## Synchronized锁优化

### 轻量级锁

引入轻量级锁的主要目的是在多没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。当关闭偏向锁功能或者多个线程竞争偏向锁导致偏向锁升级为轻量级锁，则会尝试获取轻量级锁，其步骤如下：**获取锁**

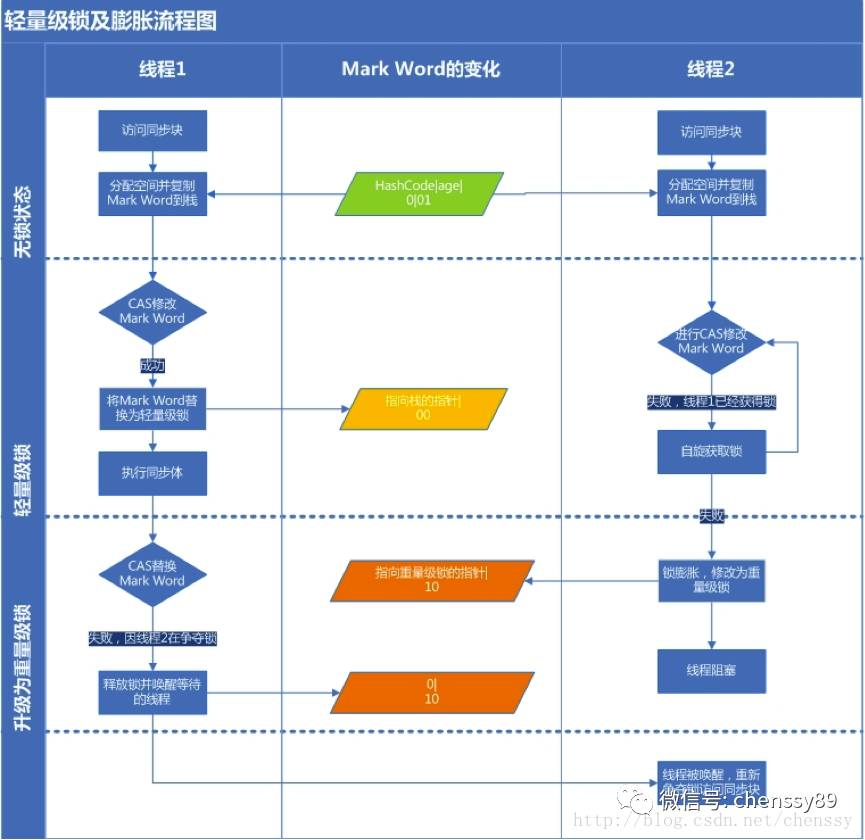
1. 判断当前对象是否处于无锁状态（hashcode、0、01），若是，则JVM首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝（官方把这份拷贝加了一个Displaced前缀，即Displaced Mark Word）；否则执行步骤（3）；
2. JVM利用CAS操作尝试将对象的Mark Word更新为指向Lock Record的指正，如果成功表示竞争到锁，则将锁标志位变成00（表示此对象处于轻量级锁状态），执行同步操作；如果失败则执行步骤（3）；
3. 判断当前对象的Mark Word是否指向当前线程的栈帧，如果是则表示当前线程已经持有当前对象的锁，则直接执行同步代码块；否则只能说明该锁对象已经被其他线程抢占了，这时轻量级锁需要膨胀为重量级锁，锁标志位变成10，后面等待的线程将会进入阻塞状态；

**释放锁**轻量级锁的释放也是通过CAS操作来进行的，主要步骤如下：

1. 取出在获取轻量级锁保存在Displaced Mark Word中的数据；
2. 用CAS操作将取出的数据替换当前对象的Mark Word中，如果成功，则说明释放锁成功，否则执行（3）；
3. 如果CAS操作替换失败，说明有其他线程尝试获取该锁，则需要在释放锁的同时需要唤醒被挂起的线程。

对于轻量级锁，其性能提升的依据是“对于绝大部分的锁，在整个生命周期内都是不会存在竞争的”，如果打破这个依据则除了互斥的开销外，还有额外的CAS操作，因此在有多线程竞争的情况下，轻量级锁比重量级锁更慢；

