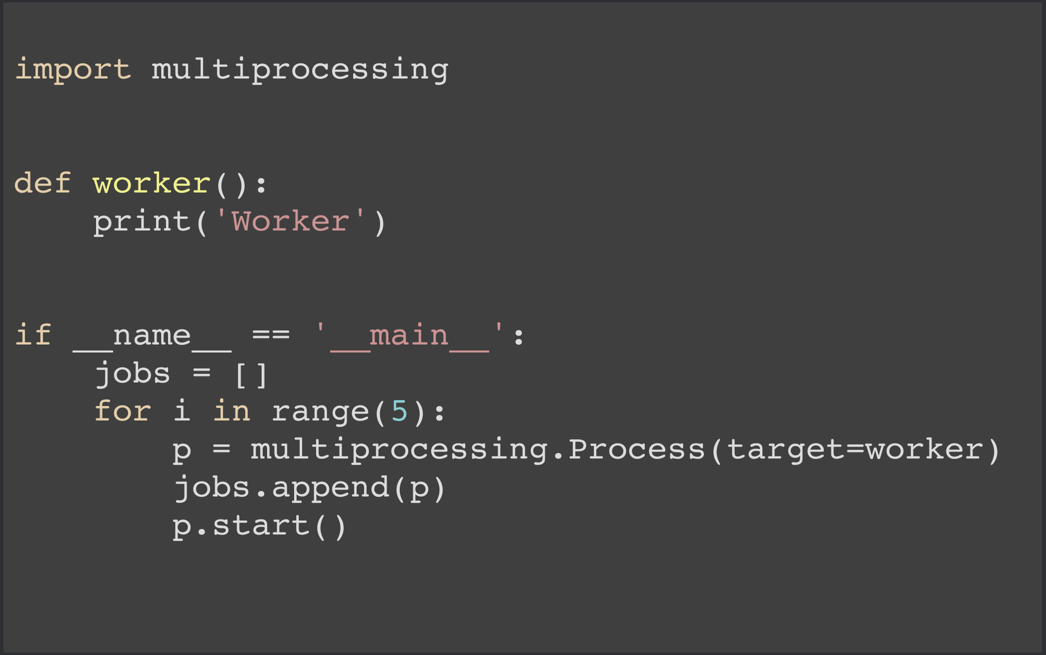
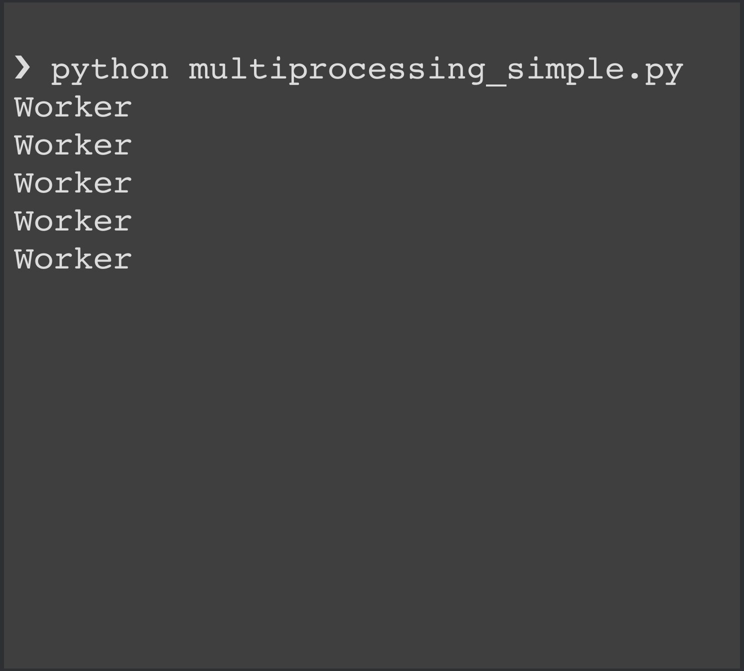
多进程编程

由于GIL（全局解释锁）的问题，多线程并不能充分利用多核处理器，如果是一个CPU计算型的任务，应该使用多进程模块 multiprocessing



```python
import multiprocessing


def worker():
    print('Worker')


if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=worker)
        jobs.append(p)
        p.start()
```



```
❯ python multiprocessing_simple.py
Worker
Worker
Worker
Worker
Worker
```

