**Java泛型类型擦除以及类型擦除带来的问题**

目录



# 一、Java泛型的实现方法：类型擦除

大家都知道，Java的泛型是伪泛型，这是因为Java在编译期间，所有的泛型信息都会被擦掉，正确理解泛型概念的首要前提是理解类型擦除。Java的泛型基本上都是在编译器这个层次上实现的，在生成的字节码中是不包含泛型中的类型信息的，使用泛型的时候加上类型参数，在编译器编译的时候会去掉，这个过程成为**类型擦除。**

如在代码中定义 `List<Object>` 和 `List<String>` 等类型，在编译后都会变成 `List` ，JVM看到的只是 `List` ，而由泛型附加的类型信息对JVM是看不到的。Java编译器会在编译时尽可能的发现可能出错的地方，但是仍然无法在运行时刻出现的类型转换异常的情况，类型擦除也是 Java 的泛型与 C++ 模板机制实现方式之间的重要区别。

通过两个例子证明Java类型的类型擦除

## 1、原始类型相等

```
public class Test {

    public static void main(String[] args) {

        ArrayList<String> list1 = new ArrayList<String>();
        list1.add("abc");

        ArrayList<Integer> list2 = new ArrayList<Integer>();
        list2.add(123);

        System.out.println(list1.getClass() == list2.getClass());
    }

}
```

在这个例子中，我们定义了两个 `ArrayList` 数组，不过一个是 `ArrayList<String>` 泛型类型的，只能存储字符串；一个是 `ArrayList<Integer>` 泛型类型的，只能存储整数，最后，我们通过 `list1` 对象和 `list2` 对象的

`getClass()` 方法获取他们的类的信息，最后发现结果为 `true` 。说明泛型类型 `String` 和 `Integer` 都被擦除掉了，只剩下原始类型。

## 2、通过反射添加其它类型元素

```
public class Test {

    public static void main(String[] args) throws Exception {

        ArrayList<Integer> list = new ArrayList<Integer>();

        list.add(1);  //这样调用 add 方法只能存储整形，因为泛型类型的实例为 Integer

        list.getClass().getMethod("add", Object.class).invoke(list, "asd");

        for (int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }
    }

}
```

在程序中定义了一个 `ArrayList` 泛型类型实例化为 `Integer` 对象，如果直接调用 `add()` 方法，那么只能存储整数数据，不过当我们利用反射调用 `add()` 方法的时候，却可以存储字符串，这说明了 `Integer` 泛型实例在编译之后被擦除掉了，只保留了原始类型。

# 二、类型擦除后保留的原始类型

在上面，两次提到了原始类型，什么是原始类型?

**原始类型** 就是擦除去了泛型信息，最后在字节码中的类型变量的真正类型，无论何时定义一个泛型，相应的原始类型都会被自动提供，类型变量擦除，并使用其**限定类型**（无限定的变量用Object）替换。

## 1、原始类型Object

```
public class Pair<T> {
    private T value;
    public T getValue() {
        return value;
    }
    public void setValue(T  value) {
        this.value = value;
    }
}
```

Pair的原始类型为:

```
public class Pair {
    private Object value;
    public Object getValue() {
        return value;
```

```
    }
    public void setValue(Object  value) {
        this.value = value;
    }
}
```

因为在 `Pair<T>` 中，T 是一个无限定的类型变量，所以用 `Object` 替换，其结果就是一个普通的类，如同泛型加入Java语言之前的已经实现的样子。在程序中可以包含不同类型的 `Pair` ，如 `Pair<String>` 或 `Pair<Integer>` ，但是擦除类型后他们的就成为原始的 `Pair` 类型了，原始类型都是 `Object` 。

从上面的"一-2"中，我们也可以明白 `ArrayList<Integer>` 被擦除类型后，原始类型也变为 `Object` ，所以通过反射我们就可以存储字符串了。