下图是轻量级锁的获取和释放过程



### 偏向锁

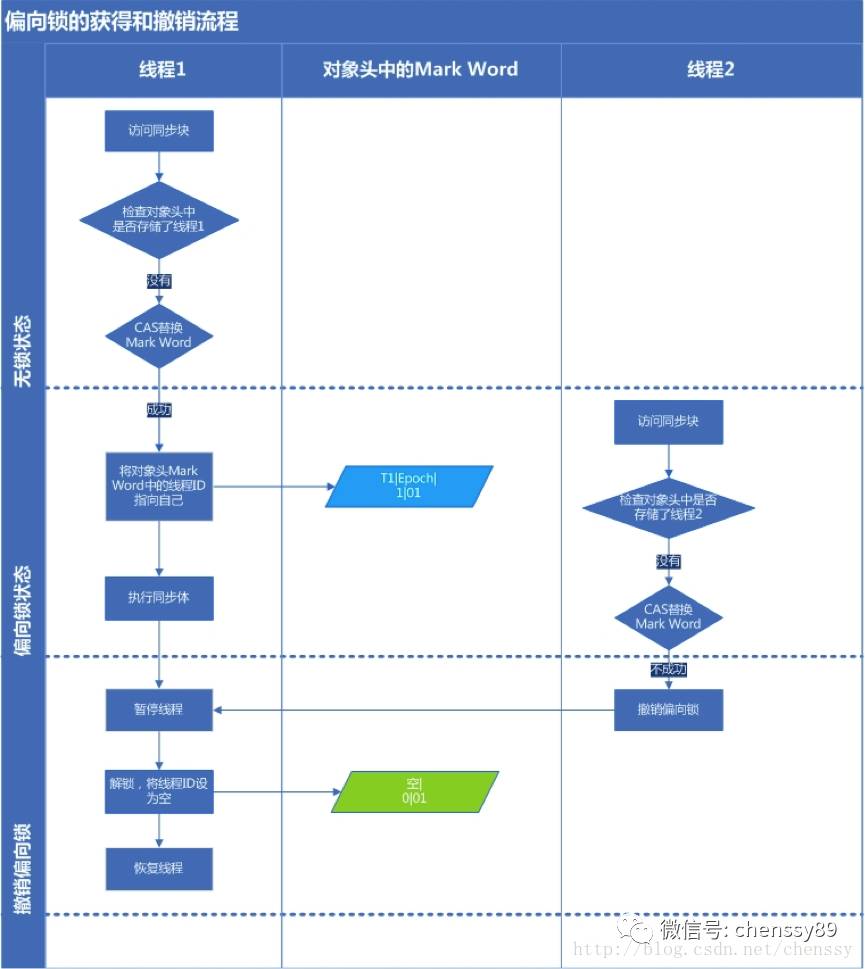
引入偏向锁主要目的是：为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径。上面提到了轻量级锁的加锁解锁操作是需要依赖多次CAS原子指令的。那么偏向锁是如何来减少不必要的CAS操作呢？我们可以查看Mark work的结构就明白了。只需要检查是否为偏向锁、锁标识为以及ThreadID即可，处理流程如下：**获取锁**

1. 检测Mark Word是否为可偏向状态，即是否为偏向锁1，锁标识位为01；
2. 若为可偏向状态，则测试线程ID是否为当前线程ID，如果是，则执行步骤（5），否则执行步骤（3）；
3. 如果线程ID不为当前线程ID，则通过CAS操作竞争锁，竞争成功，则将Mark Word的线程ID替换为当前线程ID，否则执行线程（4）；
4. 通过CAS竞争锁失败，证明当前存在多线程竞争情况，当到达全局安全点，获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码块；
5. 执行同步代码块

