Pass by Value or Reference in Java

jenny

March 5, 2014

Contents

1 FAQ2.14 变量之间传值时可分为值传递和引用传递, 那么它们有何区别?

1.1 简单类型是按值传递的

Java 方法的参数是简单类型的时候, 是按值传递的 (pass by value)。这一点我们可以通过一个简单的例子来说明:

```
public class Test {
    public static void test(boolean test) {
        test = ! test;
        System.out.println("In␣test(boolean)␣:␣test␣=␣" + test);
    }
    public static void main(String[] args) {
        boolean test = true;
        System.out.println("Before␣test(boolean)␣:␣test␣=␣" + test);
        test(test);
        System.out.println("After␣test(boolean)␣:␣test␣=␣" + test);
    }
}
```

运行结果:

```
Before test(boolean) : test = true
In test(boolean) : test = false
After test(boolean) : test = true
```

不难看出，虽然在 test(boolean) 方法中改变了传进来的参数的值，但对这个参数源变量本身并没有影响，即对 main(String[]) 方法里的 test 变量没有影响。那说明，参数类型是简单类型的时候，是按值传递的。以参数形式传递简单类型的变量时，实际上是将参数的值作了一个拷贝传进方法函数的，那么在方法函数里再怎么改变其值，其结果都是只改变了拷贝的值，而不是源值。

1.2 什么是引用

Java 是传值还是传引用，问题主要出在对象的传递上，因为 Java 中简单类型没有引用。既然争论中提到了引用这个东西，为了搞清楚这个问题，我们必须要知道引用是什么。

简单的说，引用其实就像是一个对象的名字或者别名 (alias)，一个对象在内存中会请求一块空间来保存数据，根据对象的大小，它可能需要占用的空间大小也不等。访问对象的时候，我们不会直接是访问对象在内存中的数据，而是通过引用去访问。引用也是一种数据类型，我们可以把它想象为类似 C 语言中指针的东西，它指示了对象在内存中的地址——只不过我们不能够观察到这个地址究竟是什么。

如果我们定义了不止一个引用指向同一个对象，那么这些引用是不相同的，因为引用也是一种数据类型，需要一定的内存空间来保存。但是它们的值是相同的，都指示同一个对象在内存的中位置。比如

```
String a = "Hello";
String b = a;
```

这里，a 和 b 是不同的两个引用，我们使用了两个定义语句来定义它们。但它们的值是一样的，都指向同一个对象 "Hello"。也许你还觉得不够直观，因为 String 对象的值本身是不可更改的 (像 b = "World"; b = a; 这种情况不是改变了 "World" 这一对象的值，而是改变了它的引用 b 的值使之指向了另一个 String 对象 a)。那么我们用 StringBuffer 来举一个例子：

```
public class Test {
    public static void main(String[] args) {
        StringBuffer a = new StringBuffer("Hello");
        StringBuffer b = a;
        b.append(",␣World");
        System.out.println("a␣is␣" + a);
    }
}
```

运行结果：

a is Hello, World

这个例子中 a 和 b 都是引用，当改变了 b 指示的对象的值的时候，从输出结果来看，a 所指示的对象的值也改变了。所以，a 和 b 都指向同一个对象即包含 "Hello" 的一个StringBuffer 对象。

1.2.1 这里我描述了两个要点：

1. 引用是一种数据类型，保存了对象在内存中的地址，这种类型即不是我们平时所说的简单数据类型也不是类实例(对象)；
2. 不同的引用可能指向同一个对象，换句话说，一个对象可以有多个引用，即该类类型的变量。

1.3 对象是如何传递的呢

关于对象的传递，有两种说法，即“它是按值传递的”和“它是按引用传递的”。这两种说法各有各的道理，但是它们都没有从本质上去分析，即致于产生了争论。既然现在我们已经知道了引用是什么东西，那么现在不妨来分析一下对象作是参数是如何传递的。还是先以一个程序为例：

```
public class Test {
    public static void test(StringBuffer str) {
        str.append(",␣World!");
    }
    public static void main(String[] args) {
        StringBuffer string = new StringBuffer("Hello");
        test(string);
        System.out.println(string);
    }
}
```

运行结果：

Hello, World!

test(string) 调用了 test(StringBuffer) 方法，并将 string 作为参数传递了进去。这里 string 是一个引用，这一点是勿庸置疑的。前面提到，引用是一种数据类型，而且不是对象，所以它不可能按引用传递，所以它是按值传递的，它么它的值究竟是什么呢？是对象的地址。

