1 进程以及状态

1. 进程

程序：例如 xxx.py 这是**程序**，是一个静态的

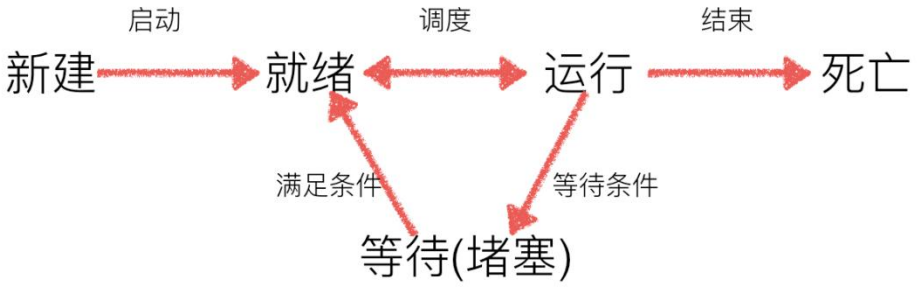
进程：一个程序运行起来后，**代码+用到的资源** 称之为进程，它**是操作系统分配资源的基本单元。**

Cpu，内存，文件，socket 对象

不仅可以通过线程完成多任务，进程也是可以的

2. 进程的状态

工作中，任务数往往大于 cpu 的核数，即一定有一些任务正在执行，而另外一些任务在等待 cpu 进行执行，因此导致了有了不同的状态



- 就绪态：运行的条件都已经满足，正在等在 cpu 执行
- 执行态：cpu 正在执行其功能
- 等待态：等待某些条件满足，例如一个程序 sleep 了，此时就处于等待态

top
ps
R 状态 运行
S 状态 睡眠
看到 R 状态是两次采样之间的一个时间片变化分析计算
cat /proc/cpuinfo 查看 Linux 的 cpu 的核数

3 Linux 下的进程管理

启动进程：手工启动 调度启动

命令	含 义
ps	查看系统中的进程
top	动态显示系统中的进程
kill	向进程发送信号（包括后台进程）
crontab	用于安装、删除或者列出用于驱动 cron 后台进程的任务。
bg	将挂起的进程放到后台执行

备注：

进程 process：是 os 的最小单元 os 会为每个进程分配大小为 4g 的虚拟内存空间，其中 1g 给内核空间 3g 给用户空间{**代码区 数据区 堆栈区**}

cat /proc/cpuinfo 查看 CPU 个数

ps 查看活动进程 ps -aux 查看所有的进程 **ps -aux| grep 'aa'** 查找指定（aa）进程 **ps -elf 可以显示父子进程关系**

进程状态：执行 就绪 等待状态

```
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root           1       0  0  80   0 - 40057 -      09:29 ?        00:00:03 /sbin/init auto noprompt
1 S root           2       0  0  80   0 -     0 -      09:29 ?        00:00:00 [kthreadd]
1 I root           3       2  0  60 -20 -     0 -      09:29 ?        00:00:00 [rcu_gp]
1 I root           4       2  0  60 -20 -     0 -      09:29 ?        00:00:00 [rcu_par_gp]
1 I root           6       2  0  60 -20 -     0 -      09:29 ?        00:00:00 [kworker/0:0H-kb]
1 I root           7       2  0  80   0 -     0 -      09:29 ?        00:00:00 [kworker/0:1-eve]
1 I root           9       2  0  60 -20 -     0 -      09:29 ?        00:00:00 [mm_percpu_wq]
1 S root          10       2  0  80   0 -     0 -      09:29 ?        00:00:00 [ksoftirqd/0]
1 I root          11       2  0  80   0 -     0 -      09:29 ?        00:00:04 [rcu_sched]
1 S root          12       2  0 -40   - -     0 -      09:29 ?        00:00:00 [migration/0]
5 S root          13       2  0   9   - -     0 -      09:29 ?        00:00:00 [idle_inject/0]
1 S root          14       2  0  80   0 -     0 -      09:29 ?        00:00:00 [cpuhp/0]
5 S root          15       2  0  80   0 -     0 -      09:29 ?        00:00:00 [kdevtmpfs]
```


