day09【线程池、死锁、线程状态、等待与唤醒、Lambda表达式、Stream流】

今日目标

- 死锁
- 线程池
- 线程状态
- 等待与唤醒
- Lambda表达式
- Stream流

教学目标

- [ ] 能够描述Java中线程池运行原理
- [ ] 能够描述死锁产生的原因
- [ ] 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 线程池方式

1.1 线程池的思想

我们使用线程的时候就去创建一个线程，这样实现起来非常简便，但是就会有一个问题：

如果并发的线程数量很多，并且每个线程都是执行一个时间很短的任务就结束了，这样频繁创建线程就会大大降低系统的效率，因为频繁创建线程和销毁线程需要时间。

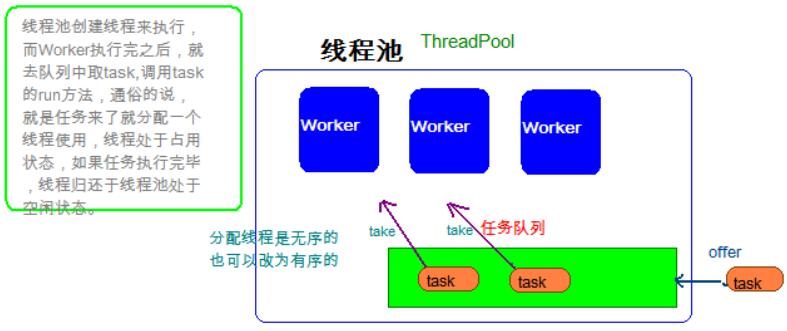
那么有没有一种办法使得线程可以复用，就是执行完一个任务，并不被销毁，而是可以继续执行其他的任务？

在Java中可以通过线程池来达到这样的效果。今天我们就来详细讲解一下Java的线程池。

1.2 线程池概念

- **线程池：**其实就是一个容纳多个线程的容器，其中的线程可以反复使用，省去了频繁创建线程对象的操作，无需反复创建线程而消耗过多资源。

由于线程池中有很多操作都是与优化资源相关的，我们在这里就不多赘述。我们通过一张图来了解线程池的工作原理：



工作线程（PoolWorker）：线程池中线程，在没有任务时处于等待状态，可以循环的执行任务；

任务队列（taskQueue）：用于存放没有处理的任务。提供一种缓冲机制。

任务接口（Task）：每个任务必须实现的接口，以供工作线程调度任务的执行，它主要规定了任务的入口，任务执行完后的收尾工作，任务的执行状态等；

线程池管理器（ThreadPool）：用于创建并管理线程池，包括 创建线程池，销毁线程池，添加新任务。

合理利用线程池能够带来三个好处：

1. 降低资源消耗。减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。
2. 提高响应速度。当任务到达时，任务可以不需要的等到线程创建就能立即执行。
3. 提高线程的可管理性。可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为消耗过多的内存，而把服务器累趴下(每个线程需要大约1MB内存，线程开的越多，消耗的内存也就越大，最后死机)。

1.3 线程池的使用

Java里面线程池的顶级接口是`java.util.concurrent.Executor`，但是严格意义上讲`Executor`并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是`java.util.concurrent.ExecutorService`。

要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，很有可能配置的线程池不是较优的，因此在`java.util.concurrent.Executors`线程工厂类里面提供了一些静态工厂，生成一些常用的线程池。官方建议使用Executors工程类来创建线程池对象。

Executors类中有个创建线程池的方法如下：

- `public static ExecutorService newFixedThreadPool(int nThreads)`：返回线程池对象。(创建的是有界线程池,也就是池中的线程个数可以指定最大数量)

获取到了一个线程池ExecutorService 对象，那么怎么使用呢，在这里定义了一个使用线程池对象的方法如下：

- `public Future<?> submit(Runnable task)`:获取线程池中的某一个线程对象，并执行

  > Future接口：用来记录线程任务执行完毕后产生的结果。

使用线程池中线程对象的步骤：

1. 创建线程池对象。
2. 创建Runnable接口子类对象。(task)
3. 提交Runnable接口子类对象。(take task)
4. 关闭线程池(一般不做)。

Runnable实现类代码：

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("我要一个教练");
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("教练来了： " + Thread.currentThread().getName());
        System.out.println("教我游泳,交完后，教练回到了游泳池");
    }
}
```

线程池测试类：

```
public class ThreadPoolDemo {
    public static void main(String[] args) {
        // 创建线程池对象
```

```
        ExecutorService service = Executors.newFixedThreadPool(2);//包含2个线程对
象
        // 创建Runnable实例对象
        MyRunnable r = new MyRunnable();

        //自己创建线程对象的方式
        // Thread t = new Thread(r);
        // t.start(); ---> 调用MyRunnable中的run()

        // 从线程池中获取线程对象,然后调用MyRunnable中的run()
        service.submit(r);
        // 再获取个线程对象，调用MyRunnable中的run()
        service.submit(r);
        service.submit(r);
        // 注意：submit方法调用结束后，程序并不终止，是因为线程池控制了线程的关闭。
        // 将使用完的线程又归还到了线程池中
        // 关闭线程池
        //service.shutdown();
    }
}
```

Callable测试代码:

- `<T> Future<T> submit(Callable<T> task)` : 获取线程池中的某一个线程对象，并执行.

  Future : 表示计算的结果.
- `V get()` : 获取计算完成的结果。

