J.U.C

一、AQS原理

1.1 概述

全称AbstractQueuedSynchronizer，是阻塞式锁（和synchronized相同）和相关的同步器工具的框架

特点：

- 用state属性来表示资源的状态（分独占模式和共享模式），子类需要定义如何维护这个状态，控制如何获取锁和释放锁
  - 独占模式是只有一个线程能够访问资源，而共享模式可以允许多个线程访问资源
  - getState - 获取state状态
  - setState - 设置state状态
  - compareAndSetStatus - 乐观锁机制设置 state 状态
- 提供了基于FIFO的等待队列
- 条件变量来实现等待、唤醒机制，支持多个条件变量

子类主要实现这样一些方法（默认抛出 UnsupportedOperationException）

- **tryAcquire**
- **tryRelease**
- **tryAcquireShared**
- **tryReleaseShared**
- **isHeldExclusively**

```java
//获取锁的姿势
// 如果获取锁失败
if (!tryAcquire(arg)) {
 // 入队，可以选择阻塞当前线程 通过park unpark来阻塞或者恢复当前进程
}

//释放锁的姿势
// 如果释放锁成功
if (tryRelease(arg)) {
 // 让阻塞线程恢复运行
}
```

下面实现一个不可重入的阻塞式锁：使用 `AbstractQueuedSynchronizer` 自定义一个同步器来实现自定义锁！

```java
@Slf4j(topic = "guizy.TestAQS")
@SuppressWarnings("all")
public class TestAqs {
    public static void main(String[] args) {
        MyLock lock = new MyLock();
        new Thread(() -> {
            lock.lock();
            log.debug("locking...");
            // 不可重入锁，同一线程在锁释放前，只能加一次锁
//            lock.lock();
//            log.debug("locking...");
            try {
                log.debug("locking...");
                Sleeper.sleep(1);
            } finally {
                log.debug("unlocking...");
                lock.unlock();
            }
        }, "t1").start();

        new Thread(() -> {
            lock.lock();
            try {
                log.debug("locking...");
            } finally {
                log.debug("unlocking...");
                lock.unlock();
            }
```

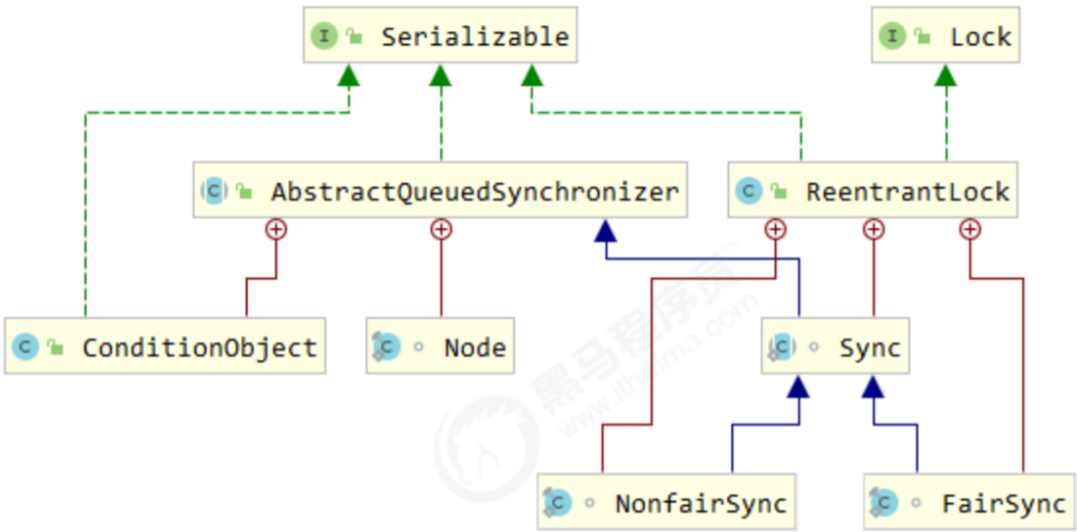
```java
        }, "t2").start();
    }
}

// 自定义锁（不可重入锁）
class MyLock implements Lock {

    // 独占锁  同步器类
    class MySync extends AbstractQueuedSynchronizer {
        @Override
        protected boolean tryAcquire(int arg) {
            // 确保原子性
            if (compareAndSetState(0, 1)) {
                // 加上了锁，并设置 owner 为当前线程
                setExclusiveOwnerThread(Thread.currentThread());
                return true;
            }
            return false;
        }

        @Override
        protected boolean tryRelease(int arg) {
            // 这里不需要确定原子性，因为是是独占锁，由持锁者进行释放
            // 在setState(0)上面设置Owner为null，防止指令重排序带来的问
题
            setExclusiveOwnerThread(null);
            setState(0); // state是volatile修饰的，在setState(0)前面
的属性修改，对于其他线程也是可见的，具体见volatile原理(写屏障)
            return true;
        }

        @Override // 是否持有独占锁
        protected boolean isHeldExclusively() {
            return getState() == 1;
        }

        public Condition newCondition() {
            return new ConditionObject();
        }
    }

    private MySync sync = new MySync();

    @Override // 加锁（不成功会进入等待队列）
    public void lock() {
```

```java
        sync.acquire(1);
    }

    @Override // 加锁，可打断
    public void lockInterruptibly() throws InterruptedException {
        sync.acquireInterruptibly(1);
    }

    @Override // 尝试加锁（一次）
    public boolean tryLock() {
        return sync.tryAcquire(1);
    }

    @Override // 尝试加锁，带超时
    public boolean tryLock(long time, TimeUnit unit) throws
InterruptedException {
        return sync.tryAcquireNanos(1, unit.toNanos(time));
    }

    @Override // 解锁
    public void unlock() {
        sync.release(1);
    }

    @Override // 创建条件变量
    public Condition newCondition() {
        return sync.newCondition();
    }
}
```

二、ReentrantLock

2.1 概述

ReentrantLock提供了两个同步器，实现 公平锁 和 非公平锁 ，默认是非公平锁！



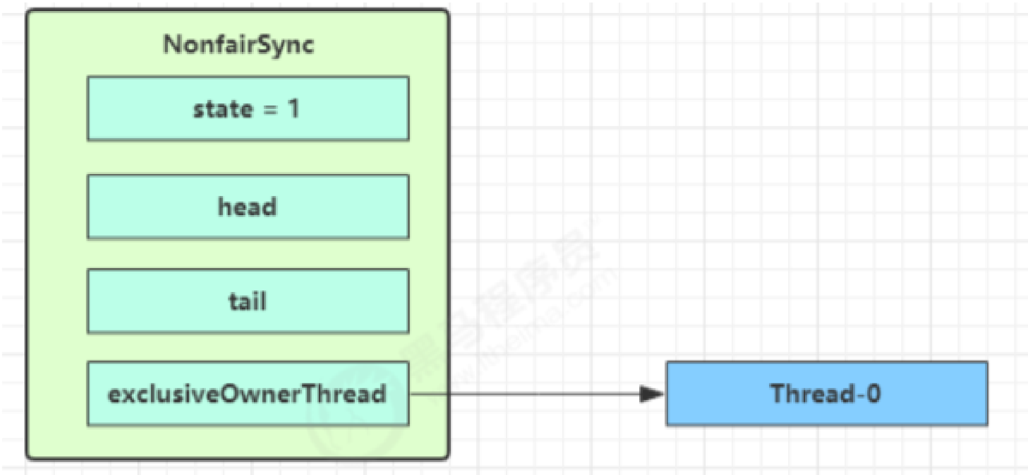
2.2 非公平锁实现原理

- 加锁，解锁 流程，先从构造器开始看，默认为非公平锁实现

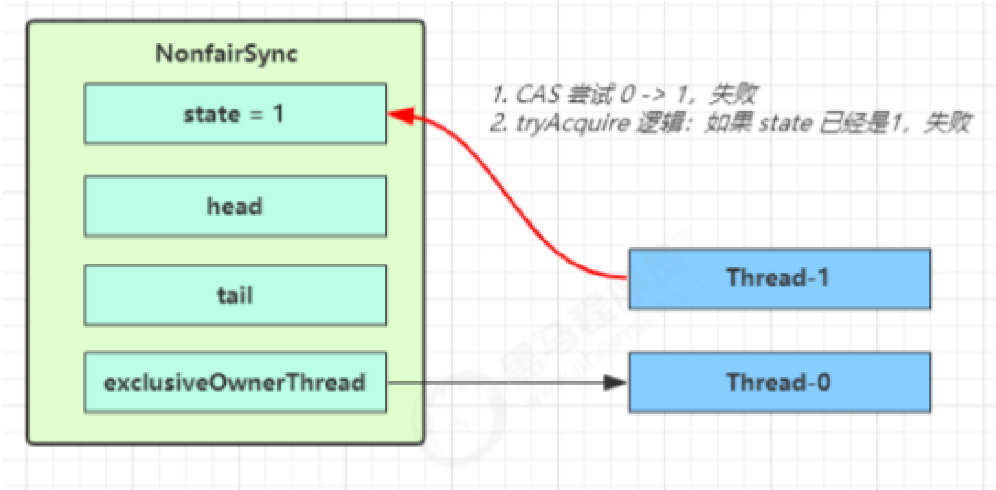
```
public ReentrantLock() {
    sync = new NonfairSync();
}
```

