Java 基础 - 泛型机制详解

Java泛型这个特性是从JDK 1.5才开始加入的，因此为了兼容之前的版本，Java泛型的实现采取了“**伪泛型**”的策略，即Java在语法上支持泛型，但是在编译阶段会进行所谓的“**类型擦除**”（Type Erasure），将所有的泛型表示（尖括号中的内容）都替换为具体的类型（其对应的原生态类型），就像完全没有泛型一样。

为什么会引入泛型

> 泛型的本质是为了参数化类型（在不创建新的类型的情况下，通过泛型指定的不同类型来控制形参具体限制的类型）。也就是说在泛型使用过程中，操作的数据类型被指定为一个参数，这种参数类型可以用在类、接口和方法中，分别被称为泛型类、泛型接口、泛型方法。

引入泛型的意义在于：

- **适用于多种数据类型执行相同的代码**（代码复用）

我们通过一个例子来阐述，先看下下面的代码：

```
private static int add(int a, int b) {
    System.out.println(a + "+" + b + "=" + (a + b));
    return a + b;
}

private static float add(float a, float b) {
    System.out.println(a + "+" + b + "=" + (a + b));
    return a + b;
}

private static double add(double a, double b) {
    System.out.println(a + "+" + b + "=" + (a + b));
    return a + b;
}
```

如果没有泛型，要实现不同类型的加法，每种类型都需要重载一个add方法；通过泛型，我们可以复用为一个方法：

```
private static <T extends Number> double add(T a, T b) {
    System.out.println(a + "+" + b + "=" + (a.doubleValue() + b.doubleValue()));
    return a.doubleValue() + b.doubleValue();
}
```

- 泛型中的类型在使用时指定，不需要强制类型转换（**类型安全**，编译器会**检查类型**）

看下这个例子：

```
List list = new ArrayList();
list.add("xxString");
list.add(100d);
list.add(new Person());
```

我们在使用上述list中，list中的元素都是Object类型（无法约束其中的类型），所以在取出集合元素时需要人为的强制类型转化到具体的目标类型，且很容易出现java.lang.ClassCastException异常。

引入泛型，它将提供类型的约束，提供编译前的检查：

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

// list中只能放String，不能放其它类型的元素
```

泛型的基本使用

TIP

我们通过一些例子来学习泛型的使用；泛型有三种使用方式，分别为：泛型类、泛型接口、泛型方法。一些例子可以参考《李兴华 - Java实战经典》。@pdai

泛型类

- 从一个简单的泛型类看起：

```
class Point<T>{         // 此处可以随便写标识符号，T是type的简称
    private T var ;     // var的类型由T指定，即：由外部指定
    public T getVar(){  // 返回值的类型由外部决定
        return var ;
    }
    public void setVar(T var){  // 设置的类型也由外部决定
        this.var = var ;
    }
}
public class GenericsDemo06{
    public static void main(String args[]){
        Point<String> p = new Point<String>() ;     // 里面的var类型为String类型
        p.setVar("it") ;                            // 设置字符串
        System.out.println(p.getVar().length()) ;   // 取得字符串的长度
    }
}
```

- 多元泛型

```
class Notepad<K,V>{       // 此处指定了两个泛型类型
    private K key ;     // 此变量的类型由外部决定
    private V value ;   // 此变量的类型由外部决定
    public K getKey(){
        return this.key ;
    }
    public V getValue(){
        return this.value ;
```

```
    }
    public void setKey(K key){
        this.key = key ;
    }
    public void setValue(V value){
        this.value = value ;
    }
}
public class GenericsDemo09{
    public static void main(String args[]){
        Notepad<String,Integer> t = null ;        // 定义两个泛型类型的对象
        t = new Notepad<String,Integer>() ;       // 里面的key为String，value为Integer
        t.setKey("汤姆") ;        // 设置第一个内容
        t.setValue(20) ;            // 设置第二个内容
        System.out.print("姓名：" + t.getKey()) ;      // 取得信息
        System.out.print("，年龄：" + t.getValue()) ;       // 取得信息

