一文带你整明白 Java 的 N 种锁

作者：wangning1018
原文链接：https://ld246.com/article/1575451871336
来源网站：链滴

温馨提示：本文内容较长废话较多，如有心脏病、精神病史等请酌情看。

一、概述

本文源码基于openJDK8u。在阅读本文前，你需要对**并发**有所了解。

在并发中，为了**解决程序中多个进程和线程对资源的抢占问题**，在 Java 中引入了锁的概念。

各种各样的锁，对于初碰 Java 并发的同学来说，面对多达 20 种的锁，瞬间懵逼，退游戏这把鸡劳不吃了.....

其实不要紧张，虽然锁的种类很多，但是都是根据其特性衍生出来的概念而已，如果你对 Java 锁不很清晰，希望这篇文章能够对你有所帮助。朋友们，如果你不会做饭或者不知道吃什么请关注我麻辣子感谢您的双...呸，学好 Java ， 拒绝沉迷某音，哈哈哈哈哈~

下面，是一张关于锁的思维导图，带大家有一个总体的认识，配合此图食用效果更佳哈。

ps：如果看不清楚可以点击点击原图查看哦。

二、synchronized 带你跑毒

为了方便大家由浅入深（怀疑作者开车但是没有证据...），我们从大家比较熟悉的 synchronized 说。对于 Java 使用者来说，synchronized 关键字是实现锁的一种重要方式。

```
package com.aysaml.demo.test;


import com.google.common.util.concurrent.ThreadFactoryBuilder;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ThreadFactory;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

/**
 * SynchronizedDemo 示例
 *
 * @author wangning
 * @date 2019-11-26
 */
```

```
public class SynchronizedDemo {

    private static int count = 0;


    private static void addCount() {
        count++;
    }

    public static void main(String[] args) {

        int loopCount = 1000;

        ThreadFactory namedThreadFactory = new ThreadFactoryBuilder()
                .setNameFormat("demo-pool-%d").build();
        ExecutorService executorService = new ThreadPoolExecutor(10, 1000,
                60L, TimeUnit.SECONDS,
                new LinkedBlockingQueue<>(10), namedThreadFactory, new ThreadPoolExecutor.
bortPolicy());

        for (int i = 0; i < loopCount; i++) {
            Runnable r = SynchronizedDemo::addCount;
            executorService.execute(r);
        }
        executorService.shutdown();
        System.out.println(SynchronizedDemo.count);
    }

}
```

措不及防的代码粘贴，哈哈哈。

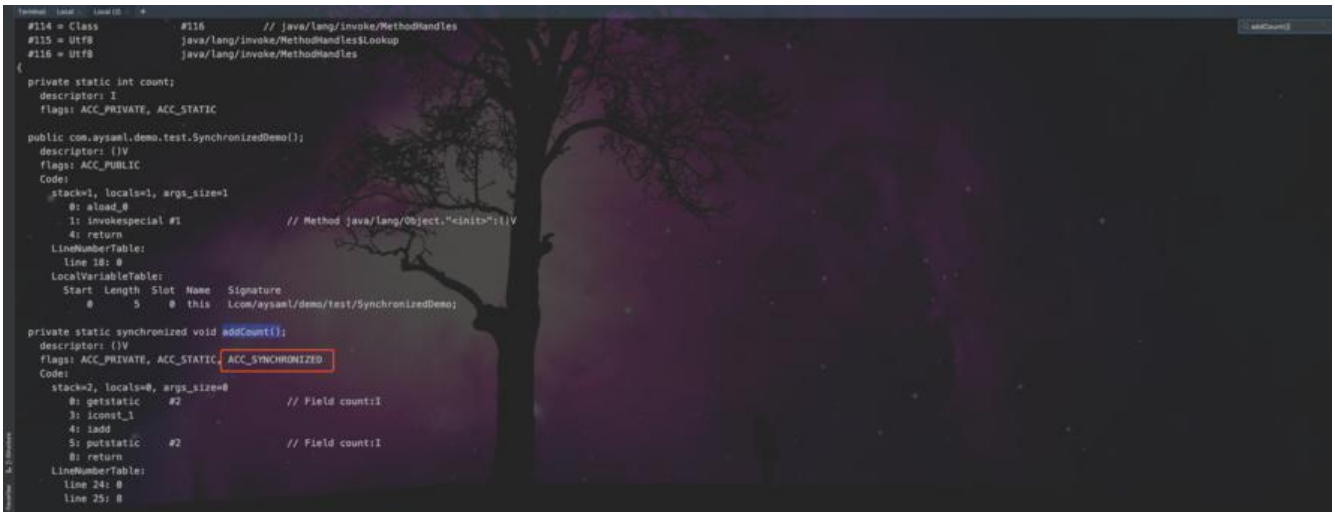
上面是比较经典的线程并发问题示例。运行这段代码，得到的结果多种多样，996、997、998.....

结果并不像我们预期的那样是1000，遇到多线程资源竞争时我们可能第一反应的就是加 synchronized 简单粗暴，于是变成下边这样：

```
private static synchronized void addCount() {
        count++;
    }
```

在累加方法加上 synchronized，就给这个方法加了锁，如此，每次执行的结果就符合了我们的预期。

Java 内置了一个反编译工具 javap 可以反编译字节码文件，通过执行命令 javap -verbose -p SynchronizedDemo.class 可以看到上述这个类的字节码文件。



查找被加锁的方法 addCount() 发现，synchronized **修饰方法** 时是通过 **ACC_SYNCHRONIZED** 标符号指定该方法是一个同步方法，从而执行相应的同步调用。

同样查看字节码，可以知道 synchronized **修饰代码块** 时 通过**monitorenter** 和 **monitorexit** 指令解决同步问题，其中 **monitorenter** 指令指向同步代码块的开始位置，**monitorexit** 指令指明同步码块的结束位置。

三、各种锁的解释，猥琐发育捡枪捡子弹

这部分只是简单说一下各种锁以及相关概念，让大家有一个简单了解，其中相应的在 Java 中的实现用较长的篇幅做介绍。

偏向锁、轻量级锁、重量级锁

在程序第一次执行到 synchronized 代码块的时候，锁对象变成 **偏向锁** ，即偏向于第一个获得它的程的锁。在程序第二次执行到改代码块时，线程会判断此时持有锁的线程是否就是它自己，如果是就续往下面执行。值得注意的是，在第一次执行完同步代码块时，并不会释放这个偏向锁。从效率角度看，如果第二次执行同步代码块的线程一直是一个，并不需要重新做加锁操作，没有额外开销，效率高。

前面说的只有一个线程同步执行代码块只是理想的状态下（如果只有一个线程也不用考虑并发的问题，虽然这么说有点不太严谨哈...），一旦有第二个线程加入 **锁竞争** ，偏向锁就自动升级为 **轻量级锁** 。这里不同情况需值得注意：当第二个线程想要获取锁时，且这个锁时偏向锁时，会判断当前持有锁的程是否仍然存活，如果改**持有锁的线程没有存活**，**那么偏向锁并不会升级为轻量级锁** 。什么是锁竞争：**一个线程想要获取另一个线程持有的锁** 。

在此状态下各个线程继续做锁竞争，没有抢到锁的线程循环判断是否能够成功获取锁，这种状态称为**旋** ，故轻量级锁是一种 **自旋锁** 。虚拟机中有个计数器用来记录自旋次数，默认允许循环10次，这个可以通过虚拟机参数-XX：PreBlockSpin来进行修改。如果锁竞争特别严重，达到这个自旋次数最大限度，轻量级锁就会升级为**重量级锁**。当其他线程尝试获取锁的时候，发现现在的锁是重量级锁，则接将自己挂起，等待将来被唤醒。

