java 泛型详解-绝对是对泛型方法讲解最详细的，没有之一

> 对java的泛型特性的了解仅限于表面的浅浅一层，直到在学习设计模式时发现有不了解的用法，才想起详细的记录一下。
> 本文参考java 泛型详解、Java中的泛型方法、 java泛型详解

1. 概述

泛型在java中有很重要的地位，在面向对象编程及各种设计模式中有非常广泛的应用。

什么是泛型？为什么要使用泛型？

> **泛型，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？顾名思义，就是将类型由原来的具体的**
> **类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。**
> **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
> 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。

2. 一个栗子

一个被举了无数次的例子：

```
1  List arrayList = new ArrayList();
2  arrayList.add("aaaa");
3  arrayList.add(100);
4
5  for(int i = 0; i< arrayList.size();i++){
6      String item = (String)arrayList.get(i);
7      Log.d("泛型测试","item = " + item);
8  }
```

毫无疑问，程序的运行结果会以崩溃结束：

```
1  java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
```

ArrayList可以存放任意类型，例子中添加了一个String类型，添加了一个Integer类型，再使用时都以String的方式使用，因此程序崩溃了。为了解决类似
编译阶段就可以解决），泛型应运而生。

我们将第一行声明初始化list的代码更改一下，编译器会在编译阶段就能够帮我们发现类似这样的问题。

```
1  List<String> arrayList = new ArrayList<String>();
2  ...
3  //arrayList.add(100); 在编译阶段，编译器就会报错
```

3. 特性

泛型只在编译阶段有效。看下面的代码：

```
1  List<String> stringArrayList = new ArrayList<String>();
2  List<Integer> integerArrayList = new ArrayList<Integer>();
3
4  Class classStringArrayList = stringArrayList.getClass();
5  Class classIntegerArrayList = integerArrayList.getClass();
6
7  if(classStringArrayList.equals(classIntegerArrayList)){
8      Log.d("泛型测试","类型相同");
9  }
```

输出结果： `D/泛型测试: 类型相同` 。

通过上面的例子可以证明，在编译之后程序会采取去泛型化的措施。也就是说Java中的泛型，只在编译阶段有效。在编译过程中，正确检验泛型结果后，
信息擦出，并且在对象进入和离开方法的边界处添加类型检查和类型转换的方法。也就是说，泛型信息不会进入到运行时阶段。

对此总结成一句话：泛型类型在逻辑上看以看成是多个不同的类型，实际上都是相同的基本类型。

4. 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

4.3 泛型类

泛型类型用于类的定义中，被称为泛型类。通过泛型可以完成对一组类的操作对外开放相同的接口。最典型的就是各种容器类，如：List、Set、Map。

泛型类的最基本写法（这么看可能会有点晕，会在下面的例子中详解）：

```
class 类名称 <泛型标识：可以随便写任意标识号，标识指定的泛型的类型>{
  private 泛型标识 /*（成员变量类型）*/ var;
  .....

  }
}
```

一个最普通的泛型类：

```
//此处T可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型
//在实例化泛型类时，必须指定T的具体类型
public class Generic<T>{
    //key这个成员变量的类型为T,T的类型由外部指定
    private T key;

    public Generic(T key) { //泛型构造方法形参key的类型也为T，T的类型由外部指定
        this.key = key;
    }

    public T getKey(){ //泛型方法getKey的返回值类型为T，T的类型由外部指定
        return key;
    }
}
```

```
//泛型的类型参数只能是类类型（包括自定义类），不能是简单类型
//传入的实参类型需与泛型的类型参数类型相同，即为Integer.
Generic<Integer> genericInteger = new Generic<Integer>(123456);

//传入的实参类型需与泛型的类型参数类型相同，即为String.
Generic<String> genericString = new Generic<String>("key_vlaue");
Log.d("泛型测试","key is " + genericInteger.getKey());
Log.d("泛型测试","key is " + genericString.getKey());
```

```
12-27 09:20:04.432 13063-13063/? D/泛型测试: key is 123456
12-27 09:20:04.432 13063-13063/? D/泛型测试: key is key_vlaue
```