由此可见，对象作为参数的时候是按值传递的，对吗？错！为什么错，让我们看另一个例子：

```
public class Test {
    public static void test(String str) {
        str = "World";
    }
    public static void main(String[] args) {
        String string = "Hello";
        test(string);
        System.out.println(string);
    }
}
```

运行结果：

Hello

为什么会这样呢？因为参数 str 是一个引用，而且它与 string 是不同的引用，虽然它们都是同一个对象的引用。str = "World" 则改变了 str 的值，使之指向了另一个对象，然而 str指向的对象改变了，但它并没有对 "Hello" 造成任何影响，而且由于 string 和 str 是不同的引用，str 的改变也没有对 string 造成任何影响，结果就如例中所示。

其结果是推翻了参数按值传递的说法。那么，对象作为参数的时候是按引用传递的了？也错！因为上一个例子的确能够说明它是按值传递的。

1.3.1 结果，

就像光到底是波还是粒子的问题一样，Java 方法的参数是按什么传递的问题，其答案就只能是：即是按值传递也是按引用传递，只是参照物不同，结果也就不同。

①单纯考虑参数str存的也是一种数据类型，可以看成是值传递。

②考虑参数str它是对象string的一个引用，此时就可看做是引用传递。

1.4 正确看待传值还是传引用的问题

要正确的看待这个问题必须要搞清楚为什么会有这样一个问题。

实际上，问题来源于 C，而不是 Java。

C 语言中有一种数据类型叫做指针，于是将一个数据作为参数传递给某个函数的时候，就有两种方式：传值，或是传指针，它们的区别，可以用一个简单的例子说明：

```
void SwapValue(int a, int b) {
    int t = a;
    a = b;
    b = t;
}

void SwapPointer(int * a, int * b) {
    int t = * a;
    * a = * b;
    * b = t;
}

void main() {
    int a = 0, b = 1;
    printf("1 : a = %d, b = %d\n", a, b);
    SwapValue(a, b);
    printf("2 : a = %d, b = %d\n", a, b);
    SwapPointer(&a, &b);
    printf("3 : a = %d, b = %d\n", a, b);
}
```

运行结果：

```
1 : a = 0, b = 1
2 : a = 0, b = 1
3 : a = 1, b = 0
```

大家可以明显的看到，按指针传递参数可以方便的修改通过参数传递进来的值，而按值传递就不行。

当 Java 成长起来的时候，许多的 C 程序员开始转向学习 Java，他们发现，使用类似SwapValue 的方法仍然不能改变通过参数传递进来的简单数据类型的值，但是如果是一个对象，则可能将其成员随意更改。于是他们觉得这很像是 C 语言中传

值/传指针的问题。但是 Java 中没有指针，那么这个问题就演变成了传值/传引用的问题。可惜将这个问题放在 Java 中进行讨论并不恰当。

讨论这样一个问题的最终目的只是为了搞清楚何种情况才能在方法函数中方便的更改参数的值并使之长期有效。

1.4.1 Java 中，改变参数的值有两种情况，

第一种，使用赋值号“=”直接进行赋值使其改变，如例 1 和例 4;

第二种，对于某些对象的引用，通过一定途径对其成员数据进行改变，如例 3。

对于第一种情况，其改变不会影响到方法该方法以外的数据，或者直接说源数据。

而第二种方法，则相反，会影响到源数据——因为引用指示的对象没有变，对其成员数据进行改变则实质上是改变的该对象。

1.5 面试题:

当一个对象被当作参数传递到一个方法后，此方法可改变这个对象的属性，并可返回变化后的结果，那么这里到底是值传递还是引用传递?

答:是值传递。Java 编程语言只有值传递参数。当一个对象实例作为一个参数被传递到方法中时，参数的值就是该对象的引用一个副本。指向同一个对象,对象的内容可以在被调用的方法中改变，但对象的引用(不是引用的副本)是永远不会改变的。

在 Java 应用程序中永远不会传递对象，而只传递对象引用。因此是按引用传递对象。但重要的是要区分参数是如何传递的，这才是该节选的意图。Java 应用程序按引用传递对象这一事实并不意味着 Java 应用程序按引用传递参数。参数可以是对象引用，而 Java 应用程序是按值传递对象引用的。

Java 应用程序中的变量可以为以下两种类型之一：引用类型或基本类型。当作为参数传递给一个方法时，处理这两种类型的方式是相同的。两种类型都是按值传递的；没有一种按引用传递。

