链滴

Java 中的泛型（Generics）

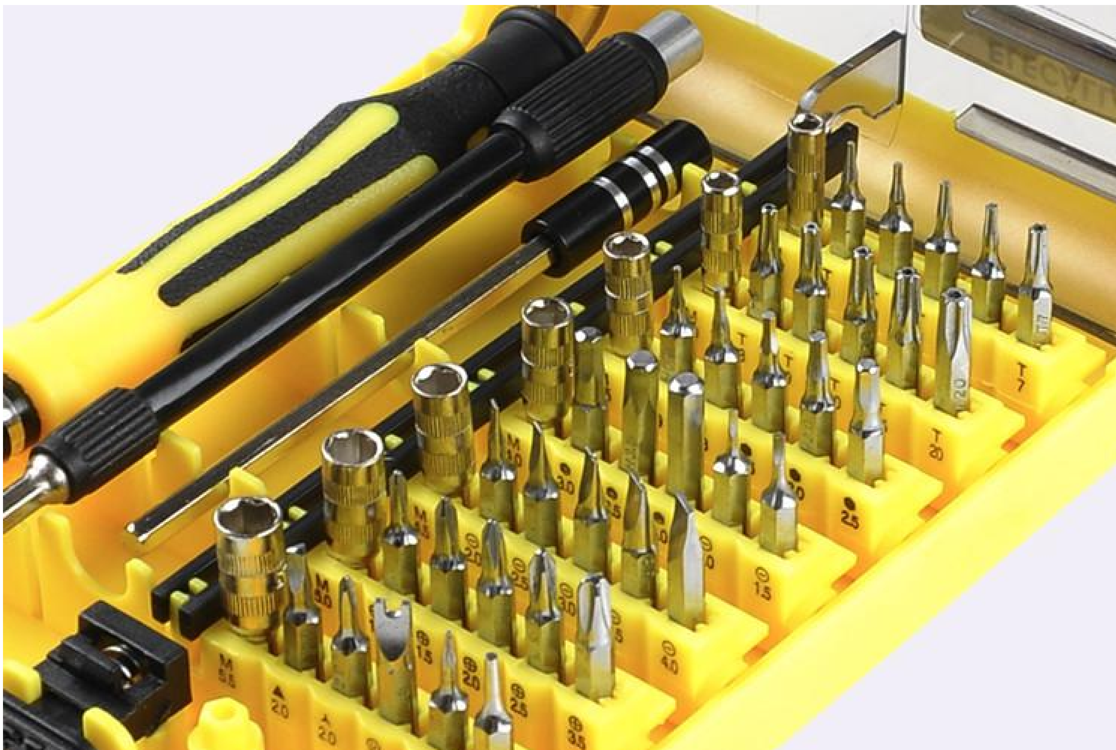
作者：lvtaos

原文链接：https://ld246.com/article/1563691619949

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

Java 中的泛型（Generics）



一、什么是泛型

泛型就是在定义类、接口、方法的时候将参数也转换为一种广泛的类型，即参数化类型。与方法中的式参数相比，泛型可以使方法传入多种数据类型，同一套代码可以适用于多种数据结构类型。

二、为什么要使用泛型

泛型有哪些好处呢?

- 更高的安全性
- 避免强制类型转换
- 开发人员可以实现通用的算法

举例来说

```
public static void main(String[] args) {
    List arrList = new ArrayList();
    arrList.add(1);
    arrList.add("str");
    for (int i = 0; i < arrList.size(); i++) {
        String str = (String) arrList.get(i);
        System.out.println("str=" + str);
    }
}
```

毫无疑问，程序会输出错误

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
```

如果我们使用泛型，就可以这样来写

```
public static void main(String[] args) {
    List<String> arrList = new ArrayList();
    arrList.add(1); // 程序编译时会报错
    arrList.add("str");
    for (int i = 0; i < arrList.size(); i++) {
        String str = arrList.get(i);
        System.out.println("str=" + str);
    }
}
```

再来看一个例子

```
public static void main(String[] args) {
    List<String> strList = new ArrayList<>();
    List<Integer> intList = new ArrayList<>();

    Class strClass = strList.getClass();
    Class intClass = intList.getClass();
    System.out.println(strClass == intClass);
    System.out.println(strClass.equals(intClass));
}
```

输出的结果是什么呢?

```
true
true
```

Java 有 Java 编译器和 Java 虚拟机。Java 编译器在编译泛型类的时候会将泛型擦除，替换为必要的制类型转换。在运行期的时候，Java 虚拟机是不知道有泛型这一回事的。

所以上述例子就相当于这样。

```
public static void main(String[] args) {
    List strList = new ArrayList<>();
    List intList = new ArrayList<>();

    Class strClass = strList.getClass();
    Class intClass = intList.getClass();
    System.out.println(strClass == intClass);
    System.out.println(strClass.equals(intClass));
}
```

三、泛型类

接下来我们看一个简单泛型类。

```
public class Bird<T> {

    T param1;
```

```
    T param2;

    public Bird(T param1, T param2) {
        this.param1 = param1;
        this.param2 = param2;
    }

    public T getParam1() {
        return this.param1;
    }

    public T getParam2() {
        return this.param2;
    }
}
```

我们可以这样使用。

```
public static void main(String[] args) {
    Bird<Integer> bird = new Bird<>(1, 2);
    Integer param1 = bird.getParam1();
    Integer param2 = bird.getParam2();
    System.out.println("param1=" + param1 + ",param2=" + param2);
}
```

泛型类可以当普通类来使用吗?

