传智播客™
www.itcast.cn

上海 Python 面试手册

2019 年 1 月更新

目录

一、Python 基础

1、输入与输出

1.1、a=1,b=2,不用中间变量交换 a 和 b 的值？(2018-3-29-lxy)

方法一：

```
a = a+b
b = a-b
a = a-b
```

方法二：

```
a = a^b
b = b^a
a = a^b
```

方法三：

```
a,b = b,a
```

1.2、下面这段代码的输出结果将是什么？请解释？(2018-3-30-lxy)

```
class Parent(object):
    x = 1
class Child1(Parent):
    pass
class Child2(Parent):
    pass
Print(Parent.x, Child1.x, Child2.x)
Child1.x = 2
Print(parent.x, Child1.x, Child2.x)
parent.x = 3
Print(Parent.x, Child1.x, Child2.x)
```

结果为：

```
1 1 1 #继承自父类的类属性 x，所以都一样，指向同一块内存地址。
1 2 1 #更改 Child1，Child1 的 x 指向了新的内存地址。
3 2 3 #更改 Parent，Parent 的 x 指向了新的内存地址。
```

2、条件与循环

2.1、阅读下面的代码，写出 A0，A1 至 An 的最终值。(2018-3-30-lxy)

```
A0 = dict(zip(('a', 'b', 'c', 'd', 'e'), (1, 2, 3, 4, 5)))
A1 = range(10)
A2 = [i for i in A1 if i in A0]
A3 = [A0[s] for s in A0]
A4 = [i for i in A1 if i in A3]
A5 = {i:i*i for i in A1}
A6 = [[i, i*i] for i in A1]
```

答：

```
A0 = {'a': 1, 'c': 3, 'b': 2, 'e': 5, 'd': 4}
A1 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
A2 = []
A3 = [1, 3, 2, 5, 4]
```

```
    A4 = [1, 2, 3, 4, 5]
    A5 = {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
    A6 = [[0, 0], [1, 1], [2, 4], [3, 9], [4, 16], [5, 25], [6, 36], [7, 49], [8, 64]
[9, 81]]
```

2.2、考虑以下 Python 代码，命令行中的运行结果是什么？(2018-3-30-lxy)

```
    l = []
    for i in xrange(10):
        l.append({'num':i})
    Print(l)
```

考虑以下代码，运行结束后的结果是什么？

```
    l = []
    a = {'num':0}
    for i in xrange(10):
        a['num'] = i
        l.append(a)
    Print(l)
```

以上两段代码的运行结果是否相同，如果不相同，原因是什么？

上方代码的结果：

```
    [{'num':0}, {'num':1}, {'num':2}, {'num':3}, {'num':4}, {'num':5}, {'num':6}, {'num':7}, {'num':8}, {'num':9}]
```

下方代码结果：

```
    [{'num':9}, {'num':9}, {'num':9}, {'num':9}, {'num':9}, {'num':9}, {'num':9}, {'num':9}, {'num':9}, {'num':9}]
```

原因是：字典是可变对象，在下方的 `l.append(a)`的操作中是把字典 `a` 的引用传到列表 `l` 中，当后续操作修改 `a['num']`的值的时候，`l` 中的值也会跟着改变，相当于浅拷贝。

2.3、以下 Python 程序的输出？(2018-3-30-lxy)

```
    for i in range(5, 0, -1):
        print(i)
```

```
    答: 5 4 3 2 1
```

3、文件操作

3.1、有一个 jsonline 格式的文件 file.txt 大小约为 10K (2018-3-30-lxy)

```
    def get_lines():
        l = []
        with open('file.txt', 'rb') as f:
            for eachline in f:
                l.append(eachline)
            return l
    if __name__ == '__main__':
        for e in get_lines():
            process(e) #处理每一行数据
```

现在要处理一个大小为 `10G` 的文件，但是内存只有 `4G`，如果在只修改 `get_lines` 函数而其他代码保持不变的情况下，应该如何实现？需要考虑的问题都有哪些？

```
    def get_lines():
        l = []
        with open('file.txt', 'rb') as f:
            data = f.readlines(60000)
```

```
        l.append(data)
        yield l
```

要考虑到的问题有：
内存只有 4G 无法一次性读入 10G 的文件，需要分批读入。分批读入数据要记录每次读入数据的位置。分批每次读入数据的大小，太小就会在读取操作上花费过多时间。

3.2、补充缺失的代码？(2018-4-16-lxy)

```
def print_directory_contents(sPath):
    """
    这个函数接收文件夹的名称作为输入参数
    返回该文件夹中文件的路径
    以及其包含文件夹中文件的路径
    """
    # 补充代码
    ------------代码如下--------------------
    import os
    for sChild in os.listdir(sPath):
      sChildPath = os.path.join(sPath, sChild)
      if os.path.isdir(sChildPath):
         print_directory_contents(sChildPath)
      else:
         print(sChildPath)
```

4、模块与包

4.1 输入日期，判断这一天是这一年的第几天？(2018-3-30-lxy)

```
import datetime
def dayofyear():
    year = input("请输入年份：")
    month = input("请输入月份：")
    day = input("请输入天：")
    date1 = datetime.date(year=int(year)，month=int(month)，day=int(day))
    date2 = datetime.date(year=int(year)，month=1，day=1)
    return (date1 - date2 + 1).days
```

4.2 打乱一个排好序的 list 对象 alist？(2018-3-30-lxy)

```
import random
alist = [1,2,3,4,5]
random.shuffle(alist)
Print(alist)
```

5、数据类型

5.1、现有字典 d={‘a’:24，‘g’:52，‘i’:12，‘k’:33}请按 value 值进行排序？ (2018-3-30-lxy)

```
sorted(d.items()，key = lambda x:x[1]) 。
```

5.2、字典推导式？ （2018-4-23-lxy)

```
d = {key: value for (key, value) in iterable}
```

5.3、请反转字符串“aStr”?(2018-3-30-lxy)

```
print('aStr'[::-1])
```

5.4、将字符串"k:1|k1:2|k2:3|k3:4"，处理成字典：{k:1， k1:2， ... } (2018-3-30-lxy)

```
str1 = "k:1|k1:2|k2:3|k3:4"
def str2dict(str1):
    dict1 = {}
    for iterms in str1.split('|'):
        key，value = iterms.split(':')
        dict1[key] = value
    return dict1
```

5.5、请按 alist 中元素的 age 由大到小排序(2018-3-30-lxy)

```
alist [{'name':'a'，'age':20}，{'name':'b'，'age':30}，{'name':'c'，'age':25}]
def sort_by_age(list1):
    return sorted(alist，key=lambda x:x['age']，reverse=True)
```

5.6 下面代码的输出结果将是什么？(2018-3-30-lxy)

```
list = ['a'， 'b'， 'c'， 'd'， 'e']
    print(list[10:])
```

代码将输出[]，不会产生 IndexError 错误。就像所期望的那样，尝试用超出成员的个数的 index 来获取某个列表的成员。
例如，尝试获取 list[10]和之后的成员，会导致 IndexError。
然而，尝试获取列表的切片，开始的 index 超过了成员个数不会产生 IndexError，而是仅仅返回一个空列表。这成为特别让人恶心的疑难杂症，因为运行的时候没有错误产生，导致 bug 很难被追踪到。

5.7、写一个列表生成式，产生一个公差为 11 的等差数列(2018-3-30-lxy)

```
 print([x*11 for x in range(10)])
```

5.8、给定两个列表，怎么找出他们相同的元素和不同的元素? (2018-3-30-lxy)

```
    list1 = [1，2，3]
    list2 = [3，4，5]
    set1 = set(list1)
    set2 = set(list2)
    print(set1&set2)
    print(set1^set2)
```

5.9、请写出一段 Python 代码实现删除一个 list 里面的重复元素?(2018-3-30-lxy)

比较容易记忆的是用内置的 set：

```
  l1 = ['b'，'c'，'d'，'b'，'c'，'a'，'a']
  l2 = list(set(l1))
  print l2
```

如果想要保持他们原来的排序：

用 list 类的 sort 方法：

```
l1 = ['b', 'c', 'd', 'b', 'c', 'a', 'a']
l2 = list(set(l1))
l2.sort(key=l1.index)
print l2
```

也可以这样写：

```
l1 = ['b', 'c', 'd', 'b', 'c', 'a', 'a']
l2 = sorted(set(l1), key=l1.index)
print l2
```

也可以用遍历：

```
l1 = ['b', 'c', 'd', 'b', 'c', 'a', 'a']
l2 = []
for i in l1:
   if not i in l2:
      l2.append(i)
print l2
```

5.10、给定两个 list A ,B，请用找出 A ,B 中相同与不同的元素(2018-3-30-lxy)

```
A、B 中相同元素：print(set(A)&set(B))
A、B 中不同元素：print(set(A)^set(B))
```

5.11、有如下数组 list = range(10)我想取以下几个数组，应该如何切片？(2018-3-30-lxy)

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6]
[3, 4, 5, 6]
[9]
[1, 3, 5, 7, 9]
```

答：

```
[1:]
[1:7]
[3:7]
[-1]
[1::2]
```

5.12、下面这段代码的输出结果是什么？请解释？(2018-4-16-lxy)

```
def extendlist(val, list=[]):
   list.append(val)
   return list

list1 = extendlist(10)
list2 = extendlist(123, [])
list3 = extendlist('a')

print("list1 = %s" %list1)
print("list2 = %s" %list2)
print("list3 = %s" %list3)
```

输出结果：

```
list1 = [10, 'a']
list2 = [123]
list3 = [10, 'a']
```

新的默认列表只在函数被定义的那一刻创建一次。当 extendList 被没有指定特定参数 list 调用时，这组 list 的值随后将被使用。这是因为带有默认参数的表达式在函数被定义的时候被计算，不是在调用的时候被计算。

5.13、将以下 3 个函数按照执行效率高低排序(2018-4-16-lxy)

```
def f1(lIn):
    l1 = sorted(lIn)
    l2 = [i for i in l1 if i<0.5]
    return [i*i for i in l2]

def f2(lIn):
    l1 = [i for i in l1 if i<0.5]
    l2 = sorted(l1)
    return [i*i for i in l2]

def f3(lIn):
    l1 = [i*i for i in lIn]
    l2 = sorted(l1)
    return [i for i in l1 if i<(0.5*0.5)]
```

按执行效率从高到低排列：f2、f1 和 f3。要证明这个答案是正确的，你应该知道如何分析自己代码的性能。Python 中有一个很好的程序分析包，可以满足这个需求。

```
import random
import cProfile
lIn = [random.random() for i in range(100000)]
cProfile.run('f1(lIn)')
cProfile.run('f2(lIn)')
cProfile.run('f3(lIn)')
3.9 获取 1~100 被 6 整除的偶数？(2018-4-23-lxy)
def A():
    alist = []
    for i in range(1, 100):
        if i % 6 == 0:
            alist.append(i)
    last_num = alist[-3:]
   print(last_num)
```

5.14、如何用 Python 实现删除一个 list 里面的重复元素？

答：转换为集合，再转换回列表

6、企业面试题

6.1、Python 新式类和经典类的区别？（2019-01-19-cz）

a.在 Python 里凡是继承了 object 的类，都是新式类
b.Python3 里只有新式类
c.Python2 里继承 object 的时新式类，没有写父类的是经典类
d.经典类目前在 Python 里基本已经没有应用了，所以没有去学习

6.2、python 中内置的数据结构有几种？(2019-01-19-cz)

a. 整型 int、长整型 long、浮点型 float、复数 complex
b. 字符串 str、列表 list、元组 tuple
c. 字典 dict、集合 set

6.3、Python 如何实现单例模式？请写出两种实现方(2019-01-19-cz)

第一种方法：使用装饰器

装饰器卫华一个字典对象 instances,缓存了所有单例类，只要单例不存在就创建，已经存在直接返回该实例对象

```
def singleton(cls):
    instances = {}

    def wrapper(*args, **kwargs):
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)
        return instances[cls]

    return wrapper

@singleton
class Foo(object):
    pass

foo1 = Foo()
foo2 = Foo()

print foo1 is foo2  # True
```

第二种方法：使用基类

New 是真正创建实例对象的方法，所以重写基类的 new 方法，以此来保证创建对象的时候只生成一个实例

```
class Singleton(object):
    def __new__(cls, *args, **kwargs):
        if not hasattr(cls, '_instance'):
            cls._instance = super(Singleton, cls).__new__(cls, *args, **kwargs)
        return cls._instance

class Foo(Singleton):
    pass

foo1 = Foo()
foo2 = Foo()

print foo1 is foo2  # True
```

第三种方法：元类

元类（参考：深刻理解 Python 中的元类）是用于创建类对象的类，类对象创建实例对象时一定要调用 call 方法，因此在调用 call 时候保证始终只创建一个实例即可，type 是 python 中的元类

```
class Singleton(type):
    def __call__(cls, *args, **kwargs):
        if not hasattr(cls, '_instance'):
            cls._instance = super(Singleton, cls).__call__(*args, **kwargs)
        return cls._instance
```

```
class Foo(object):
    __metaclass__ = Singleton

foo1 = Foo()
foo2 = Foo()

print foo1 is foo2  # True
```

6.4、反转一个整数，例如 123-->321,-123-->-321,使用 Python 语言实现(2019-01-19-cz)

```
class Solution(object):
    def reverse(self, x):
```

```
        if -10 < x < 10:
            return x
        str_x = str(x)
        if str_x[0] != "-":
            str_x = str_x[::-1]
            x = int(str_x)
        else:
            str_x = str_x[1:][::-1]
            x = int(str_x)
            x = -x
        return x if -2147483648 < x < 2147483647 else 0

if __name__ == '__main__':
    s = Solution()
    reverse_int = s.reverse(-120)
    print(reverse_int)
```

6.5、设计实现遍历目录与子目录，抓取.pyc 文件(2019-01-19-cz)

第一种方法：

```
import os

def getFiles(dir, suffix):

    res = []
    for root, dirs, files in os.walk(dir):
        for filename in files:
            name, suf = os.path.splitext(filename)
            if suf == suffix:
                res.append(os.path.join(root, filename))

    print(res)

getFiles("./", '.pyc')
```

第二种方法：

```
import os

def pick(obj):
 try:
  if obj.[-4:] == ".pyc":
   print(obj)
 except:
  return None

def scan_path(ph):
    file_list = os.listdir(ph)
    for obj in file_list:
        if os.path.isfile(obj):
   pick(obj)
        elif os.path.isdir(obj):
            scan_path(obj)

if __name__ == '__main__':
    path = input('输入目录:')
    scan_path(path)
```

6.6、一行代码实现 1-100 之和。 注意：是左闭右开区间，所以要 101(2019-01-19-cz)

```
count = sum(range(0,101))
print(count)
```

6.7、Python-遍历列表时删除元素的正确做法(2019-01-19-cz)

问题描述
这是在工作中遇到的一段代码，原理大概和下面类似(判断某一个元素是否符合要求，不符合删除该元素，最后得到符合要求的列表)：

```
a = [1,2,3,4,5,6,7,8]
for i in a:
   if i>5:
      pass
   else:
      a.remove(i)
   print(a)
```

运行结果：

```
[2, 3, 4, 5, 6, 7, 8]
[2, 4, 5, 6, 7, 8]
[2, 4, 6, 7, 8]
[2, 4, 6, 7, 8]
[2, 4, 6, 7, 8]
```

问题分析
因为删除元素后，整个列表的元素会往前移动，而 i 却是在最初就已经确定了，是不断增大的，所以并不能得到想要的结果

解决方法
遍历在新的列表操作，删除时在原来的列表操作

```
a = [1,2,3,4,5,6,7,8]
print(id(a))
print(id(a[:]))
for i in a[:]:
   if i>5:
      pass
   else:
      a.remove(i)
   print(a)
print('-------------------------')
print(id(a))
```

运行结果

```
4545442312
4545447560
[2, 3, 4, 5, 6, 7, 8]
[3, 4, 5, 6, 7, 8]
[4, 5, 6, 7, 8]
[5, 6, 7, 8]
[6, 7, 8]
[6, 7, 8]
[6, 7, 8]
[6, 7, 8]
-------------------------
4545442312
```

```
    filter
    a = [1,2,3,4,5,6,7,8]
    b = filter(lambda x: x>5,a)
    print(list(b))
```

运行结果：

```
    [6, 7, 8]
```

列表解析

```
    a = [1,2,3,4,5,6,7,8]
    b = [i for i in a if i >5]
    print(b)
```

运行结果：

```
    [6, 7, 8]
```

倒序删除

因为列表总是“向前移”，所以可以倒序遍历，即使后面的元素被修改了，还没有被遍历的元素和其坐标还是保持不变的。

```
    a = [1,2,3,4,5,6,7,8]
    print(id(a))
    for i in range(len(a)-1,-1,-1):
       if a[i] > 5:
          pass
       else:
          a.remove(a[i])
    print(id(a))
    print('--------------------')
    print(a)
```

运行结果

```
    4545400712
    4545400712
    --------------------
    [6, 7, 8]
```

6.8、字符串的操作题目(2019-01-19-cz)

全字母短句是包含所有英文字母的句子，比如：A QUICK BROWN FOX JUMPS OVER THE LAZY DOG. 定义并实现一个方法 GETMISSINGLETTERS,传入一个字符串采纳数，返回参数字符串变成一个 PANGRAM 中所缺失的字符。应该忽略传入字符串参数中的大小写，返回应该都是小写字符并按字母顺序排序，（请忽略所有非 ASCII 字符）
下面示例是用来解释，双引号不需要考虑:
0）输入：A quick brown fox jumps over the lazy dog
返回：""

1)输入："A slow yellow fox crawls under the proactive dog"
返回:"bjkmqz"

2)输入:"Lions,and tigers,and bears,oh my!"
返回:"cfjkpquvwxz"

3)输入：""
返回:"abcdefghijklmnopqrstuvwxyz"

```
def getMissingLetter(a):
    s1 = set("abcdefghijklmnopqrstuvwxyz")
    ret = ""
    s2 = set(a)
    for i in sorted(s1-s2):
        ret += i
    return ret

print(getMissingLetter("python"))
```

6.9、可变类型和不可变类型

1、可变类型有 list，dict。不可变类型有 string，number，tuple。
2、当进行修改操作时，可变类型传递的是内存中的地址，也就是说，直接修改内存中的值，并没有开辟新的内存。
3、不可变类型被改变时，并没有改变原内存地址中的值，而是开辟一块新的内存，将原地址中的值复制过去，对这块新开辟的内存中的值进行操作。

6.10、is 和==有什么区别?

变量、内存理解

变量：用来标识（identify）一块内存区域。为了方便表示内存，我们操作变量实质上是在操作变量指向的内存单元。编译器负责分配，我们可以使用 Python 内建函数 id()来获取变量的地址；

内存：内存是我们电脑硬件，用来存放数据，形象的理解就是内存有一个一个的小格子组成，每个格子的大小是一个字节，每个格子可以存放一个字节大小的数据，我们如何才能知道，数据存放在那些格子中，需要靠内存地址，内存地址类似楼房的门牌号码，是内存的标识符；

id()
id(object)函数是返回对象 object 在其生命周期内位于内存中的地址，id 函数的参数类型是一个对象

is 和==是什么?
在 Python 当中一切都是对象，毫无例外整数也是对象，对象之间是比较是否相等可以用==，也可以用 is.
is：比较的是两个对象的 id 值是否相等，也就是比较俩对象是否为同一个实例对象，是否指向同一个内存地址
==：比较的是两个对象的内容/值是否相等，默认会调用对象的 eq()方法；
数据池
为了引入数据池概念，我们看下面一段代码

```
>>> a = 1
>>> b = 1
>>> a is b
True
>>> a == b
True
>>> c = 257
>>> d = 257
>>> c == d
True
>>> c is d
False   # ? why
```

对于 257，c is d 返回的竟然是 False ，大家可能有点疑惑了，下面开始解惑

解惑一：
出于对性能的考虑，Python 内部做了很多的优化工作，对于整数对象，Python 把一些频繁使用的整数对象缓存起来，保存到一个叫 small_ints 的链表中，在 Python 的整个生命周期内，任何需要引用这些整数对象的地方，都不再重新创建新的对象，而是直接引用缓存中的对象。Python 把这些可能频繁使用的整数对象规定在范围【-5，256】之间的小对象放在 small_ints 中，但凡是需要用些小整数时，就从这里面取，不再去临时创建新的对象。因为 257 不再小整数范围内，因此尽管 c 和 d 的值都是一样，但是在 Python 内部他们的内存地址不一样。
我们继续看代码

```
>>> a = 257
>>> def foo():
...     b = 257
...     c = 257
...     print(a is b)
...     print(b is c)
...
>>> foo()
False
```

True # why?不是说 257 不在小整数范围，内存地址不一样？为什么 b is c 是 True
很多人会疑惑？为什么是 True
解惑二
为了弄清楚这个问题，我们有必要弄明白代码块的概念。Python 程序由代码块构成，代码块作为程序的一个最小基本单位来执行。一个模块文件，一个函数体，一个类，交互命令行中的单行代码都叫做一个代码块。在上文的代码中 a 和 b 不处于一个代码块，而 b 和 c 所属一个函数属于一个代码块。Python 内部为了将性能进一步提高，但凡是不可变对象，在同一个代码块，只要不可变对象的值相同，就不会重复创建，而是直接引用。因此不仅是整数数字，字符串、元祖也同样遵循这个原则。

一番长篇大论以后，得出两个结论：
1、小整数对象【-5，256】是全局解释器范围内被重复使用，永远不会被 GC 回收
2、同一个代码块中的不可变对象，只要值是相等的就不会重复创建新的对象

总结：
1、在同一代码块内，a == b 如果是 True，a is b 肯定是 True
2、不在同一代码块内，a == b 如果是 True，a is b 可能是 True 也可能是 False ；如果是 a 和 b 属于-5,256 小整数范围就是 True，如果不是就是 False

6.11、filter 方法求出列表所有奇数并构造新列表，a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

第一种：

```
a = {"name1":"gj","age1":24}
b = {"name2":"hr","age2":22}
c = {}
c.update(a)
c.update(b)
print(c)
```

运行结果：

```
{'name1': 'gj', 'age1': 24, 'name2': 'hr', 'age2': 22}
```

第二种：

```
a = {"name1":"gj","age1":24}
b = {"name2":"hr","age2":22}
c = dict()

for key,value in a.items():
    c[key] = value
```

```
    for key,value in b.items():
        c[key] = value
    print(c)
```

```
{'name1': 'gj', 'age1': 24, 'name2': 'hr', 'age2': 22}
```

python 位运算计算
运行下列代码输出打印的结果：

```
    # x = 9
    # y = 12
    # print(x &amp;amp; y)
    # print(x | y)
    # print(x ^ y)
    # print(~x)
```

考察技能点：
本题考察的是对 python 中位运算的运用，与（&），或（|），异或（^），取反（~），下图是具体讲解：
计算过程及结果：
与（&），按位与运算符：参与运算的两个值,如果两个相应位都为 1,则该位的结果为 1,否则为 0
或（|），按位或运算符：只要对应的二个二进位有一个为 1 时，结果位就为 1
异或（^），按位异或运算符：当两对应的二进位相异时，结果为 1
取反（~），按位取反运算符：对数据的每个二进制位取反,即把 1 变为 0,把 0 变为 1

计算过程及结果：

```
    x = 9  #二进制表达为 1001
    y = 12 #二进制表达为 1100
    print(x &amp;amp; y)  #二进制计算结果为 1000，即 8
    print(x | y)  #二进制计算结果为 1101，即 13
    print(x ^ y)  #二进制计算结果为 0101，即 5
    print(~x)  #二进制计算取反结果为-10
```

6.12、用一行 python 代码写出 1+2+3+10248

两种方法如下：

```
    # 用一行 python 代码写出 1+2+3+10348
    from functools import reduce
    # 1.使用 sum 内置求和函数
    num = sum([1,2,3,10348])
    print(num)
```

```
    # 2.reduce 函数是对一个序列的每个项迭代调用函数,下面是求和
    num1 = reduce(lambda x,y:x+y,[1,2,3,10348])
    print(num1)
```

6.13、Python 中变量的作用域？（变量查找顺序）

函数作用域的 LEGB 顺序
1.什么是 LEGB?
L:local 函数内部作用域
E:enclosing 函数内部与内嵌函数之间
G:global 全局作用域

B:build-in 内置作用域

2.它们是作什么用的
为什么非要介绍这个呢?或者说它们的作用是什么?
原因是因为我们的在学习 Python 函数的时候，经常会遇到很多定义域的问题，全部变量，内部变量，内部嵌入的函数，等等，Python 是如何查找的呢？以及 Python 又是按照什么顺序来查找的呢？这里做一个顺序的说明

3.顺序是什么
跟名字一样，Python 在函数里面的查找分为 4 种，称之为 LEGB，也正是按照这种顺序来查找的。
首先，是 local,先查找函数内部
然后，是 enclosing，再查找函数内部与嵌入函数之间（是指在函数内部再次定义一个函数）
其次，是 global，查找全局
最后，是 build-in，内置作用域

6.14、字符串”123″ 转换成 123，不使用内置 api，例如 int（）

方法一：利用 str 函数

既然不能用 int 函数，那我们就反其道而行，用 str 函数找出每一位字符表示的数字大写。

```
def atoi(s):
    s = s[::-1]
    num = 0
    for i, v in enumerate(s):
        for j in range(0, 10):
            if v == str(j):
                num += j * (10 ** i)
    return num
```

方法二：利用 ord 函数

利用 ord 求出每一位字符的 ASCII 码再减去字符 0 的 ASCII 码求出每位表示的数字大写。

```
def atoi(s):
    s = s[::-1]
    num = 0
    for i, v in enumerate(s):
        offset = ord(v) - ord('0')
        num += offset * (10 ** i)

    return num
```

方法三：利用 eval 函数

eval 的功能是将字符串 str 当成有效的表达式来求值并返回计算结果。我们利用这特点可以利用每位字符构造成和 1 相乘的表达式，再用 eval 算出该表达式的返回值就表示数字大写。

```
def atoi(s):
    s = s[::-1]
    num = 0
    for i, v in enumerate(s):
        t = '%s * 1' % v
        n = eval(t)
        num += n * (10 ** i)
```

```
        return num
```

6.15、Given an array of integers

return indices of the two numbers such that they add up to a specific target. You may assume that each input would have exactly one solution, and you may not use the same element twice. Example: Given nums = [2, 7, 11, 15], target = 9, Because nums[0] + nums[1] = 2 + 7 = 9, return [0, 1].
给定一个整数数组和一个目标值，找出数组中和为目标值的两个数。你可以假设每个输入只对应一种答案，且同样的元素不能被重复利用。 示例: 给定 nums = [2, 7, 11, 15], target = 9 因为 nums[0] + nums[1] = 2 + 7 = 9 所以返回 [0, 1]

```
    class Solution:
        def twoSum(self, nums, target):
            """
            :type nums: List[int]
            :type target: int
            :rtype: List[int]
            """
            d={}
            size=0
            while size < len(nums):
                if target-nums[size] in d:
                    if d[target-nums[size]] < size:
                        return [d[target-nums[size]],size]
                else:
                    d[nums[size]] = size
                size = size + 1

    solution = Solution()
    list = [2, 7, 11, 15]
    target = 9
    nums = solution.twoSum(list,target)
    print(nums)
```

#输出

```
    [0,1]
```

给列表中的字典排序：假设有如下 list 对象，alist=[{“name”:”a”, “age”:20},{“name”:”b”,”age”:30}, {“name”:”c”,”age”:25}]，将 alist 中的元素按照 age 从大到小排序
alist = [{"name":"a","age":20},{"name":"b","age":30},{"name":"c","age":25}]

```
    # 单级排序，按照 age 从大到小排序

    alist_sort = sorted(alist, key=lambda e: e.__getitem__('age'),reverse=True)

    print("sort by 1 key: ", alist_sort)

    # 多级排序，先按照 age，再按照 name 排序

    # alist_sort1 = sorted(alist, key=lambda e:(e.__getitem__('age'), e.__getitem__('name')))

    # print( "sort by 2 keys: ", alist_sort1)
```

6.16、python 代码实现删除一个 list 里面的重复元素

```
def distFunc1(a):
    '''使用集合去重'''
    a=list(set(a))
    print(a)

def distFunc2(a):
    '''将一个列表的数据取出放到另一个列表中，中间作判断'''
    list = []
    for i in a:
        if i not in list:
            list.append(i)
    # 如果需要排序的话用 sort
    list.sort()
    print(list)

def distFunc3(a):
    '''使用字典'''
    b = {}
    b = b.fromkeys(a)
    c = list(b.keys())
    print(c)

if __name__ == '__main__':
    a = [1,2,4,2,4,5,7,10,5,5,7,8,9,0,3]
    distFunc1(a)
    distFunc2(a)
    distFunc3(a)
```

6.17、统计一个文本中单词频次最高的 10 个单词?

