Thread

1. Node {
2. **int** waitStatus;
3. Node prev;
4. Node next;
5. Node nextWaiter;
6. Thread **thread**;
7. }

# 第一篇 J2SE

## 1.5 Thread

### 1.5.5 Concurrent包

#### 1.5.5.1 Pool

##### ExecutorService

Code: ThreadPool101



Future



* isDone()： 执行状态
* get(): 获取执行结果。 Waits if necessary for the computation to complete, and then retrieves its result.

##### 实现原理

复用执行前面任务的线程，add过程受阻将添加到workQueue

[Reference](https://blog.csdn.net/cjh94520/article/details/70545202)

1. 线程被包装成worker



2. getTask()中有条件的进行着死循环，从而可以不断接受任务来进行。

1. **final** **void** runWorker(Worker w) {
2. **while** (task != **null** || (task = getTask()) != **null**) {
3. task.run();
4. }
5. }

执行任务时直接调用线程的run方法。所以池内线程可以执行不同任务。

线程池作用

* 减少在创建和销毁线程上花的时间
* 控制系统资源的开销。如果不使用线程池，有可能造成系统创建大量同类线程而导致消耗完内存或者“过度切换”的问题

#### 1.5.5.2 ReentrantLock

实现原理

[Reference](https://www.cnblogs.com/maypattis/p/6403682.html)

**ReentrantLock源码**



可见：默认为非公平锁

内部类继承关系



### AQS

即AbstractQueuedSynchronizer 。 AQS实现FIFO等待队列，构建锁和同步容器提供了框架。Concurrent包内许多类都是基AQS构建。

#### CLH Lock

[Reference](https://www.jianshu.com/p/f43e581976b9)

CLH锁是一个自旋锁，能确保无饥饿性，提供先来先服务的公平性。线程必须等待占用的锁被释放后，通过竞争锁来获取资源的控制权。等待获取锁方式有“被动”和“主动”两种。

被动模式 VS 主动模式

被动模式：竞争线程自身进入阻塞状态，由某种调度机制唤醒；

主动模式：通常使用自旋的方式实现。竞争不激烈，锁资源很快就可以被释放的前提下，自旋锁的计算开销，要低于被动模式。

FIFO

在自旋模式下，依赖线程自身的抢占可能会导致其中某些线程始终无法获得锁，因此需要一个策略来保证争夺锁的线程可以以某种顺序获得锁的控制权，

实现方式：1.线程请求锁的顺序 2.线程优先级来。

#### AQS原理

。AQS基于CLH锁的思想，主要从两方面进行了改造：**节点的结构与节点等待机制。**

* 在结构上引入了头结点和尾节点。
* 在等待机制上由原来的自旋改成阻塞唤醒。

#### 实现细节

**AbstractOwnableSynchronizer**

1. **class** AbstractOwnableSynchronizer{
2. //当前占用该锁的线程
3. **private** **transient** Thread exclusiveOwnerThread;
4. //同步状态
5. **private** **volatile** **int** state;
6. **private** **transient** **volatile** Node head;
7. **private** **transient** **volatile** Node tail;
8. }

state

1 占用

0 未占用

##### Node

节点。

1. Node {
2. **int** waitStatus;
3. Node prev;
4. Node next;
5. Node nextWaiter;
6. Thread thread;
7. }

**waitStatus**

1.CANCELLED( 1)，表示当前的线程被取消；

2.SIGNAL(-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；

3.CONDITION(-2)，表示当前节点在等待condition，也就是在condition队列中；

4.PROPAGATE(-3)，表示当前场景下后续的acquireShared能够得以执行；

5.0，默认。表示当前节点在sync队列中，等待着获取锁。

**nextWaiter**

节点模式。

AQL构造的队列实际是Node的链表



AQS在结构上引入了头结点和尾节点，他们分别指向队列的头和尾，尝试获取锁、入队列、释放锁等实现都与头尾节点相关，并且每个节点都引入前驱节点和后后续节点的引用。

##### Lock

**ReentrantLock.lock()**

1. **final** **void** lock() {
2. acquire(1);
3. }

**acquire**

1. **public** **final** **void** acquire(**int** arg) {
2. **if** (!tryAcquire(arg) &&
3. acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
4. selfInterrupt();
5. }

selfInterrupt: Thread.interrupted()