如果类型变量有限定，那么原始类型就用第一个边界的类型变量类替换。

比如: Pair这样声明的话

```
public class Pair<T extends Comparable> {}
```

那么原始类型就是 `Comparable` 。

要区分**原始类型**和**泛型变量的类型**。

在调用泛型方法时，可以指定泛型，也可以不指定泛型。

- 在不指定泛型的情况下，泛型变量的类型为该方法中的几种类型的同一父类的最小级，直到 Object。
- 在指定泛型的情况下，该方法的几种类型必须是该泛型的实例的类型或者其子类。

```
public class Test {
    public static void main(String[] args) {

        /**不指定泛型的时候*/
        int i = Test.add(1, 2); //这两个参数都是Integer，所以T为Integer类型
        Number f = Test.add(1, 1.2); //这两个参数一个是Integer，以风格是Float，所以取同一父类的最小级
        Object o = Test.add(1, "asd"); //这两个参数一个是Integer，以风格是Float，所以取同一父类的最

        /**指定泛型的时候*/
        int a = Test.<Integer>add(1, 2); //指定了Integer，所以只能为Integer类型或者其子类
        int b = Test.<Integer>add(1, 2.2); //编译错误，指定了Integer，不能为Float
        Number c = Test.<Number>add(1, 2.2); //指定为Number，所以可以为Integer和Float
    }

    //这是一个简单的泛型方法
    public static <T> T add(T x,T y){
        return y;
    }
}
```

其实在泛型类中，不指定泛型的时候，也差不多，只不过这个时候的泛型为 `Object` ，就比如 `ArrayList` 中，如果不指定泛型，那么这个 `ArrayList` 可以存储任意的对象。

## 2、Object泛型

```
public static void main(String[] args) {
    ArrayList list = new ArrayList();
    list.add(1);
    list.add("121");
    list.add(new Date());
}
```

# 三、类型擦除引起的问题及解决方法

因为种种原因，Java不能实现真正的泛型，只能使用类型擦除来实现伪泛型，这样虽然不会有类型膨胀问题，但是也引起来许多新问题，所以，SUN对这些问题做出了种种限制，避免我们发生各种错误。

## 1、先检查再编译以及编译的对象和引用传递问题

**Q**: 既然说类型变量会在编译的时候擦除掉，那为什么我们往 ArrayList 创建的对象中添加整数会报错呢？不是说泛型变量String会在编译的时候变为Object类型吗？为什么不能存别的类型呢？既然类型擦除了，如何保证我们只能使用泛型变量限定的类型呢？

**A**: Java编译器是通过先检查代码中泛型的类型，然后在进行类型擦除，再进行编译。

例如：

```
public static  void main(String[] args) {

    ArrayList<String> list = new ArrayList<String>();
    list.add("123");
    list.add(123);//编译错误
}
```

在上面的程序中，使用 `add` 方法添加一个整型，在IDE中，直接会报错，说明这就是在编译之前的检查，因为如果是在编译之后检查，类型擦除后，原始类型为 `Object` ，是应该允许任意引用类型添加的。可实际上却不是这样的，这恰恰说明了关于泛型变量的使用，是会在编译之前检查的。

那么，这个类型检查是针对谁的呢？我们先看看参数化类型和原始类型的兼容。

以 ArrayList举例子，以前的写法:

```
ArrayList list = new ArrayList();
```

现在的写法:

```
ArrayList<String> list = new ArrayList<String>();
```

如果是与以前的代码兼容，各种引用传值之间，必然会出现如下的情况：

```
ArrayList<String> list1 = new ArrayList(); //第一种 情况
ArrayList list2 = new ArrayList<String>(); //第二种 情况
```

这样是没有错误的，不过会有个编译时警告。

不过在第一种情况，可以实现与完全使用泛型参数一样的效果，第二种则没有效果。

因为类型检查就是编译时完成的， `new ArrayList()` 只是在内存中开辟了一个存储空间，可以存储任何类型对象，而**真正设计类型检查的是它的引用**，因为我们是使用它引用 `list1` 来调用它的方法，比如说调用 `add` 方法，所以 `list1` 引用能完成泛型类型的检查。而引用 `list2` 没有使用泛型，所以不行。

