Java　Java 编程　Java EE　Java 程序员

关注者 574　被浏览 96,426

Java 到底是值传递还是引用传递？

public class TestMain { public static void main(String[] args) { List&l...显示全部

关注问题　写回答　2 条评论　分享　邀请回答　···

61 个回答

默认排序

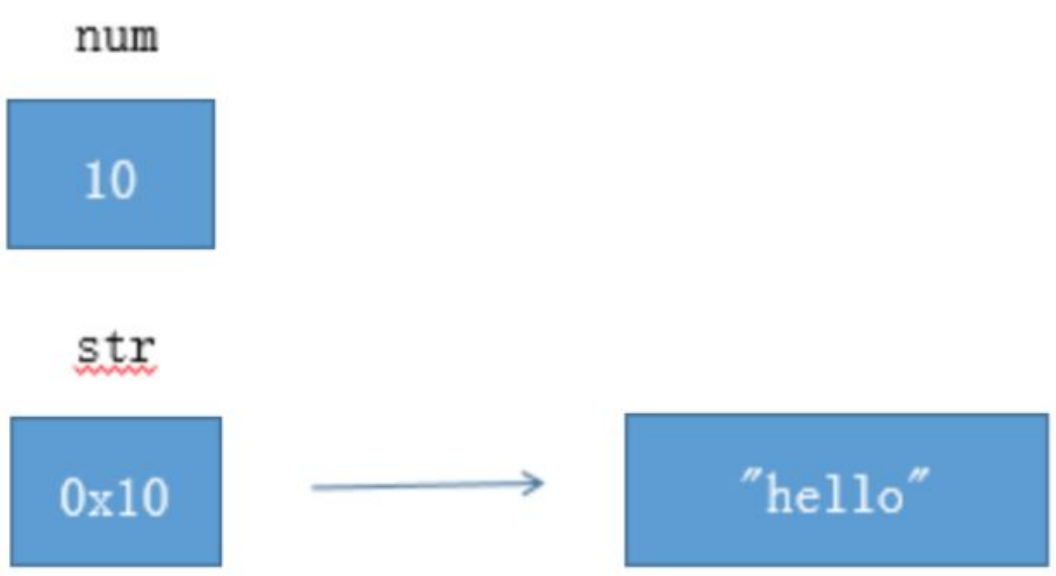
Intopass
程序员，近期沉迷于动漫ING

821 人赞同了该回答

首先，不要纠结于 Pass By Value 和 Pass By Reference 的字面上的意义，否则很容易陷入所谓的“一切传引用其实本质上是传值”这种并不能解决问题无意义论战中。
更何况，要想知道Java到底是传值还是传引用，起码你要先知道传值和传引用的准确含义吧？可是如果你已经知道了这两个名字的准确含义，那么你自己就能判断Java到底是传值还是传引用。
这就好像用大学的名词来解释高中的题目，对于初学者根本没有任何意义。

