

## Primeri iz apstrakcija serverskih procesa (managers) u paketu “multiprocessing”

Osnovne klase vezane za serverske procese su: BaseManager i Manager

Glavna referenca je Python online dokumentacija:

<https://docs.python.org/3/library/multiprocessing.html>

### Deljenje stanja između procesa pomoću serverskih procesa

**Primer 1:** Glavni proces main prvo napravi serverski proces manager (pomoću konstruktora Manager), zatim koristeći objekat manager napravi deljeni rečnik d i deljenu listu l (pomoću metoda dict i list), i na kraju pokrene proces potomak (kome prosledi d i l kao parametre). Proces potomak postavi tri para ključ-vrednost u rečniku d i obrne elemente u listi l.

Napomena: Ovaj primer je već rađen u osnova multiprocessinga (primer 7), ovde ga samo ponavljamo.

Tabela 1.a: Programski kod modula w1\_manager.py.

```
from multiprocessing import Process, Manager
```

```
def f(d, l):  
    d[1] = '1'  
    d['2'] = 2  
    d[0.25] = None  
    l.reverse()
```

```
if __name__ == '__main__':  
    with Manager() as manager:  
        d = manager.dict()  
        l = manager.list(range(10))  
  
        p = Process(target=f, args=(d, l))  
        p.start()  
        p.join()  
  
        print(d)  
        print(l)
```

Tabela 1.b: Rezultat izvršenja modula w1\_manager.py.

```
C:\Z>python w1_manager.py  
{1: '1', '2': 2, 0.25: None}  
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

## **Klijent i server u zasebnim procesima**

**Primer 2:** U ovom primeru klijent i server se izvršavaju u zasebnim procesima (terminalima). Najpre je potrebno pokrenuti servera, a zatim klijenta.

Glavni proces u terminalu servera najpre pravi objekat manager pomoću konstruktora BaseManager (parametri su par (IP adresa, port) i autentifikacioni ključ b'abc'), zatim pravi objekat server pomoću metode get\_server i na kraju pokreće serverski proces pomoću metode serve\_forever.

Da bi pokrenuli server, prekopirajte kod iz donje tabele u Pjaton interpreter u serverskom terminalu:

Tabela 2.a: Programski kod za proces u serverskom terminalu.
--

<pre>from multiprocessing.managers import BaseManager manager = BaseManager(address=('', 50000), authkey=b'abc') server = manager.get_server() server.serve_forever()</pre>
---

Glavni proces u terminalu klijenta najpre pravi objekat m pomoću konstruktora BaseManager (sa istim parametrima kao server), a zatim uspostavlja vezu sa serverskim procesom pomoću metode connect.

Da bi pokrenuli klijenta, prekopirajte kod iz donje tabele u Pajton interpreter u klijentskom terminalu:

Tabela 2.b: Programski kod za proces u klijentskom terminalu.
---

<pre>from multiprocessing.managers import BaseManager m = BaseManager(address=('127.0.0.1', 50000), authkey=b'abc') m.connect()</pre>
---

## **Korisnički definisani serverski procesi**

**Primer 3:** U ovom primeru se prvo definiše korisnička klasa MyManager kao klasa izvedena iz BaseManager, a zatim se na njoj registruje tip podataka 'Maths' koji je definisan kao klasa MathsClass. Glavni process main pravi objekat manager ovog tipa, zatim pravi objekat zastupnika (proxy) maths, i na kraju nad objektom maths poziva metode add i mul.

Tabela 3.a: Programski kod modula w2_customized_manager.py.
---

<pre>from multiprocessing.managers import BaseManager  class MathsClass:     def add(self, x, y):         return x + y     def mul(self, x, y):         return x * y  class MyManager(BaseManager):     pass</pre>
--

```
MyManager.register('Maths', MathsClass)
```

```
if __name__ == '__main__':  
    with MyManager() as manager:  
        maths = manager.Maths()  
        print(maths.add(4, 3))    # prints 7  
        print(maths.mul(7, 8))   # prints 56
```

Tabela 3.b: Rezultat izvršenja modula w2\_customized\_manager.py.

