Day08-多任务编程vs深浅拷贝

今日课程学习目标

```
掌握多任务编程中进程和线程的概念
熟悉python中多进程和多线程的使用
常握深拷贝和浅拷贝的区别
```

今日课程内容大纲

```
# 1. 多任务编程
    多任务的基本概念
    进程：资源分配的基本单位
        python中多进程的基本使用
        主进程和子进程
        进程执行带有参数的任务
        进程使用的2个注意点
        守护进程和终止子进程
    线程：CPU调度的基本单位
        python中多线程的基本使用
        线程执行带有参数的任务
        线程使用的3个注意点
        守护线程的设置
        线程资源共享问题与解决
# 2. 深拷贝和浅拷贝
    深拷贝
    浅拷贝
```

知识点1：多任务的基本概念【掌握】

思考：

```
以我们现在所学的知识，能让两个函数同时执行吗？
```

问题：

```
1.什么是多任务？多任务有什么优点？
2.多任务是如何实现的？
3.多任务是如何执行的？
```

1. 什么是多任务?

多任务是指多个任务同时执行。比如：百度网盘同时下载多部电影；一边听歌软件听歌，一边写代码；

多任务的最大好处是充分**利用CPU资源，提高程序的执行效率。**

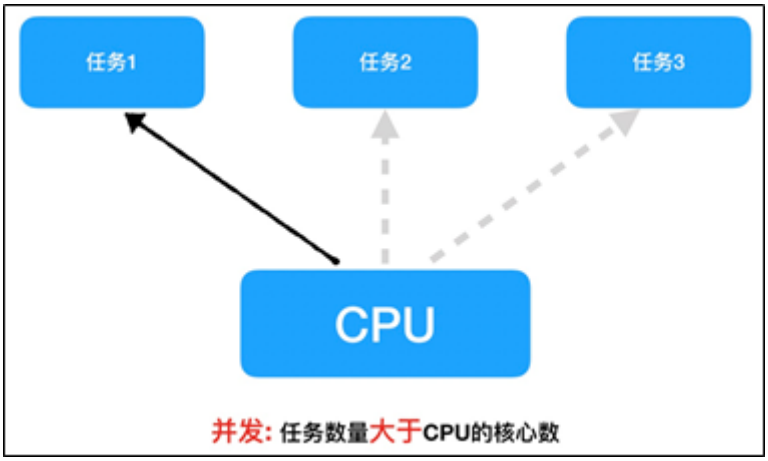
2. 多任务是如何实现的?

多任务的实现方式有 2 种：多进程和多线程。

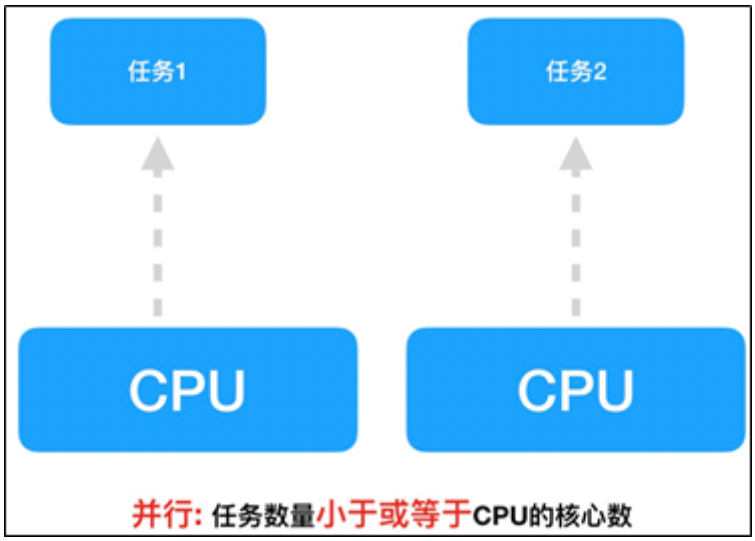
3. 多任务是如何执行的?

