Pickle模块使用指南

（http://www.cnblogs.com/cobbliu/archive/2012/09/04/2670178.html）

持久性就是指保持对象，甚至在多次执行同一程序之间也保持对象。通过本文，您会对 Python对象的各种持久性机制（从关系数据库到 Python 的 pickle以及其它机制）有一个总体认识。另外，还会让您更深一步地了解Python 的对象序列化能力。

什么是持久性？

持久性的基本思想很简单。假定有一个 Python 程序，它可能是一个管理日常待办事项的程序，您希望在多次执行这个程序之间可以保存应用程序对象（待办事项）。换句话说，您希望将对象存储在磁盘上，便于以后检索。这就是持久性。要达到这个目的，有几种方法，每一种方法都有其优缺点。

例如，可以将对象数据存储在某种格式的文本文件中，譬如 CSV 文件。或者可以用关系数据库，譬如 Gadfly、MySQL、PostgreSQL 或者 DB2。这些文件格式和数据库都非常优秀，对于所有这些存储机制，Python 都有健壮的接口。

这些存储机制都有一个共同点：存储的数据是独立于对这些数据进行操作的对象和程序。这样做的好处是，数据可以作为共享的资源，供其它应用程序使用。缺点是，用这种方式，可以允许其它程序访问对象的数据，这违背了面向对象的封装性原则 — 即对象的数据只能通过这个对象自身的公共（public）接口来访问。

另外，对于某些应用程序，关系数据库方法可能不是很理想。尤其是，关系数据库不理解对象。相反，关系数据库会强行使用自己的类型系统和关系数据模型（表），每张表包含一组元组（行），每行包含具有固定数目的静态类型字段（列）。如果应用程序的对象模型不能够方便地转换到关系模型，那么在将对象映射到元组以及将元组映射回对象方面，会碰到一定难度。这种困难常被称为阻碍性不匹配（impedence-mismatch）问题。

对象持久性

如果希望透明地存储 Python 对象，而不丢失其身份和类型等信息，则需要某种形式的对象序列化：它是一个将任意复杂的对象转成对象的文本或二进制表示的过程。同样，必须能够将对象经过序列化后的形式恢复到原有的对象。在 Python 中，这种序列化过程称为 pickle，可以将对象 pickle 成字符串、磁盘上的文件或者任何类似于文件的对象，也可以将这些字符串、文件或任何类似于文件的对象 unpickle 成原来的对象。我们将在本文后面详细讨论 pickle。

假定您喜欢将任何事物都保存成对象，而且希望避免将对象转换成某种基于非对象存储的开销；那么 pickle 文件可以提供这些好处，但有时可能需要比这种简单的 pickle 文件更健壮以及更具有可伸缩性的事物。例如，只用 pickle 不能解决命名和查找 pickle 文件这样的问题，另外，它也不能支持并发地访问持久性对象。如果需要这些方面的功能，则要求助类似于 ZODB（针对 Python 的 Z 对象数据库）这类数据库。ZODB 是一个健壮的、多用户的和面向对象的数据库系统，它能够存储和管理任意复杂的 Python 对象，并支持事务操作和并发控制。（请参阅 参考资料，以下载 ZODB。）令人足够感兴趣的是，甚至 ZODB 也依靠 Python 的本机序列化能力，而且要有效地使用 ZODB，必须充分了解 pickle。

另一种令人感兴趣的解决持久性问题的方法是 Prevayler，它最初是用 Java 实现的（有关 Prevaylor 方面的developerWorks 文章，请参阅 参考资料）。最近，一群 Python 程序员将 Prevayler 移植到了 Python 上，另起名为 PyPerSyst，由 SourceForge 托管（有关至 PyPerSyst 项目的链接，请参阅 参考资料）。Prevayler/PyPerSyst 概念也是建立在 Java 和 Python 语言的本机序列化能力之上。PyPerSyst 将整个对象系统保存在内存中，并通过不时地将系统快照 pickle 到磁盘以及维护一个命令日志（通过此日志可以重新应用最新的快照）来提供灾难恢复。所以，尽管使用 PyPerSyst 的应用程序受到可用内存的限制，但好处是本机对象系统可以完全装入到内存中，因而速度极快，而且实现起来要比如 ZODB 这样的数据库简单，ZODB 允许对象的数目比同时在能内存中所保持的对象要多。