```
public class ThreadPoolDemo2 {
    public static void main(String[] args) throws Exception {
        // 创建线程池对象
      ExecutorService service = Executors.newFixedThreadPool(2);//包含2个线程对象

        // 创建Runnable实例对象
        Callable<Double> c = new Callable<Double>() {
            @Override
            public Double call() throws Exception {
                return Math.random();
            }
        };

        // 从线程池中获取线程对象,然后调用Callable中的call()
        Future<Double> f1 = service.submit(c);
        // Futur 调用get() 获取运算结果
        System.out.println(f1.get());

        Future<Double> f2 = service.submit(c);
        System.out.println(f2.get());

        Future<Double> f3 = service.submit(c);
        System.out.println(f3.get());
    }
}
```

1.4 线程池的练习

需求: 使用线程池方式执行任务,返回1-n的和

分析: 因为需要返回求和结果,所以使用Callable方式的任务

代码:

```
public class Demo04 {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        ExecutorService pool = Executors.newFixedThreadPool(3);

        SumCallable sc = new SumCallable(100);
        Future<Integer> fu = pool.submit(sc);
        Integer integer = fu.get();
        System.out.println("结果: " + integer);

        SumCallable sc2 = new SumCallable(200);
        Future<Integer> fu2 = pool.submit(sc2);
        Integer integer2 = fu2.get();
        System.out.println("结果: " + integer2);

        pool.shutdown();
    }
}
```

SumCallable.java

```
public class SumCallable implements Callable<Integer> {
    private int n;

    public SumCallable(int n) {
        this.n = n;
    }

    @Override
    public Integer call() throws Exception {
        // 求1-n的和?
        int sum = 0;
        for (int i = 1; i <= n; i++) {
            sum += i;
        }
        return sum;
    }
}
```

第二章 死锁

2.1 什么是死锁

在多线程程序中,使用了多把锁,造成线程之间相互等待.程序不往下走了。

2.2 产生死锁的条件

1.有多把锁 2.有多个线程 3.有同步代码块嵌套

2.3 死锁代码

```
public class DemoO5 {
    public static void main(String[] args) {
        MyRunnable mr = new MyRunnable();

        new Thread(mr).start();
        new Thread(mr).start();
    }
}

class MyRunnable implements Runnable {
    Object objA = new Object();
    Object objB = new Object();

    /*
    嵌套1 objA
    嵌套1 objB
    嵌套2 objB
    嵌套1 objA
     */
    @Override
    public void run() {
        synchronized (objA) {
            System.out.println("嵌套1 objA");
            synchronized (objB) {// t2, objA, 拿不到B锁,等待
                System.out.println("嵌套1 objB");
            }
        }

        synchronized (objB) {
            System.out.println("嵌套2 objB");
            synchronized (objA) {// t1 , objB, 拿不到A锁,等待
                System.out.println("嵌套2 objA");
            }
        }
    }
}
```

注意:**我们应该尽量避免死锁**

第三章 线程状态

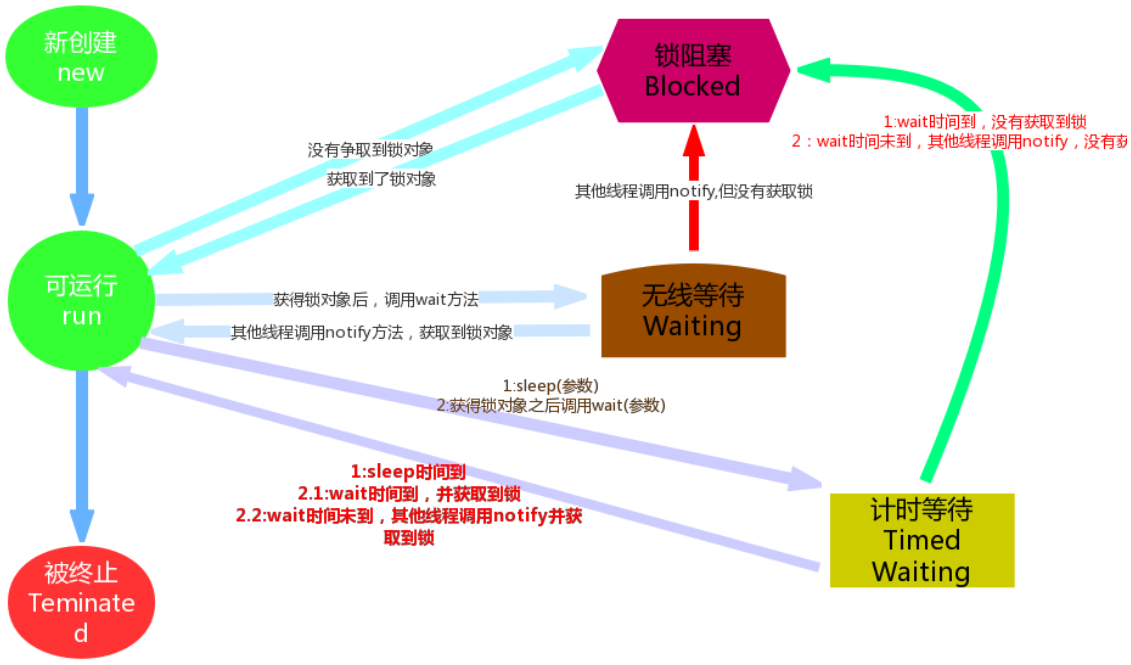
3.1 线程状态概述

线程由生到死的完整过程：技术素养和面试的要求。

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。在线程的生命周期中，有几种状态呢？在API中`java.lang.Thread.State`这个枚举中给出了六种线程状态：

这里先列出各个线程状态发生的条件，下面将会对每种状态进行详细解析

线程状态	导致状态发生条件
NEW(新建)	线程刚被创建，但是并未启动。还没调用start方法。MyThread t = new MyThread只有线程对象，没有线程特征。
Runnable(可运行)	线程可以在java虚拟机中运行的状态，可能正在运行自己代码，也可能没有，这取决于操作系统处理器。调用了t.start()方法： 就绪（经典教法）
Blocked(锁阻塞)	当一个线程试图获取一个对象锁，而该对象锁被其他的线程持有，则该线程进入Blocked状态；当该线程持有锁时，该线程将变成Runnable状态。
Waiting(无限等待)	一个线程在等待另一个线程执行一个（唤醒）动作时，该线程进入Waiting状态。进入这个状态后是不能自动唤醒的，必须等待另一个线程调用notify或者notifyAll方法才能够唤醒。
Timed Waiting(计时等待)	同waiting状态，有几个方法有超时参数，调用他们将进入Timed Waiting状态。这一状态将一直保持到超时期满或者接收到唤醒通知。带有超时参数的常用方法有Thread.sleep 、Object.wait。
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。



我们不需要去研究这几种状态的实现原理，我们只需知道在做线程操作中存在这样的状态。那我们怎么去理解这几个状态呢，新建与被终止还是很容易理解的，我们就研究一下线程从Runnable（可运行）状态与非运行状态之间的转换问题。

3.2 睡眠sleep方法

我们看到状态中有一个状态叫做计时等待，可以通过Thread类的方法来进行演示.

`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

