进程与线程

- 讲师：pansir

基本概念

多任务与多进程

操作系统 -> 进程 -> 线程

子任务与多线程

某个进程下启动的子任务就叫做线程

Python实现多任务

主要方案：

1. 多进程
2. 一个进程+多线程
3. 多进程+多线程，同时执行更多的任务，但是模型很复杂，实际很少用

进程通信

多任务执行时候，任务之间通信需要通过特殊手段，比如：队列

小结

1. 线程是最小的执行单元
2. 多进程和多线程涉及到同步，数据共享，编写和调试更困难

Python多进程

multiprocessing

`multiprocessing`支持跨平台多进程

```
from multiprocessing import Process
import os

def run_proc(name):
    print('正在执行的子进程 %s (%s)...' % (name, os.getpid()))
```

```
if __name__ == '__main__':
    print('父进程 %s.' % os.getpid())
    p = Process(target=run_proc, args=('test',))
    print('子进程准备执行.')
    p.start()
    p.join()
    print('子进程执行完毕.')
```

进程池

通过进程池批量创建子进程

```
from multiprocessing import Pool
import os, time, random


def long_time_task(name):
    print('执行任务  %s (%s)...' % (name, os.getpid()))
    start = time.time()
    time.sleep(random.random() * 3)
    end = time.time()
    print('任务 %s 执行了 %0.2f 秒.' % (name, (end - start)))


if __name__ == '__main__':
    print('当前进程的父进程号 %s.' % os.getpid())
    p = Pool(4)
    for i in range(5):
        p.apply_async(long_time_task, args=(i,))
    print('等待所有子进程加载...')
    p.close()
    p.join()
    print('所有子进程执行完毕.')
```

进程间通信

进程之间可以通过`multiprocessing`提供的`Queue`交换数据

以`Queue`为例，模拟两个子进程，一个往queue里写数据，一个从queue里读数据

```
from multiprocessing import Process, Queue
import os, time, random

def write(q):
    print('执行写操作的进程id: %s' % os.getpid())
    for value in ['A', 'B', 'C']:
```

```
        print('往队列写入 %s ...' % value)
        q.put(value)
        time.sleep(random.random())

def read(q):
    print('执行读操作的进程id: %s' % os.getpid())
    while True:
        value = q.get(True)
        print('从队列读取 %s .' % value)


if __name__ == '__main__':
    q = Queue()
    pw = Process(target=write, args=(q,))
    pr = Process(target=read, args=(q,))
    pw.start()
    pr.start()
    pw.join()
    pr.terminate()
```

小结

1. windows下可以使用`multiprocessing`实现多进程
2. 进程间通信通过`Queue`，`Pipes`实现

Python多线程

多进程可以完成多任务，也可以由一个进程内的多线程完成

`threading`封装了`_thread`模块，足够使用日常多线程操作

启动线程

把函数传入`Thread`，调用`start()`执行

```
import time, threading

def loop():
    print('当前执行的线程是:  %s ' % threading.current_thread().name)
    n = 0
    while n < 5:
        n = n + 1
        print('线程 %s >>> 迭代 %s' % (threading.current_thread().name, n))
        time.sleep(1)
    print('当前线程 %s 执行结束.' % threading.current_thread().name)


print('当前执行的线程是:  %s ' % threading.current_thread().name)
t = threading.Thread(target=loop, name='LoopThread')
```

```
t.start()
t.join()
print('当前线程 %s 执行完毕.' % threading.current_thread().name)
```

线程锁

多进程：每个进程独享一个变量，各进程间互不影响

多线程：所有变量由所有线程共享，任何线程都可以修改，存在同时修改一个变量情况，导致变量错乱

模拟银行存取

```
import time, threading

balance = 0

def change_it(n):
    global balance
    balance = balance + n
    balance = balance - n


def run_thread(n):
    for i in range(1000000):
        change_it(n)

t1 = threading.Thread(target=run_thread, args=(5,))
t2 = threading.Thread(target=run_thread, args=(8,))
t1.start()
t2.start()
t1.join()
t2.join()
print(balance)
```

引入线程锁

当线程进行读写操作时候，加上线程锁，用完之后再释放

```
import time, threading
lock = threading.Lock()

balance = 0


def change_it(n):
    global balance
    balance = balance + n
    balance = balance - n
```

```
def run_thread(n):
    for i in range(1000000):
        lock.acquire()
        try:
            change_it(n)
        finally:
            lock.release()

t1 = threading.Thread(target=run_thread, args=(5,))
t2 = threading.Thread(target=run_thread, args=(8,))
t1.start()
t2.start()
t1.join()
t2.join()
print(balance)
```

```
0
```

锁的优点与缺点

优点：某段关键代码，只能由一个线程从头到尾执行，保证数据准确

缺点：

1. 线程不能并发
2. 出现死锁

多线程局部变量

引入`TheadLocal`来管理每个线程下当变量

```
import threading

local_school = threading.local()

def process_student():
    std = local_school.student
    print('Hello, %s (in %s)' % (std, threading.current_thread().name))

def process_thread(name):
    local_school.student = name
    process_student()

t1 = threading.Thread(target= process_thread, args=('liudan',), name='Thread-A')
t2 = threading.Thread(target= process_thread, args=('shangcl',), name='Thread-B')
```

```
t1.start()
t2.start()
t1.join()
t2.join()
```

解释：

1. `local_school.student`是局部变量，线程自己独享
2. 常用场景：每个线程绑定一个数据库连接，HTTP请求，用户身份信息

小结

1. 多线程编程很复杂，既要用锁来隔离，又要小心死锁
2. python解释器由于本身的基础设计有GIL锁，导致多线程无法利用多核心
3. `ThradLocal`解决线程内的传参问题

进程与线程对比

多任务设计模式

	多进程模式	**多线程模式**
设计模式	Master-Worker	Master-Worker
稳定性	高，一个子进程崩了，不会影响其他子进程	低，任何一个线程崩掉，导致整个进程崩溃，因为所有线程共享进程的内存
开销	大，操作系统能同时运行的进程数有限，太多了调度成问题	低，速度略快

线程切换

只要数量多了，效率一定低。当任务数过多，操作系统忙着切换任务，浪费很多时间，就没时间执行任务了

密集型操作

计算密集型：需要大量计算，消耗CPU资源

IO密集型：涉及网络/磁盘IO都是IO密集型任务，消耗CPU资源少，任务大部分时间都在等待IO操作完成

异步IO

因为CPU和IO速度差异太大，考虑使用多进程/多线程来支持多任务并发执行