ps -aux 看%cpu(cpu 使用量) %mem（内存使用量） stat 状态{**S 睡眠 T 暂停 R 运行 Z 僵尸**}

top 显示前 20 条进程，动态的改变，按 q 退出

```
top - 16:24:25 up 284 days,  4:59,  1 user,  load average: 0.10, 0.05, 0.01

Tasks: 115 total,   1 running, 114 sleeping,   0 stopped,   0 zombie

Cpu(s):  0.1%us,  0.0%sy,  0.0%ni, 99.8%id,  0.0%wa,  0.0%hi,  0.1%si,  0.0%st

Mem:   4074364k total,  3733628k used,   340736k free,   296520k buffers
```

```
Swap:  2104504k total,    40272k used,  2064232k free,   931680k cached


PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
11836 root      15   0  2324 1028  800 R  0.3  0.0   0:00.02 top
27225 root      25   0 1494m 696m  11m S  0.3 17.5   2304:03 java
1 root      18   0  2072  620  532 S  0.0  0.0   7:04.48 init
```

第一行分别显示：

```
16:24:25 当前时间、
up 284 days,  4:59 系统启动时间、
1 user 当前系统登录用户数目、
load average: 0.10, 0.05, 0.01 平均负载（1 分钟,10 分钟,15 分钟）。
```

平均负载（load average），一般对于单个 cpu 来说，负载在 0～1.00 之间是正常的，超过 1.00 须引起注意。在多核 cpu 中，系统平均负载不应该高于 cpu 核心的总数。

第二行分别显示：

```
115 total 进程总数、
1 running 运行进程数、
114 sleeping 休眠进程数、
0 stopped 终止进程数、
0 zombie 僵死进程数。
```

第三行：

```
0.1%us          %us 用户空间占用 cpu 百分比;
0.0%sy        %sy 内核空间占用 cpu 百分比;
0.0%ni        %ni 用户进程空间内改变过优先级的进程占用 cpu 百分比;
99.8%id       %id 空闲 cpu 百分比，反映一个系统 cpu 的闲忙程度。越大越空闲;
0.0%wa        %wa 等待输入输出（I/O）的 cpu 百分比;
0.0%hi        %hi 指的是 cpu 处理硬件中断的时间;
0.1%si        %si 值的是 cpu 处理软件中断的时间;
0.0%st        %st 用于有虚拟 cpu 的情况，用来指示被虚拟机偷掉的 cpu 时间。
```

第四行（Mem）：

```
4074364k total          total 总的物理内存;

3733628k used           used 使用物理内存大小;

340736k free free 空闲物理内存;

296520k buffers         buffers 用于内核缓存的内存大小

查看系统内存占用情况

free   内存泄露     python 循环引用会发生内存泄露
```

第五行（Swap）：

```
2104504k total          total 总的交换空间大小;

40272k used  used 已经使用交换空间大小;

2064232k free           free 空间交换空间大小;

931680k cached          cached 缓冲的交换空间大小
```

buffers 与 cached 区别：**buffers 指的是块设备的读写缓冲区，cached 指的是文件系统本身的页面缓存**。他们都是 Linux 系统底层的机制，为了加速对磁盘的访问。

然后下面就是和 ps 相仿的各进程情况列表了

第六行：
PID　　进程号
USER 运行用户
PR
优先级，PR(Priority)优先级
NI　　任务 nice 值
VIRT　　进程使用的虚拟内存总量，单位 kb。VIRT=SWAP+RES
RES　物理内存用量
SHR　共享内存用量
S　该进程的状态。其中 S 代表休眠状态；D 代表不可中断的休眠状态；R 代表运行状态；Z 代表僵死状态；T 代表停止或跟踪状态
%CPU　该进程自最近一次刷新以来所占用的 CPU 时间和总时间的百分比
%MEM　该进程占用的物理内存占总内存的百分比
TIME+　累计 cpu 占用时间
COMMAND 该进程的命令名称，如果一行显示不下，则会进行截取。内存中的进程会有一个完整的命令行

