

python 高级

1.GIL（全局解释器锁）

GIL 面试题如下

描述 Python GIL 的概念，以及它对 python 多线程的影响？编写一个多线程抓取网页的程序，并阐明多线程抓取程序是否可比单线程性能有提升，并解释原因。

Guido 的声明：

<http://www.artima.com/forums/flat.jsp?forum=106&thread=214235>

he language doesn't require the GIL -- it's only the CPython virtual machine that has historically been unable to shed it.

参考答案：

1. Python 语言和 GIL 没有半毛钱关系。仅仅是由于历史原因在 Cpython 虚拟机(解释器)，难以移除 GIL。
2. GIL：全局解释器锁。每个线程在执行的过程都需要先获取 GIL，保证同一时刻只有一个线程可以执行代码。
3. 线程释放 GIL 锁的情况：在 IO 操作等可能会引起阻塞的 system call 之前,可以暂时释放 GIL,但在执行完毕后,必须重新获取 GIL Python 3.x 使用计时器（执行时间达到阈值后，当前线程释放 GIL）或 Python 2.x，tickets 计数达到 100
4. Python 使用多进程是可以利用多核的 CPU 资源的。
5. 多线程爬取比单线程性能有提升，因为遇到 IO 阻塞会自动释放 GIL 锁

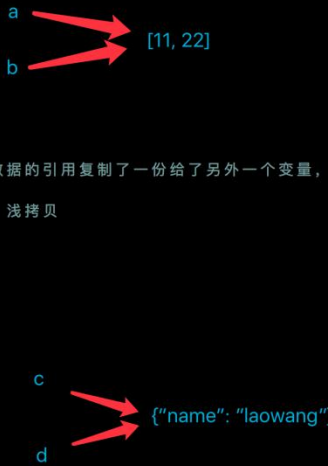
2. 深拷贝、浅拷贝

1. 浅拷贝

- 浅拷贝是对于一个对象的顶层拷贝

通俗的理解是：拷贝了引用，并没有拷贝内容

```
In [1]: a = [11, 22]
In [2]: b = a
In [3]: id(a) # 用来显示a指向的数据的内存地址
Out[3]: 4399944520
In [4]: id(b) # 用来显示b指向的数据的内存地址
Out[4]: 4399944520
In [5]: # 以上结果相同，说明了当给一个变量赋值时，其实就是将数据的引用复制了一份给了另外一个变量，这其实就是最简单的浅拷贝
In [6]: # 不仅列表是这样只要是 类似于 xx1 = xx2 的这种基本都是 浅拷贝
In [7]:
In [7]: c = {"name": "laowang"}
In [8]: d = c
In [9]: id(c)
Out[9]: 4396398704
In [10]: id(d)
Out[10]: 4396398704
In [11]:
In [11]: c["passwd"] = "123456"
In [12]: c
Out[12]: {'name': 'laowang', 'passwd': '123456'}
In [13]: d
Out[13]: {'name': 'laowang', 'passwd': '123456'}
In [14]: # 因为都是浅拷贝，所以只要通过一个引用进行了修改，那么另外一个变量就看到的也就变化了
```



```
In [9]: a = [11,22]
In [10]: b = [33,44]
In [11]: c = [a,b]
In [12]: id(a)
Out[12]: 4395286984
In [13]: id(b)
Out[13]: 4398005384
In [14]: id(c)
Out[14]: 4395392520
In [15]: import copy
In [16]: d = copy.copy(c)
In [17]: id(d)
Out[17]: 4395293000
In [18]: id(d[0])
Out[18]: 4395286984
In [19]: id(d[1])
Out[19]: 4398005384
In [20]: a.append(11)
In [21]: c
Out[21]: [[11, 22, 11], [33, 44]]
In [22]: d
Out[22]: [[11, 22, 11], [33, 44]]
```

浅copy，只会复制最顶层的那个列表

2. 深拷贝

- 深拷贝是对于一个对象所有层次的拷贝(递归)

```
In [16]: import copy
In [17]: a = [11, 22]
In [18]: b = copy.deepcopy(a) # 对a指向的列表进行深copy
In [19]: a
Out[19]: [11, 22]
In [20]: b
Out[20]: [11, 22]
In [21]: id(a)
Out[21]: 4399904456
In [22]: id(b)
Out[22]: 4400396296
In [23]: # 以上结果说明了通过deepcopy确实将a列表中所有的数据的引用copy了，而不是只copy了a指向的列表的引用
In [24]:
In [24]: a.append(33)
In [25]: a
Out[25]: [11, 22, 33]
In [26]: b
Out[26]: [11, 22]
In [27]:
```

The diagram illustrates the state of variables 'a' and 'b' after the execution of the code. Variable 'a' is shown with a red arrow pointing to the list [11, 22]. Variable 'b' is shown with a red arrow pointing to the list [11, 22, 33]. This visualizes that 'b' is a deep copy of 'a' and contains the additional element 33 that was added to 'a' in the subsequent code block.