```
D:\Z>python w2_customized_manager.py  
7  
56
```

### Udaljeni serverski procesi

**Primer 4:** U ovom primeru se koriste tri terminala – jedan serverski i dva klijentska. Serverski proces rukuje redom (queue), prvi klijentski proces dodaje poruku u red pozivajući metodu put nad objektom zastupnika, a drugi klijentski proces vadi poruku iz reda pomoću metode get.

Modul w3\_remote\_manger: Glavni proces u terminalu servera najpre napravi red queue (pomoću konstruktora Queue), zatim izvede klasu QueueManager iz klase BaseManager, i na njoj registruje novi tip podataka 'get\_queue' koji rukuje sa redom queue.

Tabela 4.a: Programski kod skripte w3\_remote\_manger.py.

```
from multiprocessing.managers import BaseManager  
from queue import Queue  
queue = Queue()  
class QueueManager(BaseManager):  
    pass  
  
QueueManager.register('get_queue', callable=lambda:queue)  
m = QueueManager(address=('', 50000), authkey=b'abracadabra')  
s = m.get_server()  
s.serve_forever()
```

Tabela 4.b: Rezultat izvršenja skripte w3\_remote\_manger.py u terminalu servera.

```
D:\Z>python w3_remote_manger.py  
(napomena: ovaj program ništa ne ispisuje)
```

Glavni proces u terminalu prvog klijenta prvo radi isto kao u Primeru 2, zatim pravi objekat zastupnika queue, i na kraju dodaje u red poruku 'hello' pomoću metode put.

Tabela 4.c: Programski kod skripte w3\_remote\_client\_1.py.

```
from multiprocessing.managers import BaseManager
class QueueManager(BaseManager):
    pass

QueueManager.register('get_queue')
m = QueueManager(address=('127.0.0.1', 50000), authkey=b'abracadabra')
m.connect()
queue = m.get_queue()
queue.put('hello')
```

Tabela 4.d: Rezultat izvršenja skipte w3\_remote\_client\_1.py u terminalu prvog klijenta.

(Argument '-i' omogućava nastavak izvršenja u režimu interpretera; sledeće komande se izdaju ručno.)

```
D:\Z>python -i w3_remote_client_1.py
>>> queue.put('poruka 1')
>>> exit()
```

Glavni proces u terminalu drugog klijenta prvo radi isto kao proces u terminalu prvog klijenta, s tim što na kraju vadi poruku 'hello' pomoću metode get.

Tabela 4.e: Programski kod skripte w3\_remote\_client\_2.py.

```
from multiprocessing.managers import BaseManager
class QueueManager(BaseManager):
    pass

QueueManager.register('get_queue')
m = QueueManager(address=('127.0.0.1', 50000), authkey=b'abracadabra')
m.connect()
queue = m.get_queue()
print(queue.get())
```

Tabela 4.f: Rezultat izvršenja skipte w3\_remote\_client\_2.py u terminalu drugog klijenta.

(Argument '-i' omogućava nastavak izvršenja u režimu interpretera; sledeće komande se izdaju ručno.)

```
D:\Z>python -i w3_remote_client_2.py
hello
>>> queue.get()
'poruka 1'
>>> exit()
```

**Primer 5:** Ovaj primer je vrlo sličan predhodnom, s tim što se sada serverski proces i jedan lokalni klijentski proces izvršavaju u jednom terminalu, a drugi klijentski proces se izvršava u drugom terminalu.

Serverski proces i lokalni klijentski proces su definisani u modulu `w3_manager_and_local_client.py`. Ovde je na početku izvedena klasa `Worker` iz klase `Process`, čiji konstruktor postavlja referencu na red `q`, a metoda `run` upisuje poruku 'local hello' u red `q`.

Glavni process `main` prvo napravi red queue, a zatim napravi i pokrene lokalnog klijenta (objekt `w`).

Tabela 5.a: Modul python `w3_manager_and_local_client.py`, koji se izvršava u prvom terminalu.

```
from multiprocessing import Process, Queue
from multiprocessing.managers import BaseManager

class Worker(Process):
    def __init__(self, q):
        self.q = q
        super(Worker, self).__init__()
    def run(self):
        self.q.put('local hello')

class QueueManager(BaseManager):
    pass

if __name__ == '__main__':
    queue = Queue()
    w = Worker(queue)
    w.start()

    QueueManager.register('get_queue', callable=lambda: queue)
    m = QueueManager(address=('', 50000), authkey=b'abracadabra')
    s = m.get_server()
    s.serve_forever()
```

Tabela 5.b: Rezultat izvršenja modula `w3_manager_and_local_client.py` u prvom terminalu.

