死磕Synchronized底层实现--重量级锁


往之farmer 关注
0.313 2018.12.06 18:33:18 字数 2,457 阅读 1,770

本文为死磕Synchronized底层实现第三篇文章，内容为重量级锁实现。

本系列文章将对HotSpot的 `synchronized` 锁实现进行全面分析，内容包括偏向锁、轻量级锁、重量级锁的加锁、解锁、锁升级流程的原理及源码分析，希望给在研究 `synchronized` 路上的同学一些帮助。主要包括以下几篇文章：

死磕Synchronized底层实现--概论

死磕Synchronized底层实现--偏向锁

死磕Synchronized底层实现--轻量级锁

死磕Synchronized底层实现--重量级锁

更多文章见个人博客：https://github.com/farmerjohngit/myblog

重量级的膨胀和加锁流程

当出现多个线程同时竞争锁时，会进入到 `synchronizer.cpp#slow_enter` 方法

```
void ObjectSynchronizer::slow_enter(Handle obj, BasicLock* lock, TRAPS) {
  markOop mark = obj->mark();
  assert(!mark->has_bias_pattern(), "should not see bias pattern here");
  // 如果是无锁状态
  if (mark->is_neutral()) {
    lock->set_displaced_header(mark);
    if (mark == (markOop) Atomic::cmpxchg_ptr(lock, obj()->mark_addr(), mark)) {
      TEVENT (slow_enter: release stacklock) ;
      return ;
    }
    // Fall through to inflate() ...
  } else
  // 如果是轻量级锁重入
  if (mark->has_locker() && THREAD->is_lock_owned((address)mark->locker())) {
    assert(lock != mark->locker(), "must not re-lock the same lock");
    assert(lock != (BasicLock*)obj->mark(), "don't relock with same BasicLock");
    lock->set_displaced_header(NULL);
    return;
  }

 ...


  // 这时候需要膨胀为重量级锁，膨胀前，设置Displaced Mark Word为一个特殊值，代表该锁正在用一个重量级锁
  lock->set_displaced_header(markOopDesc::unused_mark());
  //先调用inflate膨胀为重量级锁，该方法返回一个ObjectMonitor对象，然后调用其enter方法
  ObjectSynchronizer::inflate(THREAD, obj())->enter(THREAD);
}
```

在 `inflate` 中完成膨胀过程。

```
ObjectMonitor * ATTR ObjectSynchronizer::inflate (Thread * Self, oop object) {
  ...

  for (;;) {
      const markOop mark = object->mark() ;
```