多任务的执行分成 2 种方式：并发和并行。

并发：在一段时间内**快速交替**去执行多个任务



并行：在一段时间内**真正的同时一起**执行多个任务



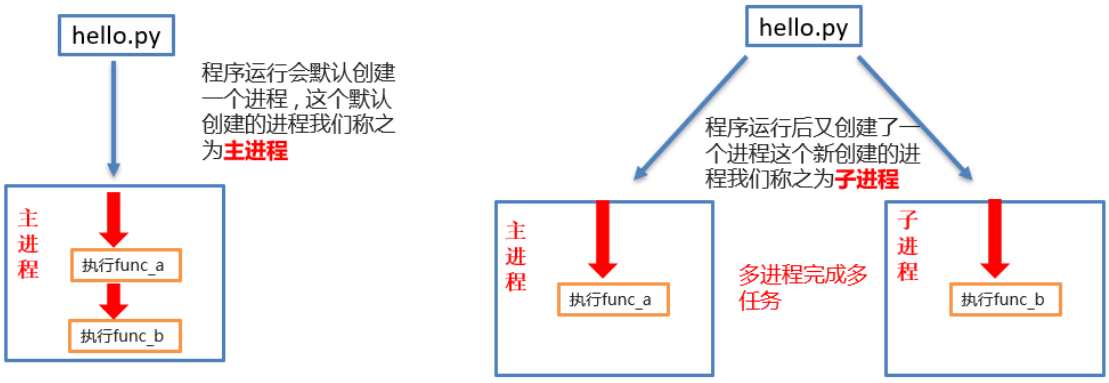
知识点2：进程的基本概念【掌握】

进程（Process）是资源分配的最小单位，**它是操作系统进行资源分配和调度运行的基本单位**，通俗理解：一个正在运行的程序就是一个进程。例如：正在运行的QQ、微信等他们都是一个进程。

> **一个程序运行后至少有一个进程**

多进程可以完成多任务，每个进程就好比一家独立的公司，每个公司都各自在运营，每个进程也各自在运行，执行各自的任务。

多进程的作用：



知识点3：Python中多进程的基本使用【熟悉】

多进程的创建步骤：

1）导入进程包

```
import multiprocessing
```

2）通过进程类创建进程对象

```
进程对象 = multiprocessing.Process(target=进程执行的任务函数)

其他参数：
- group：指定进程组，目前只能使用None
- target：执行的目标任务名
- name：进程名字
- args：以元组方式给执行任务传参
- kwargs：以字典方式给执行任务传参
```

3）启动进程执行任务

```
进程对象.start()
```

多进程示例代码：

```
"""
多进程的基本使用
学习目标：能够使用多进程同时执行两个不同的任务函数
"""
import time
# 导入进程包
import multiprocessing


```

```python
# 跳舞函数
def dance():
    for i in range(5):
        print('跳舞中...')
        time.sleep(1)


# 唱歌函数
def sing():
    for i in range(5):
        print('唱歌中...')
        time.sleep(1)


if __name__ == '__main__':
    # 创建一个进程，执行 dance 函数
    # 注意：target指定的是函数名或方法名，切忌！！！不要再函数名或方法名后加()
    dance_process = multiprocessing.Process(target=dance)

    # 再创建一个进程，执行 sing 函数
    sing_process = multiprocessing.Process(target=sing)

    # 启动这两个进程
    dance_process.start()
    sing_process.start()
```

知识点4：进程关系-主进程和子进程【常握】

思考题：

```
1. 上面的代码执行时，一共有几个进程？
```

补充命令：

- tasklist：查看 Windows 系统中整体执行的进程
- tasklist | findstr python：查看 Windows 系统中正在运行的 python 进程

获取进程编号函数：

```
import os
os.getpid()：获取当前进程的编号
os.getppid()：获取当前进程父进程的编号
```

知识点5：进程执行带有参数的任务【熟悉】

Process 类执行任务并给任务传参数有两种方式：

- args 表示以元组的方式给执行任务传参：传参一定要和参数的顺序保持一致
- kwargs 表示以字典方式给执行任务传参：传参字典中的key一定要和参数名保持一致

示例代码：

```python
"""
进程执行带有参数的任务(函数)
学习目录：能够使用多进程执行带有参数的任务
"""

import multiprocessing
import time
```

```python


# 带有参数的任务(函数)
def task(count):
    for i in range(count):
        print('任务执行中...')
        time.sleep(0.2)
    else:
        print('任务执行完成')


if __name__ == '__main__':
    # 创建一个进程，执行 task 任务函数
    # 通过元祖指定任务函数的参数：args=(5, )，按照参数顺序指定参数的值
    # sub_process = multiprocessing.Process(target=task, args=(5, ))

    # 通过字典指定任务函数的参数：kwargs={'count': 3}，按照参数名称指定参数的值
    sub_process = multiprocessing.Process(target=task, kwargs={'count': 3})

    # 启动进程
    sub_process.start()
```