进一步理解深拷贝

```
In [32]: import copy
```

```
In [33]: a = [11, 22]
```

```
In [34]: b = [33, 44]
```

```
In [35]: c = [a, b]
```

```
In [36]: d = copy.deepcopy(c)
```

```
In [37]: id(a)
```

```
Out[37]: 4400357768
```

```
In [38]: id(b)
```

```
Out[38]: 4399676872
```

```
In [39]: id(c)
```

```
Out[39]: 4400148872
```

```
In [40]: id(c[0])
```

```
Out[40]: 4400357768
```

```
In [41]: id(c[1])
```

```
Out[41]: 4399676872
```

```
In [42]:
```

```
In [42]: id(d[0])
```

```
Out[42]: 4400003144
```

```
In [43]: id(d[1])
```

```
Out[43]: 4400139016
```

```
In [44]: id(d)
```

```
Out[44]: 4400089544
```

```
In [45]:
```

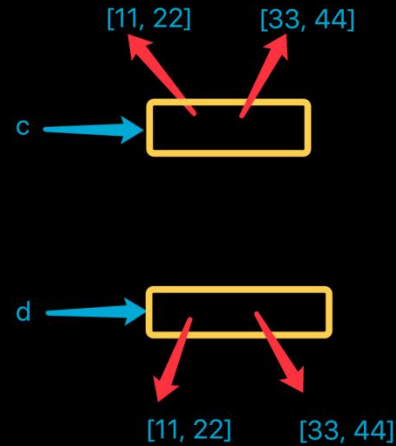
```
In [45]: c[0].append(55)
```

```
In [46]: c
```

```
Out[46]: [[11, 22, 55], [33, 44]]
```

```
In [47]: d
```

```
Out[47]: [[11, 22], [33, 44]]
```



3. 拷贝的其他方式

- 分片表达式可以赋值一个序列

```
In [13]: a = [11, 22]
```

```
In [14]: b = [33, 44]
```

```
In [15]: c = [a, b]
```

```
In [16]: d = c[:]
```

```
In [17]: id(c)
```

```
Out[17]: 4500666120
```

```
In [18]: id(d)
```

```
Out[18]: 4499927112
```

```
In [19]: id(c[0])
```

```
Out[19]: 4499881672
```

```
In [20]: id(d[0])
```

```
Out[20]: 4499881672
```

```
In [21]: a
```

```
Out[21]: [11, 22]
```

```
In [22]: a.append(33)
```

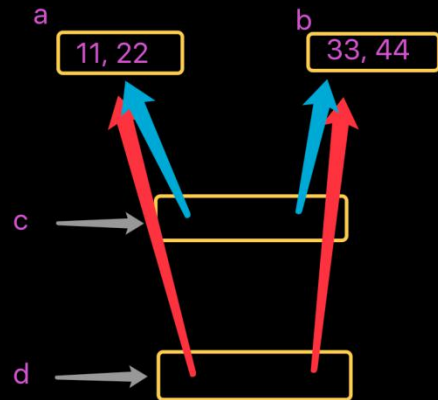
```
In [23]: c
```

```
Out[23]: [[11, 22, 33], [33, 44]]
```

```
In [24]: d
```

```
Out[24]: [[11, 22, 33], [33, 44]]
```

```
In [25]:
```



d = c[:] 与 d=copy.copy(c)一样 属于浅copy

- 字典的 copy 方法可以拷贝一个字典

```

In [62]: d = dict(name="zhangsan", age=27)
In [63]: co = d.copy()
In [64]: id(d)
Out[64]: 4397681184
In [65]: id(co)
Out[65]: 4378467208
In [66]: d
Out[66]: {'age': 27, 'name': 'zhangsan'}
In [67]: co
Out[67]: {'age': 27, 'name': 'zhangsan'}
In [68]:
In [68]: d = dict(name="zhangsan", age=27, children_ages = [11, 22])
In [69]: co = d.copy()
In [70]: d["children_ages"].append(9)
In [71]: d
Out[71]: {'age': 27, 'children_ages': [11, 22, 9], 'name': 'zhangsan'}
In [72]: co
Out[72]: {'age': 27, 'children_ages': [11, 22, 9], 'name': 'zhangsan'}
In [73]:

```

```

d=dict(name='zhangsan', age=20)
d={'name': 'zhangsan', 'age': 20}
哪一种效率更高

```

4. 注意点

浅拷贝对不可变类型和可变类型的 copy 不同

1. copy.copy 对于可变类型，会进行浅拷贝
2. copy.copy 对于不可变类型，不会拷贝，仅仅是指向

```

In [88]: a = [11, 22, 33]
In [89]: b = copy.copy(a)
In [90]: id(a)
Out[90]: 59275144
In [91]: id(b)
Out[91]: 59525600
In [92]: a.append(44)
In [93]: a
Out[93]: [11, 22, 33, 44]

