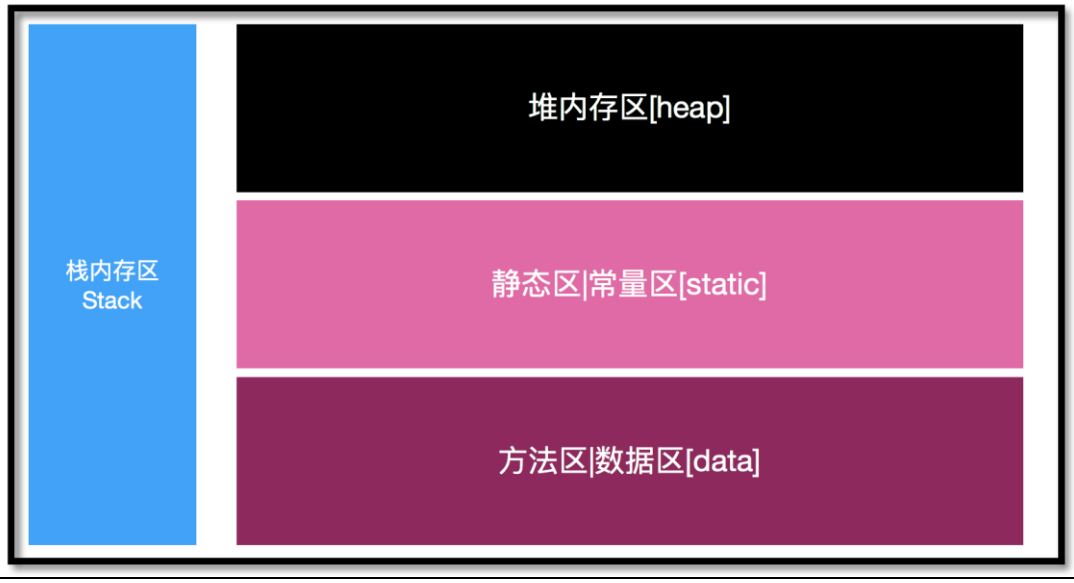
1. 内存分析和处理

程序的运行离不开对内存的操作，一个软件要运行，需要将数据加载到内存中，通过 CPU 进行内存数据的读写，完成数据的运算。

1.1. 程序内存浅析

软件程序在计算机中的执行，主要是通过数据单元、控制单元、执行单元共同协作，完成数据的交互，达到程序处理数据的目的，在软件执行的过程中，由于系统内存和 CPU 的资源非常有限，所以有效的分解软件中的各项数据，将不同的数据加载到不同的内存部分以有效的运行程序，同时可以达到在一个计算机中有效 运行更多软件的目的

