# 1、线程的创建方式?

① 继承Thread类，作为线程对象存在（继承Thread对象）

public class MyThread extends Thread {    @Override public void run() { System.out.println("MyThread...run..."); }    public static void main(String[] args) {        *// 创建MyThread对象* MyThread t1 = new MyThread() ; MyThread t2 = new MyThread() ;        *// 调用start方法启动线程* t1.start(); t2.start();    }}

② 实现runnable接口，作为线程任务存在

public class MyRunnable implements Runnable{    @Override public void run() { System.out.println("MyRunnable...run..."); }    public static void main(String[] args) {        *// 创建MyRunnable对象* MyRunnable mr = new MyRunnable() ;        *// 创建Thread对象* Thread t1 = new Thread(mr) ; Thread t2 = new Thread(mr) ;        *// 调用start方法启动线程* t1.start(); t2.start();    }}

③ 创建带返回值的线程

public class MyCallable implements Callable<String> {    @Override public String call() throws Exception { System.out.println("MyCallable...call..."); return "OK"; }    public static void main(String[] args) throws ExecutionException, InterruptedException {        *// 创建MyCallable对象* MyCallable mc = new MyCallable() ;        *// 创建FutureTask对象* FutureTask<String> ft = new FutureTask<String>(mc) ;        *// 创建Thread对象* Thread t1 = new Thread(ft) ; Thread t2 = new Thread(ft) ;        *// 调用start方法启动线程* t1.start();        *// 调用ft的get方法获取执行结果* String result = ft.get();        *// 输出* System.out.println(result); }}

④ 线程池创建线程

public class MyRunnable implements Runnable{    @Override public void run() { System.out.println("MyRunnable...run..."); }    public static void main(String[] args) {        *// 创建线程池对象* ExecutorService threadPool = Executors.newFixedThreadPool(3); threadPool.submit(new MyRunnable()) ;        *// 关闭线程池* threadPool.shutdown();    }}

# 2、线程有哪些基本状态?

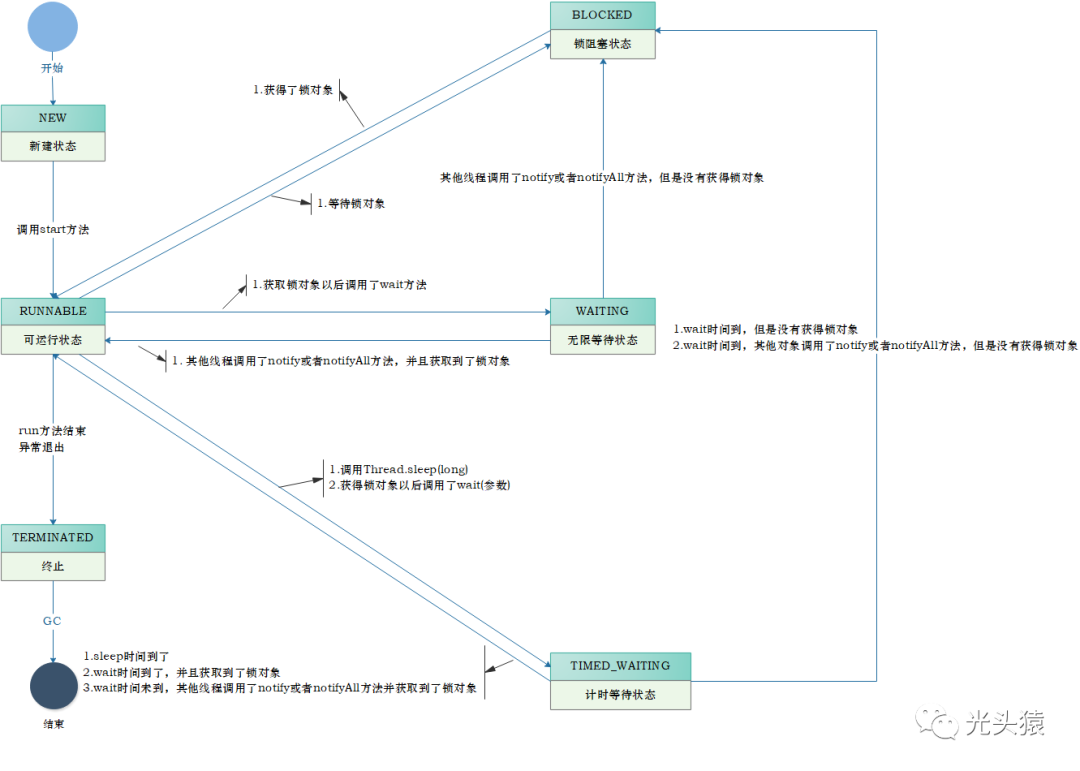
Java中的线程状态被定义在了java.lang.Thread.State枚举类中，State枚举类的源码如下：

