IO并发编程

Tedu Python 教学部
Author：吕泽

IO

1. 定义

> IO指数据流的输入输出，从计算机应用层编程层面来说，在内存中存在数据交换的操作一般认为是IO操作,比如和终端交互 ,和磁盘交互，和网络交互等

2. 程序分类

> - IO密集型程序：在程序执行中有大量IO操作，而cpu运算较少。消耗cpu较少，耗时长。
>
> - 计算密集型程序：程序运行中计算较多，IO操作相对较少。cpu消耗多，执行速度快，几乎没有阻塞。

文件

文件是保存在持久化存储设备(硬盘、U盘、光盘..)上的一段数据。从格式编码角度分为文本文件（打开后会自动解码为字符）、二进制文件(视频、音频等)。在Python里把文件视作一种类型的对象，类似之前学习过的其它数据类型。

字节串（bytes）

在python3中引入了字节串的概念，与str不同，字节串以字节序列值表达数据，更方便用来处理二进程数据。因此在python3中字节串是常见的二进制数据展现方式。

- 普通的ascii编码字符串可以在前面加b转换为字节串，例如：b'hello'
- 字符串转换为字节串方法： str.encode()
- 字节串转换为字符串方法 : bytes.decode()

文件读写

对文件实现读写的基本操作步骤为：打开文件，读写文件，关闭文件

代码实现： day2/file_open.py
代码实现： day2/file_read.py

代码实现： day2/file_write.py

1. 打开文件

```
file_object = open(file_name, access_mode='r', buffering=-1)
功能：打开一个文件，返回一个文件对象。
参数：file_name   文件名；
     access_mode   打开文件的方式,如果不写默认为‘r’
          文件模式                          操作
              r                        以读方式打开  文件必须存在
              w                        以写方式打开
                                       文件不存在则创建，存在清空原有内容
              a                        以追加模式打开
              r+                       以读写模式打开  文件必须存在
              w+                       以读写模式打开文件
                                       不存在则创建，存在清空原有内容
              a+                       以读写模式打开  追加模式
              rb                       以二进制读模式打开  同r
              wb                       以二进制写模式打开  同w
              ab                       以二进制追加模式打开  同a
              rb+                      以二进制读写模式打开  同r+
              wb+                      以二进制读写模式打开  同w+
              ab+                      以二进制读写模式打开  同a+
     buffering   1表示有行缓冲，默认则表示使用系统默认提供的缓冲机制。
返回值：成功返回文件操作对象。
```

1. 读取文件

> read([size])
> 功能： 来直接读取文件中字符。
> 参数： 如果没有给定size参数（默认值为-1）或者size值为负，文件将被读取直至末尾，给定size最多读取给定数目个字符（字节）。
> 返回值： 返回读取到的内容
>
> - 注意：文件过大时候不建议直接读取到文件结尾，读到文件结尾会返回空字符串。

> readline([size])
> 功能： 用来读取文件中一行
> 参数： 如果没有给定size参数（默认值为-1）或者size值为负，表示读取一行，给定size表示最多读取制定的字符（字节）。
> 返回值： 返回读取到的内容

> readlines([sizeint])
> 功能： 读取文件中的每一行作为列表中的一项
> 参数： 如果没有给定size参数（默认值为-1）或者size值为负，文件将被读取直至末尾，给定size表示读取到size字符所在行为止。
> 返回值： 返回读取到的内容列表

> 文件对象本身也是一个可迭代对象，在for循环中可以迭代文件的每一行。

```
for line in f:
     print(line)
```

3. 写入文件

> write(string)
> 功能: 把文本数据或二进制数据块的字符串写入到文件中去
> 参数：要写入的内容
> 返回值：写入的字符个数
>
> - 如果需要换行要自己在写入内容中添加\n

> writelines(str_list)
> 功能：接受一个字符串列表作为参数，将它们写入文件。
> 参数: 要写入的内容列表

4. 关闭文件

打开一个文件后我们就可以通过文件对象对文件进行操作了，当操作结束后使用close（）关闭这个对象可以防止一些误操作，也可以节省资源。

> file_object.close()

5. with操作

python中的with语句使用于对资源进行访问的场合，保证不管处理过程中是否发生错误或者异常都会执行规定的“清理”操作，释放被访问的资源，比如有文件读写后自动关闭、线程中锁的自动获取和释放等。

with语句的语法格式如下：

```
with context_expression [as obj]:
    with-body
```

通过with方法可以不用close(),因为with生成的对象在语句块结束后会自动处理，所以也就不需要close了，但是这个文件对象只能在with语句块内使用。

```
with open('file','r+') as f:
    f.read()
```

> 注意
>
> > 1. 加b的打开方式读写要求必须都是字节串

> 2. 无论什么文件都可以使用二进制方式打开，但是二进制文件使用文本方式打开读写会出错

其他操作

刷新缓冲区

缓冲:系统自动的在内存中为每一个正在使用的文件开辟一个缓冲区，从内存向磁盘输出数据必须先送到内存缓冲区，再由缓冲区送到磁盘中去。从磁盘中读数据，则一次从磁盘文件将一批数据读入到内存缓冲区中，然后再从缓冲区将数据送到程序的数据区。

刷新缓冲区条件:

1. 缓冲区被写满
2. 程序执行结束或者文件对象被关闭
3. 行缓冲遇到换行
4. 程序中调用flush()函数

代码实现： day3/buffer.py

> flush()
> 该函数调用后会进行一次磁盘交互，将缓冲区中的内容写入到磁盘。

文件偏移量

代码实现： day3/seek.py

1. 定义

> 打开一个文件进行操作时系统会自动生成一个记录，记录中描述了我们对文件的一系列操作。其中包括每次操作到的文件位置。文件的读写操作都是从这个位置开始进行的。

2. 基本操作

> tell()
> 功能：获取文件偏移量大小

> seek(offset[,whence])
> 功能:移动文件偏移量位置
> 参数：offset 代表相对于某个位置移动的字节数。负数表示向前移动，正数表示向后移动。
> whence是基准位置的默认值为 0，代表从文件开头算起，1代表从当前位置算起，2 代表从文件末尾算起。
>
> - 必须以二进制方式打开文件时基准位置才能是1或者2

文件描述符

1. 定义

> 系统中每一个IO操作都会分配一个整数作为编号，该整数即这个IO操作的文件描述符。

2. 获取文件描述符

> fileno()
> 通过IO对象获取对应的文件描述符

常用文件操作函数

1. 获取文件大小

> os.path.getsize(file)

2. 查看文件列表

> os.listdir(dir)

3. 查看文件是否存在

> os.path.exists(file)

4. 判断文件类型

> os.path.isfile(file)

5. 删除文件

> os.remove(file)

网络编程基础

计算机网络功能主要包括实现资源共享，实现数据信息的快速传递。

OSI七层模型

> 制定组织： ISO（国际标准化组织）

> 作用：使网络通信工作流程标准化

> 应用层： 提供用户服务，具体功能有应用程序实现
> 表示层： 数据的压缩优化加密
> 会话层： 建立用户级的连接，选择适当的传输服务
> 传输层： 提供传输服务
> 网络层： 路由选择，网络互联

> 链路层： 进行数据交换，控制具体数据的发送
> 物理层： 提供数据传输的硬件保证，网卡接口，传输介质

> 优点
>
> 1. 建立了统一的工作流程
> 2. 分部清晰，各司其职，每个步骤分工明确
> 3. 降低了各个模块之间的耦合度，便于开发

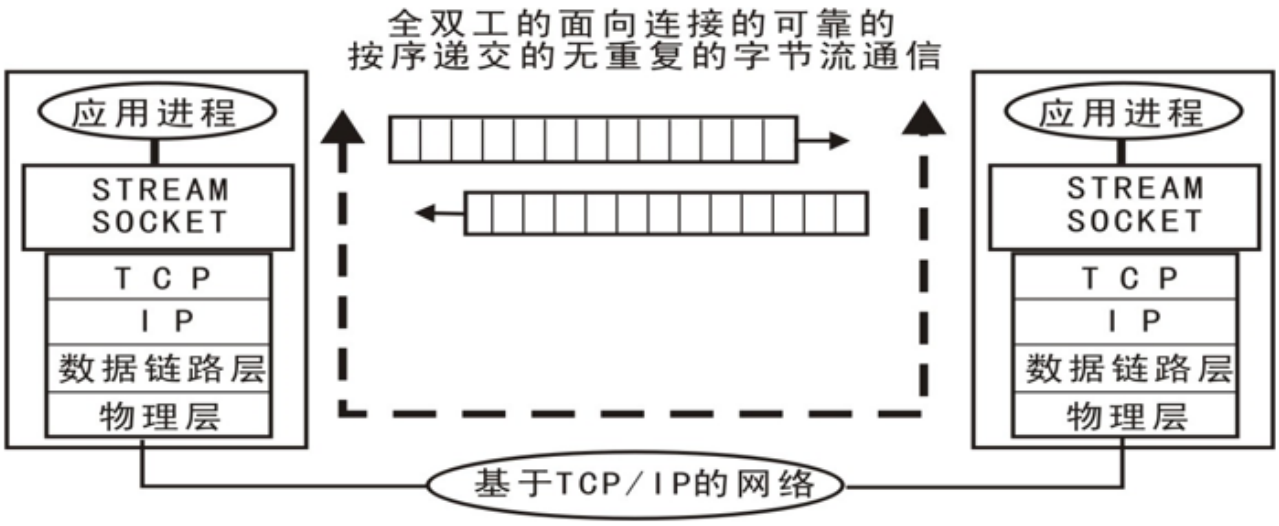
四层模型（TCP/IP模型）

背景： 实际工作中工程师无法完全按照七层模型要求操作，逐渐演化为更符合实际情况的四层

OSI七层网络模型	TCP/IP四层概念模型	对应网络协议
应用层（Application）	应用层	TFTP, FTP, NFS, WAIS http
表示层（Presentation）		Telnet, Rlogin, SNMP, Gopher
会话层（Session）		SMTP, DNS
传输层（Transport）	传输层	TCP, UDP
网络层（Network）	网际层	IP, ICMP, ARP, RARP, AKP, UUCP
数据链路层（Data Link）	网络接口	FDDI, Ethernet, Arpanet, PDN, SLIP, PPP
物理层（Physical）		IEEE 802.1A, IEEE 802.2到IEEE 802.11

