相比synchronize的监视器锁，Condition提供了更加灵活和精确的线程控制。它的最大特点是可以为不同的线程建立多个Condition，从而达到精确控制某一些线程的休眠与唤醒。

Condition是一个接口，内部主要提供了一些线程休眠与唤醒相关的方法，代码如下：

public interface Condition {

// 使当前线程进入等待状态,可以相应中断请求

void await() throws InterruptedException;

// 使当前线程进入等待状态，不响应中断请求

void awaitUninterruptibly();

// 使当前线程进入等待状态，直到被唤醒或中断，或者经过指定的等待时间。nanosTimeout单位纳秒

long awaitNanos(long nanosTimeout) throws InterruptedException;

// 同awaitNanos方法，可以指定时间单位

boolean await(long time, TimeUnit unit) throws InterruptedException;

// 使线程进入等待状态，直到被被唤醒或者中断，或者到截止的时间

boolean awaitUntil(Date deadline) throws InterruptedException;

// 唤醒一个等待在Condition上的线程，与notify功能类似

void signal();

// 唤醒所有等待在Condition上的线程，与notifyAll类似

void signalAll();

}

Condition的实现类是在AQS中的ConditionObject，关于ConditionObject我们后边再看，接下来看下如何使用Condition来实现线程的等待与唤醒。

Condition实现“生产者-消费者”模式

仍然以“生产者-消费者”模式来看Condition的使用，沿用上篇文章生产面包的例子，稍加改动后的面包容器类如下：

public class BreadContainer {

LinkedList<Bread> list = new LinkedList<>();

private final static int CAPACITY = 10;

Lock lock = new ReentrantLock();

private final Condition providerCondition = lock.newCondition();

private final Condition consumerCondition = lock.newCondition();

public void put(Bread bread) {

try {

lock.lock();

while (list.size() == CAPACITY) {

try {

// 如果容器已满，则阻塞生产者线程

providerCondition.await();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

list.add(bread);

// 面包生产成功后通知消费者线程

consumerCondition.signalAll();

System.out.println(Thread.currentThread().getName() + " product a bread" + bread.toString() + " size = " + list.size());

} catch (Exception e) {

e.printStackTrace();

} finally {

lock.unlock();

}

}

public void take() {

try {

lock.lock();

while (list.isEmpty()) {

try {

// 如果容器为空，则阻塞消费者线程

consumerCondition.await();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

Bread bread = list.removeFirst();

// 消费后通知生产者生产面包

providerCondition.signalAll();

System.out.println("Consumer " + Thread.currentThread().getName() + " consume a bread" + bread.toString() + " size = " + list.size());

} catch (Exception e) {

e.printStackTrace();

} finally {

lock.unlock();

}

}

}

可以看到，在上述代码中我们声明了两个Condition，一个生产者Condition，一个消费者Condition。在put方法中使用ReentrantLock来实现同步，同时，当容器满时调用生产者Condition的await方法使生产者线程进入等待状态。如果生成成功，则调用消费者Condition的signalAll方法来唤醒消费者线程。take方法与put类似，不再赘述。这里要注意的是在使用Condition前必须先获得锁。

生产者消费者类与synchronize的实现一致，代码如下：

// 生产者

public class Producer implements Runnable {

private final BreadContainer container;

public Producer(BreadContainer container) {

this.container = container;

}

@Override

public void run() {

container.put(new Bread());

}

}

// 消费者

public class Consumer implements Runnable {

private final BreadContainer container;

public Consumer(BreadContainer container) {

this.container = container;

}

@Override

public void run() {

container.take();

}

}

那接下来测试类我们仍然实例化多个生产者线程与多个消费者线程，如下：

public class Test {

public static void main(String[] args) {

BreadContainer container = new BreadContainer();

new Thread(() -> {

for (int i = 0; i < 100; i++) {

new Thread(new Producer(container)).start();

}

}).start();

new Thread(() -> {

for (int i = 0; i < 100; i++) {

new Thread(new Consumer(container)).start();

}

}).start();

}

}

运行后生产者线程与消费者线程可以很好的实现线程协作。与使用synchronized不同的是这里有两个Condition，分别来控制生产者和消费者。

Condition仅仅是一个接口，它的具体实现是在AQS的内部类ConditionObject中。调用ReentrantLock的newCondition实际上就是实例化了一个ConditionObject，代码如下：

// ReentrantLock#Sync

final ConditionObject newCondition() {

return new ConditionObject();

