10.4 将模块分割成多个文件¶

问题¶

你想将一个模块分割成多个文件。但是你不想将分离的文件统一成一个逻辑模块时使已有的代码遭到破坏。

解决方案¶

程序模块可以通过变成包来分割成多个独立的文件。考虑下下面简单的模块：

```
# mymodule.py
class A:
    def spam(self):
        print('A.spam')

class B(A):
    def bar(self):
        print('B.bar')
```

假设你想mymodule.py分为两个文件，每个定义的一个类。要做到这一点，首先用mymodule目录来替换文件mymodule.py。 这这个目录下，创建以下文件：

```
mymodule/
    __init__.py
    a.py
    b.py
```

在a.py文件中插入以下代码：

```
# a.py
class A:
    def spam(self):
        print('A.spam')
```

在b.py文件中插入以下代码：

```
# b.py
from .a import A
class B(A):
    def bar(self):
        print('B.bar')
```

最后，在 __init__.py 中，将2个文件粘合在一起：

```
# __init__.py
from .a import A
from .b import B
```

如果按照这些步骤，所产生的包MyModule将作为一个单一的逻辑模块：

```
>>> import mymodule
>>> a = mymodule.A()
>>> a.spam()
A.spam
>>> b = mymodule.B()
>>> b.bar()
B.bar
>>>
```

讨论¶

在这个章节中的主要问题是一个设计问题，不管你是否希望用户使用很多小模块或只是一个模块。举个例子，在一个大型的代码库中，你可以将这一切都分割成独立的文件，让用户使用大量的import语句，就像这样：

```
from mymodule.a import A
from mymodule.b import B
...
```

这样能工作，但这让用户承受更多的负担，用户要知道不同的部分位于何处。通常情况下，将这些统一起来，使用一条import将更加容易，就像这样：

```
from mymodule import A, B
```

对后者而言，让mymodule成为一个大的源文件是最常见的。但是，这一章节展示了如何合并多个文件合并成一个单一的逻辑命名空间。 这样做的关键是创建一个包目录，使用 __init__.py 文件来将每部分粘合在一起。

当一个模块被分割，你需要特别注意交叉引用的文件名。举个例子，在这一章节中，B类需要访问A类作为基类。用包的相对导入 from .a import A 来获取。

整个章节都使用包的相对导入来避免将顶层模块名硬编码到源代码中。这使得重命名模块或者将它移动到别的位置更容易。（见10.3小节）

作为这一章节的延伸，将介绍延迟导入。如图所示，__init__.py文件一次导入所有必需的组件的。但是对于一个很大的模块，可能你只想组件在需要时被加载。 要做到这一点，__init__.py有细微的变化：

```
# __init__.py
def A():
    from .a import A
    return A()

def B():
    from .b import B
    return B()
```

在这个版本中，类A和类B被替换为在第一次访问时加载所需的类的函数。对于用户，这看起来不会有太大的不同。例如：

```
>>> import mymodule
>>> a = mymodule.A()
>>> a.spam()
A.spam
>>>
```

延迟加载的主要缺点是继承和类型检查可能会中断。你可能会稍微改变你的代码，例如:

```
if isinstance(x, mymodule.A): # Error
...

if isinstance(x, mymodule.a.A): # Ok
...
```

延迟加载的真实例子, 见标准库 multiprocessing/__init__.py 的源码.