Python的包引用机制

前言

在C语言中，我们经常写这样一句：

```
#include <stdio.h>
//....
```

这句代码引入了一个名为"stdio.h"的头函数，引入这个函数之后，我们就可以使用一些标准输入输出函数，例如printf()；可以认为在C语言中，printf()这个函数，是stdio.h这个头文件提供的功能；

python中同样有类似于“头函数”的东西，我们称之为“模块”，模块也可以被组织成更复杂结构，我们称之为包；

模块

在介绍包之前，先介绍一下模块，模块就是一个python文件，例如：

```
# 这个文件叫做 my_module.py

variable_in_my_module = 1

def my_func_in_my_module():
    print("调用了my_module中的方法")
```

```
# 这个文件叫做 main.py

import my_module

print("当前正在执行main.py")
my_module.my_func_in_my_module()
print(my_module.variable_in_my_module)
```

```
#执行结果如下
lifugui@LAPTOP-55A4PF8J:/tmp$ python3 ./main.py
当前正在执行main.py
调用了my_module中的方法
1
```

在上面这个例子中，我们在一个叫做main.py的文件中，调用了另一个叫做my_module.py的文件中的函数和变量；

如果不想引入整个模块，也可以仅引入模块中的一部分：

```
# 对main.py文件进行一些修改
from my_module import my_func_in_my_module
# from my_module import *就可以把my_module中全部的内容引用过来

print("当前正在执行main.py")
my_func_in_my_module() # 此时无需再使用 模块.方法 这种模式去引用方法，直接用方法名调用即可
print(variable_in_my_module)
```

```
# 执行结果如下
lifugui@LAPTOP-55A4PF8J:/tmp$ python3 ./main.py
当前正在执行main.py
调用了my_module中的方法 # 可见my_func_in_my_module确实被引入了main，在main中可以调用
Traceback (most recent call last): # 但是variable_in_my_module没有被引入main中
  File "./main.py", line 6, in <module>
    print(variable_in_my_module)
NameError: name 'variable_in_my_module' is not defined
```

利用form xxx import xxx的语法，我们可以仅引入模块中的一部分；

除此之外还可以对引入的模块进行重命名，例如：

```
# 对main.py文件再进行一些修改
import my_module as mm

print("当前正在执行main.py")
mm.my_func_in_my_module()
print(mm.variable_in_my_module)
```

利用上述的这些机制，我们可以将一个项目分成好几个文件写，就像C语言中所谓的模块化编程一样；这样对代码的易读性和对编码人员的分工都有很多的好处；

包

包是一种模块的组织方式，接下来我会构建一个简单的包，命名为my_pack：

这个包的结构如下：

```
lifugui@LAPTOP-55A4PF8J:/tmp/my_pack$ tree
.
├── __init__.py # __init__.py是一个文件
├── a # a是一个文件夹
|   └── a.py # a.py是一个文件（模块）
└── b # b是一个文件夹
    └── b.py # b.py是一个文件（模块）
```

根据我对包的理解，一个包可以被分成两个部分，一个部分是各种模块，另一个部分是__init__.py文件

第一部分：包中的模块

其中文件a.py的内容如下

```
# a.py
def fun_in_a():
    print("调用了a.py中的函数")
```

文件b.py的内容如下

```
# b.py
def fun_in_b():
    print("调用了b.py中的函数")
```

很明显，包里的模块是提供具体功能的，并且你可以根据自己的想法自由的组织模块的结构：可以放在根目录下面，也可以创建一个文件夹把模块放在文件夹里，这些操作都是被允许的；

在构筑包的时候应当尽量的把模块按照规则组织，这样之后在调用包的时候会减少很多痛苦....

第二部分：__init__.py文件

如果一个文件夹下存在__init__.py，那么python解释器会把这个文件夹当作包处理；

__init__.py中描述了模块的组织模式，没有这个文件的话引用包是一个很痛苦的事情，举一个例子：

例：如果my_pack的__init__.py中是空的，那么我将无法通过如下的代码调用my_pack中的函数:

```
# test.py
import my_pack

my_pack.a.fun_in_a()
```

执行结果如下：

```
lifugui@LAPTOP-55A4PF8J:/tmp/my_pack$ python3 ../test.py
Traceback (most recent call last):
  File "../test.py", line 3, in <module>
    my_pack.a.fun_in_a()
AttributeError: module 'my_pack' has no attribute 'a'
```

我们明明在my_pack下面创建了文件夹a，但解释器认为my_pack下面没有一个叫做a的东西；

也许你会好奇，如果我直接手动从my_pack中引入a，是不是就能调用a中的内容了？对不起，还是不行..

```
# test.py
from my_pack import a

a.fun_in_a()
```

```
lifugui@LAPTOP-55A4PF8J:/tmp$ python3 test.py
Traceback (most recent call last):
  File "test.py", line 4, in <module>
    a.fun_in_a()
AttributeError: module 'my_pack.a' has no attribute 'fun_in_a'
```

from my_pack import a只是从my_pack文件夹中引入了文件夹a，而解释器**还是不知道**文件夹a下面还有一个叫做a.py的模块...

在没有__init__.py正确定义的情况下，想要从my_pack中引用到a.py，正确的写法如下：

```
# test.py
from my_pack.a import a

a.fun_in_a()
```

如果包中的内容需要这样去引用的话那就太弱智了，包根本都没有存在的价值了，引用一个包里的内容比直接引用模块还繁琐，我还要包做什么？

不过幸运的是python提供了__init__.py；我们可以在这个文件中告诉解释器，这个包下面到底都有哪些资源，是以什么方式组织起来的:

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

这两行代码实际上是在描述my_pack中的模块的路径，之后，我们只需要引用my_pack就可以按照my_pack中的路径去引用方法了

```
# test.py
from my_pack

my_pack.a.fun_in_a()
# 现在这个文件变回了我们最初没有写__init__.py时举的那个例子
```

执行结果：

```
lifugui@LAPTOP-55A4PF8J:/tmp$ python3 test.py
调用了a.py中的函数
```

这次是可以调用到的