lambda表达式&集合框架1

一 内容回顾（列举前一天重点难点内容）

1.1 教学重点:

```
1.理解字符串的原理
2.掌握String的常用方法使用
3.掌握StringBuffer/StringBuilder的常用方法使用
4.熟练编写简单的正则表达式
```

1.2 教学难点:

```
1.字符串原理的理解
字符串作为一个特殊的存在,使用广泛,理解了原理使用起来会更加得心应手,不过在学习字符串时应该先学会常用使用方法,再慢慢理解原理
2.正则表达式的进阶使用
正则表达式的作用很强大,如果有精力,可以深入学习一下,基本要求是先能看懂正则表达式,然后可以自己写一些简单的即可.
```

二 教学目标

```
1.掌握Lambda表达式的概念
2.掌握Lambda表达式的基本使用
3.熟悉Lambda表达式的使用场景
4.掌握集合的分类特点
5.掌握集合的常用方法
6.掌握集合的遍历
7.掌握迭代器的工作原理
8.掌握List的常用方法
9.掌握List的遍历排序
```

三 教学导读

3.1. Lambda表达式

```
lambda表达式， 是Java8的一个新特性， 也是Java8中最值得学习的新特性之一。

lambda表达式， 从本质来讲， 是一个匿名函数。 可以使用这个匿名函数， 实现接口中的方法。 对接口进行非常简洁的实现， 从而简化代码。
```

3.2. 集合框架

3.2.1. 集合的概念

集合与数组类似， 是一个数据容器， 用来存储引用数据类型的数据。 在Java中， 集合不是泛指某一个类， 而是若干个类组成的数据结构的实现。

Java的集合类是 **java.util** 包中的重要内容，它允许以各种方式将元素分组，并定义了各种使这些元素更容易操作的方法。

Java集合类是Java将一些基本的和使用频率极高的基础类进行封装和增强后再以一个类的形式提供。

Java集合类是可以往里面保存多个对象的类，存放的是对象，不同的集合类有不同的功能和特点，适合不同的场合，用以解决一些实际问题。

3.2.2. 集合的特点

- 集合类这种框架是高性能的。对基本类集（动态数组，链接表，树和散列表）的实现是高效率的。
- 集合类允许不同类型的集合以相同的方式和高度互操作方式工作
- 集合类容易扩展和修改，程序员可以很容易地稍加改造就能满足自己的数据结构需求

3.2.3. 集合和数组的区别

- **存储的数据类型**
  - 数组中可以存储基本数据类型的数据， 也可以存储引用数据类型的数据
  - 集合中只能存储引用数据类型的数据， 基本数据类型的数据需要进行装箱， 才能存入集合中。
- **长度**
  - 数组是定长的容器， 一旦实例化完成， 数组的长度不能发生改变。即数组不能动态添加、删除元素。
  - 集合是变长的容器， 长度可以发生改变。 即集合中可以随时添加、删除元素。
- **元素操作**
  - 数组只能通过下标进行元素的访问， 如果需要其他的操作， 例如排序， 需要自己写算法实现。
  - 集合中封装了若干对元素进行操作的方法， 直接使用集合， 比较方便。

3.2.4. 使用集合的好处

- **降低编程难度：**

  在编程中会经常需要链表、向量等集合类，如果自己动手写代码实现这些类，需要花费较多的时间和精力。调用Java中提供的这些接口和类，可以很容易的处理数据。

- **提升程序的运行速度和质量：**

  Java提供的集合类具有较高的质量，运行时速度也较快。使用这些集合类提供的数据结构，程序员可以从“重复造轮子”中解脱出来，将精力专注于提升程序的质量和性能。

- **无需再学习新的API：**

  借助泛型，只要了解了这些类的使用方法，就可以将它们应用到很多数据类型中。如果知道了LinkedList<String>的使用方法，也会知道LinkedList<Double>怎么用，则无需为每一种数据类型学习不同的API。

