

🔗 楼+之Python实战第10期 (/courses/1190)

Python 多进程与多线程

简介

本节实验重点学习 Python 中的多进程和多线程的实现方法，同时会涉及到多进程编程的进程通信、进程同步、进程池等概念和实现方式。

知识点

- Python3 开发多进程程序
- 进程间通信
- 进程同步
- 进程池
- Python3 多线程程序

多进程

我们常见的 Linux、Windows、Mac OS 操作系统，都是支持多进程的多核操作系统。所谓多进程，就是系统可以同时运行多个任务。例如我们的电脑上运行着 QQ、浏览器、音乐播放器、影音播放器等。在操作系统中，每个任务就是一个进程。每个进程至少做一件事，多数进程会做很多事，例如影音播放器，要播放画面，同时要播放声音，在一个进程中，就有很多线程，每个线程做一件事，多个线程同时运行就是多线程。可以在实验环境终端执行 `ps -ef` 命令来查看当前系统中正在运行的进程。

一个简单的多进程程序

`multiprocessing` 下的 `Process` 类封装了多进程操作，我们通过一个多进程版本的程序来看看它的使用方法，该程序写入 `/home/shiyanlou/multi.py`：

📁 楼+之Python实战第10期 (/courses/1190)
from multiprocessing import Process

```
def hello(name):
    # os.getpid() 用来获取当前进程 ID
    print('child process: {}'.format(os.getpid()))
    print('Hello ' + name)

# 当我们运行一个 Python 程序，系统会创建一个进程来运行程序，被称为主进程或父进程
# 前面课程中，我们写的程序都是单进程程序，即所有代码都运行在一个主进程中
# 下面的 main() 函数就运行在主进程中
def main():
    # 打印当前进程即主进程 ID
    print('parent process: {}'.format(os.getpid()))
    # 注意 Process 对象只是一个子任务，运行该任务时系统会自动创建一个子进程
    # 注意: args 参数要以 tuple 方式传入
    p = Process(target=hello, args=('shianlou', ))
    print('child process start')
    # 启动一个子进程来运行子任务，该进程运行的是 hello() 函数中的代码
    p.start()
    p.join()
    # 子进程完成后，继续运行主进程
    print('child process stop')
    print('parent process: {}'.format(os.getpid()))

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

在上面的程序中，首先从 `multiprocessing` 中导入 `Process` 类，然后定义了一个 `hello` 函数，打印 `hello + 传入的 name 值`，在 `main` 函数中，用 `Process` 类定义了一个子进程，这个子进程要执行的函数是 `hello`，传入的参数是 `shianlou`，然后调用 `start()` 方法，启动子进程，这时候子进程会调用 `hello` 函数，将 `shianlou` 作为参数传入，打印当前进程 ID 和 `hello shianlou` 后返回。`join()` 方法表示等待子进程运行结束后继续执行，所以在子进程返回后会继续打印父进程的 ID。

运行这个程序，会输出：

```
$ cd /home/shianlou
$ python3 multi.py
parent process: 22222
child process start
child process: 22223
Hello shianlou
child process stop
parent process: 22222
```

注意，每次运行程序，进程 ID 会有所不同。

利用多进程提高程序运行效率

计算机的两大核心为运算器和存储器。常说的手机配置四核、八核，指的就是 CUP 的数量，它决定了手机的运算能力；128G、256G 超大存储空间，指的就是手机存储数据的能力。当我们运行一个程序来计算 $3 + 5$ ，计算机操作系统会启动一个进程，并要求运算器派过来一个 CPU 来完成任

务；当我们运行一个程序来打开文件，操作系统会启动存储器的功能将硬盘中的文件数据导入到内存中。

一个 CPU 在某一时刻只能做一项任务，即在一个进程（或线程）中工作，当它闲置时，会被系统派到其它进程中。单核计算机也可以实现多进程，原理是第 1 秒的时间段内运行 A 进程，其它进程等待；第 2 秒的时间段内运行 B 进程，其它进程等待。。。第 5 秒的时间段内又运行 A 进程，往复循环。当然实际上 CPU 在各个进程间的切换是极快的，在毫秒（千分之一）、微秒（百万分之一）级，以至于我们看起来这些程序就像在同时运行。现代的计算机都是多核配置，四核八核等，但计算机启动的瞬间，往往就有几十上百个进程在运行了，所以进程切换是一定会发生的，CPU 在忙不迭停地到处赶场。注意，什么时候进行进程切换是由操作系统决定的，无法人为干预。

下面我们来编写一个单进程程序，并打印程序运行时间，将以下代码写入 `single_process.py`：