    }
}
```

泛型接口

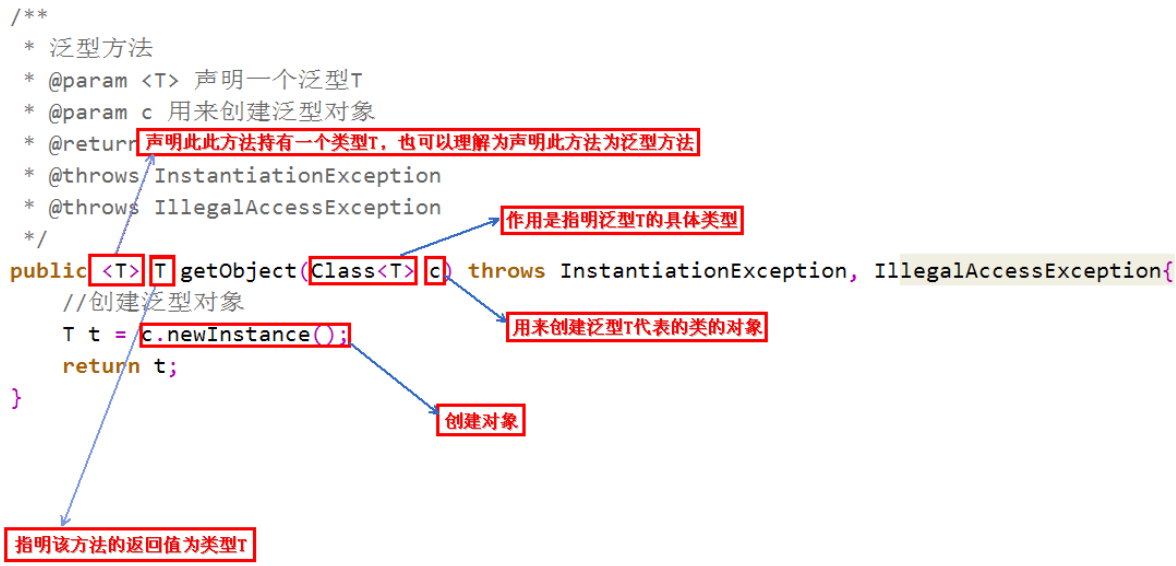
- 简单的泛型接口

```
interface Info<T>{        // 在接口上定义泛型
    public T getVar() ; // 定义抽象方法，抽象方法的返回值就是泛型类型
}
class InfoImpl<T> implements Info<T>{   // 定义泛型接口的子类
    private T var ;             // 定义属性
    public InfoImpl(T var){     // 通过构造方法设置属性内容
        this.setVar(var) ;
    }
    public void setVar(T var){
        this.var = var ;
    }
    public T getVar(){
        return this.var ;
    }
}
public class GenericsDemo24{
    public static void main(String arsg[]){
        Info<String> i = null;        // 声明接口对象
        i = new InfoImpl<String>("汤姆") ;  // 通过子类实例化对象
        System.out.println("内容：" + i.getVar()) ;
    }
}
```

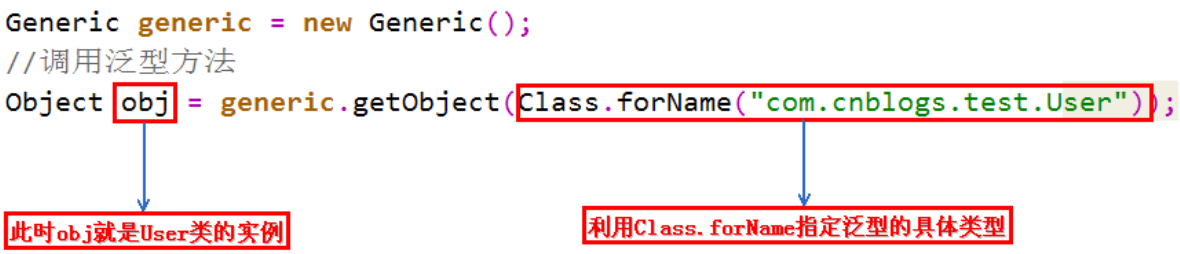
泛型方法

泛型方法，是在调用方法的时候指明泛型的具体类型。重点看下泛型的方法（图参考自：https://www.cnblogs.com/iyangyuan/archive/2013/04/09/3011274.html）

- 定义泛型方法语法格式



- 调用泛型方法语法格式



说明一下，定义泛型方法时，必须在返回值前边加一个<T>，来声明这是一个泛型方法，持有一个泛型T，然后才可以用泛型T作为方法的返回值。

Class<T>的作用就是指明泛型的具体类型，而Class<T>类型的变量c，可以用来创建泛型类的对象。

为什么要用变量c来创建对象呢？既然是泛型方法，就代表着我们不知道具体的类型是什么，也不知道构造方法如何，因此没有办法去new一个对象，但可以利用变量c的newInstance方法去创建对象，也就是利用反射创建对象。

泛型方法要求的参数是Class<T>类型，而Class.forName()方法的返回值也是Class<T>，因此可以用Class.forName()作为参数。其中，forName()方法中的参数是何种类型，返回的Class<T>就是何种类型。在本例中，forName()方法中传入的是User类的完整路径，因此返回的是Class<User>类型的对象，因此调用泛型方法时，变量c的类型就是Class<User>，因此泛型方法中的泛型T就被指明为User，因此变量obj的类型为User。

当然，泛型方法不是仅仅可以有一个参数Class<T>，可以根据需要添加其他参数。

为什么要使用泛型方法呢？因为泛型类要在实例化的时候就指明类型，如果想换一种类型，不得不重新new一次，可能不够灵活；而泛型方法可以在调用的时候指明类型，更加灵活。

泛型的上下限

- **先看下如下的代码，很明显是会报错的** （具体错误原因请参考后文）。

