**1. 网络编程：**

应用调用操作系统，操作系统控制硬件

osi七层协议：应用层，表示层，会话层，传输层，网络层，数据链路层，物理层

传统的五层协议：应用层（应用层，表示层，会话层），传输层，网络层，数据链路层，物理层。

**客户端：**

应用层： 应用产生数据，遵循协议（如:http,ftp协议）的标准来对数据包装

传输层： 通过TCP/UDP端口协议传输（端口范围0—65535）

网络层： 遵循IP协议；由于客户端和服务端可能在不同的子网中，要想发送接收方就需要找到目标的具体位置。

数据链路层： 遵循以太网协议；其中[报头：data]；报头18个字节，前6位源地址后六位目标地址，然后6位描述信息。网卡（eth0）。用MAC地址通过广播形式找到**同一子网（局域网）**目标地址位置，通过MAC+IP可以找到全世界目标在网中的具体位置

物理层： 比如数据链路层的数据解析为01011001通过网络并传输。

应用层 客户端产生数据，通过协议包装。服务端协议解包装，获取数据

传输层 通过tcp/ip发送数据，即要知道发给哪个端口

网络层 包装 ip头：数据 IP

数据链路层 在包装 报头：ip头：数据 IP+MAC(arp协议解析ip对应的mac)

物理层 解析为二进制并通过网络传输

**TCP/IP协议（数据流形式发送）：**

TCP：三次握手和四次挥手。效率低，可靠性高

三次握手：

1. 客向服发起请求连接syn=1。

2.1 服确认客的请求ack=1。

2.2 服向客连接syn = 1。

3. 客收到服的确认信息。

2.1和2.2可以合成一步，即客同意服的请求，并建立连接

四次挥手

1. 客向服发起断开请求FIN=1,此时客不在向fuw发送数据。

2.服确认客的断开请求ack=1，此时服可能还没发完客请求的数据

3. 服向客发送数据已经完毕，准备断开FIN=1

4. 客收到服的信息，并确认ack=1。

2 和 3 不 可以合成一步，因此为服可能还没发完客请求的数据

UDP：效率高，可靠性低

**socket：**

socket是基于C/S架构的，也就是说进行socket网络编程，通常需要编写两个py文件，一个服务端，一个客户端。

**1.tcp数据流协议**

**不是一个send对应一个recv且会粘包**

服务端（tcp）：

import socket

# 绑定端口端口范围0——6535,前1024为给操作系统用
ip\_port = ('127.0.0.1', 9999)

# 默认使用 socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)，即tcp协议
sk = socket.socket() # 创建套接字
sk.bind(ip\_port) # 绑定服务地址
sk.listen(5) # 监听连接请求

print('启动socket服务，等待客户端连接...')
conn, address = sk.accept() # 等待连接，此处自动阻塞

while True: # 一个死循环，直到客户端发送‘exit’的信号，才关闭连接
client\_data = conn.recv(1024).decode() # 接收信息 1024表示最大可接受字节数
if client\_data == "exit": # 判断是否退出连接
exit("通信结束")
print("来自%s的客户端向你发来信息：%s" % (address, client\_data))
conn.sendall('服务器已经收到你的信息'.encode()) # 回馈信息给客户端
conn.close() # 关闭连接

客户端（tcp)：

import socket
ip\_port = ('127.0.0.1', 9999)
s = socket.socket() # 创建套接字
s.connect(ip\_port) # 连接服务器
while True: # 通过一个死循环不断接收用户输入，并发送给服务器
inp = input("请输入要发送的信息： ").strip()
if not inp: # 防止输入空信息，导致异常退出
continue
s.sendall(inp.encode())
if inp == "exit": # 如果输入的是‘exit’，表示断开连接
print("结束通信！")
break
server\_reply = s.recv(1024).decode()
print(server\_reply)
s.close() # 关闭连接

**想要执行系统命令需要调用import os或者 import subprocess**

1. import os

os.system('ls /')

# 如果为linux系统，则执行ls命令，但是默认输出到终端，不能保存

2. import subprocess

obj = subprocess.Popen('ls /', shell=True,

stdout=subprocess.PIPE,

stderr=subprocess.PIPE,)

# windows默认gbk编码

# shell=True,表示可以以shell方式运行命令，

# stdout,stderr是输出管道，如果命令错误则输出stderr里管道的内容

print('stdout--->1', obj.stdout.read().decode('utf-8'))

print('stderr--->1', obj.stderr.read().decode('utf-8'))

**如下是windows**

a = subprocess.Popen('dir c:\\', shell=True,

stdout=subprocess.PIPE,

stderr=subprocess.PIPE,)

print(a.stdout.read().decode('gbk'))

