Python高级编程技巧

ju7ran

目　录

1-深入类和对象

深入类和对象

鸭子类型和多态

多态的概念是应用于Java和C#这一类强类型语言中，而Python崇尚"鸭子类型"

动态语言调用实例方法时不检查类型，只要方法存在，参数正确，就可以调用。这就是动态语言的“鸭子类型”，它并不要求严格的继承体系，一个对象只要“看起来像鸭子，走起路来像鸭子”，那它就可以被看做是鸭子。

所谓多态：定义时的类型和运行时的类型不一样，此时就成为多态。

抽象基类(abc模块)

抽象基类（abstract base class,ABC）：抽象基类就是类里定义了纯虚成员函数的类。纯虚函数只提供了接口，并没有具体实现。抽象基类不能被实例化(不能创建对象)，通常是作为基类供子类继承，子类中重写虚函数，实现具体的接口。

抽象基类就是定义各种方法而不做具体实现的类，任何继承自抽象基类的类必须实现这些方法，否则无法实例化。

应用场景

- 判断某个对象的类型
- 我们需要强制某个子类必须实现某些方法

使用isinstance和type的区别

```
i = 1
s = 'a'
```

```
isinstance(i,int)
isinstance(s,str)
isinstance(s,int)
```

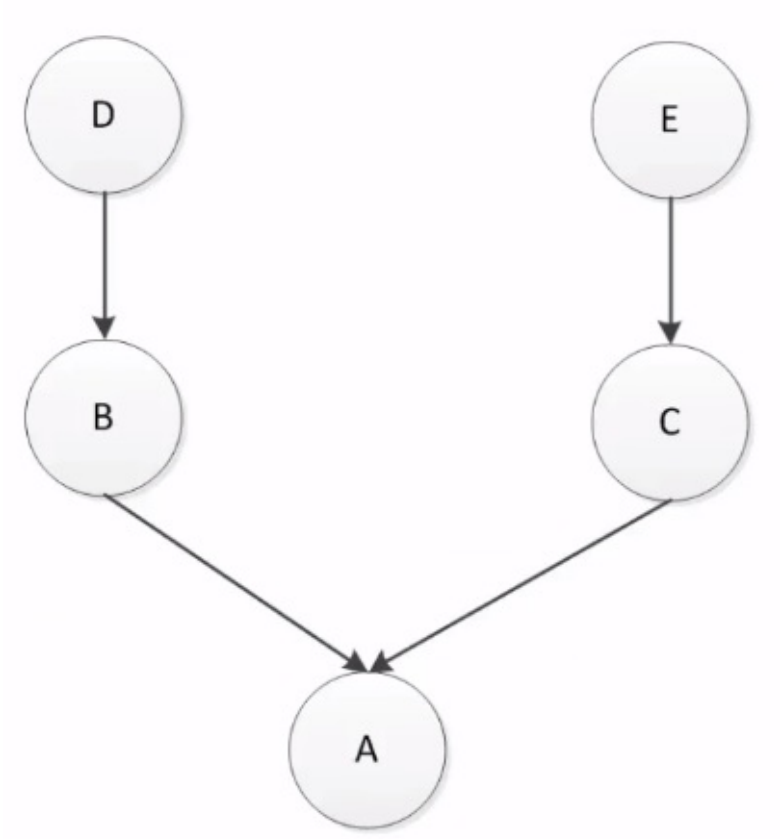
类变量和对象变量

```
class A:
    aa = 1

    def __init__(self,x,y):
        self.x = x
        self.y = y
```

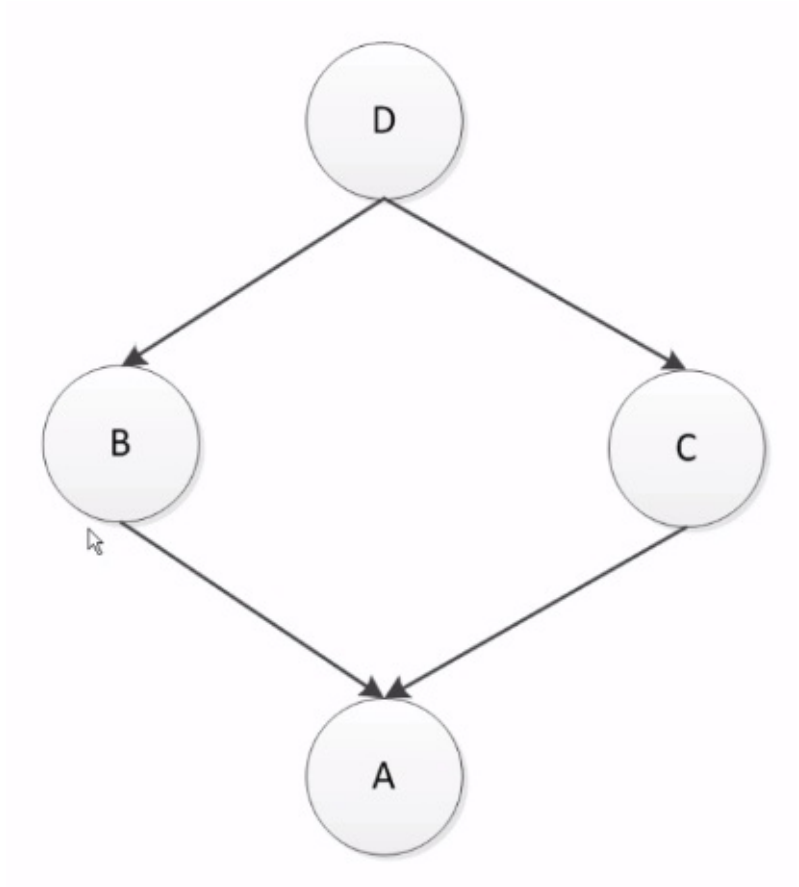
类属性和实例属性以及查找顺序

MRO算法



Python2.2之前的算法:金典类

DFS(deep first search)：A->B->D->C->E



Python2.2版本之后,引入了BFS(广度优先搜索)

BFS:A->B->C->D

在Python2.3之后,Python采用了C3算法

Python新式类继承的C3算法：https://www.cnblogs.com/blackmatrix/p/5644023.html

Python对象的自省机制

自省是通过一定的机制查询到对象的内部结构

Python中比较常见的自省（introspection）机制(函数用法)有： dir()，type(), hasattr(), isinstance()，通过这些函数，我们能够在程序运行时得知对象的类型，判断对象是否存在某个属性，访问对象的属性。

super函数

在类的继承中，如果重定义某个方法，该方法会覆盖父类的同名方法，但有时，我们希望能同时实现父类的功能，这时，我们就需要调用父类的方法了，可通过使用 `super` 来实现

2-类与对象深度问题与解决技巧

类与对象深度问题与解决技巧

1.如何派生内置不可变类型并修改其实例化行为

我们想自定义一种新类型的元组,对于传入的可迭代对象,我们只保留其中int类型且值大于0的元素，例如:

```
IntTuple([2,-2,'jr',['x','y'],4]) => (2,4)
```