查看当前窗口启动的任务情况

python 1.while 死循环.py & 让进程后台运行

vi a.c &(&表示后台运行)，一个死循环，按 ctrl+z 可以把进程暂停，再执行[bg 作业 ID]可以将该进程带入后台。利用 jobs 可以查看后台任务，fg 1 把后台任务带到前台，这里的 1 表示作业 ID

bg 让暂停的进程在后台运行

fg 拉到前台

jobs 看后台任务

kill -l 查看系统的所有信号

kill -9 进程号➔表示向某个进程发送 9 号信号，从而杀掉某个进程

利用 pkill -f a 可以杀死进程名为 a 的进程,如果有空格，用\转义

pkill -f python\ 1.while 死循环.py

free 命令用来查看物理内存

fdisk -l 查看磁盘及磁盘分区情况

df -h 查看磁盘剩余空间

设置定时任务

crontab –e 设置当前用户定时任务

vim /etc/crontab 设置定时任务

crontab -l 查看当前自己设置的定时任务

killall wd_08_三个进程.py 按进程名字杀掉进程

2 进程的创建-multiprocessing

multiprocessing 模块就是跨平台版本的多进程模块，提供了一个 Process 类来代表一个进程对象，这个对象可以理解为是一个独立的进程，可以执行另外的事情

1. 两个 while 循环一起执行

```
# -*- coding:utf-8 -*-
from multiprocessing import Process
import time


def run_proc():
    """子进程要执行的代码"""
    while True:
        print("----2----")
        time.sleep(1)


if __name__=='__main__':
    p = Process(target=run_proc)
    p.start()
    while True:
        print("----1----")
        time.sleep(1)
```

说明

- 创建子进程时，只需要传入一个执行函数和函数的参数，创建一个 Process 实例，用 start()方法启动

孤儿进程 --- 父进程退出（kill 杀死父进程），子进程变为孤儿

僵尸进程 --- 子进程退出，父进程在忙碌，没有回收它，要避免僵尸

Python 进程变为僵尸进程后，名字会改变

2. 获取进程 pid

```
# -*- coding:utf-8 -*-
from multiprocessing import Process
import os
import time

def run_proc():
    """子进程要执行的代码"""
    print('子进程运行中，pid=%d...' % os.getpid())  # os.getpid 获取当前进程的进程号
    print('子进程将要结束...')

if __name__ == '__main__':
    print('父进程 pid: %d' % os.getpid())  # os.getpid 获取当前进程的进程号
    p = Process(target=run_proc)
    p.start()
```

获取父亲的 pid

os.getppid()

进程组　方便管理进程

kill -9 -3811　可以直接杀掉某一个进程组

3. Process 语法结构如下：

Process(group , target , name , args , kwargs)

- target：如果传递了函数的引用，可以让这个子进程就执行这里的代码
- args：给 target 指定的函数传递的参数，以元组的方式传递
- kwargs：给 target 指定的函数传递命名参数，keyword 参数
- name：给进程设定一个名字，可以不设定
- group：指定进程组，大多数情况下用不到

Process 创建的实例对象的常用方法：

- start()：启动子进程实例（创建子进程）
- is_alive()：判断进程子进程是否还在活着
- join([timeout])：是否等待子进程执行结束，或等待多少秒--**回收子进程尸体**
- terminate()：不管任务是否完成，立即终止子进程

Process 创建的实例对象的常用属性：

- name：当前进程的别名，默认为 Process-N，N 为从 1 开始递增的整数
- pid：当前进程的 pid（进程号）

4. 给子进程指定的函数传递参数

```
# -*- coding:utf-8 -*-
from multiprocessing import Process
import os
from time import sleep


def run_proc(name, age, **kwargs):
    for i in range(10):
        print('子进程运行中，name= %s,age=%d ,pid=%d...' % (name, age, o
s.getpid()))
        print(kwargs)
        sleep(0.2)

if __name__=='__main__':
    p = Process(target=run_proc, args=('test',18), kwargs={"m":20})
```

```
    p.start()
    sleep(1)  # 1 秒中之后，立即结束子进程
    p.terminate()
    p.join()
```

