Table of Contents

Python 语言特性

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。

通过 id 来看引用 a 的内存地址可以比较理解：

```
a = 1
def fun(a):
    print "func_in",id(a)   # func_in 41322472
    a = 2
    print "re-point",id(a), id(2)   # re-point 41322448 41322448
print "func_out",id(a), id(1)  # func_out 41322472 41322472
fun(a)
print a  # 1
```

注：具体的值在不同电脑上运行时可能不同。

可以看到，在执行完 a = 2 之后，a 引用中保存的值，即内存地址发生变化，由原来 1 对象的所在的地址变成了 2 这个实体对象的内存地址。

而第 2 个例子 a 引用保存的内存值就不会发生变化：

```
a = []
def fun(a):
    print "func_in",id(a)  # func_in 53629256
    a.append(1)
print "func_out",id(a)     # func_out 53629256
fun(a)
print a  # [1]
```

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, set 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.

如果还不明白的话,这里有更好的解释: http://stackoverflow.com/questions/986006/how-do-i-pass-a-variable-by-reference

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:http://stackoverflow.com/questions/100003/what-is-a-metaclass-in-python

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
a = A()	a.foo(x)	a.class_foo(x)	a.stati
A	不可用	A.class_foo(x)	A.stati

更多关于这个问题:http://stackoverflow.com/questions/136097/what-is-the-difference-between-staticmethod-and-classmethod-in-python

4 类变量和实例变量

类变量：

是可在类的所有实例之间共享的值（也就是说，它们不是单独分配给每个实例的）。例如下例中，num_of_instance 就是类变量，用于跟踪存在着多少个 Test 的实例。

实例变量：

实例化之后，每个实例单独拥有的变量。

```
class Test(object):
    num_of_instance = 0
    def __init__(self, name):
        self.name = name
        Test.num_of_instance += 1

if __name__ == '__main__':
    print Test.num_of_instance   # 0
    t1 = Test('jack')
    print Test.num_of_instance   # 1
    t2 = Test('lucy')
    print t1.name , t1.num_of_instance  # jack 2
    print t2.name , t2.num_of_instance  # lucy 2
```

补充的例子

```
class Person:
    name="aaa"


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

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]
```

参考:http://stackoverflow.com/questions/6470428/catch-multiple-exceptions-in-one-line-except-block

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().

```
a = [1,2,3]
b = {'a':1,'b':2,'c':3}
c = True
print type(a),type(b),type(c) # <type 'list'> <type 'dict'> <type 'bool'>
print isinstance(a,list)  # True
```

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，就是例如__init__(),__del__(),__call__()这些特殊方法

_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.不能用 from module import * 导入，其他方面和公有一样访问；

__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,它无法直接像公有成员一样随便访问,通过对象名._类名__xxx 这样的方式可以访问.

详情见:http://stackoverflow.com/questions/1301346/the-meaning-of-a-single-and-a-double-underscore-before-an-object-name-in-python

或者: http://www.zhihu.com/question/19754941

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))

http://stackoverflow.com/questions/5082452/python-string-formatting-vs-format

9 迭代器和生成器

这个是 stackoverflow 里 python 排名第一的问题,值得一看: http://stackoverflow.com/questions/231767/what-does-the-yield-keyword-do-in-python

这是中文版: http://taizilongxu.gitbooks.io/stackoverflow-about-python/content/1/README.html

这里有个关于生成器的创建问题面试官有考： 问： 将列表生成式中[]改成() 之后数据结构是否改变？ 答案：是，从列表变为生成器

```
>>> L = [x*x for x in range(10)]
>>> L
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> g = (x*x for x in range(10))
>>> g
<generator object <genexpr> at 0x0000028F8B774200>
```

通过列表生成式，可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。而且，创建一个包含百万元素的列表，不仅是占用很大的内存空间，如：我们只需要访问前面的几个元素，后面大部分元素所占的空间都是浪费的。因此，没有必要创建完整的列表（节省大量内存空间）。在 Python 中，我们可以采用生成器：边循环，边计算的机制—>generator

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*.

http://stackoverflow.com/questions/3394835/args-and-kwargs

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。**

这个问题比较大,推荐: http://stackoverflow.com/questions/739654/how-can-i-make-a-chain-of-function-decorators-in-python

中文: http://taizilongxu.gitbooks.io/stackoverflow-about-python/content/3/README.html

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

引自知乎:http://www.zhihu.com/question/20053359

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

这个面试官问了,我说了老半天,不知道他问的真正意图是什么.

stackoverflow

这篇文章很好的介绍了新式类的特性: http://www.cnblogs.com/btchenguang/archive/2012/09/17/2689146.html

新式类很早在 2.2 就出现了,所以旧式类完全是兼容的问题,Python3 里的类全部都是新式类.这里有一个 MRO 问题可以了解下(新式类是广度优先,旧式类是深度优先),<Python 核心编程>里讲的也很多.