推荐阅读

恐怖:这份Github神仙面试笔记,简直把所有Java知识面试题写出来了
阅读 30,865

阿里、华为、字节跳动，大厂面试算法题，这些你会吗?
阅读 14,266

InnoDB的索引
阅读 341

史上最全！2020面试阿里，字节跳动90%被问到的JVM面试题（附答案）
阅读 10,251

嵌套事务、挂起事务，Spring 是怎样给事务又实现传播特性的?
阅读 149

```
// *  Neutral（无锁状态）       - 膨胀
// *  BIASED（偏向锁）          - 非法状态，在这里不会出现

// CASE: inflated
if (mark->has_monitor()) {
    // 已经是重量级锁状态了，直接返回
    ObjectMonitor * inf = mark->monitor() ;
    ...
    return inf ;
}

// CASE: inflation in progress
if (mark == markOopDesc::INFLATING()) {
   // 正在膨胀中，说明另一个线程正在进行锁膨胀，continue重试
   TEVENT (Inflate: spin while INFLATING) ;
   // 在该方法中会进行spin/yield/park等操作完成自旋动作
   ReadStableMark(object) ;
   continue ;
}

if (mark->has_locker()) {
    // 当前轻量级锁状态，先分配一个ObjectMonitor对象，并初始化值
    ObjectMonitor * m = omAlloc (Self) ;

    m->Recycle();
    m->_Responsible  = NULL ;
    m->OwnerIsThread = 0 ;
    m->_recursions   = 0 ;
    m->_SpinDuration = ObjectMonitor::Knob_SpinLimit ;   // Consider: maintain by type/c
    // 将锁对象的mark word设置为INFLATING (0)状态
    markOop cmp = (markOop) Atomic::cmpxchg_ptr (markOopDesc::INFLATING(), object->mark_
    if (cmp != mark) {
       omRelease (Self, m, true) ;
       continue ;       // Interference -- just retry
    }

    // 栈中的displaced mark word
    markOop dmw = mark->displaced_mark_helper() ;
    assert (dmw->is_neutral(), "invariant") ;

    // 设置monitor的字段
    m->set_header(dmw) ;
    // owner为Lock Record
    m->set_owner(mark->locker());
    m->set_object(object);
    ...
    // 将锁对象头设置为重量级锁状态
    object->release_set_mark(markOopDesc::encode(m));

   ...
    return m ;
}

// CASE: neutral

// 分配以及初始化ObjectMonitor对象
ObjectMonitor * m = omAlloc (Self) ;
// prepare m for installation - set monitor to initial state
m->Recycle();
m->set_header(mark);
// owner为NULL
m->set_owner(NULL);
m->set_object(object);
m->OwnerIsThread = 1 ;
m->_recursions   = 0 ;
m->_Responsible  = NULL ;
m->_SpinDuration = ObjectMonitor::Knob_SpinLimit ;       // consider: keep metastats by
// 用CAS替换对象头的mark word为重量级锁状态
if (Atomic::cmpxchg_ptr (markOopDesc::encode(m), object->mark_addr(), mark) != mark) {
    // 不成功说明有另外一个线程在执行inflate，释放monitor对象
    m->set_object (NULL) ;
    m->set_owner  (NULL) ;
    m->OwnerIsThread = 0 ;
    m->Recycle() ;
    omRelease (Self, m, true) ;
    m = NULL ;
    continue ;
    // interference - the markword changed - just retry
```

```
        return m ;
    }
}
```

`inflate` 中是一个for循环，主要是为了处理多线程同时调用inflate的情况。然后会根据锁对象的状态进行不同的处理：

1.已经是重量级状态，说明膨胀已经完成，直接返回

2.如果是轻量级锁则需要进行膨胀操作

3.如果是膨胀中状态，则进行忙等待

4.如果是无锁状态则需要进行膨胀操作

其中轻量级锁和无锁状态需要进行膨胀操作，轻量级锁膨胀流程如下：

1.调用 `omAlloc` 分配一个 `ObjectMonitor` 对象(以下简称monitor)，在 `omAlloc` 方法中会先从线程私有的 `monitor` 集合 `omFreeList` 中分配对象，如果 `omFreeList` 中已经没有 `monitor` 对象，则从JVM全局的 `gFreeList` 中分配一批 `monitor` 到 `omFreeList` 中。

2.初始化 `monitor` 对象

3.将状态设置为膨胀中（INFLATING）状态

4.设置 `monitor` 的header字段为 `displaced mark word` ，owner字段为 `Lock Record` ，obj字段为锁对象

5.设置锁对象头的 `mark word` 为重量级锁状态，指向第一步分配的 `monitor` 对象

无锁状态下的膨胀流程如下：

1.调用 `omAlloc` 分配一个 `ObjectMonitor` 对象(以下简称monitor)

2.初始化 `monitor` 对象

3.设置 `monitor` 的header字段为 `mark word` ，owner字段为 `null` ，obj字段为锁对象

4.设置锁对象头的 `mark word` 为重量级锁状态，指向第一步分配的 `monitor` 对象

至于为什么轻量级锁需要一个膨胀中（INFLATING）状态，代码中的注释是：

```
// Why do we CAS a 0 into the mark-word instead of just CASing the
// mark-word from the stack-locked value directly to the new inflated state?
// Consider what happens when a thread unlocks a stack-locked object.
// It attempts to use CAS to swing the displaced header value from the
// on-stack basiclock back into the object header.  Recall also that the
// header value (hashcode, etc) can reside in (a) the object header, or
// (b) a displaced header associated with the stack-lock, or (c) a displaced
// header in an objectMonitor.  The inflate() routine must copy the header
// value from the basiclock on the owner's stack to the objectMonitor, all
// the while preserving the hashCode stability invariants.  If the owner
// decides to release the lock while the value is 0, the unlock will fail
// and control will eventually pass from slow_exit() to inflate.  The owner
// will then spin, waiting for the 0 value to disappear.   Put another way,
// the 0 causes the owner to stall if the owner happens to try to
// drop the lock (restoring the header from the basiclock to the object)
// while inflation is in-progress.  This protocol avoids races that might
// would otherwise permit hashCode values to change or "flicker" for an object.
// Critically, while object->mark is 0 mark->displaced_mark_helper() is stable.
// 0 serves as a "BUSY" inflate-in-progress indicator.
```

