

Python 深度解析之

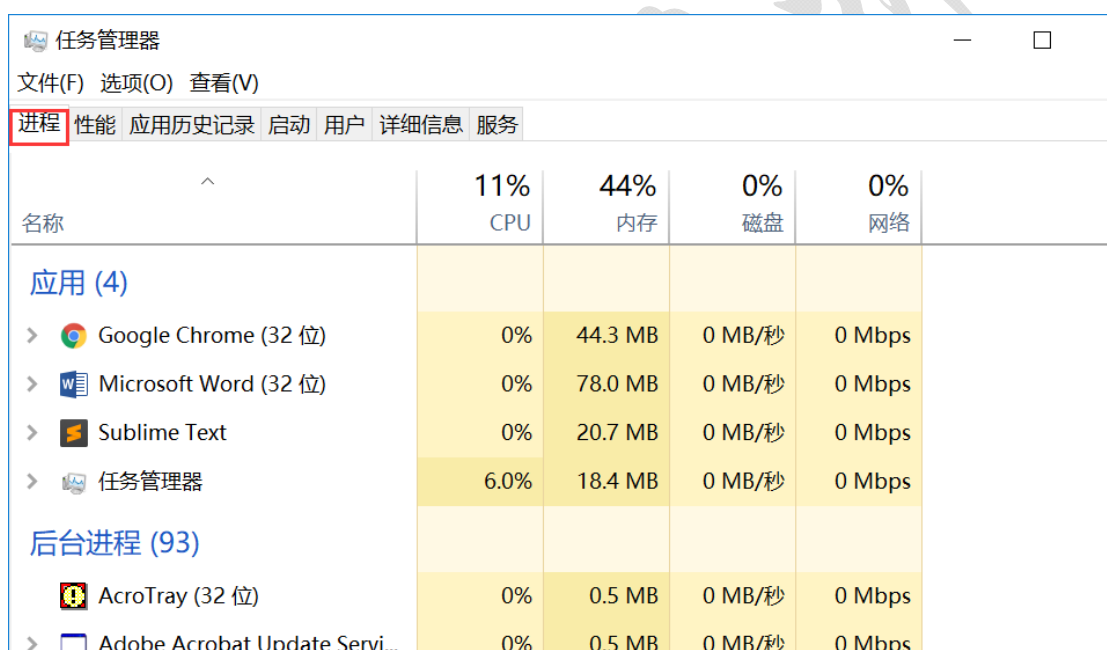
多线程与多进程

前言：记得小学的时候，语文老师经常让我们抄课文，一遍一遍重复，那时候我经常一只手拿着两支笔，一下写两行文字，因为总觉得一遍抄完才能抄另一遍实在是太不效率。在如今的社会，什么事情都要要求效率，程序也不例外。前边我们学习了爬虫的用法，比如让爬 20 万张图片，你会选择一张一张的爬，还是选择让 20 个照片同时爬呢？答案不言而喻，那么 python 如何同时让多个函数一起工作呢，这就是我们今天要讲的多线程与多进程。

By 浪ふ沕沙

A. 多线程与多进程的简介

什么是进程？对电脑稍微熟悉一点的都知道 ctrl+alt+delete 能打开任务管理器，里边有个进程，找不到这三个键的右键任务栏也能打开，我们看一下。



名称	11% CPU	44% 内存	0% 磁盘	0% 网络
应用 (4)				
Google Chrome (32 位)	0%	44.3 MB	0 MB/秒	0 Mbps
Microsoft Word (32 位)	0%	78.0 MB	0 MB/秒	0 Mbps
Sublime Text	0%	20.7 MB	0 MB/秒	0 Mbps
任务管理器	6.0%	18.4 MB	0 MB/秒	0 Mbps
后台进程 (93)				
AcroTray (32 位)	0%	0.5 MB	0 MB/秒	0 Mbps
Adobe Acrobat Update Servi...	0%	0.5 MB	0 MB/秒	0 Mbps