运行结果:

```
子进程运行中，name= test,age=18 ,pid=45097...
{'m': 20}
子进程运行中，name= test,age=18 ,pid=45097...
{'m': 20}
子进程运行中，name= test,age=18 ,pid=45097...
{'m': 20}
子进程运行中，name= test,age=18 ,pid=45097...
{'m': 20}
子进程运行中，name= test,age=18 ,pid=45097...
{'m': 20}
```

5. 进程间是否共享全局变量

```
# -*- coding:utf-8 -*-
from multiprocessing import Process
import os
import time

nums = [11, 22]

def work1():
    """子进程要执行的代码"""
    print("in process1 pid=%d ,nums=%s" % (os.getpid(), nums))
    for i in range(3):
        nums.append(i)
        time.sleep(1)
        print("in process1 pid=%d ,nums=%s" % (os.getpid(), nums))

def work2():
    """子进程要执行的代码"""
    print("in process2 pid=%d ,nums=%s" % (os.getpid(), nums))

if __name__ == '__main__':
    p1 = Process(target=work1)
    p1.start()
    p1.join()

    p2 = Process(target=work2)
    p2.start()
```

运行结果:

```
in process1 pid=11349 ,nums=[11, 22]
in process1 pid=11349 ,nums=[11, 22, 0]
in process1 pid=11349 ,nums=[11, 22, 0, 1]
in process1 pid=11349 ,nums=[11, 22, 0, 1, 2]
in process2 pid=11350 ,nums=[11, 22]
```

3 进程、线程对比

功能

- 进程，能够完成多任务，比如 在一台电脑上能够同时运行多个 QQ
- 线程，能够完成多任务，比如 一个 QQ 中的多个聊天窗口

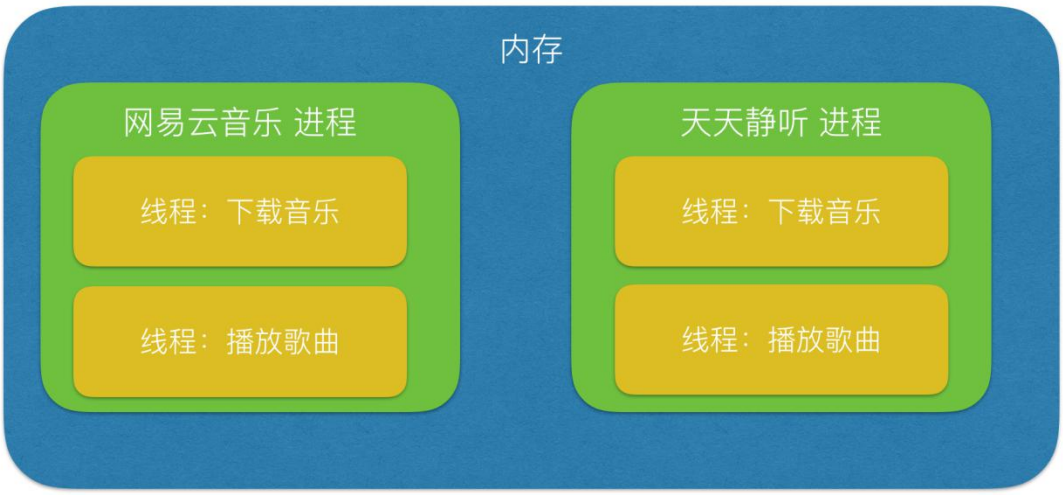

1.8K/s
18:14
41%
社交
QQ1
QQ2
QQ3
QQ4
QQ5
QQ6
QQ7
QQ8
QQ9

定义的不同

- 进程是系统进行资源分配和调度的一个独立单位.
- 线程是进程的一个实体,是 CPU 调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源.