举例子：

```
public class Test {

    public static void main(String[] args) {

        ArrayList<String> list1 = new ArrayList();
        list1.add("1"); //编译通过
        list1.add(1); //编译错误
        String str1 = list1.get(0); //返回类型就是String

        ArrayList list2 = new ArrayList<String>();
        list2.add("1"); //编译通过
        list2.add(1); //编译通过
        Object object = list2.get(0); //返回类型就是Object

        new ArrayList<String>().add("11"); //编译通过
        new ArrayList<String>().add(22); //编译错误

        String str2 = new ArrayList<String>().get(0); //返回类型就是String
    }

}
```

通过上面的例子，我们可以明白，**类型检查就是针对引用的**，谁是一个引用，用这个引用调用泛型方法，就会对这个引用调用的方法进行类型检测，而无关它真正引用的对象。

泛型中参数话类型为什么不考虑继承关系？

在Java中，像下面形式的引用传递是不允许的：

```
ArrayList<String> list1 = new ArrayList<Object>(); //编译错误
ArrayList<Object> list2 = new ArrayList<String>(); //编译错误
```

我们先看第一种情况，将第一种情况拓展成下面的形式：

```
ArrayList<Object> list1 = new ArrayList<Object>();
list1.add(new Object());
list1.add(new Object());
ArrayList<String> list2 = list1; //编译错误
```

实际上，在第4行代码的时候，就会有编译错误。那么，我们先假设它编译没错。那么当我们使用 `list2` 引用用 `get()` 方法取值的时候，返回的都是 `String` 类型的对象（上面提到了，类型检测是根据引用来决定的），可是它里面实际上已经被我们存放了 `Object` 类型的对象，这样就会有 `ClassCastException` 了。所以为了避免这种极易出现的错误，Java不允许进行这样的引用传递。（这也是泛型出现的原因，就是为了解决类型转换的问题，我们不能违背它的初衷）。

再看第二种情况，将第二种情况拓展成下面的形式：

```
ArrayList<String> list1 = new ArrayList<String>();
list1.add(new String());
list1.add(new String());

ArrayList<Object> list2 = list1; //编译错误
```

没错，这样的情况比第一种情况好的多，最起码，在我们用 `list2` 取值的时候不会出现 `ClassCastException` ，因为是从 `String` 转换为 `Object` 。可是，这样做有什么意义呢，泛型出现的原因，就是为了解决类型转换的问题。我们使用了泛型，到头来，还是要自己强转，违背了泛型设计的初衷。所以java不允许这么干。再说，你如果又用 `list2` 往里面 `add()` 新的对象，那么到时候取得时候，我怎么知道我取出来的到底是 `String` 类型的，还是 `Object` 类型的呢?

**所以，要格外注意，泛型中的引用传递的问题。**

## 2、自动类型转换

因为类型擦除的问题，所以所有的泛型类型变量最后都会被替换为原始类型。

既然都被替换为原始类型，那么为什么我们在获取的时候，不需要进行强制类型转换呢?

看下 `ArrayList.get()` 方法:

```
public E get(int index) {

    RangeCheck(index);

    return (E) elementData[index];

}
```

可以看到，在 `return` 之前，会根据泛型变量进行强转。假设泛型类型变量为 `Date` ，虽然泛型信息会被擦除掉，但是会将 `(E) elementData[index]` ，编译为 `(Date) elementData[index]` 。所以我们不用自己进行强转。当存取一个泛型域时也会自动插入强制类型转换。假设 `Pair` 类的 `value` 域是 `public` 的，那么表达式:

```
Date date = pair.value;
```

也会自动地在结果字节码中插入强制类型转换。

## 3、类型擦除与多态的冲突和解决方法

现在有这样一个泛型类:

```
class Pair<T> {

    private T value;

    public T getValue() {
        return value;
    }
```

```
    public void setValue(T value) {
        this.value = value;
    }
}
```