```
class A{}
class B extends A {}

// 如下两个方法不会报错
public static void funA(A a) {
    // ...
}
public static void funB(B b) {
    funA(b);
    // ...
}

// 如下funD方法会报错
public static void funC(List<A> listA) {
    // ...
}
public static void funD(List<B> listB) {
    funC(listB); // Unresolved compilation problem: The method doPrint(List<A>) in the type test is not applicable for the arguments (List<B>)
    // ...
}
```

那么如何解决呢?

为了解决泛型中隐含的转换问题，Java泛型加入了类型参数的上下边界机制。`<? extends A>`表示该类型参数可以是A(上边界)或者A的子类类型。编译时擦除到类型A，即用A类型代替类型参数。这种方法可以解决开始遇到的问题，编译器知道类型参数的范围，如果传入的实例类型B是在这个范围内的话允许转换，这时只要一次类型转换就可以了，运行时会把对象当做A的实例看待。

```
public static void funC(List<? extends A> listA) {
    // ...
}
public static void funD(List<B> listB) {
    funC(listB); // OK
    // ...
}
```

- **泛型上下限的引入**

在使用泛型的时候，我们可以为传入的泛型类型实参进行上下边界的限制，如：类型实参只准传入某种类型的父类或某种类型的子类。

上限

```
class Info<T extends Number>{    // 此处泛型只能是数字类型
    private T var ;        // 定义泛型变量
    public void setVar(T var){
        this.var = var ;
    }
    public T getVar(){
        return this.var ;
    }
    public String toString(){    // 直接打印
        return this.var.toString() ;
```

```
    }
}
public class demo1{
    public static void main(String args[]){
        Info<Integer> i1 = new Info<Integer>() ;        // 声明Integer的泛型对象
    }
}
```

下限

```
class Info<T>{
    private T var ;        // 定义泛型变量
    public void setVar(T var){
        this.var = var ;
    }
    public T getVar(){
        return this.var ;
    }
    public String toString(){    // 直接打印
        return this.var.toString() ;
    }
}
public class GenericsDemo21{
    public static void main(String args[]){
        Info<String> i1 = new Info<String>() ;        // 声明String的泛型对象
        Info<Object> i2 = new Info<Object>() ;        // 声明Object的泛型对象
        i1.setVar("hello") ;
        i2.setVar(new Object()) ;
        fun(i1) ;
        fun(i2) ;
    }
    public static void fun(Info<? super String> temp){    // 只能接收String或Object类型的泛型，
String类的父类只有Object类
        System.out.print(temp + ", ") ;
    }
}
```

小结

```
<?> 无限制通配符
<? extends E> extends 关键字声明了类型的上界，表示参数化的类型可能是所指定的类型，或者是此类型的子类
<? super E> super 关键字声明了类型的下界，表示参数化的类型可能是指定的类型，或者是此类型的父类

