Programação assíncrona em python

Programação assíncrona

Suponhamos um programa que faz pedidos a vários servidores.
Enquanto espera, pode ser importante que possa continuar a execução fazendo outra tarefa.
Mais tarde deverá poder aceder ao resultado que pediu.

O módulo **Concurrent.futures** permite este tipo de programação

Fornece dois tipos de interface que permitem:

- gerir uma pool de threads;

- gerir uma pool de processos,

Concurrent.futures

Para interactuar com as pools, estão definidas duas classes:

- Executors – para gerir pools de workers

- Futures - para gerir a obtenção dos resultados

Exemplo:

```
from concurrent import futures
import threading
import time

def task(n):
    print ( "%s: sleeping %s" % (threading.current_thread().name, n))
    time.sleep(n / 10)
    print( "%s: done with %s" % (threading.current_thread().name, n))
    return n / 10

ex = futures.ThreadPoolExecutor(max_workers=2) #pool com duas threads
print('main: starting')
results = ex.map(task, range(5, 0, -1)) # lança a execução assíncrona da
função (task) para cada elemento da lista

print("main: unprocessed results %s " % results)
print("main: waiting for real results")
real_results = list(results)   # espera por cada resposta
print("main: results: %s" % real_results)
```

Output:

```
main: starting
Thread-5: sleeping 5
Thread-6: sleeping 4
main: unprocessed results <generator object result_iterator at
0x00000000076492D0>
main: waiting for real results
Thread-6: done with 4
Thread-6: sleeping 3
Thread-5: done with 5
Thread-5: sleeping 2
Thread-6: done with 3
Thread-6: sleeping 1
Thread-5: done with 2
Thread-6: done with 1
main: results: [0.5, 0.4, 0.3, 0.2, 0.1]   # resultados ordenados segundo a ordem dos argumentos
```

Quantas vezes é executada a função task??

E se fosse **ex = futures.ThreadPoolExecutor(max_workers=10) , o que era diferente?**

Submeter uma só tarefa

```
executor = futures.ThreadPoolExecutor(max_workers=2)
print("main: starting")

f = executor.submit (task, 5)
print("main: future: %s" , f)
print("main: waiting for results")
result = f.result()     # espera que devolva o resultado

print("main: result: %s" , result)
print("main: future after result: %s" , f)
```

Output:

main: starting
Thread-5: sleeping 5

main: future: <Future at 0x76063c8 state=running>

main: waiting for results

Thread-5: done with 5

main: result: 0.5

main: future after result: <Future at 0x76063c8 state=finished returned float>

Esperar pelo resultado das tarefas em qualquer ordem

```
from concurrent import futures
import random
import time

def task(n):
    time.sleep(random.random())
    return (n, n / 10)

ex = futures.ThreadPoolExecutor(max_workers=5)
print("main: starting")

wait_for = [ ex.submit (task, i) for i in range(5, 0, -1) ]

for f in futures.as_completed( wait_for ):
    print("main: result: " , f.result() )
```

Output:

(1)

main: starting
main: result: (5, 0.5)
main: result: (4, 0.4)
main: result: (2, 0.2)
main: result: (1, 0.1)
main: result: (3, 0.3)

(2)

main: starting
main: result: (3, 0.3)
main: result: (1, 0.1)
main: result: (2, 0.2)
main: result: (4, 0.4)
main: result: (5, 0.5)
…

Cada execução pode ter uma sequência de resultados diferente

Se fosse:

```
for i in wait_for:
    print(i.result())
```

O output seria:

```
main: starting
(5, 0.5)
(4, 0.4)
(3, 0.3)
(2, 0.2)
(1, 0.1)
```

O resultado corresponde à sequência dos dados de entrada.

Future Callbacks

Para executar um ação quando termina a execução de uma tarefa, sem explicitamente esperar que a tarefa termine:

A função **add_done_callback()** da classe Future, permite especificar uma função que será invocada quando o future estiver concluído

A função callback deve ter como argumento uma instância da classe Future.

O callback é invocado independentemente da razão porque o future é considerado concluído (fim de código ou erro). É necessário verificar o seu status.