数据传输过程

1. 发送端由应用程序发送消息，逐层添加首部信息，最终在物理层发送消息包。
2. 发送的消息经过多个节点（交换机，路由器）传输，最终到达目标主机。
3. 目标主机由物理层逐层解析首部消息包，最终到应用程序呈现消息。



网络协议

> 在网络数据传输中，都遵循的规定，包括建立什么样的数据结构，什么样的特殊标志等。

网络地址

- IP地址

> 功能：确定一台主机的网络路由位置

> 结构
>
> > IPv4 点分十进制表示 172.40.91.185 每部分取值范围0--255
> > IPv6 128位 扩大了地址范围
>
> 查看本机网络地址命令： ifconfig
>
> 测试和某个主机是否联通：ping [ip]

- 端口号（port）

> 作用：端口是网络地址的一部分，用于区分主机上不同的网络应用程序。

> 特点：一个系统中的应用监听端口不能重复

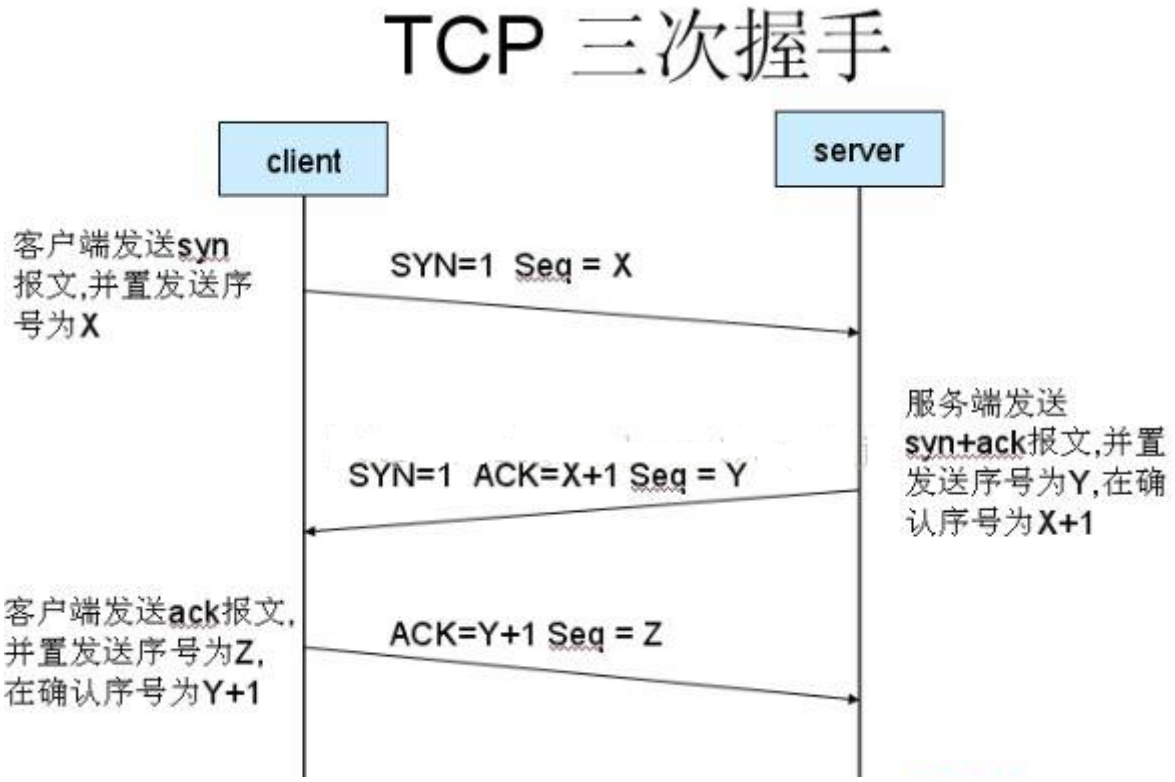
> 取值范围： 1 -- 65535
>
> > 1--1023 系统应用或者大众程序监听端口
> > 1024--65535 自用端口

传输层服务

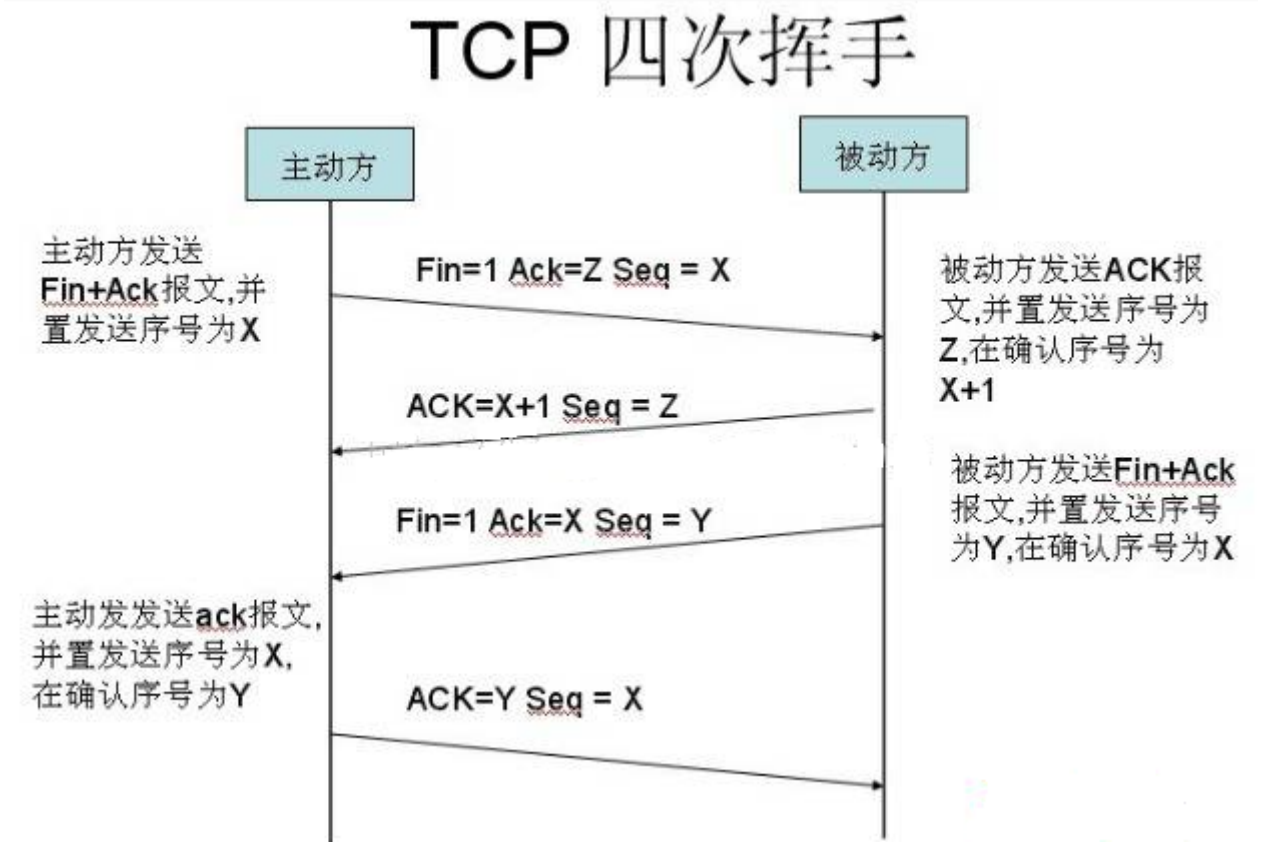
面向连接的传输服务（基于TCP协议的数据传输）

1. 传输特征： 提供了可靠的数据传输，可靠性指数据传输过程中无丢失，无失序，无差错，无重复。
2. 实现手段： 在通信前需要建立数据连接，通信结束要正常断开连接。

