day07_等待唤醒案例、线程池、Lambda表达式

一.等待唤醒案例

在模拟卖票的案例中，我们使用 "同步机制" 来解决 "多个线程操作同一份共享数据产生的线程安全问题"，当时多个线程的线程任务是一样的，即多个线程对共享数据的操作是一样的。同步机制仅仅是为了解决线程安全问题。

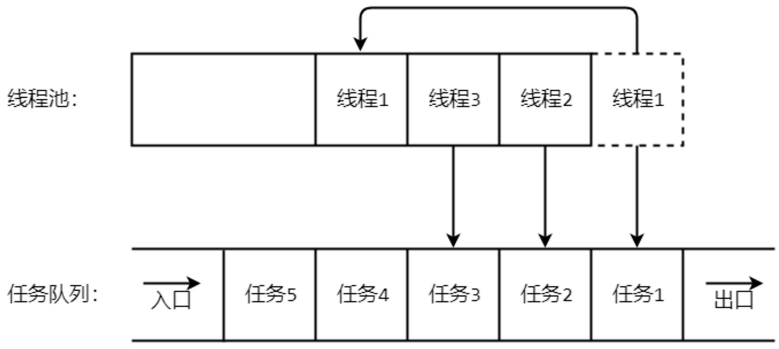
当多个线程处理同一份共享数据(资源)，但是线程任务不同时，就需要线程之间的协调通信来解决共享数据的线程安全问题，我们使用 "等待唤醒机制" 来实现多个线程之间的通信。等待唤醒机制不仅仅是解决线程安全问题，更多的是让多个线程之间有规律地执行代码，达到灵活处理共享资源的目的。

等待唤醒机制其实就是经典的 "生产者与消费者" 问题，下面我们以一个案例来演示，如何使用 "等待唤醒机制" 来实现多个线程之间的协调通信，以此来对共享资源进行灵活地处理。

二.线程池

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被终止了，这样频繁地创建、销毁线程会降低系统效率，消耗系统资源。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任务呢？在 Java 中可以通过线程池来达到这样的效果。线程池：其实就是一个可以容纳多个线程的容器(集合)，在线程池中的线程可以被重复利用。

1.线程池的底层原理



线程池的底层就是一个用来存储线程的LinkedList集合

创建一个线程池集合：`LinkedList<Thread> linked = new LinkedList<>();`

添加线程到集合中：

```
linked.add(new Thread(xxx, "线程1"));
linked.add(new Thread(xxx, "线程2"));
linked.add(new Thread(xxx, "线程3"));
```

当任务队列中有任务时：

调用 `linked.removeFirst();` 从集合中移除并返回第一个线程，用来执行任务1。

线程1被移除后会调用 `linked.addLast();` 将线程1再次添加到集合末尾进行重复利用。

此时线程2就变成了集合中第一个线程：再次执行上一步的操作，如此往复...

合理利用线程池能够带来以下好处：

- 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。
- 提高响应速度：当任务到达时，任务可以不需要等到线程创建就能立即执行。
- 提高线程的可管理性：可以根据系统的承受能力，调整线程池中的线程数目，防止空闲线程消耗过多的内存

2.线程池的使用

三.Lambda 表达式

1.函数式编程思想

面向对象的编程思想过分强调 "必须通过对象的形式来做事情"，即找一个能解决这件事情的对象，通过调用对象的方法来完成任务。然而函数式编程思想则尽量忽略面向对象的复杂语法，强调去做什么事情，谁去做的、怎么做的并不重要，只注重结果、不注重过程。

2.Lambda 表达式的产生背景及用法

首先我们来回顾一下，创建多线程程序的第二种方式——利用 Runnable 接口的实现类来创建一个线程对象。由于 Thread 类的构造方法需要 Runnable 接口类型作为参数，所以在 new 一个 Thread 类的线程对象时，我们不得不给其传递一个 Runnable 接口的实现类对象。所以我们首先要创建一个 Runnable 接口的实现类，并重写接口中的 run( ) 方法来设置线程任务，然后再 new 一个该实现类的对象将其传递进 Thread 类构造方法。后面我们为了省去定义一个 RunnableImpl 实现类的麻烦，不得不使用匿名内部类来简化代码的书写。代码如下：

```java
public class RunnableImpl implements Runnable {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
}
public class DemoMain {
    public static void main(String[] args) {

        // 使用Runnable接口的实现类创建线程
        RunnableImpl r = new RunnableImpl();
        new Thread(r).start();

        // 使用匿名内部类简化代码书写
        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName());
            }
        }).start();

    }
}
```

我们真的想创建一个 RunnableImpl 实现类对象、或匿名内部类对象吗？并不是，我们只是为了给 Thread 线程对象传递一个线程任务而已，即将 run( ) 方法体内的代码传递给 Thread 类让其知晓。所以我们只是为了传递一段代码，然而由于受限于面向对象的语法，而不得不要多创建一个实现类及其对象，并且方法名称、方法参数、方法返回值不得不再写一遍，就造成了很多的代码冗余。而似乎只有 run( ) 方法的方法体才是关键所在。

所以在 JDK8 中加入了 Lambda 表达式的重量级新特性，给我们打开了函数式编程思想的大门。借助 Lambda 表达式的全新语法，我们可以使用更简洁的方式来代替上述接口实现类对象的创建过程、或匿名内部类的复杂语法。Lambda 表达式的标准语法格式如下：

```
(参数列表) -> {一些重写方法的代码};
```

- ( )：接口中抽象方法的参数列表，没有参数就空着，有参数就写出参数，并且多个参数使用逗号分隔
- ->：传递的意思，把参数传递给方法体{ }
- { }：重写接口中抽象方法的方法体

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

        new Thread(() -> {
            System.out.println(Thread.currentThread().getName());
        }).start();

    }
}
```

所以 Lambda 表达式就相当于是接口的一个实现类对象，这样说其实是不太对的。Lambda 表达式是以 "传递接口中抽象方法体代码" 的形式，来代替 "使用接口时创建实现类对象的繁琐过程"。但是使用 Lambda 表达式有以下前提：

- 只有当接口中的抽象方法有且仅有一个时，才可以使用 Lambda 表达式。这种接口称为 "函数式接口"
- Lambda 表达式的必须要由相应接口类型的方法参数、局部变量来接收 (相当于是返回了接口的一个实现类对象)

3.Lambda 表达式的练习

4.Lambda 表达式的省略格式

在Lambda 表达式的标准格式中，凡是可以根据上下文(接口中的抽象方法)推导出来的信息，都可以省略。省略规则如下：

- (参数列表)：小括号中参数列表的数据类型，可以省略不写
- (参数列表)：小括号中的参数如果只有一个，那么类型和 ( ) 都可以省略
- {一些代码}：如果 { } 中的代码只有一条语句，那么可以省略 { } 以及该语句后的分号。若这一条语句中有 return ，那么 { } 、return、分号可以一起省略。