1.不可变集合

1.1 什么是不可变集合

是一个长度不可变，内容也无法修改的集合

1.2 使用场景

如果某个数据不能被修改，把它防御性地拷贝到不可变集合中是个很好的实践。

当集合对象被不可信的库调用时，不可变形式是安全的。

简单理解：

不想让别人修改集合中的内容

比如说：

1，斗地主的54张牌，是不能添加，不能删除，不能修改的

2，斗地主的打牌规则：单张，对子，三张，顺子等，也是不能修改的

3，用代码获取的操作系统硬件信息，也是不能被修改的

1.3 不可变集合分类

- 不可变的list集合
- 不可变的set集合
- 不可变的map集合

1.4 不可变的list集合

```
public class ImmutableDemo1 {
    public static void main(String[] args) {
        /*
            创建不可变的List集合
            "张三", "李四", "王五", "赵六"
        */
```

```
7
8           //一旦创建完毕之后，是无法进行修改的，在下面的代码中，只能进行查询操
    作
9           List<String> list = List.of("张三", "李四", "王五", "赵
    六");
10
11          System.out.println(list.get(0));
12          System.out.println(list.get(1));
13          System.out.println(list.get(2));
14          System.out.println(list.get(3));
15
16          System.out.println("----------------------------");
17
18          for (String s : list) {
19              System.out.println(s);
20          }
21
22          System.out.println("----------------------------");
23
24
25          Iterator<String> it = list.iterator();
26          while(it.hasNext()){
27              String s = it.next();
28              System.out.println(s);
29          }
30          System.out.println("----------------------------");
31
32          for (int i = 0; i < list.size(); i++) {
33              String s = list.get(i);
34              System.out.println(s);
35          }
36          System.out.println("----------------------------");
37
38          //list.remove("李四");
39          //list.add("aaa");
40          list.set(0,"aaa");
41      }
42  }
```

1.5 不可变的Set集合

```java
public class ImmutableDemo2 {
    public static void main(String[] args) {
        /*
            创建不可变的Set集合
            "张三", "李四", "王五", "赵六"


            细节:
                  当我们要获取一个不可变的Set集合时，里面的参数一定要保证唯一性
        */

        //一旦创建完毕之后，是无法进行修改的，在下面的代码中，只能进行查询操作
        Set<String> set = Set.of("张三", "张三", "李四", "王五", "赵六");

        for (String s : set) {
            System.out.println(s);
        }

        System.out.println("-----------------------");

        Iterator<String> it = set.iterator();
        while(it.hasNext()){
            String s = it.next();
            System.out.println(s);
        }

        System.out.println("-----------------------");
        //set.remove("王五");
    }
}
```

1.6 不可变的Map集合

1.6.1：键值对个数小于等于10

```
public class ImmutableDemo3 {
    public static void main(String[] args) {
        /*
         创建Map的不可变集合
             细节1：
                 键是不能重复的
             细节2：
                 Map里面的of方法，参数是有上限的，最多只能传递20个参数，10个键值对
             细节3：
                 如果我们要传递多个键值对对象，数量大于10个，在Map接口中还有一个方法
         */

        //一旦创建完毕之后，是无法进行修改的，在下面的代码中，只能进行查询操作
        Map<String, String> map = Map.of("张三", "南京", "张三", "北京", "王五", "上海",
                "赵六", "广州", "孙七", "深圳", "周八", "杭州",
                "吴九", "宁波", "郑十", "苏州", "刘一", "无锡",
                "陈二", "嘉兴");

        Set<String> keys = map.keySet();
        for (String key : keys) {
            String value = map.get(key);
            System.out.println(key + "=" + value);
        }

        System.out.println("--------------------------");

        Set<Map.Entry<String, String>> entries = map.entrySet();
        for (Map.Entry<String, String> entry : entries) {
            String key = entry.getKey();
            String value = entry.getValue();
            System.out.println(key + "=" + value);
        }
        System.out.println("--------------------------");
```

```
    }
}
```

