目录

[**零碎的问题** 2](#_Toc48593573)

[**内存管理和垃圾回收** 2](#_Toc48593574)

[**内存池机制** 2](#_Toc48593575)

[**垃圾回收机制** 2](#_Toc48593576)

[**python中有垃圾回收机制，在以下情况下仍然会有内存泄漏的问题：** 2](#_Toc48593577)

[**list、str、dict、set实现细节和区别** 2](#_Toc48593578)

[**list** 2](#_Toc48593579)

[**tuple** 3](#_Toc48593580)

[**dict:** 3](#_Toc48593581)

[**set** 3](#_Toc48593582)

[**frozenset** 3](#_Toc48593583)

[**为什么list不能当dict的key** 3](#_Toc48593584)

[**tuple是不是冗余设计** 3](#_Toc48593585)

[**可迭代对象collections.abc.Iterable、迭代器Iterator、生成器Generator** 3](#_Toc48593586)

[**哪些是可迭代对象** 3](#_Toc48593587)

[**为什么要有迭代器：** 3](#_Toc48593588)

[**关于for循环** 4](#_Toc48593589)

[**生成器既是可迭代对象也是迭代器** 4](#_Toc48593590)

[**什么是协程** 4](#_Toc48593591)

[**内存溢出和内存泄漏的区别** 4](#_Toc48593592)

[**直接赋值、深拷贝和浅拷贝** 5](#_Toc48593593)

[**可变类型和不可变类型** 5](#_Toc48593594)

[**解释型语言和编译型语言的区别** 5](#_Toc48593595)

[**为什么说python比较慢？** 5](#_Toc48593596)

[**‘\_\_new\_\_’与‘\_\_init\_\_’的区别** 5](#_Toc48593597)

[**@classmethod 和 @staticmethod 的区别，以及分别运用在哪些使用场景？** 6](#_Toc48593598)

[**切片** 6](#_Toc48593599)

[**python2和python3的区别** 6](#_Toc48593600)

[**python v.s. java** 6](#_Toc48593601)

[**高阶函数** 7](#_Toc48593602)

[**is与==的区别** 7](#_Toc48593603)

[**python中的参数类型** 7](#_Toc48593604)

[**单下划线和双下划线的区别** 7](#_Toc48593605)

[**重写和重载的区别** 7](#_Toc48593606)

[上下文管理器 7](#_Toc48593607)

[**强类型、弱类型、动态语言、静态语言** 8](#_Toc48593608)

[**鸭子类型** 8](#_Toc48593609)

[**元类** 9](#_Toc48593610)

[**python多继承的顺序是什么** 9](#_Toc48593611)

**零碎的问题**

**内存管理和垃圾回收**

**内存池机制**

预先在内存中申请一定数量的，大小相等的内存块留作备用，当有新的内存需求时，就先从内存池中分配内存给这个需求，不够了之后再申请新的内存。这样做最显著的优势就是能够减少内存碎片，提升效率。

**垃圾回收机制**

内存池机制具有内存的创建那就需要垃圾回收。python 采用了 引用计数 为主， 标志清除和分代回收 为辅测策略。

**引用计数**：每一个对象，在源码里就是一个结构体表示，都会有一个计数字段。当一个对象有新的引用时，它的引用计数就会增加，当引用它的对象被删除，它的引用计数就会减少。 一旦对象的引用计数为0，该对象立即被回收，对象占用的内存空间将被释放。

优点:1)简单2)实时:一旦没有引用，内存就直接释放了。不用像其他机制等到特定时机。

缺点:1)不能解决对象的循环引用2) 需要额外的空间维护引用计数。

**标记清除：**

1. 标记阶段，gc会把所有的活动对象打上标记。

对象之间通过引用（指针）连在一起，构成一个有向图，对象构成这个有向图的节点，而引用关系构成这个有向图的边。从根对象出发，沿着有向边遍历对象，可达的对象标记为活动对象，不可达的对象就是要被清除的非活动对象。**根对象就是全局变量、调用栈、寄存器。**