```

```
In [94]: b
Out[94]: [11, 22, 33]
```

```
In [95]: a = (11,22,33)
In [96]: b = copy.copy(a)
In [97]: id(a)
Out[97]: 58890680
In [98]: id(b)
Out[98]: 58890680
```

```
In [24]: a = [11,22]
In [25]: b = [33,44]
In [26]: c = (a,b)
In [27]: d = copy.copy(c)
In [28]: id(c)
Out[28]: 4398226568
In [29]: id(d)
Out[29]: 4398226568
In [30]: a.append(33)
In [31]: c
Out[31]: ([11, 22, 33], [33, 44])
In [32]: d
Out[32]: ([11, 22, 33], [33, 44])
In [33]:
In [33]: e = copy.deepcopy(c)
In [34]: id(c)
Out[34]: 4398226568
In [35]: id(e)
Out[35]: 4399065544
In [36]: a.append(44)
In [37]: c
Out[37]: ([11, 22, 33, 44], [33, 44])
In [38]: e
Out[38]: ([11, 22, 33], [33, 44])
In [39]:
```

如果c是元组，那么 copy时会，仅仅是元组的引用copy
而deepcopy依然是深copy，即递归copy所有

copy.copy 和 copy.deepcopy 的区别

copy.copy


```
In [81]: a = [11, 22]
```

```
In [82]: b = (a, )
```

```
In [83]: c = [b,]
```

```
In [84]:
```

```
In [84]: d = copy.copy(c)
```

```
In [85]:
```

```
In [85]: c
```

```
Out[85]: [(11, 22),]
```

```
In [86]: d
```

```
Out[86]: [(11, 22),]
```

```
In [87]: a.append(33)
```

```
In [88]: c
```

```
Out[88]: [(11, 22, 33),]
```

```
In [89]: d
```

```
Out[89]: [(11, 22, 33),]
```

```
In [90]: id(c)
```

```
Out[90]: 4400371720
```

```
In [91]: id(d)
```

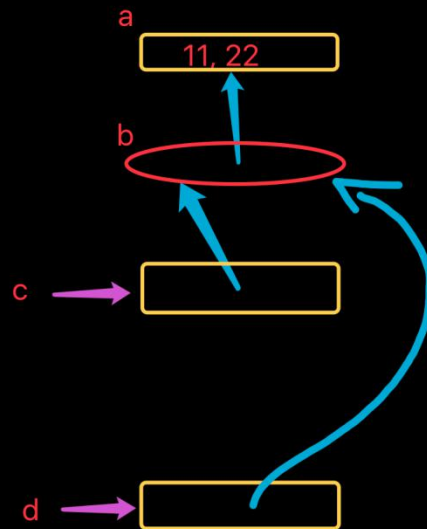
```
Out[91]: 4400343688
```

```
In [92]: id(c[0])
```

```
Out[92]: 4400563256
```

```
In [93]: id(d[0])
```

```
Out[93]: 4400563256
```



`d = c` # 让d这个变量指向c指向的空间
`d = copy.copy(c)` # 复制所有c指向的数据到一个新空间，但是不会递归copy

```
In [98]: a = [11, 22]
```

```
In [99]: b = [a]
```

```
In [100]: c = [b]
```

```
In [101]:
```

```
In [101]: d = copy.copy(c)
```

```
In [102]:
```

```
In [102]: c
```

```
Out[102]: [[[11, 22]]]
```

```
In [103]: d
```

```
Out[103]: [[[11, 22]]]
```

```
In [104]: id(c)
```

```
Out[104]: 4395608584
```

```
In [105]: id(d)
```

```
Out[105]: 4400474632
```

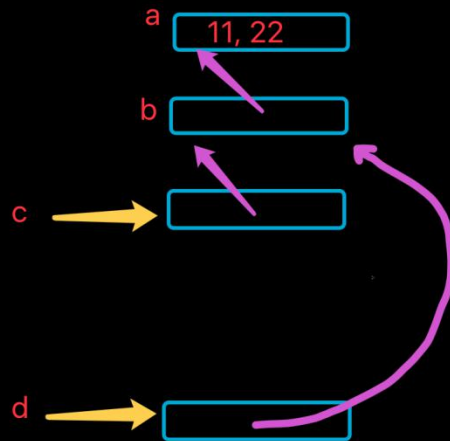
```
In [106]: id(c[0])
```

```
Out[106]: 4400474824
```

```
In [107]: id(d[0])
```

```
Out[107]: 4400474824
```

```
In [108]:
```



copy.deepcopy

```
In [121]: a = [11, 22]

In [122]: b = [a]

In [123]: c = [b]

In [124]:

In [124]: d = copy.deepcopy(c)

In [125]:

In [125]: c
Out[125]: [[11, 22]]

In [126]: d
Out[126]: [[11, 22]]

In [127]: id(c)
Out[127]: 4399971144

In [128]: id(d)
Out[128]: 4399853064

In [129]: id(c[0])
Out[129]: 4400242312

In [130]: id(d[0])
Out[130]: 4400473736

In [132]: a
Out[132]: [11, 22]

In [133]: a.append(33)
Out[133]: [11, 22, 33]

In [134]: a
Out[134]: [11, 22, 33]

In [135]: c
Out[135]: [[11, 22, 33]]

In [136]: d
Out[136]: [[11, 22]]

In [137]: []
```