> 三次握手（建立连接）
>
> > 客户端向服务器发送消息报文请求连接
> > 服务器收到请求后，回复报文确定可以连接
> > 客户端收到回复，发送最终报文连接建立



> 四次挥手（断开连接）
>
> > 主动方发送报文请求断开连接
> > 被动方收到请求后，立即回复，表示准备断开
> > 被动方准备就绪，再次发送报文表示可以断开
> > 主动方收到确定，发送最终报文完成断开



3. 适用情况： 对数据传输准确性有明确要求，传数文件较大，需要确保可靠性的情况。比如：网页获取，文件下载，邮件收发。

面向无连接的传输服务（基于UDP协议的数据传输）

1. 传输特点： 不保证传输的可靠性，传输过程没有连接和断开，数据收发自由随意。
2. 适用情况： 网络较差，对传输可靠性要求不高。比如：网络视频，群聊，广播

面试要求

- OSI七层模型介绍一下，tcp/ip模型是什么?
- tcp服务和udp服务有什么区别?
- 三次握手和四次挥手指什么，过程是怎样的?

socket套接字编程

套接字介绍

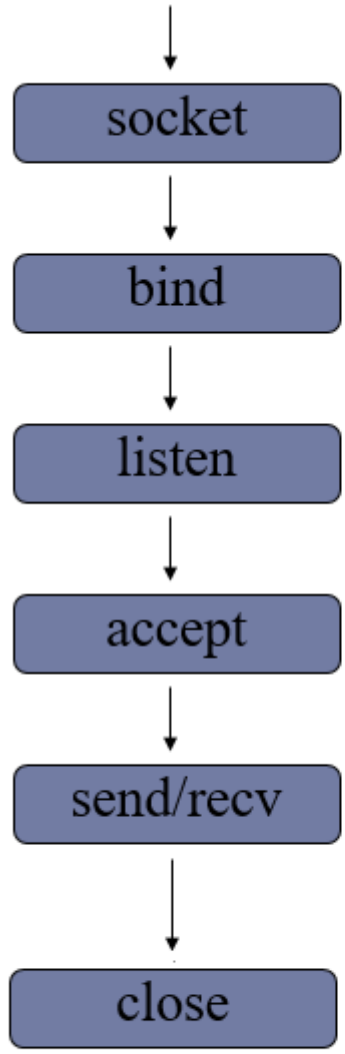
1. 套接字： 实现网络编程进行数据传输的一种技术手段
2. Python实现套接字编程：import socket
3. 套接字分类

> 流式套接字(SOCK_STREAM): 以字节流方式传输数据，实现tcp网络传输方案。(面向连接--tcp协议--可靠的--流式套接字)

> 数据报套接字(SOCK_DGRAM):以数据报形式传输数据，实现udp网络传输方案。(无连接--udp协议--不可靠--数据报套接字)

tcp套接字编程

服务端流程



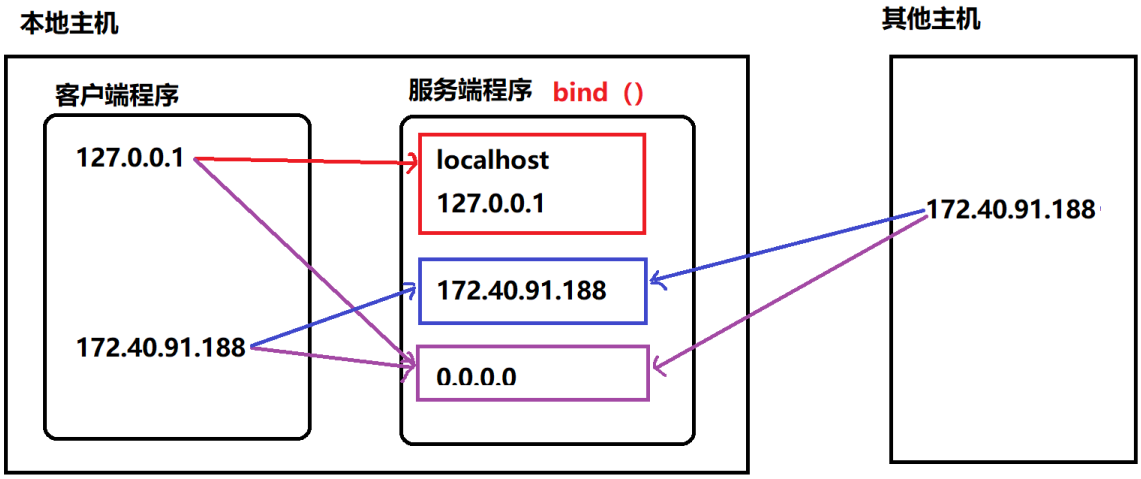
代码实现：day4/tcp_server.py

1. 创建套接字

```
sockfd=socket.socket(socket_family=AF_INET,socket_type=SOCK_STREAM,proto=0)
功能：创建套接字
参数：   socket_family  网络地址类型 AF_INET表示ipv4
         socket_type  套接字类型 SOCK_STREAM(流式)  SOCK_DGRAM(数据报)
         proto  通常为0  选择子协议
返回值：  套接字对象
```

2. 绑定地址

> 本地地址： 'localhost' , '127.0.0.1'
> 网络地址： '172.40.91.185'
> 自动获取地址： '0.0.0.0'




```
sockfd.bind(addr)
功能： 绑定本机网络地址
参数： 二元元组 (ip,port)  ('0.0.0.0',8888)
```

3. 设置监听

```
sockfd.listen(n)
功能 ： 将套接字设置为监听套接字，确定监听队列大小
参数 ： 监听队列大小
```

4. 等待处理客户端连接请求

```
connfd,addr = sockfd.accept()
功能： 阻塞等待处理客户端请求
返回值： connfd  客户端连接套接字
         addr  连接的客户端地址
```

5. 消息收发

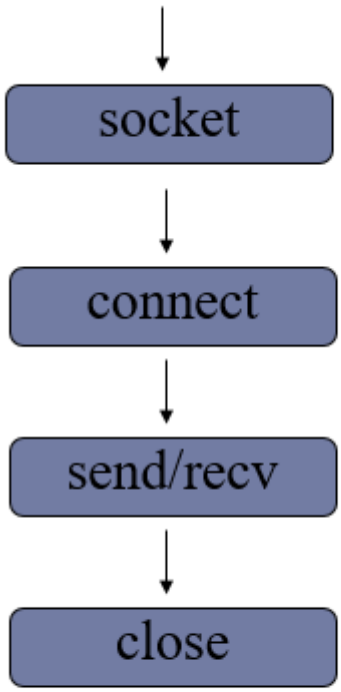
```
data = connfd.recv(buffersize)
功能 : 接受客户端消息
参数 : 每次最多接收消息的大小
返回值:  接收到的内容

n = connfd.send(data)
功能 : 发送消息
参数 : 要发送的内容  bytes格式
返回值:  发送的字节数
```

6. 关闭套接字

```
sockfd.close()
功能：关闭套接字
```

客户端流程

代码实现：day4/tcp_client.py



1. 创建套接字

> 注意:只有相同类型的套接字才能进行通信

2. 请求连接

```
sockfd.connect(server_addr)
功能：连接服务器
参数：元组  服务器地址
```

3. 收发消息

> 注意： 防止两端都阻塞，recv send要配合

4. 关闭套接字

tcp 套接字数据传输特点

> - tcp连接中当一端退出，另一端如果阻塞在recv，此时recv会立即返回一个空字串。
> - tcp连接中如果一端已经不存在，仍然试图通过send发送则会产生BrokenPipeError
> - 一个监听套接字可以同时连接多个客户端，也能够重复被连接

网络收发缓冲区

1. 网络缓冲区有效的协调了消息的收发速度
2. send和recv实际是向缓冲区发送接收消息，当缓冲区不为空recv就不会阻塞。

tcp粘包

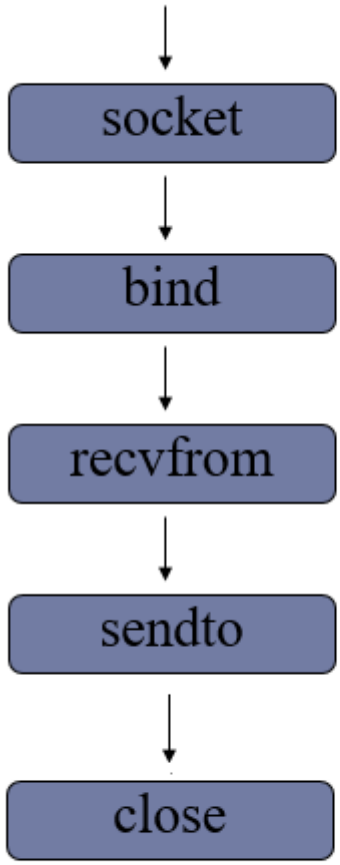
> 原因：tcp以字节流方式传输，没有消息边界。多次发送的消息被一次接收，此时就会形成粘包。

