并发网络编程

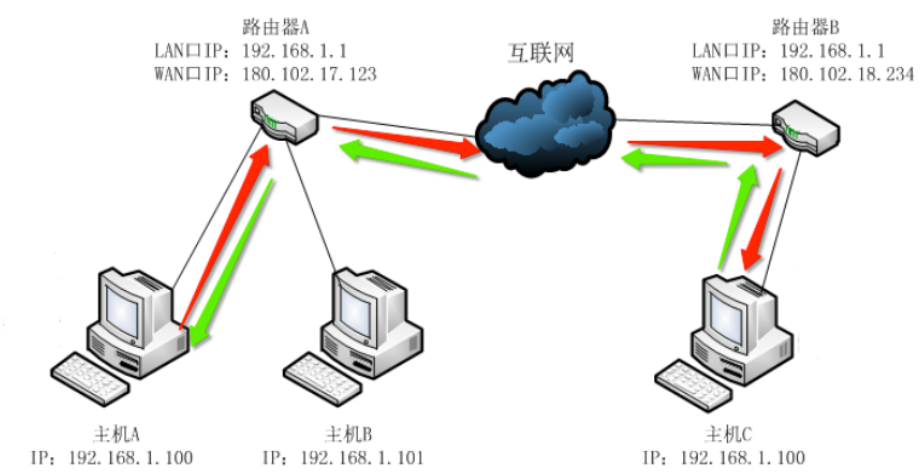
Tedu Python 教学部
Author：吕泽

1. 网络编程

1.1 网络基础知识

1.1.1 什么是网络

- 什么是网络 : 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。



1.1.2 网络通信标准

- 面临问题
  1. 不同的国家和公司都建立自己的通信标准不利于网络互连
  2. 多种标准并行情况下不利于技术的发展融合

- OSI 7层模型

  - 好处
    1. 建立了统一的通信标准
    2. 降低开发难度，每层功能明确，各司其职
    3. 七层模型实际规定了每一层的任务，该完成什么事情

- 网络协议
  - 什么是网络协议：在网络数据传输中，都遵循的执行规则。
  - 网络协议实际上规定了每一层在完成自己的任务时应该遵循什么规范。

- TCP/IP模型
  - 七层模型过于理想，结构细节太复杂
  - 在工程中应用实践难度大
  - 实际工作中以TCP/IP模型为工作标准流程

<table>
<tr><th>OSI</th><th colspan="2">TCP/IP协议集</th></tr>
<tr><td>应用层</td><td rowspan="3">应用层</td><td rowspan="3">Telnet，FTP，SMTP，DNS，HTTP
以及其他应用协议</td></tr>
<tr><td>表示层</td></tr>
<tr><td>会话层</td></tr>
<tr><td>传输层</td><td>传输层</td><td>TCP，　UDP</td></tr>
<tr><td>网络层</td><td>网络层</td><td>IP，ARP，RARP，ICMP</td></tr>
<tr><td>数据链路层</td><td rowspan="2">网络接口</td><td rowspan="2">各种通信网络接口（以太网等）
（物理网络）</td></tr>
<tr><td>物理层</td></tr>
</table>

  - 需要应用工程师做的工作： 编写应用工功能，明确对方地址，选择传输服务。

1.1.3 通信地址

- IP地址
  - IP地址： 即在网络中标识一台计算机的地址编号。
  - IP地址分类
    - IPv4： 192.168.1.5
    - IPv6： fe80::80a:76cf:ab11:2d73
  - IPv4 特点
    - 分为4个部分，每部分是一个整数，取值分为0-255
  - IPv6 特点（了解）
    - 分为8个部分，每部分4个16进制数，如果出现连续的数字 0 则可以用：：省略中间的0
  - IP地址相关命令
    - ifconfig : 查看Linux系统下计算机的IP地址

```
tarena@tedu:~$ ifconfig
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.94.136  本机IP地址        55.0  broadcast 192.168.94.255
        inet6 fe80::80a:76cf:ab11:2d73  IPv6地址          peid 0x20<link>
        ether 00:0c:29:a0:7d:55  txqueuelen 1000  (以太网)
        RX packets 82512  bytes 10750221 (10.7 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 14248  bytes 1308719 (1.3 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  用于本机本地测试地址
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (本地环回)
        RX packets 801580  bytes 102760244 (102.7 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 801580  bytes 102760244 (102.7 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

- ping [ip]：查看计算机的连通性

```
tarena@tedu:~$ ping www.baidu.com
PING www.a.shifen.com (182.61.200.7) 56(84) bytes of data.
64 bytes from 182.61.200.7 (182.61.200.7): icmp_seq=1 ttl=128 time=24.0 ms
64 bytes from 182.61.200.7 (182.61.200.7): icmp_seq=2 ttl=128 time=5.39 ms
64 bytes from 182.61.200.7 (182.61.200.7): icmp_seq=3 ttl=128 time=7.02 ms
^C
--- www.a.shifen.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 5.398/12.174/24.099/8.458 ms
```

- 公网IP和内网IP
  - 公网IP指的是连接到互联网上的公共IP地址，大家都可以访问。（将来进公司，公司会申请公网IP作为网络项目的被访问地址）
  - 内网IP指的是一个局域网络范围内由网络设备分配的IP地址。

- 端口号
  - 端口：网络地址的一部分，在一台计算机上，每个网络程序对应一个端口。

  - 端口号特点

- 取值范围： 0 —— 65535 的整数
- 一台计算机上的网络应用所使用的端口不会重复
- 通常 0——1023 的端口会被一些有名的程序或者系统服务占用，个人一般使用 > 1024的端口

1.2 UDP 传输方法

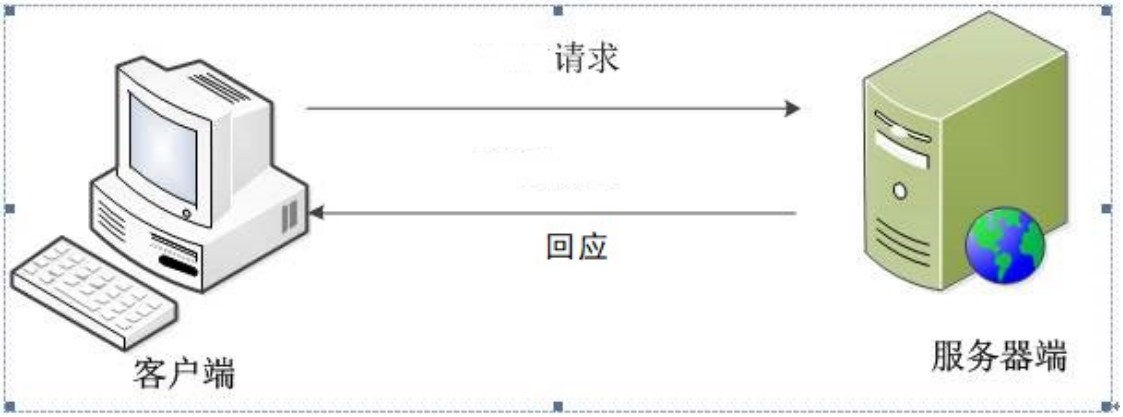
1.2.1 套接字简介

- 套接字(Socket)： 实现网络编程进行数据传输的一种技术手段,网络上各种各样的网络服务大多都是基于 Socket 来完成通信的。

- Python套接字编程模块：import socket

1.2.2 服务端与客户端

- 服务端（Server）：服务端是为客户端服务的，服务的内容诸如向客户端提供资源，保存客户端数据，处理客户端请求等。
- 客户端（Client）：也称为用户端，是指与服务端相对应，为客户提供一定应用功能的程序，我们平时使用的手机或者电脑上的程序基本都是客户端程序。

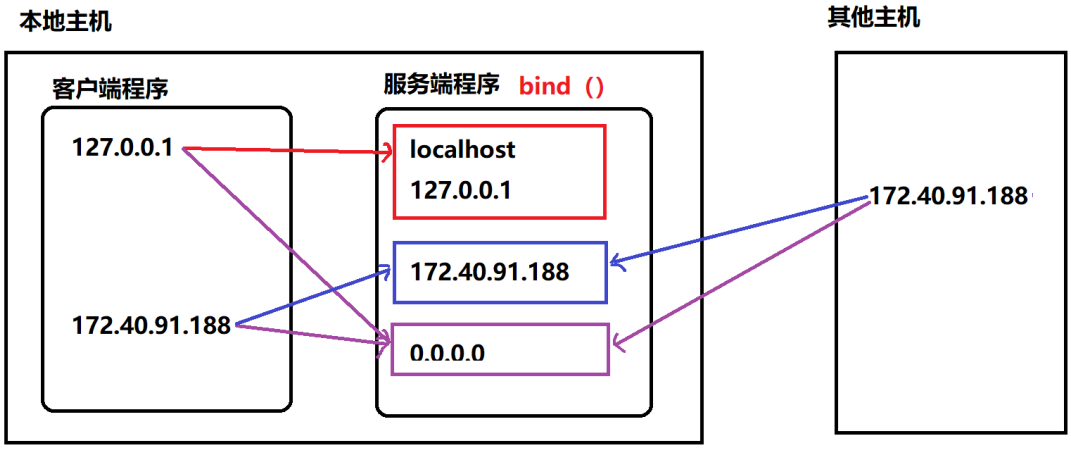


1.2.3 UDP套接字编程

- 创建套接字

```
sockfd=socket.socket(socket_family,socket_type,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_DGRAM 表示udp套接字 （也叫数据报套接字）
   proto 通常为0 选择子协议
返回值： 套接字对象
```

- 绑定地址
  - 本地地址： 'localhost' , '127.0.0.1'
  - 网络地址： '172.40.91.185' （通过ifconfig查看）
  - 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)
```

