Collection接口详解

Collection接口层次结构

Java集合框架中的Collection接口是所有集合类的基础，它定义了集合类应该具备的基本操作和行为。其子接口和实现类形成了一层层的层次结构，方便了集合类的分类和使用。

结构示意图

```
java.lang.Object
    └ java.util.Collection<E>
            ├ java.util.List<E>
            |     ├ java.util.ArrayList<E>
            |     ├ java.util.LinkedList<E>
            |     └ java.util.Vector<E>
            |           └ java.util.Stack<E>
            ├ java.util.Set<E>
            |     ├ java.util.HashSet<E>
            |     |     └ java.util.LinkedHashSet<E>
            |     ├ java.util.TreeSet<E>
            |     └ java.util.EnumSet<E>
            ├ java.util.Queue<E>
            |     ├ java.util.ArrayDeque<E>
            |     ├ java.util.LinkedList<E>
            |     └ java.util.PriorityQueue<E>
            └ java.util.Deque<E>
                  ├ java.util.ArrayDeque<E>
                  └ java.util.LinkedList<E>
```

Collection接口的主要子接口包括：

- List接口：继承自Collection接口，允许有重复元素，元素有序，支持按照下标访问元素。
- Set接口：继承自Collection接口，不允许有重复元素，元素无序。
- Queue接口：继承自Collection接口，用于实现队列数据结构，支持在队列头部插入元素，队列尾部删除元素，元素有序。
- Deque接口：继承自Queue接口，支持在队列头部和尾部都可以插入和删除元素，因此也可以用于实现栈数据结构。

Collection接口的主要实现类包括：

- ArrayList类：实现了List接口，底层基于动态数组实现，支持随机访问和快速插入、删除元素。
- LinkedList类：实现了List接口，底层基于双向链表实现，支持快速插入、删除元素，但访问元素需要遍历链表，效率较低。
- HashSet类：实现了Set接口，底层基于哈希表实现，元素无序，查询、插入、删除元素的时间复杂度都为O(1)。
- TreeSet类：实现了SortedSet接口，底层基于红黑树实现，元素有序，查询、插入、删除元素的时间复杂度都为O(logN)。

- PriorityQueue类：实现了Queue接口，底层基于堆实现，元素按照优先级有序，插入、删除元素的时间复杂度为O(logN)。
- ArrayDeque类：实现了Deque接口，底层基于数组实现，支持双向插入、删除元素，效率较高。

还有一些其他的实现类，如LinkedHashSet、HashMap、TreeMap等，都是在上述基础上进行了一些扩展和优化。这些实现类的存在，使得Java集合框架可以满足不同场景下的需求。

Collection 接口中常用的方法

方法	描述
boolean add(E e)	向集合中添加元素
boolean addAll(Collection<? extends E> c)	将另一个集合中的所有元素添加到当前集合中
void clear()	清空集合中的所有元素
boolean contains(Object o)	判断集合中是否包含某个元素
boolean containsAll(Collection<?> c)	判断当前集合是否包含另一个集合中的所有元素
boolean isEmpty()	判断集合是否为空
Iterator iterator()	返回一个迭代器，用于遍历集合中的元素
boolean remove(Object o)	从集合中删除指定的元素
boolean removeAll(Collection<?> c)	删除当前集合中与另一个集合相同的所有元素
boolean retainAll(Collection<?> c)	保留当前集合中与另一个集合相同的所有元素，删除不同的元素
int size()	返回集合中元素的数量
Object[] toArray()	将集合转换为数组
T[] toArray(T[] a)	将集合转换为指定类型的数组

迭代器Iterator

Iterator 是 Java 集合框架中提供的一种用于遍历集合元素的接口，可以对任何实现了 java.util.Collection 接口的集合类进行遍历操作。通过使用迭代器，可以不依赖于集合类的具体实现，而对集合进行迭代遍历，从而使得集合与迭代算法相分离。

Iterator 接口主要定义了以下方法：

- boolean hasNext()：判断是否还有下一个元素，如果有返回 true，否则返回 false。
- E next()：返回下一个元素，并将迭代器的指针向后移动一个位置。
- void remove()：从集合中移除上一个元素。

