JAVA中的指针,引用及对象的clone

级别： 初级

倪大鹏 (userid@us.ibm.com),

2002 年 12 月 23 日

Java语言的一个优点就是取消了指针的概念，但也导致了许多程序员在编程中常常忽略了对象与引用的区别，本文会试图澄清这一概念。并且由于Java不能通过简单的赋值来解决对象复制的问题，在开发过程中，也常常要要应用clone（）方法来复制对象。本文会让你了解什么是影子clone与深度clone，认识它们的区别、优点及缺点。

看到这个标题，是不是有点困惑：Java语言明确说明取消了指针，因为指针往往是在带来方便的同时也是导致代码不安全的根源，同时也会使程序的变得非常复杂难以理解，滥用指针写成的代码不亚于使用早已臭名昭著的"GOTO"语句。Java放弃指针的概念绝对是极其明智的。但这只是在Java语言中没有明确的指针定义，实质上每一个new语句返回的都是一个指针的引用，只不过在大多时候Java中不用关心如何操作这个"指针"，更不用象在操作C＋＋的指针那样胆战心惊。唯一要多多关心的是在给函数传递对象的时候。如下例程：

```
package reference;
class Obj{
    String str = "init value";
    public String toString(){
        return str;
    }
}
public class ObjRef{
    Obj aObj = new Obj();
    int aInt = 11;
    public void changeObj(Obj inObj){
        inObj.str = "changed value";
    }
    public void changePri(int inInt){
        inInt = 22;
    }
    public static void main(String[] args)
    {
        ObjRef oRef = new ObjRef();

        System.out.println("Before call changeObj() method: " + oRef.aObj);
        oRef.changeObj(oRef.aObj);
        System.out.println("After call changeObj() method: " + oRef.aObj);
        System.out.println("==================Print Primtive================");
        System.out.println("Before call changePri() method: " + oRef.aInt);
        oRef.changePri(oRef.aInt);
        System.out.println("After call changePri() method: " + oRef.aInt);
    }
}
/* RUN RESULT
Before call changeObj() method: init value
After call changeObj() method: changed value
==================Print Primtive================
Before call changePri() method: 11
After call changePri() method: 11
*
*/
```

这段代码的主要部分调用了两个很相近的方法，changeObj()和changePri()。唯一不同的是它们一个把对象作为输入参数，另一个把Java中的基本类型int作为输入参数。并且在这两个函数体内部都对输入的参数进行了改动。看似一样的方法，程序输出的结果却不太一样。changeObj()方法真正的把输入的参数改变了，而changePri()方法对输入的参数没有任何的改变。

从这个例子知道Java对对象和基本的数据类型的处理是不一样的。和C语言一样，当把Java的基本数据类型（如int，char，double等）作为入口参数传给函数体的时候，传入的参数在函数体内部变成了局部变量，这个局部变量是输入参数的一个拷贝，所有的函数体内部的操作都是针对这个拷贝的操作，函数执行结束后，这个局部变量也就完成了它的使命，它影响不到作为输入参数的变量。这种方式的参数传递被称为"值传递"。而在Java中用对象的作为入口参数的传递则缺省为"引用传递"，也就是说仅仅传递了对象的一个"引用"，这个"引用"的概念同C语言中的指针引用是一样的。当函数体内部对输入变量改变时，实质上就是在对这个对象的直接操作。

除了在函数传值的时候是"引用传递"，在任何用"＝"向对象变量赋值的时候都是"引用传递"。如：

```
package reference;
class PassObj
{
    String str = "init value";
}
public class ObjPassValue
{
    public static void main(String[] args)
    {
        PassObj objA = new PassObj();
        PassObj objB = objA;
        objA.str = "changed in objA";
        System.out.println("Print objB.str value: " + objB.str);
    }
}
/* RUN RESULT
Print objB.str value: changed in objA
*/
```

第一句是在内存中生成一个新的PassObj对象，然后把这个PassObj的引用赋给变量objA，第二句是把PassObj对象的引用又赋给了变量objB。此时objA和objB是两个完全一致的变量，以后任何对objA的改变都等同于对objB的改变。