膨胀完成之后，会调用 `enter` 方法获得锁

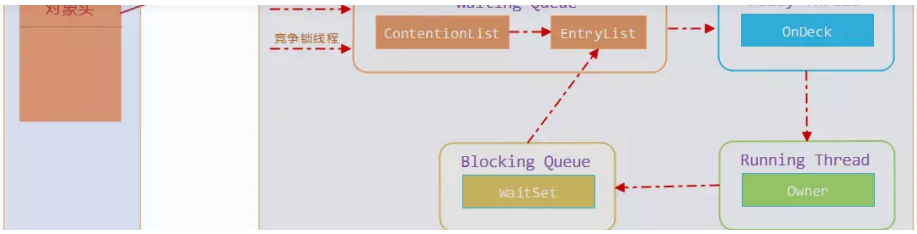
```
void ATTR ObjectMonitor::enter(TRAPS) {

  Thread * const Self = THREAD ;
  void * cur ;
  // owner为null代表无锁状态，如果能CAS设置成功，则当前线程直接获得锁
  cur = Atomic::cmpxchg_ptr (Self, &_owner, NULL) ;
  if (cur == NULL) {
     ...
     return ;
  }
  // 如果是重入的情况
  if (cur == Self) {
     // TODO-FIXME: check for integer overflow!  BUGID 6557169.
     _recursions ++ ;
     return ;
  }
  // 当前线程是之前持有轻量级锁的线程。由轻量级锁膨胀且第一次调用enter方法，那cur是指向Lock Record的指
  if (Self->is_lock_owned ((address)cur)) {
    assert (_recursions == 0, "internal state error");
    // 重入计数重置为1
    _recursions = 1 ;
    // 设置owner字段为当前线程（之前owner是指向Lock Record的指针）
    _owner = Self ;
    OwnerIsThread = 1 ;
    return ;
  }

  ...

  // 在调用系统的同步操作之前，先尝试自旋获得锁
  if (Knob_SpinEarly && TrySpin (Self) > 0) {
     ...
     //自旋的过程中获得了锁，则直接返回
     Self->_Stalled = 0 ;
     return ;
  }

  ...

  {
    ...

    for (;;) {
      jt->set_suspend_equivalent();
      // 在该方法中调用系统同步操作
      EnterI (THREAD) ;
      ...
    }
    Self->set_current_pending_monitor(NULL);

  }

  ...

}
```

1. 如果当前是无锁状态、锁重入、当前线程是之前持有轻量级锁的线程则进行简单操作后返回。
2. 先自旋尝试获得锁，这样做的目的是为了减少执行操作系统同步操作带来的开销
3. 调用 `EnterI` 方法获得锁或阻塞

`EnterI` 方法比较长，在看之前，我们先阐述下其大致原理：

一个 `ObjectMonitor` 对象包括这么几个关键字段：cxq（下图中的ContentionList），EntryList，WaitSet，owner。

其中cxq、EntryList、WaitSet都是由ObjectWaiter的链表结构，owner指向持有锁的线程



sync-d.png

当一个线程尝试获得锁时，如果该锁已经被占用，则会将该线程封装成一个 `ObjectWaiter` 对象插入到cxq的队列的队首，然后调用 `park` 函数挂起当前线程。在linux系统上， `park` 函数底层调用的是gclib库的 `pthread_cond_wait` ，JDK的 `ReentrantLock` 底层也是用该方法挂起线程的。更多细节可以看我之前的两篇文章：关于同步的一点思考-下，linux内核级同步机制--futex

当线程释放锁时，会从cxq或EntryList中挑选一个线程唤醒，被选中的线程叫做 `Heir presumptive` 即假定继承人（应该是这样翻译），就是图中的 `Ready Thread` ，假定继承人被唤醒后会尝试获得锁，但 `synchronized` 是非公平的，所以假定继承人不一定能获得锁（这也是它叫"假定"继承人的原因）。

