1 多任务的概念

什么叫“多任务”呢？简单地说，就是操作系统可以同时运行多个任务。打个比方，你一边在用浏览器上网，一边在听 MP3，一边在用 Word 赶作业，这就是多任务，至少同时有 3 个任务正在运行。还有很多任务悄悄地在后台同时运行着，只是桌面上没有显示而已。

现在，多核 CPU 已经非常普及了，但是，即使过去的单核 CPU，也可以执行多任务。由于 CPU 执行代码都是顺序执行的，那么，单核 CPU 是怎么执行多任务的呢？

答案就是操作系统轮流让各个任务交替执行，任务 1 执行 0.01 秒，切换到任务 2，任务 2 执行 0.01 秒，再切换到任务 3，执行 0.01 秒……这样反复执行下去。表面上看，每个任务都是交替执行的，但是，由于 CPU 的执行速度实在是太快了，我们感觉就像所有任务都在同时执行一样。

真正的并行执行多任务只能在多核 CPU 上实现，但是，由于任务数量远远多于 CPU 的核心数量，所以，操作系统也会自动把很多任务轮流调度到每个核心上执行。

注意：

- 并发：指的是任务数多余 cpu 核数，通过操作系统的各种任务调度算法，实现用多个任务“一起”执行（实际上总有一些任务不在执行，因为切换任务的速度相当快，看上去一起执行而已）

- 并行：**指的是任务数小于等于 cpu 核数，即任务真的是一起执行的**

2 多任务介绍

现实生活中

有很多的场景中的事情是同时进行的，比如开车的时候 手和脚共同来驾驶汽车，再比如唱歌跳舞也是同时进行的；

试想，如果把唱歌和跳舞这 2 件事情分开依次完成的话，估计就没有那么好的效果了（想一下场景：先唱歌，然后在跳舞，O(∩_∩)O 哈哈~）

程序中

如下程序，来模拟“唱歌跳舞”这件事情

```
#coding=utf-8

from time import sleep

def sing():
    for i in range(3):
        print("正在唱歌...%d"%i)
        sleep(1)

def dance():
    for i in range(3):
```

```
        print("正在跳舞...%d"%i)
        sleep(1)

if __name__ == '__main__':
    sing() #唱歌
    dance() #跳舞
```

运行结果如下：

```
正在唱歌...0
正在唱歌...1
正在唱歌...2
正在跳舞...0
正在跳舞...1
正在跳舞...2
```

!!!注意

- 很显然刚刚的程序并没有完成唱歌和跳舞同时进行的要求
- 如果想要实现“唱歌跳舞”同时进行，那么就需要一个新的方法，叫做：**多任务**

3 线程

python 的 thread 模块是比较底层的模块，python 的 threading 模块是对 thread 做了一些包装的，可以更加方便的被使用

1. 使用 threading 模块

单线程执行

```
#coding=utf-8
import time

def saySorry():
    print("亲爱的，我错了，我能吃饭了吗？")
    time.sleep(1)

if __name__ == "__main__":
    for i in range(5):
        saySorry()
```

运行结果：

多线程执行

```
#coding=utf-8
import threading
import time

def saySorry():
    print("亲爱的，我错了，我能吃饭了吗？")
    time.sleep(1)

if __name__ == "__main__":
    for i in range(5):
        t = threading.Thread(target=saySorry)
        t.start() #启动线程，即让线程开始执行
```

运行结果：

说明

1. 可以明显看出使用了多线程并发的操作，花费时间要短很多
2. 当调用 start()时，才会真正的创建线程，并且开始执行

2. 主线程会等待所有的子线程结束后才结束

```
#coding=utf-8
import threading
from time import sleep,ctime

def sing():
    for i in range(3):
        print("正在唱歌...%d"%i)
        sleep(1)

def dance():
    for i in range(3):
        print("正在跳舞...%d"%i)
        sleep(1)

if __name__ == '__main__':
    print('---开始---:%s'%ctime())

    t1 = threading.Thread(target=sing)
    t2 = threading.Thread(target=dance)

    t1.start()
    t2.start()

    #sleep(5) # 屏蔽此行代码，试试看，程序是否会立马结束？
    print('---结束---:%s'%ctime())
```

