网络编程就是，让在不同的电脑上的软件能够进行数据传递，即进程之间的通信。

计算机都遵守的网络通信协议叫做TCP/IP协议

端口

ip地址和网络服务是一对多的关系，通过ip地址+端口号可以区分不同服务，双方端口并非一一对应。一个进程可能与多个计算机链接，因此会申请很多端口。

每次重新运行网络程序，若端口号未指定，标识程序的端口号都会重新随机分配

知名端口 众所周知的端口号，0—1023

80端口分配给HTTP，21端口分配给FTP

动态端口 不固定分配某种服务 1024—65535

netstat –an 查看端口状态

私有ip 不在公网中使用，属于私网ip，局域网使用

10.0.0.0～10.255.255.255

172.16.0.0～172.31.255.255

192.168.0.0～192.168.255.255

IP地址127．0．0．1~127．255．255．255用于回路测试，

如：127.0.0.1可以代表本机IP地址，用http://127.0.0.1就可以测试本机中配置的Web服务器。

子网掩码

某个IP地址划分成网络地址和主机地址两部分子网掩码的设定必须遵循一定的规则。只有通过子网掩码，才能表明一台主机所在的子网与其他子网的关系

左边是网络位，用二进制数字“1”表示；右边是主机位，用二进制数字“0”表示。

IP地址为“192.168.1.1”子网掩码为“255.255.255.0”。

其中，“1”有24个，代表与此相对应的IP地址左边24位是网络号；

“0”有8个，代表与此相对应的IP地址右边8位是主机号。

**socket**

socket(简称 套接字) 是进程间通信的一种方式，它与其他进程间通信的一个主要不同是：

它能实现不同主机间的进程间通信，我们网络上各种各样的服务大多都是基于 Socket 来完成通信的

**创建socket**

使用socket 模块的函数 socket socket.socket(AddressFamily, Type)

* Address Family：可以选择 AF\_INET（用于 Internet 进程间通信） 或者 AF\_UNIX（用于同一台机器进程间通信）,实际工作中常用AF\_INET
* Type：套接字类型，可以是 SOCK\_STREAM（流式套接字，主要用于 TCP 协议）或者 SOCK\_DGRAM（数据报套接字，主要用于 UDP 协议）

UDP --- 用户数据报协议，是一个无连接的简单的面向数据报的运输层协议

UDP是面向无连接的通讯协议，UDP数据包括目的端口号和源端口号信息，通讯不需要连接

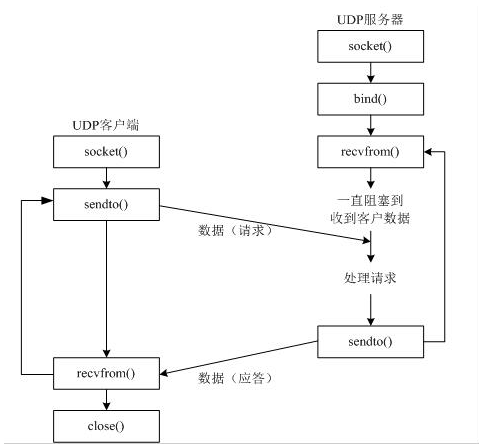
UDP通信双方分为客户端和服务器。

请求服务的一方称为：客户端。提供服务的一方称为：服务器

创建socket之后，服务器首先要绑定端口，目的是让客户端能够正确发送到此进程。

客户端一般不需要绑定，是让系统随机分配，就不会产生绑定端口被占用情况

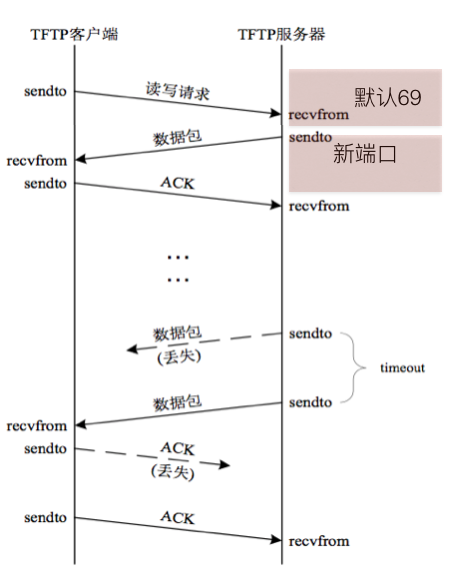
udpsocket=socket(AF\_INET,SOCK\_DGRAM)  
bindaddr=('',8002) # 匹配本地服务器ip和端口  
udpsocket.bind(bindaddr) #服务器端绑定端口