// 使用原则《Effictive Java》
// 为了获得最大限度的灵活性，要在表示 生产者或者消费者 的输入参数上使用通配符，使用的规则就是：生产者有上限、消费者有下限
1. 如果参数化类型表示一个 T 的生产者，使用 < ? extends T>;
2. 如果它表示一个 T 的消费者，就使用 < ? super T>;
3. 如果既是生产又是消费，那使用通配符就没什么意义了，因为你需要的是精确的参数类型。
```

- 再看一个实际例子，**加深印象**

```
private  <E extends Comparable<? super E>> E max(List<? extends E> e1) {
    if (e1 == null){
        return null;
    }
    //迭代器返回的元素属于 E 的某个子类型
    Iterator<? extends E> iterator = e1.iterator();
```

```
    E result = iterator.next();
    while (iterator.hasNext()){
        E next = iterator.next();
        if (next.compareTo(result) > 0){
            result = next;
        }
    }
    return result;
}
```

上述代码中的类型参数 E 的范围是`<E extends Comparable<? super E>>`，我们可以分步查看：

- 要进行比较，所以 E 需要是可比较的类，因此需要 `extends Comparable<…>`（注意这里不要和继承的 `extends` 搞混了，不一样）
- `Comparable< ? super E>` 要对 E 进行比较，即 E 的消费者，所以需要用 super
- 而参数 `List< ? extends E>` 表示要操作的数据是 E 的子类的列表，指定上限，这样容器才够大
- **多个限制**

使用&符号

```
public class Client {
    //工资低于2500元的上班族并且站立的乘客车票打8折
    public static <T extends Staff & Passenger> void discount(T t){
        if(t.getSalary()<2500 && t.isStanding()){
            System.out.println("恭喜你！您的车票打八折！");
        }
    }
    public static void main(String[] args) {
        discount(new Me());
    }
}
```

泛型数组

> 具体可以参考下文中关于泛型数组的理解。

首先，我们泛型数组相关的申明：

```
List<String>[] list11 = new ArrayList<String>[10]; //编译错误，非法创建
List<String>[] list12 = new ArrayList<?>[10]; //编译错误，需要强转类型
List<String>[] list13 = (List<String>[]) new ArrayList<?>[10]; //OK，但是会有警告
List<?>[] list14 = new ArrayList<String>[10]; //编译错误，非法创建
List<?>[] list15 = new ArrayList<?>[10]; //OK
List<String>[] list6 = new ArrayList[10]; //OK，但是会有警告
```

那么通常我们如何用呢？

- 讨巧的使用场景

```
public class GenericsDemo30{
    public static void main(String args[]){
        Integer i[] = fun1(1,2,3,4,5,6) ;    // 返回泛型数组
        fun2(i) ;
    }
    public static <T> T[] fun1(T...arg){  // 接收可变参数
```

```
        return arg ;            // 返回泛型数组
    }
    public static <T> void fun2(T param[]){   // 输出
        System.out.print("接收泛型数组：") ;
        for(T t:param){
            System.out.print(t + "、") ;
        }
    }
}
```

- 合理使用