1.6.2：键值对个数大于10

```
public class ImmutableDemo4 {
    public static void main(String[] args) {

        /*
            创建Map的不可变集合,键值对的数量超过10个
        */

        //1.创建一个普通的Map集合
        HashMap<String, String> hm = new HashMap<>();
        hm.put("张三", "南京");
        hm.put("李四", "北京");
        hm.put("王五", "上海");
        hm.put("赵六", "北京");
        hm.put("孙七", "深圳");
        hm.put("周八", "杭州");
        hm.put("吴九", "宁波");
        hm.put("郑十", "苏州");
        hm.put("刘一", "无锡");
        hm.put("陈二", "嘉兴");
        hm.put("aaa", "111");

        //2.利用上面的数据来获取一个不可变的集合
/*
        //获取到所有的键值对对象（Entry对象）
        Set<Map.Entry<String, String>> entries = hm.entrySet();
        //把entries变成一个数组
        Map.Entry[] arr1 = new Map.Entry[0];
        //toArray方法在底层会比较集合的长度跟数组的长度两者的大小
        //如果集合的长度 > 数组的长度 ：数据在数组中放不下，此时会根据实际数据的个数，重新创建数组
        //如果集合的长度 <= 数组的长度：数据在数组中放的下，此时不会创建新的数组，而是直接用
        Map.Entry[] arr2 = entries.toArray(arr1);
        //不可变的map集合
        Map map = Map.ofEntries(arr2);
        map.put("bbb","222");*/
```

```
35
36
37          //Map<Object, Object> map =
    Map.ofEntries(hm.entrySet().toArray(new Map.Entry[0]));
38
39          Map<String, String> map = Map.copyOf(hm);
40          map.put("bbb","222");
41      }
42  }
```

2.Stream流

2.1体验Stream流【理解】

- 案例需求

  按照下面的要求完成集合的创建和遍历

  - 创建一个集合，存储多个字符串元素
  - 把集合中所有以"张"开头的元素存储到一个新的集合
  - 把"张"开头的集合中的长度为3的元素存储到一个新的集合
  - 遍历上一步得到的集合

- 原始方式示例代码

```
1  public class MyStream1 {
2      public static void main(String[] args) {
3          //集合的批量添加
4          ArrayList<String> list1 = new ArrayList<>
    (List.of("张三丰","张无忌","张翠山","王二麻子","张良","谢广
    坤"));
5          //list.add()
6
7          //遍历list1把以张开头的元素添加到list2中。
8          ArrayList<String> list2 = new ArrayList<>();
9          for (String s : list1) {
10             if(s.startsWith("张")){
11                 list2.add(s);
12             }
13         }
14         //遍历list2集合，把其中长度为3的元素，再添加到list3中。
15         ArrayList<String> list3 = new ArrayList<>();
```

```
        for (String s : list2) {
            if(s.length() == 3){
                list3.add(s);
            }
        }
        for (String s : list3) {
            System.out.println(s);
        }
    }
}
```

- 使用Stream流示例代码

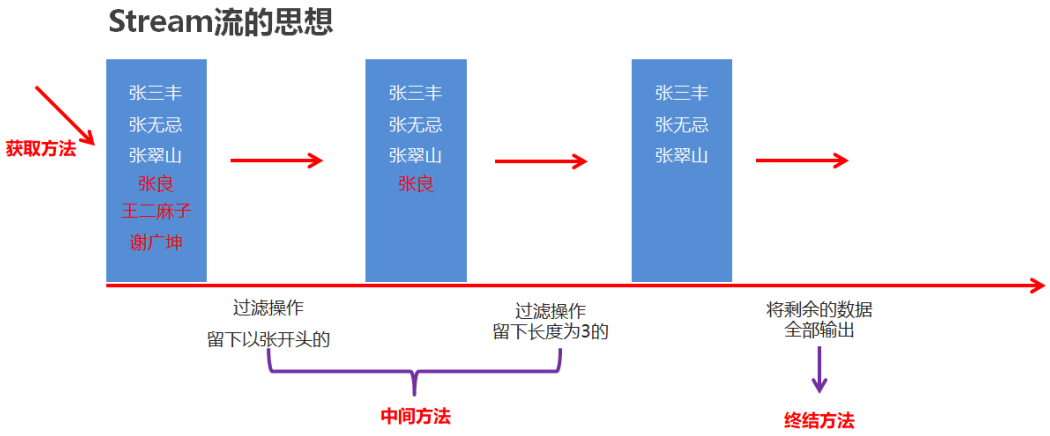
```
public class StreamDemo {
    public static void main(String[] args) {
        //集合的批量添加
        ArrayList<String> list1 = new ArrayList<>(List.of("张三丰","张无忌","张翠山","王二麻子","张良","谢广坤"));

        //Stream流
        list1.stream().filter(s->s.startsWith("张"))
                .filter(s->s.length() == 3)
                .forEach(s-> System.out.println(s));
    }
}
```