关于这块的更多信息，可以参考《Synchronized与三种锁态》

公平锁、非公平锁

当一个线程持有的锁释放时，其他线程按照先后顺序，**先申请的先得到锁**，那么这个锁就是**公平锁**。

之，如果**后申请的线程有可能先获取到锁**，就是**非公平锁** 。

Java 中的 ReentrantLock 可以通过其构造函数来指定是否是公平锁，默认是非公平锁。一般来说，用非公平锁可以获得较大的吞吐量，所以推荐优先使用非公平锁。

synchronized 就是一种非公平锁。

乐观锁、悲观锁

先说悲观锁，即在**读数据的时候总认为其他线程会对数据进行修改**，所以采取**加锁**的形式，一旦本线要读取数据时，就加锁，其他线程被阻塞，等待锁的释放。所以悲观锁总结为**悲观加锁阻塞线程。**

在**读数据**时总认为其他线程**不会对数据做修改**，在**更新数据时**会判断其他线程**有没有更新数据**，如果更新，则重新读取，再次尝试更新，循环上述步骤直到更新成功，即为**乐观锁。**

这样来看乐观锁实际上是没有锁的，只是通过一种**比较交换**的方法来保证数据同步，总结为**乐观无锁滚重试。**

CAS（比较和交换）

CAS， 英文直译为 compare and swap，即**比较和交换。**上面乐观锁也说了，其实就是一种比较与换的过程。

简单描述一下就是：**读取到一个值为 A ，在要将这个值更新为B 之前，检查是否等于 A （比较），果是则将 A 更新为 B ，否则什么都不做。**

通过这种方式，可以实现不必使用加锁的方式，就能保证资源在多线程之间的同步，显然，不阻塞线，可以大大提高吞吐量。方式虽好，但是也存在问题。

- **ABA 问题**，即如果一个值从 A 变为 B 再变回 A 时，这样 CAS 就会认为值没有发生变化。
    - 对于这个问题，已经有了使用版本号的解决方式，即每次变量更新的时候变量的 **版本号**都 **+1** 即由 A->B->A 就变成了 1A->2B->3A 。
- **循环时间长开销大**，如果锁的竞争比较激烈，就会导致 CAS 不断的重复执行，一直循环，耗费 CPU 资源。
- **只能保证一个变量的同步**，显然，由于其特性，CAS 只能保证一个共享变量的原子操作。

可重入锁

可重入锁即**允许多个线程多次获取同一把锁**，那从锁本身的角度来看，就是可以重新进入该锁。比如一个递归函数里面有加锁操作，如果这个锁不阻塞自己，就是**可重入锁**，故也称**递归锁** 。

再看上面的偏向锁、轻量级锁、重量级锁可以知道**synchronized关键字加锁是可重入的** ，不仅如此，**DK 中实现 Lock 接口的锁都是可重入的** 。感兴趣的读者可以自行了解怎么实现不可重入锁，这里只

一下锁的定义。

可中断锁

如果线程A持有锁，线程B等待获取该锁。由于线程A持有锁的时间过长，线程B不想继续等待了，我可以让线程B中断自己或者在别的线程里中断它，这种就是**可中断锁**。

在 Java 中，synchronized就是**不可中断锁**，而Lock的实现类都是**可中断锁**。

独享锁、共享锁

独享锁亦称**互斥锁**，**排它锁**，容易理解这种锁**每次只允许一个线程持有**。反之，就是**共享锁**啦。

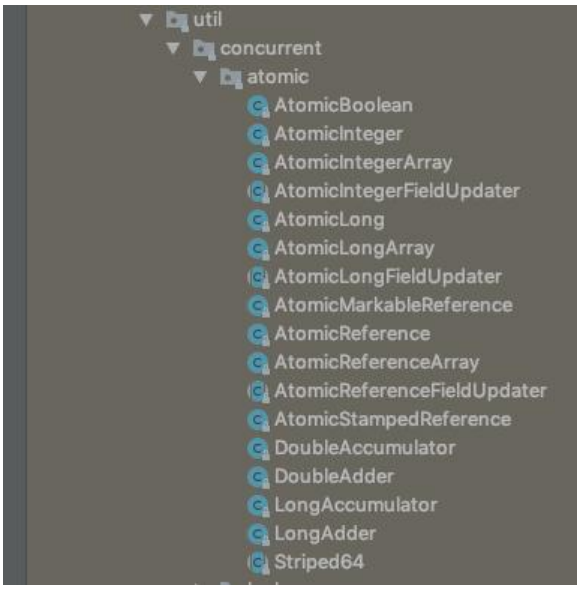
读锁、写锁

上面说独享锁和共享锁，其实读写锁就是其最典型的锁。写锁是独享锁，读锁是共享锁。在后面我们着重说一下Java 中的读写锁实现。

三、CAS 在 Java 中的实现，带上这把 M4-CAS

通过上面对 CAS 的简单介绍，相信大家对 CAS 也有了一个比较简单的概念：**通过比较和交换实现单变量的线程安全** 。

JDK 中对 CAS 的实现在 java.util.concurrent.atomic 包中：



类名	描述
AtomicBoolean	可以用原子方式更新的 boolean 值。
AtomicInteger	可以用原子方式更新的 int 值。
AtomicIntegerArray	可以用原子方式更新其元的 int 数组。
AtomicIntegerFieldUpdater<T>	基于反射的

用工具，可以对指定类的指定 volatile int 字段进行原子更新。

AtomicLong 可以用原子方式更新的 long 值。

AtomicLongArray 可以用原子方式更新其元素的 ong 数组。

AtomicLongFieldUpdater<T> 基于反射的实 工具，可以对指定类的指定 volatile long 字段进行原子更新。

AtomicMarkableReference<V> AtomicMark bleReference 维护带有标记位的对象引用，可以原子方式对其进行更新。用来解决 ABA 问题，只关 有没有被修改过。

AtomicReference<V> 可以用原子方式更新的 象引用。

AtomicReferenceArray<E> 可以用原子方式 新其元素的对象引用数组。

AtomicReferenceFieldUpdater<T,V> 基于反 的实用工具，可以对指定类的指定 volatile 字段进行原子更新。

AtomicStampedReference<V> AtomicStam edReference 维护带有整数“标志”的对象引用，可以用原子方式对其进行更新。用来解决 ABA 问 ，与上面的 AtomicMarkableReference<V> 相比，除了关系有没有被修改过之外，还关心修改了 次。