```
import re

def test(filepath):

distone={}
numTen=[]

with open(filepath, 'r', encoding='utf-8') as f:
    for line in f:
        line = re.sub('\W'," ",line)
        lineone=line.split()
        for keyone in lineone:
            if not distone.get(keyone):
                distone[keyone]=1
            else:
                distone[keyone]+=1

numTen = sorted(distone.items(), key= lambda x: x[1], reverse=True)[: 10]
numTen = [x[0] for x in numTen]

return numTen
filepath = r"C:\Users\Administrator\Desktop\qwe.txt"

res = test(filepath) print(res)
```

6.18、请写出一个函数满足以下条件

该函数的输入是一个仅包含数字的 list,输出一个新的 list，其中每一个元素(element)要满足以下条件：
1、该元素是偶数
2、该元素在原 list 中是在偶数的位置（index 是偶数）

```
def num_list(num):

    new_list = []
    for i in num:
        if i % 2 == 0 and num.index(i) % 2 == 0:
            new_list.append(i)

    return new_list
# 输入列表数据
num = [0, 1, 2, 3, 5, 4, 6, 7, 8, 9, 10]

result = num_list(num)

print(result)
```

6.19、使用单一的列表生成式来产生一个新的列表，该列表只包含满足以下条件的值：

偶数值
(b)元素为原始列表中偶数切片

```
list_data = [1, 2, 5, 8, 10, 3, 18, 6, 20]

res = [x for x in list_data[::2] if x % 2 == 0]

print(res)#[10,18,20]
```

遍历一个 object 的所有属性，并 print 每一个属性名
#dir()方法中放入相应的 object 即可例如：

```
for i in dir([]):
print(i)
```

6.20、用一行代码生成[1,4,9,16,25,36,49,64,81,100]

```
[x * x for x in range(1, 11)]
```

6.21、输入某年某月某日，判断这一天是这一年的第几天?

```
#方法一：
dat = input('请输入某年某月某日，格式为 yyyy-mm-dd ： ')
y = int(dat[0:4])  #获取年份
m = int(dat[5:7])  #获取月份
d = int(dat[8:])  #获取日

ly = False

if y%100 == 0:  #若年份能被 100 整除
    if y%400 == 0:  #且能被 400 整除
```

```
        ly = True  #则是闰年
    else:
        ly = False
elif y%4 == 0:  #其它情况下，若能被 4 整除
    ly = True  #则为闰年
else:
    ly = False

if ly == True:  #若为闰年，则 2 月份有 29 天
    ms = [31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
else:
    ms = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

days = 0
for i in range(1, 13):  #从 1 到 12 逐一判断，以确定月份
    if i == m:
        for j in range(i-1):  #确定月份 i 之后，则将 ms 列表中的前 i-1 项相加
            days += ms[j]
        print('%s 是该年份的第%s 天。' % (dat, (days + d))) #最后再加上“日”，即是答案
```

```
# 方法二：
#使用库
import datetime

y = int(input('请输入 4 位数字的年份：'))  #获取年份
m = int(input('请输入月份：'))  #获取月份
d = int(input('请输入是哪一天：'))  #获取“日”

targetDay = datetime.date(y, m, d)  #将输入的日期格式化成标准的日期
dayCount = targetDay - datetime.date(targetDay.year - 1, 12, 31)  #减去上一年最后一天
print('%s 是%s 年的第%s 天。'% (targetDay, y, dayCount.days))
```

6.22、两个有序列表，l1,l2，写个算法对这两个列表进行合并（不可使用 extend）

```
#方法一：
def loop_merge_sort(l1, l2):

    tmp = []
    while len(l1) > 0 and len(l2) > 0:
        if l1[0] < l2[0]:
            tmp.append(l1[0])
            del l1[0]
        else:
            tmp.append(l2[0])
            del l2[0]
     # 有一个为空表时，把非空表拼到 tmp 后面 。。。
```

```
#方法二：
 # 列表 1 加到列表 2 中再进行排序
```

6.23、给定一个任意长度数组，实现一个函数

让所有奇数都在偶数前面，而且奇数升序排列，偶数降序排列，如字符串’1982376455’ ，变成’135798642’

```
def func1(l):
    if isinstance(l, str):
        l = list(l)
        l = [int(i) for i in l]
    l.sort(reverse=True)

    for i in range(len(l)):

        if l[i] % 2 > 0:
            l.insert(0, l.pop(i))

    print(l)
```

6.23、写一个函数找出一个整数数组中，第二大的数

```
def find_Second_large_num(num_list):
  '''''
  找出数组中第 2 大的数字
  '''
  #直接排序,输出倒数第二个数即可
  tmp_list=sorted(num_list)
  print 'Second_large_num is:', tmp_list[-2]
  #设置两个标志位一个存储最大数一个存储次大数
  #two 存储次大值，one 存储最大值，遍历一次数组即可，先判断是否大于 one，若大于将 one 的
  #值给 two，将 num_list[i]的值给 one；否则比较是否大于 two，若大于直接将 num_list[i]的
  #值给 two；否则 pass
  one=num_list[0]
  two=num_list[0]
  for i in range(1,len(num_list)):
    if num_list[i]>one:
      two=one
      one=num_list[i]
    elif num_list[i]>two:
       two=num_list[i]
    else:
      pass
  print 'Second_large_num is:', two
if __name__ == '__main__':
  num_list=[34,11,23,56,78,0,9,12,3,7,5]
  find_Second_large_num(num_list)
```

6.24、阅读一下代码他们的输出结果是什么？

```
def multi():
    return [lambda x: i * x for i in range(4)]
print([m(3) for m in multi()])
```

正确答案是[9,9,9,9]，而不是[0,3,6,9]
产生的原因是 Python 的闭包的后期绑定导致的，这意味着在闭包中的变量是在内部函数被调用的时候被查找的
因此，最后函数被调用的时候，for 循环已经完成，i 的值最后是 3，因此每一个返回值的 i 都是 3，所以最后的结果是[9,9,9,9]
知识要点：熟练理解闭包和匿名函数

6.25、统计一段字符串中字符出现的次数

```
    def count_str(str_data):
       """定义一个字符出现次数的函数"""
        dict_str = {} # 定义空字典
        for i in str_data:
            dict_str[i] = dict_str.get(i, 0) + 1 # 对每个字符出现的次数做累加操作
        return dict_str

    dict_str = count_str("AAABBCCAC")

    str_cout_data = ""

    for k,v in dict_str.items():# 遍历字典取出 key 和 value 进行拼接

        str_cout_data += k+str(v)

    print(str_cout_data)
```

6.26、super 函数的具体用法和场景

super 本质上就是使用 MRO 这个顺序去调用 当前类在 MRO 顺序中下一个类。 super().init()则调用了下一个类的初始化方法进行构造。

print("******多继承使用 super().__init__ 发生的状态******")

```
    class Parent(object):
        def __init__(self, name, *args, **kwargs):  # 为避免多继承报错，使用不定长参数，接受参数
            print('parent 的 init 开始被调用')
            self.name = name
            print('parent 的 init 结束被调用')

    class Son1(Parent):
        def __init__(self, name, age, *args, **kwargs):  # 为避免多继承报错，使用不定长参数，接受参数
            print('Son1 的 init 开始被调用')
            self.age = age
            super().__init__(name, *args, **kwargs)  # 为避免多继承报错，使用不定长参数，接受参数
            print('Son1 的 init 结束被调用')

    class Son2(Parent):
        def __init__(self, name, gender, *args, **kwargs):  # 为避免多继承报错，使用不定长参数，接受参数
            print('Son2 的 init 开始被调用')
            self.gender = gender
            super().__init__(name, *args, **kwargs)  # 为避免多继承报错，使用不定长参数，接受参数
            print('Son2 的 init 结束被调用')

    class Grandson(Son1, Son2):
        def __init__(self, name, age, gender):
            print('Grandson 的 init 开始被调用')
            # 多继承时，相对于使用类名.__init__方法，要把每个父类全部写一遍
            # 而 super 只用一句话，执行了全部父类的方法，这也是为何多继承需要全部传参的一个原因
            # super(Grandson, self).__init__(name, age, gender)
            super().__init__(name, age, gender)
            print('Grandson 的 init 结束被调用')

    print(Grandson.__mro__)
```

```
gs = Grandson('grandson', 12, '男')
print('姓名：', gs.name)
print('年龄：', gs.age)
print('性别：', gs.gender)
print("******多继承使用 super().__init__ 发生的状态******\n\n")
```

总结： 1. MRO 保证了多继承情况 每个类只出现一次 2. super().init 相对于类名.init，在单继承上用法基本无差 3. 但在多继承上有区别，super 方法能保证每个父类的方法只会执行一次，而使用类名的方法会导致方法被执行多次 4. 多继承时，使用 super 方法，对父类的传参数，应该是由于 python 中 super 的算法导致的原因，必须把参数全部传递，否则会报错 5. 单继承时，使用 super 方法，则不能全部传递，只能传父类方法所需的参数，否则会报错 6. 多继承时，相对于使用类名.init 方法，要把每个父类全部写一遍，而使用 super 方法，只需写一句话便执行了全部父类的方法，这也是为何多继承需要全部传参的一个原因

二、Python 高级

1、元类

1.1、Python 中类方法、类实例方法、静态方法有何区别？(2018-3-30-lxy)

类方法：是类对象的方法，在定义时需要在上方使用“@classmethod”进行装饰，形参为 cls，表示类对象，类对象和实例对象都可调用；

类实例方法：是类实例化对象的方法，只有实例对象可以调用，形参为 self，指代对象本身；

静态方法：是一个任意函数，在其上方使用“@staticmethod”进行装饰，可以用对象直接调用，静态方法实际上跟该类没有太大关系。

1.2、super 函数的具体用法和场景？(2019-1-19-yzh)

super() 函数用于调用下一个父类(超类)并返回该父类实例的方法；是用来解决多重继承问题的，直接用类名调用父类方法在使用单继承的时候没问题，但是如果使用多继承，会涉及到查找顺序（MRO）、重复调用（钻石继承）等种种问题。MRO 就是类的方法解析顺序表，其实也就是继承父类方法时的顺序表。

以下是 super() 方法的语法: super(type[, object-or-type])

```
    class FooParent(object):
        def __init__(self):
        self.parent = 'I\'m the parent.'
        print ('Parent')

        def bar(self,message):
            print ("%s from Parent" % message)

    class FooChild(FooParent):
        def __init__(self):
        # super(FooChild,self) 首先找到 FooChild 的父类（就是类 FooParent），然后把类 B 的对象 FooChild 转换为类 FooParent 的对象
            super(FooChild,self).__init__()
            print ('Child')

        def bar(self,message):
            super(FooChild, self).bar(message)
            print ('Child bar fuction')
            print (self.parent)

    if __name__ == '__main__':
        fooChild = FooChild()
        fooChild.bar('HelloWorld')
```

1.3、历一个 object 的所有属性，并 print 每一个属性名？(2019-1-19-yzh)

```
    class Car(object):
       def __init__(self, name, loss):  # loss [价格，油耗，公里数]
          self.name = name
          self.loss = loss
```

```
    def getName(self):
        return self.name

    def getPrice(self):
        # 获取汽车价格
        return self.loss[0]

    def getLoss(self):
        # 获取汽车损耗值
        return self.loss[1]*self.loss[2]


Bmw = Car("宝马", [60, 9, 500])  # 实例化一个宝马车对象
print(getattr(Bmw, "name"))  # 使用 getattr() 传入对象名字，属性值。输出值：宝马
print(dir(Bmw))  # 获取 Bmw 所有的属性和方法
```

1.4、写一个类，并让它尽可能多的支持操作符？(2019-1-19-yzh)

```
class Array:
    __list = []

    def __init__(self):
        print "constructor"

    def __del__(self):
        print "destructor"

    def __str__(self):
        return "this self-defined array class"

    def __getitem__(self, key):
        return self.__list[key]

    def __len__(self):
        return len(self.__list)

    def Add(self, value):
        self.__list.append(value)

    def Remove(self, index):
        del self.__list[index]

    def DisplayItems(self):
        print "show all items----"
        for item in self.__list:
            print item

arr = Array()   #constructor
print arr    #this self-defined array class
print len(arr)   #0
arr.Add(1)
arr.Add(2)
arr.Add(3)
print len(arr)   #3
print arr[0]   #1
arr.DisplayItems()
```

```
    #show all items----
    #1
    #2
    #3
    arr.Remove(1)
    arr.DisplayItems()
    #show all items----
    #1
    #3
    #destructor
```

1.5、介绍 Cpython，Pypy Cpython Numba 各有什优缺点(2019-1-19-yzh)

CPython 是用 C 语言实现 Python，是目前应用最广泛的解释器。Python 最新的语言特性都是在这个上面先实现，Linux，OS X 等自带的也是这个版本，包括 Anaconda 里面用的也是 CPython。CPython 是官方版本加上对于 C/Python API 的全面支持，基本包含了所有第三方库支持，例如 Numpy，Scipy 等。但是 CPython 有几个缺陷，一是全局锁使 Python 在多线程效能上表现不佳，二是 CPython 无法支持 JIT（即时编译），导致其执行速度不及 Java 和 Javascipt 等语言。于是出现了 Pypy。

Pypy 是用 Python 自身实现的解释器。针对 CPython 的缺点进行了各方面的改良，性能得到很大的提升。最重要的一点就是 Pypy 集成了 JIT。但是，Pypy 无法支持官方的 C/Python API，导致无法使用例如 Numpy，Scipy 等重要的第三方库。这也是现在 Pypy 没有被广泛使用的原因吧。

Jython 是将 Python code 在 JVM 上面跑和调用 java code 的解释器。

1.6、请描述抽象类和接口类的区别和联系？(2019-1-19-yzh)

1）抽象类：规定了一系列的方法，并规定了必须由继承类实现的方法。由于有抽象方法的存在，所以抽象类不能实例化。可以将抽象类理解为毛坯房，门窗、墙面的样式由你自己来定，所以抽象类与作为基类的普通类的区别在于约束性更强。

2）接口类：与抽象类很相似，表现在接口中定义的方法，必须由引用类实现，但他与抽象类的根本区别在于用途：与不同个体间沟通的规则（方法），你要进宿舍需要有钥匙，这个钥匙就是你与宿舍的接口，你的同室也有这个接口，所以他也能进入宿舍，你用手机通话，那么手机就是你与他人交流的接口。

3）区别和关联：

1. 接口是抽象类的变体，接口中所有的方法都是抽象的。而抽象类中可以有非抽象方法。抽象类是声明方法的存在而不去实现它的类。

2. 接口可以继承，抽象类不行。

3.接口定义方法，没有实现的代码，而抽象类可以实现部分方法。

4.接口中基本数据类型为 static 而抽类象不是。

5.接口可以继承，抽象类不行。

6.可以在一个类中同时实现多个接口。

7.接口的使用方式通过 implements 关键字进行，抽象类则是通过继承 extends 关键字进行。

1.7、Python 中如何动态获取和设置对象的属性？(2018-3-30-lxy)

```
    if hasattr(Parent, 'x'):
        print(getattr(Parent, 'x'))
        setattr(Parent, 'x', 3)
    print(getattr(Parent, 'x'))
```

2、内存管理与垃圾回收机制

2.1、关于 Python 内存管理,下列说法错误的是(B)(2018-5-1-lxy)

A、变量不必事先声明　　B、变量无须先创建和赋值而直接使用
C、变量无须指定类型　　D、可以使用 del 释放资源

2.2、哪些操作会导致 Python 内存溢出，怎么处理？(2019-1-19-yzh)

内存泄漏:你使用 malloc 或 new 向内存申请了一块内存空间,但没有用 free 以及 delete 对该块内存进行释放,造成程序失去了对该块内存的控制.

内存溢出:你申请了 10 个字节的内存,但写入了大于 10 个字节的数据

内存泄漏指由于疏忽或错误造成程序未能释放已经不再使用的内存的情况。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制，因而造成了内存的浪费。导致程序运行速度减慢甚至系统崩溃等严重后果。有 del()函数的对象间的循环引用是导致内存泄漏的主凶。不使用一个对象时使用:del object 来删除一个对象的引用计数就可以有效防止内存泄漏问题。通过 Python 扩展模块 gc 来查看不能回收的对象的详细信息。可以通过 sys.getrefcount(obj) 来获取对象的引用计数，并根据返回值是否为 0 来判断是否内存泄漏

分类:

常发性内存泄漏：发生内存泄漏的代码会被多次执行到，每次被执行的时候都会导致一块内存泄漏。

偶发性内存泄漏：发生内存泄漏的代码只有在某些特定环境或操作过程下才会发生。常发性和偶发性是相对的。对于特定的环境，偶发性的也许就变成了常发性的。所以测试环境和测试方法对检测内存泄漏至关重要。

一次性内存泄漏：发生内存泄漏的代码只会被执行一次，或者由于算法上的缺陷，导致总会有一块仅且一块内存发生泄漏。比如，在类的构造函数中分配内存，在析构函数中却没有释放该内存，所以内存泄漏只会发生一次

隐式内存泄漏：程序在运行过程中不停的分配内存，但是直到结束的时候才释放内存。严格的说这里并没有发生内存泄漏，因为最终程序释放了所有申请的内存。但是对于一个服务器程序，需要运行几天，几周甚至几个月，不及时释放内存也可能导致最终耗尽系统的所有内存。所以，我们称这类内存泄漏为隐式内存泄漏。

内存泄漏解决办法：内存泄漏也许是因为活动已经被使用完毕，但是仍然在其他地方被引用，导致无法对其进行回收。我们只需要给对活动进行引用的类独立出来或者将其变为静态类，该类随着活动的结束而结束，也就没有了当活动结束但仍然还被其他类引用的情况。

资源性对象在不使用的时候，应该调用它的 close()函数将其关闭掉。。

集合容器中的内存泄露 ，我们通常把一些对象的引用加入到了集合容器（比如 ArrayList）中，当我们不需要该对象时，并没有把它的引用从集合中清理掉，这样这个集合就会越来越大。如果这个集合是 static 的话，那情况就更严重了。

需要在退出程序之前，将集合里的东西 clear，然后置为 null，再退出程序。

WebView 造成的泄露，当我们不使用 WebView 对象时，应该调用它的 destory()函数来销毁它，并释放其占用的内存，否则其长期占用的内存也不能被回收，从而造成内存泄露。

我们应该为 WebView 另外开启一个进程，通过 AIDL 与主线程进行通信，WebView 所在的进程可以根据业务的需要选择合适的时机进行销毁，从而达到内存的完整释放

内存溢出解决办法

内存溢出原因:

1.内存中加载的数据量过于庞大，如一次从数据库取出过多数据；

2.集合类中有对对象的引用，使用完后未清空，产生了堆积，使得 JVM 不能回收；

3.代码中存在死循环或循环产生过多重复的对象实体；

4.使用的第三方软件中的 BUG；

5.启动参数内存值设定的过小

内存溢出的解决方案：

第一步，修改 JVM 启动参数，直接增加内存。(-Xms，-Xmx 参数一定不要忘记加。)

第二步，检查错误日志，查看“OutOfMemory”错误前是否有其 它异常或错误。

第三步，对代码进行走查和分析，找出可能发生内存溢出的位置。

2.3、Python 的内存管理机制及调优手段？(2018-3-30-lxy)

内存管理机制：引用计数、垃圾回收、内存池。

引用计数：引用计数是一种非常高效的内存管理手段， 当一个 Python 对象被引用时其引用计数增加 1，当其不再被一个变量引用时则计数减 1. 当引用计数等于 0 时对象被删除。

垃圾回收 ：

1. 引用计数

引用计数也是一种垃圾收集机制，而且也是一种最直观，最简单的垃圾收集技术。当 Python 的某个对象的引用计数降为 0 时，说明没有任何引用指向该对象，该对象就成为要被回收的垃圾了。比如某个新建对象，它被分配给某个引用，对象的引用计数变为 1。如果引用被删除，对象的引用计数为 0，那么该对象就可以被垃圾回收。不过如果出现循环引用的话，引用计数机制就不再起有效的作用了

2. 标记清除

如果两个对象的引用计数都为 1，但是仅仅存在他们之间的循环引用，那么这两个对象都是需要被回收的，也就是说，它们的引用计数虽然表现为非 0，但实际上有效的引用计数为 0。所以先将循环引用摘掉，就会得出这两个对象的有效计数。

3. 分代回收

从前面“标记-清除”这样的垃圾收集机制来看，这种垃圾收集机制所带来的额外操作实际上与系统中总的内存块的数量是相关的，当需要回收的内存块越多时，垃圾检测带来的额外操作就越多，而垃圾回收带来的额外操作就越少；反之，当需回收的内存块越少时，垃圾检测就将比垃圾回收带来更少的额外操作。

举个例子：

当某些内存块 M 经过了 3 次垃圾收集的清洗之后还存活时，我们就将内存块 M 划到一个集合 A 中去，而新分配的内存都划分到集合 B 中去。当垃圾收集开始工作时，大多数情况都只对集合 B 进行垃圾回收，而对集合 A 进行垃圾回收要隔相当长一段时间后才进行，这就使得垃圾收集机制需要处理的内存少了，效率自然就提高了。在这个过程中，集合 B 中的某些内存块由于存活时间长而会被转移到集合 A 中，当然，集合 A 中实际上也存在一些垃圾，这些垃圾的回收会因为这种分代的机制而被延迟。

内存池：

Python 的内存机制呈现金字塔形状，-1，-2 层主要有操作系统进行操作；

第 0 层是 C 中的 malloc，free 等内存分配和释放函数进行操作；

第 1 层和第 2 层是内存池，有 Python 的接口函数 PyMem_Malloc 函数实现，当对象小于 256K 时有该层直接分配内存；

第 3 层是最上层，也就是我们对 Python 对象的直接操作；

Python 在运行期间会大量地执行 malloc 和 free 的操作，频繁地在用户态和核心态之间进行切换，这将严重影响 Python 的执行效率。为了加速 Python 的执行效率，Python 引入了一个内存池机制，用于管理对小块内存的申请和释放。

Python 内部默认的小块内存与大块内存的分界点定在 256 个字节，当申请的内存小于 256 字节时，PyObject_Malloc 会在内存池中申请内存；当申请的内存大于 256 字节时，PyObject_Malloc 的行为将蜕化为 malloc 的行为。当然，通过修改 Python 源代码，我们可以改变这个默认值，从而改变 Python 的默认内存管理行为。

调优手段（了解）

1.手动垃圾回收

2.调高垃圾回收阈值

3.避免循环引用（手动解循环引用和使用弱引用）

2.4、内存泄露是什么？如何避免？(2018-3-30-lxy)

指由于疏忽或错误造成程序未能释放已经不再使用的内存的情况。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制，因而造成了内存的浪费。导致程序运行速度减慢甚至系统崩溃等严重后果。

有 __del__() 函数的对象间的循环引用是导致内存泄漏的主凶。

不使用一个对象时使用:del object 来删除一个对象的引用计数就可以有效防止内存泄漏问题。

通过 Python 扩展模块 gc 来查看不能回收的对象的详细信息。

可以通过 sys.getrefcount(obj) 来获取对象的引用计数，并根据返回值是否为 0 来判断是否内存泄漏。

3、函数

3.1、函数调用参数的传递方式是值传递还是引用传递？(2018-3-30-lxy)

Python 的参数传递有：位置参数、默认参数、可变参数、关键字参数。

函数的传值到底是值传递还是引用传递，要分情况：

不可变参数用值传递：像整数和字符串这样的不可变对象，是通过拷贝进行传递的，因为你无论如何都不可能在原处改变不可变对象

可变参数是引用传递的：比如像列表，字典这样的对象是通过引用传递、和 C 语言里面的用指针传递数组很相似，可变对象能在函数内部改变。

3.2、python 常见的列表推导式？(2019-1-19-yzh)

[表达式 for 变量 in 列表]　或者　[表达式 for 变量 in 列表 if 条件]

3.3、如何在 function 里面设置一个全局变量(2018-5-2-xhq)

```
globals()

globals()  返回包含当前作用域全局变量的字典。
global 变量    设置使用全局变量
```

3.4、对缺省参数的理解 ？(2018-3-30-lxy)

缺省参数指在调用函数的时候没有传入参数的情况下，调用默认的参数，在调用函数的同时赋值时，所传入的参数会替代默认参数。

*args 是不定长参数，他可以表示输入参数是不确定的，可以是任意多个。

**kwargs 是关键字参数，赋值的时候是以键 = 值的方式，参数是可以任意多对在定义函数的时候不确定会有多少参数会传入时，就可以使用两个参数。

3.5、Mysql 怎么限制 IP 访问？(2018-4-16-lxy)

grant all privileges on . to ‘数据库中用户名’@’ip 地址’ identified by ‘数据库密码’;

3.6、带参数的装饰器? (2018-4-16-lxy)

带定长参数的装饰器。

```
def new_func(func):
    def wrappedfun(username,passwd):
        if username == 'root' and passwd == '123456789':
            print('通过认证！')
            print('开始执行附加功能')
            return func()
        else:
            print('用户名或密码错误')
            return
    return wrappedfun

@new_func
def orign():
    print('开始执行函数')
orign('root','123456789')
```

带不定长参数的装饰器。

```
def new_func(func):
    def wrappedfun(*parts):
        if parts:
            counts = len(parts)
            print('本系统包含 ', end='')
            for part in parts:
                print(part, ' ', end='')
            print('等', counts, '部分')
            return func()
        else:
            print('用户名或密码错误')
            return func()

    return wrappedfun

@new_func
def orign():
    print('开始执行函数')
 orign('硬件', '软件', '用户数据')
```

同时带不定长、关键字参数的装饰器。

```
def new_func(func):
    def wrappedfun(*args,**kwargs):
        if args:
            counts = len(args)
            print('本系统包含 ',end='')
            for arg in args:
                print(arg,' ',end='')
            print('等',counts,'部分')
```

```
            if kwargs:
                for k in kwargs:
                    v= kwargs[k]
                    print(k,'为：',v)
            return func()
        else:
            if kwargs:
                for kwarg in kwargs:
                    print(kwarg)
                    k,v = kwarg
                    print(k,'为：',v)
            return func()
    return wrappedfun

@new_func
def orign():
    print('开始执行函数')

orign('硬件','软件','用户数据',总用户数=5,系统版本='CentOS 7.4')
```

3.7、为什么函数名字可以当做参数用?(2018-3-30-lxy)

Python 中一切皆对象，函数名是函数在内存中的空间，也是一个对象。

3.8、Python 中 pass 语句的作用是什么？(2018-3-30-lxy)

在编写代码时只写框架思路，具体实现还未编写就可以用 pass 进行占位，使程序不报错，不会进行任何操作。

3.9、有这样一段代码，print c 会输出什么，为什么？(2018-3-30-lxy)

```
a = 10
b = 20
c = [a]
a = 15
```

答：10 对于字符串、数字，传递是相应的值。

3.10、交换两个变量的值？(2018-4-16-lxy)

```
a,b = b,a
```

3.11、简述 read、readline、readlines 的区别？(2019-1-19-yzh)

read	读取整个文件
readline	读取下一行
readlines	读取整个文件到一个迭代器以供我们遍历（读取到一个 list 中，以供使用，比较方便）

3.12、什么是 Hash（散列函数）？(2019-1-19-yzh)

Hash，一般翻译做“散列”，也有直接音译为“哈希”的，就是把任意长度的输入（又叫做预映射 pre-image）通过散列算法变换成固定长度的输出，该输出就是散列值。这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来确定唯一的输入值。简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。

3.13、python 函数重载机制？(2019-1-19-yzh)

所谓函数的重载是指多个函数的名称以及返回值类型均相同，仅参数类型或参数个数不同。函数重载大大提高了代码重用率和程序员开发效率。

函数重载主要是为了解决两个问题： 1.可变参数类型 2.可变参数个数

并且函数重载一个基本的设计原则是，仅仅当两个函数除了参数类型和参数个数不同以外，其功能是完全相同的，此时才使用函数重载，如果两个函数的功能 其实不同，不应当使用重载，而应当使用一个名字不同的函数。

那么对于情况 1，函数功能相同，但是参数类型不同，对于这种情况 python 根本不需要进行处理， 因为 Python 可以函数可以接受任意类型的参数，所以如果函数功能相同，只是不同的参数类型在 python 中没有什么不同，所以没有必要做成两个函数就没有必 要使用函数的重载。对于情况 2，如果是函数拥有同样的功能只是参数个数不同，这种情况对于 Python 同样不需要函数的重载，因为 Python 中存在缺省参数，对 于个数不确定的函数采用缺省参数的方法就能完成所有的工作。函数重载为了解决的两个问题在 Python 中都不是问题因此 Python 就没有再进行函数重载的必要了，所以 Python 中不存在函数的重载。

3.14、写一个函数找出一个整数数组中，第二大的数？(2019-1-19-yzh)

```
    def find_Second_large_num(num_list):
        ''''
        找出数组中第 2 大的数字
        '''
        #直接排序,输出倒数第二个数即可
        tmp_list=sorted(num_list)
        print 'Second_large_num is:', tmp_list[-2]
        #设置两个标志位一个存储最大数一个存储次大数
        #two 存储次大值，one 存储最大值，遍历一次数组即可，先判断是否大于 one，若大于将 one
的   值给 two，将 num_list[i]的值给 one；否则比较是否大于 two，若大于直接将 num_list[i]的
        #值给 two；否则 pass
        one=num_list[0]
        two=num_list[0]
        for i in range(1,len(num_list)):
            if num_list[i]>one:
            two=one
            one=num_list[i]
        elif num_list[i]>two:
            two=num_list[i]
        else:
            pass
        print 'Second_large_num is:', two
    if __name__ == '__main__':
        num_list=[34,11,23,56,78,0,9,12,3,7,5]
        find_Second_large_num(num_list)
```

3.15、map 函数和 reduce 函数？ (2018-3-30-lxy)

①从参数方面来讲：

map()包含两个参数，第一个参数是一个函数，第二个是序列（列表 或元组）。其中，函数（即 map 的第一个参数位置的函数）可以接收一个或多个参数。

reduce()第一个参数是函数，第二个是序列（列表或元组）。但是，其函数必须接收两个参数。

②从对传进去的数值作用来讲：

map()是将传入的函数依次作用到序列的每个元素，每个元素都是独自被函数“作用”一次 。

reduce()是将传人的函数作用在序列的第一个元素得到结果后，把这个结果继续与下一个元素作用（累积计算）。

3.16、使用 Python 内置的 filter()方法来过滤？(2019-1-19-yzh)

filter()函数用于过滤序列，过滤掉不符合条件的元素，返回一个迭代器对象，可以通过 list()转化为列表或者通过 for 循环取值。

filter()函数接收两个参数，第一个为函数，第二个为可迭代对象。返回序列中能使函数为真的元素的迭代器对象；如果函数为 None，则返回序列中值为 True 的元素。

