# 多线程基础

## 概述:

1.是指从软件或者硬件上实现多个线程并发执行的技术;

2.具有多线程能力的计算机因有硬件支持而能够再同一时间执行多个线程,提升性能

## 线程相关的概念

并发和并行

并行:在同一时刻,有多个指令在多个CPU上同时进行;

并发:在同一时刻,有多个指令在单个CPU上交替执行.

进程和线程

进程:是操作系统中正在运行的一个程序

1.独立性:进程是一个能独立运行的基本单位,同时也是系统分配资源和调度的独立单位;

2.动态性:进程的实质是程序的一次执行过程,进程是动态产生,动态消亡的;

3.并发性:任何进程都可以同其他进程一起并发执行.

线程:是每个应用程序中做的事情

1.单线程:一个进程如果只有一条执行路径,则称为单线程程序;

2.多线程:一个程序如果有多条执行路径,则称之为多线程程序’

## 多线程的三个实现方式

### 继承Thread类的方式进行实现

**步骤:**

**1. 定义一个类MyThread继承Thread类;**

**2. 在MyThread类中重写run()方法;**

Run();的方法体就是开启线程后执行的代码

**3. 创建MyThread类的对象;**

如果创建了多个线程对象,在调用start方法后CPU是随机进行多个线程之间的切换的,在控制台表现的形式就是不同线程对象的内容交替出现

**4. 启动线程;**

对象名.start();

**特别的:**

在多个线程对象启动线程后,CUP在不同线程之间切换,是随机的

**两个问题:**

为什么要重写run()方法?

因为run()方法用来封装被线程执行的代码

Run()和start()方法有什么区别?

run() : 封装线程执行的代码,直接调用,相当于普通方法的调用,并没有开启线程;

start() : 启动线程,然后由JVM调用此线程的run()方法

### 实现Runnable接口的方式进行实现

**步骤:**

1.定义一个类MyRunnable实现Runnable接口;

2.在MyRunnable类中重写run()方法;

3.在测试类中创建MyRunnable类的对象;

MyRunnable myRunnable1 = new MyRunnable();

MyRunnable myRunnable2 = new MyRunnable();

4.创建Thread类的对象,把MyRunnable对象作为构造方法的参数

Thread thread1 = new Thread(myRunnable1);

Thread thread2 = new Thread(myRunnable2);

5.启动线程

thread1.start();

thread2.start();

### 利用Callable和Future接口方式实现

**步骤:**

**1.定义一个类MyCallable实现Callable接口;**

public class MyCallable implements Callable<String> {

**2.在MyCallable类中重写call()方法;**

@Override

public String call() throws Exception {

for (int i = 0; i < 100; i++) {

System.out.println("给老婆买礼物" + i);

}

return "老婆很开心!";

}

}

**注意点:**重写的call()方法是有返回值的方法,返回值和类定义实现的接口泛型相同,返回值也相同,表示线程运行结束后的结果

**3.在测试类中创建MyCallable类的对象;**

MyCallable myCallable = new MyCallable();

**注意点:**线程开启之后需要执行里面的call();

**4.创建Future的实现类FutureTask,把MyCallable对象作为构造方法的参数;**

FutureTask<String> future = new FutureTask<>(myCallable);

**注意点:**可以获取线程执行完毕之后的结果,也可以作为参数传递类Thread对象

**5.创建Thread类的对象,把FutureTask对象作为构造方法的参数;**

Thread thread3 = new Thread(future);

**6.启动线程;**

thread3.start();

**注意点:**main()方法也是一条线程;

**7.再调用get方法,就可以获取线程结束之后的结果了.**

String s = future.get();

System.out.println(s);

**注意点:**get()方法得到线程运行结束后的结果,必须放在start()后面,如果放在前面,程序会卡在get()这里.

### 三种实现方式的对比

实现Runnable接口和利用Callable,Future接口实现接口,可以视为一类;

继承Thread类看成另一类.

根据不同场景选择不同的实现方式.

|  |  |  |
| --- | --- | --- |
|  | 优点 | 缺点 |
| 实现Runnable,Callable接口 | 扩展性强,实现该接口的同时还可以继承其他类 | 编程相对复杂,不能直接使用Thread类中的方法 |
| 继承Thread类 | 编程相对简单,可以直接使用Thread类中的方法 | 可扩展较差,不能  再继承其他的类 |

## Thread方法

### 获取和设置获取名字

获取线程的名字

String getName() : 返回此线程的名称,默认名称”Thread-0”;

Thread类中设置线程名字的方法

void setName(String name) : 将此线程的名称更改为参数String的name;

通过带参构造方法也可以设置线程名称,构造方法不能继承,在Thread子类中需要手动给出.

### 获得线程对象

public static Thread currentThread() : 返回当前正在执行的线程对象,返回值是一个Thread,代表得到的是Thread类的一个对象,对象是可以调方法的哦!