- 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
```

- 关闭套接字

```
sockfd.close()
功能：关闭套接字
```

- 服务端客户端流程

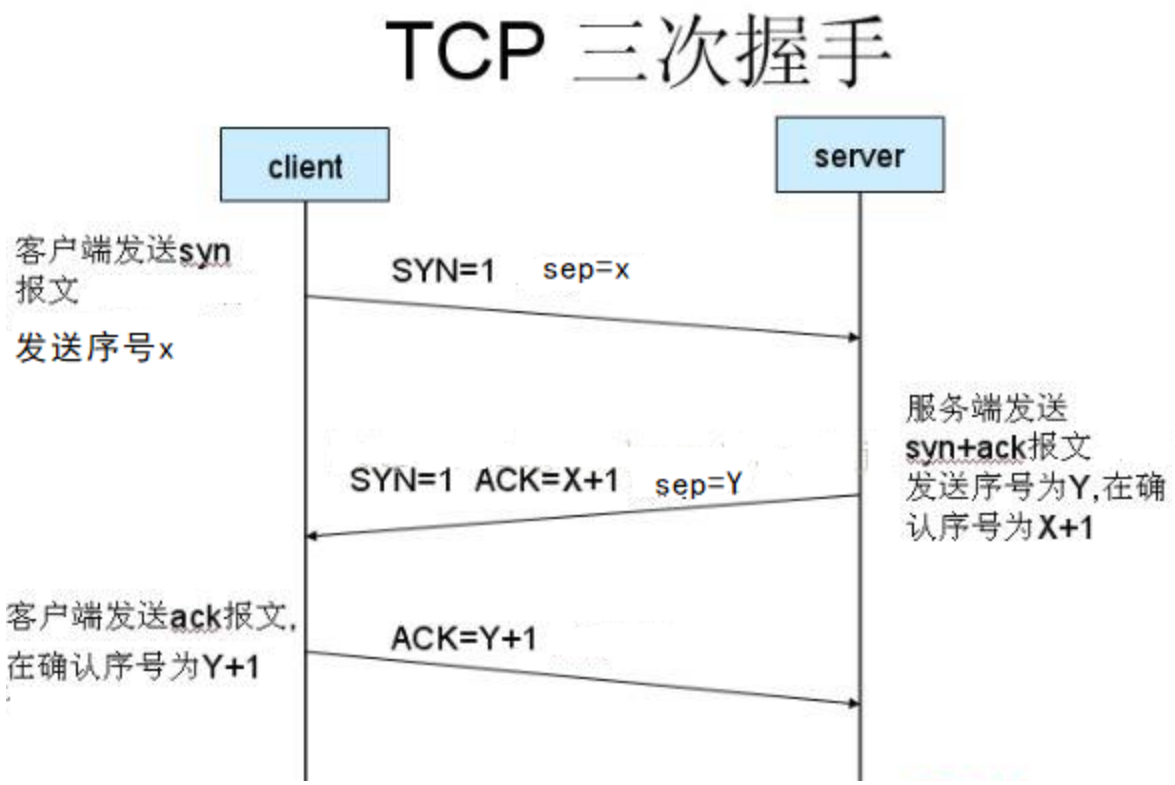
1.2.4 UDP套接字特点

- 可能会出现数据丢失的情况
- 传输过程简单，实现容易
- 数据以数据包形式表达传输
- 数据传输效率较高

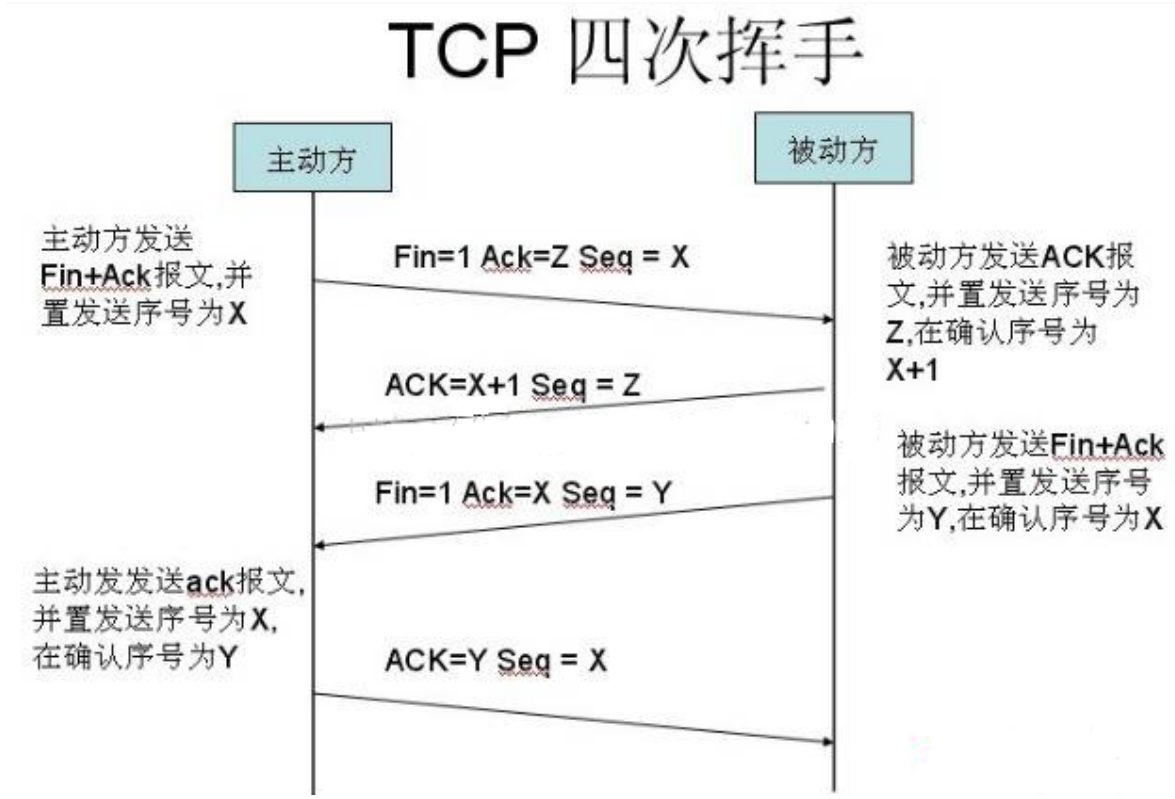
1.3 TCP 传输方法

1.3.1 TCP传输特点

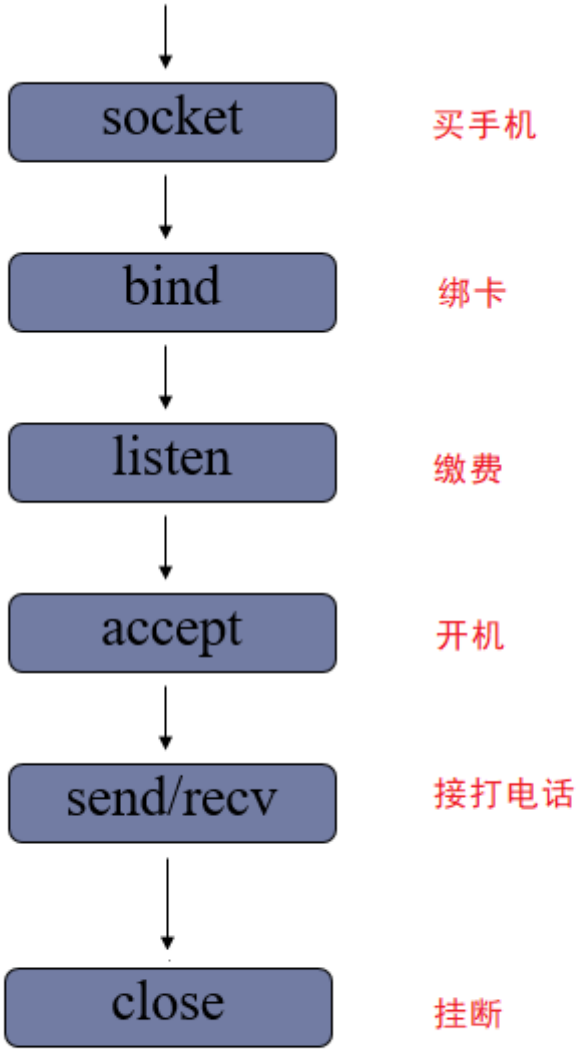
- 面向连接的传输服务
    - 传输特征： 提供了可靠的数据传输，可靠性指数据传输过程中无丢失，无失序，无差错，无重复。
    - 可靠性保障机制（都是操作系统网络服务自动帮应用完成的）：
        - 在通信前需要建立数据连接
        - 确认应答机制
        - 通信结束要正常断开连接
- 三次握手（建立连接）
    - 客户端向服务器发送消息报文请求连接
    - 服务器收到请求后，回复报文确定可以连接
    - 客户端收到回复，发送最终报文连接建立



- 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



1.3.2 TCP服务端



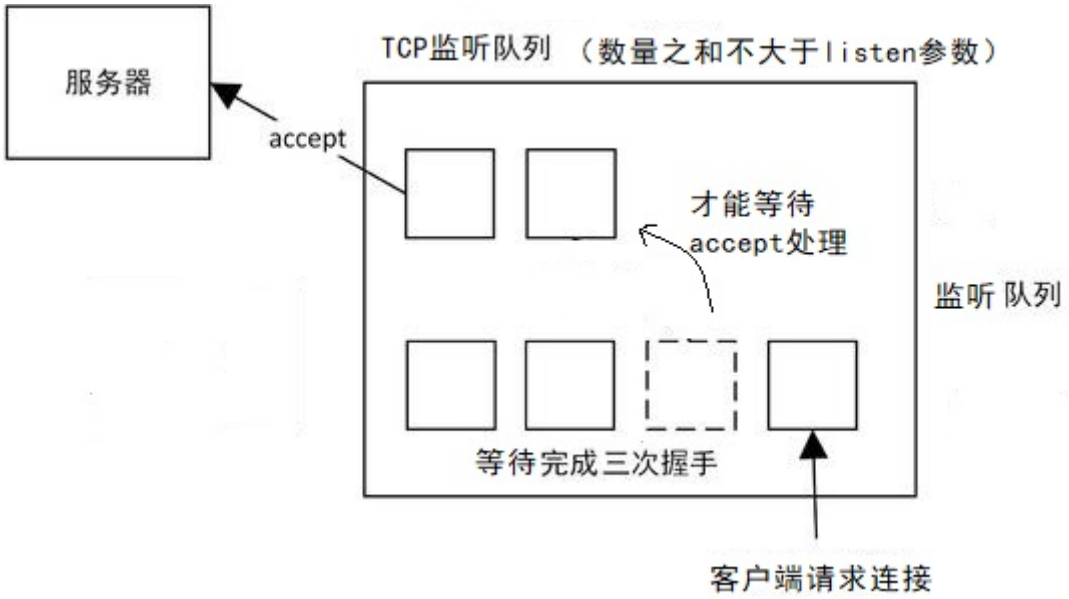
- 创建套接字