区别

- 一个程序至少有一个进程,一个进程至少有一个线程.
- 线程的划分尺度小于进程(资源比进程少)，使得多线程程序的并发性高。
- 进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率



- 线线程不能够独立执行，必须依存在进程中

- 可以将进程理解为工厂中的一条流水线，而其中的线程就是这个流水线上的工人

优缺点

线程和进程在使用上各有优缺点：线程执行开销小，但不利于资源的管理和保护；而进程正相反。

4 进程间通信-Queue

Process 之间有时需要通信，操作系统提供了很多机制来实现进程间的通信。--还**有管道，共享内存**

1. Queue 的使用

可以使用 multiprocessing 模块的 Queue 实现多进程之间的数据传递，Queue 本身是一个消息列队程序，首先用一个小实例来演示一下 Queue 的工作原理：

```
#coding=utf-8
from multiprocessing import Queue
q=Queue(3) #初始化一个 Queue 对象，最多可接收三条 put 消息
q.put("消息 1")
q.put("消息 2")
print(q.full())  #False
q.put("消息 3")
print(q.full()) #True

#因为消息列队已满下面的 try 都会抛出异常，第一个 try 会等待 2 秒后再抛出异常，第二个 Try 会立刻抛出异常
try:
    q.put("消息 4",True,2)
except:
    print("消息列队已满，现有消息数量:%s"%q.qsize())

try:
    q.put_nowait("消息 4")
except:
    print("消息列队已满，现有消息数量:%s"%q.qsize())

#推荐的方式，先判断消息列队是否已满，再写入
if not q.full():
    q.put_nowait("消息 4")

#读取消息时，先判断消息列队是否为空，再读取
if not q.empty():
    for i in range(q.qsize()):
        print(q.get_nowait())
```

运行结果:

```
False
True
消息列队已满，现有消息数量:3
消息列队已满，现有消息数量:3
```

消息 1
消息 2
消息 3

说明

初始化 Queue()对象时（例如：q=Queue()），若括号中没有指定最大可接收的消息数量，或数量为负值，那么就代表可接受的消息数量没有上限（直到内存的尽头）；

- Queue.qsize()：返回当前队列包含的消息数量；
- Queue.empty()：如果队列为空，返回 True，反之 False ；
- Queue.full()：如果队列满了，返回 True,反之 False；
- Queue.get([block[, timeout]])：获取队列中的一条消息，然后将其从列队中移除，block 默认值为 True；

1）如果 block 使用默认值，且没有设置 timeout（单位秒），消息列队如果为空，此时程序将被阻塞（停在读取状态），直到从消息列队读到消息为止，如果设置了 timeout，则会等待 timeout 秒，若还没读取到任何消息，则抛出"Queue.Empty"异常；

2）如果 block 值为 False，消息列队如果为空，则会立刻抛出"Queue.Empty"异常；

- Queue.get_nowait()：相当 Queue.get(block=False)；
- Queue.put(item,[block[, timeout]])：将 item 消息写入队列，block 默认值为 True；

1）如果 block 使用默认值，且没有设置 timeout（单位秒），消息列队如果已经没有空间可写入，此时程序将被阻塞（停在写入状态），直到从消息列队腾出空间为止，如果设置了 timeout，则会等待 timeout 秒，若还没空间，则抛出"Queue.Full"异常；

2）如果 block 值为 False，消息列队如果没有空间可写入，则会立刻抛出"Queue.Full"异常；

- Queue.put_nowait(item)：相当 Queue.put(item, False)；

2. Queue 实例

我们以 Queue 为例，在父进程中创建两个子进程，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

# 写数据进程执行的代码:
def write(q):
    for value in ['A', 'B', 'C']:
        print('Put %s to queue...' % value)
        q.put(value)
        time.sleep(1)