一个旧式类的深度优先的例子

```
class A():
    def foo1(self):
        print "A"
class B(A):
    def foo2(self):
        pass
class C(A):
    def foo1(self):
        print "C"
class D(B, C):
    pass


d = D()
d.foo1()


# A
```

按照经典类的查找顺序从左到右深度优先的规则，在访问 d.foo1()的时候,D 这个类是没有的..那么往上查找,先找到 B,里面没有,深度优先,访问 A,找到了 foo1(),所以这时候调用的是 A 的 foo1()，从而导致 C 重写的 foo1()被绕过

15 __new__和__init__的区别

这个__new__确实很少见到,先做了解吧.

1. __new__是一个静态方法,而__init__是一个实例方法.
2. __new__方法会返回一个创建的实例,而__init__什么都不返回.
3. 只有在__new__返回一个 cls 的实例时后面的__init__才能被调用.
4. 当创建一个新实例时调用__new__,初始化一个实例时用__init__.

stackoverflow

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

单例模式是一种常用的软件设计模式。在它的核心结构中只包含一个被称为单例类的特殊类。通过单例模式可以保证系统中一个类只有一个实例而且该实例易于外界访问，从而方便对实例个数的控制并节约系统资源。如果希望在系统中某个类的对象只能存在一个，单例模式是最好的解决方案。

__new__()在__init__()之前被调用，用于生成实例对象。利用这个方法和类的属性的特点可以实现设计模式的单例模式。单例模式是指创建唯一对象，单例模式设计的类只能实例

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 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.**对于 io 密集型任务，python 的多线程起到作用，但对于 cpu 密集型任务，python 的多线程几乎占不到任何优势，还有可能因为争夺资源而变慢。**

见 Python 最难的问题

解决办法就是多进程和下面的协程(协程也只是单 CPU,但是能减小切换代价提升性能).

19 协程

知乎被问到了,呵呵哒,跪了

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.

Python 里最常见的 yield 就是协程的思想!可以查看第九个问题.

20 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:

1. 必须有一个内嵌函数
2. 内嵌函数必须引用外部函数中的变量
3. 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.

21 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.

推荐: 知乎

22 Python 函数式编程

这个需要适当的了解一下吧,毕竟函数式编程在 Python 中也做了引用.

推荐: 酷壳

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
```

23 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']]
```

24 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收效率。

1 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:

1. 简单
2. 实时性

缺点:

1. 维护引用计数消耗资源
2. 循环引用

2 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。

3 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。

25 Python 的 List

推荐: http://www.jianshu.com/p/J4U6rR

26 Python 的 is

is 是对比地址,==是对比值

27 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

28 Python2 和 3 的区别

推荐：Python 2.7.x 与 Python 3.x 的主要差异

29 super init

super() lets you avoid referring to the base class explicitly, which can be nice. But the main advantage comes with multiple inheritance, where all sorts of fun stuff can happen. See the standard docs on super if you haven't already.

Note that the syntax changed in Python 3.0: you can just say super().__init__() instead of super(ChildB, self).__init__() which IMO is quite a bit nicer. http://stackoverflow.com/questions/576169/understanding-python-super-with-init-methods

Python2.7 中的 super 方法浅见

30 range and xrange

都在循环时使用，xrange 内存性能更好。 for i in range(0, 20): for i in xrange(0, 20): What is the difference between range and xrange functions in Python 2.X? range creates a list, so if you do range(1, 10000000) it creates a list in memory with 9999999 elements. xrange is a sequence object that evaluates lazily.

http://stackoverflow.com/questions/94935/what-is-the-difference-between-range-and-xrange-functions-in-python-2-x

操作系统

1 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:

1. 连接数受限
2. 查找配对速度慢
3. 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.