一：搞清楚 基本类型 和 引用类型的不同之处

```
int num = 10;
String str = "hello";
```

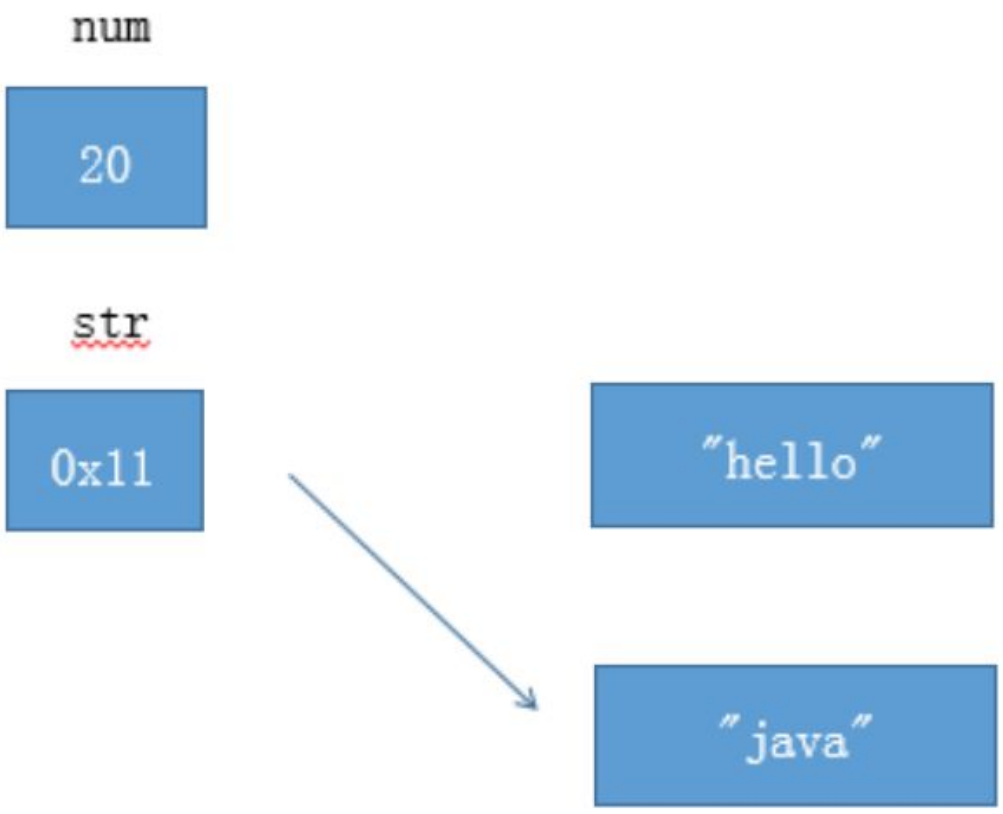


如图所示，num是基本类型，值就直接保存在变量中。而str是引用类型，变量中保存的只是实际对象的地址。一般称这种变量为"引用"，引用指向实际对象，实际对象中保存着内容。

二：搞清楚赋值运算符（=）的作用

```
num = 20;
str = "java";
```

▲ 75　12 条评论　收藏　感谢



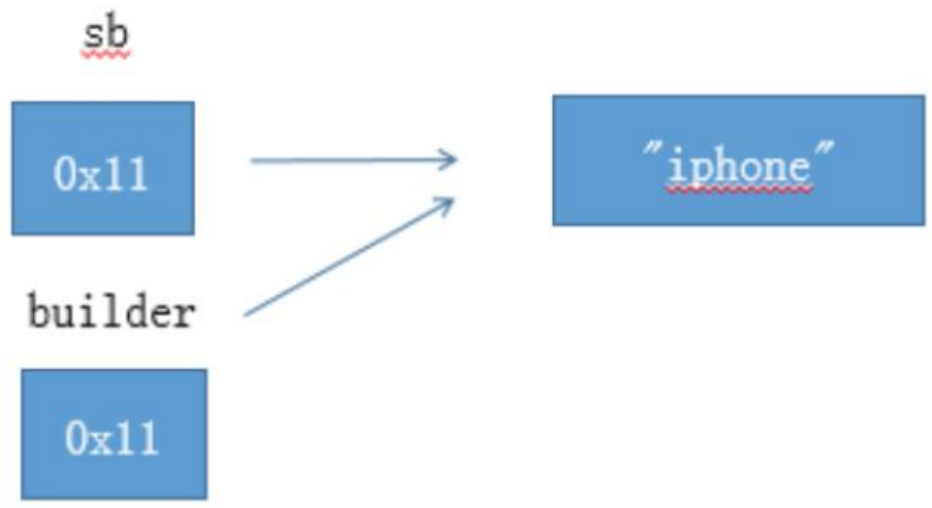
对于基本类型 num ，赋值运算符会直接改变变量的值，原来的值被覆盖掉。
对于引用类型 str，赋值运算符会改变引用中所保存的地址，原来的地址被覆盖掉。**但是原来的对象不会被改变（重要）。**
如上图所示，"hello" 字符串对象没有被改变。（没有被任何引用所指向的对象是垃圾，会被垃圾回收器回收）

三：调用方法时发生了什么？**参数传递基本上就是赋值操作。**

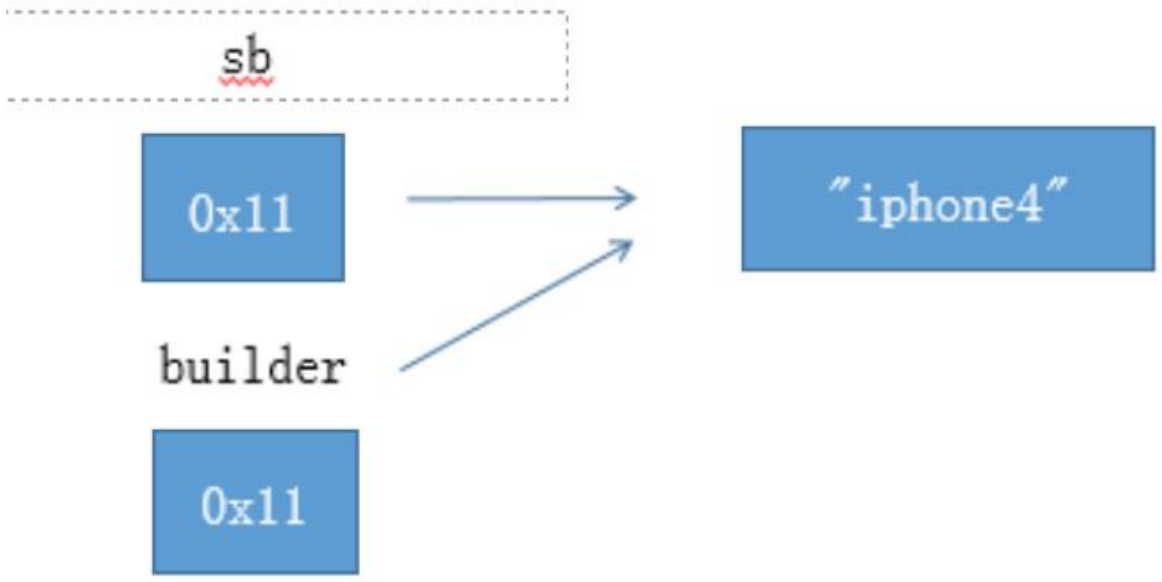
```
第一个例子：基本类型
void foo(int value) {
    value = 100;
}
foo(num); // num 没有被改变

第二个例子：没有提供改变自身方法的引用类型
void foo(String text) {
    text = "windows";
}
foo(str); // str 也没有被改变

第三个例子：提供了改变自身方法的引用类型
StringBuilder sb = new StringBuilder("iphone");
void foo(StringBuilder builder) {
    builder.append("4");
}
foo(sb); // sb 被改变了，变成了"iphone4"。

第四个例子：提供了改变自身方法的引用类型，但是不使用，而是使用赋值运算符。
StringBuilder sb = new StringBuilder("iphone");
void foo(StringBuilder builder) {
    builder = new StringBuilder("ipad");
}
foo(sb); // sb 没有被改变，还是 "iphone"。
```