即使明白了Java语言中的"指针"概念也许还会不经意间犯下面的错误。

Hashtable真的能存储对象吗？

看一看下面的很简单的代码，先是声明了一个Hashtable和StringBuffer对象，然后分四次把StriingBuffer对象放入到Hashtable表中，在每次放入之前都对这个StringBuffer对象append()了一些新的字符串：

```
package reference;
import java.util.*;
public class HashtableAdd{
    public static void main(String[] args){
        Hashtable ht = new Hashtable();
        StringBuffer sb = new StringBuffer();
        sb.append("abc,");
        ht.put("1",sb);
        sb.append("def,");
        ht.put("2",sb);
        sb.append("mno,");
        ht.put("3",sb);
        sb.append("xyz.");
        ht.put("4",sb);
```

```
        int numObj=0;
        Enumeration it = ht.elements();
        while(it.hasMoreElements()){
            System.out.print("get StringBufffer "+(++numObj)+" from Hashtable: ");
            System.out.println(it.nextElement());
        }
    }
}
```

如果你认为输出的结果是：
get StringBufffer 1 from Hashtable: abc,
get StringBufffer 2 from Hashtable: abc,def，
get StringBufffer 3 from Hashtable: abc,def,mno,
get StringBufffer 4 from Hashtable: abc,def,mno,xyz.

那么你就要回过头再仔细看一看上一个问题了，把对象时作为入口参数传给函数，实质上是传递了对象的引用，向Hashtable传递StringBuffer对象也是只传递了这个StringBuffer对象的引用！每一次向Hashtable表中put一次StringBuffer，并没有生成新的StringBuffer对象，只是在Hashtable表中又放入了一个指向同一StringBuffer对象的引用而已。

对Hashtable表存储的任何一个StringBuffer对象（更确切的说应该是对象的引用）的改动，实际上都是对同一个"StringBuffer"的改动。所以Hashtable并不能真正存储能对象，而只能存储对象的引用。也应该知道这条原则对与Hashtable相似的Vector, List, Map, Set等都是一样的。

上面的例程的实际输出的结果是：

```
/* RUN RESULT
get StringBufffer 1 from Hashtable: abc,def,mno,xyz.
get StringBufffer 2 from Hashtable: abc,def,mno,xyz.
get StringBufffer 3 from Hashtable: abc,def,mno,xyz.
get StringBufffer 4 from Hashtable: abc,def,mno,xyz.
*/
```

类，对象与引用

Java最基本的概念就是类，类包括函数和变量。如果想要应用类，就要把类生成对象，这个过程被称作"类的实例化"。有几种方法把类实例化成对象，最常用的就是用"new"操作符。类实例化成对象后，就意味着要在内存中占据一块空间存放实例。想要对这块空间操作就要应用到对象的引用。引用在Java语言中的体现就是变量，而变量的类型就是这个引用的对象。虽然在语法上可以在生成一个对象后直接调用该对象的函数或变量，如：

```
new String("Hello NDP")).substring(0,3)   //RETURN RESULT: Hel
```

但由于没有相应的引用，对这个对象的使用也只能局限这条语句中了。

1. 产生：引用总是在把对象作参数"传递"的过程中自动发生，不需要人为的产生，也不能人为的控制引用的产生。这个传递包括把对象作为函数的入口参数的情况，也包括用"＝"进行对象赋值的时候。
2. 范围：只有局部的引用，没有局部的对象。引用在Java语言的体现就是变量，而变量在Java语言中是有范围的，可以是局部的，也可以是全局的。
3. 生存期：程序只能控制引用的生存周期。对象的生存期是由Java控制。用"new Object()"语句生成一个新的对象，是在计算机的内存中声明一块区域存储对象，只有Java的垃圾收集器才能决定在适当的时候回收对象占用的内存。
4. 没有办法阻止对引用的改动。

什么是"clone"？

在实际编程过程中，我们常常要遇到这种情况：有一个对象A，在某一时刻A中已经包含了一些有效值，此时可能会需要一个和A完全相同新对象B，并且此后对B任何改动都不会影响到A中的值，也就是说，A与B是两个独立的对象，但B的初始值是由A对象确定的。在Java语言中，用简单的赋值语句是不能满足这种需求的。要满足这种需求虽然有很多途径，但实现clone（）方法是其中最简单，也是最高效的手段。