没有竞争时

- Thread-0成为锁的持有者



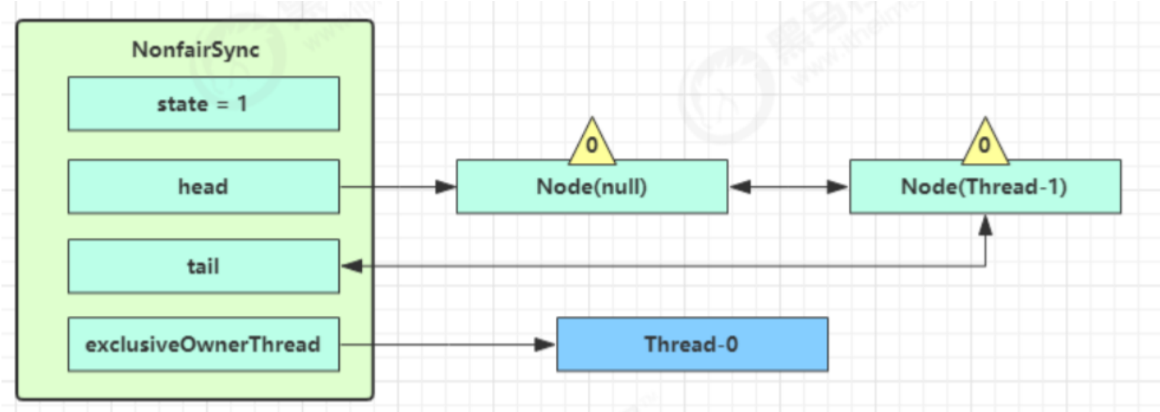
第一个线程Thread-0竞争出现时，查看源码的 `NonfairSync` 的 `lock` 方法，Thread-0将state改成1，并且设置自己独占锁




```java
abstract void lock();

// 非公平锁的lock
final void lock() {
    if (compareAndSetState(0, 1))
        setExclusiveOwnerThread(Thread.currentThread());
    else
        acquire(1);
}

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

当第二个进程Thread-1 进行加锁流程时，会发生失败

- lock方法中CAS 尝试将 state 由 0 改为 1，结果失败 (因为此时CAS操作, 已经state已经为1了)
- lock方法中进一步调用 `acquire` 方法，进入 `tryAcquire` 逻辑，这里我们认为这时 state 已经是1，结果仍然失败
- 接下来进入 acquire方法的addWaiter逻辑，构造Node 队列 (双向链表实现)
    - 下图中 黄色三角 表示该 Node 的 `waitStatus` 状态，其中 0 为默认正常状态
    - Node 的创建是懒惰的
    - **其中第一个 Node 称为 Dummy（哑元）或哨兵，用来占位，并不关联线程**



当前线程进入 acquire方法的 `acquireQueued` 逻辑

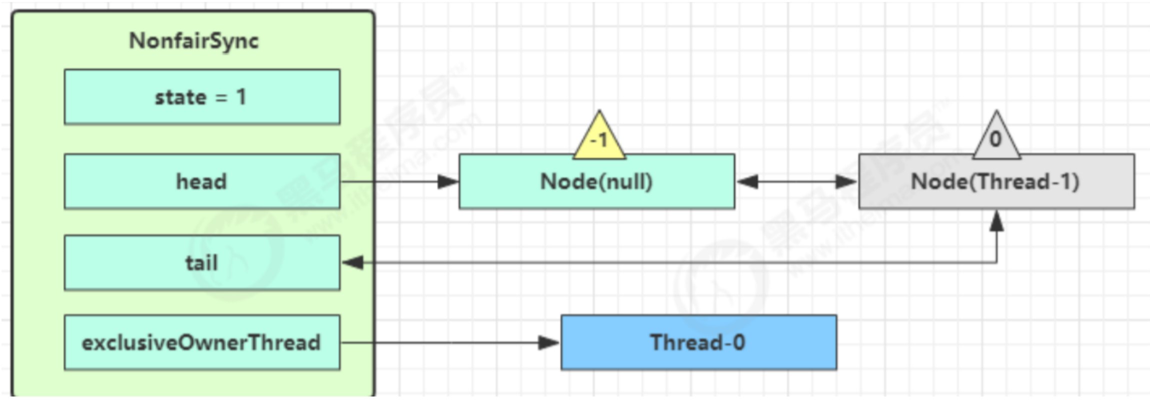
1. acquireQueued 会在一个死循环中不断尝试获得锁，失败后进入 `park` 阻塞
2. 如果自己是紧邻着 head（排第二位），那么再次 tryAcquire 尝试获取锁，我们这里设置这时 state 仍为 1，失败
3. 进入 `shouldParkAfterFailedAcquire` 逻辑，将 `前驱 node` ，即 head 的 `waitStatus` 改为 `-1` （waitStatus value to indicate successor's thread needs unparking），这次返回 false

```
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor(); // 寻找前驱结点
            if (p == head && tryAcquire(arg)) { //如果紧邻head，可以再次通过tryAcquire获得锁
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return interrupted;
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

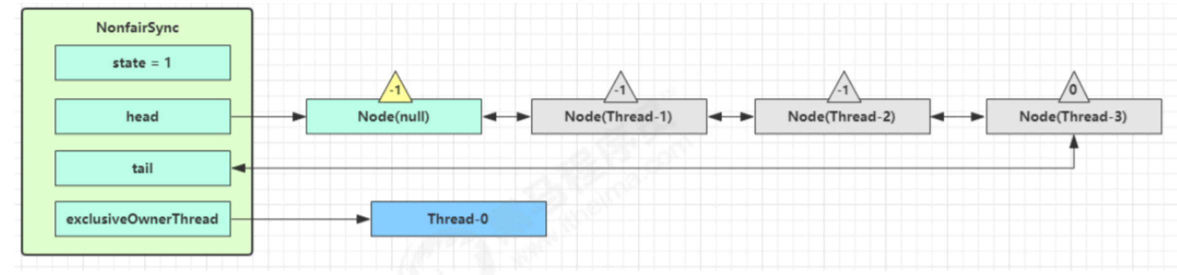
4. shouldParkAfterFailedAcquire 执行完毕回到 `acquireQueued` ，再次 tryAcquire 尝试获取锁，当然这时 state 仍为 1，失败

5. 当再次进入 shouldParkAfterFailedAcquire 时，这时因为其前驱 node 的 waitStatus 已经是 -1，这次返回 true
6. 进入 `parkAndCheckInterrupt` ， Thread-1 被 `park` 挂起（灰色表示已经阻塞）

```
private final boolean parkAndCheckInterrupt() {
    LockSupport.park(this);
    return Thread.interrupted();
}
```



后续有多个线程经历上述过程竞争失败，变成这个样子



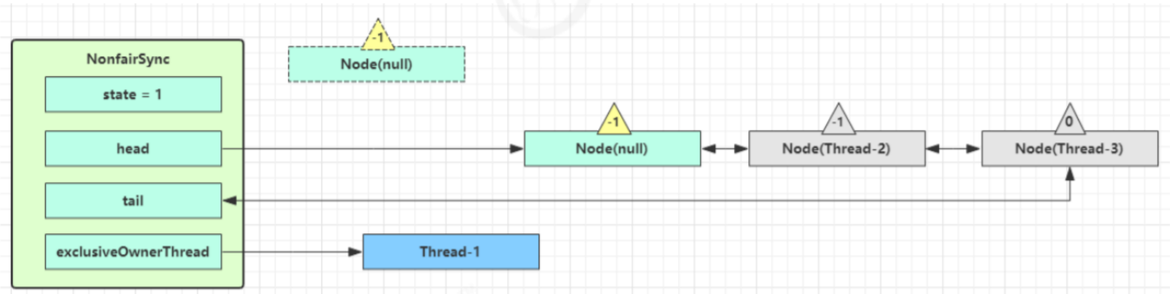
Thread-0 调用 `unlock方法` 里的 `release方法` 释放锁，进入 `tryRelease` 流程，如果成功，

- 设置 exclusiveOwnerThread 为 null
- 设置 state 为 0

```
public void unlock() {
    sync.release(1);
}
```

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

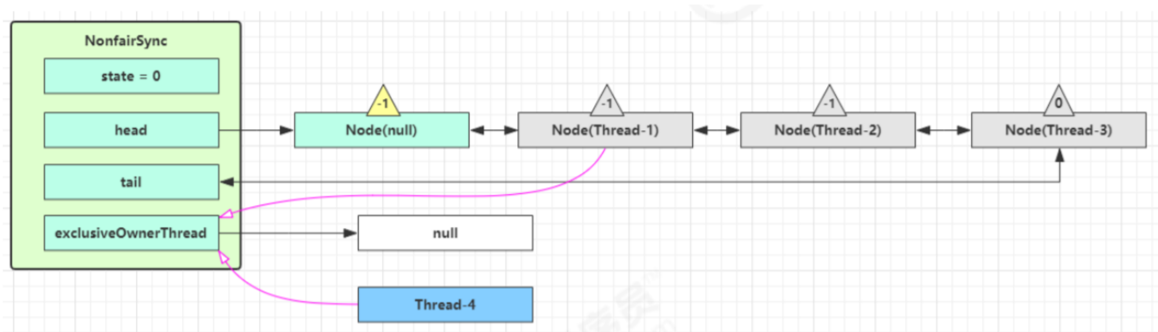
unlock方法里的release方法方法中，如果当前队列不为 null，并且 head 的 waitStatus = -1，进入 unparkSuccessor 流程： **unparkSuccessor中会找到队列中离 head 最近的一个 Node（没取消的）即后继节点，unpark 唤醒Thread-1 恢复其运行**，本例中即为 Thread-1 回到 Thread-1 阻塞的位置继续执行, 会继续执行 acquireQueued 流程



如果加锁成功（没有竞争），会设置 （acquireQueued 方法中）

1. exclusiveOwnerThread 为 Thread-1，state = 1
2. head 指向刚刚 Thread-1 所在的 Node，该 Node 清空 Thread
3. 原本的 head 因为从链表断开，而可被垃圾回收

如果这时候有其它线程来竞争（非公平的体现），例如这时有 Thread-4 来了



如果不巧又被 Thread-4 占了先

1. Thread-4 被设置为 exclusiveOwnerThread，state = 1
2. Thread-1 再次进入 acquireQueued 流程，获取锁失败，重新进入 park 阻塞

2.3 可重入原理

所谓重入锁，指的是以线程为单位，当一个线程获取对象锁之后，这个线程可以再次获取本对象上的锁，而其他的线程是不可以的

同一个线程, 锁重入, 会对 `state` 进行自增. 释放锁的时候, state进行自减; 当state自减为0的时候. 此时线程才会将锁释放成功, 才会进一步去唤醒其他线程来竞争锁

```
final boolean nonfairTryAcquire(int acquires) {
    // 获取当前进程
    final Thread current = Thread.currentThread();
    int c = getState();
    // 当前进程状态是0（未获得锁），会将其设置为acquires
    if (c == 0) {
        if (compareAndSetState(0, acquires)) {
            // 设置状态成功，会将owner进程设置为当前进程
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    // 如果当前进程已经获得锁，线程还是当前线程，表示发生锁重入，会将当前状态
在加上acquires
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}

protected final boolean tryRelease(int releases) {
    int c = getState() - releases;
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    // 只有state为0，才释放成功
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);
```

```java
    return free;
}
```