重点理解为什么，第三个例子和第四个例子结果不同？

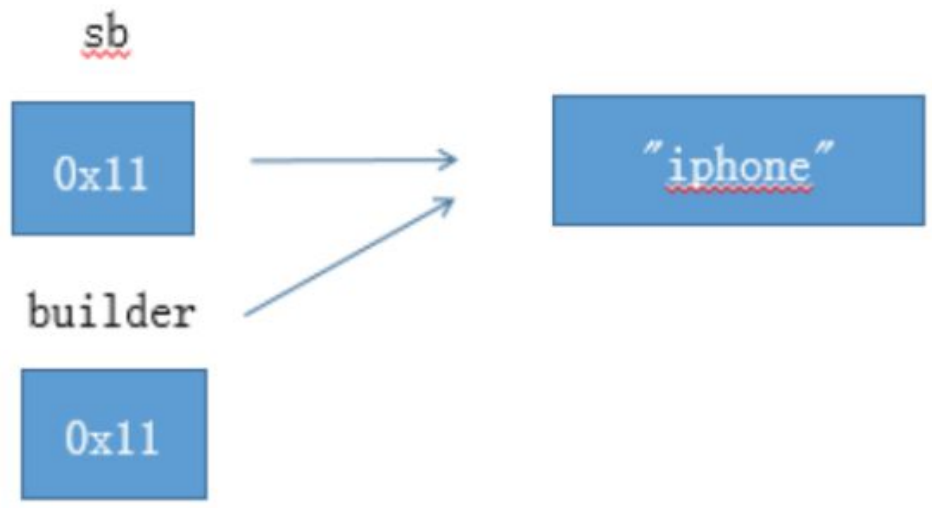
下面是第三个例子的图解：



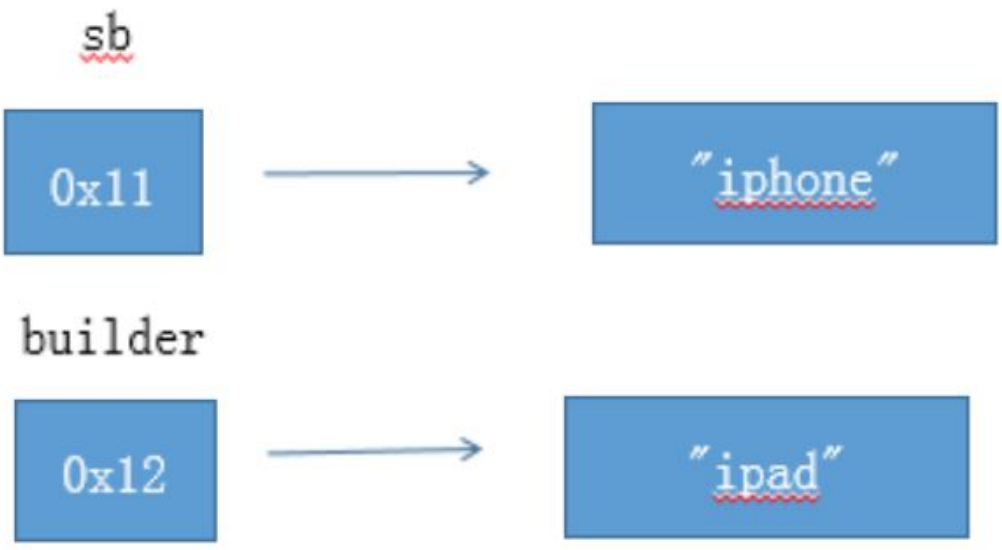
builder.append("4")之后



下面是第四个例子的图解：



builder = new StringBuilder("ipad"); 之后



2018年1月31日添加部分内容：

这个答案点赞的不少，虽然当时回答时并没有讲的特别详细，今天就稍微多讲一些各种类型数据在内存中的存储方式。

从局部变量/方法参数开始讲起：

局部变量和方法参数在jvm中的储存方法是相同的，都是在栈上开辟空间来储存的，随着进入方法开辟，退出方法回收。以32位JVM为例，boolean/byte/short/char/int/float以及引用都是分配4字节空间，long/double分配8字节空间。对于每个方法来说，最多占用多少空间是一定的，这在编译时就可以计算好。