```
import time

def io_task():
    # time.sleep 强行挂起当前进程 1 秒钟
    # 所谓“挂起”，就是进程停滞，后续代码无法运行，CPU 无法进行工作的状态
    # 相当于进行了一个耗时 1 秒钟的 IO 操作
    # 上文提到过的，IO 操作可能会比较耗时，但它不会占用 CPU
    time.sleep(1)

def main():
    start_time = time.time()
    # 循环 IO 操作 5 次
    for i in range(5):
        io_task()
    end_time = time.time()
    # 打印运行耗时，保留 2 位小数
    print('程序运行耗时: {:.2f}s'.format(end_time-start_time))

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

执行程序，查看运行时间：

```
shiyancelou:~/ $ python3 single_process.py
程序运行耗时: 5.01s
```

现在我们来改写上文中的单进程代码，引入 `Process` 类并修改 `main()` 函数来实现多进程，将以下代码写入 `multi_process.py` 中：

🔗 楼之Python实战第10期 (/courses/1190)

引入 Process 类

```
from multiprocessing import Process
```

```
def io_task():
    # time.sleep 强行挂起当前进程 1 秒钟
    # 所谓“挂起”，就是进程停滞，后续代码无法运行，CPU 无法进行工作的状态
    # 相当于进行了一个耗时 1 秒钟的 IO 操作
    # 上文提到过的，IO 操作可能会比较耗时，但它不会占用 CPU
    time.sleep(1)

def main():
    start_time = time.time()
    ...

    for i in range(5):
        io_task()
    ...

    # 创建一个列表存放子任务备用
    process_list = []
    # 创建 5 个多进程任务并加入到任务列表中
    for i in range(5):
        process_list.append(Process(target=io_task))
    # 启动所有子任务
    # 此时操作系统会创建 5 个子进程并派出闲置的 CPU 来运行 io_task() 函数
    for process in process_list:
        process.start()
    # join 方法将主进程挂起并释放 CPU，在一旁候着，直到所有子进程运行完毕
    for process in process_list:
        process.join()
    # 子进程运行完毕，以下代码运行在主进程中
    end_time = time.time()
    print('程序运行耗时: {:.2f}s'.format(end_time-start_time))

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

运行程序，查看大致运行时间：

```
shiyancelou:~/ $ python3 multi_process.py
程序运行耗时: 1.04s
```

可以看到多进程程序的运行时间明显少于单进程程序，看起来我们成功优化了代码。但是多进程的运行时间难道不应该是单进程的五分之一（1.01s）左右吗，怎么看起来稍稍多了些？我们来对这两个程序进行一下简单的分析：单进程程序中的 IO 任务 `io_task()` 是依次循环运行的，即全部任务的运行时间为每个任务的运行时间乘以 5；多进程代码中，5 个任务分别在各自的进程中被不同的 CPU 处理，而且主进程中创建了列表并执行了两个列表的循环，再加上创建子进程以及进程切换耗掉的时间，结果就会比理论上多一些。

一个查看系统 CPU 核数的小技巧：

```
shyanlou@py:~$ python3
Python 3.5.2 (default, Jul 17 2016, 00:00:00)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from multiprocessing import cpu_count
>>> cpu_count()
4
```

进程间通信

进程有自己独立的运行空间，它们之间所有的变量、数据是隔绝的，A 进程中的变量 test 与 B 进程的变量没有任何关系，这就意味要使用一些特殊的手段才能实现它们之间的数据交换。multiprocessing 模块提供了管道 Pipe 和队列 Queue 两种方式。

这两种方式的基本概念和实现机制可以通过后续的实验代码进行学习。

Pipe

如果把两个进程想象成两个密封的箱子，那 Pipe 就像是连接两个箱子的一个管道，借助它可以实现两个箱子之间简单的数据交换。看一下它的使用方法：

