关键字: java对象 引用

Java对象及其引用

关于对象与引用之间的一些基本概念。

       初学Java时，在很长一段时间里，总觉得基本概念很模糊。后来才知道，在许多Java书中，把对象和对象的引用混为一谈。可是，如果我分不清对象与对象引用，

       那实在没法很好地理解下面的面向对象技术。把自己的一点认识写下来，或许能让初学Java的朋友们少走一点弯路。

       为便于说明，我们先定义一个简单的类：

       class Vehicle {

       int passengers;

       int fuelcap;

       int mpg;

                   }

有了这个模板，就可以用它来创建对象：

       Vehicle veh1 = new Vehicle();

通常把这条语句的动作称之为创建一个对象，其实，它包含了四个动作。

1）右边的“new Vehicle”，是以Vehicle类为模板，在堆空间里创建一个Vehicle类对象（也简称为Vehicle对象）。

2）末尾的()意味着，在对象创建后，立即调用Vehicle类的构造函数，对刚生成的对象进行初始化。构造函数是肯定有的。如果你没写，Java会给你补上一个默认的构造函数。

3）左边的“Vehicle veh 1”创建了一个Vehicle类引用变量。所谓Vehicle类引用，就是以后可以用来指向Vehicle对象的对象引用。

4）“=”操作符使对象引用指向刚创建的那个Vehicle对象。

我们可以把这条语句拆成两部分：

Vehicle veh1;

veh1 = new Vehicle();

效果是一样的。这样写，就比较清楚了，有两个实体：一是对象引用变量，一是对象本身。

       在堆空间里创建的实体，与在数据段以及栈空间里创建的实体不同。尽管它们也是确确实实存在的实体，但是，我们看不见，也摸不着。不仅如此，

       我们仔细研究一下第二句，找找刚创建的对象叫什么名字？有人说，它叫“Vehicle”。不对，“Vehicle”是类（对象的创建模板）的名字。

       一个Vehicle类可以据此创建出无数个对象，这些对象不可能全叫“Vehicle”。

       对象连名都没有，没法直接访问它。我们只能通过对象引用来间接访问对象。

       为了形象地说明对象、引用及它们之间的关系，可以做一个或许不很妥当的比喻。对象好比是一只很大的气球，大到我们抓不住它。引用变量是一根绳， 可以用来系汽球。

       如果只执行了第一条语句，还没执行第二条，此时创建的引用变量veh1还没指向任何一个对象，它的值是null。引用变量可以指向某个对象，或者为null。

       它是一根绳，一根还没有系上任何一个汽球的绳。执行了第二句后，一只新汽球做出来了，并被系在veh1这根绳上。我们抓住这根绳，就等于抓住了那只汽球。

       再来一句：

       Vehicle veh2;

就又做了一根绳，还没系上汽球。如果再加一句：

       veh2 = veh1;

系上了。这里，发生了复制行为。但是，要说明的是，对象本身并没有被复制，被复制的只是对象引用。结果是，veh2也指向了veh1所指向的对象。两根绳系的是同一只汽球。

       如果用下句再创建一个对象：

veh2 = new Vehicle();

则引用变量veh2改指向第二个对象。

       从以上叙述再推演下去，我们可以获得以下结论：

（1）一个对象引用可以指向0个或1个对象（一根绳子可以不系汽球，也可以系一个汽球）；

（2）一个对象可以有N个引用指向它（可以有N条绳子系住一个汽球）。

       如果再来下面语句：

       veh1 = veh2;

按上面的推断，veh1也指向了第二个对象。这个没问题。问题是第一个对象呢？没有一条绳子系住它，它飞了。多数书里说，它被Java的垃圾回收机制回收了。

这不确切。正确地说，它已成为垃圾回收机制的处理对象。至于什么时候真正被回收，那要看垃圾回收机制的心情了。

       由此看来，下面的语句应该不合法吧？至少是没用的吧？

new Vehicle();

不对。它是合法的，而且可用的。譬如，如果我们仅仅为了打印而生成一个对象，就不需要用引用变量来系住它。最常见的就是打印字符串：

    System.out.println(“I am Java!”);

字符串对象“I am Java!”在打印后即被丢弃。有人把这种对象称之为临时对象。

       对象与引用的关系将持续到对象回收。

Java对象及引用

Java对象及引用是容易混淆却又必须掌握的基础知识，本章阐述Java对象和引用的概念，以及与其密切相关的参数传递。

先看下面的程序：

StringBuffer s;

s = new StringBuffer("Hello World!");