定义的泛型类，就一定要传入泛型类型实参么？并不是这样，在使用泛型的时候如果传入泛型实参，则会根据传入的泛型实参做相应的限制，此时泛型才
限制作用。如果不传入泛型类型实参的话，在泛型类中使用泛型的方法或成员变量定义的类型可以为任何的类型。

看一个例子：

```
Generic generic = new Generic("111111");
Generic generic1 = new Generic(4444);
Generic generic2 = new Generic(55.55);
Generic generic3 = new Generic(false);

Log.d("泛型测试","key is " + generic.getKey());
Log.d("泛型测试","key is " + generic1.getKey());
Log.d("泛型测试","key is " + generic2.getKey());
Log.d("泛型测试","key is " + generic3.getKey());
```

```
D/泛型测试: key is 111111
D/泛型测试: key is 4444
D/泛型测试: key is 55.55
D/泛型测试: key is false
```

注意：

- 1. 泛型的类型参数只能是类类型，不能是简单类型。
- 1. 不能对确切的泛型类型使用instanceof操作。如下面的操作是非法的，编译时会出错。

```
if(ex_num instanceof Generic<Number>){
}
```

4.4 泛型接口

泛型接口与泛型类的定义及使用基本相同。泛型接口常被用在各种类的生产器中，可以看一个例子：

```
//定义一个泛型接口
public interface Generator<T> {
    public T next();
}
```

当实现泛型接口的类，未传入泛型实参时：

```
/**
 * 未传入泛型实参时，与泛型类的定义相同，在声明类的时候，需将泛型的声明也一起加到类中
 * 即：class FruitGenerator<T> implements Generator<T>{
 * 如果不声明泛型，如：class FruitGenerator implements Generator<T>，编译器会报错："Unknown class"
 */
class FruitGenerator<T> implements Generator<T>{
    @Override
    public T next() {
        return null;
    }
}
```

当实现泛型接口的类，传入泛型实参时：

```
/**
 * 传入泛型实参时：
 * 定义一个生产器实现这个接口,虽然我们只创建了一个泛型接口Generator<T>
 * 但是我们可以为T传入无数个实参，形成无数种类型的Generator接口。
 * 在实现类实现泛型接口时，如已将泛型类型传入实参类型，则所有使用泛型的地方都要替换成传入的实参类型
 * 即：Generator<T>，public T next();中的的T都要替换成传入的String类型。
 */
public class FruitGenerator implements Generator<String> {

    private String[] fruits = new String[]{"Apple", "Banana", "Pear"};

    @Override
    public String next() {
        Random rand = new Random();
        return fruits[rand.nextInt(3)];
    }
}
```

4.5 泛型通配符

我们知道 `Ingeter` 是 `Number` 的一个子类，同时在特性章节中我们也验证过 `Generic<Ingeter>` 与 `Generic<Number>` 实际上是相同的一种基本类型。那么用 `Generic<Number>` 作为形参的方法中，能否使用 `Generic<Ingeter>` 的实例传入呢？在逻辑上类似于 `Generic<Number>` 和 `Generic<Ingeter>` 是否可以系的泛型类型呢?

为了弄清楚这个问题，我们使用 `Generic<T>` 这个泛型类继续看下面的例子：

```
public void showKeyValue1(Generic<Number> obj){
    Log.d("泛型测试","key value is " + obj.getKey());
}
```

```
Generic<Integer> gInteger = new Generic<Integer>(123);
Generic<Number> gNumber = new Generic<Number>(456);

```

```
showKeyValue(gNumber);

// showKeyValue这个方法编译器会为我们报错：Generic<java.lang.Integer>
// cannot be applied to Generic<java.lang.Number>
// showKeyValue(gInteger);
```

通过提示信息我们可以看到 `Generic<Integer>` 不能被看作为 `Generic<Number>` 的子类。由此可以看出:**同一种泛型可以对应多个版本（因为参数类型是不同版本的泛型类实例是不兼容的。**

