读写锁介绍

现实中有这样一种场景：对共享资源有读和写的操作，且写操作没有读操作那么频繁（读多写少）。在没有写操作的时候，多个线程同时读一个资源没有任何问题，所以应该允许多个线程同时读取共享资源（读读可以并发）；但是如果一个线程想去写这些共享资源，就不应该允许其他线程对该资源进行读和写操作了（读写，写读，写写互斥）。在读多于写的情况下，读写锁能够提供比排它锁更好的并发性和吞吐量。

针对这种场景，JAVA的并发包提供了读写锁ReentrantReadWriteLock，它内部，维护了一对相关的锁，一个用于只读操作，称为读锁；一个用于写入操作，称为写锁，描述如下：

线程进入读锁的前提条件：

- 没有其他线程的写锁

- 没有写请求或者有写请求，但调用线程和持有锁的线程是同一个。

线程进入写锁的前提条件：

- 没有其他线程的读锁
- 没有其他线程的写锁

而读写锁有以下三个重要的特性：

- 公平选择性：支持非公平（默认）和公平的锁获取方式，吞吐量还是非公平优于公平。
- 可重入：读锁和写锁都支持线程重入。以读写线程为例：读线程获取读锁后，能够再次获取读锁。写线程在获取写锁之后能够再次获取写锁，同时也可以获取读锁。
- 锁降级：遵循获取写锁、再获取读锁最后释放写锁的次序，写锁能够降级成为读锁。

ReentrantReadWriteLock的使用

读写锁接口ReadWriteLock

一对方法，分别获得读锁和写锁 Lock 对象。

```
public interface ReadWriteLock {
    /**
     * Returns the lock used for reading.
     *
     * @return the lock used for reading
     */
    @NotNull Lock readLock();

    /**
     * Returns the lock used for writing.
     *
     * @return the lock used for writing
     */
    @NotNull Lock writeLock();
}
```

ReentrantReadWriteLock类结构

ReentrantReadWriteLock是可重入的读写锁实现类。在它内部，维护了一对相关的锁，一个用于只读操作，另一个用于写入操作。只要没有 Writer 线程，读锁可以由多个 Reader 线程同时持有。也就是说，写锁是独占的，读锁是共享的。

```
/** Inner class providing readlock */
private final ReentrantReadWriteLock.ReadLock readerLock;
/** Inner class providing writelock */
private final ReentrantReadWriteLock.WriteLock writerLock;
/** Performs all synchronization mechanics */
final Sync sync;

/**...*/
public ReentrantReadWriteLock() {
    this(false);
}

/**...*/
public ReentrantReadWriteLock(boolean fair) {...}

public ReentrantReadWriteLock.WriteLock writeLock() { return writerLock; }
public ReentrantReadWriteLock.ReadLock  readLock()  { return readerLock; }
```

如何使用读写锁

```
1 private ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
2 private Lock r = readWriteLock.readLock();
3 private Lock w = readWriteLock.writeLock();
```

```
4
5  // 读操作上读锁
6  public Data get(String key) {
7    r.lock();
8    try {
9        // TODO 业务逻辑
10   }finally {
11       r.unlock();
12    }
13 }
14
15 // 写操作上写锁
16 public Data put(String key, Data value) {
17   w.lock();
18   try {
19       // TODO 业务逻辑
20   }finally {
21       w.unlock();
22    }
```

注意事项

- 读锁不支持条件变量
- 重入时升级不支持：持有读锁的情况下去获取写锁，会导致获取永久等待
- 重入时支持降级： 持有写锁的情况下可以去获取读锁

应用场景

ReentrantReadWriteLock适合读多写少的场景

示例Demo

```
1  public class Cache {
2      static Map<String, Object> map = new HashMap<String, Object>();
3      static ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
4      static Lock r = rwl.readLock();
5      static Lock w = rwl.writeLock();
6
7      // 获取一个key对应的value
8      public static final Object get(String key) {
9          r.lock();
10         try {
```

```
            return map.get(key);
        } finally {
            r.unlock();
        }
    }

    // 设置key对应的value，并返回旧的value
    public static final Object put(String key, Object value) {
        w.lock();
        try {
            return map.put(key, value);
        } finally {
            w.unlock();
        }
    }

    // 清空所有的内容
    public static final void clear() {
        w.lock();
        try {
            map.clear();
        } finally {
            w.unlock();
        }
    }
```