```
public class Test{
  public static void main(String[] args){
    for(int i = 1;i<=5;i++){
        Thread.sleep(1000);
        System.out.println(i)
    }
  }
}
```

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。

3.3 等待和唤醒

Object类的方法

`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 class Demo1_wait {
    public static void main(String[] args) throws InterruptedException {
       // 步骤1 : 子线程开启,进入无限等待状态, 没有被唤醒,无法继续运行.
        new Thread(() -> {
            try {

                System.out.println("begin wait ....");
                synchronized ("") {
                    "".wait();
                }
                System.out.println("over");
            } catch (Exception e) {
            }
        }).start();
    }
```

`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.

```
public class Demo2_notify {
    public static void main(String[] args) throws InterruptedException {
       // 步骤1 : 子线程开启,进入无限等待状态, 没有被唤醒,无法继续运行.
        new Thread(() -> {
            try {

                System.out.println("begin wait ....");
                synchronized ("") {
                    "".wait();
                }
                System.out.println("over");
            } catch (Exception e) {
            }
        }).start();

        //步骤2:  加入如下代码后, 3秒后,会执行notify方法, 唤醒wait中线程.
        Thread.sleep(3000);
        new Thread(() -> {
            try {
                synchronized ("") {
```

```
                    System.out.println("唤醒");
                    "".notify();
                }
            } catch (Exception e) {
            }
        }).start();
    }
}
```

3.4 等待唤醒案例（包子铺卖包子）

定义一个集合，包子铺线程完成生产包子，包子添加到集合中；吃货线程完成购买包子，包子从集合中移除。
1. 当包子没有时（包子状态为false），吃货线程等待.
2. 包子铺线程生产包子（即包子状态为true），并通知吃货线程（解除吃货的等待状态）

代码示例：

生成包子类：

```
public class BaoZiPu extends Thread{
    private List<String> list ;
    public BaoZiPu(String name,ArrayList<String> list){
        super(name);
        this.list = list;
    }
    @Override
    public void run() {
            int i = 0;
            while(true){
                    //list作为锁对象
                    synchronized (list){
                        if(list.size()>0){
                            //存元素的线程进入到等待状态
                            try {
                                list.wait();
                            } catch (InterruptedException e) {
                                e.printStackTrace();
                            }
                        }

                        //如果线程没进入到等待状态 说明集合中没有元素
                        //向集合中添加元素
                        list.add("包子"+i++);
                        System.out.println(list);
                        //集合中已经有元素了 唤醒获取元素的线程
                        list.notify();
                    }
                }
            }
    }
}
```