```
sockfd=socket.socket(socket_family,socket_type,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 表示tcp套接字 （也叫流式套接字）
   proto 通常为0 选择子协议
返回值： 套接字对象
```

- 绑定地址 （与udp套接字相同）

- 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
```



- 处理客户端连接请求

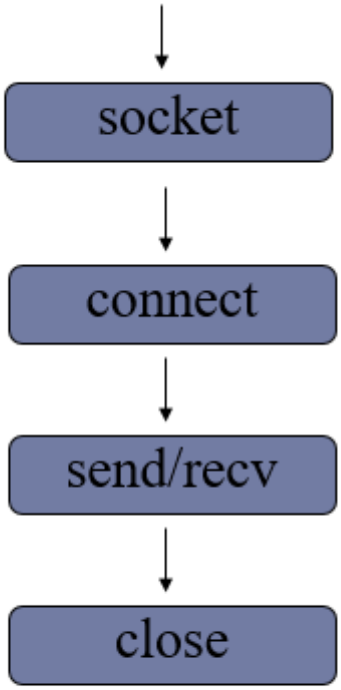
```
connfd,addr = sockfd.accept()
功能： 阻塞等待处理客户端请求
返回值： connfd 客户端连接套接字
     addr 连接的客户端地址
```

- 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 (与udp套接字相同)

1.3.3 TCP客户端



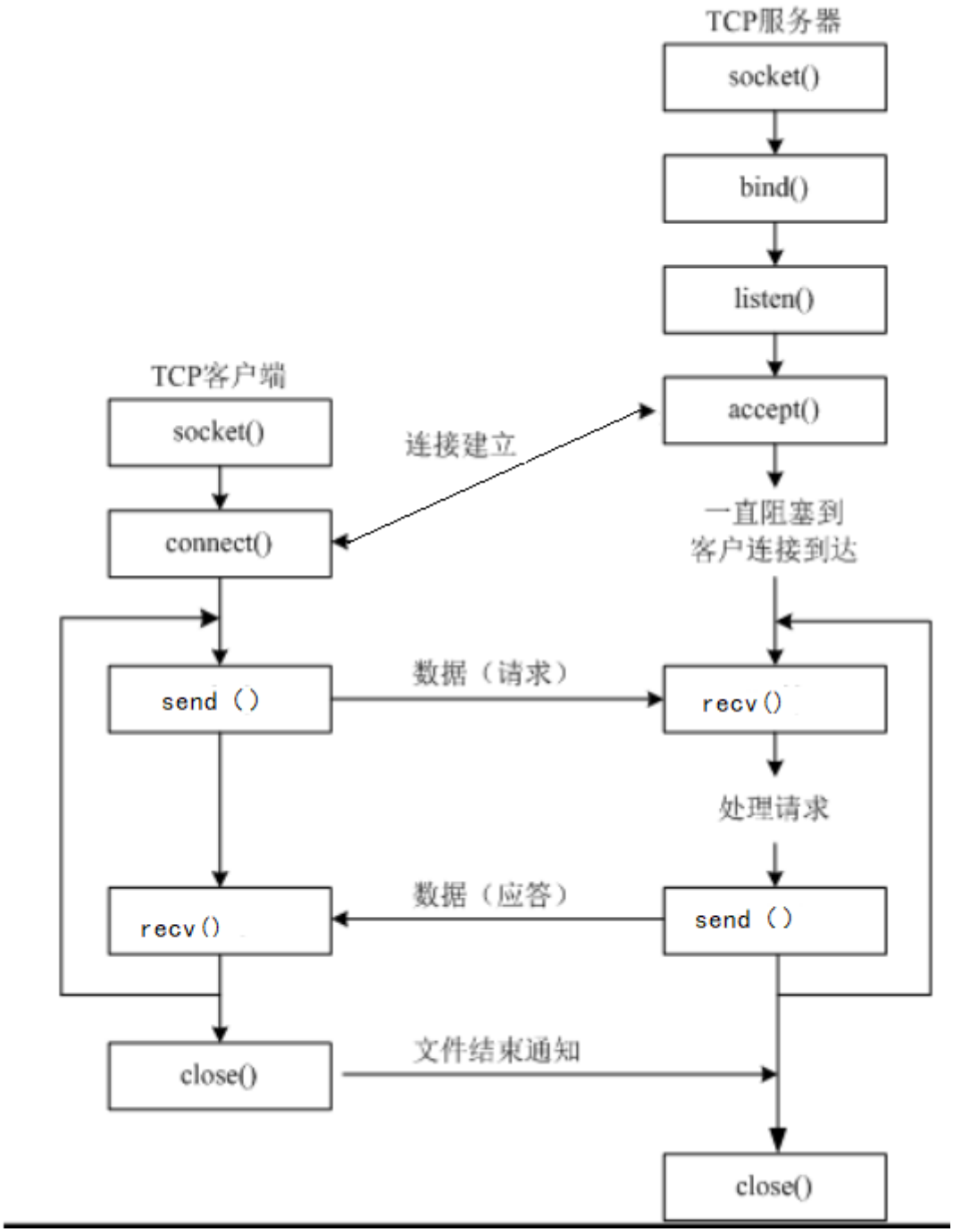
- 创建TCP套接字
- 请求连接

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

- 收发消息

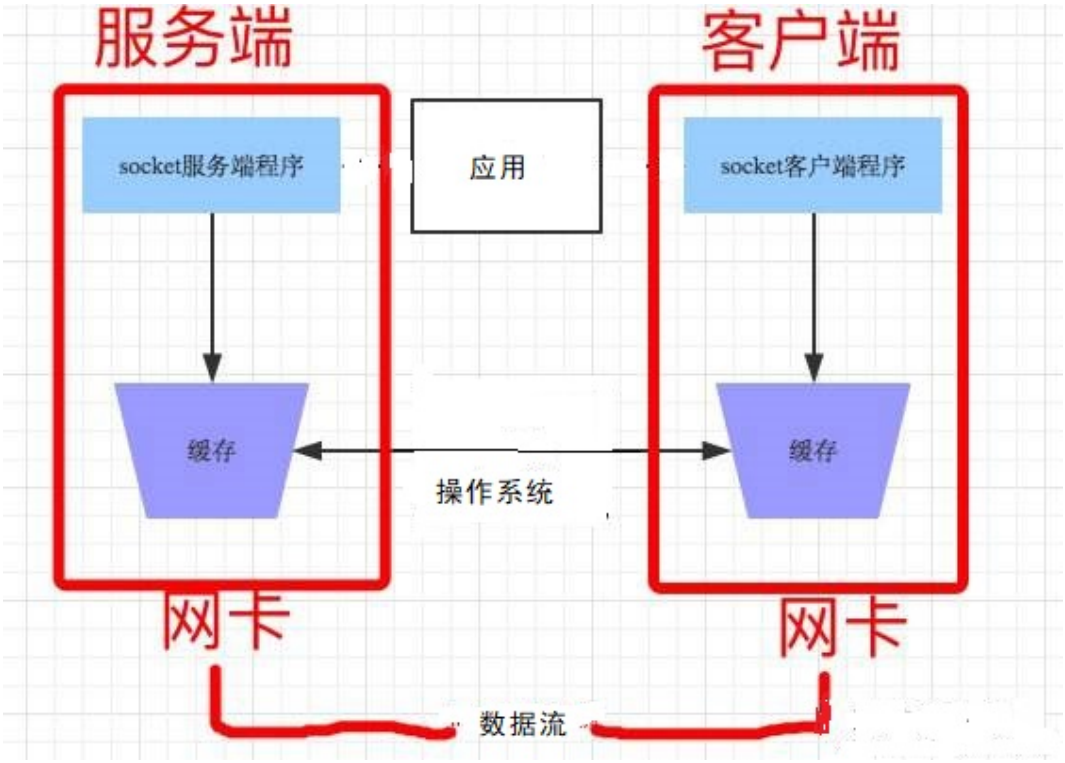
> 注意： 防止两端都阻塞，recv send要配合

- 关闭套接字



1.3.4 TCP套接字细节

- 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向其发送数据则会产生BrokenPipeError
- 一个服务端可以同时连接多个客户端，也能够重复被连接
- tcp粘包问题
  - 产生原因
    - 为了解决数据再传输过程中可能产生的速度不协调问题，操作系统设置了缓冲区
    - 实际网络工作过程比较复杂，导致消息收发速度不一致
    - tcp以字节流方式进行数据传输，在接收时不区分消息边界



- 带来的影响
  - 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
  - 人为的添加消息边界，用作消息之间的分割
  - 控制发送的速度

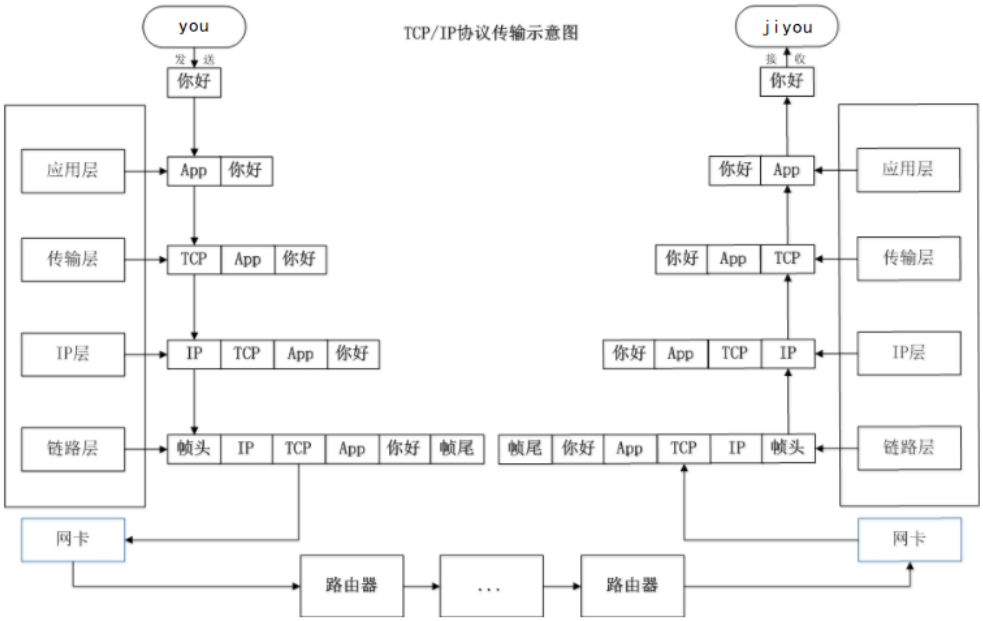
1.3.5 TCP与UDP对比

- 传输特征
  - TCP提供可靠的数据传输，但是UDP则不保证传输的可靠性
  - TCP传输在数据处理为字节流，而UDP处理为数据包形式
  - TCP传输需要建立连接才能进行数据传，效率相对较低，UDP比较自由，无需连接，效率较高
- 套接字编程区别
  - 创建的套接字类型不同
  - tcp套接字会有粘包，udp套接字有消息边界不会粘包
  - 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
  - tcp套接字使用send，recv收发消息，udp套接字使用sendto，recvfrom
- 使用场景
  - tcp更适合对准确性要求高，传输数据较大的场景
    - 文件传输：如下载电影，访问网页，上传照片
    - 邮件收发
    - 点对点数据传输：如点对点聊天，登录请求，远程访问，发红包
  - udp更适合对可靠性要求不竟没有那么高，传输方式比较自由的场景
    - 视频流的传输： 如直播，视频聊天
    - 广播：如网络广播，群发消息
    - 实时传输：如游戏画面
  - 在一个大型的项目中，可能既涉及到TCP网络又有UDP网络

