... continued

Continues the discussion on using a manager to share objects among processes.

The `BaseManager`'s register method takes in a number of parameters and it's important that we discuss how they are used. We'll explain each one of them below:

- typeid
- callable
- proxytype
- exposed
- method_to_typeid
- create_method

In the previous sections, we already discussed the first two parameters. We'll study the remaining using an example. Consider the following `Pair` class that holds two values x and y.

```
class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def get_x(self):
        # return self.x
        return fancy_type(self.x)

    def set_x(self, new_x):
        self.x = new_x

    def set_y(self, new_y):
        self.y = new_y
```

Let's say we want to register an object of this class with a manager so that it can be shared among processes. The most straight-forward way to register would be:

```
pair = Pair(5, 6)
manager.register('get_pair', callable=lambda: pair)
```

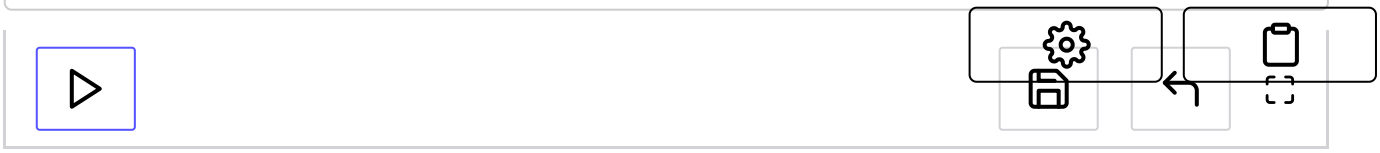
All the other register parameters will take on default values. The proxy returned for the pair would be **`AutoProxy`** and all the public methods on the pair object are by default available for consuming processes.

If you wanted to restrict the methods available to consumers you can specify the list of available methods using the **`exposed`** parameter. The register call would be as follows:

```
pair = Pair(5, 6)
manager.register('get_pair', callable=lambda: pair, exposed=[
'get_x'])
```

The above change would only make the `get_x()` method available on the pair object. The below runnable script barfs when we try to access `set_x()` since it has not been exposed.

```
from multiprocessing.managers import BaseManager
from multiprocessing import Process
import multiprocessing
import time

port_num = 55555


def process_task():
    manager = BaseManager(address=('', port_num))
    manager.register('get_pair')
    manager.connect()

    p = manager.get_pair()
    p.set_x(7)
    p.set_y(7)
    print(p.get_x())
    print(p.get_y())


class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def get_x(self):
        # return self.x
        return self.x

    def get_y(self):
        # return self.x
        return self.y

    def set_x(self, new_x):
        self.x = new_x

    def set_y(self, new_y):
        self.y = new_y


if __name__ == '__main__':
    p1 = Process(target=process_task)

    manager = BaseManager(address=('', port_num))
    pair = Pair(5, 6)
    manager.register('get_pair', callable=lambda: pair, exposed=['get_x'])
    manager.start()
    p1.start()

    time.sleep(3)
    manager.shutdown()
```



The next parameter is **`proxytype`**. We can use a proxy class of our own choice if we don't like the default **`AutoProxy`**. Say for instance if we want a proxy to always return the square of the return value of **`get_x()`**, we can specify a proxy as below and use it when invoking **`register()`** in the consumer process.

Using custom proxy

```
class MyProxy(BaseProxy):
    def get_x(self):
        x = self._callmethod('get_x')
        return x * x
```

And the register call in the consumer process will look like:

```
manager.register('get_pair', proxytype=MyProxy)
```

The complete runnable code appears below:

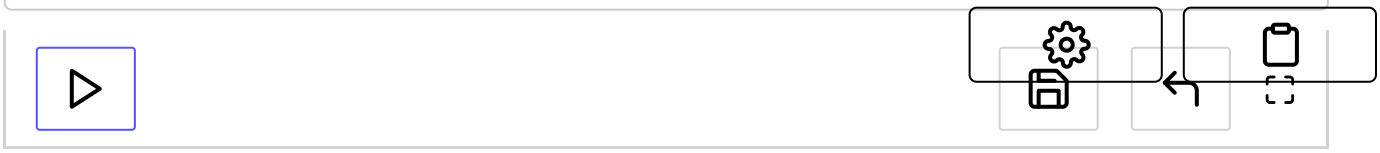
```
from multiprocessing.managers import BaseManager, BaseProxy
from multiprocessing import Process
import time

port_num = 55555

class MyProxy(BaseProxy):
    def get_x(self):
        x = self._callmethod('get_x')
        return x * x


def process_task():
    manager = BaseManager(address=('', port_num))
    manager.register('get_pair', proxytype=MyProxy)
    manager.connect()

    p = manager.get_pair()
    print(p.get_x())



class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def get_x(self):
        return self.x

    def get_y(self):
        return self.y

    def set_x(self, new_x):
        self.x = new_x

    def set_y(self, new_y):
        self.y = new_y

if __name__ == '__main__':
    p1 = Process(target=process_task)
    manager = BaseManager(address=('', port_num))

    pair = Pair(15, 6)

    manager.register('get_pair', callable=lambda: pair, exposed=['get_x'])

    manager.start()
    p1.start()
    time.sleep(3)
    manager.shutdown()
```