**则:**

有了这个静态方法,在使用Runnable实现类创建的线程和Callable,Future实现类创建线程获得线程的名字的时候,就解决了这个问题.

### sleep

线程休眠:

public static void sleep(long time) : 让线程休眠指定的时间,单位为毫秒.

### 线程的优先级

**线程调度 : 多线程的并发运行**

计算机中的CPU,在任意时刻只能执行一条机器指令,每个线程只有获得CPU的使用权才能执行代码.各个线程轮流获得CPU的使用权,分别执行各自的任务

**线程的两种调度模式**

1.分时调度模型:所有线程轮流使用CPU的使用权,平均分配每个线程占用CPU的时间;

2.抢占调度模式:有限让优先级高的线程使用CPU,如果线程的优先级相同,那么会随机选择一个,优先级高的线程获取的CPU是将会相对多一些;

**优先级相关方法**

public final void setPriority(int newPriority) : 设置线程优先级;

public final int getPriority() : 获取线程的优先级;

**注意点**

1.优先级是一个int类型的整数,取值范围[1,10],数字越大优先级越高;

2.优先级是相对而言的,就是一个概率问题,并不是绝对的先后;

3.如果用户不设置优先级,线程的优先级默认值是5(对半开);

### 守护线程/后台线程

public final void setDaemon(boolean on) : 设置守护线程,参数传true

当普通线程执行完之后,那么守护线程也没有继续运行下去你的必要了,也会跟着停止,但是不会马上停止的,因为内存运行很快,普通线程运行完以后,还会在挣扎一会,运行一下.

## 线程的安全问题

### 卖票案例实现

需求:

某电影院目前正在上映电影,共有100张票,而他又三个窗口,请设计一个程序模拟电影院卖票.

**注意:**

用Runnable接口的实现类对象为参数传递类线程(Thread)对象的时候,在这个需求中应该是三条线程传递同一个Runnable接口的实现类对象,如果创建了三个实现类对象,那就表示卖了100+100+100共300张票了

### 原因分析

在实际生活中,出票是需要时间的,所以在出售一张票的时候,需要一点时间延迟,使用sleep()方法实现.

卖票出现的问题:

1.卖重复票;

2.出现了负数票;

原因:

多线程同时执行的时候,共享数据,且存在sleep(),以及在执行每条代码语句的时候当前线程的执行权都有可能被其他线程抢夺执行权**<**而不是一条线程一但有了执行权就能执行到底**>,**由于这三个原因,导致可重复和负数票存在.

### 同步代码块

**同步:synchronized**

使用同步代码块解决上述两个问题

把多条语句操作共享数据的代码给锁起来,让任意时刻只能有一个线程执行;

Java中提供了同步到马快的方式来解决;

同步代码块格式:

Synchronized(任意对象){

多条语句操作共享数据的代码;

<说白了就是线程需要执行的共享代码>

}

特点:

默认情况下是打开的,只要有一个线程进去执行了代码,锁就会就会关闭

当线程执行 完出来,锁才会解开

同步的好处和弊端:

好处 : 解决了多线程数据安全问题;

弊端 : 当线程很多时,因为每个线程都会去判断同步**<synchronized>**上的锁,这是很耗费资源的,无形中会降低程序的运行效率.

**示例:**

private int ticket = 100;

private Object obj = new Object();

@Override

public void run() {

/\*

\* 同步代码块,是一把锁,同一时间是能有一个线程进来

\* 小括号参数暂时不清楚具体作用

\*/

synchronized (obj) {

while (ticket > 0) {

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

ticket--;

System.out.println(Thread.currentThread().getName() + "正在卖票,还剩" + ticket + "张票");

}

}

}

**注 :**

**Synchronized后面的小括号对象的作用,开启一条线程就会在内存中开辟一块空间,不同线程new的不同...**

### 锁对象唯一

同步代码块的对象就是一把锁,锁的对象应唯一,不然同步代码块就不起作用

在使用第一种方法Thread在子类创建线程的时候,锁对象在成员位置定义,那么必须要静态修饰,

在使用第二种方法实现Runnable接口创建线程的时候,锁对象在成员位置定义,则不需要使用静态修饰,因为在测试类中创建Thread对象的时候参数只有一个Runnable实现类对象.

第三种应该类似.

### 同步方法

就是把synchronized关键字加到方法上放在返回值前面

格式:

修饰符 synchronized 返回值类型 方法名(方法参数){}

同步代码块和同步方法的区别:

1.同步代码块可以锁住指定代码,同步方法是锁住方法中所有代码;

2.同步代码块可以指定锁对象,同步方法不能指定锁对象

同步方法的锁对象是什么呢?