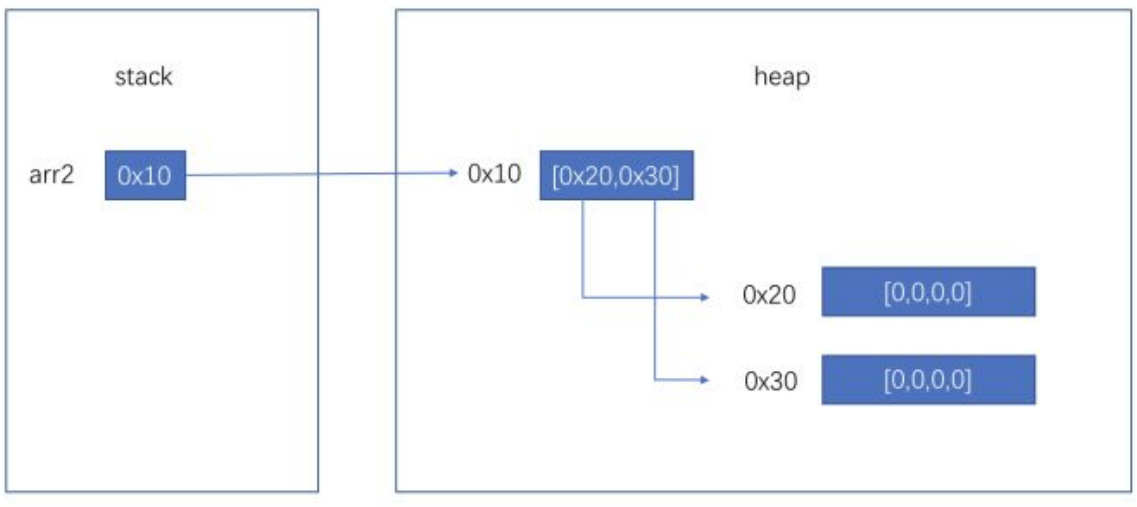
我们都知道JVM内存模型中有，stack和heap的存在，但是更准确的说，是每个线程都分配一个独享的stack，所有线程共享一个heap。对于每个方法的局部变量来说，是绝对无法被其他方法，甚至其他线程的同一方法所访问到的，更遑论修改。

当我们在方法中声明一个 int i = 0，或者 Object obj = null 时，仅仅涉及stack，不影响到heap，当我们 new Object() 时，会在heap中开辟一段内存并初始化Object对象。当我们将这个对象赋予obj变量时，仅仅是stack中代表obj的那4个字节变更为这个对象的地址。

数组类型引用和对象：

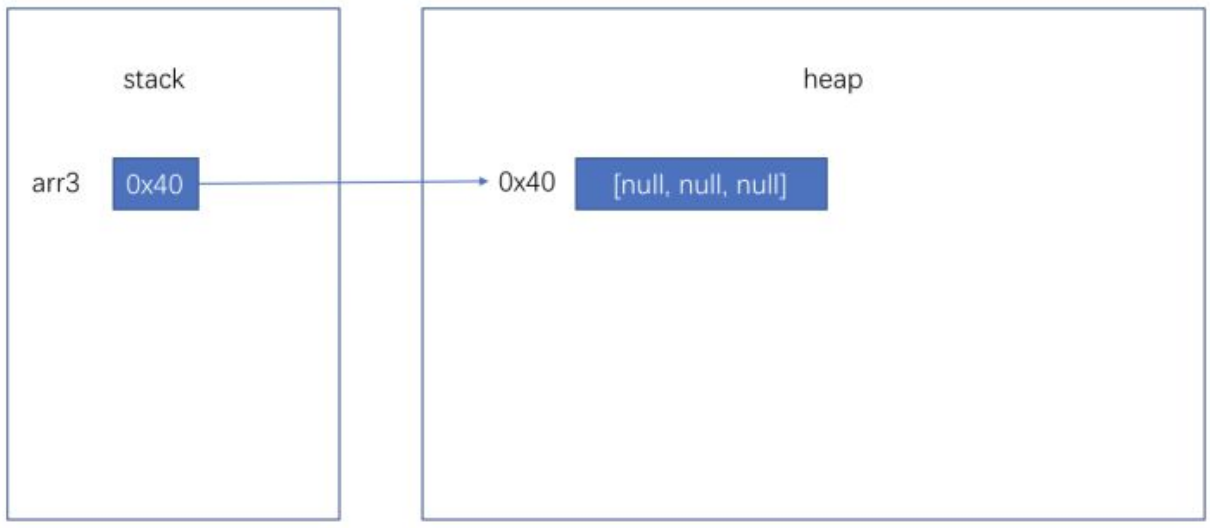
当我们声明一个数组时，如int[] arr = new int[10]，因为数组也是对象，arr实际上是引用，stack上仅仅占用4字节空间，new int[10]会在heap中开辟一个数组对象，然后arr指向它。

当我们声明一个二维数组时，如 int[][] arr2 = new int[2][4]，arr2同样仅在stack中占用4个字节，会在内存中开辟一个长度为2的，类型为int[]的数组，然后arr2指向这个数组。这个数组内部有两个引用（大小为4字节），分别指向两个长度为4的类型为int的数组。