**粘包：**

**理解：**

**1. 不管recv还是send都不是直接接受对方资源，而是操作自己的操作系统内存**

**即不是一个send对应一个recv**

**2. send方法时间特点：**速度快，（应用层）只需要把发送的数据复制给操作系统缓存，剩下的全部交给操作系统处理（把应用产生的数据通过操作系统读取到硬盘中 ）

**3. recv方法时间特点：**

1. 等待时间长

2. copy时间（把操作系统缓存上的数据拷贝到内存中然后读取 ）

**原因：**

发送端为了将多个发往接收端的包更有效的发到对方，使用了优化算法（Nagle）：

即将多次间隔较小且数据量较小的数据包合并成一个数据块，然后进行封包。

**即粘包是由于Nagle算法特性决定的**

报头+数据:且报头必须为固定长度

自己定制固定长度的报头，目的是为了让对方知道我要发送的数据信息（长度）

一般解决粘包

基于struct模块制定报头

数据长度4个bytes（发文件的时候容易超出范围）

好的解决粘包问题：

发送是通过字典形式发送，然后对字典求len()在基于struct模块制定报头

字典长度4个bytes（里面的数据可能很大很大）。而且还可以发送其他有用的信息

**2.udp数据报协议**

**一个sendto()对应一个recvfrom()且不会粘包**

**2. 并发编程**

**操作系统：**

1. 帮你封装好硬件的接口，来给应用程序调用，应用只需要操作接口就可以操作硬件

2. 操作系统管理电脑上运行的多个进程，并且将多个进程的竞争变得有序

**进程：from multiprocessing import Process**

开启进程的两种方式

join用法

os.getpid()和os.gitppid()

**守护进程：**

1. 守护进程会在主进程代码执行结束后立马终止(不管其他进程执行情况)。

2. 守护进程内无法再开启子进程。

3. 在子进程(p)开启之前把p.daemon=True

**互斥锁和join区别()**

都是保证数据不错乱，但是效率变低。Join把整个进程都弄成串行，而互斥锁可以实现把需要串行的部分实现，而非整体

**多个进程之间通信使用队列（IPC通信机制）：**

mutiprocessing模块为我们提供基于消息的IPC通信机制：队列和管道。队列和管道都i是把数据存在内存中，而队列是基于（管道和锁实现的)

生产者消费者模型（数据产生和数据处理)：

1. 生产者产生数据并存放于队列中（q=Queue()）q中，即q.put()

2. 消费者在队列中取出数据，即q.put

3. 注意如果队列中不存在数据则会在哪里一直等待

4. 生产者生产完之后，等到消费者全部消费之后（p1.join()），通过发送信号来告诉消费者生产结束(q.put(None))。然后消费者通过判断 接收到的值来相应处理

**IPC通信机制生产者消费者模型好处：**

生产者<——>队列<——>消费者

1. 平衡生产者和消费者之间的速度差

2. 解开生产者和消费者之间的耦合（）

**注意：**

实际要用rabbitmq实现中间管理生产者消费者,因为用原生的queue需要各个组件(生产者,消费者)在同一电脑上,性能和稳定性较低。

**线程：from threading import Thread**

1. 两种开启方式，同Process调用方式一样，只需要改改调用名字

2. join用法

3. os.getpid()和os.gitppid()

4. p.is\_alive() 判断线程是否死掉

**守护线程：**

**开启一个进程**：默认就只有一个主线程，且在执行完代码之后立即死掉

**一个进程开多线程：**主线程在在执行玩代码之后不会死掉，因为主线程开了其子线程，就要等其他子线程都结束之后在死掉。

即在同一进程,守护线程要在其他非守护线程执行完毕之后才死(因为主线程要等其他子线程都结束之后才死,此时守护主线程的线程也该死掉)

**进程线程区别：**

1. 进程比线程消耗资源

2. 进程间数据不共享，线程共享同一个进程里所有信息

**GIL（全局解释性锁）：**

**运行一个python程序大致经历如下步骤：**

1. 把解释器加载到内存

2. 把python代码加载到内存

3. 把python代码传给解释器并解释

**在多线程中：**

1. 在CPython解释器中：一个进程的多个线程同一时刻只能有一个线程获得GIL,是为了保证python的**垃圾回收**是安全的

2. 针对不同的数据就应该加不同的锁,因为解释器的锁只是为了保证解释器的安全,并不能保证你操作的其他共享数据是安全的

**GIL与多线程：**

IO密集型：多线程

CPU(计算)密集型：多进程

**死锁：**

比如线程A要等线程B释放锁，线程B要等线程A释放锁，结果就产生死锁。

注意：(LOCK)互斥锁只能acquire一次，再次acquire需要先release

**递归锁：**

(RLOCK)可以连续acquire多次，每acquire一次计数器加一，release计数减一。只有计数为0时才能被其他线程acquire，可以用递归锁解除死锁。

**信号量（本质为锁）:**

from threading import Semaphore,currentThread

import time,random

sm = Semaphore(3) # 表示同一时间能有三个人抢到锁

可以通过with 调用锁如

def task()：

with sm:

print("%s is " % currentThread().getName())

time.sleep(random.randint(1,3))

main里：

for i in range(10)

t = Thread(targer=task)

t.start()

**Event:**

from threading import Event

event = Event()

event.wait()

表示一直等待，直到发出event.set()

event.wait(5)

表示最多要等待5秒，超出时间还没有等到event.set()，则不等待，继续执行代码

event.is\_set()判断是否发出event.set()请求，发出则为True

**定时器:**

from threading import Timer

t = Timer(5, task, args=("..."), ) 本质就是定时开启线程，然后执行任务

t.strat()

t.canccel()取消定时器

表示5秒之后执行task任务，参数...