既然我们已经简要讨论了存储持久对象的各种方法，那么现在该详细探讨 pickle 过程了。虽然我们主要感兴趣的是探索以各种方式来保存 Python 对象，而不必将其转换成某种其它格式，但我们仍然还有一些需要关注的地方，譬如：如何有效地 pickle 和 unpickle 简单对象以及复杂对象，包括定制类的实例；如何维护对象的引用，包括循环引用和递归引用；以及如何处理类定义发生的变化，从而使用以前经过 pickle 的实例时不会发生问题。我们将在随后关于 Python 的 pickle 能力探讨中涉及所有这些问题。

一些经过 pickle 的 Python

pickle 模块及其同类模块 cPickle 向 Python 提供了 pickle 支持。后者是用 C 编码的，它具有更好的性能，对于大多数应用程序，推荐使用该模块。我们将继续讨论 pickle ，但本文的示例实际是利用了 cPickle 。由于其中大多数示例要用 Python shell 来显示，所以先展示一下如何导入 cPickle ，并可以作为 pickle 来引用它：

>>> import cPickle as pickle

现在已经导入了该模块，接下来让我们看一下 pickle 接口。 pickle 模块提供了以下函数对： dumps(object) 返回一个字符串，它包含一个 pickle 格式的对象； loads(string) 返回包含在 pickle 字符串中的对象； dump(object, file) 将对象写到文件，这个文件可以是实际的物理文件，但也可以是任何类似于文件的对象，这个对象具有 write() 方法，可以接受单个的字符串参数； load(file) 返回包含在 pickle 文件中的对象。

缺省情况下， dumps() 和 dump() 使用可打印的 ASCII 表示来创建 pickle。两者都有一个 final 参数（可选），如果为 True ，则该参数指定用更快以及更小的二进制表示来创建 pickle。 loads() 和 load() 函数自动检测 pickle 是二进制格式还是文本格式。

清单 1 显示了一个交互式会话，这里使用了刚才所描述的 dumps() 和 loads() 函数：

清单 1. dumps() 和 loads() 的演示

**[python]** [view plaincopyprint?](http://blog.csdn.net/bh20077/article/details/6070278)

1. >>> import cPickle as pickle
2. >>> t1 = ('this is a string', 42, [1, 2, 3], None)

* >>> t1
* ('this is a string', 42, [1, 2, 3], None)
* >>> p1 = pickle.dumps(t1)
* >>> p1
* "(S'this is a string'/nI42/n(lp1/nI1/naI2/naI3/naNtp2/n."
* >>> print p1
* (S'this is a string'
* I42
* (lp1
* I1
* aI2
* aI3
* aNtp2
* .
* >>> t2 = pickle.loads(p1)
* >>> t2
* ('this is a string', 42, [1, 2, 3], None)
* >>> p2 = pickle.dumps(t1, True)
* >>> p2
* '(U/x10this is a stringK\*]q/x01(K/x01K/x02K/x03eNtq/x02.'
* >>> t3 = pickle.loads(p2)
* >>> t3
* ('this is a string', 42, [1, 2, 3], None)



注：该文本 pickle 格式很简单，这里就不解释了。事实上，在 pickle 模块中记录了所有使用的约定。我们还应该指出，在我们的示例中使用的都是简单对象，因此使用二进制 pickle 格式不会在节省空间上显示出太大的效率。然而，在实际使用复杂对象的系统中，您会看到，使用二进制格式可以在大小和速度方面带来显著的改进。

接下来，我们看一些示例，这些示例用到了 dump() 和 load() ，它们使用文件和类似文件的对象。这些函数的操作非常类似于我们刚才所看到的 dumps() 和 loads() ，区别在于它们还有另一种能力 — dump() 函数能一个接着一个地将几个对象转储到同一个文件。随后调用 load() 来以同样的顺序检索这些对象。清单 2 显示了这种能力的实际应用：