消费包子类：

```
public class ChiHuo extends Thread {
```

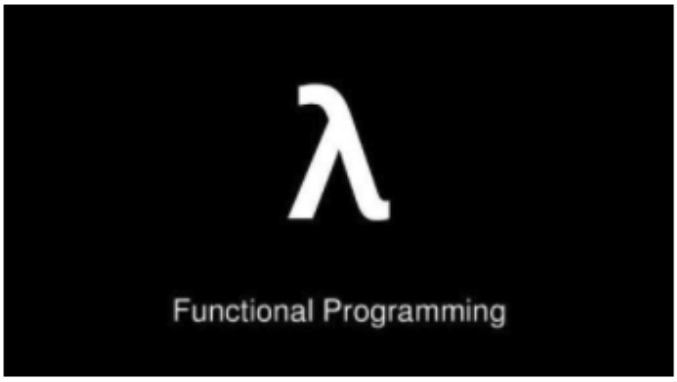
```
    private List<String> list ;
    public ChiHuo(String name,ArrayList<String> list){
        super(name);
        this.list = list;
    }

    @Override
    public void run() {
            //为了能看到效果 写个死循环
                while(true){
                    //由于使用的同一个集合 list作为锁对象
                    synchronized (list){
                        //如果集合中没有元素 获取元素的线程进入到等待状态
                        if(list.size()==0){
                            try {
                                list.wait();
                            } catch (InterruptedException e) {
                                e.printStackTrace();
                            }
                        }
                        //如果集合中有元素 则获取元素的线程获取元素(删除)
                        list.remove(0);
                        //打印集合 集合中没有元素了
                        System.out.println(list);
                        //集合中已经没有元素 则唤醒添加元素的线程 向集合中添加元素
                        list.notify();
                    }
                }
            }
    }
}
```

测试类:

```
public class Demo {
    public static void main(String[] args) {
        //等待唤醒案例
        List<String> list = new ArrayList<>();
        // 创建线程对象
         BaoZiPu bzp = new BaoZiPu("包子铺",list);
        ChiHuo ch = new ChiHuo("吃货",list);
        // 开启线程
        bzp.start();
        ch.start();
    }
}
```

第四章 Lambda表达式，研究

4.1 函数式编程思想概述



在数学中，**函数**就是有输入量、输出量的一套计算方案，也就是“拿什么东西做什么事情”。相对而言，面向对象过分强调“必须通过对象的形式来做事情”，而函数式思想则尽量忽略面向对象的复杂语法——**强调做什么，而不是以什么形式做。**

做什么，而不是怎么做

我们真的希望创建一个匿名内部类对象吗？不。我们只是为了做这件事情而**不得不**创建一个对象。我们真正希望做的事情是：将`run`方法体内的代码传递给`Thread`类知晓。

传递一段代码——这才是我们真正的目的。而创建对象只是受限于面向对象语法而不得不采取的一种手段方式。那，有没有更加简单的办法？如果我们将关注点从“怎么做”回归到“做什么”的本质上，就会发现只要能够更好地达到目的，过程与形式其实并不重要。

4.2 Lambda的优化

当需要启动一个线程去完成任务时，通常会通过`java.lang.Runnable`接口来定义任务内容，并使用`java.lang.Thread`类来启动该线程。

传统写法,代码如下：

```
public class Demo01ThreadNameless {
    public static void main(String[] args) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("多线程任务执行！");
            }
        }).start();
    }
}
```

本着“一切皆对象”的思想，这种做法是无可厚非的：首先创建一个`Runnable`接口的匿名内部类对象来指定任务内容，再将其交给一个线程来启动。

代码分析:

对于`Runnable`的匿名内部类用法，可以分析出几点内容：

- `Thread`类需要`Runnable`接口作为参数，其中的抽象`run`方法是用来指定线程任务内容的核心；
- 为了指定`run`的方法体，**不得不**需要`Runnable`接口的实现类；
- 为了省去定义一个`RunnableImpl`实现类的麻烦，**不得不**使用匿名内部类；
- 必须覆盖重写抽象`run`方法，所以方法名称、方法参数、方法返回值**不得不**再写一遍，且不能写错；
- 而实际上，**似乎只有方法体才是关键所在。**

Lambda表达式写法,代码如下:

借助Java 8的全新语法，上述`Runnable`接口的匿名内部类写法可以通过更简单的Lambda表达式达到等效:

```
public class DemoO2LambdaRunnable {
    public static void main(String[] args) {
        new Thread(() -> System.out.println("多线程任务执行！")).start(); // 启动线
程
    }
}
```

这段代码和刚才的执行效果是完全一样的，可以在1.8或更高的编译级别下通过。从代码的语义中可以看出：我们启动了一个线程，而线程任务的内容以一种更加简洁的形式被指定。

不再有“不得不创建接口对象”的束缚，不再有“抽象方法覆盖重写”的负担，就是这么简单！

4.3 Lambda的格式

标准格式:

Lambda省去面向对象的条条框框，格式由**3个部分**组成:

- 一些参数
- 一个箭头
- 一段代码

Lambda表达式的**标准格式**为:

```
(参数类型 参数名称) -> { 代码语句 }
```

格式说明:

- 小括号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。
- 大括号内的语法与传统方法体要求基本一致。

匿名内部类与lambda对比:

```
new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("多线程任务执行！");
            }
}).start();
```

仔细分析该代码中，`Runnable`接口只有一个`run`方法的定义：

- `public abstract void run();`

即制定了一种做事情的方案（其实就是一个方法）：

- **无参数**：不需要任何条件即可执行该方案。
- **无返回值**：该方案不产生任何结果。
- **代码块**（方法体）：该方案的具体执行步骤。

同样的语义体现在`Lambda`语法中，要更加简单：

```
() -> System.out.println("多线程任务执行！")
```

- 前面的一对小括号即`run`方法的参数（无），代表不需要任何条件；
- 中间的一个箭头代表将前面的参数传递给后面的代码；
- 后面的输出语句即业务逻辑代码。

参数和返回值:

下面举例演示`java.util.Comparator<T>`接口的使用场景代码，其中的抽象方法定义为：

- `public abstract int compare(T o1, T o2);`

当需要对一个对象数组进行排序时，`Arrays.sort`方法需要一个`Comparator`接口实例来指定排序的规则。假设有一个`Person`类，含有`String name`和`int age`两个成员变量：

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

    // 省略构造器、toString方法与Getter Setter
}
```

传统写法

如果使用传统的代码对`Person[]`数组进行排序，写法如下：

```
public class Demo06Comparator {
    public static void main(String[] args) {
        // 本来年龄乱序的对象数组
        Person[] array = { new Person("古力娜扎", 19),          new Person("迪丽热巴", 18),             new Person("马尔扎哈", 20) };

        // 匿名内部类
        Comparator<Person> comp = new Comparator<Person>() {
            @Override
            public int compare(Person o1, Person o2) {
                return o1.getAge() - o2.getAge();
```

```
            }
        };
        Arrays.sort(array, comp); // 第二个参数为排序规则，即Comparator接口实例

        for (Person person : array) {
            System.out.println(person);
        }
    }
}
```

这种做法在面向对象的思想中，似乎也是“理所当然”的。其中`Comparator`接口的实例（使用了匿名内部类）代表了“按照年龄从小到大”的排序规则。

代码分析

下面我们来搞清楚上述代码真正要做什么事情。

- 为了排序，`Arrays.sort`方法需要排序规则，即`Comparator`接口的实例，抽象方法`compare`是关键；
- 为了指定`compare`的方法体，**不得不**需要`Comparator`接口的实现类；
- 为了省去定义一个`ComparatorImpl`实现类的麻烦，**不得不**使用匿名内部类；
- 必须覆盖重写抽象`compare`方法，所以方法名称、方法参数、方法返回值**不得不**再写一遍，且不能写错；
- 实际上，**只有参数和方法体才是关键。**

Lambda写法

```
public class Demo07ComparatorLambda {
    public static void main(String[] args) {
        Person[] array = {
            new Person("古力娜扎", 19),
            new Person("迪丽热巴", 18),
            new Person("马尔扎哈", 20) };

        Arrays.sort(array, (Person a, Person b) -> {
            return a.getAge() - b.getAge();
        });

        for (Person person : array) {
            System.out.println(person);
        }
    }
}
```

省略格式:

省略规则

在Lambda标准格式的基础上，使用省略写法的规则为：

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**，则小括号可以省略；
3. 如果大括号内**有且仅有一个语句**，则无论是否有返回值，都可以省略大括号、return关键字及语句分号。

> 备注：掌握这些省略规则后，请对应地回顾本章开头的多线程案例。

可推导即可省略

Lambda强调的是“做什么”而不是“怎么做”，所以凡是可以根据上下文推导得知的信息，都可以省略。例如上例还可以使用Lambda的省略写法：

```
Runnable接口简化：
1. () -> System.out.println("多线程任务执行！")
Comparator接口简化：
2. Arrays.sort(array, (a, b) -> a.getAge() - b.getAge());
```

4.4 Lambda的前提条件

Lambda的语法非常简洁，完全没有面向对象复杂的束缚。但是使用时有几个问题需要特别注意：

1. 使用Lambda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。
2. 使用Lambda必须具有**上下文推断。** 也就是方法的参数或局部变量类型必须为Lambda对应的接口类型，才能使用Lambda作为该接口的实例。

> 备注：有且仅有一个抽象方法的接口，称为“**函数式接口**”。

第五章 Stream

在Java 8中，得益于Lambda所带来的函数式编程，引入了一个**全新的Stream概念**，用于解决已有集合类库既有的弊端。

5.1 引言

传统集合的多步遍历代码

几乎所有的集合（如`Collection`接口或`Map`接口等）都支持直接或间接的遍历操作。而当我们需要对集合中的元素进行操作的时候，除了必需的添加、删除、获取外，最典型的就是集合遍历。例如：

```
public class DemoO1ForEach {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("张无忌");
        list.add("周芷若");
        list.add("赵敏");
        list.add("张强");
        list.add("张三丰");
        for (String name : list) {
            System.out.println(name);
        }
    }
}
```

这是一段非常简单的集合遍历操作：对集合中的每一个字符串都进行打印输出操作。

循环遍历的弊端

Java 8的Lambda让我们可以更加专注于**做什么**（What），而不是**怎么做**（How），这点此前已经结合内部类进行了对比说明。现在，我们仔细体会一下上例代码，可以发现：

- for循环的语法就是“**怎么做**”
- for循环的循环体才是“**做什么**”

为什么使用循环？因为要进行遍历。但循环是遍历的唯一方式吗？遍历是指每一个元素逐一进行处理，**而并不是从第一个到最后一个顺次处理的循环**。前者是目的，后者是方式。

试想一下，如果希望对集合中的元素进行筛选过滤：

1. 将集合A根据条件一过滤为**子集B**；
2. 然后再根据条件二过滤为**子集C**。

那怎么办？在Java 8之前的做法可能为：

```
public class Demo02NormalFilter {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("张无忌");
        list.add("周芷若");
        list.add("赵敏");
        list.add("张强");
        list.add("张三丰");