目标函数可传入参数

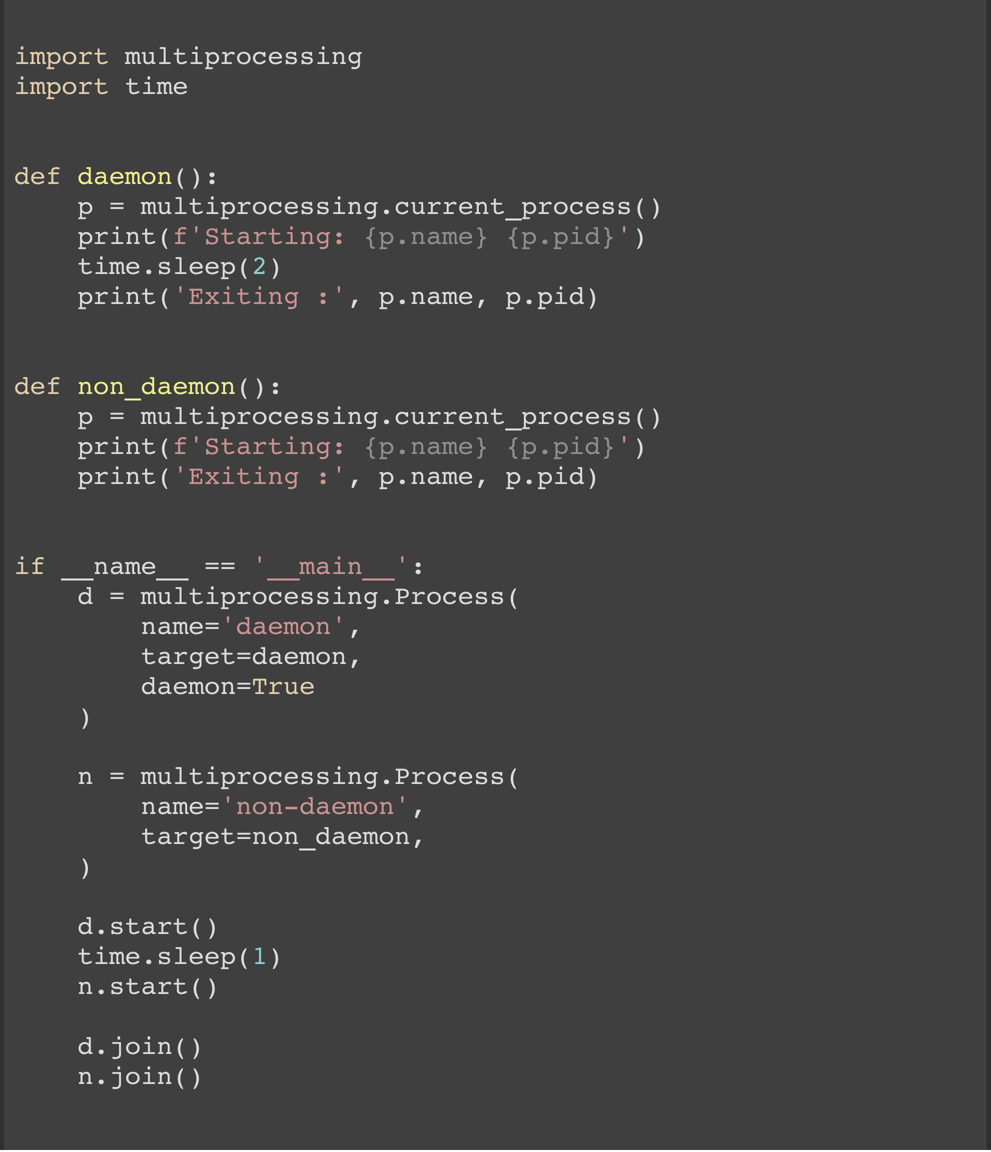
```python
import multiprocessing


def worker(num):
    print(f'Worker: {num}')


if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=worker,
                                    args=(i,))
        jobs.append(p)
        p.start()
```

```
❯ python multiprocessing_simpleargs.py
Worker: 0
Worker: 1
Worker: 2
Worker: 3
Worker: 4
```