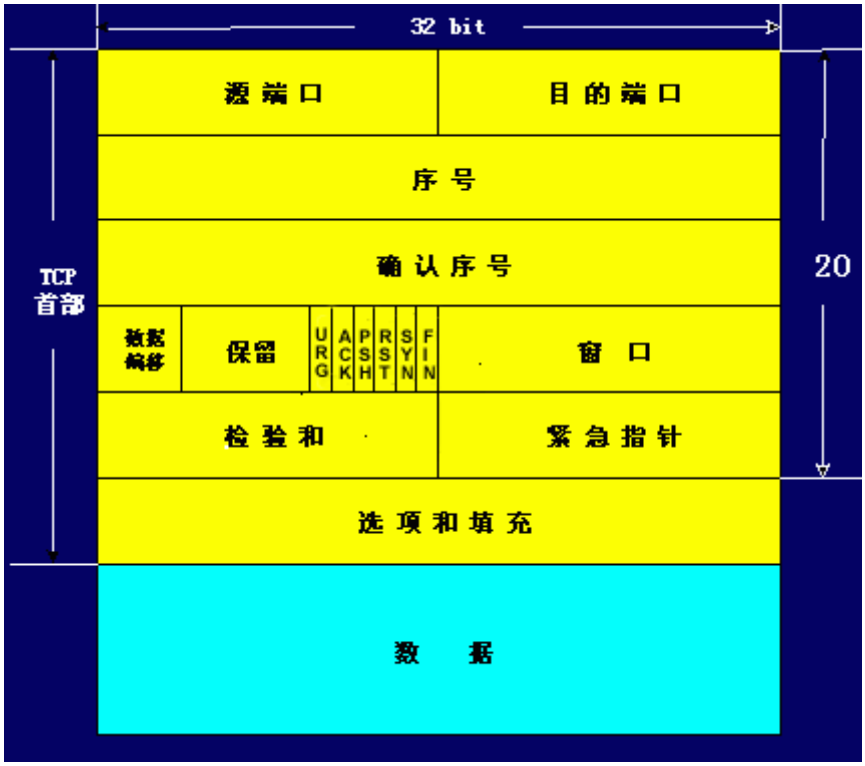
1.4 数据传输过程

1.4.1 传输流程

- 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。
- 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。
- 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。



1.4.2 TCP协议首部（了解）



- 源端口和目的端口 各占2个字节，分别写入源端口和目的端口。

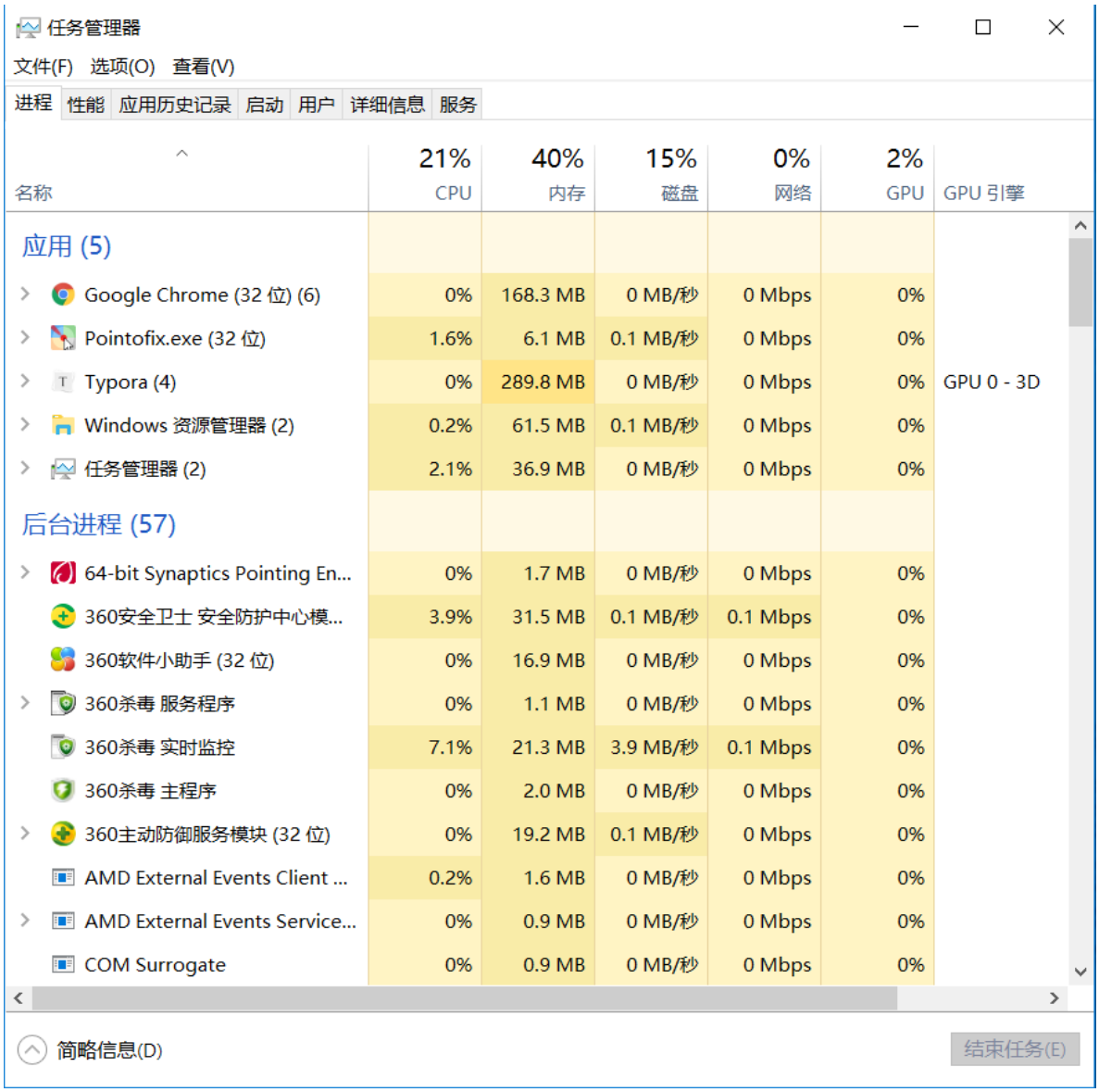
- 序号 占4字节。TCP是面向字节流的。在一个TCP连接中传送的字节流中的每一个字节都按顺序编号。例如，一报文段的序号是301，而接待的数据共有100字节。这就表明本报文段的数据的第一个字节的序号是301，最后一个字节的序号是400。
- 确认号 占4字节，是期望收到对方下一个报文段的第一个数据字节的序号。例如，B正确收到了A发送过来的一个报文段，其序号字段值是501，而数据长度是200字节（序号501~700），这表明B正确收到了A发送的到序号700为止的数据。因此，B期望收到A的下一个数据序号是701，于是B在发送给A的确认报文段中把确认号置为701。
- 确认ACK（ACKnowledgment） 仅当ACK = 1时确认号字段才有效，当ACK = 0时确认号无效。TCP规定，在连接建立后所有的传送的报文段都必须把ACK置为1。
- 同步SYN（SYNchronization） 在连接建立时用来同步序号。当SYN=1而ACK=0时，表明这是一个连接请求报文段。对方若同意建立连接，则应在响应的报文段中使SYN=1和ACK=1，因此SYN置为1就表示这是一个连接请求或连接接受报文。
- 终止FIN（FINis，意思是“完”“终”） 用来释放一个连接。当FIN=1时，表明此报文段的发送发的数据已发送完毕，并要求释放运输连接。

2. 多任务编程

2.1 多任务概述

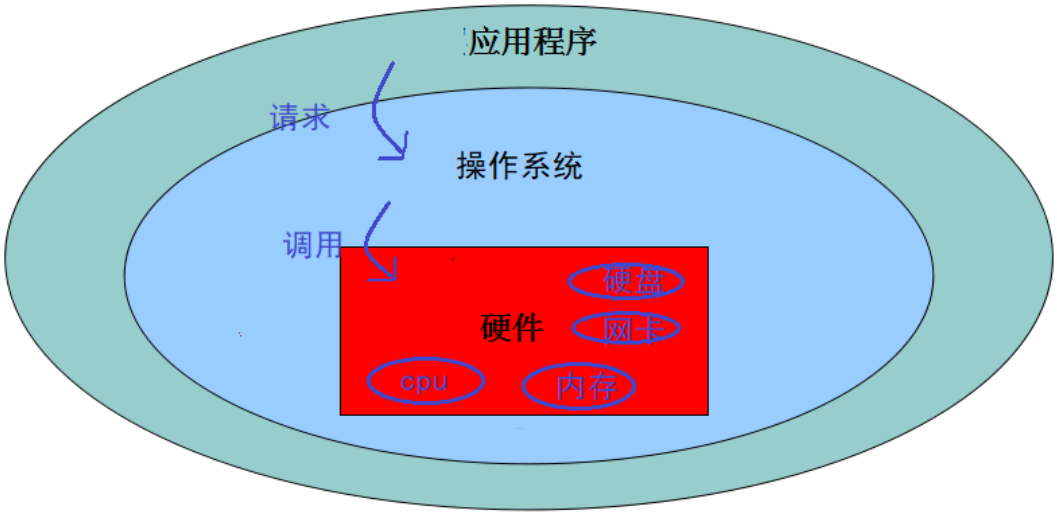
- 多任务

  即操作系统中可以同时运行多个任务。比如我们可以同时挂着qq，听音乐，同时上网浏览网页。这是我们看得到的任务，在系统中还有很多系统任务在执行,现在的操作系统基本都是多任务操作系统，具备运行多任务的能力。