        List<String> zhangList = new ArrayList<>();
        for (String name : list) {
            if (name.startsWith("张")) {
                zhangList.add(name);
            }
        }

        List<String> shortList = new ArrayList<>();
        for (String name : zhangList) {
            if (name.length() == 3) {
                shortList.add(name);
            }
        }

        for (String name : shortList) {
            System.out.println(name);
        }
    }
}
```

这段代码中含有三个循环，每一个作用不同：

1. 首先筛选所有姓张的人；
2. 然后筛选名字有三个字的人；
3. 最后进行对结果进行打印输出。

每当我们需要对集合中的元素进行操作的时候，总是需要进行循环、循环、再循环。这是理所当然的么？**不是。**循环是做事情的方式，而不是目的。另一方面，使用线性循环就意味着只能遍历一次。如果希望再次遍历，只能再使用另一个循环从头开始。

那，Lambda的衍生物Stream能给我们带来怎样更加优雅的写法呢？

Stream的更优写法

下面来看一下借助Java 8的Stream API，什么才叫优雅：

```
public class Demo03StreamFilter {
```

```
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("张无忌");
        list.add("周芷若");
        list.add("赵敏");
        list.add("张强");
        list.add("张三丰");

        list.stream()
            .filter(s -> s.startsWith("张"))
            .filter(s -> s.length() == 3)
            .forEach(System.out::println);
    }
}
```

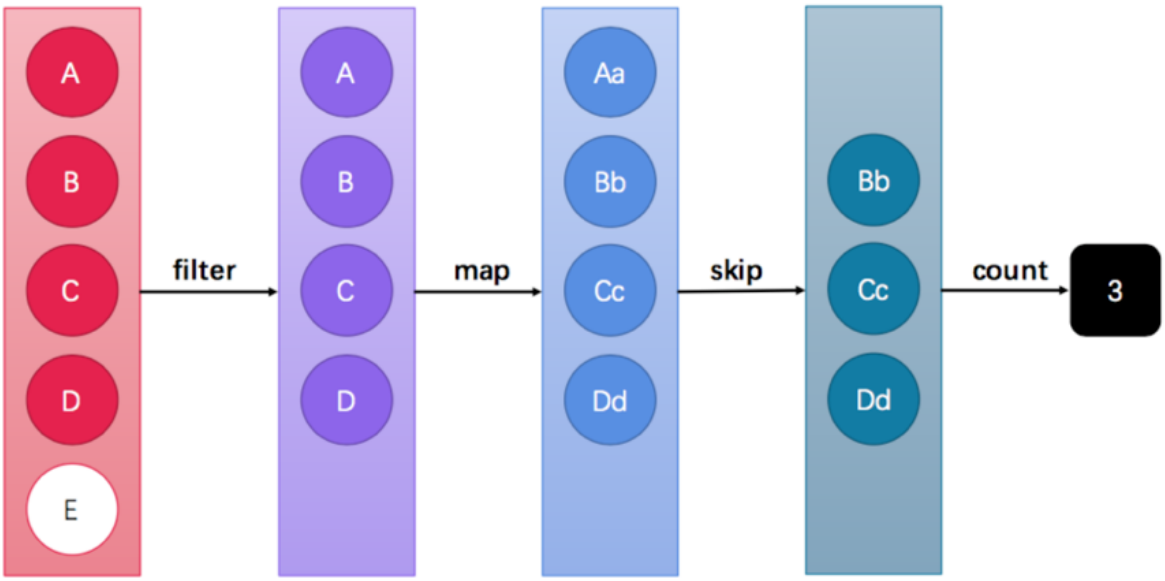
直接阅读代码的字面意思即可完美展示无关逻辑方式的语义：**获取流、过滤姓张、过滤长度为3、逐一打印**。代码中并没有体现使用线性循环或是其他任何算法进行遍历，我们真正要做的事情内容被更好地体现在代码中。

5.2 流式思想概述

注意：请暂时忘记对传统IO流的固有印象！

整体来看，流式思想类似于工厂车间的“**生产流水线**”。

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，然后再按照方案去执行它。



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。

> 备注：“Stream流”其实是一个集合元素的函数模型，它并不是集合，也不是数据结构，其本身并不存储任何元素（或其地址值）。

5.3 获取流方式

`java.util.stream.Stream<T>`是Java 8新加入的最常用的流接口。（这并不是一个函数式接口。）

获取一个流非常简单，有以下几种常用的方式：

- 所有的`Collection`集合都可以通过`stream`默认方法获取流；
- `Stream`接口的静态方法`of`可以获取数组对应的流。

方式1：根据Collection获取流

首先，`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。

```
import java.util.*;
import java.util.stream.Stream;

public class Demo04GetStream {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        // ...
        Stream<String> stream1 = list.stream();

        Set<String> set = new HashSet<>();
        // ...
        Stream<String> stream2 = set.stream();

        Vector<String> vector = new Vector<>();
        // ...
        Stream<String> stream3 = vector.stream();
    }
```

```
}
```

方式2：根据Map获取流

`java.util.Map`接口不是`Collection`的子接口，且其K-V数据结构不符合流元素的单一特征，所以获取对应的流需要分key、value或entry等情况：

