链滴

JAVA 多线程

作者：XinyiZhang

原文链接：https://ld246.com/article/1601261770704

来源网站：链滴

线程基础概念

线程的状态及相互转换

1.初始(NEW) -- 新创建了一个线程对象，但还没有调用start()方法;

2.运行(RUNNABLE) -- 处于可运行状态的线程正在JVM中执行,但它可能正在等待来自操作系统的其资源;

```
//RUNNABLE
new Thread(() -> {
    try {
        System.in.read();
    } catch (IOException e) {
    e.printStackTrace();
    }
}).start();
```

3.阻塞(BLOCKED) -- 线程阻塞于synchronized锁,等待获取synchronized锁的状态;

```
//BLOCKED
Object object = new Object();
new Thread(() -> {
    synchronized (object){
    try {
        TimeUnit.MINUTES.sleep(5);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    }
```

```
}).start();
Thread.sleep(2000L);
new Thread(() -> {
    synchronized (object){
    }
}).start();
```

4.等待(WAITING) -- Object.wait()、join()、 LockSupport.park(),进入该状态的线程需要等待其他程做出一些特定动作(通知或中断);

```
//WAITING
Object object = new Object();
new Thread(() -> {
    synchronized (object){
    try {
        object.wait();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    }
}).start();
```

5.超时等待(TIME_WAITING) -- Object.wait(long)、Thread.join()、LockSupport.parkNanos()、LockSupport.parkUntil,该状态不同于WAITING,它可以在指定的时间内自行返回;

6.终止(TERMINATED) -- 表示该线程已经执行完毕;

JAVA多线程

创建多线程的方式

1. 继承Thread类,重写run方法;

2. 实现Runnable接口并实现run方法;(实际开发中选用,java只允许单继承 增加程序的健壮性,代码可共享,代码跟数据独立);

3. 匿名内部类

4. Lambda表达式

5. 线程池

线程挂起及唤醒

- suspend() -- 挂起,已废弃,因挂起时不会释放线程所占用的资源.如果使用该方法将某个线程挂起,则能会使其他等待资源的线程死锁;
- resume() -- 唤醒,已废弃,本身并无问题,但是不能独立于suspend()方法存在;
- wait() -- 暂停执行、放弃已经获得的锁、进入等待状态;
- notify() -- 随机唤醒一个在等待锁的线程;
- notifyAll() -- 唤醒所有在等待锁的线程,自行抢占cpu资源;

线程的中断

- stop() -- 废弃方法,开发中不要使用.因为一调用,线程就立刻停止,可能引发相应的线程安全性问题;
- interrupt() -- 中断方法;
- 自行定义一个标志,用来判断是否继续执行;

线程的优先级

- 线程的优先级设置可以为1-10的任一数值;
- Thread类中定义了三个线程优先级;
  - MIN_PRIORITY(1) -- 最小;
  - NORM_PRIORITY(5) -- 默认;
  - MAX_PRIORITY(10) -- 最大;
- 一般情况下推荐使用这几个常量,不要自行设置数值;

线程分类

- 线程分为用户线程、守护线程;
- 守护线程 -- 任何一个守护线程都是整个程序中所有用户线程的守护者,只要有活着的用户线程,守护程就活着.当JVM实例中最后一个非守护线程结束时,也随JVM一起退出;
- 守护线程的用处 -- jvm垃圾清理线程;
- 尽量少使用守护线程,因其不可控不要在守护线程里去进行读写操作、执行计算逻辑;
- 设置守护线程 -- setDaemon(true);

锁分类

内置锁

- 每个java对象都可以用做一个实现同步的锁,这些锁称为内置锁;
- 线程进入同步代码块或方法的时候会自动获得该锁,在退出同步代码块或方法时会释放该锁;
- 获得内置锁的唯一途径就是进入这个锁的保护的同步代码块或方法;
- 内置锁本身就是互斥锁;