没错，我们在电脑上启动的每一个软件都是进程，当然电脑的操作系统也会在后台开启的有系统进程，不然我们的电脑就运行不起来了是不是？

那么什么是进程呢？准确来讲，每一个线程中都至少会有一个线程的启动，这个默认启动的进程，我们称之为主线程 MainThread。为了方便大家理解，我举个例子，我们小时候玩的魂斗罗都还记得吧，当我们打开这个游戏的时候，是启动了一个进程，我们可以在任务管理器里看见，在游戏中，我们操作的两个战士是两个线程，里边的子弹又会是另外一个进程，这些线程单独执行，互不影响，或许会调用全局变量，产生联系。再不懂的可以想象一下你正在用网易音乐听歌，然后你觉得这首歌唱的很带感，于是你选择了下载，但是下载并不影响你继续听这首歌，那么下载歌曲和播放歌曲就是两个线程。

B. 多线程

在 python 中有两个标准库，_thread 和 threading，_thread 被称为上古时代的产物，我们已经不怎么使用了，而 threading 是一个高级库，对_thread 进行了很好的封装，我们平时一般只使用 threading 就可以了。我们来导入 threading，按住 alt+鼠标左键打开源码，看一下。启动多线程都是用使用

threading.Thread() 这个类来实现, 我们看下这个类就可以了。

```

738 ▼ class Thread:
739     """A class that represents a thread of control.
740
741     This class can be safely subclassed in a limited fashion. There are two ways
742     to specify the activity: by passing a callable object to the constructor, or
743     by overriding the run() method in a subclass.
744
745     """
746
747     _initialized = False
748     # Need to store a reference to sys.exc_info for printing
749     # out exceptions when a thread tries to use a global var. during interp.
750     # shutdown and thus raises an exception about trying to perform some
751     # operation on/with a NoneType
752     _exc_info = _sys.exc_info
753     # Keep sys.exc_clear too to clear the exception just before
754     # allowing .join() to return.
755     #XXX _exc_clear = _sys.exc_clear
756
757     def __init__(self, group=None, target=None, name=None,
758                  args=(), kwargs=None, *, daemon=None):

```

我们看下启动需要的参数, target 是我们要多次执行的函数名字, name 是我们要启动的线程的名字, args 是我们要给执行函数传递的参数, 注意这里是一个元组, 如果只有一个参数必须要(i,)这么写, 少了逗号会报错, daemon 参数表示主线程是否为守护进程, 参数是一个布尔值。另外的几个参数, 不太常用, 我也不太懂, 有兴趣的同学可以自行百度, 如果后期需要我会再及时补充。线程的启动用 start 方法。我们来一段代码。

```

× 类似爱情.py
× 多线程.py — C
× threading.py
• 多线程.py — D
1 # -*- coding:utf-8 -*-
2 import threading
3 import time
4
5 ▼ def test_fun(i):
6     y = i**2
7     time.sleep(1)
8     print('thread name is %s,result is %s' % (threading.current_thread().name,y))
9
10 ▼ for i in range(6):
11     #print('thread name is %s' % threading.current_thread().name)
12     t = threading.Thread(target=test_fun,args=(i,),name='第'+str(i)+'个线程')
13     t.start()

```

```

thread name is 第0个线程,result is 0
thread name is 第3个线程,result is 9
thread name is 第1个线程,result is 1
thread name is 第2个线程,result is 4
thread name is 第5个线程,result is 25
thread name is 第4个线程,result is 16
[Finished in 1.1s]

```

采用多线程的执行, 完成耗时 1.2 秒。我们看看不采用多线程。

```

× 类似爱情.py 1 # -*- coding:utf-8 -*-
× 多线程.py — C 2 import threading
× threading.py 3 import time
• 多线程.py — D 4
5 def test_fun(i):
6     y = i**2
7     time.sleep(1)
8     print('thread name is %s,result is %s' % (threading.current_thread().name,y))
9     for i in range(6):
10        test_fun(i)

thread name is MainThread,result is 0
thread name is MainThread,result is 1
thread name is MainThread,result is 4
thread name is MainThread,result is 9
thread name is MainThread,result is 16
thread name is MainThread,result is 25
[Finished in 6.1s]

