目录

第一阶段

1、Python 的函数参数传递

看两个例子:

```
a = 1
def fun(a):
  a = 2
fun(a)
print a  # 1
a = []
def fun(a):
  a.append(1)
fun(a)
print a  # [1]
```

所有的变量都可以理解是内存中一个对象的“引用”，或者，也可以看似 c 中 void*的感觉。这里记住的是类型是属于对象的，而不是变量。而对象有两种,“可更改”（mutable）与“不可更改”（immutable）对象。在 python 中，strings, tuples, 和 numbers 是不可更改的对象，而 list,dict 等则是可以修改的对象。(这就是这个问题的重点)

当一个引用传递给函数的时候,函数自动复制一份引用,这个函数里的引用和外边的引用没有半毛关系了.所以第一个例子里函数把引用指向了一个不可变对象,当函数返回的时候,外面的引用没半毛感觉.而第二个例子就不一样了,函数内的引用指向的是可变对象,对它的操作就和定位了指针地址一样,在内存里进行修改.

2、Python 中的元类(metaclass)

这个非常的不常用,但是像 ORM 这种复杂的结构还是会需要的,详情请看：《深刻理解 Python 中的元类(metaclass)》

3、@staticmethod 和@classmethod

Python 其实有 3 个方法,即静态方法(staticmethod),类方法(classmethod)和实例方法,如下:

```
def  foo(x):
        print  "executing  foo(%s)"%(x)
class  A(object):
        def  foo(self,x):
                print  "executing  foo(%s,%s)"%(self,x)
        @classmethod
        def  class_foo(cls,x):
                print  "executing  class_foo(%s,%s)"%(cls,x)
        @staticmethod
        def  static_foo(x):
                print  "executing  static_foo(%s)"%x
```

a=A()
这里先理解下函数参数里面的 self 和 cls.这个 self 和 cls 是对类或者实例的绑定,对于一般的函数来说我们可以这么调用 foo(x),这个函数就是最常用的,它的工作跟任何东西(类,实例)无关.对于实例方法,我们知道在类里每次定义方法的时候都需要绑定这个实例,就是 foo(self, x),为什么要这么做呢?因为实例方法的调用离不开实例,我们需要把实例自己传给函数,调用的时候是这样的 a.foo(x)(其实是 foo(a, x)).类方法一样,只不过它传递的是类而不是实例,A.class_foo(x).注意这里的 self 和 cls 可以替换别的参数,但是 python 的约定是这俩,还是不要改的好.
对于静态方法其实和普通的方法一样,不需要对谁进行绑定,唯一的区别是调用的时候需要使用 a.static_foo(x)或者 A.static_foo(x)来调用.
\ 实例方法 类方法 静态方法
a = A() a.foo(x) a.class_foo(x) a.static_foo(x)
A 不可用 A.class_foo(x) A.static_foo(x)

4、类变量和实例变量

```
class  Person:
        name="aaa"
p1=Person()
p2=Person()
p1.name="bbb"
print  p1.name    #  bbb
print  p2.name    #  aaa
print  Person.name    #  aaa
print  Person.name    #  aaa
```

类变量就是供类使用的变量,实例变量就是供实例使用的.
这里 p1.name="bbb"是实例调用了类变量,这其实和上面第一个问题一样,就是函数传参的问题,p1.name 一开始是指向的类变量 name="aaa",但是在实例的作用域里把类变量的引用改变了,就变成了一个实例变量,self.name 不再引用 Person 的类变量 name 了.
可以看看下面的例子:

```
class  Person:
        name=[]
p1=Person()
p2=Person()
p1.name.append(1)
print  p1.name    #  [1]
print  p2.name    #  [1]
print  Person.name    #  [1]
```

5、Python 自省

这个也是 python 彪悍的特性.

自省就是面向对象的语言所写的程序在运行时,所能知道对象的类型.简单一句就是运行时能够获得对象的类型.比如
type(),dir(),getattr(),hasattr(),isinstance().

6、 字典推导式

可能你见过列表推导时,却没有见过字典推导式,在 2.7 中才加入的:

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

7、 Python 中单下划线和双下划线

```
>>> class MyClass():
...          def __init__(self):
...                          self.__superprivate = "Hello"
...                          self._semiprivate = ", world!"
>>> mc = MyClass()
>>> print mc.__superprivate
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AttributeError: myClass instance has no attribute '__superprivate'
>>> print mc._semiprivate
, world!
>>> print mc.__dict__
{'_MyClass__superprivate': 'Hello', '_semiprivate': ', world!'}
```

__foo__:一种约定,Python 内部的名字,用来区别其他用户自定义的命名,以防冲突.
_foo:一种约定,用来指定变量私有.程序员用来指定私有变量的一种方式.
__foo:这个有真正的意义:解析器用_classname__foo 来代替这个名字,以区别和其他类相同的命名.

8、 字符串格式化:%和.format

.format 在许多方面看起来更便利.对于%最烦人的是它无法同时传递一个变量和元组.你可能会想下面的代码不会有什么问题:

```
"hi there %s" % name
```

但是,如果 name 恰好是(1,2,3),它将会抛出一个 TypeError 异常.为了保证它总是正确的,你必须这样做:

```
"hi there %s" % (name,)       # 提供一个单元素的数组而不是一个参数
```

但是有点丑..format 就没有这些问题.你给的第二个问题也是这样,.format 好看多了.
你为什么不用它?
不知道它(在读这个之前)

为了和 Python2.5 兼容(譬如 logging 库建议使用%(issue #4))

9、 迭代器和生成器

迭代器
对序列（列表、元组）、字典和文件都可以用 iter()方法生成迭代对象，然后用 next()方法访问。
python3.x，迭代器对象实现的是__next__()方法，不是 next()。
在 python3.x 中有一个内建函数 next()，可以实现 next(it)，访问迭代器，这相当于于 python2.x 中的 it.next()（it 是迭代对象）。
通过 range()得到的列表，会一次性被读入内存，而 xrange()返回的对象，则是需要一个数值才从返回一个数值。
迭代器的确有迷人之处，但是它也不是万能之物。比如迭代器不能回退，只能如过河的卒子，不断向前。另外，迭代器也不适合在多线程环境中对可变集合使用如果是迭代器，就可以用 for 循环来依次读出其值。
生成器
生成器必须是可迭代的,可以理解为非常方便的自定义迭代器

10、 *args and **kwargs

用*args 和**kwargs 只是为了方便并没有强制使用它们.
当你不确定你的函数里将要传递多少参数时你可以用*args.例如,它可以传递任意数量的参数:

```
>>> def print_everything(*args):
                for count, thing in enumerate(args):
...                 print '{0}. {1}'.format(count, thing)
>>> print_everything('apple', 'banana', 'cabbage')
0. apple
1. banana
2. cabbage
```

相似的,**kwargs 允许你使用没有事先定义的参数名:

```
>>> def table_things(**kwargs):
...         for name, value in kwargs.items():
...                 print '{0} = {1}'.format(name, value)
>>> table_things(apple = 'fruit', cabbage = 'vegetable')
cabbage = vegetable
apple = fruit
```

你也可以混着用.命名参数首先获得参数值然后所有的其他参数都传递给*args 和**kwargs.命名参数在列表的最前端.例如:

```
def table_things(titlestring, **kwargs)
```

*args 和**kwargs 可以同时在函数的定义中,但是*args 必须在**kwargs 前面.
当调用函数时你也可以用*和**语法.例如:

```
>>> def print_three_things(a, b, c):
...         print 'a = {0}, b = {1}, c = {2}'.format(a,b,c)
```

```
...
>>> mylist = ['aardvark', 'baboon', 'cat']
>>> print_three_things(*mylist)
a = aardvark, b = baboon, c = cat
```

就像你看到的一样,它可以传递列表(或者元组)的每一项并把它们解包.注意必须与它们在函数里的参数相吻合.当然,你也可以在函数定义或者函数调用时用*.

