11.8 实现远程方法调用¶

问题¶

你想在一个消息传输层如 `sockets` 、 `multiprocessing connections` 或 `ZeroMQ` 的基础之上实现一个简单的远程过程调用（RPC）。

解决方案¶

将函数请求、参数和返回值使用pickle编码后，在不同的解释器直接传送pickle字节字符串，可以很容易的实现RPC。 下面是一个简单的PRC处理器，可以被整合到一个服务器中去：

```
# rpcserver.py

import pickle
class RPCHandler:
    def __init__(self):
        self._functions = { }

    def register_function(self, func):
        self._functions[func.__name__] = func

    def handle_connection(self, connection):
        try:
            while True:
                # Receive a message
                func_name, args, kwargs = pickle.loads(connection.recv())
                # Run the RPC and send a response
                try:
                    r = self._functions[func_name](*args,**kwargs)
                    connection.send(pickle.dumps(r))
                except Exception as e:
                    connection.send(pickle.dumps(e))
        except EOFError:
            pass
```

要使用这个处理器，你需要将它加入到一个消息服务器中。你有很多种选择， 但是使用 `multiprocessing` 库是最简单的。下面是一个RPC服务器例子：

```
from multiprocessing.connection import Listener
from threading import Thread

def rpc_server(handler, address, authkey):
    sock = Listener(address, authkey=authkey)
    while True:
        client = sock.accept()
        t = Thread(target=handler.handle_connection, args=(client,))
        t.daemon = True
        t.start()

# Some remote functions
def add(x, y):
    return x + y
```

```
def sub(x, y):
    return x - y

# Register with a handler
handler = RPCHandler()
handler.register_function(add)
handler.register_function(sub)

# Run the server
rpc_server(handler, ('localhost', 17000), authkey=b'peekaboo')
```

为了从一个远程客户端访问服务器，你需要创建一个对应的用来传送请求的RPC代理类。例如

```
import pickle

class RPCProxy:
    def __init__(self, connection):
        self._connection = connection
    def __getattr__(self, name):
        def do_rpc(*args, **kwargs):
            self._connection.send(pickle.dumps((name, args, kwargs)))
            result = pickle.loads(self._connection.recv())
            if isinstance(result, Exception):
                raise result
            return result
        return do_rpc
```

要使用这个代理类，你需要将其包装到一个服务器的连接上面，例如：

```
>>> from multiprocessing.connection import Client
>>> c = Client(('localhost', 17000), authkey=b'peekaboo')
>>> proxy = RPCProxy(c)
>>> proxy.add(2, 3)

5
>>> proxy.sub(2, 3)
-1
>>> proxy.sub([1, 2], 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "rpcserver.py", line 37, in do_rpc
    raise result
TypeError: unsupported operand type(s) for -: 'list' and 'int'
>>>
```

要注意的是很多消息层（比如 `multiprocessing` ）已经使用pickle序列化了数据。 如果是这样的话，对 `pickle.dumps()` 和 `pickle.loads()` 的调用要去掉。

讨论¶

`RPCHandler` 和 `RPCProxy` 的基本思路是很比较简单的。 如果一个客户端想要调用一个远程函数，比如 `foo(1, 2, z=3)` ,代理类创建一个包含了函数名和参数的元组 `('foo', (1, 2), {'z': 3})` 。 这个元组被pickle序列化后通过网络连接发生出去。 这一步在 `RPCProxy` 的 `__getattr__()` 方法返回的 `do_rpc()` 闭包中完成。 服务器接收后通过pickle反序列化消息，查找函数名看看是否已经注册过，然后执行相应的函数。 执行结果(或异常)被pickle序列化后返回发送给客户端。我们的实例需要依赖 `multiprocessing` 进行通信。 不过，这种方式可以适用于其他任何消息系统。例如，如果你想在ZeroMQ之上实习RPC，

仅仅只需要将连接对象换成合适的ZeroMQ的socket对象即可。

由于底层需要依赖pickle，那么安全问题就需要考虑了（因为一个聪明的黑客可以创建特定的消息，能够让任意函数通过pickle反序列化后被执行）。因此你永远不要允许来自不信任或未认证的客户端的RPC。特别是你绝对不要允许来自Internet的任意机器的访问，这种只能在内部被使用，位于防火墙后面并且不要对外暴露。

作为pickle的替代，你也许可以考虑使用JSON、XML或一些其他的编码格式来序列化消息。例如，本机实例可以很容易的改写成JSON编码方案。还需要将 `pickle.loads()` 和 `pickle.dumps()` 替换成 `json.loads()` 和 `json.dumps()` 即可：

```
# jsonrpcserver.py
import json

class RPCHandler:
    def __init__(self):
        self._functions = { }

    def register_function(self, func):
        self._functions[func.__name__] = func

    def handle_connection(self, connection):
        try:
            while True:
                # Receive a message
                func_name, args, kwargs = json.loads(connection.recv())
                # Run the RPC and send a response
                try:
                    r = self._functions[func_name](*args,**kwargs)
                    connection.send(json.dumps(r))
                except Exception as e:
                    connection.send(json.dumps(str(e)))
        except EOFError:
            pass

# jsonrpcclient.py
import json

class RPCProxy:
    def __init__(self, connection):
        self._connection = connection
    def __getattr__(self, name):
        def do_rpc(*args, **kwargs):
            self._connection.send(json.dumps((name, args, kwargs)))
            result = json.loads(self._connection.recv())
            return result
        return do_rpc
```

实现RPC的一个比较复杂的问题是如何去处理异常。至少，当方法产生异常时服务器不应该奔溃。因此，返回给客户端的异常所代表的含义就要好好设计了。如果你使用pickle，异常对象实例在客户端能被反序列化并抛出。如果你使用其他的协议，那得想想另外的方法了。不过至少，你应该在响应中返回异常字符串。我们在JSON的例子中就是使用的这种方式。

对于其他的RPC实现例子，我推荐你看看在XML-RPC中使用的 `SimpleXMLRPCServer` 和 `ServerProxy` 的实现，也就是11.6小节中的内容。