主讲老师：Fox

> 有道云笔记链接：https://note.youdao.com/s/YlbEATOq

1. 管程 — Java同步的设计思想

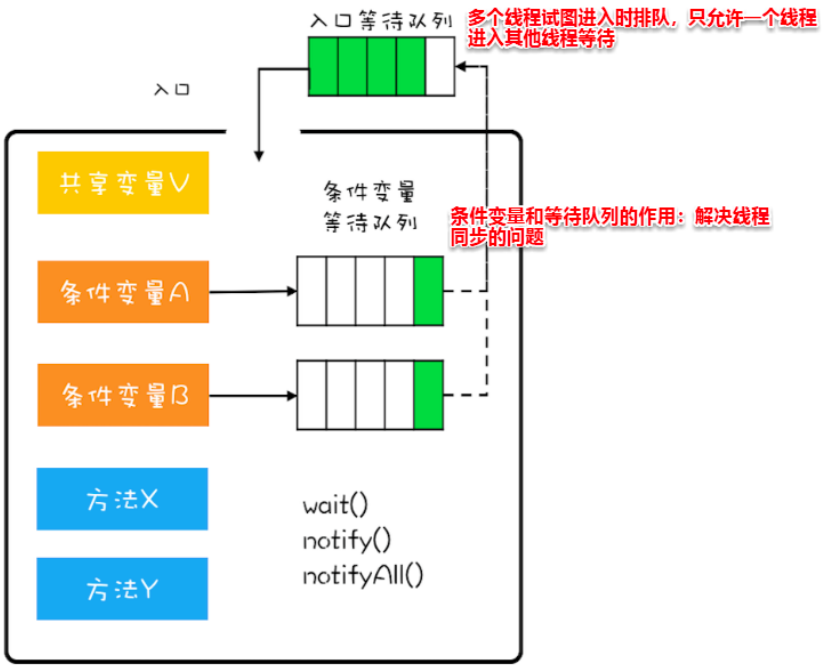
管程：指的是管理共享变量以及对共享变量的操作过程，让他们支持并发。

互斥：同一时刻只允许一个线程访问共享资源；

同步：线程之间如何通信、协作。

MESA模型

在管程的发展史上，先后出现过三种不同的管程模型，分别是Hasen模型、Hoare模型和MESA模型。现在正在广泛使用的是**MESA模型**。



MESA 管程模型

管程中引入了条件变量的概念，而且每个条件变量都对应有一个等待队列。条件变量和等待队列的作用是解决线程之间的同步问题。

Java中针对管程有两种实现

- **一种是基于Object的Monitor机制，用于synchronized内置锁的实现**
- **一种是抽象队列同步器AQS，用于JUC包下Lock锁机制的实现**

示例代码

```java
@Slf4j
public class ConditionDemo2 {

    private static final ReentrantLock lock = new ReentrantLock();
    private static final Condition condition = lock.newCondition();

    public static void main(String[] args) throws InterruptedException {
        new Thread(() -> {
            log.debug("t1开始执行....");
            lock.lock();
            try {
                log.debug("t1获取锁....");
                // 让线程在obj上一直等待下去
                condition.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
                log.debug("t1执行完成....");
            }
        }, "t1").start();

        new Thread(() -> {
            log.debug("t2开始执行....");
            lock.lock();
            try {
                log.debug("t2获取锁....");
                // 让线程在obj上一直等待下去
                condition.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
                log.debug("t2执行完成....");
            }
        }, "t2").start();

        // 主线程两秒后执行
```

```
39          Thread.sleep(2000);
40          log.debug("准备获取锁，去唤醒 condition上阻塞的线程");
41          lock.lock();
42          try {
43              // 唤醒condition上所有阻塞的线程
44              condition.signalAll();
45              log.debug("唤醒condition上阻塞的线程");
46          } catch (Exception e) {
47              e.printStackTrace();
48          } finally {
49              lock.unlock();
50          }
51
52
53      }
54  }
```