关于 epoll 的: http://www.cnblogs.com/my_life/articles/3968782.html

2 调度算法

1. 先来先服务(FCFS, First Come First Serve)
2. 短作业优先(SJF, Shortest Job First)
3. 最高优先权调度(Priority Scheduling)
4. 时间片轮转(RR, Round Robin)
5. 多级反馈队列调度(multilevel feedback queue scheduling)

常见的调度算法总结:http://www.jianshu.com/p/6edf8174c1eb

实时调度算法:

1. 最早截至时间优先 EDF
2. 最低松弛度优先 LLF

3 死锁

原因:

1. 竞争资源
2. 程序推进顺序不当

必要条件:

1. 互斥条件
2. 请求和保持条件
3. 不剥夺条件
4. 环路等待条件

处理死锁基本方法:

1. 预防死锁(摒弃除 1 以外的条件)
2. 避免死锁(银行家算法)
3. 检测死锁(资源分配图)
4. 解除死锁
    i. 剥夺资源
    ii. 撤销进程

死锁概念处理策略详细介绍:https://wizardforcel.gitbooks.io/wangdaokaoyan-os/content/10.html

4 程序编译与链接

推荐: http://www.ruanyifeng.com/blog/2014/11/compiler.html

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：

1. 将所有的“#define”删除，并展开所用的宏定义
2. 处理所有条件预编译指令，比如“#if”、“#ifdef”、 “#elif”、“#endif”
3. 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
4. 删除所有注释
5. 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
6. 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。 链接的主要过程包块 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。

5 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

6 虚拟内存技术

虚拟存储器是指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.

7 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

1. 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.

2. 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.
3. 分页的作业地址空间是一维的.分段的地址空间是二维的.

8 页面置换算法

1. 最佳置换算法 OPT:不可能实现
2. 先进先出 FIFO
3. 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.
4. clock 算法

9 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

数据库

1 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。 彻底理解数据库事务: http://www.hollischuang.com/archives/898

2 数据库索引

推荐: http://tech.meituan.com/mysql-index.html

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

3 Redis 原理

Redis 是什么？

1. 是一个完全开源免费的 key-value 内存数据库
2. 通常被认为是一个数据结构服务器，主要是因为其有着丰富的数据结构 strings、map、 list、sets、 sorted sets

Redis 数据库

通常局限点来说，Redis 也以消息队列的形式存在，作为内嵌的 List 存在，满足实时的高并发需求。在使用缓存的时候，redis 比 memcached 具有更多的优势，并且支持更多的数据类型，把 redis 当作一个中间存储系统，用来处理高并发的数据库操作

- 速度快：使用标准 C 写，所有数据都在内存中完成，读写速度分别达到 10 万/20 万
- 持久化：对数据的更新采用 Copy-on-write 技术，可以异步地保存到磁盘上，主要有两种策略，一是根据时间，更新次数的快照（save 300 10 ）二是基于语句追加方式(Append-only file，aof)
- 自动操作：对不同数据类型的操作都是自动的，很安全
- 快速的主--从复制，官方提供了一个数据，Slave 在 21 秒即完成了对 Amazon 网站 10G key set 的复制。
- Sharding 技术： 很容易将数据分布到多个 Redis 实例中，数据库的扩展是个永恒的话题，在关系型数据库中，主要是以添加硬件、以分区为主要技术形式的纵向扩展解决了很多的应用场景，但随着 web2.0、移动互联网、云计算等应用的兴起，这种扩展模式已经不太适合了，所以近年来，像采用主从配置、数据库复制形式的，Sharding 这种技术把负载分布到多个特理节点上去的横向扩展方式用处越来越多。

Redis 缺点

- 是数据库容量受到物理内存的限制,不能用作海量数据的高性能读写,因此 Redis 适合的场景主要局限在较小数据量的高性能操作和运算上。
- Redis 较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复杂。为避免这一问题，运维人员在系统上线时必须确保有足够的空间，这对资源造成了很大的浪费。

4 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。

乐观锁与悲观锁的具体区别: http://www.cnblogs.com/Bob-FD/p/3352216.html

5 MVCC

全称是 Multi-Version Concurrent Control，即多版本并发控制，在 MVCC 协议下，每个读操作会看到一个一致性的 snapshot，并且可以实现非阻塞的读。MVCC 允许数据具有多个版本，这个版本可以是时间戳或者是全局递增的事务 ID，在同一个时间点，不同的事务看到的数据是不同的。

MySQL 的 innodb 引擎是如何实现 MVCC 的

innodb 会为每一行添加两个字段，分别表示该行**创建的版本**和**删除的版本**，填入的是事务的版本号，这个版本号随着事务的创建不断递增。在 repeated read 的隔离级别（事务的隔离级别请看这篇文章）下，具体各种数据库操作的实现：