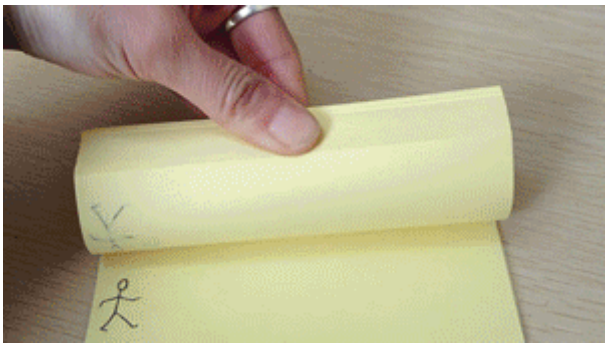


- 计算机原理
  - CPU：计算机硬件的核心部件，用于对任务进行执行运算。

  - 操作系统调用CPU执行任务



- cpu轮训机制： cpu都在多个任务之间快速的切换执行，切换速度在微秒级别，其实cpu同时只执行一个任务，但是因为切换太快了，从应用层看好像所有任务同时在执行。

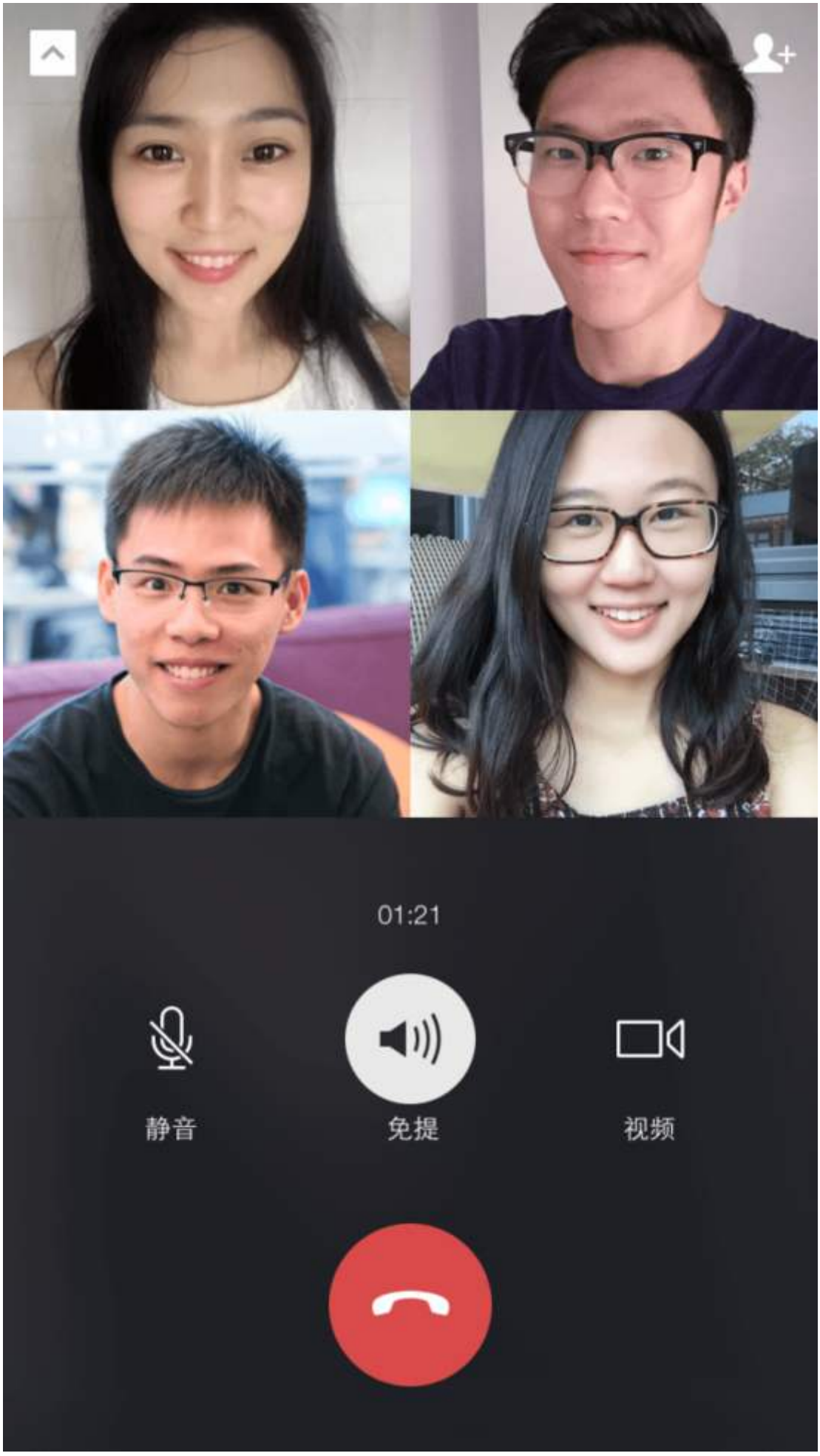


- 多核CPU：现在的计算机一般都是多核CPU，比如四核，八核，我们可以理解为由多个单核CPU的集合。这时候在执行任务时就有了选择，可以将多个任务分配给某一个cpu核心，也可以将多个任务分配给多个cpu核心，操作系统会自动根据任务的复杂程度选择最优的分配方案。
  - 并发： 多个任务如果被分配给了一个cpu内核，那么这多个任务之间就是并发关系，并发关系的多个任务之间并不是真正的 “"同时"。
  - 并行： 多个任务如果被分配给了不同的cpu内核，那么这多个任务之间执行时就是并行关系，并行关系的多个任务时真正的“同时”执行。

- 什么是多任务编程

  多任务编程即一个程序中编写多个任务，在程序运行时让这多个任务一起运行，而不是一个一个的顺次执行。

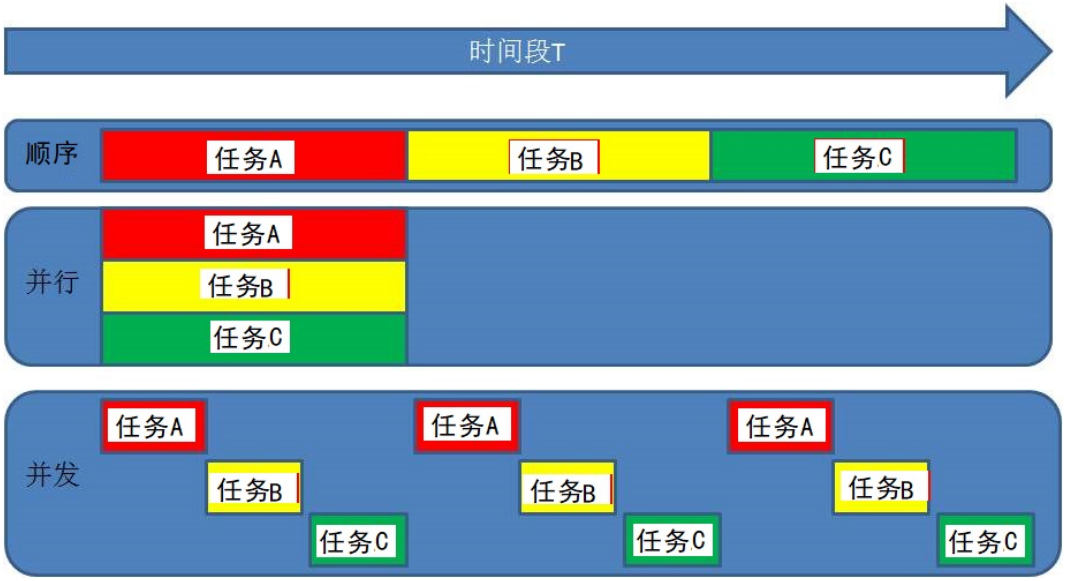
  比如微信视频聊天，这时候在微信运行过程中既用到了视频任务也用到了音频任务，甚至同时还能发消息。这就是典型的多任务。而实际的开发过程中这样的情况比比皆是。



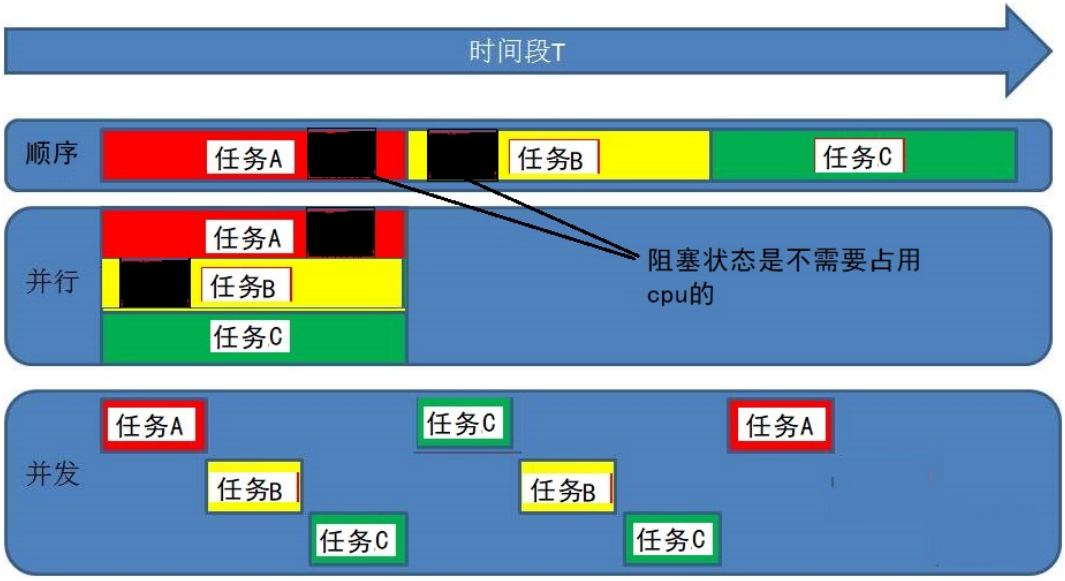
- 实现多任务编程的方法： 多进程编程，多线程编程

- 多任务意义
  - 提高了任务之间的配合，可以根据运行情况进行任务创建。

    比如： 你也不知道用户在微信使用中是否会进行视频聊天，总不能提前启动起来吧，这是需要根据用户的行为启动新任务。
  - 充分利用计算机资源，提高了任务的执行效率。
    - 在任务中无阻塞时只有并行状态才能提高效率



