为什么要有线程
什么是线程
线程和进程之间的关系
线程的特点
Python中的线程

- 进程是计算机中最小的资源分配单位
- 进程对于操作系统来说还是有一定的负担
- 创建一个进程 操作系统要分配的资源大致有：
  - 代码
  - 数据
  - 文件

1. 为什么要有线程？
   a. 轻量级的概念
   b. 他没有属于自己的资源
      i. 一条线程只负责执行代码
      ii. 没有自己独立的代码、变量、文件资源
2. 什么是线程？
   a. 线程是计算机中被cpu调度的最小单位
   b. 我们的计算机当中的cpu都是执行的线程中的代码
3. 进程和线程之间的关系
   a. 每一个进程中都有至少一条线程在工作（就像每一辆车要有一个司机开）
4. 线程的特点
   a. 同一个进程中的所有线程的资源是共享的
   b. 轻量级 没有自己的资源
   c. 可以并发执行
5. 线程和进程的区别
   a. 占用的资源
   b. 调度的效率

c. 资源是否共享

问题：一个进程中的多个线程能够并行吗？（Python中不行）（java c++ c# 是可以的）

见ppt

6. Python线程模块（threading）

```python
from threading import Thread
from multiprocessing import Process
import time
import os

# def func(i):
#     print("我是线程{}".format(i))


# if __name__ == '__main__':
#     for i in range(10):
#         t = Thread(target=func, args=(i, ))
#         t.start()


# 为什么会轻量级？
# if __name__ == '__main__':
#     start1 = time.time()
#     t_lis = []
#     for i in range(100):
#         t = Thread(target=func, args=(i, ))
#         t.start()
#         t_lis.append(t)
#     for t in t_lis:
#         t.join()
#     tt = time.time() - start1
#
#     start2 = time.time()
#     p_lis = []
#     for i in range(100):
#         p = Process(target=func, args=(i,))
#         p.start()
#         p_lis.append(p)
```

```
#     for p in p_lis:
#         p.join()
#     pp = time.time() - start2
#
#     print('启动100个线程需要: %s' % tt)
#     print('启动100个进程需要: %s' % pp)

# 数据共享
num = 100

def func():
    global num
    num -= 1

t_lis = []
for i in range(100):
    t = Thread(target=func)
    t.start()
    t_lis.append(t)
for t in t_lis:
    t.join()

print(num)
```

7. 守护线程

```
import time
from threading import Thread


def func1():
    while True:
        time.sleep(0.5)
        print(123)


def func2():
    print('func2, start')
    time.sleep(3)
```

```
    print('func2 end')


t1 = Thread(target=func1)
t2 = Thread(target=func2)
# 设置守护线程
t1.setDaemon(True)
t1.start()
t2.start()
print('主线程的代码结束了')


# 结论：
# 守护线程是在主线程代码结束之后，还等待了子线程结束才结束的
# 主线程结束，就是意味着主进程的结束
# 主线程等待所有的线程结束
# 主线程结束之后，守护线程随着主进程的结束自然结束的
```

8. 线程的安全问题

a. 写一个线程不安全的情况

```
from threading import Thread

num = 0


def func1():
    global num
    for i in range(1000000):
        num -= 1

def func2():
    global num
    for i in range(1000000):
        num += 1


if __name__ == '__main__':
    t_lis = []
```

```
    for i in range(10):
        t1 = Thread(target=func1)
        t2 = Thread(target=func2)
        t1.start()
        t2.start()
        t_lis.append(t1)
        t_lis.append(t2)
    for t in t_lis:
        t.join()

    print("num =", num)
```

