

迭代器

★ 集合类都是支持 `foreach` 语句的，但是虽然你在代码里面写的是 `foreach` 写法实际上是语法糖，但是在编译后就会切换到迭代器的语法

```
1 //foreach语法
2 public static void main(String[] args) {
3     List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C"));
4     for(String s : list){
5         System.out.println(s);
6     }
7 }
8
9 //编译后
10 public static void main(String[] args) {
11     List<String> list = new ArrayList(Arrays.asList("A", "B", "C"));
12     Iterator var2 = list.iterator(); //使用了list的迭代器
13
14     while(var2.hasNext()) {
15         String s = (String)var2.next();
16         System.out.println(s);
17     }
```

1 自己来解读一下这个啥意思，`List` 的父类 `Collection` 继承了可迭代接口 `Iterable`，里面有一个 `Iterator` 的迭代器方法需要实现

```
1 public interface Collection<E> extends Iterable<E> //继承
2
3 public interface Iterable<T> {
4     /**
5      * Returns an iterator over elements of type {@code T}.
6      *
7      * @return an Iterator.
8      */
9     Iterator<T> iterator(); //迭代器方法
10 }
```

2 可以看看 `Iterator` 迭代器内的方法

```
1 public interface Iterator<E> {
2     //看看是否还有下一个元素，放回boolean值
3     boolean hasNext();
4
5     //遍历当前元素，并将下一个元素作为待遍历元素
6     E next();
7
8     //移除上一个被遍历的元素（某些集合不支持这种操作）
```

```

9      default void remove() {
10          throw new UnsupportedOperationException("remove");
11      }
12
13      //对剩下的元素进行自定义遍历操作
14      default void forEachRemaining(Consumer<? super E> action) {
15          Objects.requireNonNull(action);
16          while (hasNext())
17              action.accept(next());
18      }
19  }

```

3 在回头去看看编译后的代码，这样就好理解了

```

1  //编译后
2  public static void main(String[] args) {
3      List<String> list = new ArrayList(Arrays.asList("A", "B", "C"));
4      Iterator var2 = list.iterator(); //使用了list的迭代器
5
6      while(var2.hasNext()) { //去调用Itertor迭代器hasNext方法，查看下一个元素有就返回
true
7          String s = (String)var2.next(); //遍历当前的元素
8          System.out.println(s); //打印当前元素
9      }

```

★ 迭代器的运作机制大概是



一个新的迭代器就像上面这样，默认有一个指向集合中第一个元素的指针：



每一次 `next` 操作，都会将指针后移一位，直到完成每一个元素的遍历，此时再调用 `next` 将不能再得到下一个元素。至于为什么要这样设计，是因为集合类的实现方案有很多，可能是链式存储，也有可能是数组存储，不同的实现有着不同的遍历方式，而迭代器则可以将多种多样不同的集合类遍历方式进行统一，只需要各个集合类根据自己的情况进行对应实现就行了

1 在不同的集合类型中，实现 `next` 的方法也是不一样的可以看看

```

1  //ArrayList直接访问下标
2  public E next() {
3      ...
4      cursor = i + 1; //移动指针
5      return (E) elementData[lastRet = i]; //直接返回指针所指元素

```

```

6   }
7
8
9   //LinkedList不断向后寻找终点
10
11  public E next() {
12      ...
13      next = next.next;    //向后继续寻找结点
14      nextIndex++;
15      return lastReturned.item; //返回结点内部存放的元素
16  }