上述示例中，Cache组合一个非线程安全的HashMap作为缓存的实现，同时使用读写锁的读锁和写锁来保证Cache是线程安全的。在读操作get(String key)方法中，需要获取读锁，这使得并发访问该方法时不会被阻塞。写操作put(String key,Object value)方法和clear()方法，在更新 HashMap时必须提前获取写锁，当获取写锁后，其他线程对于读锁和写锁的获取均被阻塞，而 只有写锁被释放之后，其他读写操作才能继续。Cache使用读写锁提升读操作的并发性，也保证每次写操作对所有的读写操作的可见性，同时简化了编程方式

锁降级

锁降级指的是写锁降级成为读锁。如果当前线程拥有写锁，然后将其释放，最后再获取读锁，这种分段完成的过程不能称之为锁降级。锁降级是指把持住（当前拥有的）写锁，再获取到读锁，随后释放（先前拥有的）写锁的过程。锁降级可以帮助我们拿到当前线程修改后的结果而不被其他线程所破坏，防止更新丢失。

锁降级的使用示例

因为数据不常变化，所以多个线程可以并发地进行数据处理，当数据变更后，如果当前线程感知到数据变化，则进行数据的准备工作，同时其他处理线程被阻塞，直到当前线程完成数据的准备工作。

```
private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
private final Lock r = rwl.readLock();
private final Lock w = rwl.writeLock();
private volatile boolean update = false;

public void processData() {
    readLock.lock();
    if (!update) {
        // 必须先释放读锁
        readLock.unlock();
        // 锁降级从写锁获取到开始
        writeLock.lock();
        try {
            if (!update) {
                // TODO 准备数据的流程（略）
                update = true;
            }
            readLock.lock();
        } finally {
            writeLock.unlock();
        }
        // 锁降级完成，写锁降级为读锁
    }
    try {
        //TODO  使用数据的流程（略）
    } finally {
        readLock.unlock();
    }
}
```

锁降级中读锁的获取是否必要呢？答案是必要的。主要是为了保证数据的可见性，如果当前线程不获取读锁而是直接释放写锁，假设此刻另一个线程（记作线程T）获取了写锁并修改了数据，那么当前线程无法感知线程T的数据更新。如果当前线程获取读锁，即遵循锁降级的步骤，则线程T将会被阻塞，直到当前线程使用数据并释放读锁之后，线程T才能获取写锁进行数据更新。

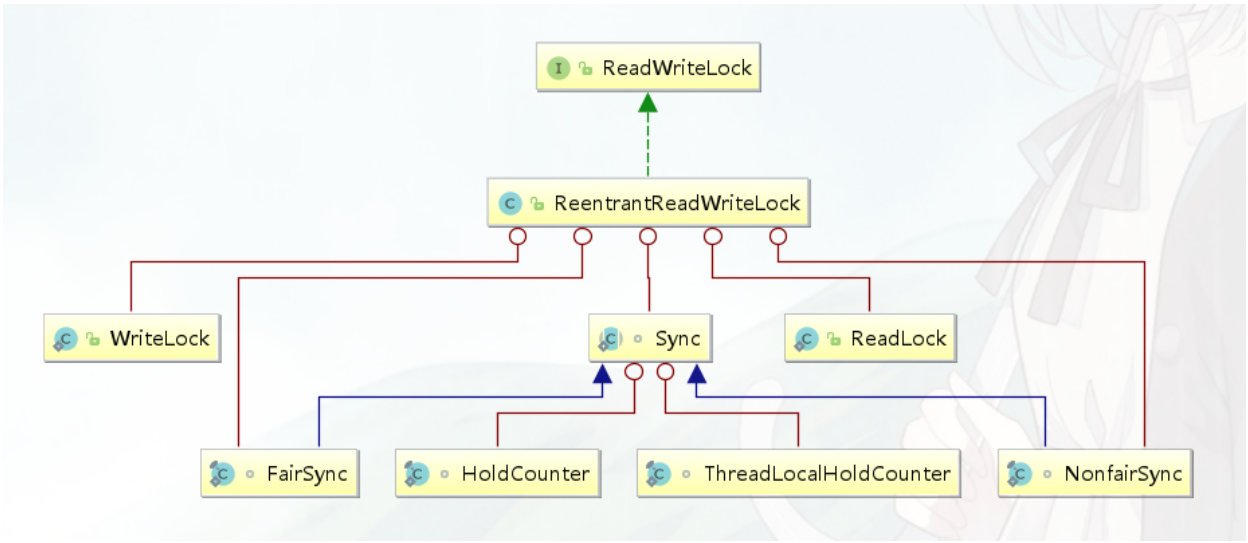
RentrantReadWriteLock不支持锁升级（把持读锁、获取写锁，最后释放读锁的过程）。目的也是保证数据可见性，如果读锁已被多个线程获取，其中任意线程成功获取了写锁并更新了数据，则其更新对其他获取到读锁的线程是不可见的。