> 影响：如果每次发送内容是一个独立的含义，需要接收端独立解析此时粘包会有影响。

> 处理方法
>
> > 1. 人为的添加消息边界
> > 2. 控制发送速度

UDP套接字编程

服务端流程



代码实现：day4/udp_server.py

1. 创建数据报套接字

```
sockfd = socket(AF_INET,SOCK_DGRAM)
```

2. 绑定地址

```
sockfd.bind(addr)
```

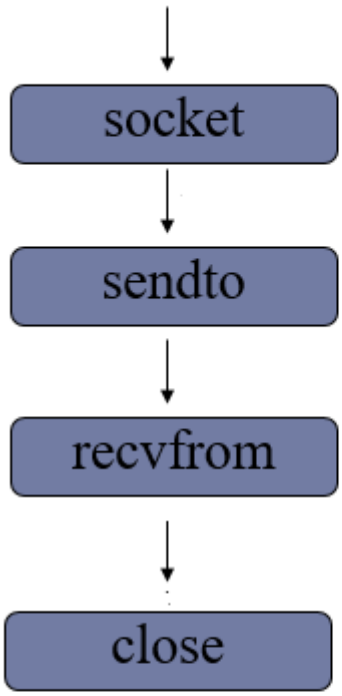
3. 消息收发

```
data,addr = sockfd.recvfrom(buffersize)
功能： 接收UDP消息
参数： 每次最多接收多少字节
返回值： data  接收到的内容
        addr  消息发送方地址

n = sockfd.sendto(data,addr)
功能： 发送UDP消息
参数： data  发送的内容 bytes格式
        addr  目标地址
返回值：发送的字节数
```

4. 关闭套接字

```
sockfd.close()
```

客户端流程



代码实现：day4/udp_client.py

1. 创建套接字
2. 收发消息
3. 关闭套接字

> 总结： tcp套接字和udp套接字编程区别
>
> 1. 流式套接字是以字节流方式传输数据，数据报套接字以数据报形式传输
> 2. tcp套接字会有粘包，udp套接字有消息边界不会粘包
> 3. tcp套接字保证消息的完整性，udp套接字则不能
> 4. tcp套接字依赖listen accept建立连接才能收发消息，udp套接字则不需要
> 5. tcp套接字使用send，recv收发消息，udp套接字使用sendto，recvfrom

socket套接字常用属性

代码实现：day4/sock_attr.py

【1】 sockfd.type 套接字类型

【2】 sockfd.family 套接字地址类型

【3】 sockfd.getsockname() 获取套接字绑定地址

【4】 sockfd.fileno() 获取套接字的文件描述符

【5】 sockfd.getpeername() 获取连接套接字客户端地址

【6】 sockfd.setsockopt(level,option,value)
功能：设置套接字选项
参数： level 选项类别 SOL_SOCKET
option 具体选项内容
value 选项值

选项	意义	期望值
SO_BINDTODEVICE	可以使socket只在某个特殊的网络接口（网卡）有效。也许不能是移动便携设备	一个字符串给出设备的名称或者一个空字符串返回默认值
SO_BROADCAST	允许广播地址发送和接收信息包。只对UDP有效。如何发送和接收广播信息包	布尔型整数
SO_DONTROUTE	禁止通过路由器和网关往外发送信息包。这主要是为了安全而用在以太网上UDP通信的一种方法。不管目的地址使用什么IP地址，都可以防止数据离开本地网络	布尔型整数
SO_KEEPALIVE	可以使TCP通信的信息包保持连续性。这些信息包可以在没有信息传输的时候，使通信的双方确定连接是保持的	布尔型整数
SO_OOBINLINE	可以把收到的不正常数据看成是正常的数据，也就是说会通过一个标准的对recv()的调用来接收这些数据	布尔型整数
SO_REUSEADDR	当socket关闭后，本地端用于该socket的端口号立刻就可以被重用。通常来说，只有经过系统定义一段时间后，才能被重用。	布尔型整数

struct模块进行数据打包

代码实现：day5/struct_recv.py
代码实现：day5/struct_send.py

1. 原理： 将一组简单数据进行打包，转换为bytes格式发送。或者将一组bytes格式数据，进行解析。
2. 接口使用

```
Struct(fmt)
功能：生成结构化对象
参数：fmt  定制的数据结构

st.pack(v1,v2,v3....)
功能：将一组数据按照指定格式打包转换为bytes
参数：要打包的数据
返回值： bytes字节串

st.unpack(bytes_data)
功能： 将bytes字节串按照指定的格式解析
参数： 要解析的字节串
返回值： 解析后的内容

struct.pack(fmt,v1,v2,v3...)
struct.unpack(fmt,bytes_data)
```

> 说明： 可以使用struct模块直接调用pack unpack。此时这两函数第一个参数传入fmt。其他用法功能相同

Format	C Type	Python type	Standard size	Notes
x	pad byte	no value		
c	char	bytes of length 1	1	
b	signed char	integer	1	(1),(3)
B	unsigned char	integer	1	(3)
?	_Bool	bool	1	(1)
h	short	integer	2	(3)
H	unsigned short	integer	2	(3)
i	int	integer	4	(3)
I	unsigned int	integer	4	(3)
l	long	integer	4	(3)
L	unsigned long	integer	4	(3)
q	long long	integer	8	(2), (3)
Q	unsigned long long	integer	8	(2), (3)
n	ssize_t	integer		(4)
N	size_t	integer		(4)
e	(7)	float	2	(5)
f	float	float	4	(5)
d	double	float	8	(5)
s	char[]	bytes		

HTTP传输

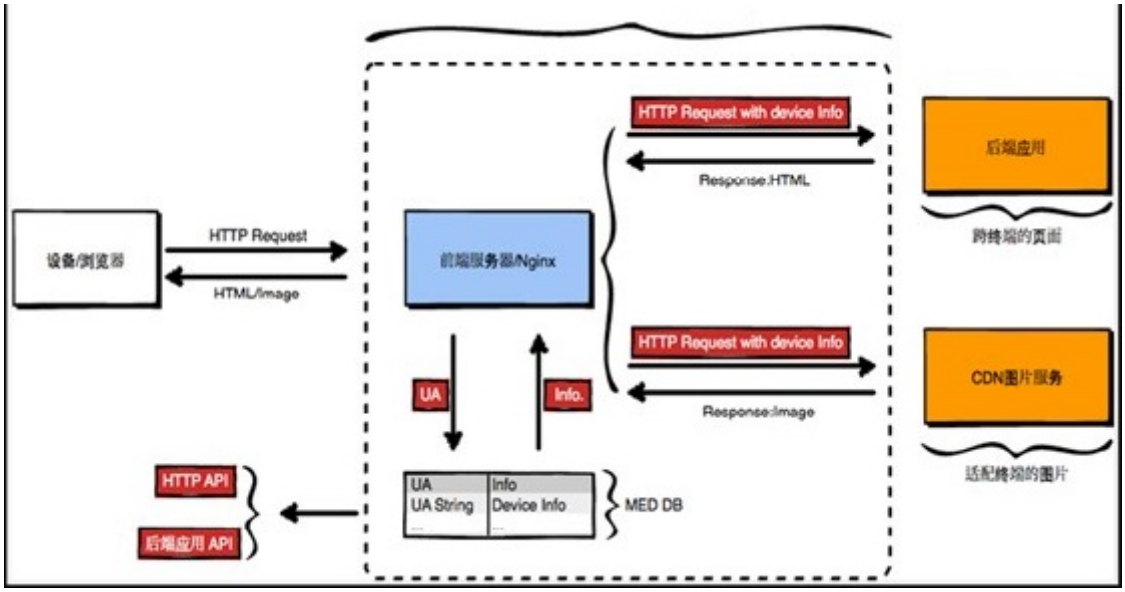
HTTP协议 （超文本传输协议）

1. 用途： 网页获取，数据的传输
2. 特点

> - 应用层协议，传输层使用tcp传输
> - 简单，灵活，很多语言都有HTTP专门接口
> - 无状态，协议不记录传输内容
> - http1.1 支持持久连接，丰富了请求类型

3. 网页请求过程

> 1.客户端（浏览器）通过tcp传输，发送http请求给服务端
> 2.服务端接收到http请求后进行解析
> 3.服务端处理请求内容，组织响应内容
> 4.服务端将响应内容以http响应格式发送给浏览器
> 5.浏览器接收到响应内容，解析展示



HTTP请求（request）

代码实现：day5/http_test.py
代码实现：day5/http_server.py

- 请求行： 具体的请求类别和请求内容

```
GET         /         HTTP/1.1
请求类别    请求内容      协议版本
```

请求类别：每个请求类别表示要做不同的事情

```
GET ： 获取网络资源
POST ：提交一定的信息，得到反馈
HEAD ：  只获取网络资源的响应头
PUT ：  更新服务器资源
DELETE ：  删除服务器资源
CONNECT
TRACE ：  测试
OPTIONS ：  获取服务器性能信息
```

- 请求头：对请求的进一步解释和描述

```
Accept-Encoding: gzip
```

- 空行
- 请求体: 请求参数或者提交内容

http响应（response）

1. 响应格式：响应行，响应头，空行，响应体

- 响应行： 反馈基本的响应情况

```
HTTP/1.1     200       OK
版本信息     响应码    附加信息
```

响应码：

```
1xx  提示信息，表示请求被接收
2xx  响应成功
3xx  响应需要进一步操作，重定向
4xx  客户端错误
5xx  服务器错误
```

