**参考：java核心技术**

**一、Java泛型的实现方法：类型擦除**

前面已经说了，**Java**的泛型是伪泛型。为什么说Java的泛型是伪泛型呢？因为，在编译期间，所有的泛型信息都会被擦除掉。正确理解泛型概念的首要前提是理解类型擦除（type erasure）。

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，会在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，在编译后都会编程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。类型擦除也是Java的泛型实现方法与C++模版机制实现方式之间的重要区别。

可以通过两个简单的例子，来证明java泛型的类型擦除。

例1、

1. **public** **class** Test4 {
2. **public** **static** **void** main(String[] args) {
3. ArrayList<String> arrayList1=**new** ArrayList<String>();
4. arrayList1.add("abc");
5. ArrayList<Integer> arrayList2=**new** ArrayList<Integer>();
6. arrayList2.add(123);
7. System.out.println(arrayList1.getClass()==arrayList2.getClass());
8. }
9. }

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，只能存储字符串。一个是ArrayList<Integer>泛型类型，只能存储整形。最后，我们通过arrayList1对象和arrayList2对象的getClass方法获取它们的类的信息，最后发现结果为true。说明泛型类型String和Integer都被擦除掉了，只剩下了原始类型。

例2、

1. **public** **class** Test4 {
2. **public** **static** **void** main(String[] args) **throws** IllegalArgumentException, SecurityException, IllegalAccessException, InvocationTargetException, NoSuchMethodException {
3. ArrayList<Integer> arrayList3=**new** ArrayList<Integer>();
4. arrayList3.add(1);//这样调用add方法只能存储整型，因为泛型类型的实例为Integer
5. arrayList3.getClass().getMethod("add", Object.**class**).invoke(arrayList3, "asd");
6. **for** (**int** i=0;i<arrayList3.size();i++) {
7. System.out.println(arrayList3.get(i));
8. }
9. }

在程序中定义了一个ArrayList泛型类型实例化为Integer的对象，如果直接调用add方法，那么只能存储整型的数据。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（raw type）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类型，相应的原始类型都会被自动地提供。类型变量被擦除（crased），并使用其限定类型（无限定的变量用Object）替换。

例3：

1. **class** Pair<T> {
2. **private** T value;
3. **public** T getValue() {
4. **return** value;
5. }
6. **public** **void** setValue(T  value) {
7. **this**.value = value;
8. }
9. }

Pair<T>的原始类型为：

1. **class** Pair {
2. **private** Object value;
3. **public** Object getValue() {
4. **return** value;
5. }
6. **public** **void** setValue(Object  value) {
7. **this**.value = value;
8. }
9. }

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变成语言之前已经实现的那样。在程序中可以包含不同类型的Pair，如Pair<String>或Pair<Integer>，但是，擦除类型后它们就成为原始的Pair类型了，原始类型都是Object。

从上面的那个例2中，我们也可以明白ArrayList<Integer>被擦除类型后，原始类型也变成了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来替换。

比如Pair这样声明

例4：

1. **public** **class** Pair<T **extends** Comparable& Serializable> {

那么原始类型就是Comparable

注意：

如果Pair这样声明public class Pair<T extends Serializable&Comparable> ，那么原始类型就用Serializable替换，而编译器在必要的时要向Comparable插入强制类型转换。为了提高效率，应该将标签（tagging）接口（即没有方法的接口）放在边界限定列表的末尾。

**要区分原始类型和泛型变量的类型**

在调用泛型方法的时候，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个父类的最小级，直到Object。

在指定泛型的时候，该方法中的几种类型必须是该泛型实例类型或者其子类。

1. **public** **class** Test2{
2. **public** **static** **void** main(String[] args) {
3. /\*\*不指定泛型的时候\*/
4. **int** i=Test2.add(1, 2); //这两个参数都是Integer，所以T为Integer类型
5. 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
6. 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
8. /\*\*指定泛型的时候\*/
9. **int** a=Test2.<Integer>add(1, 2);//指定了Integer，所以只能为Integer类型或者其子类
10. **int** b=Test2.<Integer>add(1, 2.2);//编译错误，指定了Integer，不能为Float
11. Number c=Test2.<Number>add(1, 2.2); //指定为Number，所以可以为Integer和Float
12. }
14. //这是一个简单的泛型方法
15. **public** **static** <T> T add(T x,T y){
16. **return** y;
17. }
18. }

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类型为Object，就比如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。

举例：

1. **public** **static** **void** main(String[] args) {
2. ArrayList arrayList=**new** ArrayList();
3. arrayList.add(1);
4. arrayList.add("121");
5. arrayList.add(**new** Date());
6. }

