泛型(了解)

本节目标

> - **以能阅读 java 集合源码为目标学习泛型**
> - 了解泛型的分类
> - 了解如何定义泛型类和泛型方法
> - 了解定义泛型类时的一些注意事项
> - 区分泛型的定义和泛型的使用

1. 泛型类的定义

1.1 语法

```
class 泛型类名称<类型形参列表> {
    // 这里可以使用类型参数
}

class ClassName<T1, T2, ..., Tn> {
}
```

```
class 泛型类名称<类型形参列表> extends 继承类/* 这里可以使用类型参数 */ {
    // 这里可以使用类型参数
}

class ClassName<T1, T2, ..., Tn> extends ParentClass<T1> {
    // 可以只使用部分类型参数
}
```

了解： 【规范】类型形参一般使用一个大写字母表示，常用的名称有：

- E 表示 Element
- K 表示 Key
- V 表示 Value
- N 表示 Number
- T 表示 Type
- S, U, V 等等 - 第二、第三、第四个类型

1.2 简单示例

定义一个泛型类顺序表

```
// 演示泛型目的，没有做错误处理和扩容
public class MyArrayList<E> {
    private E[] array;
    private int size;
```

```
    public MyArrayList() {
        // 泛型类型无法直接创建数组，具体的见下面的注意事项
        array = (E[])new Object[16];
        size = 0;
    }

    // 尾插
    public void add(E e) {
        array[size++] = e;
    }

    // 尾删
    public E remove() {
        E element = array[size - 1];
        array[size - 1] = null; // 将容器置空，保证对象被正确释放
        size--;
        return element;
    }
}
```

1.3 加入静态内部类的示例

定义一个泛型类链表

```
public class MyLinkedList<E> {
    public static class Node<E> {
        private E value;
        private Node<E> next;

        private Node(E e) {
            value = e;
            next = null;
        }
    }

    private Node<E> head;
    private int size;

    public MyLinkedList() {
        head = null;
        size = 0;
    }

    // 头插
    public void pushFront(E e) {
        Node<E> node = new Node<>(e);
        node.next = head;
        head = node;
        size++;
    }

    // 尾插
```

```
    public void pushBack(E e) {
        if (size == 0) {
            pushFront(e);
            return;
        }

        Node<E> cur = head;
        while (cur.next != null) {
            cur = cur.next;
        }

        cur.next = new Node<E>(e);
        size++;
    }
}
```

1.4 加入继承或实现的示例

定义一个泛型类顺序表

```
public interface MyList<E> {
    // 尾插
    void add(E e);
    // 尾删
    E remove();
}
```

```
public class MyArrayList<E> implements MyList<E> {
    // TODO: 未完成
}
```

2. 泛型类的使用

2.1 语法

```
泛型类<类型实参> 变量名;  // 定义一个泛型类引用
new 泛型类<类型实参>(构造方法实参);  // 实例化一个泛型类对象
```

2.2 示例

```
MyArrayList<String> list = new MyArrayList<String>();
```

2.3 类型推导(Type Inference)

当编译器可以根据上下文推导出类型实参时，可以省略类型实参的填写

```
MyArrayList<String> list = new MyArrayList<>(); // 可以推导出实例化需要的类型实参为 String
```

3. 裸类型(Raw Type) （了解）

3.1 说明

裸类型是一个泛型类但没有带着类型实参，例如 MyArrayList 就是一个裸类型

```
MyArrayList list = new MyArrayList();
```

注意： 我们不要自己去使用裸类型，裸类型是为了兼容老版本的 API 保留的机制

下面的类型擦除部分，我们也会讲到编译器是如何使用裸类型的。

3.2 未检查错误

```
MyArrayList<String> list = new MyArrayList();   // 自己永远不要这么用
```

上述代码，会产生编译警告

```
Note: Example.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

可以使用 `@SuppressWarnings` 注解进行警告压制

```
@SuppressWarnings("unchecked")
```

4. 泛型类的定义-类型边界

在定义泛型类时，有时需要对传入的类型变量做一定的约束，可以通过类型边界来约束。

4.1 语法

```
class 泛型类名称<类型形参 extends 类型边界> {
    ...
}
```

4.2 示例

```
public class MyArrayList<E extends Number> {
    ...
}
```

只接受 Number 的子类型作为 E 的类型实参

```
MyArrayList<Integer> l1;        // 正常，因为 Integer 是 Number 的子类型
MyArrayList<String> l2;     // 编译错误，因为 String 不是 Number 的子类型
```

