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)

更多关于这个问题:

1. http://stackoverflow.com/questions/136097/what-is-the-difference-between-staticmethod-and-classmethod-in-python
2. https://realpython.com/blog/python/instance-class-and-static-methods-demystified/

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不像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(新式类继承是根据C3算法,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):
    instances = {}
    def getinstance(*args, **kw):
        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()
```

[单例模式伯乐在线详细解释]

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
    1. 剥夺资源
    2. 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