按值传递 意味着当将一个参数传递给一个函数时，函数接收的是原始值的一个副本。因此，如果函数修改了该参数，仅改变副本，而原始值保持不变。

按引用传递 意味着当将一个参数传递给一个函数时，函数接收的是原始值的内存地址，而不是值的副本。因此，如果函数修改了该参数，调用代码中的原始值也随之改变。

当传递给函数的参数不是引用时，传递的都是该值的一个副本（按值传递）。区别在于 **按引用传递** 。在 **C++** 中当传递给函数的参数是引用时，您传递的就是这个引用，或者内存地址（按引用传递）。在 **Java** 应用程序中，当对象引用是传递给方法的一个参数时，您传递的是该引用的一个副本（按值传递），而不是引用本身。

Java 应用程序按值传递参数(引用类型或基本类型)，其实都是传递他们的一份拷贝.而不是数据本身.(不是像 C++ 中那样对原始值进行操作。)

1.5.1 例1:

```
//在函数中传递基本数据类型,
public class Test {
    public static void change(int i, int j) {
        int temp = i;
        i = j;
        j = temp;
    }
    public static void main(String[] args) {
        int a = 3;
        int b = 4;
        change(a, b);
        System.out.println("a=" + a);
        System.out.println("b=" + b);
    }
}
```

public static void change(int i, int j) { int temp = i; i = j; j = temp; }

结果为:

```
a=3
b=4
```

原因就是 参数中传递的是 基本类型 a 和 b 的拷贝,在函数中交换的也是那份拷贝的值 而不是数据本身;

1.5.2 例2:

```
//传的是引用数据类型
public class Test {
    public static void change(int[] counts) {
        counts[0] = 6;
```

```
        System.out.println(counts[0]);
    }
    public static void main(String[] args) {
        int[] count = { 1, 2, 3, 4, 5 };
        change(count);
    }
}
```

在方法中 传递引用数据类型int数组，实际上传递的是其引用count的拷贝，他们都指向数组对象，在方法中可以改变数组对象的内容。即:对复制的引用所调用的方法更改的是同一个对象。

1.5.3 例3:

```
//对象的引用不是引用的副本是永远不会改变的()
class A {
        int i = 0;
}

public class Test {
        public static void add(A a) {
                a = new A();
                a.i++;
        }
        public static void main(String args[]) {
                A a = new A();
                add(a);
                System.out.println(a.i);
        }
}
```

输出结果是0

在该程序中，对象的引用指向的是A ,而在change方法中，传递的引用的一份副本则指向了一个新的OBJECT，并对其进行操作。

而原来的A对象并没有发生任何变化。 引用指向的是还是原来的A对象。

1.5.4 例4:

String 不改变，数组改变

```
public class Example {
    String str = new String("good");
    char[] ch = { 'a', 'b', 'c' };

    public static void main(String args[]) {
        Example ex = new Example();
        ex.change(ex.str, ex.ch);
        System.out.print(ex.str + " and ");
        System.out.println(ex.ch);
    }
    public void change(String str, char ch[]) {
        str = "test ok";
        ch[0] = 'g';
    }
}
```

程序3输出的是 good and gbc.

String 比较特别，看过String 代码的都知道， String 是 final的。所以值是不变的。 函数中String对象引用的副本指向了另外一个新String对象,而数组对象引用的副本没有改变,而是改变对象中数据的内容.

对于对象类型，也就是Object的子类，如果你在方法中修改了它的成员的值，那个修改是生效的，方法调用结束后，它的成员是新的值，但是如果你把它指向一个其它的对象，方法调用结束后，原来对它的引用并没用指向新的对象。

```
class Person {
    private int age;
    private String name;
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
```

```
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public static void main(String[] args) {
        Person person = new Person();//---1
        Person p2 = new Person();
        person.setAge(12);
        person.setName("lj");
        test.changePerson(person,p2);
        //---2 该处不同于处的，而是前者的一个拷贝，他们指向堆中同一个对象person1person

        System.out.println("in main...");
        System.out.println("person.age:" + person.getAge());
        System.out.println("person.name:" + person.getName());
    }

    void changePerson(Person p1,Person p2){
        p2.setAge(13);
        p2.setName("yb");
        p1 = p2;
        System.out.println("in changePerson...");
        System.out.println("person.age:" + p1.getAge());
        System.out.println("person.name:" + p1.getName());
    }
}

void changePerson2(Person p){
    p.setAge(100);
    p.setName("ljyb");
}

test.changePerson2(person);
System.out.println("in changePerson...");
System.out.println("person.age:" + person.getAge());
System.out.println("person.name:" + person.getName());
```

运行结果：

```
in changePerson...
person.age:13
person.name:yb
in main...
person.age:12
person.name:lj
```

也许会很奇怪，java对象在方法里传递时不是引用传递么？为什么传过去的person在changePerson()方法体内把引用指向了p2，但回到主方法后又把引用指向了原来的对象