如何继承内置tuple 实现IntTuple

2.如何为创建大量实例节省内存

在游戏中,定义了玩家类player,每有一个在线玩家,在服务器内则有一个player的实例,当在线人数很多时,将产生大量实例(百万级)

如何降低这些大量实例的内存开销？

解决方案：

- 定义类的__slots__属性,声明实例有哪些属性(关闭动态绑定)

3.Python中的with语句

上下文管理器协议

contextlib简化上下文管理器

```
import contextlib

@contextlib.contextmanager
def file_open(filename):
```

```
    print("file open")
    yield {}
    print("file end")

with file_open('lib1.py') as f:
    print("file operation")
```

4.如何创建可管理的对象属性

在面向对象编程中,我们把方法看做对象的接口。直接访问对象的属性可能是不安全的,或设计上不够灵活,但是使用调用方法在形式上不如访问属性简洁。

5.如何让类支持比较操作

有时我们希望自定义类的实例间可以使用,<,<=,>,>=,==,!=符号进行比较,我们自定义比较的行业,例如,有一个矩形的类,比较两个矩形的实例时,比较的是他们的面积

6.如何在环状数据结构中管理内存

双向循环链表

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    def add_right(self, node):
        self.right = node
        node.left = self

    def __str__(self):
        return 'Node:<%s>' % self.data

    def __del__(self):
        print('in __del__: delete %s' % self)

def create_linklist(n):
    head = current = Node(1)
    for i in range(2, n + 1):
        node = Node(i)
        current.add_right(node)
        current = node
    return head

head = create_linklist(1000)
head = None
```

```
import time
for _ in range(1000):
    time.sleep(1)
    print('run...')
input('wait...')
```

通过实例方法名字的字符串调用方法

我们有三个图形类

```
Circle,Triangle,Rectangle
```

他们都有一个获取图形面积的方法,但是方法名字不同,我们可以实现一个统一的获取面积的函数,使用每种方法名进行尝试,调用相应类的接口

```
class Triangle:
    def __init__(self,a,b,c):
        self.a,self.b,self.c = a,b,c

    def get_area(self):
        a,b,c = self.a,self.b,self.c
        p = (a+b+c)/2
        return (p * (p-a)*(p-b)*(p-c)) ** 0.5

class Rectangle:
    def __init__(self,a,b):
        self.a,self.b = a,b

    def getArea(self):
        return self.a * self.b

class Circle:
    def __init__(self,r):
        self.r = r

    def area(self):
        return self.r ** 2 * 3.14159
```

3-Python 垃圾回收机制

Python 垃圾回收机制

计数引用我们反复提过好几次， Python 中一切皆对象。因此，你所看到的一切变量，本质上都是对象的一个指针。

那么，怎么知道一个对象，是否永远都不能被调用了呢？

就是当这个对象的引用计数（指针数）为 0 的时候，说明这个对象永不可达，自然它也就成为了垃圾，需要被回收。

```
import os
import psutil

# 显示当前 python 程序占用的内存大小
def show_memory_info(hint):
    pid = os.getpid()
    p = psutil.Process(pid)

    info = p.memory_full_info()
    memory = info.uss / 1024. / 1024
    print('{} memory used: {} MB'.format(hint, memory))


def func():
    show_memory_info('initial')
    a = [i for i in range(10000000)]
    show_memory_info('after a created')

func()
show_memory_info('finished')
```

循环引用

如果有两个对象，它们互相引用，并且不再被别的对象所引用，那么它们应该被垃圾回收吗？

```
def func():
    show_memory_info('initial')
    a = [i for i in range(10000000)]
    b = [i for i in range(10000000)]
    show_memory_info('after a, b created')
    a.append(b)
    b.append(a)

func()
show_memory_info('finished')
```

调试内存泄漏

虽然有了自动回收机制，但这也不是万能的，难免还是会有漏网之鱼。内存泄漏是我们不想见到的，而且还会严重影响性能。有没有什么好的调试手段呢？

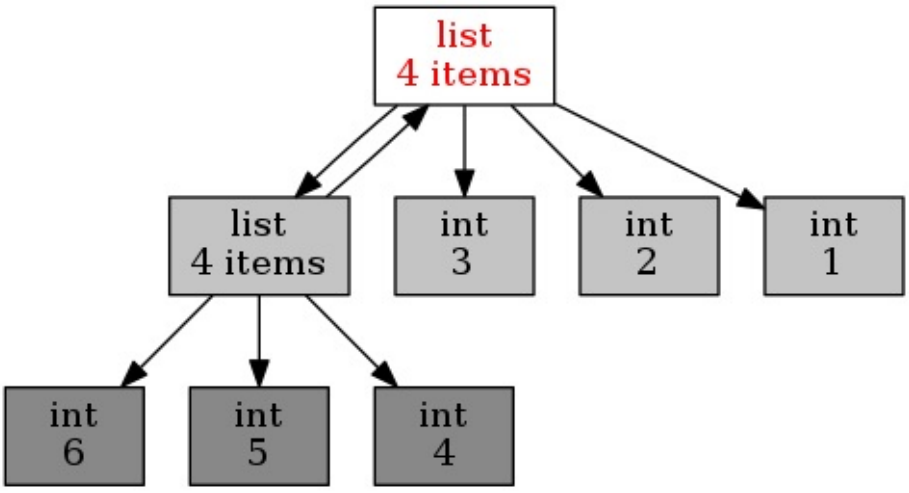
它就是 objgraph，一个非常好用的可视化引用关系的包。在这个包中，我主要推荐两个函数，第一个是 show_refs()，它可以生成清晰的引用关系图。

```
import objgraph

a = [1, 2, 3]
b = [4, 5, 6]

a.append(b)
b.append(a)

objgraph.show_refs([a])
```



dot转图片:https://onlineconvertfree.com/zh/

总结

- 垃圾回收是 Python 自带的机制，用于自动释放不会再用到的内存空间；
- 引用计数是其中最简单的实现，不过切记，这只是充分非必要条件，因为循环引用需要通过不可达判定，来确定是否可以回收；
- Python 的自动回收算法包括标记清除和分代收集，主要针对的是循环引用的垃圾收集；
- 调试内存泄漏方面， objgraph 是很好的可视化分析工具。

4-代码调试和性能分析

调试和性能分析

用 pdb 进行代码调试

首先，我们来看代码的调试。也许不少人会有疑问：代码调试？说白了不就是在程序中使用 print() 语句吗？

没错，在程序中相应的地方打印，的确是调试程序的一个常用手段，但这只适用于小型程序。因为你每次都得重新运行整个程序，或是一个完整的功能模块，才能看到打印出来的变量值。如果程序不大，每次运行都非常快，那么使用 print()，的确是很方便的。

可能又有人会说，现在很多的 IDE 不都有内置的 debug 工具吗？

如何使用 pdb

首先，要启动 pdb 调试，我们只需要在程序中，加入 `import pdb` 和 `pdb.set_trace()` 这两行代码就行了