以 AtomicInteger 为例，看看它是如何保证对 int 的操作线程安全的。

```
package java.util.concurrent.atomic;
import java.util.function.IntUnaryOperator;
import java.util.function.IntBinaryOperator;
import sun.misc.Unsafe;

public class AtomicInteger extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 6214790243416807050L;

    // 使用 Unsafe.compareAndSwapInt 进行数据更新
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    // 内存偏移量，即内存地址
    private static final long valueOffset;

    static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    private volatile int value;

    /**
     * 带有初值的构造函数
     *
     * @param initialValue the initial value
     */
    public AtomicInteger(int initialValue) {
```

```
        value = initialValue;
    }

    /**
     * 无参构造函数默认值为0
     */
    public AtomicInteger() {
    }

    /**
     * 获得当前值
     *
     * @return the current value
     */
    public final int get() {
        return value;
    }

    /**
     * 设置所给值，因为value是使用volatile关键字修饰，所以一经修改，其他线程会立即看到value
修改
     *
     * @param newValue the new value
     */
    public final void set(int newValue) {
        value = newValue;
    }

    /**
     * 设置给定的值，通过调用Unsafe的延迟设置方法不保证结果被其他线程立即看到
     *
     * @param newValue the new value
     * @since 1.6
     */
    public final void lazySet(int newValue) {
        unsafe.putOrderedInt(this, valueOffset, newValue);
    }

    /**
     * 原子方式设置新值，返回旧值
     *
     * @param newValue the new value
     * @return the previous value
     */
    public final int getAndSet(int newValue) {
        return unsafe.getAndSetInt(this, valueOffset, newValue);
    }

    /**
     * 使用 CAS 方式设置新值，成功返回true
     *
     * @param expect the expected value
     * @param update the new value
     * @return {@code true} if successful. False return indicates that
```

```
     * the actual value was not equal to the expected value.
     */
    public final boolean compareAndSet(int expect, int update) {
        return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
    }

    /**
     * 使用 CAS 方式更新值
     *
     * <p><a href="package-summary.html#weakCompareAndSet">May fail
     * spuriously and does not provide ordering guarantees</a>, so is
     * only rarely an appropriate alternative to {@code compareAndSet}.
     *
     * @param expect the expected value
     * @param update the new value
     * @return {@code true} if successful
     */
    public final boolean weakCompareAndSet(int expect, int update) {
        return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
    }

    /**
     * 原子增加
     *
     * @return the previous value
     */
    public final int getAndIncrement() {
        return unsafe.getAndAddInt(this, valueOffset, 1);
    }

    /**
     * 原子减1
     *
     * @return the previous value
     */
    public final int getAndDecrement() {
        return unsafe.getAndAddInt(this, valueOffset, -1);
    }

    /**
     * 原子增加给定值
     *
     * @param delta the value to add
     * @return the previous value
     */
    public final int getAndAdd(int delta) {
        return unsafe.getAndAddInt(this, valueOffset, delta);
    }

    /**
     * Atomically increments by one the current value.
     *
     * @return the updated value
     */
```

```
public final int incrementAndGet() {
    return unsafe.getAndAddInt(this, valueOffset, 1) + 1;
}

/**
 * Atomically decrements by one the current value.
 *
 * @return the updated value
 */
public final int decrementAndGet() {
    return unsafe.getAndAddInt(this, valueOffset, -1) - 1;
}

/**
 * Atomically adds the given value to the current value.
 *
 * @param delta the value to add
 * @return the updated value
 */
public final int addAndGet(int delta) {
    return unsafe.getAndAddInt(this, valueOffset, delta) + delta;
}

/**
 * Atomically updates the current value with the results of
 * applying the given function, returning the previous value. The
 * function should be side-effect-free, since it may be re-applied
 * when attempted updates fail due to contention among threads.
 *
 * @param updateFunction a side-effect-free function
 * @return the previous value
 * @since 1.8
 */
public final int getAndUpdate(IntUnaryOperator updateFunction) {
    int prev, next;
    do {
        prev = get();
        next = updateFunction.applyAsInt(prev);
    } while (!compareAndSet(prev, next));
    return prev;
}

/**
 * Atomically updates the current value with the results of
 * applying the given function, returning the updated value. The
 * function should be side-effect-free, since it may be re-applied
 * when attempted updates fail due to contention among threads.
 *
 * @param updateFunction a side-effect-free function
 * @return the updated value
 * @since 1.8
 */
public final int updateAndGet(IntUnaryOperator updateFunction) {
    int prev, next;
```

```
    do {
        prev = get();
        next = updateFunction.applyAsInt(prev);
    } while (!compareAndSet(prev, next));
    return next;
}

/**
 * Atomically updates the current value with the results of
 * applying the given function to the current and given values,
 * returning the previous value. The function should be
 * side-effect-free, since it may be re-applied when attempted
 * updates fail due to contention among threads.  The function
 * is applied with the current value as its first argument,
 * and the given update as the second argument.
 *
 * @param x the update value
 * @param accumulatorFunction a side-effect-free function of two arguments
 * @return the previous value
 * @since 1.8
 */
public final int getAndAccumulate(int x,
                                  IntBinaryOperator accumulatorFunction) {
    int prev, next;
    do {
        prev = get();
        next = accumulatorFunction.applyAsInt(prev, x);
    } while (!compareAndSet(prev, next));
    return prev;
}

/**
 * Atomically updates the current value with the results of
 * applying the given function to the current and given values,
 * returning the updated value. The function should be
 * side-effect-free, since it may be re-applied when attempted
 * updates fail due to contention among threads.  The function
 * is applied with the current value as its first argument,
 * and the given update as the second argument.
 *
 * @param x the update value
 * @param accumulatorFunction a side-effect-free function of two arguments
 * @return the updated value
 * @since 1.8
 */
public final int accumulateAndGet(int x,
                                  IntBinaryOperator accumulatorFunction) {
    int prev, next;
    do {
        prev = get();
        next = accumulatorFunction.applyAsInt(prev, x);
    } while (!compareAndSet(prev, next));
    return next;
}
```

```
    /**
     * Returns the String representation of the current value.
     * @return the String representation of the current value
     */
    public String toString() {
        return Integer.toString(get());
    }

    /**
     * Returns the value of this {@code AtomicInteger} as an {@code int}.
     */
    public int intValue() {
        return get();
    }

    /**
     * Returns the value of this {@code AtomicInteger} as a {@code long}
     * after a widening primitive conversion.
     * @jls 5.1.2 Widening Primitive Conversions
     */
    public long longValue() {
        return (long)get();
    }

    /**
     * Returns the value of this {@code AtomicInteger} as a {@code float}
     * after a widening primitive conversion.
     * @jls 5.1.2 Widening Primitive Conversions
     */
    public float floatValue() {
        return (float)get();
    }

    /**
     * Returns the value of this {@code AtomicInteger} as a {@code double}
     * after a widening primitive conversion.
     * @jls 5.1.2 Widening Primitive Conversions
     */
    public double doubleValue() {
        return (double)get();
    }

}
```

源码中的注释写的很详细，写注释写到一半决定不做谷歌翻译了...主要是通过 Unsafe 提供的 compar AndSwapInt(Object var1, long var2, int var4, int var5) 等方法来实现 CAS 原子操作，Unsafe 提了执行低级别、不安全操作的方法，如直接访问系统内存资源、自主管理内存资源等。

CAS操作包含三个操作数---内存位置、预期原值及新值。执行 CAS 操作的时候，将内存位置的值与期原值比较，如果相匹配，那么处理器会自动将该位置值更新为新值，否则，处理器不做任何操作。们都知道，CAS 是一条 CPU 的原子指令（ cmpxchg 指令），不会造成所谓的数据不一致问题，Uns fe 提供的 CAS 方法（如 compareAndSwapXXX ）底层实现即为 CPU 指令 cmpxchg 。如果小伙

对其感兴趣可以参考 《Java魔法类：Unsafe应用解析》

四、AQS 给你穿上三级甲

AQS，全名 AbstractQueuedSynchronizer，直译为抽象队列同步器，是构建锁或者其他同步组件基础框架，可以解决大部分同步问题。实现原理可以简单理解为：**同步状态(state) + FIFO 线程等队列** 。

● 资源 state

AQS使用了一个 int 类型的成员变量 state 来表示同步状态，使用了 volatile 关键字来保证线程间的见性，当 state > 0 时表示已经获取了锁，当 state = 0 时表示释放了锁。它提供了三个方法（getSta e()、setState(int newState)、compareAndSetState(int expect,int update)）来对同步状态state 行操作，确保对state的操作是安全的。

 而对于不同的锁，state 也有不同的值：

- 独享锁中 state =0 代表释放了锁，state = 1 代表获取了锁。
- 共享锁中 state 即持有锁的数量。
- 可重入锁 state 即代表重入的次数。
- 读写锁比较特殊，因 state 是 int 类型的变量，为 32 位，所以采取了中间切割的方式， **高 16 位标识读锁的数量** ，**低 16 位标识写锁的数量** 。