PYTHON 程序在运行过程中，主要是解释器从系统中申请内存空间以运行 PYTHON 软件
解释器将申请的内存主要区分为这样几个部分用于处理执行的程序软件

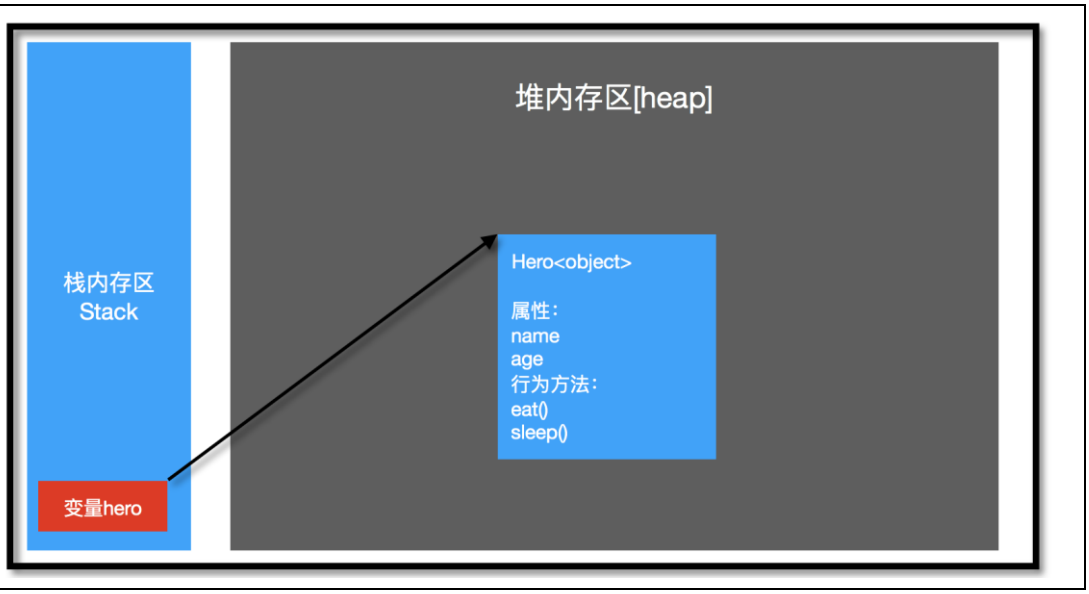


栈内存区[stack]：用于直接分配数据，存取速度较快，数据存储不稳定，适用于小数据块的快速存取，一般在程序中用于存储变量数据
方法区[data]：主要用于加载程序中使用的代码数据、二进制数据、方法数据等等程序运行需要的预加载数据
静态区[static]：主要用于加载存储程序中的一些静态数据、常量数据等等，在 PYTHON 中的不可变数据类型的数据，也会存储在静态常量区内存中

堆内存[heap]：存储数据稳定持久，一般用于存储加载较为重量级的数据，

如程序运行过程中的对象都是存在堆内存中的

程序中变量和对象的基本表示方式
程序中有一个类型 Hero，我们在代码中创建了一个 Hero 对象，将对象的值赋值给一个变量 hero，内存中内存信息的分配如下：



1.1.1. 不可变数据类型 VS 可变数据类型

PYTHON 中根据数据是否可以进行修改提供了两种不同的数据类型

- 不可变数据类型：一般基本数据类型都是不可变数据类型
- 可变数据类型：一般组合数据类型或者自定义数据类都是可变数据类型

怎么区分可变和不可变？为什么要有这样的规则?
PYTHON 中的一切都是对象，可以通过 id()函数查询对象在内存中的地址数据
可变数据类型是在定义了数据之后，修改变量的数据，内存地址不会发生变化
不可变数据类型是在定义了数据之后，修改变量的数据，变量不会修改原来内存地址的数据而是会指向新的地址，原有的数据保留，这样更加方便程序中基本数据的利用率。

```
>>> a = 10
>>> id(a)
4319454848
>>> a = 12
>>> id(a)
4319454912
```

```
>>> u = list()
>>> id(u)
4322937928
>>> u.append("hello")
>>> id(u)
4322937928
>>> u.append("world")
>>> u
['hello', 'world']
>>> id(u)
4322937928
>>>
```

那么就比较容易理解下面代码的含义了，请从内存的角度分析如下代码的执行过程。

```
def chg_data_1(x):
    x = 12
    print("method: {}".format(x))


def chg_data_2(y):
    y.append("hello")
    print("method: {}".format(y))


if __name__ == "__main__":
    a = 10
    chg_data_1(a)# 实际参数传递不可变类型

    print("main:", a) # 这里 a 是多少?


    b = [1,2,3]
    chg_data_2(b)# 实际参数传递可变类型

    print("main:", b) # 这里的 b 是多少?
```

1.1.2. 代码和代码块

PYTHON 中的最小运行单元是代码块，代码块的最小单元是一行代码

在实际开发过程中，需要注意的是 python 有两种操作方式

- 交互模式

- IDE 开发模式

在交互模式下，每行命令是一个独立运行的代码块，每个代码块运行会独立申请一次内存，在操作过程中交互模式没有退出的情况下遵循 PYTHON 官方操作标准
如：对基本数据类型进行了基本优化操作，将一定范围内的数据存储在常量区以提升性能

```
Python 3.6.5 |Anaconda, Inc.| (default, Apr 26 2018, 08:42:37)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 12
>>> b = 12
>>> id(a)
4426725056
>>> id(b)
4426725056
>>> a = "hello"
>>> b = "hello"
>>> id(a)
4430210920
>>> id(b)
4430210920
>>> a = 1000
>>> b = 1000
>>> id(a)
4429864720
>>> id(b)
4429864752
```

-5~256之间的数据自动缓存

字符串自动缓存

超出范围的数据重新申请内存

但是在 IDE 开发模式下，代码封装在模块中，通过 python 命令运行模块时，模块整体作为一个代码块向系统申请内存并执行程序，执行过程中对于基本数据类型进行缓存优化操作，通过 pycharm 工具执行上述代码测试：

```
1   a = 12
2   b = 12
3   print(id(a), id(b))
4
5   a = "hello"
6   b = "hello"
7   print(id(a), id(b))
8
9   a = 1000
10  b = 1000
11  print(id(a), id(b))
```

```
4338726592 4338726592
4341200016 4341200016
4339572432 4339572432
```

1.2. 程序内存代码检测

PYTHON 对于内存的操作，社区开发了一款比较强大的专门用于检测代码内存使用率，用于

项目代码调优的模块 memory_profile 是一个使用较为简单，并且可视化比较直观的工具模块，通过 pip 工具安装即可使用

```
pip install memory_profiler
```