```
a = 1
b = 2
import pdb
pdb.set_trace()
c = 3
print(a + b + c)
```

这时，我们就可以执行，在 IDE 断点调试器中可以执行的一切操作，比如打印，语法是"p "：

```
(pdb) p a
1
(pdb) p b
2
```

除了打印，常见的操作还有“n”，表示继续执行代码到下一行

```
(pdb) n
-> print(a + b + c)
```

而命令 `l` ，则表示列举出当前代码行上下的 11 行源代码，方便开发者熟悉当前断点周围的代码状态

```
(pdb) l
  1     a = 1
  2     b = 2
  3     import pdb
  4     pdb.set_trace()
  5  ->  c = 3
  6     print(a + b + c)
```

命令“s ”，就是 step into 的意思，即进入相对应的代码内部。

当然，除了这些常用命令，还有许多其他的命令可以使用

参考对应的官方文档：https://docs.python.org/3/library/pdb.html#module-pdb%EF%BC%89

用 cProfile 进行性能分析

关于调试的内容，我主要先讲这么多。事实上，除了要对程序进行调试，性能分析也是每个开发者的必备技能。

日常工作中，我们常常会遇到这样的问题：在线上，我发现产品的某个功能模块效率低下，延迟高，占用的资源多，但却不知道是哪里出了问题。

这时，对代码进行 profile 就显得异常重要了。

这里所谓的 profile，是指对代码的每个部分进行动态的分析，比如准确计算出每个模块消耗的时间等。

计算斐波拉契数列，运用递归思想

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def fib_seq(n):
    res = []
    if n > 0:
        res.extend(fib_seq(n-1))
```

```
    res.append(fib(n))
    return res

fib_seq(30)
```

接下来，我想要测试一下这段代码总的效率以及各个部分的效率

```
import cProfile

cProfile.run('fib_seq(30)')
```

```
         7049218 function calls (96 primitive calls) in 1.417 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    1.417    1.417 test.py:1(<module>)
7049123/31    1.417    0.000    1.417    0.046 test.py:1(fib)
     31/1    0.000    0.000    1.417    1.417 test.py:9(fib_seq)
        1    0.000    0.000    1.417    1.417 {built-in method builtins.exec}
       31    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
       30    0.000    0.000    0.000    0.000 {method 'extend' of 'list' objects}
```

参数介绍：

- ncalls，是指相应代码 / 函数被调用的次数
- tottime，是指对应代码 / 函数总共执行所需要的时间（注意，并不包括它调用的其他代码 / 函数的执行时间）
- tottime percall，就是上述两者相除的结果，也就是tottime / ncalls
- cumtime，则是指对应代码 / 函数总共执行所需要的时间，这里包括了它调用的其他代码 / 函数的执行时间
- cumtime percall，则是 cumtime 和 ncalls 相除的平均结果。

5-经典的参数错误

经典的参数错误

```
def add(a,b):
    a += b
    return a

a = 1
b = 2
c = add(a,b)
print(c)
print(a,b)

a = [1,2]
b = [3,4]
c = add(a,b)
print(c)
print(a,b)

a = (1,2)
b = (3,4)
c = add(a,b)
print(c)
print(a,b)
```

不可变类型

以int类型为例:实际上 i += 1 并不是真的在原有的int对象上+1，而是重新创建一个value为6的int对象，i引用自这个新的对象。

可变类型

以list为例。list在append之后，还是指向同个内存地址，因为list是可变类型，可以在原处修改。

6-元类编程

__getattr__和__getattribute__魔法函数

```
from datetime import date,datetime

class User:
    def __init__(self,name,birthday):
        self.name = name

        self.birthday = birthday

    def __getattr__(self,item):
        print("not find attr")

    def __getattribute__(self,item):
        return "juran"

if __name__ == "__main__":
    user = User("juran",date(year=1990,month=1,day=1))
    print(user.age)
```

属性描述符

```
class User:
    def __init__(self,age):
        self.age = age

    def get_age(self):
        return (str(self.age) + '岁')

    def set_age(self,age):
        if not isinstance(age,int):
            raise TypeError('Type Error')
        self.age = age
```

如果User类中有多个属性都需要判断,那么就需要写多个方法,这些方法怎么复用呢？这个时候就要用到属性描述符

> 属性描述符,只要实现了__get__,__set__,__delete__任何一个方法,就被称为属性描述符

属性查找顺序

```
user = User(), 那么user.age 顺序如下:

1 如果"age"是出现在User或其基类的__dict__中, 且age是data descriptor,那么调用其__get__方法, 否则

2 如果"age"出现在user的__dict__中, 那么直接返回 obj.__dict__['age'],否则

3 如果"age"出现在User或其基类的__dict__中

3.1 如果age是non-data descriptor,那么调用其__get__方法,  否则

3.2 返回 __dict__['age']

4 如果User有__getattr__方法, 调用__getattr__方法, 否则

5 抛出AttributeError
```

自定义元类

动态创建类

```
def create_class(name):
    if name == "user":
        class User:
            def __str__(self):
                return "user"
        return User
    elif name == "student":
        class Student:
            def __str__(self):
                return "Student"
        return Student

if __name__ == "__main__":
    myclass = create_class('user')
    obj = myclass()
    print(obj)
    print(type(obj))
```

使用type创建类

type还可以动态的创建类,type(类名,由父类组成的元组,包含属性的字典)

- 第一个参数：name表示类名称，字符串类型
- 第二个参数：bases表示继承对象（父类），元组类型，单元素使用逗号
- 第三个参数：attr表示属性，这里可以填写类属性、类方式、静态方法，采用字典格式，key为属性名，value为属性值

metaclass属性

如果一个类中定义了metalass = xxx,Python就会用元类的方式来创建类

7-迭代器和生成器

迭代器

在介绍迭代器之前，先说明下迭代的概念：

迭代：通过for循环遍历对象的每一个元素的过程。

Python的for语法功能非常强大，可以遍历任何可迭代的对象。

在Python中，list/tuple/string/dict/set/bytes都是可以迭代的数据类型。

迭代器是什么？

迭代器是一种可以被遍历的对象，并且能作用于next()函数。迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。 迭代器只能往后遍历不能回溯，不像列表，你随时可以取后面的数据，也可以返回头取前面的数据。

生成器

有时候，序列或集合内的元素的个数非常巨大，如果全制造出来并放入内存，对计算机的压力是非常大的

生成器如何读取大文件

文件300G,文件比较特殊,一行 分隔符 {|}

```
def readlines(f,newline):
    buf = ""
    while True:
        while newline in buf:
            pos = buf.index(newline)
            yield buf[:pos]
            buf = buf[pos + len(newline):]
        chunk = f.read(4096*10)
        if not chunk:
            yield buf
            break
        buf += chunk
```

```
with open('demo.txt') as f:
    for line in readlines(f,"{|}"):
        print(line)
```