**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**

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<String> arrayList=new ArrayList<String>();所创建的数组列表arrayList中，不能使用add方法添加整形呢？不是说泛型变量Integer会在编译时候擦除变为原始类型Object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？

java是如何解决这个问题的呢？java编译器是通过先检查代码中泛型的类型，然后再进行类型擦除，在进行编译的。

举个例子说明：

1. **public** **static**  **void** main(String[] args) {
2. ArrayList<String> arrayList=**new** ArrayList<String>();
3. arrayList.add("123");
4. arrayList.add(123);//编译错误
5. }

在上面的程序中，使用add方法添加一个整型，在eclipse中，直接就会报错，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。

那么，这么类型检查是针对谁的呢？我们先看看**参数化类型与原始类型的兼容**

以ArrayList举例子，以前的写法：

1. ArrayList arrayList=**new** ArrayList();

现在的写法：

1. 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：

1. ArrayList<String> arrayList1=**new** ArrayList(); //第一种 情况
2. ArrayList arrayList2=**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，第二种则完全没效果。

因为，本来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。

举例子：

1. **public** **class** Test10 {
2. **public** **static** **void** main(String[] args) {
4. ArrayList<String> arrayList1=**new** ArrayList();
5. arrayList1.add("1");//编译通过
6. arrayList1.add(1);//编译错误
7. String str1=arrayList1.get(0);//返回类型就是String
9. ArrayList arrayList2=**new** ArrayList<String>();
10. arrayList2.add("1");//编译通过
11. arrayList2.add(1);//编译通过
12. Object object=arrayList2.get(0);//返回类型就是Object
14. **new** ArrayList<String>().add("11");//编译通过
15. **new** ArrayList<String>().add(22);//编译错误
16. String string=**new** ArrayList<String>().get(0);//返回类型就是String
17. }
18. }

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。

**从这里，我们可以再讨论下 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：

1. ArrayList<String> arrayList1=**new** ArrayList<Object>();//编译错误
2. 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：

1. ArrayList<Object> arrayList1=**new** ArrayList<Object>();
2. arrayList1.add(**new** Object());
3. arrayList1.add(**new** Object());
4. ArrayList<String> arrayList2=arrayList1;//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）。

在看第二种情况，将第二种情况拓展成下面的形式：

1. ArrayList<String> arrayList1=**new** ArrayList<String>();
2. arrayList1.add(**new** String());
3. arrayList1.add(**new** String());
4. 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**

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。这样就引起了一个问题，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和get方法：

1. **public** E get(**int** index) {
2. RangeCheck(index);
3. **return** (E) elementData[index];
4. }

可以看到，在return之前，会根据泛型变量进行强转。

写了个简单的**测试**代码：

1. **public** **class** Test {
2. **public** **static** **void** main(String[] args) {
3. ArrayList<Date> list=**new** ArrayList<Date>();
4. list.add(**new** Date());
5. Date myDate=list.get(0);
6. }

然后反编了下字节码，如下

1. **public** **static** **void** main(java.lang.String[]);
2. Code:
3. 0: **new** #16 // class java/util/ArrayList
4. 3: dup
5. 4: invokespecial #18 // Method java/util/ArrayList."<init
6. :()V
7. 7: astore\_1
8. 8: aload\_1
9. 9: **new** #19 // class java/util/Date
10. 12: dup
11. 13: invokespecial #21 // Method java/util/Date."<init>":()
13. 16: invokevirtual #22 // Method java/util/ArrayList.add:(L
14. va/lang/Object;)Z
15. 19: pop
16. 20: aload\_1
17. 21: iconst\_0
18. 22: invokevirtual #26 // Method java/util/ArrayList.get:(I
19. java/lang/Object;
20. 25: checkcast #19 // class java/util/Date
21. 28: astore\_2
22. 29: **return**

看第22 ，它调用的是ArrayList.get()方法，方法返回值是Object，说明类型擦除了。然后第25，它做了一个checkcast操作，即检查类型#19， 在在上面找#19引用的类型，他是  
9: new #19 // class java/util/Date  
是一个Date类型，即做Date类型的强转。  
所以不是在get方法里强转的，是在你调用的地方强转的。

附关于checkcast的解释：  
checkcast checks that the top item on the operand stack (a reference to an object or array) can be cast to a given type. For example, if you write in Java:  
  
return ((String)obj);  
  
then the **Java**compiler will generate something like:  
  
aload\_1 ; push -obj- onto the stack  
checkcast java/lang/String ; check its a String  
areturn ; return it  
  
checkcast is actually a shortand for writing **java**code like:  
  
if (! (obj == null || obj instanceof <class>)) {  
throw new ClassCastException();  
}  
// if this point is reached, then object is either null, or an instance of  
// <class> or one of its superclasses.