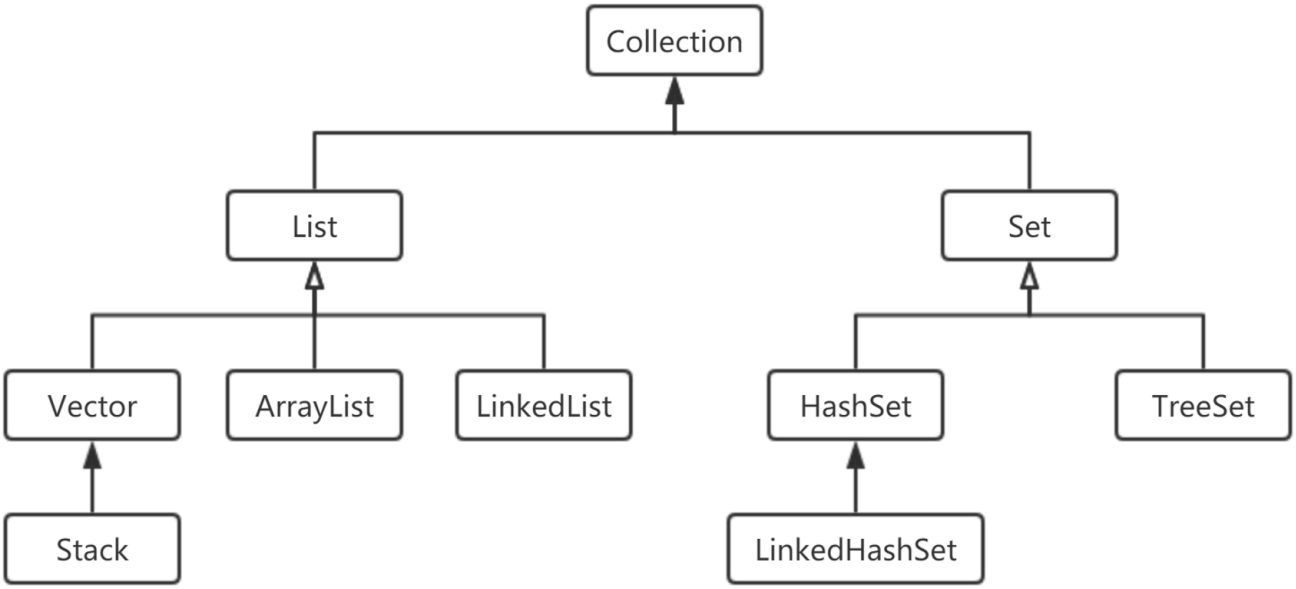
- **增加代码重用性：**

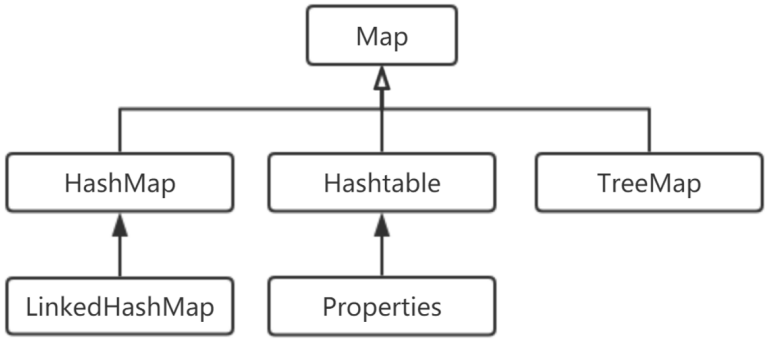
  也是借助泛型，就算对集合类中的元素类型进行了修改，集合类相关的代码也几乎不用修改。

3.2.5. Java中的集合框架图

Java中的集合，大致分为两类。分别是 Collection 集合和 Map 集合。

其中，Collection是单列集合的顶级接口，Map接口是双列集合的顶级接口。





四 教学内容

> **课程重点**
>
> - 函数式接口
>   - 函数式接口的概念
>   - 函数式接口的判断
>   - @FunctionalInterface
> - **lambda表达式的语法**
>   - **基础的语法**
>   - **语法的精简（参数、方法体）**
> - 函数引用

- 静态、非静态、构造方法的引用
- 对象方法的特殊引用

4.1 Lambda表达式(会)

4.1.1. Lambda表达式的简介

4.1.1.1. Lambda表达式的使用场景

通常来讲， 使用lambda表达式， 是为了简化接口实现的。

关于接口实现， 可以有很多种方式来实现。 例如： 设计接口的实现类、使用匿名内部类。 但是lambda表达式， 比这两种方式都简单。

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Date 2020/4/8
 * @Description
 */
interface SingleReturnSingleParameter{
    int test(int a);
}

public class Program {
    public static void main(String[] args) {
        // 无参、无返回值的函数式接口
        interfaceImpl();
    }

    private static void interfaceImpl() {
        // 1. 使用显式的实现类对象
        SingleReturnSingleParameter parameter1 = new
Impl();
```

```
        // 2. 使用匿名内部类实现
        SingleReturnSingleParameter parameter2 = new SingleReturnSingleParameter() {
            @Override
            public int test(int a) {
                return a * a;
            }
        };

        // 3. 使用lambda表达式实现
        SingleReturnSingleParameter parameter3 = a -> a * a;

        System.out.println(parameter1.test(10));
        System.out.println(parameter2.test(10));
        System.out.println(parameter3.test(10));
    }

    private static class Impl implements SingleReturnSingleParameter {

        @Override
        public int test(int a) {
            return a * a;
        }
    }
}
```

4.1.1.2. Lambda表达式对接口的要求

虽然说， lambda表达式可以在一定程度上简化接口的实现。 但是， 并不是所有的接口都可以使用lambda表达式来简洁实现的。

lambda表达式毕竟只是一个匿名方法。 当实现的接口中的方法过多或者多少的时候， lambda表达式都是不适用的。

lambda表达式，只能实现**函数式接口**。

4.1.1.3. 函数式接口

- 基础概念

如果说， 一个接口中， 要求实现类**必须**实现的抽象方法， 有且只有一个！ 这样的接口， 就是函数式接口。

```
1  // 这个接口中， 有且只有一个方法， 是实现类必须实现的， 因此是一个函数式接口
2  interface Test1 {
3      void test();
4  }
5  // 这个接口中， 实现类必须要实现的方法， 有两个！ 因此不是一个函数式接口
6  interface Test2 {
7      void test1();
8      void test2();
9  }
10 // 这个接口中， 实现类必须要实现的方法， 有零个！ 因此不是一个函数式接口
11 interface Test3 {
12
13 }
```

```java
// 这个接口中， 虽然没有定义任何的方法， 但是可以从父接口中继承到一个抽象方法的。 是一个函数式接口
interface Test4 extends Test1 {

}
// 这个接口， 虽然里面定义了两个方法， 但是defualt方法子类不是必须实现的。
// 因此， 实现类实现这个接口的时候， 必须实现的方法只有一个！是一个函数式接口。
interface Test5 {
    void test5();
    default void test() {}
}
// 这个接口中的 toString 方法， 是Object类中定义的方法。
// 此时， 实现类在实现接口的时候， toString可以不重写的！ 因为可以从父类Object中继承到！
// 此时， 实现类在实现接口的时候， 有且只有一个方法是必须要重写的。 是一个函数式接口！
interface Test6 {
    void test6();
    String toString();
}
```

> 思考题： 下面的两个接口是不是函数式接口?

```java
interface Test7 {
String toString();
}

interface Test8 {
void test();
default void test1() {}
static void test2() {}
String toString();
}
```

- @FunctionalInterface

是一个注解， 用在接口之前， 判断这个接口是否是一个函数式接口。 如果是函数式接口， 没有任何问题。 如果不是函数式接口， 则会报错。 功能类似于 @Override。