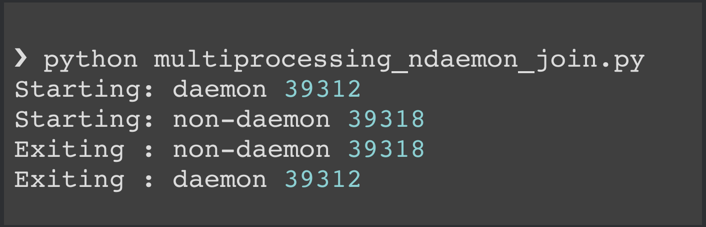
守护进程

```python
import multiprocessing
import time


def daemon():
    p = multiprocessing.current_process()
    print(f'Starting: {p.name} {p.pid}')
    time.sleep(2)
    print('Exiting :', p.name, p.pid)


def non_daemon():
    p = multiprocessing.current_process()
    print(f'Starting: {p.name} {p.pid}')
    print('Exiting :', p.name, p.pid)


if __name__ == '__main__':
    d = multiprocessing.Process(
        name='daemon',
        target=daemon,
        daemon=True
    )

    n = multiprocessing.Process(
        name='non-daemon',
        target=non_daemon,
    )

    d.start()
    time.sleep(1)
    n.start()
```

```
❯ python multiprocessing_daemon.py
Starting: daemon 38439
Starting: non-daemon 38442
Exiting : non-daemon 38442
```

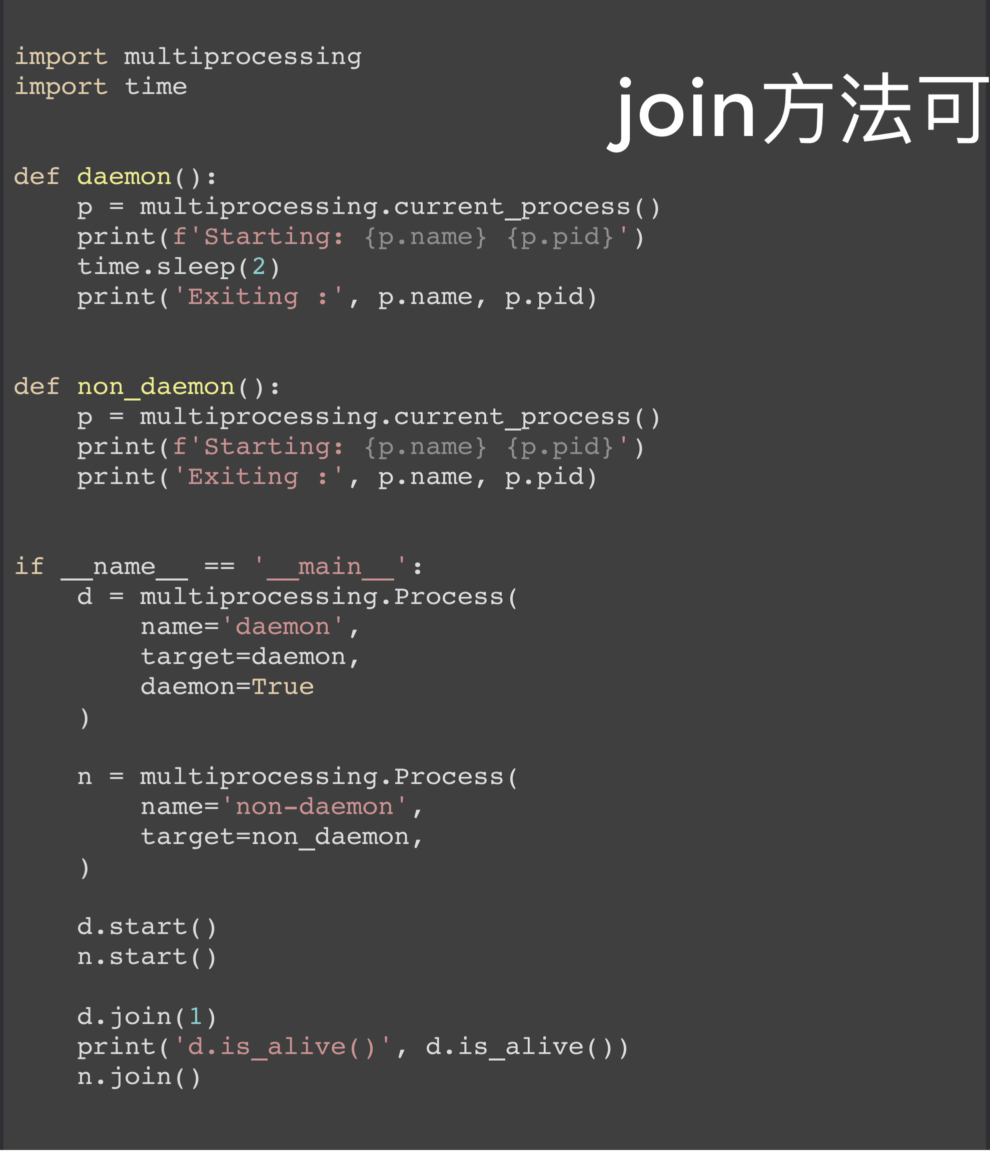


```python
import multiprocessing
import time


def daemon():
    p = multiprocessing.current_process()
    print(f'Starting: {p.name} {p.pid}')
    time.sleep(2)
    print('Exiting :', p.name, p.pid)


def non_daemon():
    p = multiprocessing.current_process()
    print(f'Starting: {p.name} {p.pid}')
    print('Exiting :', p.name, p.pid)


if __name__ == '__main__':
    d = multiprocessing.Process(
        name='daemon',
        target=daemon,
        daemon=True
    )

    n = multiprocessing.Process(
        name='non-daemon',
        target=non_daemon,
    )

    d.start()
    time.sleep(1)
    n.start()

    d.join()
    n.join()
```