2. AQS原理分析

2.1 什么是AQS

java.util.concurrent包中的大多数同步器实现都是围绕着共同的基础行为，比如等待队列、条件队列、独占获取、共享获取等，而这些行为的抽象就是基于AbstractQueuedSynchronizer（简称AQS）实现的，AQS是一个抽象同步框架，可以用来实现一个依赖状态的同步器。

JDK中提供的大多数的同步器如Lock, Latch, Barrier等，都是基于AQS框架来实现的

- 一般是通过一个内部类Sync继承 AQS
- 将同步器所有调用都映射到Sync对应的方法

AQS具备的特性：

- 阻塞等待队列
- 共享/独占
- 公平/非公平
- 可重入
- 允许中断

2.2 AQS核心结构

```
private volatile int state;//共享变量，使用volatile修饰保证线程可见性
//返回同步状态的当前值
protected final int getState() {
        return state;
}
 // 设置同步状态的值
protected final void setState(int newState) {
        state = newState;
}
//原子地（CAS操作）将同步状态值设置为给定值update如果当前同步状态的值等于expect（期望值）
protected final boolean compareAndSetState(int expect, int update) {
        return unsafe.compareAndSwapInt(this, stateOffset, expect, update);
}
```

AQS内部维护属性volatile int state

- state表示资源的可用状态

State三种访问方式：

- getState()
- setState()
- compareAndSetState()

定义了两种资源访问方式：

- Exclusive-独占，只有一个线程能执行，如ReentrantLock
- Share-共享，多个线程可以同时执行，如Semaphore/CountDownLatch

AQS实现时主要实现以下几种方法：

- isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。
- tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。
- tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。
- tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
- tryReleaseShared(int)：共享方式。尝试释放资源，如果释放后允许唤醒后续等待结点返回true，否则返回false。

2.3 AQS定义两种队列

- 同步等待队列： 主要用于维护获取锁失败时入队的线程。
- 条件等待队列： 调用await()的时候会释放锁，然后线程会加入到条件队列，调用signal()唤醒的时候会把条件队列中的线程节点移动到同步队列中，等待再次获得锁。

AQS 定义了5个队列中节点状态：

1. 值为0，初始化状态，表示当前节点在sync队列中，等待着获取锁。
2. CANCELLED，值为1，表示当前的线程被取消；
3. SIGNAL，值为-1，表示当前节点的后继节点包含的线程需要运行，也就是unpark；
4. CONDITION，值为-2，表示当前节点在等待condition，也就是在condition队列中；
5. PROPAGATE，值为-3，表示当前场景下后续的acquireShared能够得以执行；

同步等待队列

AQS当中的同步等待队列也称CLH队列，CLH队列是Craig、Landin、Hagersten三人发明的一种基于双向链表数据结构的队列，是FIFO先进先出线程等待队列，Java中的CLH队列是原CLH队列的一个变种,线程由原自旋机制改为阻塞机制。

AQS 依赖CLH同步队列来完成同步状态的管理：

- 当前线程如果获取同步状态失败时，AQS则会将当前线程已经等待状态等信息构造成一个节点（Node）并将其加入到CLH同步队列，同时会阻塞当前线程
- 当同步状态释放时，会把首节点唤醒（公平锁），使其再次尝试获取同步状态。
- 通过signal或signalAll将条件队列中的节点转移到同步队列。（由条件队列转化为同步队列）

条件等待队列

AQS中条件队列是使用单向列表保存的，用nextWaiter来连接:

- 调用await方法阻塞线程；
- 当前线程存在于同步队列的头结点，调用await方法进行阻塞（从同步队列转化到条件队列）

2.4 基于AQS实现一把独占锁

思考： 基于AQS 如何设计一把独占锁?

```java
/**
 * @author  Fox
 *  基于AQS实现一把独占锁
 */
public class TulingLock extends AbstractQueuedSynchronizer{

    @Override
    protected boolean tryAcquire(int unused) {
        //cas 加锁  state=0
        if (compareAndSetState(0, 1)) {
            setExclusiveOwnerThread(Thread.currentThread());
            return true;
        }

        return false;
    }

    @Override
    protected boolean tryRelease(int unused) {
        //释放锁
        setExclusiveOwnerThread(null);
        setState(0);
        return true;
    }

    public void lock() {
        acquire(1);
    }

    public boolean tryLock() {
        return tryAcquire(1);
    }

    public void unlock() {
        release(1);
    }

    public boolean isLocked() {
```

```
        return getState() != 0;
    }


}
```

