Synchronizer In Hotspot (Base OpenJDK 8)

feihui 关注

0.56 2019.10.27 11:30:50 字数 3,208 阅读 11,093

> 同步是程序中一个永恒的话题，无论我们怎么避免，也总是不能完全绕开，JVM 本身提供了同步原语关键字 -- synchronized (对应字节码：monitorenter 和 monitorexit)，在 Hotspot 早期实现中 (1.6 之前)，其执行效率非常低， 1.6 的时候进行优化极大地提升了性能，那么我们今天就来看看其中的奥秘。

我们首先来看下 monitorenter 主要逻辑：

```
IRT_ENTRY_NO_ASYNC(void, InterpreterRuntime::monitorenter(JavaThread* thread, BasicObjectLock* elem))
  ...... //
  if (UseBiasedLocking) {
    // Retry fast entry if bias is revoked to avoid unnecessary inflation
    ObjectSynchronizer::fast_enter(h_obj, elem->lock(), true, CHECK);
  } else {
    ObjectSynchronizer::slow_enter(h_obj, elem->lock(), CHECK);
  }
  ...... //
IRT_END
```

不难看出，Hotspot 对同步原语的实现是通过 lock 来实现的，那这些 lock 又是什么呢？

> 接下来的内容将以《Evaluating and improving biased locking in the HotSpot virtual machine》这篇非常有名的论文为依据，并结合源码进行表述。

相信大家或多或少都知道，JAVA 对象内存布局包含三部分 -- Mark Word / Metadata Ref / Actual Data，而其中的 Mark Word 最为特殊，如下三图如所示：

bitfields			tag bits	state
hash	age	0	01	unlocked
ptr to lock record			00	lightweight locked
ptr to heavyweight monitor			10	inflated
			11	marked for GC

pasted-image.png

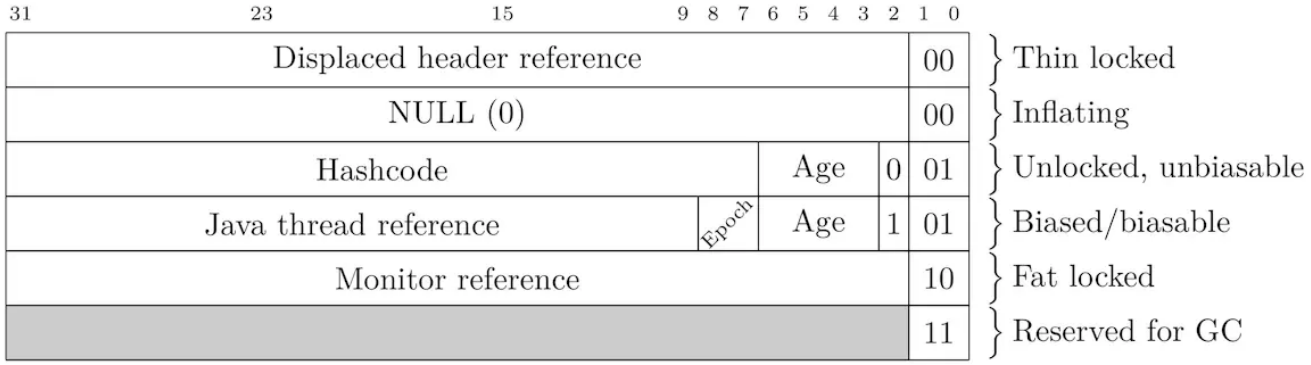


Figure 3.2. The different states and layouts of the 32-bit mark word in HotSpot.