- 响应头：对响应内容的描述

```
Content-Type: text/html
```

- 响应体：响应的主体内容信息

进程线程编程

1. 意义： 充分利用计算机CPU的多核资源，同时处理多个应用程序任务，以此提高程序的运行效率。
2. 实现方案： 多进程， 多线程

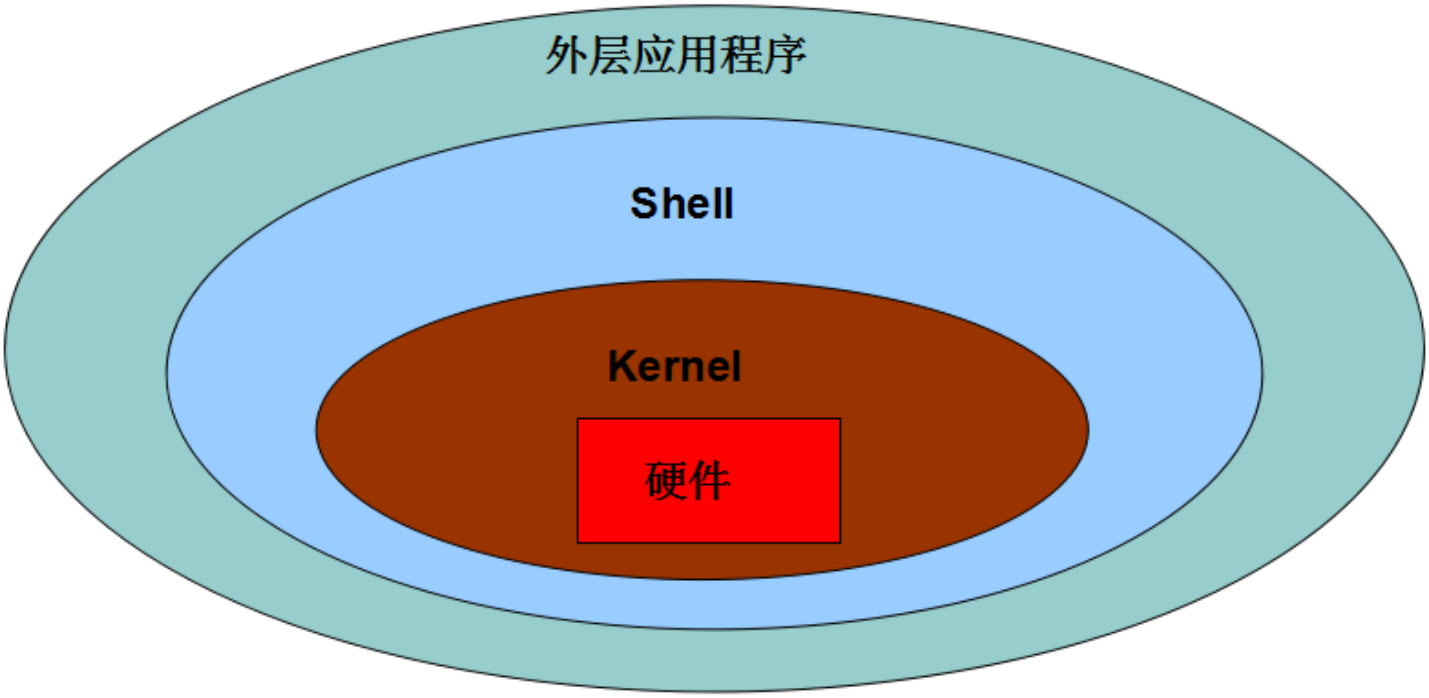
进程（process）

进程理论基础

1. 定义： 程序在计算机中的一次运行。

> - 程序是一个可执行的文件，是静态的占有磁盘。
> - 进程是一个动态的过程描述，占有计算机运行资源，有一定的生命周期。

2. 系统中如何产生一个进程
   【1】 用户空间通过调用程序接口或者命令发起请求
   【2】 操作系统接收用户请求，开始创建进程
   【3】 操作系统调配计算机资源，确定进程状态等
   【4】 操作系统将创建的进程提供给用户使用



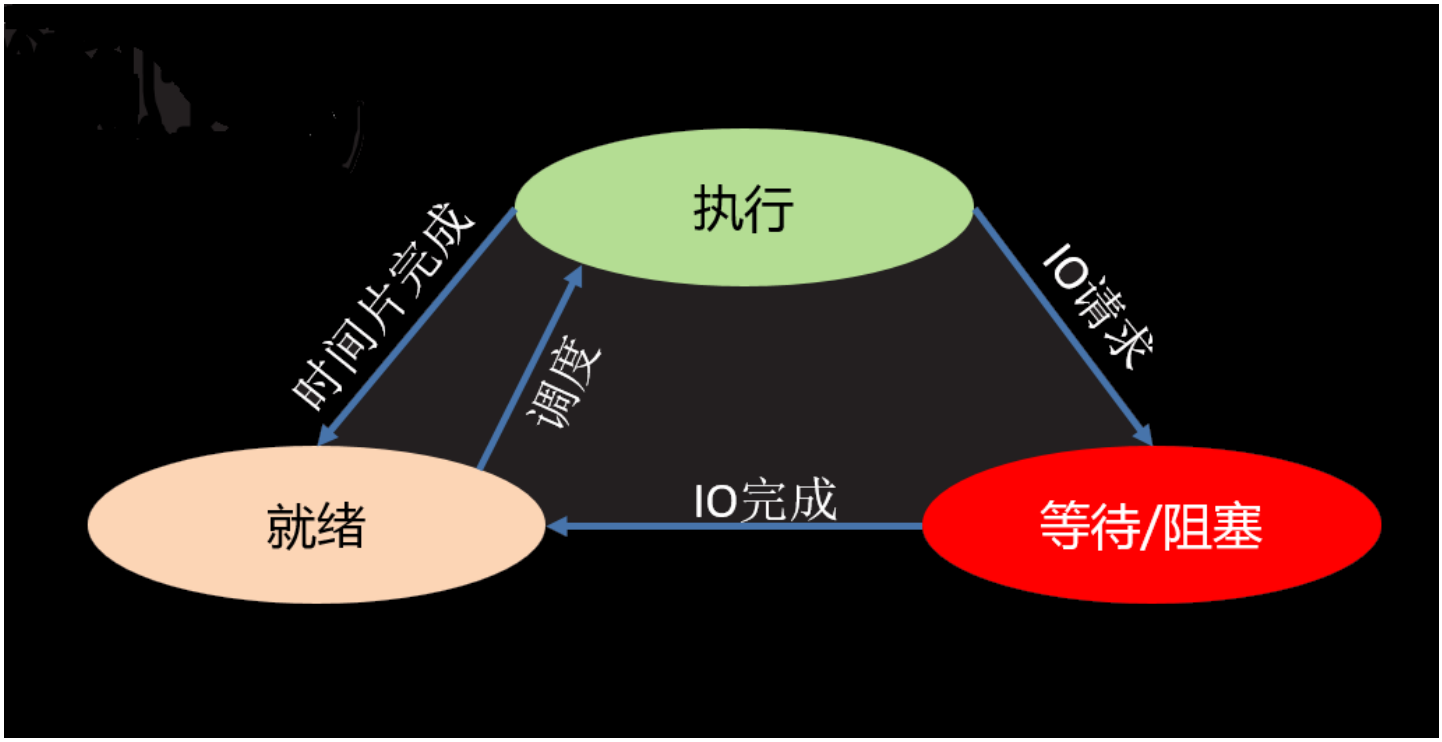
3. 进程基本概念

- cpu时间片：如果一个进程占有cpu内核则称这个进程在cpu时间片上。
- PCB(进程控制块)：在内存中开辟的一块空间，用于存放进程的基本信息，也用于系统查找识别进程。
- 进程ID（PID）： 系统为每个进程分配的一个大于0的整数，作为进程ID。每个进程ID不重复。
  > Linux查看进程ID： ps -aux
- 父子进程： 系统中每一个进程(除了系统初始化进程)都有唯一的父进程，可以有0个或多个子进程。父子进程关系便于进程管理。