The next parameter we'll discuss is **`create_method`** which takes on a boolean true or false. In the consumer process we invoke the **`register()`** method as follows:

```
manager.register('get_pair', proxytype=MyProxy)
```

If **`create_method`** is true (which it is be default) then the manager object has a method added to it as an attribute by the same name as the typeid parameter. We can then get the proxy for the shared object using the call **`manager.typeid()`**. If we set **`create_method`** to false, the method won't be created as an attribute of the manager object. It makes sense to set it to false on the server process if sharing of objects isn't desired on that node. As an example, we repeat the code from the previous example and set **`create_method`** to false in the consumer process and see that the script runs into errors.

```
from multiprocessing.managers import BaseManager, BaseProxy
from multiprocessing import Process
import time

port_num = 55555

class MyProxy(BaseProxy):
    def get_x(self):
        x = self._callmethod('get_x')
        return x * x


def process_task():
    manager = BaseManager(address=('', port_num))
    manager.register('get_pair', proxytype=MyProxy, create_method=False)
    manager.connect()

    p = manager.get_pair()
    print(p.get_x())



class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def get_x(self):
        return self.x

    def get_y(self):
        return self.y

    def set_x(self, new_x):
        self.x = new_x

    def set_y(self, new_y):
        self.y = new_y

if __name__ == '__main__':
    p1 = Process(target=process_task)
    manager = BaseManager(address=('', port_num))

    pair = Pair(15, 6)

    manager.register('get_pair', callable=lambda: pair, exposed=['get_x'], create_m

    manager.start()
    p1.start()
    time.sleep(3)
    manager.shutdown()
```



Note on **line#48** we are setting `create_method` to false too. Set it to true and re-run the program and you'll see 15 being printed. This may seem counterintuitive but realize that by default the child process was created as a fork and receives a copy of the memory of the parent process. The parent process which is also the server manager registers the `get_pair()` with the manager object, which internally adds it to the registry of the `BaseManager` class. The child process receives the same copy of the `BaseManager` class and thus already has the `get_pair()` method in its registry. If we changed the start method to "spawn" then instead of 15 being printed, an error would occur.

The `create_method` parameter makes more sense when looking at the following usage:

```
manager.register('get_pair', Pair)
```

Instead of sharing an object, we are passing the class `Pair` and each time we invoke the `get_pair()` method, a new pair object is *created* and returned. The `create_method` if set to false would leave no way to create new objects of a shared class or callable. The below runnable script shows how a class is shared.

```
from multiprocessing.managers import BaseManager, BaseProxy
from multiprocessing import Process
import time

port_num = 55555


def process_task():
    manager = BaseManager(address=('', port_num))
    manager.register('get_pair')
    manager.connect()

    obj1 = manager.get_pair(1,3)
    obj2 = manager.get_pair(2,4)

    # verify two different objects are created
    print(obj1.get_x())
    print(obj2.get_x())


class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def get_x(self):
        return self.x

    def get_y(self):
        return self.y

    def set_x(self, new_x):
        self.x = new_x

    def set_y(self, new_y):
        self.y = new_y

if __name__ == '__main__':
    p1 = Process(target=process_task)
    manager = BaseManager(address=('', port_num))

    manager.register('get_pair', Pair)

    manager.start()
    p1.start()
    time.sleep(3)
    manager.shutdown()
```

Last but not the least, is the **`method_to_typeid`** parameter. The parameter comes in handy if we want to return proxies from exposed methods of the proxy returned by the manager object. It is a mapping between method names and typeids. Let's see an example in action. We'll create an **`Item`** class and add an object of that type to the **`Pair`** class.

```
class Item:

    def __init__(self):
        self.item = "intialized"

    def change(self, new_item):
        self.item = new_item

    def retrieve(self):
        return self.item;
```

The changed pair class would become:

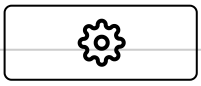
```
class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.i = Item()

    def get_item(self):
        return self.i

    def get_x(self):
        # return self.x
        return self.x

    def get_y(self):
        # return self.x
        return self.y

    def set_x(self, new_x):
        self.x = new_x

    def set_y(self, new_y):
        self.y = new_y
```

Notice the **`get_item()`** method returns an instance of the **`Item`** class. A process that interacts with the proxy of an object of the **`Pair`** class will receive a copy of the item object held by the pair object if the **`get_item()`** method is invoked. It may be desirable that instead of the copy a proxy to the item object is returned so that any changes in the consumer process are also reflected in the shared object.