第一个语句仅为引用(reference)分配了空间，而第二个语句则通过调用类(StringBuffer)的构造函数StringBuffer(String str)为类生成了一个实例（或称为对象）。这两个操作被完成后，对象的内容则可通过s进行访问——在Java里都是通过引用来操纵对象的。

Java对象和引用的关系可以说是互相关联，却又彼此独立。彼此独立主要表现在：引用是可以改变的，它可以指向别的对象，譬如上面的s，你可以给它另外的对象，如：

s = new StringBuffer("Java");

这样一来，s就和它指向的第一个对象脱离关系。

从存储空间上来说，对象和引用也是独立的，它们存储在不同的地方，对象一般存储在堆中，而引用存储在速度更快的堆栈中。

引用可以指向不同的对象，对象也可以被多个引用操纵，如：

StringBuffer s1 = s;

这条语句使得s1和s指向同一个对象。既然两个引用指向同一个对象，那么不管使用哪个引用操纵对象，对象的内容都发生改变，并且只有一份，通过s1和s得到的内容自然也一样，(String除外，因为String始终不变，String s1=”AAAA”; String s=s1,操作s,s1由于始终不变，所以为s另外开辟了空间来存储s,)如下面的程序：

StringBuffer s;

s = new StringBuffer("Java");

StringBuffer s1 = s;

s1.append(" World");

System.out.println("s1=" + s1.toString());//打印结果为：s1=Java World

System.out.println("s=" + s.toString());//打印结果为：s=Java World

上面的程序表明，s1和s打印出来的内容是一样的，这样的结果看起来让人非常疑惑，但是仔细想想，s1和s只是两个引用，它们只是操纵杆而已，它们指向同一个对象，操纵的也是同一个对象，通过它们得到的是同一个对象的内容。这就像汽车的刹车和油门，它们操纵的都是车速，假如汽车开始的速度是80，然后你踩了一次油门，汽车加速了，假如车速升到了120，然后你踩一下刹车，此时车速是从120开始下降的，假如下降到60，再踩一次油门，车速则从60开始上升，而不是从第一次踩油门后的120开始。也就是说车速同时受油门和刹车影响，它们的影响是累积起来的，而不是各自独立（除非刹车和油门不在一辆车上）。所以，在上面的程序中，不管使用s1还是s操纵对象，它们对对象的影响也是累积起来的（更多的引用同理）。

只有理解了对象和引用的关系，才能理解参数传递。

一般面试题中都会考Java传参的问题，并且它的标准答案是Java只有一种参数传递方式：那就是按值传递，即Java中传递任何东西都是传值。如果传入方法的是基本类型的东西，你就得到此基本类型的一份拷贝。如果是传递引用，就得到引用的拷贝。

一般来说，对于基本类型的传递，我们很容易理解，而对于对象，总让人感觉是按引用传递，看下面的程序：

public class ObjectRef {

    //基本类型的参数传递

    public static void testBasicType(int m) {

        System.out.println("m=" + m);//m=50

        m = 100;

        System.out.println("m=" + m);//m=100

    }

    //参数为对象，不改变引用的值 ？？？？？？

    public static void add(StringBuffer s) {

        s.append("\_add");

    }

    //参数为对象，改变引用的值 ？？？？？

    public static void changeRef(StringBuffer s) {

        s = new StringBuffer("Java");

    }

    public static void main(String[] args) {

        int i = 50;

        testBasicType(i);

        System.out.println(i);//i=50

        StringBuffer sMain = new StringBuffer("init");

        System.out.println("sMain=" + sMain.toString());//sMain=init

        add(sMain);

        System.out.println("sMain=" + sMain.toString());//sMain=init\_add

        changeRef(sMain);

        System.out.println("sMain=" + sMain.toString());//sMain=init\_add

    }

}

以上程序的允许结果显示出，testBasicType方法的参数是基本类型，尽管参数m的值发生改变，但并不影响i。

      add方法的参数是一个对象，当把sMain传给参数s时，s得到的是sMain的拷贝，所以s和sMain指向同一个对象，因此，使用s操作影响的其实就是sMain指向的对象，故调用add方法后，sMain指向的对象的内容发生了改变。

      在changeRef方法中，参数也是对象，当把sMain传给参数s时，s得到的是sMain的拷贝，但与add方法不同的是，**在方法体内改变了s指向的对象（也就是s指向了别的对象,牵着气球的绳子换气球了），给s重新赋值后，s与sMain已经毫无关联**，它和sMain指向了不同的对象，所以不管对s做什么操作，都不会影响sMain指向的对象，故调用changeRef方法前后sMain指向的对象内容并未发生改变。