通过在测试的函数或者类型等前面添加@profile 注解，让内存分析模块可以直接进行代码运行检测，修改上一节代码如下：

```
from memory_profiler import profile

@profile
def chg_data_1(x):
    x = 12
    print("method: {}".format(x))

@profile
def chg_data_2(y):
    y.append("hello")
    print("method: {}".format(y))


if __name__ == "__main__":
    a = 10
    chg_data_1(a)# 实际参数传递不可变类型

    print("main:", a) # 这里 a 是多少?


    b = [1,2,3]
    chg_data_2(b)# 实际参数传递可变类型

    print("main:", b) # 这里的 b 是多少?
```

执行代码程序，运行结果如下：

```
method: 12
Filename:  /Users/wenbinmu/workspace/work_py_spider/case_adv/demo08_ 内 存 分 析.py

Line #    Mem usage    Increment   Line Contents
================================================
    3    12.5 MiB    12.5 MiB   @memory_profiler.profile
    4                           def chg_data_1(x):
    5    12.5 MiB     0.0 MiB       x = 12
    6    12.5 MiB     0.0 MiB       print("method: {}".format(x))
```

```
main: 10
method: [1, 2, 3, 'hello']
Filename:  /Users/wenbinmu/workspace/work_py_spider/case_adv/demo08_内存分析.py

Line #    Mem usage    Increment   Line Contents
================================================
     8     12.5 MiB     12.5 MiB   @memory_profiler.profile
     9                             def chg_data_2(y):
    10     12.5 MiB      0.0 MiB       y.append("hello")
    11     12.5 MiB      0.0 MiB       print("method: {}".format(y))


main: [1, 2, 3, 'hello']

Process finished with exit code 0
```

1.3. 操作符号：is 和==的使用

前面看过了变量以及对象在内存中的分配和执行情况，那么在程序执行过程中，怎么判断对象和对象之间的关系呢，PYTHON 提供了对象操作符号 is 和内容操作符号==，用于判断对象和对象中的值的情况

A is B：判断对象 A 和对象 B 是否同一个内存地址，即是否同一个对象
A == B：判断 A 中的内容是否和 B 中的内容一致

不论是基本类型的数据，还是内容复杂的对象，都可以通过对象判断符号 is 和内容判断操作符号==来进行确定

不可变数据类型的数据判断

```
>>> a = 100
>>> b = 100
>>> id(a)
4362084800
>>> id(b)
4362084800
>>> a is b
True
>>> a == b
True
>>>
>>> a = 1000
>>> b = 1000
>>> id(a)
4363706096
>>> id(b)
4365221648
>>> a is b
False
>>> a == b
True
```

组合数据类型的数据判断

创建的每个组合数据类型的对象都是独立的，如下面的代码中的 a 和 b 变量中分别存放了两个不同的列表类型的对象，所以 is 判断是 False，但是值又是相同的所以==判断 True

```
>>> a = [1,2,3,4,5]
>>> b = [1,2,3,4,5]
>>>
>>> a == b
True
>>> a is b
False
>>> id(a)
4365529608
>>> id(b)
4365565192
>>>
```

引用数据类型的操作：自定义数据类型，变量中存放的是对象在内存中的地址

自定义类型的对象，每次创建同样也是在堆内存中单独创建的对象，所以 is 判断 False

但是为什么我们创建的两个 name 属性都为 tom 的对象，通过==判断还是 Flase 呢？哪是因为创建的对象在通过==判断时，会自动调用父类的__eq__()函数进行判断，默认情况下创建的对象内部变量，使用的是对象自己的内存空间，所以==判断是 False

```
>>> class Person:
...     def __init__(self, name):
...         self.name = name
...
>>> p = Person("tom")
>>> p2 = Person("tom")
>>> id(p)
4365568600
>>> id(p2)
4365568768
>>> p is p2
False
>>> p == p2
False
>>>
```

1.4. 引用、浅拷贝、深拷贝

对象和对象之间，通过内存的操作存在各种各样的关系，那么这些关系都有什么样的特点又是怎么分配的呢?
这里就要说到对象的内存分配，对象的引用赋值、对象的深浅拷贝了

1.4.1. 对象的内存分配

对象的创建，依赖于申请的内存空间中数据的加载，对象在内存中的创建过程依赖于三部分内存处理：对象分配内存地址、引用变量分配内存地址、对象和引用变量之间的关联

参考代码:

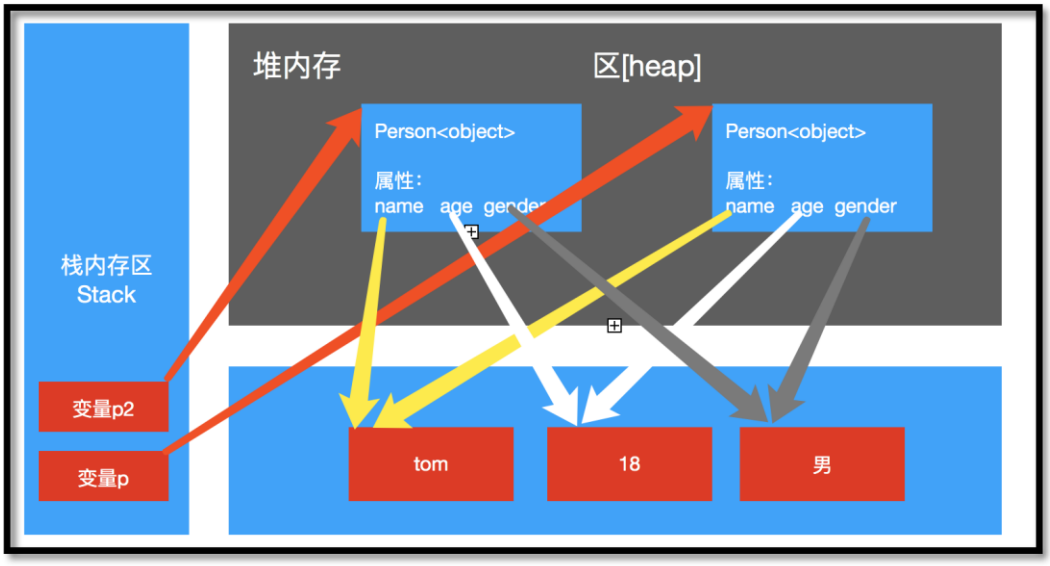
```python
from memory_profiler import profile


class Person:
    '''自定义类型'''
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

@profile(precision=10)
def main():
    '''入口函数'''

    p = Person("tom", 18, "男")
```

```
    print(p)
    p2 = Person("tom", 18, "男")
    print(p)

if __name__ == "__main__":
    # 运行代码
    main()
```

执行内存处理

```
/Users/wenbinmu/workspace/work_py_spider/venv36/bin/python /Users/wenbinmu/workspace/work_py_spider/case
<__main__.Person object at 0x107928320>
<__main__.Person object at 0x107928320>
Filename: /Users/wenbinmu/workspace/work_py_spider/case_adv/demo09_内存分配.py

Line #    Mem usage    Increment   Line Contents
================================================
    11  12.4765625000 MiB  12.4765625000 MiB   @profile(precision=10)
    12                             def main():
    13                                 '''入口函数'''
    14  12.4804687500 MiB   0.0039062500 MiB       p = Person("tom", 18, "男")
    15  12.4804687500 MiB   0.0000000000 MiB       print(p)
    16  12.4804687500 MiB   0.0000000000 MiB       p2 = Person("tom", 18, "男")
    17  12.4804687500 MiB   0.0000000000 MiB       print(p)
```

内存分配图解



由于对象的创建，是将堆内存中创建的对象的地址临时存储在栈内存中的变量中，那么在程序中如果要在多个地方使用一个对象数据时应该怎么办呢？一般想到的都是将对象像文件一样复制一份不就好了。

PYTHON 中对于这样的情况，有三种不同的操作方式

- 如果程序中多个不同的地方都要使用同一个对象，通过对象的引用赋值，将同一个对象

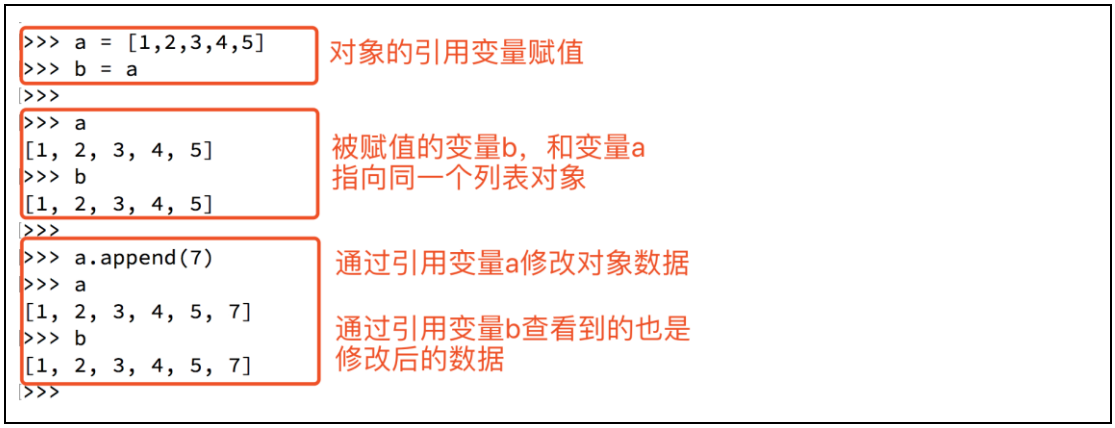
赋值给多个变量

- 如果程序中多个不同的地方都要使用相同的对象数据，通过对象的拷贝完成数据的简单复制即可，对象中的包含的数据要求必须统一
- 如果程序中多个不同的地方使用相同的而且独立的对象数据，通过对象的深层次的复制将对象的数据完整复制成独立的另一份即可