```
def is_odd(n):
    """过滤出列表中的奇数"""
    return n % 2 == 1


res = filter(is_odd, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print(list(res))  # [1, 3, 5, 7, 9]
```

3.17、编写函数的 4 个原则(2019-1-19-yzh)

1）函数设计要尽量短小

2）函数声明要做到合理、简单、易于使用

3）函数参数设计应该考虑向下兼容

4）一个函数只做一件事情，尽量保证函数语句粒度的一致性

3.18、递归函数停止的条件？(2018-3-30-lxy)

递归的终止条件一般定义在递归函数内部，在递归调用前要做一个条件判断，根据判断的结果选择是继续调用自身，还是 return;返回终止递归。

终止的条件：

判断递归的次数是否达到某一限定值

2. 判断运算的结果是否达到某个范围等，根据设计的目的来选择

3.19、回调函数，如何通信的? (2018-3-30-lxy)

回调函数是把函数的指针(地址)作为参数传递给另一个函数，将整个函数当作一个对象，赋值给调用的函数。

3.20、Python 主要的内置数据类型都有哪些？ print dir（‘a ’）的输出？(2018-3-30-lxy)

内建类型：布尔类型、数字、字符串、列表、元组、字典、集合；

输出字符串‘a’的内建方法；

3.21、map(lambda x:x*x，[y for y in range(3)])的输出？(2018-3-30-lxy)

```
[0， 1， 4]
```

3.22、 hasattr() getattr() setattr() 函数使用详解？(2018-4-16-lxy)

hasattr(object, name)函数：

判断一个对象里面是否有 name 属性或者 name 方法，返回 bool 值，有 name 属性(方法)返回 True，否则返回 False。

注意：name 要使用引号括起来。

```
class function_demo(object):
    name = 'demo'
    def run(self):
        return "hello function"
functiondemo = function_demo()
res = hasattr(functiondemo, 'name')  #判断对象是否有 name 属性，True
res = hasattr(functiondemo, "run") #判断对象是否有 run 方法，True
res = hasattr(functiondemo, "age") #判断对象是否有 age 属性，Falsw
print(res)
```

getattr(object, name[,default]) 函数：

获取对象 object 的属性或者方法，如果存在则打印出来，如果不存在，打印默认值，默认值可选。注意：如果返回的是对象的方法，则打印结果是：方法的内存地址，如果需要运行这个方法，可以在后面添加括号()。

```
functiondemo = function_demo()
getattr(functiondemo, 'name') #获取 name 属性，存在就打印出来--- demo
getattr(functiondemo, "run") #获取 run 方法，存在打印出 方法的内存地址---<bound
method function_demo.run of <__main__.function_demo object at 0x10244f320>>
getattr(functiondemo, "age") #获取不存在的属性，报错如下：
Traceback (most recent call last):
  File "/Users/liuhuiling/Desktop/MT_code/OpAPIDemo/conf/OPCommUtil.py",
line 39, in <module>
    res = getattr(functiondemo, "age")
AttributeError: 'function_demo' object has no attribute 'age'
getattr(functiondemo, "age", 18)  #获取不存在的属性，返回一个默认值
```

setattr(object,name,values)函数：

给对象的属性赋值，若属性不存在，先创建再赋值

```
class function_demo(object):
    name = 'demo'
    def run(self):
        return "hello function"
functiondemo = function_demo()
res = hasattr(functiondemo, 'age')  # 判断 age 属性是否存在，False
print(res)
setattr(functiondemo, 'age', 18 )  #对 age 属性进行赋值，无返回值
res1 = hasattr(functiondemo, 'age') #再次判断属性是否存在，True
```

综合使用：

```
class function_demo(object):
    name = 'demo'
    def run(self):
        return "hello function"
functiondemo = function_demo()
res = hasattr(functiondemo, 'addr') # 先判断是否存在 if res:
    addr = getattr(functiondemo, 'addr')
    print(addr)else:
    addr = getattr(functiondemo, 'addr', setattr(functiondemo, 'addr', '北
京首都'))
    #addr = getattr(functiondemo, 'addr', '美国纽约')
    print(addr)
```

3.23、一句话解决阶乘函数？(2018-4-16-lxy)

```
reduce(lambda x,y: x*y, range(1,n+1))
```

3.24、Lambda

3.25、什么是 lambda 函数？ 有什么好处？(2018-4-16-lxy)

lambda 函数是一个可以接收任意多个参数(包括可选参数)并且返回单个表达式值的函数

1、lambda 函数比较轻便，即用即仍，很适合需要完成一项功能，但是此功能只在此一处使用，连名字都很随意的情况下；

2、匿名函数，一般用来给 filter， map 这样的函数式编程服务；

3、作为回调函数，传递给某些应用，比如消息处理

3.26、手写一个判断时间的装饰器(2019-1-19-yzh)

```
def zhuangshiqi(fun):
    def warrper(*args,**kwargs):
        a = time.clock()     #clock()函数以浮点数计算的秒数返回当前的 CPU 时间.用来衡量不同程序的耗时，比 time.time()更有用
        c = fun(*args,**kwargs)
        b = time.clock()
        print(b - c)
       return c
    return warrper
```

3.27、下面这段代码的输出结果将是什么？请解释。(2018-3-30-lxy)

```
def multipliers():
        return [lambda x : i * x for i in range(4)]
        print [m(2) for m in multipliers()]
```

上面代码输出的结果是[6， 6， 6， 6] (不是我们想的[0， 2， 4， 6])。

你如何修改上面的 multipliers 的定义产生想要的结果？

上述问题产生的原因是 Python 闭包的延迟绑定。这意味着内部函数被调用时，参数的值在闭包内进行查找。因此，当任何由 multipliers()返回的函数被调用时，i 的值将在附近的范围进行查找。那时，不管返回的函数是否被调用，for 循环已经完成，i 被赋予了最终的值 3。

因此，每次返回的函数乘以传递过来的值 3，因为上段代码传过来的值是 2，它们最终返回的都是 6。(3*2)碰巧的是，《The Hitchhiker's Guide to Python》也指出，在与 lambdas 函数相关也有一个被广泛被误解的知识点，不过跟这个 case 不一样。由 lambda 表达式创造的函数没有什么特殊的地方，它其实是和 def 创造的函数式一样的。

下面是解决这一问题的一些方法。

一种解决方法就是用 Python 生成器。

```
def multipliers():
2.      for i in range(4): yield lambda x : i * x
```

另外一个解决方案就是创造一个闭包，利用默认函数立即绑定。

```
1. def  multipliers():
2.  return [lambda x,  i=i : i * x for i in range(4)]
```

3.28、什么是 lambda 函数？它有什么好处？写一个匿名函数求两个数的(2018-3-30-lxy)

lambda 函数是匿名函数；使用 lambda 函数能创建小型匿名函数。这种函数得名于省略了用 def 声明函数的标准步骤；

```
f = lambda x, y:x+y
print(f(2017, 2018))
```

4、设计模式

4.1、Python 如何实现单例模式？请写出两种实现方法？

在 Python 中，我们可以用多种方法来实现单例模式：

1. 使用模块； 2.使用__new__； 3.使用装饰器； 4. 使用元类（metaclass）。

1）使用模块：其实，Python 的模块就是天然的单例模式，因为模块在第一次导入时，会生成.pyc 文件，当第二次导入时，就会直接加载.pyc 文件，而不会再次执行模块代码。因此我们只需把相关的函数和数据定义在一个模块中，就可以获得一个单例对象了。

```
# mysingle.py
class MySingle:
    def foo(self):
        pass

sinleton = MySingle()

将上面的代码保存在文件 mysingle.py 中，然后这样使用：
from mysingle import sinleton
singleton.foo()
```

2）使用__new__：为了使类只能出现一个实例，我们可以使用__new__来控制实例的创建过程，

```
class Singleton(object):
    def __new__(cls):
        # 关键在于这，每一次实例化的时候，我们都只会返回这同一个 instance 对象
        if not hasattr(cls, 'instance'):
            cls.instance = super(Singleton, cls).__new__(cls)
        return cls.instance

obj1 = Singleton()
obj2 = Singleton()

obj1.attr1 = 'value1'
print obj1.attr1, obj2.attr1
print obj1 is obj2

输出结果：
value1  value1
```

3）使用装饰器：装饰器可以动态的修改一个类或函数的功能。这里，我们也可以使用装饰器来装饰某个类，使其只能生成一个实例

```
def singleton(cls):
    instances = {}
    def getinstance(*args,**kwargs):
        if cls not in instances:
            instances[cls] = cls(*args,**kwargs)
        return instances[cls]
    return getinstance
```

```
    @singleton
    class MyClass:
        a = 1

    c1 = MyClass()
    c2 = MyClass()
    print(c1 == c2) # True
```

在上面，我们定义了一个装饰器 singleton，它返回了一个内部函数 getinstance，

该函数会判断某个类是否在字典 instances 中，如果不存在，则会将 cls 作为 key，cls(*args, **kw) 作为 value 存到 instances 中，

否则，直接返回 instances[cls]。

4）使用 metaclass（元类）：元类可以控制类的创建过程，它主要做三件事：

- 拦截类的创建
- 修改类的定义
- 返回修改后的类

```
    class Singleton2(type):
        def __init__(self, *args, **kwargs):
            self.__instance = None
            super(Singleton2,self).__init__(*args, **kwargs)

        def __call__(self, *args, **kwargs):
            if self.__instance is None:
                self.__instance = super(Singleton2,self).__call__(*args,
**kwargs)
            return self.__instance


    class Foo(object):
        __metaclass__ = Singleton2 #在代码执行到这里的时候，元类中的__new__方法和
__init__方法其实已经被执行了，而不是在 Foo 实例化的时候执行。且仅会执行一次。


    foo1 = Foo()
    foo2 = Foo()
    print (Foo.__dict__)  #_Singleton__instance': <__main__.Foo object at
0x100c52f10> 存在一个私有属性来保存属性，而不会污染 Foo 类（其实还是会污染，只是无法直接通
过__instance 属性访问）
    print (foo1 is foo2)  # True
```

4.2、请手写一个单例(2018-3-30-lxy)

```
    class A(object):
        __instance = None
        def __new__(cls,  *args,  **kwargs):
            if  cls.__instance is None:
                cls.__instance = object.__new__(cls)
                return cls.__instance
            else:
                return cls.__instance
```

4.3、单例模式的应用场景有哪些？(2018-4-16-lxy)

单例模式应用的场景一般发现在以下条件下：

资源共享的情况下，避免由于资源操作时导致的性能或损耗等。如日志文件，应用配置。

（2）控制资源的情况下，方便资源之间的互相通信。如线程池等。 1.网站的计数器 2.应用配置 3.多线程池 4.数据库配置，数据库连接池 5.应用程序的日志应用....

4.4、谈一下对设计模式的认识和理解，并简述几个你了解的设计模式？(2019-1-19-yzh)

设计模式是经过总结、优化的，对我们经常会碰到的一些编程问题的可重用解决方案。一个设计模式并不像一个类或一个库那样能够直接作用于我们的代码。反之，设计模式更为高级，它是一种必须在特定情形下实现的一种方法模板。

常见的是工厂模式和单例模式。

4.5、对装饰器的理解 ，并写出一个计时器记录方法执行性能的装饰器？(2018-3-30-lxy)

装饰器本质上是一个 Python 函数，它可以让其他函数在不需要做任何代码变动的前提下增加额外功能，装饰器的返回值也是一个函数对象。

```
import time
def timeit(func):
   def wrapper():
        start = time.clock()
        func() end =time.clock()
        print 'used:', end - start
        return wrapper
@timeit
def foo():
    print 'in foo()'foo()
```

4.6、解释一下什么是闭包?(2018-3-30-lxy)

在函数内部再定义一个函数，并且这个函数用到了外边函数的变量，那么将这个函数以及用到的一些变量称之为闭包。

4.7、函数装饰器有什么作用？ (2018-4-16-lxy)

装饰器本质上是一个 Python 函数，它可以在让其他函数在不需要做任何代码的变动的前提下增加额外的功能。 装饰器的返回值也是一个函数的对象，它经常用于有切面需求的场景。 比如：插入日志、性能测试、事务处理、缓存、权限的校验等场景 有了装饰器就可以抽离出大量的与函数功能本身无关的雷同代码并发并继续使用。

4.8、生成器、迭代器的区别？(2018-3-30-lxy)

迭代器是一个更抽象的概念，任何对象，如果它的类有 next 方法和 iter 方法返回自己本身，对于 string、list、dict、tuple 等这类容器对象，使用 for 循环遍历是很方便的。在后台 for 语句对容器对象调用 iter()函数，iter()是 python 的内置函数。iter()会返回一个定义了 next()方法的迭代器对象，它在容器中逐个访问容器内元素，next()也是 python 的内置函数。在没有后续元素时，next()会抛出一个 StopIteration 异常。生成器（Generator）是创建迭代器的简单而强大的工具。它们写起来就像是正规的函数，只是在需要返回数据的时候使用 yield 语句。每次 next()被调用时，生成器会返回它脱离的位置（它记忆语句最后一次执行的位置和所有的数据值）

区别：生成器能做到迭代器能做的所有事,而且因为自动创建了 iter()和 next()方法,生成器显得特别简洁,而且生成器也是高效的，使用生成器表达式取代列表解析可以同时节省内存。除了创建和保存程序状态的自动方法,当发生器终结时,还会自动抛出 StopIteration 异常。

4.9 X 是什么类型？(2018-3-30-lxy)

```
X = (for i in ramg(10))
```

答：X 是 generator 类型。

4.10、请用“一行代码”实现将 1-N 的整数列表以 3 为单位分组 (2018-4-20-lxy)

比如 1-100 分组后为?

```
print([[x for x in range(1, 100)][i:i+3] for i in range(0, len(list_a), 3)])
```

4.11、Python 中 yield 的用法？(2018-4-16-lxy)

yield 就是保存当前程序执行状态。你用 for 循环的时候，每次取一个元素的时候就会计算一次。用 yield 的函数叫 generator，和 iterator 一样，它的好处是不用一次计算所有元素，而是用一次算一次，可以节省很多空间。generator 每次计算需要上一次计算结果，所以用 yield，否则一 return，上次计算结果就没了。

```
>>> def createGenerator():
...    mylist = range(3)
...    for i in mylist:
...       yield i*i
...
>>> mygenerator = createGenerator() # create a generator
>>> print(mygenerator) # mygenerator is an object!
<generator object createGenerator at 0xb7555c34>
>>> for i in mygenerator:
...     print(i)
0
1
4
```

5、面向对象

5.1、Python 中的可变对象和不可变对象？(2018-3-30-lxy)

不可变对象，该对象所指向的内存中的值不能被改变。当改变某个变量时候，由于其所指的值不能被改变，相当于把原来的值复制一份后再改变，这会开辟一个新的地址，变量再指向这个新的地址。

可变对象，该对象所指向的内存中的值可以被改变。变量（准确的说是引用）改变后，实际上是其所指的值直接发生改变，并没有发生复制行为，也没有开辟新的出地址，通俗点说就是原地改变。

Python 中，数值类型（int 和 float）、字符串 str、元组 tuple 都是不可变类型。而列表 list、字典 dict、集合 set 是可变类型。

5.2、Python 中 is 和==的区别？(2018-3-30-lxy)

is 判断的是 a 对象是否就是 b 对象，是通过 id 来判断的。

==判断的是 a 对象的值是否和 b 对象的值相等，是通过 value 来判断的。

5.3、Python 的魔法方法 (2018-3-30-lxy)

魔法方法就是可以给你的类增加魔力的特殊方法，如果你的对象实现 （重载）了这些方法中的某一个，那么这个方法就会在特殊的情况下被 Python 所调用，你可以定义自己想要的行为，而这一切都是自动发生的。 它们经常是两个下划线包围来命名的（比如 __init__，__lt__），Python 的魔法方法是非常强大的，所以了解其使用方法也变得尤为重要！

__init__ 构造器，当一个实例被创建的时候初始化的方法。但是它并 不是实例化调用的第一个方法。

__new__才是实例化对象调用的第一个方法，它只取下 cls 参数，并把 其他参数传给 __init__。__new__很少使用，但是也有它适合的场景，尤其 是当类继承自一个像元组或者字符串这样不经常改变的类型的时候。

__call__ 允许一个类的实例像函数一样被调用 。

__getitem__ 定义获取容器中指定元素的行为，相当于 self[key] 。

__getattr__ 定义当用户试图访问一个不存在属性的时候的行为 。

__setattr__ 定义当一个属性被设置的时候的行为 。

__getattribute__ 定义当一个属性被访问的时候的行为 。

5.4、面向对象中怎么实现只读属性? (2018-3-30-lxy)

将对象私有化，通过共有方法提供一个读取数据的接口。

```
class person:
    def __init__(self,x):
        self.__age = 10;
    def age(self):
        return self.__age;
t = person(22)
# t.__age = 100
print(t.age())
```

最好的方法

```
class MyCls(object):
    __weight = 50

    @property  #以访问属性的方式来访问 weight 方法
    def weight(self):
        return self.__weight

if __name__ == '__main__':
    obj = MyCls()
    print(obj.weight)
    obj.weight = 12

Traceback (most recent call last):
50
  File "C:/PythonTest/test.py", line 11, in <module>
    obj.weight = 12
AttributeError: can't set attribute
```

5.5、谈谈你对面向对象的理解?

面向对象是相对于面向过程而言的。面向过程语言是一种基于功能分析的、以算法为中心的程序设计方法；而面向对象是一种基于结构分析的、以数据为中心的程序设计思想。在面向对象语言中有一个有很重要东西，叫做类。面向对象有三大特性：封装、继承、多态。

6、正则表达式

6.1、Python 里 match 与 search 的区别？(2018-3-30-lxy)

match()函数只检测 RE 是不是在 string 的开始位置匹配，search()会扫描整个 string 查找匹配；也就是说 match()只有在 0 位置匹配成功的话才有返回，如果不是开始位置匹配成功的话，match()就返回 none。

6.2、a = “abbbccc”，用正则匹配为 abccc,不管有多少 b，就出现一次？(2018-4-16-lxy)

```
思路：不管有多少个 b 替换成一个
re.sub(r'b+', 'b', a)
```

6.3、Python 字符串查找和替换？(2018-3-30-lxy)

```
re.findall(r’目的字符串’, ’原有字符串’) #查询
re.findall(r'cast', 'itcast.cn')[0]
re.sub(r‘要替换原字符’, ’要替换新字符’, ’原始字符串’)
re.sub(r'cast', 'heima', 'itcast.cn')
```

6.4、 用 Python 匹配 HTML g tag 的时候，<.*> 和 <.*?> 有什么区别(2018-3-30-lxy)

<.*>是贪婪匹配，会从第一个“<”开始匹配，直到最后一个“>”中间所有的字符都会匹配到，中间可能会包含“<>”。

<.*?>是非贪婪匹配，从第一个“<”开始往后，遇到第一个“>”结束匹配，这中间的字符串都会匹配到，但是不会有“<>”。

6.5、正则表达式贪婪与非贪婪模式的区别？(2018-4-16-lxy)

在形式上非贪婪模式有一个“？”作为该部分的结束标志。

在功能上贪婪模式是尽可能多的匹配当前正则表达式，可能会包含好几个满足正则表达式的字符串，非贪婪模式，在满足所有正则表达式的情况下尽可能少的匹配当前正则表达式。

6.6、写出开头匹配字母和下划线，末尾是数字的正则表达式？(2018-4-16-lxy)

```
^[A-Za-z]|_.*\d$
```

6.7、正则表达式操作(2018-5-1-lxy)

匹配手机号

分析：

（1）手机号位数为 11 位；

（2）开头为 1，第二位为 3 或 4 或 5 或 7 或 8;

表达式为：`/^[1][3,4,5,7,8][0-9]{9}$/;`

6.8、请匹配出变量 A = 'json({"Adam":95,"Lisa":85,"Bart":59})'中的 json 字符串。

```
A = 'json({"Adam":95,"Lisa":85,"Bart":59})'
b = re.search(r'json.*?({.*?}).*',A,re.S)
print(b.group(1))
```

6.9、怎么过滤评论中的表情?

```
    co = re.compile(u'[\uD800-\uDBFF][\uDC00-\uDFFF]')
    co.sub(‘’,text)
```

6.10、简述 Python 里面 search 和 match 的区别（2018-5-2-zcz）

match()函数只检测 RE 是不是在 string 的开始位置匹配，search()会扫描整个 string 查找匹配；也就是说 match()只有在 0 位置匹配成功的话才有返回，如果不是开始位置匹配成功的话，match()就返回 none；

例如：

```
    print(re.match('super', 'superstition').span())   会返回(0, 5)
    而 print(re.match('super', 'insuperable'))   则返回 None
    search()会扫描整个字符串并返回第一个成功的匹配：
    例如：
    print(re.search('super', 'superstition').span())返回(0, 5)
    print(re.search('super', 'insuperable').span())返回(2, 7)
    其中 span 函数定义如下，返回位置信息：
    span([group]):
    返回(start(group), end(group))。
```

6.11、请写出匹配 ip 的 Python 正则表达式（2018-5-2-xhq）

iPv4 的 ip 地址都是（1~255）.（0~255）.（0~255）.（0~255）的格式。

下面给出相对应的正则表达式：

```
    "^(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|[1-9])\\."
    +"(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|\\d)\\."
    +"(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|\\d)\\."
    +"(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|\\d)$"
```

简单的讲解一下：

\\d 表示 0~9 的任何一个数字

{2}表示正好出现两次

[0-4]表示 0~4 的任何一个数字

| 的意思是或者

( )上面的括号不能少，是为了提取匹配的字符串，表达式中有几个()就表示有几个相应的匹配字符串

1\\d{2}的意思就是 100~199 之间的任意一个数字

2[0-4]\\d 的意思是 200~249 之间的任意一个数字

25[0-5]的意思是 250~255 之间的任意一个数字

[1-9]\\d 的意思是 10~99 之间的任意一个数字

[1-9])的意思是 1~9 之间的任意一个数字

\\.的意思是.点要转义（特殊字符类似，@都要加\\转义）

6.12、请写出一段代码用正则匹配出 ip? (2019-1-19-yzh)

```
    import re src = "security/afafsff/?ip=123.4.56.78&id=45"
    print(re.findall( r'([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|2
5[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])',src))
```

7、系统编程

7.1、进程总结(2019-1-19-yzh)

进程：程序运行在操作系统上的一个实例，就称之为进程。进程需要相应的系统资源：内存、时间片、pid。

创建进程：

首先要导入 multiprocessing 中的 Process；

创建一个 Process 对象；

创建 Process 对象时，可以传递参数；

```
p = Process(target=XXX, args=(元组,) , kwargs={key:value})
target = XXX 指定的任务函数,不用加()
args=(元组,) , kwargs={key:value} 给任务函数传递的参数
```

使用 start()启动进程；

结束进程。

Process 语法结构：

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

target：如果传递了函数的引用，可以让这个子进程就执行函数中的代码

args：给 target 指定的函数传递的参数，以元组的形式进行传递

kwargs：给 target 指定的函数传递参数，以字典的形式进行传递

name：给进程设定一个名字，可以省略

group：指定进程组，大多数情况下用不到

Process 创建的实例对象的常用方法有：

start()：启动子进程实例(创建子进程)

is_alive()：判断进程子进程是否还在活着

join(timeout)：是否等待子进程执行结束，或者等待多少秒

terminate()：不管任务是否完成，立即终止子进程

Process 创建的实例对象的常用属性：

name：当前进程的别名，默认为 Process-N,N 为从 1 开始递增的整数

pid：当前进程的 pid(进程号)

给子进程指定函数传递参数 Demo：

```
import osfrom multiprocessing import Process
import time

def pro_func(name, age, **kwargs):
    for i in range(5):
        print("子进程正在运行中,name=%s, age=%d, pid=%d" %(name, age,
os.getpid()))
        print(kwargs)
        time.sleep(0.2)

if __name__ == '__main__':
    # 创建 Process 对象
    p = Process(target=pro_func, args=('小明',18), kwargs={'m': 20})
    # 启动进程
    p.start()
    time.sleep(1)
    # 1 秒钟之后，立刻结束子进程
    p.terminate()
```

```
    p.join()
```

注意：进程间不共享全局变量。

进程之间的通信-Queue

在初始化 Queue()对象时，(例如 q=Queue()，若在括号中没有指定最大可接受的消息数量，或数量为负值时，那么就代表可接受的消息数量没有上限-直到内存的尽头)

Queue.qsize()：返回当前队列包含的消息数量。

Queue.empty()：如果队列为空，返回 True,反之 False。

Queue.full()：如果队列满了，返回 True，反之 False。

Queue.get([block[,timeout]])：获取队列中的一条消息，然后将其从队列中移除，block 默认值为 True。

如果 block 使用默认值，且没有设置 timeout（单位秒），消息列队如果为空，此时程序将被阻塞（停在读取状态），直到从消息列队读到消息为止，如果设置了 timeout，则会等待 timeout 秒，若还没读取到任何消息，则抛出"Queue.Empty"异常；

如果 block 值为 False，消息列队如果为空，则会立刻抛出"Queue.Empty"异常；

Queue.get_nowait()：相当 Queue.get(False)；

Queue.put(item,[block[, timeout]])：将 item 消息写入队列，block 默认值为 True；

如果 block 使用默认值，且没有设置 timeout（单位秒），消息列队如果已经没有空间可写入，此时程序将被阻塞（停在写入状态），直到从消息列队腾出空间为止，如果设置了 timeout，则会等待 timeout 秒，若还没空间，则抛出"Queue.Full"异常；

如果 block 值为 False，消息列队如果没有空间可写入，则会立刻抛出"Queue.Full"异常；

Queue.put_nowait(item)：相当 Queue.put(item, False)；

进程间通信 Demo：

```
from multiprocessing import Process, Queueimport os, time, random
# 写数据进程执行的代码:def write(q):
    for value in ['A', 'B', 'C']:
        print('Put %s to queue...' % value)
        q.put(value)
        time.sleep(random.random())
# 读数据进程执行的代码:def read(q):
    while True:
        if not q.empty():
            value = q.get(True)
            print('Get %s from queue.' % value)
            time.sleep(random.random())
        else:
            break
if __name__=='__main__':
    # 父进程创建 Queue，并传给各个子进程：
    q = Queue()
    pw = Process(target=write, args=(q,))
    pr = Process(target=read, args=(q,))
    # 启动子进程 pw，写入：
    pw.start()
    # 等待 pw 结束：
    pw.join()
    # 启动子进程 pr，读取：
    pr.start()
    pr.join()
    # pr 进程里是死循环，无法等待其结束，只能强行终止：
    print('')
    print('所有数据都写入并且读完')
```

进程池 Pool

```
# -*- coding:utf-8 -*-
```

```
from multiprocessing import Poolimport os, time, random
def worker(msg):
    t_start = time.time()
    print("%s 开始执行,进程号为%d" % (msg,os.getpid()))
    # random.random()随机生成 0~1 之间的浮点数
    time.sleep(random.random()*2)
    t_stop = time.time()
    print(msg,"执行完毕，耗时%0.2f" % (t_stop-t_start))

po = Pool(3)  # 定义一个进程池，最大进程数 3
for i in range(0,10):
    # Pool().apply_async(要调用的目标,(传递给目标的参数元祖,))
    # 每次循环将会用空闲出来的子进程去调用目标
    po.apply_async(worker,(i,))

print("----start----")
po.close()  # 关闭进程池，关闭后 po 不再接收新的请求
po.join()  # 等待 po 中所有子进程执行完成，必须放在 close 语句之后
print("-----end-----")
```

multiprocessing.Pool 常用函数解析：

apply_async(func[, args[, kwds]]) ：使用非阻塞方式调用 func（并行执行，堵塞方式必须等待上一个进程退出才能执行下一个进程），args 为传递给 func 的参数列表，kwds 为传递给 func 的关键字参数列表；

close()：关闭 Pool，使其不再接受新的任务；

terminate()：不管任务是否完成，立即终止；

join()：主进程阻塞，等待子进程的退出， 必须在 close 或 terminate 之后使用；

进程池中使用 Queue

如果要使用 Pool 创建进程，就需要使用 multiprocessing.Manager()中的 Queue()，而不是 multiprocessing.Queue()，否则会得到一条如下的错误信息：

RuntimeError: Queue objects should only be shared between processes through inheritance.