3. 查看线程数量

```
#coding=utf-8
import threading
from time import sleep,ctime

def sing():
    for i in range(3):
        print("正在唱歌...%d"%i)
        sleep(1)

def dance():
    for i in range(3):
        print("正在跳舞...%d"%i)
        sleep(1)

if __name__ == '__main__':
    print('---开始---:%s'%ctime())
```

```
    t1 = threading.Thread(target=sing)
    t2 = threading.Thread(target=dance)

    t1.start()
    t2.start()

    while True:
        length = len(threading.enumerate())
        print('当前运行的线程数为：%d'%length)
        if length<=1:
            break

        sleep(0.5)
```

命令方法

ps -elLf|grep wd

top

按 H 来查看线程

在大多数系统上，Python同时支持消息传递和基于线程的并发编程。尽管大多数程序员熟悉的往往是线程接口，但实际上Python线程受到的限制有很多。尽管最低限度是线程安全的，但Python解释器还是使用了内部的GIL（Global Interpreter Lock，全局解释器锁定），在任意指定的时刻只允许单个Python线程执行。无论系统上存在多少个可用的CPU核心，这限制了Python程序只能在一个处理器上运行。尽管GIL经常是Python社区中争论的热点，但在可以预见的将来它都不可能消失。

4 线程-注意点

1. 线程执行代码的封装

通过上一小节，能够看出，通过使用 threading 模块能完成多任务的程序开发，为了让每个线程的封装性更完美，所以使用 threading 模块时，往往会定义一个新的子类 class，只要继承 threading.Thread 就可以了，然后重写 run 方法

示例如下：

```
#coding=utf-8
import threading
import time

class MyThread(threading.Thread):
    def run(self):
        for i in range(3):
            time.sleep(1)
            msg = "I'm "+self.name+' @ '+str(i) #name 属性中保存的是当前线程的名字
            print(msg)


if __name__ == '__main__':
    t = MyThread()
    t.start()
```

说明

- python 的 threading.Thread 类有一个 run 方法，用于定义线程的功能函数，可以在自己的线程类中覆盖该方法。而创建自己的线程实例后，通过 Thread 类的 start 方法，可以启动该线程，交给 python 虚拟机进行调度，当该线程获得执行的机会时，就会调用 run 方法执行线程。

2. 线程的执行顺序

```
#coding=utf-8
import threading
import time

class MyThread(threading.Thread):
    def run(self):
        for i in range(3):
            time.sleep(1)
```

```
            msg = "I'm "+self.name+' @ '+str(i)
            print(msg)
def test():
    for i in range(5):
        t = MyThread()
        t.start()
if __name__ == '__main__':
    test()
```

执行结果：(运行的结果可能不一样，但是大体是一致的)

```
    I'm Thread-1 @ 0
    I'm Thread-2 @ 0
    I'm Thread-5 @ 0
    I'm Thread-3 @ 0
    I'm Thread-4 @ 0
    I'm Thread-3 @ 1
    I'm Thread-4 @ 1
    I'm Thread-5 @ 1
    I'm Thread-1 @ 1
    I'm Thread-2 @ 1
    I'm Thread-4 @ 2
    I'm Thread-5 @ 2
    I'm Thread-2 @ 2
    I'm Thread-1 @ 2
    I'm Thread-3 @ 2
```

说明

从代码和执行结果我们可以看出，多线程程序的执行顺序是不确定的。当执行到 sleep 语句时，线程将被阻塞（Blocked），到 sleep 结束后，线程进入就绪（Runnable）状态，等待调度。而线程调度将自行选择一个线程执行。上面的代码中只能保证每个线程都运行完整个 run 函数，但是线程的启动顺序、run 函数中每次循环的执行顺序都不能确定。

3. 总结

1. 每个线程默认有一个名字，尽管上面的例子中没有指定线程对象的 name，但是 python 会自动为线程指定一个名字。
2. 当线程的 run()方法结束时该线程完成。
3. 无法控制线程调度程序，但可以通过别的方式来影响线程调度的方式。