- Stream流的好处
    - 直接阅读代码的字面意思即可完美展示无关逻辑方式的语义：获取流、过滤姓张、过滤长度为3、逐一打印
    - Stream流把真正的函数式编程风格引入到Java中
    - 代码简洁

2.2Stream流的常见生成方式【应用】

- Stream流的思想



- Stream流的三类方法
    - 获取Stream流
        - 创建一条流水线,并把数据放到流水线上准备进行操作
    - 中间方法
        - 流水线上的操作
        - 一次操作完毕之后,还可以继续进行其他操作
    - 终结方法
        - 一个Stream流只能有一个终结方法
        - 是流水线上的最后一个操作
- 生成Stream流的方式
    - Collection体系集合

      使用默认方法stream()生成流， default Stream stream()
    - Map体系集合

      把Map转成Set集合，间接的生成流
    - 数组

      通过Arrays中的静态方法stream生成流
    - 同种数据类型的多个数据

      通过Stream接口的静态方法of(T... values)生成流
- 代码演示

```
public class StreamDemo {
    public static void main(String[] args) {
        //Collection体系的集合可以使用默认方法stream()生成流
        List<String> list = new ArrayList<String>();
        Stream<String> listStream = list.stream();

        Set<String> set = new HashSet<String>();
```

```
        Stream<String> setStream = set.stream();

        //Map体系的集合间接的生成流
        Map<String,Integer> map = new HashMap<String,
Integer>();
        Stream<String> keyStream =
map.keySet().stream();
        Stream<Integer> valueStream =
map.values().stream();
        Stream<Map.Entry<String, Integer>> entryStream =
map.entrySet().stream();

        //数组可以通过Arrays中的静态方法stream生成流
        String[] strArray = {"hello","world","java"};
        Stream<String> strArrayStream =
Arrays.stream(strArray);

        //同种数据类型的多个数据可以通过Stream接口的静态方法
of(T... values)生成流
        Stream<String> strArrayStream2 =
Stream.of("hello", "world", "java");
        Stream<Integer> intStream = Stream.of(10, 20,
30);
    }
}
```

2.3Stream流中间操作方法【应用】

- 概念

  中间操作的意思是,执行完此方法之后,Stream流依然可以继续执行其他操作

- 常见方法

方法名	说明
Stream filter(Predicate predicate)	用于对流中的数据进行过滤
Stream limit(long maxSize)	返回此流中的元素组成的流，截取前指定参数个数的数据
Stream skip(long n)	跳过指定参数个数的数据，返回由该流的剩余元素组成的流

方法名	说明
static Stream concat(Stream a, Stream b)	合并a和b两个流为一个流
Stream distinct()	返回由该流的不同元素（根据Object.equals(Object) ）组成的流