知识点6：进程使用的 2 个注意点【常握】

进程使用的注意点介绍：

1）进程之间不共享全局变量

> 注意：创建子进程时，子进程会主进程的东西全部复制一份

2）主进程会等待所有的子进程执行结束再结束

示例代码1：

```python
# 注意点1：进程之间不共享全局变量
import multiprocessing
import time

# 定义全局变量
g_list = []


# 添加数据的函数
def add_data():
    for i in range(5):
        g_list.append(i)
        print('add：', i)
        time.sleep(0.2)

    print('add_data：', g_list)


# 读取数据的函数
def read_data():
    print('read_data：', g_list)


if __name__ == '__main__':
```

```
    # 创建添加数据的子进程
    add_data_process = multiprocessing.Process(target=add_data)
    # 创建读取数据的子进程
    read_data_process = multiprocessing.Process(target=read_data)

    # 启动添加数据子进程
    add_data_process.start()
    # 主进程等待 add_data_process 执行完成，再向下继续执行
    add_data_process.join()
    # 启动读取数据子进程
    read_data_process.start()

    print('main：', g_list)
```

示例代码2：

```
# 注意点2：主进程会等待所有的子进程执行结束再结束
import multiprocessing
import time


# 任务函数
def task():
    for i in range(20):
        print('任务执行中...')
        time.sleep(0.2)


if __name__ == '__main__':
    # 创建子进程并启动
    sub_process = multiprocessing.Process(target=task)
    sub_process.start()

    # 主进程延时 1s
    time.sleep(1)
    print('主进程结束！')
    # 退出程序
    exit()
```

知识点7：守护进程和终止子进程【熟悉】

如何让主进程执行结束时，子进程就结束执行？

1）方式1：将子进程设置为守护进程

```
进程对象.daemon = True # 注意点：设置守护进程必须在子进程启动之前
```

2）方式2：主进程结束时直接终止子进程

```
进程对象.terminate()
```

示例代码1：将子进程设置为守护进程

```
# 任务函数
def task():
    for i in range(10):
        print('任务执行中...')
        time.sleep(0.2)

```

```
7
8   if __name__ == '__main__':
9       # 创建子进程并启动
10      sub_process = multiprocessing.Process(target=task)
11      # TODO：设置子进程为守护进程
12      sub_process.daemon = True # 注意点：设置守护进程必须在子进程启动之前
13
14      sub_process.start()
15
16      # 主进程延时 1s
17      time.sleep(1)
18      print('主进程结束！')
19      # 退出程序
20      exit()
```

示例代码2：直接终止子进程

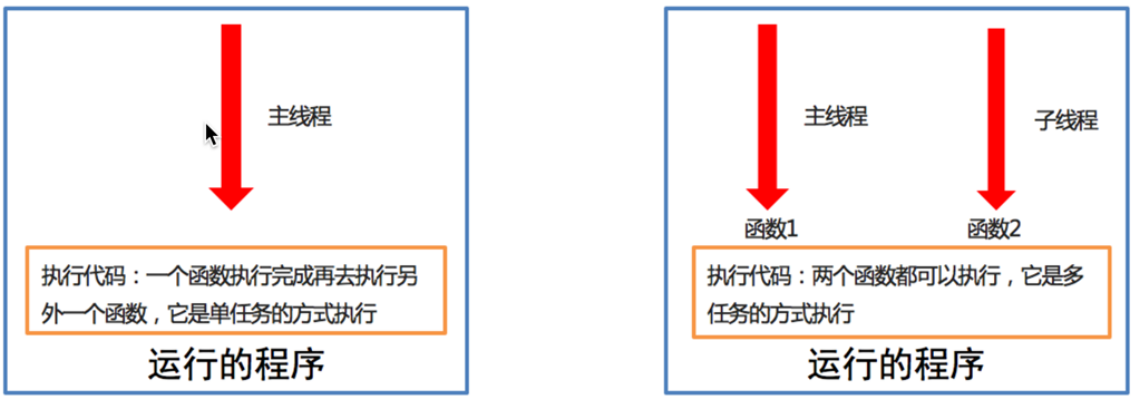
```
1   import multiprocessing
2   import time
3
4
5   # 任务函数
6   def task():
7       for i in range(10):
8           print('任务执行中...')
9           time.sleep(0.2)
10
11
12  if __name__ == '__main__':
13      # 创建子进程并启动
14      sub_process = multiprocessing.Process(target=task)
15      sub_process.start()
16
17      # 主进程延时 1s
18      time.sleep(1)
19      print('主进程结束！')
20      # TODO: 终止子进程
21      sub_process.terminate()
22
23      # 退出程序
24      exit()
```

知识点8：线程的基本概念【掌握】

线程是进程中执行代码的一个分支，每个执行分支（线程）要想工作执行代码需要 CPU 进行调度，也就是说**线程是 CPU 调度的基本单位**，每个进程默认都有一个线程，而这个线程就是我们通常说的主线程。