```
# 首先引入 Pipe 类
from multiprocessing import Pipe
# 创建一个管道，注意 Pipe 的实例是一个元组，里面有两个连接器，可以把它们想象成通讯员
conn1, conn2 = Pipe()
```

默认情况下，打开的管道是全双工的，也就是说你可以在任何一端读写数据，写入数据使用 send 方法，读取数据使用 recv 方法。下面看一个例子：

📌 楼+之Python实战第10期 (/courses/1190)

```
from multiprocessing import Pipe

conn1, conn2 = Pipe()

def send():
    data = 'hello shiyanlou'
    conn1.send(data)
    print('Send Data: {}'.format(data))

def recv():
    data = conn2.recv()
    print('Receive Data: {}'.format(data))

def main():
    Process(target=send).start()
    Process(target=recv).start()

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

这个程序启动了两个进程，第一个进程在 send 函数中向 pipe 管道写入 Hello shiyanlou，第二个进程在 recv 函数中从管道中读取数据并打印。将以上代码写入 /home/shiyanlou/mult_pipe.py 文件，程序运行结果：

```
$ python3 mult_pipe.py
Send Data: hello shiyanlou
Receive Data: hello shiyanlou
```

- 拓展阅读：pipes — Interface to shell pipelines (英文)
(<https://docs.python.org/3.5/library/pipes.html?highlight=pipe#module-pipes>)

Queue

除了 Pipe 外，multiprocessing 模块还实现了一个可以在多进程下使用的队列结构 Queue，改写上面的程序：

📌 楼+之Python实战第10期 (/courses/1190)

```
# 创建队列实例
queue = Queue()

# 该函数的任务是向队列中发送数据
def f1(q):
    i = 'Hello shiyanlou'
    q.put(i)
    print('Send Data: {}'.format(i))

# 该函数的任务是从队列中获取数据
def f2(q):
    data = q.get()
    print('Receive Data: {}'.format(data))

def main():
    Process(target=f1, args=(queue,)).start()
    Process(target=f2, args=(queue,)).start()

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

将以上代码写入 `/home/shiyanlou/multi_queue.py` 文件中，运行程序结果如下：

```
$ python3 multi_queue.py
Send Data: Hello shiyanlou
Receive Data: Hello shiyanlou
```

Queue 可以在初始化时指定一个最大容量：

```
queue = Queue(maxsize=10)
```

另外通过 `Queue.empty()` 方法可以判断队列中是否为空，是否还有数据可以读取，如果返回为 `True` 表示已经没有数据了。

进程同步

看一个例子，有一个计数器 `i`，初始值为 0，现在创建并运行 10 个进程，每个进程对 `i` 进行 50 次加 1 操作，也就是说理论上最后的结果是 500。

`Value` 是一个方法，可以实现数据在多进程之间共享，该方法的返回值即 `Value` 对象可以看做是一个全局共享的变量。使用 `help(Value)` 查看参数，第一个参数是共享的数据类型，`'i'` 指的是 `ctypes.c_int` 就是整数，第二个参数是 `Value` 对象的值，在进程中可以通过 `Value` 对象的 `value` 属性获取。

将以下代码写入 `multi_value.py` 文件中：

实验楼之Python实战第10期 (/courses/1190)

