JDK 中的内置锁 synchronized 和对象锁 ReentrantLock

屈春河

创建日期：2020-06-07

1 基础知识

在共享内存的计算机架构中，同步机制是并发程序和并行程序不可或缺的功能，用于保护临界区 (Critical Section)。临界区指的是一个使用复用资源（例如：公用设备或共享数据）的程序片段。这些复用资源具有无法被多个线程同时访问的特性，因此线程需要以排他方式执行临界区中的代码，访问复用资源，即仅仅许可一个线程进入临界区段，并且一旦有一个线程位于临界区内，则其他线程必须在临界区的入口处等待，直到临界区内的线程离开为止。同步机制可以划分为两大类

- 阻塞 (blocking) 方式：如果没有获准进入临界区，则线程释放资源并停止运行，等待被系统重新调度。
- 忙等待 (busy-wait) 方式：线程持续测试共享变量，以确定其是否许可进入临界区。

由于阻塞方式需要上下文切换等操作，会增加额外的性能开销。因此，如果临界区的执行时间非常短，那么采用忙等待方式是一个比较好的方案。相反地，如果临界区需要耗时较长的执行时间，那么忙等待方式则会一直空耗 CPU 检查共享变量，为此采用阻塞方式是一个更佳的选择。

锁是广泛使用的一种阻塞同步方式。在 JDK 中提供了两种锁

- 内置锁 synchronized
- 对象锁 ReentrantLock

上述两种锁都是可重入锁，一个线程可以重复地进入同一个锁保护的临界区。也就是说，一个线程在获取一个锁后、在没有释放这个锁之前，还可以再次获取这个锁，而不会被阻塞。

synchronized 是 JDK 中的关键字，用于对方法或者代码块进行加锁保护。因为每个对象内部都会关联一个监视器和相应的等待集合，可以被 synchronized 修饰实现加锁功能，因此 synchronized 方式也被称为为内置锁 (intrinsic lock) 或者监视器锁 (monitor lock)。在经过编译之后，在 synchronized 所保护的代码前后会分别插入 monitorenter 和 monitorexit 两个字节码指令，分别用于获取和释放内置锁。synchronized 的基本使用方式如下面代码所示。

```
public class SharedData {
    private final Object lock = new Object();
    ...
    public void accessSharedData() {
        .....
        synchronized (lock) {
            //access shared data
        }
```

```
    }
     ...
}
```

在使用内置锁的过程中，需要注意如下几个方面

- 尽量采用 final 修饰加锁的变量，以确保这个变量被初始化赋值之后，在程序的运行期间不会再被赋值。否则，一旦被重新赋值，虽然在加锁时采用同一个变量名，但是所引用的对象已经改变，内置锁也会随之不同，使得多个线程可以同时进入临界区。
- 尽量采用对代码块进行加锁，而不是对方法加锁，以尽量减小临界区的大小，提高程序的并发性。临界区中的代码是顺序执行的，因此临界区越大，耗时越长，程序的并发执行能力越低，多线程执行的效率越低。因此，除非是非常简单的方法，否则尽量避免对于方法进行加锁。
- 尽量采用内部对象作为加锁变量，以避免加锁的对象被第三方或者外部系统再次加锁，从而埋下隐患。由于无法限制第三方以何种方式使用和集成，如果第三方也采用相同的对象加锁，那么就有可能出现不可预料的情况。为此，要尽量把加锁变量隐藏到内部，便于系统集成和功能组合。
- 尽量避免使用 String 或 Integer 等类型的变量作为加锁变量，因为对于这些类型而言，看似不同的变量却可能引用同一个对象。如下代码所示，String 数据存储在常量池，导致相同内容的字符串，引用同一个字符串对象，而 Integer 和 Long 会缓存 [-128, 127] 之间的对象，当数字位于此范围时，工厂方法 valueOf 会返回缓存的对象，而不是创建新的对象。

```
    String s1 = "abcde";
    String s2 = "abcde";
    System.out.println(s1==s2);

    Integer i1 = Integer.valueOf(2);
    Integer i2 = Integer.valueOf("2");
    System.out.println(i1==i2);

    Long l1 = Long.valueOf(2);
    Long l2 = Long.valueOf("2");
    System.out.println(l1==l2);
}
```

相对于内置锁，ReentrantLock 的使用要稍微繁琐一些。如下代码所示，其需要以显式的方式获取锁和释放锁。

```
public class SharedData {
    private final ReentrantLock lock = new ReentrantLock();
    ...
    public void accessSharedData()() {
       lock.lock();  // block until condition holds
       //can not insert any codes
       try {
           ...
```

```
        } finally {
            //can not insert any codes
            lock.unlock()
        }
    }
}
```