public class Thread {   public enum State {        */\* 新建 \*/* NEW ,         */\* 可运行状态 \*/*        RUNNABLE , */\* 阻塞状态 \*/*        BLOCKED , */\* 无限等待状态 \*/*        WAITING ,  */\* 计时等待 \*/*        TIMED\_WAITING , */\* 终止 \*/*        TERMINATED;  } *// 获取当前线程的状态* public State getState() { return jdk.internal.misc.VM.toThreadState(threadStatus);    }}

通过源码我们可以看到Java中的线程存在6种状态，每种线程状态的含义如下：

|  |  |
| --- | --- |
| **线程状态** | **具体含义** |
| NEW | 一个尚未启动的线程的状态。也称之为初始状态、开始状态。线程刚被创建，但是并未启动。还没调用start方法。MyThread t = new MyThread()只有线程象，没有线程特征。 |
| RUNNABLE | 当我们调用线程对象的start方法，那么此时线程对象进入了RUNNABLE状态。那么此时才是真正的在JVM进程中创建了一个线程，线程一经启动并不是立即得到执行，线程的运行与否要听令与CPU的调度，那么我们把这个中间状态称之为可执行状态(RUNNABLE)也就是说它具备执行的资格，但是并没有真正的执行起来而是在等待CPU的度。 |
| BLOCKED | 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 | 一个正在等待的线程的状态。也称之为等待状态。造成线程等待的原因有两种，分别是调用Object.wait()、join()方法。处于等待状态的线程，正在等待其他线程去执行一个特定的操作。例如：因为wait()而等待的线程正在等待另一个线程去调用notify()或notifyAll()；一个因为join()而等待的线程正在等待另一个线程结束。 |
| TIMED\_WAITING | 一个在限定时间内等待的线程的状态。也称之为限时等待状态。造成线程限时等待状态的原因有三种，分别是：Thread.sleep(long)，Object.wait(long)、join(long)。 |
| TERMINATED | 一个完全运行完成的线程的状态。也称之为终止状态、结束状态 |

各个状态的转换，如下图所示：



# 3、如何停止一个正在运行的线程？

① 使用退出标志，使线程正常退出。

public class MyThread extends Thread {    volatile boolean flag = false ;     *// 线程执行的退出标记*    @Override public void run() { while(!flag) { System.out.println("MyThread...run..."); try { Thread.sleep(3000); } catch (InterruptedException e) { e.printStackTrace(); } } }    public static void main(String[] args) throws InterruptedException {        *// 创建MyThread对象* MyThread t1 = new MyThread() ; t1.start();        *// 主线程休眠2秒* Thread.sleep(6000);        *// 更改标记为true* t1.flag = true ;    }}

② 使用stop方法强行终止

public class MyThread extends Thread {    volatile boolean flag = false ;     *// 线程执行的退出标记*    @Override public void run() { while(!flag) { System.out.println("MyThread...run..."); try { Thread.sleep(3000); } catch (InterruptedException e) { e.printStackTrace(); } } }    public static void main(String[] args) throws InterruptedException { *// 创建MyThread对象* MyThread t1 = new MyThread() ;        t1.start();        *// 主线程休眠2秒* Thread.sleep(6000);        *// 调用stop方法* t1.stop();    }}