```
from multiprocessing import Manager,Poolimport os,time,random
def reader(q):
    print("reader 启动(%s),父进程为(%s)" % (os.getpid(), os.getppid()))
    for i in range(q.qsize()):
        print("reader 从 Queue 获取到消息：%s" % q.get(True))
def writer(q):
    print("writer 启动(%s),父进程为(%s)" % (os.getpid(), os.getppid()))
    for i in "itcast":
        q.put(i)
if __name__=="__main__":
    print("(%s) start" % os.getpid())
    q = Manager().Queue()  # 使用 Manager 中的 Queue
    po = Pool()
    po.apply_async(writer, (q,))

    time.sleep(1)  # 先让上面的任务向 Queue 存入数据，然后再让下面的任务开始从中取数据

    po.apply_async(reader, (q,))
    po.close()
    po.join()
    print("(%s) End" % os.getpid())
```

7.2、谈谈你对多进程，多线程，以及协程的理解，项目是否用？(2018-3-30-lxy)

这个问题被问的概率相当之大，其实多线程，多进程，在实际开发中用到的很少，除非是那些对项目性能要求特别高的，有的开发工作几年了，也确实没用过，你可以这么回答，给他扯扯什么是进程，线程（cpython 中是伪多线程）的概念就行，实在不行你就说你之前写过下载文件时，用过多线程技术，或者业余时间用过多线程写爬虫，提升效率。

进程：一个运行的程序（代码）就是一个进程，没有运行的代码叫程序，进程是系统资源分配的最小单位，进程拥有自己独立的内存空间，所以进程间数据不共享，开销大。

线程： 调度执行的最小单位，也叫执行路径，不能独立存在，依赖进程存在一个进程至少有一个线程，叫主线程，而多个线程共享内存(数据共享，共享全局变量)，从而极大地提高了程序的运行效率。

协程：是一种用户态的轻量级线程，协程的调度完全由用户控制。协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈，直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快。

7.3、Python 异步使用场景有那些？(2019-1-19-yzh)

异步的使用场景：

1、不涉及共享资源，或对共享资源只读，即非互斥操作

2、没有时序上的严格关系

3、不需要原子操作，或可以通过其他方式控制原子性

4、常用于 IO 操作等耗时操作，因为比较影响客户体验和使用性能

5、不影响主线程逻辑

7.4、多线程共同操作同一个数据互斥锁同步？(2019-1-19-yzh)

```
import threadingimport time
class MyThread(threading.Thread):
    def run(self):
        global num
        time.sleep(1)

        if mutex.acquire(1):
            num += 1
            msg = self.name+' set num to '+str(num)
            print msg
            mutex.release()
num = 0
mutex = threading.Lock()def test():
    for i in range(5):
        t = MyThread()
        t.start()if __name__ == '__main__':
    test()
```

7.5、什么是多线程竞争？(2018-3-30-lxy)

线程是非独立的，同一个进程里线程是数据共享的，当各个线程访问数据资源时会出现竞争状态即：数据几乎同步会被多个线程占用，造成数据混乱 ，即所谓的线程不安全

那么怎么解决多线程竞争问题？-- 锁。

锁的好处：

确保了某段关键代码(共享数据资源)只能由一个线程从头到尾完整地执行能解决多线程资源竞争下的原子操作问题。

锁的坏处:

阻止了多线程并发执行，包含锁的某段代码实际上只能以单线程模式执行，效率就大大地下降了

锁的致命问题：死锁。

7.6、请介绍一下 Python 的线程同步？(2019-1-19-yzh)

一、setDaemon(False)

当一个进程启动之后，会默认产生一个主线程，因为线程是程序执行的最小单位，当设置多线程时，主线程会创建多个子线程，在 Python 中，默认情况下就是 setDaemon(False)，主线程执行完自己的任务以后，就退出了，此时子线程会继续执行自己的任务，直到自己的任务结束；

案例

```
import threadingimport time
def thread():
    time.sleep(2)
    print('---子线程结束---')
def main():
    t1 = threading.Thread(target=thread)
    t1.start()
    print('---主线程---结束')
if __name__ == '__main__':
    main()
Python
Copy
执行结果
---主线程结束------子线程结束---
```

二、setDaemon(True)

当我们使用 setDaemon(True)时，这是子线程为守护线程，主线程一旦执行结束，则全部子线程被强制终止。

案例

```
import threadingimport time
def thread():
    time.sleep(2)
    print('---子线程结束---')
def main():
    t1 = threading.Thread(target=thread)
    t1.setDaemon(True)  # 设置子线程守护主线程
    t1.start()
    print('---主线程结束---')
if __name__ == '__main__':
    main()
```

执行结果

```
---主线程结束---  # 只有主线程结束，子线程来不及执行就被强制结束
```

三、join(线程同步)

join 所完成的工作就是线程同步，即主线程任务结束以后，进入堵塞状态，一直等待所有的子线程结束以后，主线程再终止

当设置守护线程时，含义是主线程对于子线程等待 timeout 的时间将会杀死该子线程，最后退出程序。所以说，如果有 10 个子线程，全部的等待时间就是每个 timeout 的累加和。简单的来说，就是给每个子线程一个 timeout 的时间，让他去执行，时间一到，不管任务有没有完成，直接杀死。

没有设置守护线程时，主线程将会等待 timeout 的累加和这样的一段时间，时间一到，主线程结束，但是并没有杀死子线程，子线程依然可以继续执行，直到子线程全部结束，程序退出。

案例

```
import threadingimport time
def thread():
    time.sleep(2)
    print('---子线程结束---')
def main():
    t1 = threading.Thread(target=thread)
    t1.setDaemon(True)
    t1.start()
    t1.join(timeout=1)  # 1、线程同步，主线程堵塞 1s 然后主线程结束，子线程继续执行
                        # 2、如果不设置 timeout 参数就等子线程结束主线程再结束
                        # 3、如果设置了 setDaemon=True 和 timeout=1 主线程等待 1s 后会强制杀死子线程，然后主线程结束
    print('---主线程结束---')
if __name__ == '__main__':
    main()
Python
Copy
执行结果
---主线程结束------子线程结束---
```

7.7、解释一下什么是锁，有哪几种锁? (2018-3-30-lxy)

锁(Lock)是 Python 提供的对线程控制的对象。有互斥锁、可重入锁、死锁。

7.8、什么是死锁呢？(2018-3-30-lxy)

若干子线程在系统资源竞争时，都在等待对方对某部分资源解除占用状态，结果是谁也不愿先解锁，互相干等着，程序无法执行下去，这就是死锁。

GIL 锁（有时候，面试官不问，你自己要主动说，增加 b 格，尽量别一问一答的尬聊，不然最后等到的一句话就是：你还有什么想问的么？）

GIL 锁 全局解释器锁（只在 cpython 里才有）

作用：限制多线程同时执行，保证同一时间只有一个线程执行，所以 cpython 里的多线程其实是伪多线程!

所以 Python 里常常使用协程技术来代替多线程，协程是一种更轻量级的线程，

进程和线程的切换时由系统决定，而协程由我们程序员自己决定，而模块 gevent 下切换是遇到了耗时操作才会切换。

三者的关系：进程里有线程，线程里有协程。

7.9、多线程交互访问数据，如果访问到了就不访问了(2018-4-16-lxy)

怎么避免重读?

创建一个已访问数据列表，用于存储已经访问过的数据，并加上互斥锁，在多线程访问数据的时候先查看数据是否已经在已访问的列表中，若已存在就直接跳过。

7.10、什么是线程安全，什么是互斥锁？(2018-3-30-lxy)

每个对象都对应于一个可称为" 互斥锁" 的标记，这个标记用来保证在任一时刻，只能有一个线程访问该对象。

同一个进程中的多线程之间是共享系统资源的，多个线程同时对一个对象进行操作，一个线程操作尚未结束，另一个线程已经对其进行操作，导致最终结果出现错误，此时需要对被操作对象添加互斥锁，保证每个线程对该对象的操作都得到正确的结果。

7.11、说说下面几个概念：同步，异步，阻塞，非阻塞?(2018-3-30-lxy)

同步：多个任务之间有先后顺序执行，一个执行完下个才能执行。

异步：多个任务之间没有先后顺序，可以同时执行有时候一个任务可能要在必要的时候获取另一个同时执行的任务的结果，这个就叫回调！

阻塞：如果卡住了调用者，调用者不能继续往下执行，就是说调用者阻塞了。

非阻塞：如果不会卡住，可以继续执行，就是说非阻塞的。

同步异步相对于多任务而言，阻塞非阻塞相对于代码执行而言。

7.12、什么是僵尸进程和孤儿进程？怎么避免僵尸进程? (2018-3-30-lxy)

孤儿进程：父进程退出，子进程还在运行的这些子进程都是孤儿进程，孤儿进程将被 init 进程(进程号为 1)所收养，并由 init 进程对它们完成状态收集工作。

僵尸进程：进程使用 fork 创建子进程，如果子进程退出，而父进程并没有调用 wait 或 waitpid 获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中的这些进程是僵尸进程。

避免僵尸进程的方法：

1.fork 两次用孙子进程去完成子进程的任务；

2.用 wait()函数使父进程阻塞；

3.使用信号量，在 signal handler 中调用 waitpid，这样父进程不用阻塞。

7.13、Python 中的进程与线程的使用场景? (2018-3-30-lxy)

多进程适合在 CPU 密集型操作(cpu 操作指令比较多，如位数多的浮点运算)。

多线程适合在 IO 密集型操作(读写数据操作较多的，比如爬虫)。

7.14、线程是并发还是并行，进程是并发还是并行？(2018-3-30-lxy)

线程是并发，进程是并行；

进程之间相互独立，是系统分配资源的最小单位，同一个线程中的所有线程共享资源。

7.15、并行（parallel）和并发（concurrency）？(2018-3-30-lxy)

并行：同一时刻多个任务同时在运行。

并发：在同一时间间隔内多个任务都在运行，但是并不会在同一时刻同时运行，存在交替执行的情况。

实现并行的库有：multiprocessing

实现并发的库有：threading

程序需要执行较多的读写、请求和回复任务的需要大量的 IO 操作，IO 密集型操作使用并发更好。

CPU 运算量大的程序程序，使用并行会更好。

7.16、IO 密集型和 CPU 密集型区别？(2018-4-16-lxy)

IO 密集型：系统运作，大部分的状况是 CPU 在等 I/O (硬盘/内存)的读/写。
CPU 密集型：大部份时间用来做计算、逻辑判断等 CPU 动作的程序称之 CPU 密集型。

8、网络编程

8.1、怎么实现强行关闭客户端和服务器之间的连接? (2018-3-30-lxy)

在 socket 通信过程中不断循环检测一个全局变量(开关标记变量)，一旦标记变量变为关闭，则 调用 socket 的 close 方法，循环结束，从而达到关闭连接的目的。

8.2、简述 TCP 和 UDP 的区别以及优缺点? (2018-4-16-lxy)

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。
优点：UDP 速度快、操作简单、要求系统资源较少，由于通讯不需要连接，可以实现广播发送
缺点：UDP 传送数据前并不与对方建立连接，对接收到的数据也不发送确认信号，发送端不知道数据是否会正确接收，也不重复发送，不可靠。
TCP 是面向连接的通讯协议，通过三次握手建立连接，通讯完成时四次挥手
优点：TCP 在数据传递时，有确认、窗口、重传、阻塞等控制机制，能保证数据正确性，较为可靠。
缺点：TCP 相对于 UDP 速度慢一点，要求系统资源较多。

8.3、简述浏览器通过 WSGI 请求动态资源的过程? (2018-4-16-lxy)

1.发送 http 请求动态资源给 web 服务器
2.web 服务器收到请求后通过 WSGI 调用一个属性给应用程序框架
3.应用程序框架通过引用 WSGI 调用 web 服务器的方法，设置返回的状态和头信息。
4.调用后返回，此时 web 服务器保存了刚刚设置的信息
5.应用程序框架查询数据库，生成动态页面的 body 的信息
6.把生成的 body 信息返回给 web 服务器
7.web 服务器吧数据返回给浏览器

8.4、描述用浏览器访问 www.baidu.com 的过程(2018-4-16-lxy)

先要解析出 baidu.com 对应的 ip 地址
要先使用 arp 获取默认网关的 mac 地址
组织数据发送给默认网关(ip 还是 dns 服务器的 ip，但是 mac 地址是默认网关的 mac 地址)
默认网关拥有转发数据的能力，把数据转发给路由器
路由器根据自己的路由协议，来选择一个合适的较快的路径转发数据给目的网关
目的网关(dns 服务器所在的网关)，把数据转发给 dns 服务器
dns 服务器查询解析出 baidu.com 对应的 ip 地址，并原路返回请求这个域名的 client
得到了 baidu.com 对应的 ip 地址之后，会发送 tcp 的 3 次握手，进行连接
使用 http 协议发送请求数据给 web 服务器
web 服务器收到数据请求之后，通过查询自己的服务器得到相应的结果，原路返回给浏览器。
浏览器接收到数据之后通过浏览器自己的渲染功能来显示这个网页。
浏览器关闭 tcp 连接，即 4 次挥手结束，完成整个访问过程

8.5、Post 和 Get 请求的区别? (2018-4-16-lxy)

GET 请求，请求的数据会附加在 URL 之后，以?分割 URL 和传输数据，多个参数用&连接。URL 的编码格式采用的是 ASCII 编码，而不是 uniclde，即是说所有的非 ASCII 字符都要编码之后再传输。

POST 请求：POST 请求会把请求的数据放置在 HTTP 请求包的包体中。上面的 item=bandsaw 就是实际的传输数据。

因此，GET 请求的数据会暴露在地址栏中，而 POST 请求则不会

传输数据的大小：

在 HTTP 规范中，没有对 URL 的长度和传输的数据大小进行限制。但是在实际开发过程中，对于 GET，特定的浏览器和服务器对 URL 的长度有限制。因此，在使用 GET 请求时，传输数据会受到 URL 长度的限制。

对于 POST，由于不是 URL 传值，理论上是不会受限制的，但是实际上各个服务器会规定对 POST 提交数据大小进行限制，Apache、IIS 都有各自的配置。

安全性：

POST 的安全性比 GET 的高。这里的安全是指真正的安全，而不同于上面 GET 提到的安全方法中的安全，上面提到的安全仅仅是不修改服务器的数据。比如，在进行登录操作，通过 GET 请求，用户名和密码都会暴露再 URL 上，因为登录页面有可能被浏览器缓存以及其他人查看浏览器的历史记录的原因，此时的用户名和密码就很容易被他人拿到了。除此之外，GET 请求提交的数据还可能会造成 Cross-site request frogery 攻击。

8.6、cookie 和 session 的区别？(2018-4-16-lxy)

1、cookie 数据存放在客户的浏览器上，session 数据放在服务器上。

2、cookie 不是很安全，别人可以分析存放在本地的 cookie 并进行 cookie 欺骗考虑　到安全应当使用 session。

3、session 会在一定时间内保存在服务器上。当访问增多，会比较占用服务器的性能 考虑到减轻服务器性能方面，应当使用 cookie。

4、单个 cookie 保存的数据不能超过 4K，很多浏览器都限制一个站点最多保存 20 个 cookie。

5、建议：　将登陆信息等重要信息存放为 SESSION 其他信息如果需要保留，可以放在　　cookie 中

6.cookie 限制

一次打开网页会生成一个随机 cookie，如果再次打开网页这个 cookie 不存在，那么再次设置，第三次打开仍然不存在，这就非常有可能是爬虫在工作了。

解决措施：在 headers 挂上相应的 cookie 或者根据其方法进行构造（例如从中选取几个字母进行构造）。如果过于复杂，可以考虑使用 selenium 模块（可以完全模拟浏览器行为）。

8.7、列出你知道的 HTTP 协议的状态码，说出表示什么意思？(2018-4-16-lxy)

通过状态码告诉客户端服务器的执行状态，以判断下一步该执行什么操作。

常见的状态机器码有：

100-199：表示服务器成功接收部分请求，要求客户端继续提交其余请求才能完成整个处理过程。

200-299：表示服务器成功接收请求并已完成处理过程，常用 200（OK 请求成功）。

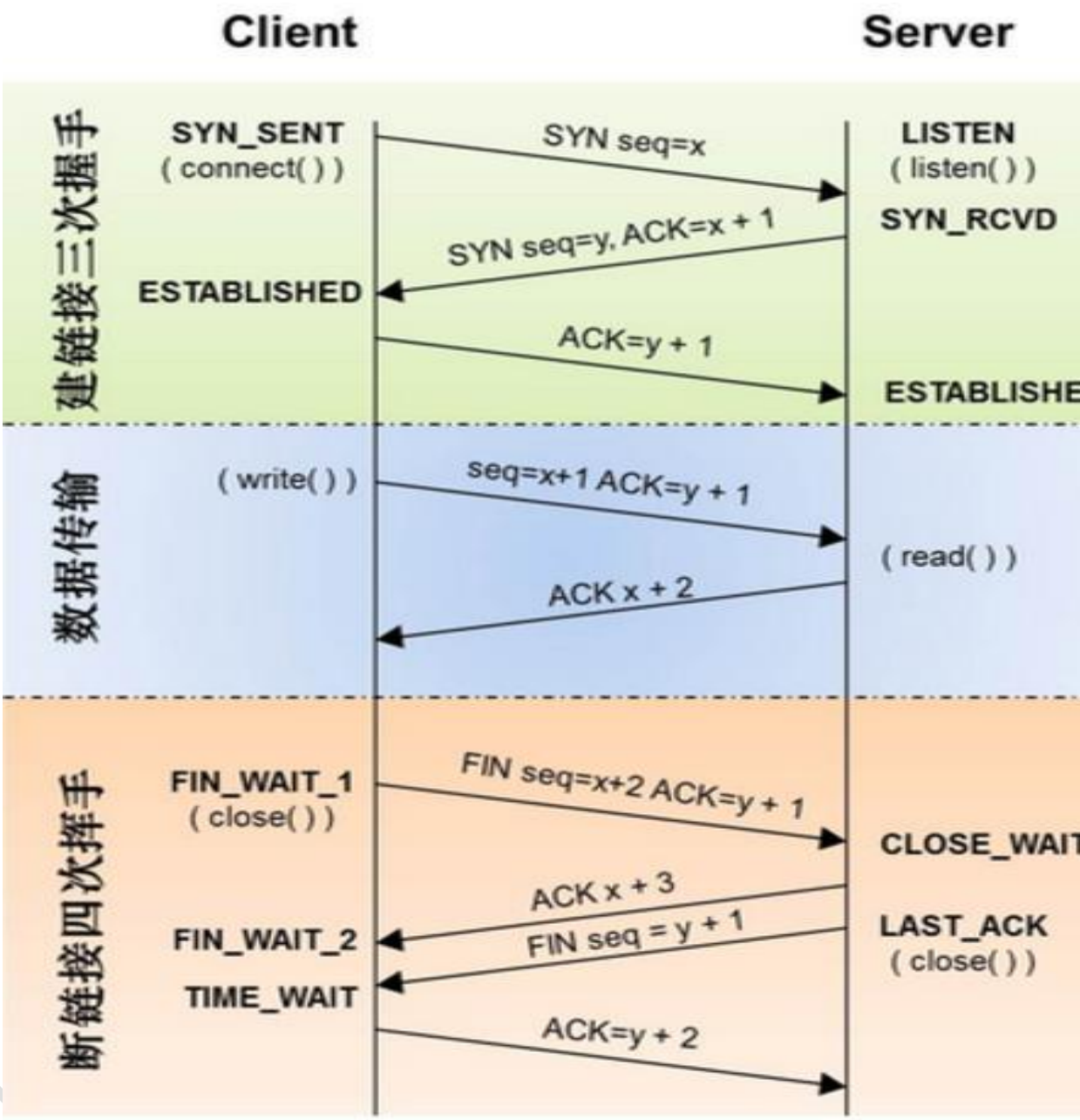
300-399：为完成请求，客户需要进一步细化请求。302（所有请求页面已经临时转移到新的 url）。

304、307（使用缓存资源）。

400-499：客户端请求有错误，常用 404（服务器无法找到被请求页面），403（服务器拒绝访问，权限不够）。

500-599：服务器端出现错误，常用 500（请求未完成，服务器遇到不可预知的情况）。

8.8、请简单说一下三次握手和四次挥手？ (2018-4-20-lxy)



三次握手过程：

1 首先客户端向服务端发送一个带有 SYN 标志，以及随机生成的序号 100(0 字节)的报文

2 服务端收到报文后返回一个报文(SYN200(0 字节)，ACk1001(字节+1))给客户端

3 客户端再次发送带有 ACk 标志 201(字节+)序号的报文给服务端

至此三次握手过程结束，客户端开始向服务端发送数据

1 客户端向服务端发起请求：我想给你通信，你准备好了么？

2 服务端收到请求后回应客户端：I'ok，你准备好了么

3 客户端礼貌的再次回一下客户端：准备就绪，咱们开始通信吧！

整个过程跟打电话的过程一模一样:1 喂，你在吗 2 在，我说的你听得到不 3 恩，听得到(接下来请开始你的表演)

补充：SYN：请求询问，ACk：回复，回应

四次挥手过程：

由于 TCP 连接是可以双向通信的（全双工），因此每个方向都必须单独进行关闭（这句话才是精辟，后面四个挥手过程都是其具体实现的语言描述）

四次挥手过程，客户端和服务端都可以先开始断开连接

1 客户端发送带有 fin 标识的报文给服务端，请求通信关闭

2 服务端收到信息后，回复 ACK 答应关闭客户端通信(连接)请求

3 服务端发送带有 fin 标识的报文给客户端，也请求关闭通信

4 客户端回应 ack 给服务端，答应关闭服务端的通信(连接)请求

8.9、说一下什么是 tcp 的 2MSL？(2018-4-20-lxy)

主动发送 fin 关闭的一方，在 4 次挥手最后一次要等待一段时间我们称这段时间为 2MSL

TIME_WAIT 状态的存在有两个理由：

1.让 4 次挥手关闭流程更加可靠

2.防止丢包后对后续新建的正常连接的传输造成破坏

8.10、为什么客户端在 TIME-WAIT 状态必须等待 2MSL 的时间？(2018-4-20-lxy)

1、为了保证客户端发送的最后一个 ACK 报文段能够达到服务器。 这个 ACK 报文段可能丢失，因而使处在 LAST-ACK 状态的服务器收不到确认。服务器会超时重传 FIN+ACK 报文段，客户端就能在 2MSL 时间内收到这个重传的 FIN+ACK 报文段，接着客户端重传一次确认，重启计时器。最好，客户端和服务器都正常进入到 CLOSED 状态。如果客户端在 TIME-WAIT 状态不等待一段时间，而是再发送完 ACK 报文后立即释放连接，那么就无法收到服务器重传的 FIN+ACK 报文段，因而也不会再发送一次确认报文。这样，服务器就无法按照正常步骤进入 CLOSED 状态。

2、防止已失效的连接请求报文段出现在本连接中。客户端在发送完最后一个 ACK 确认报文段后，再经过时间 2MSL，就可以使本连接持续的时间内所产生的所有报文段都从网络中消失。这样就可以使下一个新的连接中不会出现这种旧的连接请求报文段。

8.11、说说 HTTP 和 HTTPS 区别？ （2018-4-23-lxy）

HTTP 协议传输的数据都是未加密的，也就是明文的，因此使用 HTTP 协议传输隐私信息非常不安全，为了保证这些隐私数据能加密传输，于是网景公司设计了 SSL（Secure Sockets Layer）协议用于对 HTTP 协议传输的数据进行加密，从而就诞生了 HTTPS。简单来说，HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，要比 http 协议安全。

HTTPS 和 HTTP 的区别主要如下：

1、https 协议需要到 ca 申请证书，一般免费证书较少，因而需要一定费用。

2、http 是超文本传输协议，信息是明文传输，https 则是具有安全性的 ssl 加密传输协议。

3、http 和 https 使用的是完全不同的连接方式，用的端口也不一样，前者是 80，后者是 443。

4、http 的连接很简单，是无状态的；HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比 http 协议安全。

8.12、谈一下 HTTP 协议以及协议头部中表示数据类型的字段？ （2018-4-23-lxy）

HTTP 协议是 Hyper Text Transfer Protocol（超文本传输协议）的缩写，是用于从万维网（WWW:World Wide Web）服务器传输超文本到本地浏览器的传送协议。

HTTP 是一个基于 TCP/IP 通信协议来传递数据（HTML 文件， 图片文件， 查询结果等）。

HTTP 是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。它于 1990 年提出，经过几年的使用与发展，得到不断地完善和扩展。目前在 WWW 中使用的是 HTTP/1.0 的第六版，HTTP/1.1 的规范化工作正在进行之中，而且 HTTP-NG(Next Generation of HTTP) 的建议已经提出。

HTTP 协议工作于客户端-服务端架构为上。浏览器作为 HTTP 客户端通过 URL 向 HTTP 服务端即 WEB 服务器发送所有请求。Web 服务器根据接收到的请求后，向客户端发送响应信息。

表示数据类型字段： Content-Type

8.13、HTTP 请求方法都有什么？（2018-4-23-lxy）

根据 HTTP 标准，HTTP 请求可以使用多种请求方法。

HTTP1.0 定义了三种请求方法： GET， POST 和 HEAD 方法。

HTTP1.1 新增了五种请求方法：OPTIONS， PUT， DELETE， TRACE 和 CONNECT 方法。

1 GET 请求指定的页面信息，并返回实体主体。

2 HEAD 类似于 get 请求，只不过返回的响应中没有具体的内容，用于获取报头

3 POST 向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST 请求可能会导致新的资源的建立和/或已有资源的修改。

4 PUT 从客户端向服务器传送的数据取代指定的文档的内容。

5 DELETE 请求服务器删除指定的页面。

6 CONNECT HTTP/1.1 协议中预留给能够将连接改为管道方式的代理服务器。

7 OPTIONS 允许客户端查看服务器的性能。

8 TRACE 回显服务器收到的请求，主要用于测试或诊断。

8.14、使用 Socket 套接字需要传入哪些参数 ？（2018-4-23-lxy）

Address Family 和 Type，分别表示套接字应用场景和类型。

family 的值可以是 AF_UNIX(Unix 域，用于同一台机器上的进程间通讯)，也可以是 AF_INET（对于 IPV4 协议的 TCP 和 UDP），至于 type 参数，SOCK_STREAM（流套接字）或者 SOCK_DGRAM（数据报文套接字），SOCK_RAW（raw 套接字）。

8.15、HTTP 常见请求头？（2018-4-23-lxy）

1. Host (主机和端口号)
2. Connection (链接类型)
3. Upgrade-Insecure-Requests (升级为 HTTPS 请求)
4. User-Agent (浏览器名称)
5. Accept (传输文件类型)
6. Referer (页面跳转处)
7. Accept-Encoding（文件编解码格式）
8. Cookie （Cookie）
9. x-requested-with :XMLHttpRequest （是 Ajax 异步请求）

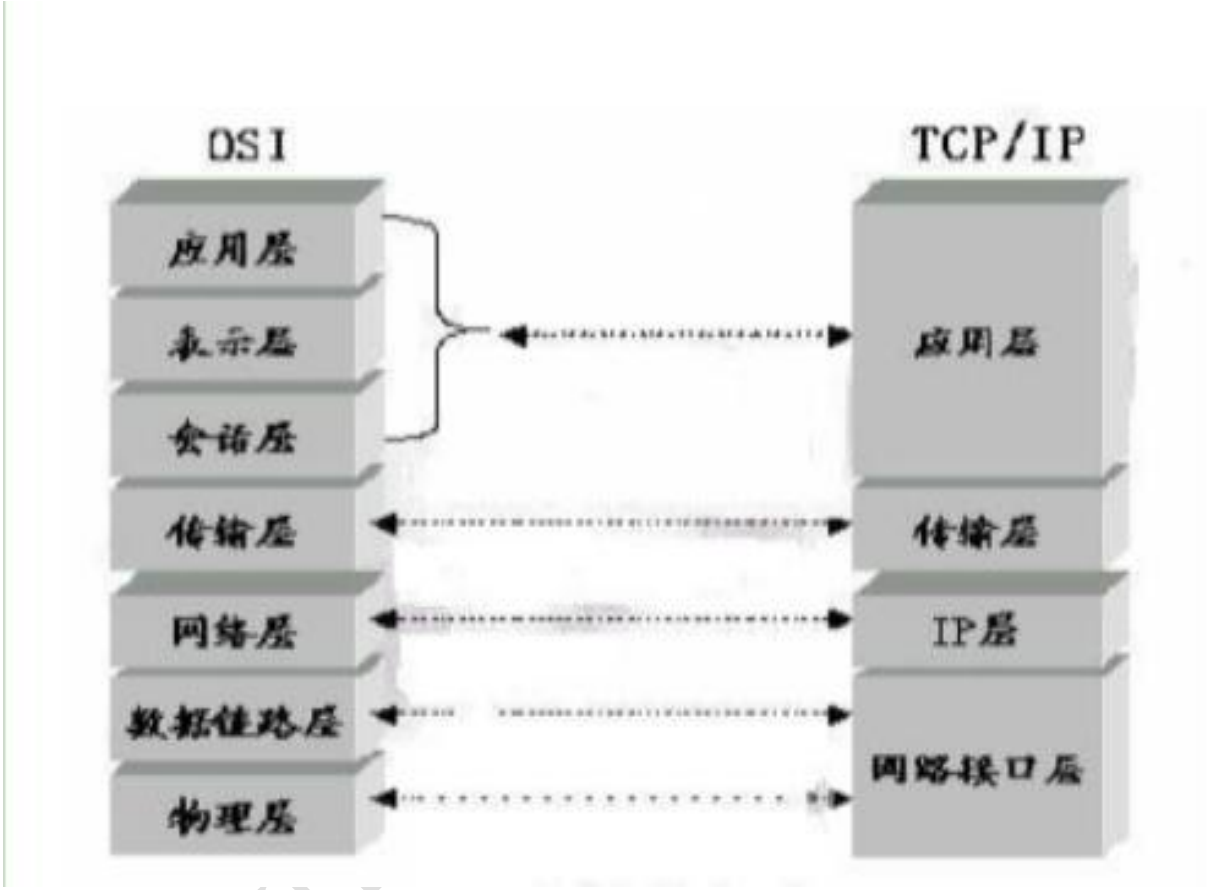
8.16、七层模型？（2018-4-23-lxy）

IP ， TCP/UDP ， HTTP ， RTSP ， FTP 分别在哪层？

IP： 网络层

TCP/UDP： 传输层

HTTP、RTSP、FTP： 应用层协议



8.17、url 的形式？（2018-4-23-lxy）

形式： scheme://host[:port#]/path/…/[?query-string][#anchor]
scheme：协议(例如：http， https， ftp)
host：服务器的 IP 地址或者域名
port：服务器的端口（如果是走协议默认端口，80 or 443）
path：访问资源的路径
query-string：参数，发送给 http 服务器的数据
anchor：锚（跳转到网页的指定锚点位置）
http://localhost:4000/file/part01/1.2.html

三、Web

1、Flask

1.1、对 Flask 蓝图(Blueprint)的理解？ (2018-4-14-lxy)

蓝图的定义

蓝图 /Blueprint 是 Flask 应用程序组件化的方法，可以在一个应用内或跨越多个项目共用蓝图。使用蓝图可以极大地简化大型应用的开发难度，也为 Flask 扩展 提供了一种在应用中注册服务的集中式机制。

蓝图的应用场景

把一个应用分解为一个蓝图的集合。这对大型应用是理想的。一个项目可以实例化一个应用对象，初始化几个扩展，并注册一集合的蓝图。

以 URL 前缀和/或子域名，在应用上注册一个蓝图。 URL 前缀/子域名中的参数即成为这个蓝图下的所有视图函数的共同的视图参数（默认情况下）。

在一个应用中用不同的 URL 规则多次注册一个蓝图。

通过蓝图提供模板过滤器、静态文件、模板和其它功能。一个蓝图不一定要实现应用或者视图函数。

初始化一个 Flask 扩展时，在这些情况中注册一个蓝图。

蓝图的缺点

不能在应用创建后撤销注册一个蓝图而不销毁整个应用对象。

使用蓝图的三个步骤

1.创建 一个蓝图对象

```
blue = Blueprint("blue", __name__)
```

2.在这个蓝图对象上进行操作 ，例如注册路由、指定静态文件夹、注册模板过滤器...