- select：满足以下两个条件 innodb 会返回该行数据：
    - 该行的创建版本号小于等于当前版本号，用于保证在 select 操作之前所有的操作已经执行落地。

    - 该行的删除版本号大于当前版本或者为空。删除版本号大于当前版本意味着有一个并发事务将该行删除了。
- insert：将新插入的行的创建版本号设置为当前系统的版本号。
- delete：将要删除的行的删除版本号设置为当前系统的版本号。
- update：不执行原地 update，而是转换成 insert + delete。将旧行的删除版本号设置为当前版本号，并将新行 insert 同时设置创建版本号为当前版本号。

其中，写操作（insert、delete 和 update）执行时，需要将系统版本号递增。

由于旧数据并不真正的删除，所以必须对这些数据进行清理，innodb 会开启一个后台线程执行清理工作，具体的规则是将删除版本号小于当前系统版本的行删除，这个过程叫做 purge。

通过 MVCC 很好的实现了事务的隔离性，可以达到 repeated read 级别，要实现 serializable 还必须加锁。

参考：MVCC 浅析

6 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 对于 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。

mysql 数据库引擎: http://www.cnblogs.com/0201zcr/p/5296843.html MySQL 存储引擎－－MyISAM 与 InnoDB 区别: https://segmentfault.com/a/1190000008227211

网络

1 三次握手

1. 客户端通过向服务器端发送一个 SYN 来创建一个主动打开，作为三次握手的一部分。客户端把这段连接的序号设定为随机数 A。
2. 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。
3. 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。

2 四次挥手

注意：中断连接端可以是客户端，也可以是服务器端. 下面仅以客户端断开连接举例, 反之亦然.

1. 客户端发送一个数据分段, 其中的 FIN 标记设置为 1. 客户端进入 FIN-WAIT 状态. 该状态下客户端只接收数据, 不再发送数据.
2. 服务器接收到带有 FIN = 1 的数据分段, 发送带有 ACK = 1 的剩余数据分段, 确认收到客户端发来的 FIN 信息.
3. 服务器等到所有数据传输结束, 向客户端发送一个带有 FIN = 1 的数据分段, 并进入 CLOSE-WAIT 状态, 等待客户端发来带有 ACK = 1 的确认报文.
4. 客户端收到服务器发来带有 FIN = 1 的报文, 返回 ACK = 1 的报文确认, 为了防止服务器端未收到需要重发, 进入 TIME-WAIT 状态. 服务器接收到报文后关闭连接. 客户端等待 2MSL 后未收到回复, 则认为服务器成功关闭, 客户端关闭连接.

图解: http://blog.csdn.net/whuslei/article/details/6667471

3 ARP 协议

地址解析协议(Address Resolution Protocol)，其基本功能为透过目标设备的 IP 地址，查询目标的 MAC 地址，以保证通信的顺利进行。它是 IPv4 网络层必不可少的协议，不过在 IPv6 中已不再适用，并被邻居发现协议（NDP）所替代。

4 urllib 和 urllib2 的区别

这个面试官确实问过,当时答的 urllib2 可以 Post 而 urllib 不可以.

1. 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。
2. 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。

5 Post 和 Get

GET 和 POST 有什么区别？及为什么网上的多数答案都是错的 知乎回答

get: RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1 post: RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1

6 Cookie 和 Session

	Cookie
储存位置	客户端
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
安全性	不安全

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。

7 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
- 社区活跃

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

8 网站用户密码保存

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

9 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

10 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

11 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。

12 RESTful 架构(SOAP,RPC)

推荐: http://www.ruanyifeng.com/blog/2011/09/restful.html

13 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。

14 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

15 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。

官方说明：PEP-3333

16 中间人攻击

在 GFW 里屡见不鲜的,呵呵.

