12.10 定义一个Actor任务¶

问题¶

你想定义跟actor模式中类似“actors”角色的任务

解决方案¶

actor模式是一种最古老的也是最简单的并行和分布式计算解决方案。 事实上，它天生的简单性是它如此受欢迎的重要原因之一。 简单来讲，一个actor就是一个并发执行的任务，只是简单的执行发送给它的消息任务。 响应这些消息时，它可能还会给其他actor发送更进一步的消息。 actor之间的通信是单向和异步的。因此，消息发送者不知道消息是什么时候被发送， 也不会接收到一个消息已被处理的回应或通知。

结合使用一个线程和一个队列可以很容易的定义actor，例如：

```
from queue import Queue
from threading import Thread, Event

# Sentinel used for shutdown
class ActorExit(Exception):
    pass

class Actor:
    def __init__(self):
        self._mailbox = Queue()

    def send(self, msg):
        '''
        Send a message to the actor
        '''
        self._mailbox.put(msg)

    def recv(self):
        '''
        Receive an incoming message
        '''
        msg = self._mailbox.get()
        if msg is ActorExit:
            raise ActorExit()
        return msg

    def close(self):
        '''
        Close the actor, thus shutting it down
        '''
        self.send(ActorExit)

    def start(self):
        '''
        Start concurrent execution
        '''
        self._terminated = Event()
        t = Thread(target=self._bootstrap)

        t.daemon = True
        t.start()

    def _bootstrap(self):
        try:
            self.run()
        except ActorExit:
            pass
        finally:
            self._terminated.set()

    def join(self):
```

```
        self._terminated.wait()

    def run(self):
        '''
        Run method to be implemented by the user
        '''
        while True:
            msg = self.recv()

# Sample ActorTask
class PrintActor(Actor):
    def run(self):
        while True:
            msg = self.recv()
            print('Got:', msg)

# Sample use
p = PrintActor()
p.start()
p.send('Hello')
p.send('World')
p.close()
p.join()
```

这个例子中，你使用actor实例的 `send()` 方法发送消息给它们。 其机制是，这个方法会将消息放入一个队里中， 然后将其转交给处理被接受消息的一个内部线程。 `close()` 方法通过在队列中放入一个特殊的哨兵值（ActorExit）来关闭这个actor。 用户可以通过继承Actor并定义实现自己处理逻辑run()方法来定义新的actor。 `ActorExit` 异常的使用就是用户自定义代码可以在需要的时候来捕获终止请求 （异常被get()方法抛出并传播出去）。

如果你放宽对于同步和异步消息发送的要求， 类actor对象还可以通过生成器来简化定义。例如：

```
def print_actor():
    while True:

        try:
            msg = yield      # Get a message
            print('Got:', msg)
        except GeneratorExit:
            print('Actor terminating')

# Sample use
p = print_actor()
next(p)     # Advance to the yield (ready to receive)
p.send('Hello')
p.send('World')
p.close()
```

讨论¶

actor模式的魅力就在于它的简单性。 实际上，这里仅仅只有一个核心操作 `send()` . 甚至，对于在基于actor系统中的“消息”的泛化概念可以已多种方式被扩展。 例如，你可以以元组形式传递标签消息，让actor执行不同的操作，如下：

```
class TaggedActor(Actor):
    def run(self):
        while True:
             tag, *payload = self.recv()
             getattr(self,'do_'+tag)(*payload)

    # Methods correponding to different message tags
    def do_A(self, x):
        print('Running A', x)

    def do_B(self, x, y):
        print('Running B', x, y)

# Example
a = TaggedActor()
a.start()
```

```
a.send(('A', 1))      # Invokes do_A(1)
a.send(('B', 2, 3))   # Invokes do_B(2,3)
a.close()
a.join()
```

作为另外一个例子，下面的actor允许在一个工作者中运行任意的函数， 并且通过一个特殊的Result对象返回结果：

```
from threading import Event
class Result:
    def __init__(self):
        self._evt = Event()
        self._result = None

    def set_result(self, value):
        self._result = value

        self._evt.set()

    def result(self):
        self._evt.wait()
        return self._result

class Worker(Actor):
    def submit(self, func, *args, **kwargs):
        r = Result()
        self.send((func, args, kwargs, r))
        return r

    def run(self):
        while True:
            func, args, kwargs, r = self.recv()
            r.set_result(func(*args, **kwargs))

# Example use
worker = Worker()
worker.start()
r = worker.submit(pow, 2, 3)
worker.close()
worker.join()
print(r.result())
```

最后，“发送”一个任务消息的概念可以被扩展到多进程甚至是大型分布式系统中去。 例如，一个类actor对象的 `send()` 方法可以被编程让它能在一个套接字连接上传输数据 或通过某些消息中间件（比如AMQP、ZMQ等）来发送。