**这一次，让你彻底明白Java的值传递和引用传递！**

2018年10月29日 08:39:00 [Java后端技术](https://me.csdn.net/bntX2jSQfEHy7) 阅读数 2199

本文旨在用最通俗的语言讲述最枯燥的基本知识

学过Java基础的人都知道：值传递和引用传递是初次接触Java时的一个难点，有时候记得了语法却记不得怎么实际运用，有时候会的了运用却解释不出原理，而且坊间讨论的话题又是充满争议：有的论坛帖子说Java只有值传递，有的博客说两者皆有；这让人有点摸不着头脑，下面我们就这个话题做一些探讨，对书籍、对论坛博客的说法，做一次考证，以得出信得过的答案。

其实，对于值传递和引用传递的语法和运用，百度一下，就能出来可观的解释和例子数目，或许你看一下例子好像就懂，但是当你参加面试，做一道这个知识点的笔试题时感觉自己会，胸有成熟的写了答案，却发现是错的，或者是你根本不会做。

是什么原因？

那是因为你对知识点没有了解透彻，只知道其皮毛。要熟读一个语法很简单，要理解一行代码也不难，但是能把学过的知识融会贯通，串联起来理解，那就是非常难了，在此，关于值传递和引用传递，小编会从以前学过的基础知识开始，从内存模型开始，一步步的引出值传递和引用传递的本质原理，故篇幅较长，知识点较多，望读者多有包涵。

1. 形参与实参

我们先来重温一组语法：

1. 形参：方法被调用时需要传递进来的参数，如：func(int a)中的a，它只有在func被调用期间a才有意义，也就是会被分配内存空间，在方法func执行完成后，a就会被销毁释放空间，也就是不存在了
2. 实参：方法被调用时是传入的实际值，它在方法被调用前就已经被初始化并且在方法被调用时传入。

举个栗子：

1public static void func(int a){  
2 a=20;  
3 System.out.println(a);  
4}  
5public static void main(String[] args) {  
6 int a=10;//变量  
7 func(a);  
8}

例子中  
int a=10;中的a在被调用之前就已经创建并初始化，在调用func方法时，他被当做参数传入，所以这个a是实参。  
而func(int a)中的a只有在func被调用时它的生命周期才开始，而在func调用结束之后，它也随之被JVM释放掉，，所以这个a是形参。

2. Java的数据类型

所谓数据类型，是编程语言中对内存的一种抽象表达方式，我们知道程序是由代码文件和静态资源组成，在程序被运行前，这些代码存在在硬盘里，程序开始运行，这些代码会被转成计算机能识别的内容放到内存中被执行。  
因此