● FIFO 线程等待队列

实现队列的方式无外乎两种，一是使用数组，二是使用 Node 。AQS 使用了 Node 的方式实现队列。

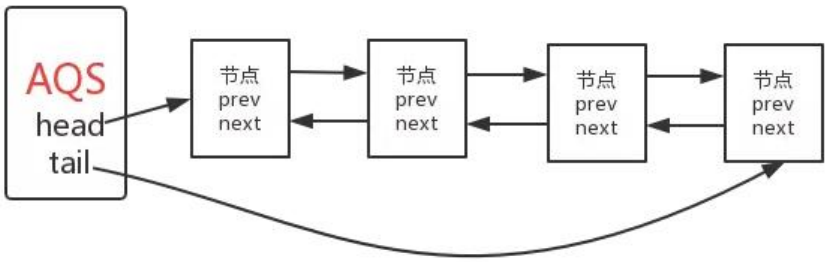
```
static final class Node {
    /** 标记一个节点是在共享模式，默认为共享模式 */
    static final Node SHARED = new Node();
    /** 标记一个节点为独占模式 */
    static final Node EXCLUSIVE = null;


    static final int CANCELLED =  1;

    static final int SIGNAL    = -1;

    static final int CONDITION = -2;

    static final int PROPAGATE = -3;

    /** 以此变量来表示当前线程的状态 */
    volatile int waitStatus;

  /** 前驱 */
    volatile Node prev;

  /** 后继 */
    volatile Node next;

  /** 用于保存线程 */
    volatile Thread thread;
```

```
    /** 保存下一个处于等待状态的Node */
        Node nextWaiter;

        /**
         * 用来判断是否是共享模式
         */
        final boolean isShared() {
            return nextWaiter == SHARED;
        }

        /**
         * Returns previous node, or throws NullPointerException if null.
         * Use when predecessor cannot be null.  The null check could
         * be elided, but is present to help the VM.
         *
         * @return the predecessor of this node
         */
        final Node predecessor() throws NullPointerException {
            Node p = prev;
            if (p == null)
                throw new NullPointerException();
            else
                return p;
        }

        Node() {    // 无参构造方法，默认为共享模式
        }

        Node(Thread thread, Node mode) {     // 用于构造下一个等待线程Node节点
            this.nextWaiter = mode;
            this.thread = thread;
        }

        Node(Thread thread, int waitStatus) { // 用于构造带有本线程状态的Node
            this.waitStatus = waitStatus;
            this.thread = thread;
        }
    }
```

在 AQS 中定义了两个节点,分别为头尾节：

```
    /** 等待队列的头结点，作为队列的初始化节点，只能通过setHead()方法设置值，
     *      而这个方法将Node的变量值都置空，便于及时GC。当其有值时，必须保证waiteStatus为CA
CELLED状态。
     */
    private transient volatile Node head;

    /**
     * 用于保存线程等待队列的尾结点。
     * 通过enq()方法设置值。
     */
    private transient volatile Node tail;
```

这个队列的结构见下图：



AQS 的一堆方法，按照获取锁和解锁的维度可以分为下面这样：

获取锁相关方法

方法	描述
acquire(int arg)	独占模式获取锁，忽略中断。
acquireInterruptibly(int arg)	独占模式获取锁，果被中断则中止。
acquireShared(int arg)	共享模式获取锁，忽略断。
acquireSharedInterruptibly(int arg)	共享模式取锁，如果被中断则中止。
tryAcquire(int arg)	尝试在独占模式获取锁。**由类自行实现。**
tryAcquireNanos(int arg, long nanosTimeout)	试在独占模式获取锁，如果被中断则中止，如果到了给定超时时间 nanosTimeout ，则会失败。
tryAcquireShared(int arg)	尝试在共享模式获取。
tryAcquireSharedNanos(int arg, long nanosTimeout)	试在共享模式获取锁，如果被中断则中止，如果到了给定超时时间 nanosTimeout ，则会失败。
addWaiter(Node mode)	将当前线程加入到CL队列队尾。
acquireQueued(final Node node, int arg)	当线程会根据公平性原则来进行阻塞等待,直到获取锁为止；并且返回当前线程在等待过程中有没有中断。
selfInterrupt()	产生一个中断。

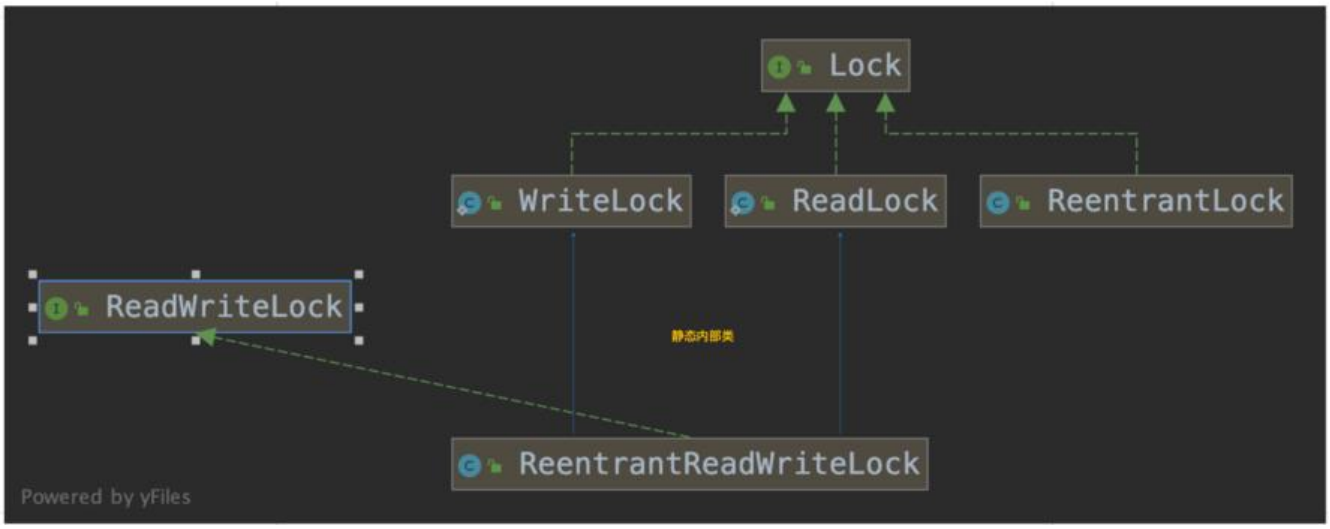
解锁相关方法

方法	描述
release(int arg)	以独占模式释放对象。
releaseShared(int arg)	以共享模式释放对象。

tryRelease(int arg) 下的一个释放。**由子类自行实现。**	试图设置状态来反映独占模
tryReleaseShared(int arg) 享模式下的一个释放。	试图设置状态来反映
unparkSuccessor(Node node)	用来唤醒节点。

五、Lock 带你吃鸡

除了 synchronized 外，在 Java 中还有 Lock 接口的一系列实现来加锁。



如上绿色虚线表示实现，WriteLock、ReadLock、ReentrantLock 都实现了 Lock 接口，三者分别应读锁、写锁和可重入锁，其中 ReadWriteLock 定义了读锁和写锁，ReentrantReadWriteLock 以态内部类的形式实现了读写锁。

首先创建一个单例读写锁：

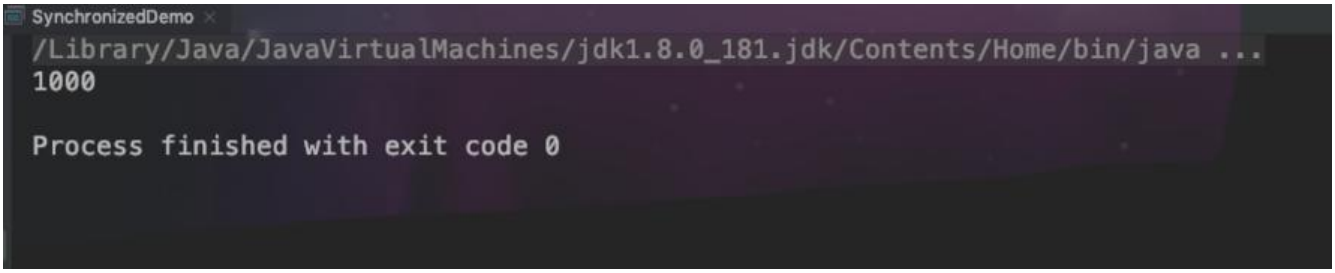
```
package com.aysaml.demo.test;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

/**
 * 单例读写锁
 *
 * @author wangning
 * @date 2019-11-26
 */
public enum Locker {

    instance;

    private Locker() {
    }

    private static final ReadWriteLock lock = new ReentrantReadWriteLock();
```

```
    public Lock writeLock() {
        return lock.writeLock();
    }
}
```