```

按照我们循环的写法, 耗时 6.1 秒, 这个效率不用我多说了吧。

```

× 类似爱情.py 1 # -*- coding:utf-8 -*-
× 多线程.py — C 2 import threading
× threading.py 3 import time
• 多线程.py — D 4
5 def test_fun(i):
6     y = i**2
7     time.sleep(1)
8     print('thread name is %s,result is %s' % (threading.current_thread().name,y))
9     for i in range(6):
10        t = threading.Thread(target=test_fun,args=(i,),name='第'+str(i)+'个线程')
11        t.start()
12    print('总共%s个线程'% threading.activeCount())

总共7个线程

```

threading 的 activeCount() 方法获取当前总共线程的个数, 包含主线程, 而主线程永远只有一个, 如果我们需要计算当前子线程的个数只需要-1 就可以了。

threading.enumerate() 方法返回当前运行中的 Thread 对象列表。

```

× 类似爱情.py 1 # -*- coding:utf-8 -*-
× 多线程.py — C 2 import threading
× threading.py 3 import time
• 多线程.py — D 4
5 def test_fun(i):
6     y = i**2
7     time.sleep(1)
8     print('thread name is %s,result is %s' % (threading.current_thread().name,y))
9     for i in range(6):
10        t = threading.Thread(target=test_fun,args=(i,),name='第'+str(i)+'个线程')
11        t.start()
12    for item in threading.enumerate():
13        print(item)

<_MainThread(MainThread, started 7952)>
<Thread(第0个线程, started 2772)>
<Thread(第1个线程, started 12112)>
<Thread(第2个线程, started 9160)>
<Thread(第3个线程, started 8872)>
<Thread(第4个线程, started 15256)>
<Thread(第5个线程, started 9936)>
thread name is 第0个线程,result is 0
thread name is 第2个线程,result is 4
thread name is 第1个线程,result is 1
thread name is 第4个线程,result is 16
thread name is 第3个线程,result is 9
thread name is 第5个线程,result is 25
[Finished in 1.1s]

```

如果想要程序成为主线程守护线程可以设置 daemon 参数, 也可以用 setdaemon() 方法, 如果你能看懂源码, 你会发现 setdaemon 接收的值就是 Thread 类中的 daemon 的参数。我们来看看 daemon 的作用。

```

x 类似爱情.py
x 多线程.py — C
x threading.py
• 多线程.py — D
1  # _*_ coding:utf-8 _*_
2  import threading
3  import time
4
5  def test_fun(i):
6      y = i**2
7      time.sleep(1)
8      print('thread name is %s,result is %s' % (threading.current_thread().name,y))
9  for i in range(6):
10     t = threading.Thread(target=test_fun,args=(i,),name='第'+str(i)+'个线程')
11     t.setDaemon(True)
12     t.start()
13
[Finished in 0.1s]

```

这里为什么没有打印子线程呢？这是因为我们设置了主线程为守护线程，也即主线程单独执行，并不会去关心子线程是否执行完毕，主线程执行完毕之后程序就直接退出了，而子线程还没有执行完就一并退出了。Setdaemon 的反方法就是 join()。join() 方法被调用之后，被作用的线程执行完毕之后就会判断子线程是否执行完毕，如果子线程没有执行完，就会等待子线程执行完毕之后，一并退出，我们上图看一下。唯一不同的就是 setdaemon(False) 只是对所有子进程同时起作用，而 join() 是针对每一个进程单独起作用。

```

x 多线程.py — C
x threading.py
• 多线程.py — D
5  def test_fun(i):
6      y = i**2
7      time.sleep(1)
8      print('thread name is %s,result is %s' % (threading.current_thread().name,y))
9  for i in range(6):
10     t = threading.Thread(target=test_fun,args=(i,),name='第'+str(i)+'个线程')
11     #t.setDaemon(False)
12     t.start()
13     t.join()
14     print('hello')
15
thread name is 第0个线程,result is 0
hello
thread name is 第1个线程,result is 1
hello
thread name is 第2个线程,result is 4
hello
thread name is 第3个线程,result is 9
hello
thread name is 第4个线程,result is 16
hello
thread name is 第5个线程,result is 25
hello
[Finished in 6.2s]

