12.7 创建一个线程池¶

问题¶

你创建一个工作者线程池，用来响应客户端请求或执行其他的工作。

解决方案¶

`concurrent.futures` 函数库有一个 `ThreadPoolExecutor` 类可以被用来完成这个任务。 下面是一个简单的TCP服务器，使用了一个线程池来响应客户端：

```
from socket import AF_INET, SOCK_STREAM, socket
from concurrent.futures import ThreadPoolExecutor

def echo_client(sock, client_addr):
    '''
    Handle a client connection
    '''
    print('Got connection from', client_addr)
    while True:
        msg = sock.recv(65536)
        if not msg:
            break
        sock.sendall(msg)
    print('Client closed connection')
    sock.close()

def echo_server(addr):
    pool = ThreadPoolExecutor(128)
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        pool.submit(echo_client, client_sock, client_addr)

echo_server(('',15000))
```

如果你想手动创建你自己的线程池， 通常可以使用一个Queue来轻松实现。下面是一个稍微不同但是手动实现的例子：

```
from socket import socket, AF_INET, SOCK_STREAM
from threading import Thread
from queue import Queue

def echo_client(q):
    '''
    Handle a client connection
    '''
    sock, client_addr = q.get()
    print('Got connection from', client_addr)
    while True:
        msg = sock.recv(65536)
        if not msg:
            break
        sock.sendall(msg)
    print('Client closed connection')

    sock.close()

def echo_server(addr, nworkers):
    # Launch the client workers
    q = Queue()
    for n in range(nworkers):
```

```
        t = Thread(target=echo_client, args=(q,))
        t.daemon = True
        t.start()

    # Run the server
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        q.put((client_sock, client_addr))

echo_server(('',15000), 128)
```

使用 `ThreadPoolExecutor` 相对于手动实现的一个好处在于它使得 任务提交者更方便的从被调用函数中获取返回值。例如，你可能会像下面这样写：

```
from concurrent.futures import ThreadPoolExecutor
import urllib.request

def fetch_url(url):
    u = urllib.request.urlopen(url)
    data = u.read()
    return data

pool = ThreadPoolExecutor(10)
# Submit work to the pool
a = pool.submit(fetch_url, 'http://www.python.org')
b = pool.submit(fetch_url, 'http://www.pypy.org')

# Get the results back
x = a.result()
y = b.result()
```

例子中返回的handle对象会帮你处理所有的阻塞与协作，然后从工作线程中返回数据给你。 特别的， `a.result()` 操作会阻塞进程直到对应的函数执行完成并返回一个结果。

讨论¶

通常来讲，你应该避免编写线程数量可以无限制增长的程序。例如，看看下面这个服务器：

```
from threading import Thread
from socket import socket, AF_INET, SOCK_STREAM

def echo_client(sock, client_addr):
    '''
    Handle a client connection
    '''
    print('Got connection from', client_addr)
    while True:
        msg = sock.recv(65536)
        if not msg:
            break
        sock.sendall(msg)
    print('Client closed connection')
    sock.close()

def echo_server(addr, nworkers):
    # Run the server
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        t = Thread(target=echo_client, args=(client_sock, client_addr))
        t.daemon = True
```

```
        t.daemon = True
        t.start()

echo_server(('',15000))
```

尽管这个也可以工作， 但是它不能抵御有人试图通过创建大量线程让你服务器资源枯竭而崩溃的攻击行为。 通过使用预先初始化的线程池，你可以设置同时运行线程的上限数量。

你可能会关心创建大量线程会有什么后果。 现代操作系统可以很轻松的创建几千个线程的线程池。 甚至，同时几千个线程等待工作并不会对其他代码产生性能影响。 当然了，如果所有线程同时被唤醒并立即在CPU上执行，那就不同了——特别是有了全局解释器锁GIL。 通常，你应该只在I/O处理相关代码中使用线程池。

创建大的线程池的一个可能需要关注的问题是内存的使用。 例如，如果你在OS X系统上面创建2000个线程，系统显示Python进程使用了超过9GB的虚拟内存。 不过，这个计算通常是有误差的。当创建一个线程时，操作系统会预留一个虚拟内存区域来 放置线程的执行栈（通常是8MB大小）。但是这个内存只有一小片段被实际映射到真实内存中。 因此，Python进程使用到的真实内存其实很小 （比如，对于2000个线程来讲，只使用到了70MB的真实内存，而不是9GB）。 如果你担心虚拟内存大小，可以使用 `threading.stack_size()` 函数来降低它。例如：

```
import threading
threading.stack_size(65536)
```

如果你加上这条语句并再次运行前面的创建2000个线程试验， 你会发现Python进程只使用到了大概210MB的虚拟内存，而真实内存使用量没有变。 注意线程栈大小必须至少为32768字节，通常是系统内存页大小（4096、8192等）的整数倍。