如果线程获得锁后调用 `Object#wait` 方法，则会将线程加入到WaitSet中，当被 `Object#notify` 唤醒后，会将线程从WaitSet移动到cxq或EntryList中去。需要注意的是，当调用一个锁对象的 `wait` 或 `notify` 方法时，**如当前锁的状态是偏向锁或轻量级锁则会先膨胀成重量级锁。**

`synchronized` 的 `monitor` 锁机制和JDK的 `ReentrantLock` 与 `Condition` 是很相似的， `ReentrantLock` 也有一个存放等待获取锁线程的链表， `Condition` 也有一个类似 `WaitSet` 的集合用来存放调用了 `await` 的线程。如果你之前对 `ReentrantLock` 有深入了解，那理解起 `monitor` 应该是很简单。

回到代码上，开始分析 `EnterI` 方法：

```
void ATTR ObjectMonitor::EnterI (TRAPS) {
    Thread * Self = THREAD ;
    ...
    // 尝试获得锁
    if (TryLock (Self) > 0) {
        ...
        return ;
    }

    DeferredInitialize () ;

    // 自旋
    if (TrySpin (Self) > 0) {
        ...
        return ;
    }

    ...

    // 将线程封装成node节点中
    ObjectWaiter node(Self) ;
    Self->_ParkEvent->reset() ;
    node._prev   = (ObjectWaiter *) 0xBAD ;
    node.TState  = ObjectWaiter::TS_CXQ ;

    // 将node节点插入到_cxq队列的头部，cxq是一个单向链表
    ObjectWaiter * nxt ;
    for (;;) {
        node._next = nxt = _cxq ;
        if (Atomic::cmpxchg_ptr (&node, &_cxq, nxt) == nxt) break ;

        // CAS失败的话 再尝试获得锁，这样可以降低插入到_cxq队列的频率
        if (TryLock (Self) > 0) {
            ...
            return ;
```

```
    }


    TEVENT (Inflated enter - Contention) ;
    int nWakeups = 0 ;
    int RecheckInterval = 1 ;

    for (;;) {

        if (TryLock (Self) > 0) break ;
        assert (_owner != Self, "invariant") ;

        ...

        // park self
        if (_Responsible == Self || (SyncFlags & 1)) {
            // 当前线程是_Responsible时，调用的是带时间参数的park
            TEVENT (Inflated enter - park TIMED) ;
            Self->_ParkEvent->park ((jlong) RecheckInterval) ;
            // Increase the RecheckInterval, but clamp the value.
            RecheckInterval *= 8 ;
            if (RecheckInterval > 1000) RecheckInterval = 1000 ;
        } else {
            //否则直接调用park挂起当前线程
            TEVENT (Inflated enter - park UNTIMED) ;
            Self->_ParkEvent->park() ;
        }

        if (TryLock(Self) > 0) break ;

        ...

        if ((Knob_SpinAfterFutile & 1) && TrySpin (Self) > 0) break ;

        ...
        // 在释放锁时，_succ会被设置为EntryList或_cxq中的一个线程
        if (_succ == Self) _succ = NULL ;

        // Invariant: after clearing _succ a thread *must* retry _owner before parking.
        OrderAccess::fence() ;
    }

   // 走到这里说明已经获得锁了

    assert (_owner == Self      , "invariant") ;
    assert (object() != NULL    , "invariant") ;

    // 将当前线程的node从cxq或EntryList中移除
    UnlinkAfterAcquire (Self, &node) ;
    if (_succ == Self) _succ = NULL ;
    if (_Responsible == Self) {
        _Responsible = NULL ;
        OrderAccess::fence();
    }
    ...
    return ;
}

```

主要步骤有3步：

1. 将当前线程插入到cxq队列的队首
2. 然后park当前线程
3. 当被唤醒后再尝试获得锁

这里需要特别说明的是 `_Responsible` 和 `_succ` 两个字段的作用：

当竞争发生时，选取一个线程作为 `_Responsible` ， `_Responsible` 线程调用的是有时间限制的 `park` 方法，其目的是防止出现 `搁浅` 现象。

