二

27 什么是自旋锁？自旋的好处和后果是什么呢？

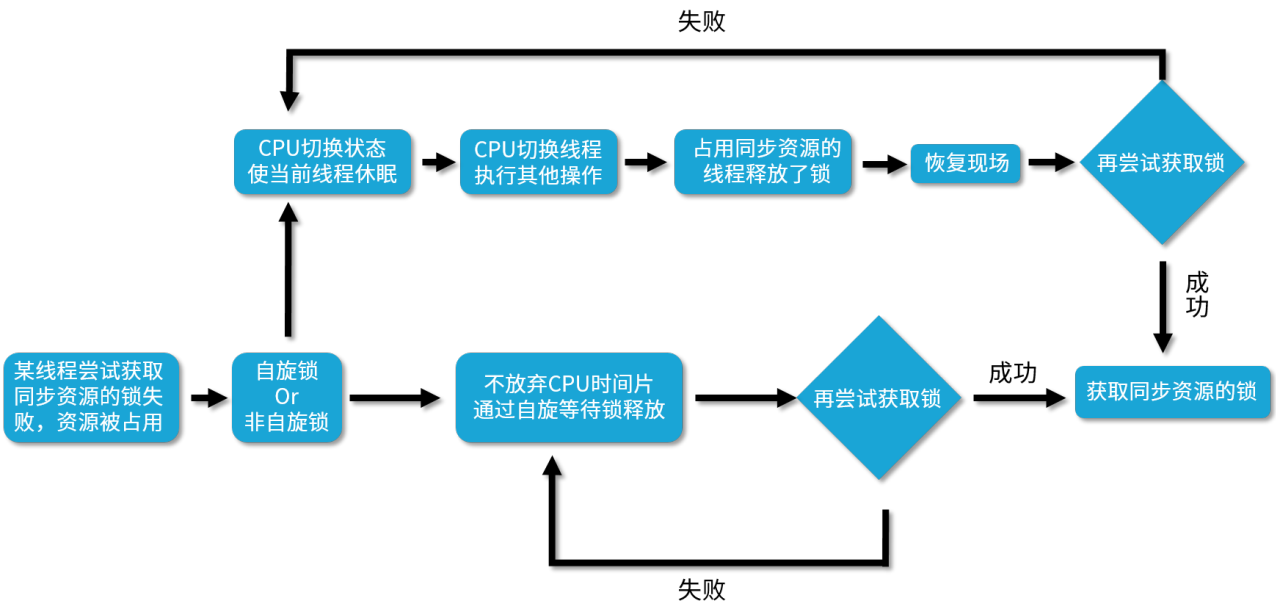
在本课时我们主要讲解什么是自旋锁？以及使用自旋锁的好处和后果分别是什么呢？

什么是自旋

首先，我们了解什么叫自旋？“自旋”可以理解为“自我旋转”，这里的“旋转”指“循环”，比如 while 循环或者 for 循环。“自旋”就是自己在这里不停地循环，直到目标达成。而不像普通的锁那样，如果获取不到锁就进入阻塞。

对比自旋和非自旋的获取锁的流程

下面我们用这样一张流程图来对比一下自旋锁和非自旋锁的获取锁的过程。



首先，我们来看自旋锁，它并不会放弃 CPU 时间片，而是通过自旋等待锁的释放，也就是说，它会不停地再次地尝试获取锁，如果失败就再次尝试，直到成功为止。

我们再来看下非自旋锁，非自旋锁和自旋锁是完全不一样的，如果它发现此时获取不到锁，它就把自己的线程切换状态，让线程休眠，然后 CPU 就可以在这段时间去做很多其他的事情，直到之前持有这把锁的线程释放了锁，于是 CPU 再把之前的线程恢复回来，让这个线

程再去尝试获取这把锁。如果再次失败，就再次让线程休眠，如果成功，一样可以成功获取到同步资源的锁。

可以看出，非自旋锁和自旋锁最大的区别，就是如果它遇到拿不到锁的情况，它会把线程阻塞，直到被唤醒。而自旋锁会不停地尝试。那么，自旋锁这样不停尝试的好处是什么呢？

自旋锁的好处

首先，阻塞和唤醒线程都是需要高昂的开销的，如果同步代码块中的内容不复杂，那么可能转换线程带来的开销比实际业务代码执行的开销还要大。

在很多场景下，可能我们的同步代码块的内容并不多，所以需要的执行时间也很短，如果我们仅仅为了这点时间就去切换线程状态，那么其实不如让线程不切换状态，而是让它自旋地尝试获取锁，等待其他线程释放锁，有时我只需要稍等一下，就可以避免上下文切换等开销，提高了效率。

