泛型相关问题

1、泛型类型引用传递问题

在 Java 中，像下面形式的引用传递是不允许的：

```
ArrayList<String> arrayList1=new ArrayList<Object>();//编译错误
ArrayList<Object> arrayList1=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> arrayList1=new ArrayList<Object>();
arrayList1.add(new Object());
arrayList1.add(new Object());
ArrayList<String> arrayList2=arrayList1;//编译错误
```

实际上，在第 4 行代码处，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 arrayList2 引用用 get()方法取值的时候，返回的都是 String 类型的对象，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）。

在看第二种情况，将第二种情况拓展成下面的形式：

```
ArrayList<String> arrayList1=new ArrayList<String>();
          arrayList1.add(new String());
          arrayList1.add(new String());
          ArrayList<Object> arrayList2=arrayList1;//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 arrayList2 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 arrayList2 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，要格外注意泛型中引用传递问题。

2、泛型类型变量不能是基本数据类型

就比如，没有 ArrayList<double>，只有 ArrayList<Double>。因为当类型擦除后，ArrayList 的原始类中的类型变量（T）替换为 Object，但 Object 类型不能存储 double 值。

3、运行时类型查询

```
ArrayList<String> arrayList=new ArrayList<String>();if( arrayList instanceof ArrayLi
```

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。

4、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***

```
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***泛型类型对象（例如ArrayList<Integer>）***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。

泛型相关面试题

1. Java 中的泛型是什么 ？ 使用泛型的好处是什么?

泛型是一种参数化类型的机制。它可以使得代码适用于各种类型，从而编写更加通用的代码，例如集合框架。

泛型是一种编译时类型确认机制。它提供了编译期的**类型安全**，确保在泛型类型（通常为泛型集合）上只能使用正确类型的对象，避免了在运行时出现 ClassCastException。

2、Java 的泛型是如何工作的 ？ 什么是类型擦除 ？

泛型的正常工作是依赖编译器在编译源码的时候，先进行类型检查，然后进行类型擦除并且在类型参数出现的地方插入强制转换的相关指令实现的。

编译器在编译时擦除了所有类型相关的信息，所以在运行时不存在任何类型相关的信息。例如 List<String>在运行时仅用一个 List 类型来表示。为什么要进行擦除呢？这是为了避免**类型膨胀**。

3. 什么是泛型中的限定通配符和非限定通配符 ？

限定通配符对类型进行了限制。有两种限定通配符，一种是<? extends T>它通过确保类型必须是 T 的子类来设定类型的上界，另一种是<? super T>它通过确保类型必须是 T 的父类来设定类型的下界。泛型类型必须用限定内的类型来进行初始化，否则会导致编译错误。另一方面<?>表示了非限定通配符，因为<?>可以用任意类型来替代。

4. List<? extends T>和 List <? super T>之间有什么区别 ？

这和上一个面试题有联系，有时面试官会用这个问题来评估你对泛型的理解，而不是直接问你什么是限定通配符和非限定通配符。这两个 List 的声明都是限定通配符的例子，List<? extends T>可以接受任何继承自 T 的类型的 List，而 List<? super T>可以接受任何 T 的父类构成的 List。例如 List<? extends Number>可以接受 List<Integer>或 List<Float>。在本段出现的连接中可以找到更多信息。

5. 如何编写一个泛型方法，让它能接受泛型参数并返回泛型类型?

编写泛型方法并不困难，你需要用泛型类型来替代原始类型，比如使用 T, E or K,V 等被广泛认可的类型占位符。泛型方法的例子请参阅 Java 集合类框架。最简单的情况下，一个泛型方法可能会像这样:

```
public V put(K key, V value) {
    return cache.put(key, value);
}
```

6. Java 中如何使用泛型编写带有参数的类?

这是上一道面试题的延伸。面试官可能会要求你用泛型编写一个类型安全的类，而不是编写一个泛型方法。关键仍然是使用泛型类型来代替原始类型，而且要使用 JDK 中采用的标准占位符。

7. 编写一段泛型程序来实现 LRU 缓存?

对于喜欢 Java 编程的人来说这相当于是一次练习。给你个提示，LinkedHashMap 可以用来实现固定大小的 LRU 缓存，当 LRU 缓存已经满了的时候，它会把最老的键值对移出缓存。LinkedHashMap 提供了一个称为 removeEldestEntry()的方法，该方法会被 put()和 putAll()调用来删除最老的键值对。

8. 你可以把 List<String>传递给一个接受 List<Object>参数的方法吗？(见上面说明)

对任何一个不太熟悉泛型的人来说，这个 Java 泛型题目看起来令人疑惑，因为乍看起来 String 是一种 Object，所以 List<String>应当可以用在需要 List<Object>的地方，但是事实并非如此。真这样做的话会导致编译错误。如果你再深一步考虑，你会发现 Java 这样做是有意义的，因为 List<Object>可以存储任何类型的对象包括 String, Integer 等等，而 List<String>却只能用来存储 Strings。

9. Array 中可以用泛型吗?

这可能是 Java 泛型面试题中最简单的一个了，当然前提是你要知道 Array 事实上并不支持泛型，这也是为什么 Joshua Bloch 在 Effective Java 一书中建议使用 List 来代替 Array，因为 List 可以提供编译期的类型安全保证，而 Array 却不能。

10. 如何阻止 Java 中的类型未检查的警告?

如果你把泛型和原始类型混合起来使用，例如下列代码，Java 5 的 javac 编译器会产生类型未检查的警告

，例如 List<String> rawList = new ArrayList()

注意: Hello.java 使用了未检查或称为不安全的操作;

这种警告可以使用@SuppressWarnings("unchecked")注解来屏蔽。

11、Java 中 List<Object>和原始类型 List 之间的区别?