8-Python socket编程

Python socket编程

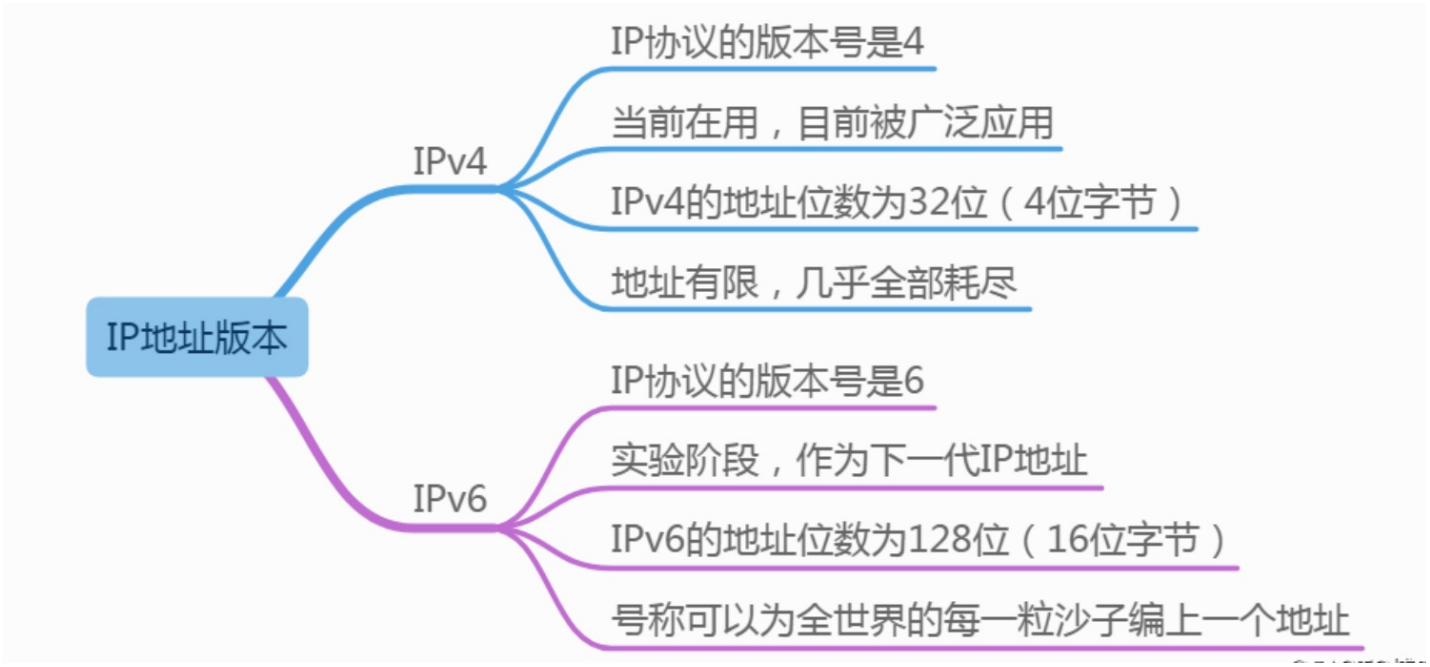
IP地址

目的:用来标记网络上的一台电脑

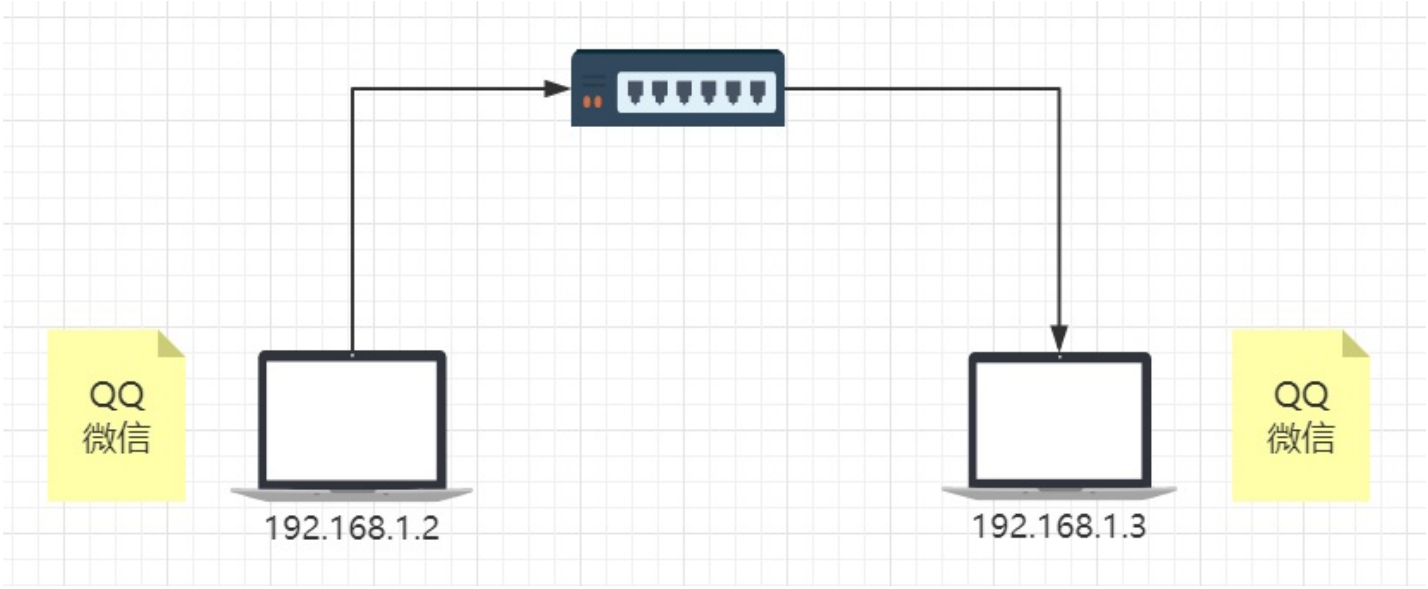
windows和Linux查看网卡信息

- Linux中 ifconfig
- windows中 ipconfig

IP地址的分类



端口



端口分类

1.知名端口(well known ports)

- 80端口分配给HTTP服务
- 21端口分配给FTP服务
  - 范围是从0到1023

2.动态端口

动态端口的范围是从1024-65535

socket简介

TCP/IP协议

TCP/IP协议是Transmission Control Protocol/Internet Protocol的简写，即传输控制协议/因特网互联协议，又名网络通讯协议，是Internet最基本的协议、Internet国际互联网络的基础，由网络层的IP协议和传输层的TCP协议组成。