pasted-image.png

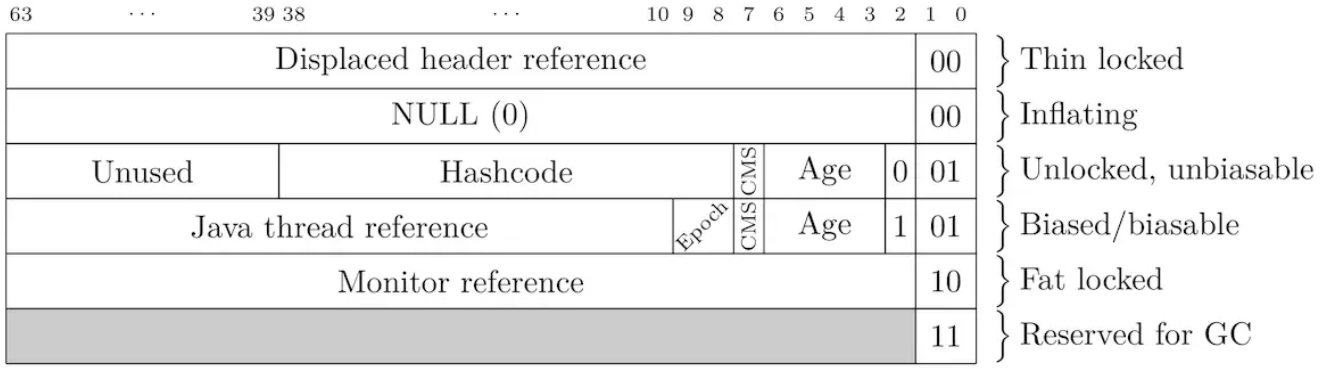


Figure 3.1. The different states and layouts of the 64-bit mark word in HotSpot.

pasted-image.png

从上面看的锁状态位，我们可以看到，在 Hotstpot 中一共有三种类型的锁：

```
Biased Lock // 使用在大部分情况下只有同一个线程持有锁
Thin Lock // 使用在大部分情况下只有一个线程持有锁
```

```
    Fat Lock // else
    // 关于自旋锁，我个人更倾向于归其为一种 Lock 策略而非一种 Lock 实现。
```

接下来会将从结构 / 获取 / 竞争 / 场景 这几方面来介绍这三种锁。

偏向锁

结构

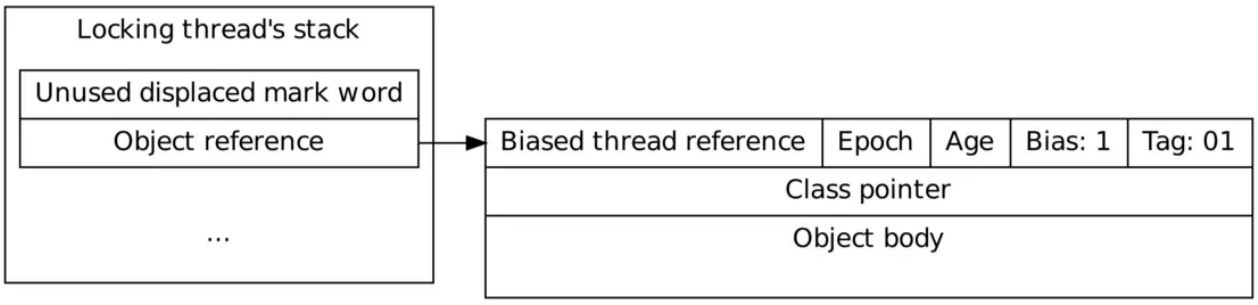


Figure 3.5. Example of a bias-locked object.

pasted-image.png

获取

1. Test the bias bit of the object

   If it is 0, the lock is not biased, and the thin locking algorithm should be used.

2. Test the bias bit of the object's class

   Check that the bias bit is set in the prototype mark word for the object's class. If it is not, biasing is globally disallowed on objects of this class and the bias bit of the object should be reset and thin locking used instead.

3. Verify the epoch

   Check that the epoch in the object's mark word matches that of the prototype mark word. If not, the bias has expired and the lock is rebiasable. In this case the locking thread can simply try to rebias the lock with an atomic CAS.