```
import java.util.HashMap;
import java.util.Map;
import java.util.stream.Stream;

public class Demo05GetStream {
    public static void main(String[] args) {
        Map<String, String> map = new HashMap<>();
        // ...
        Stream<String> keyStream = map.keySet().stream();
        Stream<String> valueStream = map.values().stream();
        Stream<Map.Entry<String, String>> entryStream = map.entrySet().stream();
    }
}
```

方式3：根据数组获取流

如果使用的不是集合或映射而是数组，由于数组对象不可能添加默认方法，所以`Stream`接口中提供了静态方法`of`，使用很简单：

```
import java.util.stream.Stream;

public class Demo06GetStream {
    public static void main(String[] args) {
        String[] array = { "张无忌", "张翠山", "张三丰", "张一元" };
        Stream<String> stream = Stream.of(array);
    }
}
```

> 备注：`of`方法的参数其实是一个可变参数，所以支持数组。

5.4 常用方法

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`接口自身类型的方法，因此不再支持类似`StringBuilder`那样的链式调用。本小节中，终结方法包括`count`和`forEach`方法。
- **非终结方法**：返回值类型仍然是`Stream`接口自身类型的方法，因此支持链式调用。（除了终结方法外，其余方法均为非终结方法。）

函数拼接与终结方法

在上述介绍的各种方法中，凡是返回值仍然为`Stream`接口的为**函数拼接方法**，它们支持链式调用；而返回值不再为`Stream`接口的为**终结方法**，不再支持链式调用。如下表所示：

方法名	方法作用	方法种类	是否支持链式调用
count	统计个数	终结	否
forEach	逐一处理	终结	否
filter	过滤	函数拼接	是
limit	取用前几个	函数拼接	是
skip	跳过前几个	函数拼接	是
map	映射	函数拼接	是
concat	组合	函数拼接	是

> 备注：本小节之外的更多方法，请自行参考API文档。

forEach：逐一处理

虽然方法名字叫`forEach`，但是与for循环中的“for-each”昵称不同，该方法**并不保证元素的逐一消费动作在流中是被有序执行的。**

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

该方法接收一个`Consumer`接口函数，会将每一个流元素交给该函数进行处理。例如：

```
import java.util.stream.Stream;

public class Demo12StreamForEach {
    public static void main(String[] args) {
        Stream<String> stream = Stream.of("张无忌", "张三丰", "周芷若");
        stream.forEach(s->System.out.println(s));
    }
}
```

count：统计个数

正如旧集合`Collection`当中的`size`方法一样，流提供`count`方法来数一数其中的元素个数：

```
long count();
```

该方法返回一个long值代表元素个数（不再像旧集合那样是int值）。基本使用：

```
public class Demo09StreamCount {
    public static void main(String[] args) {
        Stream<String> original = Stream.of("张无忌", "张三丰", "周芷若");
        Stream<String> result = original.filter(s -> s.startsWith("张"));
        System.out.println(result.count()); // 2
    }
}
```

filter：过滤

可以通过`filter`方法将一个流转换成另一个子集流。方法声明：

```
Stream<T> filter(Predicate<? super T> predicate);
```

该接口接收一个`Predicate`函数式接口参数（可以是一个Lambda或方法引用）作为筛选条件。

基本使用

Stream流中的`filter`方法基本使用的代码如：

```
public class Demo07StreamFilter {
    public static void main(String[] args) {
        Stream<String> original = Stream.of("张无忌", "张三丰", "周芷若");
        Stream<String> result = original.filter(s -> s.startsWith("张"));
    }
}
```

在这里通过Lambda表达式来指定了筛选的条件：必须姓张。

limit：取用前几个

`limit`方法可以对流进行截取，只取用前n个。方法签名：

```
Stream<T> limit(long maxSize);
```

参数是一个long型，如果集合当前长度大于参数则进行截取；否则不进行操作。基本使用：

```
import java.util.stream.Stream;

public class Demo10StreamLimit {
    public static void main(String[] args) {
        Stream<String> original = Stream.of("张无忌", "张三丰", "周芷若");
        Stream<String> result = original.limit(2);
        System.out.println(result.count()); // 2
    }
}
```

skip：跳过前几个

如果希望跳过前几个元素，可以使用`skip`方法获取一个截取之后的新流：

```
Stream<T> skip(long n);
```

如果流的当前长度大于n，则跳过前n个；否则将会得到一个长度为0的空流。基本使用：

```
import java.util.stream.Stream;

public class Demo11StreamSkip {
    public static void main(String[] args) {
        Stream<String> original = Stream.of("张无忌", "张三丰", "周芷若");
        Stream<String> result = original.skip(2);
        System.out.println(result.count()); // 1
    }
}
```

map：映射

如果需要将流中的元素映射到另一个流中，可以使用`map`方法。方法签名：

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper);
```

该接口需要一个`Function`函数式接口参数，可以将当前流中的T类型数据转换为另一种R类型的流。

基本使用

Stream流中的`map`方法基本使用的代码如：

```
import java.util.stream.Stream;

public class Demo08StreamMap {
    public static void main(String[] args) {
        Stream<String> original = Stream.of("10", "12", "18");
        Stream<Integer> result = original.map(s->Integer.parseInt(s));
    }
}
```

这段代码中，`map`方法的参数通过方法引用，将字符串类型转换成为了int类型（并自动装箱为`Integer`类对象）。

concat：组合

如果有两个流，希望合并成为一个流，那么可以使用`Stream`接口的静态方法`concat`：

```
static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)
```

