17 导入时与运行时

@(SIGAI课程录制)

概述

第一板块：Python程序的运行

- 建立对Python运行机制的基本认知
- 为第三板块内容做一下铺垫

第二板块：Python的模块与包

- 讲解本门课程最后一个概念性知识点：模块与包
- 重新认识`import`语句，知道这个语句并不像看上去那样简单
- 为第三板块内容做一下铺垫

第三板块：Python的导入时与运行时

- 能够将程序划分为导入时与运行时
- 理解`import`语句是如何模糊导入时与运行时之间的界线的
- 透过对Python导入时与运行时的理解，初步建立对Python进行性能优化的感觉

Python是怎么运行起来的

安装Python的时候到底安装的是什么?

- Python解释器
- 支持它的Built-in库文件

Python解释器是什么?

- 解释器其实也是一个软件
- 解释器可以被任何语言实现

解释器由什么组成?

- 编译器：负责将源代码编译为byte code
- 虚拟机：负责调用库文件并执行byte code

Python程序运行完整流程

- Python编译器将源代码在内存中编译为byte code
- Python虚拟机调用库文件，同时将内存中的byte code逐行解析并执行
- 运行结束后，byte code在内存中直接销毁
- 若存在`import`的脚本，则将其byte code持久化到硬盘上，成为`__pycache__`文件夹中的`.pyc`文件
- 第二次运行相同且无修改模块时，免去编译过程，直接读取`__pycache__`放入虚拟机中执行

Python的模块与包

模块是一个对象

```
>>> import time
>>> time.__class__
<class 'module'>
>>> time.__class__.__base__
<class 'object'>
```

> 说明：既然是对象，那就有在内存中构建这个对象的过程，所以，Python中的`import`语句不简单

模块有自己的作用域

编写`Temp.py`脚本：

```
x, y = 1, 2
```

通过`import`语句在交互式解释器中导入：

```
>>> import Temp
>>> Temp.x
1
>>> Temp.y
2
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

> 说明：使用`from Temp import *`语句可以解决这个问题，但通常不建议这么使用

模块只需要导入一次

修改`Temp.py`模块：

```
x, y = 1, 2
print("x = ", x)
print("y = ", y)
```

重新启动一个交互式解释器：

```
>>> import Temp
x =  1
y =  2
>>> import Temp
>>>
```

再写一个`Temp2.py`模块：

```
import Temp

print("importing Temp Module.")
print(Temp.__name__)
```

将`Temp.py`模块修改为：

```
import Temp2

x, y = 1, 2
print("x = ", x)
print("y = ", y)
```

重新启动交互式解释器：

```
>>> import Temp
importing Temp Module.
Temp
x =  1
y =  2
```

> 说明：
>
> 1. 重复使用`import`语句导入相同的模块是不起作用的
> 2. 解释器如果发现模块已经被导入，则跳过，继续向后执行
> 3. 只允许导入一次的限制，避免了写出互相导入的死循环代码
> 4. 但同时带来了修改调试代码的便利性问题

模块通常怎么用（一）：工具包

- 形式：在模块里定义一组函数或类
- 不负责处理具体问题
- 有明确的输入输出定义
- 可直接拿去使用
- 目标：让工具与业务解耦，充分复用代码

模块通常怎么用（二）：测试包

- 形式1：针对一个工具包，编写了一组单元测试
- 形式2：针对一个业务场景，编写了一组测试用例
- 形式3：使用脚本自动执行整个工程的全部测试代码，给出报告
- 一般与对应的工具包和业务代码放在一起
- 目标：确保改动代码或新增代码时不影响过去已开发功能或已修补的bug

模块通常怎么用（三）：配置包

- 形式：将整个工程中可能会改动的所有参数单独放在一起

- 目的：确保可移植；易于调参；强化解耦；

模块怎么变成包

- 模块是一个`.py`的文件
- 而包是一个文件夹，里面有很多模块，但必须再增加一个`__init__.py`
- 包里面还可以包含其他包

建一个名为`pkg`的文件夹，里面创建一个`__init__.py`

```
name = "sigai"
print("my name is ", name)
```

启动一个交互式解释器：

```
>>> import pkg
my name is  sigai
>>> pkg.name
'sigai'
```

> 说明：
>
> 1. `import pkg`语句，执行了`__init__.py`中的代码
> 2. `__init__.py`中定义的变量，自动绑定在了`pkg`上

模块有自己的预设字段

- `__file__`：在不在`sys.path`里面会影响是否是绝对路径
- `__name__`：从包的起点开始的引用关系
- `__doc__`：模块顶部的字符串说明文档
- `__cached__`：缓存对应的`.pyc`的绝对地址

> 入口模块的`__name__`自动赋值为`__main__`

如果使用`from pkg import *`，你需要知道模块的访问限制变量：`__all__`

- 解释器的导入系统会检测非包模块中以及包内的`__init__.py`文件中的`__all__`变量
- `__all__`变量是一个序列，包含一系列名称字符串
- 如果找到了`__all__`，则仅将该变量包含的名称暴露出来
- 若没有找到`__all__`，则除了`_`开头以外的所有变量名均暴露出来

在包`pkg`中的`__init__.py`文件中写入：

```
__all__ = ['x', 'add']

x = 1
y = 2
_z = 3

def add(x, y):
    return x + y
```

启动交互式解释器：

```
>>> from pkg import *
>>> x
1
>>> y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
>>> _z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_z' is not defined
>>> add(1,2)
3
```

> 说明：
>
> 1. 虽然不建议使用`from`导入模块，尤其是`from pkg import *`，但可以配合`__all__`使用
> 2. 如果不使用`from`语句而直接`import pkg`，试试看会得到什么结果，思考一下为什么？
> 3. 感兴趣的同学，可以课下自学一下`__slots__`

深入理解Python的导入时与运行时

案例：模块，类，函数，在导入时与运行时的行为

编写代码`import_and_runtime.py`：