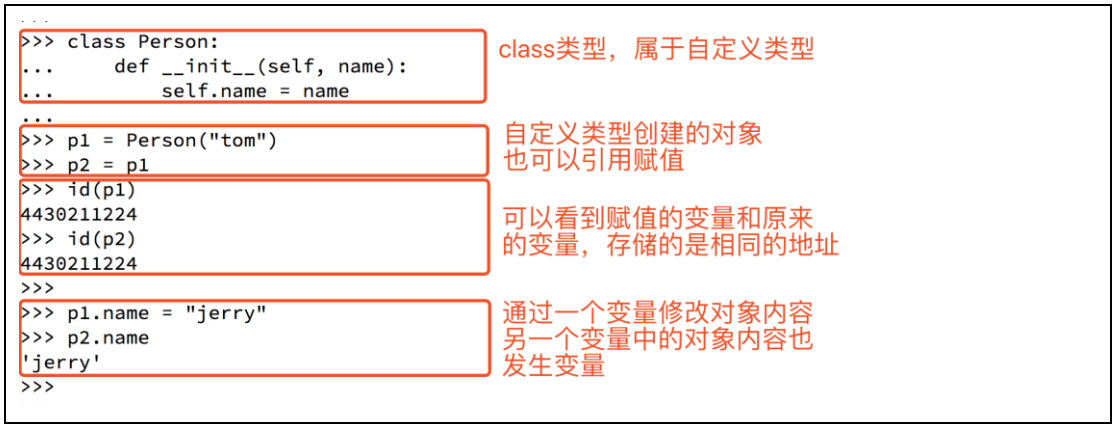
1.4.2. 对象的引用赋值

对象的引用赋值，可以将对象的内存地址同时赋值给多个变量，这多个变量中存放的都是同一个对象的引用地址，如果通过一个变量修改了对象内容，那么其他变量指向的对象内容也会同步发生改变