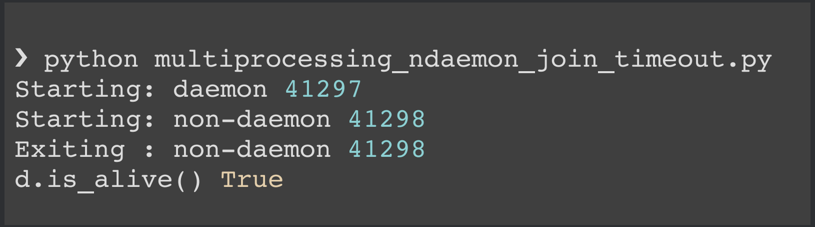


```
❯ python multiprocessing_ndaemon_join.py
Starting: daemon 39312
Starting: non-daemon 39318
Exiting : non-daemon 39318
Exiting : daemon 39312
```

join方法可设置超时参数



```python
import multiprocessing
import time


def daemon():
    p = multiprocessing.current_process()
    print(f'Starting: {p.name} {p.pid}')
    time.sleep(2)
    print('Exiting :', p.name, p.pid)


def non_daemon():
    p = multiprocessing.current_process()
    print(f'Starting: {p.name} {p.pid}')
    print('Exiting :', p.name, p.pid)


if __name__ == '__main__':
    d = multiprocessing.Process(
        name='daemon',
        target=daemon,
        daemon=True
    )

    n = multiprocessing.Process(
        name='non-daemon',
        target=non_daemon,
    )

    d.start()
    n.start()

    d.join(1)
    print('d.is_alive()', d.is_alive())
    n.join()
```



```
❯ python multiprocessing_ndaemon_join_timeout.py
Starting: daemon 41297
Starting: non-daemon 41298
Exiting : non-daemon 41298
d.is_alive() True
```

进程池

```python
from functools import lru_cache
from multiprocessing import Pool

@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)


pool = Pool(2)
pool.map(fib, [35] * 2)
```