**释放锁**偏向锁的释放采用了一种只有竞争才会释放锁的机制，线程是不会主动去释放偏向锁，需要等待其他线程来竞争。偏向锁的撤销需要等待全局安全点（这个时间点是上没有正在执行的代码）。其步骤如下：

1. 暂停拥有偏向锁的线程，判断锁对象石是否还处于被锁定状态；
2. 撤销偏向苏，恢复到无锁状态（01）或者轻量级锁的状态；

下图是偏向锁的获取和释放流程



### 重量级锁

重量级锁通过对象内部的监视器（monitor）实现，其中monitor的本质是依赖于底层操作系统的Mutex Lock实现，操作系统实现线程之间的切换需要从用户态到内核态的切换，切换成本非常高。

## AQS：

AQS的设计模式采用的模板方法模式，子类通过继承的方式，实现它的抽象方法来管理同步状态，对于子类而言它并没有太多的活要做，AQS提供了大量的模板方法来实现同步，主要是分为三类：独占式获取和释放同步状态、共享式获取和释放同步状态、查询同步队列中的等待线程情况。自定义子类使用AQS提供的模板方法就可以实现自己的同步语义。

### 独占式

独占式，同一时刻仅有一个线程持有同步状态。

#### 独占式同步状态获取

acquire(int arg)方法为AQS提供的模板方法，该方法为独占式获取同步状态，但是该方法对中断不敏感，也就是说由于线程获取同步状态失败加入到CLH同步队列中，后续对线程进行中断操作时，线程不会从同步队列中移除。代码如下：

public final void acquire(int arg) {

if (!tryAcquire(arg) &&

acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

selfInterrupt();

}

各个方法定义如下：

1. tryAcquire：去尝试获取锁，获取成功则设置锁状态并返回true，否则返回false。该方法自定义同步组件自己实现，该方法必须要保证线程安全的获取同步状态。
2. addWaiter：如果tryAcquire返回FALSE（获取同步状态失败），则调用该方法将当前线程加入到CLH同步队列尾部。
3. acquireQueued：当前线程会根据公平性原则来进行阻塞等待（自旋）,直到获取锁为止；并且返回当前线程在等待过程中有没有中断过。
4. selfInterrupt：产生一个中断。

acquireQueued方法为一个自旋的过程，也就是说当前线程（Node）进入同步队列后，就会进入一个自旋的过程，每个节点都会自省地观察，当条件满足，获取到同步状态后，就可以从这个自旋过程中退出，否则会一直执行下去。如下：

final boolean acquireQueued(final Node node, int arg) {

boolean failed = true;

try {

//中断标志

boolean interrupted = false;

/\*

\* 自旋过程，其实就是一个死循环而已

\*/

for (;;) {

//当前线程的前驱节点

final Node p = node.predecessor();

//当前线程的前驱节点是头结点，且同步状态成功

if (p == head && tryAcquire(arg)) {

setHead(node);

p.next = null; // help GC

failed = false;

return interrupted;

}

//获取失败，线程等待--具体后面介绍

if (shouldParkAfterFailedAcquire(p, node) &&

parkAndCheckInterrupt())

interrupted = true;

}

} finally {

if (failed)

cancelAcquire(node);

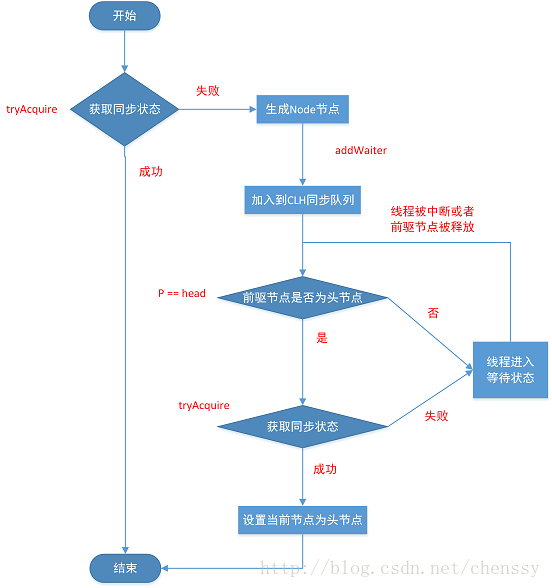
}

}

从上面代码中可以看到，当前线程会一直尝试获取同步状态，当然前提是只有其前驱节点为头结点才能够尝试获取同步状态，理由：

1. 保持FIFO同步队列原则。
2. 头节点释放同步状态后，将会唤醒其后继节点，后继节点被唤醒后需要检查自己是否为头节点。

acquire(int arg)方法流程图如下：

[](https://gitee.com/chenssy/blog-home/raw/master/image/sijava/2018120811001.png)