原始类型和带参数类型<Object>之间的主要区别是，***在编译时编译器不会对原始类型进行类型安全检查***，却会对带参数的类型进行检查，通过使用 Object 作为类型，可以告知编译器该方法可以接受任何类型的对象，比如 String 或 Integer。这道题的考察点在于对泛型中原始类型的正确理解。它们之间的第二点区别是，你可以把任何带参数的泛型类型传递给接受原始类型 List 的方法，但却不能把 List<String>传递给接受 List<Object>的方法，因为会产生编译错误。

12、Java 中 List<?>和 List<Object>之间的区别是什么?

这道题跟上一道题看起来很像，实质上却完全不同。List<?> 是一个未知类型的 List，而 List<Object>其实是任意类型的 List。你可以把 List<String>, List<Integer>赋值给 List<?>，却不能把 List<String>赋值给 List<Object>。

```
List<?> listOfAnyType;
List<Object> listOfObject = new ArrayList<Object>();
List<String> listOfString = new ArrayList<String>();
List<Integer> listOfInteger = new ArrayList<Integer>();

listOfAnyType = listOfString; //legal
listOfAnyType = listOfInteger; //legal
listOfObjectType = (List<Object>) listOfString; //compiler error - in-convertible ty
```

13、List<String>和原始类型 List 之间的区别.

该题类似于"原始类型和带参数类型之间有什么区别"。带参数类型是类型安全的，而且其类型安全是由编译器保证的，但原始类型 List 却不是类型安全的。你不能把 String 之外的任何其它类型的 Object 存入

String 类型的 List 中，而你可以把任何类型的对象存入原始 List 中。使用泛型的带参数类型你不需要进行类型转换，但是对于原始类型，你则需要进行显式的类型转换。

```
List listOfRawTypes = new ArrayList();
listOfRawTypes.add("abc");
listOfRawTypes.add(123); //编译器允许这样 - 运行时却会出现异常
String item = (String) listOfRawTypes.get(0); //需要显式的类型转换
item = (String) listOfRawTypes.get(1); //抛 ClassCastException，因为 Integer 不能被转换

List<String> listOfString = new ArrayList();
listOfString.add("abcd");
listOfString.add(1234); //编译错误，比在运行时抛异常要好
item = listOfString.get(0); //不需要显式的类型转换 - 编译器自动转换
```

通配符

通配符上界

常规使用

```
public class Test {
    public static void printIntValue(List<? extends Number> list) {
        for (Number number : list) {
            System.out.print(number.intValue()+" ");
        }
        System.out.println();
    }
    public static void main(String[] args) {
        List<Integer> integerList=new ArrayList<Integer>();
        integerList.add(2);
        integerList.add(2);
        printIntValue(integerList);
        List<Float> floatList=new ArrayList<Float>();
        floatList.add((float) 3.3);
        floatList.add((float) 0.3);
        printIntValue(floatList);
    }
}
```

输出：

```
2 2
3 0
```

非法使用

```
public class Test {
    public static void fillNumberList(List<? extends Number> list) {
        list.add(new Integer(0));//编译错误
        list.add(new Float(1.0));//编译错误
    }
```

```
    public static void main(String[] args) {
        List<? extends Number> list=new ArrayList();
        list.add(new Integer(1));//编译错误
        list.add(new Float(1.0));//编译错误
    }
}
```

List<? extends Number>可以代表 List<Integer>或 List<Float>，为什么不能像其中加入 Integer 或者 Float 呢？

首先，我们知道 List<Integer>之中只能加入 Integer。并且如下代码是可行的：

```
List<? extends Number> list1=new ArrayList<Integer>();
List<? extends Number> list2=new ArrayList<Float>();
```

假设前面的例子没有编译错误，如果我们把 list1 或者 list2 传入方法 fillNumberList，显然都会出现类型不匹配的情况，***假设不成立***。

因此，我们得出结论：不能往 List<? extends T> 中添加任意对象，除了 null。

那为什么对 List<? extends T>进行迭代可以呢，因为子类必定有父类相同的接口，这正是我们所期望的。

通配符下界

常规使用

```
public class Test {
    public static void fillNumberList(List<? super Number> list) {
        list.add(new Integer(0));
        list.add(new Float(1.0));
    }
    public static void main(String[] args) {
        List<? super Number> list=new ArrayList();
        list.add(new Integer(1));
        list.add(new Float(1.1));
    }
}
```

可以添加 Number 的任何子类，为什么呢？

List<? super Number>可以代表 List<T>，其中 T 为 Number 父类，（虽然 Number 没有父类）。如果说，T 为 Number 的父类，我们想 List<T>中加入 Number 的子类肯定是可以的。

非法使用

对 List<? superT>进行迭代是不允许的。为什么呢？你知道用哪种接口去迭代 List 吗？只有用 Object 类的接口才能保证集合中的元素都拥有该接口，显然这个意义不大。其应用场景略。

无界通配符

知道了通配符的上界和下界，其实也等同于知道了无界通配符，不加任何修饰即可，单独一个“？”。如 List<?>，“？”可以代表任意类型，“任意”也就是未知类型。

List<Object>与List<?>并不等同，List<Object>是 List<?>的子类。还有不能往 List<?> list 里添加任意对象，除了 null。

常规使用

1、当方法是使用原始的 Object 类型作为参数时，如下：

```
public static void printList(List<Object> list) {
    for (Object elem : list)
        System.out.println(elem + "");
    System.out.println();
}
```

可以选择改为如下实现：

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + "");
    System.out.println();
}
```

这样就可以兼容更多的输出，而不单纯是 List<Object>,如下：

```
List<Integer> li = Arrays.asList(1, 2, 3);
List<String>  ls = Arrays.asList("one", "two", "three");
printList(li);
printList(ls);
```