### **一、类型擦除**

Java的泛型是伪泛型。  
问题：为什么说Java的泛型是伪泛型？  
因为，在编译期间，所有的泛型信息都会被擦除掉。

类型擦除的定义：

Java字节码中是不包含泛型的类型信息。使用泛型的时候加上的类型参数，会在编译器在编译的时候去掉。这个过程就称为类型擦除。

例：

ArrayList<String> aS = new ArrayList<>();

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

if(aS.getClass()==aI.getClass()){

System.out.println(aS.getClass().getName());

System.out.println("类型擦除，Integer、String被擦除。返回的类型是原始类型：ArrayList");

}

aI.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

try {

//add()参数被类型擦除，Integer被擦去，用Object代替成为add()的参数

//因此利用反射却可以储存字符串等其他类型，即：字节码中

aI.getClass().getMethod("add", Object.class).invoke(aI, "字符");

} catch (IllegalAccessException | IllegalArgumentException | InvocationTargetException

| NoSuchMethodException | SecurityException e){

e.printStackTrace();

}

for(int i=0;i<aI.size();i++){

System.out.print(aI.get(i)+" ");

}

//打印出 1 字符

**二、类型擦除后出现的原始类型**

**原始类型：**就是擦除了泛型信息，在编译后的字节码中的类型变量的真正类型

一个泛型类型再编译时，类型变量会被擦除。然后由编译器用限定类型替换（无限定时，使用Object）。

**例，一个泛型类型：**

class Pair<T> {

private T value;

public T getValue() {

return value;

}

public void setValue(T value) {

this.value = value;

}

}

Pari<T>的原始类型：

class Pair {

private Object value;

public Object getValue() {

return value;

}

public void setValue(Object value) {

this.value = value;

}

}

**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来替换**

**例：**

class Pair<T extends Comparable& Serializable> {

private T value;

public T getValue() {

return value;

}

public void setValue(T value) {

this.value = value;

}

}

Pair<T extends Comparable& Serializable>的原始类型：

class Pair {

    private Comparable value;

    public Comparable getValue() {

        return value;

    }

    public void setValue(Comparable  value) {

        this.value = value;

    }

}

（public class Pair<T extends Serializable&Comparable> ）   上边代码里就变成了Serializable

**区分原始类型和泛型变量的类型：**

调用泛型方法的时：

**1、指定泛型时**，该方法中的几种类型必须是该泛型实例类型或者其子类

**2、不指定泛型时**，泛型变量的类型为该方法中的几种类型的同一个父类的最小级，直到Object

public class Test2{

public static void main(String[] args) {

/\*\*不指定泛型的时候\*/

int i=Test2.add(1, 2); //这两个参数都是Integer，所以T为Integer类型

Number f=Test2.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，为Number

Object o=Test2.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小级，为Object

/\*\*指定泛型的时候\*/

int a=Test2.<Integer>add(1, 2);//指定了Integer，所以只能为Integer类型或者其子类

int b=Test2.<Integer>add(1, 2.2);//编译错误，指定了Integer，不能为Float

Number c=Test2.<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;

}

}

**在泛型类中，不指定泛型的时，泛型类型为Object**

**如ArrayList中，如果不指定泛型，那么这个ArrayList中可以放任意类型的对象**

**例：**

public static void main(String[] args) {

ArrayList arrayList=new ArrayList();

arrayList.add(1);

arrayList.add("121");

arrayList.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的问题，但是也引起了许多新的问题。所以，Sun对这些问题作出了许多限制，避免我们犯各种错误。

**1、先检查，在编译，以及检查编译的对象和引用传递的问题**

类型变量会在编译的时候擦除掉，引发以下思考：

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

**问题：**

为什么arrayList中，不能使用add方法添加整型？

泛型变量Integer会在编译时候擦除，变为原始类型Object，那么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？

Java是如何解决这些问题的呢？

Java编译器是通过先检查代码中泛型的类型符不符合，然后进行类型擦除，再进行编译的。

**例：**

public static void main(String[] args) {

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

arrayList.add("123");

arrayList.add(123);//编译之前的检查 出现错误

}

Java会在编译前检查

如果是在编译之后检查，类型擦除后，原始类型为Object，是应该运行任意引用类型的添加的。可实际上却不是这样，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。

**类型检查是针对谁的？**

先看看参数化类型与原始类型的兼容：

以ArrayList举例：

以前的写法：

ArrayList arrayList=new ArrayList();

现在的写法：

ArrayList<String> arrayList=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：

（这样是没有错误的，不过会有个编译时警告。）

ArrayList<String> arrayList1=new ArrayList(); //第一种情况：可以实现与 完全使用泛型参数一样的效果

ArrayList arrayList2=new ArrayList<String>();//第二种情况 ：完全没效果

因为，类型检查就是编译时完成的。  
new ArrayList()只是在内存中开辟一个存储空间，可以存储任何的类型对象。  
**而真正涉及类型检查的是它的引用**  
因为我们是使用它引用arrayList1 来调用它的方法，比如说调用add()方法。所以arrayList1引用能完成泛型类型的检查。

而引用arrayList2没有使用泛型，所以不行。

**例：**

public class Test10 {

public static void main(String[] args) {

//

ArrayList<String> arrayList1=new ArrayList();

arrayList1.add("1");//编译通过

arrayList1.add(1);//编译错误

String str1=arrayList1.get(0);//返回类型就是String

ArrayList arrayList2=new ArrayList<String>();

arrayList2.add("1");//编译通过

arrayList2.add(1);//编译通过

Object object=arrayList2.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 string=new ArrayList<String>().get(0);//返回类型就是String

}

}

通过以上例子，可以知道：**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。**

**再讨论下，泛型中参数化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> arrayList1=new ArrayList<Object>();//编译错误

ArrayList<Object> arrayList1=new ArrayList<String>();//编译错误

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

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arrayList2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）。

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，泛型中的引用传递的问题。**

**2、自动类型转换**

**3、类型擦除与多态的冲突和解决方法（桥方法）**

**4、泛型类型变量不能是基本数据类型**

**5、运行时类型查询**

**6、异常中使用泛型的问题**

**1、不能抛出也不能捕获泛型类的对象**

**2、不能再catch子句中使用泛型变量**

**7、不能声明参数化类型的数组（不属于类型擦除引起的问题）**

**8、不能实例化泛型类型**

**9、类型擦除后的冲突**

**1、当泛型类型被擦除后，创建条件不能产生冲突**

**2、泛型规范说明提及另一个原则“要支持擦除的转换，需要强行制一个类或者类型变量不能同时成为两个接口的子类，而这两个子类是同一接品的不同参数化。”**

**10、泛型在静态方法和静态类中的问题**

**学习笔记**

**参考：http://blog.csdn.net/lonelyroamer/article/details/7868820#**