在使用迭代器遍历集合时，通常会采用如下的模板代码：

```
Iterator<E> it = collection.iterator();
while (it.hasNext()) {
    E e = it.next();
    // do something with e
}
```

> 其中 collection 是要遍历的集合，E 是集合中元素的类型，it 是迭代器对象，通过调用 iterator() 方法来获取。在遍历集合时，通过 hasNext() 方法判断是否还有下一个元素，如果有则通过 next() 方法获取下一个元素并对其进行处理。如果需要从集合中移除元素，则可以调用 remove() 方法，该方法会将上一个元素从集合中移除。
>
> 需要注意的是，在使用迭代器遍历集合时，不能通过集合的 add()、remove() 方法添加或删除元素，否则会抛出 ConcurrentModificationException 异常。如果需要添加或删除元素，则必须使用迭代器的 remove() 方法来完成。

示例代码

使用 ArrayList 实现类创建集合对象

```
import java.util.ArrayList;
import java.util.Collection;

//定义一个Book类
class Book {
    String title;
    String author;

    Book(String title, String author) {
        this.title = title;
        this.author = author;
    }

    public String toString() {
        return "Book[title=" + title + ", author=" + author + "]";
    }
}

public class CollectionTest {
    public static void main(String[] args) {
        //创建一个ArrayList实例
        Collection<Book> books = new ArrayList<>();

        //向集合中添加元素
        books.add(new Book("Java编程思想", "Bruce Eckel"));
        books.add(new Book("Effective Java", "Joshua Bloch"));
        books.add(new Book("深入理解Java虚拟机", "周志明"));

        //将集合转换为Object类型的数组，并遍历该数组
        Object[] objs = books.toArray();
        for (int i = 0; i < objs.length; i++) {
            System.out.println(objs[i]);
```

```
        }

        //获取元素的个数
        System.out.println(books.size()); //3
        System.out.println(books.isEmpty()); //false

        //清空元素
        books.clear();
        System.out.println(books.size()); //0
        System.out.println(books.isEmpty()); //true
    }
}
```

> 在这个示例中，我们创建了一个 `Book` 类，它有一个标题和作者属性，并实现了 `toString()` 方法。然后我们创建了一个 `Collection` 实例 `books`，它的类型是 `ArrayList<Book>`。我们向该集合中添加了三本书，并使用 `toArray()` 方法将集合转换为对象数组，并遍历了该数组。最后，我们打印了元素的个数和是否为空，并清空了该集合。

Collection 接口常用用法

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

//定义一个Person类
class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String toString() {
        return "Person[name=" + name + ", age=" + age + "]";
    }
}

public class CollectionExample {
    public static void main(String[] args) {
        // 创建一个ArrayList实例
        Collection<Person> c = new ArrayList<Person>();

        // 向集合中添加元素
        c.add(new Person("张三", 20));
        c.add(new Person("李四", 25));
        c.add(new Person("王五", 30));
        c.add(new Person("赵六", 35));

        // 判断集合中是否包含某个元素
```

```
        Person p = new Person("张三", 20);
        System.out.println(c.contains(p)); // true

        // 将集合转换为Object类型的数组，并遍历该数组
        Object[] objs = c.toArray();
        for (int i = 0; i < objs.length; i++) {
            System.out.println(objs[i]);
        }

        // 使用Iterator迭代器遍历集合
        Iterator<Person> it = c.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }

        // 获取元素的个数
        System.out.println(c.size()); // 4

        // 判断集合是否为空
        System.out.println(c.isEmpty()); // false

        // 清空集合
        c.clear();
        System.out.println(c.size()); // 0
        System.out.println(c.isEmpty()); // true
    }
}
```

> 以上代码演示了 Collection 接口的常用方法，包括添加元素、判断集合中是否包含某个元素、将集合转换为数组、使用 Iterator 迭代器遍历集合、获取元素的个数、判断集合是否为空和清空集合等操作。

使用迭代器来遍历集合

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

//定义一个Person类
class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String toString() {
        return "Person[name=" + name + ", age=" + age + "]";
    }
}
```

```
public class IteratorExample {
    public static void main(String[] args) {
        // 创建一个ArrayList实例
        Collection<Person> c = new ArrayList<Person>();

        // 向集合中添加元素
        c.add(new Person("张三", 20));
        c.add(new Person("李四", 25));
        c.add(new Person("王五", 30));
        c.add(new Person("赵六", 35));

        // 使用迭代器遍历集合
        Iterator<Person> it = c.iterator();
        while (it.hasNext()) {
            Person p = it.next();
            if (p.age > 30) {
                it.remove(); // 删除集合中的元素
            } else {
                System.out.println(p);
            }
        }

        // 判断集合是否为空
        System.out.println(c.isEmpty()); // false

        // 清空集合
        c.clear();
        System.out.println(c.size()); // 0
        System.out.println(c.isEmpty()); // true
    }
}
```