> 备注：这是一个静态方法，与`java.lang.String`当中的`concat`方法是不同的。

该方法的基本使用代码如：

```
import java.util.stream.Stream;

public class Demo12StreamConcat {
    public static void main(String[] args) {
        Stream<String> streamA = Stream.of("张无忌");
        Stream<String> streamB = Stream.of("张翠山");
        Stream<String> result = Stream.concat(streamA, streamB);
    }
}
```

5.5 Stream综合案例

现在有两个`ArrayList`集合存储队伍当中的多个成员姓名，要求使用传统的for循环（或增强for循环）**依次**进行以下若干操作步骤：

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3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；
7. 打印整个队伍的Person对象信息。

两个队伍（集合）的代码如下：

```
public class DemoArrayListNames {
    public static void main(String[] args) {
        List<String> one = new ArrayList<>();
        one.add("迪丽热巴");
        one.add("宋远桥");
        one.add("苏星河");
        one.add("老子");
        one.add("庄子");
        one.add("孙子");
        one.add("洪七公");

        List<String> two = new ArrayList<>();
        two.add("古力娜扎");
        two.add("张无忌");
        two.add("张三丰");
        two.add("赵丽颖");
        two.add("张二狗");
        two.add("张天爱");
        two.add("张三");
        // ....
    }
}
```

而`Person`类的代码为：

```
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;
    }
}
```

传统方式

使用for循环，示例代码:

```
public class DemoArrayListNames {
    public static void main(String[] args) {
        List<String> one = new ArrayList<>();
        // ...

        List<String> two = new ArrayList<>();
        // ...

        // 第一个队伍只要名字为3个字的成员姓名；
        List<String> oneA = new ArrayList<>();
        for (String name : one) {
            if (name.length() == 3) {
                oneA.add(name);
            }
        }

        // 第一个队伍筛选之后只要前3个人；
        List<String> oneB = new ArrayList<>();
        for (int i = 0; i < 3; i++) {
            oneB.add(oneA.get(i));
        }

        // 第二个队伍只要姓张的成员姓名；
        List<String> twoA = new ArrayList<>();
        for (String name : two) {
            if (name.startsWith("张")) {
                twoA.add(name);
            }
        }

        // 第二个队伍筛选之后不要前2个人；
        List<String> twoB = new ArrayList<>();
        for (int i = 2; i < twoA.size(); i++) {
            twoB.add(twoA.get(i));
        }

        // 将两个队伍合并为一个队伍；
        List<String> totalNames = new ArrayList<>();
        totalNames.addAll(oneB);
        totalNames.addAll(twoB);

        // 根据姓名创建Person对象；
        List<Person> totalPersonList = new ArrayList<>();
```

```
        for (String name : totalNames) {
            totalPersonList.add(new Person(name));
        }

        // 打印整个队伍的Person对象信息。
        for (Person person : totalPersonList) {
            System.out.println(person);
        }
    }
}
```

运行结果为：

```
Person{name='宋远桥'}
Person{name='苏星河'}
Person{name='洪七公'}
Person{name='张二狗'}
Person{name='张天爱'}
Person{name='张三'}
```

Stream方式

等效的Stream流式处理代码为：

```
public class DemoStreamNames {
    public static void main(String[] args) {
        List<String> one = new ArrayList<>();
        // ...

        List<String> two = new ArrayList<>();
        // ...

        // 第一个队伍只要名字为3个字的成员姓名；
        // 第一个队伍筛选之后只要前3个人；
        Stream<String> streamOne = one.stream().filter(s -> s.length() ==
3).limit(3);

        // 第二个队伍只要姓张的成员姓名；
        // 第二个队伍筛选之后不要前2个人；
        Stream<String> streamTwo = two.stream().filter(s ->
s.startsWith("张")).skip(2);

        // 将两个队伍合并为一个队伍；
        // 根据姓名创建Person对象；
        // 打印整个队伍的Person对象信息。
        Stream.concat(streamOne, streamTwo).map(s-> new Person(s)).forEach(s-
>System.out.println(s));
    }
}
```

运行效果完全一样：

```
Person{name='宋远桥'}
Person{name='苏星河'}
Person{name='洪七公'}
Person{name='张二狗'}
Person{name='张天爱'}
Person{name='张三'}
```

5.6 收集Stream结果

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?

收集到集合中

Stream流提供`collect`方法，其参数需要一个`java.util.stream.Collector<T,A, R>`接口对象来指定收集到哪种集合中。幸运的是，`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。

下面是这两个方法的基本使用代码：

```
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Demo15StreamCollect {
    public static void main(String[] args) {
        Stream<String> stream = Stream.of("10", "20", "30", "40", "50");
        List<String> list = stream.collect(Collectors.toList());
        Set<String> set = stream.collect(Collectors.toSet());
    }
}
```

收集到数组中

Stream提供`toArray`方法来将结果放到一个数组中，由于泛型擦除的原因，返回值类型是Object[]的：

```
Object[] toArray();
```

其使用场景如：

```
import java.util.stream.Stream;

public class Demo16StreamArray {
    public static void main(String[] args) {
        Stream<String> stream = Stream.of("10", "20", "30", "40", "50");
        Object[] objArray = stream.toArray();
    }
}
```