如此可以在上述的累加方法上面加锁做同步：

```
 private static void addCount() {
        Lock locker = Locker.instance.writeLock();
        locker.lock();
        count++;
        locker.unlock();
    }
```

看下运行结果，如愿以偿（总感觉这个词用在这不太对，真是实在想不出什么词了，哈哈，意思你们就好）。



两种加锁方式比较：

synchronized属于互斥锁，任何时候只允许一个线程的读写操作，其他线程必须等待；

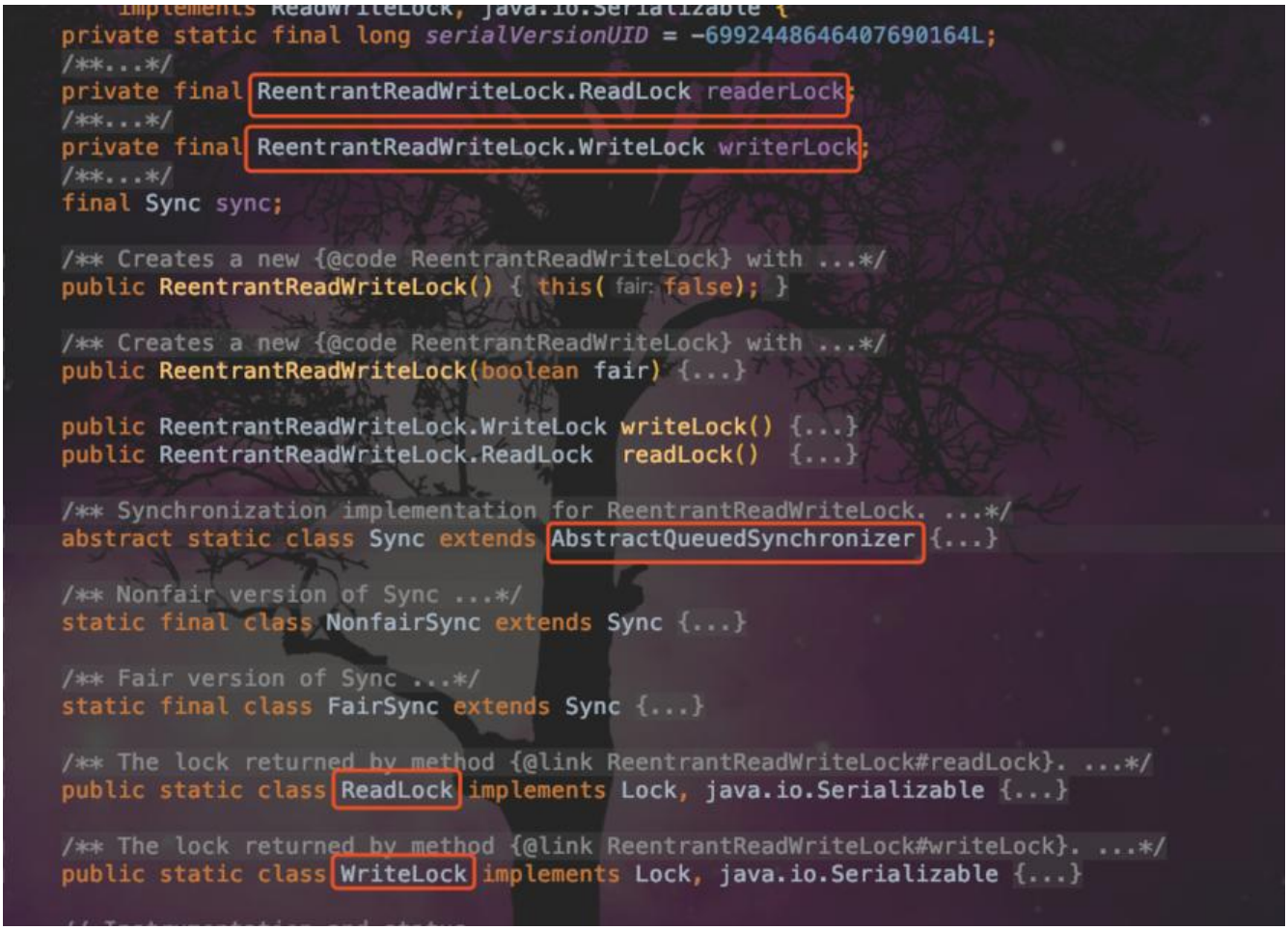
ReadWriteLock允许多个线程获得读锁，但只允许一个线程获得写锁，效率相对较高。

看了上面的 Lock 接口的实现图，我们知道在 Java 中锁有三个重要实现，下面一一来看。

读锁写锁抽象队列同步器三级甲

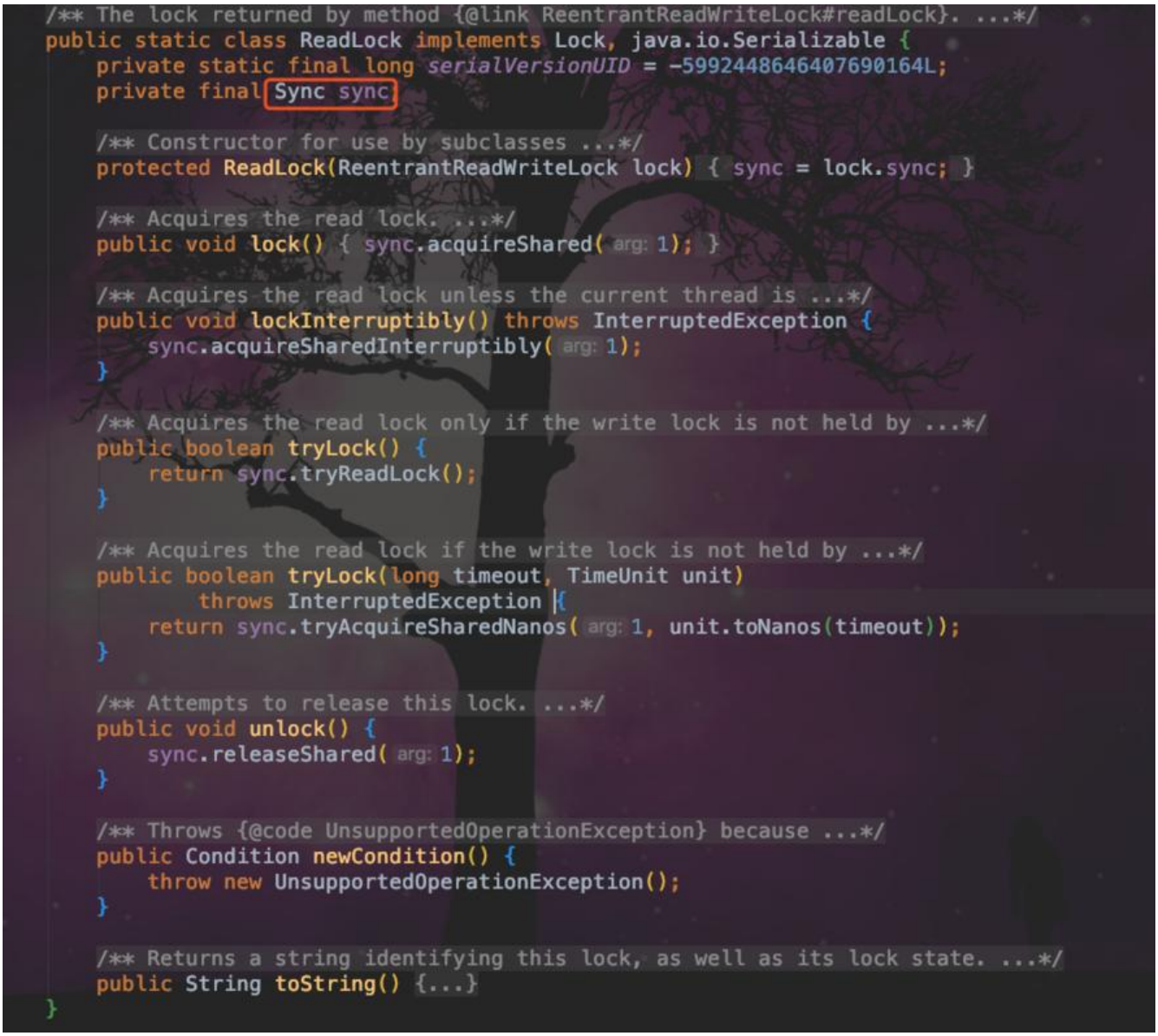
上面说了读写锁属于共享锁，即允许同一时刻有多个线程获取锁。在一些业务中，读的操作比写的操多，相比较 synchronized 而言，读写锁采用 CAS 方式保证资源同步，所以使用读写锁可以大大增吞吐量。