```java
@FunctionalInterface
interface FunctionalInterfaceTest {
    void test();
}
```

- 系统内置的若干函数式接口

接口名字	参数	返回值	特殊接口
Predicate<T>	T	boolean	IntPredicate: 参数int， 返回值 boolean LongPredicate: 参数 long， 返回值 boolean DoublePredicate: 参数 double， 返回值 boolean

Consumer<T>	T	void	IntConsumer: 参数int， 返回值 void LongConsumer: 参数 long， 返回值 void DoubleConsumer: 参数 double， 返回值 void
Function<T, R>	T	R	IntFunction<R>: 参数 int， 返回值 R IntToDoubleFunction: 参数 int， 返回值 double IntToLongFunction: 参数 int， 返回值 long LongFunction<R>: 参数 long， 返回值 R LongToIntFunction: 参数 long， 返回值 int LongToDoubleFunction: 参数 long， 返回值 double DoubleFunction<R>: 参数 double， 返回值 R DoubleToIntFunction: 参数 double， 返回值 int DoubleToLongFunction: 参数 double， 返回值 long
Supplier<T>	无	T	BooleanSupplier: 参数无， 返回值 boolean IntSupplier: 参数无， 返回值 int LongSupplier: 参数无， 返回值 long DoubleSupplier: 参数无， 返回值 double
UnaryOperator<T>	T	T	IntUnaryOperator: 参数 int， 返回值 int LongUnaryOperator: 参数 long, 返回值 long DoubleUnaryOperator: 参数 double, 返回值 double
BinaryOperator<T>	T, T	T	IntBinaryOperator: 参数 int, int， 返回值 int LongBinaryOperator: 参数 long, long， 返回值 long DoubleBinaryOperator: 参数 double, double, 返回值 double
BiPredicate<L, R>	L, R	boolean	
BiConsumer<T, U>	T, U	void	
BiFunction<T, U, R>	T, U	R	

4.1.2. Lambda表达式的语法

2.1. Lambda表达式的基础语法

lambda表达式， 其实本质来讲， 就是一个匿名函数。 因此在写lambda表达式的时候， 不需要关心方法名是什么。

实际上， 我们在写lambda表达式的时候， 也不需要关心返回值类型。

我们在写lambda表达式的时候， 只需要关注两部分内容即可： **参数列表** 和 **方法体**

lambda表达式的基础语法：

```
(参数)    ->   {
    方法体
};
```

参数部分： 方法的参数列表， 要求和实现的接口中的方法参数部分一致， 包括参数的数量和类型。

方法体部分： 方法的实现部分， 如果接口中定义的方法有返回值， 则在实现的时候， 注意返回值的返回。

-> : 分隔参数部分和方法体部分。

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
```

```
 * @Date 2020/4/8
 * @Description
 */
interface NoneReturnNoneParameter{
    void test();
}
public class Syntax {
    public static void main(String[] args) {
        // 1. 无参、无返回值的方法实现
        NoneReturnNoneParameter lambda1 = () -> {
            System.out.println("无参、无返回值方法的实现");
        };
        lambda1.test();

        // 2. 有参、无返回值的方法实现
        NoneReturnSingleParameter lambda2 = (int a) ->
{
            System.out.println("一个参数、无返回值方法的实
现: 参数是 " + a);
        };
        lambda2.test(10);

        // 3. 多个参数、无返回值方法的实现
        NoneReturnMutipleParameter lambda3 = (int a,
int b) -> {
            System.out.println("多个参数、无返回值方法的实
现: 参数a是 " + a + ", 参数b是 " + b);
        };
        lambda3.test(10, 20);

        // 4. 无参、有返回值的方法的实现
        SingleReturnNoneParameter lambda4 = () -> {
            System.out.println("无参、有返回值方法的实现");
```

```
33              return 666;
34          };
35          System.out.println(lambda4.test());
36
37          // 5. 一个参数、有返回值的方法实现
38          SingleReturnSingleParameter lambda5 = (int a) -
    > {
39              System.out.println("一个参数、有返回值的方法实
    现：参数是 " + a);
40              return a * a;
41          };
42          System.out.println(lambda5.test(9));
43
44          // 6. 多个参数、有返回值的方法实现
45          SingleReturnMutipleParameter lambda6 = (int a,
    int b) -> {
46              System.out.println("多个参数、有返回值的方法实
    现：参数a是 " + a + ", 参数b是 " + b);
47              return a * b;
48          };
49          System.out.println(lambda6.test(10, 20));
50      }
51  }
```

2.2. Lambda表达式的语法进阶

在上述代码中， 的确可以使用lambda表达式实现接口， 但是依然不够简洁， 有简化的空间。

4.1.2.2.1. 参数部分的精简

- 参数的类型
  - 由于在接口的方法中，已经定义了每一个参数的类型是什么。 而且在使用lambda表达式实现接口的时候， 必须要保证参数的数量和类型需要和接口中的方法保持一致。 因此， 此时lambda表达式中的参数的类型可以省略不写。
  - 注意事项:
    - 如果需要省略参数的类型， 要保证: 要省略， 每一个参数的类型都必须省略不写。 绝对不能出现， 有的参数类型省略了， 有的参数类型没有省略。

```
// 多个参数、无返回值的方法实现
 NoneReturnMutipleParameter lambda1 = (a, b) -> {
     System.out.println("多个参数、无返回值方法的实现:参数a是 " + a + ", 参数b是 " + b);
 };
```

- 参数的小括号
  - 如果方法的参数列表中的参数数量 **有且只有一个** ，此时，参数列表的小括号是可以省略不写的。
  - 注意事项:
    - 只有当参数的数量是一个的时候， 多了、少了都不能省略。
    - 省略掉小括号的同时， 必须要省略参数的类型。

```
// 有参、无返回值的方法实现
NoneReturnSingleParameter lambda2 = a -> {
    System.out.println("一个参数、无返回值方法的实现：参数是 " + a);
};
```

4.1.2.2.2. 方法体部分的精简

- 方法体大括号的精简
  - 当一个方法体中的逻辑， 有且只有一句的情况下， 大括号可以省略。

```
// 有参、无返回值的方法实现
NoneReturnSingleParameter lambda2 = a -> System.out.println("一个参数、无返回值方法的实现：参数是 " + a);
```

- return的精简
  - 如果一个方法中唯一的一条语句是一个返回语句， 此时在省略掉大括号的同时， 也必须省略掉return。

```
SingleReturnMutipleParameter lambda3 = (a, b) -> a + b;
```

4.1.3. 函数引用(了解)

lambda表达式是为了简化接口的实现的。 在lambda表达式中， 不应该出现比较复杂的逻辑。 如果在lambda表达式中出现了过于复杂的逻辑， 会对程序的可读性造成非常大的影响。 如果在lambda表达式中需要处理的逻辑比较复杂， 一般情况会单独的写一个方法。 在lambda表达式中直接引用这个方法即可。

