day09【线程状态、等待与唤醒、Lambda表达式、Stream流】

今日内容

- 线程状态
  - 线程6种状态
  - 等待唤醒
- Lambda表达式
  - 面向接口编程
  - 使用格式
  - 使用形式
- Stream流
  - 理解
  - 获取方式
  - 常用方法
  - 结果收集

教学目标

- [ ] 能够说出线程6个状态的名称
- [ ] 能够理解等待唤醒案例
- [ ] 能够掌握Lambda表达式的标准格式与省略格式
- [ ] 能够通过集合、映射或数组方式获取流
- [ ] 能够掌握常用的流操作
- [ ] 能够将流中的内容收集到集合和数组中

第一章 线程状态

知识点-- 线程状态

目标

- 能够说出线程6个状态的名称

路径

- 线程状态概述
- 演示睡眠方法
- 演示无限等待
- 演示等待和唤醒

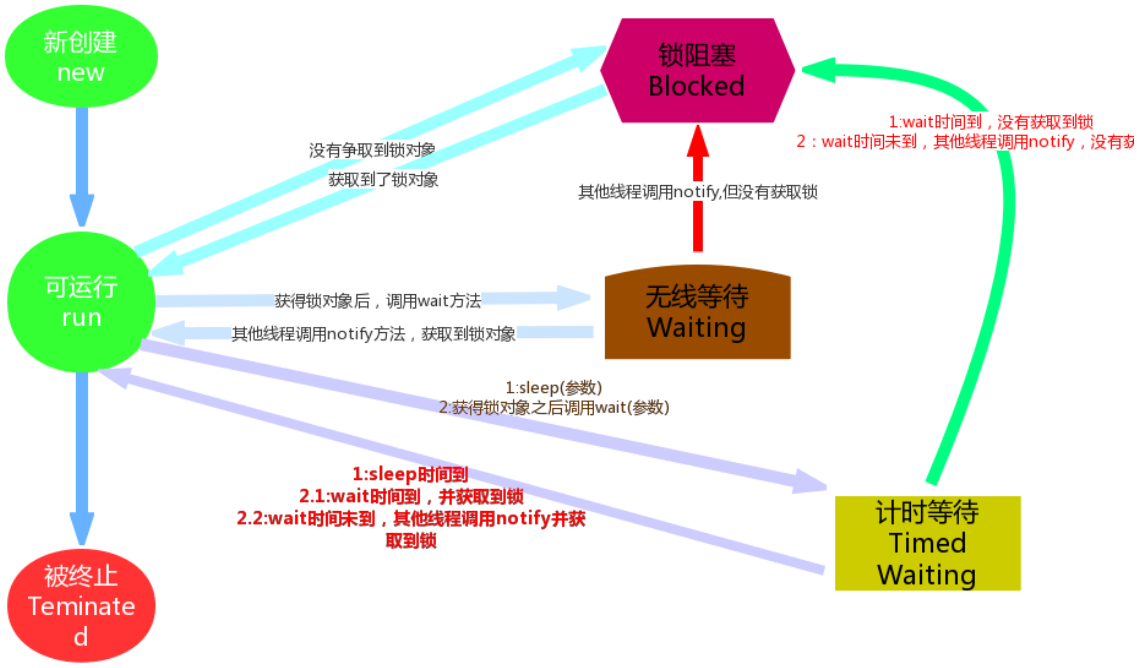
讲解

1.1.1线程状态概述

当线程被创建并启动以后，它既不是一启动就进入了执行状态，也不是一直处于执行状态。

在API中`java.lang.Thread.State`这个枚举中给出了六种线程状态：

线程状态	导致状态发生条件
新建:NEW	线程刚创建，并未启动(未调用start方法)。 MyThread t = new MyThread()只有线程对象，没有线程特征。
可运行:Runnable 线程就绪(经典教法)	线程启动后(调用start方法)，具备执行资格，等待cpu提供执行权限 线程在java虚拟机，可能正在运行，也可能没有。
锁阻塞:Blocked	线程试图获取对象锁，该对象锁被其他线程持有，该线程出于锁阻塞状态； 当该线程持有锁时，该线程将变成Runnable状态。
无限等待:Waiting	一个线程等待另一个线程执行(唤醒)动作时，该线程出于无限等待状态，该状态不能自动唤醒。 当收到唤醒通知，该线程若获取锁将变成Runnable状态，未获取锁将进入锁阻塞。 常见功能:Object-wait() 、Object-notify()、Object-notifyAll()
计时等待:Timed Waiting	一个线程在指定时间内，等待另一个线程执行(唤醒)动作，该线程出于计时等待状态 当时间超时或收到唤醒通知，该线程若获取锁将变成Runnable状态，未获取锁将进入锁阻塞。 常见功能:Object-wait(long time) Thread-sleep(long time)
Teminated(被终止)	因为run方法正常退出而死亡，或者因为没有捕获的异常终止了run方法而死亡。