#### 独占式获取响应中断

AQS提供了acquire(int arg)方法以供独占式获取同步状态，但是该方法对中断不响应，对线程进行中断操作后，该线程会依然位于CLH同步队列中等待着获取同步状态。为了响应中断，AQS提供了acquireInterruptibly(int arg)方法，该方法在等待获取同步状态时，如果当前线程被中断了，会立刻响应中断抛出异常InterruptedException。

public final void acquireInterruptibly(int arg)

throws InterruptedException {

if (Thread.interrupted())

throw new InterruptedException();

if (!tryAcquire(arg))

doAcquireInterruptibly(arg);

}

首先校验该线程是否已经中断了，如果是则抛出InterruptedException，否则执行tryAcquire(int arg)方法获取同步状态，如果获取成功，则直接返回，否则执行doAcquireInterruptibly(int arg)。doAcquireInterruptibly(int arg)定义如下：

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())

throw new InterruptedException();

}

} finally {

if (failed)

cancelAcquire(node);

}

}

doAcquireInterruptibly(int arg)方法与acquire(int arg)方法仅有两个差别。1.方法声明抛出InterruptedException异常，2.在中断方法处不再是使用interrupted标志，而是直接抛出InterruptedException异常。

#### 独占式超时获取

AQS除了提供上面两个方法外，还提供了一个增强版的方法：tryAcquireNanos(int arg,long nanos)。该方法为acquireInterruptibly方法的进一步增强，它除了响应中断外，还有超时控制。即如果当前线程没有在指定时间内获取同步状态，则会返回false，否则返回true。如下：

public final boolean tryAcquireNanos(int arg, long nanosTimeout)

throws InterruptedException {

if (Thread.interrupted())

throw new InterruptedException();

return tryAcquire(arg) ||

doAcquireNanos(arg, nanosTimeout);

}

tryAcquireNanos(int arg, long nanosTimeout)方法超时获取最终是在doAcquireNanos(int arg, long nanosTimeout)中实现的，如下：

private boolean doAcquireNanos(int arg, long nanosTimeout)

throws InterruptedException {

//nanosTimeout <= 0

if (nanosTimeout <= 0L)

return false;

//超时时间

final long deadline = System.nanoTime() + nanosTimeout;

//新增Node节点

final Node node = addWaiter(Node.EXCLUSIVE);

boolean failed = true;

try {

//自旋

for (;;) {

final Node p = node.predecessor();

//获取同步状态成功

if (p == head && tryAcquire(arg)) {

setHead(node);

p.next = null; // help GC

failed = false;

return true;

}

/\*

\* 获取失败，做超时、中断判断

\*/

//重新计算需要休眠的时间

nanosTimeout = deadline - System.nanoTime();

//已经超时，返回false

if (nanosTimeout <= 0L)

return false;

//如果没有超时，则等待nanosTimeout纳秒

//注：该线程会直接从LockSupport.parkNanos中返回，

//LockSupport为JUC提供的一个阻塞和唤醒的工具类，后面做详细介绍

if (shouldParkAfterFailedAcquire(p, node) &&

nanosTimeout > spinForTimeoutThreshold)

LockSupport.parkNanos(this, nanosTimeout);

//线程是否已经中断了

if (Thread.interrupted())

throw new InterruptedException();

}

} finally {

if (failed)

