第十四章 执行环境

Python

可调用对象

Python 有 4 种可调用对象： **函数， 方法，类**， 以及一些**类的实例。**

所说的可调用的， 即是任何能通过函数操作符 `()` 来调用的对象。要调用可调用对象，函数操作符得紧跟在可调用对象之后。比方说，用“foo()”来调用函数”foo”。

14.1.1 函数

1. 内建函数-BIFs
   BIFs的属性有 `__doc__,__name__,__self__,__module__`

这些函数在`_bulitin_`模块里，并作为`__builtins__`模
块导入到解释器中。

2. 用户自定义函数-UDF
   定义在模块的最高级，因此会作为**全局名字空间的一部分**(一旦创建好内建名字空间）装载到系统中。

UDF的函数属性
`udf.__doc__,udf.__name__,udf.func_*`

一旦函数声明以后（且函数对象可用），程序员也可以自定义函数属性。所有的新属性变成`udf.__dict__`对象的一部分。

14.1.2 方法

许多 python 数据类型，比如列表和字典，也有方法，这些被称为内建方法。

内建方法
内建方法BIM属性
`bim.__doc__`
`bim.__name__`–字符串类型的函数名字
`bim.__self__`–绑定的对象

只有内建类型(BIT)有 BIM.

对于内建方法，type()工厂函数给出了和 BIF 相同的输出

```
type([].append) # <type 'builtin_function_or_method'>
```

BIM 和 BIF 两者也都享有相同属性。不同之处在于 BIM 的**self**属性指向一个 Python对象，而 BIF 指向 None。

用户定义的方法UDM
包含在类定义之中， 只是拥有标准函数的包装。

14.1.3 类

利用类的可调用性来创建实例。“调用”类的结果便是创建了实例，即大家所知道的实例化。

14.1.4 类的实例

python 给类提供了名为 `__call__` 的特别方法，该方法允许程序员创建可调用的对象（实例）。
默认是没有实现的。在类定义中覆盖这个方法，那么这个类实例就成为可调用的了。

调用这样的实例对象等同于调用 `__call__()` 方法。
`foo()` 就和 `foo.__call__(foo)` 的效果相同

`foo(arg)` 就和调用 `foo.__call__(foo, arg)` 一样。

代码对象

python 语句， 赋值，表达式，甚至还有模块这些代码块称为代码对象。

如果要执行 python 代码，那么该代码必须先要转换成字节编译的代码（又称字节码）。

14.3 可执行的对象声明和内建函数

Python 提供了大量的 BIF 来支持可调用/可执行对象，其中包括 exec 语句。
这些函数帮助程序员执行代码对象，也可以用内建函数 `complie()` 来生成代码对象。

可执行对象和内建函数
`callable()`–确定一个对象是否可以通过函数调用符(`()`)来调用。
`compile()`–在运行时生成代码对象，然后直接使用 `exec` 或者 `eval` 来进行求值。该函数一次性字节码预编译，以后都不用编译了
`compile(string,file, type)` 最后的参数是个字符串，它表明代码的类型
’eval‘—-可求值得表达式，和eval()一起使用
’single‘—-单一执行语句，和exec一起使用
’exec‘—-可执行的与剧组，和exec一起使用
`eval()`–对表达式求值。可以这样理解 eval()函数的工作方式：对表达式两端的引号视而不见
`exec()`–执行代码对象或字符串形式的Python代码。
`input(prompt='')`–等价于 `eval(raw_input())`，`input()` 把输入作为 python 对象来求值并返回表达式的结果。

可以这样理解 `eval()` 函数的工作方式：对表达式两端
的引号视而不见，

执行其他Python程序

只有属于模块最高级的代码才是全局变量，全局类，和全局函数声明。

> 导入模块的副作用是导致最高级代码运行。

当模块导入后，就执行所有的模块，运行所有最高级别的(即没有缩进的)Python代码。
`execfile(filename, globals=globals(), locals=locals())`

14.5 执行其他（非 Python）程序

针对不同的环境，python 提供了各种执行非 python 程序的方法。
所有的函数都可以在os模块中找到。
`system(cmd)`
`fork()`–通常和`exec*()`一起使用
`execl(file,arg0,arf1,...)`–用参数列表`arg0`，`arg1`等等执行文件
`execv(file,arflist)`
`execle(file,arg0,arg1,...,env)`–提供环境变量字典
`execve(file,arglist,env)`
`execlp(cmd, arg0,arg1,...)`–在用户的搜索路径下搜索完全的文件路径名
`execvp(cmd,arglist)`
`execlpe(cmd, arg0, arg1,... env)`
`execvpe(cmd,arglist,env)`
`spawn*a(mode, file, args[, env])`–在新进程中执行
`popen(cmd, mode='r',buffering=-1)`–执行cmd命令，将结果作为文件对象，默认为读取模式。

当子进程返回的时候， 其返回值永远是 0； 当父进程返回时， 其返回值永远是子进程的进程标识符 PID

```
ret = os.fork() # spawn 2 processes, both return #产生两个进程，都返回
if ret == 0: # child returns with PID of 0 #子进程返回的 PID 是 0
    child_suite # child code #子进程的代码
else: # parent returns with child's PID #父进程返回是子进程的 PID
    parent_suite # parent code #父进程的代码
```

在Python2.4版本后在 ,subprocess 作为面向进程函数的模块。

随着越来越接近软件的操作系统层面，你就会发现执行跨平台程序（甚至是 python 脚本）的一致性开始有些不确定了。上面我们提到在这个小节中描述的程序在 os 模块中。事实上，有多个 os模块。

基于 Unix 衍生系统(Linux,MacOS X, Solaris,BSD 等等)的模块是 posix 模块，windows 的是 nt(无论你现在用的是哪个版本的 windows;dos 用户有 dos 模块)

结束执行

`sys.exit(status=0)`
`os._exit()`

14.9 相关模块 os,sys

`os`,`sys`