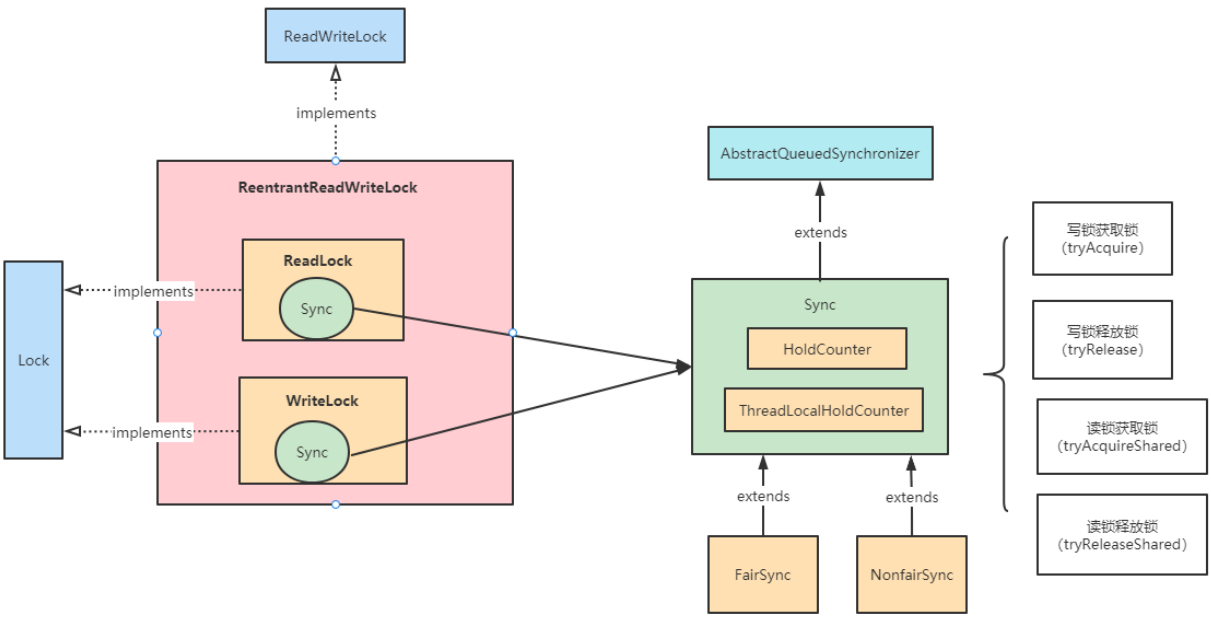
ReentrantReadWriteLock源码分析

思考：

1. 读写锁是怎样实现分别记录读写状态的？
2. 写锁是怎样获取和释放的？
3. 读锁是怎样获取和释放的？

ReentrantReadWriteLock结构





读写状态的设计

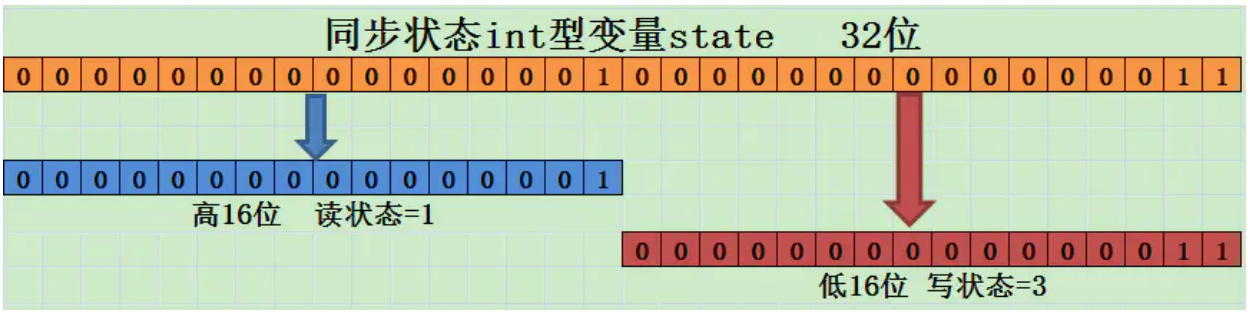
设计的精髓：用一个变量如何维护多种状态

在 ReentrantLock 中，使用 Sync (实际是 AQS)的 int 类型的 state 来表示同步状态，表示锁被一个线程重复获取的次数。但是，读写锁 ReentrantReadWriteLock 内部维护着一对读写锁，如果要用一个变量维护多种状态，需要采用“按位切割使用”的方式来维护这个变量，将其切分为两部分：高16为表示读，低16为表示写。

分割之后，读写锁是如何迅速确定读锁和写锁的状态呢？通过位运算。假如当前同步状态为S，那么：

- 写状态，等于 S & 0x0000FFFF（将高 16 位全部抹去）。 当写状态加1，等于S+1.
- 读状态，等于 S >>> 16 (无符号补 0 右移 16 位)。当读状态加1，等于S+（1<<16）,也就是S+0x00010000

根据状态的划分能得出一个推论：S不等于0时，当写状态（S&0x0000FFFF）等于0时，则读状态（S>>>16）大于0，即读锁已被获取。



代码实现：java.util.concurrent.locks.ReentrantReadWriteLock.Sync

```
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;

/** Returns the number of shared holds represented in count  */
static int sharedCount(int c)    { return c >>> SHARED_SHIFT; }
/** Returns the number of exclusive holds represented in count  */
static int exclusiveCount(int c) { return c & EXCLUSIVE_MASK; }