> 或者， 在有些情况下， 我们需要在lambda表达式中实现的逻辑， 在另外一个地方已经写好了。 此时我们就不需要再单独写一遍， 只需要直接引用这个已经存在的方法即可。

函数引用： 引用一个已经存在的方法， 使其替代lambda表达式完成接口的实现。

4.1.3.1. 静态方法的引用

- **语法：**
  - 类::静态方法
- **注意事项:**
  - 在引用的方法后面， 不要添加小括号。
  - 引用的这个方法， 参数（数量、类型） 和 返回值， 必须要跟接口中定义的一致。
- **示例:**

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Date 2020/4/8
 * @Description
```

```
 */
interface SingleReturnMutipleParameter{
    void test(int a,int b);
}

public class Syntax1 {
    // 静态方法的引用
    public static void main(String[] args) {
        // 实现一个多个参数的、一个返回值的接口
        // 对一个静态方法的引用
        // 类::静态方法
        SingleReturnMutipleParameter lambda11 = (a,b)-> Calculator.calculate(a,b);
        //简化后
        SingleReturnMutipleParameter lambda1 = Calculator::calculate;
        System.out.println(lambda1.test(10, 20));
    }

    private static class Calculator {
        public static int calculate(int a, int b) {
            // 稍微复杂的逻辑：计算a和b的差值的绝对值
            if (a > b) {
                return a - b;
            }
            return b - a;
        }
    }
}
```

课上练习

```
//* 1.引用类方法
```

```
@FunctionalInterface
interface Converter{
    //将字符串转换成整数
    Integer convert(String value);
}
class Test1{
    public static void fun1() {
        //原来的方法
        Converter converter = value->Integer.valueOf(value);
        Integer v1 = converter.convert("222");
        System.out.println(v1);

        //简化
        //引用类方法
        //通过::实现,这里会自动将lambda表达式方法的参数全部传递给当前的方法
        Converter converter2 = Integer::valueOf;
        Integer v2 = converter2.convert("333");
        System.out.println(v2);
    }
}
```

4.1.3.2. 非静态方法的引用

- **语法:**
  - 对象::非静态方法
- **注意事项:**
  - 在引用的方法后面， 不要添加小括号。
  - 引用的这个方法， 参数（数量、类型） 和 返回值， 必须要跟接口中定义的一致。

- 示例:

```java
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Date 2020/4/8
 * @Description
 */
public class Syntax2 {
    public static void main(String[] args) {
        // 对非静态方法的引用，需要使用对象来完成
        SingleReturnMutipleParameter1 lambda1 = (a,b)->new Calculator().calculate(a,b);
        //简化
        SingleReturnMutipleParameter lambda = new Calculator()::calculate;
        System.out.println(lambda.test(10, 30));
    }

    private static class Calculator {
        public int calculate(int a, int b) {
            return a > b ? a - b : b - a;
        }
    }
}
```

课上练习

```java
//* 2.引用特定对象的实例方法
interface IA{
    public void show(String message);
}
class A{
    public void play(String i) {
```

```
        System.out.println("这里是A的方法play"+"  i:"+i);
    }
}
class Test2{
    public static void fun2() {
        //原来
        IA ia = message->new A().play(message);
        ia.show("hello");
        //简化
        IA ia2 = new A()::play;
        ia2.show("world");
    }
}

```

4.1.3.3. 构造方法的引用

- **使用场景**
  - 如果某一个函数式接口中定义的方法， 仅仅是为了得到一个类的对象。 此时我们就可以使用构造方法的引用， 简化这个方法的实现。
- **语法**
  - 类名::new
- **注意事项**
  - 可以通过接口中的方法的参数， 区分引用不同的构造方法。
- **示例**

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Date 2020/4/8
```

```
 * @Description
 */
public class Syntax3 {
    private static class Person {
        String name;
        int age;
        public Person() {
            System.out.println("一个Person对象被实例化了");
        }
        public Person(String name, int age) {
            System.out.println("一个Person对象被有参的实例化了");
            this.name = name;
            this.age = age;
        }
    }

    @FunctionalInterface
    private interface GetPerson {
        // 仅仅是希望获取到一个Person对象作为返回值
        Person test();
    }

    private interface GetPersonWithParameter {
        Person test(String name, int age);
    }

    public static void main(String[] args) {
        // lambda表达式实现接口
        GetPerson lambda = Person::new;     // 引用到Person类中的无参构造方法，获取到一个Person对象
        Person person = lambda.test();
```

```
35
36          GetPersonWithParameter lambda2 =
    Person::new;  // 引用到Person类中的有参构造方法，获取到
    一个Person对象
37          lambda2.test("xiaoming", 1);
38      }
39  }
```

4.1.3.4. 对象方法的特殊引用

如果在使用lambda表达式，实现某些接口的时候。 lambda表达式中包含了某一个对象， 此时方法体中， 直接使用这个对象调用它的某一个方法就可以完成整体的逻辑。 其他的参数， 可以作为调用方法的参数。 此时，可以对这种实现进行简化。

```
1  /**
2   * @Author 千锋大数据教学团队
3   * @Company 千锋好程序员大数据
4   * @Date 2020/4/8
5   * @Description
6   */
7  public class Syntax {
8      public static void main(String[] args) {
9          // 如果对于这个方法的实现逻辑，是为了获取到对象的名字
10         GetField field = person -> person.getName();
11         // 对于对象方法的特殊引用
12         GetField field = Person::getName;
13
14         // 如果对于这个方法的实现逻辑，是为了给对象的某些属性进
    行赋值
15         SetField lambda = (person, name) ->
    person.setName(name);
```

```
        SetField lambda = Person::setName;

        // 如果对于这个方法的实现逻辑，正好是参数对象的某一个方法
        ShowTest lambda2 = person -> person.show();
        ShowTest lambda2 = Person::show;
    }
}

interface ShowTest {
    void test(Person person);
}

interface SetField {
    void set(Person person, String name);
}

interface GetField {
    String get(Person person);
}

class Person {
    private String name;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void show() {
```

```
48
49      }
50  }
```

