# Java泛型

开发人员在使用泛型的时候，很容易根据自己的直觉而犯一些错误。比如一个方法如果接收List<Object>作为形式参数，那么如果尝试将一个List<String>的对象作为实际参数传进去，却发现无法通过编译。虽然从直觉上来说，Object是String的父类，这种类型转换应该是合理的。**但是实际上这会产生隐含的类型转换问题，因此编译器直接就禁止这样的行为**。

## 类型擦除

Java中的泛型基本上都是在编译器这个层次来实现的，**在生成的Java字节代码中是不包含泛型中的类型信息的。使用泛型的时候加上的类型参数，会被编译器在编译的时候去掉，这个过程就称为类型擦除**。如在代码中定义的List<Object>和List<String>等类型，在编译之后都会变成List。**JVM看到的只是List，而由泛型附加的类型信息对JVM来说是不可见的**。Java编译器会在编译时尽可能的发现可能出错的地方，但是仍然无法避免在运行时刻出现类型转换异常的情况。

很多泛型的奇怪特性都与这个类型擦除的存在有关，包括：

* **泛型类并没有自己独有的Class类对象**。比如并不存在List<String>.class或是List<Integer>.class，而只有List.class。
* **静态变量是被泛型类的所有实例所共享的**。对于声明为MyClass<T>的类，访问其中的静态变量的方法仍然是 MyClass.myStaticVar。不管是通过new MyClass<String>还是new MyClass<Integer>创建的对象，都是共享一个静态变量。
* **泛型的类型参数不能用在Java异常处理的catch语句中**。因为异常处理是由JVM在运行时刻来进行的。由于类型信息被擦除，JVM是无法区分两个异常类型MyException<String>和MyException<Integer>的。对于JVM来说，它们都是 MyException类型的。也就无法执行与异常对应的catch语句。

类型擦除的基本过程也比较简单，首先是找到用来替换类型参数的具体类。这个具体类一般是Object。如果指定了类型参数的上界的话，则使用这个上界。把代码中的类型参数都替换成具体的类。同时去掉出现的类型声明，即去掉<>的内容。比如T get()方法声明就变成了Object get()；List<String>就变成了List。接下来就可能需要生成一些桥接方法（bridge method）。这是由于擦除了类型之后的类可能缺少某些必须的方法。比如考虑下面的代码：

class MyString implements Comparable<String> { public int compareTo(String str) { return 0; } }

当类型信息被擦除之后，上述类的声明变成了class MyString implements Comparable。但是这样的话，类MyString就会有编译错误，因为没有实现接口Comparable声明的int compareTo(Object)方法。这个时候就由编译器来动态生成这个方法。

## 通配符

在使用泛型类的时候，既可以指定一个具体的类型，如List<String>就声明了具体的类型是String；也可以用通配符?来表示未知类型，如List<?>就声明了List中包含的元素类型是未知的。 通配符所代表的其实是一组类型，但具体的类型是未知的。List<?>所声明的就是所有类型都是可以的。**但是List<?>并不等同于List<Object>。List<Object>实际上确定了List中包含的是Object及其子类，在使用的时候都可以通过Object来进行引用。而List<?>则其中所包含的元素类型是不确定**。其中可能包含的是String，也可能是 Integer。如果它包含了String的话，往里面添加Integer类型的元素就是错误的。**正因为类型未知，就不能通过new ArrayList<?>()的方法来创建一个新的ArrayList对象。因为编译器无法知道具体的类型是什么。但是对于 List<?>中的元素确总是可以用Object来引用的，因为虽然类型未知，但肯定是Object及其子类**。考虑下面的代码：

public void wildcard(List<?> list) {  
 list.add(1);//编译错误  
}

如上所示，试图对一个带通配符的泛型类进行操作的时候，总是会出现编译错误。其原因在于通配符所表示的类型是未知的。

因为对于List<?>中的元素只能用Object来引用，在有些情况下不是很方便。在这些情况下，可以使用上下界来限制未知类型的范围。 如 **List<? extends Number>说明List中可能包含的元素类型是Number及其子类。而List<? super Number>则说明List中包含的是Number及其父类**。当引入了上界之后，在使用类型的时候就可以使用上界类中定义的方法。

## 类型系统

在Java中，大家比较熟悉的是通过继承机制而产生的类型体系结构。比如String继承自Object。根据Liskov替换原则，子类是可以替换父类的。当需要Object类的引用的时候，如果传入一个String对象是没有任何问题的。但是反过来的话，即用父类的引用替换子类引用的时候，就需要进行强制类型转换。编译器并不能保证运行时刻这种转换一定是合法的。**这种自动的子类替换父类的类型转换机制，对于数组也是适用的。 String[]可以替换Object[]**。但是泛型的引入，对于这个类型系统产生了一定的影响。**正如前面提到的List是不能替换掉List的**。

引入泛型之后的类型系统增加了两个维度：**一个是类型参数自身的继承体系结构，另外一个是泛型类或接口自身的继承体系结构**。第一个指的是对于 List<String>和List<Object>这样的情况，类型参数String是继承自Object的。而第二种指的是 List接口继承自Collection接口。对于这个类型系统，有如下的一些规则：

* **相同类型参数的泛型类的关系取决于泛型类自身的继承体系结构**。即List<String>是Collection<String> 的子类型，List<String>可以替换Collection<String>。这种情况也适用于带有上下界的类型声明。
* **当泛型类的类型声明中使用了通配符的时候，其子类型可以在两个维度上分别展开**。如对Collection<? extends Number>来说，其子类型可以在Collection这个维度上展开，即List<? extends Number>和Set<? extends Number>等；也可以在Number这个层次上展开，即Collection<Double>和Collection<Integer>等。如此循环下去，ArrayList<Long>和 HashSet<Double>等也都算是Collection<? extends Number>的子类型。
* 如果泛型类中包含多个类型参数，则对于每个类型参数分别应用上面的规则。