> 可以把进程理解为公司，线程理解为公司中的员工，一个公司至少有一个员工。公司(进程)负责提供资源，员工(线程)负责实际干活。

多线程效果图：



说明：程序启动默认会有一个主线程，程序员自己创建的线程可以称为子线程，多线程可以完成多任务。

知识点9：Python中多线程的基本使用【熟悉】

多线程的创建步骤：

1）导入线程模块

```
import threading
```

2）通过线程类创建线程对象

```
线程对象 = threading.Thread(target=进程执行的任务函数)

其他参数：
- group：线程组，目前只能使用None
- target：执行的目标任务名
- args：以元组的方式给执行任务传参
- kwargs：以字典方式给执行任务传参
- name：线程名，一般不用设置
```

3）启动线程执行任务

```
线程对象.start()
```

多线程示例代码：

```
"""
多线程的基本使用
学习目标：能够使用多线程同时执行两个不同的函数
"""

import time
# 导入模块
import threading


# 跳舞任务函数
def dance():
    for i in range(5):
        print('正在跳舞...%d' % i)
        time.sleep(1)


# 唱歌任务函数
```

```python
def sing():
    for i in range(5):
        print('正在唱歌...%d' % i)
        time.sleep(1)


if __name__ == '__main__':
    # 创建一个线程，执行 dance 任务函数
    dance_thread = threading.Thread(target=dance)

    # 再创建一个线程，执行 sing 任务函数
    sing_thread = threading.Thread(target=sing)

    # 启动这两个线程
    dance_thread.start()
    sing_thread.start()
```

知识点10：线程执行带有参数的任务【熟悉】

Thread 类执行任务并给任务传参数有两种方式：

- args 表示以元组的方式给执行任务传参：传参一定要和参数的顺序保持一致
- kwargs 表示以字典方式给执行任务传参：传参字典中的key一定要和参数名保持一致

示例代码：

```python
"""
线程执行带有参数的任务(函数)
学习目录：能够使用多线程执行带有参数的任务
"""
# 导入线程模块
import threading
import time


# 带有参数的任务(函数)
def task(count):
    for i in range(count):
        print('任务执行中...')
        time.sleep(0.2)
    else:
        print('任务执行完成')


if __name__ == '__main__':
    # 创建一个线程，执行 task 任务函数
    # sub_thread = threading.Thread(target=task, args=(3, ))
    sub_thread = threading.Thread(target=task, kwargs={'count': 5})
    # 启动线程
    sub_thread.start()
```

知识点11：线程使用的 3 个注意点【常握】

线程使用的注意点介绍：

1）线程之间执行是无序的

2）主线程会等待所有的子线程执行结束再结束

3）线程之间共享全局变量

示例代码1：线程之间执行是无序的

```python
import threading
import time


def task():
    time.sleep(1)
    print(f'当前线程：{threading.current_thread().name}')


if __name__ == '__main__':
    for i in range(5):
        sub_thread = threading.Thread(target=task)
        sub_thread.start()
```

示例代码2：主线程会等待所有的子线程执行结束再结束

```python
import threading
import time


def task():
    for i in range(5):
        print('任务执行中...')
        time.sleep(0.5)


if __name__ == '__main__':
    # 创建子线程
    sub_thread = threading.Thread(target=task)
    sub_thread.start()

    # 主进程延时 1s
    time.sleep(1)
    print('主线程结束！')
```

示例代码3：线程之间共享全局变量

```python
import threading
import time

# 定义全局变量
g_list = []


# 添加数据的函数
def add_data():
    for i in range(5):
        g_list.append(i)
        print('add：', i)
        time.sleep(0.2)

    print('add_data：', g_list)


```

```python
# 读取数据的函数
def read_data():
    print('read_data：', g_list)


if __name__ == '__main__':
    # 创建添加数据的子线程
    add_data_thread = threading.Thread(target=add_data)
    # 创建读取数据的子线程
    read_data_thread = threading.Thread(target=read_data)

    # 启动添加数据子线程
    add_data_thread.start()
    # 主线程等待 add_data_thread 执行完成，再向下继续执行
    add_data_thread.join()
    # 启动读取数据子线程
    read_data_thread.start()

    print('main：', g_list)
```

知识点12：守护线程设置【熟悉】

如何让主线程执行结束时，子线程就结束执行?