4.1.4. Lambda表达式需要注意的问题

这里类似于局部内部类、匿名内部类，依然存在闭包的问题。

如果在lambda表达式中，使用到了局部变量，那么这个局部变量会被隐式的声明为 final。 是一个常量， 不能修改值。

```
1  interface IB{
2      String subString(String string, int stat, int end);
3  }
4  class Outer{
5
6      public  void  show(){
7          int age = 0;
8          class  Inner{
9
10         }
11
12         IB ib = (string,stat,end)->{
13             //这里类似于局部内部类、匿名内部类，依然存在闭包的
   问题。
14             //
15             //如果在lambda表达式中，使用到了局部变量，那么这个
   局部变量会被隐式的声明为 final。 是一个常量， 不能修改值。
16             //age = 5;
17             System.out.println(age);
18             string.substring(stat,end);
19             return "";
```

```
        };
    }
}
```

4.1.5. Lambda表达式的实例

4.1.5.1. 线程的实例化

```
Thread thread = new Thread(() -> {
    // 线程中的处理
});
```

4.1.5.2. 集合的常见方法

```
ArrayList<String> list = new ArrayList<>();
Collections.addAll(list, "千锋", "大数据", "好程序员", "严选", "高薪");

// 按照条件进行删除
list.removeIf(ele -> ele.endsWith(".m"));
// 批量替换
list.replaceAll(ele -> ele.concat("!"));
// 自定义排序
list.sort((e1, e2) -> e2.compareTo(e1));
// 遍历
list.forEach(System.out::println);
```

4.1.5.3. 集合的流式编程(见附件文档)

```
ArrayList<String> list = new ArrayList<>();
Collections.addAll(list, "千锋", "大数据", "好程序员", "严选", "高薪");

list.parallelStream().filter(ele -> ele.length() > 2).forEach(System.out::println);
```

4.2 集合框架(01)(会)

4.2.1 Collection集合

4.2.1.1. 存储特点

Collection接口是单列集合的顶级接口。 在这种集合中存储的数据， 只占一列。 所有的元素， 直接存储于各种数据结构中。

Collection集合中， 没有下标的概念。

4.2.1.2. Collection API

- 接口方法

由于这个接口是单列集合的顶级接口， 在这里定义的所有的方法， 在所有的实现类中都是可以使用的。

修饰&返回值	方法	描述
boolean	**add(E element)**	将一个指定类型的元素， 添加到集合的末尾。

boolean	addAll(Collection coll)	批量添加，将一个集合中的所有的元素，添加到当前集合的末尾。
boolean	**remove(E ele)**	删除集合中指定的元素。
boolean	removeAll(Collection coll)	删除集合中所有的在参数集合中存在的元素（删除交集）。
boolean	retainAll(Collection coll)	保留集合中所有的在参数集合中存在的元素，删除其他元素（保留交集）。
void	**clear()**	清空集合中的所有的数据。
boolean	**removeIf(Predicate predicate)**	删除集合中满足指定条件的数据。
boolean	**contains(E ele)**	判断一个集合中是否包含指定的元素。
boolean	containsAll(Collection coll)	判断参数集合中， 是否所有的元素都在当前集合中包含。
int	**size()**	获取集合中元素的数量， 类似于数组 length。
boolean	isEmpty()	判断一个集合是否是空集合。
Object[]	toArray()	将集合中的元素，转成 Object 数组。
T[]	toArray(T[] arr)	将集合中的元素，转成指定类型的数组。

- 示例代码

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

/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description Collection集合的常用的方法
 */
public class Test {
    public static void main(String[] args) {
        // 1. 实例化一个Collection接口的实现类对象，并向上转型为接口类型
        Collection<String> collection = new ArrayList<>();
        Collection<String> temp = new ArrayList<>();

        // 2. 添加一个元素到集合的末尾
        //    返回值是boolean类型的
        //    如果本次添加操作，对集合中的数据造成影响了。这个数据的确加到集合中了，会返回true。
        //    如果本次添加操作，对集合中的数据没有造成影响，此时会返回false。(Set集合，因为Set集合是排重的集合，有时候添加会失败)
        collection.add("lucy");
        collection.add("lily");
        collection.add("polly");
        temp.add("Uncle wang");
        temp.add("polly");
        temp.add("lily");
        temp.add("Li Lei");
```