在Java 中 ReentrantReadWriteLock 中以内部类的形式实现了读写锁，如下：



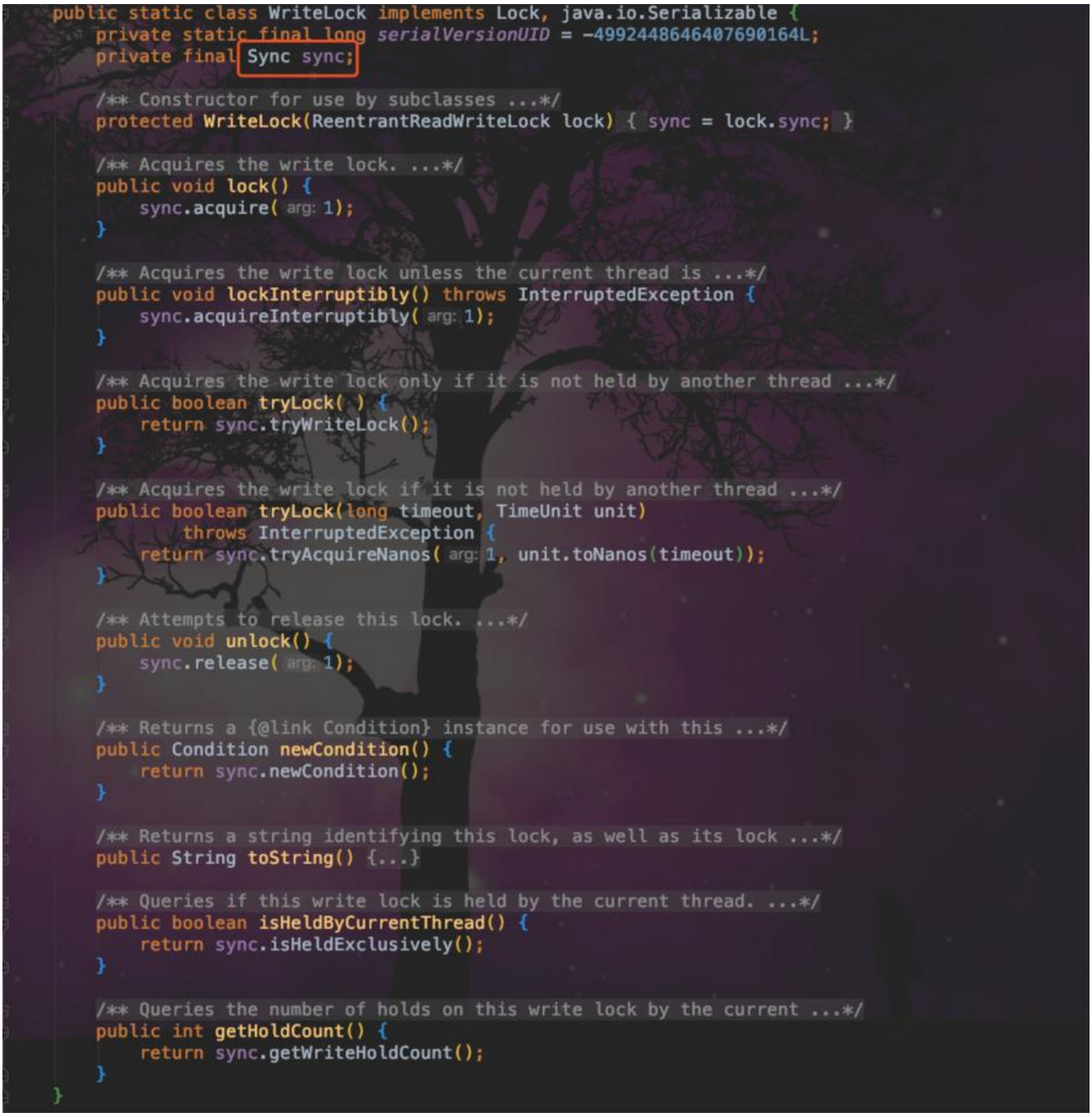
再接着分别看他们的实现：

- ReadLock



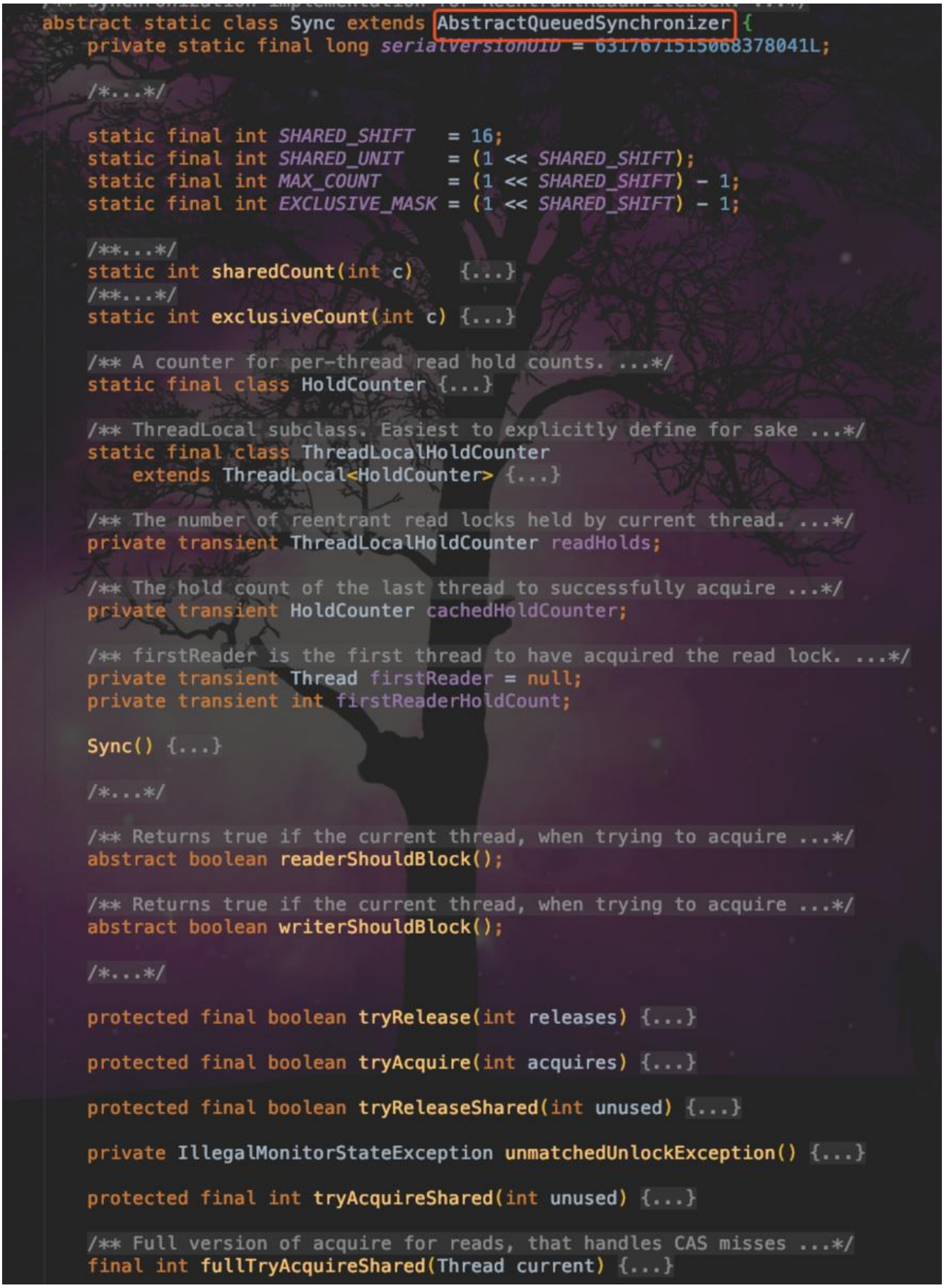
```java
/** The lock returned by method {@link ReentrantReadWriteLock#readLock}. ...*/
public static class ReadLock implements Lock, java.io.Serializable {
    private static final long serialVersionUID = -5992448646407690164L;
    private final Sync sync;

    /** Constructor for use by subclasses ...*/
    protected ReadLock(ReentrantReadWriteLock lock) { sync = lock.sync; }

    /** Acquires the read lock. ...*/
    public void lock() { sync.acquireShared( arg: 1); }

    /** Acquires the read lock unless the current thread is ...*/
    public void lockInterruptibly() throws InterruptedException {
        sync.acquireSharedInterruptibly( arg: 1);
    }

    /** Acquires the read lock only if the write lock is not held by ...*/
    public boolean tryLock() {
        return sync.tryReadLock();
    }

    /** Acquires the read lock if the write lock is not held by ...*/
    public boolean tryLock(long timeout, TimeUnit unit)
            throws InterruptedException {
        return sync.tryAcquireSharedNanos( arg: 1, unit.toNanos(timeout));
    }

    /** Attempts to release this lock. ...*/
    public void unlock() {
        sync.releaseShared( arg: 1);
    }

    /** Throws {@code UnsupportedOperationException} because ...*/
    public Condition newCondition() {
        throw new UnsupportedOperationException();
    }

    /** Returns a string identifying this lock, as well as its lock state. ...*/
    public String toString() {...}
}
```