Java的所有类都默认继承java.lang.Object类，在java.lang.Object类中有一个方法clone()。JDK API的说明文档解释这个方法将返回Object对象的一个拷贝。要说明的有两点：一是拷贝对象返回的是一个新对象，而不是一个引用。二是拷贝对象与用new操作符返回的新对象的区别就是这个拷贝已经包含了一些原来对象的信息，而不是对象的初始信息。

怎样应用clone()方法？

一个很典型的调用clone()代码如下：

```
class CloneClass implements Cloneable{
    public int aInt;
    public Object clone(){
        CloneClass o = null;
        try{
            o = (CloneClass)super.clone();
        }catch(CloneNotSupportedException e){
            e.printStackTrace();
        }
        return o;
    }
}
```

有三个值得注意的地方，一是希望能实现clone功能的CloneClass类实现了Cloneable接口，这个接口属于java.lang包，java.lang包已经被缺省的导入类中，所以不需要写成java.lang.Cloneable。另一个值得请注意的是重载了clone()方法。最后在clone()方法中调用了super.clone()，这也意味着无论clone类的继承结构是什么样的，super.clone()直接或间接调用了java.lang.Object类的clone()方法。下面再详细的解释一下这几点。

应该说第三点是最重要的，仔细观察一下Object类的clone()一个native方法，native方法的效率一般来说都是远高于java中的非native方法。这也解释了为什么要用Object中clone()方法而不是先new一个类，然后把原始对象中的信息赋到新对象中，虽然这也实现了clone功能。对于第二点，也要观察Object类中的clone()还是一个protected属性的方法。这也意味着如果要应用clone()方法，必须继承Object类，在Java中所有的类是缺省继承Object类的，也就不用关心这点了。然后重载clone()方法。还有一点要考虑的是为了让其它类能调用这个clone类的clone()方法，重载之后要把clone()方法的属性设置为public。

那么clone类为什么还要实现Cloneable接口呢？稍微注意一下，Cloneable接口是不包含任何方法的！其实这个接口仅仅是一个标志，而且这个标志也仅仅是针对Object类中clone()方法的，如果clone类没有实现Cloneable接口，并调用了Object的clone()方法（也就是调用了super.Clone()方法），那么Object的clone()方法就会抛出CloneNotSupportedException异常。

以上是clone的最基本的步骤，想要完成一个成功的clone，还要了解什么是"影子clone"和"深度clone"。

什么是影子clone?

下面的例子包含三个类UnCloneA，CloneB，CloneMain。CloneB类包含了一个UnCloneA的实例和一个int类型变量，并且重载clone()方法。CloneMain类初始化UnCloneA类的一个实例b1，然后调用clone()方法生成了一个b1的拷贝b2。最后考察一下b1和b2的输出：

```
package clone;
class UnCloneA {
    private int i;
    public UnCloneA(int ii) { i = ii; }
    public void doubleValue() { i *= 2; }
    public String toString() {
        return Integer.toString(i);
    }
}
class CloneB implements Cloneable{
    public int aInt;
    public UnCloneA unCA = new UnCloneA(111);
    public Object clone(){
        CloneB o = null;
        try{
            o = (CloneB)super.clone();
        }catch(CloneNotSupportedException e){
            e.printStackTrace();
        }
        return o;
    }
}
public class CloneMain {
    public static void main(String[] a){
        CloneB b1 = new CloneB();
        b1.aInt = 11;
        System.out.println("before clone,b1.aInt = "+ b1.aInt);
        System.out.println("before clone,b1.unCA = "+ b1.unCA);

        CloneB b2 = (CloneB)b1.clone();
        b2.aInt = 22;
        b2.unCA.doubleValue();
        System.out.println("================================");
        System.out.println("after clone,b1.aInt = "+ b1.aInt);
        System.out.println("after clone,b1.unCA = "+ b1.unCA);
        System.out.println("================================");
        System.out.println("after clone,b2.aInt = "+ b2.aInt);
        System.out.println("after clone,b2.unCA = "+ b2.unCA);
    }
}
/** RUN RESULT:
before clone,b1.aInt = 11
before clone,b1.unCA = 111
================================
after clone,b1.aInt = 11
after clone,b1.unCA = 222
================================
after clone,b2.aInt = 22
after clone,b2.unCA = 222
*/
```

