# 多线程高级

## 线程状态

**特别的:** 线程在运行的过程当中是有可能被抢夺执行权的,在被抢夺在执行权时,线程处于三种状态分别是阻塞,等待,计时等待.线程在运行过程中是和CPU产生关系,JVM是不定义运行状态的.

在JVM中线程的六种状态:

新建: (NEW) → 创建线程对象

就绪: (RUNNABLE) → start方法

阻塞: (BLOCKED) → 无法获得锁对象

等待: (WAITING) → wait方法

计时等待: (TIMED\_WAITING) → sleep方法

结束: (TERMINATED) → 全部代码运行完毕

## 线程池

### 基本原理

**是线程池创建了线程对象还是调用者人为创建线程对象再放入线程池中?**

在内存中创建一个存储线程的容器空间,我们将需要执行的任务交给线程池,线程池自动匹配线程执行,如果执行的时候没有就创建,有就不创建新的.

**1.**创建线程池,开始时空的;

**创建Executors中的静态方法**

**2.**有任务要执行,创建线程对象,任务执行完毕,线程对象归还给线程池;

**submit方法: 池子会自动帮我们创建对象,任务执行完毕,也会自动把线程对象归还给池子**

**3.**所有任务执行完毕,关闭线程池.

**shutdown方法**

### Executors默认线程池

**ExecutorService executorService = Executors.newCachedThreadPool();**

**1.**创建默认的线程池(Executors的静态方法,不是构造方法!)

static ExecutorService newCachedThreadPool​()

创建一个默认的线程池;

static ExecutorService newFixedThreadPool​(int nThreads)

创建一个指定最大容量的线程池;

注意点: Executors-------用于创建线程池对象

ExecutorService—用于控制线程池

**2.** ExecutorService控制线程池的方法

submit() 提交一个可运行的任务执行,并返回一个表示该任务的未来.方法的形参可以是Runnable的实现类对象(重写run方法),也可以是Callable接口的实现类对象(重写call方法),匿名内部类lambda

**executorService.submit(() -> {**

**System.out.println(Thread.currentThread().getName() + "执行了");**

**});**

shutdown() 关闭线程池

### Executors创建指定上限的线程池

**ExecutorService executorService = Executors.newFixedThreadPool(10);**

static ExecutorService newFixedThreadPool​(int nThreads) 创建一个指定最大容量的线程池;

构造传递的参数时最大容量,初始容量是0;

### ThreadPoolExecutor

Module java.base

软件包 java.util.concurrent

线程池类都是在这个体系下面的

创建自定义线程池及对象

public class ThreadPoolExecutor extends AbstractExecutorService{}

**1.构造方法**

ThreadPoolExecutor​(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory, RejectedExecutionHandler handler);

**七个参数**

**参数一 int corePoolSize**

核心线程数量,除非线程池关闭,不然不会消失(合同工).>=0

**参数二 int maximumPoolSize**

最大线程数量,线程池线程的最大容量,>=0,且要>=核心线程数量

**参数三 long keepAliveTime**

空闲线程最大存活时间,>=0.空闲线程(临时工),最大数量(总-核心).

**参数四 TimeUnit unit**

空闲线程存活时间的单位,时?分?秒? 用TimeUnit类中的常量来表示

**参数五 BlockingQueue<Runnable> workQueue**

任务队列(阻塞队列),允许多少任务等待,任务排队的地方(让任务在队列中等着,等有线程空闲了,线程池再从这个队列中获取任务并执行),不能为null

**参数六 ThreadFactory threadFactory**

创建线程工厂,线程的出处,表示怎么创建线程对象.不能为null, Executors线程池类中的defaultThreadFactory()方法(是一个静态方法)来创建新线程的默认线程工厂.

**参数七 RejectedExecutionHandler handler**

任务的拒绝策略,不能为null. 在ThreadPoolExecutor类中的内部类默认处理方案.

**示例:**

ThreadPoolExecutor pool= New ThreadPoolExecutor(2, 5, 30,

TimeUnit.SECONDS, new ArrayBlockingQueue<>(10),

Executors.defaultThreadFactory(), new ThreadPoolExecutor.AbortPolicy());

**2.控制自定义线程池的方法**

pool.submit(()->{

System.out.println(Thread.currentThread().getName() + "执行了");

});

pool.shutdown();

控制方法是一样的,主要是构造方法的参数,是重点.