Future Callbacks

```
from concurrent import futures
import time

def task(n):
    print(" %s: sleeping" % n)
    time.sleep(0.5)
    print(" %s: done" % n)
    return n / 10


def done(fn):          Função callback
    if fn.cancelled():
        print("%s: canceled " % fn.arg)
    elif fn.done():
        error = fn.exception()
        if error:
            print("%s: error returned: %s" % (fn.arg, error))
        else:
            result = fn.result()
            print("%s: value returned: %s" % (fn.arg, result))
```

Future Callbacks

```
if __name__ == '__main__':
    ex = futures.ThreadPoolExecutor(max_workers=2)
    print("main: starting")
    f = ex.submit(task, 5)
    f.arg = 5
    f.add_done_callback(done)
    result = f.result()
    print (result)
```

Output:

```
main: starting
5: sleeping
5: done
5: value returned: 0.5
0.5
```

“Executors” podem se usados como “context managers”:

```
from concurrent import futures

def task(n):
    print(n)

with futures.ThreadPoolExecutor(max_workers=2) as ex:
    print('main: starting')
    ex.submit(task, 1)
    ex.submit(task, 2)
    ex.submit(task, 3)
    ex.submit(task, 4)
# quando termina o bloco with todos os recursos são libertados
print('main: done')
```

Pool de Processos

A classe ProcessPoolExecutor tem a mesma interface que a Thread PoolExecutor mas usa processos em vez de threads

- Um conjunto de processos podes ser reusado para múltiplas tarefas.

Pool de Processos

```
from concurrent import futures
import os

def task(n):
    return (n, os.getpid())

if __name__ == '__main__':
    ex = futures.ProcessPoolExecutor(max_workers=2)
    results = ex.map(task, range(5, 0, -1))
    for n, pid in results:
        print("run task %s in process %s" % (n, pid))
```

Output:

```
run task 5 in process 1812
run task 4 in process 1812
run task 3 in process 7180
run task 2 in process 1812
run task 1 in process 7180
```

Desempenho:

##Concurrent.Futures Pooling - Asynchronous Programming

```
import concurrent.futures
import time
number_list = [1,2,3,4,5,6,7,8,9,10]

def evaluate_item(x):
    #count...just to make an operation
    result_item = count(x)
    #print the input item and the result
    print ("item " + str(x) + " result " + str(result_item))

def count(number) :
    for i in range(0,10000000):
        i=i+1
    return i*number
```

Desempenho:

##Concurrent.Futures Pooling - Asynchronous Programming

```
if __name__ == "__main__":
    ##Sequential Execution
    start_time = time.clock()

    for item in number_list:
        evaluate_item(item)

    print ("Sequential execution in " + \
                                    str(time.clock() - start_time), "seconds")
```

Desempenho:

##Concurrent.Futures Pooling - Asynchronous Programming

```
##Thread pool Execution
start_time_1 = time.clock()
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    for item in number_list:
        executor.submit(evaluate_item, item)
 print ("Thread pool execution in " + str(time.clock() - start_time_1), "seconds")


##Process pool Execution
start_time_2 = time.clock()
with concurrent.futures.ProcessPoolExecutor(max_workers=5) as executor:
    for item in number_list:
        executor.submit(evaluate_item, item)
 print ("Process pool execution in " + str(time.clock() - start_time_2), "seconds")
```

Output

item 1 result 10000000
item 2 result 20000000
item 3 result 30000000
item 4 result 40000000
item 5 result 50000000
item 6 result 60000000
item 7 result 70000000
item 8 result 80000000
item 9 result 90000000
item 10 result 100000000

Sequential execution in 6.395211333567499 seconds

item 3 result 30000000
item 2 result 20000000
item 5 result 50000000
item 4 result 40000000
item 1 result 10000000
item 7 result 70000000
item 6 result 60000000
item 9 result 90000000
item 10 result 100000000
item 8 result 80000000

Thread pool execution in 6.492513954842721 seconds

item 1 result 10000000
item 7 result 70000000
item 3 result 30000000
item 6 result 60000000

Process pool execution in 4.258110230712969 seconds

item 4 result 40000000
item 9 result 90000000
item 5 result 50000000
item 10 result 100000000
item 2 result 20000000
item 8 result 80000000