3. ReentrantLock源码分析

ReentrantLock是一种基于AQS框架的应用实现，是JDK中的一种线程并发访问的同步手段，它的功能类似于synchronized是一种互斥锁，可以保证线程安全。

ReentrantLock使用方式

```
public class ReentrantLockTest {
    private final ReentrantLock lock = new ReentrantLock();
    // ...

    public void doSomething() {
        lock.lock();  // block until condition holds
        try {
            // ... method body
        } finally {
            lock.unlock();
        }
    }
}
```

3.1 ReentrantLock原理

ReentrantLock基于 AQS + CAS 实现。

lock()流程图

ReentrantLock基于抽象队列同步器AQS + CAS 实现的加锁、释放锁。ReentrantLock实现了公平锁、非公平锁，公平锁与非公平锁唯一的区别在于，非公平锁不会判断等待队列中是否节点等待获取锁，而是直接尝试获取锁，获取不到，再将当前线程节点添加进等待队列的尾节点，判断当前线程节点是否挂起。

unlock()流程图

ReentrantLock释放锁的流程较为简单，优先判断持有锁资源的线程是否为当前线程，若不为当前线程抛出异常；若为当前线程，AQS的state的属性值减1，再判断减1后的值是否为0，若为0表示当前线程彻底释放锁资源，唤醒等待队列中的挂起线程节点，开始抢占锁资源。

3.2 ReentrantLock源码分析

构造函数

```
private final Sync sync;

// 默认使用非公平锁
public ReentrantLock() {
    sync = new NonfairSync();
}

// fair=true，公平锁；否则，非公平锁
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

Sync是ReentrantLock的抽象静态内部类，继承自AQS(AbstractQueuedSynchronizer) - 抽象队列同步器，AQS中定义了锁的基本行为，AQS中用volatile修饰的state表示当前锁重入的次数。
NonfairSync、FairSync是ReentrantLock的静态内部类，继承ReentrantLock$Sync，NonfairSync实现非公平锁，FairSync实现公平锁。

lock()加锁

```
private final Sync sync;

// 加锁
public void lock() {
    sync.lock();
}
```

公平锁

调用AQS的acquire方法。ReentrantLock$FairSync#lock() 核心代码：

```
// 加锁
final void lock() {
    acquire(1);
}
```

非公平锁

通过CAS尝试获取锁(将AQS的state由0修改为1)，若成功，代表当前线程获取锁资源成功；若失败调用AQS的acquire方法。ReentrantLock$NonfairSync#lock() 核心代码：

```
// 加锁
final void lock() {
    // 获取锁资源，CAS 修改 AQS 的 state 属性值,，获取成功，设置当前线程
    if (compareAndSetState(0, 1))
        setExclusiveOwnerThread(Thread.currentThread());
    // 获取失败，执行AQS的acquire
    else
        acquire(1);
}
```

acquire()

acquire()方法是Sync父类AQS中的方法，AbstractQueuedSynchronizer#acquire() 核心代码：

```
// 获取锁资源
public final void acquire(int arg) {
    // 尝试获取锁资源
    if (!tryAcquire(arg) &&
        // 当前线程为获取到锁资源，加入等待队列，同时挂起线程，等待唤醒
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

tryAcquire()