1. 清除阶段 把没有标记的对象，视作非活动对象，进行回收。

优点：主要解决循环引用问题。主要处理的是容器对象，e.g.list、dict、tuple，因为数值、字符串对象不可能有循环引用问题。

缺点:清楚非活动对象前，必须顺序扫描整个堆内存，哪怕只剩下小部分活动对象，也要扫描所有对象。

**分代回收**：一种空间换时间的方式。根据对象的存活时间分为不同的集合，每个集合称为一个代，python分了三代，年轻代，中年代，老年代，对应三个链表，每个链表垃圾收集的频率随着对象存活时间的增大而减小。新创建的对象都会分配在年轻代，年轻代链表的总数达到上限时，Python垃圾收集机制就会被触发，把那些可以被回收的对象回收掉，而那些不会回收的对象就会被移到中年代去，依此类推，老年代中的对象是存活时间最久的对象，甚至是存活于整个系统的生命周期内。同时，分代回收是建立在标记清除技术基础之上。也是处理的容器对象

**python中有垃圾回收机制，在以下情况下仍然会有内存泄漏的问题：**

1）对象一直被全局变量所引用, 全局变量生命周期长.

2）垃圾回收机被禁用或者设置成debug状态, 垃圾回收的内存不会被释放.

3）循环引用过多, gc 无法及时释放。

**list、str、dict、set实现细节和区别**

**list**

是一个指针数组 PyObject \*\* obitem。可以这么理解，链表的每一个节点是一个元素，而在list中，链表的每一个节点是一个数组。而且长度可变，所以一开始申请list的时候，会多申请一些冗余空间，减少重新分配的次数

**tuple**

也是一个指针数组，但是空间大小固定 static pyobject \* free\_list[Tuple\_SIZE].而且python还做了优化，e.g. 小于一定大小的元组，在释放的时候会被放进这个 free\_list 中以供下次使用。也就是说，如果以后需要再去创建同样的 tuple，Python 就可以直接从缓存中载入。

**dict:**

字典底层维护一张哈希表，把哈希表看成一个列表，列表中的值存储了哈希值、键、值3个元素。E.g. entities = [[‘—‘,’—‘,’—'],[hash,key,value]…]

当插入一个键值对时，首先计算index=hash(key)&mask(mask为字典长度-1),若index已被占用，则判断两者key是否相等，相等就直接更新value值，不相等就继续向下寻找空位置(开放寻址法)。

p.s. entities是稀疏的，无序的，会动态扩展长度，每次的扩展都会重新计算所有key的hash值。

Python3.7以后，使用了indice表辅助字典，使得遍历字典时，按照插入的顺序遍历。

**set**

可变、无序。与字典的实现颇为相似，集合被实现为带有空值的字典，键是集合元素。

**frozenset**

不可变，无序

**为什么list不能当dict的key**

dict必须要保证当同样的内容进来的时候，能够找到该内容对应的值，也就是如果内容相同对应的哈希值应当相同。

而对于list的hash函数，可能存在两种实现方式：

1）基于id。基于id实现hash可以保证，两个对象的hash值不同，他们的id不同，但是,当两个对象的内容相同时，他们的id也可能不同，创建以个一模一样的list用字典查找永远会得到一个KeyError。

2)基于内容。tuple就是这样做的，tuple不可修改，基于内容方法对于同一个tuple来说其hash值不会改变，而对于同一个list而言，list的内容改变了，其hash值就改变了，因此，对于某一个list来说，基于内容的方法不可能生成一个固定的哈希值。

**tuple是不是冗余设计**

1）tuple不可变，可哈希，可作为字典的key

2）当给一个函数传值时，不想让该函数改变参数的值，使用tuple

3）tuple不可变，不需要申请冗余空间，而且python对tuple做了优化，会比list快一点点点。

**可迭代对象collections.abc.Iterable、迭代器Iterator、生成器Generator**

**哪些是可迭代对象**

字符串、列表、元组、集合、字典、文件

类中定义了\_\_iter\_\_的对象。

**为什么要有迭代器：**

字符串、列表、元组支持索引，我们可以使用while循环取出其中的元素，但是集合、字典、文件就不可以了，因此引入迭代器。

优点：1)为有序的和无序的类型提供了一种统一的迭代取值方式。

2)惰性计算，每迭代一次才会计算一次。这样使得迭代器可以表示一个计算流，对于迭代器来说，同一时刻，内存中只有一个值，所以可以给出无限量的数据流。而可迭代对象则需要将所有数据全部放入内存中。