`succ` 线程是在线程释放锁是被设置，其含义是 `Heir presumptive` ，也就是我们上面说的假定继

重量级锁释放的代码在 `ObjectMonitor::exit`：

```
void ATTR ObjectMonitor::exit(bool not_suspended, TRAPS) {
   Thread * Self = THREAD ;
   // 如果_owner不是当前线程
   if (THREAD != _owner) {
     // 当前线程是之前持有轻量级锁的线程。由轻量级锁膨胀后还没调用过enter方法，_owner会是指向Lock Rec
     if (THREAD->is_lock_owned((address) _owner)) {
       assert (_recursions == 0, "invariant") ;
       _owner = THREAD ;
       _recursions = 0 ;
       OwnerIsThread = 1 ;
     } else {
       // 异常情况:当前不是持有锁的线程
       TEVENT (Exit - Throw IMSX) ;
       assert(false, "Non-balanced monitor enter/exit!");
       if (false) {
          THROW(vmSymbols::java_lang_IllegalMonitorStateException());
       }
       return;
     }
   }
   // 重入计数器还不为0，则计数器-1后返回
   if (_recursions != 0) {
     _recursions--;        // this is simple recursive enter
     TEVENT (Inflated exit - recursive) ;
     return ;
   }

   // _Responsible设置为null
   if ((SyncFlags & 4) == 0) {
      _Responsible = NULL ;
   }

   ...

   for (;;) {
      assert (THREAD == _owner, "invariant") ;

      // Knob_ExitPolicy默认为0
      if (Knob_ExitPolicy == 0) {
         // code 1: 先释放锁，这时如果有其他线程进入同步块则能获得锁
         OrderAccess::release_store_ptr (&_owner, NULL) ;   // drop the lock
         OrderAccess::storeload() ;                         // See if we need to wake a succes
         // code 2: 如果没有等待的线程或已经有假定继承人
         if ((intptr_t(_EntryList)|intptr_t(_cxq)) == 0 || _succ != NULL) {
            TEVENT (Inflated exit - simple egress) ;
            return ;
         }
         TEVENT (Inflated exit - complex egress) ;

         // code 3: 要执行之后的操作需要重新获得锁，即设置_owner为当前线程
         if (Atomic::cmpxchg_ptr (THREAD, &_owner, NULL) != NULL) {
            return ;
         }
         TEVENT (Exit - Reacquired) ;
      }
      ...

      ObjectWaiter * w = NULL ;
      // code 4: 根据QMode的不同会有不同的唤醒策略，默认为0
      int QMode = Knob_QMode ;

      if (QMode == 2 && _cxq != NULL) {
          // QMode == 2 : cxq中的线程有更高优先级，直接唤醒cxq的队首线程
          w = _cxq ;
          assert (w != NULL, "invariant") ;
          assert (w->TState == ObjectWaiter::TS_CXQ, "Invariant") ;
          ExitEpilog (Self, w) ;
          return ;
      }

      if (QMode == 3 && _cxq != NULL) {
          // 将cxq中的元素插入到EntryList的末尾
          w = _cxq ;
          for (;;) {
```