    - 在任务中有阻塞时并行并发都能提高效率



2.2 进程（Process）

2.2.1 进程概述

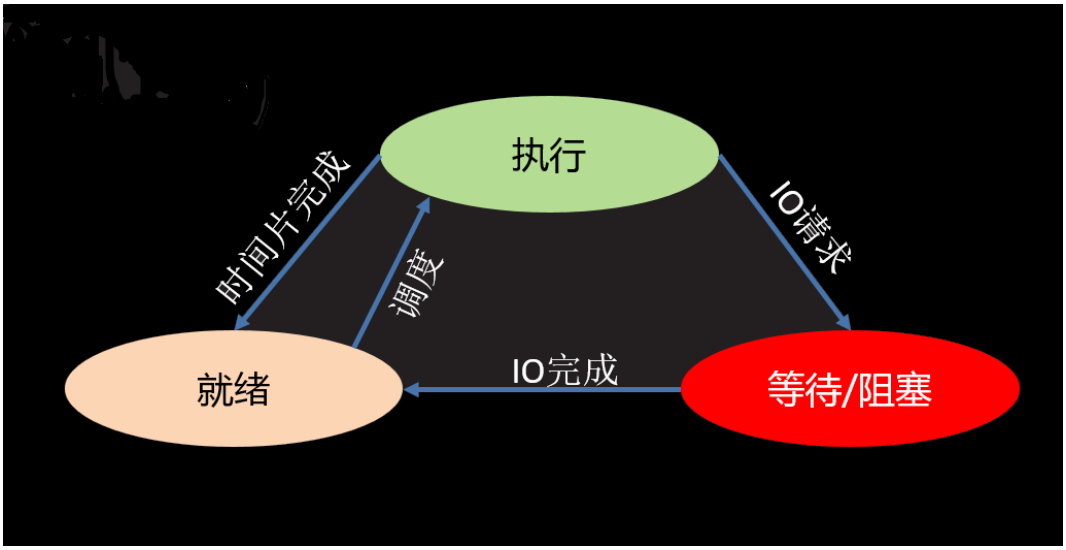
- 定义： 程序在计算机中的一次执行过程。
  - 程序是一个可执行的文件，是静态的占有磁盘。
  - 进程是一个动态的过程描述，占有计算机运行资源，有一定的生命周期。

- 进程状态
  - 三态

    就绪态： 进程具备执行条件，等待系统调度分配cpu资源

    运行态： 进程占有cpu正在运行

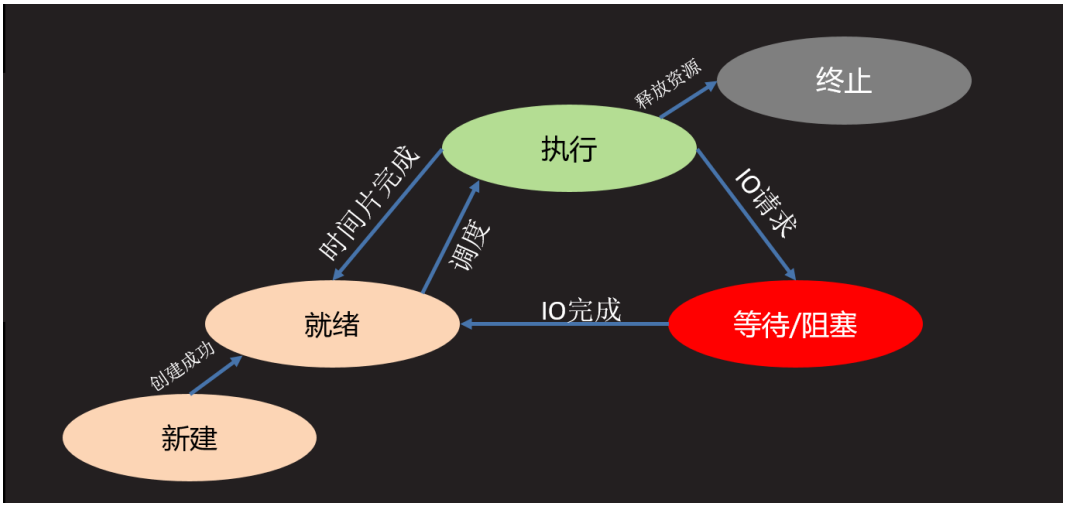
    等待态： 进程阻塞等待，此时会让出cpu



  - 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



- 进程命令
  - 查看进程信息

```
ps -aux
```

```
tarena@tedu:~$ ps -aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.7  0.3 159896  9272 ?        Ss   18:36   0:02 /sbin/init splash
root           2  0.0  0.0      0     0 ?        S    18:36   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   18:36   0:00 [rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   18:36   0:00 [rcu_par_gp]
root           5  0.0  0.0      0     0 ?        I    18:36   0:00 [kworker/0:0-eve]
root           6  0.0  0.0      0     0 ?        I<   18:36   0:00 [kworker/0:0H-kb]
root           7  0.0  0.0      0     0 ?        I    18:36   0:00 [kworker/0:1-cgr]
```

- USER： 进程的创建者
- PID ：操作系统分配给进程的编号,大于0的整数，系统中每个进程的PID都不重复。PID也是重要的区分进程的标志。
- %CPU,%MEM : 占有的CPU和内存
- STAT： 进程状态信息，S I 表示阻塞状态 ，R 表示就绪状态或者运行状态
- START : 进程启动时间
- COMMAND : 通过什么程序启动的进程

- 进程树形结构

```
pstree
```

- 父子进程：在Linux操作系统中，进程形成树形关系，任务上一级进程是下一级的父进程，下一级进程是上一级的子进程。

2.2.2 多进程编程

- 使用模块： multiprocessing
- 创建流程
  【1】 将需要新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回收进程资源

- 主要类和函数使用

```
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，该函数作为新进程执行内容，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
```

- 进程执行现象理解 （难点）
  - 新的进程是原有进程的子进程，子进程复制父进程全部内存空间代码段，一个进程可以创建多个子进程。
  - 子进程只执行指定的函数，其余内容均是父进程执行内容，但是子进程也拥有其他父进程资源。
  - 各个进程在执行上互不影响，也没有先后顺序关系。
  - 进程创建后，各个进程空间独立，相互没有影响。
  - 各个进程空间独立
  - multiprocessing 创建的子进程中无法使用标准输入（input）。

- 进程对象属性
  - 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则该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()

2.2.3 进程处理细节

- 进程相关函数

```
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
```

```
sys.exit(info)
功能：退出进程
参数：字符串 表示退出时打印内容
```

- 孤儿和僵尸
  - 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。
  - 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，但是会存留部分进程信息资源在内存中，大量的僵尸进程会浪费系统的内存资源。
- 如何避免僵尸进程产生
    1. 使用join()回收
    2. 在父进程中使用signal方法处理

```
from signal import *
signal(SIGCHLD,SIG_IGN)
```

2.2.5 创建进程类

进程的基本创建方法将子进程执行的内容封装为函数。如果我们更热衷于面向对象的编程思想，也可以使用类来封装进程内容。

- 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
- 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

2.2.4 进程池

- 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。
- 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。

- 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
```

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
```

2.2.5 进程通信

- 必要性： 进程间空间独立，资源不共享，此时在需要进程间数据传输时就需要特定的手段进行数据通信。
- 常用进程间通信方法：消息队列，套接字等。
- 消息队列使用
    - 通信原理： 在内存中开辟空间，建立队列模型，进程通过队列将消息存入，或者从队列取出完成进程间通信。
    - 实现方法

```
from multiprocessing import Queue

q = Queue(maxsize=0)
功能: 创建队列对象
参数：最多存放消息个数
返回值：队列对象

q.put(data,[block,timeout])
功能：向队列存入消息
参数：data 要存入的内容
   block 设置是否阻塞 False为非阻塞
   timeout 超时检测

q.get([block,timeout])
功能：从队列取出消息
参数：block 设置是否阻塞 False为非阻塞
    timeout 超时检测
返回值： 返回获取到的内容

q.full() 判断队列是否为满
q.empty() 判断队列是否为空
q.qsize() 获取队列中消息个数
q.close() 关闭队列
```

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： xxxxxxxxxxx
> 【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

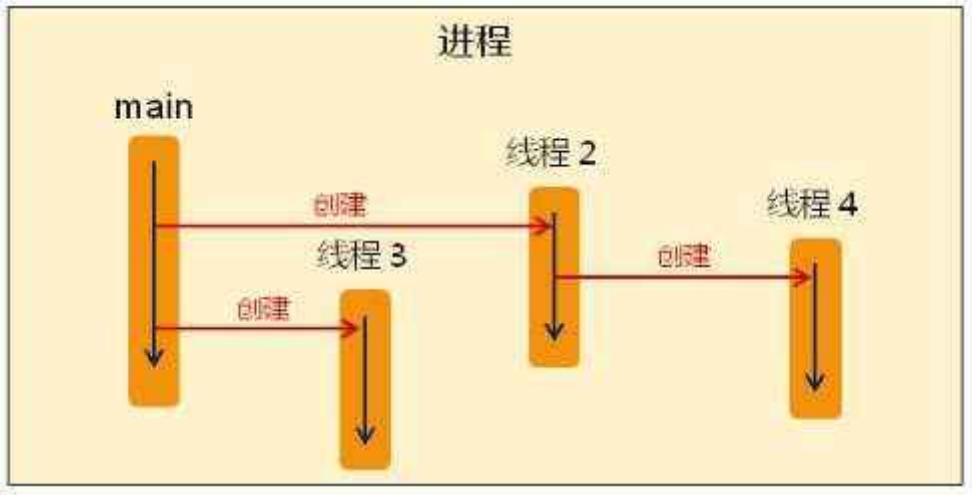
2.3 线程 (Thread)