在使用 ReentrantLock 的过程中，需要注意如下几个方面

- 尽量采用 final 修饰 ReentrantLock 对象，原因与内置锁相同，避免在程序的运行期间被重新赋值。
- 在获取锁的操作 lock 和临界区的入口 try 之间，尽量不要插入代码。一旦在此执行代码，就可能抛出非检查型异常 (Unchecked Exception), 从而跳过后续代码的执行，造成 unlock 无法执行，锁将不会被释放。这使得后续针对这个锁的加锁操作不会成功，获取锁的线程将永远处于阻塞状态。
- 将释放锁的操作 unlock 放置在 finally 块中，以确保无论在任何情况下，已经获得的锁都会被正常释放。此外，在 finally 块中还需要尽量避免在 unlock 之前插入任何可执行代码，以确保不会因为抛出非检查型异常而不执行 unlock 操作。

内置锁的使用较为简单，并且由于 JVM 对于内置锁进行了优化，因此当线程数比较少或者并发度比较低时，内置锁的加锁性能会比较好。ReentrantLock 的使用较为复杂，如果使用不当，非常容易造成不正常的获取锁和释放锁，从而导致系统异常或者错误。但是，ReentrantLock 提供了更丰富、更多样的的锁操作能力，包括

- 提供 fair 和 unfair 两种不同的锁策略。
- 提供多种加锁方式，包括普通加锁以及尝试加锁和带超时的尝试加锁。
- 支持可以被中断的加锁过程 (lockInterruptibly)。
- 一个锁可以关联多个条件 (Condition) 并且支持可中断、不可中断和有时限三种等待方式。
- 能够获得当前持有锁的线程 (getOwner)、没有获得锁而处于阻塞队列中的线程 (getQueuedThreads) 和等待特定条件的线程 (getWaitingThreads)

这些操作能力可以相互组合，更加灵活和更加高效地支持不同的同步场景和需求。

2 公平性

ReentrantLock 的构造函数提供了可选的公平性参数。根据 JDK Doc[Oracle] 的说法，如果设置为真，那么在竞争条件下，会倾向于让等待最长时间的线程获得锁，即最先请求锁的线程将会最先获得锁。否则，如果设置为假，那么不会保证任何特定的访问顺序。采用公平方式，如果 n 个线程以 T1,T2, ...,Tn 顺序依次获取锁，那么即使在并发情况存在线程之间的竞争，公平锁也会保证线程以 T1,T2, ...,Tn 的先后顺序获取锁并进入临界区。在 JDK Doc 还着重声明：在大量线程访问的情况下，使用公平锁的程序将会比使用默认配置 (非公平锁) 情况下的程序呈现更低的整体吞吐，并且在获取锁的时间和确保减小饥饿方面，两种方式仅仅具有较小的差异。因此，除非特殊情况，尽量采用默认设置，即使用非公平锁。

```
public class SharedData {
```

```
    private final ReentrantLock unfairLock = new ReentrantLock();
    private final ReentrantLock fairLock = new ReentrantLock(true);
    ...
}
```

对于需要确保线程依照先后顺序进入临界区的情况，例如在一些资源分配和商品抢购的场景中必须满足先到先得原则，公平锁可能是不二选择。

3 条件 (Condition)

在并发程序中线程的执行会依赖或者影响一些共享的状态或者标志。针对于此，提出了条件 (Condition) 机制。条件是从锁衍生而来的多线程协作机制，其代表了共享的状态或者标志，并能基于这些状态或者标志，阻塞或者唤醒线程。如果一个线程所依赖于的条件（状态或者标志）没有得到满足，那么该线程就停止执行并处于等待状态，直到当其他线程的运行改变了状态或者标志，满足了所依赖的条件，该线程才会被唤醒并继续执行。基于条件，多个线程可以相互协同，合作完成复杂的并发处理功能。例如在经典的生产者/消费者模式中，生产者是一个或多个线程，生成数据或者任务，而消费者是一个或者多个线程，消费数据或者完成任务。在消费者和生产者之间，一个队列用于缓存生产出来的数据或者任务。