③ 使用interrupt方法中断线程。

public class MyThread extends Thread {    volatile boolean flag = false ;     *// 线程执行的退出标记*    @Override public void run() { while(!flag) { System.out.println("MyThread...run..."); try { Thread.sleep(3000); } catch (InterruptedException e) { e.printStackTrace(); } } }    public static void main(String[] args) throws InterruptedException {        *// 创建MyThread对象* MyThread t1 = new MyThread() ; t1.start();        *// 主线程休眠2秒* Thread.sleep(6000);        *// 调用interrupt方法* t1.interrupt();    }}

# 4、有三个线程T1,T2,T3,如何保证顺序执行？

在多线程中有多种方法让线程按特定顺序执行，你可以用线程类的**join**()方法在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。

public class JoinTest {    public static void main(String[] args) {        *// 创建线程对象* Thread t1 = new Thread(() -> { System.out.println("t1"); }) ;        Thread t2 = new Thread(() -> { try { t1.join(); *// 加入线程t1,只有t1线程执行完毕以后，再次执行该线程* } catch (InterruptedException e) { e.printStackTrace(); } System.out.println("t2"); }) ;        Thread t3 = new Thread(() -> { try { t2.join(); *// 加入线程t2,只有t2线程执行完毕以后，再次执行该线程* } catch (InterruptedException e) { e.printStackTrace(); } System.out.println("t3"); }) ;        *// 启动线程* t1.start(); t2.start(); t3.start();    }}

# 5、在线程中你怎么处理不可控制异常？

在Java中有两种异常：

1、非运行时异常（Checked Exception）：这种异常必须在方法声明的throws语句指定，或者在方法体内捕获。例如：IOException和ClassNotFoundException。

2、运行时异常（Unchecked Exception）：这种异常不必在方法声明中指定，也不需要在方法体中捕获。例如，NumberFormatException。

因为run()方法不支持throws语句，所以当线程对象的run()方法抛出非运行异常时，我们必须捕获并且处理它们。当运行时异常从run()方法中抛出时，默认行为是在控制台输出堆栈记录并且退出程序。好在，java提供给我们一种在线程对象里捕获和处理运行时异常的一种机制。实现用来处理运行时异常的类，这个类实现UncaughtExceptionHandler接口并且实现这个接口的uncaughtException()方法。

代码如下所示：

public class MyRunnable implements Runnable{    @Override public void run() { int a = Integer.parseInt("itheima"); System.out.println(a); }    public static void main(String[] args) {        *// 创建任务类对象* MyRunnable mr = new MyRunnable() ;        *// 创建线程对象* Thread t1 = new Thread(mr) ; t1.setUncaughtExceptionHandler(new ThreadExceptionHandler()); *// 设置运行期异常的处理器类* t1.start();    }}  
class ThreadExceptionHandler implements Thread.UncaughtExceptionHandler {    @Override public void uncaughtException(Thread t, Throwable e) { System.out.println(t.getId() + "---" + t.getState()); e.printStackTrace(); }}

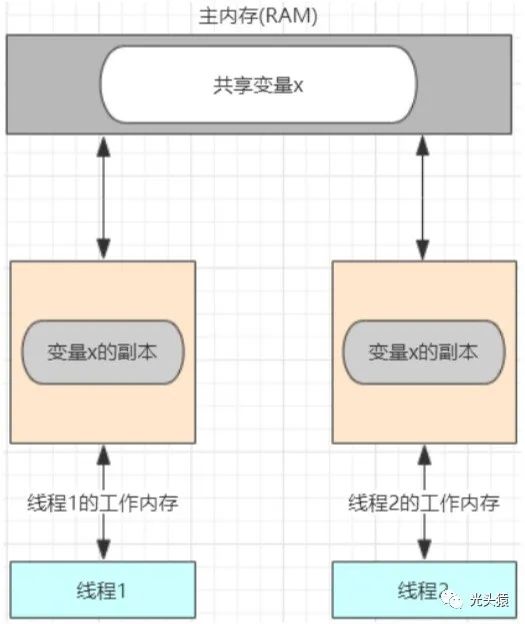
# 6、什么是Java内存模型？

JMM(Java Memory Model)Java内存模型,是java虚拟机规范中所定义的一种内存模型。

Java内存模型(Java Memory Model)描述了Java程序中各种变量(线程共享变量)的访问规则，以及在JVM中将变量存储到内存和从内存中读取变量这样的底层细节。