然后我们想要一个子类继承它。

```
class DateInter extends Pair<Date> {

    @Override
    public void setValue(Date value) {
        super.setValue(value);
    }

    @Override
    public Date getValue() {
        return super.getValue();
    }
}
```

在这个子类中，我们设定父类的泛型类型为 `Pair<Date>` ，在子类中，我们覆盖了父类的两个方法，我们的原意是这样的：将父类的泛型类型限定为 `Date` ，那么父类里面的两个方法的参数都为 `Date` 类型。

```
public Date getValue() {
    return value;
}

public void setValue(Date value) {
    this.value = value;
}
```

所以，我们在子类中重写这两个方法一点问题也没有，实际上，从他们的 `@Override` 标签中也可以看到，一点问题也没有，实际上是这样的吗？

分析：实际上，类型擦除后，父类的的泛型类型全部变为了原始类型 `Object` ，所以父类编译之后会变成下面的样子：

```
class Pair {
    private Object value;

    public Object getValue() {
        return value;
    }

    public void setValue(Object  value) {
        this.value = value;
    }
}
```

再看子类的两个重写的方法的类型：

```
@Override
public void setValue(Date value) {
    super.setValue(value);
}
@Override
public Date getValue() {
    return super.getValue();
}
```

先来分析 `setValue` 方法，父类的类型是 `Object` ，而子类的类型是 `Date` ，参数类型不一样，这如果实在普通的继承关系中，根本就不会是重写，而是重载。

我们在一个main方法测试一下：

```
public static void main(String[] args) throws ClassNotFoundException {
        DateInter dateInter = new DateInter();
        dateInter.setValue(new Date());
        dateInter.setValue(new Object()); //编译错误
}
```

如果是重载，那么子类中两个 `setValue` 方法，一个是参数 `Object` 类型，一个是 `Date` 类型，可是我们发现，根本就没有这样的一个子类继承自父类的Object类型参数的方法。所以说，却是是重写了，而不是重载了。

为什么会这样呢?

原因是这样的，我们传入父类的泛型类型是 `Date, Pair<Date>` ，我们的本意是将泛型类变为如下：

```
class Pair {
    private Date value;
    public Date getValue() {
        return value;
    }
    public void setValue(Date value) {
        this.value = value;
    }
}
```

然后再子类中重写参数类型为Date的那两个方法，实现继承中的多态。

可是由于种种原因，虚拟机并不能将泛型类型变为 `Date` ，只能将类型擦除掉，变为原始类型 `Object` 。这样，我们的本意是进行重写，实现多态。可是类型擦除后，只能变为了重载。这样，类型擦除就和多态有了冲突。JVM知道你的本意吗？知道！！！可是它能直接实现吗，不能！！！如果真的不能的话，那我们怎么去重写我们想要的 `Date` 类型参数的方法啊。

于是JVM采用了一个特殊的方法，来完成这项功能，那就是**桥方法。**

首先，我们用 `javap -c className` 的方式反编译下 `DateInter` 子类的字节码，结果如下：

```
class com.tao.test.DateInter extends com.tao.test.Pair<java.util.Date> {
  com.tao.test.DateInter();
    Code:
       0: aload_0
       1: invokespecial #8                  // Method com/tao/test/Pair."<init>":()V
```

```
      4: return

  public void setValue(java.util.Date);  //我们重写的setValue方法
    Code:
      0: aload_0
      1: aload_1
      2: invokespecial #16                 // Method com/tao/test/Pair.setValue:(Ljava/lang/Object;)V
      5: return

  public java.util.Date getValue();    //我们重写的getValue方法
    Code:
      0: aload_0
      1: invokespecial #23                 // Method com/tao/test/Pair.getValue:()Ljava/lang/Object;
      4: checkcast     #26                 // class java/util/Date
      7: areturn

  public java.lang.Object getValue();     //编译时由编译器生成的桥方法
    Code:
      0: aload_0
      1: invokevirtual #28                 // Method getValue:()Ljava/util/Date 去调用我们重写的getValue方法;
      4: areturn

  public void setValue(java.lang.Object);   //编译时由编译器生成的桥方法
    Code:
      0: aload_0
      1: aload_1
      2: checkcast     #26                 // class java/util/Date
      5: invokevirtual #30                 // Method setValue:(Ljava/util/Date; 去调用我们重写的setValue方法)V
      8: return
}
```