11、面向切面编程 AOP 和装饰器

这个 AOP 一听起来有点懵,同学面阿里的时候就被问懵了…
装饰器是一个很著名的设计模式，经常被用于有切面需求的场景，较为经典的有插入日志、性能测试、事务处理等。装饰器是解决这类问题的绝佳设计，有了装饰器,我们就可以抽离出大量函数中与函数功能本身无关的雷同代码并继续重用。概括的讲，装饰器的作用就是为已经存在的对象添加额外的功能。

12、鸭子类型

“当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。”
我们并不关心对象是什么类型，到底是不是鸭子，只关心行为。
比如在 python 中,有很多 file-like 的东西,比如 StringIO,GzipFile,socket。它们有很多相同的方法，我们把它们当作文件使用。
又比如 list.extend()方法中,我们并不关心它的参数是不是 list,只要它是可迭代的,所以它的参数可以是 list/tuple/dict/字符串/生成器等.
鸭子类型在动态语言中经常使用，非常灵活，使得 python 不想 java 那样专门去弄一大堆的设计模式。

13、 Python 中重载

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。
另外，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,那么不应当使用重载，而应当使用一个名字不同的函数。
好吧，那么对于情况 1，函数功能相同，但是参数类型不同，python 如何处理？答案是根本不需要处理，因为 python 可以接受任何类型的参数，如果函数的功能相同，那么不同的参数类型在 python 中很可能是相同的代码，没有必要做成两个不同函数。
那么对于情况 2,函数功能相同,但参数个数不同,python 如何处理？大家知道，答案就是缺省参数。对那些缺少的参数设定为缺省参数即可解决问题。因为你假设函数功能相同，那么那些缺少的参数终归是需要用的。
好了，鉴于情况 1 跟情况 2 都有了解决方案，python 自然就不需要函数重载了。

14、新式类和旧式类

Python 中类分两种：旧式类和新式类：
新式类都从 object 继承，经典类不需要。
新式类的 MRO(method resolution order 基类搜索顺序)算法采用 C3 算法广度优先搜索，而旧式类的 MRO 算法是采用深度优先搜索
新式类相同父类只执行一次构造函数，经典类重复执行多次。
其中：
截止到 python2.1，只存在旧式类。旧式类中，类名和 type 是无关的：如果 x 是一个旧式类，那么 x.__class__定义了 x 的类名，但是 type(x)总是返回<type 'instance'>。这反映了所有的旧式类的实例是通过一个单一的叫做 instance 的内建类型来实现的，这是它和类不同的地方。
新式类是在 python2.2 为了统一类和实例引入的。一个新式类只能由用户自定义。如果 x 是一个新式类的实例，那么 type(x)和 x.__class__是一样的结果（尽管这不能得到保证，因为新式类的实例的__class__方法是允许被用户覆盖的）。
Python 2.x 中默认都是经典类，只有显式继承了 object 才是新式类
Python 3.x 中默认都是新式类,经典类被移除，不必显式的继承 object

15、 __new__和__init__的区别

这个__new__确实很少见到,先做了解吧.
__new__是一个静态方法,而__init__是一个实例方法.
__new__方法会返回一个创建的实例,而__init__什么都不返回.
只有在__new__返回一个 cls 的实例时后面的__init__才能被调用.
当创建一个新实例时调用__new__,初始化一个实例时用__init__.
ps: __metaclass__是创建类时起作用.所以我们可以分别使用__metaclass__,__new__和__init__来分别在类创建,实例创建和实例初始化的时候做一些小手脚.

16、单例模式

这个绝对常考啊.绝对要记住 1~2 个方法,当时面试官是让手写的.
1 使用__new__方法

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

class  MyClass(Singleton):
```

```
        a = 1
```

2 共享属性

创建实例时把所有实例的__dict__指向同一个字典,这样它们具有相同的属性和方法.

```
class Borg(object):
        _state = {}
        def __new__(cls, *args, **kw):
                ob = super(Borg, cls).__new__(cls, *args, **kw)
                ob.__dict__ = cls._state
                return ob
class MyClass2(Borg):
        a = 1
```

3 装饰器版本

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

@singleton
class MyClass:
    ...
```

4 import 方法

作为 python 的模块是天然的单例模式

```
# mysingleton.py
class My_Singleton(object):
        def foo(self):
                pass

my_singleton = My_Singleton()
# to use
from mysingleton import my_singleton
my_singleton.foo()
```

17、Python 中的作用域

Python 中，一个变量的作用域总是由在代码中被赋值的地方所决定的。

当 Python 遇到一个变量的话他会按照这样的顺序进行搜索：

本地作用域（Local）→当前作用域被嵌入的本地作用域（Enclosing locals）→全局/模块作用域（Global）→内置作用域（Built-in）

18、闭包

闭包(closure)是函数式编程的重要的语法结构。闭包也是一种组织代码的结构，它同样提高了代码的可重复使用性。
当一个内嵌函数引用其外部作作用域的变量,我们就会得到一个闭包. 总结一下,创建一个闭包必须满足以下几点:
必须有一个内嵌函数
内嵌函数必须引用外部函数中的变量
外部函数的返回值必须是内嵌函数
感觉闭包还是有难度的,几句话是说不明白的,还是查查相关资料.
重点是函数运行后并不会被撤销,就像16题的instance字典一样,当函数运行完后,instance并不被销毁,而是继续留在内存空间里.这个功能类似类里的类变量,只不过迁移到了函数上.
闭包就像个空心球一样,你知道外面和里面,但你不知道中间是什么样.

19、lambda 函数

其实就是一个匿名函数,为什么叫lambda?因为和后面的函数式编程有关.

20、程序编译与链接

Bulid过程可以分解为4个步骤:预处理(Prepressing)， 编译(Compilation)、汇编(Assembly)、链接(Linking)
以c语言为例:
1 预处理
预编译过程主要处理那些源文件中的以“#”开始的预编译指令，主要处理规则有：
将所有的“#define”删除，并展开所用的宏定义
处理所有条件预编译指令,比如“#if”、“#ifdef”、 “#elif”、“#endif”
处理“#include”预编译指令，将被包含的文件插入到该编译指令的位置，注：此过程是递归进行的
删除所有注释
添加行号和文件名标识,以便于编译时编译器产生调试用的行号信息以及用于编译时产生编译错误或警告时可显示行号保留所有的#pragma编译器指令。
2 编译
编译过程就是把预处理完的文件进行一系列的词法分析、语法分析、语义分析及优化后生成相应的汇编代码文件。这个过程是整个程序构建的核心部分。
3 汇编
汇编器是将汇编代码转化成机器可以执行的指令,每一条汇编语句几乎都是一条机器指令。经过编译、链接、汇编输出的文件成为目标文件(Object File)
4 链接
链接的主要内容就是把各个模块之间相互引用的部分处理好,使各个模块可以正确的拼接。

链接的主要过程包括地址和空间的分配(Address and Storage Allocation)、符号决议(Symbol Resolution)和重定位(Relocation)等步骤。

21、静态链接和动态链接

静态链接方法：静态链接的时候，载入代码就会把程序会用到的动态代码或动态代码的地址确定下来
静态库的链接可以使用静态链接，动态链接库也可以使用这种方法链接导入库
动态链接方法：使用这种方式的程序并不在一开始就完成动态链接，而是直到真正调用动态库代码时，载入程序才计算(被调用的那部分)动态代码的逻辑地址，然后等到某个时候，程序又需要调用另外某块动态代码时，载入程序又去计算这部分代码的逻辑地址，所以，这种方式使程序初始化时间较短，但运行期间的性能比不上静态链接的程序

22、虚拟内存技术

虚拟存储器是值具有请求调入功能和置换功能,能从逻辑上对内存容量加以扩充的一种存储系统.

23、分页和分段

分页： 用户程序的地址空间被划分成若干固定大小的区域，称为“页”，相应地，内存空间分成若干个物理块，页和块的大小相等。可将用户程序的任一页放在内存的任一块中，实现了离散分配。
分段： 将用户程序地址空间分成若干个大小不等的段，每段可以定义一组相对完整的逻辑信息。存储分配时，以段为单位，段与段在内存中可以不相邻接，也实现了离散分配。
分页与分段的主要区别：
页是信息的物理单位,分页是为了实现非连续分配,以便解决内存碎片问题,或者说分页是由于系统管理的需要.段是信息的逻辑单位,它含有一组意义相对完整的信息,分段的目的是为了更好地实现共享,满足用户的需要.
页的大小固定,由系统确定,将逻辑地址划分为页号和页内地址是由机器硬件实现的.而段的长度却不固定,决定于用户所编写的程序,通常由编译程序在对源程序进行编译时根据信息的性质来划分.
分页的作业地址空间是一维的.分段的地址空间是二维的.