特点：

1. 所有的共享变量都存储于主内存(计算机的RAM)这里所说的变量指的是实例变量和类变量。不包含局部变量，因为局部变量是线程私有的，因此不存在竞争问题。
2. 每一个线程还存在自己的工作内存，线程的工作内存，保留了被线程使用的变量的工作副本。
3. 线程对变量的所有的操作(读，写)都必须在工作内存中完成，而不能直接读写主内存中的变量，不同线程之间也不能直接访问对方工作内存中的变量，线程间变量的值的传递需要通过主内存完成。



# 7、volatile 是什么？可以保证有序性吗？

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

① 保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的,volatile关键字会强制将修改的值立即写入主存。

② 禁止进行指令重排序，可以保证有序性。

指令重排：计算机在执行程序时，为了提高性能，编译器和处理器常常会对指令重排。处理器在进行重排序时，必须要考虑指令之间的数据依赖性。如下代码：

public void mySort() { int x = 11; int y = 12; x = x + 5; y = x \* x;}

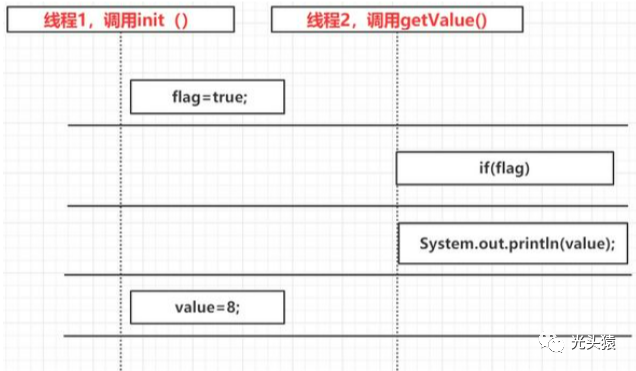
按照正常的顺序进行执行，那么执行顺序应该是：1 2 3 4 。但是如果发生了指令重排，那么此时的执行顺序可能是：① 1 3 2 4  ② 2 1 3 4 但是肯定不会出现：4 3 2 1这种顺序，因为处理器在进行重排时候，必须考虑到指令之间的数据依赖性。多线程环境里编译器和CPU指令优化根本无法识别多个线程之间存在的数据依赖性，比如说下面的程序代码如果两个方法在两个不同的线程里面调用就可能出现问题。

private static int value;private static boolean flag;  
public static void init(){ value=8; *//语句1* flag=true; *//语句2*}  
public static void getValue(){ if(flag){ System.out.println(value); }}

根据上面代码，如果程序代码运行都是按顺序的，那么getValue() 中打印的value值必定是等于8的，不过如果init()方法经过了指令重排序，那么结果就不一定了。进行重排序后代码执行顺序可能如下。

flag=true; //语句2 value=8; //语句1

如果init()方法经过了指令重排序后，这个时候两个线程分别调用 init()和getValue()方法，那么就有可能出现下图的情况，导致最终打印出来的value数据等于0。



解决方案：使用volatile修饰flag，禁止指令重排。

原理说明：添加了一个内存屏障，通过插入内存屏障禁止在内存屏障前后的指令执行重排序优

# 8、说一说自己对于synchronized关键字的了解？

说明：

① synchronized关键字解决的是多个线程之间访问资源的同步性，synchronized关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。

② 在Java早期版本中，synchronized属于重量级锁，效率低下

* 因为监视器锁（monitor）是依赖于底层的操作系统的 Mutex Lock 来实现的，Java 的线程是映射到操作系统的原生线程之上的。
* 如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态转换到内核态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高。

