福建師範大學
FUJIAN NORMAL UNIVERSITY

Java语言

学　　期：2025-2026-1

教　　师：赵珊珊

第11章 线程

第11章 线程

- 11.1线程概述
- 11.2 线程机制
- 11.3线程的生命周期及状态转换
- 11.4线程的调度
- 11.5多线程同步
- 11.6多线程通信
- 11.7线程组和未处理的异常
- 11.8线程池

11.1线程概述

- 多线程是实现并发机制的一种有效手段。进程和线程一样，都是实现并发的一个基本单位。线程是比进程更小的执行单位，线程是在进程的基础之上进行的进一步划分。所谓多线程，是指一个进程在执行过程中可以产生多个更小的程序单元，这些更小的单元称为线程

11.1线程概述

- 11.1.1进程
- 在一个操作系统中，每个独立执行的程序都可以称为一个进程。

11.1线程概述

- 11.1.2线程
- 操作系统可以同时执行多个任务，每个任务就是进程，进程可以同时执行多个任务，每个任务就是线程。

11.2 线程机制

- Java为多线程开发提供了非常优秀的技术支持，在Java中，可以通过三种方式来实现多线程：
- 第一种是继承Thread类，重写run()方法；
- 第二种是实现Runnable接口，重写run()方法；
- 第三种是实现Callable接口，重写call()方法，并使用Future来获取call()方法的返回结果。

11.2 线程机制

- 11.2.1Thread类创建线程
- Java提供了Thread类代表线程，它位于java.lang包中，下面介绍Thread类创建并启动多线程的步骤，具体步骤如下：
  - 定义Thread类的子类，并重写run()方法，run()方法称为线程执行体。
  - 创建Thread子类的实例，即创建了线程对象。
  - 调用线程对象start()方法启动线程。

方法声明	功能描述
start()	启动线程，使其进入就绪状态，最终JVM会自动调用其run()方法
run()	线程的任务执行体，包含线程要执行的代码。通常需要重写此方法
setName(), getName()	设置和获取线程的名称
Thread.State getState()	返回该线程的状态
isAlive()	测试线程是否处于活动状态
sleep(long millis)	让当前正在执行的线程休眠指定毫秒数

11.2 线程机制

【Thread逐个创建线程对象与创建线程对象数组】

```
2  public class Thread0 {
3
4      public static void main(String[] args) {
5          // TODO Auto-generated method stub
6
7      }
8
9  }
10
```

Problems Javadoc Declaration Console ×

<terminated> ThreadExam [Java Application] D:\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.7.v20230425-1502\jre\bin\javaw.exe (2025

11.2 线程机制

- 11.2.2Runnable接口创建线程
- Java只支持单继承，一个类只能有一个父类，继承Thread类后，就不能再继承其他类，为了解决这个问题，可以用实现Runnable接口的方式创建多线程，下面介绍实现Runnable接口创建并启动多线程的具体步骤：
  - 定义Runnable接口实现类，并重写run()方法。
  - 创建Runnable实现类的示例，并将实例对象传给Thread类的target来创建线程对象。
  - 调用线程对象的start()方法启动线程。

11.2 线程机制

【Runnable接口创建对象数组】

【11.2的runnable和callable程序代码可以如何简化？】

```
2  public class Thread0 {
3      static int floor =3;
4      static String[] str=new String[floor];
5      public static void main(String[] args) {
6          MyThread[] mt =new MyThread[floor];
7          for(int i=0;i<floor;i++) {
8              str[i]=(i+1)+"楼";
9              mt[i]=new MyThread(str[i]);
10             mt[i].start();
11         }
12     }
13 }
14 class MyThread extends Thread{
15     MyThread(String str){
16         super(str);
17     }
18     public void run() {
19         int i=0;
20         while(i++<5)
21             System.out.println(Thread.currentThread().getName()+"的run方法正在运行。");
22     }
23 }
```

Problems @ Javadoc Declaration Console ×

<terminated> Thread0 [Java Application] D:\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.7.v20230425-1502\jre\bin\javaw.exe (2025年10

11.2 线程机制

- 11.2.3Callable接口和Future接口创建线程
- Java提供了Callable接口，接口内有一个call()方法可以作为线程执行体，call()方法有返回值且可以抛出异常。下面介绍实现Callable接口创建并启动多线程的具体步骤：
  - 定义Callable接口实现类，指定返回值类型，并重写call()方法。
  - 创建Callable实现类的实例，使用FutureTask类来包装Callable对象，该FutureTask对象封装了该Callable对象的call()方法的返回值。
  - 使用FutureTask对象作为Thread对象的target创建并启动新线程。
  - 调用FutureTask对象的get()方法来获得子线程执行结束后的返回值。