`d = copy.deepcopy(c)`
会将c指向的空间进行递归copy

```
In [138]: a = [11, 22]
```

```
In [139]: b = (a,)
```

```
In [140]: c = [b]
```

```
In [141]:
```

```
In [141]: d = copy.deepcopy(c)
```

```
In [142]: c
```

```
Out[142]: [(11, 22),]
```

```
In [143]: d
```

```
Out[143]: [(11, 22),]
```

```
In [144]: id(c)
```

```
Out[144]: 4399912328
```

```
In [145]: id(d)
```

```
Out[145]: 4400358024
```

```
In [146]: id(c[0])
```

```
Out[146]: 4399871536
```

```
In [147]: id(d[0])
```

```
Out[147]: 4399936344
```

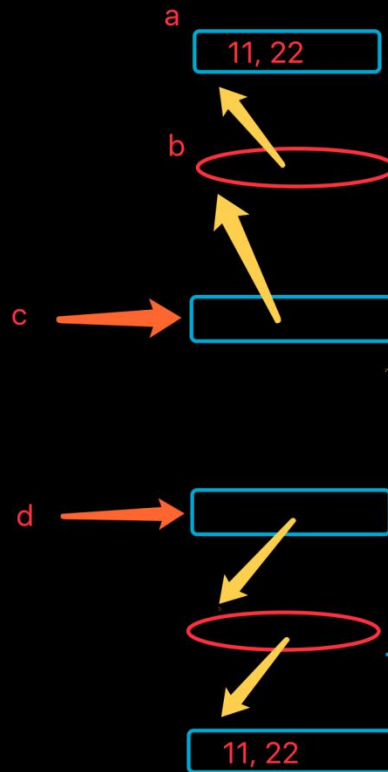
```
In [148]: a.append(33)
```

```
In [149]: c
```

```
Out[149]: [(11, 22, 33),]
```

```
In [150]: d
```

```
Out[150]: [(11, 22),]
```



3.私有化

- `xx`: 公有变量
- `_x`: 单前置下划线,私有化属性或方法, `from somemodule import *`禁止导入,类对象和子类可以访问
- `__xx`: 双前置下划线,避免与子类中的属性命名冲突,无法在外部直接访问(名字重整所以访问不到)
- `__xx__`: 双前后下划线,用户名字空间的魔法对象或属性。例如: `__init__`, `__` 不要自己发明这样的名字
- `xx_`: 单后置下划线,用于避免与 Python 关键词的冲突 (班里有个同学爱用)

通过 name mangling (名字重整(目的就是以防子类意外重写基类的方法或者属性)如: `_Class_object`) 机制就可以访问 private 了。

```
#coding=utf-8
```

```
class Person(object):
    def __init__(self, name, age, taste):
        self.name = name
        self._age = age
        self.__taste = taste

    def showperson(self):
        print(self.name)
        print(self._age)
        print(self.__taste)

    def dowork(self):
        self._work()
        self.__away()

    def _work(self):
        print('my _work')

    def __away(self):
        print('my __away')

class Student(Person):
    def construction(self, name, age, taste):
        self.name = name
        self._age = age
        self.__taste = taste

    def showstudent(self):
        print(self.name)
```

```
        print(self._age)
        print(self.__taste)

    @staticmethod
    def testbug():
        _Bug.showbug()

# 模块内可以访问, 当 from cur_module import *时, 不导入
class _Bug(object):
    @staticmethod
    def showbug():
        print("showbug")

s1 = Student('jack', 25, 'football')
s1.showperson()
print('*'*20)

# 无法访问__taste, 导致报错
# s1.showstudent()
s1.construction('rose', 30, 'basketball')
s1.showperson()
print('*'*20)

s1.showstudent()
print('*'*20)

Student.testbug()
```

```
python@ubuntu:~/workspace/test$ python t10.py
jack
25
football
*****
rose
30
football
*****
rose
30
basketball
*****
showbug
python@ubuntu:~/workspace/test$
```

总结

- 父类中属性名为__名字的, 子类不继承, 子类不能访问

- 如果在子类中向__名字赋值，那么会在子类中定义的一个与父类相同名字的属性
- __名的变量、函数、类在使用 `from xxx import *` 时都不会被导入

4.import 导入模块

1. import 搜索路径

```
In [152]: import sys

In [153]: sys.path
Out[153]:
['',
 '/usr/local/bin',
 '/usr/local/Cellar/python3/3.6.0/Frameworks/Python.framework/Versions/3.6/lib/python3.6.zip',
 '/usr/local/Cellar/python3/3.6.0/Frameworks/Python.framework/Versions/3.6/lib/python3.6',
 '/usr/local/Cellar/python3/3.6.0/Frameworks/Python.framework/Versions/3.6/lib/python3.6/lib-dynload',
 '/usr/local/lib/python3.6/site-packages',
 '/usr/local/lib/python3.6/site-packages/pytesseract-0.1.6-py3.6.egg',
 '/usr/local/lib/python3.6/site-packages/easytrader-0.11.17-py3.6.egg',
 '/usr/local/lib/python3.6/site-packages/rqopen_client-0.0.4-py3.6.egg',
 '/usr/local/lib/python3.6/site-packages/PyOpenGL-3.1.1a1-py3.6.egg',
 '/usr/local/lib/python3.6/site-packages/progressbar-2.3-py3.6.egg',
 '/usr/local/lib/python3.6/site-packages/IPython/extensions',
 '/Users/dongge/.ipython']

In [154]:
```