清单 2. dump() 和 load() 示例

**[python]** [view plaincopyprint?](http://blog.csdn.net/bh20077/article/details/6070278)

1. >>> a1 = 'apple'
2. >>> b1 = {1: 'One', 2: 'Two', 3: 'Three'}
3. >>> c1 = ['fee', 'fie', 'foe', 'fum']
4. >>> f1 = file('temp.pkl', 'wb')
5. >>> pickle.dump(a1, f1, True)
6. >>> pickle.dump(b1, f1, True)
7. >>> pickle.dump(c1, f1, True)
8. >>> f1.close()
9. >>> f2 = file('temp.pkl', 'rb')
10. >>> a2 = pickle.load(f2)
11. >>> a2
12. 'apple'
13. >>> b2 = pickle.load(f2)
14. >>> b2
15. {1: 'One', 2: 'Two', 3: 'Three'}
16. >>> c2 = pickle.load(f2)
17. >>> c2
18. ['fee', 'fie', 'foe', 'fum']
19. >>> f2.close()



Pickle 的威力

到目前为止，我们讲述了关于 pickle 方面的基本知识。在这一节，将讨论一些高级问题，当您开始 pickle 复杂对象时，会遇到这些问题，其中包括定制类的实例。幸运的是，Python 可以很容易地处理这种情形。

可移植性

从空间和时间上说，Pickle 是可移植的。换句话说，pickle 文件格式独立于机器的体系结构，这意味着，例如，可以在 Linux 下创建一个 pickle，然后将它发送到在 Windows 或 Mac OS 下运行的 Python 程序。并且，当升级到更新版本的 Python 时，不必担心可能要废弃已有的 pickle。Python 开发人员已经保证 pickle 格式将可以向后兼容 Python 各个版本。事实上，在 pickle 模块中提供了有关目前以及所支持的格式方面的详细信息.

清单 3. 检索所支持的格式

**[python]** [view plaincopyprint?](http://blog.csdn.net/bh20077/article/details/6070278)

1. >>> pickle.format\_version
2. '1.3'
3. >>> pickle.compatible\_formats
4. ['1.0', '1.1', '1.2']



多个引用，同一对象

在 Python 中，变量是对象的引用。同时，也可以用多个变量引用同一个对象。经证明，Python 在用经过 pickle 的对象维护这种行为方面丝毫没有困难，如清单 4 所示：

清单 4. 对象引用的维护

**[python]** [view plaincopyprint?](http://blog.csdn.net/bh20077/article/details/6070278)

1. >>> a = [1, 2, 3]
2. >>> b = a
3. >>> a
4. [1, 2, 3]
5. >>> b
6. [1, 2, 3]
7. >>> a.append(4)
8. >>> a
9. [1, 2, 3, 4]
10. >>> b
11. [1, 2, 3, 4]
12. >>> c = pickle.dumps((a, b))
13. >>> d, e = pickle.loads(c)
14. >>> d
15. [1, 2, 3, 4]
16. >>> e
17. [1, 2, 3, 4]
18. >>> d.append(5)
19. >>> d
20. [1, 2, 3, 4, 5]
21. >>> e
22. [1, 2, 3, 4, 5]



循环引用和递归引用

可以将刚才演示过的对象引用支持扩展到 循环引用（两个对象各自包含对对方的引用）和 递归引用（一个对象包含对其自身的引用）。下面两个清单着重显示这种能力。我们先看一下递归引用：

>清单 5. 递归引用

**[python]** [view plaincopyprint?](http://blog.csdn.net/bh20077/article/details/6070278)

1. >>> l = [1, 2, 3]
2. >>> l.append(l)
3. >>> l
4. [1, 2, 3, [...]]
5. >>> l[3]
6. [1, 2, 3, [...]]
7. >>> l[3][3]
8. [1, 2, 3, [...]]
9. >>> p = pickle.dumps(l)
10. >>> l2 = pickle.loads(p)
11. >>> l2
12. [1, 2, 3, [...]]
13. >>> l2[3]
14. [1, 2, 3, [...]]
15. >>> l2[3][3]
16. [1, 2, 3, [...]]