答：将子线程设置为守护线程

```
方式1：子线程对象.daemon = True
方式2：子线程对象.setDaemon(True)
```

示例代码：

```python
import threading
import time


# 任务函数
def task():
    for i in range(10):
        print('任务执行中...')
        time.sleep(0.2)


if __name__ == '__main__':
    # 创建子线程并启动
    sub_thread = threading.Thread(target=task)
    # TODO：设置子线程为守护线程
    # sub_thread.daemon = True
    sub_thread.setDaemon(True)

    sub_thread.start()

    # 主线程延时 1s
    time.sleep(1)
    print('主线程结束！')
```

知识点13：线程的资源共享问题【常握】

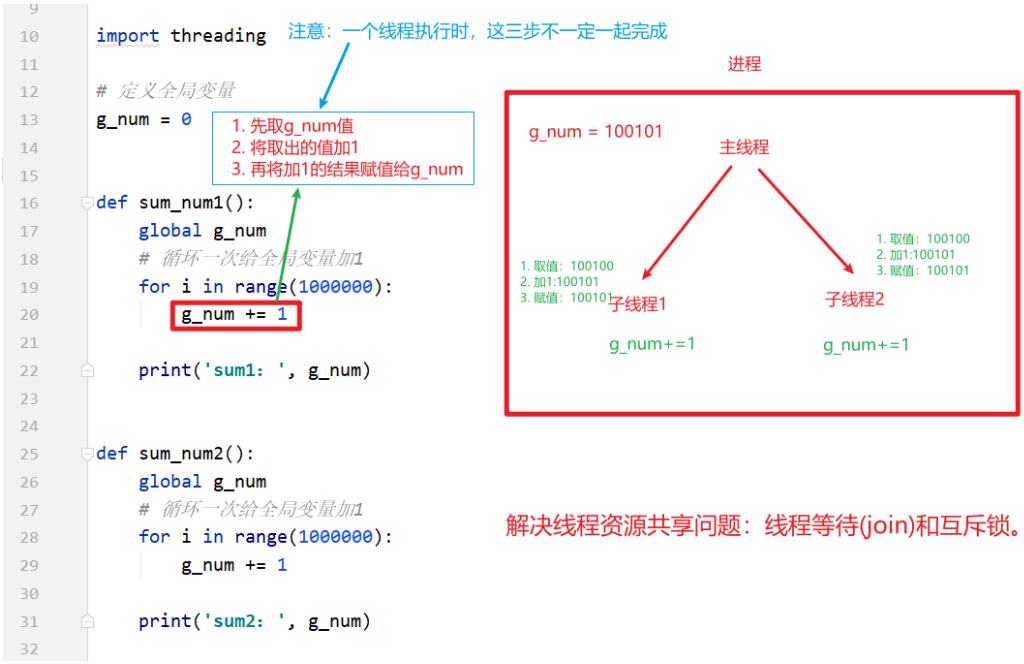
多线程会共享全局变量，当多个线程同时操作同一个共享的全局变量时，可能会造成错误的结果！

示例代码：

```
import threading

# 定义全局变量
g_num = 0


def sum_num1():
    global g_num
    # 循环一次给全局变量加1
    for i in range(1000000):
        g_num += 1

    print('sum1：', g_num)


def sum_num2():
    global g_num
    # 循环一次给全局变量加1
    for i in range(1000000):
        g_num += 1

    print('sum2：', g_num)


if __name__ == '__main__':
    # 创建两个线程
    first_thread = threading.Thread(target=sum_num1)
    second_thread = threading.Thread(target=sum_num2)

    # 启动两个线程
    first_thread.start()
    second_thread.start()
```

思考题：

```
上面的代码执行完成之后，为什么 g_num 的结果不是 200000？
```



知识点14：线程资源共享问题解决：线程等待vs互斥锁【熟悉】

思考题：

```
如何解决线程资源共享出现的错误问题？
答：线程同步：保证同一时刻只能有一个线程去操作共享资源(全局变量)

线程同步的方式：
1）线程等待(join)
2）互斥锁
```

线程等待：等一个线程完全执行结束之后，再执行另外一个线程。

```python
import threading

# 定义全局变量
g_num = 0


def sum_num1():
    global g_num
    # 循环一次给全局变量加1
    for i in range(1000000):
        g_num += 1

    print('sum1: ', g_num)


def sum_num2():
    global g_num
    # 循环一次给全局变量加1
    for i in range(1000000):
        g_num += 1

    print('sum2: ', g_num)


if __name__ == '__main__':
    # 创建两个线程
```

```
    first_thread = threading.Thread(target=sum_num1)
    second_thread = threading.Thread(target=sum_num2)

    # 启动两个线程
    first_thread.start()
    # 线程等待：等待 first_thread 执行完成，主线程的代码再继续向下执行
    first_thread.join()

    second_thread.start()
```

