ReetrantLock -可重入锁

1. ReentrantLock介绍 -(lock中使用频率最高的类)

ReentrantLock重入锁，是实现Lock接口的一个类，也是在实际编程中使用频率很高的一个锁，支持重入性，表示能够对共享资源能够重复加锁，即当前线程获取该锁再次获取不会被阻塞。

在java关键字synchronized隐式支持重入性，synchronized通过获取自增（每个锁对象拥有一个锁计数器和一个指向持有该锁 的线程的指针。锁计数器：**引用计数** 器），释放自减的方式实现重入。

与此同时，ReentrantLock还支持公平锁和非公平锁两种方式

2.特性

2.1.重入性锁的特点

I.线程获取锁的时候，如果已经获得锁的线程时，当前线程在次进入代码块时直接再次获取锁。

II.由于锁会被获取N次，因此锁只有被释放N次后才算真正的释放成功。

2.1公平，非公平性

3.特性的实现

3.1重入锁的实现

```
final boolean nonfairTryAcquire(int acquires) {
 //拿到线程
 final Thread current = Thread.currentThread();
 int c = getState();
 if (c == 0) {
 //当前同步状态还未被获取
 //当前线程使用CAS尝试获取同步状态
 if (compareAndSetState(0, acquires)) {
 setExclusiveOwnerThread(current);
  return true;
  }
  }
  //此时同步状态不为0，表述当前线程获取到了同步状态
  //判断持有线程是否是当前线程
  else if (current == getExclusiveOwnerThread()) {
  //若是当前线程，同步状态+1（计数器+1）
  int nextc = c + acquires;
  if (nextc < 0) // overflow //锁计数器异常，
  throw new Error("Maximum lock count exceeded");
  //把最新状态写回内存中（再次+1的同步状态）
  setState(nextc);
  return true;
  }
  //当前锁已经被线程持有且 当前线程不是持有锁线程，
  return false;
 }
```

- [x] ~~问题1：setState(nextc)：为什么不用CAS操作：~~

已解决:state（同步状态是用volatile关键字修饰的，保持内存可见性，而`setState`原子性操作)

```
private volatile int state;

```

```
3
4
5  /**
6   * Sets the value of synchronization state.
7   * This operation has memory semantics of a {@code volatile} wri
 te.
8   * @param newState the new state value
9   */
10 protected final void setState(int newState) {
11   state = newState;
12 }
```

- [x] ~~问题2：所计数器异常，为什么是这样判断。~~

已解决；计数器是int类型的值 （int最大值+1 -> int的最小值（是一个负数））；

3.2公平锁与非公平锁的实现

- **ReentrantLock支持两种锁：公平锁和非公平锁。何谓公平性，是针对获取锁而言的，如果一个锁是公平的，那么锁 的获取顺序就应该符合请求上的绝对时间顺序，满足FIFO。**

- **ReentrantLock的构造方法无参时是构造非公平锁（默认非公平锁）**

> **公平同步器，非公平同步器都是AQS子类Sync接口，再ReetrantLock中的实现子类 覆写Sync的protected抽象方法的**

公平锁：锁的获取顺序一定满足时间上的绝对顺序，等待时间最长的线程一定最先获得锁。

3.2.1非公平同步器

```
1 public ReentrantLock() { //默认无参构造
2   sync = new NonfairSync();
3 }
```

另外还提供了另外一种方式，可传入一个boolean值，true时为公平锁，false时为非公平锁，源码为：

3.2.2公平同步器

```
public ReentrantLock(boolean fair) {
 //如果是true表示构造了公平锁，反之非公平锁
 sync = fair ? new FairSync() : new NonfairSync();
}
```

ReentrantLock默认使用非公平锁

> **特点：**
> **速度快，吞吐量大**

3.2.3非公平锁的实现 nonFairSync

```
final void lock() {
 //不再同步队列中的线程，可能会直接获取到锁
 if (compareAndSetState(0, 1))
 setExclusiveOwnerThread(Thread.currentThread());
 else
 acquire(1);
}

//-----------相同的acquire(1)
public final void acquire(int arg) {
 if (!tryAcquire(arg) &&
 acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
 selfInterrupt();
}
```