TCP/IP 定义了电子设备如何连入因特网，以及数据如何在它们之间传输的标准。协议采用了4层的层级结构，每一层都呼叫它的下一层所提供的协议来完成自己的需求。

TCP/IP协议

TCP/IP网络模型四层模型从根本上和OSI七层网络模型是一样的，只是合并了几层

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	HTTP、TFTP, FTP, NFS, WAIS、SMTP
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）	网络层	IP, ICMP, ARP, RARP, AKP, UUCP
数据链路层（Data Link）	数据链路层	FDDI, Ethernet, Arpanet, PDN, SLIP, PPP
物理层（Physical）		IEEE 802.1A, IEEE 802.2到IEEE 802.11

socket

socket又称"套接字",应用程序通常通过"套接字"向网络发出请求或者应答网络请求,使主机间或者一台计算机上的进程间可以通讯。白话说,socket就是两个节点为了互相通信,而在各自家里装的一部'电话'。

socket的使用

1.创建套接字

2.使用套接字收/发数据

3.关闭套接字

udp发送与接收程序

> udp发送数据

```
import socket

def main():
    # 创建一个udp套接字
    udp_socket = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
    udp_socket.sendto(b'nihao',('192.168.0.162',8080))
    # 关闭socket
    udp_socket.close()

if __name__ == '__main__':
    main()
```

> udp接收数据

1 创建套接字

2 绑定本地信息(IP和端口)

3 接受数据

4 打印数据

5 关闭套接字

端口绑定问题

如果程序运行时，没有绑定端口，那么操作系统会自动分配一个端口给程序。 而且同一端口，不能用两次。

udp聊天器

1.创建套接字 套接字是可以同时收发数据的

2.发送数据

3.接收数据

TCP介绍

- TCP协议，传输控制协议，是一种面向连接的、可靠的、基于字节流的传输层通信协议
- TCP通信需要经过创建连接、数据传送、终止连接三个步骤。
- TCP通信模型中，在通信开始之前，一定要先建立相关连接，才能发生数据。

TCP特点

- 面向连接
  - 通信双方必须先建立连接才能进行数据的传输
- 可靠传输
  - TCP采用发送应答机制
  - 超时重传
  - 错误校验
  - 流量控制和阻塞管理

TCP与UDP区别总结

1、TCP面向连接;UDP是无连接的，即发送数据之前不需要建立连接

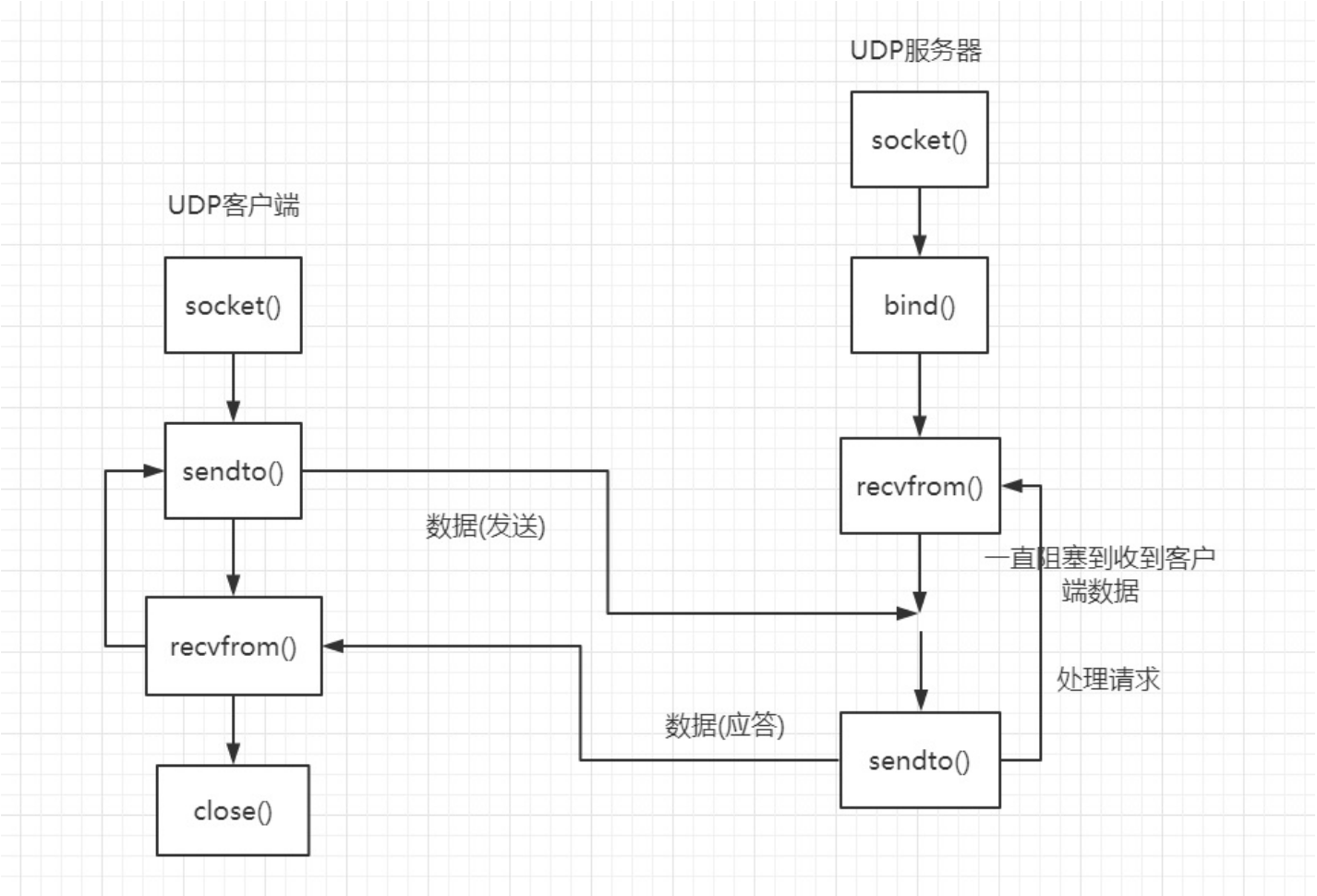
2、TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达;UDP尽最大努力交付，即不保证可靠交付