24、页面置换算法

最佳置换算法 OPT:不可能实现
先进先出 FIFO
最近最久未使用算法 LRU:最近一段时间里最久没有使用过的页面予以置换.
clock 算法

25、边沿触发和水平触发

边缘触发是指每当状态变化时发生一个 io 事件，条件触发是只要满足条件就发生一个 io 事件

26、标记-清除机制

基本思路是先按需分配，等到没有空闲内存的时候从寄存器和程序栈上的引用出发，遍历以对象为节点、以引用为边构成的图，把所有可以访问到的对象打上标记，然后清扫一遍内存空间，把所有没标记的对象释放。

27、分代技术

分代回收的整体思想是：将系统中的所有内存块根据其存活时间划分为不同的集合，每个集合就成为一个“代”，垃圾收集频率随着“代”的存活时间的增大而减小，存活时间通常利用经过几次垃圾回收来度量。
Python 默认定义了三代对象集合，索引数越大，对象存活时间越长。
举例：
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。

28、找零问题

```
def coinChange(values, money, coinsUsed):
    #values          T[1:n]数组
    #valuesCounts      钱币对应的种类数
    #money    找出来的总钱数
    #coinsUsed      对应于目前钱币总数 i 所使用的硬币数目
    for cents in range(1, money+1):
            minCoins = cents #从第一个开始到 money 的所有情况初始
            for value in values:
                if value <= cents:
                    temp = coinsUsed[cents-value] + 1
                    if temp < minCoins:
                        minCoins = temp
            coinsUsed[cents] = minCoins
            print('面值为：{0} 的最小硬币数目为：
{1} '.format(cents, coinsUsed[cents]) )
if __name__ == '__main__':
```

```
values = [ 25, 21, 10, 5, 1]
money = 63
coinsUsed = {i:0 for i in range(money+1)}
coinChange(values, money, coinsUsed)
```

29、Python 函数式编程

python 中函数式编程支持:
filter 函数的功能相当于过滤器。调用一个布尔函数 bool_func 来迭代遍历每个 seq 中的元素；返回一个使 bool_seq 返回值为 true 的元素的序列。

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

map 函数是对一个序列的每个项依次执行函数，下面是对一个序列每个项都乘以 2：

```
>>> a = map(lambda x:x*2,[1,2,3])
>>> list(a)
[2, 4, 6]
```

reduce 函数是对一个序列的每个项迭代调用函数，下面是求 3 的阶乘：

```
>>> reduce(lambda x,y:x*y,range(1,4))
6
```

30、Python 里的拷贝

引用和 copy(),deepcopy()的区别

```
import copy
a = [1, 2, 3, 4, ['a', 'b']] #原始对象
b = a #赋值，传对象的引用
c = copy.copy(a) #对象拷贝，浅拷贝
d = copy.deepcopy(a) #对象拷贝，深拷贝
a.append(5) #修改对象 a
a[4].append('c') #修改对象 a 中的['a', 'b']数组对象
print 'a = ', a
print 'b = ', b
print 'c = ', c
print 'd = ', d
```

输出结果：

```
a = [1, 2, 3, 4, ['a', 'b', 'c'], 5]
b = [1, 2, 3, 4, ['a', 'b', 'c'], 5]
c = [1, 2, 3, 4, ['a', 'b', 'c']]
d = [1, 2, 3, 4, ['a', 'b']]
```

31、 Python 垃圾回收机制

Python GC 主要使用引用计数（reference counting）来跟踪和回收垃圾。在引用计数的基础上，通过“标记-清除”（mark and sweep）解决容器对象可能产生的循环引用问题，通过“分代回收”（generation collection）以空间换时间的方法提高垃圾回

32、到底什么是 Python？

答案：（关键点）

Python 是一种解释型语言。这就是说，与 C 语言和 C 的衍生语言不同，Python 代码在运行之前不需要编译。其他解释型语言还包括 PHP 和 Ruby。

Python 是动态类型语言，指的是你在声明变量时，不需要说明变量的类型。你可以直接编写类似 x=111 和 x="I'm a string"这样的代码，程序不会报错。

Python 非常适合面向对象的编程（OOP），因为它支持通过组合（composition）与继承(inheritance)的方式定义类(class)。Python 中没有访问说明符(access specifier，类似 C++中的 public 和 private），这么设计的依据是“大家都是成年人了”。

在 Python 语言中，函数是第一类对象（first-class objects）。这指的是它们可以被指定给变量，函数既能返回函数类型，也可以接受函数作为输入。类(class)也是第一类对象。

Python 代码编写快，但是运行速度比编译语言通常要慢。好在 Python 允许加入基于 C 语言编写的扩展，因此我们能够优化代码，消除瓶颈，这点通常是可以实现的。numpy 就是一个很好地例子，它的运行速度真的非常快，因为很多算术运算其实并不是通过 Python 实现的。

Python 用途非常广泛——网络应用，自动化，科学建模，大数据应用，等等。它也常被用作“胶水语言”，帮助其他语言和组件改善运行状况。

Python 让困难的事情变得容易，因此程序员可以专注于算法和数据结构的设计，而不用处理底层的细节。

33、补充缺失的代码

```
def  print_directory_contents(sPath):
        """
        这个函数接受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
```

答案：

```
def  print_directory_contents(sPath):
        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
```

34、阅读下面的代码，写出 A0，A1 至 An 的最终值。

```
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]
```

答案：

```
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]]
```

35、Python 是如何进行内存管理的？

答:从三个方面来说,一对象的引用计数机制,二垃圾回收机制,三内存池机制

一、对象的引用计数机制

Python 内部使用引用计数，来保持追踪内存中的对象，所有对象都有引用计数。

引用计数增加的情况：

1，一个对象分配一个新名称

2，将其放入一个容器中（如列表、元组或字典）

引用计数减少的情况：

1，使用 del 语句对对象别名显示的销毁

2，引用超出作用域或被重新赋值

sys.getrefcount( )函数可以获得对象的当前引用计数

多数情况下,引用计数比你猜测得要大得多。对于不可变数据(如数字和字符串)，解释器会在程序的不同部分共享内存，以便节约内存。

二、垃圾回收

1，当一个对象的引用计数归零时，它将被垃圾收集机制处理掉。

2，当两个对象 a 和 b 相互引用时，del 语句可以减少 a 和 b 的引用计数，并销毁用于引用底层对象的名称。然而由于每个对象都包含一个对其他对象的应用，

因此引用计数不会归零，对象也不会销毁。（从而导致内存泄露）。为解决这一问题，解释器会定期执行一个循环检测器，搜索不可访问对象的循环并删除它们。
三、内存池机制
Python 提供了对内存的垃圾收集机制，但是它将不用的内存放到内存池而不是返回给操作系统。
1，Pymalloc 机制。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。
2，Python 中所有小于 256 个字节的对象都使用 pymalloc 实现的分配器，而大的对象则使用系统的 malloc。
3，对于 Python 对象，如整数，浮点数和 List，都有其独立的私有内存池，对象间不共享他们的内存池。也就是说如果你分配又释放了大量的整数，用于缓存这些整数的内存就不能再分配给浮点数。

36、什么是 lambda 函数？它有什么好处?

答：lambda 表达式，通常是在需要一个函数，但是又不想费神去命名一个函数的场合下使用，也就是指匿名函数
lambda 函数：首要用途是指点短小的回调函数
lambda [arguments]:expression

```
>>> a=lambdax,y:x+y
>>> a(3,11)
```

37、Python 里面如何实现 tuple 和 list 的转换？

答：直接使用 tuple 和 list 函数就行了，type()可以判断对象的类型

38、请写出一段 Python 代码实现删除一个 list 里面的重复元素

答：
1. 使用 set 函数，set(list)
2. 使用字典函数，

```
>>>a=[1,2,4,2,4,5,6,5,7,8,9,0]
>>> b={}
>>>b=b.fromkeys(a)
>>>c=list(b.keys())
>>> c
```

39、Python 里面如何拷贝一个对象？

（赋值，浅拷贝，深拷贝的区别）
答：赋值（=），就是创建了对象的一个新的引用，修改其中任意一个变量都会影响到另一个。
浅拷贝：创建一个新的对象，但它包含的是对原始对象中包含项的引用（如果用引用的方式修改其中一个对象，另外一个也会修改改变）{1,完全切片方法；2，工厂函数，如 list()；3，copy 模块的 copy()函数}
深拷贝：创建一个新的对象，并且递归的复制它所包含的对象（修改其中一个，另外一个不会改变）{copy 模块的 deep.deepcopy()函数}

40、介绍一下 except 的用法和作用？

答：try…except…except…[else…][finally…]
执行 try 下的语句，如果引发异常，则执行过程会跳到 except 语句。对每个 except 分支顺序尝试执行，如果引发的异常与 except 中的异常组匹配，执行相应的语句。
如果所有的 except 都不匹配，则异常会传递到下一个调用本代码的最高层 try 代码中。
try 下的语句正常执行，则执行 else 块代码。如果发生异常，就不会执行
如果存在 finally 语句，最后总是会执行。

41、介绍一下 Python 下 range()函数的用法？

答：列出一组数据，经常用在 for in range()循环中

42、如何用 Python 来进行查询和替换一个文本字符串？

答：可以使用 re 模块中的 sub()函数或者 subn()函数来进行查询和替换，
格式：sub(replacement, string[,count=0])(replacement 是被替换成的文本，string 是需要被替换的文本，count 是一个可选参数，指最大被替换的数量）

```
>>> import re
>>>p=re.compile( ‘blue|white|red’ )
>>>print(p.sub( ‘colour’ ,’blue socks and red shoes’ ))
colour socks and colourshoes
>>>print(p.sub( ‘colour’ ,’blue socks and red shoes’ ,count=1))
colour socks and redshoes
```

subn()方法执行的效果跟 sub()一样，不过它会返回一个二维数组，包括替换后的新的字符串和总共替换的数量

43、Python 里面 match()和 search()的区别？

答：re 模块中 match(pattern,string[,flags]),检查 string 的开头是否与 pattern 匹配。

re 模块中 re.search(pattern,string[,flags]),在 string 搜索 pattern 的第一个匹配值。

```
>>>print(re.match(‘super’, ‘superstition’).span())
(0, 5)
>>>print(re.match(‘super’, ‘insuperable’))
None
>>>print(re.search(‘super’, ‘superstition’).span())
(0, 5)
>>>print(re.search(‘super’, ‘insuperable’).span())
(2, 7)
```

44、Python 里面如何生成随机数？

答：random 模块

随机整数：random.randint(a,b)：返回随机整数 x,a<=x<=b

random.randrange(start,stop,[,step])：返回一个范围在(start,stop,step)之间的随机整数，不包括结束值。

随机实数：random.random( ):返回 0 到 1 之间的浮点数

random.uniform(a,b):返回指定范围内的浮点数。

45、有没有一个工具可以帮助查找 python 的 bug 和进行静态的代码分析？

答：PyChecker 是一个 python 代码的静态分析工具，它可以帮助查找 python 代码的 bug, 会对代码的复杂度和格式提出警告

Pylint 是另外一个工具可以进行 codingstandard 检查

46、如何在一个 function 里面设置一个全局的变量？

答：解决方法是在 function 的开始插入一个 global 声明：

```
def  f():
    global  x
```

47、单引号，双引号，三引号的区别?