```
@blue.route('/')
def blue_index():
        return 'Welcome to my blueprint'
```

3.在应用对象上注册这个蓝图对象

```
app.register_blueprint(blue, url_prefix='/blue')
```

1.2、Flask 和 Django 路由映射的区别？(2018-4-19-lyf)

在 django 中，路由是浏览器访问服务器时，先访问的项目中的 url，再由项目中的 url 找到应用中 url，这些 url 是放在一个列表里，遵从从前往后匹配的规则。在 flask 中，路由是通过装饰器给每个视图函数提供的，而且根据请求方式的不同可以一个 url 用于不同的作用。

2、Django

2.1、django 中间件的使用？(2018-4-14-lxy)

Django 在中间件中预置了六个方法，这六个方法的区别在于不同的阶段执行，对输入或输出进行干预，方法如下：

1.初始化：无需任何参数，服务器响应第一个请求的时候调用一次，用于确定是否启用当前中间件。

```
def __init__():
    pass
```

2.处理请求前：在每个请求上调用，返回 None 或 HttpResponse 对象。

```
def process_request(request):
    pass
```

3.处理视图前：在每个请求上调用，返回 None 或 HttpResponse 对象。

```
def process_view(request, view_func, view_args, view_kwargs):
    pass
```

4.处理模板响应前：在每个请求上调用，返回实现了 render 方法的响应对象。

```
def process_template_response(request, response):
    pass
```

5.处理响应后：所有响应返回浏览器之前被调用，在每个请求上调用，返回 HttpResponse 对象。

```
def process_response(request, response):
    pass
```

6.异常处理：当视图抛出异常时调用，在每个请求上调用，返回一个 HttpResponse 对象。

```
def process_exception(request,exception):
    pass
```

2.2、谈一下你对 uWSGI 和 nginx 的理解？(2018-4-14-lxy)

1.uWSGI 是一个 Web 服务器，它实现了 WSGI 协议、uwsgi、http 等协议。Nginx 中 HttpUwsgiModule 的作用是与 uWSGI 服务器进行交换。WSGI 是一种 Web 服务器网关接口。它是一个 Web 服务器（如 nginx，uWSGI 等服务器）与 web 应用（如用 Flask 框架写的程序）通信的一种规范。

要注意 WSGI / uwsgi / uWSGI 这三个概念的区分。

WSGI 是一种通信协议。

uwsgi 是一种线路协议而不是通信协议，在此常用于在 uWSGI 服务器与其他网络服务器的数据通信。

uWSGI 是实现了 uwsgi 和 WSGI 两种协议的 Web 服务器。

nginx 是一个开源的高性能的 HTTP 服务器和反向代理：

1.作为 web 服务器，它处理静态文件和索引文件效果非常高；

2.它的设计非常注重效率，最大支持 5 万个并发连接，但只占用很少的内存空间；

3.稳定性高，配置简洁；

4.强大的反向代理和负载均衡功能，平衡集群中各个服务器的负载压力应用。

2.3、Python 中三大框架各自的应用场景？(2018-4-14-lxy)

django：主要是用来搞快速开发的，他的亮点就是快速开发，节约成本，正常的并发量不过 10000，如果要实现高并发的话，就要对 django 进行二次开发，比如把整个笨重的框架给拆掉，自己写 socket 实现 http 的通信，底层用纯 c，c++写提升效率，ORM 框架给干掉，自己编写封装与数据库交互的框架，因为啥呢，ORM 虽然面向对象来操作数据库，但是它的效率很低，使用外键来联系表与表之间的查询；

flask：轻量级，主要是用来写接口的一个框架，实现前后端分离，提升开发效率，Flask 本身相当于一个内核，其他几乎所有的功能都要用到扩展（邮件扩展 Flask-Mail，用户认证 Flask-Login），都需要用第三方的扩展来实现。比如可以用 Flask-extension 加入 ORM、窗体验证工具，文件上传、身份验证等。Flask 没有默认使用的数据库，你可以选择 MySQL，也可以用 NoSQL。

其 WSGI 工具箱采用 Werkzeug（路由模块），模板引擎则使用 Jinja2。这两个也是 Flask 框架的核心。Python 最出名的框架要数 Django，此外还有 Flask、Tornado 等框架。虽然 Flask 不是最出名的框架，但是 Flask 应该算是最灵活的框架之一，这也是 Flask 受到广大开发者喜爱的原因。

Tornado：Tornado 是一种 Web 服务器软件的开源版本。Tornado 和现在的主流 Web 服务器框架（包括大多数 Python 的框架）有着明显的区别：它是非阻塞式服务器，而且速度相当快。

得利于其非阻塞的方式和对 epoll 的运用，Tornado 每秒可以处理数以千计的连接，因此 Tornado 是实时 Web 服务的一个 理想框架。

2.4、有过部署经验？用的什么技术？可以满足多少压力？ (2018-4-14-lxy)

1.有部署经验，在阿里云服务器上部署的
2.技术有：nginx + uwsgi 的方式来部署 Django 项目
3.无标准答案（例：压力测试一两千）

2.5、Django 中哪里用到了线程?哪里用到了协程?哪里用到了进程？ (2018-4-14-lxy)

1.Django 中耗时的任务用一个进程或者线程来执行，比如发邮件，使用 celery。
2.部署 django 项目的时候，配置文件中设置了进程和协程的相关配置。

2.6、有用过 Django REST framework 吗？(2018-4-14-lxy)

Django REST framework 是一个强大而灵活的 Web API 工具。使用 RESTframework 的理由有：
Web browsable API 对开发者有极大的好处
包括 OAuth1a 和 OAuth2 的认证策略
支持 ORM 和非 ORM 数据资源的序列化
全程自定义开发——如果不想使用更加强大的功能，可仅仅使用常规的 function-based views
额外的文档和强大的社区支持

2.7、对 cookie 与 session 的了解？他们能单独用吗？ (2018-4-14-lxy)

Session 采用的是在服务器端保持状态的方案，而 Cookie 采用的是在客户端保持状态的方案。但是禁用 Cookie 就不能得到 Session。因为 Session 是用 Session ID 来确定当前对话所对应的服务器 Session，而 Session ID 是通过 Cookie 来传递的，禁用 Cookie 相当于失去了 SessionID，也就得不到 Session。

2.8、什么是 wsgi,uwsgi,uWSGI？ （2019-01-20-cz）

WSGI:
web 服务器网关接口,是一套协议。用于接收用户请求并将请求进行初次封装，然后将请求交给 web 框架。
实现 wsgi 协议的模块:wsgiref,本质上就是编写一 socket 服务端,用于接收用户请求(django)werkzeug,本质上就是编写一个 socket 服务端，用于接收用户请求(flask)。
uwsgi:
与 WSGI 一样是一种通信协议，它是 uWSGI 服务器的独占协议,用于定义传输信息的类型。
uWSGI:
是一个 web 服务器,实现了 WSGI 协议,uWSGI 协议,http 协议。

2.9、Django 、Flask、Tornado 的对比（2019-01-20-cz）

1、Django 走的是大而全的方向,开发效率高。它的 MTV 框架,自带的 ORM,admin 后台管理,自带的 sqlite 数据库和开发测试用的服务器，给开发者提高了超高的开发效率
重量级 web 框架，功能齐全，提供一站式解决的思路，能让开发者不用在选择上花费大量时间。
自带 ORM(Object-Relational Mapping 对象关系映射)和模板引擎，支持 jinja 等非官方模板引擎。
自带 ORM 使 Django 和关系型数据库耦合度高，如果要使用非关系型数据库，需要使用第三方库
自带数据库管理 app
成熟，稳定，开发效率高，相对于 Flask，Django 的整体封闭性比较好，适合做企业级网站的开发。

python web 框架的先驱，第三方库丰富/

2、Flask 是轻量级的框架,自由,灵活,可扩展性很强,核心基于 Werkzeug WSGI 工具和 jinja2 模板引擎

轻量级 web 框架，默认依赖两个外部库：jinja2 和 Werkzeug WSGI 工具

适用于做小型网站以及 web 服务的 API，开发大型网站无压力，但架构需要自己设计

与关系型数据库的结合不弱于 Django，而与非关系型数据库的结合远远优于 Django

3、Tornado 走的是少而精的方向,性能优越。它最出名的是异步非阻塞的设计方式

Tornado 的两大核心模块：

iostraem：对非阻塞式的 socket 进行简单的封装

ioloop：对 I/O 多路复用的封装，它实现了一个单例

2.10、CORS 和 CSRF 的区别？（2019-01-20-cz）

什么是 CORS?

CORS 是一个 W3C 标准，全称是"跨域资源共享"（Cross-origin resource sharing）。

它允许浏览器向跨源服务器，发出 XMLHttpRequest 请求，从而克服了 AJAX 只能同源使用的限制。

学习资料[跨域资源共享 CORS 详解] http://www.ruanyifeng.com/blog/2016/04/cors.html

什么是 CSRF?

CSRF 主流防御方式是在后端生成表单的时候生成一串随机 token ，内置到表单里成为一个字段，同时，将此串 token 置入 session 中。每次表单提交到后端时都会检查这两个值是否一致，以此来判断此次表单提交是否是可信的。提交过一次之后，如果这个页面没有生成 CSRF token ，那么 token 将会被清空，如果有新的需求，那么 token 会被更新。

攻击者可以伪造 POST 表单提交，但是他没有后端生成的内置于表单的 token，session 中有没有 token 都无济于事。

既然已经有了同源策略，CSRF 防护是否必要?

我之前错误的理解是「form 和 Ajax 发起的 POST 请求都受到 CORS 的限制，因此只要非幂等请求不是 GET, 就可以防范 CSRF」，而我今天才发现，原来 form 发起的 POST 请求并不受到 CORS 的限制，因此可以任意地使用其他域的 Cookie 向其他域发送 POST 请求，形成 CSRF 攻击。

2.11、Session、Cookie、JWT 的理解 （2019-01-20-cz）

为什么要使用会话管理

众所周知，HTTP 协议是一个无状态的协议，也就是说每个请求都是一个独立的请求，请求与请求之间并无关系。但在实际的应用场景，这种方式并不能满足我们的需求。举个大家都喜欢用的例子，把商品加入购物车，单独考虑这个请求，服务端并不知道这个商品是谁的，应该加入谁的购物车？因此这个请求的上下文环境实际上应该包含用户的相关信息，在每次用户发出请求时把这一小部分额外信息，也做为请求的一部分，这样服务端就可以根据上下文中的信息，针对具体的用户进行操作。所以这几种技术的出现都是对 HTTP 协议的一个补充。使得我们可以用 HTTP 协议+状态管理构建一个的面向用户的 WEB 应用。

Session 和 Cookie 的区别

这里我想先谈谈 session 与 cookies，因为这两个技术是做为开发最为常见的。那么 session 与 cookies 的区别是什么？个人认为 session 与 cookies 最核心区别在于额外信息由谁来维护。利用 cookies 来实现会话管理时，用户的相关信息或者其他我们想要保持在每个请求中的信息，都是放在 cookies 中，而 cookies 是由客户端来保存，每当客户端发出新请求时，就会稍带上 cookies，服务端会根据其中的信息进行操作。当利用 session 来进行会话管理时，客户端实际上只存了一个由服务端发送的 session_id，而由这个 session_id，可以在服务端还原出所需要的所有状态信息，从这里可以看出这部分信息是由服务端来维护的。

除此以外，session 与 cookies 都有一些自己的缺点：

cookies 的安全性不好，攻击者可以通过获取本地 cookies 进行欺骗或者利用 cookies 进行 CSRF 攻击。

使用 cookies 时，在多个域名下，会存在跨域问题。

session 在一定的时间里，需要存放在服务端，因此当拥有大量用户时，也会大幅度降低服务端的性能。当有多台机器时，如何共享 session 也会是一个问题，也就是说，用户第一个访问的时候是服务器 A，而第二个请求被转发给了服务器 B，那服务器 B 如何得知其状态。

实际上，session 与 cookies 是有联系的，比如，我们可以把 session_id 存放在 cookies 中的。

JWT 是如何工作的

首先用户发出登录请求，服务端根据用户的登录请求进行匹配，如果匹配成功，将相关的信息放入 payload 中，利用上述算法，加上服务端的密钥生成 token，这里需要注意的是 secret_key 很重要，如果这个泄露的话，客户端就可以随意篡改发送的额外信息，它是信息完整性的保证。生成 token 后服务端将其返回给客户端，客户端可以在下次请求时，将 token 一起交给服务端，一般来说我们可以将其放在 Authorization 首部中，这样也就可以避免跨域问题。接下来，服务端根据 token 进行信息解析，再根据用户信息作出相应的操作

JWT 的优点、缺点、及解决方案

考虑 JWT 的实现，上面所述的关于 session，cookies 的缺点都不复存在了，不易被攻击者利用，安全性提高。利用 Authorization 首部传输 token，无跨域问题。额外信息存储在客户端，服务端占用资源不多，也不存在 session 共享问题。感觉 JWT 优势很明显，但其仍然有一些缺点：

登录状态信息续签问题。比如设置 token 的有效期为一个小时，那么一个小时后，如果用户仍然在这个 web 应用上，这个时候当然不能指望用户再登录一次。目前可用的解决办法是在每次用户发出请求都返回一个新的 token，前端再用这个新的 token 来替代旧的，这样每一次请求都会刷新 token 的有效期。但是这样，需要频繁的生成 token。另外一种方案是判断还有多久这个 token 会过期，在 token 快要过期时，返回一个新的 token。下面是我在项目里的一个实现。

用户主动注销。JWT 并不支持用户主动退出登录，当然，可以在客户端删除这个 token，但在别处使用的 token 仍然可以正常访问。为了支持注销，我的解决方案是在注销时将该 token 加入黑名单。当用户发出请求后，如果该 token 在黑名单中，则阻止用户的后续操作，返回 Invalid token 错误。这个地方我再稍微补充一下，其实这里的黑名单操作也比较简单，把已经注销的 token 存入比如说一个 set 中，那么在每次进行 token 验证时，先检查在 set 中是否已经存在，如果已经存在的话，则视为 token 已经失效，直接返回未授权。这一部分在上面的授权代码中也可以看到，不过我是放到 redis 缓存中的。

总结

无论 session 还是 cookies 或是 jwt。目前情况是 jwt 仍然无法代替 session，cookies 也会有人用。它们各自有自己的优势和缺点，不能因为有一些缺点就否认技术的存在，缺点仍然可以采用一些技术手段来弥补，比如通过添加 csrf token 来阻止来自 CSRF 的攻击，比如利用 redis 集群来做 session 的存储和共享。技术只是工具，选择最适合你的才是最重要的。

2.12、简述 Django 请求生命周期 （2019-01-20-cz）

一般是用户通过浏览器向我们的服务器发起一个请求(request)，这个请求会去访问视图函数，（如果不涉及到数据调用，那么这个时候视图函数返回一个模板也就是一 个网页给用户），

视图函数调用模型，模型去数据库查找数据，然后逐级返回，视图函数把返回的数据填充到模板中空格中，最后返回网页给用户。

1.wsgi,请求封装后交给 web 框架 （Flask、Django）

2.中间件，对请求进行校验或在请求对象中添加其他相关数据，例如：csrf、request.session －

3.路由匹配 根据浏览器发送的不同 url 去匹配不同的视图函数

4.视图函数，在视图函数中进行业务逻辑的处理，可能涉及到：orm、templates => 渲染 －

5.中间件，对响应的数据进行处理。

6.wsgi,将响应的内容发送给浏览器。

2.13、什么是 wsgi,uwsgi,uWSGI？（2019-01-20-cz）

WSGI:

web 服务器网关接口,是一套协议。用于接收用户请求并将请求进行初次封装，然后将请求交给 web 框架。

实现 wsgi 协议的模块:wsgiref,本质上就是编写一 socket 服务端,用于接收用户请求(django)werkzeug,本质上就是编写一个 socket 服务端，用于接收用户请求(flask)。

uwsgi:

与 WSGI 一样是一种通信协议，它是 uWSGI 服务器的独占协议,用于定义传输信息的类型。

uWSGI:

是一个 web 服务器,实现了 WSGI 协议,uWSGI 协议,http 协议。

2.14、Django 、Flask、Tornado 的对比（2019-01-20-cz）

1、Django 走的是大而全的方向,开发效率高。它的 MTV 框架,自带的 ORM,admin 后台管理,自带的 sqlite 数据库和开发测试用的服务器，给开发者提高了超高的开发效率

重量级 web 框架，功能齐全，提供一站式解决的思路，能让开发者不用在选择上花费大量时间。

自带 ORM(Object-Relational Mapping 对象关系映射)和模板引擎，支持 jinja 等非官方模板引擎。

自带 ORM 使 Django 和关系型数据库耦合度高，如果要使用非关系型数据库，需要使用第三方库

自带数据库管理 app

成熟，稳定，开发效率高，相对于 Flask，Django 的整体封闭性比较好，适合做企业级网站的开发。

python web 框架的先驱，第三方库丰富/

2、Flask 是轻量级的框架,自由,灵活,可扩展性很强,核心基于 Werkzeug WSGI 工具和 jinja2 模板引擎

轻量级 web 框架，默认依赖两个外部库：jinja2 和 Werkzeug WSGI 工具

适用于做小型网站以及 web 服务的 API，开发大型网站无压力，但架构需要自己设计

与关系型数据库的结合不弱于 Django，而与非关系型数据库的结合远远优于 Django

3、Tornado 走的是少而精的方向,性能优越。它最出名的是异步非阻塞的设计方式

Tornado 的两大核心模块：

iostraem：对非阻塞式的 socket 进行简单的封装

ioloop：对 I/O 多路复用的封装，它实现了一个单例

2.15、用 django 的 restframework 完成 api 发送时间时区信息（2019-01-20-cz）

当前的问题是用 django 的 rest framework 模块做一个 get 请求的发送时间以及时区信息的 api