回到上面的例子，如何解决上面的问题？总不能为了定义一个新的方法来处理 `Generic<Integer>` 类型的类，这显然与java中的多台理念相违背。因此我们上可以表示**同时**是 `Generic<Integer>` 和 `Generic<Number>` 父类的引用类型。由此类型通配符应运而生。

我们可以将上面的方法改一下：

```
public void showKeyValue1(Generic<?> obj){
    Log.d("泛型测试","key value is " + obj.getKey());
}
```

类型通配符一般是使用？代替具体的类型实参，注意了，**此处‘？’是类型实参，而不是类型形参** 。重要说三遍！**此处‘？’是类型实参，而不是类型形处‘？’是类型实参，而不是类型形参**！再直白点的意思就是，此处的？和Number、String、Integer一样都是一种实际的类型，可以把？看成所有类型真实的类型。

可以解决当具体类型不确定的时候，这个通配符就是 **？** ；当操作类型时，不需要使用类型的具体功能时，只使用Object类中的功能。那么可以用？通配符

4.6 泛型方法

在java中,泛型类的定义非常简单，但是泛型方法就比较复杂了。

> 尤其是我们见到的大多数泛型类中的成员方法也都使用了泛型，有的甚至泛型类中也包含着泛型方法，这样在初学者中非常容易将泛型方法理解错了。

泛型类，是在实例化类的时候指明泛型的具体类型；泛型方法，是在调用方法的时候指明泛型的具体类型 。

```
/**
 * 泛型方法的基本介绍
 * @param tClass 传入的泛型实参
 * @return T 返回值为T类型
 * 说明：
 *    1）public 与 返回值中间<T>非常重要，可以理解为声明此方法为泛型方法。
 *    2）只有声明了<T>的方法才是泛型方法，泛型类中的使用了泛型的成员方法并不是泛型方法。
 *    3）<T>表明该方法将使用泛型类型T，此时才可以在方法中使用泛型类型T。
 *    4）与泛型类的定义一样，此处T可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型。
 */
public <T> T genericMethod(Class<T> tClass)throws InstantiationException ,
  IllegalAccessException{
        T instance = tClass.newInstance();
        return instance;
}
```

```
Object obj = genericMethod(Class.forName("com.test.test"));
```

4.6.1 泛型方法的基本用法

光看上面的例子有的同学可能依然会非常迷糊，我们再通过一个例子，把我泛型方法再总结一下。

```
public class GenericTest {
    //这个类是个泛型类，在上面已经介绍过
    public class Generic<T>{
        private T key;

        public Generic(T key) {
            this.key = key;
        }

        //我想说的其实是这个，虽然在方法中使用了泛型，但是这并不是一个泛型方法。
        //这只是类中一个普通的成员方法，只不过他的返回值是在声明泛型类已经声明过的泛型。
        //所以在这个方法中才可以继续使用 T 这个泛型。
```

```
13          public T getKey(){
14              return key;
15          }
16
17          /**
18       * 这个方法显然是有问题的，在编译器会给我们提示这样的错误信息"cannot reslove symbol E"
19       * 因为在类的声明中并未声明泛型E，所以在使用E做形参和返回值类型时，编译器会无法识别。
20       public E setKey(E key){
21          this.key = keu
22       }
23       */
24       }
25
26       /**
27     * 这才是一个真正的泛型方法。
28     * 首先在public与返回值之间的<T>必不可少，这表明这是一个泛型方法，并且声明了一个泛型T
29     * 这个T可以出现在这个泛型方法的任意位置.
30     * 泛型的数量也可以为任意多个
31     *   如：public <T,K> K showKeyName(Generic<T> container){
32     *       ...
33     *     }
34     */
35       public <T> T showKeyName(Generic<T> container){
36           System.out.println("container key :" + container.getKey());
37           //当然这个例子举的不太合适，只是为了说明泛型方法的特性。
38           T test = container.getKey();
39           return test;
40       }
41
42       //这也不是一个泛型方法，这就是一个普通的方法，只是使用了Generic<Number>这个泛型类做形参而已。
43       public void showKeyValue1(Generic<Number> obj){
44           Log.d("泛型测试","key value is " + obj.getKey());
45       }
46
47       //这也不是一个泛型方法，这也是一个普通的方法，只不过使用了泛型通配符?
48       //同时这也印证了泛型通配符章节所描述的，?是一种类型实参，可以看做为Number等所有类的父类
49       public void showKeyValue2(Generic<?> obj){
50           Log.d("泛型测试","key value is " + obj.getKey());
51       }
52
53        /**
54     * 这个方法是有问题的，编译器会为我们提示错误信息："UnKnown class 'E' "
55     * 虽然我们声明了<T>,也表明了这是一个可以处理泛型的类型的泛型方法。
56     * 但是只声明了泛型类型T，并未声明泛型类型E，因此编译器并不知道该如何处理E这个类型。
57     public <T> T showKeyName(Generic<E> container){
58        ...
59     }
60     */
61
62       /**
63     * 这个方法也是有问题的，编译器会为我们提示错误信息："UnKnown class 'T' "
64     * 对于编译器来说T这个类型并未项目中声明过，因此编译也不知道该如何编译这个类。
65     * 所以这也不是一个正确的泛型方法声明。
66     public void showkey(T genericObj){
67
68     }
69     */
70
71       public static void main(String[] args) {
72
73
74       }
75   }
```