3、UDP具有较好的实时性，工作效率比TCP高，适用于对高速传输和实时性有较高的通信或广播通信。

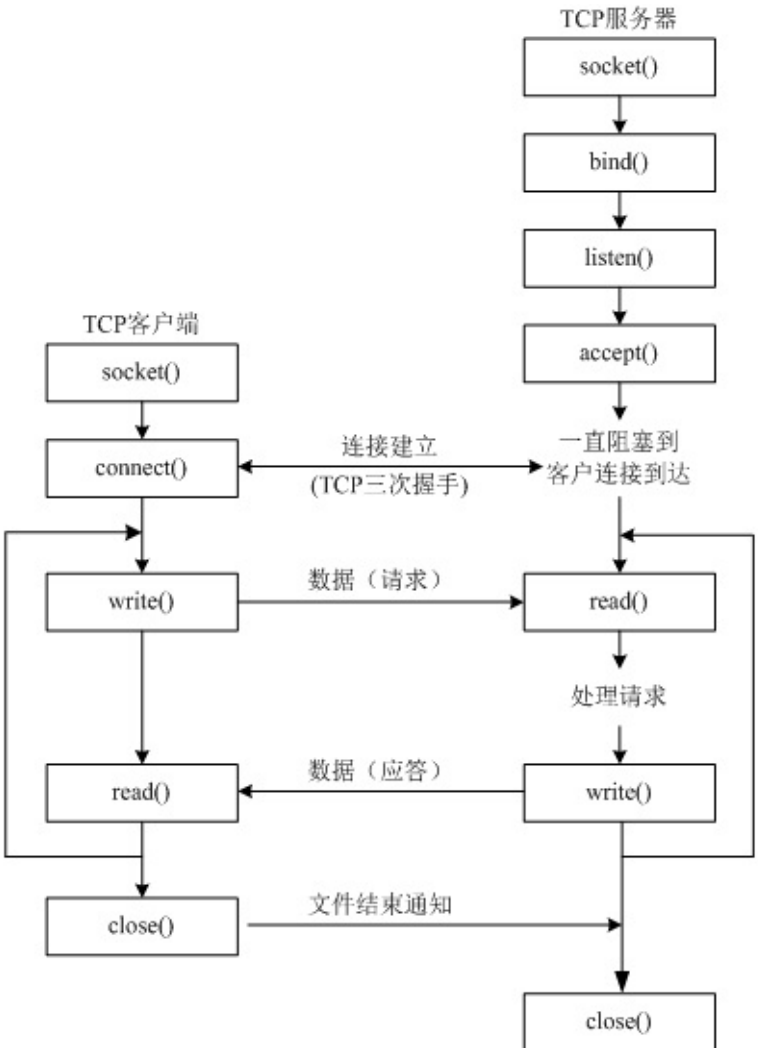
4、每一条TCP连接只能是点到点的;UDP支持一对一，一对多，多对一和多对多的交互通信

5、TCP对系统资源要求较多，UDP对系统资源要求较少。

UDP通信


UDP服务器
socket()
bind()
recvfrom()
一直阻塞到收到客户端数据
处理请求
sendto()
数据(应答)
UDP客户端
socket()
sendto()
数据(发送)
recvfrom()
close()

TCP通信



TCP客户端

服务器端：就是提供服务的一方，而客户端，就是需要被服务的一方

TCP客户端构建流程

1.创建socket

2.链接服务器

3.接收数据(最大接收2014个字节)

4.关闭套接字

TCP服务端

1 socket创建套接字

2 bind绑定IP和port

3 listen使套接字变为可以被动链接

4 accept等待客户端的链接

5 recv/send接收发送数据

TCP服务端为多个客户端服务

文件下载器

TCP客户端

1.创建套接字

2.目的信息 服务器的ip port

3.连接服务器

4.输入要下载的文件名称

5.发送文件下载请求

6.接收对方发送过来的数据

7.接收到数据在创建文件

8.关闭套接字

TCP服务端

1 socket创建套接字

2 bind绑定IP和port

3 listen使套接字变为可以被动链接

4 accept等待客户端的链接

5 recv/send接收发送数据

9-Python多任务-线程

多任务

有很多的场景中的事情是同时进行的，比如开车的时候 手和脚共同来驾驶汽车，再比如唱歌跳舞也是同时进行的

程序中模拟多任务

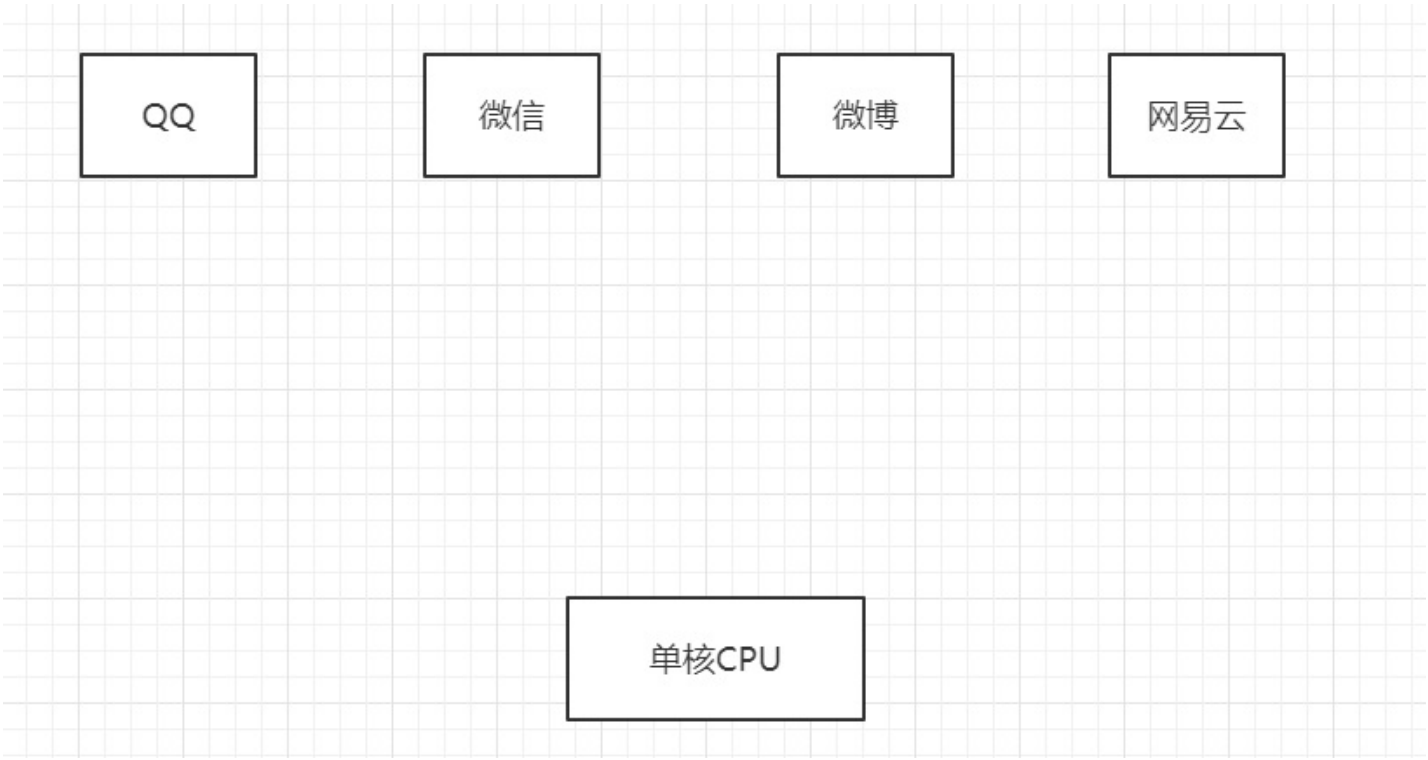
```
import time

def sing():
    for i in range(3):
        print("正在唱歌...%d"%i)
        time.sleep(1)

def dance():
    for i in range(3):
        print("正在跳舞...%d"%i)
        time.sleep(1)

if __name__ == '__main__':
    sing()
    dance()
```