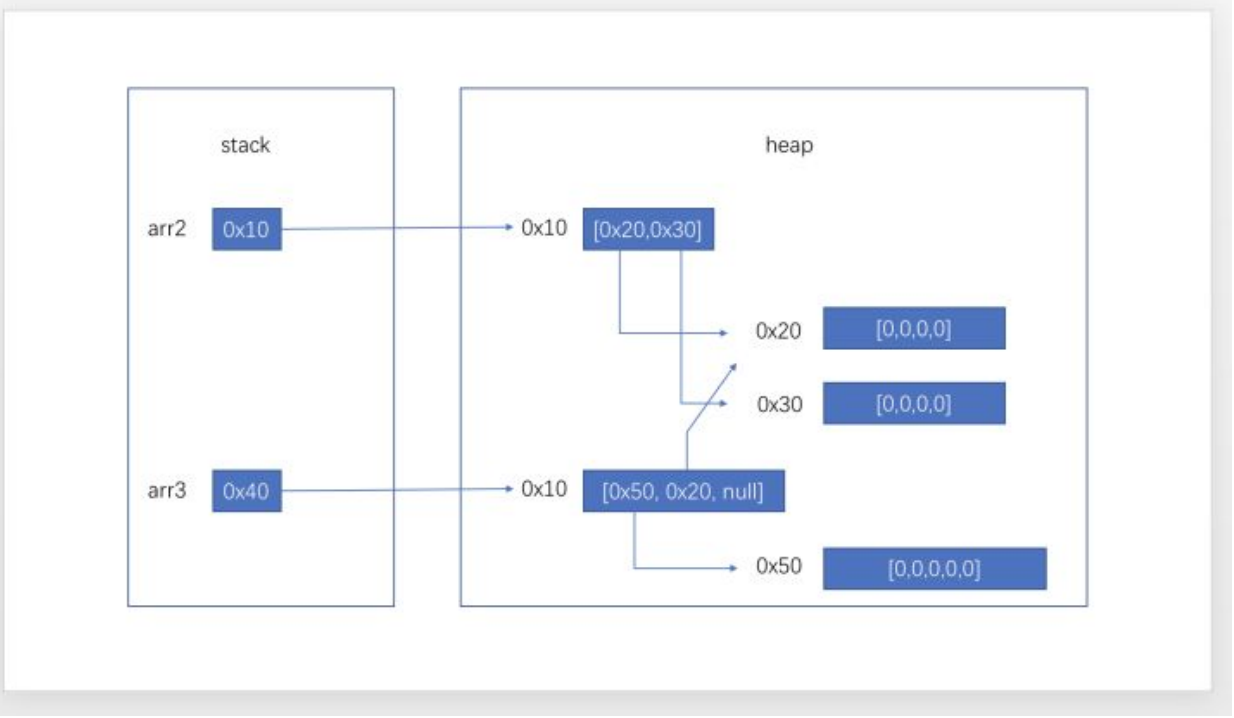


所以当我们传递一个数组引用给一个方法时，数组的元素是可以被改变的，但是无法让数组引用指向新的数组。

你还可以这样声明：int[][] arr3 = new int[3][]，这时内存情况如下图



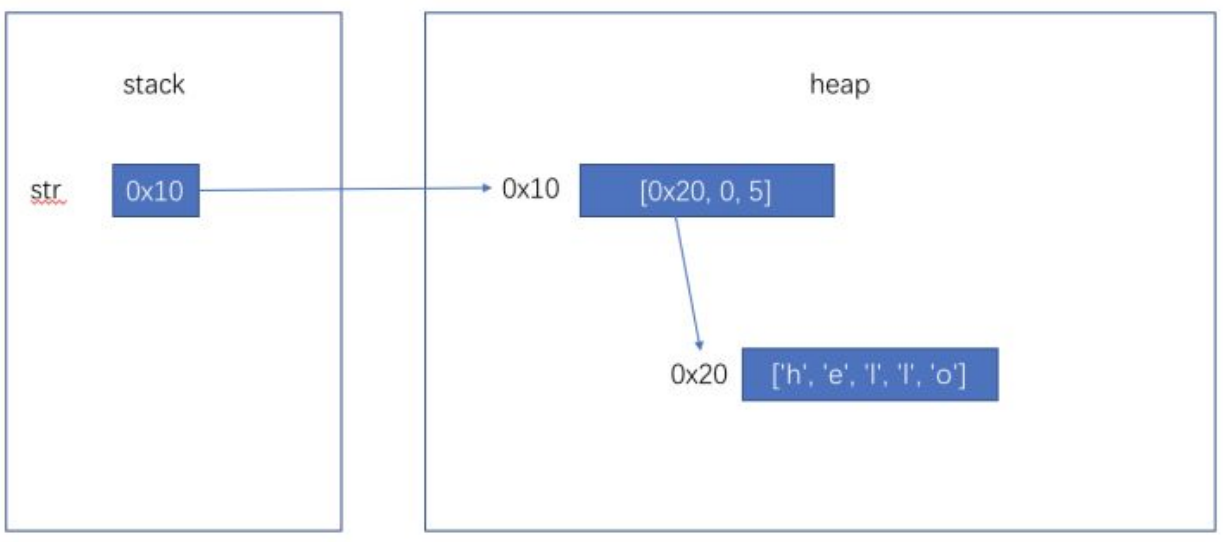
你还可以这样 arr3[0] = new int [5]; arr3[1] = arr2[0];