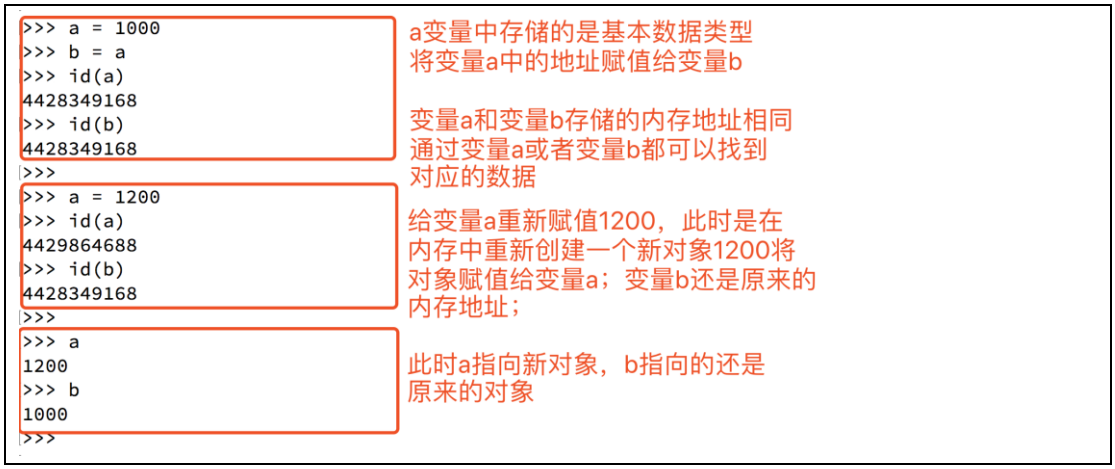
将一个变量中存放的对象的地址数据，赋值给其他变量，通过赋值操作符号就可以完成



对象的引用变量的赋值，并不是对象的复制或者备份，而仅仅是将对象的地址存储在多个变量中方便程序操作。



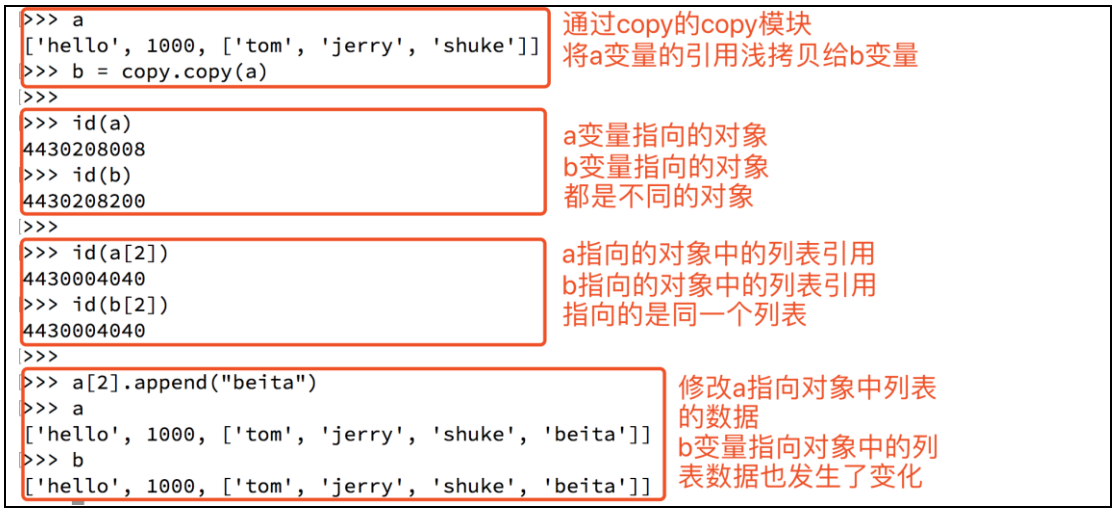
注意：PYTHON 中所谓对象的引用赋值，针对的是可变类型，不论是组合数据类型或者自定义 class 类型，都具备引用赋值的操作；但是不适合不可变类型，不可变类型的引用赋值操作有只读不写的特征，一旦通过变量重新赋值就会重新指向新的引用对象



1.4.3. 对象的浅拷贝

对于程序中对象的拷贝操作，除了前面介绍的引用赋值操作可以完成同一个对象在程序不同位置的操作，这里的浅拷贝更是一种对象的临时备份，浅拷贝的核心机制主要是赋值对象内部数据的引用

PYTHON 内建标准模块 copy 提供了一个 copy 函数可以完成



1.4.4. 对象的深拷贝

和对象的浅拷贝不同，对象的深拷贝，是对象数据的直接拷贝，而不是简单的引用拷贝

主要是通过 PYTHON 内建标准模块 copy 提供的 deepcopy 函数可以完成对象深拷贝

1.5. 垃圾回收机制

垃圾回收机制(Garbage Collection:GC)基本是所有高级语言的标准配置之一了
在一定程度上，能优化编程语言的数据处理效率和提高编程软件开发软件的安全性能

在 PYTHON 中的垃圾回收机制主要是以引用计数为主要手段
以标记清除和隔代回收机制作为辅助操作手段
完成对内存中无效数据的自动管理操作的!

1.5.1. 引用计数

引用计数[Reference Counting: RC]是 PYTHON 中的垃圾回收机制的核心操作算法
该算法最早是 George E.Collins 在 1960 年首次提出的，并在大部分高级语言中沿用至今，是很多高级语言的垃圾回收核心算法之一

(1) 什么是引用计数

引用计数算法的核心思想是：当一个对象被创建或者拷贝时，引用计数就会+1，当这个对象的多个引用变量，被销毁一个时该对象的引用计数就会-1，如果一个对象的引用计数为 0 则表示该对象已经不被引用，就可以让垃圾回收机制进行清除并释放该对象占有的内存空间了。

引用计数算法的优点是：操作简单，实时性能优秀，能在最短的时间获得并运算对象引用数

引用计数算法的缺点是：为了维护每个对象的引用计数操作算法，PYTHON 必须提供和对象对等的内存消耗来维护引用计数，这样就在无形中增加了内存负担；同时引用计数对于循环应用/对象之间的互相引用，是无法进行引用计数操作的，所以就会造成常驻内存的情况。

(2) PYTHON 中的引用计数

PYTHON 是一个面向对象的弱类型语言，所有的对象都是直接或者间接继承自 object 类型，object 类型的核心其实就是一个结构体对象

```
typedef struct_object {
   int ob_refcnt;
   struct_typeobject *ob_type;
} PyObject;
```