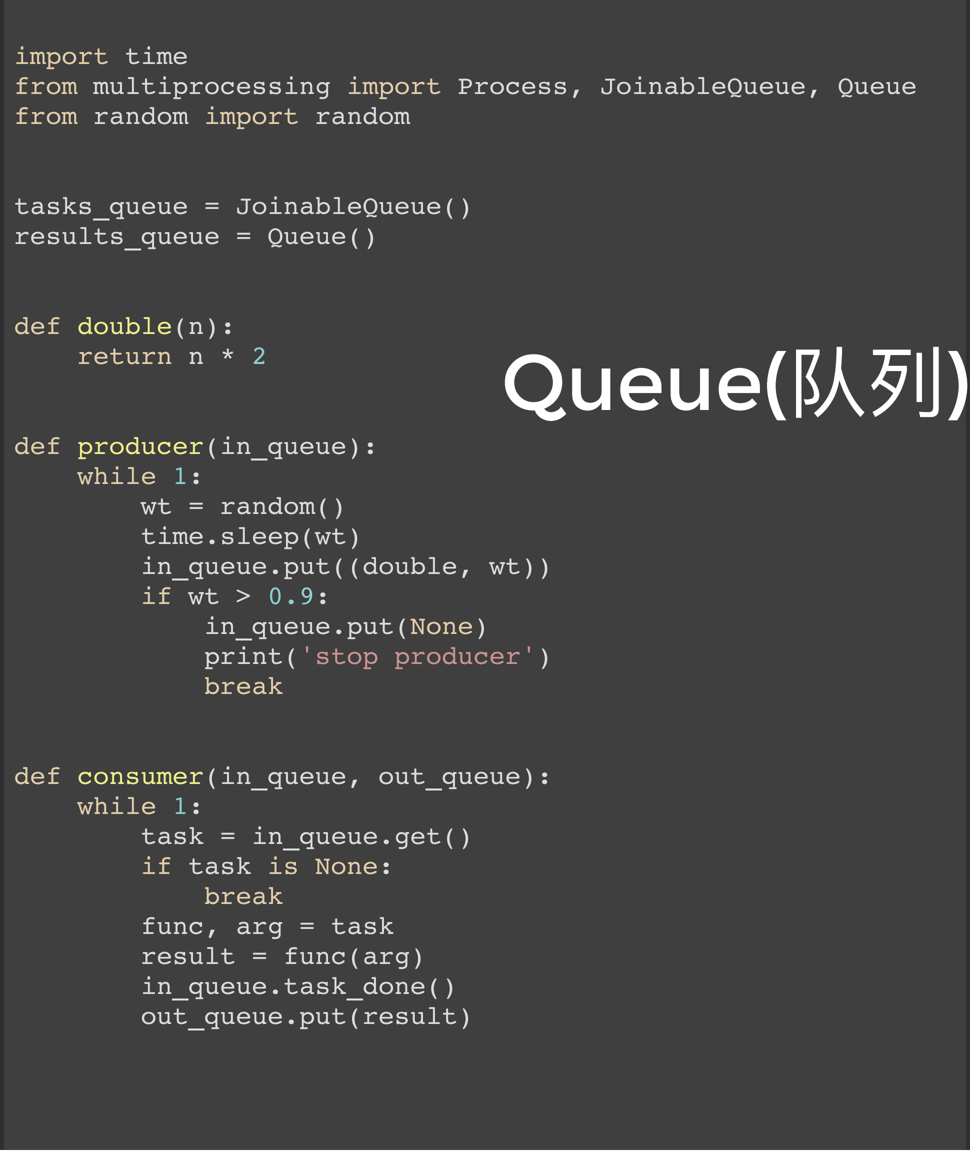
dummy

```
from multiprocessing import Pool
from multiprocessing.dummy import Pool
```