互斥锁

- 最多只有一个线程能够获得该锁;
- 当线程A尝试去获得线程B持有的内置锁时,线程A必须等待或者阻塞,直到线程B释放这个锁,如果B线不释放这个锁,那么A线程将永远等待下去;

自旋锁

- 线程状态及上下文切换消耗系统资源;
- 当访问共享资源的时间短,频繁上下文切换不值得;

- jvm实现,使线程在没获得锁的时候,不被挂起,转而执行空循环,循环几次之后,如果还没能获得锁,则被起;

阻塞锁

- 阻塞锁改变了线程的运行状态,让线程进入阻塞状态进行等待;
- 当获得相应的信号(唤醒或者时间) 时,才可以进入线程的准备就绪状态,转为就绪状态的所有线程;
- 通过竞争,进入运行状态;

重入锁

- 支持线程再次进入的锁,就跟我们有房间钥匙,可以多次进入房间类似;

读写锁

- 两把锁,读锁跟写锁;
- 写写互斥、读写互斥、读读共享;

悲观锁

- 总是假设最坏的情况,每次去拿数据的时候都认为别人会修改;
- 每次在拿数据的时候都会上锁,这样别人想拿这个数据就会阻塞直到它拿到锁;

乐观锁

- 每次去拿数据的时候都认为别人不会修改,所以不会上锁;
- 但是在更新的时候会判断一下在此期间别人有没有去更新这个数据;
- 可以使用版本号等机制;

公平锁

- 大家都老老实实排队,对大家而言都很公平;

非公平锁

- 一部分人排着队,但是新来的可能插队;

偏向锁

- 偏向锁使用了一种等到竞争出现才释放锁的机制;
- 所以当其他线程尝试竞争偏向锁时,持有偏向锁的线程才会释放锁;

独占锁

- 独占锁模式下,每次只能有一个线程能持有锁;

共享锁

- 允许多个线程同时获取锁,并发访问共享资源;

关键字

synchronized

- 修饰普通方法 -- 锁住对象的实例;
- 修饰静态方法 -- 锁住整个类;
- 修饰代码块 -- 锁住一个对象 synchronized (lock) 即synchronized后面括号里的内容;

volatile

- 能且仅能修饰变量;
- 仅仅保证可见性,并不保证原子性;
- 禁止指令重排序;
- 使用场景
  - 作为线程开关;
  - 单例,修饰对象实例,禁止指令重排序;

类

AbstractQueuedSynchronizer

- 为实现依赖于先进先出(FIFO)等待队列的阻塞锁和相关同步器(信号量、事件等等)提供一个框架;
- 设计目标是成为依靠单个原子int值来表示状态的大多数同步器的一个有用基础.
- 子类必须定义更改此状态的受保护方法,并定义哪种状态对于此对象意味着被获取或被释放.假定这条件之后,此类中的其他方法就可以实现所有排队和阻塞机制;
- 子类可以维护其他状态字段,但只是为了获得同步而只追踪使用getState()、setState(int)和 compareAndSetState(int, int)方法来操作以原子方式更新的 int 值;
- 应该将子类定义为非公共内部帮助器类,可用它们来实现其封闭类的同步属性;
- 没有实现任何同步接口.而是定义了诸如acquireInterruptibly(int)之类的一些方法,在适当的时候可通过具体的锁和相关同步器来调用它们,以实现其公共方法;
- 支持默认的独占模式和共享模式之一,或者二者都支持;
  - 处于独占模式下时,其他线程试图获取该锁将无法取得成功;
  - 在共享模式下,多个线程获取某个锁可能(但不是一定)会获得成功;
- 此类并不"了解"这些不同,除了机械地意识到当在共享模式下成功获取某一锁时,下一个等待线程(如存在)也必须确定自己是否可以成功获取该锁. 处于不同模式下的等待线程可以共享相同的FIFO队列;

● 通常,实现子类只支持其中一种模式,但两种模式都可以在 (例如)ReadWriteLock 中发挥作用.只支持占模式或者只支持共享模式的子类不必定义支持未使用模式的方法;