注：Callable接口不是Runnable接口的子接口，不能直接作为Thread的target

11.2 线程机制

```
1
2  public class CallableExam {
3
4      public static void main(String[] args) {
5          // TODO Auto-generated method stub
6
7      }
8
9  }
10
```

Problems Javadoc Declaration Console ×

<terminated> Thread0 [Java Application] D:\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.7.v20230425-1502\jre\bin\javaw.exe (2025年10

11.3线程的生命周期及状态转换

- Java中任何对象都有生命周期，线程也有自己的生命周期。当Thread对象创建完成时，线程的生命周期就开始了。当线程任务中代码正常执行完毕或者有未捕获的异常或错误时，线程的生命周期就结束了。
- 线程有新建（New）、就绪（Runnable）、运行（Running）、阻塞（Blocked）和死亡（Terminated）五种状态（图11-4)

方法/机制	所属类	是否释放已持有的同步锁	唤醒条件	实用性评价
sleep(long millis)	Thread	否	休眠时间到	☆☆☆☆☆ 常用，用于定时延迟
同步锁 (Synchronized)	语言关键字	(竞争失败被动进入)	成功获取到锁	☆☆☆☆☆ 核心同步机制，必需
wait()/ notify()	Object	是	其他线程调用 notify/notifyAll	☆☆☆☆ 重要，用于线程间条件协作
join()	Thread	是 (底层基于wait)	目标线程执行完毕	☆☆☆☆ 常用，用于线程汇合等待
suspend()/ resume()	Thread	否 (易导致死锁)	其他线程调用 resume	⚠ 已废弃，严禁使用

11.4线程的调度

- 程序中的多个线程是并发执行的，但并不是同一时刻执行，某个线程想要执行必须要获得CUP的使用权。Java虚拟机会按照特定的机制为程序中的每个线程分配CPU的使用权，这种机制被称为线程的调度。
- 分时调度模式：所有线程轮流使用CPU，平均分配每个线程占用CPU的时间片；
- 抢占式调度模式（默认模式）：可运行池中所有就绪状态的线程争抢CPU的使用权，优先级高的线程获取CPU执行权的概率大于优先级低的线程

11.4线程的调度

- 11.4.1线程的优先级
- 所有处于就绪状态的线程根据优先级存放在可运行池中，优先级低的线程运行机会较少，优先级高的线程运行机会更多。
- Thread类的setPriority(int newPriority)方法和getPriority()方法分别用于设置优先级和读取优先级。
- 优先级用整数表示，取值范围1~10，除了直接用数字表示线程的优先级，还可以用Thread类中提供的三个静态常量来表示线程的优先级
- MAX_PRIORITY（10）, NORM_PRIORITY（5）, MIN_PRIORITY（1）

11.4线程的调度

```
public class PriorityDemo {
    public static void main(String[] args) {
        myThread t1 = new myThread("优先级低的线程");
        myThread t2 = new myThread("默认优先级的线程");
        myThread t3 = new myThread("优先级高的线程");
        t1.setPriority(1);
        t2.setPriority(Thread.NORM_PRIORITY);
        t3.setPriority(10);
        t1.start();
        t2.start();
        t3.start();
    }
}
class myThread extends Thread {
    public myThread(String name) {
        super(name);
    }
    public void run() {
        for (int i = 0; i < 5; i++)
            System.out.println(Thread.currentThread().getName() + "正在输出第" +(i+1)+"次");
    }
}
```

```
优先级低的线程正在输出第1次
优先级高的线程正在输出第1次
优先级高的线程正在输出第2次
默认优先级的线程正在输出第1次
优先级高的线程正在输出第3次
优先级低的线程正在输出第2次
优先级高的线程正在输出第4次
默认优先级的线程正在输出第2次
优先级高的线程正在输出第5次
优先级低的线程正在输出第3次
默认优先级的线程正在输出第3次
默认优先级的线程正在输出第4次
优先级低的线程正在输出第4次
默认优先级的线程正在输出第5次
优先级低的线程正在输出第5次
```