```
27
28          // 3. 批量添加，将一个集合中的所有的元素，添加到当前集合的末尾
29          //     返回值boolean
30          //     如果本次添加操作，对集合中的数据造成影响了。这个数据的确加到集合中了，会返回true。
31          //     如果本次添加操作，对集合中的数据没有造成影响，此时会返回false。(Set集合，因为Set集合是排重的集合，有时候添加会失败)
32          collection.addAll(temp);
33
34          // 4. 删除集合中从前往后第一个匹配到的元素
35          //     返回值boolean
36          //     本次删除操作，是否真的删除掉了某些元素。
37          //     如果删除掉了，本次操作对集合中的数据造成影响了，返回true。
38          //     否则，返回false。(要删除的这个元素在集合中不存在)
39          // collection.remove("polly");
40
41          // 5. 批量删除。删除所有的在参数集合中存在的元素
42          //     依次判断当前集合中的每一个元素，是否在参数集合中。
43          //     如果存在，就删除；如果不存在，就不删除。
44          //     返回值boolean
45          //     本次操作，如果成功删除掉数据了，对集合中的数据造成影响了，返回true。
46          //     否则就返回false。
47          // collection.removeAll(temp);
48
49          // 6. 保留当前集合中，存在于参数集合中的数据，删除其他数据。
```

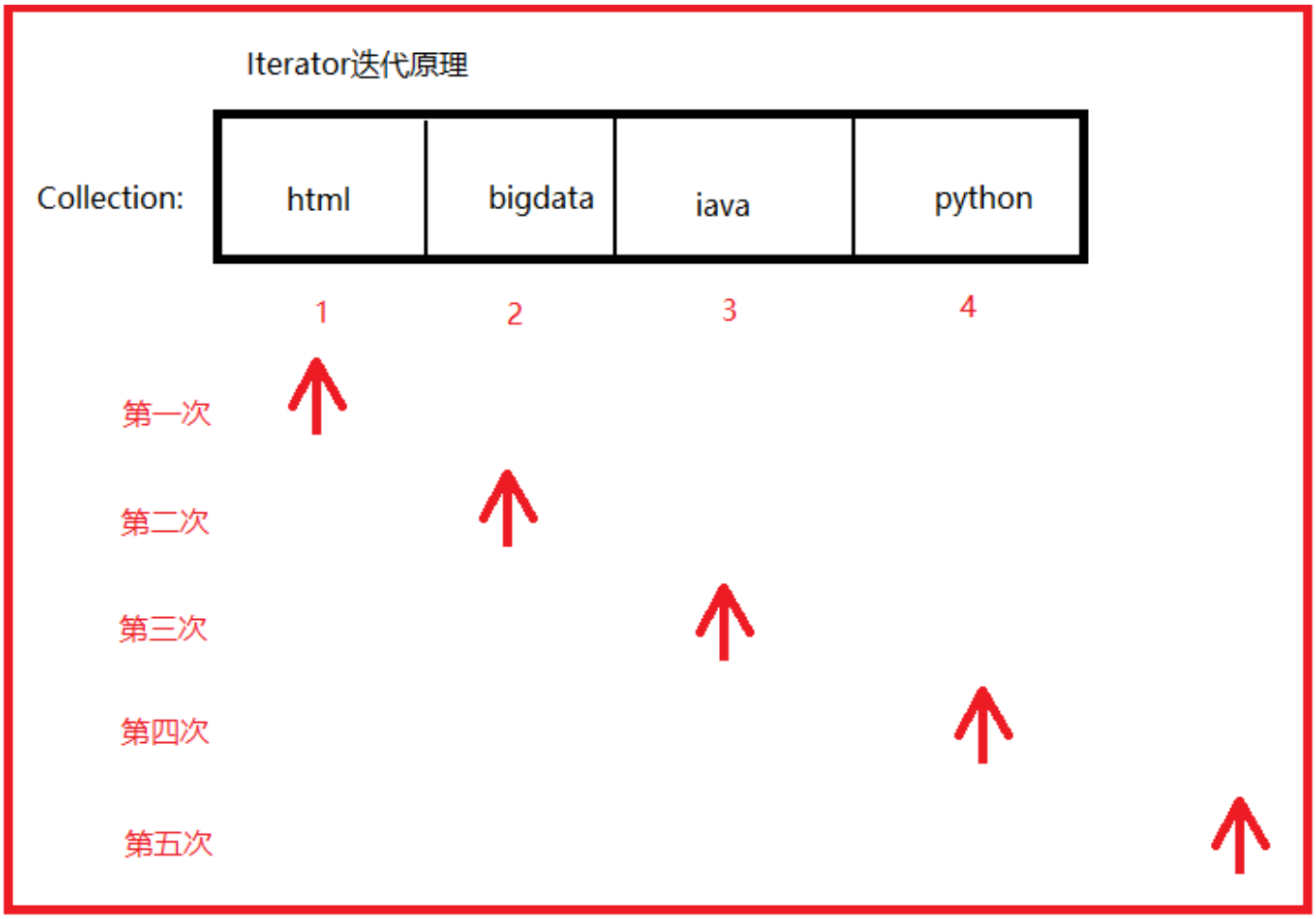
```
50          //    依次判断当前集合中的每一个元素，是否在参数集合中。
51          //    如果存在于参数集合中，就保留这个元素。
52          //    如果不存在于参数集合中，就删除这个元素。
53          //    返回值boolean
54          //    本次操作，如果成功删除掉数据了，对集合中的数据造成影响了，返回true。
55          //    否则就返回false。
56          // collection.retainAll(temp);
57
58          // 7. 清空集合中的所有的数据
59          // collection.clear();
60
61          // 8. 按照条件删除元素
62          //    依次将集合中的每一个元素，带入到参数Predicate接口的方法中，作为接口方法的参数
63          //    如果Predicate接口的方法返回值为true，就删除这个元素
64          //    如果Predicate接口的方法返回值为false，就保留这个元素
65          //    返回值boolean
66          //    本次操作，如果成功删除掉数据了，对集合中的数据造成影响了，返回true。
67          //    否则就返回false。
68          // collection.removeIf(ele -> ele.matches("[lL].*"));
69
70          // 9. 判断一个集合中是否包含指定的元素
71          boolean ret1 = collection.contains("lily");
72          System.out.println(ret1);
73
74          // 10. 判断是否参数集合中的所有元素都在当前集合中包含
```

```
75          //      只有当参数集合中的每一个元素都在当前集合中包含，
    才会返回true。
76          //      但凡参数集合中有任意的元素没有在当前的集合中包
    含，返回值都是false。
77          boolean ret2 = collection.containsAll(temp);
78          System.out.println(ret2);
79
80          // 11. 判断当前的集合中有多少元素，类似于数组的长度
81          int size = collection.size();
82          System.out.println(size);
83
84          // 12. 判断当前集合，是否是空集合
85          boolean ret3 = collection.isEmpty();
86          System.out.println(ret3);
87
88          // 13. 将集合中的元素转成Object数组
89          Object[] array = collection.toArray();
90
91          // 14. 将集合中的元素转成指定类型的数组
92          String[] ret4 = collection.toArray(new
    String[0]);
93          System.out.println(Arrays.toString(ret4));
94
95          System.out.println(collection);
96      }
97  }
```

4.2.1.3. Collection集合遍历

4.2.1.3.1. 迭代器（Iterator)

工作原理



- 使用集合的 iterator() 方法，获取到一个迭代器对象。

```
原因:迭代器在对集合进行遍历时,要与对应的集合产生关系,通过集合的一个方法获取迭代器对象,可以在方法内部创建迭代器对象,同时自动与当前集合产生关系.对于使用者来说简化了代码
```

- 这个迭代器对象， 内部维护了一个引用， 指向集合中的某一个元素。默认指向-1位。
- 使用next()方法， 向后移动一位元素指向， 并返回新的指向的元素。
- 如果使用next()方法， 向后移动指向的时候， 已经超出了集合的范围，指向了一个不存在的元素， 会抛出异常 NoSuchElenmentException。

- 一般情况下， 配合 hasNext() 方法一起使用的。

```java
private static void enumunation2(Collection<String> collection) {
    // 1. 获取到一个迭代器对象
    Iterator<String> iterator = collection.iterator();