用一句话总结自旋锁的好处，那就是自旋锁用循环去不停地尝试获取锁，让线程始终处于 Runnable 状态，节省了线程状态切换带来的开销。

AtomicLong 的实现

在 Java 1.5 版本及以上的并发包中，也就是 java.util.concurrent 的包中，里面的原子类基本都是自旋锁的实现。

比如我们看一个 AtomicLong 的实现，里面有一个 getAndIncrement 方法，源码如下：

```
public final long getAndIncrement() {
    return unsafe.getAndAddLong(this, valueOffset, 1L);
}
```

可以看到它调用了一个 unsafe.getAndAddLong，所以我们再来看这个方法：

```
public final long getAndAddLong (Object var1,long var2, long var4){
    long var6;
    do {
        var6 = this.getLongVolatile(var1, var2);
    } while (!this.compareAndSwapLong(var1, var2, var6, var6 + var4));
```

```
    return var6;
}
```

在这个方法中，它用了一个 do while 循环。这里就很明显了：

```
do {
    var6 = this.getLongVolatile(var1, var2);
}
while (!this.compareAndSwapLong(var1, var2, var6, var6 + var4));
```

这里的 do-while 循环就是一个自旋操作，如果在修改过程中遇到了其他线程竞争导致没修改成功的情况，就会 while 循环里进行死循环，直到修改成功为止。

自己实现一个可重入的自旋锁

下面我们来看一个自己实现可重入的自旋锁。

代码如下所示：

```
package lesson27;
import java.util.concurrent.atomic.AtomicReference;
import java.util.concurrent.locks.Lock;
/**
 * 描述：       实现一个可重入的自旋锁
 */
public class ReentrantSpinLock  {
    private AtomicReference<Thread> owner = new AtomicReference<>();
    //重入次数
    private int count = 0;
    public void lock() {
        Thread t = Thread.currentThread();
        if (t == owner.get()) {
```

```
            ++count;
            return;
        }
        //自旋获取锁
        while (!owner.compareAndSet(null, t)) {
            System.out.println("自旋了");
        }
    }
    public void unlock() {
        Thread t = Thread.currentThread();
        //只有持有锁的线程才能解锁
        if (t == owner.get()) {
            if (count > 0) {
                --count;
            } else {
                //此处无需CAS操作，因为没有竞争，因为只有线程持有者才能解锁
                owner.set(null);
            }
        }
    }
    public static void main(String[] args) {
        ReentrantSpinLock spinLock = new ReentrantSpinLock();
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName() + "开始尝试获取自
                spinLock.lock();
                try {
```

```
                    System.out.println(Thread.currentThread().getName() + "获取到了
                    Thread.sleep(4000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    spinLock.unlock();
                    System.out.println(Thread.currentThread().getName() + "释放了了
                }
            }
        };
        Thread thread1 = new Thread(runnable);
        Thread thread2 = new Thread(runnable);
        thread1.start();
        thread2.start();
    }
}
```

这段代码的运行结果是：

```
...
自旋了
自旋了
自旋了
自旋了
自旋了
自旋了
自旋了
自旋了
Thread-0释放了了自旋锁
```

```
Thread-1获取到了自旋锁
```

前面会打印出很多“自旋了”，说明自旋期间，CPU依然在不停运转。

缺点

那么自旋锁有没有缺点呢？其实自旋锁是有缺点的。它最大的缺点就在于虽然避免了线程切换的开销，但是它在避免线程切换开销的同时也带来了新的开销，因为它需要不停得去尝试获取锁。如果这把锁一直不能被释放，那么这种尝试只是无用的尝试，会白白浪费处理器资源。也就是说，虽然一开始自旋锁的开销低于线程切换，但是随着时间的增加，这种开销也是水涨船高，后期甚至会超过线程切换的开销，得不偿失。

适用场景

所以我们就要看一下自旋锁的适用场景。首先，自旋锁适用于并发度不是特别高的场景，以及临界区比较短小的情况，这样我们可以利用避免线程切换来提高效率。

可是如果临界区很大，线程一旦拿到锁，很久才会释放的话，那就不合适用自旋锁，因为自旋会一直占用 CPU 却无法拿到锁，白白消耗资源。

上一页 下一页