5 多线程-共享全局变量

```
from threading import Thread
import time

g_num = 100

def work1():
    global g_num
    for i in range(3):
        g_num += 1

    print("----in work1, g_num is %d---"%g_num)


def work2():
```

```
    global g_num
    print("----in work2, g_num is %d---"%g_num)


print("---线程创建之前 g_num is %d---"%g_num)

t1 = Thread(target=work1)
t1.start()

#延时一会，保证 t1 线程中的事情做完
time.sleep(1)

t2 = Thread(target=work2)
t2.start()
```

运行结果:

```
---线程创建之前 g_num is 100---
----in work1, g_num is 103---
----in work2, g_num is 103---
```

列表当做实参传递到线程中

```
from threading import Thread
import time

def work1(nums):
    nums.append(44)
    print("----in work1---",nums)


def work2(nums):
    #延时一会，保证 t1 线程中的事情做完
    time.sleep(1)
    print("----in work2---",nums)

g_nums = [11,22,33]

t1 = Thread(target=work1, args=(g_nums,))
t1.start()

t2 = Thread(target=work2, args=(g_nums,))
t2.start()
```

运行结果:

```
----in work1--- [11, 22, 33, 44]
----in work2--- [11, 22, 33, 44]
```

总结：

- 在一个进程内的所有线程共享全局变量，很方便在多个线程间共享数据
- 缺点就是，线程是对全局变量随意遂改可能造成多线程之间对全局变量的混乱（即线程非安全）

6 多线程-共享全局变量问题

多线程开发可能遇到的问题

假设两个线程 t1 和 t2 都要对全局变量 g_num(默认是 0)进行加 1 运算，t1 和 t2 都各对 g_num 加 10 次，g_num 的最终的结果应该为 20。

但是由于是多线程同时操作，有可能出现下面情况：

1. 在 g_num=0 时，t1 取得 g_num=0。此时系统把 t1 调度为”sleeping”状态，把 t2 转换为”running”状态，t2 也获得 g_num=0
2. 然后 t2 对得到的值进行加 1 并赋给 g_num，使得 g_num=1
3. 然后系统又把 t2 调度为”sleeping”，把 t1 转为”running”。线程 t1 又把它之前得到的 0 加 1 后赋值给 g_num。
4. 这样导致虽然 t1 和 t2 都对 g_num 加 1，但结果仍然是 g_num=1

测试 1

```
import threading
import time

g_num = 0
```

```
def work1(num):
    global g_num
    for i in range(num):
        g_num += 1
    print("----in work1, g_num is %d---"%g_num)


def work2(num):
    global g_num
    for i in range(num):
        g_num += 1
    print("----in work2, g_num is %d---"%g_num)


print("---线程创建之前 g_num is %d---"%g_num)

t1 = threading.Thread(target=work1, args=(100,))
t1.start()

t2 = threading.Thread(target=work2, args=(100,))
t2.start()

while len(threading.enumerate()) != 1:
    time.sleep(1)

print("2 个线程对同一个全局变量操作之后的最终结果是:%s" % g_num)
```

运行结果：

```
---线程创建之前 g_num is 0---
----in work1, g_num is 100---
----in work2, g_num is 200---
2 个线程对同一个全局变量操作之后的最终结果是:200
```

测试 2

```
import threading
import time

g_num = 0

def work1(num):
    global g_num
    for i in range(num):
        g_num += 1
    print("----in work1, g_num is %d---"%g_num)


def work2(num):
```

```
    global g_num
    for i in range(num):
        g_num += 1
    print("----in work2, g_num is %d---"%g_num)


print("---线程创建之前 g_num is %d---"%g_num)

t1 = threading.Thread(target=work1, args=(1000000,))
t1.start()

t2 = threading.Thread(target=work2, args=(1000000,))
t2.start()

while len(threading.enumerate()) != 1:
    time.sleep(1)

print("2 个线程对同一个全局变量操作之后的最终结果是:%s" % g_num)
```

运行结果：

```
---线程创建之前 g_num is 0---
----in work1, g_num is 1088005---
----in work2, g_num is 1286202---
2 个线程对同一个全局变量操作之后的最终结果是:1286202
```