In order to modify the item object held by the shared pair object in the server process, we'll change the register call as follows:

```
    manager.register('get_pair', callable=lambda: pair,
                     method_to_typeid={'get_item': 'ItemWrappe
```

```
r'})
```



The above call states that the return value of the **`get_sum()`** method call should return a proxy. The typeId for which the proxy gets created is yet another class **`ItemWrapper`**. We'll define the wrapper class as below:

```
class ItemWrapper:
    def __init__(self, res):
        self.item = res

    def set_wrapped_item(self, new_item):
        self.item.change(new_item)

    def get_wrapped_item(self):
        return self.item.retrieve()
```

The referent for the proxy will be an instance of **`ItemWrapper`**. Internally, an instance of the wrapper object is created and is passed the result of the **`get_item()`** method call. The result is nothing but the item object held by the pair object. The typeid that is provided as a mapping for a given method has its constructor invoked with the result of the method call. A proxy is created over the typeid and returned. In our example, the flow is as follows:

1. User code invokes method **`get_item()`** which returns the item object.
2. The python framework creates an object of type **`ItemWrapper`** and passes the item object from step#1 into its constructor. The typeid specified as the mapping from a method must have a single argument constructor where the single argument is the result of the method call.
3. The python framework creates an **`Autoproxy`** over the object created in step#2

created in step#2.

4. The proxy created in step#3 is returned to the user code instead of the item.

Below is the complete code. Follow the print statements and realize that we can manipulate the item object contained by the shared pair object directly in the consumer process.

```
from multiprocessing.managers import BaseManager, BaseProxy
from multiprocessing import Process
import multiprocessing
import time

port_num = 55555


def consumer_process():
    mgr = BaseManager(address=('', port_num))
    mgr.register('ItemWrapper', ItemWrapper)
    mgr.register('get_pair')
    mgr.connect()

    p = mgr.get_pair()
    proxy1 = p.get_item()
    print("proxy1 sees item as : " + str(proxy1.get_wrapped_item()))
    old_val = proxy1.get_wrapped_item()
    proxy1._callmethod('set_wrapped_item', (7,))
    print("proxy1 changes item from {0} to {1}".format(old_val, proxy1.get_wrapped_

    proxy2 = mgr.get_pair().get_item()
    print("proxy2 sees item as : " + str(proxy2.get_wrapped_item()))
    old_val = proxy2.get_wrapped_item()
    proxy2.set_wrapped_item(11)
    print("proxy2 changes item from {0} to {1}".format(old_val, proxy2.get_wrapped_
    print("proxy1 sees item as : " + str(proxy1.get_wrapped_item()))


def producer_process():
    manager = BaseManager(address=('', port_num))

    pair = Pair(15, 6)

    manager.register('ItemWrapper', ItemWrapper)
    manager.register('get_pair', callable=lambda: pair,
                     method_to_typeid={'get_item': 'ItemWrapper'})

    manager.start()
    time.sleep(3)
    print("\nAfter changes in consumer process, the producer process sees item as =
    manager.shutdown()


class ItemWrapper:
    def __init__(self, res):
        self.item = res

    def set_wrapped_item(self, new_item):
        self.item.change(new_item)
```

```
    def get_wrapped_item(self):
        return self.item.retrieve()


class Item:

    def __init__(self):
        self.item = "intialized"

    def change(self, new_item):
        self.item = new_item

    def retrieve(self):
        return self.item;


class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.i = Item()

    def get_item(self):
        return self.i

    def get_x(self):
        return self.x

    def get_y(self):
        return self.y

    def set_x(self, new_x):
        self.x = new_x

    def set_y(self, new_y):
        self.y = new_y


if __name__ == '__main__':
    p1 = Process(target=producer_process)
    p2 = Process(target=consumer_process)

    p1.start()
    time.sleep(2)
    p2.start()

    p2.join()
    p1.join()
```