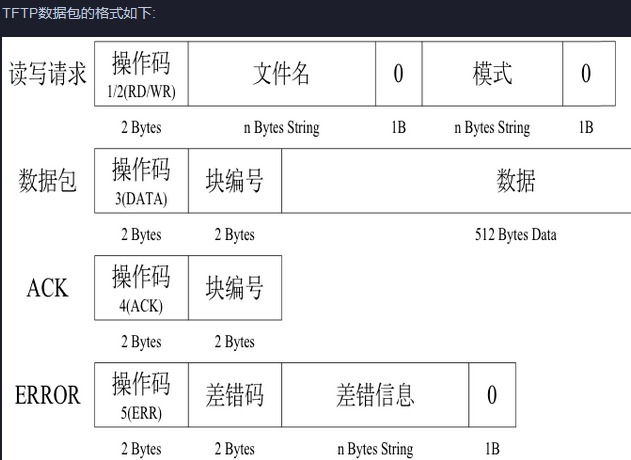


TFTP协议 （简单文件传输协议）

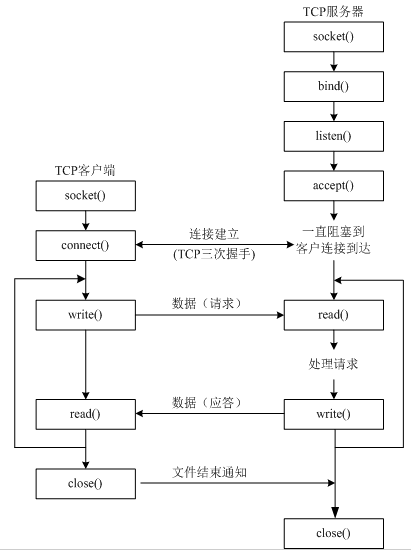
默认监听69号端口，客户端需要向服务器69端口发送下载（读）请求，服务器批准后使用一个新的临时端口进行数据传输。

客户端收到数据包后需要向服务器发送确认信息，称为ACK（应答包）为了标记数据已经发送完毕，所以规定，当客户端接收到的数据小于516（2字节操作码+2个字节的序号+512字节数据）时，就意味着服务器发送完毕了





**tcp协议**



**tcp服务器**

1. socket创建一个套接字
2. bind绑定ip和port
3. listen使套接字变为可以被动链接，传入的参数指定等待连接的最大数量，超过后排队
4. accept等待并返回一个客户端的链接
5. recv/send接收/发送数据

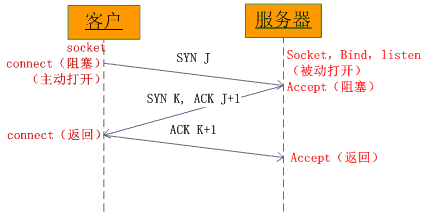
from socket import \*  
tcpsersocket=socket(AF\_INET,SOCK\_STREAM)  
tcpseraddr=('192.168.0.203',8001)  
tcpsersocket.bind(tcpseraddr) #绑定tcpserver的ip和端口  
tcpsersocket.listen(5) #listem将socket变为被动，接收连接  
while True: #循环接收来自client的连接  
 newsocket, clientaddr = tcpsersocket.accept() #新的客户端连接服务器，则对应产生一个新的socket  
 print(clientaddr)  
 recvdata = newsocket.recv(1024) #接收对方数据并打印  
 print(recvdata)  
 msg='received'  
 newsocket.send(msg.encode('utf-8')) #收到client信息后返回一条信息  
 newsocket.close() #关闭client的套接字，不再为这个client服务  
 tcpsersocket.close() #关闭监听套接字，不再接收任何client连接