数据类型实质上是用来定义编程语言中相同类型的数据的存储形式，也就是决定了如何将代表这些值的位存储到计算机的内存中。

所以，数据在内存中的存储，是根据数据类型来划定存储形式和存储位置的。  
那么  
Java的数据类型有哪些？

1. 基本类型：编程语言中内置的最小粒度的数据类型。它包括四大类八种类型：

4种整数类型：byte、short、int、long  
2种浮点数类型：float、double  
1种字符类型：char  
1种布尔类型：boolean

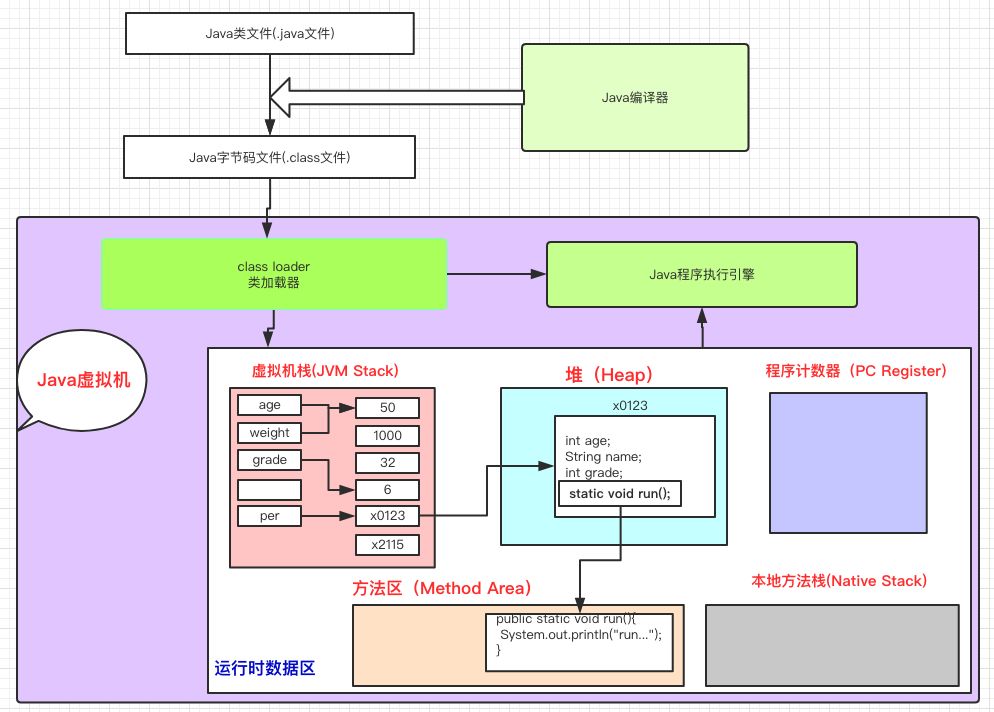
1. 引用类型：引用也叫句柄，引用类型，是编程语言中定义的在句柄中存放着实际内容所在地址的地址值的一种数据形式。它主要包括：

类  
接口  
数组

有了数据类型，JVM对程序数据的管理就规范化了，不同的数据类型，它的存储形式和位置是不一样的，要想知道JVM是怎么存储各种类型的数据，就得先了解JVM的内存划分以及每部分的职能。

3.JVM内存的划分及职能

Java语言本身是不能操作内存的，它的一切都是交给JVM来管理和控制的，因此Java内存区域的划分也就是JVM的区域划分，在说JVM的内存划分之前，我们先来看一下Java程序的执行过程，如下图：

  
有图可以看出：Java代码被编译器编译成字节码之后，JVM开辟一片内存空间（也叫运行时数据区），通过类加载器加到到运行时数据区来存储程序执行期间需要用到的数据和相关信息，在这个数据区中，它由以下几部分组成：