路径搜索

- 从上面列出的目录里依次查找要导入的模块文件
- " 表示当前路径
- 列表中的路径的先后顺序代表了 python 解释器在搜索模块时的先后顺序

程序执行时添加新的模块路径

```
sys.path.append('/home/wangdao/xxx')
sys.path.insert(0, '/home/wangdao/xxx') # 可以确保先搜索这个路径
```

```
In [37]: sys.path.insert(0, "/home/python/xxxx")
In [38]: sys.path
Out[38]:
['/home/python/xxxx',
 '',
 '/usr/bin',
 '/usr/lib/python3.5.zip',
 '/usr/lib/python3.5',
 '/usr/lib/python3.5/plat-x86_64-linux-gnu',
 '/usr/lib/python3.5/lib-dynload',
 '/usr/local/lib/python3.5/dist-packages',
 '/usr/lib/python3/dist-packages',
 '/usr/lib/python3/dist-packages/IPython/extensions',
 '/home/python/.ipython']
```

2. 重新导入模块

模块被导入后，`import module` 不能重新导入模块，重新导入需用 `reload`

```
1 def test():
2     print("---1---")
```

```
In [1]: import reload_test
```

```
In [2]: reload_test.test()
---1---
```

```
In [3]:
```

```
1 def test():
2     print("---2---")
```

```
In [3]: # 重新开一个新终端，在新终端中修改 reload_test 模块的代码
```

```
In [4]: # 然后再次 import
```

```
In [5]: import reload_test
```

```
In [6]: reload_test.test()
---1---
```

```
In [7]: # import 导入模块只会导入一次，因此即使模块进行了修改，import 也不会重新导入
```

```
In [8]: # 如果用 reload 则可以重新导入
```

```
In [9]: from imp import reload
```

```
In [10]: reload(reload_test)
```

```
Out[10]: <module 'reload_test' from '/Users/dongge/Desktop/reload_test.py'>
```

```
In [11]: reload_test.test()
---2---
```

```
In [12]:
```

3. 多模块开发时的注意点

`recv_msg.py` 模块

```
from common import RECV_DATA_LIST
# from common import HANDLE_FLAG
import common
```

```
def recv_msg():
```



```
"""模拟接收到数据，然后添加到 common 模块中的列表中"""
print("--->recv_msg")
for i in range(5):
    RECV_DATA_LIST.append(i)

def test_recv_data():
    """测试接收到的数据"""
    print("--->test_recv_data")
    print(RECV_DATA_LIST)

def recv_msg_next():
    """已经处理完成后，再接收另外的其他数据"""
    print("--->recv_msg_next")
    # if HANDLE_FLAG:
    if common.HANDLE_FLAG:
        print("-----发现之前的数据已经处理完成，这里进行接收其他的数据(模拟过程...)----")
    else:
        print("-----发现之前的数据未处理完，等待中....-----")

handle_msg.py 模块

from common import RECV_DATA_LIST
# from common import HANDLE_FLAG
import common

def handle_data():
    """模拟处理 recv_msg 模块接收的数据"""
    print("--->handle_data")
    for i in RECV_DATA_LIST:
        print(i)

    # 既然处理完成了，那么将变量 HANDLE_FLAG 设置为 True，意味着处理完成
    # global HANDLE_FLAG
    # HANDLE_FLAG = True
    common.HANDLE_FLAG = True

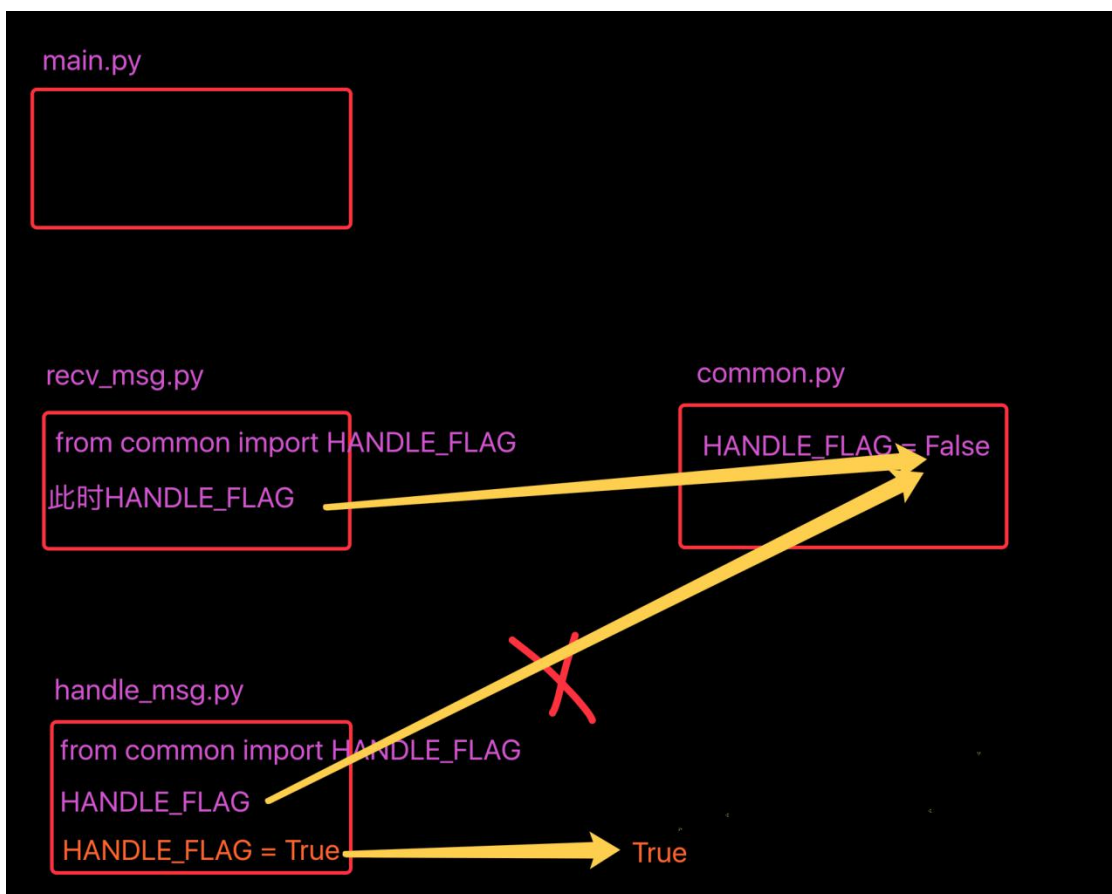
def test_handle_data():
    """测试处理是否完成，变量是否设置为 True"""
    print("--->test_handle_data")
    # if HANDLE_FLAG:
    if common.HANDLE_FLAG:
        print("=====已经处理完成=====")
    else:
        print("=====未处理完成=====")
```