现在，看一个循环引用的示例：

清单 6. 循环引用

**[python]** [view plaincopyprint?](http://blog.csdn.net/bh20077/article/details/6070278)

1. >>> a = [1, 2]
2. >>> b = [3, 4]
3. >>> a.append(b)
4. >>> a
5. [1, 2, [3, 4]]
6. >>> b.append(a)
7. >>> a
8. [1, 2, [3, 4, [...]]]
9. >>> b
10. [3, 4, [1, 2, [...]]]
11. >>> a[2]
12. [3, 4, [1, 2, [...]]]
13. >>> b[2]
14. [1, 2, [3, 4, [...]]]
15. >>> a[2] is b
16. 1
17. >>> b[2] is a
18. 1
19. >>> f = file('temp.pkl', 'w')
20. >>> pickle.dump((a, b), f)
21. >>> f.close()
22. >>> f = file('temp.pkl', 'r')
23. >>> c, d = pickle.load(f)
24. >>> f.close()
25. >>> c
26. [1, 2, [3, 4, [...]]]
27. >>> d
28. [3, 4, [1, 2, [...]]]
29. >>> c[2]
30. [3, 4, [1, 2, [...]]]
31. >>> d[2]
32. [1, 2, [3, 4, [...]]]
33. >>> c[2] is d
34. 1
35. >>> d[2] is c
36. 1



注意，如果分别 pickle 每个对象，而不是在一个元组中一起 pickle 所有对象，会得到略微不同（但很重要）的结果，如清单 7 所示：

清单 7. 分别 pickle vs. 在一个元组中一起 pickle

**[python]** [view plaincopyprint?](http://blog.csdn.net/bh20077/article/details/6070278)

1. >>> f = file('temp.pkl', 'w')
2. >>> pickle.dump(a, f)
3. >>> pickle.dump(b, f)
4. >>> f.close()
5. >>> f = file('temp.pkl', 'r')
6. >>> c = pickle.load(f)
7. >>> d = pickle.load(f)
8. >>> f.close()
9. >>> c
10. [1, 2, [3, 4, [...]]]
11. >>> d
12. [3, 4, [1, 2, [...]]]
13. >>> c[2]
14. [3, 4, [1, 2, [...]]]
15. >>> d[2]
16. [1, 2, [3, 4, [...]]]
17. >>> c[2] is d
18. 0
19. >>> d[2] is c
20. 0



相等，但并不总是相同

正如在上一个示例所暗示的，只有在这些对象引用内存中同一个对象时，它们才是相同的。在 pickle 情形中，每个对象被恢复到一个与原来对象相等的对象，但不是同一个对象。换句话说，每个 pickle 都是原来对象的一个副本：

清单 8. 作为原来对象副本的被恢复的对象

**[python]** [view plaincopyprint?](http://blog.csdn.net/bh20077/article/details/6070278)

1. >>> j = [1, 2, 3]
2. >>> k = j
3. >>> k is j
4. 1
5. >>> x = pickle.dumps(k)
6. >>> y = pickle.loads(x)
7. >>> y
8. [1, 2, 3]
9. >>> y == k
10. 1
11. >>> y is k
12. 0
13. >>> y is j
14. 0
15. >>> k is j
16. 1



同时，我们看到 Python 能够维护对象之间的引用，这些对象是作为一个单元进行 pickle 的。然而，我们还看到分别调用 dump() 会使 Python 无法维护对在该单元外部进行 pickle 的对象的引用。相反，Python 复制了被引用对象，并将副本和被 pickle 的对象存储在一起。对于 pickle 和恢复单个对象层次结构的应用程序，这是没有问题的。但要意识到还有其它情形。