缺点：1)除非迭代到抛StopIteration异常，否则无法知道迭代器长度。

2)同一时间只能取一个值，而且只能往前不能后退，若想再次取上次取到的值，就只能从头开始next。

iter(可迭代对象x)可以返回x对应的迭代器e.g.x为列表，就返回一个列表迭代器；若x不是可迭代对象，但是x内部实现了\_\_getitem\_\_，也可以，因为iter如果找不到\_\_iter\_\_,回去找\_\_getitem\_\_拿到第一个元素。

**关于for循环**

for循环又叫可迭代循环，首先for循环拿到一个可迭代对象，通过iter()将其转化成迭代器，迭代器通过next()拿到返回值，并将返回值赋值给对象，直到跑出StopIteration，for循环会捕获异常，结束循环。

**生成器既是可迭代对象也是迭代器**

1）列表生成器 L = （x for x in range(10)）

2）使用yield定义生成器函数: e.g.写一个斐波那契数列生成函数fib

python中虚拟机类似程序在x86机器上运行时栈的形式，以栈帧为基本单位，形成一个栈帧链，执行的时候在这些栈帧链中进行切换。在python中，一个模块、类以及函数的执行都会产生一个栈帧，然后执行这个栈帧。栈帧中保存了代码的信息和上下文，包括最后执行的指令、变量、异常状态等。

每次运行到yield的时候函数就主动挂起，交出运行权，并保存该函数的栈帧，下次再迭代的时候(使用next或send)，恢复栈帧中的数据。

p.s. 迭代器、生成器中的值通过next()只能读一次

**什么是协程**

Python有一个GIL（Global Interpreter Lock）机制(有了这样一个设定，就不用管线程之间的加锁、同步等问题了)，任何线程在运行之前必须获取这个全局锁才能执行，每当执行完100条字节码，全局锁才会释放，切换到其他线程执行。在python GIL之下，同一时刻只能有一个线程在运行，python只能使用到1个核。

而使用多进程虽然能够利用到多核，但是进程启动时间长，耗费内存多。

引入协程：

又称微线程、纤程，是一种伪多线程，其不需要像线程一样上下文切换(协程在用户态执行)，开销远小于线程的开销。

p.s. 进程由系统分配资源、线程是由CPU调度、协程由用户(程序)控制。

协程适合处理I/O密集型程序，当程序在执行 I/O 时操作时，CPU 是空闲的，此时可以充分利用 CPU 的时间片来处理其他任务。在单线程中，一个函数调用，一般是从函数的第一行代码开始执行，结束于 return 语句、异常或者函数执行（也可以认为是隐式地返回了 None ）。 有了协程，我们在函数的执行过程中，如果遇到了耗时的 I/O 操作，函数可以临时让出控制权(产生软件中断)，让 CPU 执行其他函数，等 I/O 操作执行完毕以后再收回控制权。

协程的发展： 1）最初的生成器变形yield/send 2）引入@asyncio.coroutine和yield from 3）在最近的Python3.5版本中引入async/await关键字。

**内存溢出和内存泄漏的区别**

内存溢出 out of memory，是指程序在申请内存时，没有足够的内存空间供其使用，出现out of memory；比如申请了一个integer,但给它存了long才能存下的数，那就是内存溢出。

内存泄露 memory leak，是指程序在申请内存后，无法释放已申请的内存空间，一次内存泄露危害可以忽略，但内存泄露堆积后果很严重，无论多少内存,迟早会被占光。

**直接赋值、深拷贝和浅拷贝**

赋值，传对象的引用（别名）

浅拷贝:拷贝父对象，不会拷贝对象的内部的子对象

深拷贝: copy 模块的 deepcopy 方法，完全拷贝了父对象及其子对象

**可变类型和不可变类型**

可变：list、set、dict

不可变：int/float、str、tuple

**解释型语言和编译型语言的区别**

我们编程都是用的高级语言(写汇编和机器语言的大牛们除外)，计算机不能直接理解高级语言，只能理解和运行机器语言，所以必须要把高级语言翻译成机器语言，计算机才能运行。

编译型语言，在运行前进行翻译(编译)，这样运行时就可以直接执行了；

而解释型语言在运行的时候才翻译。

二者各有利弊：前者由于程序执行速度快，同等条件下对系统要求较低，因此像开发操作系统、大型应用程序、数据库系统等时都采用它，像C/C++、Pascal/Object Pascal（Delphi）等都是编译语言，而一些网页脚本、服务器脚本及辅助开发接口这样的对速度要求不高、对不同系统平台间的兼容性有一定要求的程序则通常使用解释性语言，如JavaScript、VBScript、Perl、Python、Ruby、MATLAB 等等。