2.3.1 线程概述

- 什么是线程
  【1】 线程被称为轻量级的进程，也是多任务编程方式
  【2】 也可以利用计算机的多cpu资源
  【3】 线程可以理解为进程中再开辟的分支任务
- 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同样互不影响各自运行
  【5】 线程的创建和销毁消耗资源远小于进程

2.3.2 多线程编程

- 线程模块： threading



- 创建方法

  【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
  - 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.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。

2.3.3 创建线程类

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

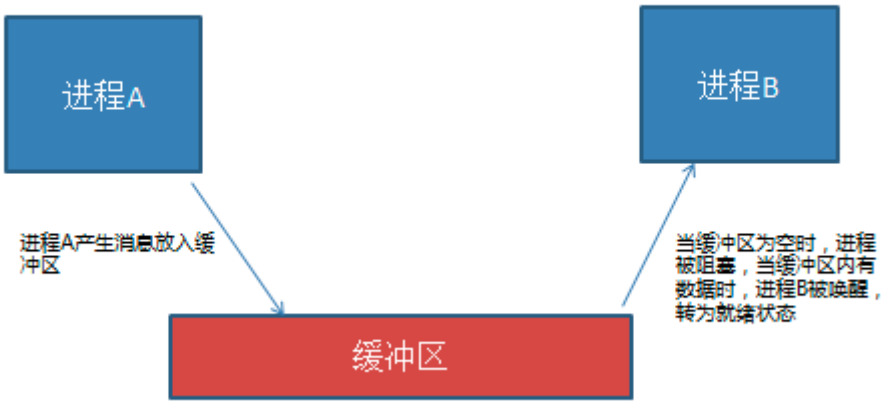
   【2】 调用start自动执行run方法

   【3】 调用join回收线程

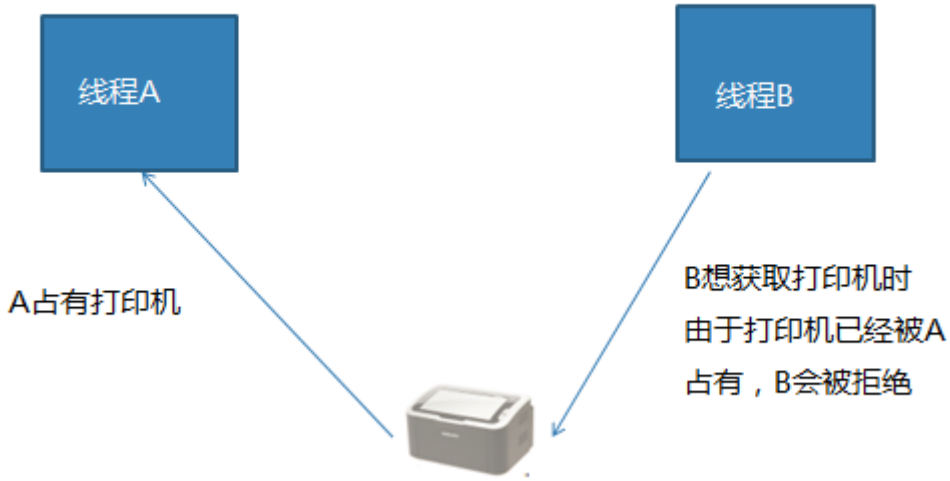
2.3.4 线程同步互斥

- 线程通信方法： 线程间使用全局变量进行通信
- 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。

- 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Event

```
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
```

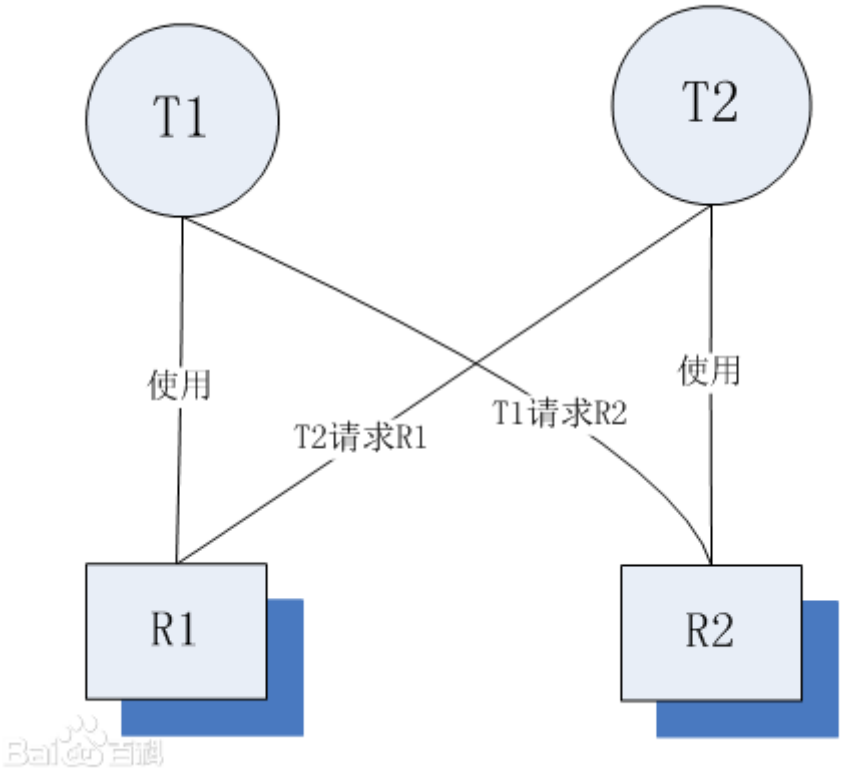
- 线程锁 Lock

```
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
```

2.3.5 死锁

- 什么是死锁

  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。



- 死锁产生条件
  - 互斥条件：指线程使用了互斥方法，使用一个资源时其他线程无法使用。
  - 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，在获取到新的资源前不会释放自己保持的资源。
  - 不剥夺条件：不会受到线程外部的干扰，如系统强制终止线程等。
  - 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，如 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。

- 如何避免死锁

- 逻辑清晰，不要同时出现上述死锁产生的四个条件
- 通过测试工程师进行死锁检测

2.3.6 GIL问题

- 什么是GIL问题 （全局解释器锁）

  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任务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#）

  Guido的声明：http://www.artima.com/forums/flat.jsp?forum=106&thread=214235

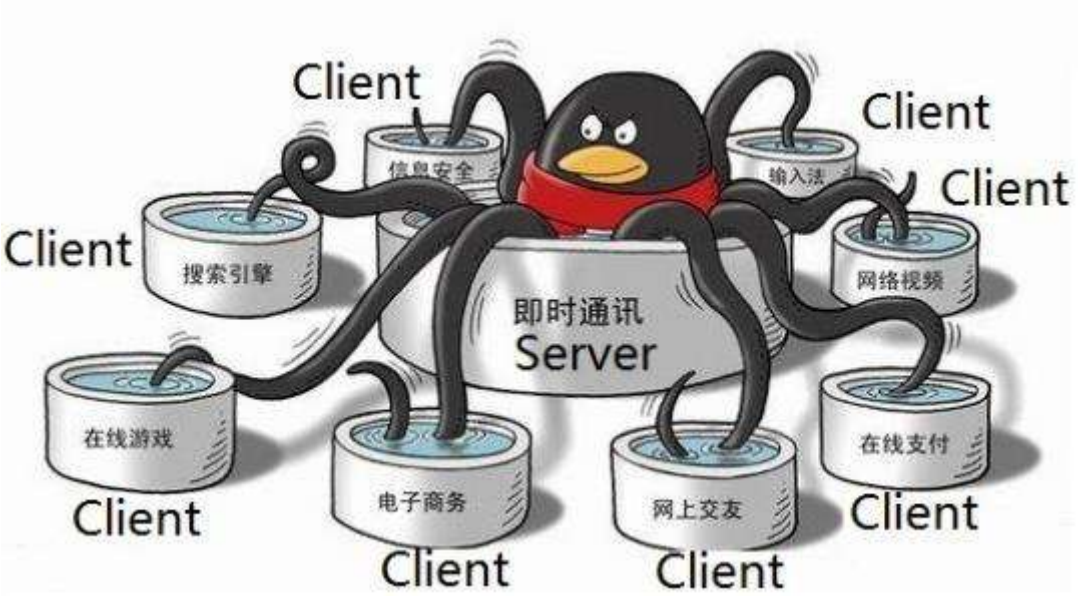
- 结论
  - GIL问题与Python语言本身并没什么关系，属于解释器设计的历史问题。
  - 在无阻塞状态下，多线程程序程序执行效率并不高，甚至还不如单线程效率。
  - Python多线程只适用于执行有阻塞延迟的任务情形。

2.3.7 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. Python线程存在GIL问题，但是进程没有。

- 使用场景



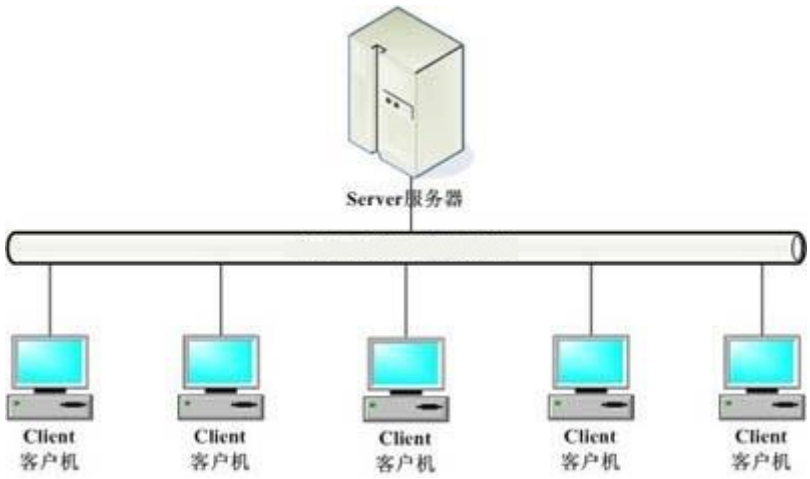
1. 任务场景：一个大型服务，往往包含多个独立的任务模块，每个任务模块又有多个小独立任务构成，此时整个项目可能有多个进程，每个进程又有多个线程。
2. 编程语言：Java,C#之类的编程语言在执行多任务时一般都是用线程完成，因为线程资源消耗少；而Python由于GIL问题往往使用多进程。