**1. 虚拟机栈**

**2. 堆**

**3. 程序计数器**

**4. 方法区**

**5. 本地方法栈**

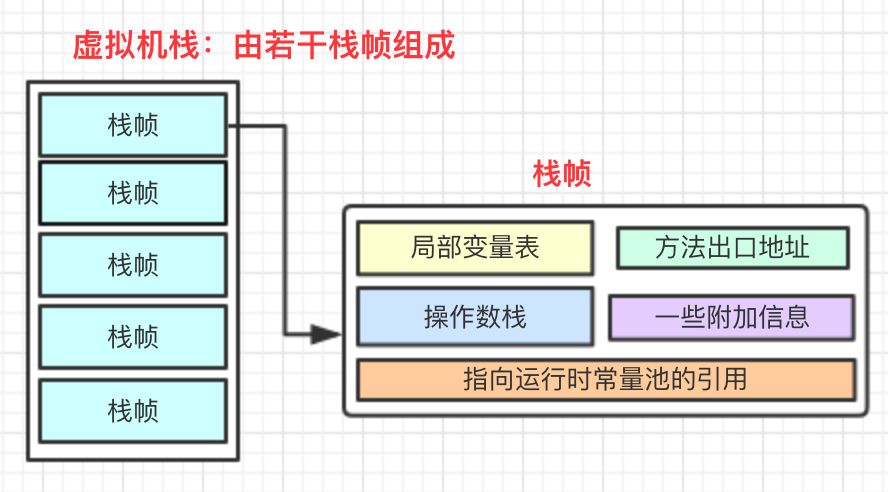
我们接着来了解一下每部分的原理以及具体用来存储程序执行过程中的哪些数据。

**1. 虚拟机栈**

虚拟机栈是Java方法执行的内存模型，栈中存放着栈帧，每个栈帧分别对应一个被调用的方法，方法的调用过程对应栈帧在虚拟机中入栈到出栈的过程。

栈是线程私有的，也就是线程之间的栈是隔离的；当程序中某个线程开始执行一个方法时就会相应的创建一个栈帧并且入栈（位于栈顶），在方法结束后，栈帧出栈。

下图表示了一个Java栈的模型以及栈帧的组成：

  
**栈帧**:是用于支持虚拟机进行方法调用和方法执行的数据结构，它是虚拟机运行时数据区中的虚拟机栈的栈元素。

每个栈帧中包括：

1. **局部变量表**:用来存储方法中的局部变量（非静态变量、函数形参）。当变量为基本数据类型时，直接存储值，当变量为引用类型时，存储的是指向具体对象的引用。
2. **操作数栈**:Java虚拟机的解释执行引擎被称为"基于栈的执行引擎"，其中所指的栈就是指操作数栈。
3. **指向运行时常量池的引用**:存储程序执行时可能用到常量的引用。
4. **方法返回地址**:存储方法执行完成后的返回地址。

**2. 堆：**

堆是用来存储对象本身和数组的，在JVM中只有一个堆，因此，堆是被所有线程共享的。

**3. 方法区：**

方法区是一块所有线程共享的内存逻辑区域，在JVM中只有一个方法区，用来存储一些线程可共享的内容，它是线程安全的，多个线程同时访问方法区中同一个内容时，只能有一个线程装载该数据，其它线程只能等待。

方法区可存储的内容有：类的全路径名、类的直接超类的权全限定名、类的访问修饰符、类的类型（类或接口）、类的直接接口全限定名的有序列表、常量池（字段，方法信息，静态变量，类型引用（class））等。

**4. 本地方法栈：**

本地方法栈的功能和虚拟机栈是基本一致的，并且也是线程私有的，它们的区别在于虚拟机栈是为执行Java方法服务的，而本地方法栈是为执行本地方法服务的。

**有人会疑惑：什么是本地方法？为什么Java还要调用本地方法？**

**5. 程序计数器：**

线程私有的。  
记录着当前线程所执行的字节码的行号指示器，在程序运行过程中，字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、异常处理、线程恢复等基础功能都需要依赖计数器完成。