**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：

1. **class** Pair<T> {
2. **private** T value;
3. **public** T getValue() {
4. **return** value;
5. }
6. **public** **void** setValue(T value) {
7. **this**.value = value;
8. }
9. }

然后我们想要一个子类继承它

1. **class** DateInter **extends** Pair<Date> {
2. @Override
3. **public** **void** setValue(Date value) {
4. **super**.setValue(value);
5. }
6. @Override
7. **public** Date getValue() {
8. **return** **super**.getValue();
9. }
10. }

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：“

1. **public** Date getValue() {
2. **return** value;
3. }
4. **public** **void** setValue(Date value) {
5. **this**.value = value;
6. }

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：

1. **class** Pair {
2. **private** Object value;
3. **public** Object getValue() {
4. **return** value;
5. }
6. **public** **void** setValue(Object  value) {
7. **this**.value = value;
8. }
9. }

再看子类的两个重写的方法的类型：

1. @Override
2. **public** **void** setValue(Date value) {
3. **super**.setValue(value);
4. }
5. @Override
6. **public** Date getValue() {
7. **return** **super**.getValue();
8. }

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：

1. **public** **static** **void** main(String[] args) **throws** ClassNotFoundException {
2. DateInter dateInter=**new** DateInter();
3. dateInter.setValue(**new** Date());
4. dateInter.setValue(**new** Object());//编译错误
5. }

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，确实是重载了，而不是重写了。（同一个方法名，不同的参数类型即为重载）

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：

1. **class** Pair {
2. **private** Date value;
3. **public** Date getValue() {
4. **return** value;
5. }
6. **public** **void** setValue(Date value) {
7. **this**.value = value;
8. }
9. }

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：

1. **class** com.tao.test.DateInter **extends** com.tao.test.Pair<java.util.Date> {
2. com.tao.test.DateInter();
3. Code:
4. 0: aload\_0
5. 1: invokespecial #8                  // Method com/tao/test/Pair."<init>"
6. :()V
7. 4: **return**
9. **public** **void** setValue(java.util.Date);  //我们重写的setValue方法
10. Code:
11. 0: aload\_0
12. 1: aload\_1
13. 2: invokespecial #16                 // Method com/tao/test/Pair.setValue
14. :(Ljava/lang/Object;)V
15. 5: **return**
17. **public** java.util.Date getValue();    //我们重写的getValue方法
18. Code:
19. 0: aload\_0
20. 1: invokespecial #23                 // Method com/tao/test/Pair.getValue
21. :()Ljava/lang/Object;
22. 4: checkcast     #26                 // class java/util/Date
23. 7: areturn
25. **public** java.lang.Object getValue();     //编译时由编译器生成的巧方法
26. Code:
27. 0: aload\_0
28. 1: invokevirtual #28                 // Method getValue:()Ljava/util/Date 去调用我们重写的getValue方法
29. ;
30. 4: areturn
32. **public** **void** setValue(java.lang.Object);   //编译时由编译器生成的巧方法
33. Code:
34. 0: aload\_0
35. 1: aload\_1
36. 2: checkcast     #26                 // class java/util/Date
37. 5: invokevirtual #30                 // Method setValue:(Ljava/util/Date;   去调用我们重写的setValue方法
38. )V
39. 8: **return**
40. }

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。而打在我们自己定义的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巧方法，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setValue方法如下：

1. **public** ObjectgetValue() {
2. **return** **super**.getValue();
3. }

而子类重写的方法是：

1. **public** Date getValue() {
2. **return** **super**.getValue();
3. }

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，子类中的巧方法  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、运行时类型查询**

举个例子:

1. 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，运行时进行类型查询的时候使用下面的方法是错误的

1. **if**( arrayList **instanceof** ArrayList<String>)

java限定了这种类型查询的方式

1. **if**( arrayList **instanceof** ArrayList<?>)

？ 是通配符的形式 ，将在后面一篇中介绍。

**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。事实上，泛型类扩展Throwable都不合法。例如：下面的定义将不会通过编译：

1. **public** **class** Problem<T> **extends** Exception{......}

为什么不能扩展Throwable，因为异常都是在运行时捕获和抛出的，而在编译的时候，泛型信息全都会被擦除掉，那么，假设上面的编译可行，那么，在看下面的定义：

1. **try**{
2. }**catch**(Problem<Integer> e1){
3. 。。
4. }**catch**(Problem<Number> e2){
5. ...
6. }

类型信息被擦除后，那么两个地方的catch都变为原始类型Object，那么也就是说，这两个地方的catch变的一模一样,就相当于下面的这样

**[java]** view plain copy

1. **try**{
2. }**catch**(Problem<Object> e1){
3. 。。
4. }**catch**(Problem<Object> e2){
5. ...