关于String：

原本回答中关于String的图解是简化过的，实际上String对象内部仅需要维护三个变量，char[] chars, int startIndex, int length。而chars在某些情况下是可以共用的。但是因为String被设计成为了不可变类型，所以你思考时把String对象简化考虑也是可以的。

String str = new String("hello")



当然某些JVM实现会把"hello"字面量生成的String对象放到常量池中，而常量池中的对象可以实际分配在heap中，有些实现也许会分配在方法区，当然这对我们理解影响不大。

编辑于 2018-02-01

Yolanda
程序员

36 人赞同了该回答

为什么 Java 只有值传递，但 C# 既有值传递，又有引用传递，这种语言设计有哪些好处？ - 编程里面：Hugo Gu的回答我十分的赞同。再次感谢作者。
作者：Hugo Gu
链接：为什么 Java 只有值传递，但 C# 既有值传递，又有引用
知乎用户的回答

来源：知乎

先强调这个问题前半句是真命题。说问题逻辑有问题，说一切都是值传递，都是没理解什么叫引用传递和值传递。

虽然这个问题根本就没有在问“Java是不是值传递”，但是看完其它答案发现，如果不先解释清楚到底什么是值传递，什么是引用传递，后面的好处也无从谈起。只关心好处的请拉到最后。

第一种误解是：Java是引用传递。（这么理解的人，大体会解释说Java的形参是对象的引用所以才叫引用传递。这个解释的错误在于：引用传递这个词不是这个意思，这个词是形容调用方式，而不是参数本质的类型的。所以，即使有人因为明白引用本身也是个值，然后觉得Java其实是值传递了，这种理解也是错的。你这种理解，叫“传递的是值”，而非“值传递”。后面展开。）

第二种误解是：值类型是值传递，引用类型用的是引用传递。

第三种误解是：认为所有的都是值传递，因为引用本质上也是个值，本质就是个指针嘛。

第四种误解是：常出现在C++程序员中，声明的参数是引用类型的，就是引用传递；声明的参数是一般类型或指针的就是值传递。（也有人把指针归为引用传递，其实它比较特殊，无论你归哪边都是错的。）

值传递与引用传递，在计算机领域是专有名词，如果你没有专门了解过，一般很难自行悟出其含义。**而且在理解下面的解释时，请不要把任何概念往你所熟悉的语言功能上套。很容易产生误解。**比如Reference，请当个全新的概念，它和C#引用类型中的引用，和C++的&，一点儿关系都没有。

值传递和引用传递，属于函数调用时参数的求值策略(Evaluation Strategy**)，这是对调用函数时，求值和传值的方式的描述，而非传递的内容的类型**（内容指：是值类型还是引用类型，是值还是指针）。值类型/引用类型，是用于区分两种内存分配方式，值类型在调用栈上分配，引用类型在堆上分配。（不要问我引用类型里定义个值类型成员或反之会发生什么，这不在这个本文的讨论范畴内，而且你看完之后，你应该可以自己想明白）。一个描述内存分配方式，一个描述参数求值策略，两者之间无任何依赖或约束关系。

在函数调用过程中，调用方提供实参，这些实参可以是常量：

Call(1);

也可以是变量：

Call(x);

也可以是他们的组合：

Call(2 * x + 1);

也可以是对其它函数的调用：

Call(GetNumber());

但是所有这些实参的形式，都统称为表达式(Expression)。求值（Evaluation）即是指对这些表达式的简化并求解其值的过程。

求值策略(值传递和引用传递)的关注的点在于，这些表达式在调用函数的过程中，求值的时机、值的形式的选取等问题。求值的时机，可以是在函数调用前，也可以是在函数调用后，由被调用者自己求值。这里所谓调用后求值，可以理解为Lazy Load或On Demand的一种求值方式。

而且，除了值传递和引用传递，还有一些其它的求值策略。这些求值策略的划分依据是：求值的时机（调用前还是调用中）和值本身的传递方式。详见下表：

<img src="pic4.zhimg.com/9d4d1d25..." data-rawwidth="524" data-rawheight="101" class="origin_image zh-lightbox-thumb" width="524" data-original="pic4.zhimg.com/9d4d1d25...">

求值策略	求值时间	传值方式
值传递(Pass by value)	调用前	值的结果（是原值的副本）
引用传递(Pass by reference)	调用前	原值（原始对象，无副本）
名传递(Pass by name)	调用后（用到才求值）	与值无关的一个名

看到这里的名传递，可能就有人联想到C++里的别名(alias)，其实也是两码事儿。语言层直接支持名传递的语言很不主流，但是在C#中，名传递的行为可以用Func<T>来模拟，说到这儿应该能大概猜出名传递的大致行为了。不过这不是重点，重点是值传递和引用传递。上面给出的传值方式的表述有些单薄，下表列出了一些二者在行为表象上的区别。