互斥锁：操作共享资源是，多个线程去抢同一把"锁"，抢到锁的线程执行，没抢到锁的线程会阻塞等待

```
import threading

# 定义全局变量
g_num = 0

# 创建一个全局的互斥锁
lock = threading.Lock()


def sum_num1():
    global g_num
    # 循环一次给全局变量加1
    for i in range(1000000):
        # 加锁：拿到锁的线程代码可以继续向下执行，拿不到锁的线程代码会阻塞等待
        lock.acquire()
        g_num += 1
        # 释放锁
        lock.release()

    print('sum1：', g_num)


def sum_num2():
    global g_num
    # 循环一次给全局变量加1
    for i in range(1000000):
        # 加锁：拿到锁的线程代码可以继续向下执行，拿不到锁的线程代码会阻塞等待
        lock.acquire()
        g_num += 1
        # 释放锁
        lock.release()

    print('sum2：', g_num)


if __name__ == '__main__':
    # 创建两个线程
    first_thread = threading.Thread(target=sum_num1)
    second_thread = threading.Thread(target=sum_num2)

    # 启动两个线程
    first_thread.start()
    second_thread.start()
```

知识点15：进程和线程对比【熟悉】

关系对比：

1. 线程是依附在进程里面的，没有进程就没有线程
2. 一个进程默认提供一条线程，进程可以创建多个线程

区别对比：

1. 进程是操作系统资源分配的基本单位，线程是CPU调度的基本单位
2. 线程不能够独立执行，必须依存在进程中
3. 创建进程的资源开销要比创建线程的资源开销要大
4. 进程之间不共享全局变量，线程之间共享全局变量，但是要注意资源竞争的问题
5. 多进程开发比单进程多线程开发稳定性要强

优缺点对比：

进程优缺点:

- 优点：可以用多核
- 缺点：资源开销大

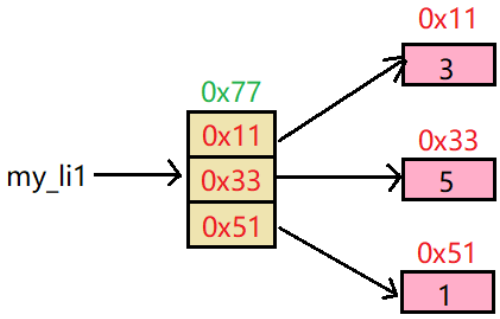
线程优缺点:

- 优点：资源开销小
- 缺点：不能使用多核

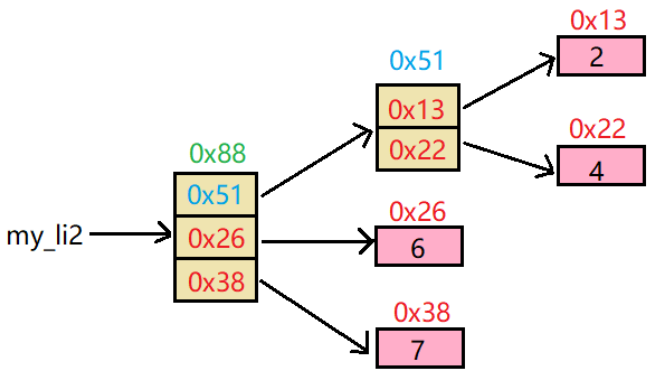
知识点16：简单容器类型和嵌套容器类型数据的内存存储【常握】

以列表为例：

```
# 简单列表
my_li1 = [3, 5, 1]
```




```
# 嵌套列表
my_li2 = [[2, 4], 6, 7]
```