4. 数据如何在内存中存储？

从上面程序运行图我们可以看到，JVM在程序运行时的内存分配有三个地方：

* 堆
* 栈
* 静态方法区
* 常量区

相应地，每个存储区域都有自己的内存分配策略：

* 堆式：
* 栈式
* 静态

我们已经知道：Java中的数据类型有基本数据类型和引用数据类型，那么这些数据的存储都使用哪一种策略呢？  
这里要分以下的情况进行探究：

**1. 基本数据类型的存储：**

* A. 基本数据类型的局部变量
* B. 基本数据类型的成员变量
* C. 基本数据类型的静态变量

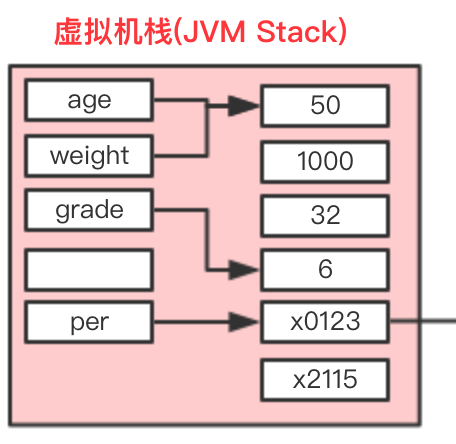
**2. 引用数据类型的存储**

**1. 基本数据类型的存储**

我们分别来研究一下：

**A.基本数据类型的局部变量**

1. 定义基本数据类型的局部变量以及数据都是直接存储在内存中的栈上，也就是前面说到的**“虚拟机栈”**，数据本身的值就是存储在栈空间里面。



如上图，在方法内定义的变量直接存储在栈中，如

1int age=50;  
2int weight=50;  
3int grade=6;

当我们写“int age=50；”，其实是分为两步的：

1int age;//定义变量  
2age=50;//赋值

首先JVM创建一个名为age的变量，存于局部变量表中，然后去栈中查找是否存在有字面量值为50的内容，如果有就直接把age指向这个地址，如果没有，JVM会在栈中开辟一块空间来存储“50”这个内容，并且把age指向这个地址。因此我们可以知道：  
**我们声明并初始化基本数据类型的局部变量时，变量名以及字面量值都是存储在栈中，而且是真实的内容。**

我们再来看“int weight=50；”，按照刚才的思路：字面量为50的内容在栈中已经存在，因此weight是直接指向这个地址的。由此可见：**栈中的数据在当前线程下是共享的**。

那么如果再执行下面的代码呢？

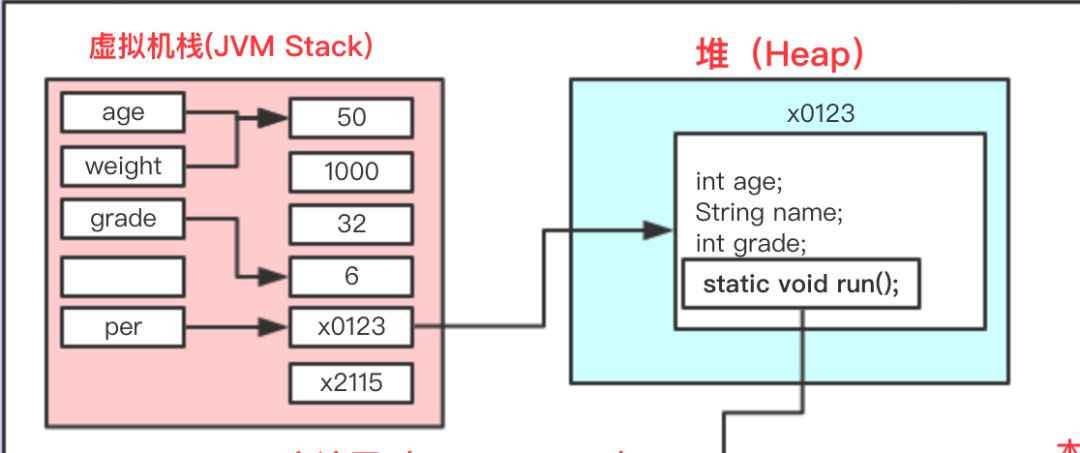
1weight=40；

当代码中重新给weight变量进行赋值时，JVM会去栈中寻找字面量为40的内容，发现没有，就会开辟一块内存空间存储40这个内容，并且把weight指向这个地址。由此可知：

**基本数据类型的数据本身是不会改变的，当局部变量重新赋值时，并不是在内存中改变字面量内容，而是重新在栈中寻找已存在的相同的数据，若栈中不存在，则重新开辟内存存新数据，并且把要重新赋值的局部变量的引用指向新数据所在地址。**

**B. 基本数据类型的成员变量**

成员变量：顾名思义，就是在类体中定义的变量。  
看下图：



我们看per的地址指向的是堆内存中的一块区域，我们来还原一下代码：

1public class Person{  
 2  private int age;  
 3  private String name;  
 4  private int grade;  
 5//篇幅较长，省略setter getter方法  
 6  static void run(){  
 7     System.out.println("run....");   
 8   };  
 9}  
10  
11//调用  
12Person per=new Person();