4. Check owner

   Compare the biased thread identifier to the locking thread's identifier. If they match, the locking thread is currently holding the lock and can safely return. If they do not match, the lock is assumed to be anonymously biased and the locking thread should try to acquire the bias with an atomic CAS. On failure, revoke the bias (possibly involving a safepoint) and fall back to thin locking. On success, the locking thread is the biased owner of the lock and can return.

```
BiasedLocking::Condition BiasedLocking::revoke_and_rebias(Handle obj, bool attempt_rebias, TRAPS) {
  assert(!SafepointSynchronize::is_at_safepoint(), "must not be called while at safepoint");

  // We can revoke the biases of anonymously-biased objects
  // efficiently enough that we should not cause these revocations to
  // update the heuristics because doing so may cause unwanted bulk
  // revocations (which are expensive) to occur.
  markOop mark = obj->mark();
  if (mark->is_biased_anonymously() && !attempt_rebias) {
    // We are probably trying to revoke the bias of this object due to
    // an identity hash code computation. Try to revoke the bias
    // without a safepoint. This is possible if we can successfully
    // compare-and-exchange an unbiased header into the mark word of
    // the object, meaning that no other thread has raced to acquire
    // the bias of the object.
    markOop biased_value       = mark;
    markOop unbiased_prototype = markOopDesc::prototype()->set_age(mark->age());
    markOop res_mark = (markOop) Atomic::cmpxchg_ptr(unbiased_prototype, obj->mark_addr(), mark);
    if (res_mark == biased_value) {
      return BIAS_REVOKED;
    }
  } else if (mark->has_bias_pattern()) {
    Klass* k = obj->klass();
    markOop prototype_header = k->prototype_header();
    if (!prototype_header->has_bias_pattern()) {
      // This object has a stale bias from before the bulk revocation
      // for this data type occurred. It's pointless to update the
      // heuristics at this point so simply update the header with a
      // CAS. If we fail this race, the object's bias has been revoked
      // by another thread so we simply return and let the caller deal
      // with it.
      markOop biased_value       = mark;
      markOop res_mark = (markOop) Atomic::cmpxchg_ptr(prototype_header, obj->mark_addr(), mark);
      assert(!(*(obj->mark_addr()))->has_bias_pattern(), "even if we raced, should still be revoked");
      return BIAS_REVOKED;
    } else if (prototype_header->bias_epoch() != mark->bias_epoch()) {
      // The epoch of this biasing has expired indicating that the
      // object is effectively unbiased. Depending on whether we need
      // to rebias or revoke the bias of this object we can do it
      // efficiently enough with a CAS that we shouldn't update the
      // heuristics. This is normally done in the assembly code but we
      // can reach this point due to various points in the runtime
```

```
        // needing to revoke biases.
        if (attempt_rebias) {
          assert(THREAD->is_Java_thread(), "");
          markOop biased_value       = mark;
          markOop rebiased_prototype = markOopDesc::encode((JavaThread*) THREAD, mark->age(), prototype_h
          markOop res_mark = (markOop) Atomic::cmpxchg_ptr(rebiased_prototype, obj->mark_addr(), mark);
          if (res_mark == biased_value) {
            return BIAS_REVOKED_AND_REBIASED;
          }
        } else {
          markOop biased_value       = mark;
          markOop unbiased_prototype = markOopDesc::prototype()->set_age(mark->age());
          markOop res_mark = (markOop) Atomic::cmpxchg_ptr(unbiased_prototype, obj->mark_addr(), mark);
          if (res_mark == biased_value) {
            return BIAS_REVOKED;
          }
        }
      }
    }
    `````` //
}
```

```
class Klass : public Metadata {
  ...... //
  // Used when biased locking is both enabled and disabled for this type
  markOop  _prototype_header;
  ...... //
}
```

关于在 2 和 3 中提到的，object 对应的 class 上同样也有 mark work，用来在 class 范围上开启/禁止偏向以及偏向有效校验 (epoch)。

偏向锁一个最大的不同之处在于，持有锁的偏向线程在推出同步区域后不会释放。那么你就会疑惑，不释放锁那岂不是导致其他线程都不能获取锁了(我们平时使用锁的习惯第一直觉)？我们来看下 Hotspot 中的如何处理这个问题的？