    // 2. 通过hasNext判断是否还有下一个元素
    while (iterator.hasNext()) {
        // 3. 向后指向，并返回这个新的指向的元素
        String ele = iterator.next();
        System.out.println(ele);
    }
          //遍历取值--通过hasNext判断是否还有下一个元素
        while(iterator.hasNext()){
            Object o = iterator.next();
            //注意点1:当集合中同时存在不同类型的数据时,需要进行容错处理和向下转型.
            //容错处理
            if (!(o instanceof String)){
                throw new ClassCastException("类型错误");
            }
            //向下转型
            String s = (String)o;
            System.out.println(s.length());
        }

        //再继续遍历
        //注意点2:直接再次使用第一次的iterator进行遍历,遍历失败.因为当前的指针已经指向了集合的最后.
        //再次使用hasnext会直接返回false.所以如果想再次遍历,要重新获取迭代器对象.
```

```
27          while (iterator.hasNext()){
28
29          }
30  }
```

注意事项

在使用迭代器进行元素的遍历过程中，不要使用集合的方法修改集合中的内容 !!! 否则，会出现 ConcurrentModificationException 。可以使用迭代器自带的remove()方法操作.

对出现ConcurrentModificationException异常的解释(扩展内容)

```
1  Fail-Fast机制：
2
3      我们知道java.util.ArrayList不是线程安全的，因此如果在使用迭代器的过程中有其他线程修改了list，那么将抛出ConcurrentModificationException，这就是所谓fail-fast策略。
4      这一策略在源码中的实现是通过modCount域，modCount顾名思义就是修改次数，对ArrayList内容的修改都将增加这个值，那么在迭代器初始化过程中会将这个值赋给迭代器的expectedModCount。
5  [java] view plain copy
6  1.HashIterator() {
7  2.     expectedModCount = modCount;
8  3.     if (size > 0) { // advance to first entry
9  4.         Entry[] t = table;
10 5.         while (index < t.length && (next = t[index++]) == null)
11 6.             ;
12 7.     }
13 8.}
14 在迭代过程中，判断modCount跟expectedModCount是否相等，如果不相等就表示已经有其他线程修改了list：
15     注意到modCount声明为volatile，保证线程之间修改的可见性。
```

```
[java] view plain copy
1.final Entry<K,V> nextEntry() {
2.    if (modCount != expectedModCount)
3.        throw new ConcurrentModificationException();
```

在HashMap的API中指出：

由所有ArrayList类的“collection 视图方法”所返回的迭代器都是快速失败的：在迭代器创建之后，如果从结构上对映射进行修改，除非通过迭代器本身的 remove 方法，其他任何时间任何方式的修改，迭代器都将抛出 ConcurrentModificationException。因此，面对并发的修改，迭代器很快就会完全失败，而不冒在将来不确定的时间发生任意不确定行为的风险。

注意，迭代器的快速失败行为不能得到保证，一般来说，存在非同步的并发修改时，不可能作出任何坚决的保证。快速失败迭代器尽最大努力抛出ConcurrentModificationException。因此，编写依赖于此异常的程序的做法是错误的，正确做法是：迭代器的快速失败行为应该仅用于检测程序错误。

4.2.1.3.2. 增强for循环

```java
// 遍历参数集合中的元素
public static void enumuration(Collection<String>
collection) {
    // 增强for循环
    for (String element : collection) {
        System.out.println(element);
    }
}
```

注意事项

在使用增强for循环进行元素的遍历过程中，不要修改集合中的内容 !!! 否则， 会出现 ConcurrentModificationException 。

原因:增强for循环内部实际调用的是一个迭代器

4.2.1.3.3. forEach

Java8之后，在Collection集合中添加了一个新的方法 forEach。

方法原型:

```
default void forEach(Consumer<? super T> action)
```

方法逻辑:

将集合中的每一个元素，带入到参数Consumer函数式接口的方法中，作为参数。在调用这个方法的时候，可以自定义实现的方式。

示例代码:

```
private static void enumunation3(Collection<String>
collection) {
    collection.forEach(System.out::println);
}
```

课上练习

有整型数组，内部元素为1到10，将数组中的元素倒序插入ArrayList中，并遍历输出结果。

```
public static void main(String[] args) {
    int[] a = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
};
    Collection con = new ArrayList<>();
    for (int i = a.length - 1; i >= 0; i--) {
      con.add(a[i]);
    }

     Iterator i = con.iterator();
     while (i.hasNext()){
         System.out.println(i.next());
     }
  }

```

4.2.2. List集合

4.2.2.1. 存储特点

List集合是单列集合，是Collection接口的子接口。Collection接口中所有的方法，这里都有。同时，这个集合比Collection集合，多个若干方法。

在List接口中，是有下标的概念的。多出来的这些方法，基本也都是围绕着下标操作的。

4.2.2.2 List与Set对比

- List:存储的数据是有序的(元素的存储顺序与添加元素的顺序一致),可以重复的.
- Set:存储的数据是无序的,不可以重复

4.2.2.3. List API

修饰&返回	方法	描述
void	add(int index, E element)	向集合中的指定的下标位插入一个元素。
void	addAll(int index, Collection coll)	在集合中指定的下标位插入另外一个集合中所有的数据。
E	remove(int index)	删除集合中指定下标位的元素。
E	set(int index, E element)	修改指定下标位的值。
E	get(int index)	获取指定下标位的元素。
int	indexOf(E element)	获取集合中的某一个元素第一次出现的下标。
int	lastIndexOf（E element)	获取集合中的某一个元素最后一次出现的下标。
List	subList(int fromIndex, int toIndex)	从一个集合中，截取一部分，作为子集合。 [from, to)。
void	replaceAll(UnaryOperator operator)	将集合中的元素带入到接口方法中， 用返回值替换原来的元素。
void	sort(Comparator comparator)	将集合中的元素进行升序排序。

4.2.2.4. 示例代码

```java
import java.util.ArrayList;
import java.util.List;

/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description List接口的方法
 */
public class Test1 {
    public static void main(String[] args) {
        // 1. 实例化一个ArrayList对象，向上转型为接口类型。
        List<String> list = new ArrayList<>();

        // 2. 增元素
        list.add("Lily");
        list.add("Lucy");
        list.add("Polly");
        list.add("Jim");

