进程：单个用户启动后的单个程序，每个进程使用单独的内存空间

涉及模块：

multiprocessing包的\_\_init\_\_模块(实际为context模块)，适用于Unix和Windows系统

方法：

\_\_init\_\_(target=None,args=(),kwargs={})：target为函数名称，args为函数参数的tuple，kwargs为函数参数的dict

start()：启动子进程

join(timeout=None)：未提供参数时，等待子进程结束；当提供参数时，至少阻塞timeout时间

代码实例：

from multiprocessing import Process  
import os  
def f(process\_name):  
 print("%s的进程ID ：%s"%(process\_name,os.getpid()))  
 print("%s的父进程ID ：%s"%(process\_name,os.getppid()))  
  
def main():  
 process1 = Process(target=f,args=("branch1 line",))  
 process1.start()  
 process2 = Process(target=f,args=("branch2 line",))  
 process2.start()  
 process1.join()  
 process2.join()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

进程池：进程池内进程依然独享内存空间，当进程执行完当前任务空闲时，立即分配下一个任务

涉及模块：

multiprocessing包的\_\_init\_\_模块(实际为context模块)

方法：

\_\_init\_\_(process=core\_number)：创建进程池实例

apply(func,args,kwargs)：进程池内进程同步进行(即完成当前进程再进行下一个)

apply\_async(func,args,kwargs)：进程池内进程异步进行

close()：保证最后没有新任务进入到进程池

join()：等待所有任务完成

代码实例：

import os,random  
from time import sleep  
from multiprocessing import Pool  
def f(i):  
 print("第%d个进程开始执行，PID：%s"%(i,os.getpid()))  
 sleep(random.random()\*3)  
 print("第%d个进程执行完毕，PID：%s" %(i, os.getpid()))  
  
def main():  
 p = Pool(processes=4)  
 for i in range(6):  
 p.apply\_async(func=f,args=(i,))  
 p.close()  
 p.join()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

进程间通信

涉及模块：

multiprocessing包的\_\_init\_\_模块(实际为context模块)

方法：

put(item[,block=True[,timeout=0]])：向队列中放入内容，当block为True

get([block=True[,timeout=0]])：从队列中取出，当block为True，则queue无数据时，会进行阻塞等待，并且等待时间为timeout(默认为0，无限等待)

empty()：队列是否为空

实例代码：

from multiprocessing import Process,Queue  
def square(numbers,queue):  
 for i in range(numbers):  
 queue.put(i\*i)  
  
def cube(numbers,queue):  
 for i in range(numbers):  
 queue.put(i\*i\*i)  
  
def get(queue):  
 while not queue.empty():  
 print(queue.get())  
  
def main():  
 numbers = 5  
 queue = Queue()  
 process1 = Process(target=square,args=(numbers,queue))  
 process2 = Process(target=cube,args=(numbers,queue))  
 process3 = Process(target=get,args=(queue,))  
 process1.start()  
 process2.start()  
 process3.start()  
 process1.join()  
 process2.join()  
 process3.join()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

线程：CPU中的最小调度单元，同一进程的不同线程，共享同一内存空间

锁：由于同一进程下的线程共享同一内存空间，所以会造成同时读取内容，所以需要使用互斥锁防止同时读取

涉及模块：

threading模块的Thread、Lock

Thread方法：

\_\_init\_\_(target=None,args=None,kwargs=None)

参数释义：

target：run方法调用的函数名称

args：函数的参数值元组

kwargs：函数的关键字参数值

Lock方法：

\_\_init\_\_(blocking=True, timeout=-1)

acquire()：请求锁，将锁置于lock状态

release()：释放锁，将锁置于unlock状态。如果试图释放unlock锁，将会引发RuntimeError异常

参数释义：

blocking：是否阻塞(block直到unlock)，默认为True

timeout：当block为True时，阻塞直到unlock和timeout中的最大值。默认-1不指定超时时间

实例代码：

from threading import Thread,Lock  
import time  
class Account:  
 def \_\_init\_\_(self):  
 self.\_balance = 0  
 self.\_lock = Lock()  
  
 def desipot(self,money):  
 self.\_lock.acquire()  
 new\_balance = self.\_balance + money  
 time.sleep(0.01)  
 self.\_balance = new\_balance  
 self.\_lock.release()  
  