答：单引号和双引号是等效的，如果要换行，需要符号(\),三引号则可以直接换行，并且可以包含注释

如果要表示 Let’s go 这个字符串

单引号：s4 = ‘Let\’s go’

双引号：s5 = “Let’s go”

s6 = ‘I realy like“python”!’

这就是单引号和双引号都可以表示字符串的原因了

48、引用计数

PyObject 是每个对象必有的内容，其中 ob_refcnt 就是做为引用计数。当一个对象有新的引用时，它的 ob_refcnt 就会增加，当引用它的对象被删除，它的 ob_refcnt 就会减少.引用计数为 0 时，该对象生命就结束了。

优点:

简单

实时性

缺点:

维护引用计数消耗资源

循环引用

49、Python 的 List

列表是 Python 中最基本的数据结构，列表是最常用的 Python 数据类型，列表的数据项不需要具有相同的类型。列表中的每个元素都分配一个数字 - 它的位置，或索引，第一个索引是 0，第二个索引是 1，依此类推。

Python 有 6 个序列的内置类型，但最常见的是列表和元组。序列都可以进行的操作包括索引，切片，加，乘，检查成员。此外，Python 已经内置确定序列的长度以及确定最大和最小的元素的方法。

列表操作包含以下函数:

cmp(list1, list2)：比较两个列表的元素

len(list)：列表元素个数

max(list)：返回列表元素最大值

min(list)：返回列表元素最小值

list(seq)：将元组转换为列表

列表操作包含以下方法:

list.append(obj)：在列表末尾添加新的对象

list.count(obj)：统计某个元素在列表中出现的次数

list.extend(seq)：在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）

list.index(obj)：从列表中找出某个值第一个匹配项的索引位置

list.insert(index, obj)：将对象插入列表

list.pop(obj=list[-1])：移除列表中的一个元素（默认最后一个元素），并且返回该元素的值

list.remove(obj)：移除列表中某个值的第一个匹配项

list.reverse()：反向列表中元素

list.sort([func])：对原列表进行排序

50、Python 的 is

is 是对比地址,==是对比值

51、read,readline 和 readlines

read 读取整个文件
readline 读取下一行,使用生成器方法
readlines 读取整个文件到一个迭代器以供我们遍历

52、Python2 和 3 的区别

print 不再是语句，而是函数，比如原来是 print 'abc' 现在是 print('abc')
但是 python2.6+ 可以使用 from __future__ import print_function 来实现相同功能

在 Python 3 中，没有旧式类，只有新式类，也就是说不用再像这样 class Foobar(object): pass 显式地子类化 object
但是最好还是加上. 主要区别在于 old-style 是 classtype 类型而 new-style 是 type 类型

原来 1/2（两个整数相除）结果是 0，现在是 0.5 了
python 2.2+ 以上都可以使用 from __future__ import division 实现改特性, 同时注意 // 取代了之前的 / 运算

新的字符串格式化方法 format 取代%
错误, 从 python2.6+ 开始已经在 str 和 unicode 中有该方法, 同时 python3 依然支持 % 算符

xrange 重命名为 range
同时更改的还有一系列内置函数及方法, 都返回迭代器对象, 而不是列表或者元组, 比如 filter, map, dict.items 等

!=取代 <>
python2 也很少有人用 <> 所以不算什么修改

long 重命名为 int
不完全对, python3 彻底废弃了 long+int 双整数实现的方法, 统一为 int , 支持高精度整数运算.

except Exception, e 变成 except (Exception) as e
只有 python2.5 及以下版本不支持该语法. python2.6 是支持的. 不算新东西

exec 变成函数
类似 print() 的变化, 之前是语句.

53、下面代码会输出什么：

```
def  f(x, l=[]):
```

```
        for  i  in  range(x):
                l.append(i*i)
        print  l
f(2)
f(3,[3,2,1])
f(3)
```

答案：

```
[0,  1]
[3,  2,  1,  0,  1,  4]
[0,  1,  0,  1,  4]
```

54、Python 和多线程（multi-threading）。这是个好主意码？列举一些让 Python 代码以并行方式运行的方法。

答案：

Python 并不支持真正意义上的多线程。Python 中提供了多线程包，但是如果你想通过多线程提高代码的速度，使用多线程包并不是个好主意。Python 中有一个被称为 Global Interpreter Lock（GIL）的东西，它会确保任何时候你的多个线程中，只有一个被执行。线程的执行速度非常之快，会让你误以为线程是并行执行的，但是实际上都是轮流执行。经过 GIL 这一道关卡处理，会增加执行的开销。这意味着，如果你想提高代码的运行速度，使用 threading 包并不是一个很好的方法。

不过还是有很多理由促使我们使用 threading 包的。如果你想同时执行一些任务，而且不考虑效率问题，那么使用这个包是完全没问题的，而且也很方便。但是大部分情况下，并不是这么一回事，你会希望把多线程的部分外包给操作系统完成（通过开启多个进程），或者是某些调用你的 Python 代码的外部程序（例如 Spark 或 Hadoop），又或者是你的 Python 代码调用的其他代码（例如，你可以在 Python 中调用 C 函数，用于处理开销较大的多线程工作）。

55、“猴子补丁”（monkey patching）指的是什么？这种做法好吗？

答案：

“猴子补丁”就是指，在函数或对象已经定义之后，再去改变它们的行为。

举个例子：

```
import  datetime
datetime.datetime.now  =  lambda:  datetime.datetime(2012,  12,  12)
```

大部分情况下，这是种很不好的做法 - 因为函数在代码库中的行为最好是都保持一致。打“猴子补丁”的原因可能是为了测试。mock 包对实现这个目的很有帮助。

56、这两个参数是什么意思：*args，**kwargs？我们为什么要使用它们？

答案：

如果我们不确定要往函数中传入多少个参数，或者我们想往函数中以列表和元组的形式传参数时，那就使要用*args；

如果我们不知道要往函数中传入多少个关键词参数，或者想传入字典的值作为关键词参数时，那就要使用**kwargs。

args 和 kwargs 这两个标识符是约定俗成的用法，你当然还可以用*bob 和**billy，但是这样就并不太妥。

下面是具体的示例：

```
def  f(*args,**kwargs):  print  args,  kwargs
l = [1,2,3]
t = (4,5,6)
d = {'a':7,'b':8,'c':9}
f()
f(1,2,3)                # (1, 2, 3) {}
f(1,2,3,"groovy")         # (1, 2, 3, 'groovy') {}
f(a=1,b=2,c=3)            # () {'a': 1, 'c': 3, 'b': 2}
f(a=1,b=2,c=3,zzz="hi")     # () {'a': 1, 'c': 3, 'b': 2, 'zzz': 'hi'}
f(1,2,3,a=1,b=2,c=3)       # (1, 2, 3) {'a': 1, 'c': 3, 'b': 2}
f(*l,**d)                # (1, 2, 3) {'a': 7, 'c': 9, 'b': 8}
f(*t,**d)                # (4, 5, 6) {'a': 7, 'c': 9, 'b': 8}
f(1,2,*t)                # (1, 2, 4, 5, 6) {}
f(q="winning",**d)         # () {'a': 7, 'q': 'winning', 'c': 9, 'b': 8}
f(1,2,*t,q="winning",**d)   # (1, 2, 4, 5, 6) {'a': 7, 'q': 'winning', 'c': 9, 'b': 8}
def f2(arg1,arg2,*args,**kwargs): print arg1,arg2, args, kwargs
f2(1,2,3)                 # 1 2 (3,) {}
f2(1,2,3,"groovy")           # 1 2 (3, 'groovy') {}
f2(arg1=1,arg2=2,c=3)         # 1 2 () {'c': 3}
f2(arg1=1,arg2=2,c=3,zzz="hi")  # 1 2 () {'c': 3, 'zzz': 'hi'}
f2(1,2,3,a=1,b=2,c=3)        # 1 2 (3,) {'a': 1, 'c': 3, 'b': 2}
f2(*l,**d)                # 1 2 (3,) {'a': 7, 'c': 9, 'b': 8}
f2(*t,**d)                # 4 5 (6,) {'a': 7, 'c': 9, 'b': 8}
f2(1,2,*t)                # 1 2 (4, 5, 6) {}
f2(1,1,q="winning",**d)     # 1 1 () {'a': 7, 'q': 'winning', 'c': 9, 'b': 8}
f2(1,2,*t,q="winning",**d)   # 1 2 (4, 5, 6) {'a': 7, 'q': 'winning', 'c': 9, 'b': 8}
```