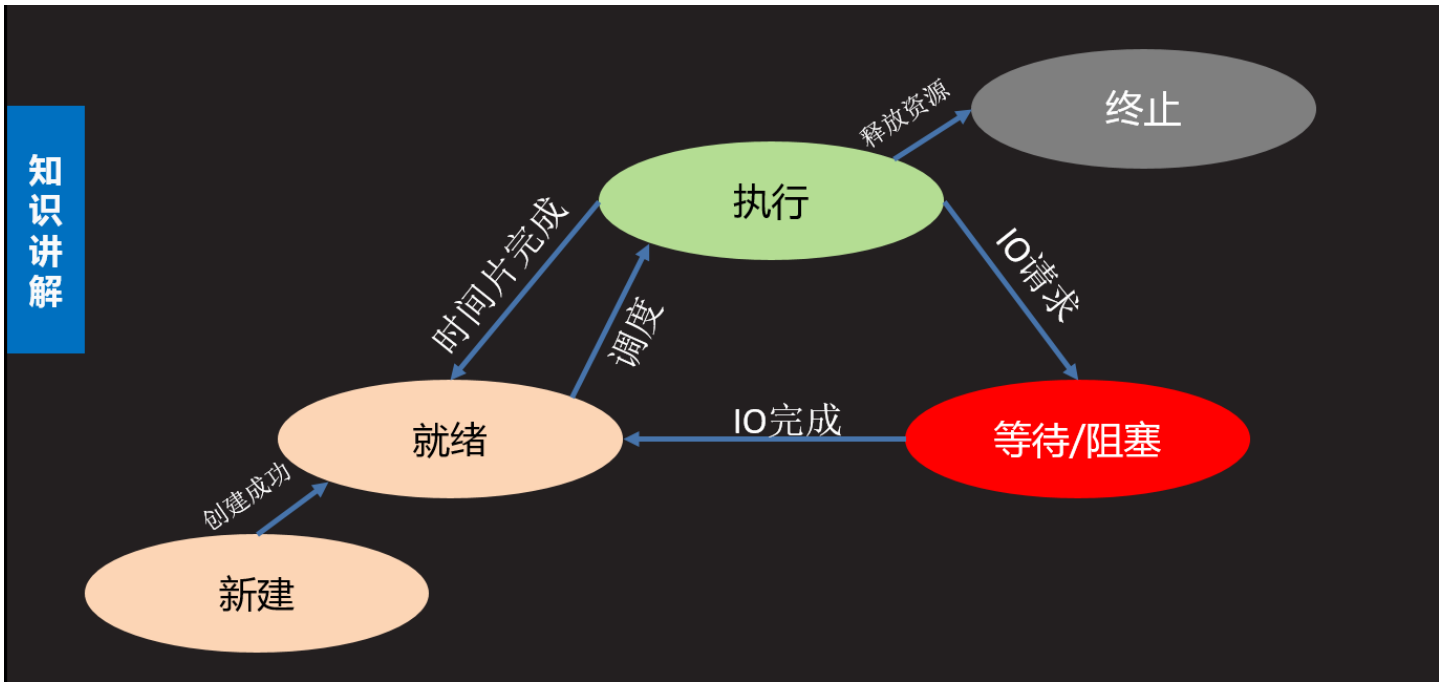
> 查看进程树： pstree

- 进程状态

- 三态
  就绪态： 进程具备执行条件，等待分配cpu资源
  运行态： 进程占有cpu时间片正在运行
  等待态： 进程暂时停止运行，让出cpu



- 五态 (在三态基础上增加新建和终止)
  新建： 创建一个进程，获取资源的过程
  终止： 进程结束，释放资源的过程



- 状态查看命令： ps -aux --> STAT列

> S 等待态
> R 执行态

> Z 僵尸

> \+ 前台进程
> l 有多线程的

- 进程的运行特征
  【1】 多进程可以更充分使用计算机多核资源
  【2】 进程之间的运行互不影响，各自独立
  【3】 每个进程拥有独立的空间，各自使用自己空间资源

> 面试要求
>
> 1. 什么是进程，进程和程序有什么区别
> 2. 进程有哪些状态，状态之间如何转化

基于fork的多进程编程

fork使用

代码示例：day5/fork.py
代码示例：day5/fork1.py

> pid = os.fork()
> 功能： 创建新的进程
> 返回值：整数，如果创建进程失败返回一个负数，如果成功则在原有进程中返回新进程的PID，在新进程中返回0

> 注意
>
> - 子进程会复制父进程全部内存空间，从fork下一句开始执行。
> - 父子进程各自独立运行，运行顺序不一定。
> - 利用父子进程fork返回值的区别，配合if结构让父子进程执行不同的内容几乎是固定搭配。
> - 父子进程有各自特有特征比如PID PCB 命令集等。
> - 父进程fork之前开辟的空间子进程同样拥有，父子进程对各自空间的操作不会相互影响。

进程相关函数

代码示例：day5/get_pid.py
代码示例：day5/exit.py

> os.getpid()
> 功能： 获取一个进程的PID值

> 返回值： 返回当前进程的PID

> os.getppid()
> 功能： 获取父进程的PID号
> 返回值： 返回父进程PID

> os._exit(status)
> 功能: 结束一个进程
> 参数：进程的终止状态

> sys.exit([status])
> 功能：退出进程
> 参数：整数 表示退出状态
> 字符串 表示退出时打印内容

孤儿和僵尸

1. 孤儿进程： 父进程先于子进程退出，此时子进程成为孤儿进程。

> 特点： 孤儿进程会被系统进程收养，此时系统进程就会成为孤儿进程新的父进程，孤儿进程退出该进程会自动处理。

2. 僵尸进程： 子进程先于父进程退出，父进程又没有处理子进程的退出状态，此时子进程就会称为僵尸进程。

> 特点： 僵尸进程虽然结束，但是会存留部分PCB在内存中，大量的僵尸进程会浪费系统的内存资源。

3. 如何避免僵尸进程产生

- 使用wait函数处理子进程退出

代码示例：day6/wait.py

```
pid,status = os.wait()
功能：在父进程中阻塞等待处理子进程退出
返回值： pid   退出的子进程的PID
         status   子进程退出状态
```

- 创建二级子进程处理僵尸

  代码示例：day6/child.py

  【1】 父进程创建子进程，等待回收子进程
  【2】 子进程创建二级子进程然后退出
  【3】 二级子进程称为孤儿，和原来父进程一同执行事件

- 通过信号处理子进程退出

> 原理： 子进程退出时会发送信号给父进程，如果父进程忽略子进程信号，则系统就会自动处理子进程退出。

> 方法： 使用signal模块在父进程创建子进程前写如下语句：

```
import signal
signal.signal(signal.SIGCHLD,signal.SIG_IGN)
```

> 特点： 非阻塞，不会影响父进程运行。可以处理所有子进程退出

群聊聊天室

> 功能： 类似qq群功能
> 【1】 有人进入聊天室需要输入姓名，姓名不能重复
> 【2】 有人进入聊天室时，其他人会收到通知：xxx 进入了聊天室
> 【3】 一个人发消息，其他人会收到：xxx： xxxxxxxxxx
> 【4】 有人退出聊天室，则其他人也会收到通知:xxx退出了聊天室
> 【5】 扩展功能：服务器可以向所有用户发送公告:管理员消息： xxxxxxxxx

multiprocessing 模块创建进程

进程创建方法

代码示例：day6/process1.py
代码示例：day6/process2.py
代码示例：day6/process3.py

1. 流程特点
   【1】 将需要子进程执行的事件封装为函数
   【2】 通过模块的Process类创建进程对象，关联函数
   【3】 可以通过进程对象设置进程信息及属性
   【4】 通过进程对象调用start启动进程
   【5】 通过进程对象调用join回收进程
2. 基本接口使用

```
Process()
功能 ： 创建进程对象
参数 ： target 绑定要执行的目标函数
        args 元组，用于给target函数位置传参
        kwargs 字典，给target函数键值传参
```

```
p.start()
功能 ： 启动进程
```

> 注意:启动进程此时target绑定函数开始执行，该函数作为子进程执行内容，此时进程真正被创建

```
p.join([timeout])
功能：阻塞等待回收进程
参数：超时时间
```

> 注意
>
> - 使用multiprocessing创建进程同样是子进程复制父进程空间代码段，父子进程运行互不影响。
> - 子进程只运行target绑定的函数部分，其余内容均是父进程执行内容。
> - multiprocessing中父进程往往只用来创建子进程回收子进程，具体事件由子进程完成。
> - multiprocessing创建的子进程中无法使用标准输入

3. 进程对象属性

代码示例：day7/process_attr.py

> p.name 进程名称

> p.pid 对应子进程的PID号

> p.is_alive() 查看子进程是否在生命周期

> p.daemon 设置父子进程的退出关系
>
> - 如果设置为True则子进程会随父进程的退出而结束
> - 要求必须在start()前设置
> - 如果daemon设置成True 通常就不会使用 join()

自定义进程类

代码示例：day7/myProcess.py

1. 创建步骤
   【1】 继承Process类
   【2】 重写 `__init__` 方法添加自己的属性，使用super()加载父类属性
   【3】 重写run()方法
2. 使用方法
   【1】 实例化对象
   【2】 调用start自动执行run方法
   【3】 调用join回收进程

进程池实现

代码示例：day7/pool.py

1. 必要性

   【1】 进程的创建和销毁过程消耗的资源较多

   【2】 当任务量众多，每个任务在很短时间内完成时，需要频繁的创建和销毁进程。此时对计算机压力较大

   【3】 进程池技术很好的解决了以上问题。

2. 原理

> 创建一定数量的进程来处理事件，事件处理完进 程不退出而是继续处理其他事件，直到所有事件全都处理完毕统一销毁。增加进程的重复利用，降低资源消耗。

3. 进程池实现

【1】 创建进程池对象，放入适当的进程

```
from multiprocessing import Pool

Pool(processes)
功能： 创建进程池对象
参数： 指定进程数量，默认根据系统自动判定
```