<img src="pic1.zhimg.com/47590cd6..." data-rawwidth="474" data-rawheight="73" class="origin_image zh-lightbox-thumb" width="474" data-original="pic1.zhimg.com/47590cd6...">

	值传递	引用传递
根本区别	会创建副本（Copy）	不创建副本
所以	函数中无法**改变**原始对象	函数中可以**改变**原始对象

这里的改变不是指mutate, 而是change，指把一个变量指向另一个对象，而不是指仅仅改变属性或是成员什么的（**如Java，所以说Java是Pass by value，原因是它调用时Copy，实参不能指向另一个对象，而不是因为被传递的东西本质上是个Value，这么讲计算机上什么不是Value?**）。

这些行为，与参数类型是值类型还是引用类型无关。对于值传递，无论是值类型还是引用类型，都会在调用栈上创建一个副本，不同是，对于值类型而言，这个副本就是整个原始值的复制。而对于引用类型而言，由于引用类型的实例在堆中，在栈上只有它的一个引用（一般情况下是指针），其副本也只是这个引用的复制，而不是整个原始对象的复制。

这便引出了值类型和引用类型（这不是在说值传递）的最大区别：值类型用做参数会被复制，但是很多人误以为这个区别是值类型的特性。其实这是值传递带来的效果，和值类型本身没有关系。只是最终结果是这样。

求值策略定义的是函数调用时的行为，并不对具体实现方式做要求，但是指针由于其汇编级支持的特性，成为实现引用传递方式的首选。但是纯理论上，你完全可以不用指针，比如用一个全局的参数名到对象地址的HashTable来实现引用传递，只是这样效率太低，所以根本没有哪个编程语言会这样做。（自己写来玩玩的不算）

综上所述，对于Java的函数调用方式最准确的描述是：**参数藉由值传递方式，传递的值是个引用。（句中两个“值”不是一个意思，第一个值是evaluation result，第二个值是value content）**

由于这个描述太绕，而且在字面上与Java总是传引用的事实冲突。于是对于Java，Python、Ruby、JavaScript等语言使用的这种求值策略，起了一个更贴切名字，叫Call by sharing。这个名字诞生于40年前。

前面讨论了各种求值策略的内涵。下面以C++为例：

```
void ByValue(int a)
{
        a = a + 1;
}

void ByRef(int& a)
{
        a = a + 1;
}

void ByPointer(int* a)
{
        *a = *a + 1;
}
```

```
int main(int argv, char** args)
{
        int v = 1;
        ByValue(v);
        ByRef(v);

        // Pass by Reference
        ByPointer(&v);

        // Pass by Value
        int* vp = &v;
        ByPointer(vp);
}
```

Main函数里的前两种方式没有什么好说，第一个是值传递，第二个函数是引用传递，但是后面两种，**同一个函数，一次调用是Call by reference, 一次是Call by value。**因为：

ByPointer(vp); 没有改变vp，其实是无法改变。

ByPointer(&v); 改变了v。（你可能会说，这传递的其实是v的地址，而ByPointer无法改变v的地址，所以这是Call by value。这听上去可以自圆其说，但是v的地址，是个纯数据，在调用的方代码中并不存在，对于调用者而言，只有v，而v的确被ByPointer函数改了，这个结果，正是Call by reference的行为。**从行为考虑，才是求值策略的本意。如果把所有东西都抽象成值，从数据考虑问题，那根本就没有必要引入求值策略的概念去混淆视听。**）

请体会一下，应该就明白上面一直在说的调用的行为的意思。

C语言不支持引用，只支持指针，但是如上文所见，使用指针的函数，不能通过签名明确其求值策略。C++引入了引用，它的求值策略可以确定是Pass by reference。于是C++的一个奇葩的地方来了，它语言本身（模拟的不算，什么都能模拟）支持Call by value和Call by reference两种求值策略，但是却提供了三种语法去做这俩事儿。

C#的设计就相对合理，函数声明里，有ref/out，就是引用传递，没有ref/out，就是值传递，与参数类型无关。

不过如果观察一下void ByRef(int& a)和void ByPointer(int* a)所生成的汇编代码，会发现在一定条件下其实是一样的。都是这个样子：

```
; 12   : {

        push    ebp
        mov     ebp, esp
        sub     esp, 192                            ; 000000c0H
        push    ebx
        push    esi
        push    edi
        lea     edi, DWORD PTR [ebp-192]
        mov     ecx, 48                             ; 00000030H
        mov     eax, -858993460                     ; cccccccH
        rep stosd

; 13   :        *a = *a + 1;

        mov     eax, DWORD PTR _a$[ebp]
        mov     ecx, DWORD PTR [eax]
        add     ecx, 1
        mov     edx, DWORD PTR _a$[ebp]
        mov     DWORD PTR [edx], ecx
```

调用方的代码也是一样的。代码就不贴了。

这两种传递方式说完了，下面回到正题说好处。问题中“这种”