线程状态相关中的方法:

```
Thread类的计时等待方法.
        public static void  sleep(long time)  让当前线程进入到睡眠状态，到毫秒后自动醒来继续执行。
Object类的等待唤醒方法
        public void wait()  让当前线程进入到等待状态 此方法必须锁对象调用.
        public void notify()  唤醒当前锁对象上等待状态的线程  此方法必须锁对象调用.
```

1.1.2演示睡眠方法

需求：通过测试类演示sleep方法的使用

//测试类

```
public class Test {
    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 100; i++) {
            System.out.println("睡一秒...");
            Thread.sleep(1000);
        }
    }
}
```

> 这时我们发现主线程执行到sleep方法会休眠1秒后再继续执行。

1.1.3演示无限等待

需求：通过测试类演示无限等待

//测试类代码

```
public class Test {
    public static void main(String[] args) throws InterruptedException {
        Object lock = new Object();
        synchronized (lock) {
            System.out.println("停下来...");
            lock.wait();
        }
        System.out.println("结束");
    }
}
```

1.1.4演示等待和唤醒

需求：通过测试类创建两个线程演示等待唤醒

//测试类代码

```
public class Test {
    public static void main(String[] args) throws InterruptedException {
        Object lock = new Object();
        new Thread() {
            @Override
            public void run() {
                synchronized (lock) {
```

```
                    System.out.println("停下来...");
                    try {
                        lock.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.println("结束");
            }
        }.start();

        new Thread() {
            @Override
            public void run() {
                synchronized (lock) {
                    System.out.println("唤醒");
                    lock.notify();
                }
            }
        }.start();
    }
}
```

小结

```
```

案例-- 等待唤醒案例（包子铺卖包子）

需求:

利用等待唤醒机制，在多线程环境下，演示如何有效利用资源开展包子的生产和消费过程。

分析:

包子类定义开关变量，标记包子的状态。
(无包子)包子铺线程生产包子，(有包子)吃货线程消费包子。
通过判断包子的状态，控制包子铺线程和吃货线程交替任务执行。

步骤:
包子类:包子状态为true,表示有包子，为false表示无包子。
吃货线程
    没包子时，包子状态为false，吃货线程进入等待(将执行权交给包子铺做包子)。
    有包子时，包子状态为true，吃货线程吃完包子，包子状态为false，唤醒包子铺线程
包子铺线程
    有包子时，包子状态为true，包子铺线程进入等待(将执行权交给吃货吃包子)。
    没包子时，包子状态为false，包子铺线程生产包子，包子状态为true，唤醒吃货线程。

实现:

//包子资源类代码

```
public class BaoZi {
    private boolean flag;//true 代表有包子    false 代表是没有包子
    private String pier;
    private String xianer;

    public BaoZi() {
    }

    public BaoZi(String pier, String xianer) {
        this.pier = pier;
        this.xianer = xianer;
    }

    public boolean isFlag() {
        return flag;
    }

    public void setFlag(boolean flag) {
        this.flag = flag;
    }

    public String getPier() {
        return pier;
    }

    public void setPier(String pier) {
        this.pier = pier;
    }

    public String getXianer() {
        return xianer;
    }

    public void setXianer(String xianer) {
        this.xianer = xianer;
    }
```

```
    @Override
    public String toString() {
        return "BaoZi{" +
                "pier='" + pier + '\'' +
                ", xianer='" + xianer + '\'' +
                '}';
    }
}
```