        // 3. 在集合中指定的下标位插入元素
        list.add(2, "Tom");

        // 4. 在集合中指定的下标位插入另外一个集合中所有的数据
        list.addAll(2, list);

        // 5. 删除集合中指定下标位的元素
        //    返回值：这个被删除的元素
        System.out.println(list.remove(2));

        // 6. 修改指定下标位的值
        //    返回被覆盖的值。
```

```
32          System.out.println(list.set(2, "AAA"));
33
34          // 7* 元素替换
35          //      将集合中的每一个元素，带入到接口的方法中，用返回值替换原来的元素
36          // list.replaceAll(ele -> ele.concat(".txt"));
37
38          // 8. 获取指定下标位的元素。
39          System.out.println(list.get(2));
40
41          // 9. 获取集合中的某一个元素第一次出现的下标
42          System.out.println(list.indexOf("Jim"));
43
44          // 10. 获取集合中的某一个元素最后一次出现的下标
45          System.out.println(list.lastIndexOf("Jim"));
46
47          // 11. 从一个集合中，截取一部分，作为子集合。 [from, to)
48          List<String> sub = list.subList(2, 60);
49
50          System.out.println(list);
51          System.out.println(sub);
52      }
53  }
```

4.2.2.5. List集合排序

在List接口中，提供了一个排序的方法 sort

方法原型

```
1  default void sort(Comparator<? super E> c)
```

方法逻辑

这是一个系统封装好的一个用来做排序的方法，由于集合中只能存储引用数据类型的数据，因此，需要明确两个元素的大小关系。参数Comparator是一个对象大小比较的接口。在这个接口的方法中，有两个参数，分别表示参与比较的两个对象。返回值是int类型，不看具体的值，只看正负。

```
/**
 * 两个对象的大小比较方法
 * @param o1 参与比较的一个对象
 * @param o2 参与比较的另一个对象
 * @return 比较的结果
 *     > 0 : o1 > o2
 *      == 0 : o1 == o2
 *     < 0 : o1 < o2
 */
int compare(T o1, T o2)
```

示例代码

```
import java.util.ArrayList;
import java.util.List;

/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 集合的元素排序
 */
public class Test2 {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();

```

```
        // 2. 增元素
        list.add("Lily");
        list.add("Lucy");
        list.add("Polly");
        list.add("Jim");

        // 排序：按照两个字符串的长度进行大小比较，升序排序
        list.sort((e1, e2) -> e1.length() - e2.length());
    }
}
```

4.2.2.6. List集合遍历

由于List接口是继承自Collection接口的，因此在Collection部分的三种遍历方式，都可以用来遍历List集合。同时，在List集合中，还添加了两种用来遍历集合的其他方式。

- 下标遍历

顾名思义， 类似于数组的下标遍历。 遍历集合中的所有的下标， 依次获取指定下标位的元素。

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.ListIterator;

/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description List集合遍历
```

```
 */
public class List1 {
    public static void main(String[] args) {
        // 1. 实例化一个List集合，存储若干数据
        List<Integer> list = new ArrayList<>();
        Collections.addAll(list, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100);
        // 2. 遍历
        index(list);
    }
    /**
     * 下标遍历法
     * @param list 需要遍历的集合
     */
    private static void index(List<Integer> list) {
        for (int i = 0; i < list.size(); i++) {
            // 获取每一个元素
            System.out.println(list.get(i));
        }
    }
}
```

- 列表迭代器

这种方式， 类似于迭代器。 在List集合中， 有一个方法 `listIterator()`， 可以获取到一个 `ListIterator` 接口的引用。 而`ListIterator`是`Iterator`的子接口。 因此在保留了传统的迭代器的迭代方法的基础上， 还添加了若干个其他的方法。

`ListIterator` 中新增了hasPrevious()和previous()方法

当我们使用hasnext()和next()方法实现从左到右遍历后,可以继续使用使用hasPrevious()和previous()方法从右到左遍历.

> 使用ListIterator， 在遍历集合中的元素的同时， 可以向集合中添加元素、删除元素、修改元素。
>
> 但是， 这里对集合中的元素操作， 并不是使用 List 接口中的方法， 而是用 ListIterator 接口中的方法完成。

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.ListIterator;

/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description List集合遍历
 */
public class List1 {
    public static void main(String[] args) {
        // 1. 实例化一个List集合，存储若干数据
        List<Integer> list = new ArrayList<>();
        Collections.addAll(list, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100);
        // 2. 遍历
        listIterator(list);
    }
    /**
     * 使用列表迭代器完成遍历
     * @param list 需要遍历的集合
     */
    private static void listIterator(List<Integer> list) {
```

```
24          // 1. 获取到 ListIterator 对象
25          ListIterator<Integer> iterator =
list.listIterator(4);
26          // 2. 循环遍历-- 从左到右遍历
27          while (iterator.hasNext()) {
28              // 2.1. 获取迭代器当前指向的元素
29              Integer ele = iterator.next();
30              System.out.println(ele);
31              // 2.2. 元素操作
32              if (ele == 30) {
33                  // 添加：在迭代器当前指向的元素的下一位插入一
个元素
34                  // 新增的元素，没有存在于迭代列表中，迭代器会
直接指向原集合中的下一个元素
35                  // iterator.add(300);
36                  // 删除：删除迭代器当前指向的这一位元素
37                  // iterator.remove();
38                  // 修改：修改迭代器当前指向的这一位元素
39                  // 注意:remove,add,set不要同时使用
40                  // iterator.set(300);
41              }
42          }
43          System.out.println(list);
44
45          //继续从右到左遍历
46          while (iterator.hasPrevious()){
47              Object o = iterator.previous();
48              System.out.println("从右到左:"+o);
49          }
50      }
51  }
```

课上练习

有某个字符串集合，长度为5，给定字母a，统计集合中的字符串元素包含字母a的个数。

```
public static void main(String[] args) {
    List list = new ArrayList<>();
    list.add("abc");
    list.add("acd");
    list.add("xyz");
    list.add("bbq");
    list.add("jack");
    int count = 0;
    for (Object string : list) {
      if (string.toString().contains("a")) {
        count++;
      }
    }
    System.out.println(count);
  }
```