# 读数据进程执行的代码:
def read(q):
    while True:
        if not q.empty():
            value = q.get(True)
            print('Get %s from queue.' % value)
            time.sleep(2)
        else:
            break

if __name__=='__main__':
    # 父进程创建 Queue，并传给各个子进程:
    q = Queue()
    pw = Process(target=write, args=(q,))
    pr = Process(target=read, args=(q,))
    # 启动子进程 pw，写入:
    pw.start()
    # 启动子进程 pr，读取:
    pr.start()
    # 等待 pw,pr 结束:
    pw.join()
    pr.join()
    # pr 进程里是死循环，无法等待其结束，只能强行终止:
    print(q.qsize())
    print('')
    print('所有数据都写入并且读完')
```

运行结果：

待给出

5 管道通信(不重要，使用起来不方便)

Pipe 直译过来的意思是“管”或“管道”，该种实现多进程编程的方式，和实际生活中的管（管道）是非常类似的。通常情况下，管道有 2 个口，而 Pipe 也常用来实现 2 个进程之间的通信，这 2 个进程分别位于管道的两端，一端用来发送数据，另一端用来接收数据。

使用 Pipe 实现进程通信，首先需要调用 multiprocessing.Pipe() 函数来创建一个管道。该函数的语法格式如下：

```
conn1, conn2 = multiprocessing.Pipe( [duplex=True] )
```

其中，conn1 和 conn2 分别用来接收 Pipe 函数返回的 2 个端口；duplex 参数默认为 True，表示该管道是双向的，即位于 2 个端口的进程既可以发送数据，也可以接受数据，而如果将 duplex 值设为 False，则表示管道是单向的，conn1 只能用来接收数据，而 conn2 只能用来发送数据。

另外值得一提的是，conn1 和 conn2 都属于 PipeConnection 对象，它们还可以调用表 2 所示的这些方法。

5 进程池 Pool

当需要创建的子进程数量不多时，可以直接利用 multiprocessing 中的 Process 动态成生多个进程，但如果是上百甚至上千个目标，手动的去创建进程的工作量巨大，此时就可以用到 multiprocessing 模块提供的 Pool 方法。

初始化 Pool 时，可以指定一个最大进程数，当有新的请求提交到 Pool 中时，如果池还没有满，那么就会创建一个新的进程用来执行该请求；但如果池中的进程数已经达到指定的最大值，那么该请求就会等待，直到池中有进程结束，才会用之前的进程来执行新的任务，请看下面的实例：

```
# -*- coding:utf-8 -*-
from multiprocessing.pool import Pool

import os, time, random

def worker(msg):
    t_start = time.time()
```

```
    print("%s 开始执行,进程号为%d" % (msg,os.getpid()))
    # random.random()随机生成 0~1 之间的浮点数
    time.sleep(random.random()*2)
    t_stop = time.time()
    print(msg,"执行完毕，耗时%0.2f" % (t_stop-t_start))

if __name__ == '__main__':

      po = Pool(3)  # 定义一个进程池，最大进程数 3
      for i in range(0,10):
          # Pool().apply_async(要调用的目标,(传递给目标的参数元祖,))
          # 每次循环将会用空闲出来的子进程去调用目标
          po.apply_async(worker,(i,))

      print("----start----")
      po.close()  # 关闭进程池，关闭后 po 不再接收新的请求
      po.join()  # 等待 po 中所有子进程执行完成，必须放在 close 语句之后
      print("-----end-----")
```

运行结果:

```
----start----
0 开始执行,进程号为 21466
1 开始执行,进程号为 21468
2 开始执行,进程号为 21467
0 执行完毕，耗时 1.01
3 开始执行,进程号为 21466
2 执行完毕，耗时 1.24
4 开始执行,进程号为 21467
3 执行完毕，耗时 0.56
5 开始执行,进程号为 21466
1 执行完毕，耗时 1.68
6 开始执行,进程号为 21468
4 执行完毕，耗时 0.67
7 开始执行,进程号为 21467
5 执行完毕，耗时 0.83
8 开始执行,进程号为 21466
6 执行完毕，耗时 0.75
9 开始执行,进程号为 21468
7 执行完毕，耗时 1.03
8 执行完毕，耗时 1.05
9 执行完毕，耗时 1.69
-----end-----
```

multiprocessing.Pool 常用函数解析：

- apply_async(func[, args[, kwds]]) ：使用非阻塞方式调用 func（并行执行，堵塞方式必须等待上一个进程退出才能执行下一个进程），args 为传递给 func 的参数列表，kwds 为传递给 func 的关键字参数列表；
- close()：关闭 Pool，使其不再接受新的任务；
- terminate()：不管任务是否完成，立即终止；
- join()：主进程阻塞，等待子进程的退出， **必须在 close 或 terminate 之后**使用；

进程池中的 Queue

如果要使用 Pool 创建进程，就需要使用 multiprocessing.Manager()中的 Queue()，而不是 multiprocessing.Queue()，否则会得到一条如下的错误信息：

RuntimeError: Queue objects should only be shared between processes through inheritance.

下面的实例演示了进程池中的进程如何通信：