支持多种求值策略可以给语言带来更高的灵活性，但是同时也需要一个“灵活”的人来良好地驾驭。Java通过牺牲这种价值不大还可能带来问题的灵活性，带来了语言自身语法一致性、逻辑鲁棒性及更容易学习等多个好处。

不仅仅Java和C#，每个语言，在设计时都需要在这些特性间做出自己独特的取舍来体现自己的设计理念，并适应不同人，不同使用环境的要求。虽然说没有什么功能是一个语言可以做，而另一个语言做不了的。但是每个语言，都有它最适合的范畴与不适合的范畴。

发布于 2016-07-20

郭无心
做好自己

12 人赞同了该回答

实名赞同 @Intopass的答案
再举个例子例证下

```
public class Employee {
        public int age;
}


public class Main {
        public static void changeEmployee(Employee employee)
        {
                employee = new Employee();
                employee.age = 1000;
        }

        public static void main(String[] args) {
                Employee employee = new Employee();
                employee.age = 100;
                changeEmployee(employee);
                System.out.println(employee.age);

        }
}
```

在changeEmployee当中是不是使用新的引用，决定原值改变与否

发布于 2015-10-01

祖春雷
无证程序员

75 人赞同了该回答

java中方法参数传递方式是按值传递。
如果参数是基本类型，传递的是基本类型的字面量值的拷贝。
如果参数是引用类型，传递的是该参量所引用的对象在堆中地址值的拷贝。

发布于 2016-05-30

知乎用户
深入一个问题，经历一些过程，然后成为一种优雅

75 人赞同了该回答

哎～～说得那么神秘干嘛呢？关于java的值传递啊、引用传递啊、指针啊blabla，记住4点黄金口诀：

> 1. *= 是赋值操作*（任何包含=的如+=、-=、 /=等等，都内含了赋值操作）。不再是你以前理解的数学含义了，而+ - * /和 = 在java中更不是一个级别，换句话说， = 是一个动作，一个可以改变内存状态的操作，一个可以改变变量的符号，而+ - * /却不会。**这里的赋值操作其实是包含了两个意思：1、放弃了原有的值或引用；2、得到了 = 右侧变量的值或引用。**
> Java中对 = 的理解很重要啊！！可惜好多人忽略了，或者理解了却没深思过。

2. *对于基本数据类型变量，= 操作是完整地复制了变量的值*。换句话说，“=之后，你我已无关联”；至于基本数据类型，就不在这科普了。
3. *对于非基本数据类型变量，= 操作是复制了变量的引用*。换句话说，“嘿，= 左侧的变量，你丫别给我瞎动！咱俩现在是一根绳上的蚂蚱，除非你再被 = 一次放弃现有的引用！！上面说了 = 是一个动作，所以我把 = 当作动词用啦！！”。而非基本数据类型变量你基本上可以
4. ***参数本身是变量，参数传递本质就是一种 = 操作***。参数是变量，所有我们对变量的操作、变量能有的行为，参数都有。所以把C语言里参数是传值啊、传指针啊的那套理论全忘掉，**参数传递就是 = 操作**。

这样你就不难理解了。add()函数和append()函数、addNum()函数不同的地方就是add()里面没有对参数进行赋值操作，换句话说就是在add()函数中，参数list始终没有放弃现有的引用，它的所作所为，都直接反应到现有引用的对象上。

而append()函数，一上来第一句话就对str变量进行了 = 操作，也就是str没有瞎动（事实上String类型的不可变性设计也决定了str也瞎动不了），而是很识趣地引用了一个新的对象。

对于add()函数，可以套口诀：**1、参数传递本质就是一种 = 操作**，= 的左边是add()函数的参变量list，=右边是main函数中给出的list；2、*非基本数据类型变量，= 操操作是复制了变量的引用*，所以add里的list获得了和main里的list同一对象引用，在add里没有对list进行=的语句，所有它会一直保持该引用，它所作的一切改变都被看在眼里。

假如把add()函数改成这个样子

```
static void add(List<Integer> list){
    list = another_list;
    list.add(100);
}
```

那么打印结果里就不会有100了

对于append()函数，可以套口诀：**1、*参数传递本质就是一种 = 操作***，= 的左边是append()函数的参变量str，=右边是main函数中给出的a；2、*非基本数据类型变量，= 操操作是复制了变量的引用*，3、= 是赋值操作，所以在第一条语句里，str就放弃了原有引用。

再假如把append()函数改成这个样子

```
static void append(String str){
    str.addmyself(" is a");
}
```

咳咳~~假设String类里真有addmys elf方法，那么打印结果就是 A is a

至于addNum()函数，就没啥纠结了，套口诀就行：**1、*参数传递本质就是一种 = 操作；2、****对于基本数据类型变量，= 操作是完整地复制了变量的值。***

=====

还有你这道题目，append函数和addNum完全就是充满深深的恶意啊！！！为啥偏偏add函数里没有赋值操作，而是方法调用？？？！！

所有还是重申一下，Java中对 = 的理解很重要啊！！你理解了第一条定律就理解了一半了。

还有就是一定要记住，把C语言里参数传值传指针那套理论全忘掉，**传参就是 = 操作。**

编辑于 2016-10-09