- WriteLock



```java
public static class WriteLock implements Lock, java.io.Serializable {
    private static final long serialVersionUID = -4992448646407690164L;
    private final Sync sync;

    /** Constructor for use by subclasses ...*/
    protected WriteLock(ReentrantReadWriteLock lock) { sync = lock.sync; }

    /** Acquires the write lock. ...*/
    public void lock() {
        sync.acquire( arg: 1);
    }

    /** Acquires the write lock unless the current thread is ...*/
    public void lockInterruptibly() throws InterruptedException {
        sync.acquireInterruptibly( arg: 1);
    }

    /** Acquires the write lock only if it is not held by another thread ...*/
    public boolean tryLock( ) {
        return sync.tryWriteLock();
    }

    /** Acquires the write lock if it is not held by another thread ...*/
    public boolean tryLock(long timeout, TimeUnit unit)
            throws InterruptedException {
        return sync.tryAcquireNanos( arg: 1, unit.toNanos(timeout));
    }

    /** Attempts to release this lock. ...*/
    public void unlock() {
        sync.release( arg: 1);
    }

    /** Returns a {@link Condition} instance for use with this ...*/
    public Condition newCondition() {
        return sync.newCondition();
    }

    /** Returns a string identifying this lock, as well as its lock ...*/
    public String toString() {...}

    /** Queries if this write lock is held by the current thread. ...*/
    public boolean isHeldByCurrentThread() {
        return sync.isHeldExclusively();
    }

    /** Queries the number of holds on this write lock by the current ...*/
    public int getHoldCount() {
        return sync.getWriteHoldCount();
    }
}
```

可以看到两个锁中的加锁操作都有一个关键的东西 Sync：



```
abstract static class Sync extends AbstractQueuedSynchronizer {
    private static final long serialVersionUID = 6317671515068378041L;

    /*...*/

    static final int SHARED_SHIFT   = 16;
    static final int SHARED_UNIT    = (1 << SHARED_SHIFT);
    static final int MAX_COUNT      = (1 << SHARED_SHIFT) - 1;
    static final int EXCLUSIVE_MASK = (1 << SHARED_SHIFT) - 1;

    /**...*/
    static int sharedCount(int c)    {...}
    /**...*/
    static int exclusiveCount(int c) {...}

    /** A counter for per-thread read hold counts. ...*/
    static final class HoldCounter {...}

    /** ThreadLocal subclass. Easiest to explicitly define for sake ...*/
    static final class ThreadLocalHoldCounter
        extends ThreadLocal<HoldCounter> {...}

    /** The number of reentrant read locks held by current thread. ...*/
    private transient ThreadLocalHoldCounter readHolds;

    /** The hold count of the last thread to successfully acquire ...*/
    private transient HoldCounter cachedHoldCounter;

    /** firstReader is the first thread to have acquired the read lock. ...*/
    private transient Thread firstReader = null;
    private transient int firstReaderHoldCount;

    Sync() {...}

    /*...*/

    /** Returns true if the current thread, when trying to acquire ...*/
    abstract boolean readerShouldBlock();

    /** Returns true if the current thread, when trying to acquire ...*/
    abstract boolean writerShouldBlock();

    /*...*/

    protected final boolean tryRelease(int releases) {...}

    protected final boolean tryAcquire(int acquires) {...}

    protected final boolean tryReleaseShared(int unused) {...}

    private IllegalMonitorStateException unmatchedUnlockException() {...}

    protected final int tryAcquireShared(int unused) {...}

    /** Full version of acquire for reads, that handles CAS misses ...*/
    final int fullTryAcquireShared(Thread current) {...}
```

Sync 是 ReentrantReadWriteLock 的一个抽象内部类，它继承了 AbstractQueuedSynchronizer 实现了共享与独享方式的同步操作。读写锁正好是一对共享&独占锁，而同步的队列也有共享和独占

分，那我们就从它们的加锁和解锁分别来看 AQS 的工作流程：

写锁（独占式）

加锁

```
        public void lock() {
            sync.acquire(1);
        }
```

这是 WriteLock 的加锁方法，可以看到加锁实际上是调用了 Sync 的 acquire(int arg) 方法，而这个法是在 AQS 中实现的，它使用 final 关键字来做修饰，在子类中不可重写。

```
public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
```

那 acquire(int arg) 做了什么呢，可以看到执行了四个方法：

- tryAcquire：尝试获取锁，获取成功则设置锁状态并返回 true ，否则返回 false 。需子类自行实现。
- addWaiter：将当前线程加入到 CLH 队列队尾。已有实现。
- acquireQueued：当前线程会根据公平性原则来进行阻塞等待,直到获取锁为止；并且返回当前线在等待过程中有没有中断过。已有实现。
- selfInterrupt：产生中断。已有实现。