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。

17 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）。 推荐: https://my.oschina.net/xianggao/blog/664275

18 socket

推荐: http://www.360doc.com/content/11/0609/15/5482098_122692444.shtml

Socket=Ip address+ TCP/UDP + port

19 浏览器缓存

推荐: http://www.cnblogs.com/skynet/archive/2012/11/28/2792503.html

304 Not Modified

20 HTTP1.0 和 HTTP1.1

推荐: http://blog.csdn.net/elifefly/article/details/3964766

1. 请求头 Host 字段,一个服务器多个网站
2. 长链接
3. 文件断点续传
4. 身份认证,状态管理,Cache 缓存

HTTP 请求 8 种方法介绍 HTTP/1.1 协议中共定义了 8 种 HTTP 请求方法，HTTP 请求方法也被叫做“请求动作”，不同的方法规定了不同的操作指定的资源方式。服务端也会根据不同的请求方法做不同的响应。

GET

GET 请求会显示请求指定的资源。一般来说 GET 方法应该只用于数据的读取，而不应当用于会产生副作用的非幂等的操作中。

GET 会方法请求指定的页面信息，并返回响应主体，GET 被认为是不安全的方法，因为 GET 方法会被网络蜘蛛等任意的访问。

HEAD

HEAD 方法与 GET 方法一样，都是向服务器发出指定资源的请求。但是，服务器在响应 HEAD 请求时不会回传资源的内容部分，即：响应主体。这样，我们可以不传输全部内容的情况下，就可以获取服务器的响应头信息。HEAD 方法常被用于客户端查看服务器的性能。

POST

POST 请求会 向指定资源提交数据，请求服务器进行处理，如：表单数据提交、文件上传等，请求数据会被包含在请求体中。POST 方法是非幂等的方法，因为这个请求可能会创建新的资源或/和修改现有资源。

PUT

PUT 请求会身向指定资源位置上传其最新内容，PUT 方法是幂等的方法。通过该方法客户端可以将指定资源的最新数据传送给服务器取代指定的资源的内容。

DELETE

DELETE 请求用于请求服务器删除所请求 URI（统一资源标识符，Uniform Resource Identifier）所标识的资源。DELETE 请求后指定资源会被删除，DELETE 方法也是幂等的。

CONNECT

CONNECT 方法是 HTTP/1.1 协议预留的，能够将连接改为管道方式的代理服务器。通常用于 SSL 加密服务器的链接与非加密的 HTTP 代理服务器的通信。

OPTIONS

OPTIONS 请求与 HEAD 类似，一般也是用于客户端查看服务器的性能。 这个方法会请求服务器返回该资源所支持的所有 HTTP 请求方法，该方法会用’*’来代替资源名称，向服务器发送 OPTIONS 请求，可以测试服务器功能是否正常。JavaScript 的

XMLHttpRequest 对象进行 CORS 跨域资源共享时，就是使用 OPTIONS 方法发送嗅探请求，以判断是否有对指定资源的访问权限。 允许

TRACE

TRACE 请求服务器回显其收到的请求信息，该方法主要用于 HTTP 请求的测试或诊断。

HTTP/1.1 之后增加的方法

在 HTTP/1.1 标准制定之后，又陆续扩展了一些方法。其中使用中较多的是 PATCH 方法：

PATCH

PATCH 方法出现的较晚，它在 2010 年的 RFC 5789 标准中被定义。PATCH 请求与 PUT 请求类似，同样用于资源的更新。二者有以下两点不同：

但 PATCH 一般用于资源的部分更新，而 PUT 一般用于资源的整体更新。 当资源不存在时，PATCH 会创建一个新的资源，而 PUT 只会对已在资源进行更新。

21 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。

*NIX

unix 进程间通信方式(IPC)

1. 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。

2. 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。
3. 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）。
4. 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
5. 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。
6. 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。
7. 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。
8. 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。

数据结构

1 红黑树

红黑树与 AVL 的比较：

AVL 是严格平衡树，因此在增加或者删除节点的时候，根据不同情况，旋转的次数比红黑树要多；

红黑是用非严格的平衡来换取增删节点时候旋转次数的降低；

所以简单说，如果你的应用中，搜索的次数远远大于插入和删除，那么选择 AVL，如果搜索，插入删除次数几乎差不多，应该选择 RB。

红黑树详解: https://xieguanglei.github.io/blog/post/red-black-tree.html

教你透彻了解红黑树: https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/03.01.md

编程题

1 台阶问题/斐波那契

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
```

2 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。

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

3 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)
```

4 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。

使用 Step-wise 线性搜索。

```
def get_value(l, r, c):
    return l[r][c]
```

```
def find(l, x):
    m = len(l) - 1
    n = len(l[0]) - 1
    r = 0
    c = n
    while c >= 0 and r <= m:
        value = get_value(l, r, c)
        if value == x:
            return True
        elif value > x:
            c = c - 1
        elif value < x:
            r = r + 1
    return False
```

5 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
```

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]
```

sorted 排序并且用列表推导式.