同样是局部变量的age、name、grade却被存储到了堆中为per对象开辟的一块空间中。因此可知：**基本数据类型的成员变量名和值都存储于堆中，其生命周期和对象的是一致的。**

**C. 基本数据类型的静态变量**

前面提到**方法区**用来存储一些共享数据，因此**基本数据类型的静态变量名以及值存储于方法区的运行时常量池中，静态变量随类加载而加载，随类消失而消失**

**2. 引用数据类型的存储:**

上面提到：堆是用来存储对象本身和数组，而引用（句柄）存放的是实际内容的地址值，因此通过上面的程序运行图，也可以看出，当我们定义一个对象时

1Person per=new Person();

实际上，它也是有两个过程：

1Person per;//定义变量  
2per=new Person();//赋值

在执行Person per;时，JVM先在虚拟机栈中的变量表中开辟一块内存存放per变量，在执行per=new Person()时，JVM会创建一个Person类的实例对象并在堆中开辟一块内存存储这个实例，同时把实例的地址值赋值给per变量。因此可见：  
**对于引用数据类型的对象/数组，变量名存在栈中，变量值存储的是对象的地址，并不是对象的实际内容。**

6. 值传递和引用传递

前面已经介绍过形参和实参，也介绍了数据类型以及数据在内存中的存储形式，接下来，就是文章的主题：值传递和引用的传递。

**值传递：**  
在方法被调用时，实参通过形参把它的内容副本传入方法内部，此时形参接收到的内容是实参值的一个拷贝，因此在方法内对形参的任何操作，都仅仅是对这个副本的操作，不影响原始值的内容。

来看个例子：

1public static void valueCrossTest(int age,float weight){  
 2    System.out.println("传入的age："+age);  
 3    System.out.println("传入的weight："+weight);  
 4    age=33;  
 5    weight=89.5f;  
 6    System.out.println("方法内重新赋值后的age："+age);  
 7    System.out.println("方法内重新赋值后的weight："+weight);  
 8    }  
 9  
10//测试  
11public static void main(String[] args) {  
12        int a=25;  
13        float w=77.5f;  
14        valueCrossTest(a,w);  
15        System.out.println("方法执行后的age："+a);  
16        System.out.println("方法执行后的weight："+w);  
17}