```

解释如下：第一个子进程执行的时候，由于设置了的 join()，所以第一个子进程没执行完的时候，并不会去执行下边的 print()，直到子进程结束，才会执行 print('hello')，进而往下交替执行。这样的不好的地方就是因为子线程没有同时执行，失去了并发的意义，所以执行时间达到了 6.2 秒。如果觉得还是不理解，大家可以看下别人的博客，我提供一篇我以前学习的时候看过的别的大神的大作，不过要求大家对类的继承非常熟悉：
<http://blog.csdn.net/zhangzheng0413/article/details/41728869/>

```

x 多线程.py — C
x threading.py
• 多线程.py — D
5 def test_fun(i):
6     y = i**2
7     time.sleep(1)
8     print('thread name is %s,result is %s' % (threading.current_thread().name,y))
9 for i in range(6):
10    t = threading.Thread(target=test_fun,args=(i,),name='第'+str(i)+'个线程')
11    t.setDaemon(False)
12    t.start()
13    #t.join()
14    print('hello')
15

hello
hello
hello
hello
hello
hello
thread name is 第0个线程,result is 0
thread name is 第4个线程,result is 16
thread name is 第5个线程,result is 25
thread name is 第2个线程,result is 4
thread name is 第1个线程,result is 1
thread name is 第3个线程,result is 9
[Finished in 1.1s]

```

由于设置了 `setdaemon(False)` 主线程执行完毕后, 会等待子线程, 所以才会出现了打印出了所有的 `hello`, 才返回了子线程的结果, 而后一起退出程序。这里需要大家细细体会, 总的来说, `setdaemon` 是针对所有的子线程, 而 `join` 需要对每一个作用的线程单独负责, 分工不同。

C. 多线程的线程锁

多线程执行的时候, 所有变量都会共享, 每个线程的执行都会对变量就行更改, 这么做最大的问题就是会导致变量的更迭, 获取错误的信息。我们先说一下 `python` 中的公共变量, 用 `global` 来定义, 我们平时用到的都是局部变量, 只会在函数内部调用。直接上代码。

```

• 多线程.py — D
15 a = 5
16 lock = threading.Lock()
17 def get_num(n):
18     global a
19     for i in range(100000):
20         a +=n
21         a -=n
22     print('a is %s' % a)
23 t1 = threading.Thread(target=get_num,args=(5,))
24 t2 = threading.Thread(target=get_num,args=(6,))
25 t1.start()
26 t2.start()
27 print('a is %s' %a)

a is 11
a is 0
a is 5
[Finished in 0.2s]

```

大家可以看到, 没有加线程锁, 因为在循环次数比较大, `t1` 和 `t2` 交替执行的时候, 在获取 `a` 的时候就会产生获取异常的 `a` 的值, 当然循环次数少的时候, 并没有什么不正常。为了防止线程在同时获取 `a`, 改变 `a` 的值, 从而去影响其它的线程, 我们可以使用 `threading.Lock()`。`threading.Lock()` 中有两个方法, `acquire()` 获取线程锁, `release()` 释放线程锁。


```

x 类似爱情.py
x 多线程.py — C
x threading.py
  • 多线程.py — D
15 a = 5
16 lock = threading.Lock()
17 def get_num(n):
18     global a
19     try:
20         lock.acquire()
21         for i in range(100000):
22             a += n
23             a -= n
24         finally:
25             lock.release()
26     print('a is %s' % a)
27 t1 = threading.Thread(target=get_num, args=(5,))
28 t2 = threading.Thread(target=get_num, args=(6,))
29 t1.start()
30 t2.start()
31 print('a is %s' % a)

a is 5
a is 5
a is 5
[Finished in 0.2s]

```

使用线程锁的时候要注意一定要加上释放线程锁, 不然别的线程拿到不到这个锁, 就会导致线程永远等待下去。所以可以使用 try---finally, 释放线程锁, 让别的线程有机会获取。