对于add方法的调用结果，可能很多人会有这种感觉：这不明明是按引用传递吗？对于这种问题，还是套用Bruce Eckel的话：这依赖于你如何看待引用，最终你会明白，这个争论并没那么重要。真正重要的是，你要理解，传引用使得（调用者的）对象的修改变得不可预期。

 public   class   Test  
{   public int   i,j;    
    public   void   test\_m(Test   a)  
    {     Test   b   =  new   Test();  
          b.i   =   1;  
          b.j   =   2;  
          a   =   b;  
    }  
    public   void   test\_m1(Test   a   )  
    {     a.i   =   1;  
        a.j   =   2;  
    }  
    public   static   void   main(String   argv[])  
    {     Test   t=   new   Test();  
          t.i   =   5;  
          t.j   =   6;  
          System.out.println( "t.i   =   "+   t.i   +   "   t.j=   "   +   t.j); //5,6  
          t.test\_m(t);  
          System.out.println( "t.i   =   "+   t.i   +   "   t.j=   "   +   t.j); //5,6,a和t都指向了一个对象，而在test\_m中s又指向了另一个对象，所以对象t不变！！！

          t.test\_m1(t);

          System.out.println( "t.i   =   "+   t.i   +   "   t.j=   "   +   t.j); //1,2

    }

}

答案只有一个：Java里都是按值传递参数。而实际上，我们要明白，当参数是对象时，传引用会发生什么状况（就像上面的add方法）？

=========================================================================

楼主，这样来记这个问题  
如下表达式：  
A a1 = new A();  
它代表A是类，a1是引用，a1不是对象，new A()才是对象，a1引用指向new A()这个对象。

在JAVA里，“=”不能被看成是一个赋值语句，它不是在把一个对象赋给另外一个对象，它的执行过程实质上是将右边对象的地址传给了左边的引用，使得左边的引用指向了右边的对象。JAVA表面上看起来没有指针，但它的引用其实质就是一个指针，引用里面存放的并不是对象，而是该对象的地址，使得该引用指向了对象。在JAVA里，“=”语句不应该被翻译成赋值语句，因为它所执行的确实不是一个赋值的过程，而是一个传地址的过程，被译成赋值语句会造成很多误解，译得不准确。

再如：  
A a2;  
它代表A是类，a2是引用，a2不是对象，a2所指向的对象为空null;

再如：  
a2 = a1;  
它代表，a2是引用，a1也是引用，a1所指向的对象的地址传给了a2(传址），使得a2和a1指向了同一对象。

综上所述，可以简单的记为，在初始化时，“=”语句左边的是引用，右边new出来的是对象。  
在后面的左右都是引用的“=”语句时，左右的引用同时指向了右边引用所指向的对象。

再所谓实例，其实就是对象的同义词。

如果需要赋值，就需要类实现Cloneable接口，实现clone()方法。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **class** D **implements** Cloneable{*//实现Cloneable接口*  String sex;  D(String sex){  **this**.sex=sex;  }  @Override  **protected** Object clone() **throws** CloneNotSupportedException {  *// 实现clone方法*  **return** **super**.clone();  }  } |

赋值的时候：

|  |  |
| --- | --- |
| 1  2 | D d=**new** D("男");  D d2=(D) d.clone();*//把d赋值给d2* |

如果类中的变量不是主类型，而是对象，也需要调用该对象的clone()方法  
下面是一个完整的例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | **public** **class** Test2 {  **public** **static** **void** main(String[] args) **throws** CloneNotSupportedException {  *// TODO Auto-generated method stub*  D d=**new** D("男");  C c=**new** C("张三","20",d);  C new\_c=(C) c.clone();*//调用clone方法来赋值*  new\_c.name="李四";  d.sex="女";*//d*  System.out.println(c.d.sex);  System.out.println(c.name);    }    }    **class** C **implements** Cloneable{  String name;  String age;  D d;  C(String name,String age,D d) **throws** CloneNotSupportedException{  **this**.name=name;  **this**.age=age;  **this**.d=(D) d.clone();*//调用clone方法来赋值，这样即便外部的d发生变化，c里的也不会变*  }  @Override  **protected** Object clone() **throws** CloneNotSupportedException {  *// TODO Auto-generated method stub*  **return** **super**.clone();  }  }  **class** D **implements** Cloneable{*//实现Cloneable接口*  String sex;  D(String sex){  **this**.sex=sex;  }  @Override  **protected** Object clone() **throws** CloneNotSupportedException {  *// 实现clone方法*  **return** **super**.clone(); |