It is interesting to see what happens if we repeat the above program without the **method_to_typeid** parameter. The changes made by the two proxies don't change the shared copy of the original pair object that the manager returns to the consumer process. This is consistent with the document which states that if the mapping **method_to_typeid** is None then the object returned by the method will be copied by value.

```
from multiprocessing.managers import BaseManager, BaseProxy
from multiprocessing import Process
import multiprocessing
import time

port_num = 55555


def consumer_process():
    mgr = BaseManager(address=('', port_num))
    mgr.register('ItemWrapper', ItemWrapper)
    mgr.register('get_pair')
    mgr.connect()

    p = mgr.get_pair()
    item1 = p.get_item()
    print("proxy1 sees item as : " + str(item1.retrieve()))
    old_val = item1.retrieve()
    item1.change(7)
    print("proxy1 changes item from {0} to {1}".format(old_val, item1.retrieve()))

    p2 = mgr.get_pair()
    item2 = p2.get_item()
    print("proxy2 sees item as : " + str(item2.retrieve()))
    old_val = item2.retrieve()
    item2.change(11)
    print("proxy2 changes item from {0} to {1}".format(old_val, item2.retrieve()))
    print("proxy1 sees item as : " + str(item1.retrieve()))


def producer_process():
    manager = BaseManager(address=('', port_num))

    pair = Pair(15, 6)

    manager.register('ItemWrapper', ItemWrapper)
    manager.register('get_pair', callable=lambda: pair)

    manager.start()
    time.sleep(3)
    print("\nAfter changes in consumer process, the producer process sees item as =
    manager.shutdown()


class ItemWrapper:
    def __init__(self, res):
        self.item = res

    def set_wrapped_item(self, new_item):
        self.item.change(new_item)
```

```
    def get_wrapped_item(self):
        return self.item.retrieve()


class Item:

    def __init__(self):
        self.item = "intialized"

    def change(self, new_item):
        self.item = new_item

    def retrieve(self):
        return self.item;


class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.i = Item()

    def get_item(self):
        return self.i

    def get_x(self):
        return self.x

    def get_y(self):
        return self.y

    def set_x(self, new_x):
        self.x = new_x

    def set_y(self, new_y):
        self.y = new_y


if __name__ == '__main__':
    p1 = Process(target=producer_process)
    p2 = Process(target=consumer_process)

    p1.start()
    time.sleep(2)
    p2.start()

    p2.join()
    p1.join()
```

Spawning vs Forking Processes

All the examples that we have discussed under the managers topic implicitly use fork to create new processes. However, if we set the start method to spawn instead of fork, we'll need to provide a picklable object as the **`callable`** parameter in the register method. For instance, the following registration will barf:

```
    manager.register('get_pair', callable=lambda: pair)
```

The callable in the above snippet is a lambda method which isn't picklable. Here (https://docs.python.org/3/library/pickle.html#what-can-be-pickled-and-unpickled) is a link listing all picklable entities in Python. We present an example below which uses spawn start method to create new processes. Specifically, the registration call looks like as follows:

```
    manager.register('get_my_items', callable=top_level_functi
on, proxytype=ListProxy)
```

Since top level module functions are picklable, we define such a method **`top_level_function()`** that always returns a new list created by the manager. We'll study the list being returned from **`top_level_function()`** in the next section.


```
from multiprocessing.managers import BaseManager, ListProxy
from multiprocessing import Process, Manager, current_process
import time, multiprocessing, random


def top_level_function():
    return (Manager()).list(["set by {0}".format(current_process().name)])


def ProcessA(port_num):
    manager = BaseManager(address=('127.0.0.1', port_num))
    manager.register('get_my_items', callable=top_level_function, proxytype=ListPro
    manager.start()

    time.sleep(3)


def ProcessB(port_num):
    manager = BaseManager(address=('127.0.0.1', port_num))

    manager.register('get_my_items')
    manager.connect()
    proxy_items = manager.get_my_items()
    proxy_items.append("Educative is Great !")

    print(proxy_items[0])
    print(proxy_items[1])


if __name__ == '__main__':
    multiprocessing.set_start_method("spawn")

    port_num = random.randint(10000, 60000)

    # Start another process which will access the shared string
    p1 = Process(target=ProcessA, args=(port_num,))
    p1.start()

    time.sleep(0.8)

    p2 = Process(target=ProcessB, args=(port_num,))
    p2.start()

    p1.join()
    p2.join()
```




Back

Next

... continued

SyncManager

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/continued__multiprocessing__python-concurrency-for-senior-engineering-interviews)