编译型和解释型语言越来越笼统，主要体现在一些新兴的高级语言上，而解释型语言的自身特点也使得编译器厂商愿意花费更多成本来优化解释器，解释型语言性能超过编译型语言也是必然的。

**为什么说python比较慢？**

1. python是动态语言，事先不知道变量的类型
2. 是解释型语言而不是编译型语言
3. 其对象模型会导致访问内存的效率低下。比如说对于整数进行操作，python会有一个额外的类型信息层。

python优点:

1. python是动态语言，用法更加灵活和兼容。
2. python开源，跨平台，移植性强。
3. 很多需要用到c或fortran进行优化的场合，python都有强大的API或库进行支持。

**‘\_\_new\_\_’与‘\_\_init\_\_’的区别**

\_\_new\_\_(cls, \*args, \*\*kwargs):

是在新式类中新出现的方法，是类的静态方法，调用\_\_new\_\_时没实力还不存在，\_\_new\_\_会返回一个实例，充当构造器的作用。

\_\_init\_\_(self, \*args, \*\*kwargs):

\_\_init\_\_初始化类的属性，不能有返回值。而在\_\_init\_\_启动之前，\_\_new\_\_ 决定是否要使用该\_\_init\_\_ 方法，因为\_\_new\_\_可以调用其他类的构造方法或者直接返回别的对象来作为本类的实例。

在旧类中没有\_\_new\_\_，\_\_init\_\_其构造器的作用。

**@classmethod 和 @staticmethod 的区别，以及分别运用在哪些使用场景？**

1）classmethod的第一个参数为类本身(cls)，正如实例方法的第一个参数为对象本身(self);staticmethod第一个参数不需要传入cls或self，故staticmethod中是无法访问类和对象的数据的。

2）都可用类名直接调用；也可用实例对象调用（不推荐，没必要）

两者特点：

classmethod可以设置修改类属性；也可以实例化对象；

staticmethod无法访问类或对象的数据，所以可把它当作一个辅助功能方法用，里面包含一些与该类有关的逻辑代码。比如validate(\*args)🡪来对要初始化数据进行有效性检查。

**切片**

反转s[::-1] 倒着，每隔两个取s[::-2]

**python2和python3的区别**

1. 编码 py2默认asscii编码(8位2进制)，需要再顶部写# coding=utf-8 (py2诞生的时代还没有unicode)，py3默认utf-8。
   1. Unicode把所有语言都统一到一套编码里，这样就不会再有乱码问题了。当需要保存到硬盘或者需要传输的时候，就转换为UTF-8编码。UTF-8 是隶属于 Unicode 的可变长的编码方式。
   2. Unicode 方式编码的字符串，可以使用 encode() 方法来编码成指定的 bytes，也可以通过 decode() 方法来把 bytes 编码成字符串。
2. 输入python2 input()只接受数字，raw\_input()接受字符串，python3中只有input() ，接收字符串。
3. print py2是把括号里的内容当成一个整体，而py3则是把字符串看成参数。
4. 字符串py2字符串有两种类型，unicode和str,分别表示文本字符串和字节序列，两者没有明显的界限，py3做了严格的区分，str表示字符串，byte表示字节序列，任何需要写入文本或网络的数据只接收字节序列，从源头阻止了编码错误的问题。

字节🡪存储、传输用 字符🡪展示用

1. py2中True和Flase是两个全局变量，可被重新赋值；py3将其变为两个关键字，不可被赋值。
2. 整除 py2和py3都认为//是整除,不同的是 / ,3/2 python2是1,py3是1.5。3./2 py2才是1.5。
3. 迭代器，py2中很多返回列表对象的内置函数和方法在py3都改成了但会类似于迭代器的对象，因为迭代器的惰性加载特性使得操作大数据更有效率。

e.g. 字典dict.keys() dict.values() dict.items()  pyt2返回列表，py3返回view对象，类似迭代器。map filter 从函数变成了类，其返回结果从列表变成了可迭代对象。

1. nonlocal：global可用于函数内部修改全局变量的值，而在嵌套函数中想要声明一个全局变量是没办法实现的，py3新增nonlocal关键字，实现了该功能。