}

可见，在第一章BreadContainer中的providerCondition与consumerCondition是两个不同的ConditionObject实例。

ConditionObject的类结构如下：

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

// 指向等待队列的头结点

private transient Node firstWaiter;

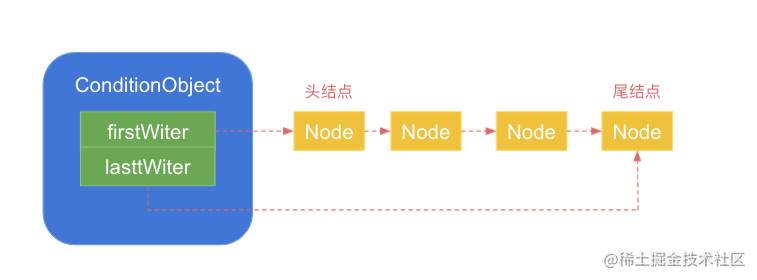
// 指向等待队列的尾结点

private transient Node lastWaiter;

public ConditionObject() { }

}

ConditionObject的结构比较简单，它内部维护了一个Node类型等待队列（这里注意与AQS中的同步队列区分）。其中firstWaiter指向队列的头结点，而lastWaiter指向队列的尾结点。关于Node节点，在ReentrantLock那篇文章中已经详细介绍过了，它封装的是一个线程的节点，这里也不再赘述。在线程中调用了Condition的await方法后，线程就会被封装成一个Node节点，并将Node的waitStatus设置成CONDITION状态，然后插入到这个Condition的等待队列中。等到收到singal或者被中断、超时就会被从等待队列中移除。其结构示意图如下：



1.Condition的await方法

public final void await() throws InterruptedException {

// 如果线程被标记位中断状态，则抛出中断异常

if (Thread.interrupted())

throw new InterruptedException();

// 将当前线程封装成一个Node节点，并添加到等待队列

Node node = addConditionWaiter();

// 释放锁

int savedState = fullyRelease(node);

int interruptMode = 0;

// 判断当前node是否在同步队列中，注意如果不在同步队列，则是一个阻塞的死循环

while (!isOnSyncQueue(node)) {

// 不在同步队列中，则挂起线程

LockSupport.park(this);

if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)

break;

}

// 到这里说明节点已加入到同步队列中，调用acquireQueued开始竞争锁

if (acquireQueued(node, savedState) && interruptMode != THROW\_IE)

interruptMode = REINTERRUPT;

if (node.nextWaiter != null) // clean up if cancelled

// 清理被标记为CANCLLED状态的节点

unlinkCancelledWaiters();

if (interruptMode != 0)

reportInterruptAfterWait(interruptMode);

}

在wait方法中首先会调用addConditionWaiter方法将线程封装成一个Node节点，并加入到等待队列中。addConditionWaiter的代码如下：

private Node addConditionWaiter() {

if (!isHeldExclusively())

throw new IllegalMonitorStateException();

Node t = lastWaiter;

// 清除CANCLLED状态的lastWaiter节点

if (t != null && t.waitStatus != Node.CONDITION) {

unlinkCancelledWaiters();

t = lastWaiter;

}

// 实例化一个Node节点，并标记为CONDITION状态

Node node = new Node(Node.CONDITION);

// 将node加入到等待队列

if (t == null)

firstWaiter = node;

else

t.nextWaiter = node;

lastWaiter = node;

return node;

}

addConditionWaiter方法的逻辑比较简单，就是将线程封装成Node并加入等待队列的操作。加入队列后，await方法又调用了fullyRelease去释放锁，在fullyRelease方法中会将state置为0，代码如下：

final int fullyRelease(Node node) {

try {

// 获取AQS中的state

int savedState = getState();

// 调用release释放锁

if (release(savedState))

return savedState;

throw new IllegalMonitorStateException();

} catch (Throwable t) {

// 释放失败则将节点置为CANCELLED状态

node.waitStatus = Node.CANCELLED;

throw t;

}

}

这个方法主要是调用了release方法来释放锁，如果释放失败，则将节点置为CANCELLED状态。关于release这个方法在ReentrantLock中已经分析过，这里不再赘述。