- filter代码演示

```
public class MyStream3 {
    public static void main(String[] args) {
//        Stream<T> filter(Predicate predicate)：过滤
//        Predicate接口中的方法   boolean test(T t)：对给定的参数进行判断，返回一个布尔值

        ArrayList<String> list = new ArrayList<>();
        list.add("张三丰");
        list.add("张无忌");
        list.add("张翠山");
        list.add("王二麻子");
        list.add("张良");
        list.add("谢广坤");

        //filter方法获取流中的 每一个数据.
        //而test方法中的s,就依次表示流中的每一个数据.
        //我们只要在test方法中对s进行判断就可以了.
        //如果判断的结果为true,则当前的数据留下
        //如果判断的结果为false,则当前数据就不要.
//        list.stream().filter(
//                new Predicate<String>() {
//                    @Override
//                    public boolean test(String s) {
//                        boolean result = s.startsWith("张");
//                        return result;
//                    }
//                }
//        ).forEach(s-> System.out.println(s));

        //因为Predicate接口中只有一个抽象方法test
        //所以我们可以使用lambda表达式来简化
//        list.stream().filter(
//                (String s)->{
```

```
//                    boolean result =
s.startsWith("张");
//                        return result;
//                }
//        ).forEach(s-> System.out.println(s));

        list.stream().filter(s ->
>s.startsWith("张")).forEach(s-> System.out.println(s));

    }
}
```

- limit&skip代码演示

```
public class StreamDemo02 {
    public static void main(String[] args) {
        //创建一个集合，存储多个字符串元素
        ArrayList<String> list = new ArrayList<String>
();

        list.add("林青霞");
        list.add("张曼玉");
        list.add("王祖贤");
        list.add("柳岩");
        list.add("张敏");
        list.add("张无忌");

        //需求1：取前3个数据在控制台输出
        list.stream().limit(3).forEach(s->
System.out.println(s));
        System.out.println("--------");

        //需求2：跳过3个元素，把剩下的元素在控制台输出
        list.stream().skip(3).forEach(s->
System.out.println(s));
        System.out.println("--------");

        //需求3：跳过2个元素，把剩下的元素中前2个在控制台输出
        list.stream().skip(2).limit(2).forEach(s->
System.out.println(s));
    }
}
```

- concat&distinct代码演示

```
public class StreamDemo03 {
    public static void main(String[] args) {
        //创建一个集合，存储多个字符串元素
        ArrayList<String> list = new ArrayList<String>
();

        list.add("林青霞");
        list.add("张曼玉");
        list.add("王祖贤");
        list.add("柳岩");
        list.add("张敏");
        list.add("张无忌");

        //需求1：取前4个数据组成一个流
        Stream<String> s1 = list.stream().limit(4);

        //需求2：跳过2个数据组成一个流
        Stream<String> s2 = list.stream().skip(2);

        //需求3：合并需求1和需求2得到的流，并把结果在控制台输出
//        Stream.concat(s1,s2).forEach(s->
System.out.println(s));

        //需求4：合并需求1和需求2得到的流，并把结果在控制台输出，要
求字符串元素不能重复
        Stream.concat(s1,s2).distinct().forEach(s->
System.out.println(s));
    }
}
```

2.4Stream流终结操作方法【应用】

- 概念

  终结操作的意思是,执行完此方法之后,Stream流将不能再执行其他操作

- 常见方法

方法名	说明
void forEach(Consumer action)	对此流的每个元素执行操作
long count()	返回此流中的元素数

- 代码演示

```java
public class MyStream5 {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("张三丰");
        list.add("张无忌");
        list.add("张翠山");
        list.add("王二麻子");
        list.add("张良");
        list.add("谢广坤");

        //method1(list);

//        long count(): 返回此流中的元素数
        long count = list.stream().count();
        System.out.println(count);
    }

    private static void method1(ArrayList<String> list)
{
        //  void forEach(Consumer action): 对此流的每个元素
执行操作
        //  Consumer接口中的方法void accept(T t): 对给定的参数
执行此操作
        //在forEach方法的底层,会循环获取到流中的每一个数据.
        //并循环调用accept方法,并把每一个数据传递给accept方法
        //s就依次表示了流中的每一个数据.
        //所以,我们只要在accept方法中,写上处理的业务逻辑就可以了.
        list.stream().forEach(
                new Consumer<String>() {
                    @Override
                    public void accept(String s) {
                        System.out.println(s);
                    }
                }
        );

        System.out.println("====================");
        //lambda表达式的简化格式
        //是因为Consumer接口中,只有一个accept方法
        list.stream().forEach(
```

```
                (String s)->{
                    System.out.println(s);
                }
        );
        System.out.println("====================");
        //lambda表达式还是可以进一步简化的.
        list.stream().forEach(s->System.out.println(s));
    }
}
```