```
public ArrayWithTypeToken(Class<T> type, int size) {
    array = (T[]) Array.newInstance(type, size);
}
```

具体可以查看后文解释。

深入理解泛型

TIP

我们通过泛型背后的类型擦除以及相关的问题来进一步理解泛型。@pdai

如何理解*Java*中的泛型是伪泛型？泛型中类型擦除

> Java泛型这个特性是从JDK 1.5才开始加入的，因此为了兼容之前的版本，Java泛型的实现采取了“**伪泛型**”的策略，即Java在语法上支持泛型，但是在编译阶段会进行所谓的“**类型擦除**”（Type Erasure），将所有的泛型表示（尖括号中的内容）都替换为具体的类型（其对应的原生态类型），就像完全没有泛型一样。理解类型擦除对于用好泛型是很有帮助的，尤其是一些看起来“疑难杂症”的问题，弄明白了类型擦除也就迎刃而解了。

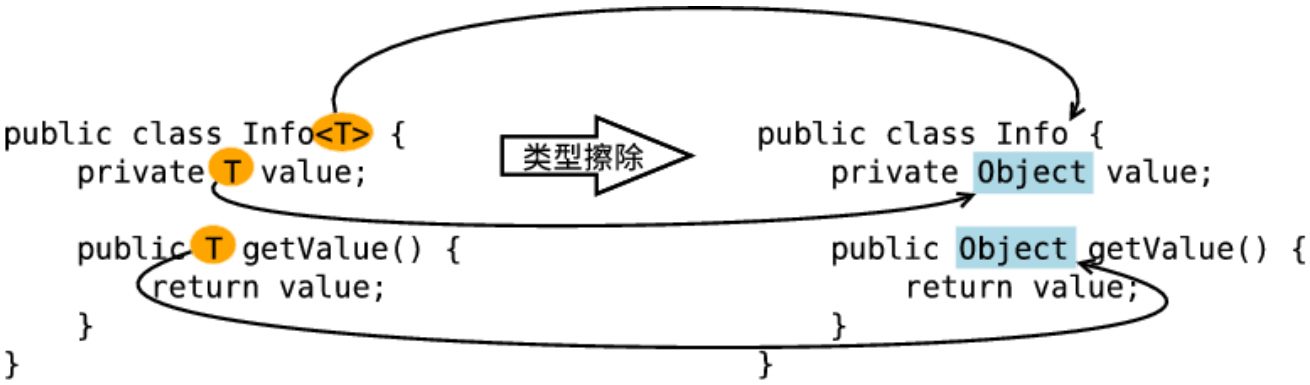
泛型的类型擦除原则是：

- 消除类型参数声明，即删除<>及其包围的部分。
- 根据类型参数的上下界推断并替换所有的类型参数为原生态类型：如果类型参数是无限制通配符或没有上下界限定则替换为Object，如果存在上下界限定则根据子类替换原则取类型参数的最左边限定类型（即父类）。
- 为了保证类型安全，必要时插入强制类型转换代码。
- 自动产生“桥接方法”以保证擦除类型后的代码仍然具有泛型的“多态性”。

那么如何进行擦除的呢?

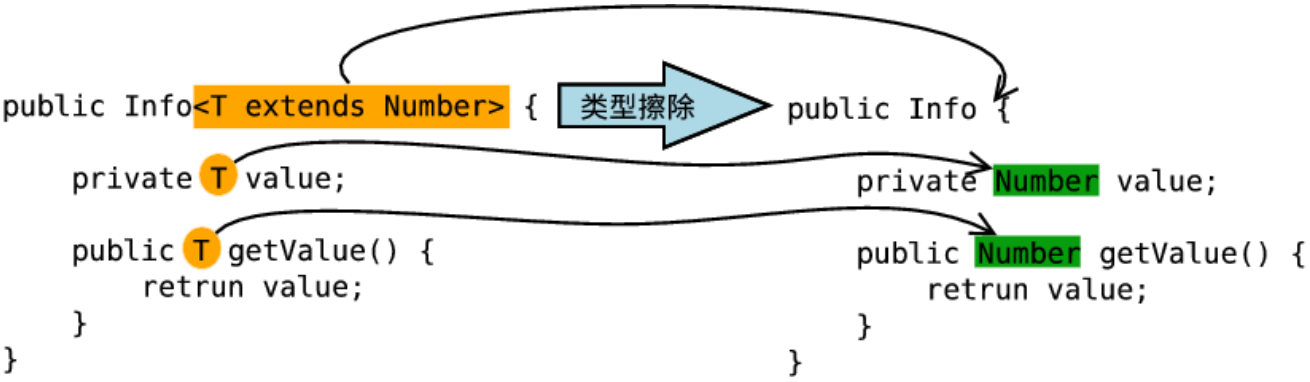
- 擦除类定义中的类型参数 - 无限制类型擦除

当类定义中的类型参数没有任何限制时，在类型擦除中直接被替换为Object，即形如<T>和<?>的类型参数都被替换为Object。



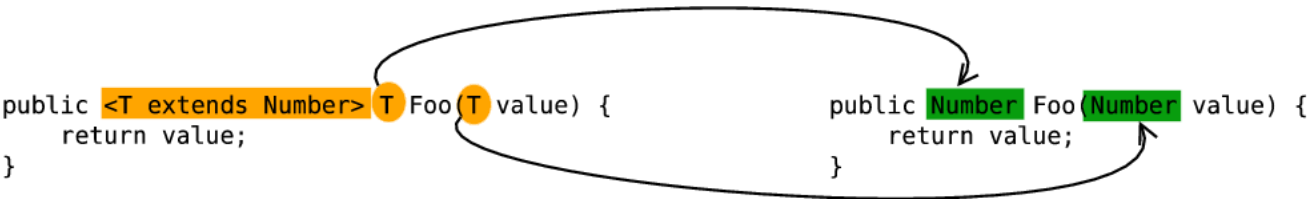
- 擦除类定义中的类型参数 - 有限制类型擦除

当类定义中的类型参数存在限制（上下界）时，在类型擦除中替换为类型参数的上界或者下界，比如形如`<T extends Number>`和`<? extends Number>`的类型参数被替换为`Number`，`<? super Number>`被替换为`Object`。



- 擦除方法定义中的类型参数

擦除方法定义中的类型参数原则和擦除类定义中的类型参数是一样的，这里仅以擦除方法定义中的有限制类型参数为例。



如何证明类型的擦除呢?

> 通过两个例子证明Java类型的类型擦除

- 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()); // true
    }
}
```