在这个结构体中，ob_refcnt 就是对象的引用计数，当对象被创建或者拷贝时该计数就会增加+1，当对象的引用变量被删除时，该计数就会减少-1，当引用计数为 0 时，对象数据就会被回收释放了。在 python 中，可以通过 sys.getrefcount()来获取一个对象的引用计数

```
>>> class Person:
...     pass
...
>>>
>>>
>>> p = Person()
>>>
>>> sys.getrefcount(p)
2
>>>
>>> p2 = p
>>> sys.getrefcount(p)
3
>>>
>>> del p2
>>>
>>> sys.getrefcount(p)
2
```

创建了一个类型

创建了一个对象，获取引用计数
p变量是一个引用
getrefcount()函数是一个引用
所以引用计数值：2

这里进行了对象的引用赋值
所以对象的引用计数值：3

这里删除了一个引用变量p2
所以对象的引用计数-1：2

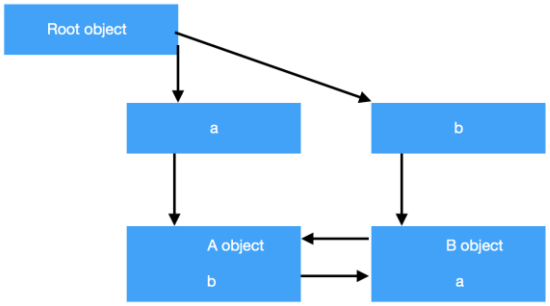
1.5.2. 标记清除

PYTHON 中的标记-清除机制主要是针对可能产生循环引用的对象进行的检测机制
在 PYTHON 中的基本不可变类型如 PyIntObject，PyStringObject 等对象的内部不会内聚其他对象的引用，所以不会产生循环引用，一般情况下循环引用总是发生在其他可变对象的内部属性中，如 list,dict,class 等等，使得该方法消耗的资源和程序中可变对象的数量息息相关!

标记清除算法核心思想：首先找到 PYTHON 中的一批根节点对象，如 object 对象，通过根节点对象可以找到他们指向的子节点对象，如果搜索过程中有这个指向是从上往下的指向，表示这个对象是可达的，否则该对象是不可达的，可达部分的对象在程序中需要保留下来，不可达部分的对象在程序中是不需要保留的

```
class A:
    pass

class B:
    pass
```

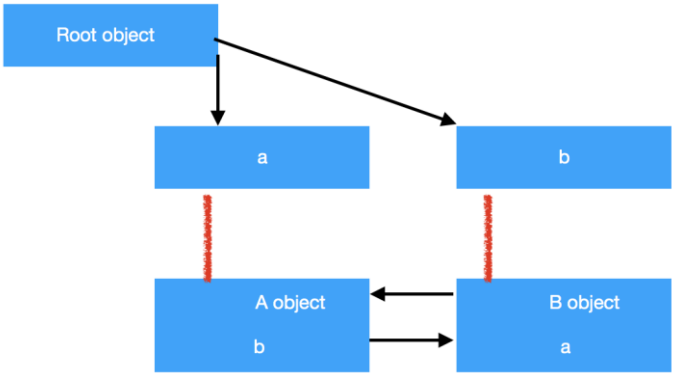
```
a = A()
b = B()
a.bb = b
b.aa = a
```



如果代码中执行了

```
del a
del b
```

我们会发现，对象 A()和对象 B()依然有引用指向他们，如果是之前的引用计数的方式明显区分不了这样的对象是否应该删除；但是标记-清除的方式，就可以标记出来对象 A()和对象 B()是不可达对象，不需要保留，直接删除即可！