57、下面这些是什么意思：@classmethod, @staticmethod, @property ？

答案：

@classmethod, @staticmethod 和@property 这三个装饰器的使用对象是在类中定义的函数。下面的例子展示了它们的用法和行为：

```
class MyClass(object):
        def __init__(self):
                self._some_property = "properties are nice"
                self._some_other_property = "VERY nice"
        def normal_method(*args,**kwargs):
                print "calling normal_method({0},{1})".format(args,
kwargs)
        @classmethod
        def class_method(*args,**kwargs):
                print "calling class_method({0},{1})".format(args,k
wargs)
        @staticmethod
        def static_method(*args,**kwargs):
                print "calling static_method({0},{1})".format(args,
kwargs)
        @property
        def some_property(self,*args,**kwargs):
                print "calling some_property getter({0},{1},{2})".
format(self,args,kwargs)
                return self._some_property
        @some_property.setter
        def some_property(self,*args,**kwargs):
                print "calling some_property setter({0},{1},{2})".
format(self,args,kwargs)
                self._some_property = args[0]
        @property
        def some_other_property(self,*args,**kwargs):
                print "calling some_other_property getter({0},{1},
{2})".format(self,args,kwargs)
                return self._some_other_property
o = MyClass()
# 未装饰的方法还是正常的行为方式，需要当前的类实例（self）作为第一个参数。
o.normal_method
# <bound method MyClass.normal_method of <__main__.MyClass inst
ance at 0x7fdd2537ea28>>
o.normal_method()
# normal_method((<__main__.MyClass instance at 0x7fdd2537ea28>,),
{})
o.normal_method(1,2,x=3,y=4)
# normal_method((<__main__.MyClass instance at 0x7fdd2537ea28>,
1, 2),{'y': 4, 'x': 3})
```

```
#  类方法的第一个参数永远是该类
o.class_method
#  <bound method classobj.class_method of <class __main__.MyClas
s at 0x7fdd2536a390>>
o.class_method()
#  class_method((<class __main__.MyClass at 0x7fdd2536a390>,),{})
o.class_method(1,2,x=3,y=4)
#  class_method((<class __main__.MyClass at 0x7fdd2536a390>, 1,
2),{'y': 4, 'x': 3})
#  静态方法（static method）中除了你调用时传入的参数以外，没有其他的
参数。
o.static_method
#  <function static_method at 0x7fdd25375848>
o.static_method()
#  static_method((),{})
o.static_method(1,2,x=3,y=4)
#  static_method((1, 2),{'y': 4, 'x': 3})
#  @property 是实现 getter 和 setter 方法的一种方式。直接调用它们是错误
的。
#  “只读”属性可以通过只定义 getter 方法，不定义 setter 方法实现。
o.some_property
#  调用 some_property 的
getter(<__main__.MyClass instance at 0x7fb2b70877e8>,(),{})
#  'properties are nice'
#  “属性”是很好的功能
o.some_property()
#  calling some_property getter(<__main__.MyClass instance at 0x
7fb2b70877e8>,(),{})
#  Traceback (most recent call last):
#    File "<stdin>", line 1, in <module>
#  TypeError: 'str' object is not callable
o.some_other_property
#  calling some_other_property getter(<__main__.MyClass instance
at 0x7fb2b70877e8>,(),{})
#  'VERY nice'
#  o.some_other_property()
#  calling some_other_property getter(<__main__.MyClass instance
at 0x7fb2b70877e8>,(),{})
#  Traceback (most recent call last):
#    File "<stdin>", line 1, in <module>
#  TypeError: 'str' object is not callable
o.some_property = "groovy"
#  calling some_property setter(<__main__.MyClass object at 0x7f
b2b7077890>,('groovy',),{})
```

```
o.some_property
# calling some_property getter(<__main__.MyClass object at 0x7f
b2b7077890>,(),{})
# 'groovy'
o.some_other_property = "very groovy"
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# AttributeError: can't set attribute
o.some_other_property
# calling some_other_property getter(<__main__.MyClass object at
  0x7fb2b7077890>,(),{})
```

58、阅读下面的代码，它的输出结果是什么？

```
class Node(object):
    def __init__(self,sName):
        self._lChildren = []
        self.sName = sName
    def __repr__(self):
        return "<Node '{}'>".format(self.sName)
    def append(self,*args,**kwargs):
        self._lChildren.append(*args,**kwargs)
    def print_all_1(self):
        print self
        for oChild in self._lChildren:
        oChild.print_all_1()
     def print_all_2(self):
        def gen(o):
            lAll = [o,]
            while lAll:
                oNext = lAll.pop(0)
                lAll.extend(oNext._lChildren)
                yield oNext
        for oNode in gen(self):
            print oNode
oRoot = Node("root")
oChild1 = Node("child1")
oChild2 = Node("child2")
oChild3 = Node("child3")
oChild4 = Node("child4")
oChild5 = Node("child5")
oChild6 = Node("child6")
oChild7 = Node("child7")
```

```
oChild8 = Node("child8")
oChild9 = Node("child9")
oChild10 = Node("child10")
oRoot.append(oChild1)
oRoot.append(oChild2)
oRoot.append(oChild3)
oChild1.append(oChild4)
oChild1.append(oChild5)
oChild2.append(oChild6)
oChild4.append(oChild7)
oChild3.append(oChild8)
oChild3.append(oChild9)
oChild6.append(oChild10)
# 说明下面代码的输出结果
oRoot.print_all_1()
oRoot.print_all_2()
```

答案：

oRoot.print_all_1()会打印下面的结果：

```
<Node 'root'>
<Node 'child1'>
<Node 'child4'>
<Node 'child7'>
<Node 'child5'>
<Node 'child2'>
<Node 'child6'>
<Node 'child10'>
<Node 'child3'>
<Node 'child8'>
<Node 'child9'>
```

oRoot.print_all_1()会打印下面的结果：

```
<Node 'root'>
<Node 'child1'>
<Node 'child2'>
<Node 'child3'>
<Node 'child4'>
<Node 'child5'>
<Node 'child6'>
<Node 'child8'>
<Node 'child9'>
<Node 'child7'>
<Node 'child10'>
```

59、简要描述 Python 的垃圾回收机制（garbage collection）。

答案：

Python 在内存中存储了每个对象的引用计数（reference count）。如果计数值变成 0，那么相应的对象就会小时，分配给该对象的内存就会释放出来用作他用。偶尔也会出现引用循环（reference cycle）。垃圾回收器会定时寻找这个循环，并将其回收。举个例子，假设有两个对象 o1 和 o2，而且符合 o1.x == o2 和 o2.x == o1 这两个条件。如果 o1 和 o2 没有其他代码引用，那么它们就不应该继续存在。但它们的引用计数都是 1。

Python 中使用了某些启发式算法（heuristics）来加速垃圾回收。例如，越晚创建的对象更有可能被回收。对象被创建之后，垃圾回收器会分配它们所属的代（generation）。每个对象都会被分配一个代，而被分配更年轻代的对象是优先被处理的

60、简述函数式编程

在函数式编程中，函数是基本单位，变量只是一个名称，而不是一个存储单元。除了匿名函数外，Python 还使用 fliter(),map(),reduce(),apply()函数来支持函数式编程。

61 、什么是匿名函数，匿名函数有什么局限性

匿名函数，也就是 lambda 函数，通常用在函数体比较简单的函数上。匿名函数顾名思义就是函数没有名字，因此不用担心函数名冲突。不过 Python 对匿名函数的支持有限，只有一些简单的情况下可以使用匿名函数。

62、如何捕获异常，常用的异常机制有哪些？

如果我们没有对异常进行任何预防，那么在程序执行的过程中发生异常，就会中断程序，调用 python 默认的异常处理器，并在终端输出异常信息。

try...except...finally 语句:当 try 语句执行时发生异常，回到 try 语句层，寻找后面是否有 except 语句。找到 except 语句后，会调用这个自定义的异常处理器。except 将异常处理完毕后，程序继续往下执行。finally 语句表示，无论异常发生与否，finally 中的语句都要执行。

assert 语句：判断 assert 后面紧跟的语句是 True 还是 False，如果是 True 则继续执行 print，如果是 False 则中断程序，调用默认的异常处理器，同时输出 assert 语句逗号后面的提示信息。

with 语句:如果 with 语句或语句块中发生异常,会调用默认的异常处理器处理，

但文件还是会正常关闭。

63 、copy()与 deepcopy()的区别?

copy 是浅拷贝，只拷贝可变对象的父级元素。 deepcopy 是深拷贝，递归拷贝可变对象的所有元素。

64、函数装饰器有什么作用（常考）?