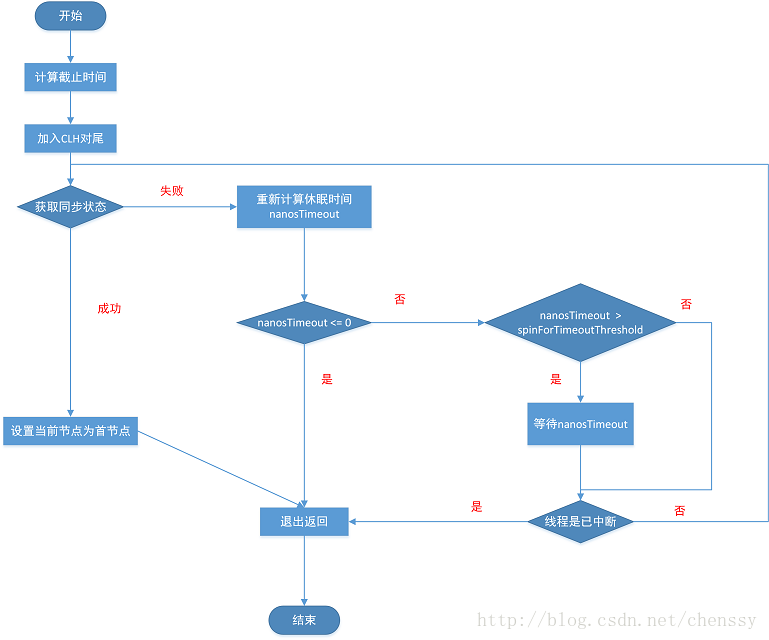
cancelAcquire(node);

}

}

针对超时控制，程序首先记录唤醒时间deadline ，deadline = System.nanoTime() + nanosTimeout（时间间隔）。如果获取同步状态失败，则需要计算出需要休眠的时间间隔nanosTimeout（= deadline – System.nanoTime()），如果nanosTimeout <= 0 表示已经超时了，返回false，如果大于spinForTimeoutThreshold（1000L）则需要休眠nanosTimeout ，如果nanosTimeout <= spinForTimeoutThreshold ，就不需要休眠了，直接进入快速自旋的过程。原因在于 spinForTimeoutThreshold 已经非常小了，非常短的时间等待无法做到十分精确，如果这时再次进行超时等待，相反会让nanosTimeout 的超时从整体上面表现得不是那么精确，所以在超时非常短的场景中，AQS会进行无条件的快速自旋。

整个流程如下：

[](https://gitee.com/chenssy/blog-home/raw/master/image/sijava/2018120811002.png)

#### 独占式同步状态释放

当线程获取同步状态后，执行完相应逻辑后就需要释放同步状态。AQS提供了release(int arg)方法释放同步状态：

public final boolean release(int arg) {

if (tryRelease(arg)) {

Node h = head;

if (h != null && h.waitStatus != 0)

unparkSuccessor(h);

return true;

}

return false;

}

该方法同样是先调用自定义同步器自定义的tryRelease(int arg)方法来释放同步状态，释放成功后，会调用unparkSuccessor(Node node)方法唤醒后继节点（如何唤醒LZ后面介绍）。

这里稍微总结下：

在AQS中维护着一个FIFO的同步队列，当线程获取同步状态失败后，则会加入到这个CLH同步队列的对尾并一直保持着自旋。在CLH同步队列中的线程在自旋时会判断其前驱节点是否为首节点，如果为首节点则不断尝试获取同步状态，获取成功则退出CLH同步队列。当线程执行完逻辑后，会释放同步状态，释放后会唤醒其后继节点。

### 共享式

共享式与独占式的最主要区别在于同一时刻独占式只能有一个线程获取同步状态，而共享式在同一时刻可以有多个线程获取同步状态。例如读操作可以有多个线程同时进行，而写操作同一时刻只能有一个线程进行写操作，其他操作都会被阻塞。

#### 共享式同步状态获取

AQS提供acquireShared(int arg)方法共享式获取同步状态：

public final void acquireShared(int arg) {

if (tryAcquireShared(arg) < 0)

//获取失败，自旋获取同步状态

doAcquireShared(arg);