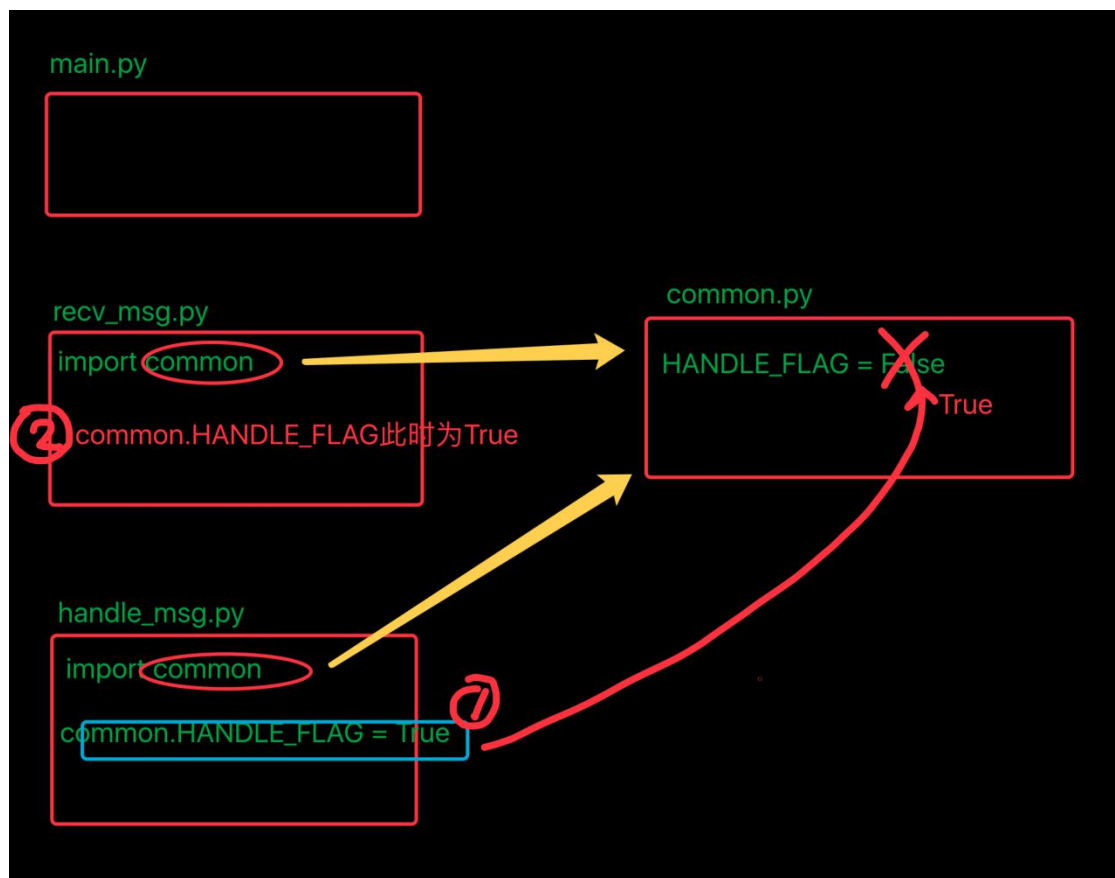
main.py 模块

```
from recv_msg import *
from handle_msg import *

def main():
    # 1. 接收数据
    recv_msg()
    # 2. 测试是否接收完毕
    test_recv_data()
    # 3. 判断如果处理完成，则接收其它数据
    recv_msg_next()
    # 4. 处理数据
    handle_data()
    # 5. 测试是否处理完毕
    test_handle_data()
    # 6. 判断如果处理完成，则接收其它数据
    recv_msg_next()

if __name__ == "__main__":
    main()
```

