链滴

泛型之解析通配符

作者：AutisticV5
原文链接：https://ld246.com/article/1573691496428
来源网站：链滴

通配符有着令人费解和混淆的语法，但通配符大量应用于Java容器类中。

1. 更简洁的参数类型限定

在上一篇中，为了将Integer对象添加到Number容器中，我们的类型参数使用了其他类型参数作为界，我们提到，这种写法有点繁琐，它可以替换为更为简洁的通配符形式：

```
public void addAll(DynamicArray<? extends E> c) {
    for (int i = 0; i< c.size; i++) {
        add(c.get(i));
    }
}
```

这个方法没有定义类型参数，c的类型是DynamicArray<? extends E>，？ 表示通配符，<? extends >表示有限定通配符，匹配E或E的某个子类型，具体什么子类型是未知的。

```
DynamicArray<Number> numbers = new DynamicArray<>();
DynamicArray<Integer> ints = new DynamicArray<>();
ints.add(100);
ints.add(34);
numbers.addAll(ints);
```

这里，E是Number类型，DynamicArray<? extends E>可以匹配DynamicArray<Integer>。

那么问题来了，同样是extends关键字，同样应用于泛型，<T extends E>和<? extends E>到底有么关系？它们用的地方不一样：

1. <T extends E>用于定义类型参数，它声明了一个类型参数T，可放在泛型类定义中类名后面、型方法返回值前面。

2.

<? extends E>用于实例化类型参数，它用于实例化泛型变量中的类型参数，只是这个具体类型是未的，只知道它是E或E的某个子类型。

虽然他们不一样，但两种写法经常可以达成相同目标。

```
public void addAll(DynamicArray<? extends E> c)
public <T extends E> void addAll(DynamicArray<T> c)
```

那么到底应该用哪种形式?

2. 理解通配符

除了有限定通配符，还有一种通配符，形如DynamicArray<?>，称为无限定通配符。

```
public static int indexOf(DynamicArray<?> arr, Object elm) {
    for (int i = 0; i< arr.size(); i++) {
        if (arr.get(i).equals(elm)) {
            return i;
        }
    }
    return -1;
}
```

其实这种无限通配符形式也可以改为使用类型参数。

```
public static int indexOf(DynamicArray<?> arr, Object elm)
```

可以改为:

```
public static <T> int indexOf(DynamicArray<T> arr, Object elm)
```

不过，通配符形式更为简洁。虽然通配符形式更为简洁，但上面两种通配符都有一个重要的限制：只读，不能写。

```
DynamicArray<Integer> ints = new DynamicArray<>();
DynamicArray<? extends Number> numbers = ints;
Integer a = 200;
numbers.add(a):    //错误
numbers.add((Number) a); //错误
numbers.add((Object) a); //错误
```

三种add方法都是非法的，无论是Integer，还是Number或Object，编译器都会报错。因为问号就表示类型安全无知，？ extends Number表示Number的某个子类型，但不知道具体子类型，如果允写入，Java就无法确保类型安全性，所以干脆禁止。

```
DynamicArray<Integer> ints = new DynamicArray<>();
DynamicArray<? extends Number> numbers = ints;
Number n = new Double(23.0)
Object o = new String("hello world");
numbers.add(n);
numbers.add(o);
```

如果允许写入Object或Number类型，则最后两行编译就是正确的，也就是说，Java将允许把Doubl或String对象放入Integer容器，这显然违背了Java关于类型安全的承诺。

大部分情况下，这种限制是好的，但这使得一些理应正确的基本操作无法完成，比如交换两个元素的置。

```
public static void swap(DynamicArray<?> arr, int i, int j) {
    Object tmp = arr.get(i);
    arr.set(i, arr.get(j));
    arr.set(j, tmp);
```

```
}
```

这个代码看上去应该是正确的，但Java会提示编译错误，两行set语句都是非法的。不过，借助待类参数的方法，这个问题可以如下解决。

```
private static <T> void swapInternal(DynamicArray<T> arr, int i, int j) {
    T tmp = arr.get(i);
    arr.set(i, arr.get(j));
    arr.set(j, tmp);
}
public static void swap(DynamicArray<?> arr, int i, int j) {
    swapInternal(arr, i, j);
}
```

swap可以调用swapInternal，而带类型参数的swapInternal可以写入。Java容器类中就有类似这样用法，公共的API是通配的形式，形式更简单，但内部调用带类型参数的方法。

除了这种需要写的场合，如果参数类型之间有依赖关系，也只能用类型参数，比如，将src容器中的内复制到dest中：

```
public static <D, S extends D> void copy(DynamicArray<D> dest, DynamicArray<S> src) {
    for (int i = 0; i< src.size(); i++) {
        dest.add(src.get(i));
    }
}
```

S和D有依赖关系，要么相同，要么S是D的子类，否则类型不兼容，有编译错误。不过上面的声明可使用通配符简化，两个参数可以简化为一个，如下：

```
public static <D> void copy(DynamicArray<D> dest, DynamicArray<? extends D> src) {
    for (int i = 0; i< src.size(); i++) {
        dest.add(src.get(i));
    }
}
```

如果返回值依赖于类型参数，也不能用通配符，比如，计算动态数组中的最大值。

```
public static <T extends Comparable<T>> T max(DynamicArray arr) {
    T max = arr.get(0);
    for(int i = 1; i< arr.size(); i++) {
        if(arr.get(i).compareTo(max) > 0) {
            max = arr.get(i);
        }
    }
    return max;
}
```