```
class getCurrentTime(APIView):
    def get(self, request):
        local_time = time.localtime()
        time_zone = settings.TIME_ZONE
        temp = {'localtime':local_time, 'timezone': time_zone}
        return Response(temp)
```

2.16、nginx,tomcat,apache 都是什么？（2019-01-20-cz）

Nginx (engine x) 是一个高性能的 HTTP 和反向代理服务器，也是一个 IMAP/POP3/SMTP 服务器。

Apache HTTP Server 是一个模块化的服务器，源于 NCSAhttpd 服务器

Tomcat 服务器是一个免费的开放源代码的 Web 应用服务器，属于轻量级应用服务器，是开发和调试 JSP 程序的首选。

2.17、请给出你熟悉关系数据库范式有那些，有什么作用？（2019-01-20-cz）

在进行数据库的设计时，所遵循的一些规范，只要按照设计规范进行设计，就能设计出没有数据冗余和数据维护异常的数据库结构

数据库的设计的规范有很多，通常来说我们在设计数据库时只要达到其中一些规范就可以了，这些规范又称之为数据库的三范式，一共有三条，也存在着其他的范式，我们只要做到满足前三个范式的要求，就能设计出符合我们的数据库了，我们也不能全部来按照范式的要求来做，还要考虑实际的业务使用情况，所以有时候也需要做一些违反范式的要求

1.数据库设计的第一范式(最基本)，基本上所有数据库的范式都是符合第一范式的，符合第一范式的表具有以下几个特点：

数据库表中的所有字段都只具有单一属性单一属性的列是由基本的数据类型(整型，浮点型，字符型等)所构成的设计出来的表都是简单的二维表

2.数据库设计的第二范式(是在第一范式的基础上设计的)，要求一个表中只具有一个业务主键，也就是说符合第二范式的表中不能存在非主键列对只对部分主键的依赖关系

3.数据库设计的第三范式，指每一个非主属性既不部分依赖于也不传递依赖于业务主键，也就是在第二范式的基础上消除了非主属性对主键的传递依赖 a

2.18、简述 QQ 登陆过程（2019-01-20-cz）

qq 登录，在我们的项目中分为了三个接口，

第一个接口是请求 qq 服务器返回一个 qq 登录的界面；

第二个接口是通过扫码或者账号登陆进行验证，qq 服务器会返回给浏览器一个 code 和 state，利用这个 code 通过本地服务器去向 qq 服务器

获取 access_token 并返回给本地服务器,凭介 access_token 再向 qq 服务器获取用户的 openid（openid 用户的唯一标识）

第三个接口是判断用户是否是第一次 qq 登录，如果不是的话直接登录返回的 jwt-token 给用户；不是的话，对没有绑定过本网站的用户，对 openid 进行加密生成 token 进行绑定

2.19、post 和 get 的区别？（2019-01-20-cz）

1、GET 是从服务器上获取数据，POST 是向服务器传送数据。

2、在客户端， GET 方式在通过 URL 提交数据，数据在 URL 中可以看到；POST 方式，数据放置在 HTML HEADER 内提交

3、对于 GET 方式，服务器端用 Request.QueryString 获取变量的值，对于 POST 方式，服务器端用 Request.Form 获取提交的数据。

4、GET 方式提交的数据最多只能有 1024 字节，而 POST 则没有此限制

安全性问题。正如在（2）中提到，使用 GET 的时候，参数会显示在地址栏上，而 POST 不会。所以，如果这些数据是中文数据而且是非敏感数据，那么使用 GET ；如果用户输入的数据不是中文字符而且包含敏感数据，那么还是使用 POST 为好

2.20、项目中日志的作用（2019-01-20-cz）

一、日志相关概念

1.日志是一种可以追踪某些软件运行时所发生事件的方法

2.软件开发人员可以向他们的代码中调用日志记录相关的方法来表明发生了某些事情

3.一个事件可以用一个可包含可选变量数据的消息来描述

4.此外，事件也有重要性的概念，这个重要性也可以被称为严重性级别（level）

二、日志的作用

1.通过 log 的分析，可以方便用户了解系统或软件、应用的运行情况；

2.如果你的应用 log 足够丰富，可以分析以往用户的操作行为、类型喜好、地域分布或其他更多信息；

3.如果一个应用的 log 同时也分了多个级别，那么可以很轻易地分析得到该应用的健康状况，及时发现问题并快速定位、解决问题，补救损失。

4.简单来讲就是我们通过记录和分析日志可以了解一个系统或软件程序运行情况是否正常，也可以在应用程序出现故障时快速定位问题。不仅在开发中，在运维中日志也很重要日志的作用可以简单

总结为以下几点：

1.程序调试

2.了解软件程序运行情况，是否正常

3.软件程序运行故障分析与问题定位

4.如果应用的日志信息足够详细和丰富，还可以用来做用户行为分析

四、爬虫

1.1、试列出至少三种目前流行的大型数据库

名称:_____、_____、_____,其中您最熟悉的是_____,从_____年开始使用（2018-4-1-ydy）

（考察对数据可的熟悉程度，同时考察你的工作年限注意和自己简历一致。Oracle，Mysql，SQLServer Mysql 、MongoDB 根据自己情况推荐 Mysql 、MongoDB）

1.2、列举您使用过的 Python 网络爬虫所用到的网络数据包? (2018-4-16-lxy)

requests、urllib、urllib2、httplib2。

1.3、列举您使用过的 Python 网络爬虫所用到的解析数据包？（2018-4-1-ydy）

BeautifulSoup、pyquery、Xpath、lxml。

1.4、爬取数据后使用哪个数据库存储数据的，为什么？（2018-4-1-ydy）

MongoDB 是使用比较多的数据库，这里以 MongoDB 为例，大家需要结合自己真实开发环境回答。

原因：1）与关系型数据库相比，MongoDB 的优点如下。

①弱一致性（最终一致），更能保证用户的访问速度

举例来说，在传统的关系型数据库中，一个 COUNT 类型的操作会锁定数据集，这样可以保证得到“当前”情况下的较精确值。这在某些情况下，例 如通过 ATM 查看账户信息的时候很重要，但对于 Wordnik 来说，数据是不断更新和增长的，这种“较精确”的保证几乎没有任何意义，反而会产生很大的延 迟。他们需要的是一个“大约”的数字以及更快的处理速度。

但某些情况下 MongoDB 会锁住数据库。如果此时正有数百个请求，则它们会堆积起来，造成许多问题。我们使用了下面的优化方式来避免锁定。

每次更新前，我们会先查询记录。查询操作会将对象放入内存，于是更新则会尽可能的迅速。在主/从部署方案中，从节点可以使用“-pretouch”参数运行，这也可以得到相同的效果。

使用多个 mongod 进程。我们根据访问模式将数据库拆分成多个进程。

②文档结构的存储方式，能够更便捷的获取数据。

对于一个层级式的数据结构来说，如果要将这样的数据使用扁平式的，表状的结构来保存数据，这无论是在查询还是获取数据时都十分困难。

③内置 GridFS，支持大容量的存储。

GridFS 是一个出色的分布式文件系统，可以支持海量的数据存储。内置了 GridFS 了 MongoDB，能够满足对大数据集的快速范围查询。

④内置 Sharding。

提供基于 Range 的 Auto Sharding 机制：一个 collection 可按照记录的范围，分成若干个段，切分到不同的 Shard 上。Shards 可以和复制结合，配合 Replica sets 能够实现 Sharding+fail-over，不同的 Shard 之间可以负载均衡。查询是对 客户端是透明的。客户端执行查询，统计，MapReduce 等操作，这些会被 MongoDB 自动路由到后端的数据节点。这让我们关注于自己的业务，适当的 时候可以无痛的升级。MongoDB 的 Sharding 设计能力较大可支持约 20 petabytes，足以支撑一般应用。

这可以保证 MongoDB 运行在便宜的 PC 服务器集群上。PC 集群扩充起来非常方便并且成本很低，避免了“sharding”操作的复杂性和成本。

⑤第三方支持丰富。(这是与其他的 NoSQL 相比，MongoDB 也具有的优势)

现在网络上的很多 NoSQL 开源数据库完全属于社区型的，没有官方支持，给使用者带来了很大的风险。而开源文档数据库 MongoDB 背后有商业公司 10gen 为其提供供商业培训和支持。

而且 MongoDB 社区非常活跃，很多开发框架都迅速提供了对 MongDB 的支持。不少知名大公司和网站也在生产环境中使用 MongoDB，越来越多的创新型企业转而使用 MongoDB 作为和 Django，RoR 来搭配的技术方案。

⑥性能优越

在使用场合下，千万级别的文档对象，近 10G 的数据，对有索引的 ID 的查询不会比 mysql 慢，而对非索引字段的查询，则是全面胜出。 mysql 实际无法胜任大数据量下任意字段的查询，而 mongodb 的查询性能实在让我惊讶。写入性能同样很令人满意，同样写入百万级别的数 据，mongodb 比我以前试用过的 couchdb 要快得多，基本 10 分钟以下可以解决。补上一句，观察过程中 mongodb 都远算不上是 CPU 杀手。

2)Mongodb 与 redis 相比较

①mongodb 文件存储是 BSON 格式类似 JSON，或自定义的二进制格式。

mongodb 与 redis 性能都很依赖内存的大小，mongodb 有丰富的数据表达、索引；最类似于关系数据库，支持丰富的查询语言，redis 数据丰富，较少的 IO ，这方面 mongodb 优势明显。

②mongodb 不支持事务，靠客户端自身保证，redis 支持事物，比较弱，仅能保证事物中的操作按顺序执行，这方面 redis 优于 mongodb。

③mongodb 对海量数据的访问效率提升，redis 较小数据量的性能及运算,这方面 mongodb 性能优于 redis .monbgodb 有 mapredurce 功能，提供数据分析，redis 没有 ，这方面 mongodb 优于 redis 。

1.5、你用过的爬虫框架或者模块有哪些？优缺点？(2018-4-16-lxy)

Python 自带：urllib、urllib2

第三方：requests

框架： Scrapy

urllib 和 urllib2 模块都做与请求 URL 相关的操作，但他们提供不同的功能。

urllib2：urllib2.urlopen 可以接受一个 Request 对象或者 url，（在接受 Request 对象时候，并以此可以来设置一个 URL 的 headers），urllib.urlopen 只接收一个 url。

urllib 有 urlencode,urllib2 没有，因此总是 urllib、urllib2 常会一起使用的原因*

scrapy 是封装起来的框架，他包含了下载器，解析器，日志及异常处理，基于多线程，twisted 的方式处理，对于固定单个网站的爬取开发，有优势，但是对于多网站爬取 100 个网站，并发及分布式处理方面，不够灵活，不便调整与括展。

request 是一个 HTTP 库， 它只是用来，进行请求，对于 HTTP 请求，他是一个强大的库，下载，解析全部自己处理，灵活性更高，高并发与分布式部署也非常灵活，对于功能可以更好实现

Scrapy 优点：

scrapy 是异步的；

采取可读性更强的 xpath 代替正则；

强大的统计和 log 系统；

同时在不同的 url 上爬行；

支持 shell 方式，方便独立调试；

写 middleware,方便写一些统一的过滤器；

通过管道的方式存入数据库；

Scrapy 缺点：

基于 python 的爬虫框架，扩展性比较差；

基于 twisted 框架，运行中的 exception 是不会干掉 reactor，并且异步框架出错后是不会停掉其他任务的，数据出错后难以察觉。

1.6、写爬虫是用多进程好？还是多线程好？(2018-4-16-lxy)

IO 密集型代码(文件处理、网络爬虫等)，多线程能够有效提升效率(单线程下有 IO 操作会进行 IO 等待，造成不必要的时间浪费，而开启多线程能在线程 A 等待时，自动切换到线程 B，可以不浪费 CPU 的资源，从而能提升程序执行效率)。在实际的数据采集过程中，既考虑网速和响应的问题，也需要考虑自身机器的硬件情况，来设置多进程或多线程。

1.7、常见的反爬虫和应对方法？(2018-4-16-lxy)

通过 Headers 反爬虫：

从用户请求的 Headers 反爬虫是最常见的反爬虫策略。很多网站都会对 Headers 的 User-Agent 进行检测，还有一部分网站会对 Referer 进行检测（一些资源网站的防盗链就是检测 Referer）。如果遇到了这类反爬虫机制，可以直接在爬虫中添加 Headers，将浏览器的 User-Agent 复制到爬虫的 Headers 中；或者将 Referer 值修改为目标网站域名。对于检测 Headers 的反爬虫，在爬虫中修改或者添加 Headers 就能很好的绕过。

基于用户行为反爬虫：

还有一部分网站是通过检测用户行为，例如同一 IP 短时间内多次访问同一页面，或者同一账户短时间内多次进行相同操作。大多数网站都是前一种情况，对于这种情况，使用 IP 代理就可以解决。可以专门写一个爬虫，爬取网上公开的代理 ip，检测后全部保存起来。这样的代理 ip 爬虫经常会用到，最好自己准备一个。有了大量代理 ip 后可以每请求几次更换一个 ip，这在 requests 或者 urllib2 中很容易做到，这样就能很容易的绕过第一种反爬虫。对于第二种情况，可以在每次请求后随机间隔几秒再进行下一次请求。有些有逻辑漏洞的网站，可以通过请求几次，退出登录，重新登录，继续请求来绕过同一账号短时间内不能多次进行相同请求的限制。

动态页面的反爬虫：

上述的几种情况大多都是出现在静态页面，还有一部分网站，我们需要爬取的数据是通过 ajax 请求得到，或者通过 JavaScript 生成的。首先用 Fiddler 对网络请求进行分析。如果能够找到 ajax 请求，也能分析出具体的参数和响应的具体含义，我们就能采用上面的方法，直接利用 requests 或者 urllib2 模拟 ajax 请求，对响应的 json 进行分析得到需要的数据。能够直接模拟 ajax 请求获取数据固然是极好的，但是有些网站把 ajax 请求的所有参数全部加密了。我们根本没办法构造自己所需要的数据的请求。这种情况下就用 selenium+phantomJS，调用浏览器内核，并利用 phantomJS 执行 js 来模拟人为操作以及触发页面中的 js 脚本。从填写表单到点击按钮再到滚动页面，全部都可以模拟，不考虑具体的请求和响应过程，只是完完整整的把人浏览页面获取数据的过程模拟一遍。用这套框架几乎能绕过大多数的反爬虫，因为它不是在伪装成浏览器来获取数据（上述的通过添加 Headers 一定程度上就是为了伪装成浏览器），它本身就是浏览器，phantomJS 就是一个没有界面的浏览器，只是操控这个浏览器的不是人。利用 selenium+phantomJS 能干很多事情，例如识别点触式（12306）或者滑动式的验证码，对页面表单进行暴力破解等。

1.8、解析网页的解析器使用最多的是哪几个? (2018-4-16-lxy)

lxml，html5lib，html.parser,lxml-xml，正则表达式。

1.9、需要登录的网页，如何解决同时限制 ip，cookie,session

（其中有一些是动态生成的）在不使用动态爬取的情况下？(2018-4-16-lxy)

解决限制 IP 可以使用代理 IP 地址池、服务器；不适用动态爬取的情况下可以使用反编译 JS 文件获取相应的文件，或者换用其他平台（比如手机端）看看是否可以获取相应的 json 文件。

1.10、验证码的解决? (2018-4-16-lxy)

图形验证码：干扰、杂色不是特别多的图片可以使用开源库 Tesseract 进行识别，太过复杂的需要借助第三方打码平台。点击和拖动滑块验证码可以借助 selenium、无图形界面浏览器（chromedirver 或者 phantomjs）和 pillow 包来模拟人的点击和滑动操作，pillow 可以根据色差识别需要滑动的位置。

1.11、使用最多的数据库，对他们的理解？(2018-4-16-lxy)

MySQL 数据库：开源免费的关系型数据库，需要实现创建数据库、数据表和表的字段，表与表之间可以进行关联（一对多、多对多），是持久化存储。

Mongodb 数据库：是非关系型数据库，数据库的三元素是，数据库、集合、文档，可以进行持久化存储，也可作为内存数据库，存储数据不需要事先设定格式，数据以键值对的形式存储。

redis 数据库：非关系型数据库，使用前可以不用设置格式，以键值对的方式保存，文件格式相对自由，主要用与缓存数据库，也可以进行持久化存储。

1.12、编写过哪些爬虫中间件？(2018-4-23-lyf)

user-agent、代理池等。

怎么获取加密的数据？(2018-4-23-lyf)

1. Web 端加密可尝试移动端（app）
2. 解析加密格式，看能否破解
3. 反爬手段层出不穷，js 加密较多，只能具体问题具体分析，js 逆向

1.13、“极验”滑动验证码如何破解？(2018-4-23-lyf)

1.selenium 控制鼠标实现，速度太机械化，成功率比较低

2.计算缺口的偏移量（推荐博客：

http://blog.csdn.net/paololiu/article/details/52514504?%3E）

3.“极验”滑动验证码需要具体网站具体分析，一般牵扯算法乃至深度学习相关知识。

1.14、爬虫多久爬一次，爬下来的数据是怎么存储？（2018-4-20-xhq）

参考回答：多久爬一次这个问题要根据公司的要求去处理，不一定是每天都爬。

Mongo 建立唯一索引键（id）可以做数据重复 前提是数据量不大 2 台电脑几百万的情况 数据库需要做分片 （数据库要设计合理）。

例：租房的网站数据量每天大概是几十万条 ，每周固定爬取。

1.15、cookie 过期的处理问题？（2018-4-20-xhq）

因为 cookie 存在过期的现象，一个很好的处理方法就是做一个异常类，如果有异常的话 cookie 抛出异常类在执行程序。

1.16、动态加载又对及时性要求很高怎么处理？（2018-4-20-xhq）

Selenium+Phantomjs

尽量不使用 sleep 而使用 WebDriverWait

1.17、HTTPS 有什么优点和缺点？（2018-4-20-xhq）

优点：

1、使用 HTTPS 协议可认证用户和服务器，确保数据发送到正确的客户机和服务器；

2、HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，要比 http 协议安全，可防止数据在传输过程中不被窃取、改变，确保数据的完整性。

3、HTTPS 是现行架构下最安全的解决方案，虽然不是绝对安全，但它大幅增加了中间人攻击的成本

缺点：

1.HTTPS 协议的加密范围也比较有限，在黑客攻击、拒绝服务攻击、服务器劫持等方面几乎起不到什么作用

2.HTTPS 协议还会影响缓存，增加数据开销和功耗，甚至已有安全措施也会受到影响也会因此而受到影响。

3.SSL 证书需要钱。功能越强大的证书费用越高。个人网站、小网站没有必要一般不会用。

4.HTTPS 连接服务器端资源占用高很多，握手阶段比较费时对网站的相应速度有负面影响。

5.HTTPS 连接缓存不如 HTTP 高效。

1.18、HTTPS 是如何实现安全传输数据的？（2018-4-20-xhq）

HTTPS 其实就是在 HTTP 跟 TCP 中间加多了一层加密层 TLS/SSL。SSL 是个加密套件，负责对 HTTP 的数据进行加密。TLS 是 SSL 的升级版。现在提到 HTTPS，加密套件基本指的是 TLS。原先是应用层将数据直接给到 TCP 进行传输，现在改成应用层将数据给到 TLS/SSL，将数据加密后，再给到 TCP 进行传输。

1.19、TTL，MSL，RTT 各是什么？（2018-4-20-xhq）

MSL：报文最大生存时间”，他是任何报文在网络上存在的最长时间，超过这个时间报文将被丢弃。

TTL：TTL 是 time to live 的缩写，中文可以译为“生存时间”，这个生存时间是由源主机设置初始值但不是存的具体时间，而是存储了一个 ip 数据报可以经过的最大路由数，每经过一个处理他的路由器此值就减 1，当此值为 0 则数据报将被丢弃，同时发送 ICMP 报文通知源主机。RFC 793 中规定 MSL 为 2 分钟，实际应用中常用的是 30 秒，1 分钟和 2 分钟等。TTL 与 MSL 是有关系的但不是简单的相等的关系，MSL 要大于等于 TTL。

RTT：RTT 是客户到服务器往返所花时间（round-trip time，简称 RTT），TCP 含有动态估算 RTT 的算法。TCP 还持续估算一个给定连接的 RTT，这是因为 RTT 受网络传输拥塞程序的变化而变化。

1.20、谈一谈你对 Selenium 和 PhantomJS 了解（2018-4-20-xhq）

Selenium 是一个 Web 的自动化测试工具，可以根据我们的指令，让浏览器自动加载页面，获取需要的数据，甚至页面截屏，或者判断网站上某些动作是否发生。Selenium 自己不带浏览器，不支持浏览器的功能，它需要与第三方浏览器结合在一起才能使用。但是我们有时候需要让它内嵌在代码中运行，所以我们可以用一个叫 PhantomJS 的工具代替真实的浏览器。Selenium 库里有个叫 WebDriver 的 API。WebDriver 有点儿像可以加载网站的浏览器，但是它也可以像 BeautifulSoup 或者其他 Selector 对象一样用来查找页面元素，与页面上的元素进行交互（发送文本、点击等），以及执行其他动作来运行网络爬虫。

PhantomJS 是一个基于 Webkit 的“无界面”(headless)浏览器，它会把网站加载到内存并执行页面上的 JavaScript，因为不会展示图形界面，所以运行起来比完整的浏览器要高效。相比传统的 Chrome 或 Firefox 浏览器等，资源消耗会更少。

如果我们把 Selenium 和 PhantomJS 结合在一起，就可以运行一个非常强大的网络爬虫了，这个爬虫可以处理 JavaScrip、Cookie、headers，以及任何我们真实用户需要做的事情。

主程序退出后，selenium 不保证 phantomJS 也成功退出，最好手动关闭 phantomJS 进程。（有可能会导致多个 phantomJS 进程运行，占用内存）。

WebDriverWait 虽然可能会减少延时，但是目前存在 bug（各种报错），这种情况可以采用 sleep。

phantomJS 爬数据比较慢，可以选择多线程。如果运行的时候发现有的可以运行，有的不能，可以尝试将 phantomJS 改成 Chrome。

1.21、平常怎么使用代理的 ？（2018-4-20-xhq）

1. 自己维护代理池
2. 付费购买(目前市场上有很多 ip 代理商,可自行百度了解,建议看看他们的接口文档(API&SDK))

IP 存放在哪里？怎么维护 IP？对于封了多个 ip 的，怎么判定 IP 没被封？

1.22、存放在数据库(redis、mysql 等)。（2018-4-20-xhq）

维护多个代理网站：

一般代理的存活时间往往在十几分钟左右，定时任务，加上代理 IP 去访问网页，验证其是否可用，如果返回状态为 200，表示这个代理是可以使用的。

假如每天爬取量在 5、6 万条数据，一般开几个线程，每个线程 ip 需要加锁限定吗？（2018-4-20-xhq）

5、6 万条数据相对来说数据量比较小，线程数量不做强制要求(做除法得一个合理值即可）

多线程使用代理，应保证不在同时一刻使用一个代理 IP

1.23、怎么监控爬虫的状态？（2018-4-20-xhq）

1. 使用 python 的 STMP 包将爬虫的状态信心发送到指定的邮箱
2. Scrapyd、pyspider

1.24、描述下 scrapy 框架运行的机制？(2018-4-16-lxy)

从 start_urls 里获取第一批 url 并发送请求，请求由引擎交给调度器入请求队列，获取完毕后，调度器将请求队列里的请求交给下载器去获取请求对应的响应资源，并将响应交给自己编写的解析方法做提取处理：

如果提取出需要的数据，则交给管道文件处理；

如果提取出 url，则继续执行之前的步骤（发送 url 请求，并由引擎将请求交给调度器入队列...)，直到请求队列里没有请求，程序结束。

（要会画流程图，并能把流程讲出来。）

1.25、谈谈你对 Scrapy 的理解？(2018-4-23-lyf)

scrapy 是一个为了爬取网站数据，提取结构性数据而编写的应用框架，我们只需要实现少量代码，就能够快速的抓取到数据内容。Scrapy 使用了 Twisted (其主要对手是 Tornado)异步网络框架来处理网络通讯，可以加快我们的下载速度，不用自己去实现异步框架，并且包含了各种中间件接口，可以灵活的完成各种需求。

scrapy 框架的工作流程：

1.首先 Spiders（爬虫）将需要发送请求的 url(requests)经 ScrapyEngine（引擎）交给 Scheduler（调度器）。

2.Scheduler（排序，入队）处理后，经 ScrapyEngine，DownloaderMiddlewares(可选，主要有 User_Agent，Proxy 代理)交给 Downloader。

3.Downloader 向互联网发送请求，并接收下载响应（response）。将响应（response）经 ScrapyEngine，SpiderMiddlewares(可选)交给 Spiders。

4.Spiders 处理 response，提取数据并将数据经 ScrapyEngine 交给 ItemPipeline 保存（可以是本地，可以是数据库）。提取 url 重新经 ScrapyEngine 交给 Scheduler 进行下一个循环。直到无 Url 请求程序停止结束。

1.26、怎么样让 scrapy 框架发送一个 post 请求（具体写出来）(2018-4-23-lyf)

使用 FormRequest

```
class mySpider(scrapy.Spider):
        # start_urls = ["http://www.taobao.com/"]
        def start_requests(self):
        url = 'http://http://www.taobao.com//login'
        # FormRequest 是 Scrapy 发送 POST 请求的方法
            yield scrapy.FormRequest(
                                url = url,
formdata = {"email" : "xxx",  "password" : "xxxxx"},
                                callback = self.parse_page
                                                )
        def parse_page(self,  response):
            # do something
```

1.27、怎么监控爬虫的状态 ？(2018-4-23-lyf)

1. 使用 python 的 STMP 包将爬虫的状态信心发送到指定的邮箱
2. Scrapyd、pyspider

1.28、怎么判断网站是否更新？(2018-4-23-lyf)

使用 MD5 数字签名：

每次下载网页时，把服务器返回的数据流 ResponseStream 先放在内存缓冲区，然后对 ResponseStream 生成 MD5 数字签名 S1，下次下载同样生成签名 S2，比较 S2 和 S1，如果相同，则页面没有跟新，否则网页就有跟新

1.29、图片、视频爬取怎么绕过防盗连接(2018-4-23-lyf)

或者说怎么获取正确的链接地址？

自定义 Referer(建议自行 Google 相关知识)。

1.30、你爬出来的数据量大概有多大？大概多长时间爬一次？(2018-4-23-lyf)

无标准答案，根据自己爬取网站回答即可（几百万，几千万，亿级）。

1.31、用什么数据库存爬下来的数据？部署是你做的吗？怎么部署？(2018-4-23-lyf)

常用 MongoDB、mysql、redis 等

部署过程稍复杂，建议自行谷歌 scrapyd 部署爬虫项目。

1.32、增量爬取 (2018-4-23-lyf)

增量爬取即保存上一次状态，本次抓取时与上次比对，如果不在上次的状态中，便视为增量，保存下来。对于 scrapy 来说，上一次的状态是抓取的特征数据和上次爬取的 request 队列（url 列表），request 队列可以通过 request 队列可以通过 scrapy.core.scheduler 的 pending_requests 成员得到，在爬虫启动时导入上次爬取的特征数据，并且用上次 request 队列的数据作为 start url 进行爬取，不在上一次状态中的数据便保存。选用 BloomFilter 原因：对爬虫爬取数据的保存有多种形式，可以是数据库，可以是磁盘文件等，不管是数据库，还是磁盘文件，进行扫描和存储都有很大的时间和空间上的开销，为了从时间和空间上提升性能，故选用 BloomFilter 作为上一次爬取数据的保存。保存的特征数据可以是数据的某几项，即监控这几项数据，一旦这几项数据有变化，便视为增量持久化下来，根据增量的规则可以对保存的状态数据进行约束。比如：可以选网页更新的时间，索引次数或是网页的实际内容，cookie 的更新等。

爬虫向数据库存数据开始和结束都会发一条消息，是 scrapy 哪个模块实现的？（2018-4-20-xhq）

Scrapy 使用信号来通知事情发生，因此答案是 signals 模块。

1.33、爬取下来的数据如何去重，说一下 scrapy 的具体的算法依据。（2018-4-20-xhq）

1.通过 MD5 生成电子指纹来判断页面是否改变

2.nutch 去重。nutch 中 digest 是对采集的每一个网页内容的 32 位哈希值，如果两个网页内容完全一样，它们的 digest 值肯定会一样。数据量不大时，可以直接放在内存里面进行去重，python 可以使用 set() 进行去重。当去重数据需要持久化时可以使用 redis 的 set 数据结构。当数据量再大一点时，可以用不同的加密算法先将长字符串压缩成 16/32/40 个字符，再使用上面两种方法去重。当数据量达到亿（甚至十亿、百亿）数量级时，内存有限，必须用"位"来去重，才能够满足需求。Bloomfilter 就是将去重对象映射到几个内存"位"，通过几个位的 0/1 值来判断一个对象是否已经存在。然而 Bloomfilter 运行在一台机器的内存上，不方便持久化（机器 down 掉就什么都没啦），也不方便分布式爬虫的统一去重。如果可以在 Redis 上申请内存进行 Bloomfilter，以上两个问题就都能解决了。simhash 最牛逼的一点就是将一个文档，最后转换成一个 64 位的字节，暂且称之为特征字，然后判断重复只需要判断他们的特征字的距离是不是<n（根据经验这个 n 一般取值为 3），就可以判断两个文档是否相似。可见 scrapy_redis 是利用 set 数据结构来去重的，去重的对象是 request 的 fingerprint（其实就是用 hashlib.sha1()对 request 对象的某些字段信息进行压缩）。其实 fp 就是 request 对象加密压缩后的一个字符串（40 个字符，0~f）。

1.34、Scrapy 的优缺点？（2018-4-20-xhq）

优点：scrapy 是异步的、采取可读性更强的 xpath 代替正则、强大的统计和 log 系统、同时在不同的 url 上爬行、支持 shell 方式，方便独立调试、写 middleware,方便写一些统一的过滤器、通过管道的方式存入数据库

缺点：基于 python 的爬虫框架，扩展性比较差、基于 twisted 框架，运行中的 exception 是不会干掉 reactor（反应器），并且异步框架出错后是不会停掉其他任务的，数据出错后难以察觉。

1.35、怎么设置爬取深度？（2018-4-20-xhq）

通过在 settings.py 中设置 depth_limit 的值可以限制爬取深度，这个深度是与 start_urls 中定义 url 的相对值。也就是相对 url 的深度。若定义 url 为 http://www.domz.com/game/,depth_limit=1 那么限制爬取的只能是此 url 下一级的网页。深度大于设置值的将被忽视。

1.36、scrapy 和 scrapy-redis 有什么区别？为什么选择 redis 数据库？(2018-4-16-lxy)

scrapy 是一个 Python 爬虫框架，爬取效率极高，具有高度定制性，但是不支持分布式。而 scrapy-redis 一套基于 redis 数据库、运行在 scrapy 框架之上的组件，可以让 scrapy 支持分布式策略，Slaver 端共享 Master 端 redis 数据库里的 item 队列、请求队列和请求指纹集合。

为什么选择 redis 数据库，因为 redis 支持主从同步，而且数据都是缓存在内存中的，所以基于 redis 的分布式爬虫，对请求和数据的高频读取效率非常高。

1.37、分布式爬虫主要解决什么问题？(2018-4-16-lxy)

Ip，带宽，cpu，io

1.38、什么是分布式存储？(2018-4-23-lyf)

传统定义：分布式存储系统是大量 PC 服务器通过 Internet 互联，对外提供一个整体的服务。
分布式存储系统具有以下的几个特性：

可扩展 ：分布式存储系统可以扩展到几百台甚至几千台这样的一个集群规模，系统的整体性能线性增长。

低成本 ：分布式存储系统的自动容错、自动负载均衡的特性，允许分布式存储系统可以构建在低成本的服务器上。另外，线性的扩展能力也使得增加、减少服务器的成本低，实现分布式存储系统的自动运维。

高性能 ：无论是针对单台服务器，还是针对整个分布式的存储集群，都要求分布式存储系统具备高性能。

易用 ：分布式存储系统需要对外提供方便易用的接口，另外，也需要具备完善的监控、运维工具，并且可以方便的与其他的系统进行集成。布式存储系统的挑战主要在于数据和状态信息的持久化，要求在自动迁移、自动容错和并发读写的过程中，保证数据的一致性。

容错：如何可以快速检测到服务器故障，并自动的将在故障服务器上的数据进行迁移
负载均衡：新增的服务器如何在集群中保障负载均衡？数据迁移过程中如何保障不影响现有的服务。
事务与并发控制：如何实现分布式事务。
易用性：如何设计对外接口，使得设计的系统易于使用。

1.39、你所知道的分布式爬虫方案有哪些？(2018-4-23-lyf)

三种分布式爬虫策略：

1.Slaver 端从 Master 端拿任务（Request/url/ID）进行数据抓取，在抓取数据的同时也生成新任务，并将任务抛给 Master。Master 端只有一个 Redis 数据库，负责对 Slaver 提交的任务进行去重、加入待爬队列。

优点： scrapy-redis 默认使用的就是这种策略，我们实现起来很简单，因为任务调度等工作 scrapy-redis 都已经帮我们做好了，我们只需要继承 RedisSpider、指定 redis_key 就行了。

缺点： scrapy-redis 调度的任务是 Request 对象，里面信息量比较大（不仅包含 url，还有 callback 函数、headers 等信息），导致的结果就是会降低爬虫速度、而且会占用 Redis 大量的存储空间。当然我们可以重写方法实现调度 url 或者用户 ID。

2.Master 端跑一个程序去生成任务（Request/url/ID）。Master 端负责的是生产任务，并把任务去重、加入到待爬队列。Slaver 只管从 Master 端拿任务去爬。

优点： 将生成任务和抓取数据分开，分工明确，减少了 Master 和 Slaver 之间的数据交流；Master 端生成任务还有一个好处就是：可以很方便地重写判重策略（当数据量大时优化判重的性能和速度还是很重要的）。

缺点： 像 QQ 或者新浪微博这种网站，发送一个请求，返回的内容里面可能包含几十个待爬的用户 ID，即几十个新爬虫任务。但有些网站一个请求只能得到一两个新任务，并且返回的内容里也包含爬虫要抓取的目标信息，如果将生成任务和抓取任务分开反而会降低爬虫抓取效率。毕竟带宽也是爬虫的一个瓶颈问题，我们要秉着发送尽量少的请求为原则，同时也是为了减轻网站服务器的压力，要做一只有道德的 Crawler。所以，视情况而定。

3.Master 中只有一个集合，它只有查询的作用。Slaver 在遇到新任务时询问 Master 此任务是否已爬，如果未爬则加入 Slaver 自己的待爬队列中，Master 把此任务记为已爬。它和策略一比较像，但明显比策略一简单。策略一的简单是因为有 scrapy-redis 实现了 scheduler 中间件，它并不适用于非 scrapy 框架的爬虫。

优点： 实现简单，非 scrapy 框架的爬虫也适用。Master 端压力比较小，Master 与 Slaver 的数据交流也不大。

缺点："健壮性"不够，需要另外定时保存待爬队列以实现"断点续爬"功能。各 Slaver 的待爬任务不通用。

如果把 Slaver 比作工人，把 Master 比作工头。策略一就是工人遇到新任务都上报给工头，需要干活的时候就去工头那里领任务；策略二就是工头去找新任务，工人只管从工头那里领任务干活；策略三就是工人遇到新任务时询问工头此任务是否有人做了，没有的话工人就将此任务加到自己的"行程表"。除了

1.40、scrapy-redis，有做过其他的分布式爬虫吗？(2018-4-23-lyf)

Celery、gearman 等，参考其他分布式爬虫策略。

五、数据库

1、MySQL

1.1、主键 超键 候选键 外键(2019-01-19 whb)

主键：数据库表中对储存数据对象予以唯一和完整标识的数据列或属性的组合。一个数据列只能有一个主键，且主键的取值不能缺失，即不能为空值（Null）。

超键：在关系中能唯一标识元组的属性集称为关系模式的超键。一个属性可以为作为一个超键，多个属性组合在一起也可以作为一个超键。超键包含候选键和主键。

候选键：是最小超键，即没有冗余元素的超键。

外键：在一个表中存在的另一个表的主键称此表的外键。

1.2、视图的作用，视图可以更改么？(2019-01-19 whb)

视图是虚拟的表，与包含数据的表不一样，视图只包含使用时动态检索数据的查询；不包含任何列或数据。使用视图可以简化复杂的 sql 操作，隐藏具体的细节，保护数据；视图创建后，可以使用与表相同的方式利用它们。

视图不能被索引，也不能有关联的触发器或默认值，如果视图本身内有 order by 则对视图再次 order by 将被覆盖。

创建视图：create view XXX as XXXXXXXXXXXXX;

对于某些视图比如未使用联结子查询分组聚集函数 Distinct Union 等，是可以对其更新的，对视图的更新将对基表进行更新；但是视图主要用于简化检索，保护数据，并不用于更新，而且大部分视图都不可以更新。

1.3、drop,delete 与 truncate 的区别(2019-01-19 whb)

drop 直接删掉表 truncate 删除表中数据，再插入时自增长 id 又从 1 开始 delete 删除表中数据，可以加 where 字句。

（1） DELETE 语句执行删除的过程是每次从表中删除一行，并且同时将该行的删除操作作为事务记录在日志中保存以便进行进行回滚操作。TRUNCATE TABLE 则一次性地从表中删除所有的数据并不把单独的删除操作记录记入日志保存，删除行是不能恢复的。并且在删除的过程中不会激活与表有关的删除触发器。执行速度快。

（2） 表和索引所占空间。当表被 TRUNCATE 后，这个表和索引所占用的空间会恢复到初始大小，而 DELETE 操作不会减少表或索引所占用的空间。drop 语句将表所占用的空间全释放掉。

（3） 一般而言，drop > truncate > delete

（4） 应用范围。TRUNCATE 只能对 TABLE；DELETE 可以是 table 和 view

（5） TRUNCATE 和 DELETE 只删除数据，而 DROP 则删除整个表（结构和数据）。

（6） truncate 与不带 where 的 delete ：只删除数据，而不删除表的结构（定义）drop 语句将删除表的结构被依赖的约束（constrain),触发器（trigger)索引（index);依赖于该表的存储过程/函数将被保留，但其状态会变为：invalid。

（7） delete 语句为 DML（data maintain Language),这个操作会被放到 rollback segment 中,事务提交后才生效。如果有相应的 tigger,执行的时候将被触发。

（8） truncate、drop 是 DLL（data define language),操作立即生效，原数据不放到 rollback segment 中，不能回滚

（9） 在没有备份情况下，谨慎使用 drop 与 truncate。要删除部分数据行采用 delete 且注意结合 where 来约束影响范围。回滚段要足够大。要删除表用 drop;若想保留表而将表中数据删除，如果于事务无关，用 truncate 即可实现。如果和事务有关，或老师想触发 trigger,还是用 delete。

（10） Truncate table 表名 速度快,而且效率高,因为:

truncate table 在功能上与不带 WHERE 子句的 DELETE 语句相同：二者均删除表中的全部行。但 TRUNCATE TABLE 比 DELETE 速度快，且使用的系统和事务日志资源少。DELETE 语句每次删除一行，并在事务日志中为所删除的每行记录一项。TRUNCATE TABLE 通过释放存储表数据所用的数据页来删除数据，并且只在事务日志中记录页的释放。

（11） TRUNCATE TABLE 删除表中的所有行，但表结构及其列、约束、索引等保持不变。新行标识所用的计数值重置为该列的种子。如果想保留标识计数值，请改用 DELETE。如果要删除表定义及其数据，请使用 DROP TABLE 语句。

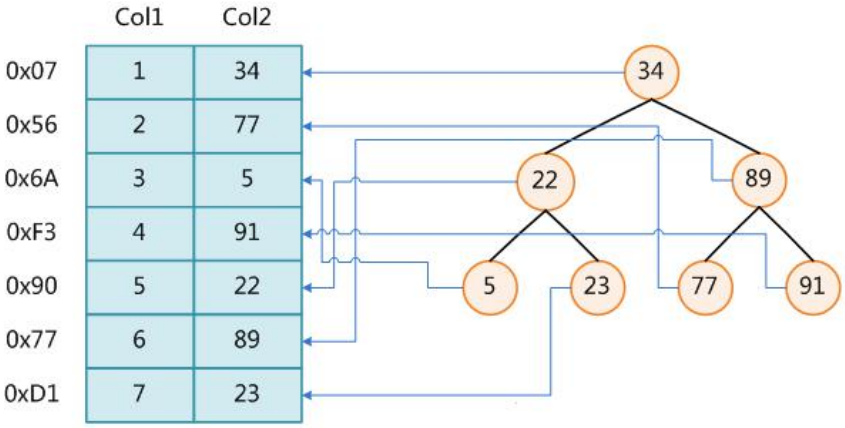
（12） 对于由 FOREIGN KEY 约束引用的表，不能使用 TRUNCATE TABLE，而应使用不带 WHERE 子句的 DELETE 语句。由于 TRUNCATE TABLE 不记录在日志中，所以它不能激活触发器。

1.4、索引的工作原理及其种类(2019-01-19 whb)

数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。索引的实现通常使用 B 树及其变种 B+树。

在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

为表设置索引要付出代价的：一是增加了数据库的存储空间，二是在插入和修改数据时要花费较多的时间(因为索引也要随之变动)。



图展示了一种可能的索引方式。左边是数据表，一共有两列七条记录，最左边的是数据记录的物理地址（注意逻辑上相邻的记录在磁盘上也并不是一定物理相邻的）。为了加快 Col2 的查找，可以维护一个右边所示的二叉查找树，每个节点分别包含索引键值和一个指向对应数据记录物理地址的指针，这样就可以运用二叉查找在 O(log2n)的复杂度内获取到相应数据。

创建索引可以大大提高系统的性能。

第一，通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。

第二，可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

第三，可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

第四，在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

第五，通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

也许会有人要问：增加索引有如此多的优点，为什么不对表中的每一个列创建一个索引呢？因为，增加索引也有许多不利的方面。

第一，创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

第二，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

第三，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

索引是建立在数据库表中的某些列的上面。在创建索引的时候，应该考虑在哪些列上可以创建索引，在哪些列上不能创建索引。一般来说，应该在这些列上创建索引：在经常需要搜索的列上，可以加快搜索的速度；在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；在经常使用在 WHERE 子句中的列上面创建索引，加快条件的判断速度。

同样，对于有些列不应该创建索引。一般来说，不应该创建索引的的这些列具有下列特点：

第一，对于那些在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。

第二，对于那些只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。

第三，对于那些定义为 text, image 和 bit 数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少。

第四，当修改性能远远大于检索性能时，不应该创建索引。这是因为，修改性能和检索性能是互相矛盾的。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

根据数据库的功能，可以在数据库设计器中创建三种索引：唯一索引、主键索引和聚集索引。

唯一索引

唯一索引是不允许其中任何两行具有相同索引值的索引。

当现有数据中存在重复的键值时，大多数数据库不允许将新创建的唯一索引与表一起保存。数据库还可能防止添加将在表中创建重复键值的新数据。例如，如果在 employee 表中职员的姓(lname)上创建了唯一索引，则任何两个员工都不能同姓。 主键索引 数据库表经常有一列或列组合，其值唯一标识表中的每一行。该列称为表的主键。 在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问。聚集索引 在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。一个表只能包含一个聚集索引。

如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。与非聚集索引相比，聚集索引通常提供更快的数据访问速度。

局部性原理与磁盘预读

由于存储介质的特性，磁盘本身存取就比主存慢很多，再加上机械运动耗费，磁盘的存取速度往往是主存的几百分分之一，因此为了提高效率，要尽量减少磁盘 I/O。为了达到这个目的，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。这样做的理论依据是计算机科学中著名的局部性原理：当一个数据被用到时，其附近的数据也通常会马上被使用。程序运行期间所需要的数据通常比较集中。

由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），因此对于具有局部性的程序来说，预读可以提高 I/O 效率。

预读的长度一般为页（page）的整倍数。页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，页得大小通常为4k），主存和磁盘以页为单位交换数据。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返回，程序继续运行。

B-/+Tree 索引的性能分析

到这里终于可以分析 B-/+Tree 索引的性能了。

上文说过一般使用磁盘 I/O 次数评价索引结构的优劣。先从 B-Tree 分析，根据 B-Tree 的定义，可知检索一次最多需要访问 h 个节点。数据库系统的设计者巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次 I/O 就可以完全载入。为了达到这个目的，在实际实现 B-Tree 还需要使用如下技巧：

每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个 node 只需一次 I/O。

B-Tree 中一次检索最多需要 h-1 次 I/O（根节点常驻内存），渐进复杂度为 O(h)=O(logdN)。一般实际应用中，出度 d 是非常大的数字，通常超过 100，因此 h 非常小（通常不超过 3）。

而红黑树这种结构，h 明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，所以红黑树的 I/O 渐进复杂度也为 O(h)，效率明显比 B-Tree 差很多。

综上所述，用 B-Tree 作为索引结构效率是非常高的。

1.5、连接的种类(2019-01-19 whb)

查询分析器中执行：