```

- exclusiveCount(int c) 静态方法，获得持有写状态的锁的次数。
- sharedCount(int c) 静态方法，获得持有读状态的锁的线程数量。不同于写锁，读锁可以同时被多个线程持有。而每个线程持有的读锁支持重入的特性，所以需要对每个线程持有的读锁的数量单独计数，这就需要用到 HoldCounter 计数器

HoldCounter 计数器

读锁的内在机制其实就是一个共享锁。一次共享锁的操作就相当于对HoldCounter 计数器的操作。获取共享锁，则该计数器 + 1，释放共享锁，该计数器 - 1。只有当线程获取共享锁后才能对共享锁进行释放、重入操作。

```
/**...*/
static final class HoldCounter {
    int count = 0;
    // Use id, not reference, to avoid garbage retention
    final long tid = getThreadId(Thread.currentThread());
}

/**...*/
static final class ThreadLocalHoldCounter
    extends ThreadLocal<HoldCounter> {
    public HoldCounter initialValue() {
        return new HoldCounter();
    }
}
```

通过 ThreadLocalHoldCounter 类，HoldCounter 与线程进行绑定。HoldCounter 是绑定线程的一个计数器，而 ThreadLocalHoldCounter 则是线程绑定的 ThreadLocal。

- HoldCounter是用来记录读锁重入数的对象
- ThreadLocalHoldCounter是ThreadLocal变量，用来存放不是第一个获取读锁的线程的其他线程的读锁重入数对象

写锁的获取

写锁是一个支持重进入的排它锁。如果当前线程已经获取了写锁，则增加写状态。如果当前线程在获取写锁时，读锁已经被获取（读状态不为0）或者该线程不是已经获取写锁的线程， 则当前线程进入等待状态。