```
D:\Z>python w3_manager_and_local_client.py
(napomena: ovaj program ništa ne ispisuje)
```

Drugi klijentski proces radi isto kao drugi klijentski procesi u primeru 4 (ali sad prima poruku 'local hello').

Tabela 4.f: Rezultat izvršenja skipte `w3_remote_client_2.py` u drugom terminalu.

```
D:\Z>python w3_remote_client_2.py
local hello
```

## **Objekti zastupnici (Proxies)**

**Primer 6:** Glavni proces prvo napravi objekat manager, a zatim napravi deljenu listu l pomoću metode list, čija povratna vrednost je objekt zastupnika liste l. Dalje radi sa l kao sa lokalnom listom, a u suštini l je objekat zastupnika deljene liste, što pokazuje rezultat poziva print(repr(l)): <ListProxy object, typeid 'list' at 0x1af4da8f208>.

Da bi se pokrenuo ovaj primer treba iskopirati sledeći kod u Python interpreter:

```
from multiprocessing import Manager
manager = Manager()
l = manager.list([i*i for i in range(10)])
print(l)
print(repr(l))
l[4]
l[2:5]
```

**Primer 7:** Ovo je primer sa ugnježdavanjem objekata zastupnika (b je ugnježđen u a). Glavni proces prvo napravi dve prazne deljene liste a i b, a zatim doda b u a. Dalje radi sa njima kao da su a i b lokalne liste, a u stvari a i b su objekti zastupnici: print(a, b) [<ListProxy object, typeid 'list' at 0x1ac2bb40898>] [].

Prekopirajte sledeći kod u istoj sesiji bez izlaska iz interpretera:

```
a = manager.list()
b = manager.list()
a.append(b)      # referent of a now contains referent of b
print(a, b)
b.append('hello')
print(a[0], b)
```

**Primer 8:** Slično, objekti zastupnika rečnika i liste mogu biti ugnježđeni jedni u drugima. U ovom primeru, glavni process prvo napravi deljenu listu, koja sadrži dva deljena rečnika, a zatim postavlja po dva para ključ-vrednost u oba rečnika. Nastavite u istoj sesiji:

```
l_outer = manager.list([ manager.dict() for i in range(2) ])
d_first_inner = l_outer[0]
d_first_inner['a'] = 1
d_first_inner['b'] = 2
l_outer[1]['c'] = 3
l_outer[1]['z'] = 26
print(l_outer[0])
print(l_outer[1])
```

**Primer 9:** Važno upozorenje: Ažuriranje deljenog kontejnera preko objekta zastupnika se mora učiniti eksplicitno, kako bi se lokalna ažuriranja nad objektom zastupnika obavila i nad deljenim kontejnerom. U ovom primeru, naredbe `d['a'] = 1` i `d['b'] = 2` su lokalna ažuriranja, a naredba `lproxy[0] = d` dovodi do ažuriranja deljenog kontejnera (rečnika). Nastavite u istoj sesiji:

```
# create a list proxy and append a mutable object (a dictionary)
lproxy = manager.list()
lproxy.append({})
# now mutate the dictionary
d = lproxy[0]
d['a'] = 1
d['b'] = 2
# at this point, the changes to d are not yet synchronized,
# by updating the dictionary, the proxy is notified of the change
lproxy[0] = d
```

**Primer 10:** Upozorenje: Objekti zastupnici ne podržavaju poređenje sa vrednošću. Sledeći izraz ima vrednost `False`. Nastavite u istoj sesiji:

```
manager.list([1,2,3]) == [1,2,3]
```

Zato je potrebno koristiti kopiju referenta u poređenjima. Nastavite u istoj sesiji:

```
l123 = manager.list([1,2,3])
cp = []
for i in range(0,3):
    cp.append(l123[i])

print(cp)
cp == [1,2,3]
```

**Primer 11:** Generalno, metoda objekta zastupnika `_callmethod` poziva i vraća rezultat metode zastupničkog referenta (objekta koji on zastupa). Nastavite u istoj sesiji:

```
l = manager.list(range(10))
l._callmethod('__len__')
l._callmethod('__getitem__', (slice(2, 7),)) # equivalent to l[2:7]
l._callmethod('__getitem__', (8,))          # equivalent to l[8]
```