2.4 可打断原理

2.4.1 不可打断原理（默认模式）

在此模式下，即使它被打断（只是将打断标记被设置为true），仍然会驻留在AQS队列中，**等获得锁后方能继续运行**

AbstractQueuedSynchronizer.java

```java
private final boolean parkAndCheckInterrupt() {
    LockSupport.park(this);
    return Thread.interrupted();
}

final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return interrupted;
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

2.4.2 可打断模式

```
static final class NonfairSync extends Sync {
    public final void acquireInterruptibly(int arg) throws
InterruptedException {
        if (Thread.interrupted())
            throw new InterruptedException();
        // 如果没有获得到锁, 进入 ㈠
        if (!tryAcquire(arg))
            doAcquireInterruptibly(arg);
    }

    // ㈠ 可打断的获取锁流程
    private void doAcquireInterruptibly(int arg) throws
InterruptedException {
        final Node node = addWaiter(Node.EXCLUSIVE);
        boolean failed = true;
        try {
            for (;;) {
                final Node p = node.predecessor();
                if (p == head && tryAcquire(arg)) {
                    setHead(node);
                    p.next = null; // help GC
                    failed = false;
                    return;
                }
                if (shouldParkAfterFailedAcquire(p, node) &&
                        parkAndCheckInterrupt()) {
                    // 在 park 过程中如果被 interrupt, 这时候抛出异常,
而不会再次进入 for (;;)
                    throw new InterruptedException();
                }
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }
}
```

2.5 公平锁实现原理

非公平锁下，线程竞争锁的时候不回去查看AQS队列，直接竞争锁

公平锁下，线程会查看AQS队列中, 自己有没有前驱节点， 以及该节点不是占位的哨兵节点； 如果有就不去竞争锁. 如果没有, 才会去CAS操作

- if (!hasQueuedPredecessors() &&
  compareAndSetState(0, acquires)) {
  setExclusiveOwnerThread(current);
  return true;
  }

```
static final class FairSync extends Sync {
    private static final long serialVersionUID =
-3000897897090466540L;
    final void lock() {
        acquire(1);
    }

    // AQS 继承过来的方法, 方便阅读, 放在此处
    public final void acquire(int arg) {
        if (
                !tryAcquire(arg) &&
                        acquireQueued(addWaiter(Node.EXCLUSIVE),
arg)
        ) {
            selfInterrupt();
        }
    }
    // 与非公平锁主要区别在于 tryAcquire 方法的实现
    protected final boolean tryAcquire(int acquires) {
        final Thread current = Thread.currentThread();
        int c = getState();
        if (c == 0) {
            // 先检查 AQS 队列中是否有前驱节点, 没有才去竞争
            if (!hasQueuedPredecessors() &&
                    compareAndSetState(0, acquires)) {
                setExclusiveOwnerThread(current);
                return true;
            }
        }
        else if (current == getExclusiveOwnerThread()) {
            int nextc = c + acquires;
            if (nextc < 0)
```

```
                throw new Error("Maximum lock count exceeded");
            setState(nextc);
            return true;
        }
        return false;
    }

    // ㈠ AQS 继承过来的方法, 方便阅读, 放在此处
    public final boolean hasQueuedPredecessors() {
        Node t = tail;
        Node h = head;
        Node s;
        // h != t 时表示队列中有 Node
        return h != t &&
                (
                        // (s = h.next) == null 表示队列中还有没有老
二
                        (s = h.next) == null || // 或者队列中老二线
程不是此线程
                                s.thread !=
Thread.currentThread()
                );
    }
}
```

2.6 条件变量实现原理

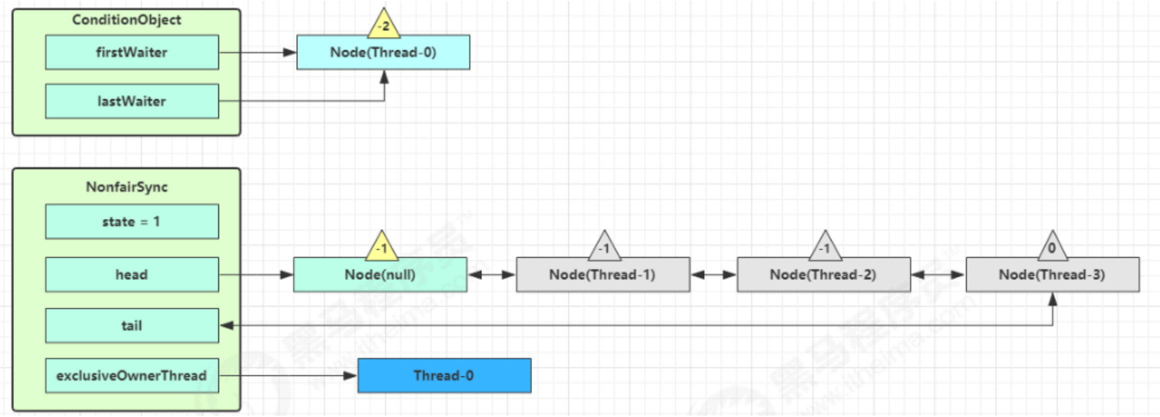
每个条件变量其实就对应着一个等待队列，其实现类是 `ConditionObject`

2.6.1 await流程

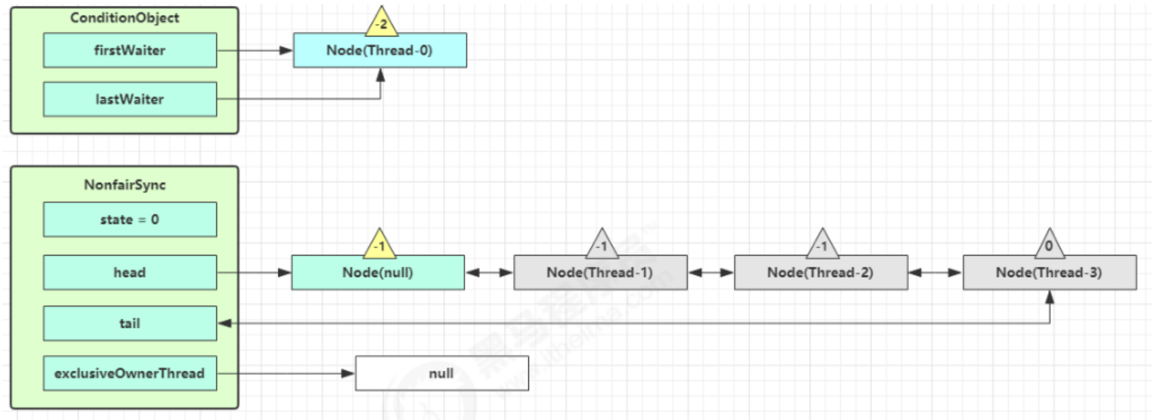
- 开始 Thread-0 持有锁，conditionObject对象调用 `await`，进入 ConditionObject 的 `addConditionWaiter` 流程（将线程加入 ConditionWaiter）创建新的 Node 状态为 -2（Node.CONDITION，waitStatus value to indicate thread is waiting on condition），关联 Thread-0，加入等待队列尾部

```
public final void await() throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    Node node = addConditionWaiter();
    int savedState = fullyRelease(node);
    int interruptMode = 0;
    while (!isOnSyncQueue(node)) {
```

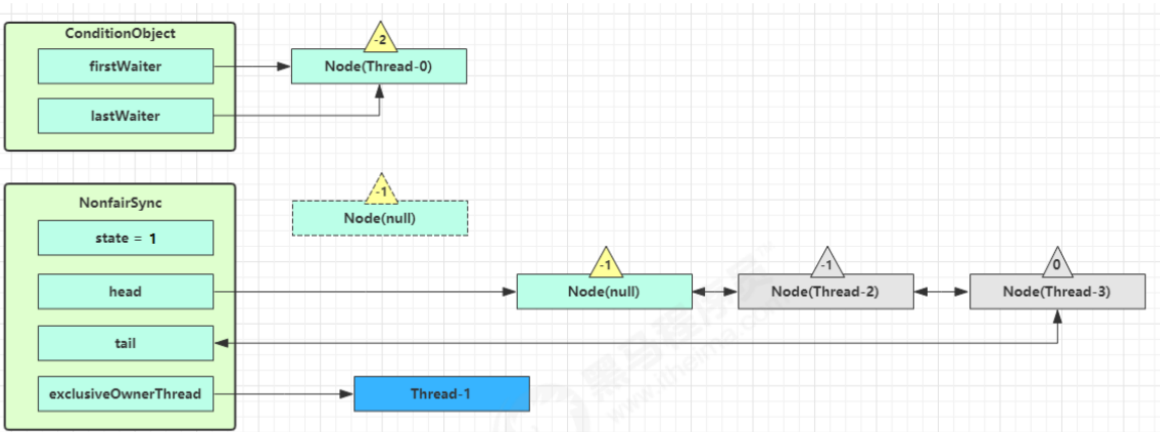
```
        LockSupport.park(this);
        if ((interruptMode = checkInterruptWhileWaiting(node)) !=
0)
            break;
    }
    if (acquireQueued(node, savedState) && interruptMode !=
THROW_IE)
        interruptMode = REINTERRUPT;
    if (node.nextWaiter != null) // clean up if cancelled
        unlinkCancelledWaiters();
    if (interruptMode != 0)
        reportInterruptAfterWait(interruptMode);
}

final int fullyRelease(Node node) {
    boolean failed = true;
    try {
        int savedState = getState();
        if (release(savedState)) {
            failed = false;
            return savedState;
        } else {
            throw new IllegalMonitorStateException();
        }
    } finally {
        if (failed)
            // Node.CANCELLED,waitStatus value to indicate thread
has cancelled
            node.waitStatus = Node.CANCELLED;
    }
}

