5.21 序列化Python对象¶

问题¶

你需要将一个Python对象序列化为一个字节流，以便将它保存到一个文件、存储到数据库或者通过网络传输它。

解决方案¶

对于序列化最普遍的做法就是使用 `pickle` 模块。为了将一个对象保存到一个文件中，可以这样做：

```
import pickle

data = ... # Some Python object
f = open('somefile', 'wb')
pickle.dump(data, f)
```

为了将一个对象转储为一个字符串，可以使用 `pickle.dumps()` ：

```
s = pickle.dumps(data)
```

为了从字节流中恢复一个对象，使用 `pickle.load()` 或 `pickle.loads()` 函数。比如：

```
# Restore from a file
f = open('somefile', 'rb')
data = pickle.load(f)

# Restore from a string
data = pickle.loads(s)
```

讨论¶

对于大多数应用程序来讲， `dump()` 和 `load()` 函数的使用就是你有效使用 `pickle` 模块所需的全部了。 它可适用于绝大部分Python数据类型和用户自定义类的对象实例。 如果你碰到某个库可以让你在数据库中保存/恢复Python对象或者是通过网络传输对象的话， 那么很有可能这个库的底层就使用了 `pickle` 模块。

`pickle` 是一种Python特有的自描述的数据编码。 通过自描述，被序列化后的数据包含每个对象开始和结束以及它的类型信息。 因此，你无需担心对象记录的定义，它总是能工作。 举个例子，如果要处理多个对象，你可以这样做：

```
>>> import pickle
>>> f = open('somedata', 'wb')
>>> pickle.dump([1, 2, 3, 4], f)
>>> pickle.dump('hello', f)
>>> pickle.dump({'Apple', 'Pear', 'Banana'}, f)
>>> f.close()
>>> f = open('somedata', 'rb')
>>> pickle.load(f)
[1, 2, 3, 4]
>>> pickle.load(f)
'hello'
>>> pickle.load(f)
{'Apple', 'Pear', 'Banana'}
>>>
```

你还能序列化函数，类，还有接口，但是结果数据仅仅将它们的名称编码成对应的代码对象。例如：

```
>>> import math
>>> import pickle.
>>> pickle.dumps(math.cos)
b'\x80\x03cmath\ncos\nq\x00.'
>>>
```

当数据反序列化回来的时候，会先假定所有的源数据时可用的。 模块、类和函数会自动按需导入进来。对于Python数据被不同机器上的解析器所共享的应用程序而言， 数据的保存可能会有问题，因为所有的机器都必须访问同一个源代码。

注

千万不要对不信任的数据使用pickle.load()。
pickle在加载时有一个副作用就是它会自动加载相应模块并构造实例对象。
但是某个坏人如果知道pickle的工作原理，
他就可以创建一个恶意的数据导致Python执行随意指定的系统命令。
因此，一定要保证pickle只在相互之间可以认证对方的解析器的内部使用。

有些类型的对象是不能被序列化的。这些通常是那些依赖外部系统状态的对象， 比如打开的文件，网络连接，线程，进程，栈帧等等。 用户自定义类可以通过提供 `__getstate__()` 和 `__setstate__()` 方法来绕过这些限制。 如果定义了这两个方法， `pickle.dump()` 就会调用 `__getstate__()` 获取序列化的对象。 类似的， `__setstate__()` 在反序列化时被调用。为了演示这个工作原理， 下面是一个在内部定义了一个线程但仍然可以序列化和反序列化的类：

```
# countdown.py
import time
import threading

class Countdown:
   def __init__(self, n):
      self.n = n
      self.thr = threading.Thread(target=self.run)
      self.thr.daemon = True
      self.thr.start()

   def run(self):
      while self.n > 0:
         print('T-minus', self.n)
         self.n -= 1
         time.sleep(5)

   def __getstate__(self):
      return self.n

   def __setstate__(self, n):
      self.__init__(n)
```

试着运行下面的序列化试验代码：

```
>>> import countdown
>>> c = countdown.Countdown(30)
>>> T-minus 30
T-minus 29
T-minus 28
...
```

```
>>> # After a few moments
>>> f = open('cstate.p', 'wb')
>>> import pickle
>>> pickle.dump(c, f)
>>> f.close()
```

然后退出Python解析器并重启后再试验下：

```
>>> f = open('cstate.p', 'rb')
>>> pickle.load(f)
countdown.Countdown object at 0x10069e2d0>
T-minus 19
T-minus 18
...
```

你可以看到线程又奇迹般的重生了，从你第一次序列化它的地方又恢复过来。

`pickle` 对于大型的数据结构比如使用 `array` 或 `numpy` 模块创建的二进制数组效率并不是一个高效的编码方式。 如果你需要移动大量的数组数据，你最好是先在一个文件中将其保存为数组数据块或使用更高级的标准编码方式如HDF5 (需要第三方库的支持)。

由于 `pickle` 是Python特有的并且附着在源码上，所有如果需要长期存储数据的时候不应该选用它。 例如，如果源码变动了，你所有的存储数据可能会被破坏并且变得不可读取。 坦白来讲，对于在数据库和存档文件中存储数据时，你最好使用更加标准的数据编码格式如XML，CSV或JSON。 这些编码格式更标准，可以被不同的语言支持，并且也能很好的适应源码变更。

最后一点要注意的是 `pickle` 有大量的配置选项和一些棘手的问题。 对于最常见的使用场景，你不需要去担心这个，但是如果你要在一个重要的程序中使用pickle去做序列化的话， 最好去查阅一下 官方文档 。