每个链接必须创建新线程处理，单线程在处理连接过程中无法接受其他client连接

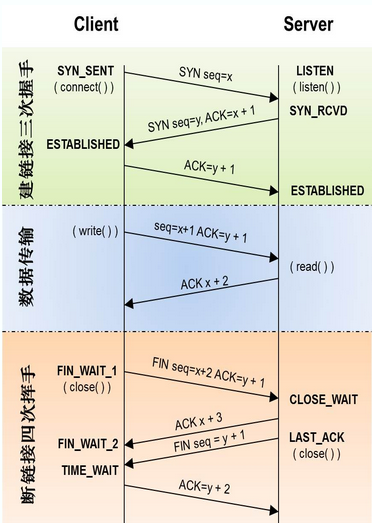
**tcp客户端**

from socket import \*  
tcpclisocket=socket(AF\_INET,SOCK\_STREAM)  
cliaddr=('192.168.0.203',8001)  
tcpclisocket.connect(cliaddr) #连接服务器  
while True:  
 senddata = input('输入：')  
 tcpclisocket.send(bytes(senddata, encoding='utf8'))   
 recvdata = tcpclisocket.recv(1024)  
 print('接收', recvdata)  
tcpclisocket.close()

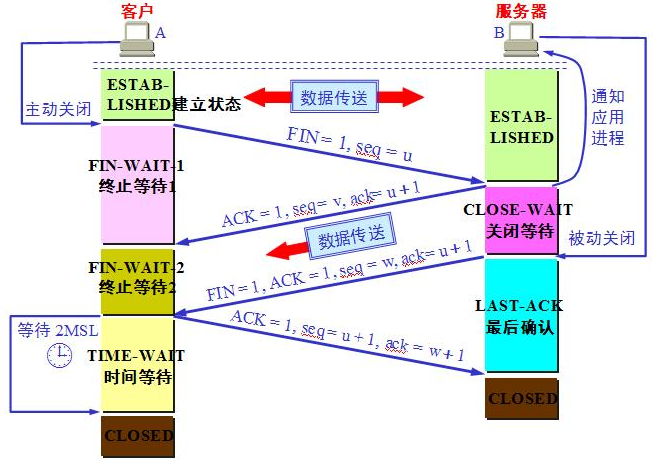
tcp三次握手



tcp四次挥手。连接的建立需要三次握手，释放需要四次握手。



tcp的2MSL



tcp的TIME\_WAIT状态称为2MSL（2MSL即两倍的MSL）等待状态，当TCP的一端发起主动关闭，在发出最后一个ACK包后，即第3次握手完成后发送了第四次握手的ACK包后就进入了TIME\_WAIT状态，必须在此状态上停留两倍的MSL时间，等待2MSL时间主要目的是怕最后一个 ACK包对方没收到，那么对方在超时后将重发第三次握手的FIN包，主动关闭端接到重发的FIN包后可以再发一个ACK应答包

在TIME\_WAIT状态 时两端的端口不能使用，要等到2MSL时间结束才可继续使用。当连接处于2MSL等待阶段时任何迟到的报文段都将被丢弃。不过在实际应用中可以通过设置 SO\_REUSEADDR选项达到不必等待2MSL时间结束再使用此端口。

tcp长连接和短连接

tcp短连接一般只会在client和server之间传递一次读写操作。

client向server发起连接请求，server接收后建立连接，client发送消息，server回应，close

一般是client先发起close操作

tcp长连接一次读写完毕后连接不关闭，继续读写操作，长时间操作后client发起关闭请求

进程和线程

“多任务”就是操作系统可以同时运行多个任务。

单核CPU操作系统轮流让各个任务交替执行，任务1执行0.01秒，切换到任务2执行0.01秒反复执行。表面上看，每个任务都是交替执行的，但是，由于CPU的执行速度实在是太快了，感觉就像所有任务都在同时执行一样。真正的并行执行多任务只能在多核CPU上实现，但是，由于任务数量远远多于CPU的核心数量，所以，操作系统也会自动把很多任务轮流调度到每个核心上执行。

一个任务就是一个进程（Process），比如打开一个浏览器就是启动一个浏览器进程。

有些进程不止同时干一件事，比如Word，它可以同时进行打字、拼写检查、打印等事情。在一个进程内部，要同时干多件事，就需要同时运行多个“子任务”，我们把进程内的这些“子任务”称为线程（Thread）。一个进程至少有一个线程。

