msg = input('请输入发送内容: ')
    client.sendto(msg.encode('utf-8'), server_addr)
    data, addr = client.recvfrom(1024)
    print('来自服务器的内容>>>', data.decode('utf-8'))
    if msg == 'q':
        break
client.close()

'''
粘包原因:

udp不会出现粘包问题 ---报式收发
```

```
tcp会出现粘包问题 ---流式收发
解决粘包问题
1. 拿到数据总大小（先收固定长度的头 解析数据 ）
    struct模块 --- 将数据转化为固定长度的bytes
    struct.pack('模式', 数据)
    struct.unpack('模式', 数据)
2. 统计接受数据长度
3. 接收长度 == 数据总长度 停止接收
'''
```

并发

```
# 基于TCP
import socketserver

class MyRequestHandle(socketserver, BaseRequestHandler):
    def handle(self):
        print(self.request)  # 如果tcp协议 self.request --> conn
        print(self.client_address)
        while True:
            try:
                msg = self.request.recv(1024)
                if len(msg) == 0: break
                print(msg.decode('utf-8'))
                self.request.send(msg.upper())
            except Exception:
                break
        self.request.close()

server = socketserver.ThreadingTCPServer(('127.0.0.1', 8888), MyRequestHandle)
server.serve_forever()

# 基于UDP
import socketserver

class MyRequestHandle(socketserver, BaseRequestHandler):
    def handle(self):
        client_data = self.request[0]
        server = self.request[1]
        client_address = self.client_address
        server.sendto(client_data.upper(), client_address)


server = socketserver.ThreadingUDPServer(('127.0.0.1', 8888), MyRequestHandle)
server.serve_forever()
------------------------------------------------------------
# 并行是并发 并发只是看起来像并行
# 单核计算机只能实现并发
# 同步 和 异步  ---描述任务提交方式
    同步：任务提交之后 原地等待任务的返回结果
    异步：任务提交之后 不等待 完成其他任务

# 阻塞 和 非阻塞  ---描述程序执行状态
```

```
    阻塞（阻塞态）：
    非阻塞（就绪态 运行态）：
------------------------------------------------------------
# 创建进行的俩种方式
# 方式一
from mutilprocessing import Process
import time

def tash(name):
    print('%s is running' % name)
    time,sleep(3)
    print('%s is over' % name)

def mian():
    p = Process(target=tash, args=('参数', )) # 创建进程对象
    p.start() # 告诉操作系统创建一个进程
    print('主') # 由于是异步 所以此行先被打印

if __name__ == '__main__':
    mian()

# 方式二
from mutilprocessing import Process
import time

class MyProcess(Process)
    def __init__(self, name):
        self.name = name

    def run(self):
        print('%s is running' % self.name)
        time,sleep(3)
        print('%s is over' % self.name)


if __name__ == '__main__':
    p = MyProcess()
    p.start()
    print('主')

'''
创建进程就是在内存中申请一块内存空间
每一个进程都对应一个独立的内存空间
默认情况下 进程间无法交互
进程间数据时相互隔离的
'''

# join方法 --让主进程等待子进程运行结束后再执行
p.join()

p.terminate() # 杀死当前进程
p.is_alive()  # 查看进程是否存活  返回布尔值

# mutilprocessing 模块的相关方法
```

```
'''
查看pid  tasklist --windows   tasklist |findstr PID
        ps aux   --linux     ps aux |grep PID
    os.getpid() 获取进程号
    os.fetppid() 获取父进程号
'''

# 僵尸进程和孤儿进程
'''
僵尸进程：开设子进程之后，该进程死后不会立即释放占用的进程号（父进程需要查看）
        所有的进程都会经历此状态

孤儿进程：主进程结束 子进程继续运行 --操作系统会回收资源
'''
# 守护进程
from mutilprocessing import Process
def task(name):
    print('%s is running' % self.name)
    time,sleep(3)
    print('%s is over' % self.name)

if __name__ == '__main__':
    p = Process(target=task, args=('egon,'))
    p.daemon = True  # 将进程设置为守护进程
    p.start()
```

互斥锁