输出的结果说明int类型的变量aInt和UnCloneA的实例对象unCA的clone结果不一致，int类型是真正的被clone了，因为改变了b2中的aInt变量，对b1的aInt没有产生影响，也就是说，b2.aInt与b1.aInt已经占据了不同的内存空间，b2.aInt是b1.aInt的一个真正拷贝。相反，对b2.unCA的改变同时改变了b1.unCA，很明显，b2.unCA和b1.unCA是仅仅指向同一个对象的不同引用！从中可以看出，调用Object类中clone()方法产生的效果是：先在内存中开辟一块和原始对象一样的空间，然后原样拷贝原始对象中的内容。对基本数据类型，这样的操作是没有问题的，但对非基本类型变量，我们知道它们保存的仅仅是对象的引用，这也导致clone后的非基本类型变量和原始对象中相应的变量指向的是同一个对象。

大多时候，这种clone的结果往往不是我们所希望的结果，这种clone也被称为"影子clone"。要想让b2.unCA指向与b2.unCA不同的对象，而且b2.unCA中还要包含b1.unCA中的信息作为初始信息，就要实现深度clone。

怎么进行深度clone？

把上面的例子改成深度clone很简单，需要两个改变：一是让UnCloneA类也实现和CloneB类一样的clone功能（实现Cloneable接口，重载clone()方法）。二是在CloneB的clone()方法中加入一句o.unCA = (UnCloneA)unCA.clone();

程序如下：

```
package clone.ext;
class UnCloneA implements Cloneable{
    private int i;
    public UnCloneA(int ii) { i = ii; }
    public void doubleValue() { i *= 2; }
    public String toString() {
        return Integer.toString(i);
    }
    public Object clone(){
        UnCloneA o = null;
        try{
            o = (UnCloneA)super.clone();
        }catch(CloneNotSupportedException e){
            e.printStackTrace();
        }
        return o;
    }
}
class CloneB implements Cloneable{
    public int aInt;
    public UnCloneA unCA = new UnCloneA(111);
    public Object clone(){
        CloneB o = null;
        try{
            o = (CloneB)super.clone();
        }catch(CloneNotSupportedException e){
            e.printStackTrace();
        }
        o.unCA = (UnCloneA)unCA.clone();
        return o;
    }
}
public class CloneMain {
    public static void main(String[] a){
        CloneB b1 = new CloneB();
        b1.aInt = 11;
```

```
        System.out.println("before clone,b1.aInt = "+ b1.aInt);
        System.out.println("before clone,b1.unCA = "+ b1.unCA);

        CloneB b2 = (CloneB)b1.clone();
        b2.aInt = 22;
        b2.unCA.doubleValue();
        System.out.println("================================");
        System.out.println("after clone,b1.aInt = "+ b1.aInt);
        System.out.println("after clone,b1.unCA = "+ b1.unCA);
        System.out.println("================================");
        System.out.println("after clone,b2.aInt = "+ b2.aInt);
        System.out.println("after clone,b2.unCA = "+ b2.unCA);
    }
}
/** RUN RESULT:
before clone,b1.aInt = 11
before clone,b1.unCA = 111
================================
after clone,b1.aInt = 11
after clone,b1.unCA = 111
================================
after clone,b2.aInt = 22
after clone,b2.unCA = 222
*/
```

可以看出，现在b2.unCA的改变对b1.unCA没有产生影响。此时b1.unCA与b2.unCA指向了两个不同的UnCloneA实例，而且在CloneB b2 = (CloneB)b1.clone();调用的那一刻b1和b2拥有相同的值，在这里，b1.i = b2.i = 11。

要知道不是所有的类都能实现深度clone的。例如，如果把上面的CloneB类中的UnCloneA类型变量改成StringBuffer类型，看一下JDK API中关于StringBuffer的说明，StringBuffer没有重载clone()方法，更为严重的是StringBuffer还是一个final类，这也是说我们也不能用继承的办法间接实现StringBuffer的clone。如果一个类中包含有StringBuffer类型对象或和StringBuffer相似类的对象，我们有两种选择：要么只能实现影子clone，要么就在类的clone()方法中加一句（假设是SringBuffer对象，而且变量名仍是unCA）： o.unCA = new StringBuffer(unCA.toString()); //原来的是：o.unCA = (UnCloneA)unCA.clone();