在普通同步方法的锁对象是this

如果是静态同步方法,锁对象是<类名.class>,也就是字节码文件

### Lock

随但我们可以理解同步代码块和同步方法的锁对象问题,但是我们并没有直接看到代码在哪里上锁,在哪里开锁.(反正JDK有新的处理方式了),在JDK5以后,提供了一个新的锁对象Lock.

Lock实现提供比使用synchronized方法和语句可以获得更加广泛的锁定操作

Lock中提供了获取锁和释放锁的方法

void lock(); 获得锁

void unlock(); 释放锁

Lock是接口不能直接实例化,这里采用它的实现类ReentrantLock来实例化

ReentrantLock的构造方法

ReentrantLock(); 创建一个ReentrantLock的实例对象.

**示例:**

private ReentrantLock lock = new ReentrantLock();

@Override

public void run() {

while (true) {

//synchronized (obj) {

try {

lock.lock();

if (ticket <= 0) {

break;

} else {

Thread.sleep(10);

ticket--;

System.out.println(Thread.currentThread().getName() + "正在卖票,还剩" + ticket + "张票");

}

} catch (InterruptedException e) {

e.printStackTrace();

} finally {

lock.unlock();

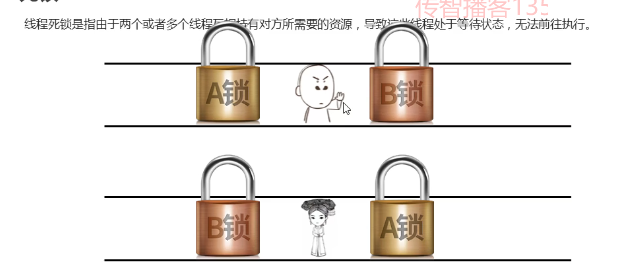
}

//}

}

### 死锁

线程死锁是指由于两个或者多个线程互相持有对方所欲要的资源,导致线程处于互相等待的状态,无法继续执行



Object objA = new Object();

Object objB = new Object();

new Thread(()->{

while (true){

synchronized (objA){

synchronized (objB){

System.out.println("尔康来了!");

}

}

}

}).start();

new Thread(()->{

while (true){

synchronized (objB){

synchronized (objA){

System.out.println("紫薇来了!");

}

}

}

}).start();

## 生产者和消费者模式

### 生产者和消费者模式-概述

等待唤醒机制的核心目的就是控制线程执行的情况(执行流程)

在这个案例中,控制的流程就是,有汉堡,允许消费者执行;没有汉堡,允许生产者执行

### 生产者和消费者模式-代码实现

为了体现生产和消费过程中的等待和唤醒,Java中提供了相对应的方法,这几个方法在Object类中:

void wait() 导致当前线程等待,直到另一个线程调用该对象的notify()或者notifyAll()方法;

void notify() 唤醒正在等待对象监视器的单个线程;

void notifyAll() 唤醒正在等待对象监视器的所有线程;

**代码实现:**

**public class Foodie extends Thread {**

**@Override**

**public void run() {**

**while (true) {**

**synchronized (Desk.lock) {**

**if (Desk.count == 0) {**

**break;**

**} else {**

**if (Desk.flag) {**

**System.out.println("吃货在吃汉堡");**

**Desk.flag = false;**

**Desk.lock.notifyAll();**

**Desk.count--;**

**} else {**

**try {**

**Desk.lock.wait();**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**}**

**}**

**}**

**}**

**}**

**}**

**套路:**

**1.while(true)死循环**

**2.synchronized锁,锁对象唯一**

**3.if判断 共享数据是否结束**

**true: 结束就跳出**

**false: 没结束就执行需求对应的代码逻辑**

**在完成前两步套路之后,主要在第三步判断中写清楚代码逻辑**

**1.注意锁唯一;**

**2.注意调用等待以及唤醒使用锁对象调用;**

**3.正确理解需求,不要摆错等待以及唤醒的时机**

**4...**

### 生产者和消费者模式-代码改写

Desk类改写成一个Javabean类

## **阻塞**队列

### **阻塞**队列-基本使用

**阻塞队列继承结构**

Iterable→Collection→Queue→BlockingQueue→

ArrayBlockingQueue\LinkedBlockingQueue

前四个接口,后面两个实现类

BlockingQueue接口的核心方法

Put(anObject) : 将参数放入队列,如果放不进去就阻塞(等着)

<参数是...>

take() : 取出第一个数据,娶不到就阻塞(等着)

常见BlockingQueue

ArrayBlockingQueue<E> : 底层是数组,有界;

**示例: 构造的参数表示阻塞队列的容量**

ArrayBlockingQueue<String> sbq = new ArrayBlockingQueue<>(2);

LinkedBlockingQueue<E> : 底层是链表,无界,但不是真正的无界,最大为int的最大值;

### **阻塞**队列-实现等待唤醒机制

使用put和take方法,在put和take方法底层已经加了锁了,简化了代码书写