输出结果：

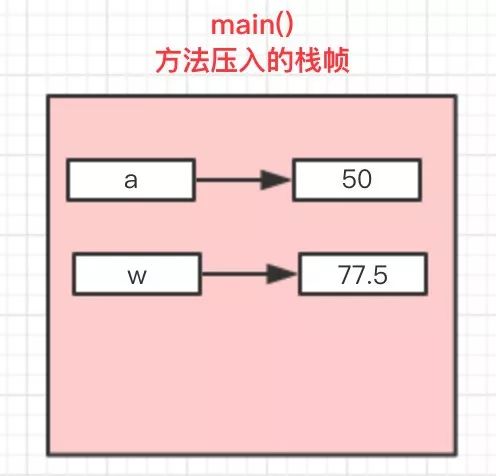
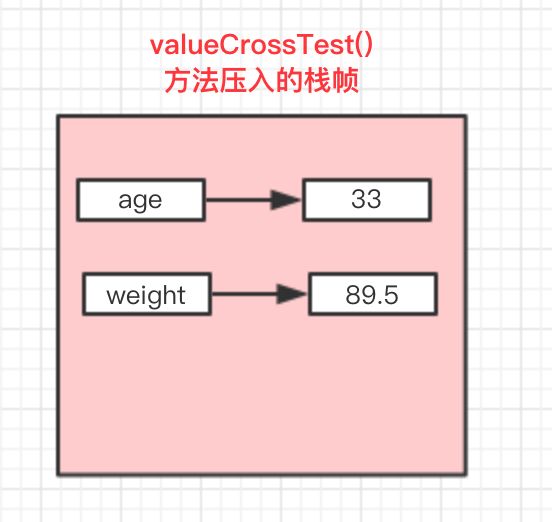
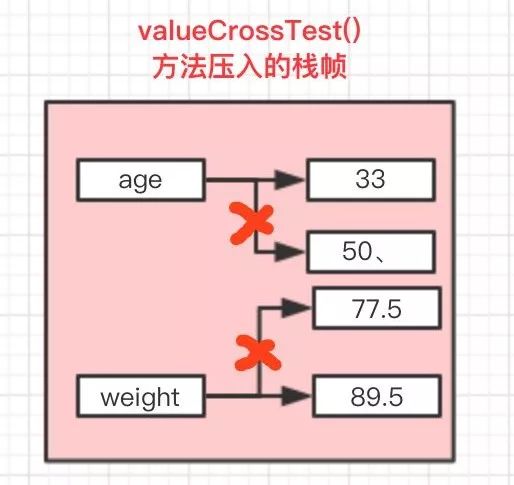
1传入的age：25  
2传入的weight：77.5  
3  
4方法内重新赋值后的age：33  
5方法内重新赋值后的weight：89.5  
6  
7方法执行后的age：25  
8方法执行后的weight：77.5

从上面的打印结果可以看到：  
a和w作为实参传入valueCrossTest之后，无论在方法内做了什么操作，最终a和w都没变化。

这是什么造型呢？！！

下面我们根据上面学到的知识点，进行详细的分析：

首先程序运行时，调用mian()方法，此时JVM为main()方法往虚拟机栈中压入一个栈帧，即为当前栈帧，用来存放main()中的局部变量表(包括参数)、操作栈、方法出口等信息，如a和w都是mian()方法中的局部变量，因此可以断定，a和w是躺着mian方法所在的栈帧中  
如图：

  
而当执行到valueCrossTest()方法时，JVM也为其往虚拟机栈中压入一个栈，即为当前栈帧，用来存放valueCrossTest()中的局部变量等信息，因此age和weight是躺着valueCrossTest方法所在的栈帧中，而他们的值是从a和w的值copy了一份副本而得，如图：  
因而可以a和age、w和weight对应的内容是不一致的，所以当在方法内重新赋值时，实际流程如图：  
也就是说，age和weight的改动，只是改变了当前栈帧（valueCrossTest方法所在栈帧）里的内容，当方法执行结束之后，这些局部变量都会被销毁，mian方法所在栈帧重新回到栈顶，成为当前栈帧，再次输出a和w时，依然是初始化时的内容。  
因此：  
**值传递传递的是真实内容的一个副本，对副本的操作不影响原内容，也就是形参怎么变化，不会影响实参对应的内容。**

**引用传递：**  
”引用”也就是指向真实内容的地址值，在方法调用时，实参的地址通过方法调用被传递给相应的形参，在方法体内，形参和实参指向通愉快内存地址，对形参的操作会影响的真实内容。

举个栗子：  
先定义一个对象：

1public class Person {  
 2        private String name;  
 3        private int age;  
 4  
 5        public String getName() {  
 6            return name;  
 7        }  
 8        public void setName(String name) {  
 9            this.name = name;  
10        }  
11        public int getAge() {  
12            return age;  
13        }  
14        public void setAge(int age) {  
15            this.age = age;  
16        }  
17}

我们写个函数测试一下：

1public static void PersonCrossTest(Person person){  
 2        System.out.println("传入的person的name："+person.getName());  
 3        person.setName("我是张小龙");  
 4        System.out.println("方法内重新赋值后的name："+person.getName());  
 5    }  
 6//测试  
 7public static void main(String[] args) {  
 8        Person p=new Person();  
 9        p.setName("我是马化腾");  
10        p.setAge(45);  
11        PersonCrossTest(p);  
12        System.out.println("方法执行后的name："+p.getName());  
13}

