Multiprocessing in Python

Multiprocessing

- Semelhante ao módulo threading
- Cria processos que executam em paralelo
- Executado no Windows e Linux
- Possui diversas APIs auxiliares
  - Pools
  - Queues
  - Event
  - Pipe

Diferenças

Threading	Multiprocessing
Memória Compartilhada	Memória Separada
Facilidade em compartilhar objetos	Dificuldade em compartilhar objetos
Facilidade em ocasionar condição de corrida	Mais difícil em ocasionar condição de corrida
Comunicação com memória compartilhada gera menos overhead	Comunicação IPC gera mais overhead
Estrutura como Queues e Pipes deve ser utilizados de pacotes externos	Estruturas como Queues e Pipe são definidos e otimizados pelo próprio pacote
Código mais difícil de entender e acertar	Código mais fácil de entender e acertar

A classe Process

- Usamos o objeto Process
- Chamamos o método
  - start()
- Semelhante ao pacote threading

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

```python
from multiprocessing import Process
import os

def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

Contexto e métodos de início

- Multiprocessamento suporta três maneiras de iniciar um processo
  - Spawn
  - Fork
  - Forkserver

Spawn

- Processo filho herda apenas os recursos necessários para executar o método run()
- Mais lento comparado ao demais métodos
- Disponível no Windows e Unix
- Padrão do Windows

Fork

- O processo usa o os.fork() para criar u novo processo
- “Processo filho” criado é idêntico ao processo pai
- Todos os recursos do pai são herdados
- Disponível apenas no Unix
- O padrão Unix

Forkserver

- Quando o programa é iniciado e selecionado o método start() do forkserver, um processo do servidor é iniciado
- A partir de então, sempre que um novo processo é necessário, o processo pai se conecta ao servidor e solicita que ele execute um novo processo
- O processo do servidor fork é uma simples thread que executa o os.fork()
- Nenhum recurso desnecessário é herdado
- Disponível em plataformas Unix que suportam a passagem de descritores de arquivos em pipes Unix.

```python
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    mp.set_start_method('spawn')
    q = mp.Queue()
    p = mp.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

```python
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    ctx = mp.get_context('spawn')
    q = ctx.Queue()
    p = ctx.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

Troca de Objetos entre processos

- Suporta 2 tipos de canais de comunicação
  - Queues
  - Pipe

```python
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints "[42, None, 'hello']"
    p.join()
```

```python
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())   # prints "[42, None, 'hello']"
    p.join()
```

Event

- Permite comunicação de estados entre os processos.
- Possuem apenas 2 estados
  - Set()
  - Unset()
- Por exemplo, o meu programa está executando enquanto uma condição é verdadeira
- A medida que ela muda o estado do programa muda
- Determina uma determinada variável ou situação do meu programa
- No fim das contas, serve para sincronizar os processos

Sincronização entre Processos

- Contém todas as primitivas de sincronização do pacote threading

```
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    try:
        print('hello world', i)
    finally:
        l.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()
```

Compartilhando Dados entre Processos

- Geralmente é melhor evitar o uso do estado compartilhado, tanto quanto possível
- No entanto, caso precise, os dados podem ser armazenados em uma memória compartilhada usando
  - Value
  - Array

```python
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])
```

```python
from multiprocessing import Pool, TimeoutError
import time
import os

def f(x):
    return x*x

if __name__ == '__main__':
    # start 4 worker processes
    with Pool(processes=4) as pool:

        # print "[0, 1, 4,..., 81]"
        print(pool.map(f, range(10)))

        # print same numbers in arbitrary order
        for i in pool.imap_unordered(f, range(10)):
            print(i)

        # evaluate "f(20)" asynchronously
        res = pool.apply_async(f, (20,))      # runs in *only* one process
        print(res.get(timeout=1))             # prints "400"

        # evaluate "os.getpid()" asynchronously
        res = pool.apply_async(os.getpid, ()) # runs in *only* one process
        print(res.get(timeout=1))             # prints the PID of that process
```

```
    # launching multiple evaluations asynchronously *may* use more processes
    multiple_results = [pool.apply_async(os.getpid, ()) for i in range(4)]
    print([res.get(timeout=1) for res in multiple_results])

    # make a single worker sleep for 10 secs
    res = pool.apply_async(time.sleep, (10,))
    try:
        print(res.get(timeout=1))
    except TimeoutError:
        print("We lacked patience and got a multiprocessing.TimeoutError")

    print("For the moment, the pool remains available for more work")

# exiting the 'with'-block has stopped the pool
print("Now the pool is closed and no longer available")
```

Primitivas de Sincronização

- Class.multiprocessing.Barrier(parties, action=None, timeout=None)
- Class.multiprocessing.BoundedSemaphore(value=1)
- Class.multiprocessing.Condition(lock=None)
- Class.multiprocessing.Event()
- Class.multiprocessing.Lock()
- Class.multiprocessing.RLock()
- Class.multiprocessing.Semaphore(value=None)