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



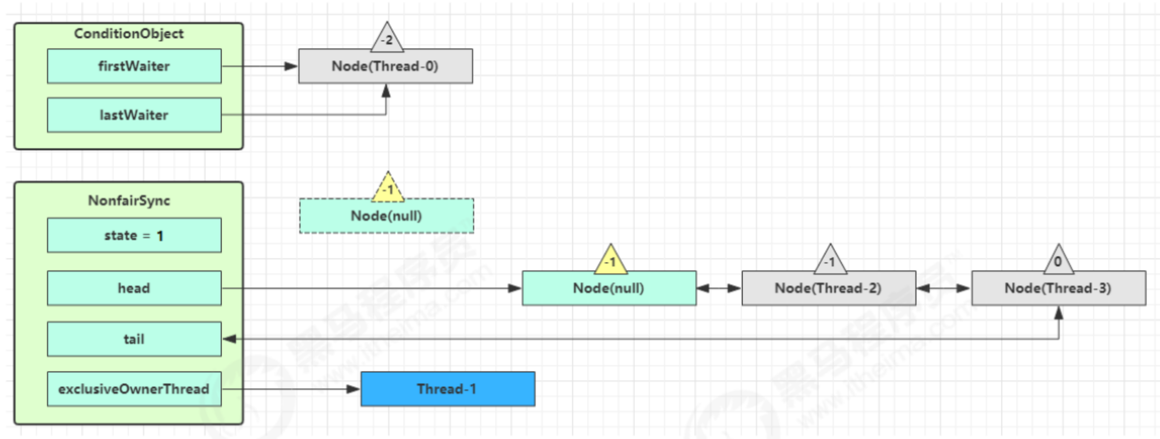
接下来进入 AQS 的 `fullyRelease` 流程，释放同步器上的所有的锁 (因为可能线程发生可重入, 锁有很多层)



`unparkSuccessor(h);` —> unpark唤醒 AQS 队列中的下一个节点，竞争锁，假设没有其他竞争线程，那么 `Thread-1` 竞争成功



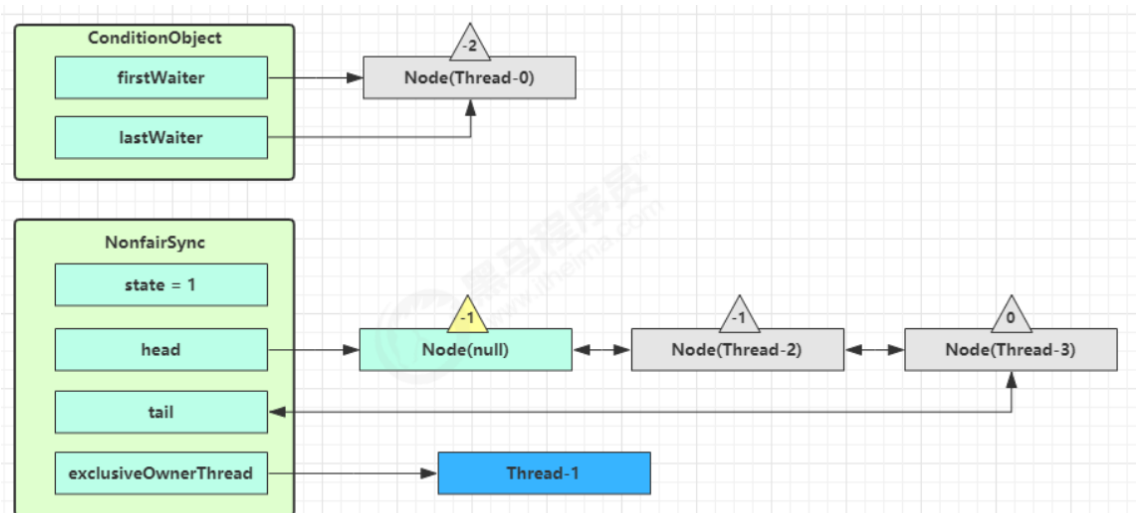
`LockSupport.park(this)` —> park 阻塞 Thread-0



2.6.2 signal 流程

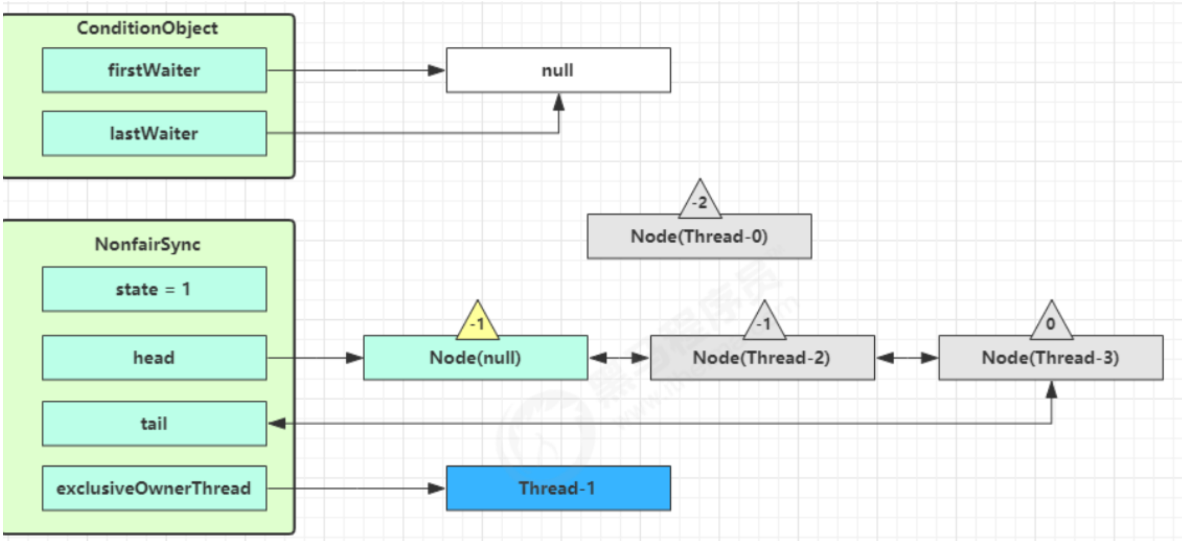
假设 Thread-1 要来唤醒 Thread-0，此时Thread-0的条件满足，需要加入AQS队列竞争锁

```
// 如果没有持有锁，会抛出异常 --> 这里表示Thread-1要持有锁，
//才可以去条件变量中去唤醒等待的线程
public final void signal() {
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
    Node first = firstWaiter;
    if (first != null)
        doSignal(first);
}
```

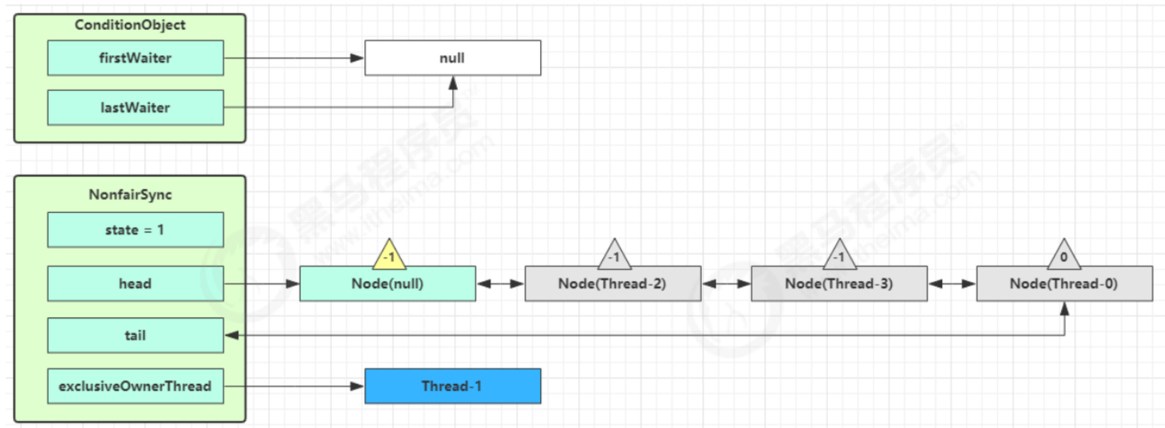


进入 ConditionObject 的 `doSignal` 流程，取得等待队列中第一个 Node，即 Thread-0 所在 Node

```java
private void doSignal(Node first) {
    do {
        // 去firstWaiter条件变量中将等待的线程拿出来.
        if ( (firstWaiter = first.nextWaiter) == null)
            lastWaiter = null;
        first.nextWaiter = null;
        // 转移到AQS的队列中，等待竞争锁
    } while (!transferForSignal(first) &&
             (first = firstWaiter) != null);
}
```



执行 `transferForSignal` 流程，**将该 Node 加入 AQS 队列尾部，将 Thread-0 的 waitStatus 改为 0，Thread-3 的waitStatus 改为 -1**，改为 `-1` 就有责任去唤醒自己的后继节点