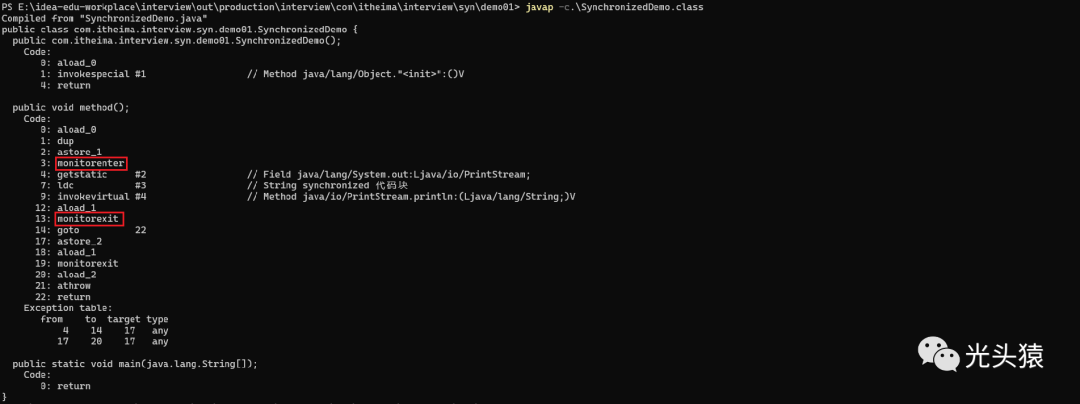
③ 在Java6之后Java 官方对从JVM层面对synchronized 较大优化，所以现在的 synchronized 锁效率也优化得很不错了。

# 9、讲一下synchronized关键字的底层原理？

① synchronized同步代码块的情况

public class SynchronizedDemo {    public void method() {        synchronized (this) { System.out.println("synchronized 代码块"); }    } }

通过javap查看字节码文件信息，如下所示：

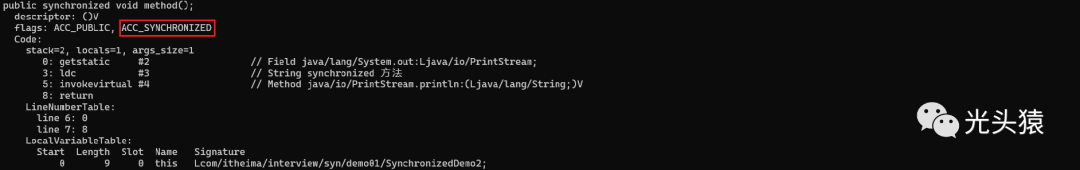


从上面我们可以看出：synchronized 同步语句块的实现使用的是 monitorenter 和 monitorexit 指令，其中monitorenter 指令指向同步代码块的开始位置，monitorexit 指令则指明同步代码块的结束位置。当执行 monitorenter 指令时，线程试图获取锁也就是获取 monitor(monitor对象存在于每个Java对象的对象头中，synchronized 锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因) 的持有权。当计数器为0则可以成功获取，获取后将锁计数器设为1也就是加1。相应的在执行monitorexit 指令后，将锁计数器设为0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止。

② synchronized修饰方法的的情况

public class SynchronizedDemo2 { public synchronized void method() { System.out.println("synchronized 方法"); }}

通过javap(javap -v xxx.class)查看字节码文件信息，如下所示：

synchronized 修饰的方法并没有 monitorenter 指令和 monitorexit 指令，取得代之的确实是ACC\_SYNCHRONIZED 标识，该标识指明了该方法是一个同步方法，JVM 通过该ACC\_SYNCHRONIZED访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。

# 10、Java中synchronized 和 ReentrantLock 有什么不同？

相似点：

这两种同步方式有很多相似之处，它们都是加锁方式同步，而且都是阻塞式的同步，也就是说当如果一个线程获得了对象锁，进入了同步块，其他访问该同步块的线程都必须阻塞在同步块外面等待，而进行线程阻塞和唤醒的代价是比较高的.