**创建进程**

编写完毕的代码，在没有运行的时候，称之为程序，正在运行着的代码，就成为进程

fork（）只能在linux/unix操作系统中使用

Unix/Linux操作系统提供了一个fork()系统调用，它非常特殊。普通的函数调用，调用一次，返回一次，但是fork()调用一次，返回两次，因为操作系统自动把当前进程（称为父进程）复制了一份（称为子进程），然后，分别在父进程和子进程内返回。

子进程永远返回0，而父进程返回子进程的ID。这样做的理由是，一个父进程可以fork出很多子进程，所以，父进程要记下每个子进程的ID，而子进程只需要调用getppid()就可以拿到父进程的ID。getpid()子进程id、getppid()父进程id。

multiprocessing模块是跨平台版本的多进程模块。

multiprocessing模块提供了一个Process类来代表一个进程对象。

创建子进程时，只需要传入一个执行函数和函数的参数，创建一个Process实例，用start()方法启动。

join()方法可以等待子进程结束后再继续往下运行，通常用于进程间的同步。

def proc(name): #子进程要执行的代码

print('运行子进程 %s (%s)...' %(name, os.getpid())) #得到子进程id

p=Process(target=proc,args=('test',)) #传入函数名和参数  
print('子进程将要执行...')  
p.start()  
p.join()

print('子进程结束')

Process([group [, target [, name [, args [, kwargs]]]]])

target：表示这个进程实例所调用对象；

args：表示调用对象的位置参数元组；

kwargs：表示调用对象的关键字参数字典；

name：为当前进程实例的别名；

group：大多数情况下用不到；

Process类常用方法：

is\_alive()：判断进程实例是否还在执行；

join([timeout])：是否等待进程实例执行结束，或等待多少秒；

start()：启动进程实例（创建子进程）；

run()：如果没有给定target参数，对这个对象调用start()方法时，就将执行对象中的run()方法；

terminate()：不管任务是否完成，立即终止；

Process类常用属性：

name：当前进程实例别名，默认为Process-N，N为从1开始递增的整数；

pid：当前进程实例的PID值；

pool

如果要启动大量的子进程，可以用进程池的方式批量创建子进程

def task(name):  
 print('运行子进程%s(%s)'%(name,os.getpid()))  
 start=time()  
 sleep(random.random()\*3)  
 end=time()  
 print('子进程%s运行 %.2f 秒'%(name,(end-start))) #子进程01213是同时进行的，某个进程运行完毕后再运行子进程4  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print('父进程%s'%os.getpid())  
 p=Pool(4) #最多同时跑4个子进程，默认大小为cpu核数  
 for i in range(5):  
 p.apply\_async(task,args=(i,))  
 p.close() #调用close之后不能添加新的进程了  
 p.join()  
 print('所有进程完毕')

子进程

很多时候，子进程并不是自身，而是一个外部进程。我们创建了子进程后，还需要控制子进程的输入和输出。

subprocess模块启动一个子进程，控制其输入和输出。

子进程可以通过communicate()方法输入

**多线程（threading）**

多任务可以由多进程完成，也可以由一个进程内多线程完成。

启动一个线程就是把一个函数传入并创建Thread实例，然后调用start()开始执行。

import threading,time  
def loop1():  
 print('线程 %s 运行...'%threading.current\_thread().name) #显示子线程实例名字  
 n=0  
 while n<3:  
 n=n+1  
 print('%s >>> %s'%(threading.current\_thread().name,n)) #current\_thread()函数返回当前线程实例  
 time.sleep(0.5)  
 print('线程 %s 结束...'%threading.current\_thread().name)  
  