释放锁之后，开启while来调用isOnSyncQueue方法，这个方法是用来判断当前节点是否在同步队列中。如果不在同步队列，则会进入自旋，并阻塞线程，等待节点进入同步队列。isOnSyncQueue的代码如下：

final boolean isOnSyncQueue(Node node) {

// 如果waitStatus是CONDITION状态或者node的前驱节点是null，说明该节点在等待队列中，而非同步队列。

if (node.waitStatus == Node.CONDITION || node.prev == null)

return false;

// 如果node.next不为null，则一定在同步队列

if (node.next != null)

return true;

// 如果前面没有确定node是否在同步队列，则遍历同步队列查看是否存在node节点

return findNodeFromTail(node);

}

private boolean findNodeFromTail(Node node) {

// tail即同步队列的队尾，从队尾遍历并与node对比

for (Node p = tail;;) {

if (p == node)

return true;

if (p == null)

return false;

p = p.prev;

}

}

如果isOnSyncQueue返回了true，那么说明该node节点已经进入同步队列中了，则会结束自旋并调用acquireQueued，关于acquireQueued在ReentrantLock文章中已经详细分析过了，即一个获取锁的操作。

总的来说，调用await方法会让线程进入等待队列，并释放锁。当等待队列中的节点被唤醒时，会将节点移入到同步队列，然后await结束自旋，并调用acquireQueued来获取锁。

2.Condition的signal方法

这里我们选用signal方法来分析，signal方法类似Object中的notify方法，调用signal方法会将等待队列的首节点移入同步队列并唤醒。它的实现相比await来说比较简单，看下代码：

public final void signal() {

if (!isHeldExclusively())

throw new IllegalMonitorStateException();

Node first = firstWaiter;

if (first != null)

// 唤醒等待队列的第一个节点

doSignal(first);

}

final boolean transferForSignal(Node node) {

Node p = enq(node);

int ws = p.waitStatus;

if (ws > 0 || !p.compareAndSetWaitStatus(ws, Node.SIGNAL))

LockSupport.unpark(node.thread);

return true;

}

在signal中会拿到等待队列的首节点并调用doSignal方法将其唤醒，doSignal代码如下：

private void doSignal(Node first) {

do {

if ( (firstWaiter = first.nextWaiter) == null)

lastWaiter = null;

first.nextWaiter = null;

// 尝试唤醒等待队列的首节点，如果唤醒失败则继续尝试

} while (!transferForSignal(first) &&

(first = firstWaiter) != null);

}

doSignal方法中是一个循环唤醒等待队列首节点的操作，核心方法是transferForSignal，代码如下：

final boolean transferForSignal(Node node) {

// 如果当前节点状态为CONDITION，则CAS将状态改为0,准备加入同步队列，如果状态不为CONDITION，则说明线程被中断，返回false，然后唤醒当前节点的后继节点

if (!node.compareAndSetWaitStatus(Node.CONDITION, 0))

return false;

// 将节点加入到同步队列，并返回同步队列的先驱节点

Node p = enq(node);

int ws = p.waitStatus;

// waitStatus>0为取消状态，则CAS尝试修改成SINGAL状态

if (ws > 0 || !p.compareAndSetWaitStatus(ws, Node.SIGNAL))

// 如果修改状态失败，那么久直接唤醒当前线程

LockSupport.unpark(node.thread);

return true;

}

private Node enq(Node node) {

for (;;) {

Node oldTail = tail;

if (oldTail != null) {

node.setPrevRelaxed(oldTail);

if (compareAndSetTail(oldTail, node)) {

oldTail.next = node;

return oldTail;

}

} else {

initializeSyncQueue();

}

}

}

transferForSignal实际上就是做了一个队列的转移，将node从等待队列移动到了同步队列。进入同步队列后，在wait方法中的自旋操作便能检测到node节点的状态，从而执行acquireQueued方法拿锁。

总的来说signal方法会从等待队列的队首开始，尝试唤醒队首线程，如果该节点是CANCELLED状态，则继续唤醒下一个。当节点被唤醒后会将其加入到同步队列，接着wait方法停止自旋执行acquireQueued方法。

通过对Condition的await与signal方法的分析，可以看得出来这两个方法并非独立存在，而是一个相互配合的关系。await方法会将执行的线程封装成Node加入到等待队列，然后开启一个循环检测这个node看是否被加入到了同步队列，如果被加入到同步队列，那么调用acquireQueued继续竞争锁，如果没有被加入同步队列，则会一直等待。而signal方法则是将等待队列中的队首元素移动到同步队列，这样就出发了await方法的循环终结，继而能够执行acquireQueued方法。其流程如下图所示：