多任务的理解



- 并行:真的多任务 cpu大于当前执行的任务
- 并发:假的多任务 cpu小于当前执行的任务

线程完成多任务

```
import threading
import time

def demo():
    # 子线程
    print("hello girls")
    time.sleep(1)

if __name__ == "__main__":
    for i in range(5):
        t = threading.Thread(target=demo)
        t.start()
```

查看线程数量

```
threading.enumerate()    查看当前线程的数量
```

验证子线程的执行与创建

当调用Thread的时候，不会创建线程。

当调用Thread创建出来的实例对象的start方法的时候，才会创建线程以及开始运行这个线程。

继承Thread类创建线程

```
import threading
import time

class A(threading.Thread):

    def __init__(self,name):
        super().__init__(name=name)

    def run(self):
        for i in range(5):
            print(i)

if __name__ == "__main__":
    t = A('test_name')
    t.start()
```

多线程共享全局变量(线程间通信)

修改全局变量一定需要加global嘛?

在一个函数中，对全局变量进行修改的时候，是否要加global要看是否对全局变量的指向进行了修改，如果修改了指向，那么必须使用global，仅仅是修改了指向的空间中的数据，此时不用必须使用global

多线程参数-args

```
threading.Thread(target=test, args=(num,))
```

共享全局变量资源竞争

一个线程写入,一个线程读取,没问题,如果两个线程都写入呢？

互斥锁

当多个线程几乎同时修改某一个共享数据的时候，需要进行同步控制

某个线程要更改共享数据时，先将其锁定，此时资源的状态为"锁定",其他线程不能改变，只到该线程释放资源，将资源的状态变成"非锁定"，其他的线程才能再次锁定该资源。互斥锁保证了每次只有一个线程进行写入操作，从而保证了多线程情况下数据的正确性。

```
创建锁
mutex = threading.Lock()
```

```
锁定
mutex.acquire()

解锁
mutex.release()
```

死锁

在线程间共享多个资源的时候，如果两个线程分别占有一部分资源并且同时等待对方的资源，就会造成死锁。

```
import threading
import time

class MyThread1(threading.Thread):
    def run(self):
        # 对mutexA上锁
        mutexA.acquire()

        # mutexA上锁后，延时1秒，等待另外那个线程 把mutexB上锁
        print(self.name+'----do1---up----')
        time.sleep(1)

        # 此时会堵塞，因为这个mutexB已经被另外的线程抢先上锁了
        mutexB.acquire()
        print(self.name+'----do1---down----')
        mutexB.release()

        # 对mutexA解锁
        mutexA.release()

class MyThread2(threading.Thread):
    def run(self):
        # 对mutexB上锁
        mutexB.acquire()

        # mutexB上锁后，延时1秒，等待另外那个线程 把mutexA上锁
        print(self.name+'----do2---up----')
        time.sleep(1)

        # 此时会堵塞，因为这个mutexA已经被另外的线程抢先上锁了
        mutexA.acquire()
        print(self.name+'----do2---down----')
        mutexA.release()

        # 对mutexB解锁
        mutexB.release()

mutexA = threading.Lock()
mutexB = threading.Lock()
```

```
if __name__ == '__main__':
    t1 = MyThread1()
    t2 = MyThread2()
    t1.start()
    t2.start()
```