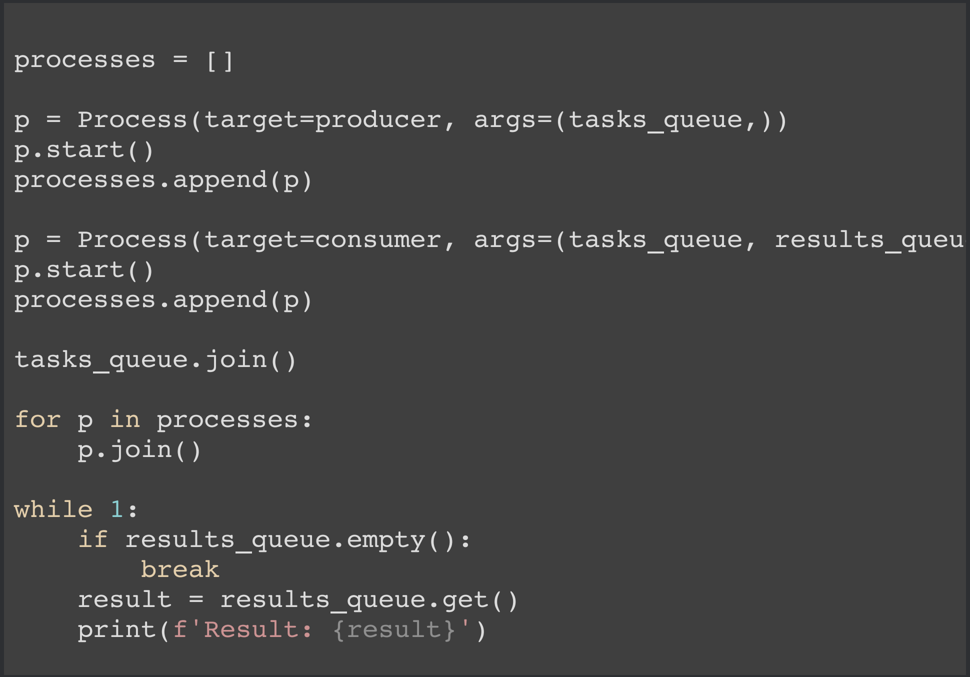
这种兼容的方式，这样在多线程/多进程之间切换非常方便。

我的技巧：如果一个任务拿不准是CPU密集还是I/O密集型，且没有其它不能选择多进程方式的因素，都统一直接上多进程模式

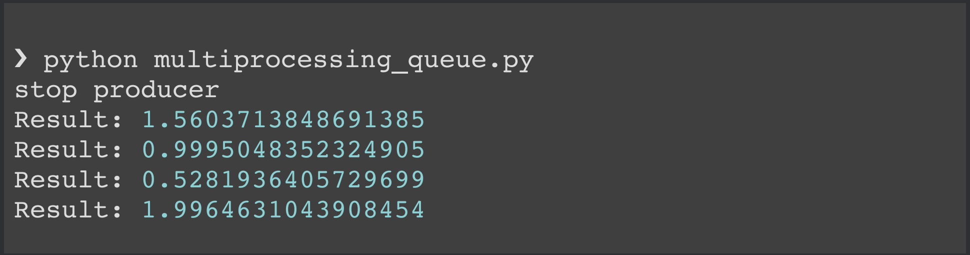
Queue(队列)



```python
import time
from multiprocessing import Process, JoinableQueue, Queue
from random import random


tasks_queue = JoinableQueue()
results_queue = Queue()


def double(n):
    return n * 2


def producer(in_queue):
    while 1:
        wt = random()
        time.sleep(wt)
        in_queue.put((double, wt))
        if wt > 0.9:
            in_queue.put(None)
            print('stop producer')
            break


def consumer(in_queue, out_queue):
    while 1:
        task = in_queue.get()
        if task is None:
            break
        func, arg = task
        result = func(arg)
        in_queue.task_done()
        out_queue.put(result)
```



```python
processes = []

p = Process(target=producer, args=(tasks_queue,))
p.start()
processes.append(p)

p = Process(target=consumer, args=(tasks_queue, results_queu
p.start()
processes.append(p)

tasks_queue.join()

for p in processes:
    p.join()

while 1:
    if results_queue.empty():
        break
    result = results_queue.get()
    print(f'Result: {result}')
```