区别：

这两种方式最大区别就是对于Synchronized来说，它是java语言的关键字，是原生语法层面的互斥，需要jvm实现。而ReentrantLock它是JDK 1.5之后提供的API层面的互斥锁，需要lock()和unlock()方法配合try/finally语句块来完成。

相比Synchronized，ReentrantLock类提供了一些高级功能，主要有以下2项：

① 等待可中断，持有锁的线程长期不释放的时候，正在等待的线程可以选择放弃等待，这相当于Synchronized来说可以避免出现死锁的情况。

② Synchronized锁非公平锁，ReentrantLock默认的构造函数是创建的非公平锁，可以通过参数true设为公平锁，但公平锁表现的性能不是很好。(公平锁，多个线程等待同一个锁时，必须按照申请锁的时间顺序获得锁。)

# 11、什么是可重入锁（ReentrantLock）？

Java.util.concurrent.lock 中的 Lock 框架是锁定的一个抽象，它允许把锁定的实现作为Java 类，而不是作为语言的特性来实现。这就为Lock 的多种实现留下了空间，各种实现可能有不同的调度算法、性能特性或者锁定语义。ReentrantLock 类实现了Lock ，它拥有与synchronized 相同的并发性和内存语义，但是添加了类似锁投票、定时锁等候和可中断锁等候的一些特性。此外，它还提供了在激烈争用情况下更佳的性能。（换句话说，当许多线程都想访问共享资源时，JVM可以花更少的时候来调度线程，把更多时间用在执行线程上。）

锁的可重入性：获取锁的线程在执行加锁的方法的时候不需要再次获取锁对象

# 12、什么是CAS？

CAS的全称是：Compare And Swap(比较再交换); 是现代CPU广泛支持的一种对内存中的共享数据进行操作的一种特殊指令。CAS可以将read-modify-write转换为原子操作，这个原子操作直接由CPU保证。CAS有3个操作数：内存值V，旧的预期值A，要修改的新值B。当且仅当旧预期值A和内存值V相同时，将内存值V修改为B并返回true，否则什么都不做，并返回false。

# 13、CAS 有什么缺陷，如何解决？

① ABA问题

并发环境下，假设初始条件是A，去修改数据时，发现是A就会执行修改。但是看到的虽然是A，中间可能发生了A变B，B又变回A的情况。此时A已经非彼A，数据即使成功修改,也可能有问题。可以通过AtomicStampedReference「解决ABA问题」，它是一个带有标记(数据版本号)的原子引用类，通过控制变量值的版本来保证CAS的正确性。

② 循环时间长开销

自旋CAS，如果一直循环执行，一直不成功，会给CPU带来非常大的执行开销。很多时候，CAS思想体现，是有个自旋次数的，就是为了避开这个耗时问题~

③ 只能保证一个变量的原子操作

CAS 保证的是对一个变量执行操作的原子性，如果对多个变量操作时，CAS 目前无法直接保证操作的原子性的。

    可以通过这两个方式解决这个问题：

* + 使用互斥锁来保证原子性
  + 将多个变量封装成对象，通过AtomicReference来保证原子性

# 14、什么是AQS？

AQS全称为：AbstractQueuedSynchronizer，翻译过来应该是抽象队列同步器。如果说java.util.concurrent的基础是CAS的话，那么AQS就是整个Java并发包的核心了，ReentrantLock、CountDownLatch、Semaphore等等都用到了它。AQS实际上以双向队列的形式连接所有的Entry，比方说ReentrantLock，所有等待的线程都被放在一个Entry中并连成双向队列，前面一个线程使用ReentrantLock好了，则双向队列实际上的第一个Entry开始运行。AQS定义了对双向队列所有的操作，而只开放了tryLock和tryRelease方法给开发者使用，开发者可以根据自己的实现重写tryLock和tryRelease方法，以实现自己的并发功能。

# 15、简述一下你对线程池的理解？

合理利用线程池能够带来三个好处：

第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

第二：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。

第三：提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

# 16、如何创建线程池？

① 通过Executor框架的工具类Executors来实现我们可以创建三种类型的：

* FixedThreadPool: 该方法返回一个固定线程数量的线程池。该线程池中的线程数量始终不变。当有一个新的任务提交时，线程池中若有空闲线程，则立即执行。若没有，则新的任务会被暂存在