```

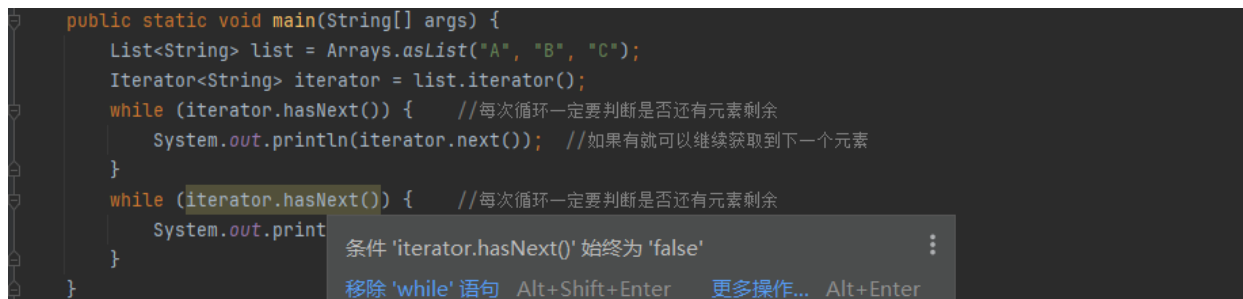
2 但是无论内部是如何实现的，我们在使用的时候直接使用迭代器就好了，使用方法都是一样的

```

1  public static void main(String[] args) {
2      List<String> list = Arrays.asList("A", "B", "C");
3      Iterator<String> iterator = list.iterator();
4      while (iterator.hasNext()) {    //每次循环一定要判断是否还有元素剩余
5          System.out.println(iterator.next()); //如果有就可以继续获取到下一个元素
6      }
7  }

```

★ 迭代器使用时一次性的，也就是一个迭代器对象，只能使用一次，如果想要再次遍历就需要再去生成一个迭代器对象，所以为了简便就直接使用 `foreach` 就好了



```

public static void main(String[] args) {
    List<String> list = Arrays.asList("A", "B", "C");
    Iterator<String> iterator = list.iterator();
    while (iterator.hasNext()) {    //每次循环一定要判断是否还有元素剩余
        System.out.println(iterator.next()); //如果有就可以继续获取到下一个元素
    }
    while (iterator.hasNext()) {    //每次循环一定要判断是否还有元素剩余
        System.out.print
    }
}

```

条件 'iterator.hasNext()' 始终为 'false'
 移除 'while' 语句 Alt+Shift+Enter 更多操作... Alt+Enter

```

1  public static void main(String[] args) {
2      List<String> list = Arrays.asList("A", "B", "C");
3      for (String s : list) {
4          System.out.println(s);
5      }
6  }

```

3 在 JAVA8 中提供了一个支持 Lambda 表达式的 `forEach` 方法，这个方法需要接受一个 `Consumer` 对象(可以去泛型笔记看看函数式接口),

```

1  public static void main(String[] args) {
2      List<String> list = Arrays.asList("A", "B", "C");
3      list.forEach(new Consumer<String>() {
4          @Override
5          public void accept(String s) {
6              System.out.println(s);
7          }
8      });
9  }

```

```

8         });
9     }
10
11    //缩减成lambda表达式
12    public static void main(String[] args) {
13        List<String> list = Arrays.asList("A", "B", "C");
14        list.forEach(s -> System.out.println(s));
15    }
16
17    //还可以继续替换成方法引用
18    public static void main(String[] args) {
19        List<String> list = Arrays.asList("A", "B", "C");
20        list.forEach(System.out::println);
21    }

```

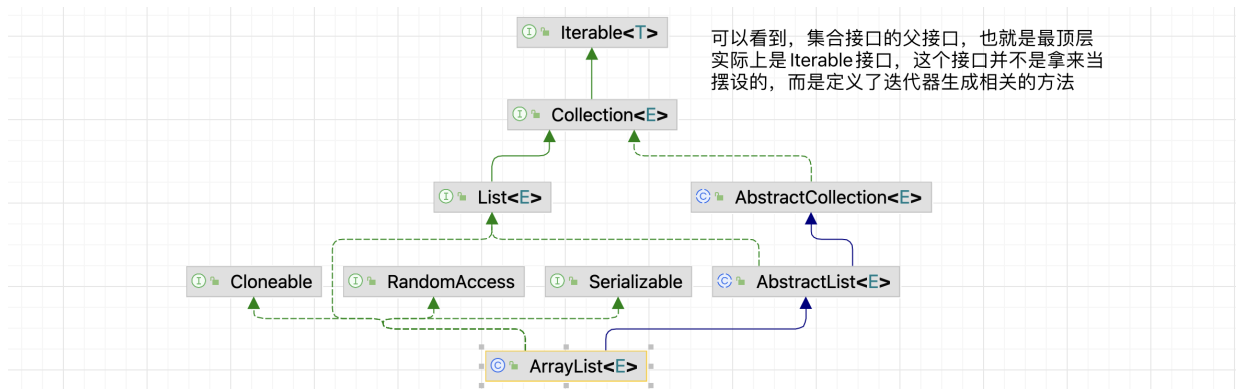
但是它本质也是使用的 `foreach` 循环来遍历打印，这个方法实在 `Iterable` 接口中定义的

```

1    default void forEach(Consumer<? super T> action) {
2        Objects.requireNonNull(action);
3        for (T t : this) { //foreach语法遍历每一个元素
4            action.accept(t); //调用Consumer的accept来对每一个元素进行消费
5        }
6    }