结论

- 如果多个线程同时对同一个全局变量操作，会出现资源竞争问题，从而数据结果会不正确

7 同步的概念

同步就是协同步调，按预定的先后次序进行运行。如:你说完，我再说。

"同"字从字面上容易理解为一起动作

其实不是，"同"字应是指协同、协助、互相配合。

如进程、线程同步，可理解为进程或线程 A 和 B 一块配合，A 执行到一定程度时要依靠 B 的某个结果，于是停下来，示意 B 运行;B 执行，再将结果给 A;A 再继续操作。

解决线程同时修改全局变量的方式

对于上一小节提出的那个计算错误的问题，可以通过线程同步来进行解决

思路，如下:

1. 系统调用 t1，然后获取到 g_num 的值为 0，此时上一把锁，即不允许其他线程操作 g_num
2. t1 对 g_num 的值进行+1
3. t1 解锁，此时 g_num 的值为 1，其他的线程就可以使用 g_num 了，而且是 g_num 的值不是 0 而是 1
4. 同理其他线程在对 g_num 进行修改时，都要先上锁，处理完后再解锁，在上锁的整个过程中不允许其他线程访问，就保证了数据的正确性

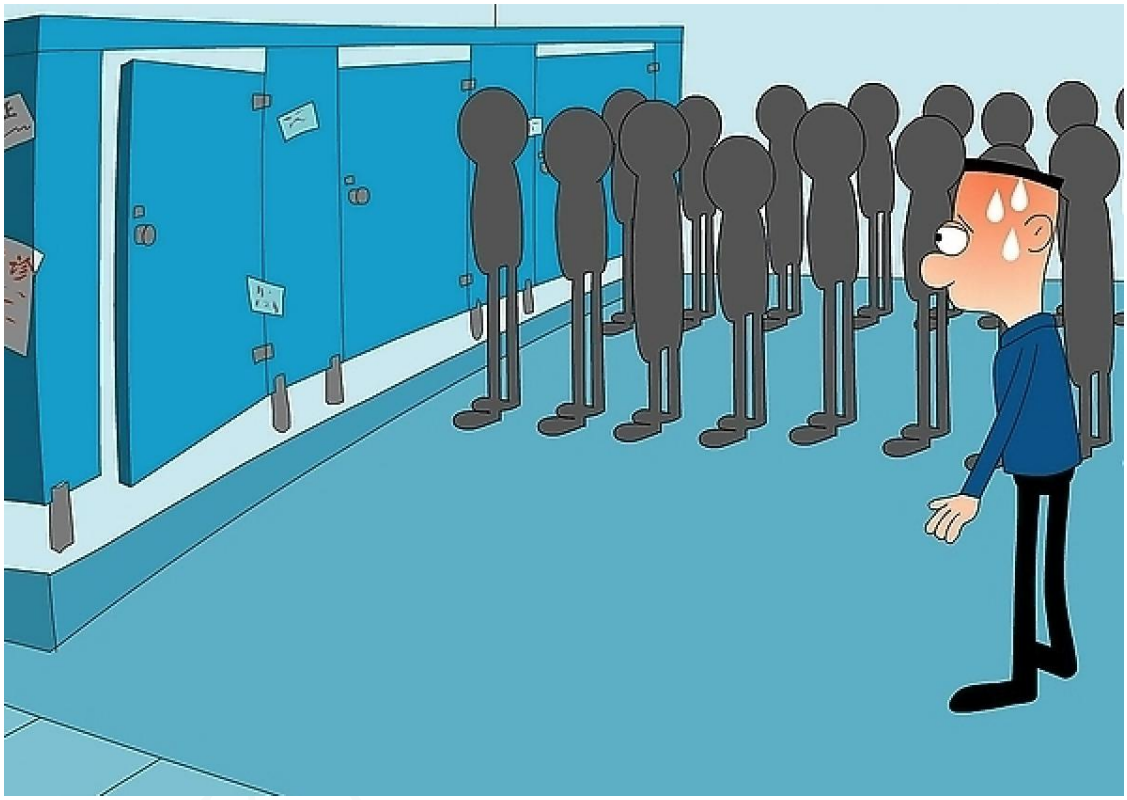
8 互斥锁

当多个线程几乎同时修改某一个共享数据的时候，需要进行同步控制

线程同步能够保证多个线程安全访问竞争资源，最简单的同步机制是引入互斥锁。

互斥锁为资源引入一个状态：锁定/非锁定

某个线程要更改共享数据时，先将其锁定，此时资源的状态为“锁定”，其他线程不能更改；直到该线程释放资源，将资源的状态变成“非锁定”，其他的线程才能再次锁定该资源。互斥锁保证了每次只有一个线程进行写入操作，从而保证了多线程情况下数据的正确性。



