泛型相關

Java 泛型是一种支持参□化类型的机制，它可以□代□更加通用和安全。通□使用泛型，我□可以在□□代□□使用占位符类型（如 T、E 等），然后在实际使用□再指定具体类型。□样就可以使代□更加灵活，可以避免类型□□□□和运行□异常。

Java 泛型的基本用法是使用尖括号 "<>" 括起□的一个或多个类型参□，□些类型参□可以用于类、接口、方法的声明中。例如：

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

    public MyClass(T data) {
        this.data = data;
    }

    public T getData() {
        return data;
    }
}
```

在上面的代□中，我□使用了一个类型参□ T，□个参□可以用于 MyClass 类中的任何地方，包括类的字段、方法参□、方法返回值等等。

而<?>和<T>的□□在于，<?>是一种无限制通配符类型，表示可以匹配任何类型，而<T>□是一个类型参□，表示在使用□需要指定具体的类型。例如：

```
List<?> list = new ArrayList<>();
List<String> strList = new ArrayList<>();
list = strList; // 合法，因□ list 可以匹配任何类型
```

```
public <T> T getValue(T[] array, int index) {
    return array[index];
}

String[] strArray = {"hello", "world"};
String str = getValue(strArray, 0); // 合法，因□在使用 getValue □指定了类型参□□ String
```

下面是一个使用无限制通配符类型（<?>）的例子，我□定义了一个方法，可以接受任何类型的 List，并打印出其中的元素：

```
public static void printList(List<?> list) {
    for (Object element : list) {
        System.out.println(element);
    }
}
```

□个方法的参□列表中使用了无限制通配符类型（<?>），表示可以匹配任何类型的 List。在方法□部，我□使用了一个 for-each 循□遍□ list，并打印出其中的元素。

使用□个方法□，我□可以□入任何类型的 List，例如：

```
List<Integer> intList = Arrays.asList(1, 2, 3);
List<String> strList = Arrays.asList("hello", "world");
List<Object> objList = Arrays.asList(1, "hello", true);

printList(intList); // 输出 1, 2, 3
printList(strList); // 输出 hello, world
printList(objList); // 输出 1, hello, true
```

如果我□□上面的例子中的方法参□从无限制通配符类型（<?>）改□类型参□（<T>），那么代□□变成下面□样：

```
public static <T> void printList(List<T> list) {
    for (T element : list) {
        System.out.println(element);
    }
}
```

□个方法的参□列表中使用了类型参□（<T>），表示在使用□需要指定 List 中元素的具体类型。在方法□部，我□使用了一个 for-each 循

□遍□ list，并打印出其中的元素。

使用□个方法□，我□需要指定 List 中元素的具体类型，例如：

```
List<Integer> intList = Arrays.asList(1, 2, 3);
List<String> strList = Arrays.asList("hello", "world");
List<Object> objList = Arrays.asList(1, "hello", true);

printList(intList); // 输出 1, 2, 3
printList(strList); // 输出 hello, world
printList(objList); // □□□□，因□ objList 中的元素类型不是 Object
```

在使用□，我□需要指定 List 中元素的具体类型，例如使用 `printList(Integer)` 或 `printList(String)`，□样才能保□参□类型的匹配，否□□会□□□□。

在 Java 泛型中，类型参□的名□是可以任意指定的，通常使用以下的□定：

- T：表示任何类型。
- E：表示集合中的元素类型。
- K：表示映射中的□类型。
- V：表示映射中的值类型。
- N：表示□字类型。
- S、U、V 等：表示其他任意类型。

因此，使用 <T> 或 <E> 在□义上是等价的，只是名□不同而已。

除了以上提到的常用类型参□名□外，□发者也可以根据实际情况自行定义类型参□的名□，只需要遵循命名□范即可。不□，□了代□可□性和可□护性，建□使用常见的类型参□名□。

需要注意的是，不同的类型参□名□并□有实□上的差异，只是在□义上有所不同，因此□□类型参□名□□□根据实际情况和需求□行□□，□□一个能更好地表达自己的代□意□的名□。

一個回傳任意型別的範例

```
public static <T> T anyTypeMethod(T arg) {
    if (arg == null) {
        // 返回空□象
        return (T) new Object();
    }
    // 方法□□
    return arg;
}
```

在□个方法中，我□首先□查参□是否□ `null`。如果参□□ `null`，□返回一个空□象，否□□行方法的□□，□参□原封不动地返回。

需要注意的是，在□个方法中，我□□建了一个新的 `Object` □象并□其□□□泛型类型，□是□了避免空□象□返回类型不匹配的□□。因□ Java 中的泛型是在□□□□行类型擦除的，如果我□直接返回一个 `null`，□□器□无法确定返回值的类型，可能会□致□□□□或运行□异常。

使用□个方法□，我□可以□入任何类型的参□，并□取返回值，如果□入参□□ `null`，□返回一个空□象，例如：

```
Integer intValue = anyTypeMethod(123);
String strValue = anyTypeMethod("hello");
Object objValue = anyTypeMethod(new Object());
Object nullValue = anyTypeMethod(null);

System.out.println(intValue); // 输出 123
System.out.println(strValue); // 输出 "hello"
System.out.println(objValue); // 输出 Object □象的 toString() 值
System.out.println(nullValue); // 输出 "java.lang.Object@hashcode"
```

在□个示例中，我□分□□入了一个整□、一个字符串、一个□象和一个 `null`，然后□取了相□的返回值。□□入参□□ `null` □，返回一个空□象，其类型□ `Object`。

修訂版本 #3
★由 treeman 建立於 14 ◆T◆◆ 2023 09:54:49
✎由 treeman 更新於 5 ◆◆◆◆ 2023 10:18:39