上面的代码就难以用通配符代替。

那么泛型方法到底应该用通配符的形式还是加类型参数？总结如下：

1. 通配符形式都可以用类型参数的形式来替代，通配符能做的，用类型参数都能做。

2. 通配符形式可以减少类型参数，形式上往往更为简单，可读性也更好，所以，能用通配符的就通配符。

3. 如果类型参数之间有依赖关系，或者返回值依赖类型参数，或者需要写操作，则只能用类型参
。

4. 通配符形式和类型参数往往配合使用。比如，上面copy方法，定义必要的类型参数，使用通配

表达依赖，并解说更广泛的数据类型。

3. 超类型通配符

还有一种通配符，与形式<? extends E>正好相反，它的形式为<? super E>，称为超类型通配符，示E的某个父类型。有了它，我们就可以更灵活地写入了。

如果没有这种语法，写入会有一些限制。

```
public void copyTo(DynamicArray<E> dest) {
    for(int i = 0; i< size; i++) {
        dest.add(get(i));
    }
}
```

这个方法也很简单，将当前容器中的元素添加到传入的目标容器中。我们可能希望这个使用：

```
DynamicArray<Integer> ints = new DynamicArray<Integer>();
ints.add(100);
ints.add(34);
DynamicArray<Number> numbers = new DynamicArray<>();
ints.copyTo(numbers);
```

Integer是Number的子类，将Integer对象拷贝进Number容器，这种用法应该是合情合理的，但Jav会提示编译错误，理由我们之前也说过，期望的参数类型是DynamicArray<Integer>，DynamicArray<Number>并不适用。

如之前所说，一般而言，不能将DynamicArray<Integer>看作DynamicArray<Number>，但我们里的用法没有问题，Java解决这个问题的方法就是超类型通配符，可以将copyTo代码该为：

```
public void copyTo(DynamicArray<? super E> dest) {
    for(int i = 0; i< size; i++) {
        dest.add(get(i));
    }
}
```

这样就没问题。

超类型通配符另一个常用的场合是Comparable/Comparator接口。同样，如果不适用会有什么限制。

```
public static <T extends Comparable<T>> T max(DynamicArray<T> arr)
```

这个声明有什么限制呢？举个简单的例子，有两个类Base和Child

```
class Base implements Comparable<Base> {
    private int sortOrder;
    public Base(int sortOrder) {
        this.sortOrder = sortOrder;
    }

    @Override
    public int compareTo(Base o) {
        if(sortOrder < o.sortOrder) {
            return -1;
        } else if(sortOrder > o.sortOrder) {
            return 1;
        } else {
            return 0;
        }
```

```
    }
}

class Child extends Base {
    public Child(int sortOrder) {
        super(sortOrder);
    }
}
```

这里，Child非常简单，只是继承了Base。注意：Child没有重新实现Comparable接口，因为Child比较规则和Base是一样的。我们可能希望使用前面的max方法操作Child容器。

```
DynamicArray<Child> childs = new DynamicArray<Child>();
childs.add(new Child(20));
childs.add(new Child(80));
Child maxChild = mac(childs);
```

遗憾的是，Java会提示错误，类型不匹配。我们可能会认为，Java会将max方法的类型参数T推断为Child类型，但类型T的要求是extends Comparable<T>，而Child并没有实现Comparable<Child>，实现的是Comparable<Base>。

但我们的需求是合理的，Base类的代码已经有了关于比较所需要的全部数据，它应该可以用于比较Child对象。解决这个问题的方法就是修改max方法声明，使用超类型通配符。

```
public static <T extends Comparable<? super T>> T max(DynamicArray<T> arr)
```

这么修改一下就可以了，这种写法比较抽象，将T替换为Child，就是：

```
Child extends Comparable<? super Child>
```

类型参数限定只有extends形式，没有super形式，比如前面的copyTo方法的通配符形式的声明：

```
public void copyTo(DynamicArray<? super E> dest)
```

如果类型参数限定支持super形式，则应该是：

```
public <T super E> void copyTo(DynamicArray<T> dest)
```

事实是，Java并不支持这种语法。

对于有限定的通配符形式<? extends E>，可以用类型参数限定替代，但是对于类似上面的超类型通符，则无法用类型参数替代。

泛型的三种通配符形式<?>、<? super E>和<? extends E>，并分析了与类型参数形式的区别和联，他们都容易混淆：

1. 他们的目的都是为了使方法接口更为灵活，可以接受更为广泛的类型。
2.

<? super E>用于灵活写入或比较，使得对象可以写入父类型的容器，使得父类型的比较方法可以应于子类对象，它不能被类型参数形式替代。

3.

<?>和<? extends E>用于灵活读取，使得方法可以读取E或E的任意子类型的容器对象，他们可以用型参数的形式替代，但通配符形式更为简洁。

在使用泛型类、方法和接口时，有一些值得注意的地方：

- 基本类型不能用于实例化类型参数
  - 运行时类型信息不适用于泛型
  - 类型擦除可能会引发一些冲突

在定义泛型类、方法和接口时，也有一些需要注意的地方：

- 不能通过类型参数创建对象
- 泛型类类型参数不能用于静态变量和方法
- 了解多个类型限定的语法

泛型与数组的关系：

- Java不支持创建泛型数组
- 如果要存放泛型对象，可以使用原始类型的数组，或者使用泛型容器
- 泛型容器内部使用Object数组，如果要转换泛型容器为对应类型的数组，需要使用反射。