值得指出的是，有一个选项确实允许分别 pickle 对象，并维护相互之间的引用，只要这些对象都是 pickle 到同一文件即可。 pickle 和 cPickle 模块提供了一个 Pickler （与此相对应是 Unpickler ），它能够跟踪已经被 pickle 的对象。通过使用这个 Pickler ，将会通过引用而不是通过值来 pickle 共享和循环引用：

清单 9. 维护分别 pickle 的对象间的引用

**[python]** [view plaincopyprint?](http://blog.csdn.net/bh20077/article/details/6070278)

1. >>> f = file('temp.pkl', 'w')
2. >>> pickler = pickle.Pickler(f)
3. >>> pickler.dump(a)
4. <cPickle.Pickler object at 0x89b0bb8>
5. >>> pickler.dump(b)
6. <cPickle.Pickler object at 0x89b0bb8>
7. >>> f.close()
8. >>> f = file('temp.pkl', 'r')
9. >>> unpickler = pickle.Unpickler(f)
10. >>> c = unpickler.load()
11. >>> d = unpickler.load()
12. >>> c[2]
13. [3, 4, [1, 2, [...]]]
14. >>> d[2]
15. [1, 2, [3, 4, [...]]]
16. >>> c[2] is d
17. 1
18. >>> d[2] is c
19. 1



不可 pickle 的对象

一些对象类型是不可 pickle 的。例如，Python 不能 pickle 文件对象（或者任何带有对文件对象引用的对象），因为 Python 在 unpickle 时不能保证它可以重建该文件的状态（另一个示例比较难懂，在这类文章中不值得提出来）。试图 pickle 文件对象会导致以下错误：

清单 10. 试图 pickle 文件对象的结果

**[python]** [view plaincopyprint?](http://blog.csdn.net/bh20077/article/details/6070278)

1. >>> f = file('temp.pkl', 'w')
2. >>> p = pickle.dumps(f)
3. Traceback (most recent call last):
4. File "<input>", line 1, in ?
5. File "/usr/lib/python2.2/copy\_reg.py", line 57, in \_reduce
6. raise TypeError, "can't pickle %s objects" % base.\_\_name\_\_
7. TypeError: can't pickle file objects



类实例

与 pickle 简单对象类型相比，pickle 类实例要多加留意。这主要由于 Python 会 pickle 实例数据（通常是 \_dict\_ 属性）和类的名称，而不会 pickle 类的代码。当 Python unpickle 类的实例时，它会试图使用在 pickle 该实例时的确切的类名称和模块名称（包括任何包的路径前缀）导入包含该类定义的模块。另外要注意，类定义必须出现在模块的最顶层，这意味着它们不能是嵌套的类（在其它类或函数中定义的类）。

当 unpickle 类的实例时，通常不会再调用它们的 \_init\_() 方法。相反，Python 创建一个通用类实例，并应用已进行过 pickle 的实例属性，同时设置该实例的 \_class\_ 属性，使其指向原来的类。

对 Python 2.2 中引入的新型类进行 unpickle 的机制与原来的略有不同。虽然处理的结果实际上与对旧型类处理的结果相同，但 Python 使用 copy\_reg 模块的 \_reconstructor() 函数来恢复新型类的实例。

如果希望对新型或旧型类的实例修改缺省的 pickle 行为，则可以定义特殊的类的方法 \_getstate\_() 和 \_setstate\_() ，在保存和恢复类实例的状态信息期间，Python 会调用这些方法。在以下几节中，我们会看到一些示例利用了这些特殊的方法。

现在，我们看一个简单的类实例。首先，创建一个 persist.py 的 Python 模块，它包含以下新型类的定义：

清单 11. 新型类的定义

**[python]** [view plaincopyprint?](http://blog.csdn.net/bh20077/article/details/6070278)

1. class Foo(object):
2. def \_\_init\_\_(self, value):
3. self.value = value



现在可以 pickle Foo 实例，并看一下它的表示：

清单 12. pickle Foo 实例

**[python]** [view plaincopyprint?](http://blog.csdn.net/bh20077/article/details/6070278)

1. >>> import cPickle as pickle
2. >>> from Orbtech.examples.persist import Foo
3. >>> foo = Foo('What is a Foo?')
4. >>> p = pickle.dumps(foo)
5. >>> print p
6. ccopy\_reg
7. \_reconstructor
8. p1
9. (cOrbtech.examples.persist
10. Foo
11. p2
12. c\_\_builtin\_\_
13. object
14. p3
15. NtRp4
16. (dp5
17. S'value'
18. p6
19. S'What is a Foo?'
20. sb.