3. 网络并发模型

3.1 网络并发模型概述

- 什么是网络并发

  在实际工作中，一个服务端程序往往要应对多个客户端同时发起访问的情况。如果让服务端程序能够更好的同时满足更多客户端网络请求的情形，这就是并发网络模型。



- 循环网络模型问题

  循环网络模型只能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。这样的网络模型虽然简单，资源占用不多，但是无法同时处理多个客户端请求就是其最大的弊端，往往只有在一些低频的小请求任务中才会使用。

3.2 多任务并发模型

多任务并发模型具体指多进程多线程网络并发模型，即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，多任务并发模型也是实际工作中最为常用的服务端处理模型。

- 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：客户端请求较复杂，需要长时间占有服务器。

3.1.1 多进程并发模型

- 创建网络套接字用于接收客户端请求
- 等待客户端连接
- 客户端连接，则创建新的进程具体处理客户端请求
- 主进程继续等待其他客户端连接
- 如果客户端退出，则销毁对应的进程

3.1.2 多线程并发模型

- 创建网络套接字用于接收客户端请求
- 等待客户端连接
- 客户端连接，则创建新的线程具体处理客户端请求
- 主线程继续等待其他客户端连接
- 如果客户端退出，则销毁对应的线程

ftp 文件服务器

【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

3.3 IO并发模型

3.2.1 IO概述

- 什么是IO

  在程序中存在读写数据操作行为的事件均是IO行为，比如终端输入输出 ,文件读写，数据库修改和网络消息收发等。

- 程序分类
  - IO密集型程序：在程序执行中有大量IO操作，而运算操作较少。消耗cpu较少，耗时长。
  - 计算密集型程序：程序运行中运算较多，IO操作相对较少。cpu消耗多，执行速度快，几乎没有阻塞。
- IO分类：阻塞IO ，非阻塞IO，IO多路复用，异步IO等。

3.2.2 阻塞IO

- 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。
- 效率：阻塞IO效率很低。但是由于逻辑简单所以是默认IO行为。
- 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

3.2.3 非阻塞IO

- 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
```

3.2.4 IO多路复用

- 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。
- 具体方案
  - select方法： windows linux unix
  - poll方法： linux unix
  - epoll方法： linux

- select 方法

```
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
```

- poll方法

```
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
```

- epoll方法

  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

  - 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）

3.2.5 IO并发模型

利用IO多路复用等技术，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： 只针对处理并发产生的IO事件
- 适用情况：HTTP请求，网络传输等都是IO行为，可以通过IO多路复用监控多个客户端的IO请求。

- 并发服务实现过程

  【1】将关注的IO准备好，通常设置为非阻塞状态。
  【2】通过IO多路复用方法提交，进行IO监控。
  【3】阻塞等待，当监控的IO有发生时，结束阻塞。
  【4】遍历返回值列表，确定就绪IO事件。
  【5】处理发生的IO事件。
  【6】继续循环监控IO发生。

3.3 并发技术探讨（扩展）

3.3.1 高并发问题

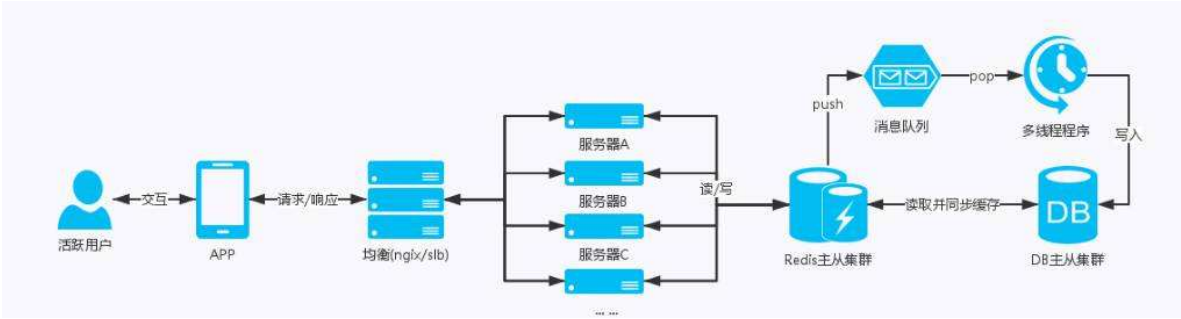
- 衡量高并发的关键指标
  - 响应时间(Response Time)： 接收请求后处理的时间
  - 吞吐量(Throughput)： 响应时间+QPS+同时在线用户数量
  - 每秒查询率QPS(Query Per Second)： 每秒接收请求的次数
  - 每秒事务处理量TPS(Transaction Per Second)：每秒处理请求的次数（包含接收，处理，响应）
  - 同时在线用户数量：同时连接服务器的用户的数量
- 多大的并发量算是高并发
  - 没有最高，只要更高

    比如在一个小公司可能QPS2000+就不错了，在一个需要频繁访问的门户网站可能要达到QPS5W+
  - C10K问题

    早先服务器都是单纯基于进程/线程模型的，新到来一个TCP连接，就需要分配1个进程（或者线程）。而进程占用操作系统资源多，一台机器无法创建很多进程。如果是C10K就要创建1万个进程，那么单机而言操作系统是无法承受的，这就是著名的C10k问题。创建的进程线程多了，数据拷贝频繁， 进程/线程切换消耗大， 导致操作系统崩溃，这就是C10K问题的本质！

3.3.2 更高并发的实现

为了解决C10K问题，现在高并发的实现已经是一个更加综合的架构艺术。涉及到进程线程编程，IO处理，数据库处理，缓存，队列，负载均衡等等，这些我们在后面的阶段还会学习。此外还有硬件的设计，服务器集群的部署，服务器负载，网络流量的处理等。



实际工作中，应对更庞大的任务场景，网络并发模型的使用有时也并不单一。比如多进程网络并发中每个进程再开辟线程，或者在每个进程中也可以使用多路复用的IO处理方法。

4. web服务

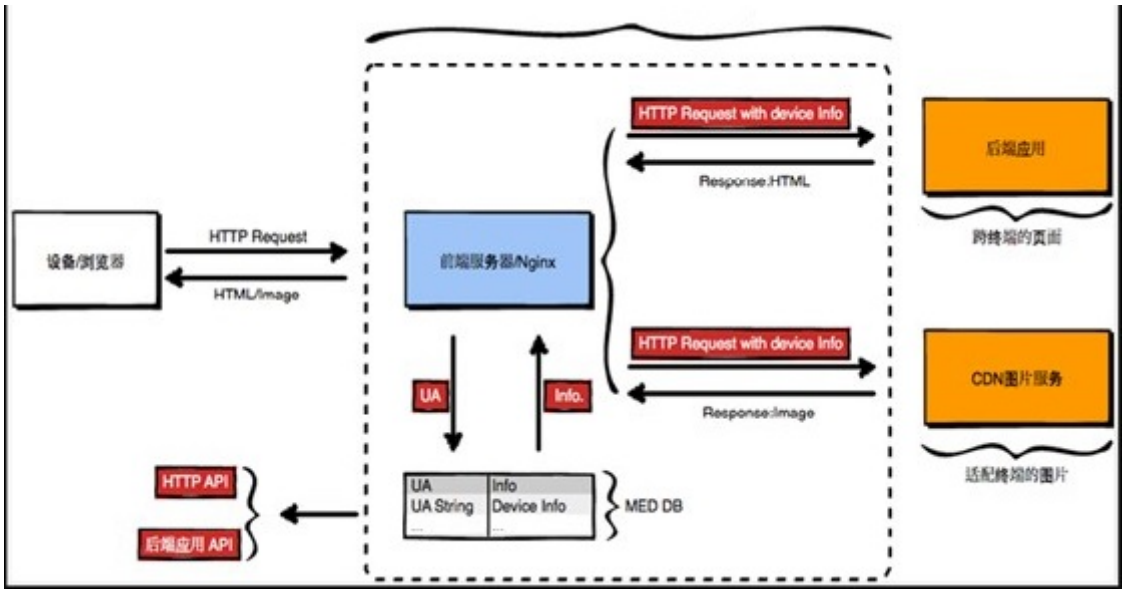
4.1 HTTP协议

4.1.1 协议概述

- 用途： 网页获取，数据的传输
- 特点
  - 应用层协议，使用tcp进行数据传输
  - 简单，灵活，很多语言都有HTTP专门接口
  - 无状态，数据传输过程中不记录传输内容
  - 有丰富了请求类型
  - 可以传输的数据类型众多

4.1.2 网页访问流程

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



4.1.2 HTTP请求

- 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
```

- 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

4.1.3 HTTP响应

- 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
```

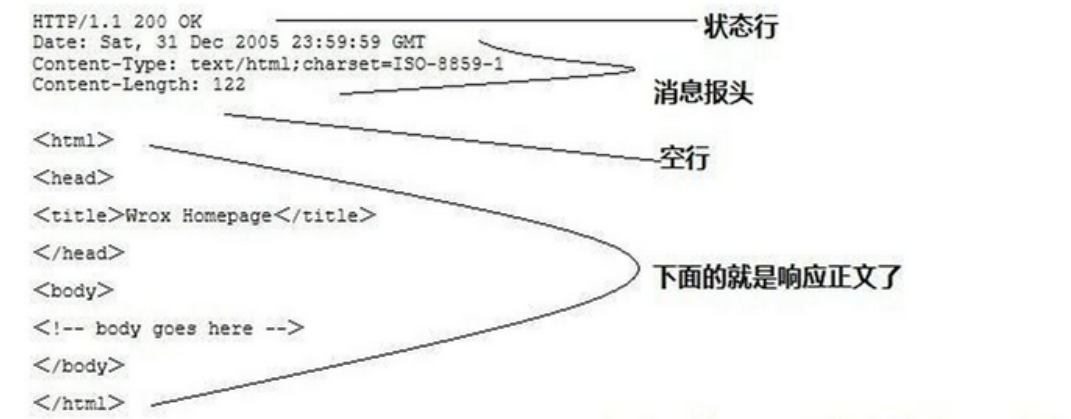
响应码：

```
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
```

- 空行
- 响应体：响应的主体内容信息



4.2 web 服务程序实现

1. 主要功能：
   【1】 接收客户端（浏览器）请求
   【2】 解析客户端发送的请求
   【3】 根据请求组织数据内容
   【4】 将数据内容形成http响应格式返回给浏览器
2. 特点：
   【1】 采用IO并发，可以满足多个客户端同时发起请求情况
   【2】 通过类接口形式进行功能封装
   【3】 做基本的请求解析，根据具体请求返回具体内容，同时处理客户端的非网页请求行为