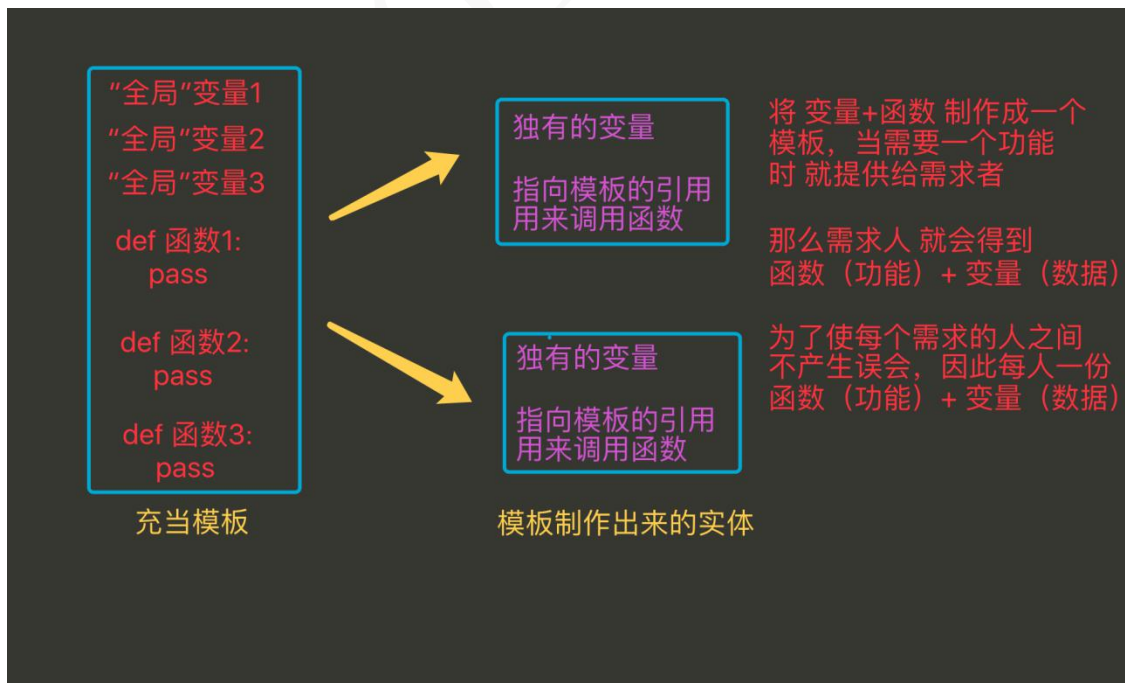
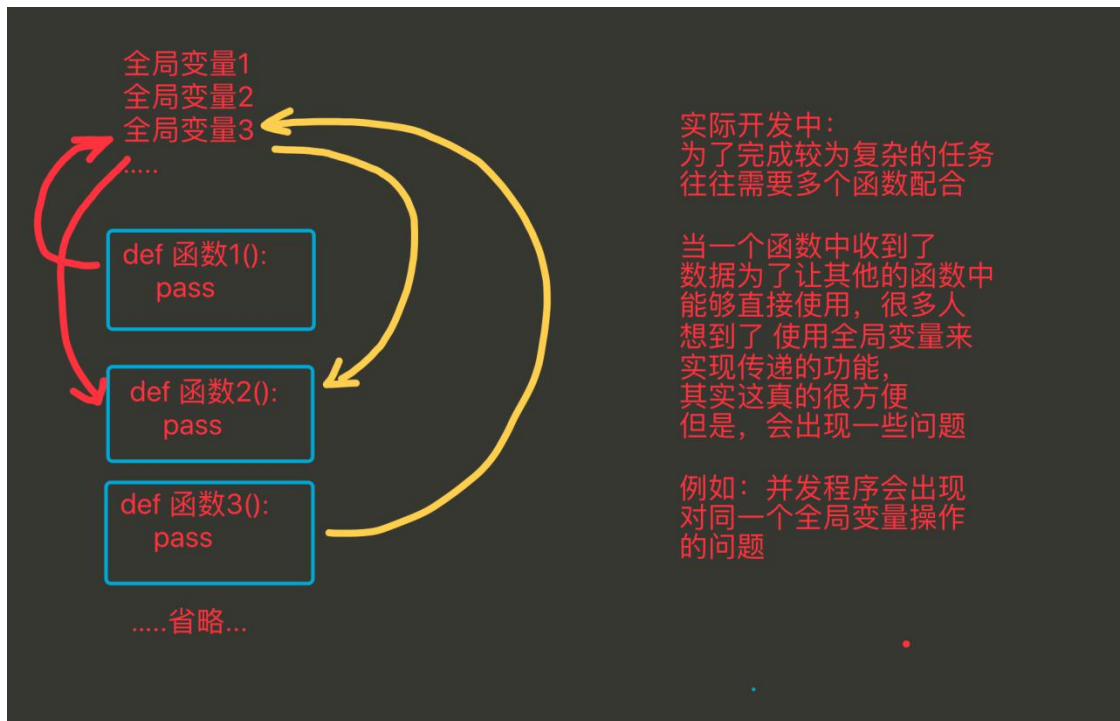