tryAcquire()方法在FairSync、NonFairSync中均有实现，尝试获取锁资源，核心代码如下：

```java
// 公平锁 FairSync#tryAcquire() 方法
protected final boolean tryAcquire(int acquires) {
    // 获取当前线程
    final Thread current = Thread.currentThread();
    // 获取AQS的 state
    int c = getState();
    // state == 0 当前没有线程占用锁资源
    if (c == 0) {
        // 判断是否有线程在排队，若有线程在排队，返回true
        if (!hasQueuedPredecessors() &&
            // 尝试抢锁
            compareAndSetState(0, acquires)) {
            // 无线程排队，将线程属性设置为当前线程
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    // state != 0  有线程占用锁资源
    // 占用锁资源的线程是否为当前线程
    else if (current == getExclusiveOwnerThread()) {
        // state + 1
        int nextc = c + acquires;
        // 锁重入超出最大限制 (int的最大值)，抛异常
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        // 将 state + 1 设置给 state
        setState(nextc);
        // 当前线程拿到锁资源，返回true
        return true;
    }
    return false;
}

// 非公平锁  NonFairSync#tryAcquire() 方法
protected final boolean tryAcquire(int acquires) {
    return nonfairTryAcquire(acquires);
}

```

```java
// 非公平锁  Sync#nonfairTryAcquire() 方法
final boolean nonfairTryAcquire(int acquires) {
     // 获取当前线程
    final Thread current = Thread.currentThread();
    // 获取AQS的 state
    int c = getState();
    // 无线程占用锁资源
    if (c == 0) {
        // CAS 修改 state 的值，修改成功，设置线程属性为当前线程，返回占用锁资源标识
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    // 有线程占用锁资源
    // 占用锁资源的线程是当前线程（重入）
    else if (current == getExclusiveOwnerThread()) {
        // AQS 的 state + acquires
        int nextc = c + acquires;
        // 超出锁重入的上限(int的最大值)，抛异常
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        // 将 state + acquires 设置到 state 属性
        setState(nextc);
        return true;
    }
    return false;
}
```

获取当前线程、AQS的state。AQS的state属性值为0，表示无线程占用锁资源，判断等待队列中是否有线程在排队，若有线程在排队，返回尝试抢锁失败标识，将线程添加进等待队列中。

若state属性值不为0，判断持有锁资源的线程是否为当前线程，若为当前线程，AQS的state属性值 + 1，返回尝试抢锁成功标识。

公平锁与非公平锁的整体实现流程类似，唯一不同的是，AQS的state属性值为0，无线程占用锁资源时，非公平锁不会判断是否有线程在等待队列中排队，而是直接通过CAS抢锁。

addWaiter()

为当前线程创建入队节点AbstractQueuedSynchronizer$Node，入参mode表示锁类型，在AQS的静态内部类Node中有SHARE、EXCLUSIVE两个属性，SHARE代表共享锁、EXCLUSIVE代表排它锁。

AbstractQueuedSynchronizer#addWaiter() 核心代码：

```
// 等待队列的尾节点，懒加载，只能通过enq方法添加节点
private transient volatile Node tail;

private Node addWaiter(Node mode) {
    // 当前线程、获取的锁类型封装为Node对象
    Node node = new Node(Thread.currentThread(), mode);
    // 获取等待队列的尾节点
    Node pred = tail;
    // 尾节点不为null
    if (pred != null) {
        // 将当前节点设置为等待队列的尾节点
        node.prev = pred;
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }
    // 等待队列为空，初始化等待队列节点信息
    enq(node);
    // 返回当前线程节点
    return node;
}
```

等待队列不为空，将当前线程封装的Node节点添加进队列尾部；若等待队列为空，先初始化等待队列，然后在将Node节点添加进队列尾部。

enq()

等待队列尾节点为空时，执行enq()方法初始化等待队列，并将Node节点添加进等待队列中。

```
private Node enq(final Node node) {
    for (;;) {
        // 获取等待队列的尾节点
        Node t = tail;
        // 等待队列为空，初始化等待队列
        if (t == null) {
            // 初始化等待队列头尾节点
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {
            // 当前线程的Node添加到等待队列中
            node.prev = t;
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
    }
}
```

acquireQueued()