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。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、权限校验等场景。有了装饰器，就可以抽离出大量与函数功能本身无关的雷同代码并继续重用。

65、简述 Python 的作用域以及 Python 搜索变量的顺序?

Python 作用域简单说就是一个变量的命名空间。代码中变量被赋值的位置，就决定了哪些范围的对象可以访问这个变量，这个范围就是变量的作用域。在 Python 中，只有模块（module），类（class）以及函数（def、lambda）才会引入新的作用域。Python 的变量名解析机制也称为 LEGB 法则：本地作用域(Local）→当前作用域被嵌入的本地作用域（Enclosing locals）→全局/模块作用域（Global）→内置作用域（Built-in）

66、新式类和旧式类的区别,如何确保使用的类是新式类?

为了统一类(class)和类型(type)，python 在 2.2 版本引进来新式类。在 2.1 版本中，类和类型是不同的。
为了确保使用的是新式类，有以下方法：
放在类模块代码的最前面 __metaclass__ = type
从内建类 object 直接或者间接地继承
在 python3 版本中，默认所有的类都是新式类。

67、有用过 with statement 吗？它的好处是什么？具体如何实现？

with 语句适用于对资源进行访问的场合，确保不管使用过程中是否发生异常都会执行必要的“清理”操作，释放资源，比如文件使用后自动关闭、线程中锁的自动获取和释放等。

68、什么是 PEP8？

PEP8 是一个编程规范，内容是一些关于如何让你的程序更具可读性的建议。

69、什么是 pickling 和 unpickling？

Pickle 模块读入任何 Python 对象，将它们转换成字符串，然后使用 dump 函数将其转储到一个文件中——这个过程叫做 pickling。
反之从存储的字符串文件中提取原始 Python 对象的过程，叫做 unpickling。

70、Python 是如何被解释的？

Python 是一种解释性语言，它的源代码可以直接运行。Python 解释器会将源代码转换成中间语言，之后再翻译成机器码再执行。

71、有哪些工具可以帮助 debug 或做静态分析？

PyChecker 是一个静态分析工具，它不仅能报告源代码中的错误，并且会报告错误类型和复杂度。Pylint 是检验模块是否达到代码标准的另一个工具。

72、什么是 Python 装饰器？

Python 装饰器是 Python 中的特有变动，可以使修改函数变得更容易。

73、数组和元组之间的区别是什么？

数组和元组之间的区别:数组内容是可以被修改的,而元组内容是只读的。另外，元组可以被哈希，比如作为字典的关键字。

74、参数按值传递和引用传递是怎样实现的？

Python 中的一切都是类，所有的变量都是一个对象的引用。引用的值是由函数确定的,因此无法被改变。但是如果一个对象是可以被修改的,你可以改动对象。

75、字典推导式和列表推导式是什么？

它们是可以轻松创建字典和列表的语法结构。

76、Python 都有哪些自带的数据结构？

Python 自带的数据结构分为可变的和不可变的。可变的有：数组、集合、字典；不可变的有：字符串、元组、数。

77、什么是 Python 的命名空间？

在 Python 中，所有的名字都存在于一个空间中，它们在该空间中存在和被操作——这就是命名空间。
它就好像一个盒子，每一个变量名字都对应装着一个对象。当查询变量的时候，会从该盒子里面寻找相应的对象。

78、Python 中的 lambda 是什么？

这是一个常被用于代码中的单个表达式的匿名函数。

79、为什么 lambda 没有语句？

匿名函数 lambda 没有语句的原因，是它被用于在代码被执行的时候构建新的函数对象并且返回。

80、Python 中的 pass 是什么？

Pass 是一个在 Python 中不会被执行的语句。在复杂语句中，如果一个地方需要暂时被留白，它常常被用于占位符。

81、Python 中什么是遍历器？

遍历器用于遍历一组元素，比如列表这样的容器。

82、Python 中的 unittest 是什么？

在 Python 中，unittest 是 Python 中的单元测试框架。它拥有支持共享搭建、自动测试、在测试中暂停代码、将不同测试迭代成一组，等等的功能。

83、在 Python 中什么是 slicing？

Slicing 是一种在有序的对象类型中(数组,元组,字符串)节选某一段的语法。

84、在 Python 中什么是构造器？

生成器是实现迭代器的一种机制。它功能的实现依赖于 yield 表达式，除此之外它跟普通的函数没有两样。

85、Python 中的 docstring 是什么？

Python 中文档字符串被称为 docstring，它在 Python 中的作用是为函数、模块和类注释生成文档。

86、如何在 Python 中拷贝一个对象？

如果要在 Python 中拷贝一个对象，大多时候你可以用 copy.copy()或者 copy.deepcopy()。但并不是所有的对象都可以被拷贝。

87、Python 中的负索引是什么？

Python 中的序列索引可以是正也可以是负。如果是正索引，0 是序列中的第一个索引，1 是第二个索引。如果是负索引，（-1）是最后一个索引而（-2）是倒数第二个索引。

88、如何将一个数字转换成一个字符串？

你可以使用自带函数 str()将一个数字转换为字符串。如果你想要八进制或者十六进制数，可以用 oct()或 hex()。

89、Xrange 和 range 的区别是什么？

Xrange 用于返回一个 xrange 对象，而 range 用于返回一个数组。不管那个范围多大，Xrange 都使用同样的内存。

90、Python 中的模块和包是什么？

在 Python 中，模块是搭建程序的一种方式。每一个 Python 代码文件都是一个模块，并可以引用其他的模块，比如对象和属性。
一个包含许多 Python 代码的文件夹是一个包。一个包可以包含模块和子文件夹。

91、输入一个整数，输出该数二进制表示中 1 的个数。其中负数用补码表示。

```
def  getOneCount(num):
        if  num  >  0:
                count  =  b_num.count('1')
                print(b_num)
                return  count
        elif  num  <  0:
                b_num  =  bin(~num)
                count  =  8  -  b_num.count('1')
                return  count
        else:
                return  8
if  __name__  ==  '__main__':
        print(getOneCount(5))
```

```
        print(getOneCount(-5))
        print(getOneCount(0))
```

92、获取最大公约数、最小公倍数

```
a = 36
b = 21

def maxCommon(a, b):
        while b: a,b = b, a%b
        return a

def minCommon(a, b):
        c = a*b
        while b: a,b = b, a%b
        return c//a

if __name__ == '__main__':
        print(maxCommon(a,b))
        print(minCommon(a,b))
```

93、获取中位数

```
def median(data):
        data.sort()
        half = len(data) // 2
        return (data[half] + data[~half])/2

l = [1,3,4,53,2,46,8,42,82]

if __name__ == '__main__':
print(median(l))
```

94、杨氏矩阵查找

在一个 m 行 n 列二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

95、去除列表中的重复元素

```
#用集合

list(set(l))
```

```
#用字典
l1 = ['b','c','d','b','c','a','a']
l2 = {}.fromkeys(l1).keys()
print l2
1
2
3
l1 = ['b','c','d','b','c','a','a']
l2 = {}.fromkeys(l1).keys()
print l2
#用字典并保持顺序
l1 = ['b','c','d','b','c','a','a']
l2 = list(set(l1))
l2.sort(key=l1.index)
print l2
#列表推导式
l1 = ['b','c','d','b','c','a','a']
l2 = []
[l2.append(i) for i in l1 if not i in l2]
```

96、矩形覆盖

我们可以用 2*1 的小矩形横着或者竖着去覆盖更大的矩形。请问用 n 个 2*1 的小矩形无重叠地覆盖一个 2*n 的大矩形，总共有多少种方法？
第 2*n 个矩形的覆盖方法等于第 2*(n-1)加上第 2*(n-2)的方法。

```
f = lambda n: 1 if n < 2 else f(n - 1) + f(n - 2)
```

97、变态台阶问题

一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级……它也可以跳上 n 级。求该

青蛙跳上一个 n 级的台阶总共有多少种跳法。
1 台阶问题/斐波纳挈
一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

```
fib = lambda n: n if n <= 2 else fib(n - 1) + fib(n - 2)
```

第二种记忆方法

```
def memo(func):
    cache = {}
def wrap(*args):
    if args not in cache:
        cache[args] = func(*args)
        return cache[args]
    return wrap
@ memo
def fib(i):
    if i < 2:
        return 1
    return fib(i-1) + fib(i-2)
```

第三种方法

```
def fib(n):
    a, b = 0, 1
    for _ in xrange(n):
        a, b = b, a + b
        return b
```