l = ['b','c','d','b','c','a','a'] [single.append(i) for i in sorted(l) if i not in single] print single

6 链表成对调换

1->2->3->4 转换成 2->1->4->3.

```
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None


class Solution:
    # @param a ListNode
    # @return a ListNode
    def swapPairs(self, head):
        if head != None and head.next != None:
            next = head.next
            head.next = self.swapPairs(next.next)
            next.next = head
            return next
        return head
```

7 创建字典的方法

1 直接创建

```
dict = {'name':'earth', 'port':'80'}
```

2 工厂方法

```
items=[('name','earth'),('port','80')]
dict2=dict(items)
dict1=dict((['name','earth'],['port','80']))
```

3 fromkeys()方法

```
dict1={}.fromkeys(('x','y'),-1)
dict={'x':-1,'y':-1}
dict2={}.fromkeys(('x','y'))
dict2={'x':None, 'y':None}
```

8 合并两个有序列表

知乎远程面试要求编程

尾递归

```
def _recursion_merge_sort2(l1, l2, tmp):
    if len(l1) == 0 or len(l2) == 0:
        tmp.extend(l1)
        tmp.extend(l2)
        return tmp
    else:
        if l1[0] < l2[0]:
            tmp.append(l1[0])
            del l1[0]
        else:
            tmp.append(l2[0])
            del l2[0]
        return _recursion_merge_sort2(l1, l2, tmp)

def recursion_merge_sort2(l1, l2):
    return _recursion_merge_sort2(l1, l2, [])
```

循环算法

思路：

定义一个新的空列表

比较两个列表的首个元素

小的就插入到新列表里

把已经插入新列表的元素从旧列表删除

直到两个旧列表有一个为空

再把旧列表加到新列表后面

```
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]
    tmp.extend(l1)
    tmp.extend(l2)
    return tmp
```

pop 弹出

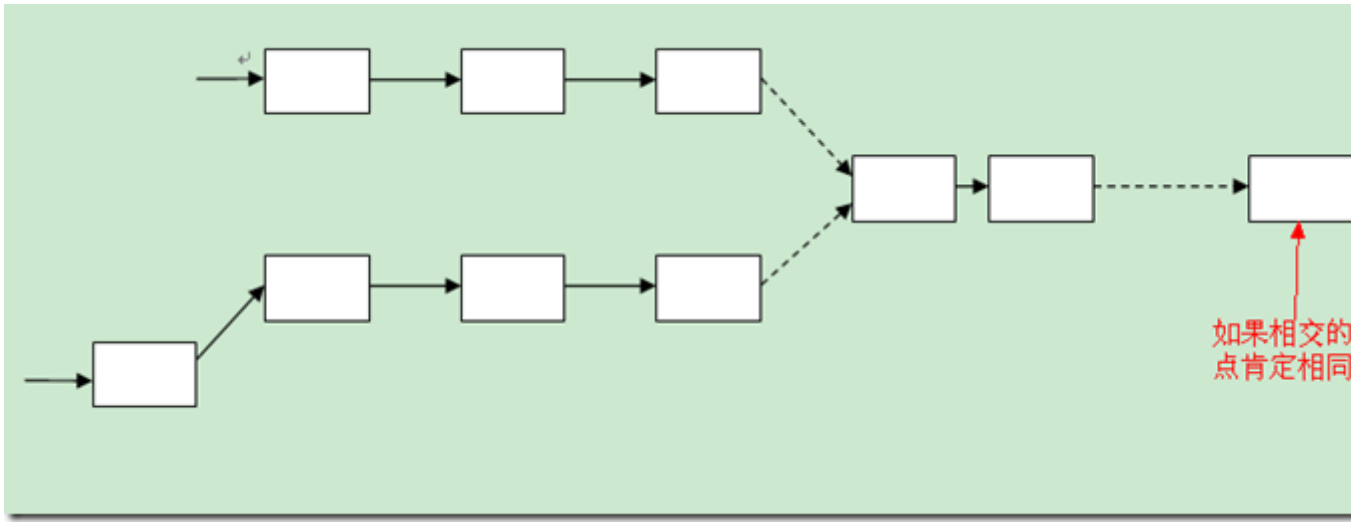
```
a = [1,2,3,7]
b = [3,4,5]

def merge_sortedlist(a,b):
    c = []
    while a and b:
        if a[0] >= b[0]:
            c.append(b.pop(0))
```

```
        else:
            c.append(a.pop(0))
    while a:
        c.append(a.pop(0))
    while b:
        c.append(b.pop(0))
    return c
print merge_sortedlist(a,b)
```