Thread-1 释放锁，进入 unlock 流程，略

2.6.3 源码分析

```java
public class ConditionObject implements Condition,
java.io.Serializable {
```

```
    private static final long serialVersionUID =
1173984872572414699L;

    // 第一个等待节点
    private transient Node firstWaiter;

    // 最后一个等待节点
    private transient Node lastWaiter;
    public ConditionObject() { }
    // ㈠ 添加一个 Node 至等待队列
    private Node addConditionWaiter() {
        Node t = lastWaiter;
        // 所有已取消的 Node 从队列链表删除, 见 ㈡
        if (t != null && t.waitStatus != Node.CONDITION) {
            unlinkCancelledWaiters();
            t = lastWaiter;
        }
        // 创建一个关联当前线程的新 Node, 添加至队列尾部
        Node node = new Node(Thread.currentThread(),
Node.CONDITION);
        if (t == null)
            firstWaiter = node;
        else
            t.nextWaiter = node;
        lastWaiter = node;
        return node;
    }
    // 唤醒 - 将没取消的第一个节点转移至 AQS 队列
    private void doSignal(Node first) {
        do {
            // 已经是尾节点了
            if ( (firstWaiter = first.nextWaiter) == null) {
                lastWaiter = null;
            }
            first.nextWaiter = null;
        } while (
            // 将等待队列中的 Node 转移至 AQS 队列, 不成功且还有节点则继
续循环 ㈢
                !transferForSignal(first) &&
                    // 队列还有节点
                    (first = firstWaiter) != null
        );
    }

    // 外部类方法, 方便阅读, 放在此处
```

```
    // ㈢ 如果节点状态是取消，返回 false 表示转移失败，否则转移成功
    final boolean transferForSignal(Node node) {
        // 设置当前node状态为0（因为处在队列末尾），如果状态已经不是
Node.CONDITION，说明被取消了
        if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))
            return false;
        // 加入 AQS 队列尾部
        Node p = enq(node);
        int ws = p.waitStatus;
        if (
            // 插入节点的上一个节点被取消
                ws > 0 ||
                        // 插入节点的上一个节点不能设置状态为
Node.SIGNAL
                        !compareAndSetWaitStatus(p, ws,
Node.SIGNAL)
        ) {
            // unpark 取消阻塞，让线程重新同步状态
            LockSupport.unpark(node.thread);
        }
        return true;
    }
// 全部唤醒 - 等待队列的所有节点转移至 AQS 队列
private void doSignalAll(Node first) {
    lastWaiter = firstWaiter = null;
    do {
        Node next = first.nextWaiter;
        first.nextWaiter = null;
        transferForSignal(first);
        first = next;
    } while (first != null);
}

    // ㈡
    private void unlinkCancelledWaiters() {
        // ...
    }
    // 唤醒 - 必须持有锁才能唤醒，因此 doSignal 内无需考虑加锁
    public final void signal() {
        // 如果没有持有锁，会抛出异常
        if (!isHeldExclusively())
            throw new IllegalMonitorStateException();
        Node first = firstWaiter;
        if (first != null)
            doSignal(first);
```

```
    }
    // 全部唤醒 - 必须持有锁才能唤醒, 因此 doSignalAll 内无需考虑加锁
    public final void signalAll() {
        if (!isHeldExclusively())
            throw new IllegalMonitorStateException();
        Node first = firstWaiter;
        if (first != null)
            doSignalAll(first);
    }
    // 不可打断等待 - 直到被唤醒
    public final void awaitUninterruptibly() {
        // 添加一个 Node 至等待队列, 见 ㈠
        Node node = addConditionWaiter();
        // 释放节点持有的锁, 见 ㈣
        int savedState = fullyRelease(node);
        boolean interrupted = false;
        // 如果该节点还没有转移至 AQS 队列, 阻塞
        while (!isOnSyncQueue(node)) {
            // park 阻塞
            LockSupport.park(this);
            // 如果被打断, 仅设置打断状态
            if (Thread.interrupted())
                interrupted = true;
        }
        // 唤醒后, 尝试竞争锁, 如果失败进入 AQS 队列
        if (acquireQueued(node, savedState) || interrupted)
            selfInterrupt();
    }
    // 外部类方法, 方便阅读, 放在此处
    // ㈣ 因为某线程可能重入, 需要将 state 全部释放, 获取state, 然后把它全
部减掉, 以全部释放
    final int fullyRelease(Node node) {
        boolean failed = true;
        try {
            int savedState = getState();
            // 唤醒等待队列队列中的下一个节点
            if (release(savedState)) {
                failed = false;
                return savedState;
            } else {
                throw new IllegalMonitorStateException();
            }
        } finally {
            if (failed)
                node.waitStatus = Node.CANCELLED;
```

```
        }
    }
    // 打断模式 - 在退出等待时重新设置打断状态
    private static final int REINTERRUPT = 1;
    // 打断模式 - 在退出等待时抛出异常
    private static final int THROW_IE = -1;
    // 判断打断模式
    private int checkInterruptWhileWaiting(Node node) {
        return Thread.interrupted() ?
                (transferAfterCancelledWait(node) ? THROW_IE :
REINTERRUPT) :
                0;
    }
    // ㈤ 应用打断模式
    private void reportInterruptAfterWait(int interruptMode)
            throws InterruptedException {
        if (interruptMode == THROW_IE)
            throw new InterruptedException();
        else if (interruptMode == REINTERRUPT)
            selfInterrupt();
    }
    // 等待 - 直到被唤醒或打断
    public final void await() throws InterruptedException {
        if (Thread.interrupted()) {
            throw new InterruptedException();
        }
        // 添加一个 Node 至等待队列, 见 ㈠
        Node node = addConditionWaiter();
        // 释放节点持有的锁
        int savedState = fullyRelease(node);
        int interruptMode = 0;
        // 如果该节点还没有转移至 AQS 队列, 阻塞
        while (!isOnSyncQueue(node)) {
            // park 阻塞
            LockSupport.park(this);
            // 如果被打断, 退出等待队列
            if ((interruptMode =
checkInterruptWhileWaiting(node)) != 0)
                break;
        }
        // 退出等待队列后, 还需要获得 AQS 队列的锁
        if (acquireQueued(node, savedState) && interruptMode !=
THROW_IE)
            interruptMode = REINTERRUPT;
        // 所有已取消的 Node 从队列链表删除, 见 ㈡
```

```java
        if (node.nextWaiter != null)
            unlinkCancelledWaiters();
        // 应用打断模式, 见 ㈤
        if (interruptMode != 0)
            reportInterruptAfterWait(interruptMode);
    }
    // 等待 - 直到被唤醒或打断或超时
    public final long awaitNanos(long nanosTimeout) throws
InterruptedException {
        if (Thread.interrupted()) {
            throw new InterruptedException();
        }
        // 添加一个 Node 至等待队列, 见 ㈠
        Node node = addConditionWaiter();
        // 释放节点持有的锁
        int savedState = fullyRelease(node);
        // 获得最后期限
        final long deadline = System.nanoTime() + nanosTimeout;
        int interruptMode = 0;
        // 如果该节点还没有转移至 AQS 队列, 阻塞
        while (!isOnSyncQueue(node)) {
            // 已超时, 退出等待队列
            if (nanosTimeout <= 0L) {
                transferAfterCancelledWait(node);
                break;
            }
            // park 阻塞一定时间, spinForTimeoutThreshold 为 1000 ns
            if (nanosTimeout >= spinForTimeoutThreshold)
                LockSupport.parkNanos(this, nanosTimeout);
            // 如果被打断, 退出等待队列
            if ((interruptMode =
checkInterruptWhileWaiting(node)) != 0)
                break;
            nanosTimeout = deadline - System.nanoTime();
        }
        // 退出等待队列后, 还需要获得 AQS 队列的锁
        if (acquireQueued(node, savedState) && interruptMode !=
THROW_IE)
            interruptMode = REINTERRUPT;
        // 所有已取消的 Node 从队列链表删除, 见 ㈡
        if (node.nextWaiter != null)
            unlinkCancelledWaiters();
        // 应用打断模式, 见 ㈤
        if (interruptMode != 0)
            reportInterruptAfterWait(interruptMode);
```

```java
        return deadline - System.nanoTime();
    }
    // 等待 - 直到被唤醒或打断或超时, 逻辑类似于 awaitNanos
    public final boolean awaitUntil(Date deadline) throws
InterruptedException {
        // ...
    }
    // 等待 - 直到被唤醒或打断或超时, 逻辑类似于 awaitNanos
    public final boolean await(long time, TimeUnit unit) throws
InterruptedException {
        // ...
    }
    // 工具方法 省略 ...
}
```

三、读写锁

3.1 ReentrantReadWriteLock

当读操作远远高于写操作时，这时候使用读写锁让 读-读 可以并发，提高性能。 类似于数据库中的 `select ... from ... lock in share mode`

提供一个数据容器类内部分别使用读锁保护数据的 read() 方法，写锁保护数据的 write() 方法

3.1.1 读-读操作

```java
@Slf4j(topic = "c.DataContainer")
public class DataContainer {

    private Object data;

    private ReentrantReadWriteLock rw = new
ReentrantReadWriteLock();

    private ReentrantReadWriteLock.ReadLock r = rw.readLock();

    private ReentrantReadWriteLock.WriteLock w = rw.writeLock();


    public Object read() {
        log.debug("获取读锁");
        r.lock();
```

```java
        try {
            log.debug("读取");
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            log.debug("释放读锁....");
            r.unlock();
            return data;
        }
    }

    public void write() {
        log.debug("获取写锁");
        w.lock();
        try {
            log.debug("写入");
        } finally {
            log.debug("释放写锁...");
            w.unlock();
        }
    }

    public static void main(String[] args) {
        DataContainer dataContainer = new DataContainer();

        new Thread(() -> {
            dataContainer.read();
        }, "t1").start();

        new Thread(() -> {
            dataContainer.read();
        }, "t2").start();
    }
}
```

运行结果

```
07:59:56.865 [t2] DEBUG c.DataContainer - 获取读锁
07:59:56.865 [t1] DEBUG c.DataContainer - 获取读锁
07:59:56.868 [t2] DEBUG c.DataContainer - 读取
07:59:56.868 [t1] DEBUG c.DataContainer - 读取
07:59:57.869 [t2] DEBUG c.DataContainer - 释放读锁....
07:59:57.869 [t1] DEBUG c.DataContainer - 释放读锁....
```

这表示多个线程进行读取操作是不互斥的，t2锁定期间，t1的读操作不受影响

3.1.2 读-写操作

```
@Slf4j(topic = "c.DataContainer")
public class DataContainer {

    private Object data;

    private ReentrantReadWriteLock rw = new
ReentrantReadWriteLock();

    private ReentrantReadWriteLock.ReadLock r = rw.readLock();

    private ReentrantReadWriteLock.WriteLock w = rw.writeLock();


    public Object read() {
        log.debug("获取读锁");
        r.lock();
        try {
            log.debug("读取");
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            log.debug("释放读锁....");
            r.unlock();
            return data;
        }
    }

    public void write() {
        log.debug("获取写锁");
        w.lock();
        try {
            log.debug("写入");
        } finally {
            log.debug("释放写锁...");
            w.unlock();
        }
    }
```

```java
    public static void main(String[] args) throws
InterruptedException {
        DataContainer dataContainer = new DataContainer();

        new Thread(() -> {
            dataContainer.read();
        }, "t1").start();

        Thread.sleep(100);

        new Thread(() -> {
            dataContainer.write();
        }, "t2").start();
    }
}
```