输出结果：

1传入的person的name：我是马化腾  
2方法内重新赋值后的name：我是张小龙  
3方法执行后的name：我是张小龙

可以看出，person经过personCrossTest()方法的执行之后，内容发生了改变，这印证了上面所说的**“引用传递”**，对形参的操作，改变了实际对象的内容。

那么，到这里就结题了吗？  
不是的，没那么简单，  
能看得到想要的效果  
是因为刚好选对了例子而已！！！

下面我们对上面的例子稍作修改，加上一行代码，

1public static void PersonCrossTest(Person person){  
2        System.out.println("传入的person的name："+person.getName());  
3        person=new Person();//加多此行代码  
4        person.setName("我是张小龙");  
5        System.out.println("方法内重新赋值后的name："+person.getName());  
6    }

输出结果：

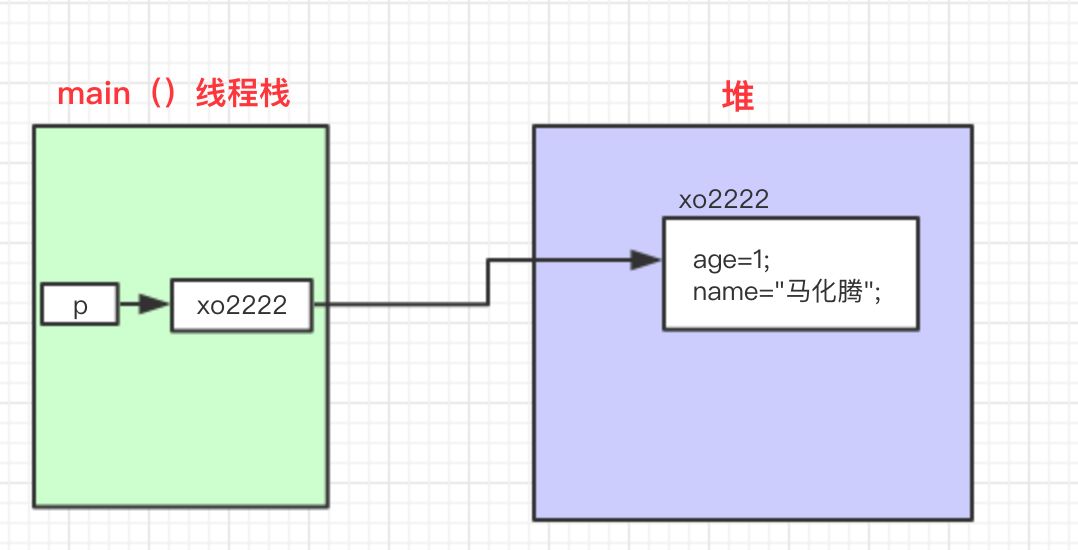
1传入的person的name：我是马化腾  
2方法内重新赋值后的name：我是张小龙  
3方法执行后的name：我是马化腾

`  
为什么这次的输出和上次的不一样了呢？  
看出什么问题了吗？

按照上面讲到JVM内存模型可以知道，对象和数组是存储在Java堆区的，而且堆区是共享的，因此程序执行到main（）方法中的下列代码时

1Person p=new Person();  
2        p.setName("我是马化腾");  
3        p.setAge(45);  
4        PersonCrossTest(p);

JVM会在堆内开辟一块内存，用来存储p对象的所有内容，同时在main（）方法所在线程的栈区中创建一个引用p存储堆区中p对象的真实地址，如图：

  
当执行到PersonCrossTest()方法时，因为方法内有这么一行代码：

1person=new Person();