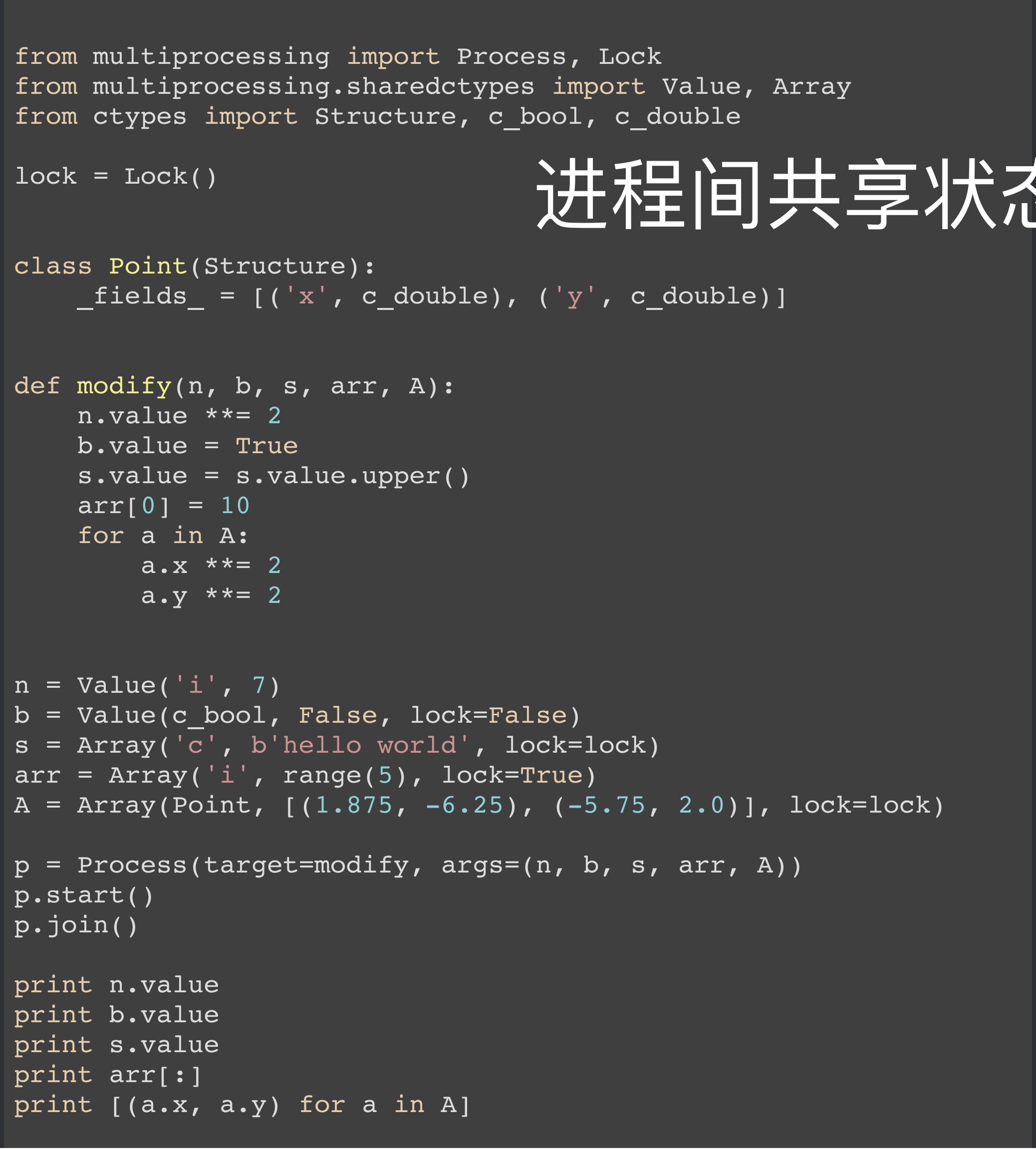


```
❯ python multiprocessing_queue.py
stop producer
Result: 1.5603713848691385
Result: 0.9995048352324905
Result: 0.5281936405729699
Result: 1.9964631043908454
```

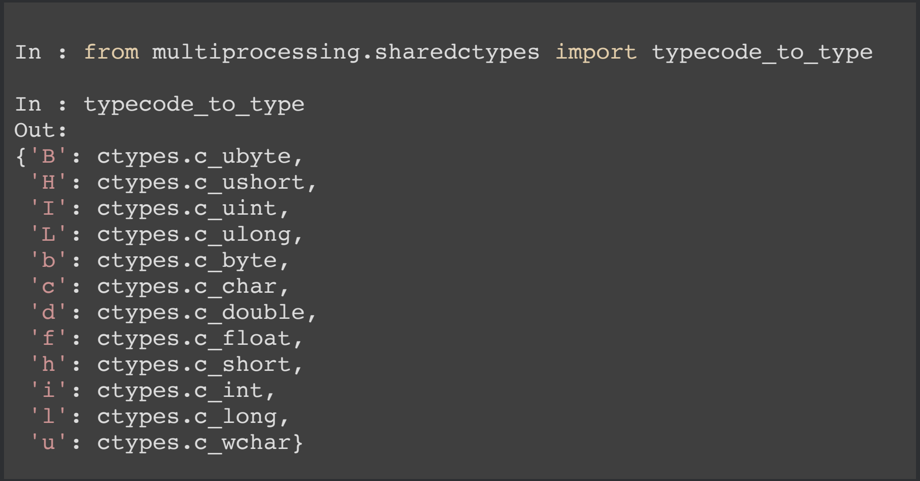
同步机制

multiprocessing的Lock、Condition、Event、RLock、Semaphore等同步原语和threading模块的API风格是一样的，用法也类似，就不展开了