```
-建表 table1,table2:
create table table1(id int,name varchar(10))
create table table2(id int,score int)
insert into table1 select 1,'lee'
insert into table1 select 2,'zhang'
insert into table1 select 4,'wang'
insert into table2 select 1,90
insert into table2 select 2,100
insert into table2 select 3,70
```

如表

```
    table1 | table2 |
    id name |id score |
1 lee |1 90|
2 zhang| 2 100|
4 wang| 3 70|
```

以下均在查询分析器中执行

一、外连接

1.概念：包括左向外联接、右向外联接或完整外部联接

2.左连接：left join 或 left outer join

(1)左向外联接的结果集包括 LEFT OUTER 子句中指定的左表的所有行，而不仅仅是联接列所匹配的行。如果左表的某行在右表中没有匹配行，则在相关联的结果集行中右表的所有选择列表列均为空值(null)。

(2)sql 语句

```
select * from table1 left join table2 on table1.id=table2.id
```

```
————-结果————-
idnameidscore
    1lee190
2zhang2100
4wangNULLNULL
```

注释：包含 table1 的所有子句，根据指定条件返回 table2 相应的字段，不符合的以 null 显示

3.右连接：right join 或 right outer join

(1)右向外联接是左向外联接的反向联接。将返回右表的所有行。如果右表的某行在左表中没有匹配行，则将为左表返回空值。

(2)sql 语句

```
select * from table1 right join table2 on table1.id=table2.id
————-结果————-
idnameidscore
    1lee190
2zhang2100
NULLNULL370
```

注释：包含 table2 的所有子句，根据指定条件返回 table1 相应的字段，不符合的以 null 显示

4.完整外部联接:full join 或 full outer join

(1)完整外部联接返回左表和右表中的所有行。当某行在另一个表中没有匹配行时，则另一个表的选择列表列包含空值。如果表之间有匹配行，则整个结果集行包含基表的数据值。

(2)sql 语句

```
select * from table1 full join table2 on table1.id=table2.id
————-结果————-
idnameidscore
    1lee190
2zhang2100
4wangNULLNULL
NULLNULL370
```

注释：返回左右连接的和（见上左、右连接）

二、内连接

1.概念：内联接是用比较运算符比较要联接列的值的联接

2.内连接：join 或 inner join

3.sql 语句

```
select * from table1 join table2 on table1.id=table2.id
————-结果————-
idnameidscore
    1lee190
2zhang2100
```

注释：只返回符合条件的 table1 和 table2 的列

4.等价（与下列执行效果相同）

```
A:select a.,b. from table1 a,table2 b where a.id=b.id
```

B:select * from table1 cross join table2 where table1.id=table2.id (注：cross join 后加条件只能用 where,不能用 on)

三、交叉连接(完全)

1.概念：没有 WHERE 子句的交叉联接将产生联接所涉及的表的笛卡尔积。第一个表的行数乘以第二个表的行数等于笛卡尔积结果集的大小。（table1 和 table2 交叉连接产生 3*3=9 条记录）

2.交叉连接：cross join （不带条件 where…）

3.sql 语句

```
select * from table1 cross join table2
————————-结果————————-
idnameidscore
    1lee190
2zhang190
4wang190
1lee2100
2zhang2100
4wang2100
1lee370
2zhang370
4wang370
```

注释：返回 3*3=9 条记录，即笛卡尔积

4.等价（与下列执行效果相同）

A:select * from table1,table2

1.6、数据库优化的思路(2019-01-19 whb)

1.SQL 语句优化

1）应尽量避免在 where 子句中使用!=或操作符，否则将引擎放弃使用索引而进行全表扫描。

2）应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：

select id from t where num is null

可以在 num 上设置默认值 0，确保表中 num 列没有 null 值，然后这样查询：

select id from t where num=0

3）很多时候用 exists 代替 in 是一个好的选择

4）用 Where 子句替换 HAVING 子句 因为 HAVING 只会在检索出所有记录之后才对结果集进行过滤

2.索引优化

看上文索引

3.数据库结构优化

1）范式优化： 比如消除冗余（节省空间。。） 2）反范式优化：比如适当加冗余等（减少 join） 3）拆分表： 分区将数据在物理上分隔开，不同分区的数据可以制定保存在处于不同磁盘上的数据文件里。这样，当对这个表进行查询时，只需要在表分区中进行扫描，而不必进行全表扫描，明显缩短了查询时间，另外处于不同磁盘的分区也将对这个表的数据传输分散在不同的磁盘 I/O，一个精心设置的分区可以将数据传输对磁盘 I/O 竞争均匀地分散开。对数据量大的时时表可采取此方法。可按月自动建表分区。

4）拆分其实又分垂直拆分和水平拆分： 案例： 简单购物系统暂设涉及如下表： 1.产品表（数据量 10w，稳定） 2.订单表（数据量 200w，且有增长趋势） 3.用户表 （数据量 100w，且有增长趋势） 以 mysql 为例讲述下水平拆分和垂直拆分，mysql 能容忍的数量级在百万静态数据可以到千万 垂直拆分：解决问题：表与表之间的 io 竞争 不解决问题：单表中数据量增长出现的压力 方案： 把产品表和用户表放到一个 server 上 订单表单独放到一个 server 上 水平拆分： 解决问题：单表中数据量增长出现的压力 不解决问

题：表与表之间的 io 争夺
方案： 用户表通过性别拆分为男用户表和女用户表 订单表通过已完成和完成中拆分为已完成订单和未完成订单 产品表 未完成订单放一个 server 上 已完成订单表盒男用户表放一个 server 上 女用户表放一个 server 上(女的爱购物 哈哈)

4.服务器硬件优化
这个么多花钱咯！

1.7、存储过程与触发器的区别(2019-01-19 whb)

触发器与存储过程非常相似，触发器也是 SQL 语句集，两者唯一的区别是触发器不能用 EXECUTE 语句调用，而是在用户执行 Transact-SQL 语句时自动触发（激活）执行。触发器是在一个修改了指定表中的数据时执行的存储过程。通常通过创建触发器来强制实现不同表中的逻辑相关数据的引用完整性和一致性。由于用户不能绕过触发器，所以可以用它来强制实施复杂的业务规则，以确保数据的完整性。触发器不同于存储过程，触发器主要是通过事件执行触发而被执行的，而存储过程可以通过存储过程名称名字而直接调用。当对某一表进行诸如 UPDATE、INSERT、DELETE 这些操作时，SQLSERVER 就会自动执行触发器所定义的 SQL 语句，从而确保对数据的处理必须符合这些 SQL 语句所定义的规则。

1.8、悲观锁和乐观锁是什么？(2019-01-19 whb)

悲观锁(Pessimistic Lock), 顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会 block 直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

乐观锁(Optimistic Lock), 顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于 write_condition 机制的其实都是提供的乐观锁。

悲观锁

格式 SELECT…FOR UPDATE

例：select * from account where name="Erica" for update

这条 sql 语句锁定了 account 表中所有符合检索条件（ name=“Erica”）的记录。 本次事务提交之前（事务提交时会释放事务过程中的锁），外界无法修改这些记录。

格式 SELECT…FOR UPDATE NOWAIT

该关键字的含义是“不用等待，立即返回”，如果当前请求的资源被其他会话锁定时，会发生阻塞，nowait 可以避免这一阻塞。

乐观锁

乐观锁，大多是基于数据版本 （ Version ）记录机制实现。何谓数据版本？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个“version”字段来实现。

读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如果提交的数据版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。

总结：

两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果经常产生冲突，上层应用会不断的进行 retry，这样反倒是降低了性能，所以这种情况下用悲观锁就比较合适。

1.9、你常用的 mysql 引擎有哪些?各引擎间有什么区别? (2019-01-19 whb)

主要 MyISAM 与 InnoDB 两个引擎，其主要区别如下：

InnoDB 支持事务，MyISAM 不支持，这一点是非常之重要。事务是一种高级的处理方式，如在一些列增删改中只要哪个出错还可以回滚还原，而 MyISAM 就不可以了；

MyISAM 适合查询以及插入为主的应用，InnoDB 适合频繁修改以及涉及到安全性较高的应用；

InnoDB 支持外键，MyISAM 不支持；

MyISAM 是默认引擎，InnoDB 需要指定；

InnoDB 不支持 FULLTEXT 类型的索引；

InnoDB 中不保存表的行数，如 select count() from table 时，InnoDB；需要扫描一遍整个表来计算有多少行，但是 MyISAM 只要简单的读出保存好的行数即可。注意的是，当 count()语句包含 where 条件时 MyISAM 也需要扫描整个表；

对于自增长的字段，InnoDB 中必须包含只有该字段的索引，但是在 MyISAM 表中可以和其他字段一起建立联合索引；清空整个表时，InnoDB 是一行一行的删除，效率非常慢。MyISAM 则会重建表；

InnoDB 支持行锁（某些情况下还是锁整表，如 update table set a=1 where user like '%lee%'

2、Redis

2.1、Redis 宕机怎么解决? (2019-01-19 whb)

宕机：服务器停止服务

如果只有一台 redis，肯定会造成数据丢失，无法挽救

多台 redis 或者是 redis 集群 ，宕机则需要分为在主从模式下区分来看：

slave 从 redis 宕机

配置主从复制的时候才配置从的 redis，从的会从主的 redis 中读取主的 redis 的操作日志，求达到主从复制。

1）在 Redis 中从库重新启动后会自动加入到主从架构中，自动完成同步数据；

2）如果从数据库实现了持久化，可以直接连接到主的上面，只要实现增量备份（宕机到重新连接过程中，主的数据库发生数据操作，复制到从数据库），重新连接到主从架构中会实现增量同步。

Master 宕机

假如主从都没数据持久化，此时千万不要立马重启服务，否则可能会造成数据丢失，正确的操作如下：

在 slave 数据上执行 SLAVEOF ON ONE,来断开主从关系并把 slave 升级为主库

此时重新启动主数据库，执行 SLAVEOF,把它设置为从库，连接到主的 redis 上面做主从复制，自动备份数据。

以上过程很容易配置错误，可以使用 redis 提供的哨兵机制来简化上面的操作。简单的方法：redis 的哨兵（sentinel）的功能。

2.2、redis 和 mecached 的区别，以及使用场景(2019-01-19 whb)

区别

1、Redis 和 Memcache 都是将数据存放在内存中，都是内存数据库。不过 memcache 还可用于缓存其他东西，例如图片、视频等等；

2、Redis 不仅仅支持简单的 k/v 类型的数据，同时还提供 list，set，hash 等数据结构的存储；

3、虚拟内存 – Redis 当物理内存用完时，可以将一些很久没用到的 value 交换到磁盘；

4、过期策略 – memcache 在 set 时就指定，例如 set key1 0 0 8,即永不过期。Redis 可以通过例如 expire 设定，例如 expire name 10；

5、分布式－设定 memcache 集群，利用 magent 做一主多从;redis 可以做一主多从。都可以一主一从；

6、存储数据安全－memcache 挂掉后，数据没了；redis 可以定期保存到磁盘（持久化）；

7、灾难恢复－memcache 挂掉后，数据不可恢复; redis 数据丢失后可以通过 aof 恢复；

8、Redis 支持数据的备份，即 master-slave 模式的数据备份；

9、应用场景不一样：Redis 出来作为 NoSQL 数据库使用外，还能用做消息队列、数据堆栈和数据缓存等；Memcached 适合于缓存 SQL 语句、数据集、用户临时性数据、延迟查询数据和 session 等。

使用场景

1、如果有持久方面的需求或对数据类型和处理有要求的应该选择 redis。

2、如果简单的 key/value 存储应该选择 memcached。

2.3、Redis 集群方案该怎么做?都有哪些方案?(2019-01-19 whb)

1.codis。

目前用的最多的集群方案，基本和 twemproxy 一致的效果，但它支持在 节点数量改变情况下，旧节点数据可恢复到新 hash 节点。

2.redis cluster3.0 自带的集群，特点在于他的分布式算法不是一致性 hash，而是 hash 槽的概念，以及自身支持节点设置从节点。具体看官方文档介绍。

3.在业务代码层实现，起几个毫无关联的 redis 实例，在代码层，对 key 进行 hash 计算，然后去对应的 redis 实例操作数据。 这种方式对 hash 层代码要求比较高，考虑部分包括，节点失效后的替代算法方案，数据震荡后的自动脚本恢复，实例的监控，等等

2.4、Redis 回收进程是如何工作的(2019-01-19 whb)

一个客户端运行了新的命令，添加了新的数据。

Redi 检查内存使用情况，如果大于 maxmemory 的限制，则根据设定好的策略进行回收。

一个新的命令被执行，等等。

所以我们不断地穿越内存限制的边界，通过不断达到边界然后不断地回收回到边界以下。

如果一个命令的结果导致大量内存被使用（例如很大的集合的交集保存到一个新的键），不用多久内存限制就会被这个内存使用量超越

3、MongoDB

3.1、MongoDB 中对多条记录做更新操作命令是什么？(2019-01-19 whb)

db.table_name.update(where,setNew,issert,multi);

参数解释:

where:类似于 sql 中的 update 语句 where 后边的查询条件

setNew:类似于 sql 中 update 语句中 set 后边的部分，也就是你要更新的部分

upsert:如果要更新的那条记录没有找到，是否插入一条新纪录，默认为 false 不插入，true 为插入

multi :是否更新满足条件的多条的记录，false：只更新第一条，true:更新多条，默认为 false

示例:

>db.sms_user.update({"state":0},{$set:{"addUser":"","nickName":"","area":""}},f alse,true);

相当于： update sms_user set addUser="",nickName="",area="" where state=0;

更新表 sms_user 中所有 state=0 的记录

3.2、MongoDB 数据在什么时候才会拓展到多个分片（shard）里(2019-01-19 whb)?

MongoDB 分片是基于区域(range)的。所以一个集合(collection)中的所有的对象都被存放到一个块(chunk)中。只有当存在多余一个块的时候，才会有多个分片获取数据的选项。现在，每个默认块的大小是64Mb，所以你需要至少 64 Mb 空间才可以实施一个迁移。

六、测试

1、编写测试计划的目的是(2019-01-18-yl)

1.使测试工作顺利进行
2.使项目参与人员沟通更舒畅
3.使测试工作更加系统化

2、测试人员在软件开发过程中的任务是什么(2018-4-23-zcz)

1.寻找 Bug；
2.避免软件开发过程中的缺陷；
3.衡量软件的品质；
4.关注用户的需求。
总的目标是：确保软件的质量。

3、一条软件 Bug 记录都包含了哪些内容？(2018-4-23-zcz)

Bug 记录最基本应包含：编号、Bug 所属模块、Bug 描述、Bug 级别、发现日期、发现人、修改日期、修改人、修改方法、回归结果等等；要有效的发现 Bug 需参考需求以及详细设计等前期文档设计出高效的测试用例，然后严格执行测试用例，对发现的问题要充分确认肯定，然后再向外发布如此才能提高提交 Bug 的质量。

4、简述黑盒测试和白盒测试的优缺点(2018-4-23-zcz)

黑盒测试的优点有：
1）比较简单，不需要了解程序内部的代码及实现；
2）与软件的内部实现无关；
3）从用户角度出发，能很容易的知道用户会用到哪些功能，会遇到哪些问题；
4）基于软件开发文档，所以也能知道软件实现了文档中的哪些功能；
5）在做软件自动化测试时较为方便。
黑盒测试的缺点有：
1）不可能覆盖所有的代码，覆盖率较低，大概只能达到总代码量的 30%；
2）自动化测试的复用性较低。
白盒测试的优点有：
帮助软件测试人员增大代码的覆盖率，提高代码的质量，发现代码中隐藏的问题。
白盒测试的缺点有：
1）程序运行会有很多不同的路径，不可能测试所有的运行路径；
2）测试基于代码，只能测试开发人员做的对不对，而不能知道设计的正确与否，可能会漏掉一些功能需求；
3）系统庞大时，测试开销会非常大。

5、请列出你所知道的软件测试种类，至少 5 项。(2018-4-23-zcz)

单元测试，集成测试，系统测试，验收测试。
系统测试包含：功能测试，性能测试，压力测试，兼容性测试，健壮性测试，冒烟测试，文档测试。

6、Alpha 测试与 Beta 测试的区别是什么？(2018-4-23-zcz)

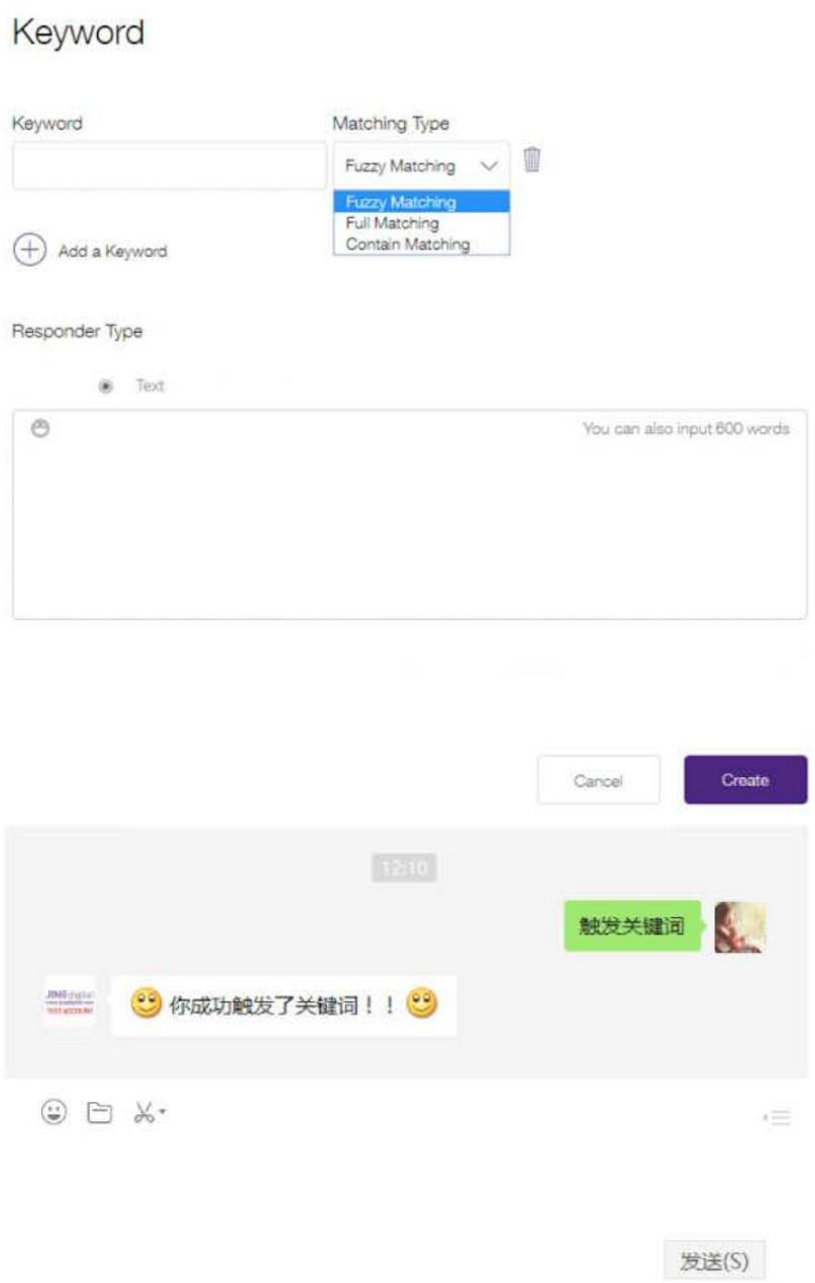
Alpha 主要是模拟用户的操作和用户的环境。
Beta 主要验证测试，准备进入发布阶段，Beta 测试是一种验收测试。

7、举例说明什么是 Bug？一个 bug report 应包含什么关键字？(2018-4-23-zcz)

比如聊天中，点击发送按钮后，无法发送消息。
标题，模块，严重程度，bug 类型，版本号，可否重现，描述，附件，日志等等。

8、对关键词触发模块进行测试(2019-1-24-yl)

如下图所示，当用户发送你设置的 keyword，
Fuzzy matching：你设置的 keyword 包含用户发送的 message，则会触发成功
Full matching：用户发送的 message 完全匹配你设置的 keyword，则会触发成功
Contain matching：用户发送的 message 包含你设置的 keyword，则会触发成功
系统会回复一条消息，这条消息由你自定义实现（见图中 responder type 下的编辑框）
对于这样一个模块，请写出你的测试点。


Keyword
Keyword
Matching Type
Fuzzy Matching
Fuzzy Matching
Full Matching
Contain Matching
Add a Keyword
Responder Type
Text
You can also input 600 words
Cancel
Create
12:10
触发关键词
你成功触发了关键词！！
发送(S)

答案如下：

关键字用例								
用例编号	测试项目	测试标题	优先级	预置条件	测试数据	执行步骤	预期结果	备注
Keyword-ST-012	关键字匹配	Keyword输入框	中	可输入任何字符	关键字	1.输入测试数据	Keyword输入框可正常输入	如限制输入字符类型，需新增字符类型验证用例
Keyword-ST-013	关键字匹配	MatchingType下拉框	中			1.选择fuzzymatching 2.选择fullmatching 3.选择containmatching	MatchingType下拉框可正常选择	
Keyword-ST-001	关键字匹配	关键字不可重复	高	关键字不可重复	关键字 关键字	1.新增keyword“关键字”，matchingtype设置为fuzzymatching 2.新增keyword“关键字”，matchingtype设置为fullmatching	提示关键字不能重复	
Keyword-ST-001	关键字匹配	关键字匹配fuzzymatching	高	关键字不重复	关键字匹配 关键字	1.输入keyword“关键字匹配” 2.输入responder“关键字” 3.点击【Create】按钮	触发成功	
Keyword-ST-002	关键字匹配	关键字匹配fuzzymatching	高	关键字不重复	关键字 AAA	1.输入keyword“关键字” 2.输入responder“AAA” 3.点击【Create】按钮	未触发成功	
Keyword-ST-003	关键字匹配	关键字匹配fullmatching	高	关键字不重复	关键字 关键字	1.输入keyword“关键字” 2.输入responder“关键字” 3.点击【Create】按钮	触发成功	
Keyword-ST-004	关键字匹配	关键字匹配fullmatching	高	关键字不重复	关键字 关键	1.输入keyword“关键字” 2.输入responder“关键” 3.点击【Create】按钮	未触发成功	
Keyword-ST-004	关键字匹配	关键字匹配fullmatching	高	关键字不重复	关键字 关键字匹配	1.输入keyword“关键字” 2.输入responder“关键字匹配” 3.点击【Create】按钮	未触发成功	
Keyword-ST-005	关键字匹配	关键字匹配containmatching	高	关键字不重复	关键 关键字	1.输入keyword“关键” 2.输入responder“关键字” 3.点击【Create】按钮	触发成功	
Keyword-ST-006	关键字匹配	关键字匹配containmatching	高	关键字不重复	关键字 关键	1.输入keyword“关键字” 2.输入responder“关键” 3.点击【Create】按钮	未触发成功	
Keyword-ST-016	关键字匹配	【Add】功能				1.点击【Add】按钮	可增加一个Keyword设置模块	
Keyword-ST-016	关键字匹配	【Add】功能（增加条数最大值）		最大值为10条		1.点击【Add】按钮，添加至第十条	【Add】按钮置灰	根据需求增减
Keyword-ST-017	关键字匹配	【删除】功能				1.点击【删除】按钮	可删除一个Keyword设置模块	
Keyword-ST-017	关键字匹配	【删除】功能（删除最后一条）		不可以删除最后一条		1.点击【删除】按钮，删除至最后一条	【删除】按钮置灰	根据需求增减
Keyword-ST-014	关键字匹配	ResponderType文本框	中	可输入任何字符	关键字	1.输入测试数据	ResponderType文本框可正常输入	
Keyword-ST-014	关键字匹配	ResponderType文本框	中	最多600字		1.输入超过600字	多出600字部分不能输入	
Keyword-ST-015	关键字匹配	【Cancel】功能				1.点击【Cancel】按钮	可以清除文本框中一个或所有内容（根据需求编写）	

9、其他常用笔试题目网址汇总(2019-1-24-yl)

1）测试按动圆珠笔
2）对一只杯子进行测试
3）主流的压力/性能/负载测试工具介绍
4）jmeter 的几个测试重要指标
5）jmeter 性能指标分析
6）笔试选择题目集锦一
7）测试用例通用写法
8）APP 常见性能测试点
9）遇到不能复现的 bug 怎么办?

七、数据结构

1.1、冒泡排序的思想？(2018-4-16-lxy)

冒泡思想：通过无序区中相邻记录的关键字间的比较和位置的交换，使关键字最小的记录像气泡一样逐渐向上漂至水面。整个算法是从最下面的记录开始，对每两个相邻的关键字进行比较，把关键字较小的记录放到关键字较大的记录的上面，经过一趟排序后，关键字最小的记录到达最上面，接着再在剩下的记录中找关键字次小的记录，把它放在第二个位置上，依次类推，一直到所有记录有序为止

复杂度：时间复杂度为 O（n2）,空间复杂度为 O(1)