11.4线程的调度

- 11.4.2线程休眠
- 线程优先级高的线程有更大的概念优先执行，而优先级低的线程可能会后执行。如果想人为控制线程执行顺序，使正在执行的线程暂停，将CPU使用权让给其他线程，可以使用sleep()方法。

```
1  import java.text.SimpleDateFormat;
2  import java.util.Date;
3  public class SleepDemo {
4          public static void main(String[] args) throws InterruptedException{
5          AThread thread=new AThread();
6          thread.start();
7      }
8  }
9  class AThread extends Thread{
10     public void run(){
11         for (int i = 0; i < 5; i++) {
12             System.out.println("当前时间："
13                     + new SimpleDateFormat("hh:mm:ss").format(new Date()));
14             try {
15                 Thread.sleep(2000);
16             } catch (InterruptedException e) {
17                 e.printStackTrace();
18             }
19         }
20     }
21 }
22
```

Problems @ Javadoc Declaration Console ×

<terminated> SleepDemo [Java Application] D:\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre

11.4线程的调度

- 11.4.3线程让步
- 前面讲解了使用sleep()方法使线程阻塞，Thread类还提供一个yield()方法，它与sleep()方法类似，它也可以让当前正在执行的线程暂停，但yield()方法不会使线程阻塞，只是将线程转换为就绪状态，也就是让当前线程暂停一下，线程调度器重新调度一次，有可能还会将暂停的程序调度出来继续执行，这也称为线程让步。

```
public class YieldDemo {
    public static void main(String[] args) {
        BThread thread1=new BThread("threadA");
        BThread thread2=new BThread("threadB");
        thread1.start();
        thread2.start();
    }
}
class BThread extends Thread{
    BThread(String name){
        super(name);
    }
    public void run(){
        for (int i = 0; i < 6; i++) {
            System.out.println(Thread.currentThread().getName()+"---"+i);
            if(i==2){
                System.out.print("线程让步:");
                Thread.yield();
            }
        }
    }
}
```

```
threadA---0
threadA---1
threadA---2
threadB---0
线程让步:threadB---1
threadA---3
threadB---2
线程让步:threadA---4
threadB---3
threadB---4
threadB---5
threadA---5
```

11.4线程的调度

- 11.4.4线程插队
- 现实生活中经常碰到“插队”的情况，同样在Thread类中也提供了一个join()方法来实现这个功能。当在某个线程中调用其他线程的join()方法时，调用的线程将被阻塞，直到被join()方法加入的线程执行完成后它才会继续执行。
- 因此join()方法不会产生线程冲突或运行时冲突。

```
1  public class TestJoin {
2      public static void main(String[] args) throws Exception {
3          CThread st = new CThread();      // 创建CThread4实例
4          Thread t = new Thread(st, "插队线程"); // 创建并开启线程
5          t.start();
6          for (int i = 1; i < 4; i++) {
7              System.out.println(Thread.
8                      currentThread().getName() + ":" + i);//main方法线程
9              if (i == 2)
10                 t.join();                    // 线程插队
11         }
12     }
13 }
14 class CThread implements Runnable {
15     public void run() {                      // 重写run()方法
16         for (int i = 1; i < 6; i++) {
17             System.out.println(Thread.
18                     currentThread().getName() + ":" + i);
19         }
20     }
21 }
22
```

Problems Javadoc Declaration Console ×

<terminated> TestJoin [Java Application] D:\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.ful

11.4线程的调度

- 11.4.5后台线程
- 线程中还有一种后台线程，它是为其他线程提供服务的，又称为“守护线程”或“精灵线程”，JVM的垃圾回收线程就是典型的后台线程。

```java
public class TestBackThread {
    public static void main(String[] args) {
        SubThread5 st1 = new SubThread5("新线程");
        st1.setDaemon(true);
        st1.start();
        for (int i = 0; i < 2; i++) {
            System.out.println(Thread.
                    currentThread().getName() + ":" + i);
        }
    }
}
class SubThread5 extends Thread {
    public SubThread5(String name) {
        super(name);
    }
    public void run() {
        for (int i = 0; i < 1000; i++) {
            if (i % 2 != 0) {
                System.out.println(Thread.
                        currentThread().getName() + ":" + i);
            }
        }
    }
}
```

<terminated> TestBackThread [Java Application]