可以看到这个类的名称 Foo 和全限定的模块名称 Orbtech.examples.persist 都存储在 pickle 中。如果将这个实例 pickle 成一个文件，稍后再 unpickle 它或在另一台机器上 unpickle，则 Python 会试图导入 Orbtech.examples.persist 模块，如果不能导入，则会抛出异常。如果重命名该类和该模块或者将该模块移到另一个目录，则也会发生类似的错误。

这里有一个 Python 发出错误消息的示例，当我们重命名 Foo 类，然后试图装入先前进行过 pickle 的 Foo 实例时会发生该错误：

清单 13. 试图装入一个被重命名的 Foo 类的经过 pickle 的实例

**[python]** [view plaincopyprint?](http://blog.csdn.net/bh20077/article/details/6070278)

1. >>> import cPickle as pickle
2. >>> f = file('temp.pkl', 'r')
3. >>> foo = pickle.load(f)
4. Traceback (most recent call last):
5. File "<input>", line 1, in ?
6. AttributeError: 'module' object has no attribute 'Foo'



在重命名 persist.py 模块之后，也会发生类似的错误：

清单 14. 试图装入一个被重命名的 persist.py 模块的经过 pickle 的实例

**[python]** [view plaincopyprint?](http://blog.csdn.net/bh20077/article/details/6070278)

1. >>> import cPickle as pickle
2. >>> f = file('temp.pkl', 'r')
3. >>> foo = pickle.load(f)
4. Traceback (most recent call last):
5. File "<input>", line 1, in ?
6. ImportError: No module named persist



我们会在下面 模式改进这一节提供一些技术来管理这类更改，而不会破坏现有的 pickle。

特殊的状态方法

前面提到对一些对象类型（譬如，文件对象）不能进行 pickle。处理这种不能 pickle 的对象的实例属性时可以使用特殊的方法（ \_getstate\_() 和 \_setstate\_() ）来修改类实例的状态。这里有一个 Foo 类的示例，我们已经对它进行了修改以处理文件对象属性：

清单 15. 处理不能 pickle 的实例属性

**[python]** [view plaincopyprint?](http://blog.csdn.net/bh20077/article/details/6070278)

1. class Foo(object):
2. def \_\_init\_\_(self, value, filename):
3. self.value = value
4. self.logfile = file(filename, 'w')
5. def \_\_getstate\_\_(self):
6. """Return state values to be pickled."""
7. f = self.logfile
8. return (self.value, f.name, f.tell())
9. def \_\_setstate\_\_(self, state):
10. """Restore state from the unpickled state values."""
11. self.value, name, position = state
12. f = file(name, 'w')
13. f.seek(position)
14. self.logfile = f



pickle Foo 的实例时，Python 将只 pickle 当它调用该实例的 \_getstate\_() 方法时返回给它的值。类似的，在 unpickle 时，Python 将提供经过 unpickle 的值作为参数传递给实例的 \_setstate\_() 方法。在 \_setstate\_() 方法内，可以根据经过 pickle 的名称和位置信息来重建文件对象，并将该文件对象分配给这个实例的 logfile 属性。

模式改进

随着时间的推移，您会发现自己必须要更改类的定义。如果已经对某个类实例进行了 pickle，而现在又需要更改这个类，则您可能要检索和更新那些实例，以便它们能在新的类定义下继续正常工作。而我们已经看到在对类或模块进行某些更改时，会出现一些错误。幸运的是，pickle 和 unpickle 过程提供了一些 hook，我们可以用它们来支持这种模式改进的需要。