不同的 tryAcquire（）子类自己覆写protected的方法

```
nonfairTryAcquire
protected final boolean tryAcquire(int acquires) {
 return nonfairTryAcquire(acquires);
 }
}



final boolean nonfairTryAcquire(int acquires) {
 final Thread current = Thread.currentThread();
 int c = getState();
 if (c == 0) {
 //相比于公平锁只少了hasQueuedPredecessors()
 if (compareAndSetState(0, acquires)) {
```

```
 setExclusiveOwnerThread(current);
 return true;
 }
 }
 //-----------------------以下相同 锁的重入机制相同
 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;
}

```

3.2.4公平锁的实现FairSvnc

```
final void lock()
 //少了一次CAS
 acquire(1);
}
//-----------相同的acquire(1)
public final void acquire(int arg) {
 if (!tryAcquire(arg) &&
 acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
 selfInterrupt();
 }


//不同的 tryAcquire（）子类自己覆写protected的方法
 FairSync

protected final boolean tryAcquire(int acquires) {
 //-----------相同
 final Thread current = Thread.currentThread();
 int c = getSt;
 //----------------------比相同的
 if (c == 0) {
 //相比于非公平锁☞增加了hasQueuedPredecessors()
 //当前同步队列中存在非空节点，当前线程直接封装为Node节点排队
 if (!hasQueuedPredecessors() &&
```

```
 compareAndSetState(0, acquires)) {
 setExclusiveOwnerThread(current);
 return true;
 }
 }
//------------------------以下相同 锁的重入机制相同
 else if (current == getExclusiveOwnerThread()) {
 int nextc = c + acquires;
 if (nextc < 0)
 throw new Error("Maximum lock count exceeded");
 setState(nextc);
 return true;
 }
 return false;
 }
}

```

3.2.5公平锁，非公平锁源码不同点

公平锁必须要有：同步队列才可以且只保证队列的头获得锁。即等待时间最长的线程，才可以获得锁。

- [ ] **源码235行**

```
//就这一句
if (!hasQueuedPredecessors() &&


* @return {@code true} if there is a queued thread preceding the
* current thread, and {@code false} if the current thread
* is at the head of the queue or the queue is empty
* @since 1.7
/**翻译：
 *如果前面有一个排队的线程，则返回{@code true}
 *当前线程，如果当前线程为{@code false}
 *是在队列的头部或队列为空
 * @since 1.7
 */


 //判断当前队列是否是当前的的同步队列的头节点
```

```
18 public final boolean hasQueuedPredecessors() {
19  // The correctness of this depends on head being initialized
20  // before tail and on head.next being accurate if the current
21  // thread is first in queue.
22  Node t = tail; // Read fields in reverse initialization order
23  Node h = head;
24  Node s;
25  //非空队列，且头节点下一个结点为null，或则当前线程不是持有锁线程的下一个结点（头节点的下一个节点）
26  //返回ture;
27  //反之返回 false;
28  return h != t &&
29  ((s = h.next) == null || s.thread != Thread.currentThread());
30 }
31
```

4.公平锁和非公平锁的对比

公平锁保证每次获取锁均为同步度列的第一个节点，**保证了请求资源上时间上的绝对顺序**，但是效率低，需要频繁的上下文切换。（上下文切换，阻塞唤醒，操作系统需要从用户态，到内核态来回切换）

优点：公平性

非公共锁会降低开销，降低了上下文切换，但是可能导致其他线程永远无法获取到锁，造成线程”**饥饿**“现象。

优点：速度快，吞吐量大

通常来讲，没有特性的公平性要求尽量选择非公平锁（ReetratLock默认选择）

因此，ReentrantLock默认选择的是非公平锁，则是为了减少一部分上下文切换，保证了系统更大的吞吐量。