原因如下：

主函数中：person–¿ [age:12 name:lj] ¡–person’

person’即为test.changePerson(person,p2)；中的person，它与前面的person具有相同的引用地址，因此java中的“对象变量”的传递叫做“引用传递”。但就其实质而言，还是值传递，只不过这个值是引用（或者说地址）而已。

changePerson函数中：person对象的拷贝p1又把引用指向了另一个对象p2，所以会打印出新对象的信息，而当这个函数结束，p1的生命周期也就结束了。

再次回到主函数时，person对象指向原来的对象的事实没有改变，所以还是打印出原来对象的信息

如果还有一个函数changePerson2如下：

```
void changePerson2(Person p){
    p.setAge(100);
          p.setName("ljyb");
}
```

则在主函数增加如下代码：

```
test.changePerson2(person);
System.out.println("in changePerson...");
System.out.println("person.age:" + person.getAge());
System.out.println("person.name:" + person.getName());
```

结果为：

```
after invoking changePerson2...
person.age:100
person.name:ljyb
```

此时的传递方式与上面相同，changePerson2中的p变量也是主函数中person变量的一份拷贝，他们指向同一个对象。p改变了这个对象的值，也就改变了person指向对象的值。

当changePerson2函数结束时，p的生命周期也结束了，但它在changePerson2内做的操作却被保存下来了。

此时打印person的信息就变成修改后的信息了。

我还找到一些相关资料，附下

下面的文字来自一个哥们的文章中的评论，他写的这篇文章也是关于java值传递的

网址：http://zangweiren.javaeye.com/blog/214369

有些争论没有意义，其实你知道传值和传引用由什么区别就可以了，但不要自以为是的认为就是别人混淆了。

“有一种说法是当一个对象或引用类型变量被当作参数传递时，也是值传递，这个值就是对象的引用，因此JAVA中只有值传递，没有引用传递。这种说法显然是混淆了值和引用的概念。”

我看到的很多书中都是只有pass by value的概念。

James Gosling，我想他对Java语言里的一些概念还是有话语权的，《The Java Programming Language》2.6.5. Parameter Values一节中，他的说法是：

1.5.5 引用

All parameters to methods are passed ”by value.” In other words, values of parameter variables in a method are copies of the values the invoker specified as arguments.。。。。。You should note that when the parameter is an object reference, it is the object reference not the object itself that is passed ”by value.” Thus, you can change which object a parameter refers to inside the method without affecting the reference that was passed. But if you change any fields of the object or invoke methods that change the object’s state, the object is changed for every part of the program that holds a reference to it.

这里说的很清楚了，方法的所有参数都是值传递。

别的书里也有提及，Core Java 卷一中也有类似的说法，举了一些例子。

Thinking In Java中也提到过，具体什么地方记不清楚了。

从编译原理的概念来说，有传值，传地址，传名，传结果等。这里的传地址明显和java的传引用不一样。C++中引入了传引用，但是C++的传引用有个明显的特点，引用一旦创建不能修改，所以引用不能指向新的对象。所以不会出现java这种令人混淆的概念。所以还是多用Java中的术语，区分pass by value中不同的情况，而不是自己引入pass by reference 的概念让人们更加混淆。

1.6 数据类型

java的数据类型有两类：

PrimitiveType(简单类型)

ReferenceType(引用类型)

1.6.1 PrimitiveType(简单类型)

（参考：langspec-3.0/typesValues.html#4.2）

1. PrimitiveType的分类如下所示：

   (a) PrimitiveType:
   NumericType
   boolean

   (b) NumericType:
   IntegralType
   FloatingPointType

   (c) IntegralType: one of
   byte short int long char

   (d) FloatingPointType: one of
   float double

   PrimitiveType是java预定义的类型，并且使用保留字命名。比如int、long、float等。由此看来其包装类不算Primiti

1.6.2 ReferenceType(引用类型)

（参考：langspec-3.0/typesValues.html#4.3）
ReferenceType有三种类型：类、接口、和数组。

1.7 变量

（参考：langspec-3.0/typesValues.html#4.12）
A variable is a storage location and has an associated type, sometimes called its compile-time type, that is either a primitive type (§4.2) or a reference type (§4.3).
变量是关联于特定类型的存储单元，所关联的类型有时叫做变量的编译时类型，即，既可以是简单类型也可以是引用类型。

1.7.1 简单类型变量

A variable of a primitive type always holds a value of that exact primitive type.
简单类型的变量总是执持简单类型的值。