避免死锁

- 程序设计时要尽量避免
- 添加超时时间等

线程同步

天猫精灵:小爱同学

小爱同学:在

天猫精灵:现在几点了？

小爱同学：你猜猜现在几点了

多任务版udp聊天

1 创建套接字

2 绑定本地信息

3 获取对方IP和端口

4 发送、接收数据

5 创建两个线程，去执行功能

10-Python多任务-进程

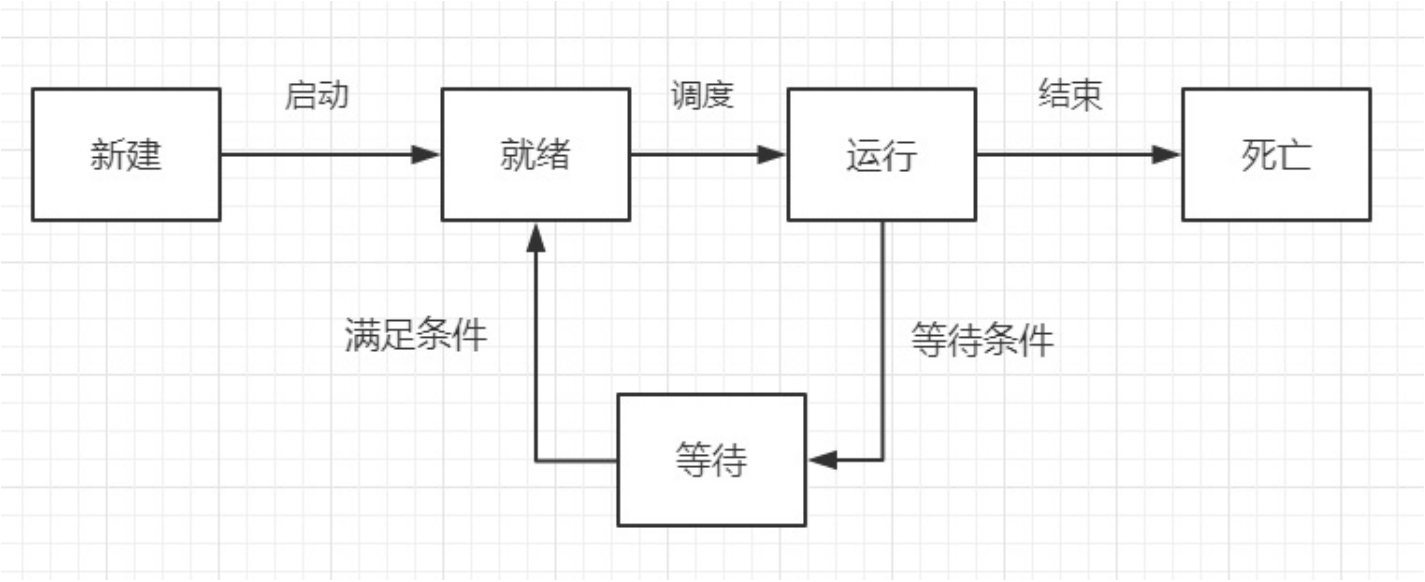
Python多任务-进程

进程和程序

进程：正在执行的程序

程序：没有执行的代码，是一个静态的

进程的状态



使用进程实现多任务

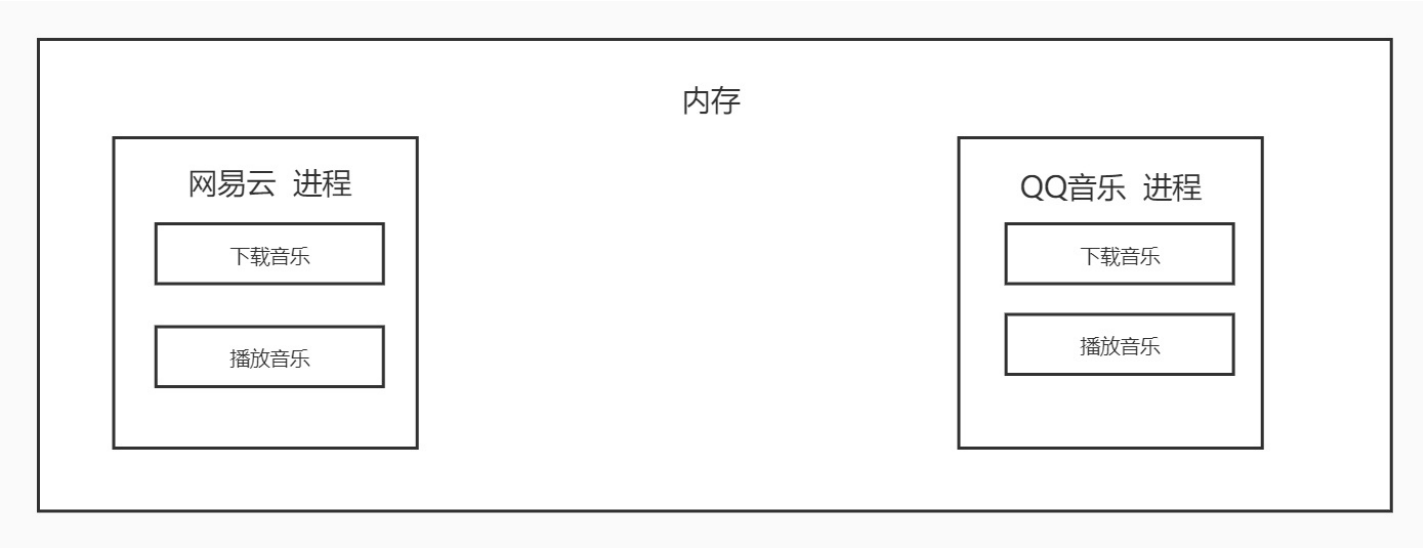
multiprocessing模块就是跨平台的多进程模块，提供了一个Process类来代表一个进程对象，这个对象可以理解为是一个独立的进程，可以执行另外的事情。

线程和进程之间的对比

进程：能够完成多任务，一台电脑上可以同时运行多个QQ

线程：能够完成多任务，一个QQ中的多个聊天窗口

根本区别：进程是操作系统资源分配的基本单位，而线程是任务调度和执行的基本单位



进程间通信-Queue

Queue-队列 先进先出

队列间简单通信

模拟下载数据,与数据处理

多进程共享全局变量

共享全局变量不适用于多进程编程

进程池

当需要创建的子进程数量不多时，可以直接利用multiprocessing中的Process动态生成多个进程，但是如果是上百甚至上千个目标，手动的去创建的进程的工作量巨大，此时就可以用到multiprocessing模块提供的Pool方法

初始化Pool时，可以指定一个最大进程数，当有新的请求提交到Pool中时，如果池还没有满，那么就会创建一个新的进程用来执行该请求，但是如果池中的进程数已经达到指定的最大值，那么该请求就会等待，直到池中有进程结束，才会用之前的进程来执行新的任务

```
from multiprocessing import Pool

import os,time,random

def worker(msg):
    t_start = time.time()
    print('%s开始执行,进程号为%d'%(msg,os.getpid()))

    time.sleep(random.random()*2)
    t_stop = time.time()
    print(msg,"执行完成,耗时%0.2f"%(t_stop-t_start))

def demo():
```

```
    pass

if __name__ == '__main__':
    po = Pool(3)        # 定义一个进程池
    for i in range(0,10):
        po.apply_async(worker,(i,))

    print("--start--")
    po.close()

    po.join()
    print("--end--")
```

进程池间的进程通信

多任务文件夹复制

1 获取用户要copy的文件夹的名次

2 创建一个新的文件夹

3 获取文件夹的所有的待copy的文件名字

4 创建进程池

5 向进程池中添加拷贝任务

11-Python多任务-协程

Python多任务-协程

同步、异步

同步:是指代码调用IO操作时,必须等待IO操作完成才返回的调用方式

异步:是指代码调用IO操作时,不必等IO操作完成就返回的调用方式

阻塞、非阻塞

阻塞:从调用者的角度出发，如果在调用的时候，被卡住，不能再继续向下运行，需要等待，就说是阻塞

非阻塞:从调用者的角度出发， 如果在调用的时候，没有被卡住，能够继续向下运行，无需等待，就说是非阻塞

生成器-send方法

send方法有一个参数，该参数指定的是上一次被挂起的yield语句的返回值

使用yield完成多任务

```
import time

def task1():
    while True:
        print("--1--")
        time.sleep(0.1)
        yield

def task2():
    while True:
        print("--2--")
        time.sleep(0.1)
        yield

def main():
    t1 = task1()
```

```
    t2 = task2()
    while True:
        next(t1)
        next(t2)

if __name__ == "__main__":
    main()
```

yield from介绍

python3.3新加了yield from语法

```
def generator_1():
    total = 0
    while True:
        x = yield
        print('加', x)
        if not x:
            break
        total += x
    return total


def generator_2():  # 委托生成器
    while True:
        total = yield from generator_1()  # 子生成器
        print('加和总数是:', total)


def main():  # 调用方
    # g1 = generator_1()
    # g1.send(None)
    # g1.send(2)
    # g1.send(3)
    # g1.send(None)
    g2 = generator_2()
    g2.send(None)
    g2.send(2)
    g2.send(3)
    g2.send(None)


if __name__ == '__main__':
    main()
```

【子生成器】：yield from后的generator_1()生成器函数是子生成器
【委托生成器】：generator_2()是程序中的委托生成器，它负责委托子生成器完成具体任务。
【调用方】：main()是程序中的调用方，负责调用委托生成器。

协程

协程，又称微线程

协程是python个中另外一种实现多任务的方式，只不过比线程更小占用更小执行单元（理解为需要的资源）

Python中的协程大概经历了如下三个阶段：

1. 最初的生成器变形yield/send
2. yield from
3. 在最近的Python3.5版本中引入async/await关键字

使用greenlet完成多任务

安装模块：pip3 install greenlet

使用gevent完成多任务

安装模块：pip3 install gevent

简单总结

- 进程是资源分配的单位
- 线程是操作系统调度的单位
- 进程切换需要的资源很最大，效率很低
- 线程切换需要的资源一般，效率一般（当然了在不考虑GIL的情况下）
- 协程切换任务资源很小，效率高
- 多进程、多线程根据cpu核数不一样可能是并行的，但是协程是在一个线程中 所以是并发