```
'''
针对多个进程操作同一份数据的时候 会出现数据错乱问题
解决方式 -- 加锁
牺牲效率 保证数据安全
'''
from mutilprocessing import Process, Lock
import time
import json
import random
def search(i):
    with open('data', mode='r', encoding='utf-8') as f:
        dic = json.loads(f)
        print('用户%s查询余票 %s' %(i, dic.get('ticket_num')))

def buy(i):
    with open('data', mode='r', encoding='utf-8') as f:
        dic = json.loads(f)
    time.sleep(random.randint(1,3))
    if dic.get('ticket_num') > 0:
        dic['ticket_num'] -= 1
        with open('data', mode='w', encoding='utf-8') as f:
        dic = json.dump(f)
        print('用户买票成功')

    else:
```

```
        print('用户买票失败')

def run(i):
    search(i)
    mutex.acquire()
    buy(i)
    mutex.releasse()

if __name__ == '__main__':
    mutex = Lock()
    for i in range(1, 11):
        p = Process(target=run, args=(i, mutex))
        p.start()

'''
行锁：锁住一行
表锁：锁住一张表
'''
```

消息队列 （管道 + 锁）

```
'''
对列：先进先出
'''
from mutilprocessing import queue

q = queue.Queue(5) # 生成一个队列 默认值30000+ 当队列存满且要继续存入数据 程序阻塞
q.put()  # 存入数据
q.get()  # 取出数据
q.get_nowait()
q.get(timeout=30)
q.full() # 判断队列是否存满
q.empty() # 判断队列是否为空

'''
---多进程下不精确 不推荐使用
q.get_nowait()
q.full() # 判断队列是否存满
q.empty() # 判断队列是否为空
'''
```

IPC 机制

```
'''
通过队列实现俩个进程之间的数据共享
'''
'''主与子'''
from mutilprocessing import Queue, Process
def producer(q):
    q.get()
    print('hello')
```

```python
if __name__ == '__main__':
    q = Queue()
    q.put(12564548)
    p = Process(target=producer, args=(q,))
    p.start()

'''子与子'''
from mutilprocessing import Queue, Process
def producer(q):
    q.put(12564548)
    print('hello')

def consumer(q):
    q.get()
    print('hello')

if __name__ == '__main__':
    q = Queue()
    p1 = Process(target=producer, args=(q,))
    p2 = Process(target=consumer, args=(q,))
    p1.start()
    p2.start()


'''
 生产者消费者模型
 生产者 + 消息队列 + 消费者
'''
from mutilprocessing import Queue, Process, JoinableQueue

def producer(name, food, q):
    for i in range(10)
        data = '%s 生产了%s%s' % (name,food,i)
        time.sleep(random.randint(1,3))
        print(data)
        q.put(data)

def consumer(name, q):
    while True:
        food = q.get()
        # if food is None:
            # breaK
        time.sleep(random.randint(1, 3))
        print( '%s 消费了%s%s' % (name,food)
        q.task_done()

if __name__ == '__main__':
    # q = Queue()
    q = JoinableQueue()
    p1 = Process(target=producer, args=('egon', '包子', q))
    p2 = Process(target=producer, args=('2hao', '馒头', q))

    c1 = Process(target=consumer, args=('3hao', '馒头', q))
```

```
    c2 = Process(target=consumer, args=('4hao', '馒头', q))
    p1.start()
    p2.start()
    c1.deamon = True
    c2.deamon = True
    c1.start()
    c2.start()

    p1.join()
    p2.join()
    # q.put(None)
    # q.put(None)
    q.join() #
'''
JoinableQueue 每当往该队列存入数据的时候 内部计数器 +1
每当调用一次task_done() 计数器 -1
q.join() 计数器为0 执行
'''
```

线程

```
'''
进程：资源单位
线程：执行单位
一个进程内可以开启多个线程 无需申请内存空间  系统开销小
同一个进程下的多个线程数据是共享的
'''
# 开启线程的俩种方式