在这一节，我们将探讨一些方法来预测常见问题以及如何解决这些问题。由于不能 pickle 类实例代码，因此可以添加、更改和除去方法，而不会影响现有的经过 pickle 的实例。出于同样的原因，可以不必担心类的属性。您必须确保包含类定义的代码模块在 unpickle 环境中可用。同时还必须为这些可能导致 unpickle 问题的更改做好规划，这些更改包括：更改类名、添加或除去实例的属性以及改变类定义模块的名称或位置。

类名的更改

要更改类名，而不破坏先前经过 pickle 的实例，请遵循以下步骤。首先，确保原来的类的定义没有被更改，以便在 unpickle 现有实例时可以找到它。不要更改原来的名称，而是在与原来类定义所在的同一个模块中，创建该类定义的一个副本，同时给它一个新的类名。然后使用实际的新类名来替代 NewClassName ，将以下方法添加到原来类的定义中：

清单 16. 更改类名：添加到原来类定义的方法

**[python]** [view plaincopyprint?](http://blog.csdn.net/bh20077/article/details/6070278)

1. def \_\_setstate\_\_(self, state):
2. self.\_\_dict\_\_.update(state)
3. self.\_\_class\_\_ = NewClassName



当 unpickle 现有实例时，Python 将查找原来类的定义，并调用实例的 \_setstate\_() 方法，同时将给新的类定义重新分配该实例的 \_class\_ 属性。一旦确定所有现有的实例都已经 unpickle、更新和重新 pickle 后，可以从源代码模块中除去旧的类定义。

属性的添加和删除

这些特殊的状态方法 \_getstate\_() 和 \_setstate\_() 再一次使我们能控制每个实例的状态，并使我们有机会处理实例属性中的更改。让我们看一个简单的类的定义，我们将向其添加和除去一些属性。这是是最初的定义：

清单 17. 最初的类定义

**[python]** [view plaincopyprint?](http://blog.csdn.net/bh20077/article/details/6070278)

1. class Person(object):
2. def \_\_init\_\_(self, firstname, lastname):
3. self.firstname = firstname
4. self.lastname = lastname



假定已经创建并 pickle 了 Person 的实例，现在我们决定真的只想存储一个名称属性，而不是分别存储姓和名。这里有一种方式可以更改类的定义，它将先前经过 pickle 的实例迁移到新的定义：

清单 18. 新的类定义

**[python]** [view plaincopyprint?](http://blog.csdn.net/bh20077/article/details/6070278)

1. class Person(object):
2. def \_\_init\_\_(self, fullname):
3. self.fullname = fullname
4. def \_\_setstate\_\_(self, state):
5. if 'fullname' not in state:
6. first = ''
7. last = ''
8. if 'firstname' in state:
9. first = state['firstname']
10. del state['firstname']
11. if 'lastname' in state:
12. last = state['lastname']
13. del state['lastname']
14. self.fullname = " ".join([first, last]).strip()
15. self.\_\_dict\_\_.update(state)



在这个示例，我们添加了一个新的属性 fullname ，并除去了两个现有的属性 firstname 和 lastname 。当对先前进行过 pickle 的实例执行 unpickle 时，其先前进行过 pickle 的状态会作为字典传递给 \_setstate\_() ，它将包括 firstname 和 lastname 属性的值。接下来，将这两个值组合起来，并将它们分配给新属性 fullname 。在这个过程中，我们删除了状态字典中旧的属性。更新和重新 pickle 先前进行过 pickle 的所有实例之后，现在可以从类定义中除去 \_setstate\_() 方法。

模块的修改

在概念上，模块的名称或位置的改变类似于类名称的改变，但处理方式却完全不同。那是因为模块的信息存储在 pickle 中，而不是通过标准的 pickle 接口就可以修改的属性。事实上，改变模块信息的唯一办法是对实际的 pickle 文件本身执行查找和替换操作。至于如何确切地去做，这取决于具体的操作系统和可使用的工具。很显然，在这种情况下，您会想备份您的文件，以免发生错误。但这种改动应该非常简单，并且对二进制 pickle 格式进行更改与对文本 pickle 格式进行更改应该一样有效。