//吃货线程类代码

```
public class ChiHuo extends Thread {
    public BaoZi bz;

    public ChiHuo(BaoZi bz) {
        this.bz = bz;
    }

    @Override
    public void run() {
        while (true) {
            synchronized (bz) {
                // 没包子时，包子状态为false，吃货线程进入等待(将执行权交给包子铺做包子)。
                // if (bz.isFlag()==false)
                if (!bz.isFlag()) {
                    try {
                        bz.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                // 有包子时，包子状态为true，吃货线程吃完包子，包子状态为false，唤醒包子铺
线程
                System.out.println("吃货开始吃包子:" + bz.toString());
                bz.setFlag(false);
                bz.notify();
            }
        }
    }
}
```

//包子铺线程类代码

```
public class BaoZiPu extends Thread {
    public BaoZi bz;

    public BaoZiPu(BaoZi bz) {
        this.bz = bz;
    }

    @Override
    public void run() {
        int index = 0;
        while (true) {
```

```
            synchronized (bz) {
                // 有包子时，包子状态为true，包子铺线程进入等待(将执行权交给吃货吃包子)。
                // if (bz.isFlag()==true)
                if (bz.isFlag()) {
                    try {
                        bz.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                // 没包子时，包子状态为false，包子铺线程生产包子，包子状态为true，唤醒吃货
线程。
                try {
                    System.out.println("做包子中...");
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                if (index % 2 == 0) {
                    bz.setPier("水晶皮");
                    bz.setXianer("五仁");
                } else {
                    bz.setPier("糯米");
                    bz.setXianer("红豆");
                }
                index++;
                bz.setFlag(true);
                bz.notify();
            }

        }
    }
}
```

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        //创建包子对象
        BaoZi bz = new BaoZi();
        //使用匿名对象创建吃货线程并启动
        new ChiHuo(bz).start();
        //使用匿名对象创建包子铺线程并启动
        new BaoZiPu(bz).start();
    }
}
```

小结:

第二章 Lambda表达式

知识点-- Lambda表达式概述

目标

- 理解Lambda表达式的基本知识

路径

- 引入
- 函数式编程思想
- 使用Lambda的前提
- 演示面向对象编程

讲解

2.1.1引入

它是一个JDK8开始一个新语法。它是一种“代替语法”。

Lambda表达式,替换以前的接口对象实现， 本质是一个匿名内部类的简易实现。

2.1.2编程思想

“面向对象”的编程思想:必须依靠对象,通过对象调用方法来完成功能

```
例如:在调用Thread()的构造方法:
1).先定义Runnable实现类; 2).创建实现类对象; 3).传入实现类对象;
```

函数式编程思想：在写法上要比较简洁，注重代码的实现过程。

```
例如:在调用Thread()的构造方法
不需要定义实现类;不需要创建具体的子类对象;只需要传入一个“方法”即可。
```

2.1.3演示"面向对象”编程

需求：通过Runnable做100次循环遍历，分别演示面向对象编程(自定义实现类、匿名内部类、匿名内部类简化)与函数式编程。

//接口实现类代码

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(Thread.currentThread().getName() + "第" + i + "次
运行");
        }
    }
}
```

//测试类代码

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

```
        //自定义实现类
        // MyRunnable mr1 = new MyRunnable();
        // Thread t1 = new Thread(mr1, "线程1");
        // t1.start();
        //匿名内部类
        // Runnable mr2 = new Runnable() {
        //     @Override
        //     public void run() {
        //         for (int i = 0; i < 100; i++) {
        //             System.out.println(Thread.currentThread().getName() + "第"
+ i + "次运行");
        //         }
        //     }
        // };
        // Thread t2 = new Thread(mr2, "线程2");
        // t2.start();
        // 匿名内部类简化
        // Thread t3 = new Thread(new Runnable() {
        //     @Override
        //     public void run() {
        //         for (int i = 0; i < 100; i++) {
        //             System.out.println(Thread.currentThread().getName() + "第"
+ i + "次运行");
        //         }
        //     }
        // }, "线程3");
        // t3.start();
        // 函数式编程
        Thread t4 = new Thread(() -> {
            for (int i = 0; i < 100; i++) {
                System.out.println(Thread.currentThread().getName() + "第" + i +
"次运行");
            }
        }, "线程4");
        t4.start();
    }
}
```

小结

知识点-- Lambda表达式标准格式

目标

- 掌握Lambda表达式的标准格式

路径

- 使用Lambda的前提
- Lambda的标准定义格式
- Lambda的格式应用场景
- 演示Lambda的标准格式应用

讲解

2.2.1使用前提

必须具有接口，且要求接口中有且仅有一个抽象方法。

无论是JDK内置的Runnable、 Comparator接口还是自定义的接口，只有当接口中的抽象方法存在且唯一时，才可以使用Lambda。

注意: 有且仅有一个抽象方法的接口，称为**函数式接口**

@FunctionalInterface注解:检查一个接口是否是函数式接口

2.2.2标准定义格式

格式：(参数类型 参数名)->{代码语句}

格式说明:

- ()内的语法与传统方法参数列表一致：无参数则留空，多个参数则用逗号分隔。
- `->`是新引入的语法格式，代表指向动作。
- {}内的语法与传统方法体要求基本一致。

2.2.3应用方式

方式1“无参、无返回值”

方式2“有参、有返回值”

2.2.4演示Lambda的标准格式

需求：通过Lambda的标准格式完成Collections中的自定义排序功能与Runnable接口的使用。

```
public class Test {
    public static void main(String[] args) {
        // “有参、有返回值”
        //定义一个集合
        List<Integer> list = new ArrayList<>();
        Collections.addAll(list, 1, 5, 2, 3, 8);
        // Comparator c = new Comparator<Integer>() {
        //     @Override
        //     public int compare(Integer o1, Integer o2) {
        //         return o1 - o2;
        //     }
        // };
        // Collections.sort(list, c);
        Collections.sort(list, (Integer i1, Integer i2) -> {
            return i1 - i2;
        });
        System.out.println(list);
        // “无参、无返回值”   匿名对象+Lambda表达式
        // new Thread(
        //         new Runnable() {
        //             @Override
        //             public void run() {
        //                 System.out.println("我是一个线程");
        //             }
        //         }
        // ).start();
```

```
        new Thread(() -> {
            System.out.println("我是一个线程");
        }).start();
    }
}
```

小结

```

```

知识点-- Lambda的省略格式

目标

- 掌握Lambda表达式的省略格式

路径

- Lambda的省略格式
- 演示Lambda的省略格式使用

讲解

2.3.1Lambda的省略格式

格式说明:

1. 小括号内参数的类型可以省略;
2. 如果小括号内有且仅有一个参数，则小括号和参数类型可以一起省略;
3. 如果大括号内有且仅有一个语句，则可以同时省略一对大括号，语句后的分号，return关键字;

2.3.2演示Lambda的省略格式

需求:在一个接口中定义一个对新闻消息的处理方法，在测试类中定义一个方法，接收一个消息集合与新闻接口的实现类对象。通过Lambda的省略格式完成上述需求。

//新闻接口

```
@FunctionalInterface
public interface NewInter {
    public String message(String message);
}
```

//新闻接口实现类

```
public class NewsInterImpl implements NewInter {
    @Override
    public String message(String message) {
        return message + "已被处理";
    }
}
```

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("消息1");
        list.add("消息2");
        list.add("消息3");
        //自定义类
        // NewsInterImpl nii = new NewsInterImpl();
        // showMessage(list, nii);
        //匿名内部类
        // showMessage(list, new NewInter() {
        //     @Override
        //     public String message(String message) {
        //         return message + "已被处理";
        //     }
        // });
        //Lambda表达式
        // showMessage(list, (String message) -> {
        //     return message + "已被处理";
        // });
        //Lambda表达式省略格式
        showMessage(list, message ->
                message + "已被处理"
        );
    }

    public static void showMessage(List<String> list, NewInter ni) {
        for (int i = 0; i < list.size(); i++) {
            String message = list.get(i);
            System.out.println(ni.message(message));
        }
    }
}
```

小结

```

```

知识点-- Lambda的应用场景

目标

- 理解Lambda应用场景

路径

- Lambda的应用场景
- 演示Lambda的应用场景

讲解

2.4Lambda的应用场景

用变量的形式

在调用方法时，作为“实参”

作为方法的“返回值”

2.4.2演示Lambda的应用场景

需求:使用Runnable接口演示Lambda的三种应用场景。

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        // 用变量的形式
        Runnable r = () -> { };//new Runnable(){}
        // 在调用方法时，作为“实参”
        showRunnable(() -> { });
        // 作为方法的“返回值”应用方式
        getRunnable();
    }

    public static void showRunnable(Runnable r) {
        r.run();
    }

    public static Runnable getRunnable() {
        return () -> { };
    }
}
```