def loop2():  
 print('线程 %s 运行...'%threading.current\_thread().name)  
 n=0  
 while n<2:  
 n+=1  
 print('%s >>> %s' %(threading.current\_thread().name,n))  
 time.sleep(0.5)  
 print('线程 %s 结束...' % threading.current\_thread().name)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print('---开始---:%s'%time.ctime(),'\n')  
 # 显示当前线程，即主线程实例名字  
 print('%s 运行'%threading.current\_thread().name)  
 # 创建子线程，子线程名字在创建时指定，loopthread为子线程的名字，不指定为默认名  
 t=threading.Thread(target=loop1,name='子线程1')  
 p=threading.Thread(target=loop2,name='子线程2')  
 #子线程可以全部start之后再join，即为同步运行，分开执行，为运行完子线程1，再运行子线程2  
 t.start()  
 t.join()  
 p.start()  
 p.join()  
 while True:  
 length=len(threading.enumerate())  
 print('当前运行线程数量：%d'%length)  
 if length<=1:  
 break  
 print('thread %s ended'%threading.current\_thread().name)

任何进程默认就会启动一个线程，该线程称为主线程，主线程又可以启动新的线程， threading模块current\_thread()函数返回当前线程的实例。

lock

多线程和多进程最大的不同在于，多进程中，同一个变量，各自有一份拷贝存在于每个进程中，互不影响，而多线程中，所有变量都由所有线程共享，所以，任何一个变量都可以被任何一个线程修改，因此，线程之间共享数据最大的危险在于多个线程同时改一个变量

t.start()  
p.start()  
t.join()  
p.join()

多线程交替运行，每个线程有自己的局部变量，导致赋予全局变量的值会打乱，确保全局变量的值为正确值，要给change\_it()上一把锁，其他线程不能同时执行change\_it()，只能等待，直到锁被释放后，获得该锁以后才能改。锁只有一个，无论多少线程，同一时刻最多只有一个线程持有该锁，所以，不会造成修改的冲突。

def change\_it(n):  
 # 先存后取，结果应该为0:  
 global balance  
 balance = balance + n  
 balance = balance - n  
  
def thread(n):  
 for i in range(1000):  
 #获取锁  
 lock.acquire()  
 try:  
 change\_it(n)  
 finally:  
 #改完释放锁  
 lock.release()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 balance=0  
 lock=threading.Lock()  
 t1 = threading.Thread(target=thread, args=(5,))  
 t2 = threading.Thread(target=thread, args=(8,))  
 t1.start()  
 t2.start()  
 t1.join()  
 t2.join()  
 print(balance)

ThreadLocal: 解决参数在一个线程中各个函数之间互相传递的问题。

local\_data = threading.local() #创建全局ThreadLocal对象

local\_data为一个ThreadLocal对象，每个thread对其可读写属性，但互不影响。可以把local\_school看成全局变量，但每个属性如local\_data.data都是线程的局部变量，可以理解为全局变量local\_data是一个dict，不但可以用local\_data.data，还可以绑定其他变量，如local\_data.teacher等等可以任意读写而互不干扰，也不用管理锁的问题，ThreadLocal内部会处理。

ThreadLocal最常用的地方就是为每个线程绑定一个数据库连接，HTTP请求，用户身份信息等，这样一个线程的所有调用到的处理函数都可以非常方便地访问这些资源。

进程 vs 线程

多进程模式稳定性高，但是创建进程代价大，消耗内存和占用cpu多

多线程模式稳定性低，任何一个子线程挂掉会导致整个进程崩溃，因为所有线程共享进程的内存。

多任务一旦多到一个限度，就会消耗掉系统所有的资源，结果效率急剧下降，所有任务都做不好。因为线程切换需要保存当前执行的现场环境，同时将新任务的执行环境准备好。

异步IO

如果充分利用操作系统提供的异步IO支持，就可以用单进程单线程模型来执行多任务，这种全新的模型称为事件驱动模型

使用异步IO可以执行基于IO的多任务，基于CPU计算的多任务执行不了

分布式进程

在Thread和Process中，应当优选Process，因为Process更稳定，而且，Process可以分布到多台机器上，而Thread最多只能分布到同一台机器的多个CPU上。

Python的multiprocessing模块不但支持多进程，其中managers子模块还支持把多进程分布到多台机器上。一个服务进程可以作为调度者，将任务分布到其他多个进程中，依靠网络通信。

单进程服务器

同一时刻只能为一个客户端服务

多进程服务器

为每个客户端创建一个进程，适用于客户端不是非常多的时候

多线程服务器