### 参数详解

**参数五:** 任务队列(阻塞队列),允许多少任务等待,任务排队的地方(让任务在队列中等着,等有线程空闲了,线程池再从这个队列中获取任务并执行),不能为null

**参数六:** 创建线程工厂,线程的出处,表示怎么创建线程对象.不能为null, Executors线程池类中的defaultThreadFactory()方法(是一个静态方法)来创建新线程的默认线程工厂.

**参数七:** 任务的拒绝策略,不能为null. 在ThreadPoolExecutor类中的内部类默认处理方案.

1.什么时候拒绝任务:当提交的任务>线程池最大线程容量+阻塞队列容量

2.如何拒绝: <四种处理方案>

<1> ThreadPoolExecutor.AbortPolicy

丢弃任务并抛出RejectedExecutionExecution异常,是默认处理方案

<2> ThreadPoolExecutor.DiscardPolicy

丢弃任务,但是不抛出异常,这是不推荐的做法

<3> ThreadPoolExecutor. DiscardOldestPolicy

抛弃队列中等待最久的任务,然后把当前的任务假如队列中

<4> ThreadPoolExecutor.CallerRunsPolicy

调用任务的run()方法绕过线程池直接执行.

### 非默认任务拒绝策略

就是进行代码演示

<2> ThreadPoolExecutor.DiscardPolicy

直接抛弃多余任务,且不会抛出异常

<3> ThreadPoolExecutor. DiscardOldestPolicy

假如多了五个任务,且阻塞队列只能等两个,那么流后两个,丢前三个

<4> ThreadPoolExecutor.CallerRunsPolicy

线程池和阻塞队列放满了,就会调用run方法将任务将给其他线程(就如main线程)

## volatile

### 问题

在多条线程共享数据时,当一条线程修改了共享数据后,另外其他线程没有第一时间获取到最新数据,导致其他线程还在之前数据的基础上运行的问题.

堆内存是唯一,栈内存是线程独有,每一条线程都有自己独有的线程栈.共享数据存放在堆内存,每条线程在抢夺到执行权后,都会到堆内存中复制一份共享数据放入自己的线程栈,名字叫做共享数据副本,而线程执行的时候都是从自己的共享数据副本中得到数据进行操作的.这样做的目的是提高了获取共享数据的速度.而在线程操作了自己当前的共享数据副本后,还会把自己的副本数据返回到堆内存中(说白了就是修改了数据源,但是其他线程不知道,还以自己的副本数据运行),只是把修改的传回堆内存,但是不会从堆中获取最新的共享数据.

**问题小结:**

1.堆内存是唯一的,每一条线程都有自己的线程栈;

2.每一条线程在使用对立面的变量时,都会先拷贝一份变量副本;

3.在县城中,每一次使用的时变量副本中的数据.

4.如果A线程修改了堆中共享变量的值,那么其他线程不一定能及时使用最新的值

### 解决

**volatile关键字 :** 他能强制线程每次使用数据时(无论自己的还是堆里的),都会区查看共享数据的最新值

**用法 :** 就加在定义共享数据的变量前面.

### synchronized解决

**同步代码块执行步骤:**

1.线程获得锁;

2.清空变量副本;

3.拷贝共享变量的最新值到变量副本中;

4.执行代码;

5.将修改后变量副本中的值赋值给共享数据;

6.释放锁;

其实解决的方式就是创建锁对象,再用synchronized同步代码块包裹线程执行流程就可以了,注意锁对象唯一.

## 原子性

原子性是指在一次操作或者多次操作中,要么所有的操作全部都得到了执行并且不受到任何因素的干扰而中断,要么所有的操作都不执行,多个操作是一个不可分割整体.

回归到线程上的意思是,每条线程操作共享数据的时候,需要符合原子性.因为线程执行的速度时很快的,在代码执行的过程中,每一行代码的执行都有可能被抢夺执行权,这和之前的卖票类似,(所以同步代码块可以解决这个问题),对共享数据进行重复或者遗漏操作,这是不正确的.

所以需要保证线程代码块的原子性.