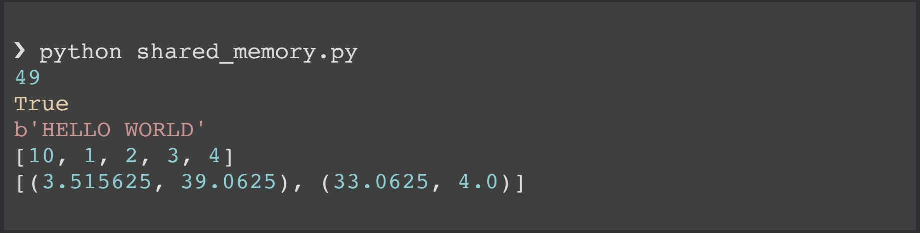
进程间共享状态 - 共享内存



```
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_bool, c_double

lock = Lock()


class Point(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]


def modify(n, b, s, arr, A):
    n.value **= 2
    b.value = True
    s.value = s.value.upper()
    arr[0] = 10
    for a in A:
        a.x **= 2
        a.y **= 2


n = Value('i', 7)
b = Value(c_bool, False, lock=False)
s = Array('c', b'hello world', lock=lock)
arr = Array('i', range(5), lock=True)
A = Array(Point, [(1.875, -6.25), (-5.75, 2.0)], lock=lock)

p = Process(target=modify, args=(n, b, s, arr, A))
p.start()
p.join()

print n.value
print b.value
print s.value
print arr[:]
print [(a.x, a.y) for a in A]
```



```
In : from multiprocessing.sharedctypes import typecode_to_type

In : typecode_to_type
Out:
{'B': ctypes.c_ubyte,
 'H': ctypes.c_ushort,
 'I': ctypes.c_uint,
 'L': ctypes.c_ulong,
 'b': ctypes.c_byte,
 'c': ctypes.c_char,
 'd': ctypes.c_double,
 'f': ctypes.c_float,
 'h': ctypes.c_short,
 'i': ctypes.c_int,
 'l': ctypes.c_long,
 'u': ctypes.c_wchar}
```



```
❯ python shared_memory.py
49
True
b'HELLO WORLD'
[10, 1, 2, 3, 4]
[(3.515625, 39.0625), (33.0625, 4.0)]
```

进程间共享状态 - 服务器进程

```python
from multiprocessing import Manager, Process

def modify(ns, lproxy, dproxy):
    ns.a **= 2
    lproxy.extend(['b', 'c'])
    dproxy['b'] = 0


manager = Manager()
ns = manager.Namespace()
ns.a = 1
lproxy = manager.list()
lproxy.append('a')
dproxy = manager.dict()
dproxy['b'] = 2

p = Process(target=modify, args=(ns, lproxy, dproxy))
p.start()
print(f'PID: {p.pid}')
p.join()

print(ns.a)
print(lproxy)
print(dproxy)
```

常见的共享方式有以下几种：

1. Namespace。创建一个可分享的命名空间。
2. Value/Array。和上面共享ctypes对象的方式一样。
3. dict/list。创建一个可分享的dict/list，支持对应数据结构的方法。
4. Condition/Event/Lock/Queue/Semaphore。创建一个可分享的对应同步原语的对象。

```
❯ python manager.py
PID: 45121
1
['a', 'b', 'c']
{'b': 0}
```

分布式的进程间通信

```python
# remote_server.py
from multiprocessing.managers import BaseManager

host = '127.0.0.1'
port = 9030
authkey = b'secret'

shared_list = []


class RemoteManager(BaseManager):
    pass


RemoteManager.register('get_list',
    callable=lambda: shared_list)
mgr = RemoteManager(address=(host, port), authkey=authkey)
server = mgr.get_server()
server.serve_forever()
```

```
❯ python remote_server.py
```

```python
# client.py
from multiprocessing.managers import BaseManager

host = '127.0.0.1'
port = 9030
authkey = b'secret'


class RemoteManager(BaseManager):
    pass


RemoteManager.register('get_list')
mgr = RemoteManager(address=(host, port), authkey=authkey)
mgr.connect()

l = mgr.get_list()
print(l)
l.append(1)
print(mgr.get_list())
```

```
❯ python3 client.py
[]
[1]
```

延伸阅读

1.https://zhuanlan.zhihu.com/p/22386793

2.https://pymotw.com/3/multiprocessing/index.html

3.https://docs.python.org/3.7/library/multiprocessing.html