小结

```

```

第三章 Stream

知识点-- Stream引入

目标

- 理解Stream流的作用

路径

- 引言
- 演示集合数据筛选过滤案例

讲解

3.1.1引言

`java.util.stream.Stream<T>`是Java 8新加入的最常用的流接口。（不是函数式接口）

JDK8中为了支持Lambda，制作了一些应用—Stream就是一个典型的应用。

Stream流：是一个接口，功能类似于迭代器，但更强大，可以对数据进行过滤、筛选、汇总等操作。

3.1.2演示集合数据筛选过滤案例

需求：使用集合与Stream流中的方法完成下列需求

```
1.定义一个集合，存储若干姓名
2.将List集合中姓张的的元素过滤到一个新的集合中
3.然后将过滤出来的姓张的元素中过滤出长度为3的元素,存储到一个新的集合中
```

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        //普通方式
        method1();
        // Stream流
        // 定义一个集合，存储若干姓名
        List<String> list = new ArrayList<>();
        list.add("张三");
        list.add("李四");
        list.add("王五");
        list.add("赵六");
        list.add("孙七");
        list.add("周八");
        list.add("张三丰");

        list.stream().filter((String name) -> {
            return name.startsWith("张");
        }).filter((String name) -> {
            return name.length() == 3;
        }).forEach((String name) -> {
            System.out.println(name);
        });
    }

    public static void method1() {
        // 定义一个集合，存储若干姓名
        List<String> list = new ArrayList<>();
        list.add("张三");
        list.add("李四");
        list.add("王五");
        list.add("赵六");
        list.add("孙七");
        list.add("周八");
        list.add("张三丰");
        // 将List集合中姓张的的元素过滤到一个新的集合中
        List<String> list1 = new ArrayList<>();
        for (int i = 0; i < list.size(); i++) {
```