```
from multiprocessing import Process, Value
```

```
# 该函数运行在子进程中，参数 val 是一个 Value 对象，是全局变量
def func(val):
    # 将 val 这个全局变量的值进行 50 次 +1 操作
    for i in range(50):
        time.sleep(0.01)
        val.value += 1

if __name__ == '__main__':
    # 多进程无法使用全局变量，multiprocessing 提供的 Value 是一个代理器，可以实现在多进程中共享这个变量
    # val 是一个 Value 对象，它是全局变量，数据类型为 int，初始值为 0
    val = Value('i', 0)
    # 创建 10 个任务，备用
    procs = [Process(target=func, args=(val,)) for i in range(10)]
    # 启动 10 个子进程来运行 procs 中的任务，对 Value 对象进行 +1 操作
    for p in procs:
        p.start()
    # join 方法使主进程挂起，直至所有子进程运行完毕
    for p in procs:
        p.join()

    print(val.value)
```

由于多进程的推进顺序是无法预测的，有可能出现几个进程同时对 i 进行加 1 操作，但由于 CPU 和内存的读写机制造成只有一个进程的加 1 的操作会被记录下来，这就导致最后的结果应该是小于 500 的，下面是程序运行 5 次的结果：

```
250
269
231
201
213
```

正确的做法是每次进行加 1 操作时候，为 i 加一把锁，也就是说当前的进程在操作 i 时，其它进程不能操作它。multiprocessing 模块的 Lock 类封装的锁操作，使用 acquire() 方法获取锁，release() 方法释放锁。下面是修正后的代码：

🔗 [Python 实战第10期 \(/courses/1190\)](#)

```
from multiprocessing import Process, Value, Lock
```

```
def func(val, lock):
    for i in range(50):
        time.sleep(0.01)
        # with lock 语句是对下面语句的简写:
        ...

        lock.acquire()
        val.value += 1
        lock.release()
        ...

    # 为 val 变量加锁, 结果就是任何时刻只有一个进程可以获得 lock 锁
    # 自然 val 的值就不会同时被多个进程改变
    with lock:
        val.value += 1

if __name__ == '__main__':

    v = Value('i', 0)
    # 初始化锁
    # Lock 和 Value 一样, 是一个函数或者叫方法, Lock 的返回值就是一把全局锁
    # 注意这把全局锁的使用范围就是当前主进程, 也就是在运行当前这个 Python 文件过程中有效
    lock = Lock()
    procs = [Process(target=func, args=(v, lock)) for i in range(10)]

    for p in procs:
        p.start()
    for p in procs:
        p.join()

    print(v.value)
```

现在每次运行的结果都是 500。加锁的作用是保证变量的修改是稳定可预期的, 副作用就是多个进程无法同时运行, 因为某个时刻只有一个进程可以获得锁, 结果程序运行耗时就会增加。由于我们的代码计算量较小, 看不出明显的运行耗时区别。

进程池

进程池 Pool 可以批量创建子进程, 进程池内维持一个固定的进程数量, 当有任务到来时, 就去池子中取一个进程处理任务, 处理完后, 进程被返回进程池。如果一个任务到来而池子中的进程都被占用了, 那么任务就需要等待某一个进程执行完。

下面举例说明进程池的原理, 将以下代码写入 multi_pool.py 文件中:

实验楼之Python实战第10期 (/courses/1190)

```
from multiprocessing import Pool
```

该函数将运行在子进程中

```
def task(name):
    # 打印进程 ID
    print('任务{}启动运行, 进程ID: {}'.format(name, os.getpid()))
    start = time.time()
    # 假设这里有一个比较耗时的 IO 操作
    # random.random() 的值是一个 0 到 1 区间内的随机小数
    time.sleep(random.random() * 3)
    end = time.time()
    print('任务{}结束运行, 耗时: {:.2f}s'.format(name, (end-start)))


if __name__ == '__main__':
    print('当前为主进程, 进程ID: {}'.format(os.getpid()))
    print('-----')
    # 创建进程池, 池子里面有 4 个进程可用
    p = Pool(4)
    # 将 5 个任务加入到进程池
    for i in range(1, 6):
        # apply_async 方法异步启动进程池
        # 即池子内的进程是随机接收任务的, 直到所有任务完成
        p.apply_async(task, args=(i,))
    # close 方法关闭进程池
    p.close()
    # join 方法挂起主进程, 直到进程池内任务全部完成
    print('开始运行子进程...')
    p.join()
    print('-----')
    print('所有子进程运行完毕, 当前为主进程, 进程ID: {}'.format(os.getpid()))
```

在终端执行上述代码：