9 交叉链表求交点

其实思想可以按照从尾开始比较两个链表，如果相交，则从尾开始必然一致，只要从尾开始比较，直至不一致的地方即为交叉点，如图所示




```
# 使用 a,b 两个 list 来模拟链表，可以看出交叉点是 7 这个节点
a = [1,2,3,7,9,1,5]
b = [4,5,7,9,1,5]

for i in range(1,min(len(a),len(b))):
    if i==1 and (a[-1] != b[-1]):
        print "No"
        break
```

```
    else:
        if a[-i] != b[-i]:
            print "交叉节点：",a[-i+1]
            break
        else:
            pass
```

另外一种比较正规的方法，构造链表类

```
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None
def node(l1, l2):
    length1, lenth2 = 0, 0
    # 求两个链表长度
    while l1.next:
        l1 = l1.next
        length1 += 1
    while l2.next:
        l2 = l2.next
        length2 += 1
    # 长的链表先走
    if length1 > lenth2:
        for _ in range(length1 - length2):
            l1 = l1.next
    else:
        for _ in range(length2 - length1):
            l2 = l2.next
    while l1 and l2:
        if l1.next == l2.next:
            return l1.next
        else:
            l1 = l1.next
            l2 = l2.next
```

修改了一下:

```
#coding:utf-8
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None


def node(l1, l2):
    length1, length2 = 0, 0
    # 求两个链表长度
    while l1.next:
        l1 = l1.next#尾节点
        length1 += 1
    while l2.next:
        l2 = l2.next#尾节点
        length2 += 1

    #如果相交
    if l1.next == l2.next:
        # 长的链表先走
        if length1 > length2:
            for _ in range(length1 - length2):
                l1 = l1.next
            return l1#返回交点
        else:
            for _ in range(length2 - length1):
                l2 = l2.next
            return l2#返回交点
    # 如果不相交
    else:
        return
```

思路: http://humaoli.blog.163.com/blog/static/13346651820141125102125995/

10 二分查找

```
#coding:utf-8
def binary_search(list,item):
    low = 0
    high = len(list)-1
    while low<=high:
        mid = (low+high)/2
        guess = list[mid]
        if guess>item:
            high = mid-1
        elif guess<item:
            low = mid+1
        else:
            return mid
    return None
mylist = [1,3,5,7,9]
print binary_search(mylist,3)
```

参考: http://blog.csdn.net/u013205877/article/details/76411718

11 快排

```
#coding:utf-8
def quicksort(list):
    if len(list)<2:
        return list
    else:
        midpivot = list[0]
        lessbeforemidpivot = [i for i in list[1:] if i<=midpivot]
        biggerafterpivot = [i for i in list[1:] if i > midpivot]
```

```
        finallylist =
quicksort(lessbeforemidpivot)+[midpivot]+quicksort(biggerafterpivot)
        return finallylist


print quicksort([2,4,6,7,1,2,5])
```

更多排序问题可见：[数据结构与算法-排序篇-Python 描述]

12 找零问题

```
#coding:utf-8
#values 是硬币的面值 values = [ 25, 21, 10, 5, 1]
#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


def coinChange(values,valuesCounts,money,coinsUsed):
    #遍历出从 1 到 money 所有的钱数可能
    for cents in range(1,money+1):
        minCoins = cents
        #把所有的硬币面值遍历出来和钱数做对比
        for kind in range(0,valuesCounts):
            if (values[kind] <= cents):
                temp = coinsUsed[cents - values[kind]] +1
                if (temp < minCoins):
                    minCoins = temp
        coinsUsed[cents] = minCoins
        print ('面值:{0}的最少硬币使用数为:{1}'.format(cents, coinsUsed[cents]))
```

思路: http://blog.csdn.net/wdxin1322/article/details/9501163

方法: http://www.cnblogs.com/ChenxofHit/archive/2011/03/18/1988431.html

13 广度遍历和深度遍历二叉树

给定一个数组，构建二叉树，并且按层次打印这个二叉树

14 二叉树节点

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

tree = Node(1, Node(3, Node(7, Node(0)), Node(6)), Node(2, Node(5), Node(4)))
```

15 层次遍历

```
def lookup(root):
    stack = [root]
    while stack:
        current = stack.pop(0)
        print current.data
        if current.left:
            stack.append(current.left)
        if current.right:
            stack.append(current.right)
```

16 深度遍历