**python v.s. java**

1. python是动态语言，java是静态语言
2. python提供了各种库，java没有这么多开源库，一般是发布jar包，看不到原始代码。
3. python的好多程序是面向过程的，而java主要是面向对象设计方法。

**高阶函数**

map(func, \*iterables)🡪map(lambda x:x\*x,[1,2,3,4])

filter(func, \*iterables)🡪 map(lambda x:x>1,[1,2,3,4])

from functools import reduce

reduce(function,sequence,initial=None)🡪 reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) 🡪((((1+2)+3)+4)+5)=15 reduce(lambda x, y: x+y, [1, 2, 3, 4, 5],10)🡪25

**is与==的区别**

is比id,==比值。

**python中的参数类型**

位置参数、默认参数、可变参数（\*arg）、关键字参数(\*\*kargs)。

\*arg,\*\*kargs都是为了实现任意个参数的方式。\*arg会把位置参数转化为tuple，\*\*kargs会把关键字参数转化为dict。

p.s.**默认参数必须指向不变类型**e.g.None，如果默认参数指向[]，会造成严重bug!!!

**单下划线和双下划线的区别**

\_\_foo\_\_: 定义的是特殊方法，一般是系统定义名字，防止与用户定义的变量名冲突 ，类似 \_\_init\_\_() 之类的。

\_foo: 以单下划线开头的表示的是 protected 类型的变量，即保护类型只能允许其本身与子类进行访问，不能用于 from module import \*

\_\_foo: 双下划线的表示的是私有类型(private)的变量, 只能是允许这个类本身进行访问了。如果子类想要访问父类中的私有变量或方法，可以使用 实例.\_父类名\_\_foo

foo:就是public方法

**重写和重载的区别**

重写是子类对父类的允许访问的方法的实现过程进行重新编写, 返回值和形参都不能改变。即外壳不变，核心重写！

重载(overloading) 是在一个类里面，方法名字相同，而参数不同。返回类型可以相同也可以不同。每个重载的方法（或者构造函数）都必须有一个独一无二的参数类型列表。

上下文管理器

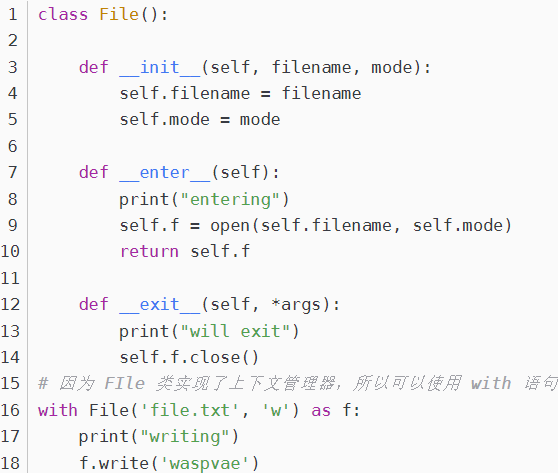
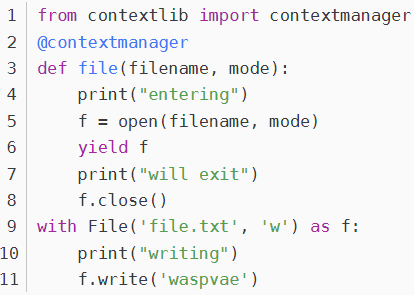
e.g.with open(‘…’,’r’) as f: with thread\_lock:

任何实现了\_\_enter\_\_和\_\_exit\_\_方法的对象都可被称为上下文管理器，上下文管理器对象可以使用with关键字。

1）with…as，紧跟with后面的语句被求值后，返回对象的\_\_enter\_\_方法，并将该方法的返回结果绑定到as后面的变量；当with后面的代码块全部被执行完以后，调用前面返回对象的\_\_exit\_\_方法。实际上，with后面代码块抛异常时，exit方法被执行。开发库时，清理资源、关闭文件等操作，都可放在exit方法中。简化了每次写finally的工作。

2）把生成器函数当成上下文管理器来使用，

contextlib.contextmanager 装饰器@contextmanager

两个代码等价，通过 yield 将函数分割成两部分，yield 之前的语句在 enter 方法中执行，yield 之后的语句在 exit 方法中执行。紧跟在 yield 后面的值是函数的返回值。