**tryAcquire：**

获取锁，根据state判断锁是否被占用。

1. **public** **boolean** tryAcquire(**int** acquires) {
2. **assert** acquires == 1; // Otherwise unused
3. **if** (compareAndSetState(0, 1)) {
4. setExclusiveOwnerThread(Thread.currentThread());
5. **return** **true**;
6. }
7. **return** **false**;
8. }

**acquireQueued**

对队列的节点进行阻塞

1. **final** **boolean** acquireQueued(**final** Node node, **int** arg) {
2. **boolean** interrupted = **false**;
3. **for** (;;) {
4. **final** Node p = node.predecessor();
5. **if** (p == head && tryAcquire(arg)) {  //
6. setHead(node);
7. p.next = **null**; // help GC
8. **return** interrupted;
9. }
10. **if** (shouldParkAfterFailedAcquire(p, node) &&
11. parkAndCheckInterrupt())
12. interrupted = **true**;
13. }
15. }

这里自选

**addWaiter**

入队。通过CAS把当前线程追加到队尾

1. **private** Node addWaiter(Node mode) {
2. Node node = **new** Node(Thread.currentThread(), mode);
3. ... Node追加到队尾
4. }

##### unLock

1. **public** **void** unlock() {
2. sync.release(1);
3. }

**release**

1. **public** **final** **boolean** release(**int** arg) {
2. **if** (tryRelease(arg)) {
3. Node h = head;
4. **if** (h != **null** && h.waitStatus != 0)
5. unparkSuccessor(h);  //
6. **return** **true**;
7. }
8. **return** **false**;
9. }

从head出释放等待队列。

**tryRelease**

1. **protected** **boolean** tryRelease(**int** releases) {
2. setExclusiveOwnerThread(**null**);
3. setState(0);
4. **return** **true**;
5. }

**unparkSuccessor**

 唤醒Node。release处传入head Node

1. **private** **void** unparkSuccessor(Node node) {
2. **int** ws = node.waitStatus;
3. **if** (ws < 0)
4. compareAndSetWaitStatus(node, ws, 0);
5. Node s = node.next;
6. **if** (s == **null** || s.waitStatus > 0) {
7. s = **null**;
8. **for** (Node t = tail; t != **null** && t != node; t = t.prev)
9. **if** (t.waitStatus <= 0)
10. s = t;
11. }
12. **if** (s != **null**)  //next node存在
13. LockSupport.unpark(s.thread);
14. }

## 1.6 JVM

### 运行时数据区

#### Cache

cache的意义

缓解CPU和内存之间速度的不匹配问题。

原理

如果CPU需要的内容在cache里

局部性原理。/2-8理论

* 时间局部性：如果某个数据被访问，那么在不久的将来它很可能被再次访问；
* 空间局部性：如果某个数据被访问，那么与它相邻的数据很快也可能被访问；

cache的写回方式

cache的写操作方式可以追溯到大学教程《计算机组成原理》一书。

* write through（写通）：每次CPU修改了cache中的内容，立即更新到内存。每次CPU写共享数据，都会导致总线事务，导致总线事务的竞争，效率非常低。
* write back（写回）：每次CPU修改了cache中的数据，不会立即更新到内存，而是等到cache line在某一个必须或合适的时机才会更新到内存中；

无论是写通还是写回，在多线程环境下都需要处理缓存cache一致性问题。

线程安全

从cache的角度，产生线程不安全的原因：

当程序在运行过程中，会将运算需要的数据从主存复制一份到CPU的高速缓存当中，那么CPU进行计算时就可以直接从它的高速缓存读取数据和向其中写入数据，当运算结束之后，再将高速缓存中的数据刷新到主存当中。

如果一个变量在多个CPU中都存在缓存（一般在多线程编程时才会出现），那么就可能存在缓存不一致的问题

缓存一致性

1）通过在总线加LOCK#锁的方式

2）通过缓存一致性协议，比如MESI协议

MESI协议

处理缓存cache一致性问题。