98、标记-清除机制

基本思路是先按需分配，等到没有空闲内存的时候从寄存器和程序栈上的引用出发，遍历以对象为节点、以引用为边构成的图，把所有可以访问到的对象打上标记，然后清扫一遍内存空间，把所有没标记的对象释放。

99、阅读下面的代码，它的输出结果是什么？

```
class  A(object):
        def  go(self):
                print  "go  A  go!"
        def  stop(self):
                print  "stop  A  stop!"
        def  pause(self):
                raise  Exception("Not  Implemented")
class  B(A):
        def  go(self):
                super(B,  self).go()
```

```
                print  "go  B  go!"
class  C(A):
        def  go(self):
                super(C,  self).go()
                print  "go  C  go!"
        def  stop(self):
                super(C,  self).stop()
                print  "stop  C  stop!"
class  D(B,C):
        def  go(self):
                super(D,  self).go()
                print  "go  D  go!"
        def  stop(self):
                super(D,  self).stop()
                print  "stop  D  stop!"
        def  pause(self):
                print  "wait  D  wait!"
class  E(B,C):  pass
a  =  A()
b  =  B()
c  =  C()
d  =  D()
e  =  E()
#  说明下列代码的输出结果
a.go()
b.go()
c.go()
d.go()
e.go()
a.stop()
b.stop()
c.stop()
d.stop()
e.stop()
a.pause()
b.pause()
c.pause()
d.pause()
e.pause()
```

答案：

输出结果以注释的形式表示：

```
a.go()
#  go  A  go!
b.go()
```

```
#  go  A  go!
#  go  B  go!
c.go()
#  go  A  go!
#  go  C  go!
d.go()
#  go  A  go!
#  go  C  go!
#  go  B  go!
#  go  D  go!

e.go()
#  go  A  go!
#  go  C  go!
#  go  B  go!
a.stop()
#  stop  A  stop!
b.stop()
#  stop  A  stop!
c.stop()
#  stop  A  stop!
#  stop  C  stop!
d.stop()
#  stop  A  stop!
#  stop  C  stop!
#  stop  D  stop!
e.stop()
#  stop  A  stop!
a.pause()
#  ..  Exception:  Not  Implemented
b.pause()
#  ...  Exception:  Not  Implemented
c.pause()
#  ...  Exception:  Not  Implemented
d.pause()
#  wait  D  wait!
e.pause()
#  ...Exception:  Not  Implemented
```

第二阶段

100、GIL 线程全局锁

线程全局锁(Global Interpreter Lock),即 Python 为了保证线程安全而采取的独立线程运行的限制,说白了就是一个核只能在同一时间运行一个线程.

101、协程

简单点说协程是进程和线程的升级版,进程和线程都面临着内核态和用户态的切换问题而耗费许多切换时间,而协程就是用户自己控制切换的时机,不再需要陷入系统的内核态.

102、select,poll 和 epoll

其实所有的 I/O 都是轮询的方法,只不过实现的层面不同罢了.
这个问题可能有点深入了,但相信能回答出这个问题是对 I/O 多路复用有很好的了解了.其中 tornado 使用的就是 epoll 的.
selec,poll 和 epoll 区别总结
基本上 select 有 3 个缺点:
连接数受限
查找配对速度慢
数据由内核拷贝到用户态
poll 改善了第一个缺点
epoll 改了三个缺点.

103、调度算法

先来先服务(FCFS, First Come First Serve)
短作业优先(SJF, Shortest Job First)
最高优先权调度(Priority Scheduling)
时间片轮转(RR, Round Robin)
多级反馈队列调度(multilevel feedback queue scheduling)
实时调度算法:
最早截至时间优先 EDF
最低松弛度优先 LLF

104、死锁

原因:
竞争资源
程序推进顺序不当
必要条件:
互斥条件
请求和保持条件

不剥夺条件
环路等待条件
处理死锁基本方法:
预防死锁(摒弃除 1 以外的条件)
避免死锁(银行家算法)
检测死锁(资源分配图)
解除死锁
剥夺资源
撤销进程
数据库

105、事务

数据库事务(Database Transaction) ，是指作为单个逻辑工作单元执行的一系列操作，要么完全地执行，要么完全地不执行。

106、数据库索引

MySQL 索引背后的数据结构及算法原理
聚集索引,非聚集索引,B-Tree,B+Tree,最左前缀原理

107、Redis 原理

redis 是一个 key-value 存储系统. 和 Memcached 类似，它支持存储的 value 类型相对更多，包括 string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和 hashs（哈希类型）

这些数据类型都支持 push/pop、add/remove 及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的.

在此基础上，redis 支持各种不同方式的排序. 与 memcached 一样，为了保证效率，数据都是缓存在内存中.

区别的是 redis 会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了 master-slave(主从)同步.

108、乐观锁和悲观锁

悲观锁：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作
乐观锁：假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性。

109、MVCC

Multi-Version Concurrency Control 多版本并发控制，MVCC 是一种并发控制的方法，一般在数据库管理系统中，实现对数据库的并发访问；在编程语言中实现事务内存。

110、MyISAM 和 InnoDB

MyISAM 适合于一些需要大量查询的应用，但其对于有大量写操作并不是很好。甚至你只是需要 update 一个字段，整个表都会被锁起来，而别的进程，就算是读进程都无法操作直到读操作完成。另外，MyISAM 于 SELECT COUNT(*) 这类的计算是超快无比的。

InnoDB 的趋势会是一个非常复杂的存储引擎，对于一些小的应用，它会比 MyISAM 还慢。他是它支持“行锁” ，于是在写操作比较多的时候，会更优秀。并且，他还支持更多的高级应用，比如：事务。

111、将下面的函数按照执行效率高低排序。它们都接受由 0 至 1 之间的数字构成的列表作为输入。这个列表可以很长。一个输入列表的示例如下：[random.random() for i in range(100000)]。你如何证明自己的答案是正确的。

```
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  lIn  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)]
```

答案：

```
import  cProfile
lIn  =  [random.random()  for  i  in  range(100000)]
```

```
cProfile.run('f1(lIn)')
cProfile.run('f2(lIn)')
cProfile.run('f3(lIn)')
```

为了向大家进行完整地说明，下面我们给出上述分析代码的输出结果：

```
>>> cProfile.run('f1(lIn)')
         4 function calls in 0.045 seconds
   Ordered by: standard name
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.009    0.009    0.044    0.044 <stdin>:1(f1)
        1    0.001    0.001    0.045    0.045 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        1    0.035    0.035    0.035    0.035 {sorted}
>>> cProfile.run('f2(lIn)')
         4 function calls in 0.024 seconds
   Ordered by: standard name
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.008    0.008    0.023    0.023 <stdin>:1(f2)
        1    0.001    0.001    0.024    0.024 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        1    0.016    0.016    0.016    0.016 {sorted}
>>> cProfile.run('f3(lIn)')
         4 function calls in 0.055 seconds
   Ordered by: standard name
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.016    0.016    0.054    0.054 <stdin>:1(f3)
        1    0.001    0.001    0.055    0.055 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        1    0.038    0.038    0.038    0.038 {sorted}
```

第三阶段

112、用 Python 匹配 HTML tag 的时候，<.*>和<.*?>有什么区别？

答：术语叫贪婪匹配(<.*>)和非贪婪匹配(<.*?>)
例如:
test
<.*> :
test
<.*?> :

113、三次握手

客户端通过向服务器端发送一个 SYN 来创建一个主动打开，作为三路握手的一部分。客户端把这段连接的序号设定为随机数?A。
服务器端应当为一个合法的 SYN 回送一个 SYN/ACK。ACK?的确认码应为?A+1，SYN/ACK?包本身又有一个随机序号?B。
最后，客户端再发送一个 ACK。当服务端受到这个 ACK 的时候，就完成了三路握手，并进入了连接创建状态。此时包序号被设定为收到的确认号?A+1，而响应则为 B+1。

114、四次挥手

由于 TCP 连接是全双工的，因此每个方向都必须单独进行关闭。这原则是当一方完成它的数据发送任务后就能发送一个 FIN 来终止这个方向的连接。收到一个 FIN 只意味着这一方向上没有数据流动，一个 TCP 连接在收到一个 FIN 后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

a. TCP 客户端发送一个 FIN，用来关闭客户到服务器的数据传送。

b. 服务器收到这个 FIN，它发回一个 ACK，确认序号为收到的序号加 1。和 SYN 一样，一个 FIN 将占用一个序号。

c. 服务器关闭客户端的连接，发送一个 FIN 给客户端。

d. 客户端发回 ACK 报文确认，并将确认序号设置为收到序号加 1。

115、ARP 协议

地址解析协议(Address Resolution Protocol):根据 IP 地址获取物理地址的一个 TCP/IP 协议

116、urllib 和 urllib2 的区别

urllib 提供 urlencode 方法用来 GET 查询字符串的产生，而 urllib2 没有。这是为何 urllib 常和 urllib2 一起使用的原因。
urllib2 可以接受一个 Request 类的实例来设置 URL 请求的 headers，urllib 仅