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。

- 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。

如何理解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。

- 原始类型Object

```
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:

```
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。

ArrayList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，为Number
        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小级，为Object

        /**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。

- Object泛型

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

如何理解泛型的编译期检查?

> 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？

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。

如何理解泛型的多态？泛型的桥接方法

> 类型擦除会造成多态的冲突，而JVM解决方法就是桥接方法。

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巧方法
    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巧方法
    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setValue方法如下：

```
public ObjectgetValue() {
    return super.getValue();
}
```

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。

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。

如何理解基本类型不能作为泛型类型?

> 比如，我们没有`ArrayList<int>`，只有`ArrayList<Integer>`, 为何?

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int值，只能引用Integer的值。

另外需要注意，我们能够使用`list.add(1)`是因为Java基础类型的自动装箱拆箱操作。

如何理解泛型类型不能实例化?

> 不能实例化泛型类型, 这本质上是由于类型擦除决定的：

我们可以看到如下代码会在编译器中报错：

```
T test = new T(); // ERROR
```

因为在 Java 编译期没法确定泛型参数化类型，也就找不到对应的类字节码文件，所以自然就不行了，此外由于T 被擦除为 Object，如果可以 new T() 则就变成了 new Object()，失去了本意。 如果我们确实需要实例化一个泛型，应该如何做呢？可以通过反射实现：

```
static <T> T newTclass (Class < T > clazz) throws InstantiationException, IllegalAccessException
{
    T obj = clazz.newInstance();
    return obj;
}
```

泛型数组：能不能采用具体的泛型类型进行初始化?

我们先来看下Oracle官网提供的一个例子：

```
List<String>[] lsa = new List<String>[10]; // Not really allowed.
Object o = lsa;
Object[] oa = (Object[]) o;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li; // Unsound, but passes run time store check
String s = lsa[1].get(0); // Run-time error ClassCastException.
```

由于 JVM 泛型的擦除机制，所以上面代码可以给 oa[1] 赋值为 ArrayList 也不会出现异常，但是在取出数据的时候却要做一次类型转换，所以就会出现 ClassCastException，如果可以进行泛型数组的声明则上面说的这种情况在编译期不会出现任何警告和错误，只有在运行时才会出错，但是泛型的出现就是为了消灭 ClassCastException，所以如果 Java 支持泛型数组初始化操作就是搬起石头砸自己的脚。

而对于下面的代码来说是成立的：

```
List<?>[] lsa = new List<?>[10]; // OK, array of unbounded wildcard type.
Object o = lsa;
Object[] oa = (Object[]) o;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li; // Correct.
Integer i = (Integer) lsa[1].get(0); // OK
```

所以说采用通配符的方式初始化泛型数组是允许的，因为对于通配符的方式最后取出数据是要做显式类型转换的，符合预期逻辑。综述就是说Java 的泛型数组初始化时数组类型不能是具体的泛型类型，只能是通配符的形式，因为具体类型会导致可存入任意类型对象，在取出时会发生类型转换异常，会与泛型的设计思想冲突，而通配符形式本来就需要自己强转，符合预期。