【2】 将事件加入进程池队列执行

```
pool.apply_async(func,args,kwds)
功能：使用进程池执行 func事件
参数： func 事件函数
      args 元组  给func按位置传参
      kwds 字典  给func按照键值传参
返回值： 返回函数事件对象
```

【3】 关闭进程池

```
pool.close()
功能： 关闭进程池
```

【4】 回收进程池中进程

```
pool.join()
功能： 回收进程池中进程
```

线程编程（Thread）

线程基本概念

1. 什么是线程
   【1】 线程被称为轻量级的进程
   【2】 线程也可以使用计算机多核资源，是多任务编程方式
   【3】 线程是系统分配内核的最小单元
   【4】 线程可以理解为进程的分支任务
2. 线程特征
   【1】 一个进程中可以包含多个线程
   【2】 线程也是一个运行行为，消耗计算机资源
   【3】 一个进程中的所有线程共享这个进程的资源
   【4】 多个线程之间的运行互不影响各自运行
   【5】 线程的创建和销毁消耗资源远小于进程
   【6】 各个线程也有自己的ID等特征

threading模块创建线程

代码示例：day7/thread1.py
代码示例：day7/thread2.py

【1】 创建线程对象

```
from threading import Thread

t = Thread()
功能：创建线程对象
参数：target 绑定线程函数
     args   元组 给线程函数位置传参
     kwargs 字典 给线程函数键值传参
```

【2】 启动线程

```
t.start()
```

【3】 回收线程

```
t.join([timeout])
```

线程对象属性

代码示例：day8/thread_attr.py

> t.name 线程名称
> t.setName() 设置线程名称

> t.getName() 获取线程名称

> t.is_alive() 查看线程是否在生命周期

> t.daemon 设置主线程和分支线程的退出关系
> t.setDaemon() 设置daemon属性值
> t.isDaemon() 查看daemon属性值
>
> > daemon为True时主线程退出分支线程也退出。要在start前设置，通常不和join一起使用。

自定义线程类

代码示例：day8/myThread.py

1. 创建步骤
   【1】 继承Thread类
   【2】 重写 `__init__` 方法添加自己的属性，使用super()加载父类属性
   【3】 重写run()方法
2. 使用方法
   【1】 实例化对象
   【2】 调用start自动执行run方法
   【3】 调用join回收线程

同步互斥

线程间通信方法

1. 通信方法

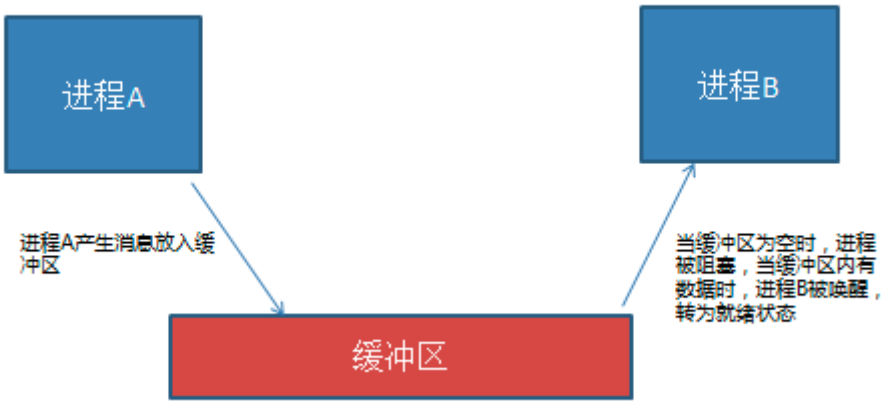
> 线程间使用全局变量进行通信

2. 共享资源争夺

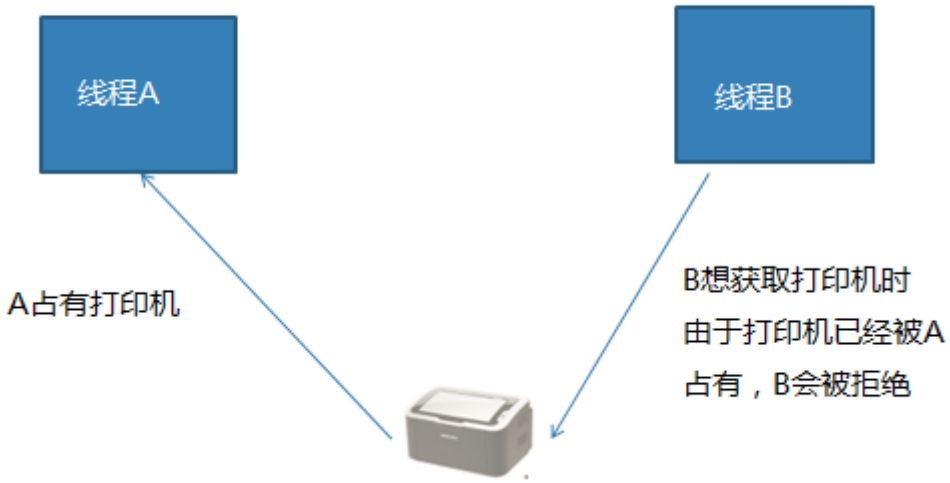
- 共享资源：多个进程或者线程都可以操作的资源称为共享资源。对共享资源的操作代码段称为临界区。
- 影响： 对共享资源的无序操作可能会带来数据的混乱，或者操作错误。此时往往需要同步互斥机制协调操作顺序。

3. 同步互斥机制

> 同步： 同步是一种协作关系，为完成操作，多进程或者线程间形成一种协调，按照必要的步骤有序执行操作。



> 互斥： 互斥是一种制约关系，当一个进程或者线程占有资源时会进行加锁处理，此时其他进程线程就无法操作该资源，直到解锁后才能操作。



线程同步互斥方法

线程Event

代码示例：day8/thread_event.py

```
from threading import Event

e = Event()  创建线程event对象

e.wait([timeout])  阻塞等待e被set

e.set()  设置e，使wait结束阻塞

e.clear() 使e回到未被设置状态

e.is_set()  查看当前e是否被设置
```

线程锁 Lock

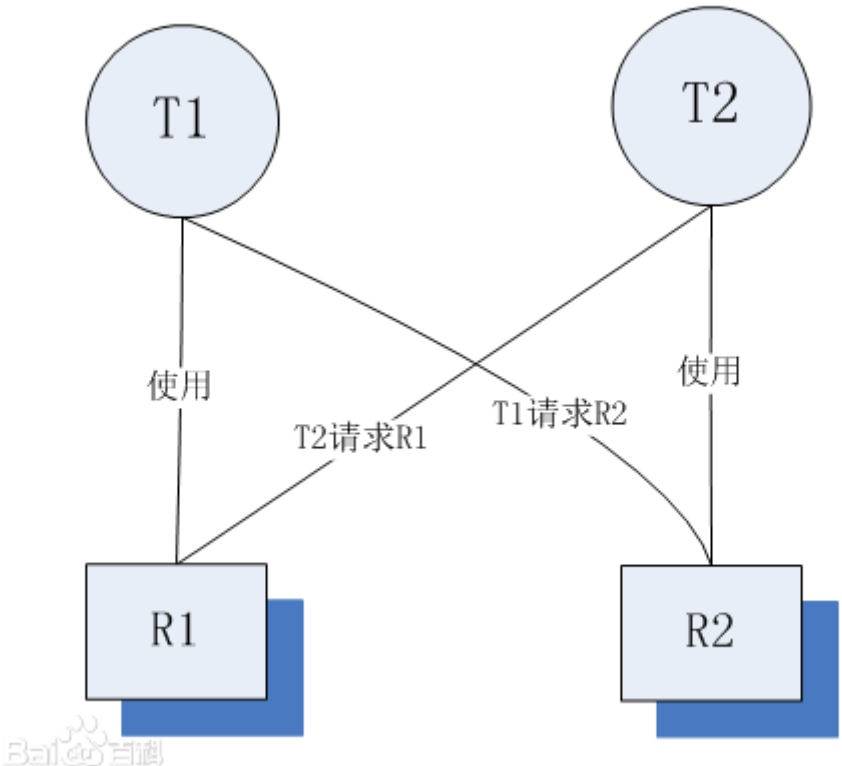
代码示例：day8/thread_lock.py

```
from  threading import Lock

lock = Lock()  创建锁对象
lock.acquire() 上锁  如果lock已经上锁再调用会阻塞
lock.release() 解锁

with  lock:  上锁
...
...
        with代码块结束自动解锁
```

死锁及其处理

1. 定义

> 死锁是指两个或两个以上的线程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁。



2. 死锁产生条件

代码示例: day8/dead_lock.py

> 死锁发生的必要条件
>
> > - 互斥条件：指线程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。
> > - 请求和保持条件：指线程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求线程阻塞，但又对自己已获得的其它资源保持不放。
> > - 不剥夺条件：指线程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放,通常CPU内存资源是可以被系统强行调配剥夺的。
> > - 环路等待条件：指在发生死锁时，必然存在一个线程——资源的环形链，即进程集合{T0，T1，T2，…，Tn}中的T0正在等待一个T1占用的资源；T1正在等待T2占用的资源，……，Tn正在等待已被T0占用的资源。