```
            String name = list.get(i);
            if (name.startsWith("张")) {
                list1.add(name);
            }
        }
        // 然后将过滤出来的姓张的元素中过滤出长度为3的元素,存储到一个新的集合中
        List<String> list2 = new ArrayList<>();
        for (int i = 0; i < list1.size(); i++) {
            String name = list1.get(i);
            if (name.length() == 3) {
                list2.add(name);
            }
        }
        System.out.println(list2);
    }
}
```

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。**

代码中并没有体现使用线性循环或是其他任何算法进行遍历，我们真正要做的事情内容被更好地体现在代码中。

小结

知识点-- 流式思想

目标

- 理解Stream中的流式思想的特点

路径

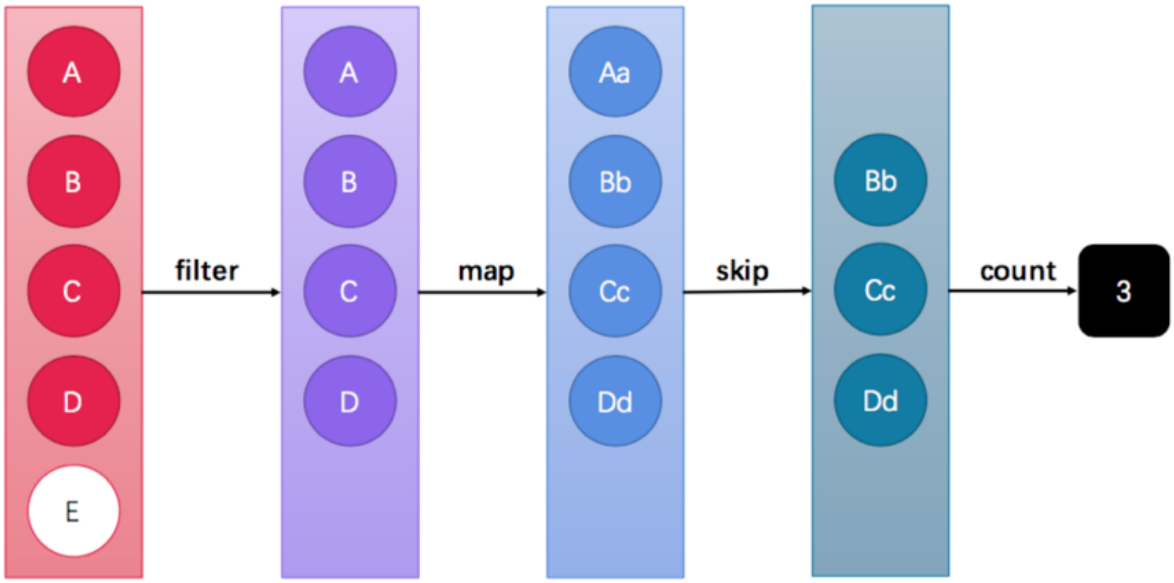
- 流式思想概述
- Stream流的特点

讲解

3.2.1流式思想概述

Stream流，类似于车间的流水线，每次操作流，都可以将结果发送给下一个操作。

当需要对多个元素进行操作（特别是多步操作）的时候，考虑到性能及便利性，我们应该首先拼好一个“模型”步骤方案，然后再使用它。



这张图中展示了过滤、映射、跳过、计数等多步操作，这是一种集合元素的处理方案，而方案就是一种“函数模型”。图中的每一个方框都是一个“流”，调用指定的方法，可以从一个流模型转换为另一个流模型。而最右侧的数字3是最终结果。

这里的`filter`、`map`、`skip`都是在对函数模型进行操作，集合元素并没有真正被处理。只有当终结方法`count`执行的时候，整个模型才会按照指定策略执行操作。而这得益于Lambda的延迟执行特性。

3.2.2Stream流的特点

- Stream流是**一次性**的,不能重复使用，当执行流的某个方法，这个流将将失效，并将结果存储到**新流**中。
- Stream流不会存储数据
- Stream流不会修改数据源
- Stream流搭建的函数模型，只有终结方法存在,前面的延迟性方法才会执行。
  - 终结方法: Stream流中返回值类型不是Stream方法
  - 延迟方法: Stream流中返回值类型是Stream方法

小结

知识点-- 获取Stream流

目标

- 掌握获取Stream流的方式

路径

- 演示根据Collection获取流
- 演示根据Map获取流
- 演示根据数组获取流

讲解

3.3.1获取方式

- `java.util.Collection`接口中加入了default方法`stream`用来获取流，所以其所有实现类均可获取流。
- `java.util.Map`接口不是`Collection`的子接口，且K-V数据结构不符合流元素的单一特征，需分别根据其键和值的集合获取流对象。
- 数组对象没有方法，所以`Stream`接口中提供了静态方法`of`获取数组对应的流。

3.3.2演示获取流的3种情况

需求：分别获取Collection、Map及数组的Stream流对象

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        //获取单列集合的流对象
        List<String> list = new ArrayList<>();
        list.add("a");
        list.add("b");
        list.add("c");
        Stream<String> stream1 = list.stream();
        //获取双列集合的流对象
        Map<String, String> m = new HashMap<>();
        m.put("1", "a");
        m.put("2", "b");
        m.put("3", "c");
        //键集的流对象
        Set<String> keys = m.keySet();
        Stream<String> stream2 = keys.stream();
        //值集的流对象
        Collection<String> values = m.values();
        Stream<String> stream3 = values.stream();
        //获取数组的流对象
        int[] arr = {1, 2, 3};
```

```
        Stream<int[]> stream4 = Stream.of(arr);//of(T t)
        Stream<Integer> stream5 = Stream.of(1, 2, 3);//of(T...t)
    }
}
```

> 备注：`of`方法的参数其实是一个可变参数，所以支持数组。

小结

知识点-- Stream流常用功能

目标

- 能够掌握常用的流操作

路径

- 概述
- 常用方法介绍
- 演示常用功能

讲解

3.4.1概述

流模型的操作很丰富，这里介绍一些常用的API，这些方法可以被分成两种：

- **终结方法**：返回值类型不再是`Stream`接口的方法，支持链式调用。
- **非终结方法**(函数拼接方法)：返回值类型仍然是`Stream`接口的方法，不支持链式调用。

3.4.2常用方法

```
方法名      作用        分类   链式调用     详解
count    统计个数     终结        否     long count();
                                          返回流中的元素个数。
forEach 逐一处理     终结        否     void forEach(Consumer<? super T> action);
                                          对此流的每个元素进行操作
filter   过滤        非终结      是      Stream<T> filter(Predicate<? super T>
predicate);
                                          返回经过筛选，满足的条件的元素组成的流
limit    取前n个     非终结      是     Stream<T> limit(long maxSize);
                                          返回由此流(包含)第maxSize个之前的元素组成
的流
skip     跳过前n     非终结      是     Stream<T> skip(long n);
                                          返回由此流(不含)第n个之后元素组成的流
map      映射        非终结      是      <R> Stream<R> map(Function<? super T, ?
extends R> mapper);
                                          返回流中旧元素经指定规则转换后的心元素组成的
流
concat   组合        非终结      是      static <T> Stream<T> concat(Stream<? extends
T> a, Stream<? extends T> b)
                                          返回将两个流中的元素合并到一其组成的流
```