```
# -*- coding:utf-8 -*-

# 修改 import 中的 Queue 为 Manager
from multiprocessing import Manager,Pool
import os,time,random

def reader(q):
    print("reader 启动(%s),父进程为(%s)" % (os.getpid(), os.getppid()))
    for i in range(q.qsize()):
        print("reader 从 Queue 获取到消息：%s" % q.get(True))

def writer(q):
    print("writer 启动(%s),父进程为(%s)" % (os.getpid(), os.getppid()))
    for i in "wangdao":
        q.put(i)

if __name__=="__main__":
    print("(%s) start" % os.getpid())
    q = Manager().Queue()  # 使用 Manager 中的 Queue
    po = Pool()
    po.apply_async(writer, (q,))

    time.sleep(1)  # 先让上面的任务向 Queue 存入数据，然后再让下面的任务开始从中取数据

    po.apply_async(reader, (q,))
    po.close()
```

```
    po.join()
    print("(%s) End" % os.getpid())
```

运行结果:

```
(11095) start
writer 启动(11097),父进程为(11095)
reader 启动(11098),父进程为(11095)
reader 从 Queue 获取到消息：w
reader 从 Queue 获取到消息：a
reader 从 Queue 获取到消息：n
reader 从 Queue 获取到消息：g
reader 从 Queue 获取到消息：d
reader 从 Queue 获取到消息：a
reader 从 Queue 获取到消息：o
(11095) End
```

6 应用：文件夹 copy 器（多进程版）

```
import multiprocessing
import os
import time
import random


def copy_file(queue, file_name,source_folder_name,  dest_folder_name):
    """copy 文件到指定的路径"""
    f_read = open(source_folder_name + "/" + file_name, "rb")
    f_write = open(dest_folder_name + "/" + file_name, "wb")
    while True:
        time.sleep(random.random())
        content = f_read.read(1024)
        if content:
            f_write.write(content)
        else:
            break
    f_read.close()
    f_write.close()
```

```
    # 发送已经拷贝完毕的文件名字
    queue.put(file_name)


def main():
    # 获取要复制的文件夹
    source_folder_name = input("请输入要复制文件夹名字:")

    # 整理目标文件夹
    dest_folder_name = source_folder_name + "[副本]"

    # 创建目标文件夹
    try:
        os.mkdir(dest_folder_name)
    except:
        pass  # 如果文件夹已经存在，那么创建会失败

    # 获取这个文件夹中所有的普通文件名
    file_names = os.listdir(source_folder_name)

    # 创建 Queue
    queue = multiprocessing.Manager().Queue()

    # 创建进程池
    pool = multiprocessing.Pool(3)

    for file_name in file_names:
        # 向进程池中添加任务
        pool.apply_async(copy_file, args=(queue, file_name, source_fold
er_name, dest_folder_name))

    # 主进程显示进度
    pool.close()

    all_file_num = len(file_names)
    while True:
        file_name = queue.get()
        if file_name in file_names:
            file_names.remove(file_name)

        copy_rate = (all_file_num-len(file_names))*100/all_file_num
        print("\r%.2f...(%s)" % (copy_rate, file_name) + " "*50, end="")
        if copy_rate >= 100:
            break
    print()
```

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