**queue(队列)**

1. 先进先出队列

2. 后进先出队列

3. 优先级队列

**进程池线程池**

通过进程池控制进程数，通过线程池控制线程数

关闭进程池入口，不能再次往进程池提交

异步提交任务

from concurrent.futures import ThreadPoolExecutor

pool = ThreadPoolExecutor(3)

指定线程池大小， 即同一时间最多只能有三个线程，一旦有线程执行完之后，则会被求他调用。进程池线程池使用方式类似，只是导入名不同

**异步调用与回调机制：**

提交任务的两种方式：异步调用，同步调用

同步调用：提交完任务之后，就在原地等待任务执行完毕，拿到执行结果，然后执行下一个任务，导致程序串行执行

**异步调用：**提交完任务之后，不等待任务执行就开始下个任务。通常一个函数就执行一个功能，实现函数间的解耦合。但是我们可以通过绑定回调函数，来执行完函数后调用另一个函数，是把函数当作参数(该函数执行完后就会)传递给回调函数，

**阻塞与同步:**

阻塞/非阻塞进程运行的一种状态；同步跟阻塞没有关系，同步只是一种运行方式，不管是计算型还是IO型，都要等待任务执行完毕。而阻塞是因为遇到了类似IO操作时，系统会剥夺cpu执行权限，直到相应的IO操作执行完毕，重新申请cpu.

**协程（单线程内实现并发）**：

单线程并发，又称微线程。简单来说：协程是一种用户态的轻量级线程，即协程是由用户程序自己控制调度的。

比如单线程内要执行五个任务，要模拟操作系统实现切换

如果是并发执行纯计算的任务：那执行总时间=计算时间+切换时间

 即：并发适用于IO操作，利用IO操作的时间执行其他东西。因此如果光是计算还不如直接用串行执行，只有计算时间。

**greenlet模块**：遇到IO不能切换

**gevent：**

gevent异步提交任务。gevent封装了greenlet模块，本身只能监测自己的IO操作！

如果想使用gevent监测所有的IO操作，就需要在所有代码前添加如下

import gevent

from gevent import monkey

monkey.patch\_all()

**IO模型：**

同步：提交完任务，等待任务执行结果，然后在往下执行

异步：提交完任务，不等待任务执行结果，就往下执行，通常要和回调函数结合使用

阻塞：遇到IO之后，如果不处理，操作系统会将cpu剥夺走，让别的使用

非阻塞

**套接字的IO在哪里会发生阻塞**

服务端：在accept和recv，send操作时会阻塞，在accept和recv会感受到明显的阻塞（等待）

**1. 阻塞IO(blocking IO)模型：**

就是一般的套接字，没做任何处理。在accept和recv会感受到明显的阻塞（等待）

单线程遇到IO就阻塞，导致程序没有并发效果。

适用于规模较小（多了服务器会跨），可以用线程池（当线程访问过多而池子太小会明显等待）

单线程遇到IO时如何利用阻塞时间做其他事？

**2. 非阻塞IO(blocking IO)模型：**

**优点：**

1. 此任务执行效率提高（一直占用cpu）

2. 单线程实现实现多并发，当有其中一个单线程遇到IO时时,会切换到其他线程执行单线程实现，不阻塞，类似死循环，但cpu占用率大大提升

**缺点：**

1. 服务端为一个死循环，完全没有阻塞，会大量时间占cpu，且做无用的访问，比如问数据好了没有。

2. 不适用于send发送大量的消息，因为send数据越多占用的缓存越多，如果此时有其他accept连接请求，则会阻塞，直到有其中一个关闭连当请求，即会导致数据不能及时响应

因此不推荐使用。

**3. 多路复用IO模型：**

**select**

rl,wl,xl = select.select(rlist, wlist, [], 0.5)

rl:可以收的套接字 wl:可以发的套接字 xl：异常信息 每隔0.5秒像操作系统询问

优点：select可以同时监测多个套接字，向操作系统询问资源是否准备好，效率提高（比阻塞和非阻塞）

缺点：代理套接字多的话，效率会变低，因为要循环列表询问socket时间越久。因此select随着套接字的增大效率逐渐变低，时间复杂度O()列表

**epoll**

异步的方式提交套接字，每隔套接字都绑定一个回调函数，然后拿走好了的。

事件通知方式，每当fd就绪，系统注册的回调函数就会被调用，将就绪fd放到readyList里面，时间复杂度O(1)哈希表

**4. 异步IO模型（效率最高）：**

1. 向操作系统询问数据，然后直接返回。

2. 当操作系统准备好将数据之后再把返回给调用者