# 1.可变参数

## 1.1可变参数【应用】

- 可变参数介绍
  - 可变参数又称参数个数可变，用作方法的形参出现，那么方法参数个数就是可变的了
  - 方法的参数类型已经确定,个数不确定,我们可以使用可变参数
- 可变参数定义格式

```
修饰符 返回值类型 方法名(数据类型... 变量名) { }
```

- 可变参数的注意事项
  - 这里的变量其实是一个数组
  - 如果一个方法有多个参数，包含可变参数，可变参数要放在最后
- 可变参数的基本使用

```
public class ArgsDemo01 {
    public static void main(String[] args) {
        System.out.println(sum(10, 20));
        System.out.println(sum(10, 20, 30));
        System.out.println(sum(10, 20, 30, 40));

        System.out.println(sum(10,20,30,40,50));
        System.out.println(sum(10,20,30,40,50,60));
        System.out.println(sum(10,20,30,40,50,60,70));
        System.out.println(sum(10,20,30,40,50,60,70,80,90,100));
    }

//    public static int sum(int b,int... a) {
//        return 0;
//    }

    public static int sum(int... a) {
        int sum = 0;
        for(int i : a) {
            sum += i;
        }
        return sum;
    }
}
```

## 1.2创建不可变集合【理解】

- 方法介绍
  - 在List、Set、Map接口中,都存在of方法,可以创建一个不可变的集合
    - 这个集合不能添加,不能删除,不能修改

        - 但是可以结合集合的带参构造,实现集合的批量添加
    - 在Map接口中,还有一个ofEntries方法可以提高代码的阅读性
        - 首先会把键值对封装成一个Entry对象,再把这个Entry对象添加到集合当中
- 示例代码

```
public class MyVariableParameter4 {
    public static void main(String[] args) {
        // static <E>  List<E>  of(E…elements)  创建一个具有指定元素的List集合对象
        //static <E>  Set<E>  of(E…elements)    创建一个具有指定元素的Set集合对象
        //static <K , V>   Map<K, V>  of(E…elements) 创建一个具有指定元素的Map集合对象

        //method1();
        //method2();
        //method3();
        //method4();

    }

    private static void method4() {
        Map<String, String> map = Map.ofEntries(
                Map.entry("zhangsan", "江苏"),
                Map.entry("lisi", "北京"));
        System.out.println(map);
    }

    private static void method3() {
        Map<String, String> map = Map.of("zhangsan", "江苏", "lisi", "北京", "wangwu", "天
津");
        System.out.println(map);
    }

    private static void method2() {
        //传递的参数当中，不能存在重复的元素。
        Set<String> set = Set.of("a", "b", "c", "d","a");
        System.out.println(set);
    }

    private static void method1() {
        List<String> list = List.of("a", "b", "c", "d");
        System.out.println(list);
        //list.add("Q");
        //list.remove("a");
        //list.set(0,"A");
        //System.out.println(list);

//        ArrayList<String> list2 = new ArrayList<>();
//        list2.add("aaa");
//        list2.add("aaa");
//        list2.add("aaa");
//        list2.add("aaa");

        //集合的批量添加。
```

```
        //首先是通过调用List.of方法来创建一个不可变的集合，of方法的形参就是一个可变参数。
        //再创建一个ArrayList集合，并把这个不可变的集合中所有的数据，都添加到ArrayList中。
        ArrayList<String> list3 = new ArrayList<>(List.of("a", "b", "c", "d"));
        System.out.println(list3);
    }
}
```

# 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

```
public class MyStream1 {
    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"));
        //list.add()

        //遍历list1把以张开头的元素添加到list2中。
        ArrayList<String> list2 = new ArrayList<>();
        for (String s : list1) {
            if(s.startsWith("张")){
                list2.add(s);
            }
        }
        //遍历list2集合，把其中长度为3的元素，再添加到list3中。
        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

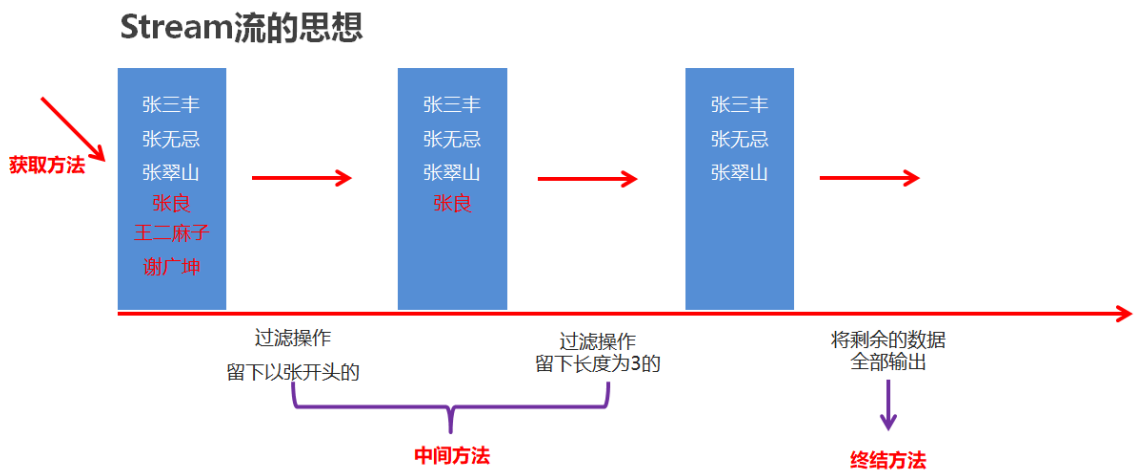
```
public class StreamDemo {
    public static void main(String[] args) {
        //集合的批量添加
        ArrayList<String> list1 = new ArrayList<>(List.of("张三丰","张无忌","张翠山","王二麻
子","张良","谢广坤"));

        //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 |
| 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

```
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);
    }
}
```

## 5.6Stream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;

    public Actor(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

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

  测试类

```
public class StreamTest {
    public static void main(String[] args) {
        //创建集合
        ArrayList<String> manList = new ArrayList<String>();
        manList.add("周润发");
        manList.add("成龙");
        manList.add("刘德华");
        manList.add("吴京");
        manList.add("周星驰");
        manList.add("李连杰");


        ArrayList<String> womanList = new ArrayList<String>();
```

```
        womanList.add("林心如");
        womanList.add("张曼玉");
        womanList.add("林青霞");
        womanList.add("柳岩");
        womanList.add("林志玲");
        womanList.add("王祖贤");

        //男演员只要名字为3个字的前三人
        Stream<String> manStream = manList.stream().filter(s -> s.length() == 3).limit(3);

        //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);
        });
    }
}
```