当前线程是否挂起，AbstractQueuedSynchronizer#acquireQueued() 核心代码：

```java
final boolean acquireQueued(final Node node, int arg) {
    // 获取锁资源标识
    boolean failed = true;
    try {
        boolean interrupted = false;
        // 自旋
        for (;;) {
            // 获取当前节点的前驱节点
            final Node p = node.predecessor();
            // 当前节点的前驱节点为头节点，并获取锁资源成功
            if (p == head && tryAcquire(arg)) {
                // 将当前节点设置到head - 头节点
                setHead(node);
                // 原头节点的下一节点指向设置为null，GC回收
                p.next = null;
                // 设置获取锁资源成功
                failed = false;
                // 不管线程GC
                return interrupted;
            }
            // 如果当前节点不是head的下一节点，获取锁资源失败,尝试将线程挂起
            if (shouldParkAfterFailedAcquire(p, node) &&
                // 线程挂起， UNSAFE.park()
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

查看当前排队的Node是否是head的next， 如果是，尝试获取锁资源， 如果不是或者获取锁资源失败那么就尝试将当前Node的线程挂起（unsafe.park()）。

shouldParkAfterFailedAcquire检查并更新未成功获取锁资源的状态，返回true表示线程被挂起。

AbstractQueuedSynchronizer#shouldParkAfterFailedAcquire() 核心代码：

```
static final class Node {
    // 线程被取消
    static final int CANCELLED =  1;
    // 等待队列中存在待被唤醒的挂起线程
    static final int SIGNAL    = -1;
    // 当前线程在Condition队列中，未在AQS对列中
    static final int CONDITION = -2;
    // 解决JDK1.5的BUG。共享锁在释放资源后，若头节点为0，无法确定真的没有后继节点
    // 如果头节点为0，需要将头节点的状态改为 -3 ，当最新拿到锁资源的线程查看
    // 是否有后继节点并且为当前锁为共享锁，需唤醒排队的线程。
    static final int PROPAGATE = -3;
}

// 获取锁资源失败，挂起线程
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
    // 获取当前节点的上一个节点的状态
    int ws = pred.waitStatus;
    // 上一节点被挂起
    if (ws == Node.SIGNAL)
        // 返回true，挂起当前线程
        return true;
    if (ws > 0) {
        // 上一节点被取消，获取最近的线程挂起节点，
        // 并将当前节点的上一节点指向最近的线程挂起节点
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        // 最近线程挂起节点的下一节点指向当前节点
        pred.next = node;
    } else {
        // 上一节点状态小于等于0，存在线程处于等待状态，但未被挂起的场景
        // 通过CAS将处于等待的线程挂起，避免在挂起前节点获取到锁资源
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    //  返回true，不挂起当前线程
    return false;
}
```

在挂起线程前，确认当前节点的上一个节点的状态。若为1，代表是取消的节点，不能挂起；若为-1，代表后续节点中有挂起的线程；若为-2 (线程在等待队列 - Condition队列中)、-3 (避免线程无法唤醒的一个状态)，需要将状态改为-1之后，才能挂起当前线程。

unlock()释放锁

释放锁，ReentrantLock#unlock() 核心代码：

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

unlock方法实际调用的是AQS的release方法，AbstractQueuedSynchronizer#release() 核心代码：

```
// 等待队列的头节点，懒加载，通过setHead方法初始化
private transient volatile Node head;

// 释放锁
public final boolean release(int arg) {
    // 当前线程释放锁资源的计数值
    if (tryRelease(arg)) {
        // 当前线程玩去释放锁资源，获取等待队列头节点
        Node h = head;
        if (h != null && h.waitStatus != 0)
            // 唤醒等待队列中待唤醒的节点
            unparkSuccessor(h);
        // 完全释放锁资源
        return true;
    }
    // 当前线程未完全释放锁资源
    return false;
}
```

tryRelease()

释放锁，Reenttrant$Sync#tryRelease()的核心代码：

```java
// 释放锁
protected final boolean tryRelease(int releases) {
    // 修改 AQS 的 state
    int c = getState() - releases;
    // 当前线程不是持有锁的线程，抛出异常
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    // 是否成功的将锁资源完全释放标识 （state == 0）
    boolean free = false;
    // 锁资源完全释放
    if (c == 0) {
        // 修改标识
        free = true;
        // 将占用锁资源的属性设置为null
        setExclusiveOwnerThread(null);
    }
    // state赋值
    setState(c);
    // 返回true表示当前线程完全释放锁资源；
    // 返回false标识当前线程是由锁资源，持有计数值减少
    return free;
}
```