> 在遍历集合时使用迭代器，可以方便地对集合进行增删改查操作。上面这段示例代码中，我们使用迭代器遍历一个包含Person对象的集合，如果该对象的年龄大于30岁，则从集合中删除该对象，否则打印该对象。
>
> 在这个示例代码中，我们使用了Iterator接口的几个常用方法：
>
> - hasNext()：判断集合中是否还有元素可以遍历。
> - next()：获取集合中下一个元素。
> - remove()：删除集合中上一次next()方法返回的元素。
>
> 这些方法可以方便地遍历集合并进行增删改查操作，是集合框架中非常实用的接口。

使用 Collection 接口常用方法

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class CollectionDemo {
```

```java
    public static void main(String[] args) {
        // 创建一个 Collection 对象
        Collection<String> c = new ArrayList<>();

        // 添加元素
        c.add("hello");
        c.add("world");
        c.add("java");

        // 输出集合元素数量
        System.out.println("集合中元素的数量为：" + c.size());

        // 判断集合是否为空
        System.out.println("集合是否为空：" + c.isEmpty());

        // 判断集合是否包含指定元素
        System.out.println("集合是否包含 \"hello\" 元素：" + c.contains("hello"));

        // 获取迭代器并遍历集合元素
        Iterator<String> it = c.iterator();
        while (it.hasNext()) {
            String s = it.next();
            System.out.println("集合元素为：" + s);
        }

        // 将集合转换为数组并输出
        Object[] arr = c.toArray();
        for (Object o : arr) {
            System.out.println("集合转换为数组后的元素为：" + o);
        }

        // 移除指定元素
        c.remove("java");
        System.out.println("移除元素 \"java\" 后，集合元素为：" + c);

        // 移除所有元素
        c.clear();
        System.out.println("移除所有元素后，集合元素为：" + c);

        // 判断两个集合是否相等
        Collection<String> c1 = new ArrayList<>();
        c1.add("hello");
        c1.add("world");
        c1.add("java");

        Collection<String> c2 = new ArrayList<>();
        c2.add("world");
        c2.add("java");
        c2.add("hello");

        System.out.println("c1 和 c2 是否相等：" + c1.equals(c2));
    }
```

```
}
```

使用迭代器和 Lambda 表达式实现遍历集合并删除指定元素

```
Collection<Integer> c = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5, 6));
c.removeIf(e -> e % 2 == 0); // 删除偶数元素
c.forEach(System.out::println); // 遍历集合
```

使用 Stream API 实现集合元素过滤、映射和统计：

```
List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5, 6));
long count = list.stream() // 获取流
        .filter(e -> e % 2 == 0) // 过滤偶数元素
        .map(e -> e * e) // 映射为平方
        .count(); // 统计个数
System.out.println(count); // 3
```

使用 Comparator 接口实现集合元素排序

```
List<Person> list = new ArrayList<>(Arrays.asList(
        new Person("Tom", 20),
        new Person("Jerry", 25),
        new Person("Alice", 18)));
Collections.sort(list, Comparator.comparing(Person::getAge)); // 按年龄升序排序
list.forEach(System.out::println); // 遍历集合
```

使用 Collections 类实现集合的二分查找和随机排序

```
List<Integer> list = new ArrayList<>(Arrays.asList(2, 3, 5, 8, 10));
int index = Collections.binarySearch(list, 5); // 二分查找元素 5 的下标
System.out.println(index); // 2

Collections.shuffle(list); // 随机排序
list.forEach(System.out::println); // 遍历集合
```