这个当然就是不行的。就好比，catch两个一模一样的普通异常，不能通过编译一样：

1. **try**{
2. }**catch**(Exception e1){
3. 。。
4. }**catch**(Exception  e2){//编译错误
5. ...

2、不能再catch子句中使用泛型变量

1. **public** **static** <T **extends** Throwable> **void** doWork(Class<T> t){
2. **try**{
3. ...
4. }**catch**(T e){ //编译错误
5. ...
6. }
7. }

因为泛型信息在编译的时候已经变味原始类型，也就是说上面的T会变为原始类型Throwable，那么如果可以再catch子句中使用泛型变量，那么，下面的定义呢：

1. **public** **static** <T **extends** Throwable> **void** doWork(Class<T> t){
2. **try**{
3. ...
4. }**catch**(T e){ //编译错误
5. ...
6. }**catch**(IndexOutOfBounds e){
7. }
8. }

根据异常捕获的原则，一定是子类在前面，父类在后面，那么上面就违背了这个原则。即使你在使用该静态方法的使用T是ArrayIndexOutofBounds，在编译之后还是会变成Throwable，ArrayIndexOutofBounds是IndexOutofBounds的子类，违背了异常捕获的原则。所以java为了避免这样的情况，禁止在catch子句中使用泛型变量。

但是在异常声明中可以使用类型变量。下面方法是合法的。

1. **public** **static**<T **extends** Throwable> **void** doWork(T t) **throws** T{
2. **try**{
3. ...
4. }**catch**(Throwable realCause){
5. t.initCause(realCause);
6. **throw** t;
7. }

上面的这样使用是没问题的。

**7、数组（这个不属于类型擦除引起的问题）**

不能声明参数化类型的数组。如：

1. Pair<String>[] table = newPair<String>(10); //ERROR

这是因为擦除后，table的类型变为Pair[]，可以转化成一个Object[]。

1. Object[] objarray =table;

  数组可以记住自己的元素类型，下面的赋值会抛出一个ArrayStoreException异常。

1. objarray ="Hello"; //ERROR

  对于泛型而言，擦除降低了这个机制的效率。下面的赋值可以通过数组存储的检测，但仍然会导致类型错误。

1. objarray =**new** Pair<Employee>();

提示：如果需要收集参数化类型对象，直接使用ArrayList：ArrayList<Pair<String>>最安全且有效。

**8、泛型类型的实例化**

不能实例化泛型类型。如，

1. first = **new** T(); //ERROR

   是错误的，类型擦除会使这个操作做成new Object()。  
   不能建立一个泛型数组。

1. **public**<T> T[] minMax(T[] a){
2. T[] mm = **new** T[2]; //ERROR
3. ...
4. }

   类似的，擦除会使这个方法总是构靠一个Object[2]数组。但是，可以用反射构造泛型对象和数组。  
   利用反射，调用Array.newInstance:

1. publicstatic <T **extends** Comparable> T[]minmax(T[] a)
3. {
5. T[] mm == (T[])Array.newInstance(a.getClass().getComponentType(),2);
7. ...
9. // 以替换掉以下代码
11. // Obeject[] mm = new Object[2];
13. // return (T[]) mm;
15. }

**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。如果在Pair类中添加下面的equals方法：

1. **class** Pair<T>   {
2. **public** **boolean** equals(T value) {
3. **return** **null**;
4. }
6. }

考虑一个Pair<String>。从概念上，它有两个equals方法：

booleanequals(String); //在Pair<T>中定义

boolean equals(Object); //从object中继承

但是，这只是一种错觉。实际上，擦除后方法

boolean equals(T)

变成了方法 boolean equals(Object)

这与Object.equals方法是冲突的！当然，补救的办法是重新命名引发错误的方法。

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。”

下面的代码是非法的：

1. **class** Calendar **implements** Comparable<Calendar>{ ... }
2. **class** GregorianCalendar **extends** Calendar **implements** Comparable<GregorianCalendar>{...} //ERROR

GregorianCalendar会实现Comparable<Calender>和Compable<GregorianCalendar>，这是同一个接口的不同参数化实现。

这一限制与类型擦除的关系并不很明确。非泛型版本：

1. **class** Calendar **implements** Comparable{ ... }
2. **class** GregorianCalendar **extends** Calendar **implements** Comparable{...} //ERROR

是合法的。

**10、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：

1. **public** **class** Test2<T> {
2. **public** **static** T one;   //编译错误
3. **public** **static**  T show(T one){ //编译错误
4. **return** **null**;
5. }
6. }

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：

1. **public** **class** Test2<T> {
2. **public** **static** <T>T show(T one){//这是正确的
3. **return** **null**;
4. }
5. }

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。