D. 多线程的并发控制

回到回头, 我们说爬图片的时候一张一张的爬太慢了, 效率起见, 我们可以同时 20 张同时爬, 也就是我们可以同时开启 20 个线程。问题来了, 我们如何控制这个线程的个数呢写 20 个 threading.Thread() 吗? 重复的动作往往都是编程的活, 因此 python 也不例外, 在处理这个重复的线程的时候, 我们可以使用线程池来控制。当然你说用循环可以吗? 可以, 但是你却没法控制线程的个数。从 Python3.2 开始, 标准库为我们提供了 concurrent.futures 模块, 它提供了 ThreadPoolExecutor 和 ProcessPoolExecutor 两个类, 实现了对 threading 和 multiprocessing 的更高级的抽象, 对编写线程池/进程池提供了直接的支持。concurrent.futures 基础模块是 executor 和 future。我们看看源码, 在 future 中有三个文件, base, thread, process。base 是基础类, 而 thread 和 process 分别为对进程和线程的控制, 都继承至 base。

« program files > Python > Lib > concurrent > futures 搜索"futures"

名称	修改日期	类型
__pycache__	2017/3/11 12:05	文件夹
__init__.py	2016/5/16 16:43	PY 文件
_base.py	2016/5/16 16:43	PY 文件
process.py	2016/9/5 19:43	PY 文件
thread.py	2016/9/11 22:51	PY 文件

我们今天的 ThreadPoolExecutor 类, 继承来至与 base 中 Executor。看看源码中 ThreadPoolExecutor。

```

83 class ThreadPoolExecutor(_base.Executor):
84     def __init__(self, max_workers=None, thread_name_prefix=''):
85         """Initializes a new ThreadPoolExecutor instance.
86
87         Args:
88             max_workers: The maximum number of threads that can be used to
89                 execute the given calls.
90             thread_name_prefix: An optional name prefix to give our threads.
91         """
92         if max_workers is None:
93             # Use this number because ThreadPoolExecutor is often
94             # used to overlap I/O instead of CPU work.
95             max_workers = (os.cpu_count() or 1) * 5
96         if max_workers <= 0:
97             raise ValueError("max_workers must be greater than 0")
98
99         self._max_workers = max_workers
100        self._work_queue = queue.Queue()
101        self._threads = set()
102        self._shutdown = False
103        self._shutdown_lock = threading.Lock()
104        self._thread_name_prefix = thread_name_prefix

```

我们看见类的实例需要一个线程池数, 如果不给的话, 程序会计算出 cpu 的核心数, 然后*5 就是自动开启的线程池数。

```

509 class Executor(object):
510     """This is an abstract base class for concrete asynchronous
511
512     def submit(self, fn, *args, **kwargs):
513         """Submits a callable to be executed with the given argu
514
515         Schedules the callable to be executed as fn(*args, **kwa
516         a Future instance representing the execution of the call
517
518         Returns:
519             A Future representing the given call.
520         """
521         raise NotImplementedError()
522
523     def map(self, fn, *iterables, timeout=None, chunksize=1):

```

```

45         if timeout is not None:
46             end_time = timeout + time.time()
47
48         fs = [self.submit(fn, *args) for args in zip(*iterables)]
49
50         # Yield must be hidden in closure so that the futures are submit
51         # before the first iterator value is required.
52         def result_iterator():
53             try:
54                 for future in fs:
55                     if timeout is None:
56                         yield future.result()
57                     else:
58                         yield future.result(end_time - time.time())
59             finally:
60                 for future in fs:
61                     future.cancel()
62         return result_iterator()

```

这是父类 Executor, 我们可以见看已经定义了 map() 的方法, 跟我们平时用到的 map() 没有什么区别, fn 代表一个函数, 而 iterables 代表一个可迭代对象, 返回一个序列。大家在日常的学习中如果有什么不懂的, 也可以找到源码自行观看。知道了这两个方法的使用方法, 我们现在就可以写我们自己的线程池了。

```

33 def get_square(i):
34     y = i**2
35     print('the thread is %s' % threading.current_thread().name)
36     return y
37
38 def main():
39     f_list = [i for i in range(1000)]#构建一个需要执行的参数
40     workers = 20 #设置线程池为20
41     with futures.ThreadPoolExecutor(max_workers=workers) as executor:
42         res = executor.map(get_square,f_list)
43     return res
44 if __name__ == '__main__':
45     a =main()
46     print(type(a))
47     print(list(a))