前面我们说 AQS 由线程状态 state 和 线程等待队列组成，**AQS 加锁解锁的过程实际上就是对线程状的修改和等待队列的出入队列操作**，而 AQS 的子类可以通过重写 tryAcquire(int acquires) 方法来对tate 进行修改操作。于是就有 ReentrantReadWriteLock 中的 Sync 重写了 tryAcquire 方法：

```
 protected final boolean tryAcquire(int acquires) {
            // 获取当前线程
            Thread current = Thread.currentThread();
        // 拿到state变量，即锁的个数
            int c = getState();
        // 获得写锁的数量，前边说了低16位表示写锁个数
            int w = exclusiveCount(c);
        // 若该线程已经持有锁
            if (c != 0) {
                // (Note: if c != 0 and w == 0 then shared count != 0)
        // 如果写锁数量为0或者持有锁的线程不是当前线程，返回 false
                if (w == 0 || current != getExclusiveOwnerThread())
                    return false;
        // 如果写入锁数量大于最大数量（65535），跑出异常
                if (w + exclusiveCount(acquires) > MAX_COUNT)
                    throw new Error("Maximum lock count exceeded");
                // Reentrant acquire
                setState(c + acquires);
                return true;
            }
        // 如果写线程数为0，并且当前线程需要阻塞那么就返回失败；或者如果通过CAS增加写线程数
```

```
败也返回失败。
            if (writerShouldBlock() ||
                !compareAndSetState(c, c + acquires))
                return false;
        // 设置当前线程为锁的拥有者
            setExclusiveOwnerThread(current);
            return true;
        }
```

再来看下 addWaiter 方法：

```
private Node addWaiter(Node mode) {
        Node node = new Node(Thread.currentThread(), mode);
        // Try the fast path of enq; backup to full enq on failure
        Node pred = tail;
        if (pred != null) {
            node.prev = pred;
        // 通过 CAS 设置尾结点
            if (compareAndSetTail(pred, node)) {
                pred.next = node;
                return node;
            }
        }
    // 如果不成功则多次尝试
        enq(node);
        return node;
    }
```

enq(Node node) 方法如下：

```
private Node enq(final Node node) {
        for (;;) {
            Node t = tail;
            if (t == null) { // Must initialize
                if (compareAndSetHead(new Node()))
                    tail = head;
            } else {
                node.prev = t;
                if (compareAndSetTail(t, node)) {
                    t.next = node;
                    return t;
                }
            }
        }
    }
```

可以看到 enq 方法使用了死循环的方式一致尝试设置尾结点，直到成功。

如此入队操作就可以简单理解为：**tail 指向新节点、新节点的 prev 指向当前最后的节点，当前最后个节点的 next 指向当前节点。**

解锁

```
 public void unlock() {
```

```
        sync.release(1);
    }
```

解锁调用了 Sync 的 release 方法，下面看看这个方法都做了什么：

```
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
    // 唤醒节点
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

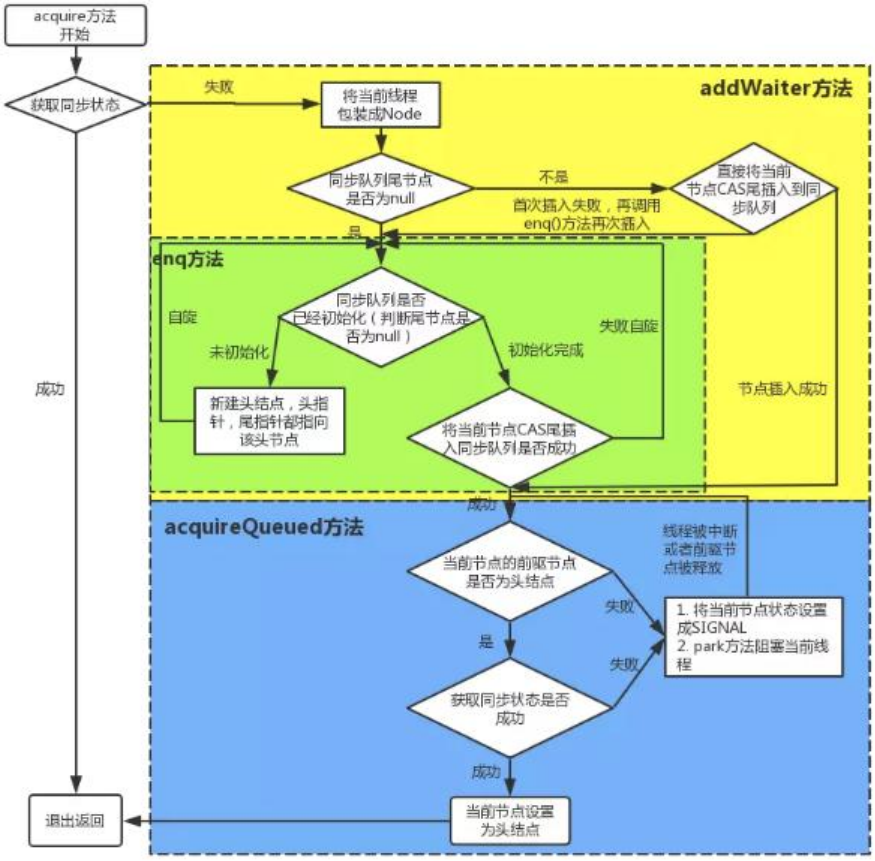
与加锁类似，tryRelease(arg) 由 AQS 的子类自行重写，

```
protected final boolean tryRelease(int releases) {
        if (!isHeldExclusively())
            throw new IllegalMonitorStateException();
        int nextc = getState() - releases;
        boolean free = exclusiveCount(nextc) == 0;
        if (free)
            setExclusiveOwnerThread(null);
        setState(nextc);
        return free;
    }
```

释放锁比较简单，**释放锁的实现是通过 CAS 修改 waitStatus 为 0 来实现的，然后通过 LockSupport.unpark(s.thread) 唤醒线程 **。

为了方便理解，在这里总结一下 WriteLock 的工作流程图：

 加锁操作



读锁 （共享式）

读锁的加锁操作与写锁类似：

```
public void lock() {
        sync.acquireShared(1);
    }
```

调用自定义的 tryAcquireShared(arg) 方法获取同步状态

```
public final void acquireShared(int arg) {
    if (tryAcquireShared(arg) < 0)
        doAcquireShared(arg);
}
```

如果获取失败，调用 doAcquireShared(int arg) 方法**自旋方式获取同步状态**：

```
private void doAcquireShared(int arg) {
    final Node node = addWaiter(Node.SHARED);
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();
            if (p == head) {
```

```
                int r = tryAcquireShared(arg);
                if (r >= 0) {
                    setHeadAndPropagate(node, r);
                    p.next = null; // help GC
                    if (interrupted)
                        selfInterrupt();
                    failed = false;
                    return;
                }
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

再看一下 tryAcquireShared(int unused) 方法：

```
protected final int tryAcquireShared(int unused) {

        Thread current = Thread.currentThread();
        int c = getState();
    // 如果其他线程已经获取了写锁，则当前线程获取读锁失败，进入等待状态
        if (exclusiveCount(c) != 0 &&
            getExclusiveOwnerThread() != current)
            return -1;
        int r = sharedCount(c);
        if (!readerShouldBlock() &&
            r < MAX_COUNT &&
            compareAndSetState(c, c + SHARED_UNIT)) {
            if (r == 0) {
                firstReader = current;
                firstReaderHoldCount = 1;
            } else if (firstReader == current) {
                firstReaderHoldCount++;
            } else {
                HoldCounter rh = cachedHoldCounter;
                if (rh == null || rh.tid != getThreadId(current))
                    cachedHoldCounter = rh = readHolds.get();
                else if (rh.count == 0)
                    readHolds.set(rh);
                rh.count++;
            }
            return 1;
        }
        return fullTryAcquireShared(current);
    }
```

在这个方法中可以知道，如果其他线程已经获取了写锁，则当前线程获取读锁失败，进入等待状态。果当前线程获取了写锁或者写锁未被获取，则当前线程增加读状态，成功获取读锁。读锁的每次释放

减少读状态，减少的值是“1<<16”。

六、结语

可能读完之后小伙伴并没有整明白，反而更懵逼了；或者根本没有读完。概念比较多，贴的代码也比多，可以配合其中几个比较重要的图去理解，几个比较关键的点：锁的分类思维导图、AQS、CAS、锁的加锁实现过程图。由于笔者水平所限，如上都是在学习的过程中总结、整理所得，仅供参考。