● 此类通过支持独占模式的子类定义了一个嵌套的AbstractQueuedSynchronizer.ConditionObject类可以将这个类用作Condition实现;

- isHeldExclusively()方法将报告同步对于当前线程是否是独占的;
- 使用当前getState()值调用release(int)方法则可以完全释放此对象;
- 如果给定保存的状态值,那么acquire(int)方法可以将此对象最终恢复为它以前获取的状态;

● 没有别的AbstractQueuedSynchronizer方法创建这样的条件,因此,如果无法满足此约束,则不要使它;

● AbstractQueuedSynchronizer.ConditionObject的行为当然取决于其同步器实现的语义;

● 此类为内部队列提供了检查、检测和监视方法,还为condition对象提供了类似方法,可以根据需要使用于其同步 机制的AbstractQueuedSynchronizer将这些方法导出到类中;

● 此类的序列化只存储维护状态的基础原子整数,因此已序列化的对象拥有空的线程队列.需要可序列的典型子类将定义一个readObject方法,该方法在反序列化时将此对象恢复到某个已知初始状态;

锁

ReentrantLock(重入锁)

非公平锁(ReentrantLock(false))

```
public class ReentrantLockTest {

    public static void main(String[] args) throws InterruptedException {

        ReentrantLock lock = new ReentrantLock();

        for (int i = 1; i <= 3; i++) {
            lock.lock();
        }

        for(int i=1;i<=3;i++){
            try {

            } finally {
                lock.unlock();
            }
        }
    }
}
```

公平锁(ReentrantLock(true))

```
public class ReentrantLockTest {

    static Lock lock = new ReentrantLock(true);

    public static void main(String[] args) throws InterruptedException {
```

```
        for(int i=0;i<5;i++){
            new Thread(new ThreadDemo(i)).start();
        }

    }

    static class ThreadDemo implements Runnable {
        Integer id;

        public ThreadDemo(Integer id) {
            this.id = id;
        }

        @Override

      public void run() {
            try {
                TimeUnit.MILLISECONDS.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            for(int i=0;i<2;i++){
                lock.lock();
                System.out.println("获得锁的线程："+id);
                lock.unlock();
            }
        }
    }
}
```

响应中断(lockInterruptibly())

```
public class ReentrantLockTest {
    static Lock lock1 = new ReentrantLock();
    static Lock lock2 = new ReentrantLock();
    public static void main(String[] args) throws InterruptedException {

        Thread thread = new Thread(new ThreadDemo(lock1, lock2));//该线程先获取锁1,再获取锁2
        Thread thread1 = new Thread(new ThreadDemo(lock2, lock1));//该线程先获取锁2,再获取
1
        thread.start();
        thread1.start();
        thread.interrupt();//是第一个线程中断
    }

    static class ThreadDemo implements Runnable {
        Lock firstLock;
        Lock secondLock;
        public ThreadDemo(Lock firstLock, Lock secondLock) {
            this.firstLock = firstLock;
            this.secondLock = secondLock;
        }
        @Override
```

```
        public void run() {
            try {
                firstLock.lockInterruptibly();
                TimeUnit.MILLISECONDS.sleep(10);//更好的触发死锁
                secondLock.lockInterruptibly();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                firstLock.unlock();
                secondLock.unlock();
                System.out.println(Thread.currentThread().getName()+"正常结束!");
            }
        }
    }
}
```