}

从上面程序可以看出，方法首先是调用tryAcquireShared(int arg)方法尝试获取同步状态，如果获取失败则调用doAcquireShared(int arg)自旋方式获取同步状态，共享式获取同步状态的标志是返回 >= 0 的值表示获取成功。自选式获取同步状态如下：

private void doAcquireShared(int arg) {

/共享式节点

final Node node = addWaiter(Node.SHARED);

boolean failed = true;

try {

boolean interrupted = false;

for (;;) {

//前驱节点

final Node p = node.predecessor();

//如果其前驱节点，获取同步状态

if (p == head) {

//尝试获取同步

int r = tryAcquireShared(arg);

if (r >= 0) {

setHeadAndPropagate(node, r);

p.next = null; // help GC

if (interrupted)

selfInterrupt();

failed = false;

return;

}

}

if (shouldParkAfterFailedAcquire(p, node) &&

parkAndCheckInterrupt())

interrupted = true;

}

} finally {

if (failed)

cancelAcquire(node);

}

}

tryAcquireShared(int arg)方法尝试获取同步状态，返回值为int，当其 >= 0 时，表示能够获取到同步状态，这个时候就可以从自旋过程中退出。

acquireShared(int arg)方法不响应中断，与独占式相似，AQS也提供了响应中断、超时的方法，分别是：acquireSharedInterruptibly(int arg)、tryAcquireSharedNanos(int arg,long nanos)，这里就不做解释了。

#### 共享式同步状态释放

获取同步状态后，需要调用release(int arg)方法释放同步状态，方法如下：

public final boolean releaseShared(int arg) {

if (tryReleaseShared(arg)) {

doReleaseShared();

return true;

}

return false;

}

因为可能会存在多个线程同时进行释放同步状态资源，所以需要确保同步状态安全地成功释放，一般都是通过CAS和循环来完成的。

final boolean acquireQueued(final Node node, int arg) {

boolean failed = true;

try {

//中断标志

boolean interrupted = false;

/\*

\* 自旋过程，其实就是一个死循环而已

\*/

for (;;) {

//当前线程的前驱节点

final Node p = node.predecessor();

//当前线程的前驱节点是头结点，且同步状态成功

if (p == head && tryAcquire(arg)) {

setHead(node);

p.next = null; // help GC

failed = false;

return interrupted;

}

//获取失败，线程等待--具体后面介绍

if (shouldParkAfterFailedAcquire(p, node) &&

parkAndCheckInterrupt())

interrupted = true;

}

} finally {

if (failed)

cancelAcquire(node);

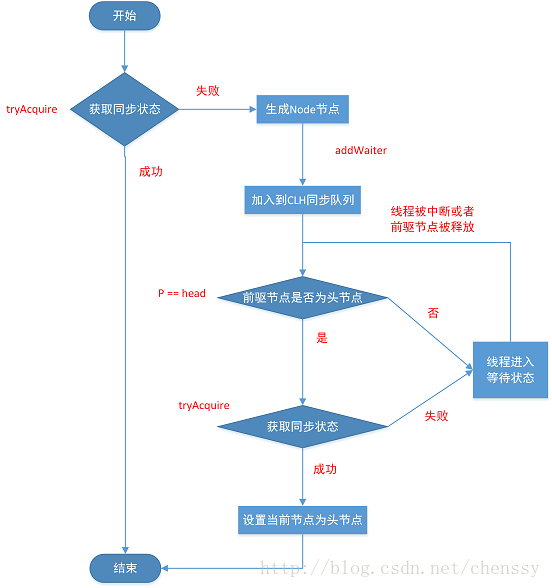
}

}

从上面代码中可以看到，当前线程会一直尝试获取同步状态，当然前提是只有其前驱节点为头结点才能够尝试获取同步状态，理由：

1. 保持FIFO同步队列原则。
2. 头节点释放同步状态后，将会唤醒其后继节点，后继节点被唤醒后需要检查自己是否为头节点。

acquire(int arg)方法流程图如下：

[](https://gitee.com/chenssy/blog-home/raw/master/image/sijava/2018120811001.png)