threading 模块中定义了 Lock 类，可以方便的处理锁定：

```
# 创建锁
mutex = threading.Lock()

# 锁定
mutex.acquire()

# 释放
mutex.release()
```

注意：

- 如果这个锁之前是没有上锁的，那么 acquire 不会堵塞

- 如果在调用 acquire 对这个锁上锁之前 它已经被 其他线程上了锁，那么此时 acquire 会堵塞，直到这个锁被解锁为止

使用互斥锁完成 2 个线程对同一个全局变量各加 100 万次的操作

```
import threading
import time

g_num = 0

def test1(num):
    global g_num
    for i in range(num):
        mutex.acquire()  # 上锁
        g_num += 1
        mutex.release()  # 解锁

    print("---test1---g_num=%d"%g_num)

def test2(num):
    global g_num
    for i in range(num):
        mutex.acquire()  # 上锁
        g_num += 1
        mutex.release()  # 解锁

    print("---test2---g_num=%d"%g_num)

# 创建一个互斥锁
# 默认是未上锁的状态
mutex = threading.Lock()

# 创建 2 个线程，让他们各自对 g_num 加 1000000 次
p1 = threading.Thread(target=test1, args=(1000000,))
p1.start()

p2 = threading.Thread(target=test2, args=(1000000,))
p2.start()

# 等待计算完成
while len(threading.enumerate()) != 1:
    time.sleep(1)

print("2 个线程对同一个全局变量操作之后的最终结果是:%s" % g_num)
```

运行结果：

```
---test1---g_num=1909909
---test2---g_num=2000000
2 个线程对同一个全局变量操作之后的最终结果是:2000000
```

可以看到最后的结果，加入互斥锁后，其结果与预期相符。

上锁解锁过程

当一个线程调用锁的 acquire()方法获得锁时，锁就进入“locked”状态。

每次只有一个线程可以获得锁。如果此时另一个线程试图获得这个锁，该线程就会变为“blocked”状态，称为“阻塞”，直到拥有锁的线程调用锁的 release()方法释放锁之后，锁进入“unlocked”状态。

线程调度程序从处于同步阻塞状态的线程中选择一个来获得锁，并使得该线程进入运行（running）状态。

总结

锁的好处：

- 确保了某段关键代码只能由一个线程从头到尾完整地执行

锁的坏处：

- 阻止了多线程并发执行，包含锁的某段代码实际上只能以单线程模式执行，效率就大大地下降了
- 由于可以存在多个锁，不同的线程持有不同的锁，并试图获取对方持有的锁时，可能会造成死锁

9 死锁

现实社会中，男女双方都在等待对方先道歉

如果双方都这样固执的等待对方先开口，弄不好，就分搜了

1. 死锁

在线程间共享多个资源的时候，如果两个线程分别占有一部分资源并且同时等待对方的资源，就会造成死锁。

尽管死锁很少发生，但一旦发生就会造成应用的停止响应。下面看一个死锁的例子

```
#coding=utf-8
import threading
import time

class MyThread1(threading.Thread):
    def run(self):
        # 对 mutexA 上锁
        mutexA.acquire()

        # mutexA 上锁后，延时 1 秒，等待另外那个线程 把 mutexB 上锁
        print(self.name+'----do1---up----')
        time.sleep(1)

        # 此时会堵塞，因为这个 mutexB 已经被另外的线程抢先上锁了
        mutexB.acquire()
        print(self.name+'----do1---down----')
        mutexB.release()

        # 对 mutexA 解锁
```

```
        mutexA.release()

class MyThread2(threading.Thread):
    def run(self):
        # 对 mutexB 上锁
        mutexB.acquire()

        # mutexB 上锁后，延时 1 秒，等待另外那个线程 把 mutexA 上锁
        print(self.name+'----do2---up----')
        time.sleep(1)

        # 此时会堵塞，因为这个 mutexA 已经被另外的线程抢先上锁了
        mutexA.acquire()
        print(self.name+'----do2---down----')
        mutexA.release()

        # 对 mutexB 解锁
        mutexB.release()

mutexA = threading.Lock()
mutexB = threading.Lock()

if __name__ == '__main__':
    t1 = MyThread1()
    t2 = MyThread2()
    t1.start()
    t2.start()
```