```
print("module {} start.".format(__name__))

class ClassX():
    print("ClassX body start.")

    def __init__(self):
        print("ClassX __init__")

    def __del__(self):
        print("ClassX __del__")
```

```
    def method(self):
        print("ClassX method.")

def func_x():
    print("func_x.")

if __name__ == "__main__":
    x = ClassX()
    x.method()
    func_x()

print("module {} end.".format(__name__))
```

在交互式解释器中导入模块：

```
>>> import import_and_runtime
module import_and_runtime start.
ClassX body start.
module import_and_runtime end.
```

直接运行此模块：

```
module __main__ start.
ClassX body start.
ClassX __init__
ClassX method.
func_x.
module __main__ end.
ClassX __del__
```

> 说明：
>
> 1. 类的定义体在导入时便全部执行
> 2. 模块如果被直接运行，成为程序的入口时，`__name__`自动变为`__main__`
> 3. 对象方法或者是模块中的函数，仅仅在调用时才执行
> 4. CPython的引用计数垃圾回收算法

案例：加上装饰器，类中类，看导入时与运行时

```
print("module {} start.".format(__name__))


class ClassX():
    print("ClassX body start.")

    def __init__(self):
```

```
        print("ClassX __init__")

    def __del__(self):
        print("ClassX __del__")

    def method(self):
        print("ClassX method.")


def func_x():
    print("function func_x.")


def deco_class(cls):
    print("decorator for class.")
    print("decorate {}.".format(cls.__name__))
    return cls


def deco_func(func):
    print("decorator for function.")
    print("decorate {}.".format(func.__name__))
    return func


@deco_class
class ClassY():
    print("ClassY body start.")

    def __init__(self):
        print("ClassY __init__.")

    def __del__(self):
        print("ClassY __del__.")

    @deco_func
    def method(self):
        print("ClassY method.")

    @deco_class
    class ClassZ():
        print("ClassZ body start.")


@deco_func
def func_y():
    print("function func_y.")
```

```
if __name__ == "__main__":
    x = ClassX()
    x.method()
    func_x()
    y = ClassY()
    y.method()
    func_y()

print("module {} end.".format(__name__))
```

在交互式解释器中导入模块：

```
>>> import import_and_runtime
module import_and_runtime start.
ClassX body start.
ClassY body start.
decorator for function.
decorate method.
ClassZ body start.
decorator for class.
decorate ClassZ.
decorator for class.
decorate ClassY.
decorator for function.
decorate func_y.
module import_and_runtime end.
```

直接运行整个脚本：

```
module __main__ start.
ClassX body start.
ClassY body start.
decorator for function.
decorate method.
ClassZ body start.
decorator for class.
decorate ClassZ.
decorator for class.
decorate ClassY.
decorator for function.
decorate func_y.
ClassX __init__
ClassX method.
function func_x.
ClassY __init__.
ClassY method.
function func_y.
module __main__ end.
```

```
ClassX __del__
ClassY __del__.
```

> 说明：
>
> 1. 再次重申，装饰器在导入时立即执行
> 2. 在导入时，解释器**按顺序**从前向后解析
> 3. 遇到类，先执行类定义体，等整个类定义体执行完毕后，最后执行类装饰器
> 4. 类定义体中如果有方法被装饰，立即执行方法装饰器
> 5. 类定义体中如果有其他类的定义体，继续执行该类的定义体
> 6. 注意：函数的装饰器，优先级比函数高，而类装饰器则相反
> 7. 由于CPython的引用计数的垃圾回收算法，占用较大内存的类与对象太多会导致资源浪费

如果没有`if __name__ == "__main__":`会发生什么?

在原来的模块中，插入一段代码：

```
......
"此处省略之前重复的代码"

if __name__ == "__main__":
    x = ClassX()
    x.method()
    func_x()
    y = ClassY()
    y.method()
    func_y()

func_x()
func_y()

yy = ClassY()
xx = ClassX()

xx.method()
yy.method()

print("module {} end.".format(__name__))
```

在交互式解释器中导入模块：

```
>>> import import_and_runtime
module import_and_runtime start.
ClassX body start.
ClassY body start.
decorator for function.
decorate method.
ClassZ body start.
decorator for class.
```

```
decorate ClassZ.
decorator for class.
decorate ClassY.
decorator for function.
decorate func_y.
function func_x.
function func_y.
ClassY __init__.
ClassX __init__
ClassX method.
ClassY method.
module import_and_runtime end.
>>> exit()
ClassY __del__.
ClassX __del__
```

> 说明：
>
> 1. 如果不对`__name__`进行判断，那么代码在`import`时便开始运行
> 2. 因此，如果一个`.py`文件的使用场景是被`import`，那么其中的代码应当仅仅定义，而不执行
> 3. 在交互式解释器中导入的模块，如果导入时对类进行了实例化，那么默认情况下在退出后才销毁
> 4. `import`语句其实很可怕，除了运行时副作用，性能问题，还可能遭遇模块注入攻击

总结

- Python的简洁，代价是让很多看上去稀松平常的关键字，内部做了大量复杂的工作（`for import with`等）
- 不清楚这些背后机理的情况下去使用Python，一定无法写出工程级别的代码，最多写出一些能运行的Demo
- 透过Python的导入时与运行时的行为，你会发现，在某些情况下，定义类并不比仅定义函数要好（开销不同）
- 学完本节课，你可能就明白为什么很多网上的Python代码都只有一个`main.py`了（`import`有坑）