从编译的结果来看，我们本意重写 `setValue` 和 `getValue` 方法的子类，竟然有4个方法，其实不用惊奇，最后的两个方法，就是编译器自己生成的桥方法。可以看到桥方法的参数类型都是Object，也就是说，子类中真正覆盖父类两个方法的就是这两个我们看不到的桥方法。而在我们自己定义的 `setvalue` 和 `getValue` 方法上面的 `@Oveerride` 只不过是假象。而桥方法的内部实现，就只是去调用我们自己重写的那两个方法。

所以，**虚拟机巧妙的使用了桥方法，来解决了类型擦除和多态的冲突。**

不过，要提到一点，这里面的 `setValue` 和 `getValue` 这两个桥方法的意义又有不同。

`setValue` 方法是为了解决类型擦除与多态之间的冲突。

而 `getValue` 却有普遍的意义，怎么说呢，如果这是一个普通的继承关系：

那么父类的 `getValue` 方法如下：

```
public Object getValue() {
    return value;
}
```

而子类重写的方法是：

```
public Date getValue() {
    return super.getValue();
```

```
}
```

其实这在普通的类继承中也是普遍存在的重写，这就是协变。

关于协变：。。。。。。

并且，还有一点也许会有疑问，子类中的桥方法 `Object getValue()` 和 `Date getValue()` 是同时存在的，可是如果是常规的两个方法，他们的方法签名是一样的，也就是说虚拟机根本不能分别这两个方法。如果是我们自己编写Java代码，这样的代码是无法通过编译器的检查的，但是虚拟机却是允许这样做的，因为虚拟机通过参数类型和返回类型来确定一个方法，所以编译器为了实现泛型的多态允许自己做这个看起来“不合法”的事情，然后交给虚拟器去区别。

## 4、泛型类型变量不能是基本数据类型

不能用类型参数替换基本类型。就比如，没有 `ArrayList<double>` ，只有 `ArrayList<Double>` 。因为当类型擦除后， `ArrayList` 的原始类型变为 `Object` ，但是 `Object` 类型不能存储 `double` 值，只能引用 `Double` 的值。

## 5、编译时集合的instanceof

```
ArrayList<String> arrayList = new ArrayList<String>();
```

因为类型擦除之后， `ArrayList<String>` 只剩下原始类型，泛型信息 `String` 不存在了。

那么，编译时进行类型查询的时候使用下面的方法是错误的

```
if( arrayList instanceof ArrayList<String>)
```

## 6、泛型在静态方法和静态类中的问题

泛型类中的静态方法和静态变量不可以使用泛型类所声明的泛型类型参数

举例说明：

```
public class Test2<T> {
    public static T one;   //编译错误
    public static  T show(T one){ //编译错误
        return null;
    }
}
```

因为泛型类中的泛型参数的实例化是在定义对象的时候指定的，而静态变量和静态方法不需要使用对象来调用。对象都没有创建，如何确定这个泛型参数是何种类型，所以当然是错误的。

但是要注意区分下面的一种情况：

```
public class Test2<T> {

    public static <T >T show(T one){ //这是正确的
        return null;
    }
}
```

因为这是一个泛型方法，在泛型方法中使用的T是自己在方法中定义的 T，而不是泛型类中的T。

写一篇好博客不容易，转自：http://blog.csdn.net/wisgood/article/details/11762427

---