一个任务队列中，待有线程空闲时，便处理在任务队列中的任务。

* SingleThreadExecutor: 方法返回一个只有一个线程的线程池。若多余一个任务被提交到该线程池，任务会被保存在一个任务队列中，待线程空闲，按先入先出的顺序执行队列中的任务。
* CachedThreadPool：该方法返回一个可根据实际情况调整线程数量的线程池。线程池的线程数量不确定，但若有空闲线程可以复用，则会优先使用可复用的线程。若所有线程均在工作，又有新

的任务提交，则会创建新的线程处理任务。所有线程在当前任务执行完毕后，将返回线程池进行复用。

② 《阿里巴巴Java开发手册》中强制线程池不允许使用 Executors 去创建，而是通过ThreadPoolExecutor 的方式：

ThreadPoolExecutor最完整的构造方法：

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

参数说明：

corePoolSize： 核心线程的最大值，不能小于0maximumPoolSize：最大线程数，不能小于等于0，maximumPoolSize >= corePoolSizekeepAliveTime： 空闲线程最大存活时间,不能小于0unit： 时间单位workQueue： 任务队列，不能为nullthreadFactory： 创建线程工厂,不能为null handler： 任务的拒绝策略,不能为null

# 17、高并发、任务执行时间短的业务怎样使用线程池？并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？

① 高并发、任务执行时间短的业务，线程池程数可以设置为CPU核数+1，减少线程上下文的切换

② 并发不高、任务执行时间长的业务要区分开看

* 假如是业务时间长集中在IO操作上，也就是IO密集型的任务，因为IO操作并不占用CPU，所以不要让所有的CPU闲下来，可以加大线程池中的线程数目，让CPU处理更多的业务
* 假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（1）一样吧，线程池中的线程数设置得少一些，减少线程上下文的切换

③ 并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考

（2）。最后，业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。

# 18、如果你提交任务时，线程池队列已满，这时会发生什么？

① 无界队列

如果使用的是无界队列LinkedBlockingQueue，也就是无界队列的话，没关系，继续添加任务到阻塞队列中等待执行，因为LinkedBlockingQueue可以近乎认为是一个无穷大的队列，可以无限存放任务

② 有界队列

如果使用的是有界队列比如ArrayBlockingQueue，任务首先会被添加到ArrayBlockingQueue中，ArrayBlockingQueue满了，会根据maximumPoolSize的值增加线程数量，如果增加了线程数量还是处理不过来，ArrayBlockingQueue继续满，那么则会使用拒绝策略RejectedExecutionHandler处理满了的任务，默认是AbortPolicy

# 19、如何控制某个方法允许并发访问线程的大小？

Semaphore两个重要的方法就是semaphore.acquire() 请求一个信号量，这时候的信号量个数-1（一旦没有可使用的信号量，也即信号量个数变为负数时，再次请求的时候就会阻塞，直到其他线程释放了信号量）semaphore.release()释放一个信号量，此时信号量个数+1

线程任务类：

public class CarThreadRunnable implements Runnable {    *// 创建一个Semaphore对象,限制只允许2个线程获取到许可证* private Semaphore semaphore = new Semaphore(2) ;    @Override public void run() { *// 这个run只允许2个线程同时执行*        try {            *// 获取许可证* semaphore.acquire(); System.out.println(Thread.currentThread().getName() + "----->>正在经过十字路口");            *// 模拟车辆经过十字路口所需要的时间* Random random = new Random(); int nextInt = random.nextInt(7); TimeUnit.SECONDS.sleep(nextInt);            System.out.println(Thread.currentThread().getName() + "----->>驶出十字路口");            *// 释放许可证* semaphore.release();        } catch (InterruptedException e) { e.printStackTrace(); }    }}

测试类：

public class SemaphoreDemo01 {    public static void main(String[] args) {        *// 创建线程任务类对象* CarThreadRunnable carThreadRunnable = new CarThreadRunnable() ;        *// 创建5个线程对象，并启动。* for(int x = 0 ; x < 5 ; x++) { new Thread(carThreadRunnable).start(); }    }}