11.4线程的调度

说明：

1.创建一个用户线程（前台线程）和一个后台线程。

2.用户线程执行一个循环，打印若干次后结束。

3.后台线程执行一个无限循环，但会因为用户线程的结束而被迫终止。

```java
public class DaemonThreadDemo {
    public static void main(String[] args) {
        // 创建一个无限循环的线程
        Thread daemonThread = new Thread(() -> {
            int count = 0;
            // 这个线程将无限循环，直到程序结束
            while (true) {
                try {
                    Thread.sleep(300); // 每0.3秒执行一次
                    count++;
                    // 判断并打印当前线程是否为守护线程
                    System.out.println("后台线程运行中... 计数: " + count +
                                       ", 是守护线程吗? " + Thread.currentThread().isDaemon());
                } catch (InterruptedException e) {e.printStackTrace();}
            }
        });
        // 在启动前将线程设置为守护线程
        daemonThread.setDaemon(true);
        // 验证设置是否成功
        System.out.println("设置后，该线程是守护线程吗? " + daemonThread.isDaemon());
       // 启动守护线程
        daemonThread.start();

        // 主线程（用户线程）执行一些工作
        System.out.println("主线程（用户线程）开始工作...");
        for (int i = 1; i <= 3; i++) {
            try {
                Thread.sleep(1000);
                System.out.println("主线程工作: " + i + "/3");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("主线程工作完成，程序即将退出。");
        // 此时，唯一的前台线程（main线程）执行完毕
        // JVM 检测到只剩下守护线程，将自动终止它们并退出程序
    }
}
```

```
<terminated> DaemonThreadDemo [Java Application] D
```

11.5多线程同步

- 11.5.1线程安全
- 关于线程安全，有一个经典的问题——窗口卖票的问题。

```
public class TestTicket {
    public static void main(String[] args) {
        Thread t[] = new Thread[3];
        for(int i=1;i<=3;i++) {
            t[i-1]=new Thread(new MyTicket(),"窗口"+i);
            t[i-1].start();
        }
    }
}
class MyTicket implements Runnable {
    private int ticket = 5;
    public void run() {
        while(true){
            if (ticket > 0) {
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName()+
                        "正在发售第" + ticket-- + "张票");
            }
        }
    }
}
```

```
<terminated> TestTicket
窗口1正在发售第5张票
窗口3正在发售第5张票
窗口2正在发售第5张票
窗口1正在发售第4张票
窗口1正在发售第3张票
窗口3正在发售第4张票
窗口2正在发售第4张票
窗口1正在发售第2张票
窗口2正在发售第3张票
窗口3正在发售第3张票
窗口1正在发售第1张票
窗口3正在发售第2张票
窗口2正在发售第2张票
窗口2正在发售第1张票
窗口3正在发售第1张票
```

11.5多线程同步

- 11.5.2同步代码块
- 为了解决线程安全的问题，Java的多线程引入了同步监视器，使用同步监视器的通用方法就是同步代码块，具体示例如下：

```
synchronized(lock){
        同步代码块
}
```

- 11.5.3同步方法
- 同步代码块可以有效解决线程的安全问题，当把共享资源的操作放在synchronized定义的区域中，便为这些操作加了同步锁。同样，在方法前面也可以使用synchronized关键字来修饰，被修饰的方法称为同步方法，它能实现和同步代码块同样的功能。

11.5多线程同步

```java
public class SynMethod {
    public static void main(String[] args) {
        Ticket1 ticket = new Ticket1();
        Thread t[] = new Thread[3];
        for(int i=0;i<3;i++) {
            t[i]=new Thread(ticket,"窗口"+(i+1));
            t[i].start();
        }
    }
}
class Ticket1 implements Runnable {
    private int ticket = 30;
    public void run() {
        while(true){
            saleTicket();
        }
    }
    private synchronized void saleTicket(){
        if (ticket > 0) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName()+
                    "正在发售第" + ticket-- + "张票");
        }
    }
}
```

```java
public class SynBlock {
    public static void main(String[] args) {
        Ticket ticket = new Ticket();
        Thread t[] = new Thread[3];
        for(int i=0;i<3;i++) {
            t[i]=new Thread(ticket,"窗口"+(i+1));
            t[i].start();
        }
    }
}
class Ticket implements Runnable {
    private int ticket = 30;
    Object lock=new Object();
    public void run() {
        while(true){
            synchronized(lock){
            if (ticket > 0) {
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread
                        "正在发售第" + ticket-- + "张票"
            }
          }
        }
    }
}
```

```
SynBlock [Java Application
窗口1正在发售第30张票
窗口1正在发售第29张票
窗口1正在发售第28张票
窗口1正在发售第27张票
窗口1正在发售第26张票
窗口1正在发售第25张票
窗口1正在发售第24张票
窗口1正在发售第23张票
窗口1正在发售第22张票
窗口1正在发售第21张票
窗口1正在发售第20张票
窗口1正在发售第19张票
窗口1正在发售第18张票
窗口1正在发售第17张票
窗口1正在发售第16张票
窗口1正在发售第15张票
窗口1正在发售第14张票
窗口1正在发售第13张票
窗口1正在发售第12张票
窗口1正在发售第11张票
窗口1正在发售第10张票
窗口1正在发售第9张票
窗口1正在发售第8张票
窗口1正在发售第7张票
窗口1正在发售第6张票
窗口1正在发售第5张票
窗口3正在发售第4张票
窗口3正在发售第3张票
窗口3正在发售第2张票
窗口2正在发售第1张票
```