5.再议 封装、继承、多态

封装、继承、多态 是面向对象的 3 大特性

为啥要封装



好处

1. 在使用面向过程编程时，当需要对数据处理时，需要考虑用哪个模板中哪个函数来进行操作，但是当用面向对象编程时，因为已经将数据存储到了这个独立的空间中，这个独立的空间（即对象）中通过一个特殊的变量（`__class__`）能够获取到类（模板），而且这个类中的方法是有一定数量的，与此类无关的将不会出现在本类中，因此需要对数据处理时，可以很快的定位到需要的方法是谁 这样更方便
2. 全局变量是只能有 1 份的，多很多个函数需要多个备份时，往往需要利用其它的变量来进行储存；而通过封装 会将用来存储数据的这个变量 变为了对象中的一个“全局”变量，只要对象不一样那么这个变量就可以再有 1 份，所以这样更方便
3. 代码划分更清晰

面向过程

全局变量 1

全局变量 2

全局变量 3

...

```
def 函数 1():  
    pass
```

```
def 函数 2():  
    pass
```

```
def 函数 3():  
    pass
```

```
def 函数 4():  
    pass
```

```
def 函数 5():  
    pass
```

面向对象

```
class 类(object):  
    属性 1  
    属性 2
```

```
    def 方法 1(self):  
        pass
```

```
def 方法 2(self):  
    pass  
  
class 类 2(object):  
    属性 3  
    def 方法 3(self):  
        pass  
  
    def 方法 4(self):  
        pass  
  
    def 方法 5(self):  
        pass
```

为啥要继承

这是一个类有 很多代码
类1

这里要对类1进行功能的扩充
多某些方法进行适当修改

方式1: 重新自己写一个 新版本
方式2: 在类1的基础上修改

你会选择哪种???

说明

1. 能够提升代码的重用率，即开发一个类，可以在多个子功能中直接使用
2. 继承能够有效的进行代码的管理，当某个类有问题只要修改这个类就行，而其继承这个类的子类往往不需要就修改

怎样理解多态

```
class MiniOS(object):
    """MiniOS 操作系统类 """
    def __init__(self, name):
        self.name = name
        self.apps = [] # 安装的应用程序名称列表

    def __str__(self):
        return "%s 安装的软件列表为 %s" % (self.name, str(self.apps))

    def install_app(self, app):
        # 判断是否已经安装了软件
        if app.name in self.apps:
            print("已经安装了 %s, 无需再次安装" % app.name)
        else:
            app.install()
            self.apps.append(app.name)

class App(object):
    def __init__(self, name, version, desc):
        self.name = name
        self.version = version
        self.desc = desc

    def __str__(self):
        return "%s 的当前版本是 %s - %s" % (self.name, self.version, self.desc)

    def install(self):
        print("将 %s [%s] 的执行程序复制到程序目录..." % (self.name, self.version))

class PyCharm(App):
    pass

class Chrome(App):
    def install(self):
        print("正在解压缩安装程序...")
        super().install()

linux = MiniOS("Linux")
print(linux)
```



```
pycharm = PyCharm("PyCharm", "1.0", "python 开发的 IDE 环境")
chrome = Chrome("Chrome", "2.0", "谷歌浏览器")

linux.install_app(pycharm)
linux.install_app(chrome)
linux.install_app(chrome)

print(linux)
```

运行结果

Linux 安装的软件列表为 []
将 PyCharm [1.0] 的执行程序复制到程序目录...
正在解压缩安装程序...
将 Chrome [2.0] 的执行程序复制到程序目录...
已经安装了 Chrome, 无需再次安装
Linux 安装的软件列表为 ['PyCharm', 'Chrome']

工厂模式

打开 python 开发工具 IDE, 新建 'factory.py' 文件, 新建一个基类代码如下:

```
class animal():
    def eat(self):
        pass
    def voice(self):
        pass

class dog(animal):
    def eat(self):
        print ('狗吃骨头')
    def voice(self):
        print ('狗叫汪汪')
```

dog 类继承 animal 类, 复写了 eat 和 voice 方法

```
class cat(animal):
    def eat(self):
```

```
print ('猫吃鱼')
def voice(self):
    print ('猫叫喵喵')
```

cat 类继承 animal 类，复写了 eat 和 voice 方法

接在在 'factory.py' 文件，编写代码，创建工厂类，编写创建函数：

```
class factoryAni:
    def createAni(self, aniType):
        if aniType == 'dog':
            return dog()
        if aniType == 'cat':
            return cat()
```

createAni 函数根据传入的参数创建指定的对象并返回

5

通过工厂对象的 createAni 方法创建 dog 对象，执行 eat 和 voice 方法

```
fa = factoryAni()
d = fa.createAni('dog')
d.eat()
d.voice()
```

运行程序，发现通过传入参数的不同，工厂对象分别创建出 dog 和 cat 对象,并执行了对应的方法，工厂模式创建对象测试成功