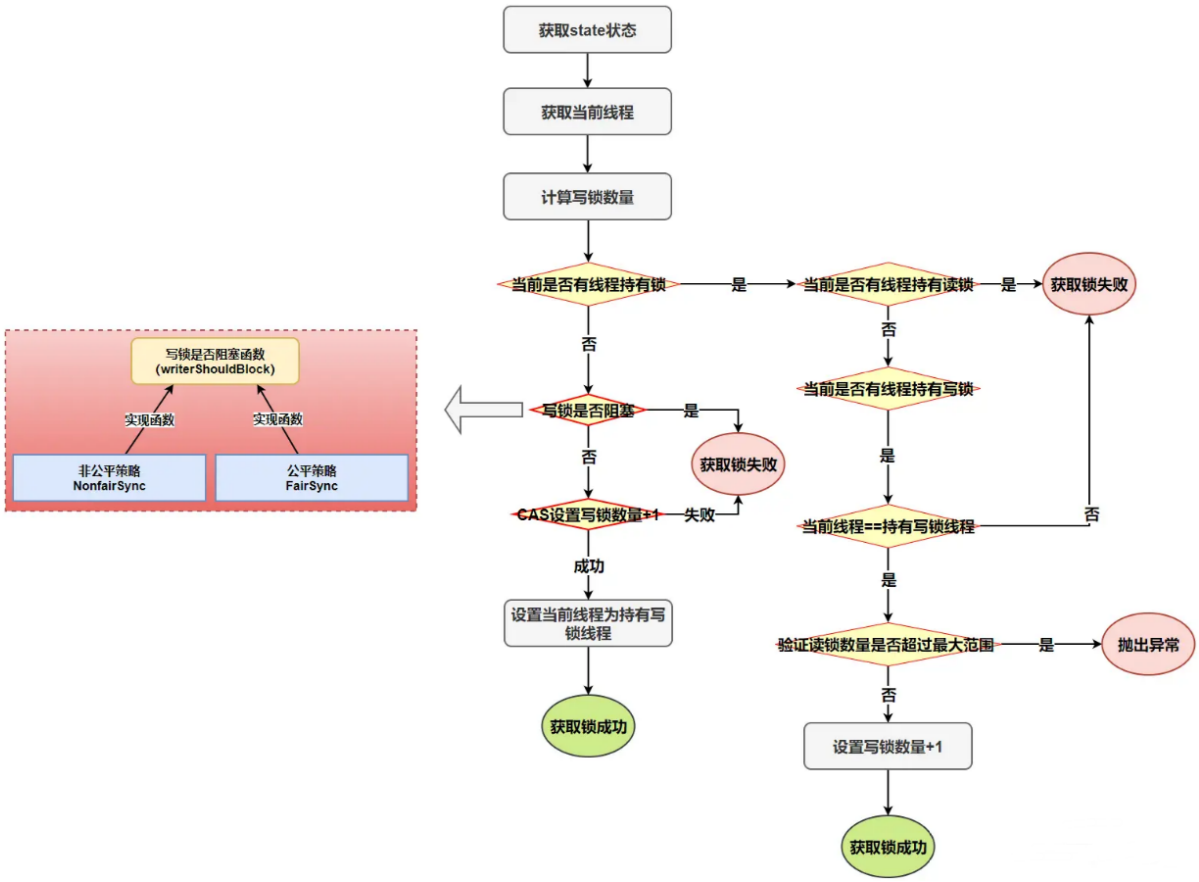
写锁的获取是通过重写AQS中的tryAcquire方法实现的。

```
protected final boolean tryAcquire(int acquires) {
    //当前线程
    Thread current = Thread.currentThread();
    //获取state状态    存在读锁或者写锁，状态就不为0
    int c = getState();
    //获取写锁的重入数
    int w = exclusiveCount(c);
    //当前同步状态state != 0，说明已经有其他线程获取了读锁或写锁
    if (c != 0) {
        // c!=0 && w==0 表示存在读锁
        // 当前存在读锁或者写锁已经被其他写线程获取，则写锁获取失败
        if (w == 0 || current != getExclusiveOwnerThread())
            return false;
        // 超出最大范围  65535
        if (w + exclusiveCount(acquires) > MAX_COUNT)
            throw new Error("Maximum lock count exceeded");
        //同步state状态
        setState(c + acquires);
```

```
19          return true;
20      }
21      // writerShouldBlock有公平与非公平的实现，非公平返回false，会尝试通过cas加锁
22      //c==0 写锁未被任何线程获取，当前线程是否阻塞或者cas尝试获取锁
23      if (writerShouldBlock() ||
24          !compareAndSetState(c, c + acquires))
25          return false;
26
27      //设置写锁为当前线程所有
28      setExclusiveOwnerThread(current);
29      return true;
```