```
error: type argument String is not within bounds of type-variable E
        MyArrayList<String> l2;
                    ^
where E is a type-variable:
        E extends Number declared in class MyArrayList
```

了解： 没有指定类型边界 E，可以视为 E extends Object

4.3 复杂一点的示例

定义一个泛型类搜索树

```
public class BSTree<K extends Comparable<K>, V> {
    ...
}
```

传入的 K 必须是 Comparable 的，并且是可以和另一个 K 类型做比较的，后边的 K 是对类型参数的使用了

5. 类型擦除

我们之前已经讲过，泛型是作用在编译期间的一种机制，实际上运行期间是没有这么多类的，那运行期间是什么类型呢？这里就是类型擦除在做的事情。

```
class MyArrayList<E> {
    // E 会被擦除为 Object
}

class MyArrayList<E extends Comparable<E>> {
    // E 被擦除为 Comprable
}
```

总结： 即类型擦除主要看其类型边界而定

了解： 编译器在类型擦除阶段在做什么？

1. 将类型变量用擦除后的类型替换，即 Object 或者 Comparable
2. 加入必要的类型转换语句
3. 加入必要的 `bridge method` 保证多态的正确性

6. 泛型类的使用-通配符(Wildcards)

6.1 基本

`?` 用于在泛型的使用，即为通配符

示例

```
public class MyArrayList<E> {...}

// 可以传入任意类型的 MyArrayList
public static void printAll(MyArrayList<?> list) {
    ...
}

// 以下调用都是正确的
printAll(new MyArrayList<String>());
printAll(new MyArrayList<Integer>());
printAll(new MyArrayList<Double>());
printAll(new MyArrayList<Number>());
printAll(new MyArrayList<Object>());
```

6.2 通配符-上界

语法

```
<? extends 上界>
```

示例

```
// 可以传入类型实参是 Number 子类的任意类型的 MyArrayList
public static void printAll(MyArrayList<? extends Number> list) {
    ...
}

// 以下调用都是正确的
printAll(new MyArrayList<Integer>());
printAll(new MyArrayList<Double>());
printAll(new MyArrayList<Number>());
// 以下调用是编译错误的
printAll(new MyArrayList<String>());
printAll(new MyArrayList<Object>());
```

注意： 需要区分 **泛型使用**中的通配符上界 和 **泛型定义**中的类型上界

6.3 通配符-下界

语法

```
<? super 下界>
```

示例

```java
// 可以传入类型实参是 Integer 父类的任意类型的 MyArrayList
public static void printAll(MyArrayList<? super Integer> list) {
    ...
}

// 以下调用都是正确的
printAll(new MyArrayList<Integer>());
printAll(new MyArrayList<Number>());
printAll(new MyArrayList<Object>());
// 以下调用是编译错误的
printAll(new MyArrayList<String>());
printAll(new MyArrayList<Double>());
```

7. 泛型中的父子类型（重要）

```java
public class MyArrayList<E> { ... }

// MyArrayList<Object> 不是 MyArrayList<Number> 的父类型
// MyArrayList<Number> 也不是 MyArrayList<Integer> 的父类型

// 需要使用通配符来确定父子类型
// MyArrayList<?> 是 MyArrayList<? extends Number> 的父类型
// MyArrayList<? extends Number> 是 MyArrayList<Integer> 的父类型
```

8. 泛型方法

8.1 定义语法

```
方法限定符 <类型形参列表> 返回值类型 方法名称(形参列表) { ... }
```

8.2 示例

```java
public class Util {
    public static <E> void swap(E[] array, int i, int j) {
        E t = array[i];
        array[i] = array[j];
        array[j] = t;
    }
}
```

8.3 使用示例-可以类型推导

```java
Integer[] a = { ... };
swap(a, 0, 9);

String[] b = { ... };
swap(b, 0, 9);
```

8.4 使用示例-不使用类型推导

```
Integer[] a = { ... };
Util.<Integer>swap(a, 0, 9);

String[] b = { ... };
Util.<String>swap(b, 0, 9);
```

9. 泛型的限制

1. 泛型类型参数不支持基本数据类型
2. 无法实例化泛型类型的对象
3. 无法使用泛型类型声明静态的属性
4. 无法使用 instanceof 判断带类型参数的泛型类型
5. 无法创建泛型类数组
6. 无法 create、catch、throw 一个泛型类异常（异常不支持泛型）
7. 泛型类型不是形参一部分，无法重载

10. 完整定义一份泛型类支持的搜索树（不使用 Comparator）