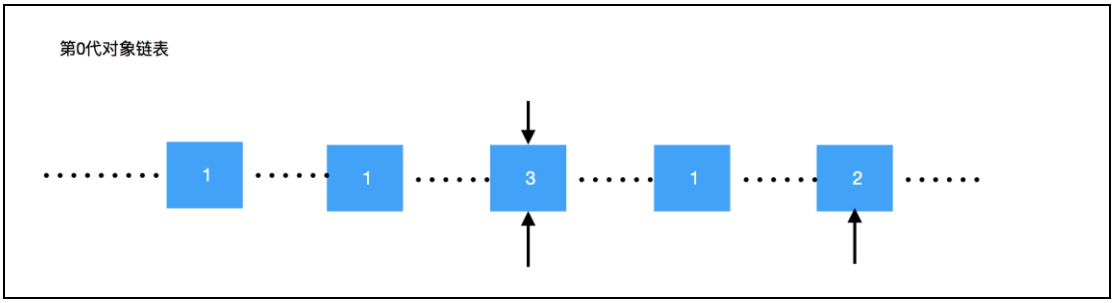


红色部分的引用被删除了

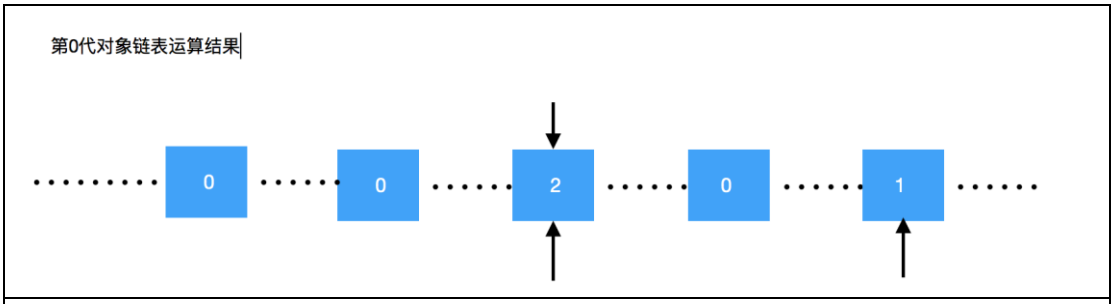
1.5.3. 分代回收

PYTHON 中的分代回收机制，是一种通过空间换取时间效率的做法，PYTHON 内部处理机制定义了三个不同的链表数据结构[第零代(年轻代)，第 1 代(中年代)，第 2 代(老年代)]PYTHON 为了提高程序执行效率，将垃圾回收机制进行了阈值限定，0 代链表中的垃圾回收机制执行最为密集，其次是 1 代，最后是 2 代;

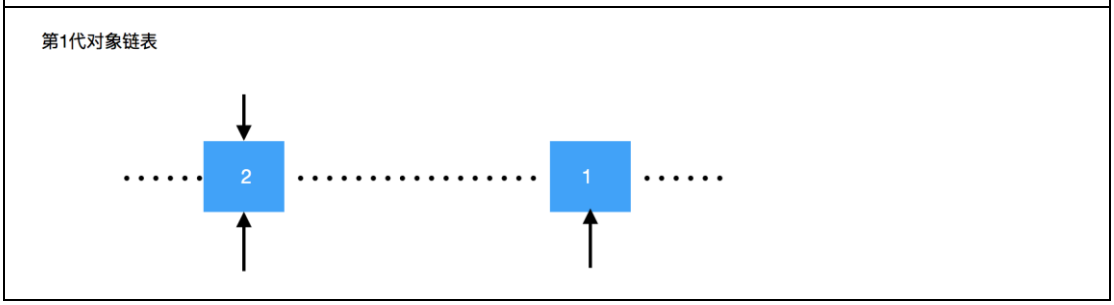
PYTHON 定义的这三个链表，主要是针对我们在程序中创建的对象，首先会添加到 0 代链表



随后 0 代链表数量达到一定的阈值之后，触发 GC 算法机制，对 0 代对象进行符合规则的引用计数运算，避免出现对象的延迟或者过早的释放



最终，触发 GC 机制将已经没有引用指向的对象进行回收，并将有引用继续指向的对象移动到第 1 代对象链表中；第 1 代对象链表的对象，就是比第 0 代对象链表中的对象可能存活更久的对象，GC 阈值更大检测频率更慢，以提高程序执行效率。



以此类推直到一部分对象存活在第 2 代对象链表中，对象周期较长的可能跟程序的生命周期一样了。

备注：弱代假说:程序中年轻的对象往往死的更快，年老的对象往往存活更久

参考文章：https://www.cnblogs.com/pinganzi/p/6646742.html

1.5.4. 垃圾回收处理

PYTHON 中的垃圾回收机制有了一定的了解之后，我们针对垃圾回收机制的操作通过代码进行测试
PYTHON 中的 gc 模块提供了垃圾回收处理的各项功能机制，必须 import gc 才能使用

gc.set_debug(flags)：设置 gc 的 debug 日志，一般为 gc.DEBUG_LEAK
gc.collect([generation])：显式进行垃圾回收处理，可以输入参数~参数表示回收的对象代数，0 表示只检查第 0 代对象，1 表示检查第 0、1 代对象，2 表示检查 0、1、2 代独对象，如果不传递参数，执行 FULL COLLECT，也就是默认传递 2
gc.set_threshold(threshold0 [, threshold2 [, threshold3]])：设置执行垃圾回收机制的频率
gc.get_count()：获取程序对象引用的计数器
gc.get_threshold()：获取程序自动执行 GC 的引用计数阈值

以上是 PYTHON 中垃圾回收机制的基本操作，在程序开发过程中需要注意：

- 项目代码中尽量避免循环引用
- 引入 gc 模块，启用 gc 模块自动清理循环引用对象的机制
- 将需要长期使用的对象集中管理，减少 GC 资源消耗
- gc 模块处理不了重写__del__方法导致的循环引用，如果一定要添加该方法，需要显式调用 gc 模块的 garbage 中对象的__del__方法进行处理