JVM需要在堆内另外开辟一块内存来存储new Person()，假如地址为“xo3333”，那此时形参person指向了这个地址，假如真的是引用传递，那么由上面讲到：**引用传递中形参实参指向同一个对象，形参的操作会改变实参对象的改变**。

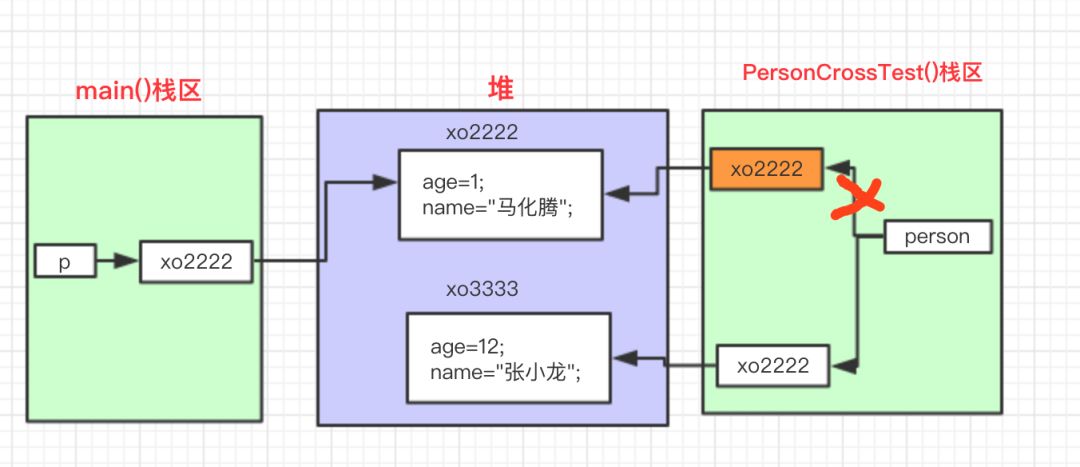
可以推出：实参也应该指向了新创建的person对象的地址，所以在执行PersonCrossTest()结束之后，最终输出的应该是后面创建的对象内容。

然而实际上，最终的输出结果却跟我们推测的不一样，最终输出的仍然是一开始创建的对象的内容。

由此可见：**引用传递，在Java中并不存在。**

但是有人会疑问：为什么第一个例子中，在方法内修改了形参的内容，会导致原始对象的内容发生改变呢？

这是因为：**无论是基本类型和是引用类型，在实参传入形参时，都是值传递，也就是说传递的都是一个副本，而不是内容本身。**



有图可以看出，方法内的形参person和实参p并无实质关联，它只是由p处copy了一份指向对象的地址，此时：

**p和person都是指向同一个对象**。

因此在第一个例子中，对形参p的操作，会影响到实参对应的对象内容。而在第二个例子中，当执行到new Person()之后，JVM在堆内开辟一块空间存储新对象，并且把person改成指向新对象的地址，此时：

**p依旧是指向旧的对象，person指向新对象的地址。**

所以此时**对person的操作，实际上是对新对象的操作，于实参p中对应的对象毫无关系**。

结语

因此可见：在Java中所有的参数传递，不管基本类型还是引用类型，都是值传递，或者说是副本传递。  
只是在传递过程中：

**如果是对基本数据类型的数据进行操作，由于原始内容和副本都是存储实际值，并且是在不同的栈区，因此形参的操作，不影响原始内容。**

**如果是对引用类型的数据进行操作，分两种情况，一种是形参和实参保持指向同一个对象地址，则形参的操作，会影响实参指向的对象的内容。一种是形参被改动指向新的对象地址（如重新赋值引用），则形参的操作，不会影响实参指向的对象的内容。**

以上为小编关于“值传递和引用传递”问题的思考和论证，对于这个问题，历来都是多有争论，在此希望和读者一起探讨和学习，有不同意见或者建议请假小编微信：sisi-ceo。理性评论，不喜勿喷。