```
import java.util.*;

public class BSTree<K extends Comparable<K>, V> {
    private static class Entry<K, V> {
        private K key;
        private V value;
        private Entry<K, V> left = null;
        private Entry<K, V> right = null;

        private Entry(K key, V value) {
            this.key = key;
            this.value = value;
        }

        @Override
        public String toString() {
            return String.format("{%s=%s}", key, value);
        }
    }

    private Entry<K, V> root = null;

    public V get(K key) {
        Entry<K, V> cur = root;
        while (cur != null) {
            int r = key.compareTo(cur.key);
            if (r == 0) {
                return cur.value;
            } else if (r < 0) {
                cur = cur.left;
```

```
            } else {
                cur = cur.right;
            }
        }

        return null;
    }

    public V put(K key, V value) {
        if (root == null) {
            root = new Entry<>(key, value);
            return null;
        }

        Entry<K, V> parent = null;
        Entry<K, V> cur = root;
        while (cur != null) {
            int r = key.compareTo(cur.key);
            if (r == 0) {
                V oldValue = cur.value;
                cur.value = value;
                return oldValue;
            } else if (r < 0) {
                parent = cur;
                cur = cur.left;
            } else {
                parent = cur;
                cur = cur.right;
            }
        }

        Entry<K, V> entry = new Entry<>(key, value);
        if (key.compareTo(parent.key) < 0) {
            parent.left = entry;
        } else {
            parent.right = entry;
        }
        return null;
    }

    public V remove(K key) {
        Entry<K, V> parent = null;
        Entry<K, V> cur = root;
        while (cur != null) {
            int r = key.compareTo(cur.key);
            if (r == 0) {
                V oldValue = cur.value;
                removeEntry(parent, cur);
                return oldValue;
            } else if (r < 0) {
                parent = cur;
                cur = cur.left;
            } else {
```

```
                parent = cur;
                cur = cur.right;
            }
        }

        return null;
    }

    private void removeEntry(Entry<K, V> parent, Entry<K, V> cur) {
        if (cur.left == null) {
            if (cur == root) {
                root = cur.right;
            } else if (cur == parent.left) {
                parent.left = cur.right;
            } else {
                parent.right = cur.right;
            }
        } else if (cur.right == null) {
            if (cur == root) {
                root = cur.left;
            } else if (cur == parent.left) {
                parent.left = cur.left;
            } else {
                parent.right = cur.left;
            }
        } else {
            Entry<K, V> gParent = cur;
            Entry<K, V> goat = cur.left;
            while (goat.right != null) {
                gParent = goat;
                goat = goat.right;
            }
            cur.key = goat.key;
            cur.value = goat.value;
            if (goat == gParent.left) {
                gParent.left = goat.left;
            } else {
                gParent.right = goat.left;
            }
        }
    }

    public static interface Function<T> {
        void apply(T entry);
    }

    public static <K, V> void preOrderTraversal(Entry<K, V> node, Function<Entry<K, V>>
func) {
        if (node != null) {
            func.apply(node);
            preOrderTraversal(node.left, func);
            preOrderTraversal(node.right, func);
        }
```

```
    }

    public static <K, V> void inOrderTraversal(Entry<K, V> node, Function<Entry<K, V>>
func) {
        if (node != null) {
            inOrderTraversal(node.left, func);
            func.apply(node);
            inOrderTraversal(node.right, func);
        }
    }

    public void print() {
        System.out.println("前序遍历: ");
        preOrderTraversal(root, (n) -> {
            System.out.print(n.key + " ");
        });
        System.out.println();
        System.out.println("中序遍历: ");
        inOrderTraversal(root, (n) -> {
            System.out.print(n.key + " ");
        });
        System.out.println();
    }

    public static void main(String[] args) {
        BSTree<Integer, String> tree = new BSTree<>();
        int count = 0;
        Random random = new Random(20190915);
        for (int i = 0; i < 20; i++) {
            int key = random.nextInt(200);
            String value = String.format("Value of %d", key);
            if (tree.put(key, value) == null) {
                count++;
            }
        }
        System.out.println("一共插入 " + count + " 个结点");
        tree.print();
    }
}
```

内容重点总结

- 以可以去阅读java集合代码为前提学习泛型