运行结果

```
08:03:56.182 [t1] DEBUG c.DataContainer - 获取读锁
08:03:56.185 [t1] DEBUG c.DataContainer - 读取
08:03:56.276 [t2] DEBUG c.DataContainer - 获取写锁
08:03:57.185 [t1] DEBUG c.DataContainer - 释放读锁....
08:03:57.185 [t2] DEBUG c.DataContainer - 写入
08:03:57.185 [t2] DEBUG c.DataContainer - 释放写锁...
```

从测试结果可以看出，读写操作互斥

3.1.3 写-写操作

```java
@Slf4j(topic = "c.DataContainer")
public class DataContainer {

    private Object data;

    private ReentrantReadWriteLock rw = new
ReentrantReadWriteLock();

    private ReentrantReadWriteLock.ReadLock r = rw.readLock();

    private ReentrantReadWriteLock.WriteLock w = rw.writeLock();


    public Object read() {
```

```
        log.debug("获取读锁");
        r.lock();
        try {
            log.debug("读取");
        } finally {
            log.debug("释放读锁....");
            r.unlock();
        }

        return data;
    }

    public void write() {
        log.debug("获取写锁");
        w.lock();
        try {
            log.debug("写入");
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            log.debug("释放写锁...");
            w.unlock();
        }
    }

    public static void main(String[] args) throws
InterruptedException {
        DataContainer dataContainer = new DataContainer();

        new Thread(() -> {
            dataContainer.write();
        }, "t1").start();

        Thread.sleep(100);

        new Thread(() -> {
            dataContainer.write();
        }, "t2").start();
    }
}
```

运行结果

```
08:10:50.799 [t1] DEBUG c.DataContainer - 获取写锁
08:10:50.802 [t1] DEBUG c.DataContainer - 写入
08:10:50.892 [t2] DEBUG c.DataContainer - 获取写锁
08:10:51.804 [t1] DEBUG c.DataContainer - 释放写锁...
08:10:51.804 [t2] DEBUG c.DataContainer - 写入
08:10:52.804 [t2] DEBUG c.DataContainer - 释放写锁...
```

可以看出，写-写操作之间互斥

> 注：
>
> - 读锁不支持条件变量
> - 重入时升级不支持：即持有读锁的情况下去读取写锁，会导致获取写锁永久等待
>
> ```
> r.lock();
> try {
>     w.lock();
>     try {
>         ...
>     } finally {
>         w.unlock();
>     }
> } finally {
>     r.unlock();
> }
> ```
>
> - 重入时支持降级支持：即拥有写锁的情况下可以获取读锁
>
> ```
> class CachedData {
>     Object data;
>     // 是否有效，如果失效，需要重新计算 data
>     volatile boolean cacheValid;
>     final ReentrantReadWriteLock rwl = new
> ReentrantReadWriteLock();
>     void processCachedData() {
>         rwl.readLock().lock();
>         // 判断是否失效
>         if (!cacheValid) {
>             // 获取写锁前必须释放读锁
>             rwl.readLock().unlock();
>             rwl.writeLock().lock();
>             try {
> ```

```java
                    // double-check, 判断是否有其它线程已经获取了写锁、更新了缓存, 避免重复更新
                    if (!cacheValid) {
                        data = ...
                            cacheValid = true;
                    }
                    // 降级为读锁, 释放写锁, 这样能够让其它线程读取缓存
                    rwl.readLock().lock();
                } finally {
                    rwl.writeLock().unlock();
                }
            }
            // 自己用完数据, 释放读锁
            try {
                use(data);
            } finally {
                rwl.readLock().unlock();
            }
        }
}
```

3.1.4 应用之缓存

可以将其应用在缓存上，保证缓存和数据库上的一致性，具体看视频P249

将更新数据库和清除缓存这两步操作一起放在写锁内，两步先后无所谓，都执行完毕后释放写锁

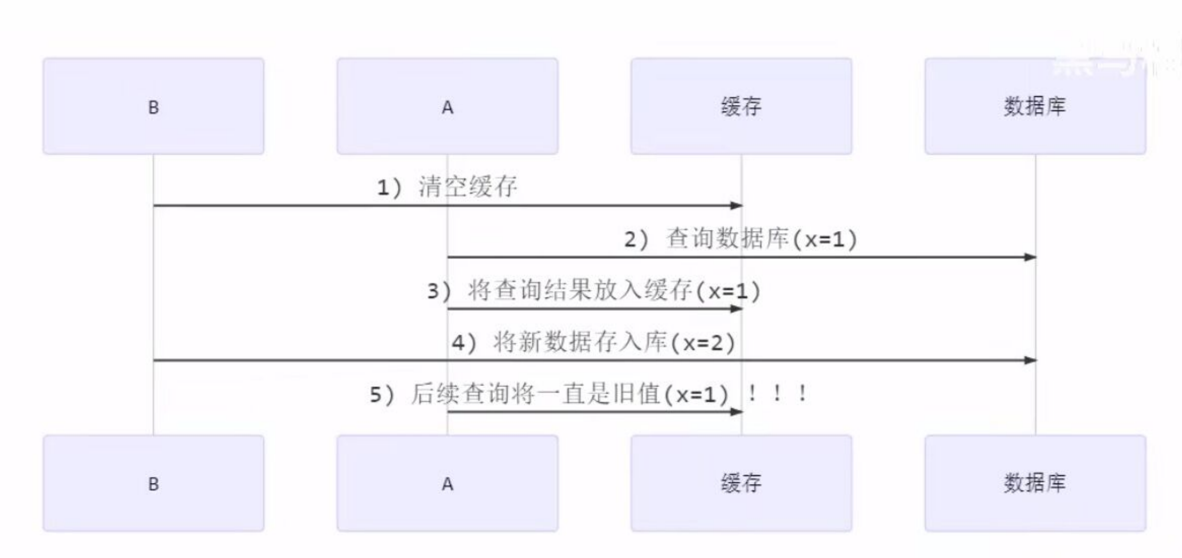
从**缓存中**获取值的操作上加上读锁，读取结束后解锁

从**数据库**获取值需要加上写锁，此时可能发生多个线程同时查询数据库中的数据，并且都向缓存中写入数据，故在获取锁后，需要用double-check检查此时缓存中是否有需要的数据，防止多次写入

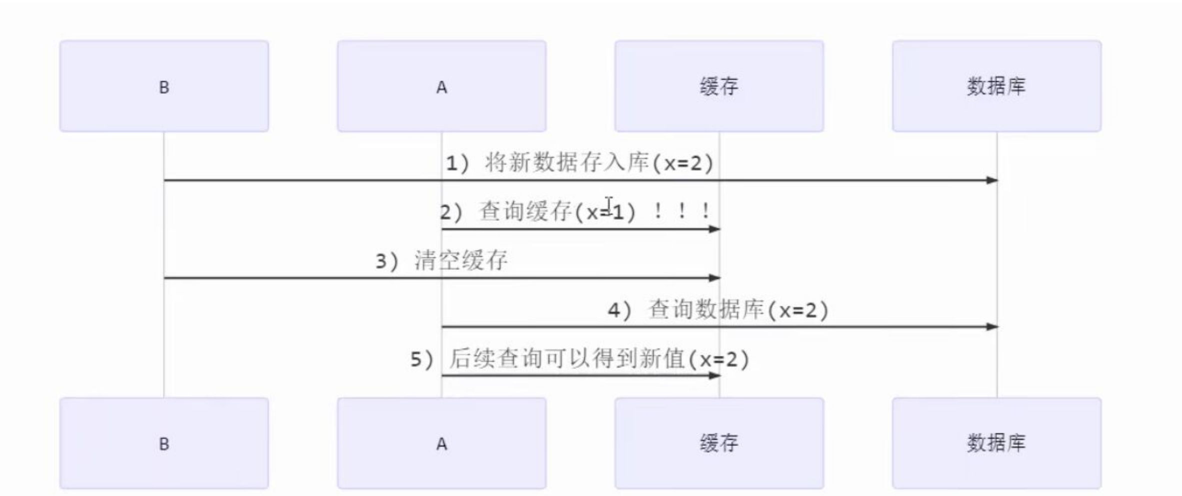
如果缓存中没有数据而去数据库中查询数据，因为查询缓存是获取读锁，查询数据库需要获取写锁，故在查询缓存之后**必须先释放读锁，再获取写锁，否则会发生永久等待**

缓存更新策略

先清理缓存后更新数据库，这时候A线程将查询数据库时将旧值写入缓存，之后每次查询都会读取缓存中的旧值，无法纠正过来



先清数据库后更新缓存，这时候虽然A线程第一次读取的是缓存中的旧值，但是之后查询只会读取到新值了，可以纠正过来



> 注意：
>
> 以上实现体现的是读写锁的应用，保证缓存和数据库的一致性，但下面的问题没有考虑
>
> - 上述实现适合读多写少，如果写操作比较频繁，性能会很低
> - 没有考虑缓存容量
> - 没有考虑缓存过期
> - 只适合单机
> - 并发性低，目前只用了一把锁，例如访问不同的表可以使用不同的锁

> - 更新方法太过简单粗暴，清空了所有的key（应该考虑按照类型分区或者重新设计key）

3.2 读写锁原理

见 `并发编程_原理.pdf`

3.3 StampedLock

3.3.1 概述

`StampedLock` 是JDK8引入的，为了进一步优化读性能，它的特点是在使用读锁、写锁时都必须配合 `戳` （下面代码中的stamp就是戳）使用

加解读锁

```
long stamp = lock.readLock();
lock.unlockRead(stamp);
```

加解写锁

```
long stamp = lock.writeLock();
lock.unlockWrite(stamp);
```

乐观读，StampedLock 支持 tryOptimisticRead() 方法（乐观读），读取完毕后需要做一次 戳校验 如果校验通 过，表示这期间确实没有写操作，数据可以安全使用，如果校验没通过，需要重新获取读锁，保证数据安全。

```
long stamp = lock.tryOptimisticRead();
// 验戳
if(!lock.validate(stamp)){
    // 锁升级
}
```