通过源码我们可以知道：

- 读写互斥
- 写写互斥
- 写锁支持同一个线程重入
- writerShouldBlock写锁是否阻塞实现取决公平与非公平的策略（FairSync和NonfairSync）

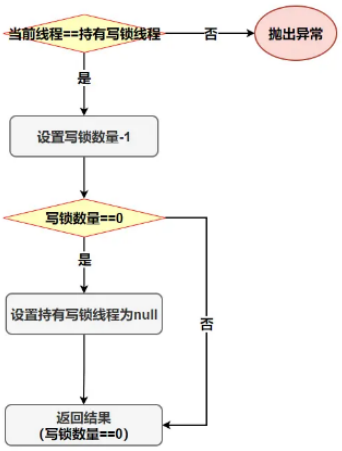


思考：有线程读的过程中不允许写，这种设计有什么问题？

写锁的释放

写锁释放通过重写AQS的tryRelease方法实现

```java
protected final boolean tryRelease(int releases) {
    //若锁的持有者不是当前线程，抛出异常
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
    int nextc = getState() - releases;
    //当前写状态是否为0，为0则释放写锁
    boolean free = exclusiveCount(nextc) == 0;
    if (free)
        setExclusiveOwnerThread(null);
    setState(nextc);
    return free;
```



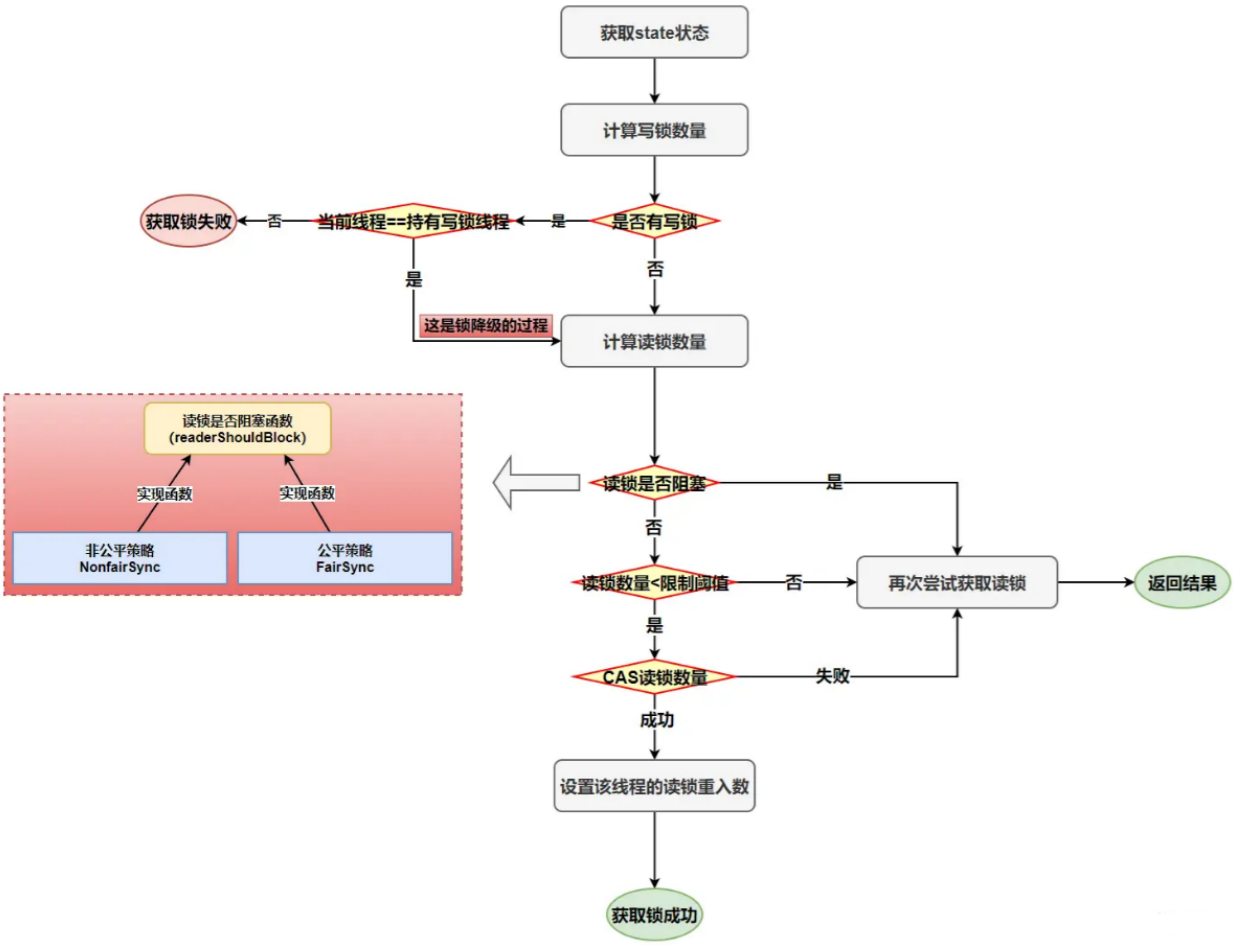
读锁的获取

实现共享式同步组件的同步语义需要通过重写AQS的tryAcquireShared方法和tryReleaseShared方法。读锁的获取实现方法为：

```java
protected final int tryAcquireShared(int unused) {
    Thread current = Thread.currentThread();
    int c = getState();
    // 如果写锁已经被获取并且获取写锁的线程不是当前线程，当前线程获取读锁失败返回-1    判断锁降级
    if (exclusiveCount(c) != 0 &&
        getExclusiveOwnerThread() != current)
        return -1;
    //计算出读锁的数量
    int r = sharedCount(c);
    /**
    * 读锁是否阻塞     readerShouldBlock()公平与非公平的实现
    * r < MAX_COUNT:  持有读锁的线程小于最大数（65535）
```

```
13      *  compareAndSetState(c, c + SHARED_UNIT) cas设置获取读锁线程的数量
14      */
15      if (!readerShouldBlock() &&
16          r < MAX_COUNT &&
17          compareAndSetState(c, c + SHARED_UNIT)) {    //当前线程获取读锁
18
19          if (r == 0) {  //设置第一个获取读锁的线程
20              firstReader = current;
21              firstReaderHoldCount = 1;  //设置第一个获取读锁线程的重入数
22          } else if (firstReader == current) { // 表示第一个获取读锁的线程重入
23              firstReaderHoldCount++;
24          } else { // 非第一个获取读锁的线程
25              HoldCounter rh = cachedHoldCounter;