### volatile关键字不能保证原子性

它只能保证获取到最新的共享数据,但是线程操作完数据可能还没来得及放回堆,就被其他线程提走了共享数据,导致了重复操作,所以volatile不能保证原子性.

### 原子性\_AtomicInteger

**原子包:**

util包→concurrent包→atomic包下的class AtomicInteger

**构造方法:**

public AtomicInteger() 初始化一个默认值为0的原子型Integer;

public AtomicInteger(int initialValue) 初始化一个指定值的原子型Integer

**成员方法:**

int get() 获取值;

int getAndlncrement() 以原子方式将当前值+1,返回自增**前**的值;

int incrementAndGet() 以原子方式将当前值+1,返回自增**后**的值;

int addAndGet(int data) 以原子方式将输入的数值与实例中的值(AtomicInteger里的value)相加,并返回相加结果;

int getAndSet(int value) 以原子方式设置为newValue的值,并返回旧值;

### AtomicInteger-内存解析

AtomicInteger原理 : 自旋锁 + CAS算法

CAS算法:

有三个操作数(内存值V,旧的预期值A,要修改的值B);

当旧的预期值A == 内存值 此时修改成功,将V改B;

当旧的预期值A!=内存值,此时修改失败,不做任何操作;

CAS算法小结:

1. 在修改共享数据的时候,把原来的旧值记录下来了.

2. 如果现在内存中的值跟原来的旧值一样,证明没有其他线程操作过内存值,则修改成功;

3. 如果现在内存中的值跟原来的旧值不一样了,证明已经有其他线程操作过内存值了,则修改失败,需要获取现在的最新值,再次进行操作,这个更新获取的过程就称**自旋.**

### AtomicInteger-源码解析

## 悲观锁和乐观锁

synchronized和CAS算法的区别:

相同点:在多线程情况下,都可以保证共享数据的安全性;

不同点:

synchronized(悲观锁)

synchronized总是从最坏的角度出发,认为每次获取数据的时候,其他线程都有可能修改.所以在每次操作共享数据之前,都会上锁;

CAS(乐观锁)

CAS是从乐观的角度出发,假设每次获取数据别人都不会修改,所以不糊上锁,只不过在修改共享数据的时候,会检查一下旧值是否等于新值:

如果其他线程修改了,那么就自旋获取最新值,

如果其他线程没有修改,那么就直接修改共享数据的值.

## 并发工具类

首先,集合有分线程安全和不安全(多线程环境下可能存在问题),为了保证数据的安全性,可以使用Hashtable,但是Hashtable的效率低下.

### Hashtable

他的底层和HashMap一样是哈希表 + 链表结构.

Hashtable源码底层采取悲观锁synchronized的形式保证数据的安全性.只要有线程访问,就会将整张表数据全部锁起来,所以Hashtable效率低下

### ConcurrentHashMap基本使用

既可以保证多线程环境下的数据安全,同时可以兼顾效率

public class ConcurrentHashMap extends Map 拥有Map的成员方法

总结:

1.HashMap是线程不安全的,多线程环境下会出现数据安全问题;

2.Hashtable是线程安全的,但是会将整张表锁起来,效率低下;

3. ConcurrentHashMap也是线程安全的,且效率较高.

在JDK1.7和JDK1.8中,底层原理是不一样的.

### ConcurrentHashMap1.7原理

### ConcurrentHashMap1.8原理

### CountDownLatch

**使用场景 :**

让某一条线程等待其他线程执行完毕之后再执行.

**构造方法 :**

CountDownLatch(int count) 参数写的是等待线程的数量,定义一个计数器;

**成员方法 :**

await() 让线程等待,当计数器为0时,会唤醒等待线程;

countDown() 线程执行完毕时调用,会将计数器-1;

### Semphore

**使用场景 :**

可以控制访问特定资源的线程数量(就是,允许同时访问共享资源的最大线程数量)

**步骤:**

1.创建Semphore对象

在Thread的子类或者Runnable\Callable的实现类里面成员位置创建,传递的参数表示允许的最带线程数量

2.对象.acquire()方法发放通行证

run()方法中,线程逻辑代码前

3.对象.release()方法回收通行证

run()方法中,线程逻辑代码后

没有通行证的线程,就等着~