```
$ python3 multi_pool.py
当前为主进程, 进程ID: 3232
-----
开始运行子进程...
任务1启动运行, 进程ID: 3234
任务2启动运行, 进程ID: 3233
任务3启动运行, 进程ID: 3235
任务4启动运行, 进程ID: 3236
任务3结束运行, 耗时: 1.11s
任务5启动运行, 进程ID: 3235
任务4结束运行, 耗时: 1.34s
任务5结束运行, 耗时: 0.70s
任务2结束运行, 耗时: 2.21s
任务1结束运行, 耗时: 2.81s
-----
所有子进程运行完毕, 当前为主进程, 进程ID: 3232
```

我们来分析一下程序的运行情况：

1、任务 1 2 3 4 是立即执行的，它们占据了进程池的全部 4 个进程。

2、任务 3 率先结束，耗时 1.11 秒，空出占用的 3235 进程，马上任务 5 开始在 3235 进程中运行。楼+之Python实战第10期 (/courses/1190)

3、等其余 4 个任务都完成了，任务 1 才结束运行，耗时 2.81 秒，甚至比任务 3 和 5 加起来的时间还要长，这就是 random.random 生成的随机小数的缘故。

4、每次运行的结果都是不同的，但流程不会变。

多线程

前面我们讲到多任务可以由多进程来完成，而在一些时候能够明显地提高代码的运行效率，多线程也可以实现类似的功能。一个进程中可以有一个或多个线程，进程内的线程共享内存空间。线程是程序执行的最小单元，一个进程中至少有一个线程（主线程）。因为解释器的设计原因，多线程在 Python 中使用得较少。

Python 的 threading 模块提供了对多线程的支持，接口和 multiprocessing 提供的多进程接口非常类似，下面我们用多线程改写一开始的多进程例子。

将以下代码写入 multi_thread.py 文件中：

```
import threading

def hello(name):
    # get_ident() 函数获取当前线程 ID
    print('当前为子线程，线程ID: {}'.format(threading.get_ident()))
    print('Hello ' + name)

def main():
    print('当前为主线程，线程ID: {}'.format(threading.get_ident()))
    print('-----')
    # 初始化一个子线程，参数传递和使用 Process 一样
    t = threading.Thread(target=hello, args=('shiyanlou',))
    # 启动线程和等待线程结束，和 Process 的接口一样
    t.start()
    t.join()
    print('-----')
    print('当前为主线程，线程ID: {}'.format(threading.get_ident()))

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

终端运行 multi_thread.py 文件：

🔗 [Python 实战第10期 \(/courses/1190\)](#)
当前为主线程，线程ID: 4583290304

当前为子线程，线程ID: 123145514283008

Hello shiyanlou

当前为主线程，线程ID: 4583290304

总结

本节实验中我们学习了 Python 的多进程和多线程编程的知识。学习了使用 multiprocessing 及 threading 模块开发多进程及多线程程序，本节实验中的知识点包括：

- 多进程：如何开发一个多进程的 Python 程序
- 进程通信：如何让多进程程序进程间使用 Pipe 和 Queue 通信
- 进程同步：如何加锁使进程处理同步
- 进程池：建立进程池的方法
- 多线程：如何开发一个多线程的 Python 程序

拓展阅读

想要加深对 Python 多线程、多进程的理解？继续阅读：

- 《Python 官方文档（中文）- 多线程》
(<http://www.pythondoc.com/pythontutorial3/stdlib2.html#tut-multi-threading>)
- 《深入分析 Python 多线程》
(<https://thief.one/2017/02/17/Python%E5%A4%9A%E7%BA%BF%E7%A8%8B%E9%B8%A1%E5%B9%B4%E4%B8%8D%E9%B8%A1%E8%82%8B/>)
- 《Python 多进程基础》(<https://thief.one/2016/11/23/Python-multiprocessing/>)

**本课程内容，由作者授权实验楼发布，未经允许，禁止转载、下载及非法传播。*

上一节：[Python 高级特性 \(/courses/1190/labs/8526/document\)](#)

下一节：[\[选学\] 挑战：多进程工资计算器 \(/courses/1190/labs/8528/document\)](#)