# 20、Java中活锁和死锁有什么区别？

活锁：一个线程通常会有会响应其他线程的活动，如果其他线程也会响应另一个线程的活动，那么就有可能发生活锁。同死锁一样，发生活锁的线程无法继续执行。然而线程并没有阻塞——他们在忙于响应对方无法恢复工作。这就相当于两个在走廊相遇的人：甲向他自己的左边靠想让乙过去，而乙向他的右边靠想让甲过去。可见他们阻塞了对方。甲向他的右边靠，而乙向他的左边靠，他们还是阻塞了对方。

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需要的资源，导致这些线程处于等待状态，无法前往执行。

死锁演示：

Thread的子类:

public class DeadThread extends Thread {    *// 定义成员变量，来切换线程去执行不同步代码块的执行* private boolean flag ; public DeadThread(boolean flag) { this.flag = flag ; }    @Override public void run() {        if(flag) {            synchronized (MyLock.R1) { System.out.println(Thread.currentThread().getName() + "---获取到了R1锁，申请R2锁...."); synchronized (MyLock.R2) { System.out.println(Thread.currentThread().getName() + "---获取到了R1锁，获取到了R2锁...."); }            }        }else {            synchronized (MyLock.R2) { System.out.println(Thread.currentThread().getName() + "---获取到了R2锁，申请R1锁...."); synchronized (MyLock.R1) { System.out.println(Thread.currentThread().getName() + "---获取到了R2锁，获取到了R1锁...."); }            }        }    }}

锁接口:

public interface MyLock {  
  
 *// 定义锁对象* public static final Object R1 = new Object() ; public static final Object R2 = new Object() ;  
  
}测试类:public class DeadThreadDemo1 {    public static void main(String[] args) {        *// 创建线程对象* DeadThread deadThread1 = new DeadThread(true) ; DeadThread deadThread2 = new DeadThread(false) ;        *// 启动两个线程* deadThread1.start(); deadThread2.start();    }}

控制台输出结果

Thread-0*---获取到了R1锁，申请R2锁....*Thread-1*---获取到了R2锁，申请R1锁....*

此时程序并没有结束，这种现象就是死锁现象...线程Thread-0持有R1的锁等待获取R2锁，线程Thread-1持有R2的锁等待获取R1的锁。

# 21、如何进行死锁诊断？

当程序出现了死锁现象，我们应该如何进行诊断呢？

使用jdk自带的工具：jstack

对上面的程序使用jstack进行死锁诊断

C:\Users\Administrator>jps74088144 DeadThreadDemo136209108 Launcher9180 JpsC:\Users\Administrator>jstack -l 81442019-10-19 14:52:01Full thread dump Java HotSpot(TM) 64-Bit Server VM (11+28 mixed mode):..."Thread-0": at com.itheima.javase.security.demo10.DeadThread.run(DeadThread.java:19) - waiting to lock <0x000000008c591218> (a java.lang.Object) // 等待锁0x000000008c591218 - locked <0x000000008c591208> (a java.lang.Object) // 已经拥有的锁0x000000008c591208"Thread-1": at com.itheima.javase.security.demo10.DeadThread.run(DeadThread.java:29) - waiting to lock <0x000000008c591208> (a java.lang.Object) // 等待锁0x000000008c591208        - locked <0x000000008c591218> (a java.lang.Object)        // 已经拥有的锁0x000000008c591218Found 1 deadlock.    // 发现了一个1个死锁C:\Users\Administrator>

# 22、Linux环境下如何查找哪个线程使用CPU最长？

步骤如下：

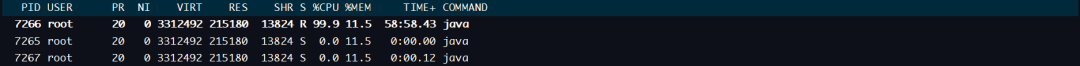
1、通过ps -ef | grep java查询指定的进程id

2、通过top -H -p pid查看指定进程中的线程信息

3、将线程的id的十进制数据转换成十六进制：printf "%x" tid

4、通过jstack -l pid查询进行中的线程nid(native thread id)

图片



图片