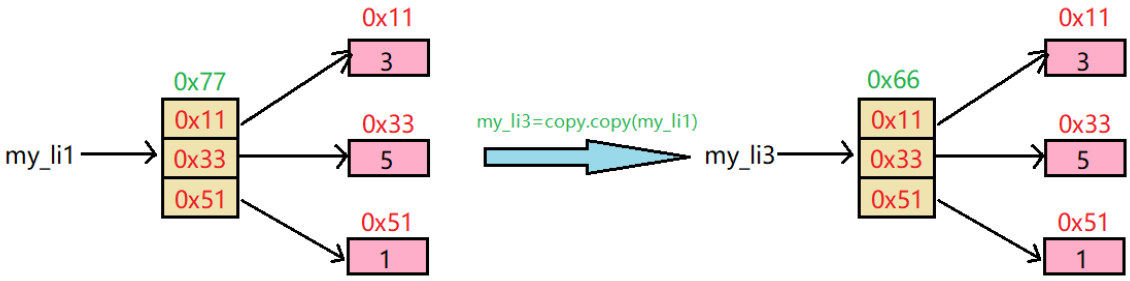


知识点17：浅拷贝-简单容器类型和嵌套容器类型数据【常握】

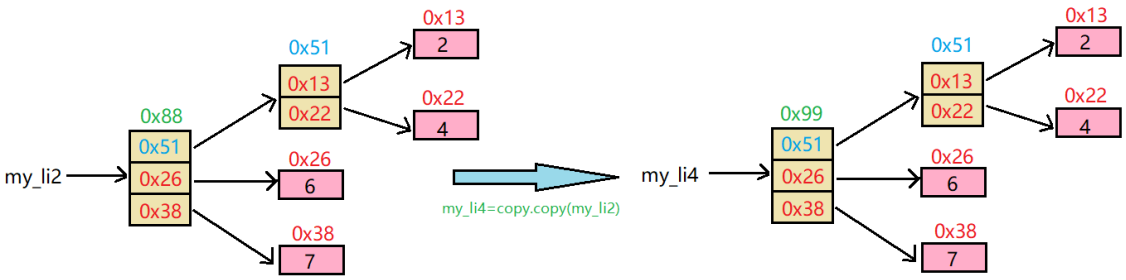
可变容器类型：列表

> 可变容器类型，进行浅拷贝时，只会对第一层数据重新开辟内存，进行拷贝。

```python
import copy

# 简单列表
my_li1 = [3, 5, 1]

# 浅拷贝
my_li3 = copy.copy(my_li1)

print('id(my_li1)：', id(my_li1))
print('id(my_li3)：', id(my_li3))

print('id(my_li1[0])：', id(my_li1[0]))
print('id(my_li3[0])：', id(my_li3[0]))
```



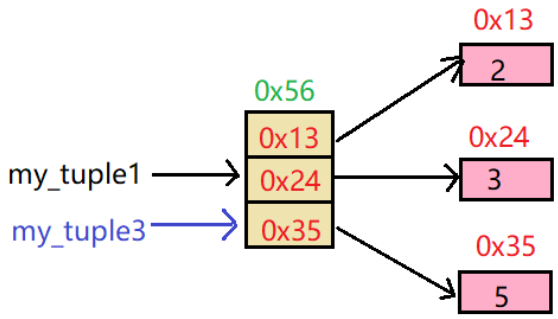

```python
# 嵌套列表
my_li2 = [[2, 4], 6, 7]

# 浅拷贝
my_li4 = copy.copy(my_li2)

print('id(my_li2)：', id(my_li2))
print('id(my_li4)：', id(my_li4))

print('id(my_li2[0])：', id(my_li2[0]))
print('id(my_li4[0])：', id(my_li4[0]))
```



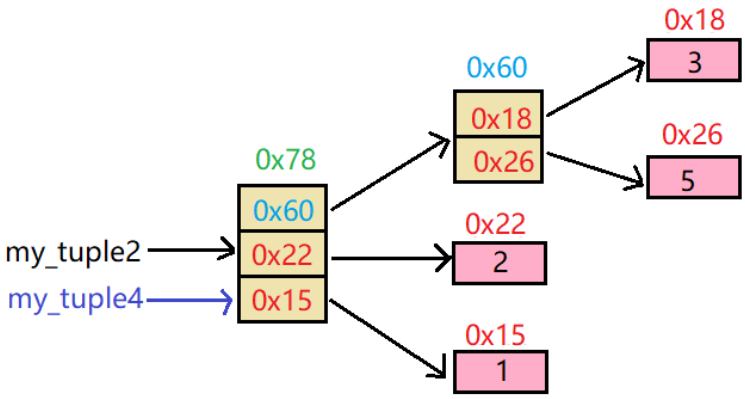
不可变容器类型：元祖

> 不可变容器类型，进行浅拷贝时，不会重新开辟内存，等同于=号赋值。

```python
# 简单元祖
my_tuple1 = (2, 3, 5)

# 浅拷贝
my_tuple3 = copy.copy(my_tuple1)

print('id(my_tuple1): ', id(my_tuple1))
print('id(my_tuple3): ', id(my_tuple3))
```




```python
# 嵌套元祖
my_tuple2 = ([3, 5], 2, 1)

# 浅拷贝
my_tuple4 = copy.copy(my_tuple2)

print('id(my_tuple2): ', id(my_tuple2))
print('id(my_tuple4): ', id(my_tuple4))
```