运行结果：

此时已经进入到了死锁状态，可以使用 ctrl-c 退出

2. 避免死锁

- 程序设计时要尽量避免（银行家算法）
- 添加超时时间等

附录-银行家算法

[背景知识]

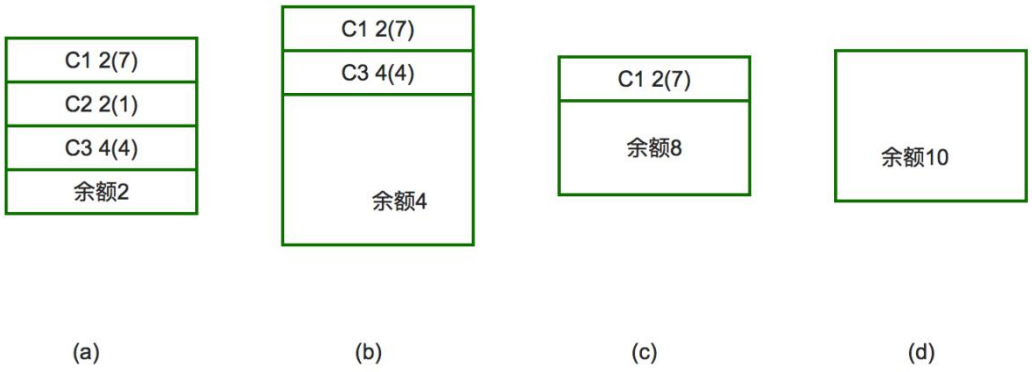
一个银行家如何将一定数目的资金安全地借给若干个客户，使这些客户既能借到钱完成要干的事，同时银行家又能收回全部资金而不至于破产，这就是银行家问题。

这个问题同操作系统中资源分配问题十分相似：银行家就像一个操作系统，客户就像运行的进程，银行家的资金就是系统的资源。

[问题的描述]

一个银行家拥有一定数量的资金，有若干个客户要贷款。每个客户须在一开始就声明他所需贷款的总额。若该客户贷款总额不超过银行家的资金总数，银行家可以接收客户的要求。客户贷款是以每次一个资金单位（如 1 万 RMB 等）的方式进行的，客户在借满所需的全部单位款额之前可能会等待，但银行家须保证这种等待是有限的，可完成的。

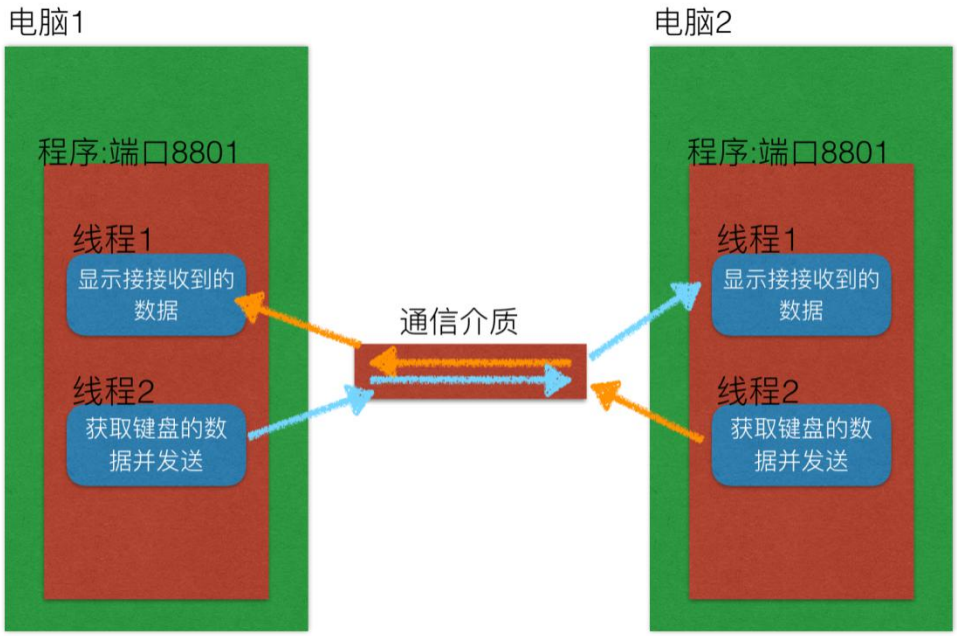
例如：有三个客户 C1，C2，C3，向银行家借款，该银行家的资金总额为 10 个资金单位，其中 C1 客户要借 9 各资金单位，C2 客户要借 3 个资金单位，C3 客户要借 8 个资金单位，总计 20 个资金单位。某一时刻的状态如图所示。