```

```

the thread is <concurrent.futures.thread.ThreadPoolExecutor object at 0x00BB6290>_2
the thread is <concurrent.futures.thread.ThreadPoolExecutor object at 0x00BB6290>_2
the thread is <concurrent.futures.thread.ThreadPoolExecutor object at 0x00BB6290>_2
<class 'generator'>
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784, 841, 900, 961, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600, 1681, 1764, 1849, 1936, 2025, 2116, 2209, 2304, 2401, 2500, 2601, 2704, 2809, 2916, 3025, 3136, 3249, 3364, 3481, 3600, 3721, 3844, 3969, 4096, 4225, 4356, 4489, 4624, 4761, 4900, 5041, 5184, 5329, 5476, 5625, 5776, 5929, 6084, 6241, 6400, 6561, 6724, 6889, 7056, 7225, 7396, 7569, 7744, 7921, 8100, 8281, 8464, 8649, 8836, 9025, 9216, 9409, 9604, 9801, 10000]

```

这里我们的作用函数实现的效果就是对传递的参数进行平方, 主函数构建线程池为 20。我们能从打印的名字中看见, 每个线程的名字, 由于条数过多所以没有全部截图。而 concurrent.futures.ThreadPoolExecutor.map() 返回的是一个生成器。我们可以用 type() 函数看到, 如果想要一个一个取得值可以用 for 来迭代, 想要看到全部, 可以用 list 转换类型。

E. 多进程

多进程与多线程类似, 在 python 中提供了 multiprocessing, 我们可以用 from multiprocessing import Process 来导入。写一段试试看。

```

51 def get_name(i):
52     print('the process is %s(%s)' % (os.getpid(),i))
53 if __name__ == '__main__':
54     p1 = Process(target=get_name,args=(1,))
55     p2 = Process(target=get_name,args=(2,))
56     p1.start()
57     p2.start()
58     p1.join()
59     p2.join()

```

```

the process is 11740(2)
the process is 13648(1)
[Finished in 0.3s]

```

Process 的参数与多线程的参数一样, os 是 python 的一个自带库, 封装了常见的系统调用。getpid() 获取当前子进程的 id, getppid() 获取当前主进程的 id。start() 和 join() 方法与多线程中的一样, 就不在赘述了。

F. 多进程的进程控制

多进程的控制, 我们可以用 multiprocessing 下的 pool 来控制, 为了方便理解,

老规矩, 我们来线看源码。

```

138 class Pool(object):
139     """
140     Class which supports an async version of applying functions to arguments.
141     """
142     _wrap_exception = True
143
144     def Process(self, *args, **kwargs):
145         return self.ctx.Process(*args, **kwargs)
146
147     def __init__(self, processes=None, initializer=None, initargs=(),
148                 maxtasksperchild=None, context=None):
149         self.ctx = context or get_context()
150         self._setup_queues()
151         self._taskqueue = queue.Queue()
152         self._cache = {}
153         self._state = RUN
154         self._maxtasksperchild = maxtasksperchild
155         self._initializer = initializer
156         self._initargs = initargs
157
158         if processes is None:
159             processes = os.cpu_count() or 1
160         if processes < 1:
161             raise ValueError("Number of processes must be at least 1")

```

很明显, `processes` 参数代表了我们要组件的进程池数, 默认的是 `cup` 的个数。而多进程的启动方法则是 `Pool` 中的 `apply_async()`。

```

def apply_async(self, func, args=(), kwargs={}, callback=None,
               error_callback=None):
    """
    Asynchronous version of `apply()` method.
    """
    if self._state != RUN:
        raise ValueError("Pool not running")
    result = ApplyResult(self._cache, callback, error_callback)
    self._taskqueue.put([(result._job, None, func, args, kwargs)], None)
    return result