推荐阅读

恐怖:这份Github神仙面试笔记,简直把所有Java知识面试题写出来了
阅读 30,865

阿里、华为、字节跳动，大厂面试算法题，这些你会吗?
阅读 14,266

InnoDB的索引
阅读 341

史上最全！2020面试阿里，字节跳动90%被问到的JVM面试题（附答案）
阅读 10,251

嵌套事务、挂起事务，Spring 是怎样给事务又实现传播特性的?
阅读 149

```
            ObjectWaiter * q = NULL ;
            ObjectWaiter * p ;
            for (p = w ; p != NULL ; p = p->_next) {
                guarantee (p->TState == ObjectWaiter::TS_CXQ, "Invariant") ;
                p->TState = ObjectWaiter::TS_ENTER ;
                p->_prev = q ;
                q = p ;
            }

            // Append the RATs to the EntryList
            // TODO: organize EntryList as a CDLL so we can locate the tail in constant-time.
            ObjectWaiter * Tail ;
            for (Tail = _EntryList ; Tail != NULL && Tail->_next != NULL ; Tail = Tail->_next) ;
            if (Tail == NULL) {
                _EntryList = w ;
            } else {
                Tail->_next = w ;
                w->_prev = Tail ;
            }

            // Fall thru into code that tries to wake a successor from EntryList
        }

        if (QMode == 4 && _cxq != NULL) {
            // 将cxq插入到EntryList的队首
            w = _cxq ;
            for (;;) {
               assert (w != NULL, "Invariant") ;
               ObjectWaiter * u = (ObjectWaiter *) Atomic::cmpxchg_ptr (NULL, &_cxq, w) ;
               if (u == w) break ;
               w = u ;
            }
            assert (w != NULL              , "invariant") ;

            ObjectWaiter * q = NULL ;
            ObjectWaiter * p ;
            for (p = w ; p != NULL ; p = p->_next) {
                guarantee (p->TState == ObjectWaiter::TS_CXQ, "Invariant") ;
                p->TState = ObjectWaiter::TS_ENTER ;
                p->_prev = q ;
                q = p ;
            }

            // Prepend the RATs to the EntryList
            if (_EntryList != NULL) {
                q->_next = _EntryList ;
                _EntryList->_prev = q ;
            }
            _EntryList = w ;

            // Fall thru into code that tries to wake a successor from EntryList
        }

        w = _EntryList  ;
        if (w != NULL) {
            // 如果EntryList不为空，则直接唤醒EntryList的队首元素
            assert (w->TState == ObjectWaiter::TS_ENTER, "invariant") ;
            ExitEpilog (Self, w) ;
            return ;
        }

        // EntryList为null，则处理cxq中的元素
        w = _cxq ;
        if (w == NULL) continue ;

        // 因为之后要将cxq的元素移动到EntryList，所以这里将cxq字段设置为null
        for (;;) {
            assert (w != NULL, "Invariant") ;
            ObjectWaiter * u = (ObjectWaiter *) Atomic::cmpxchg_ptr (NULL, &_cxq, w) ;
            if (u == w) break ;
            w = u ;
        }
        TEVENT (Inflated exit - drain cxq into EntryList) ;

        assert (w != NULL              , "invariant") ;
        assert (_EntryList  == NULL    , "invariant") ;
```

```
            ObjectWaiter * u = NULL ;
            while (t != NULL) {
                guarantee (t->TState == ObjectWaiter::TS_CXQ, "invariant") ;
                t->TState = ObjectWaiter::TS_ENTER ;
                u = t->_next ;
                t->_prev = u ;
                t->_next = s ;
                s = t;
                t = u ;
            }
            _EntryList  = s ;
            assert (s != NULL, "invariant") ;
        } else {
            // QMode == 0 or QMode == 2‘
            // 将cxq中的元素转移到EntryList
            _EntryList = w ;
            ObjectWaiter * q = NULL ;
            ObjectWaiter * p ;
            for (p = w ; p != NULL ; p = p->_next) {
                guarantee (p->TState == ObjectWaiter::TS_CXQ, "Invariant") ;
                p->TState = ObjectWaiter::TS_ENTER ;
                p->_prev = q ;
                q = p ;
            }
        }


        // _succ不为null，说明已经有个继承人了，所以不需要当前线程去唤醒，减少上下文切换的比率
        if (_succ != NULL) continue;

        w = _EntryList  ;
        // 唤醒EntryList第一个元素
        if (w != NULL) {
            guarantee (w->TState == ObjectWaiter::TS_ENTER, "invariant") ;
            ExitEpilog (Self, w) ;
            return ;
        }
    }
}
```

在进行必要的锁重入判断以及自旋优化后，进入到主要逻辑：

code 1 设置owner为null，即释放锁，这个时刻其他的线程能获取到锁。这里是一个非公平锁的优化；

code 2 如果当前没有等待的线程则直接返回就好了，因为不需要唤醒其他线程。或者如果说succ不为null，代表当前已经有个"醒着的"继承人线程，那当前线程不需要唤醒任何线程；

code 3 当前线程重新获得锁，因为之后要操作cxq和EntryList队列以及唤醒线程；

code 4 根据QMode的不同，会执行不同的唤醒策略；

根据QMode的不同，有不同的处理方式：

1. QMode = 2且cxq非空：取cxq队列队首的ObjectWaiter对象，调用ExitEpilog方法，该方法会唤醒ObjectWaiter对象的线程，然后立即返回，后面的代码不会执行了；
2. QMode = 3且cxq非空：把cxq队列插入到EntryList的尾部；
3. QMode = 4且cxq非空：把cxq队列插入到EntryList的头部；
4. QMode = 0：暂时什么都不做，继续往下看；