2.5Stream流的收集操作【应用】

- 概念

  对数据使用Stream流的方式操作完毕后,可以把流中的数据收集到集合中

- 常用方法

方法名	说明
R collect(Collector collector)	把结果收集到集合中

- 工具类Collectors提供了具体的收集方式

方法名	说明
public static Collector toList()	把元素收集到List集合中
public static Collector toSet()	把元素收集到Set集合中
public static Collector toMap(Function keyMapper,Function valueMapper)	把元素收集到Map集合中

- 代码演示

```
// toList和toSet方法演示
public class MyStream7 {
    public static void main(String[] args) {
        ArrayList<Integer> list1 = new ArrayList<>();
        for (int i = 1; i <= 10; i++) {
            list1.add(i);
        }

        list1.add(10);
        list1.add(10);
```

```
        list1.add(10);
        list1.add(10);
        list1.add(10);

        //filter负责过滤数据的.
        //collect负责收集数据.
                //获取流中剩余的数据,但是他不负责创建容器,也不负责把数据添加到容器中.
        //Collectors.toList() : 在底层会创建一个List集合.并把所有的数据添加到List集合中.
        List<Integer> list = list1.stream().filter(number -> number % 2 == 0)
                .collect(Collectors.toList());

        System.out.println(list);

    Set<Integer> set = list1.stream().filter(number -> number % 2 == 0)
            .collect(Collectors.toSet());
    System.out.println(set);
}
}
/**
Stream流的收集方法 toMap方法演示
创建一个ArrayList集合，并添加以下字符串。字符串中前面是姓名，后面是年龄
"zhangsan,23"
"lisi,24"
"wangwu,25"
保留年龄大于等于24岁的人，并将结果收集到Map集合中，姓名为键，年龄为值
*/
public class MyStream8 {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("zhangsan,23");
        list.add("lisi,24");
        list.add("wangwu,25");

        Map<String, Integer> map = list.stream().filter(
                s -> {
                    String[] split = s.split(",");
```

```
                    int age =
Integer.parseInt(split[1]);
                    return age >= 24;
                }

        //   collect方法只能获取到流中剩余的每一个数据.
        //在底层不能创建容器,也不能把数据添加到容器当中

        //Collectors.toMap 创建一个map集合并将数据添加到集合当
中

         // s 依次表示流中的每一个数据

         //第一个lambda表达式就是如何获取到Map中的键
         //第二个lambda表达式就是如何获取Map中的值
        ).collect(Collectors.toMap(
                s -> s.split(",")[0],
                s -> Integer.parseInt(s.split(",")[1])
));

        System.out.println(map);
    }
}
```

2.6Stream流综合练习【应用】

- 案例需求

  现在有两个ArrayList集合，分别存储6名男演员名称和6名女演员名称，要求完成如下的操作

  - 男演员只要名字为3个字的前三人
  - 女演员只要姓林的，并且不要第一个
  - 把过滤后的男演员姓名和女演员姓名合并到一起
  - 把上一步操作后的元素作为构造方法的参数创建演员对象,遍历数据

  演员类Actor已经提供，里面有一个成员变量，一个带参构造方法，以及成员变量对应的get/set方法

- 代码实现

  演员类

```
public class Actor {
    private String name;
```

```
3
4       public Actor(String name) {
5           this.name = name;
6       }
7
8       public String getName() {
9           return name;
10      }
11
12      public void setName(String name) {
13          this.name = name;
14      }
15  }
```