1.7.2 引用类型变量

A variable of a class type T can hold a null reference or a reference to an instance of class T or of any class that is a subclass of T. A variable of an interface type can hold a null reference or a reference to any instance of any class that implements the interface.
类型是T的类的变量可以执持null引用，或者类T及其子类的实例引用。接口类型的变量可以执持null引用，或者任何实现该接口的类的实例引用。
注：与langspec2.0不同的是，3.0引入了泛型的概念，其中有Type Variable的概念，上面的T就是一个Type Variable。

1.8 赋值与传递

1.8.1 如上所述，可以得出下面结论：

1. 对于简单类型变量的赋值是按值传递。就是说直接把数值存放到变量的存储单元里。
2. 对于引用类型的变量，赋值是把原对象的引用（可以理解为入口地址），存放在变量的存储单元里。

1.8.2 对象的赋值

简单类型的赋值很容易理解，这里仅讨论对象的赋值。所有引用类型的实例就是我们常说的对象。

1. 可以这样说，除了null以外，任何变量的 **初始赋值** 都是分两步：
   (a) 创建对象实例
   (b) 把对象实例的引用赋值给变量。

   比如：

```
1          Object o1 = new Object();
```

1.8.3 传递

传递是通过变量之间的赋值实现的。在以前的回贴中我说过这样一句话，单纯从变量的角度看，变量之间的赋值是值传递。现在我解释一下我的观点。
先举一个例子：

```
// 中所有的类的基类默认为，在此不赘述。javaObject
class Object1 {}
class Object2 {}

Object o1, o2;
o1 = new Object1();
o2 = o1;
o2 = new Object2();
```

这时候，o1的类型是什么？是Object1还是Object2？正确答案是Object1。
再举一个例子：

```
class Word {
    String word;
    public Word(String word){
        this.word = word;
    }
    public void print(){
        System.out.println(word);
    }
}

Word o1, o2;
o1 = new Word("Every␣Day");
o2 = o1;
o2 = new Word("Every␣Night!");
w1.print();
```

会出现什么结果？”Every Day” 还是 ”Every Night!”？仍然是”Every Day”。

这里面有一个很多人特别是初学者忽视了的观点 变量可以引用对象，但变量不是对象。什么是对象？对象初始化之后，会占用一块内存空间，严格意义上讲，这段内存空间才是对象。对象创建于数据段，而变量存在于代码段；对象的入口地址是不可预知的，所以程序只能通过变量来访问对象。

回到我们的问题上来，第一句

```
o1 = new Word("Every␣Day");
```

首先创建一个Word实例，即对象，然后把“引用”赋值给o1。

第二句

```
o2 = o1;
```

o1把对象的引用赋值给o2，注意赋的值是对象的引用而不是o1自身的引用。所以，在的三句

```
o2 = new Word("Every␣Night!");
```

就是又创建一个新对象，再把新对象的引用赋值给o2。

因为o1和 o2之间是值传，所以，对o2的改变丝毫不会影响到o1。

也有一种情况好像是影响到了o1，我们继续上面的例子，给Word增加一个方法

```
class Word {
    String word;
    public Word(String word){
        this.word = word;
    }
    public void print(){
        System.out.println(word);
    }
    public void setWord(String word){
        this.word = word;
    }
}

Word o1, o2;
o1 = new Word("Every␣Day");
o2 = o1;
o2.set Word("Every␣Night!");
o1.print();
```

这时的结果是”Every Night!”。

那么，这是改变了o1吗？从严格意义上讲，不是。因为o1只是保存对象的引用，执行之后，o1还是持有该对象的引用。所以，o1没变，变的是o1所引用的对象。

1.8.4 final变量能改变吗?

好了，我再出道题目：

```
final Word o3 = new Word("Every␣Day!");
o3.setWord("Every␣Night!");
```

能通过编译吗？对于final的定义大家都知道，o3是相当于一个常量，既然是常量，怎么能再改变呢？

答案是肯定的，能。道理我想大家也明白，这里不罗嗦了。

1.8.5 包装类的赋值与传递

以前看过文章说，对于java基本类型及其包装类采用值传递，对于对象采用引用传递。从langspec看，首先包装类不是PrimitiveType，那就只能是ReferenceType，而ReferenceType的变量保存的是引用。既然保存的是引用，也就无从传递数值。那么，这两个观点矛盾吗?

首先，肯定是langspec正确。

其次，虽然前一观点在原理上有错误，但却不影响正常使用。

为什么会出现这种情况？这是因为这些包装类具有一个简单类型的特征，即，不可改变。以String为例，看一下API Specification，不会找到能够改变String对象的方法。任何输出上的改变都是重建新的String对象，而不是在原对象基础上改变。改变的是变量的内容，即，不同对象的引用。