- 对于生产者而言，一方面其向队列添加数据或者任务的行为依赖于队列非满的条件，如果队列已满时，则不能再向队列中添加，而是要被阻塞，直到队列非满时为止，另一个方面生产者的添加行为有时也可能会影响队列的状态，即将队列由空变为非空，从而唤醒正在等待的消费者，执行后续的消费行为。
- 对于消费者而言，只有在队列非空的条件下，才能从队列中获取数据或任务，如果队列为空，则需要被阻塞，从而暂停消费行为，直到队列满足非空条件为止，而一旦从队列中获取了数据或任务，则有可能将满队列变为非满，从而唤醒被阻塞的生产者，完成添加过程并继续生产行为。

通过上述的介绍可以看出，通过两个条件（非空和非满）可以指挥两类线程（生产者和消费者）协调工作，完成所期望的功能。

如下代码所示，内置锁隐式地支持单个条件。

```
    synchronized (lock) {
        ... // Perform action to change condition
        lock.notify();// or lock.notifyAll();
     }
```

```
    synchronized (lock) {
        while (<condition does not hold>) {
            try{
                lock.wait();
            } catch (InterruptedException e) {

            }
```

```
        }
         ... // Perform action appropriate to condition
     }
```

由于仅仅支持一个条件，所以采用内置锁实现生产者/消费者模式会非常复杂。

不同于内置锁，ReentrantLock 能够非常简单地实现生产者/消费者模式。如下生产者/消费者模式的示例代码来自 JDK 的 ArrayBlockingQueue 类，其中省略了一些与锁和条件无关的代码。基于 ReentrantLock，在 ArrayBlockingQueue 内部构造了两个不同的条件 notFull(非满) 和 notEmpty(非空)，用于协同 put(生产) 和 take(消费) 操作，以避免队列溢出或者取出空值的异常情况发生。

```
public class ArrayBlockingQueue<E> extends AbstractQueue<E>
        implements BlockingQueue<E>, java.io.Serializable {
    ...
    /** Main lock guarding all access */
    final ReentrantLock lock;

    /** Condition for waiting takes */
    private final Condition notEmpty;

    /** Condition for waiting puts */
    private final Condition notFull;
    ...

    public ArrayBlockingQueue(int capacity, boolean fair) {
        ...
        lock = new ReentrantLock(fair);
        notEmpty = lock.newCondition();
        notFull =  lock.newCondition();
    }
    ...

   public void put(E e) throws InterruptedException {
        checkNotNull(e);
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        try {
            while (count == items.length)
                notFull.await();
            enqueue(e);
        } finally {
            lock.unlock();
        }
    }
    ...
    private void enqueue(E x) {
        ...// 在队尾添加一个元素
```

```
        notEmpty.signal();
    }
    ...

    public E take() throws InterruptedException {
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        try {
            while (count == 0)
                notEmpty.await();
            return dequeue();
        } finally {
            lock.unlock();
        }
    }
    ...
    private E dequeue() {
        ...//获取队首元素并赋值给 x，然后删除队首元素
        notFull.signal();
        return x;
    }

}
```

无论是内置锁，还是 ReentrantLock，在使用条件/信号时都需要注意如下两个方法

- 发送信号和等待信号必须位于对应锁保护的临界区内,即首先获得锁,才能调用对应该锁的 wait(或者 await) 和 signal(或者 signalAll) 方法，否则将会抛出 IllegalMonitorStateException。
- wait 必须在 while 循环中, 即无论首次调用 wait 之前，还是调用 wait 被唤醒之后，都需要判断状态和标志是否满足，以确定是继续等待条件，还是执行后续代码。之所以如此，是因为如下三个原因
  - 在开始调用 wait 之前，需要检查状态或者标志，避免出现永远不被唤醒的异常情况。例如在生产者/消费者模式中，如果队列为空，并且所有的消费者线程都处于等待非空条件的状态，此时如果有生产者线程不判断是否非满条件，就直接调用 wait，那么无论是消费者线程，还是生产者线程，都会一直处于等待状态，而不会被唤醒。
  - 防止被意外被唤醒。依赖于底层的平台，可能会出现一些意外状况，即在没有发送信号的情况下，等待信号的线程却被异常唤醒。出现这种状况时，线程所依赖的条件尚未被满足，还无法继续执行。
  - 防止条件已经改变。从发出信号到线程在接收此信号后被唤醒，可能已经间隔了一段时间。这意味着与发信号的时刻相比，所等待的条件可能已经改变，尤其是采用 signalAll 广播信号时，会唤醒所有的线程。这使得在本线程被唤醒之前，有可能其他线程已经被唤醒，并执行了代码和改变了状态。例如在生产者/消费者模式中，如果多个消费者线程在等待非空条件，如果此时一个生产者线程添加了一个数据或者任务，并采用 signalAll 广播信号，则所有等待非空条件的线程会依次被唤醒，但是由于队列