> 备注：本小节之外的更多方法，请自行参考API文档。

3.4.3演示常用功能

需求:演示 Stream流常用功能

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        // long count();返回流中的元素个数。
        Stream<Integer> stream1 = Stream.of(1, 2, 3);
        System.out.println(stream1.count());
        System.out.println("--------");
        // void forEach(Consumer<? super T> action)
        Stream<Integer> stream2 = Stream.of(1, 2, 3);
        stream2.forEach((Integer num) -> {
            System.out.println(num);
        });
        System.out.println("--------");
        // Stream<T> filter(Predicate<? super T> predicate)
        Stream<Integer> stream3 = Stream.of(1, 2, 3);
        Stream<Integer> stream31 = stream3.filter((Integer i) -> {
            return i == 2;
        });
        stream31.forEach((Integer num) -> {
            System.out.println(num);
        });
        System.out.println("--------");
        // Stream<T> limit(long maxSize)
        Stream<Integer> stream4 = Stream.of(1, 2, 3);
        Stream<Integer> stream41 = stream4.limit(2);
        stream41.forEach((Integer num) -> {
            System.out.println(num);
        });
        System.out.println("--------");
        // Stream<T> skip(long n)
        Stream<Integer> stream5 = Stream.of(1, 2, 3);
        Stream<Integer> stream51 = stream5.skip(1);
        stream51.forEach((Integer num) -> {
            System.out.println(num);
        });
        System.out.println("--------");
        // <R> Stream<R> map(Function<? super T, ? extends R> mapper)
        Stream<Integer> stream6 = Stream.of(1, 2, 3);
        Stream<Integer> stream61 =stream6.map((Integer i)->{ return i*10;});
        stream61.forEach((Integer num) -> {
            System.out.println(num);
        });
        System.out.println("--------");
        // static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends
T> b)
        Stream<Integer> stream71 = Stream.of(1, 2, 3);
        Stream<Integer> stream72 = Stream.of(4, 5, 6);
        Stream<Integer> stream7 =Stream.concat(stream71,stream72);
        stream7.forEach((Integer num) -> {
            System.out.println(num);
        });
```

```
    }
}
```

小结

```

```

案例--Stream案例

需求：使用集合与Stream流两种方式，利用循环，实现如下需求

```
0. 定义两个ArrayList集合，代表两个队伍，存储多个姓名，依次进行以下操作
1. 第一个队伍只要名字为3个字的成员姓名；
2. 第一个队伍筛选之后只要前2个人；
3. 第二个队伍只要姓张的成员姓名；
4. 第二个队伍筛选之后不要前2个人；
5. 将两个队伍合并为一个队伍；
6. 根据姓名创建`Person`对象并存储到集合；
7. 打印整个队伍的Person对象信息。
```

分析：

根据集合与Stream流的特性实现该功能

实现

//Person类代码

```
public class Person {

    private String name;

    public Person() {}

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

    @Override
    public String toString() {
        return "Person{name='" + name + "'}";
    }