# 方式一
from threading import Thread
import time

def task(name):
    print('%s is running' % name)
    time.sleep(1)
    print('%s is over' % name)

t = Thread(target=task, args=('egon,'))
t.start()

# 方式一
from threading import Thread
class MyThread(Thread):
    def __init__(self):
        super().__init__()
        self.name = name
    def run(self):
        print('%s is running' % name)
        time.sleep(1)
        print('%s is over' % name)
```

```python
if __name__ == '__main__':
    t = Mythead('egon')
    t.start()
    print('主')

# 线程对象的join方法
from threading import thread
import time

def task(name):
    print('%s is running' % name)
    time.sleep()
    print('%s is over')

if __name__ == '__main__':
    t = Thread(target=task, args=('egon',))
    t.start()
    t.join()  # 主线程等待子线程运行结束再继续执行
    print('主')

# 同一个进程下的多个线程 数据共享
# 线程对象及其他方法
active_count()  # 统计当前活跃的线程数
current_thread().name  # 获取当前线程的名字
# 守护线程
在t.start() 之前 添加 t.daemon = True
主线程运行结束之后不会立刻结束 会等待所以非守护线程结束后才会结束
因为主线程结束也就意味着进程结束

# 线程互斥锁
from threading import Thread, Lock

mutex = LOck()
mutex.acquire()
...
mutex.release()

# GIL全局解释器锁
'''
Cpython
Jpython
Pypypython

全局解释器锁是Cpython解释器的特点
GIL是保证解释器级别的数据安全
GIL会导致同一个进程下的多个线程是无法同时执行的
针对不同数据还是要加不同的锁处理

Cpython中的内存管理不是线程安全的
内存管理（垃圾回收机制） ---  1.引用计数 2.标记清除 3.分代回收

'''
```

```
'''
计算密集型：多进程占优势
IO密集型：多线程占优势
---多进程下面开设多线程  一方面利用多核 另一方面节省资源
'''
```

TCP实现并发

```
'''服务端'''
import socket
from threading import Thread
from mutilprocessing import Process

server = socket.socket(AF_INET, SOCK_STREAM)
server.bind(('127.0.0.1', 8000))
server.listen(5)

def talk(conn)
    while True:
        try:
            data = conn.recv(1024)
            if len(data) == 0:
                break
            print(data.decode('utf-8'))
        except connectionResetError as e:
            print(e)
            break
    conn.close
while True:
    conn, addr = server.accept()
    t = Thread(target=talk, args=(conn,))
    t.start()

'''客户端'''
client = socket.socket(AF_INET, SOCK_STREAM)
client.connect(('127.0.0.1', 8000))

while True:
    client.send(b'hello world')
    data = client.recv(1024)
```

死锁和递归锁

```
# 死锁  ---
# 递归锁 ---可以被连续的acquire 和 release 内部具有计数器 计数器不为 其他人无法使用锁
Rlock
# 信号量  ---在并发编程中信号量指的是锁
Seamphore
# Event事件 --一些进程/线程需要等待另外一些进程/线程运行完毕之后才能运行，类似于发射信号一样
from threading import Thread, Event
```

```
event = Event()
def light():
    print('红灯亮着的')
    time.sleep(3)
    print('绿灯亮啦')
    event.set()

def car(name):
    print('%s车正在等红灯'%name)
    event.wait()
    print('%s车加油门飙车走啦'%name)

if __name__ == '__main__':
    t = Thread(target=light)
    t.start()
    for i in range(20):
        t = Thread(target=car, args=('%s'%i,))
        t.start()

# 线程q
import queue
# 队列q  先进先出
q = queue.Queue(3)
q.put()
q.get()
q.get_nowait()
q.get(timeout=3)
q.full()
q.empty()
# 栈q 后进先出
q = queue.LifoQueue(3)
# 优先级q
q = queue.PriorityQueue(3)
q.get((优先级, 数据)) # 优先级越高数字越小
```

进程池 与 线程池