11.5多线程同步

如何解决死锁问题？

flag==0与flag==1的线程使用相同的加锁顺序(o1 -> o2)

```
flag=1
flag=0
1
0
```

- 11.5.4死锁问题
- 在多线程应用中还存在死锁的问题，不同的线程分别占用对方需要的同步资源不放弃，都在等待对方放弃自己需要的同步资源，就形成了线程的死锁。

```
public int flag = 1;
private static Object o1 = new Object();
private static Object o2 = new Object();
```

```
public class DeadLock implements Runnable {

    public static void main(String[] args) {
    public void run() {
}
```

```
flag=1
flag=0
```

```
public static void main(String[] args) {
    DeadLock td1 = new DeadLock();
    DeadLock td2 = new DeadLock();
    td1.flag = 1;
    td2.flag = 0;
    new Thread(td1).start();
    new Thread(td2).start();
}
```

```
public void run() {
    System.out.println("flag=" + flag);
    if (flag == 1) {
        synchronized (o1) {
            try {
                Thread.sleep(500);
            } catch (Exception e) {
                e.printStackTrace();
            }
            synchronized (o2) {
                System.out.println("1");
            }
        }
    }
    if (flag == 0) {
        synchronized (o2) {
            try {
                Thread.sleep(500);
            } catch (Exception e) {
                e.printStackTrace();
            }
            synchronized (o1) {
                System.out.println("0");
            }
        }
    }
}
```

11.6多线程通信

- 不同的线程执行不同的任务，如果这些任务有某种联系，线程之间必须能够通信，协调完成工作
- 售货员Clerk
- 生产者Productor
- 消费者Consumer

11.6多线程通信

```java
class Clerk {   // 售货员
    private int product = 0;
    public synchronized void addProduct() {
        if (product >= 10) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        } else {
            product++;
            System.out.println("生产者生产了第" + product + "个产品");
            notifyAll();
        }
    }
    public synchronized void getProduct() {
        if (this.product <= 0) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        } else {
            System.out.println("消费者取走了第" + product + "个产品");
            product--;
            notifyAll();
        }
    }
}
```

```java
public class TestProduct {
    public static void main(String[] args) {
        Clerk clerk = new Clerk();
        Thread productorThread = new Thread(new Productor(clerk));
        Thread consumerThread = new Thread(new Consumer(clerk));
        productorThread.start();
        consumerThread.start();
    }
}
```

```java
class Productor implements Runnable {   // 生产者
    Clerk clerk;
    public Productor(Clerk clerk) {
        this.clerk = clerk;
    }
    public void run() {
        while (true) {
            try {
                Thread.sleep((int) Math.random() * 1000);
            } catch (InterruptedException e) {
            }
            clerk.addProduct();
        }
    }
}
class Consumer implements Runnable {    // 消费者
    Clerk clerk;
    public Consumer(Clerk clerk) {
        this.clerk = clerk;
    }
    public void run() {
        while (true) {
            try {
                Thread.sleep((int) Math.random() * 1000);
            } catch (InterruptedException e) {
            }
            clerk.getProduct();
        }
    }
}
```

11.7线程组和未处理的异常

- Java中使用ThreadGroup来表示线程组，它可以对一批线程进行分类管理，Java允许程序直接对线程组进行控制。用户创建的所有线程都属于指定的线程组，若未指定线程属于哪个线程组，则该线程属于默认线程组。

11.8线程池

- 程序启动一个新线程成本是比较高的，因为它涉及到要与操作系统进行交互。而使用线程池可以很好地提高性能，尤其是当程序中要创建大量生存期很短的线程时，更应该考虑使用线程池。线程池里的每一个线程代码结束后，并不会死亡，而是再次回到线程池中成为空闲状态，等待下一个对象来使用。
- 在JDK5.0之前，我们必须手动实现自己的线程池，从JDK5.0开始，Java内置支持线程池。提供一个Executors工厂类来产生线程池。