```
def deep(root):
    if not root:
        return
    print root.data
    deep(root.left)
```

```
    deep(root.right)


if __name__ == '__main__':
    lookup(tree)
    deep(tree)
```

17 前中后序遍历

深度遍历改变顺序就 OK 了

```
#coding:utf-8
#二叉树的遍历
#简单的二叉树节点类
class Node(object):
    def __init__(self,value,left,right):
        self.value = value
        self.left = left
        self.right = right

#中序遍历:遍历左子树,访问当前节点,遍历右子树

def mid_travelsal(root):
    if root.left is None:
        mid_travelsal(root.left)
    #访问当前节点
    print(root.value)
    if root.right is not None:
        mid_travelsal(root.right)

#前序遍历:访问当前节点,遍历左子树,遍历右子树

def pre_travelsal(root):
    print (root.value)
```

```
    if root.left is not None:
        pre_travelsal(root.left)
    if root.right is not None:
        pre_travelsal(root.right)

#后续遍历:遍历左子树,遍历右子树,访问当前节点

def post_trvelsal(root):
    if root.left is not None:
        post_trvelsal(root.left)
    if root.right is not None:
        post_trvelsal(root.right)
    print (root.value)
```

18 求最大树深

```
def maxDepth(root):
        if not root:
            return 0
        return max(maxDepth(root.left), maxDepth(root.right)) + 1
```

19 求两棵树是否相同

```
def isSameTree(p, q):
    if p == None and q == None:
        return True
    elif p and q :
        return p.val == q.val and isSameTree(p.left,q.left) and
isSameTree(p.right,q.right)
    else :
        return False
```

20 前序中序求后序

推荐: http://blog.csdn.net/hinyunsin/article/details/6315502

```
def rebuild(pre, center):
    if not pre:
        return
    cur = Node(pre[0])
    index = center.index(pre[0])
    cur.left = rebuild(pre[1:index + 1], center[:index])
    cur.right = rebuild(pre[index + 1:], center[index + 1:])
    return cur


def deep(root):
    if not root:
        return
    deep(root.left)
    deep(root.right)
    print root.data
```

21 单链表逆置

```
class Node(object):
    def __init__(self, data=None, next=None):
        self.data = data
        self.next = next


link = Node(1, Node(2, Node(3, Node(4, Node(5, Node(6, Node(7, Node(8,
Node(9)))))))))


def rev(link):
    pre = link
```

```
    cur = link.next
    pre.next = None
    while cur:
        tmp = cur.next
        cur.next = pre
        pre = cur
        cur = tmp
    return pre


root = rev(link)
while root:
    print root.data
    root = root.next
```

思路: http://blog.csdn.net/feliciafay/article/details/6841115

方法: http://www.xuebuyuan.com/2066385.html?mobile=1

22 两个字符串是否是变位词

```
class Anagram:
    """
    @:param s1: The first string
    @:param s2: The second string
    @:return true or false
    """
    def Solution1(s1,s2):
        alist = list(s2)

        pos1 = 0
        stillOK = True

        while pos1 < len(s1) and stillOK:
            pos2 = 0
```

```
            found = False
            while pos2 < len(alist) and not found:
                if s1[pos1] == alist[pos2]:
                    found = True
                else:
                    pos2 = pos2 + 1

            if found:
                alist[pos2] = None
            else:
                stillOK = False

            pos1 = pos1 + 1

        return stillOK

    print(Solution1('abcd','dcba'))

    def Solution2(s1,s2):
        alist1 = list(s1)
        alist2 = list(s2)

        alist1.sort()
        alist2.sort()



        pos = 0
        matches = True

        while pos < len(s1) and matches:
            if alist1[pos] == alist2[pos]:
                pos = pos + 1
            else:
```

```
            matches = False

    return matches

print(Solution2('abcde','edcbg'))

def Solution3(s1,s2):
    c1 = [0]*26
    c2 = [0]*26

    for i in range(len(s1)):
        pos = ord(s1[i])-ord('a')
        c1[pos] = c1[pos] + 1

    for i in range(len(s2)):
        pos = ord(s2[i])-ord('a')
        c2[pos] = c2[pos] + 1

    j = 0
    stillOK = True
    while j<26 and stillOK:
        if c1[j] == c2[j]:
            j = j + 1
        else:
            stillOK = False

    return stillOK

print(Solution3('apple','pleap'))
```

23 动态规划问题

可参考：动态规划(DP)的整理-Python 描述