 @property  
 def balance(self):  
 return self.\_balance  
  
class Add\_money\_thread(Thread):  
 def \_\_init\_\_(self,account,money):  
 super().\_\_init\_\_()  
 self.\_account = account  
 self.\_money = money  
  
 def run(self):  
 self.\_account.desipot(self.\_money)  
  
def main():  
 account = Account()  
 thread\_pool = []  
 for \_ in range(100):  
 add\_money\_thread = Add\_money\_thread(account, 1)  
 thread\_pool.append(add\_money\_thread)  
 add\_money\_thread.start()  
 for i in thread\_pool:  
 i.join()  
 print("账户内余额为：%s"%account.balance)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

采用多进程(多线程)的条件：计算密集型任务 or I/O密集型任务

**多进程**：因为线程切换需要消耗CPU计算能力，并且由于GIL特性无法使用多核特性，而计算任务是不会减少的，所以多线程只会减少计算机性能，使用多进程可以在多核间灵活切换，所以计算密集型任务使用多进程

**多线程**：但是I/O密集型任务由于大部分时间处于等待I/O传输(由于CPU运算速度远高于I/O速度)，所以此时切换线程会大幅增加计算机性能，所以I/O密集型任务使用多线程

并发和并行的差异：

**并发(concurrency)**：在单CPU上，将时间段切分为更小的时间片，不同任务在不同的时间片切换进行执行，但是并不是同一个时间点同时执行。多线程和多进程都属于并发

**并行(parallel)**：在多CPU计算机上，不同任务在不同CPU上同一时间点同时执行。只有多进程属于并行

协程(coroutine)：单线程+异步I/O的编程模型，基于事件驱动

涉及模块：

asyncio

相关第三方模块

aiohttp：提供异步的HTTP客户端和服务器

aiofiles

相关关键字：

async：用来定义支持coroutine的方法

await：用于调用asyncio中的大部分方法。await必须在async方法中调用，并且调用的方法必须为async方法

函数：

run(coro, \*, debug=False)：创建事件循环，运行协成，关闭循环。通常coro为顶部函数调用main()

await sleep(delay)：暂停当前任务，允许其他任务运行

create\_task(func)：创建协程任务(func含参数)，返回Task对象task。后续搭配await task调用

await gather(\*aws,loop=None)：同时调用的不同async方法(带参数调用)，可以在其中间进行切换

Queue属性/方法：

maxsize：队列的容量

\_\_init\_\_(maxsize=0,\*,loop=None)：maxsize为队列最大容量(<=0时，容量无限大)，返回队列对象

empty()：队列是否为空

qsize()：返回队伍内的item数

full()：队列是否已满，maxsize需大于0才会返回True

await get()：从队列移除并返回item，当队列为空时，等待队列填入新值

get\_nowait()：立刻从队列返回item，如果队列为空，则引发QueueEmpty异常

await put(item)：item填入队列，当队列full，等待留出空间

put\_nowait(item)：item立刻填入队列，当队伍full，引发QueueFull异常

await join()：当item填入队列时，count增加；当执行task\_done后，count较少。最终当count为0，取消阻塞

task\_done()：意味一次排队任务完成，通常由消费者调用

Task方法：

cancel()：退出task的协程执行(含await关键字部分无法执行)

协程链(coroutine chain)：当gather中调用的某个async方法包含多个 async方法(链)调用时，该链为sync，后续方法需要前面的方法完成，然后再执行

生产者(producer)/消费者(consumer)模型：

代码实例：

import time  
import asyncio  
async def counter():  
 print("one")  
 await asyncio.sleep(1)  
 print("two")  
  
async def main():  
 start = time.perf\_counter()  
 await asyncio.gather(counter(), counter(), counter())  
 end = time.perf\_counter()  
 print("sum cost time：%.2f"%(end-start))  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 asyncio.run(main())