还要知道的是除了基本数据类型能自动实现深度clone以外，String对象是一个例外，它clone后的表现好象也实现了深度clone，虽然这只是一个假象，但却大大方便了我们的编程。

Clone中String和StringBuffer的区别

应该说明的是，这里不是着重说明String和StringBuffer的区别，但从这个例子里也能看出String类的一些与众不同的地方。

下面的例子中包括两个类，CloneC类包含一个String类型变量和一个StringBuffer类型变量，并且实现了clone()方法。在StrClone类中声明了CloneC类型变量c1，然后调用c1的clone()方法生成c1的拷贝c2，在对c2中的String和StringBuffer类型变量用相应的方法改动之后打印结果：

```
package clone;
class CloneC implements Cloneable{
    public String str;
    public StringBuffer strBuff;
    public Object clone(){
        CloneC o = null;
        try{
```

```
            o = (CloneC)super.clone();
        }catch(CloneNotSupportedException e){
            e.printStackTrace();
        }
        return o;
    }

}
public class StrClone {
    public static void main(String[] a){
        CloneC c1 = new CloneC();
        c1.str = new String("initializeStr");
        c1.strBuff = new StringBuffer("initializeStrBuff");
        System.out.println("before clone,c1.str = "+ c1.str);
        System.out.println("before clone,c1.strBuff = "+ c1.strBuff);

        CloneC c2 = (CloneC)c1.clone();
        c2.str = c2.str.substring(0,5);
        c2.strBuff = c2.strBuff.append(" change strBuff clone");
        System.out.println("================================");
        System.out.println("after clone,c1.str = "+ c1.str);
        System.out.println("after clone,c1.strBuff = "+ c1.strBuff);
        System.out.println("================================");
        System.out.println("after clone,c2.str = "+ c2.str);
        System.out.println("after clone,c2.strBuff = "+ c2.strBuff);
    }
}
/* RUN RESULT
before clone,c1.str = initializeStr
before clone,c1.strBuff = initializeStrBuff
================================
after clone,c1.str = initializeStr
after clone,c1.strBuff = initializeStrBuff change strBuff clone
================================
after clone,c2.str = initi
after clone,c2.strBuff = initializeStrBuff change strBuff clone
*
*/
```

打印的结果可以看出，String类型的变量好象已经实现了深度clone，因为对c2.str的改动并没有影响到c1.str！难道Java把Sring类看成了基本数据类型？其实不然，这里有一个小小的把戏，秘密就在于c2.str = c2.str.substring(0,5)这一语句！实质上，在clone的时候c1.str与c2.str仍然是引用，而且都指向了同一个String对象。但在执行c2.str = c2.str.substring(0,5)的时候，它作用相当于生成了一个新的String类型，然后又赋回给c2.str。这是因为String被Sun公司的工程师写成了一个不可更改的类（immutable class），在所有String类中的函数都不能更改自身的值。下面给出很简单的一个例子：

package clone; public class StrTest { public static void main(String[] args) { String str1 = "This is a test for immutable"; String str2 = str1.substring(0,8); System.out.println("print str1 : " + str1); System.out.println("print str2 : " + str2); } } /* RUN RESULT print str1 : This is a test for immutable print str2 : This is */

例子中，虽然str1调用了substring()方法，但str1的值并没有改变。类似的，String类中的其它方法也是如此。当然如果我们把最上面的例子中的这两条语句

```
c2.str = c2.str.substring(0,5);
c2.strBuff = c2.strBuff.append(" change strBuff clone");
```

改成下面这样：

```
c2.str.substring(0,5);
c2.strBuff.append(" change strBuff clone");
```

去掉了重新赋值的过程，c2.str也就不能有变化了，我们的把戏也就露馅了。但在编程过程中只调用

```
c2.str.substring(0,5);
```

语句是没有任何意义的。

应该知道的是在Java中所有的基本数据类型都有一个相对应的类，象Integer类对应int类型，Double类对应double类型等等，这些类也与String类相同，都是不可以改变的类。也就是说，这些的类中的所有方法都是不能改变其自身的值的。这也让我们在编clone类的时候有了一个更多的选择。同时我们也可以把自己的类编成不可更改的类。

关于作者

倪大鹏，有五年的软件开发经验，其中的近四年时间里是在从事与Java相关技术的应用与开发。你可以通过e-mail: ndp@21cn.com与他联系。