更进一步的，我们看看如下的代码：

```
List<String>[] list11 = new ArrayList<String>[10]; //编译错误，非法创建
List<String>[] list12 = new ArrayList<?>[10]; //编译错误，需要强转类型
List<String>[] list13 = (List<String>[]) new ArrayList<?>[10]; //OK，但是会有警告
List<?>[] list14 = new ArrayList<String>[10]; //编译错误，非法创建
List<?>[] list15 = new ArrayList<?>[10]; //OK
List<String>[] list6 = new ArrayList[10]; //OK，但是会有警告
```

因为在 Java 中是不能创建一个确切的泛型类型的数组的，除非是采用通配符的方式且要做显式类型转换才可以。

泛型数组：如何正确的初始化泛型数组实例？

> 这个无论我们通过new ArrayList[10]的形式还是通过泛型通配符的形式初始化泛型数组实例都是存在警告的，也就是说仅仅语法合格，运行时潜在的风险需要我们自己来承担，因此那些方式初始化泛型数组都不是最优雅的方式。

我们在使用到泛型数组的场景下应该尽量使用列表集合替换，此外也可以通过使用java.lang.reflect.Array.newInstance(Class<T> componentType, int length)方法来创建一个具有指定类型和维度的数组，如下：

```
public class ArrayWithTypeToken<T> {
    private T[] array;

    public ArrayWithTypeToken(Class<T> type, int size) {
        array = (T[]) Array.newInstance(type, size);
    }

    public void put(int index, T item) {
        array[index] = item;
    }

    public T get(int index) {
        return array[index];
    }

    public T[] create() {
        return array;
    }
}
//...

ArrayWithTypeToken<Integer> arrayToken = new ArrayWithTypeToken<Integer>(Integer.class, 100);
Integer[] array = arrayToken.create();
```

所以使用反射来初始化泛型数组算是优雅实现，因为泛型类型T在运行时才能被确定下来，我们能创建泛型数组也必然是在Java运行时想办法，而运行时能起作用的技术最好的就是反射了。

如何理解泛型类中的静态方法和静态变量？

> 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。

如何理解异常中使用泛型?

- **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：

```
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：

```
try{

} catch(Problem<Integer> e1) {

} catch(Problem<Number> e2) {

}
```

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

```
try{

} catch(Problem<Object> e1) {

} catch(Problem<Object> e2) {

}
```

这个当然就是不行的。

- **不能再catch子句中使用泛型变量**

```
public static <T extends Throwable> void doWork(Class<T> t) {
    try {
        ...
    } catch(T e) { //编译错误
        ...
    }
}
```

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：

```
public static <T extends Throwable> void doWork(Class<T> t){
    try {

    } catch(T e) { //编译错误

    } catch(IndexOutOfBounds e) {

    }
}
```

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。

```
public static<T extends Throwable> void doWork(T t) throws T {
    try{
        ...
    } catch(Throwable realCause) {
        t.initCause(realCause);
        throw t;
    }
}
```

上面的这样使用是没问题的。

如何获取泛型的参数类型？

> 既然类型被擦除了，那么如何获取泛型的参数类型呢？可以通过反射（java.lang.reflect.Type）获取泛型

java.lang.reflect.Type是Java中所有类型的公共高级接口, 代表了Java中的所有类型. Type体系中类型的包括：数组类型(GenericArrayType)、参数化类型(ParameterizedType)、类型变量(TypeVariable)、通配符类型(WildcardType)、原始类型(Class)、基本类型(Class), 以上这些类型都实现Type接口。

```
public class GenericType<T> {
    private T data;

    public T getData() {
        return data;
    }

    public void setData(T data) {
        this.data = data;
    }

    public static void main(String[] args) {
        GenericType<String> genericType = new GenericType<String>() {};
        Type superclass = genericType.getClass().getGenericSuperclass();
        //getActualTypeArguments 返回确切的泛型参数，如Map<String, Integer>返回[String, Integer]
        Type type = ((ParameterizedType) superclass).getActualTypeArguments()[0];
        System.out.println(type);//class java.lang.String
```

```
    }
}
```

其中 ParameterizedType:

```
public interface ParameterizedType extends Type {
    // 返回确切的泛型参数, 如Map<String, Integer>返回[String, Integer]
    Type[] getActualTypeArguments();

    //返回当前class或interface声明的类型, 如List<?>返回List
    Type getRawType();

    //返回所属类型. 如,当前类型为O<T>.I<S>, 则返回O<T>. 顶级类型将返回null
    Type getOwnerType();
}
```