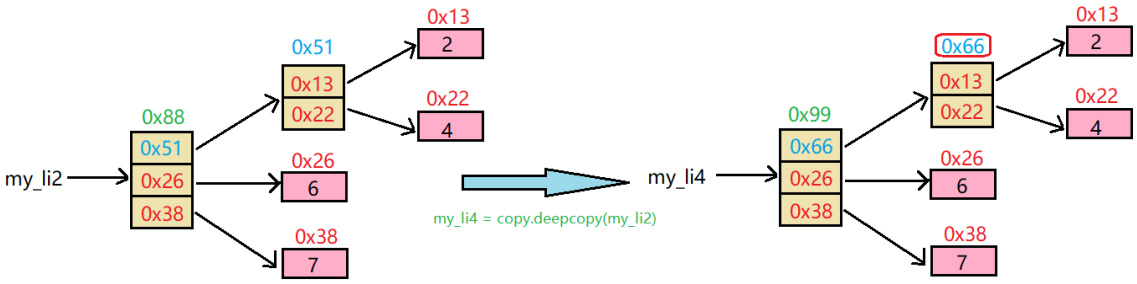
知识点18：深拷贝-简单容器类型和嵌套容器类型数据【常握】

可变容器类型：列表

> 可变容器类型，进行深拷贝时，每一层可变数据都会重新开辟内存，进行拷贝。

```python
# 简单列表
my_li1 = [3, 5, 1]

# 深拷贝
my_li3 = copy.deepcopy(my_li1)

print('id(my_li1[0]): ', id(my_li1[0]))
print('id(my_li3[0]): ', id(my_li3[0]))
```

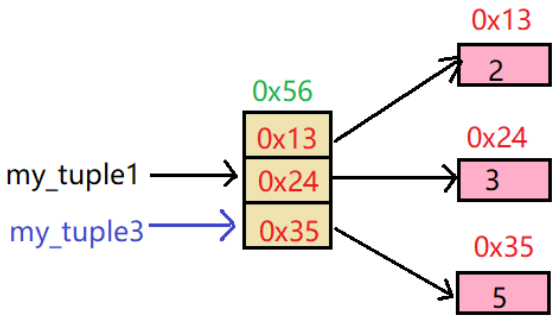

```
# 嵌套列表
my_li2 = [[2, 4], 6, 7]

# 深拷贝
my_li4 = copy.deepcopy(my_li2)

print('id(my_li2): ', id(my_li2))
print('id(my_li4): ', id(my_li4))

print('id(my_li2[0]): ', id(my_li2[0]))
print('id(my_li4[0]): ', id(my_li4[0]))
```



不可变容器类型：元祖

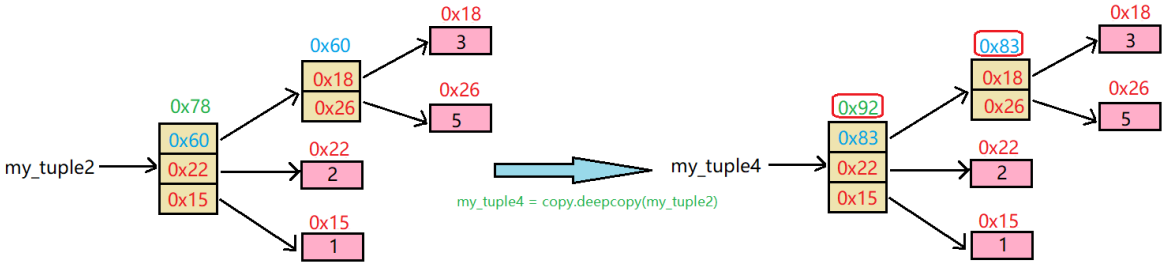
> **简单不可变容器类型，进行深拷贝时，不会重新开辟内存，等同于=号赋值。**

```
# 简单元祖
my_tuple1 = (2, 3, 5)

# 深拷贝
my_tuple3 = copy.deepcopy(my_tuple1)

print('id(my_tuple1): ', id(my_tuple1))
print('id(my_tuple3): ', id(my_tuple3))
```



> **嵌套不可变容器类型，进行深拷贝时，如果内层有可变类型时，则会重新开辟内存空间，进行拷贝**

```python
# 嵌套元祖
my_tuple2 = ([3, 5], 2, 1)

# 深拷贝
my_tuple4 = copy.deepcopy(my_tuple2)

print('id(my_tuple2)：', id(my_tuple2))
print('id(my_tuple4)：', id(my_tuple4))

print('id(my_tuple2[0])：', id(my_tuple2[0]))
print('id(my_tuple4[0])：', id(my_tuple4[0]))
```



知识点19：深拷贝和浅拷贝总结【常握】

浅拷贝：

- 可变容器类型，进行浅拷贝时，只会对第一层数据重新开辟内存，进行拷贝。
- 不可变容器类型，进行浅拷贝时，不会重新开辟内存，等同于=号赋值。

深拷贝：

- 可变容器类型，进行深拷贝时，每一层可变数据都会重新开辟内存，进行拷贝。
- 不可变容器类型
  - 简单不可变容器类型，进行深拷贝时，不会重新开辟内存，等同于=号赋值。
  - 嵌套不可变容器类型，进行深拷贝时，如果内层有可变类型时，则会重新开辟内存空间，进行拷贝。