26              if (rh == null || rh.tid != getThreadId(current))
27                  cachedHoldCounter = rh = readHolds.get();
28              else if (rh.count == 0)
29                  readHolds.set(rh);
30              rh.count++;  //记录其他获取读锁的线程的重入次数
31          }
32          return 1;
33      }
34      // 尝试通过自旋的方式获取读锁,实现了重入逻辑
35      return fullTryAcquireShared(current);
```

- 读锁共享，读读不互斥
- 读锁可重入，每个获取读锁的线程都会记录对应的重入数
- 读写互斥，锁降级场景除外
- 支持锁降级，持有写锁的线程，可以获取读锁，但是后续要记得把读锁和写锁读释放
- readerShouldBlock读锁是否阻塞实现取决公平与非公平的策略（FairSync和NonfairSync）



读锁的释放

获取到读锁，执行完临界区后，要记得释放读锁（如果重入多次要释放对应的次数），不然会阻塞其他线程的写操作。

读锁释放的实现主要通过方法tryReleaseShared：

```
protected final boolean tryReleaseShared(int unused) {
    Thread current = Thread.currentThread();
    //如果当前线程是第一个获取读锁的线程
    if (firstReader == current) {
        // assert firstReaderHoldCount > 0;
        if (firstReaderHoldCount == 1)
            firstReader = null;
        else
            firstReaderHoldCount--; //重入次数减1
    } else {  //不是第一个获取读锁的线程
        HoldCounter rh = cachedHoldCounter;
        if (rh == null || rh.tid != getThreadId(current))
            rh = readHolds.get();
        int count = rh.count;
        if (count <= 1) {
```

```
            readHolds.remove();
            if (count <= 0)
                throw unmatchedUnlockException();
        }
        --rh.count;  //重入次数减1
    }
    for (;;) {  //cas更新同步状态
        int c = getState();
        int nextc = c - SHARED_UNIT;
        if (compareAndSetState(c, nextc))
            // Releasing the read lock has no effect on readers,
            // but it may allow waiting writers to proceed if
            // both read and write locks are now free.
            return nextc == 0;
    }
```