只有QMode=2的时候会提前返回，等于0、3、4的时候都会继续往下执行：

1.如果EntryList的首元素非空，就取出来调用ExitEpilog方法，该方法会唤醒ObjectWaiter对象的线程，然后立即返回；
2.如果EntryList的首元素为空，就将cxq的所有元素放入到EntryList中，然后再从EntryList中取出

QMode默认为0，结合上面的流程我们可以看这么个demo：

```
public class SyncDemo {

    public static void main(String[] args) {

        SyncDemo syncDemo1 = new SyncDemo();
        syncDemo1.startThreadA();
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        syncDemo1.startThreadB();
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        syncDemo1.startThreadC();


    }

    final Object lock = new Object();


    public void startThreadA() {
        new Thread(() -> {
            synchronized (lock) {
                System.out.println("A get lock");
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("A release lock");
            }
        }, "thread-A").start();
    }

    public void startThreadB() {
        new Thread(() -> {
            synchronized (lock) {
                System.out.println("B get lock");
            }
        }, "thread-B").start();
    }

    public void startThreadC() {
        new Thread(() -> {
            synchronized (lock) {

                System.out.println("C get lock");
            }
        }, "thread-C").start();
    }


}
```

默认策略下，在A释放锁后一定是C线程先获得锁。因为在获取锁时，是将当前线程插入到cxq的**头部**，而释放锁时，默认策略是：如果EntryList为空，则将cxq中的元素按原有顺序插入到到EntryList，并唤醒第一个线程。也就是**当EntryList为空时，是后来的线程先获取锁。**这点JDK中的Lock机制是不一样的。

Synchronized和ReentrantLock的区别

原理弄清楚了，顺便总结了几点Synchronized和ReentrantLock的区别：

3. Synchronized是非公平锁，ReentrantLock是可以是公平也可以是非公平的；
4. Synchronized是不可以被中断的，而 `ReentrantLock#lockInterruptibly` 方法是可以被中断的；
5. 在发生异常时Synchronized会自动释放锁（由javac编译时自动实现），而ReentrantLock需要开发者在finally块中显示释放锁；
6. ReentrantLock获取锁的形式有多种：如立即返回是否成功的tryLock(),以及等待指定时长的获取，更加灵活；
7. Synchronized在特定的情况下**对于已经在等待的线程**是后来的线程先获得锁（上文有说），而ReentrantLock对于**已经在等待的线程**一定是先来的线程先获得锁；

End

总的来说Synchronized的重量级锁和ReentrantLock的实现上还是有很多相似的，包括其数据结构、挂起线程方式等等。在日常使用中，如无特殊要求用Synchronized就够了。你深入了解这两者其中一个的实现，了解另外一个或其他锁机制都比较容易，这也是我们常说的技术上的相通性。

往之farmer
总资产17 (约1.65元) 共写了4.8W字 获得163个赞 共150个粉丝

DeNaAt
4楼 07.21 17:03

老哥,关于"轻量级锁需要一个膨胀中（INFLATING）状态",结合你的文章,,我稍微翻译了下(本人能力有限,水平一般).自己也总结了一点东西..可以一起看下.
"inflate()方法必须将header value从轻量级锁持有者堆栈上的basiclock复制到objectMonitor，同时保证hashCode的稳定不变。
如果持有者决定在值为0时释放锁，解锁将失败，将从轻量级锁的slow_exit()传递到inflate()中。
然后持有者将进入[if (mark == markOopDesc::INFLATING()]自旋，等待0值消失。
换句话说，
如果持有者在inflate()过程中尝试解锁（将头从basiclock恢复到对象的mark word），则0将使持有者暂停。
此规定避免了HashCode值更改或对象"闪烁"的争用(对象闪烁征用..是啥意思没查到.)。
关键的是，当object->mark为0时，displaced_mark_helper()是稳定的[不会出现锁释

推荐阅读

恐怖:这份Github神仙面试笔记,简直把所有Java知识面试题写出来了
阅读 30,865

阿里、华为、字节跳动，大厂面试算法题，这些你会吗?
阅读 14,266

InnoDB的索引
阅读 341

史上最全！2020面试阿里，字节跳动90%被问到的JVM面试题（附答案）
阅读 10,251

嵌套事务、挂起事务，Spring 是怎样给事务又实现传播特性的?
阅读 149