提供一个 数据容器类 内部分别使用读锁保护数据的 read() 方法，写锁保护数据的 write() 方法

```
@Slf4j(topic = "c.DataContainerStamped")
public class DataContainerStamped {

    private int data;
```

```java
    private final StampedLock lock = new StampedLock();

    public DataContainerStamped(int data) {
        this.data = data;
    }

    public int read(int readTime) {
        long stamp = lock.tryOptimisticRead();
        log.debug("optimistic read locking...{}", stamp);
        try {
            Thread.sleep(readTime);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        if (lock.validate(stamp)) {
            log.debug("read finish....{}, data:{}", stamp, data);
            return data;
        }

        //锁升级 - 读锁
        log.debug("updating to read lock.... {}", stamp);
        try {
            stamp = lock.readLock();
            log.debug("read lock {}", stamp);
            try {
                Thread.sleep(readTime);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            log.debug("read finish...{}, data:{}", stamp, data);
            return data;
        } finally {
            log.debug("read unlock {}", stamp);
            lock.unlockRead(stamp);
        }
    }

    public void write(int newData) {
        long stamp = lock.writeLock();

        try {
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
```

```
                e.printStackTrace();
            }
            this.data = newData;
        } finally {
            log.debug("write unlock {}", stamp);
            lock.unlockWrite(stamp);
        }
    }
}
```

3.3.2 读-读操作

```
public static void main(String[] args) {
    DataContainerStamped dataContainer = new
DataContainerStamped(1);
    new Thread(() -> {
        dataContainer.read(1000);
    }, "t1").start();

    try {
        Thread.sleep(500);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    new Thread(() -> {
        dataContainer.read(0);
    }, "t2").start();
}
```

运行结果

```
20:35:09.173 [t1] DEBUG c.DataContainerStamped - optimistic read
locking...256
20:35:09.665 [t2] DEBUG c.DataContainerStamped - optimistic read
locking...256
20:35:09.665 [t2] DEBUG c.DataContainerStamped - read
finish....256, data:1
20:35:10.178 [t1] DEBUG c.DataContainerStamped - read
finish....256, data:1
```

3.3.3 读-写操作

```java
public static void main(String[] args) {
    DataContainerStamped dataContainer = new
DataContainerStamped(1);
    new Thread(() -> {
        dataContainer.read(1000);
    }, "t1").start();

    try {
        Thread.sleep(500);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    new Thread(() -> {
        dataContainer.write(1000);
    }, "t2").start();
}
```

运行结果

```
20:38:54.254 [t1] DEBUG c.DataContainerStamped - optimistic read
locking...256
20:38:55.257 [t1] DEBUG c.DataContainerStamped - updating to read
lock.... 256
20:38:56.749 [t2] DEBUG c.DataContainerStamped - write unlock 384
20:38:56.749 [t1] DEBUG c.DataContainerStamped - read lock 513
20:38:57.749 [t1] DEBUG c.DataContainerStamped - read
finish...513, data:1000
20:38:57.749 [t1] DEBUG c.DataContainerStamped - read unlock 513
```

其中读取操作过程中发生了写操作，故在 `lock.validate(stamp)` 会返回false，因而会升级锁，但在升级过程中必须等待写锁释放才能继续获得读锁

> 注意：
>
> - StampedLock不支持条件变量
> - StampedLock不支持可重入

四、Semaphore

4.1 概述

信号量，用来限制能同时访问共享资源的线程上限，例如停车场停车，车位（信号量）就是共享资源，并且是有限的，故需要在停车场外摆一个公示牌告诉要停车的人还有多少停车位，当车位数为0的时候，就不允许其他车过来停车，只有当停车场中的车开走了，有空位了才允许其他车进来

```java
@Slf4j
public class SemaphoreTest {
    public static void main(String[] args) {
        Semaphore semaphore = new Semaphore(3);

        // 10个线程同时运行
        for (int i = 0; i < 10; i++) {
            new Thread(() -> {
                try {
                    // 获取信号量
                    semaphore.acquire();
                    log.debug("running...");
                    Thread.sleep(1000);
                    log.debug("end...");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    semaphore.release();
                }
            }).start();
        }
    }
}
```

运行结果

```
21:00:22.335 [Thread-1] DEBUG
com.ecifics.concurrent.part8.SemaphoreTest - running...
21:00:22.335 [Thread-0] DEBUG
com.ecifics.concurrent.part8.SemaphoreTest - running...
21:00:22.335 [Thread-2] DEBUG
com.ecifics.concurrent.part8.SemaphoreTest - running...
21:00:23.338 [Thread-2] DEBUG
com.ecifics.concurrent.part8.SemaphoreTest - end...
21:00:23.338 [Thread-0] DEBUG
com.ecifics.concurrent.part8.SemaphoreTest - end...
```

```
21:00:23.338 [Thread-1] DEBUG
com.ecifics.concurrent.part8.SemaphoreTest - end...
21:00:23.338 [Thread-3] DEBUG
com.ecifics.concurrent.part8.SemaphoreTest - running...
21:00:23.338 [Thread-4] DEBUG
com.ecifics.concurrent.part8.SemaphoreTest - running...
21:00:23.338 [Thread-5] DEBUG
com.ecifics.concurrent.part8.SemaphoreTest - running...
21:00:24.339 [Thread-4] DEBUG
com.ecifics.concurrent.part8.SemaphoreTest - end...
21:00:24.339 [Thread-5] DEBUG
com.ecifics.concurrent.part8.SemaphoreTest - end...
21:00:24.339 [Thread-3] DEBUG
com.ecifics.concurrent.part8.SemaphoreTest - end...
21:00:24.339 [Thread-6] DEBUG
com.ecifics.concurrent.part8.SemaphoreTest - running...
21:00:24.339 [Thread-8] DEBUG
com.ecifics.concurrent.part8.SemaphoreTest - running...
21:00:24.339 [Thread-7] DEBUG
com.ecifics.concurrent.part8.SemaphoreTest - running...
21:00:25.339 [Thread-6] DEBUG
com.ecifics.concurrent.part8.SemaphoreTest - end...
21:00:25.339 [Thread-8] DEBUG
com.ecifics.concurrent.part8.SemaphoreTest - end...
21:00:25.339 [Thread-7] DEBUG
com.ecifics.concurrent.part8.SemaphoreTest - end...
21:00:25.339 [Thread-9] DEBUG
com.ecifics.concurrent.part8.SemaphoreTest - running...
21:00:26.340 [Thread-9] DEBUG
com.ecifics.concurrent.part8.SemaphoreTest - end...
```

4.2 应用

- 使用Semaphore限流，在访问高峰期，让请求线程阻塞，高峰期过去再释放许可，当然它只适合限制单机线程数量，并且仅是限制线程数量，而不是限制资源数量
- 使用Semaphore实现简单连接池，相比用wait&notify实现的性能和可读性更好

```
@Slf4j
public class Pool {

    private final int poolSize;

```

```java
    private Connection[] connections;

    private AtomicIntegerArray states;

    private Semaphore semaphore;

    public Pool(int poolSize) {
        this.semaphore = new Semaphore(poolSize);
        this.connections = new Connection[poolSize];
        this.states = new AtomicIntegerArray(new int[poolSize]);
        for (int i = 0; i < poolSize; i++) {
            connections[i] = new MockConnection("连接" + (i + 1));
        }
    }

    public Connection borrow() {
        try {
            semaphore.acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        for (int i = 0; i < poolSize; i++) {
            // 获取空闲连接
            if (states.get(i) == 0) {
                if (states.compareAndSet(i, 0, 1)) {
                    log.debug("borrow {}", connections[i]);
                    return connections[i];
                }
            }
        }

        return null;
    }

    public void free(Connection conn) {
        for (int i = 0; i < poolSize; i++) {
            if (connections[i] == conn) {
                states.set(i, 0);
                log.debug("free {}", conn);
                semaphore.release();
                break;
            }
        }
    }
```

```
}
```

4.3 原理

见 并发编程_原理

五、CountdownLatch

5.1 概述

用来进行线程同步协作，等待所有线程完成倒计时。

其中构造参数用来初始化等待计数值，await() 用来等待计数归零，countDown() 用来让计数减一

5.2 基本使用

```
@Slf4j
public class TestCountdownLatch {
    public static void main(String[] args) throws
InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(3);

        new Thread(() -> {
            log.debug("begin ...");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // 计数减一
            countDownLatch.countDown();
            log.debug("end ...");
        }, "t1").start();

        new Thread(() -> {
            log.debug("begin ...");
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
```

```java
            }
            // 计数减一
            countDownLatch.countDown();
            log.debug("end ...");
        }, "t2").start();

        new Thread(() -> {
            log.debug("begin ...");
            try {
                Thread.sleep(1500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // 计数减一
            countDownLatch.countDown();
            log.debug("end ...");
        }, "t3").start();

        log.debug("waiting ...");
        countDownLatch.await();
        log.debug("end waiting....");
    }
}
```

运行结果

```
08:26:46.300 [main] DEBUG
com.ecifics.concurrent.part8.TestCountdownLatch - waiting ...
08:26:46.300 [t1] DEBUG
com.ecifics.concurrent.part8.TestCountdownLatch - begin ...
08:26:46.300 [t3] DEBUG
com.ecifics.concurrent.part8.TestCountdownLatch - begin ...
08:26:46.300 [t2] DEBUG
com.ecifics.concurrent.part8.TestCountdownLatch - begin ...
08:26:47.304 [t1] DEBUG
com.ecifics.concurrent.part8.TestCountdownLatch - end ...
08:26:47.803 [t3] DEBUG
com.ecifics.concurrent.part8.TestCountdownLatch - end ...
08:26:48.303 [t2] DEBUG
com.ecifics.concurrent.part8.TestCountdownLatch - end ...
08:26:48.303 [main] DEBUG
com.ecifics.concurrent.part8.TestCountdownLatch - end waiting....
```

5.3 CountdownLatch改进

可以配合线程池一起使用

```
public static void main(String[] args) throws
InterruptedException {
    CountDownLatch latch = new CountDownLatch(3);
    ExecutorService service = Executors.newFixedThreadPool(4);
    service.submit(() -> {
        log.debug("begin...");
        sleep(1);
        latch.countDown();
        log.debug("end...{}", latch.getCount());
    });
    service.submit(() -> {
        log.debug("begin...");
        sleep(1.5);
        latch.countDown();
        log.debug("end...{}", latch.getCount());
    });
    service.submit(() -> {
        log.debug("begin...");
        sleep(2);
        latch.countDown();
        log.debug("end...{}", latch.getCount());
    });
    service.submit(()->{
        try {
            log.debug("waiting...");
            latch.await();
            log.debug("wait end...");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });
}
```