> 死锁的产生原因
>
> > 简单来说造成死锁的原因可以概括成三句话：
> >
> > - 当前线程拥有其他线程需要的资源
> > - 当前线程等待其他线程已拥有的资源
> > - 都不放弃自己拥有的资源

3. 如何避免死锁

死锁是我们非常不愿意看到的一种现象，我们要尽可能避免死锁的情况发生。通过设置某些限制条件，去破坏产生死锁的四个必要条件中的一个或者几个，来预防发生死锁。预防死锁是一种较易实现的方法。但是由于所施加的限制条件往往太严格，可能会导致系统资源利用率。

python线程GIL

1. python线程的GIL问题 （全局解释器锁）

> 什么是GIL ： 由于python解释器设计中加入了解释器锁，导致python解释器同一时刻只能解释执行一个线程，大大降低了线程的执行效率。

> 导致后果： 因为遇到阻塞时线程会主动让出解释器，去解释其他线程。所以python多线程在执行多阻塞高延迟IO时可以提升程序效率，其他情况并不能对效率有所提升。

> GIL问题建议
>
> > - 尽量使用进程完成无阻塞的并发行为
> > - 不使用c作为解释器 （Java C#）

2. 结论： 在无阻塞状态下，多线程程序和单线程程序执行效率几乎差不多，甚至还不如单线程效率。但是多进程运行相同内容却可以有明显的效率提升。

进程线程的区别联系

区别联系

1. 两者都是多任务编程方式，都能使用计算机多核资源
2. 进程的创建删除消耗的计算机资源比线程多
3. 进程空间独立，数据互不干扰，有专门通信方法；线程使用全局变量通信
4. 一个进程可以有多个分支线程，两者有包含关系
5. 多个线程共享进程资源，在共享资源操作时往往需要同步互斥处理
6. 进程线程在系统中都有自己的特有属性标志，如ID,代码段，命令集等。

使用场景

1. 任务场景：如果是相对独立的任务模块，可能使用多进程，如果是多个分支共同形成一个整体任务可能用多线程
2. 项目结构：多种编程语言实现不同任务模块，可能是多进程，或者前后端分离应该各自为一个进程。
3. 难易程度：通信难度，数据处理的复杂度来判断用进程间通信还是同步互斥方法。

要求

1. 对进程线程怎么理解/说说进程线程的差异
2. 进程间通信知道哪些，有什么特点
3. 什么是同步互斥，你什么情况下使用，怎么用
4. 给一个情形，说说用进程还是线程，为什么
5. 问一些概念，僵尸进程的处理，GIL问题，进程状态

网络并发通信

常见网络通信模型

1. 循环服务器模型 ： 循环接收客户端请求，处理请求。同一时刻只能处理一个请求，处理完毕后再处理下一个。

> 优点：实现简单，占用资源少
> 缺点：无法同时处理多个客户端请求

> 适用情况：处理的任务可以很快完成，客户端无需长期占用服务端程序。udp比tcp更适合循环。

2. 多进程/线程网络并发模型：每当一个客户端连接服务器，就创建一个新的进程/线程为该客户端服务，客户端退出时再销毁该进程/线程。

> 优点：能同时满足多个客户端长期占有服务端需求，可以处理各种请求。
> 缺点： 资源消耗较大

> 适用情况：客户端同时连接量较少，需要处理行为较复杂情况。

3. IO并发模型：利用IO多路复用,异步IO等技术，同时处理多个客户端IO请求。
   > 优点： 资源消耗少，能同时高效处理多个IO行为
   > 缺点： 只能处理并发产生的IO事件，无法处理cpu计算

   > 适用情况：HTTP请求，网络传输等都是IO行为。

基于fork的多进程网络并发模型

代码实现: day9/fork_server.py

实现步骤

1. 创建监听套接字
2. 等待接收客户端请求
3. 客户端连接创建新的进程处理客户端请求
4. 原进程继续等待其他客户端连接
5. 如果客户端退出，则销毁对应的进程

基于threading的多线程网络并发

代码实现: day9/thread_server.py

实现步骤

1. 创建监听套接字
2. 循环接收客户端连接请求
3. 当有新的客户端连接创建线程处理客户端请求
4. 主线程继续等待其他客户端连接
5. 当客户端退出，则对应分支线程退出

ftp 文件服务器

代码实现: day9/ftp

1. 功能
   【1】 分为服务端和客户端，要求可以有多个客户端同时操作。
   【2】 客户端可以查看服务器文件库中有什么文件。
   【3】 客户端可以从文件库中下载文件到本地。
   【4】 客户端可以上传一个本地文件到文件库。
   【5】 使用print在客户端打印命令输入提示，引导操作

IO并发

IO 分类

> IO分类：阻塞IO ， 非阻塞IO，IO多路复用，异步IO等

阻塞IO

1.定义：在执行IO操作时如果执行条件不满足则阻塞。阻塞IO是IO的默认形态。

2.效率：阻塞IO是效率很低的一种IO。但是由于逻辑简单所以是默认IO行为。

3.阻塞情况：

- 因为某种执行条件没有满足造成的函数阻塞
  e.g. accept input recv
- 处理IO的时间较长产生的阻塞状态
  e.g. 网络传输，大文件读写

非阻塞IO

代码实现: day9/block_io

1. 定义 ： 通过修改IO属性行为，使原本阻塞的IO变为非阻塞的状态。

- 设置套接字为非阻塞IO

> sockfd.setblocking(bool)
> 功能：设置套接字为非阻塞IO
> 参数：默认为True，表示套接字IO阻塞；设置为False则套接字IO变为非阻塞

- 超时检测 ： 设置一个最长阻塞时间，超过该时间后则不再阻塞等待。
  > sockfd.settimeout(sec)
  > 功能：设置套接字的超时时间
  > 参数：设置的时间

IO多路复用

1. 定义

> 同时监控多个IO事件，当哪个IO事件准备就绪就执行哪个IO事件。以此形成可以同时处理多个IO的行为，避免一个IO阻塞造成其他IO均无法执行，提高了IO执行效率。

2. 具体方案

> select方法： windows linux unix
> poll方法： linux unix
> epoll方法： linux

select 方法

代码实现: day10/select_server.py

```
rs, ws, xs=select(rlist, wlist, xlist[, timeout])
功能：监控IO事件，阻塞等待IO发生
参数：rlist  列表  读IO列表，添加等待发生的或者可读的IO事件
      wlist  列表  写IO列表，存放要可以主动处理的或者可写的IO事件
      xlist  列表 异常IO列表，存放出现异常要处理的IO事件
      timeout  超时时间

返回值： rs 列表  rlist中准备就绪的IO
        ws 列表  wlist中准备就绪的IO
           xs 列表  xlist中准备就绪的IO
```

select 实现tcp服务

```
【1】将关注的IO放入对应的监控类别列表
【2】通过select函数进行监控
【3】遍历select返回值列表，确定就绪IO事件
【4】处理发生的IO事件
```

> 注意
>
> > wlist中如果存在IO事件，则select立即返回给ws
> > 处理IO过程中不要出现死循环占有服务端的情况
> > IO多路复用消耗资源较少，效率较高

@@扩展: 位运算

定义： 将整数转换为二进制，按二进制位进行运算

运算符号:

> ```
>  &  按位与
> ```
>
> | 按位或
> ^ 按位异或
> << 左移
> >> 右移

```
e.g.  14 --> 01110
      19 --> 10011

14 & 19 = 00010 = 2  一0则0
14 | 19 = 11111 = 31 一1则1
14 ^ 19 = 11101 = 29 相同为0不同为1
14 << 2 = 111000 = 56 向左移动低位补0
14 >> 2 = 11 = 3  向右移动去掉低位
```

poll方法

代码实现: day10/poll_server.py

```
p = select.poll()
功能 ：  创建poll对象
返回值：  poll对象


p.register(fd,event)
功能：注册关注的IO事件
参数：fd  要关注的IO
      event  要关注的IO事件类型
             常用类型：POLLIN  读IO事件（rlist）
                      POLLOUT 写IO事件 (wlist)
                      POLLERR 异常IO  （xlist）
                      POLLHUP 断开连接
                 e.g. p.register(sockfd,POLLIN|POLLERR)

p.unregister(fd)
功能：取消对IO的关注
参数：IO对象或者IO对象的fileno


events = p.poll()
功能：  阻塞等待监控的IO事件发生
返回值：  返回发生的IO
        events格式  [(fileno,event),()....]
        每个元组为一个就绪IO，元组第一项是该IO的fileno，第二项为该IO就绪的事件类型
```

poll_server 步骤

```
【1】 创建套接字
【2】 将套接字register
【3】 创建查找字典，并维护
【4】 循环监控IO发生
【5】 处理发生的IO
```

epoll方法

代码实现: day10/epoll_server.py

1. 使用方法： 基本与poll相同
    - 生成对象改为 epoll()
    - 将所有事件类型改为EPOLL类型
2. epoll特点
    - epoll 效率比select poll要高
    - epoll 监控IO数量比select要多
    - epoll 的触发方式比poll要多 （EPOLLET边缘触发）