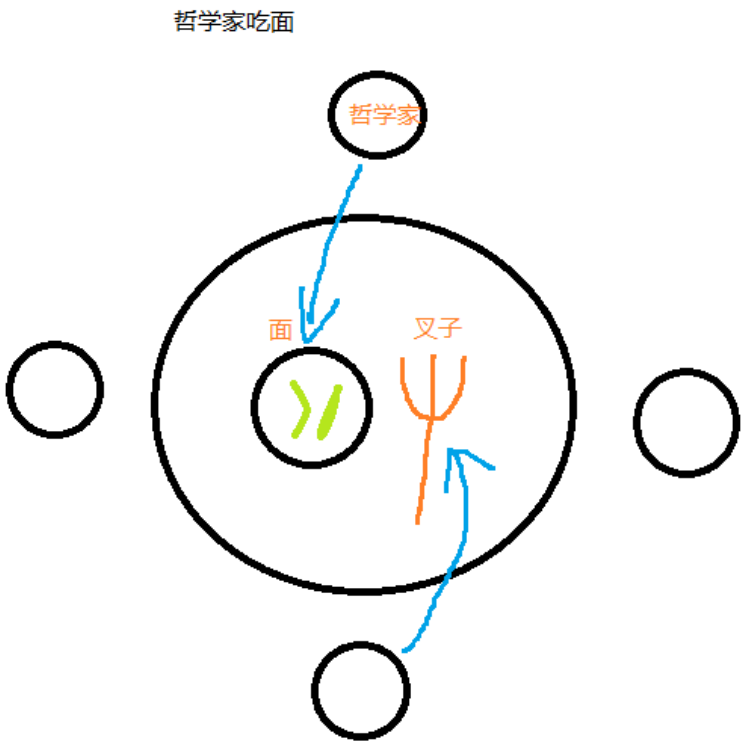
怎么解决上面的问题？（加上互斥锁）

```
from threading import Thread, Lock

num = 0


def func1(lock):
    global num
    for i in range(1000000):
        lock.acquire()
        num -= 1
        lock.release()

def func2(lock):
    global num
    for i in range(1000000):
        lock.acquire()
        num += 1
        lock.release()


if __name__ == '__main__':
    lock = Lock()
    t_lis = []
    for i in range(10):
        t1 = Thread(target=func1, args=(lock,))
        t2 = Thread(target=func2, args=(lock,))
```

```
        t1.start()
        t2.start()
        t_lis.append(t1)
        t_lis.append(t2)
    for t in t_lis:
        t.join()

    print("num =", num)
```

递归锁

举个例子



写一个死锁程序

```
import time
from threading import Thread, Lock

noodle_lock = Lock()  # 面锁
fork_lock = Lock()  # 叉子锁

def eat1(name):
    noodle_lock.acquire()
    print('%s拿到面条了' % name)
    fork_lock.acquire()
```

```
    print('%s拿到叉子了' % name)

    print('%s吃面' % name)
    time.sleep(3)
    fork_lock.release()
    print('%s放下叉子' % name)
    noodle_lock.release()
    print('%s放下面' % name)

def eat2(name):
    fork_lock.acquire()
    print('%s拿到叉子了' % name)
    noodle_lock.acquire()
    print('%s拿到面条了' % name)
    print('%s吃面' % name)
    time.sleep(3)
    noodle_lock.release()
    print('%s放下面' % name)
    fork_lock.release()
    print('%s放下叉子' % name)


if __name__ == '__main__':
    name_list1 = ["尹精赛", "郑俊杰"]
    name_list2 = ["阿莫西林", "小明哥"]
    for name in name_list1:
        Thread(target=eat1, args=(name, )).start()
    for name in name_list2:
        Thread(target=eat2, args=(name, )).start()
```

解决死锁现象（使用递归锁）

```
import time
from threading import Thread, Lock, RLock

# noodle_lock = Lock()  # 面锁
# fork_lock = Lock()  # 叉子锁


fork_lock = noodle_lock = RLock()
```

```python


def eat1(name):
    noodle_lock.acquire()
    print('%s拿到面条了' % name)
    fork_lock.acquire()
    print('%s拿到叉子了' % name)
    print('%s吃面' % name)
    time.sleep(3)
    fork_lock.release()
    print('%s放下叉子' % name)
    noodle_lock.release()
    print('%s放下面' % name)

def eat2(name):
    fork_lock.acquire()
    print('%s拿到叉子了' % name)
    noodle_lock.acquire()
    print('%s拿到面条了' % name)
    print('%s吃面' % name)
    time.sleep(3)
    noodle_lock.release()
    print('%s放下面' % name)
    fork_lock.release()
    print('%s放下叉子' % name)


if __name__ == '__main__':
    name_list1 = ["尹精赛", "郑俊杰"]
    name_list2 = ["阿莫西林", "小明哥"]
    for name in name_list1:
        Thread(target=eat1, args=(name, )).start()
    for name in name_list2:
        Thread(target=eat2, args=(name, )).start()


# 递归锁可以解决互斥锁的死锁问题
# 互斥锁
    # 两把锁
```

```
48          # 多个线程抢
49  # 递归锁
50          # 一把锁
51          # 多个线程抢
52  # 递归锁好不好
53  # 递归锁并不是一个好的解决方案
54  # 死锁现象的发生不是互斥锁的原因
55  # 而是程序员的逻辑有问题
```