对于 a 图的状态，按照安全序列的要求，我们选的第一个客户应满足该客户所需的贷款小于等于银行家当前所剩余的钱款，可以看出只有 C2 客户能被满足：C2 客户需 1 个资金单位，小银行家手中的 2 个资金单位，于是银行家把 1 个资金单位借给 C2 客户，使之完成工作并归还所借的 3 个资金单位的钱，进入 b 图。同理，银行家把 4 个资金单位借给 C3 客户，使其完成工作，在 c 图中，只剩一个客户 C1，它需 7 个资金单位，这时银行家有 8 个资金单位，所以 C1 也能顺利借到钱并完成工作。最后（见图 d）银行家收回全部 10 个资金单位，保证不赔本。那麽客户序列{C1，C2，C3}就是个安全序列，按照这个序列贷款，银行家才是安全的。否则的话，若在图 b 状态时，银行家把手中的 4 个资金单位借给了 C1，则出现不安全状态：这时 C1，C3 均不能完成工作，而银行家手中又没有钱了，系统陷入僵持局面，银行家也不能收回投资。

综上所述，银行家算法是从当前状态出发，逐个按安全序列检查各客户谁能完成其工作，然后假定其完成工作且归还全部贷款，再进而检查下一个能完成工作的客户，……。如果所有客户都能完成工作，则找到一个安全序列，银行家才是安全的。

10 案例：多任务版 udp 聊天器



说明

- 编写一个有 2 个线程的程序
- 线程 1 用来接收数据然后显示
- 线程 2 用来检测键盘数据然后通过 udp 发送数据

要求

1. 实现上述要求
2. 总结多任务程序的特点

参考代码:

```
import socket
import threading


def send_msg(udp_socket):
    """获取键盘数据，并将其发送给对方"""
    while True:
        # 1. 从键盘输入数据
        msg = input("\n 请输入要发送的数据:")
        # 2. 输入对方的 ip 地址
        dest_ip = input("\n 请输入对方的 ip 地址:")
```

```
        # 3. 输入对方的 port
        dest_port = int(input("\n 请输入对方的 port:"))
        # 4. 发送数据
        udp_socket.sendto(msg.encode("utf-8"), (dest_ip, dest_port))


def recv_msg(udp_socket):
    """接收数据并显示"""
    while True:
        # 1. 接收数据
        recv_msg = udp_socket.recvfrom(1024)
        # 2. 解码
        recv_ip = recv_msg[1]
        recv_msg = recv_msg[0].decode("utf-8")
        # 3. 显示接收到的数据
        print(">>>%s:%s" % (str(recv_ip), recv_msg))


def main():
    # 1. 创建套接字
    udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    # 2. 绑定本地信息
    udp_socket.bind(("", 7890))

    # 3. 创建一个子线程用来接收数据
    t = threading.Thread(target=recv_msg, args=(udp_socket,))
    t.start()
    # 4. 让主线程用来检测键盘数据并且发送
    send_msg(udp_socket)

if __name__ == "__main__":
    main()
```

https://blog.csdn.net/u013210620/article/details/78736153