中仅仅有一个元素，因此仅仅第一个被唤醒的消费者线程能够执行，其他后续被唤醒的线程还需要继续等待。

除了上面两个方面，还有一个特别需要关注的话题，即在发送信号时，是采用 signal，还是采用 signalAll？这两种方式各有不足之处

- notify 方法仅仅唤醒一个等待线程。如果存在多个等待线程的话，除了被唤醒的一个线程外，其他线程还会处于等待状态。一旦处理不当，就可能造成等待线程永远处于等待状态。例如在之前的 ArrayBlockingQueue 例子中，如果新增 putAll(Collection<E> c) 方法，则在该方法内要么调用 c.size() 次 signal 方法，要么调用 signalAll 方法。否则，如果仅仅调用一次 signal 方法，那么当存在多个线程被阻塞在 take 方法时，仅仅有一个线程会被唤醒并从队列中成功取出一个元素，虽然此时队列非空，但是其他调用 take 方法的线程还会一直处于等待状态。
- notifyAll 方法会唤醒所有等待的线程。如果所依赖的条件仅仅许可一个线程执行，即线程在执行后会改变条件，使得后续被唤醒的线程所依赖的条件得不到满足，那么虽然所有的线程都会被 notifyAll 陆续唤醒，但是仅仅有一个线程能够成功运行，其他线程在唤醒并检查条件不满足后，还会继续等待。当等待的线程数较多时，大量线程被唤醒然后又立刻停止运行，会引起频繁的上下文切换，导致严重的性能损失。

针对两种方式的不足之处，建议要针对不同情况，选择 notify 或 notifyAll。如果条件仅仅许可一个线程执行，比如仅仅有一个可用的资源或者向 ArrayBlockingQueue 添加 (put) 一个元素，那么采用 notfiy。反之，如果改变条件后许可多个线程执行，比如消费者生成多个数据或者任务，那么就采用 notifyAll。

4 复杂场景举例

接下来将会介绍几个使用 ReentrantLock 的例子，目的是抛砖引玉，演示如何在复杂场景中使用 ReentrantLock 的方法。需要声明的是这些代码示例忽略了很多实现细节和必要功能，在实际系统中实现这些例子往往需要大量的相关代码和相关功能。

在如下代码示例 1 中，采用 tryLock 重复尝试请求锁，直到获得锁为止，从而实现类似于忙等待的功能。

```
public class SharedData {
    private final ReentrantLock lock = new ReentrantLock();
    ...
    public void accessSharedData()() {
        ...
        while (lock.tryLock()) {
            try{
                ...
            } finally {
                lock.unlock();
            }
        }
```

```
        ...
    }
}
```

显然，上面的例子仅仅表明通过 tryLock 可以实现忙等待方式，但并不是一个好的实现。相对于代码示例 1，如下采用 CAS(Compare and Swap) 方式，从忙等待的性能来说可能会更好。

```
public class SpinLock {
    public SpinLock() {
    }
    public void lock() {
        while (unsafe.compareAndSwapObject(this, flagOffset, 0, 1));
    }
    public void unlock() {
        unsafe.compareAndSwapObject(this, flagOffset, 1, 0);
    }
    private int flag = 0;

    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long flagOffset;
    static {
        try {
            flagOffset = unsafe.objectFieldOffset(SpinLock.class.
                getDeclaredField("flag"));
        } catch (Exception e) {
            throw new Error(e);
        }
    }
}
```

tryLock 最常用的场景是在没有获得锁的情况，需要要执行不同的业务逻辑，例如在执行很多任务时，如果一个线程在执行一个任务过程中没有获得锁，那么该线程不是进入等待状态，而是继续执行其他任务。下面的代码示例 2 就演示了这种情况，即通过线程池 ThreadPoolExecutor 依次执行任务队列中的任务。在处理一个任务的过程中，如果线程没有获得访问资源的许可，那么就将访问资源和后续执行封装为一个新的 Runnable 任务，插入任务队列的尾部，然后终止此任务的执行并返回，最后此线程会被线程池回收并继续执行任务队列头部的其他任务。

```
public class RerunnableTask {
    public void setExecutor(ThreadPoolExecutor executor) {
        this.executor = executor;
    }

    public void accessSharedResource() {
        if (lock.tryLock()) {
            try {
               ... //access shared resource
            } finally {
```

```
                lock.unlock();
            }
        } else {
            //将后续执行封装为新任务插入任务队列
            executor.execute(()->this.accessSharedResource());
            return;
        }
        ... //other codes
    }

    private final ReentrantLock lock = new ReentrantLock();
    private ThreadPoolExecutor executor;
}
```