测试类

```
1   public class StreamTest {
2       public static void main(String[] args) {
3           //创建集合
4           ArrayList<String> manList = new
    ArrayList<String>();
5           manList.add("周润发");
6           manList.add("成龙");
7           manList.add("刘德华");
8           manList.add("吴京");
9           manList.add("周星驰");
10          manList.add("李连杰");
11
12          ArrayList<String> womanList = new
    ArrayList<String>();
13          womanList.add("林心如");
14          womanList.add("张曼玉");
15          womanList.add("林青霞");
16          womanList.add("柳岩");
17          womanList.add("林志玲");
18          womanList.add("王祖贤");
19
20          //男演员只要名字为3个字的前三人
21          Stream<String> manStream =
    manList.stream().filter(s -> s.length() == 3).limit(3);
22
23          //女演员只要姓林的，并且不要第一个
```

```
        Stream<String> womanStream =
womanList.stream().filter(s ->
s.startsWith("林")).skip(1);

        //把过滤后的男演员姓名和女演员姓名合并到一起
        Stream<String> stream = Stream.concat(manStream,
womanStream);

        // 将流中的数据封装成Actor对象之后打印
        stream.forEach(name -> {
            Actor actor = new Actor(name);
            System.out.println(actor);
        });
    }
}
```

3.方法引用

3.1体验方法引用【理解】

- 方法引用的出现原因

  在使用Lambda表达式的时候，我们实际上传递进去的代码就是一种解决方案：拿参数做操作

  那么考虑一种情况：如果我们在Lambda中所指定的操作方案，已经有地方存在相同方案，那是否还有必要再写重复逻辑呢？答案肯定是没有必要

  那我们又是如何使用已经存在的方案的呢？

  这就是我们要讲解的方法引用，我们是通过方法引用来使用已经存在的方案

- 代码演示

```
public interface Printable {
    void printString(String s);
}

public class PrintableDemo {
    public static void main(String[] args) {
        //在主方法中调用usePrintable方法
//        usePrintable((String s) -> {
//            System.out.println(s);
//        });
        //Lambda简化写法
```

```
12          usePrintable(s -> System.out.println(s));
13
14          //方法引用
15          usePrintable(System.out::println);
16
17      }
18
19      private static void usePrintable(Printable p) {
20          p.printString("爱生活爱Java");
21      }
22  }
23
```

3.2方法引用符【理解】

- 方法引用符

  :: 该符号为引用运算符，而它所在的表达式被称为方法引用

- 推导与省略
  - 如果使用Lambda，那么根据“可推导就是可省略”的原则，无需指定参数类型，也无需指定的重载形式，它们都将被自动推导
  - 如果使用方法引用，也是同样可以根据上下文进行推导
  - 方法引用是Lambda的孪生兄弟

3.3引用类方法【应用】

引用类方法，其实就是引用类的静态方法

- 格式

  类名::静态方法

- 范例

  Integer::parseInt

  Integer类的方法：public static int parseInt(String s) 将此String转换为int类型数据

- 练习描述
  - 定义一个接口(Converter)，里面定义一个抽象方法 int convert(String s);
  - 定义一个测试类(ConverterDemo)，在测试类中提供两个方法
    - 一个方法是：useConverter(Converter c)

        - 一个方法是主方法，在主方法中调用useConverter方法
- 代码演示

```
public interface Converter {
    int convert(String s);
}

public class ConverterDemo {
    public static void main(String[] args) {

        //Lambda写法
        useConverter(s -> Integer.parseInt(s));

        //引用类方法
        useConverter(Integer::parseInt);

    }

    private static void useConverter(Converter c) {
        int number = c.convert("666");
        System.out.println(number);
    }
}
```