    public String getName() {
        return name;
    }

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

//测试类

```

```

```java
public class Test {
    public static void main(String[] args) {
        //0. 定义两个ArrayList集合，代表两个队伍，存储多个姓名，依次进行以下操作
        List<String> listOne = new ArrayList<>();
        listOne.add("迪丽热巴");
        listOne.add("宋远桥");
        listOne.add("苏星河");
        listOne.add("老子");
        listOne.add("庄子");
        listOne.add("孙子");
        listOne.add("洪七公");
        List<String> listTwo = new ArrayList<>();
        listTwo.add("古力娜扎");
        listTwo.add("张无忌");
        listTwo.add("张三丰");
        listTwo.add("赵丽颖");
        listTwo.add("张二狗");
        listTwo.add("张天爱");
        listTwo.add("张三");
        //使用集合中的方法完成需求
        method1(listOne, listTwo);
        System.out.println("--------");
        //使用Stream流完成需求
        method2(listOne, listTwo);
    }

    public static void method2(List<String> listOne, List<String> listTwo) {
        //获取两个集合的流
        Stream<String> streamOne = listOne.stream();
        Stream<String> streamTwo = listTwo.stream();
        // 1. 第一个队伍只要名字为3个字的成员姓名；
        // 2. 第一个队伍筛选之后只要前2个人；
        Stream<String> streamOneA = streamOne.filter(s -> s.length() ==
3).limit(2);
        // 3. 第二个队伍只要姓张的成员姓名；
        // 4. 第二个队伍筛选之后不要前2个人；
        Stream<String> streamTwoA = streamTwo.filter(s ->
s.startsWith("张")).skip(2);
        // 5. 将两个队伍合并为一个队伍；
        // 6. 根据姓名创建`Person`对象并存储到集合；
        // 7. 打印整个队伍的Person对象信息。
        Stream.concat(streamOneA, streamTwoA).map(s -> new Person(s)).forEach(p
-> System.out.println(p.getName()));
    }

    public static void method1(List<String> listOne, List<String> listTwo) {

        // 1. 第一个队伍只要名字为3个字的成员姓名；
        List<String> listOneA = new ArrayList<>();
        for (int i = 0; i < listOne.size(); i++) {
            String name = listOne.get(i);
            if (name.length() == 3) {
                listOneA.add(name);
            }
        }
        // 2. 第一个队伍筛选之后只要前2个人；
        List<String> listOneB = new ArrayList<>();
        for (int i = 0; i < listOneA.size(); i++) {
```

```
            String name = listOneA.get(i);
            if (i <= 1) {
                listOneB.add(name);
            }
        }
        // 3. 第二个队伍只要姓张的成员姓名;
        List<String> listTwoA = new ArrayList<>();
        for (int i = 0; i < listTwo.size(); i++) {
            String name = listTwo.get(i);
            if (name.startsWith("张")) {
                listTwoA.add(name);
            }
        }
        // 4. 第二个队伍筛选之后不要前2个人;
        List<String> listTwoB = new ArrayList<>();
        for (int i = 0; i < listTwoA.size(); i++) {
            String name = listTwoA.get(i);
            if (i > 1) {
                listTwoB.add(name);
            }
        }
        // 5. 将两个队伍合并为一个队伍;
        List<String> listAll = new ArrayList<>();
        listAll.addAll(listOneB);
        listAll.addAll(listTwoB);
        // 6. 根据姓名创建`Person`对象并存储到集合;
        List<Person> totalPerson = new ArrayList<>();
        for (int i = 0; i < listAll.size(); i++) {
            String name = listAll.get(i);
            Person p = new Person(name);
            totalPerson.add(p);
        }
        // 7. 打印整个队伍的Person对象信息。
        for (int i = 0; i < totalPerson.size(); i++) {
            Person p = totalPerson.get(i);
            System.out.println(p.getName());
        }
    }
}
```

知识点-- 收集Stream流结果

目标

- 能够掌握收集Stream结果

路径

- 概述
- 演示常用功能

讲解

3.5.1 概述

对流操作完成之后，如果需要将其结果进行收集，例如获取对应的集合、数组等，如何操作?

```
方法名      作用               分类    链式调用  详解
collect   收集结果到集合中    终结    否       R collect(Collector<T,A, R> coll)：转换为指定的集合，R代表最终转为的集合的具体类型
toArray   收集结果到数组中    终结    否       Object[] toArray()：转换为Object数组
```

`java.util.stream.Collectors`类提供一些方法，可以作为`Collector`接口的实例。

```
public static <T> Collector<T, ?, List<T>> toList()：转换为List集合。
public static <T> Collector<T, ?, Set<T>> toSet()：转换为Set集合。
```

3.5.2演示收集Stream结果

需求：演示将流中的数据转换到集合与数组中

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        //收集结果到集合中
        Stream<String> stream1 = Stream.of("张三丰", "张无忌", "周芷若");
        List<String> list1 = stream1.collect(Collectors.toList());
        System.out.println(list1);
        Stream<String> stream2 = Stream.of("张三丰", "张无忌", "周芷若");
        Set<String> set = stream2.collect(Collectors.toSet());
        System.out.println(set);
        //收集结果到数组中
        Stream<String> stream3 = Stream.of("张三丰", "张无忌", "周芷若");
        Object[] arr = stream3.toArray();
        System.out.println(Arrays.toString(arr));
    }
}
```

小结