**强类型、弱类型、动态语言、静态语言**

**动态语言：**运行期间才做数据类型检查的语言，即动态类型语言编程时，永远不用给任何变量指定数据类型。该语言会在第一次赋值给变量时，在内部将数据类型记录下来。E.g. python javascript。

优点：方便阅读，不需要写非常多的类型相关的代码；

缺点：不方便调试，命名不规范时会造成读不懂，不利于理解等

**静态语言**：编译期间做检查数据类型的语言，即写程序时要声明所有变量的数据类型，是固定的。使用数据之前，必须先声明数据类型（int ,float,double等）。相当于使用之前，首先要为它们分配好内存空间。E.g. c c++ c# java

**强类型定义语言**：一旦变量被指定某个数据类型，如果不经强制转换，即永远是此数据类型。

**弱类型定义语言：** 一个变量可以赋不同数据类型的值。

Python属于强类型、动态语言。

例如，python不能合并类型‘x’+3。而在javascript中‘x’+3会变成‘x3’。

因为强类型化意味着值的类型不会以意外的方式改变。只包含数字的字符串不会神奇的变成数字，每种类型的更改需要显示的转换。Python能够有如下操作：

a = 1

a = ‘test’

是因为python是动态语言，动态类型化意味着运行到赋值语言时对象才具有类型。这与C语言的旧版本形成对比，c是弱静态类型，指针和证书几乎可以互换。

静态类型语言在如下操作下会出错：

int i = 1;

i = “1”;#会报错

**鸭子类型**

鸭子类型是编程语言中动态类型语言中的一种设计风格，一个对象的特征不是由父类决定(静态语言中是这样)，而是通过对象的方法决定的。e.g. 任何实现了 \_\_iter\_\_ 和 \_\_next\_\_方法的对象都可称之为迭代器，但对象本身是什么类型不受限制，可以自定义为任何类。

我们并不需要继承 Iterator 就可以实现迭代器的功能。E.g.有一函数希望接收的参数是 Iterator 类型时，但是我们传递的是 Foo 的实例对象，其实也没问题，换成是Java等静态语言，就必须传递 Iterator或者是它的子类。

鸭子类型通常得益于"不"测试方法和函数中参数的类型，而是依赖文档、清晰的代码和测试来确保正确使用，这既是优点也是缺点，缺点是需要通过文档才能知道参数类型，为了弥补这方面的不足，Python3.6引入了类型信息，定义变量的时候可以指定类型:

def greeting(name: str) -> str:

return 'Hello ' + name

**元类**

当我们使用关键字class时，python解释器在执行时就会自动的创建一个对象。

我们也可以手动处理，使用内建函数type。type两种使用方法：

1）type(‘1’) 🡪 返回参数的类的描述

2）type(类名，父类的元组(可以为空)，包含属性的字典 名称:值)

class Foo(object)

pass

Foo = type(‘Foo’,(),{}) 🡪返回一个**类对象**

Foo = Foo()🡪返回一个**类的实例**

type就是一个元类，就是python在背后用来创建所有类的元类：

a = 1

a.\_\_class\_\_ 🡪 <type ‘int’>

a.\_\_class\_\_.\_\_class\_\_ 🡪<type ‘type’>

**如何自己创建元类呢？**

在类中定义\_\_metaclass\_\_ = …

python做了如下操作：

Foo中有\_\_metaclass\_\_这个属性吗？如果是，Python会在内存中通过\_\_metaclass\_\_创建一个名字为Foo的类对象（我说的是类对象，请紧跟我的思路）。如果Python没有找到\_\_metaclass\_\_，它会继续在父类中寻找\_\_metaclass\_\_属性，并尝试做和前面同样的操作。如果Python在任何父类中都找不到\_\_metaclass\_\_，它就会在模块层次中去寻找\_\_metaclass\_\_，并尝试做同样的操作。如果还是找不到\_\_metaclass\_\_,Python就会用内置的type来创建这个类对象。

**python多继承的顺序是什么**

python3都是新式类，python2中没有显示继承object，就不是新式类，旧类并不是默认继承object类，而是继承type类。。

class D(object) class C1(D) class C2(D) class B(C1,C2)

class A(B,C1,C2)的继承顺序是什么？

新式类：广度优先搜索，A🡪B🡪C1🡪C2🡪D🡪object

经典类：从左到右，深度优先搜索，A🡪B🡪C1🡪D🡪object🡪C2