```
def bubble_sort(alist):
    for j in range(len(alist)-1, 0, -1):
        # j 表示每次遍历需要比较的次数，是逐渐减小的
        for i in range(j):
            if alist[i] > alist[i+1]:
                alist[i], alist[i+1] = alist[i+1], alist[i]

li = [54, 26, 93, 17, 77, 31, 44, 55, 20]
bubble_sort(li)
print(li)
```

1.2、快速排序的思想？(2018-4-16-lxy)

快排的基本思想：通过一躺排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一不部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

复杂度：快速排序是不稳定的排序算法，最坏的时间复杂度是 O（n2）,最好的时间复杂度是(nlogn),空间复杂度为 O(logn)

```
def quick_sort(alist, start, end):
    """快速排序"""
    # 递归的退出条件
    if start >= end:
        return
    # 设定起始元素为要寻找位置的基准元素
    mid = alist[start]
    # low 为序列左边的由左向右移动的游标
    low = start
    # high 为序列右边的由右向左移动的游标
    high = end
    while low < high:
        # 如果 low 与 high 未重合，high 指向的元素不比基准元素小，则 high 向左移动
        while low < high and alist[high] >= mid:
            high -= 1
        # 将 high 指向的元素放到 low 的位置上
        alist[low] = alist[high]
        # 如果 low 与 high 未重合，low 指向的元素比基准元素小，则 low 向右移动
```

```
        while low < high and alist[low] < mid:
            low += 1
        # 将 low 指向的元素放到 high 的位置上
        alist[high] = alist[low]
    # 退出循环后，low 与 high 重合，此时所指位置为基准元素的正确位置
    # 将基准元素放到该位置
    alist[low] = mid
    # 对基准元素左边的子序列进行快速排序
    quick_sort(alist, start, low-1)
    # 对基准元素右边的子序列进行快速排序
    quick_sort(alist, low+1, end)
alist = [54, 26, 93, 17, 77, 31, 44, 55, 20]
quick_sort(alist, 0, len(alist)-1)
print(alist)
```

1.3、如何判断单向链表中是否有环？(2018-4-16-lxy)

首先遍历链表，寻找是否有相同地址，借此判断链表中是否有环。如果程序进入死循环，则需要一块空间来存储指针，遍历新指针时将其和储存的旧指针比对，若有相同指针，则该链表有环，否则将这个新指针存下来后继续往下读取，直到遇见 NULL，这说明这个链表无环。

1.4、你知道哪些排序算法（一般是通过问题考算法）(2018-4-23-lyf)

冒泡， 选择， 快排， 归并。

1.5、斐波那契数列(2018-4-23-lyf)

斐波那契数列：简单地说，起始两项为 0 和 1，此后的项分别为它的前两项之和。

```
def fibo(num):
  numList = [0, 1]
  for i in range(num - 2):
     numList.append(numList[-2] + numList[-1])
     return numList
```

1.6、如何翻转一个单链表？(2018-4-23-lyf)

```
#!/usr/bin/env python
#coding = utf-8
class Node:
    def __init__(self, data=None, next = None):
        self.data = data
        self.next = next

def rev(link):
    pre = link
```

```
        cur = link.next
        pre.next = None
        while cur:
            temp = cur.next
            cur.next = pre
            pre =cur
            cur = temp
        return pre

    if __name__ == '__main__':
        link = Node(1, Node(2, Node(3, Node(4, Node(5, Node(6, Node(7, Node(8,
Node(9)))))))))
        root = rev(link)
        while root:
            print(root.data)
    24        root =root.next
```

1.7、青蛙跳台阶问题(2018-4-23-lyf)

一只青蛙要跳上 n 层高的台阶，一次能跳一级，也可以跳两级，请问这只青蛙有多少种跳上这个 n 层高台阶的方法?

思路分析:

这个问题有三种方法来解决，并在下面给出三处方法的 python 实现。

方法 1:递归

设青蛙跳上 n 级台阶有 f(n)种方法，把这 n 种方法分为两大类，第一种最后一次跳了一级台阶，这类方法共有 f(n-1)种，第二种最后一次跳了两级台阶，这种方法共有 f(n-2)种，则得出递推公式 f(n)=f(n-1)+f(n-2)，显然，f(1)=1，f(2)=2，递推公式如下:

* 这种方法虽然代码简单，但效率低，会超出时间上限*

代码实现如下:

```
    class Solution:
        # @param {integer} n
        # @return {integer}
        def climbStairs(self, n):
            if n==1:
                return 1
            elif n==2:
                return 2
            else:
                return self.climbStairs(n-1)+self.climbStairs(n-2)
```

方法 2: 用循环来代替递归

这种方法的原理仍然基于上面的公式，但是用循环代替了递归，比上面的代码效率上有较大的提升，可以 AC。

代码实现如下:

```
    class Solution:
        # @param {integer} n
```

```
        # @return {integer}
        def climbStairs(self, n):
            if n==1 or n==2:
                return n
            a=1;b=2;c=3
            for i in range(3, n+1):
                c=a+b;a=b;b=c
            return c
```

方法三：建立简单数学模型，利用组合数公式

设青蛙跳上这 n 级台阶一共跳了 z 次，其中有 x 次是一次跳了两级，y 次是一次跳了一级，则有 z=x+y ，2x+y=n，对一个固定的 x，利用组合可求出跳上这 n 级台阶的方法共有种方法又因为 x 在区间[0，n/2]内，所以我们只需要遍历这个区间内所有的整数，求出每个 x 对应的组合数累加到最后的结果即可。

```
python 代码实现如下：
class Solution:
    # @param {integer} n
    # @return {integer}
    def climbStairs(self, n):
        def fact(n):
            result=1
            for i in range(1, n+1):
                result*=i
            return result
        total=0
        for i in range(n/2+1):
            total+=fact(i+n-2*i)/fact(i)/fact(n-2*i)
        return total
```

1.8、两数之和 Two Sum(2018-4-23-lyf)

给一个整数数组，找到两个数使得他们的和等于一个给定的数 target。

你需要实现的函数 twoSum 需要返回这两个数的下标，并且第一个下标小于第二个下标。注意这里下标的范围是 1 到 n，不是以 0 开头。

样例：

给出 numbers = [2, 7, 11, 15], target = 9, 返回 [1, 2].

分析：

给定一个数列（注意不一定有序），和一个指定的数值 target。从这个数列中找出两个数相加刚好等于 target，要求给出这两个数的下标（注意数列下标是从 1 而不是从 0 开始）。

首先将数列排序。由于最后要得求的是两个数的原有下标，而不是两个数本身，因此要用一个新的对象 Item 来封装原有数列元素，并记录该元素的原有下标。排序是针对数列元素本身数值的，分别用一个 index1 指针和一个 index2 指针指向排序后的数列的首位，如果指向的两个数相加的和等于 target，则搜索结束；如果和小于 target，则由于 index2 此时指向的已经是数组中的最大数了，因此只能令 index1 向右移动一次；如果和大于 target，则由于此时 index1 已经指向数组中的最小数了，因此只能令 index2 向左移动一次。用一个循环重复上述过程，直到和等于 target 宣告搜索成功，或者 index1>=index2 宣告搜索失败。

```
        class Solution:
```

```
        class Item:
            def __init__(self, value, index):
                self.value = value
                self.index = index

        # @return a tuple, (index1, index2)
        def twoSum(self, num, target):
            len_num = len(num)
            if 0 == len_num:
                return (-1, -1)

            items = [Solution.Item(value, 0) for value in num]
            for i in range(0, len_num):
                items[i].index = i + 1
            items.sort(lambda x, y: cmp(x.value, y.value))
            index1 = 0
            index2 = len_num - 1
            is_find = False
            while index1 < index2:
                total = items[index1].value + items[index2].value
                if total < target:
                    index1 += 1
                elif total > target:
                    index2 -= 1
                else:
                    is_find = True
                    break
            (index1, index2) = (index1, index2) if items[index1].index <=
items[index2].index else (index2, index1)
            return (items[index1].index, items[index2].index) if is_find else (-1,
-1)
```

空间复杂度 O(n)，因为构造了一个新的 Item 序列。时间复杂度方面，如果假设 Python 的 sort 算法是用的快速排序的话，那排序的时间复杂度为 O(n*logn)，搜索过程的时间复杂度为 O(n)，因此总的时间复杂度为 O(n*logn)。

注意给的数列也许长度为 0，这样无论 target 是多少都是搜索失败。而且题目中给的 example 明显是是在误导人，不仔细看误以为数列原本就有序。此外，数列下标是从 1 而不是从 0 开始的。

但是这种算法的前提要先进行一次排序，看起来总是有点不舒服，是不是有更好的算法呢?

具体扫下面二维码查看:

1.9、数组中出现次数超过一半的数字-Python 版（2019-01-19-lq）

题目描述：

给定一个数组，如果这个数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字；如果不存在，则返回 0

思路分析：

对于一个数组而言，满足题目要求的数字最多只有一个，可以采用数字相互抵消的思想。在遍历数组时，储存两个值 now 和 count，now 是当前数字，count 是该数字的标记。当下一个数字与 now 相等时，标记 count 的值加 1，如果不相等，则减 1；当标记 count 的值变为 0 时，则将下一个数字的值用 now 来存储，并将 count 的值置为 1，继续遍历完数组。

如果数组中存在出现次数超过数组长度一半的数，那遍历过后 now 中存储的即为这个数字；

如果数组中不存在满足要求的数，最后 now 中也会存储一个不符合要求数值。

所以，在算法的最后，要重新遍历一遍数组，对变量 now 中的值进行计数并判断，从而验证结果。

解法：

根据思路编写代码，时间复杂度为 O(N)

```
def HalfLengthSolution(self, numbers):
        #判断极端情况
        if len(numbers) == 0:
            return 0
        #初始化记录变量 now 和 count
        count = 1
        now = numbers[0]
        length = len(numbers)
        #遍历数组，采用抵消的方法，寻找结果
        for i in range(1,length):
            if count == 0:
                now = numbers[i]
                count += 1
            else:
```

```
                if numbers[i] == now:
                    count += 1
                else:
                    count -= 1
            #验证 now 中存储的值是否符合题目要求
            test = 0
            for item in numbers:
                if item == now:
                    test += 1
            if test > length/2:
                return now
            else:
                return 0
```

1.10、求 100 以内的质数（2019-01-19-lq）

题目：获取 100 以内的质数

程序分析：质数又称素数，有无限个，质数定义为在大于 1 的自然数中，除了 1 和本身以外不再有其他因数的数称为质数，如：2、3、5、7、11、13、17、19

方法一：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

num=[];
i=2
for i in range(2,100):
   j=2
   for j in range(2,i):
      if(i%j==0):
         break
   else:
      num.append(i)
print(num)
```

方法二：

```
import math
def func_get_prime(n):
  return filter(lambda x: not [x%i for i in range(2, int(math.sqrt(x))+1) if x%i ==0], range(2,n+1))

print func_get_prime(100)
```

输出结果为：

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

1.11、无重复字符的最长子串-Python 实现（2019-01-19-lq）

题目：给定一个字符串，找出不含有重复字符的最长子串的长度。

示例 1:

```
输入: "abcabcbb"
输出: 3
解释: 无重复字符的最长子串是 "abc"，其长度为 3。
```

示例 2:

```
输入: "bbbbb"
输出: 1
解释: 无重复字符的最长子串是 "b"，其长度为 1。
```

示例 3:

```
输入: "pwwkew"
输出: 3
解释: 无重复字符的最长子串是 "wke"，其长度为 3。
     请注意，答案必须是一个子串，"pwke" 是一个子序列 而不是子串。
```

思路：

遍历字符串中的每一个元素。用一个字典 str_dict 来存储不重复的字符和字符所在的下标。用变量 start 存储当前最近重复字符所在的位置+1

代码：

```
class Solution:
    def lengthOfLongestSubstring(self, s):
        """
        :type s: str
        :rtype: int
        """
        # 存储历史循环中最长的子串长度
        max_len = 0
        # 判断传入的字符串是否为空
        if s is None or len(s) == 0:
            return max_len
        # 定义一个字典，存储不重复的字符和字符所在的下标
        str_dict = {}
        # 存储每次循环中最长的子串长度
        one_max = 0
        #记录最近重复字符所在的位置+1
        start = 0
        for i in range(len(s)):
            # 判断当前字符是否在字典中和当前字符的下标是否大于等于最近重复字符的所在位置
```

```
            if s[i] in str_dict and str_dict[s[i]] >= start:
                # 记录当前字符的值+1
                start = str_dict[s[i]]
            # 在此次循环中，最大的不重复子串的长度
            one_max = i - start+1
            # 把当前位置覆盖字典中的位置
            str_dict[s[i]] = i
            # 比较此次循环的最大不重复子串长度和历史循环最大不重复子串长度
            max_len = max(max_len, one_max)
        return max_len
```

1.12、通过 2 个 5/6 升得水壶从池塘得到 3 升水（2019-01-19-lq）

题目：假设有一个池塘，里面有无穷多的水，现有两个空水壶，容积分别为 5 升和 6 升，如何只用这 2 个水壶从池塘里取得 3 升的水.

解答：

第一步：先取来 6 升水，倒进 5 升桶的水桶里，即得到 6 升桶里余下的 1 升水；

第二步：把 5L 桶清掉，把取到的 1 升水放进 5 升的水桶里保留不动，然后再取 6 升水，倒进 5 升的水桶里，6 升的桶得到的是 2 升水，把 5L 桶清掉，存放这 2 升水；

第三步：5 升水桶有 2 升水．再取 6 升水，倒进 5 升水桶里，原有 2L 升+3 升=5 升，这时 6 升-3 升=3 升，5 升就没用了，要的就是 6 升里余下的这 3 升水了．

1.13、搜索旋转排序数组 Search in Rotated Sorted Array(2018-4-23-lyf)

假设有一个排序的按未知的旋转轴旋转的数组(比如，0 1 2 4 5 6 7 可能成为 4 5 6 7 0 1 2)。给定一个目标值进行搜索，如果在数组中找到目标值返回数组中的索引位置，否则返回-1。

你可以假设数组中不存在重复的元素。

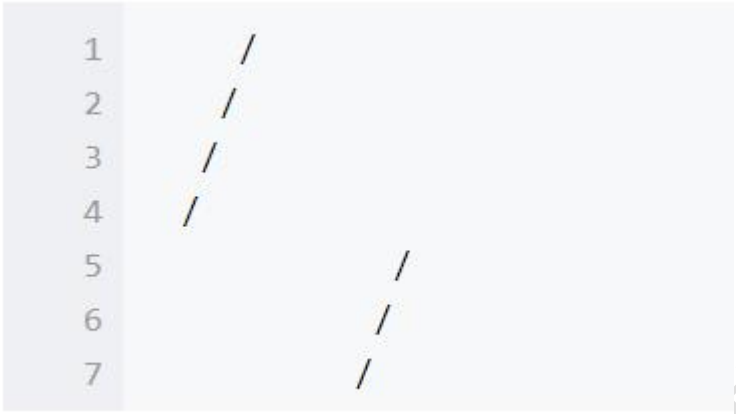
样例：

给出[4, 5, 1, 2, 3]和 target=1，返回 2

给出[4, 5, 1, 2, 3]和 target=0，返回 -1

解题思路：

采用了二分搜索，不过旋转后的数组要讨论的情况增多了。其实旋转后的数组的大小关系一般如下图：先通过中点与左顶点的关系来分类讨论中点落在了哪一部分，如果在左半边，则继续讨论目标数在中点的左边还是右边；如果在右半边，同样讨论目标数的位置。同时需要注意特殊情况只剩下两个数时，例如[3,1]，这时求出的中点也是 3，如果中点不匹配，应考虑 1。这种情况不好与上面的情况合并，单独列出。



```
class Solution:

    def doSearch(self, A, left, right, target):
        len_A = right - left + 1
        if 0 == len_A:
            return -1
        elif 1 == len_A:     ###
            return left if target == A[left] else -1
        if A[left] < A[right]:
            while left <= right:
                mid = (left + right) / 2
                if A[mid] < target:
                    left = mid + 1
                elif A[mid] > target:
                    right = mid - 1
                else:
                    return mid
        else:
            mid = (left + right) / 2
            left_result = self.doSearch(A, left, mid, target)
            if -1 != left_result:
                return left_result
            else:
                right_result = self.doSearch(A, mid + 1, right, target)
                if -1 != right_result:
                    return right_result
        return -1

    # @param A, a list of integers
    # @param target, an integer to be searched
    # @return an integer
    def search(self, A, target):
```

```
        len_A = len(A)
        return self.doSearch(A, 0, len(A) - 1, target)
```

思考：

其他条件不变，假如有重复元素又将如何？是否会影响运行时间复杂度？如何影响？为何会影响？写出一个函数判断给定的目标值是否出现在数组中。

样例：

给出[3,4,4,5,7,0,1,2]和 target=4，返回 true。

```
class Solution(object):
    def search(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int
        """
        left = 0
        right = len(nums) - 1
        while left <= right:
            mid = left + (right - left) // 2
            if nums[mid] == target:
                return True
            if nums[mid] > target:
                if nums[left] <= target or nums[mid] < nums[left]:
                    right = mid - 1
                else:
                    left = mid + 1
            else:
                if nums[left] > target or nums[mid] >= nums[left]:
                    left = mid + 1
                else:
                    right = mid - 1
        return False


if __name__ == "__main__":
    assert Solution().search([4, 5, 5, 6, 7, 0, 1, 2], 4) == True
    assert Solution().search([4, 5, 6, 7, 7, 7, 7, 7, 0, 1, 2], 7) == True
```

1.14、Python 实现一个 Stack 的数据结构

栈（有时称为“后进先出栈”） 是一个项的有序集合，其中添加移除新项总发生在同一端。这一端通常称为“顶部”。与顶部对应的端称为“底部”。

栈的底部很重要，因为在栈中靠近底部的项是存储时间最长的。最近添加的项是最先会被移除的。这种排序原则有时被称为 LIFO，后进先出。它基于在集合内的时间长度做排序。较新的项靠近顶部，较旧的项靠近底部。

下面来说说栈的几个方法：

Stack() 创建一个空的新栈。 它不需要参数，并返回一个空栈。#新建个薯片盒子

push(item)将一个新项添加到栈的顶部。它需要 item 做参数并不返回任何内容。#拿一块定制薯片扔进薯片筒

pop() 从栈中删除顶部项。它不需要参数并返回 item 。栈被修改。#吃点顶部的薯片

peek() 从栈返回顶部项，但不会删除它。不需要参数。 不修改栈。#看一下顶部的薯片，不吃

isEmpty() 测试栈是否为空。不需要参数，并返回布尔值。#看下薯片筒空了没

size() 返回栈中的 item 数量。不需要参数，并返回一个整数。#数下有几块薯片

代码:

class Stack:

```
    def __init__(self):
        self.items =[]

    def isEmpty(self):
        return self.items == []

    def push(self,item):
        return self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)

s =Stack()
print s.isEmpty()
s.push(3)
s.push(4)
print s.pop()
s.peek()
print s.size()
```

输出:

```
True
4
1
```

1.15、写一个二分查找(2018-4-23-lyf)

```
def binary_search(a, n, key):
    m = 0; l = 0; r = n - 1#闭区间[0, n - 1]
    while (l < r):
```

```
            m = l + ((r - l) >> 1)#向下取整
            if (a[m] < key): l = m + 1;
            else: r = m;

        if (a[r] == key): return r;
        return -1
```

1.16、set 用 in 时间复杂度是多少，为什么？(2018-4-23-lyf)

O(1)，因为 set 是键值相同的一个数据结构，键做了 hash 处理。

1.17、什么是 MD5 加密，有什么特点？（2019-01-19-lq）

MD5:全称是 Message Digest Algorithm 5，译为“消息摘要算法第 5 版”效果：对输入信息生成唯一的 128 位散列值（32 个字符）

MD5 的特点

对不同的数据加码，得到的结果是定长的，MD5 对不同的数据进行加密，得到的结果都是 32 个字符；

根据输出值，不能得到原始的明文，即其过程是不可逆的；

算法具有较好的安全性，并且免费；

广泛使用主要运用在数字签名，文件完整性验证以及口令加密等方面

1.18、什么是对称加密和非对称加密（2019-01-19-lq）

对称加密：A 与 B 之间的通讯数据都用同一套的密钥来进行加密解密；

优点

简单快捷，密钥较短，且破译困难

缺点

如果用户一旦多的话，管理密钥也是一种困难。不方便直接沟通的两个用户之间怎么确定密钥也需要考虑，这其中就会有密钥泄漏的风险，以及存在更换密钥的需求

对称加密通常有 DES，IDEA，3DES 加密算法

非对称加密：用公钥和私钥来加解密的算法。打个比方，A 的公钥加密过的东西只能通过 A 的私钥来解密；同理，A 的私钥加密过的东西只能通过 A 的公钥来解密。顾名思义，公钥是公开的，别人的可以获取的到；私钥是私有的，只能自己拥有；

优点

比对称加密安全

缺点

加解密比对称加密耗时，非对称加密也是存在漏洞，因为公钥是公开的，如果有 C 冒充 B 的身份利用 A 的公钥给 A 发消息，这样就乱套了，所以接下来就采用非对称加密+摘要算法+数字签名的机制来确保传输安全。常见的非对称加密算法有：RSA、ECC（移动设备用）、Diffie-Hellman、El Gamal、DSA（数字签名用）

完整的非对称加密过程：

假如现在 你向支付宝 转账（术语数据信息），为了保证信息传送的保密性、真实性、完整性和不可否认性，需要对传送的信息进行数字加密和签名，其传送过程为：

首先你要确认是否是支付宝的数字证书，如果确认为支付宝身份后，则对方真实可信。可以向对方传送信息，

你准备好要传送的数字信息（明文）计算要转的多少钱，对方支付宝账号等；

你对数字信息进行哈希运算，得到一个信息摘要（客户端主要职责）；

你用自己的私钥对信息摘要进行加密得到 你 的数字签名，并将其附在数字信息上；

你随机产生一个加密密钥，并用此密码对要发送的信息进行加密（密文）；

你用支付宝的公钥对刚才随机产生的加密密钥进行加密，将加密后的 DES 密钥连同密文一起传送给支付宝；

支付宝收到你传送来的密文和加密过的 DES 密钥，先用自己的私钥对加密的 DES 密钥进行解密，得到你随机产生的加密密钥；

支付宝然后用随机密钥对收到的密文进行解密，得到明文的数字信息，然后将随机密钥抛弃；

支付宝用你的公钥对 你的的数字签名进行解密，得到信息摘要；

支付宝用相同的哈希算法对收到的明文再进行一次哈希运算，得到一个新的信息摘要；

支付宝将收到的信息摘要和新产生的信息摘要进行比较，如果一致，说明收到的信息没有被修改过。

确定收到信息，然后进行向对方进行付款交易，一次非对称密过程结束。在这后面的流程就不属于本次非对称加密的范畴，算支付宝个人的自我流程，也就是循环以上过程。

1.19、列表中有 n 个正整数范围在[0，1000]，进行排序；(2018-4-23-lyf)

```
def quick_sort(lista, start, stop):
    if start >= stop:
        return
    low = start
    high = stop
    mid = lista[start]
    while low < high:
        while low < high and lista[high] >= mid:
            high -= 1
        lista[low] = lista[high]
     while low < high and mid > lista[low]:
            low += 1
        lista[high] = lista[low]
    lista[low] = mid
    quick_sort(lista, start, low-1)
    quick_sort(lista, low+1, stop)
    return lista
lista = [0, 1000]
print(quick_sort(lista, 0, len(lista)-1))
```

1.20、面向对象编程中有组合和继承的方法实现新的类(2018-4-23-lyf)

假设我们手头只有“栈”类，请用“组合”的方式使用“栈”（LIFO）来实现“队列”（FIFO），完成以下代码?

```
Import stack
Class queue(stack):
   def __init__(self):
   def __init__(self):
   stack1 = stack() # 进来的元素都放在里面
   stack2 = stack() # 元素都从这里出去
```

```
def push(self, element):
    self.stack1.push(element)
def pop(self):
    if self.stack2.empty():#如果没有元素， 就把负责放入元素的栈中元素全部放进来
         while not self.stack1.empty():
            self.stack2.push(self.stack1.top())
            self.stack1.pop()
       ret = self.stack2.top() # 有元素后就可以弹出了
       self.stack2.pop()
       return ret
def top(self):
     if (self.stack2.empty()):
         while not self.stack1.empty():
         self.stack2.push(self.stack1.top())
            self.stack1.pop()
    return self.stack2.top()
```

八、人工智能

1.1、找出 1G 的文件中高频词（2018-4-23-xhq）

每一行是一个词，词的大小不超过 16 字节，内存限制大小是 1M，返回频数最高的 100 个词。

使用生成器读取文件。每次读取 65536 行，一共进行 1500 次，当读取不到内容时关闭文件。每次读取，最终要得到 100 个频数最高的词。每 500 次，进行一次合并和统计，得到最多 50000 个词，对这 50000 个词提取其中频数最高的 100 个词。最终对最多 300 个筛选出来的词，进行合并和统计，提取最终频数最高的 100 个词。

筛选出 100 个高频词的步骤：

统计每个词出现的次数。维护一个 Key 为词，Value 为该词出现次数的 hash 表。每次读取一个词，如果该字串不在 hash 表中，那么加入该词，并且将 Value 值设为 1；如果该字串在 hash 表中，那么将该字串的计数加一即可。

据每个词的引用次数进行排序。冒泡、快排等等都可以。

1.2、一个大约有一万行的文本文件统计高频词（2018-4-23-xhq）

每行一个词，要求统计出其中最频繁出现的前 10 个词，请给出思想和时间复杂度分析。

用 trie 树统计每个词出现的次数，时间复杂度是 O(n*le)(le 表示单词的平准长度)。然后是找出出现最频繁的前 10 个词，可以用堆来实现，时间复杂度是 O(n*lg10)。所以总的时间复杂度，是 O(n*le)与 O(n*lg10)中较大的哪一个。

http://www.mamicode.com/info-detail-1037262.html

1.3、怎么在海量数据中找出重复次数最多的一个？（2018-4-23-xhq）

算法思想：

先做 hash，然后求模映射为小文件，求出每个小文件中重复次数最多的一个，并记录重复次数。然后找出上一步求出的数据中重复次数最多的一个就是所求。

参考博客一：https://blog.csdn.net/u010601183/article/details/56481868

参考博客二：https://www.cnblogs.com/lianghe01/p/4391804.html

1.4、判断数据是否在大量数据中

给 2 亿个不重复的整数，没排过序的，然后再给一个数，如何快速判断这个数是否在那 2 亿个数当中？

unsigned int 的取值范围是 0 到 2^32-1。我们可以申请连续的 2^32/8=512M 的内存，用每一个 bit 对应一个 unsigned int 数字。首先将 512M 内存都初始化为 0，然后每处理一个数字就将其对应的 bit 设置为 1。当需要查询时，直接找到对应 bit，看其值是 0 还是 1 即可。