- 使用说明

  Lambda表达式被类方法替代的时候，它的形式参数全部传递给静态方法作为参数

3.4引用对象的实例方法【应用】

引用对象的实例方法，其实就引用类中的成员方法

- 格式

  对象::成员方法

- 范例

  "HelloWorld"::toUpperCase

  String类中的方法：public String toUpperCase() 将此String所有字符转换为大写

- 练习描述
    - 定义一个类(PrintString)，里面定义一个方法

public void printUpper(String s)：把字符串参数变成大写的数据，然后在控制台输出

- 定义一个接口(Printer)，里面定义一个抽象方法

  void printUpperCase(String s)

- 定义一个测试类(PrinterDemo)，在测试类中提供两个方法
  - 一个方法是：usePrinter(Printer p)
  - 一个方法是主方法，在主方法中调用usePrinter方法

- 代码演示

```
public class PrintString {
    //把字符串参数变成大写的数据，然后在控制台输出
    public void printUpper(String s) {
        String result = s.toUpperCase();
        System.out.println(result);
    }
}

public interface Printer {
    void printUpperCase(String s);
}

public class PrinterDemo {
    public static void main(String[] args) {

        //Lambda简化写法
        usePrinter(s ->
System.out.println(s.toUpperCase()));

        //引用对象的实例方法
        PrintString ps = new PrintString();
        usePrinter(ps::printUpper);

    }

    private static void usePrinter(Printer p) {
        p.printUpperCase("HelloWorld");
    }
}

```

- 使用说明

Lambda表达式被对象的实例方法替代的时候，它的形式参数全部传递给该方法作为参数

3.5引用类的实例方法【应用】

引用类的实例方法，其实就是引用类中的成员方法

- 格式

  类名::成员方法

- 范例

  String::substring

  public String substring(int beginIndex,int endIndex)

  从beginIndex开始到endIndex结束，截取字符串。返回一个子串，子串的长度为endIndex-beginIndex

- 练习描述
  - 定义一个接口(MyString)，里面定义一个抽象方法：

    String mySubString(String s,int x,int y);
  - 定义一个测试类(MyStringDemo)，在测试类中提供两个方法
    - 一个方法是：useMyString(MyString my)
    - 一个方法是主方法，在主方法中调用useMyString方法
- 代码演示

```
public interface MyString {
    String mySubString(String s,int x,int y);
}

public class MyStringDemo {
    public static void main(String[] args) {
        //Lambda简化写法
        useMyString((s,x,y) -> s.substring(x,y));

        //引用类的实例方法
        useMyString(String::substring);

    }

    private static void useMyString(MyString my) {
        String s = my.mySubString("HelloWorld", 2, 5);
```

```
        System.out.println(s);
    }
}
```

- 使用说明

  Lambda表达式被类的实例方法替代的时候
  第一个参数作为调用者
  后面的参数全部传递给该方法作为参数

3.6引用构造器【应用】

引用构造器，其实就是引用构造方法

- l格式

  类名::new

- 范例

  Student::new

- 练习描述
  - 定义一个类(Student)，里面有两个成员变量(name,age)

    并提供无参构造方法和带参构造方法，以及成员变量对应的get和set方法
  - 定义一个接口(StudentBuilder)，里面定义一个抽象方法

    Student build(String name,int age);
  - 定义一个测试类(StudentDemo)，在测试类中提供两个方法
    - 一个方法是：useStudentBuilder(StudentBuilder s)
    - 一个方法是主方法，在主方法中调用useStudentBuilder方法
- 代码演示

```
public class Student {
    private String name;
    private int age;

    public Student() {
    }

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

```

```
    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}

public interface StudentBuilder {
    Student build(String name,int age);
}

public class StudentDemo {
    public static void main(String[] args) {

        //Lambda简化写法
        useStudentBuilder((name,age) -> new
Student(name,age));

        //引用构造器
        useStudentBuilder(Student::new);

    }

    private static void useStudentBuilder(StudentBuilder
sb) {
        Student s = sb.build("林青霞", 30);
        System.out.println(s.getName() + "," +
s.getAge());
    }
}
```

- 使用说明

  Lambda表达式被构造器替代的时候，它的形式参数全部传递给构造器作为参数