4.6.2 类中的泛型方法

当然这并不是泛型方法的全部，泛型方法可以出现杂任何地方和任何场景中使用。但是有一种情况是非常特殊的，当泛型方法出现在泛型类中时，我们再i
下

```java
public class GenericFruit {
    class Fruit{
        @Override
        public String toString() {
            return "fruit";
        }
    }

    class Apple extends Fruit{
        @Override
        public String toString() {
            return "apple";
        }
    }

    class Person{
        @Override
        public String toString() {
            return "Person";
        }
    }

    class GenerateTest<T>{
        public void show_1(T t){
            System.out.println(t.toString());
        }

        //在泛型类中声明了一个泛型方法，使用泛型E，这种泛型E可以为任意类型。可以类型与T相同，也可以不同。
        //由于泛型方法在声明的时候会声明泛型<E>，因此即使在泛型类中并未声明泛型，编译器也能够正确识别泛型方法中识别的泛型。
        public <E> void show_3(E t){
            System.out.println(t.toString());
        }

        //在泛型类中声明了一个泛型方法，使用泛型T，注意这个T是一种全新的类型，可以与泛型类中声明的T不是同一种类型。
        public <T> void show_2(T t){
            System.out.println(t.toString());
        }
    }

    public static void main(String[] args) {
        Apple apple = new Apple();
        Person person = new Person();

        GenerateTest<Fruit> generateTest = new GenerateTest<Fruit>();
        //apple是Fruit的子类，所以这里可以
        generateTest.show_1(apple);
        //编译器会报错，因为泛型类型实参指定的是Fruit，而传入的实参类是Person
        //generateTest.show_1(person);

        //使用这两个方法都可以成功
        generateTest.show_2(apple);
        generateTest.show_2(person);

        //使用这两个方法也都可以成功
        generateTest.show_3(apple);
        generateTest.show_3(person);
    }
}
```

4.6.3 泛型方法与可变参数

再看一个泛型方法和可变参数的例子：

```java
public <T> void printMsg( T... args){
    for(T t : args){
        Log.d("泛型测试","t is " + t);
    }
}
```

```
printMsg("111",222,"aaaa","2323.4",55.55);
```

4.6.4 静态方法与泛型

静态方法有一种情况需要注意一下，那就是在类中的静态方法使用泛型：**静态方法无法访问类上定义的泛型；如果静态方法操作的引用数据类型不确定的**
型定义在方法上。

即：**如果静态方法要使用泛型的话，必须将静态方法也定义成泛型方法** 。

```
public class StaticGenerator<T> {
    ....
    ....
    /**
   * 如果在类中定义使用泛型的静态方法，需要添加额外的泛型声明（将这个方法定义成泛型方法）
   * 即使静态方法要使用泛型类中已经声明过的泛型也不可以。
   * 如：public static void show(T t){..},此时编译器会提示错误信息：
       "StaticGenerator cannot be refrenced from static context"
   */
    public static <T> void show(T t){

    }
}
```