```
# 池 --用来保证计算机硬件安全的情况下最大限度的利用计算机
    # 降低了运行效率 保证了计算机硬件安全 从而让程序正常运行
    # 池中的进程/线程会固定存在 不会随意的创建和销毁
# 线程池
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
pool = ThreadPoolExecutor(5) # 括号内可以传数字 不传默认开启cpu个数五倍的线程
def task(n):
    print(n)
    time.sleep(2)

pool.submit(task, 1) # 向池子中提交任务 异步提交 有返回值可以接收 返回对象
pool.shutdown() # 关闭线程池 等待线程池中所有任务运行完毕
print('主')
# 进程池  -- 进程需要在__main__下开启

pool = ProcessPoolExecutor(5) # 括号内可以传数字 不传默认开启cpu个数的进程
```

```
def task(n):
    print(n)
    time.sleep(2)

pool.submit(task, 1) # 向池子中提交任务 异步提交 有返回值可以接收 返回对象
pool.shutdown() # 关闭线程池 等待线程池中所有任务运行完毕
print('主')

pool.submit(task, 1).add_done_callback() # 给该任务添加回调机制
```

协程

```
'''
进程：资源单位
线程：执行单位
协程：抽象概念  --单线程下实现并发
'''
pip install gevent
from gevent import spawn
from gevent import monkey
# from gevent import monkey; monkey.patch_all() 简写

g = spawn(函数)
g.join()
```

协程实现TCP服务端并发

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

def communication(conn):
    while True:
        try:
            data = conn.recv(1024)
            if len(data) == 0:
                break
            conn.send(data.upper())
        except Exception as e:
            print(e)
            break
    conn.close()
def server(ip, port):
    server = socket.socket(AF_INET, SOCK_STREAM)
    server.bind((ip, port))
    server.listen(5)
    while True:
        conn, addr = server.accept()
        spawn(communication, conn)

if __name__ == '__main__':
    g = spawn(server, '127.0.0.1', 8000)
```

```
    g.join()

    '''
    多进程下开设多线程 多线程下开设协程 使程序执行效率提升
    '''
```

IO模型

```
'''
同步异步
阻塞非阻塞
常见的网络阻塞状态
accpet
recv
recvfrom
'''
# 阻塞IO
# 非阻塞IO
from socket import socket
server = socket(AF_INET, SOCK_STREAM)
server.bind(('127.0.0.1', 8000))
server.listen(5)
server.setblocking(Flase) # 将所有的网络阻塞变为非阻塞

r_list = []
del_list = []
while True:
    try:
        conn, addr = server.accept()
        r_list.append(conn)
    except BlockingIOError as e:
        # print('列表长度>>' len(r_list))
        for conn in r_list:
            try:
                data = conn.recv(1024)
                if len(data) == 0:
                    conn.close()
                    del_list.append(conn)
                    continue
            except BlockingIOError as e:
                continue
            except ConnectionReserError:
                conn,close()
                del_list.append(conn)
        for conn in del_list:
            r_list.remove(conn)
        del_list.clear()
'''
非阻塞IO会长时间占着cpu但是不干活
'''
'''

IO多路复用
```

```
'''
import socket
import select

server = socket.socket()
server.bind(())
server.listen(5)
server.esrblocking(Flase)
read_list = [server]

while True:
    # res = select.select(read_list, [], [])  # 监管对象
    r_list, w_list, x_list = select.select(read_list, [], [])
    for i in r_list:
        if is server:
            conn, addr = i.accept()
            read_list.append(conn)
        else:
            res = i.recv(1024)
            if len(res) == 0:
                i.close()
                read_list.remove(i)
                continue
'''监管机制
select --windows linux
poll --linux (监管数量更多)
epoll --linux 为每一个监管对象都绑定一个回调机制 一旦有响应 立即触发回调机制
'''

# 异步IO --所有模型效率最高 使用最广泛
'''
模块：asyncio
异步框架：sanic tronado twisted
速度快！！！
'''
import threading
import asyncio

@asyncio.coroutine
def hello():
    pass

loop = asyncio.get_event_loop()
tasks = [hello(), hello()]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```