获取锁时限时等待(tryLock())

```
public class ReentrantLockTest {
    static Lock lock1 = new ReentrantLock();
    static Lock lock2 = new ReentrantLock();
    public static void main(String[] args) throws InterruptedException {

        Thread thread = new Thread(new ThreadDemo(lock1, lock2));//该线程先获取锁1,再获取锁2
        Thread thread1 = new Thread(new ThreadDemo(lock2, lock1));//该线程先获取锁2,再获取
1
        thread.start();
        thread1.start();
    }

    static class ThreadDemo implements Runnable {
        Lock firstLock;
        Lock secondLock;
        public ThreadDemo(Lock firstLock, Lock secondLock) {
            this.firstLock = firstLock;
            this.secondLock = secondLock;
        }
        @Override
        public void run() {
            try {
                while(!lock1.tryLock()){
                    TimeUnit.MILLISECONDS.sleep(10);
                }
                while(!lock2.tryLock()){
                    lock1.unlock();
                    TimeUnit.MILLISECONDS.sleep(10);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                firstLock.unlock();
                secondLock.unlock();
                System.out.println(Thread.currentThread().getName()+"正常结束!");
            }
```

```
        }
    }
}
```

Condition接口

```
public class ConditionTest {

    static ReentrantLock lock = new ReentrantLock();
    static Condition condition = lock.newCondition();
    public static void main(String[] args) throws InterruptedException {
    //Condition接口在使用前调用ReentrantLock的lock()方法获得锁
        lock.lock();
        new Thread(new SignalThread()).start();
        System.out.println("主线程等待通知");
        try {
        //调用Condition接口的await()释放锁
            condition.await();
        } finally {
            lock.unlock();
        }
        System.out.println("主线程恢复运行");
    }
    static class SignalThread implements Runnable {

        @Override
        public void run() {
            lock.lock();
            try {
        //调用Condition的signal()方法唤醒线程
                condition.signal();
                System.out.println("子线程通知");
            } finally {
                lock.unlock();
            }
        }
    }
}
```

阻塞队列的简单实现

```
public class MyBlockingQueue<E> {

    int size;//阻塞队列最大容量

    ReentrantLock lock = new ReentrantLock();

    LinkedList<E> list=new LinkedList<>();//队列底层实现

    Condition notFull = lock.newCondition();//队列满时的等待条件
    Condition notEmpty = lock.newCondition();//队列空时的等待条件

    public MyBlockingQueue(int size) {
        this.size = size;
```

```
    }

    public void enqueue(E e) throws InterruptedException {
        lock.lock();
        try {
            while (list.size() ==size)//队列已满,在notFull条件上等待
                notFull.await();
            list.add(e);//入队:加入链表末尾
            System.out.println("入队：" +e);
            notEmpty.signal(); //通知在notEmpty条件上等待的线程
        } finally {
            lock.unlock();
        }
    }

    public E dequeue() throws InterruptedException {
        E e;
        lock.lock();
        try {
            while (list.size() == 0)//队列为空,在notEmpty条件上等待
                notEmpty.await();
            e = list.removeFirst();//出队:移除链表首元素
            System.out.println("出队："+e);
            notFull.signal();//通知在notFull条件上等待的线程
            return e;
        } finally {
            lock.unlock();
        }
    }
}
```

StampedLock

特点

- 所有获取锁的方法,都返回一个邮戳(Stamp);
- Stamp为0表示获取失败,其余都表示成功;
- 所有释放锁的方法,都需要一个邮戳(Stamp),这个Stamp必须是和成功获取锁时得到的Stamp一致;
- StampedLock是不可重入的;(如果一个线程已经持有了写锁,再去获取写锁的话就会造成死锁);
- 支持锁升级跟锁降级;
- 可以乐观读也可以悲观读;
- 使用有限次自旋,增加锁获得的几率,避免上下文切换带来的开销
- 乐观读不阻塞写操作;
- 悲观读阻塞写得操作;

优点

- 相比于ReentrantReadWriteLock,吞吐量大幅提升;

缺点

- api相对复杂,容易用错;
- 内部实现相比于ReentrantReadWriteLock复杂得多;

原理

- 获取锁的时候,会返回一个邮戳(stamp),相当于mysql里的version字段;
- 释放锁的时候,再根据之前的获得的邮戳,去进行锁释放;

注意点

- 如果使用乐观读,一定要判断返回的邮戳是否是一开始获得到的,如果不是,要去获取悲观读锁,再次去取;