```

★ `Iterator` 是继承自 `Iterable`，看一下它的源代码



```

1    //注意这个接口是集合接口的父接口，不要跟之前的迭代器接口搞混了
2    public interface Iterable<T> {
3        //生成当前集合的迭代器，在Collection接口中重复定义了一次
4        Iterator<T> iterator();
5
6        //Java8新增方法，因为是在顶层接口中定义的，因此所有的集合类都有这个方法
7        default void forEach(Consumer<? super T> action) {
8            Objects.requireNonNull(action);
9            for (T t : this) {
10                action.accept(t);
11            }
12        }
13
14        //这个方法会在多线程部分中进行介绍，暂时不做讲解

```

```

15     default Spliterator<T> spliterator() {
16         return Spliterators.spliteratorUnknownSize(iterator(), 0);
17     }
18 }

```

1 Iterable 提供了迭代器的生成方法，实际上只要是实现了迭代器接口的类（我们自己写的都行），都可以使用 `foreach` 语法

```

1
2 public class Main {
3     public static void main(String[] args) {
4         Test test = new Test();
5         // test.forEach(System.out::println);
6         //或则
7         for (String s : test){
8             System.out.println(s);
9         }
10    }
11    public static class Test implements Iterable<String>{ //写一个类实现Iterable接
12        口
13
14        @Override
15        public Iterator<String> iterator() { //需要重写iterator方法
16            return new Iterator<String>() { //创建一个迭代器对象，需要通过内部类来实现
17                它的两个方法
18
19                @Override
20                public boolean hasNext() { //表示一直会返回true,也就是告诉程序我这里
21                    面会一直有数据
22
23                    return true;
24                }
25
26                @Override
27                public String next() { //返回当前数据
28                    return "TestIng...";
29                }
30            };
31        }
32    }
33 }

```

★ `ListIterator` 这个迭代器是针对 `List` 的强化版本，增加了很多操作，`List` 是有序集合，使用他还支持两种方向的遍历，从后往前，从前往后

```

1 public interface ListIterator<E> extends Iterator<E> {
2     //原本就有的
3     boolean hasNext();
4
5     //原本就有的
6     E next();
7 }

```

```

8      //查看前面是否有已经遍历的元素
9      boolean hasPrevious();
10
11     //跟next相反，这里是倒着往回遍历
12     E previous();
13
14     //返回下一个待遍历元素的下标
15     int nextIndex();
16
17     //返回上一个已遍历元素的下标
18     int previousIndex();
19
20     //原本就有的
21     void remove();
22
23     //将上一个已遍历元素修改为新的元素
24     void set(E e);
25
26     //在遍历过程中，插入新的元素到当前待遍历元素之前
27     void add(E e);
28 }

```

1 测试一下

```

1      public static void main(String[] args) {
2          List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C"));
3          ListIterator<String> iterator = list.listIterator();
4          iterator.next();    //此时得到A
5          iterator.set("X");  //将A原本位置的上的元素设定为成新的
6          System.out.println(list);
7          iterator.remove();  //会删除上一个遍历
8          System.out.println(list);
9      }
10
11     //输出：
12     [X, B, C]
13     [B, C]

```