4.6.5 泛型方法总结

泛型方法能使方法独立于类而产生变化，以下是一个基本的指导原则：

> 无论何时，如果你能做到，你就该尽量使用泛型方法。也就是说，如果使用泛型方法将整个类泛型化，那么就应该使用泛型方法。另外对于一个static的方法而已，无法访问
> 数。所以如果static方法要使用泛型能力，就必须使其成为泛型方法。

4.6 泛型上下边界

在使用泛型的时候，我们还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。

- 为泛型添加上边界，即传入的类型实参必须是指定类型的子类型

```
public void showKeyValue1(Generic<? extends Number> obj){
    Log.d("泛型测试","key value is " + obj.getKey());
}
```

```
Generic<String> generic1 = new Generic<String>("11111");
Generic<Integer> generic2 = new Generic<Integer>(2222);
Generic<Float> generic3 = new Generic<Float>(2.4f);
Generic<Double> generic4 = new Generic<Double>(2.56);

//这一行代码编译器会提示错误，因为String类型并不是Number类型的子类
//showKeyValue1(generic1);

showKeyValue1(generic2);
showKeyValue1(generic3);
showKeyValue1(generic4);
```

如果我们把泛型类的定义也改一下:

```
public class Generic<T extends Number>{
    private T key;

    public Generic(T key) {
        this.key = key;
    }

    public T getKey(){
        return key;
```

```
    }
}
```

```
//这一行代码也会报错，因为String不是Number的子类
Generic<String> generic1 = new Generic<String>("11111");
```

再来一个泛型方法的例子：

```
//在泛型方法中添加上下边界限制的时候，必须在权限声明与返回值之间的<T>上添加上下边界，即在泛型声明的时候添加
//public <T> T showKeyName(Generic<T extends Number> container)，编译器会报错："Unexpected bound"
public <T extends Number> T showKeyName(Generic<T> container){
    System.out.println("container key :" + container.getKey());
    T test = container.getKey();
    return test;
}
```

通过上面的两个例子可以看出：**泛型的上下边界添加，必须与泛型的声明在一起** 。

4.7 关于泛型数组要提一下

看到了很多文章中都会提起泛型数组，经过查看sun的说明文档，在java中是”**不能创建一个确切的泛型类型的数组**”的。

也就是说下面的这个例子是不可以的：

```
List<String>[] ls = new ArrayList<String>[10];
```

而使用通配符创建泛型数组是可以的，如下面这个例子：

```
List<?>[] ls = new ArrayList<?>[10];
```

这样也是可以的：

```
List<String>[] ls = new ArrayList[10];
```

下面使用Sun的一篇文档的一个例子来说明这个问题：

```
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.
```

> 这种情况下，由于JVM泛型的擦除机制，在运行时JVM是不知道泛型信息的，所以可以给oa[1]赋上一个ArrayList而不会出现异常，但是在取出数据的时候却要做一次类型转
> 现ClassCastException，如果可以进行泛型数组的声明，上面说的这种情况在编译期将不会出现任何的警告和错误，只有在运行时才会出错。
> 而对泛型数组的声明进行限制，对于这样的情况，可以在编译期提示代码有类型安全问题，比没有任何提示要强很多。

下面采用通配符的方式是被允许的:**数组的类型不可以是类型变量，除非是采用通配符的方式**，因为对于通配符的方式，最后取出数据是要做显式的类型转

```
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
```

5. 最后

本文中的例子主要是为了阐述泛型中的一些思想而简单举出的，并不一定有着实际的可用性。另外，一提到泛型，相信大家用到最多的就是在集合中，其
过程中，自己可以使用泛型去简化开发，且能很好的保证代码质量。