对于单个任务而言，上述方式可能会增加执行时间，但是由于能够充分利用 CPU 资源，从而在整体上可以提高线程池的任务处理能力。需要说明的是上面的代码示例 2 仅仅是一个非常简单的演示，在实际系统中还需要很多额外的工作，例如在暂停任务执行时需要保存中间状态或者上下文环境，而在继续执行任务时还需要恢复中间状态或者上下文环境。

lockInterruptibly 方法可以实现可中断的加锁方式，即在等待获取锁的过程中能够接收中断信号，并被从等待状态中唤醒处理此中断信号。如果需要线程在等待获取锁的过程中还能够处理紧急的情况，那么建议采用 lockInterruptibly。比如如下两个应用场景

- 打破死锁状态。在采用 lockInterruptibly 方式申请锁时，如果检查到两个线程之间产生了死锁，那么可以选择一个死锁线程，向其发送中断信号，将其唤醒并让其释放资源，从而解决死锁问题。
- 优雅退出系统。在接收到通过命令 kill 发送的终止信号时，程序如果被异常关闭，那么可能出现各种意外状况，比如数据不一致等。为此，需要优雅的关闭系统，即程序在释放资源和记录状态后主动退出运行。

如下的代码示例 3 是针对优雅退出情况的一个简单实现

```
public class StoppableTask {
    private final GoodReentrantLock lock = new GoodReentrantLock();
    private volatile boolean isRunning = true;
    ...

    public void accessSharedData {
       try{
            lock.lockInterruptibly();
            try {
                ...//access shared data
            } finally {
                lock.unlock();
            }
       } catch (InterruptedException e) {
            if (isRunning) {
                ...//被错误唤醒，需要重新执行
```

```
            } else {
                ...//关闭资源和记录状态
            }
        }

    }
    ...
    public void stop() {
        isRunning = false;
        lock.getQueuedThreads().stream().forEach(thread->thread.interrupt());
        final Thread owner = lock.getOwner();
        if (null != owner)
            owner.interrupt();
    }
}
```

在程序启动时，需要注册"TERM" 信号和相对应的 SignalHandler 对象。后者在接收到"TERM" 信号后，会逐个调用 StoppableTask 对象的 stop 方法，终止其运行。此外，还需要说明的是 getOwner 和 getQueuedThreads 在 ReentrantLock 中为 protected 方法。为此，创建了一个继承自 ReentrantLock 的新类 GoodReentrantLock，用于将上述两个方法转化为 public 方法。

```
public class GoodReentrantLock extends ReentrantLock {
    public GoodReentrantLock() {
        super();
    }

    public GoodReentrantLock(boolean fair) {
        super(fair);
    }

    @Override
    public Thread getOwner() {
        return super.getOwner();
    }

    @Override
    public Collection<Thread> getQueuedThreads() {
        return super.getQueuedThreads();
    }
}
```

在代码示例 3 中针对锁的持有线程，执行了中断操作，用于终止其运行。除了上面优雅退出的场景，在其他很多场景中也需要锁的持有线程能够尽快终止执行并释放锁，例如

- 存在优先级更高的线程请求锁，比如一些交互应用或实时应用的线程，需要以抢占方式优先获得锁的使用权。
- 持有锁的线程可能出现运行异常或者错误，例如拥有锁的线程已经执行过长的时间，判断

其可能出现异常的网络或者功能调用，需要终止其执行。

此外，通过 interrupt 终止线程的执行，还需要依赖如下两个条件

- 临界区没有屏蔽中断信号。如果在临界区内存在不可中断的调用或者屏蔽了中断信号，那么中断信号可能会被丢弃，持有锁的线程不会接收到中断信号。
- 需要共享的标志和状态，类似于代码示例 3 中的共享变量 isRunning，用于表示是否需要终止运行，以避免被错误中断的情况。

如果不满足上述两个条件，那么即使发送了中断信号，也无法使持有锁的线程成功退出。还需要说明的，即使成功发送了中断并且正常接收了中断，持有锁的线程也不一定会立即响应中断信号，可能还需要等待一段时间。

参考文献

CANTRILL B, BONWICK J, October 24, 2008. Real-world concurrency[J]. ACM Queue.

ORACLE. Java se 8 doc: Reentrantlock[EB/OL]. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html.