运行结果

```
18:52:25.831 c.TestCountDownLatch [pool-1-thread-3] - begin...
18:52:25.831 c.TestCountDownLatch [pool-1-thread-1] - begin...
18:52:25.831 c.TestCountDownLatch [pool-1-thread-2] - begin...
18:52:25.831 c.TestCountDownLatch [pool-1-thread-4] - waiting...
18:52:26.835 c.TestCountDownLatch [pool-1-thread-1] - end...2
18:52:27.335 c.TestCountDownLatch [pool-1-thread-2] - end...1
18:52:27.835 c.TestCountDownLatch [pool-1-thread-3] - end...0
18:52:27.835 c.TestCountDownLatch [pool-1-thread-4] - wait end...
```

5.4 应用之同步等待多线程准备完毕

在游戏匹配时，必须等待所有的线程都加载完成才回开始游戏，这也可以应用在等待多个线程准备的情况下

```
public class CountdownLatchApplication {
    public static void main(String[] args) throws
InterruptedException {
        ExecutorService service =
Executors.newFixedThreadPool(10);
        CountDownLatch countDownLatch = new CountDownLatch(10);
        Random random = new Random();
        String[] all = new String[10];

        for (int i = 0; i < 10; i++) {
            int k = i;
            service.submit(() -> {
                for (int j = 0; j <= 100; j++) {
                    try {
                        Thread.sleep(random.nextInt(100));
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    all[k] = j + "%";
                    System.out.print("\r" +
Arrays.toString(all));
                }
                countDownLatch.countDown();

            });
        }
```

```
        countDownLatch.await();
        System.out.println("\n游戏开始");
        service.shutdown();
    }
}
```

运行结果

```
[100%, 100%, 100%, 100%, 100%, 100%, 100%, 100%, 100%, 100%]
游戏开始
```

5.5 CyclicBarrier

5.5.1 引入

等待两个线程完成可以用CountdownLatch，但是如果想让这个过程重复执行三次，由于CountdownLatch无法重用，故需要重新创建三次，也就是将下线代码中的service.submit和CountdownLatch放在一个for循环中执行三次

```
@Slf4j
public class TestCyclicBarrier {
    public static void main(String[] args) {
        ExecutorService service =
Executors.newFixedThreadPool(5);
        CountDownLatch countDownLatch = new CountDownLatch(2);

        service.submit(() -> {
            log.debug("task1 start ....");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            countDownLatch.countDown();
        });

        service.submit(() -> {
            log.debug("task2 start ....");
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
                countDownLatch.countDown();
            });

            try {
                countDownLatch.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            log.debug("task1 task2 finish...");
            service.shutdown();
        }
}
```

运行结果

```
09:02:09.877 [pool-1-thread-2] DEBUG
com.ecifics.concurrent.part8.TestCyclicBarrier - task2 start ....
09:02:09.877 [pool-1-thread-1] DEBUG
com.ecifics.concurrent.part8.TestCyclicBarrier - task1 start ....
09:02:11.881 [main] DEBUG
com.ecifics.concurrent.part8.TestCyclicBarrier - task1 task2
finish...
```

和CountdownLatch相比，CyclicBarrier可以重用

5.5.2 CyclicBarrier使用

循环栅栏，用来进行线程协作，等待线程满足某个计数。构造时设置计数个数，每个线程执行到某个需要“同步”的时刻调用 await() 方法进行等待，当等待的线程数满足计数个数时，继续执行

5.5.1中的代码可以写成下面这样，并且在构造器中第二个参数可以在计数减为0后执行：

```
@Slf4j
public class TestCyclicBarrier {
    public static void main(String[] args) {
        ExecutorService service =
Executors.newFixedThreadPool(5);
        CyclicBarrier cyclicBarrier = new CyclicBarrier(2, () ->
{
            log.debug("task1, task2 finish...");
        });

```

```java
        service.submit(() -> {
            log.debug("task1 start ....");
            try {
                Thread.sleep(1000);
                log.debug("task1 end ....");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            try {
                cyclicBarrier.await();
            } catch (InterruptedException |
BrokenBarrierException e) {
                e.printStackTrace();
            }
        });

        service.submit(() -> {
            log.debug("task2 start ....");
            try {
                Thread.sleep(2000);
                log.debug("task2 end ....");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            try {
                cyclicBarrier.await();
            } catch (InterruptedException |
BrokenBarrierException e) {
                e.printStackTrace();
            }
        });

        service.shutdown();
    }
}
```

运行结果

```
09:11:35.542 [pool-1-thread-2] DEBUG
com.ecifics.concurrent.part8.TestCyclicBarrier - task2 start ....
09:11:35.543 [pool-1-thread-1] DEBUG
com.ecifics.concurrent.part8.TestCyclicBarrier - task1 start ....
09:11:36.547 [pool-1-thread-1] DEBUG
com.ecifics.concurrent.part8.TestCyclicBarrier - task1 end ....
09:11:37.546 [pool-1-thread-2] DEBUG
com.ecifics.concurrent.part8.TestCyclicBarrier - task2 end ....
09:11:37.546 [pool-1-thread-2] DEBUG
com.ecifics.concurrent.part8.TestCyclicBarrier - task1, task2
finish...
```

CyclicBarrier可以循环使用

```
@Slf4j
public class TestCyclicBarrier {
    public static void main(String[] args) {
        ExecutorService service =
Executors.newFixedThreadPool(2);
        // 线程数必须和CyclicBarrier的第一个参数相同
        CyclicBarrier cyclicBarrier = new CyclicBarrier(2, () ->
{
            log.debug("task1, task2 finish...");
        });

        for (int i = 0; i < 3; i++) {
            service.submit(() -> {
                log.debug("task1 start ....");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                try {
                    cyclicBarrier.await();
                } catch (InterruptedException |
BrokenBarrierException e) {
                    e.printStackTrace();
                }
            });

            service.submit(() -> {
                log.debug("task2 start ....");
```

```
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                try {
                    cyclicBarrier.await();
                } catch (InterruptedException |
BrokenBarrierException e) {
                    e.printStackTrace();
                }
            });
        }

        service.shutdown();
    }
}
```

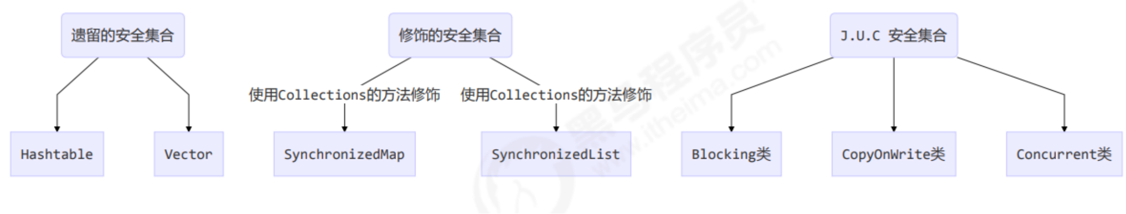
运行结果

```
21:09:37.257 [pool-1-thread-1] DEBUG
com.ecifics.concurrent.part8.TestCyclicBarrier - task1 start ....
21:09:37.257 [pool-1-thread-2] DEBUG
com.ecifics.concurrent.part8.TestCyclicBarrier - task2 start ....
21:09:39.260 [pool-1-thread-2] DEBUG
com.ecifics.concurrent.part8.TestCyclicBarrier - task1, task2
finish...
21:09:39.260 [pool-1-thread-2] DEBUG
com.ecifics.concurrent.part8.TestCyclicBarrier - task1 start ....
21:09:39.260 [pool-1-thread-1] DEBUG
com.ecifics.concurrent.part8.TestCyclicBarrier - task2 start ....
21:09:41.261 [pool-1-thread-1] DEBUG
com.ecifics.concurrent.part8.TestCyclicBarrier - task1, task2
finish...
21:09:41.261 [pool-1-thread-1] DEBUG
com.ecifics.concurrent.part8.TestCyclicBarrier - task1 start ....
21:09:41.261 [pool-1-thread-2] DEBUG
com.ecifics.concurrent.part8.TestCyclicBarrier - task2 start ....
21:09:43.262 [pool-1-thread-2] DEBUG
com.ecifics.concurrent.part8.TestCyclicBarrier - task1, task2
finish...
```

其中线程池的大小必须和CyclicBarrier的第一个参数大小相同，例如上面例子中都是2

七、线程安全集合类

7.1 概述



线程安全集合类可以分为三大类：

- 遗留的线程安全集合如 Hashtable ， Vector (所有方法都用synchronized修饰，并发性能较低，不推荐使用)
- 使用 Collections 装饰的线程安全集合（本质上是将对应集合方法上加上了synchronzied修饰），如：
  - Collections.synchronizedCollection
  - Collections.synchronizedList
  - Collections.synchronizedMap
  - Collections.synchronizedSet
  - Collections.synchronizedNavigableMap
  - Collections.synchronizedNavigableSet
  - Collections.synchronizedSortedMap
  - Collections.synchronizedSortedSet
- 推荐 java.util.concurrent.* 下的线程安全集合类，可以发现它们有规律，里面包含三类关键词： Blocking、CopyOnWrite、Concurrent
  - Blocking 大部分实现基于锁，并提供用来阻塞的方法
  - CopyOnWrite 之类容器修改开销相对较重 （适用于读多写少）
  - Concurrent 类型的容器 （性能高）
    - 内部很多操作使用 cas 优化，一般可以提供较高吞吐量
    - 弱一致性
      - 遍历时弱一致性，例如，当利用迭代器遍历时，如果容器发生修改，迭代器仍然可以继续进行遍 历，这时内容是旧的
      - 求大小弱一致性，size 操作未必是 100% 准确
      - 读取弱一致性

> 遍历时如果发生了修改，对于非安全容器来讲，使用 fail-fast 机制也就是让遍历立刻失败，抛出 ConcurrentModificationException，不再继续遍历

7.2 ConcurrentHashMap

7.2.1 概述

该类和HashMap类的相关方法使用相同，故不再赘述

7.2.2 练习-单词计数