```
public static void main(String[] args) {
    Bird<Integer> bird1 = new Bird<>(1, 2);
    Bird bird2 = bird1;

    // Unchecked call to 'Bird(T, T)' as a member of raw type 'com.lvtao.javademo.generics.Bird'
    Bird bird3 = new Bird(1, "3");
    Bird<Integer> bird4 = bird3;
}
```

泛型也可以当做普通的类来使用，但是编译器会有警告。

泛型不可以使用基本类型，也不可以使用在 instanceof 后面。

四、泛型接口

泛型接口的使用与泛型类的使用类似，我们来看 JDK 中的一个泛型接口的使用。

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

String.java 中就有引用到泛型接口 Comparable

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {

    public int compareTo(String anotherString) {
```

```
        int len1 = value.length;
        int len2 = anotherString.value.length;
        int lim = Math.min(len1, len2);
        char v1[] = value;
        char v2[] = anotherString.value;

        int k = 0;
        while (k < lim) {
            char c1 = v1[k];
            char c2 = v2[k];
            if (c1 != c2) {
                return c1 - c2;
            }
            k++;
        }
        return len1 - len2;
    }

}
```

当泛型指定了某个具体类型的时候，实现类中的所有泛型都必须替换为具体指定的类型。

五、为什么用泛型通配符

介绍泛型方法前，先简单介绍一下通配符的使用。

先来看一个例子。在 Bird 类中加入一个静态方法 fly()

```
public static void fly(Bird<Number> bird) {
    System.out.println("param1=" + bird.getParam1() + ",param2=" + bird.getParam2());
}
```

然后这样调用

```
public static void main(String[] args) {
    Bird<Number> bird1 = new Bird<>(1, 2);
    Bird<Integer> bird2 = new Bird<>(3, 4);
    fly(bird1);
    fly(bird2); // 编译错误
}
```

fly(bird2); 的调用程序会报错，提示

```
fly(com.lvtao.javademo.generics.Bird<Java.lang.Number>) in Bird cannot be applied to (com.l
tao.javademo.generics.Bird<java.lang.Integer>)
```

当 fly() 方法确定泛型类必须是 Bird<Number> 的时候，虽然 Interger 是 Number 的子类，但是 Bid<Integer> 并不是 Bird<Number> 的子类，这时候不同版本的泛型不兼容的。为了解决上述问题我们可以使用通配符。

将 fly() 方法的参数类型改用通配符代替。

```
public static void fly(Bird<?> bird) {
    System.out.println("param1=" + bird.getParam1() + ",param2=" + bird.getParam2());
```

```
}
```

使用通配符修改完 fly() 方法并调用，输出

```
param1=1,param2=2
param1=3,param2=4
```

六、泛型方法

下面是泛型方法的定义。

```
public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}

public class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

调用的时候可以这样来写

```
public static void main(String[] args) {
    Pair<Integer, String> p1 = new Pair<>(1, "A");
    Pair<Integer, String> p2 = new Pair<>(2, "B");
    boolean same = Util.compare(p1, p2);
    // 这样也可以
    // boolean same = Util.<Integer, String>compare(p1, p2);
    System.out.println(same);
}
```

使用泛型方法的时候需要注意的问题：

1. 泛型类在实例化的时候必须制定泛型的具体类型，泛型方法却不需要
2. 定义泛型方法需要用 <T> 来声明，多个泛型类型需要用逗号隔开 <K, V>
3. 静态方法需要使用泛型的时候，必须要将这个静态方法也定义为泛型方法

七、泛型的边界

```
public class Box<T> {

    private T t;

    public void set(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    /**
     * 参数 U 必须是 Number 类或者是 Number 的子类
     */
    public <U extends Number> void inspectBound(U u) {
        // TODO
    }

    /**
     * Box 的具体类型必须是 Number 类或者是 Number 的子类
     */
    public void inspectLow(Box<? extends Number> box){
        // TODO
    }

    /**
     * 用法同上
     */
    public <U extends Number> void inspectLow1(Box<U> u) {
        // TODO
    }


    /**
     * Box 的具体类型必须是 Integer 类或者是 Integer 的父类
     * @param box
     */
    public void inspectUp(Box<? super Integer> box){
        // TODO
    }
}
```

上述例子中方法 inspectLow() 和方法 inspectLow1() 的作用是相同的，只是用通配符的方法使用起更加简洁。泛型方法的返回值使用到泛型时，就不能用通配符代替。

```
public <U extends Number> U ins(Box<U> u) {
    return null;
}

/**
 * 错误的写法，不能这样定义
 */
```

```
public U ins(Box<? extends Number> u) {
    return null;
}
```

八、参考资料

- Java Documentation The Java™ Tutorials

（完）