```

`func` 显而易见是我们要启动的函数名, `args()` 和 `kwargs {}` 为函数的可变参数和关键字参数。至于 `callback` 和 `error_callback` 那就是回滚和错误回滚的参数, 我们暂时不管。写一个试试。

```

52 def get_name(i):
53     print('the process is %s(%s)' % (os.getpid(), i))
54     print('myparent is %s(%s)' % (os.getppid(), i))
55
56 if __name__ == '__main__':
57     p = Pool(10)
58     for i in range(10):
59         p.apply_async(func=get_name, args=(i,))
60     p.close()
61     p.join()

```

```

the process is 12560(0)
myparent is 4216(0)
the process is 12560(2)
myparent is 4216(2)
the process is 12560(4)
myparent is 4216(4)
the process is 12560(6)
myparent is 4216(6)

```

尤其要说一下 `apply_async()` 的函数参数是 `func`, 不要写成 `target`, 否则要

报错, 从打印结果我们可以到, 所有的子进程都有一个相同的父进程。
`close()` 是为了结束所有的子进程的添加。`join()` 是针对当前进程使之成为守护进程, 只有等到其它所有的进程运行结束, 才会退出程序。当然除了用 `Pool` 我们还可以使用 `ProcessPoolExecutor`。如果你的 python 是 3.2 以前的, 需要自行安装。

```
class ProcessPoolExecutor(_base.Executor):
    def __init__(self, max_workers=None):
        """Initializes a new ProcessPoolExecutor instance.

        Args:
            max_workers: The maximum number of processes that can be used to
                execute the given calls. If None or not given then as many
                worker processes will be created as the machine has processors.
        """
        _check_system_limits()

        if max_workers is None:
            self._max_workers = os.cpu_count() or 1
        else:
            if max_workers <= 0:
                raise ValueError("max_workers must be greater than 0")
            self._max_workers = max_workers
```

我们看到源码中, 参数简单粗暴, 只有一个进程池数量。跟上边讲的多线程的控制使用方法基本一样, 他们都继承的是 `_base.Executor`。由于上边讲了, 这里就不在细细讲了, 直接写代码。

```
× 多线程.py 53 def get_name(i):
× process.py — 54     print('the process is %s(%s)' % (os.getpid(),i))
× process.py — 55     #print('myparent is %s(%s)' % (os.getppid(),i))
× pool.py 56     return i+1
× _base.py 57 if __name__ == '__main__':
58     max_workers=20
59     test_list=[i*i for i in range(100000)]
60     with futures.ProcessPoolExecutor(max_workers=max_workers) as excutor:
61         result = excutor.map(get_name,test_list)
62         print(type(result))
63         print(list(result))
64

the process is 10140(9983806561)
the process is 10140(9987803721)
the process is 10140(9991801681)
the process is 10140(9995800441)
the process is 10140(9999800001)
<class 'itertools.chain'>
[1, 2, 5, 10, 17, 26, 37, 50, 65, 82, 101, 122, 145, 170, 197, 226, 257, 290, 325, 362, 401,
1025, 1090, 1157, 1226, 1297, 1370, 1445, 1522, 1601, 1682, 1765, 1850, 1937, 2026, 2117, 2210,
3250, 3365, 3482, 3601, 3722, 3845, 3970, 4097, 4226, 4357, 4490, 4625, 4762, 4901, 5042, 5185,
6725, 6890, 7057, 7226, 7397, 7570, 7745, 7922, 8101, 8282, 8465, 8650, 8837, 9026, 9217, 9410]
```

唯一不同的就是返回的结果是一个可迭代对象的串接, 至于为什么在多线程中是生成器而在多进程中是可迭代对象串接, 这个问题我也不懂。后续如果有了新的知识, 我再给大家做补充。

小结: 今天的内容到这也就结束了, 在多线程和多进程中, 还有许多知识未能涉及。为什么说在 python 中多线程只是一个美丽的梦? 多进程的分布式又是如何设置的? 协程的用法意义究竟是什么? 这些问题, 我们后续继续研究。由于本人也不是科班出生, 所学知识也只是从课本得来, 如果有什么书写错误或者理解错误, 还望指教。如果你有什么更好的意见或者建议请联系本人。QQ: 383750993。QQ 交流群: 518980304。