可以接受 URL。这意味着，你不可以伪装你的 User?Agent 字符串等。

117、Post 和 Get

下面的表格比较了两种 HTTP 方法：GET 和 POST。

	GET	POST
后退按钮/刷新	无害	数据会被重新提交（浏览器应该告知用户数据会被重新提交）。
书签	可收藏为书签	不可收藏为书签
缓存	能被缓存	不能缓存
编码类型	application/x-www-form-urlencoded	application/x-www-form-urlencoded 或 multipart/form-data。为二进制数据使用多重编码。
历史	参数保留在浏览器历史中。	参数不会保存在浏览器历史中。
对数据长度的限制	是的。当发送数据时，GET 方法向 URL 添加数据；URL 的长度是受限制的（URL 的最大长度是 2048 个字符）。	无限制。
对数据类型的限制	只允许 ASCII 字符。	没有限制。也允许二进制数据。
安全性	与 POST 相比，GET 的安全性较差，因为所发送的数据是 URL 的一部分。 在发送密码或其他敏感信息时绝不要使用 GET！	POST 比 GET 更安全，因为参数不会被保存在浏览器历史或 web 服务器日志中。
可见性	数据在 URL 中对所有人都是可见的。	数据不会显示在 URL 中。

118、Cookie 和 Session

Cookie Session 储存位置 客户端 服务器端
目的 跟踪会话，也可以保存用户偏好设置或者保存用户名密码等 跟踪会话
安全性 不安全 安全
session 技术是要使用到 cookie 的，之所以出现 session 技术，主要是为了安全。

119、apache 和 nginx 的区别

Nginx 相对 apache?的优点：
轻量级，同样起 web 服务，比 apache 占用更少的内存及资源
抗并发，nginx 处理请求是异步非阻塞的，支持更多的并发连接，而 apache 则是阻塞型的，在高并发下 nginx 能保持低资源低消耗高性能，配置简洁，高度模块化的设计，编写模块相对简单
Apache 相对 nginx 的优点：
Rewrite ，比 nginx 的 rewrite 强大
模块超多，基本想到的都可以找到
少 bug ，nginx 的 bug 相对较多
超稳定

120、网站用户密码保存

明文保存
明文 hash 后保存,如 md5
MD5+Salt 方式,这个 salt 可以随机
知乎使用了 Bcrypy(好像)加密

121、HTTP 和 HTTPS

状态码 定义
1xx 报告 接收到请求，继续进程
2xx 成功 步骤成功接收，被理解，并被接受
3xx 重定向 为了完成请求,必须采取进一步措施
4xx 客户端出错 请求包括错的顺序或不能完成
5xx 服务器出错 服务器无法完成显然有效的请求
403: Forbidden
404: Not?Found
HTTPS 握手,对称加密,非对称加密,TLS/SSL,RSA

122、XSRF 和 XSS

CSRF(Cross-site?request?forgery)跨站请求伪造
XSS(Cross?Site?Scripting)跨站脚本攻击
CSRF 重点在请求,XSS 重点在脚本

123、幂等 Idempotence

HTTP 方法的幂等性是指一次和多次请求某一个资源应该具有同样的副作用。(注意是副作用)
GET http://www.bank.com/account/123456，不会改变资源的状态，不论调用一次还是 N 次都没有副作用。请注意,这里强调的是一次和 N 次具有相同的副作用，而不是每次 GET 的结果相同。GET http://www.news.com/latest-news 这个 HTTP 请求可能会每次得到不同的结果，但它本身并没有产生任何副作用，因而是满足幂等性的。
DELETE 方法用于删除资源，有副作用，但它应该满足幂等性。比如：DELETE http://www.forum.com/article/4231，调用一次和 N 次对系统产生的副作用是相同的，即删掉 id 为 4231 的帖子；因此，调用者可以多次调用或刷新页面而不必担心引起错误。
POST 所对应的 URI 并非创建的资源本身，而是资源的接收者。比如：POST http://www.forum.com/articles 的语义是在 http://www.forum.com/articles 下创建一篇帖子，HTTP 响应中应包含帖子的创建状态以及帖子的 URI。两次相同的 POST 请求会在服务器端创建两份资源，它们具有不同的 URI；所以，POST 方法不具备幂等性。
PUT 所对应的 URI 是要创建或更新的资源本身。比如：PUT

http://www.forum/articles/4231 的语义是创建或更新 ID 为 4231 的帖子。对同一 URI 进行多次 PUT 的副作用和一次 PUT 是相同的；因此，PUT 方法具有幂等性。

124、SOAP

SOAP（原为 Simple Object Access Protocol 的首字母缩写，即简单对象访问协议）是交换数据的一种协议规范，使用在计算机网络 Web 服务（web service）中，交换带结构信息。SOAP 为了简化网页服务器（Web Server）从 XML 数据库中提取数据时，节省去格式化页面时间，以及不同应用程序之间按照 HTTP 通信协议，遵从 XML 格式执行资料互换，使其抽象于语言实现、平台和硬件。

125、RPC

RPC（Remote Procedure Call Protocol）——远程过程调用协议，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。RPC 协议假定某些传输协议的存在，如 TCP 或 UDP，为通信程序之间携带信息数据。在 OSI 网络通信模型中，RPC 跨越了传输层和应用层。RPC 使得开发包括网络分布式多程序在内的应用程序更加容易。
总结:服务提供的两大流派.传统意义以方法调用为导向通称 RPC。为了企业 SOA,若干厂商联合推出 webservice,制定了 wsdl 接口定义,传输 soap.当互联网时代,臃肿 SOA 被简化为 http+xml/json.但是简化出现各种混乱。以资源为导向,任何操作无非是对资源的增删改查，于是统一的 REST 出现了.
进化的顺序: RPC -> SOAP -> RESTful

126、CGI 和 WSGI

CGI 是通用网关接口，是连接 web 服务器和应用程序的接口，用户通过 CGI 来获取动态数据或文件等。
CGI 程序是一个独立的程序，它可以用几乎所有语言来写，包括 perl，c，lua，python 等等。
WSGI, Web Server Gateway?Interface，是 Python 应用程序或框架和 Web 服务器之间的一种接口，WSGI 的其中一个目的就是让用户可以用统一的语言(Python)编写前后端。

127、中间人攻击

中间人攻击（Man-in-the-middle attack，通常缩写为 MITM）是指攻击者与通讯的两端分别创建独立的联系，并交换其所收到的数据，使通讯的两端认为他们正在通过一个私密的连接与对方直接对话,但事实上整个会话都被攻击者完全控制。

128、c10k 问题

所谓 c10k 问题，指的是服务器同时支持成千上万个客户端的问题，也就是 concurrent 10 000 connection（这也是 c10k 这个名字的由来）。

129、socket

Socket=Ip address+ TCP/UDP + port

130、浏览器缓存

简单来说，浏览器缓存就是把一个已经请求过的 Web 资源（如 html 页面，图片，js，数据等）拷贝一份副本储存在浏览器中。缓存会根据进来的请求保存输出内容的副本。当下一个请求来到的时候，如果是相同的 URL，缓存会根据缓存机制决定是直接使用副本响应访问请求，还是向源服务器再次发送请求。比较常见的就是浏览器会缓存访问过网站的网页，当再次访问这个 URL 地址的时候，如果网页没有更新，就不会再次下载网页，而是直接使用本地缓存的网页。只有当网站明确标识资源已经更新，浏览器才会再次下载网页

131、Ajax

AJAX,Asynchronous JavaScript and XML（异步的 JavaScript 和 XML），是与在不重新加载整个页面的情况下，与服务器交换数据并更新部分网页的技术。

132、unix 进程间通信方式(IPC)

管道（Pipe）：管道可用于具有亲缘关系进程间的通信，允许一个进程和另一个与它有共同祖先的进程之间进行通信。
命名管道（named?pipe）：命名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。命名管道在文件系统中有对应的文件名。命名管道通过命令 mkfifo 或系统调用 mkfifo 来创建。
信号（Signal）：信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送信号给进程本身；linux 除了支持 Unix 早期信号语义函数 sigal 外，还支持语义符合 Posix.1 标准的信号函数 sigaction（实际上，该函数是基于 BSD 的，BSD 为了实现可靠信号机制，又能够统一对外接口，用 sigaction 函数重新实现了 signal 函数）。
消息(Message)队列：消息队列是消息的链接表，包括 Posix 消息队列 system?V 消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺
共享内存：使得多个进程可以访问同一块内存空间，是最快的可用 IPC 形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。
内存映射（mapped?memory）：内存映射允许任何多个进程间通信，每一个使用该机制的进程通过把一个共享的文件映射到自己的进程地址空间来实现它。
信号量（semaphore）：主要作为进程间以及同一进程不同线程之间的同步手段。

套接口（Socket）：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由 Unix 系统的 BSD 分支开发出来的，但现在一般可以移植到其它类 Unix 系统上：Linux 和 System?V 的变种都支持套接字。