竞争

> When a biased lock is acquired by a different thread than that of the biased thread, the bias must be revoked. Since the biased thread might acquire or release the lock at any time, and because it has no obligations to use atomic instructions when doing so, the biased thread must be suspended in order to safely make modifications to the lock. Once the biased thread is stopped, the lock can safely be revoked, possibly involving walking the owning thread's stack to
```

> find and modify all of its lock records. Alternatively, if the biased thread no longer exists, the lock can simply be revoked or rebiased directly. If the revoker finds that the lock is held, it converts the biased lock into a thin lock by modifying the lock field and the corresponding lock records to look as if the thin locking mechanism had been used from the start. If the lock is found not held by the bias owner, there is an alternative for the revoker to simply rebias the lock to itself. This effectively transfers the bias to the new thread, removing the previous bias of the lock. Such rebiasing is good for producer-consumer patterns, allowing the locks to be handed over from thread to thread and continue exploiting thread locality.
> Revocation is a costly operation because it requires thread suspension and possibly stack walking to find and modify lock records. In comparison to the thin locking mechanism, or even the heavy-weight monitor mechanisms, the revocation process is much slower.

由于持有偏向锁的线程在退出同步区域后并不会释放锁 -- 即锁对象上的仍偏向该线程，当发生竞争的时候，需要判断偏向线程是否已经退出同步区域（这个判断需暂停偏向线程），若退出，则简单 rebias（重新偏向）即可，否则需要进行锁升级（同时还需要修改偏向线程栈上的所有锁记录）。

> To mitigate the cost of revocations, a technique involving bulk rebias and revocation has been developed. This basically allows the JVM to rebias or revoke not just the bias on a certain object's lock, but on all locks on objects for a specific data type. Different classes will typically have different use cases, and some objects are more likely to be shared between threads than others. The use pattern of certain data types can prove to be problematic for biased locking schemes. Producer- consumer queues are typically always contended, for example. In these scenarios it is beneficial to detect the pattern at a data type level, revoking or rebiasing all objects for that type in a single pass, rather than revoking each individual object when its thread ownership changes. In other words, heuristics are added to the biased locking algorithm, estimating the cost of bias revocations per data type and rebiasing all locks of that data type when a certain threshold is reached. This bulk rebias operation resets the biased thread for each unlocked object of the given data type, allowing the next acquiring thread to bias the lock to itself without the need for revocation. During this operation locked objects can either be revoked, or left biased. The heuristics might also decide to perform a so called bulk revocation operation, completely forbidding biased locking to be used on all objects of the class. Implementing this scheme in the naive way, one would be required to scan the whole heap for

> objects in order to rebias unlocked objects. A perhaps better way to implement bulk rebias operations is to introduce an epoch field for objects. Adding this field to the class metadata, and also for each object (as part of the lock field for example), the bias of a given lock is only valid if the epoch numbers in the class metadata and object match. The invalidation then simply consists of increasing the epoch field of the class, iterating over all threads to update the epoch field of currently locked objects as well. Objects that were not updated will then have an invalid epoch number, and will thus be rebiasable. The epoch field is limited in size. There are occasions when this field wraps around, possibly causing some invalidated biases to become valid again. This is luckily not a problem for correctness, it is only a performance issue as locks are unintentionally rebiased. In the worst case this causes revocations. Experiments by Detlefs and Russell indicate the impact on performance due to epoch wrapping is negligible. Also, as a countermeasure the GC can normalize the lock field to the initial unbiased but rebiasable state when encountering objects that have an invalid epoch, further reducing the impact of epoch wrapping.
>
> To allow bulk revocation for classes, one can similarly add a field to the class metadata specifying if biasing is allowed or not. When acquiring a lock, this field will have to be checked to determine whether to use biased locking or just thin locking. If a prototype object header is kept in the class metadata, biasing can be disabled in the lock field of this prototype, and new objects of that class will retain the class biasing policy. Naturally, when bulk revocation occurs all currently biased locked objects of the class must be revoked, after which the bias bit in the prototype can be set to 0, implicitly revoking the remaining objects of the class.

同时为了减少频繁竞争带来的性能损耗 (毕竟要 suspend thread)，引入批量操作 -- 当竞争次数达到某个阈值是，针对某个 class 的所有 object 执行 rebias 或者 revoke，但这样的批量操作有个问题就是，你需要遍历堆中该 class 所有的 object，然后 suspend 相关的 thread，但这显然是不合适的 (这种遍历根本就跟堆大小成线性正相关)，论文中也提到另一种可能的方式就是只更新当前所有线程持有的 lock，并 epoch + 1 来表示更新状态(这就是为什么在获取偏向锁是会有第三步判断，可能你会认为，epoch 只用两位表示，很容易发生溢位，但发生溢位是也只是让无效的偏向有效，重新 suspend 偏向线程判断即可，论文中也说明其带来的性能损耗在可接受范围内)，这种方式下遍历只跟虚拟机栈大小线程正相关。

```
BiasedLocking::Condition BiasedLocking::revoke_and_rebias(Handle obj, bool attempt_rebias, TRAPS) {
  ...... //
  if (heuristics == HR_NOT_BIASED) {
```

```
      return NOT_BIASED;
    } else if (heuristics == HR_SINGLE_REVOKE) {
      Klass *k = obj->klass();
      markOop prototype_header = k->prototype_header();
      if (mark->biased_locker() == THREAD &&
          prototype_header->bias_epoch() == mark->bias_epoch()) {
        // A thread is trying to revoke the bias of an object biased
        // toward it, again likely due to an identity hash code
        // computation. We can again avoid a safepoint in this case
        // since we are only going to walk our own stack. There are no
        // races with revocations occurring in other threads because we
        // reach no safepoints in the revocation path.
        // Also check the epoch because even if threads match, another thread
        // can come in with a CAS to steal the bias of an object that has a
        // stale epoch.
        ResourceMark rm;
        if (TraceBiasedLocking) {
          tty->print_cr("Revoking bias by walking my own stack:");
        }
        BiasedLocking::Condition cond = revoke_bias(obj(), false, false, (JavaThread*) THREAD);
        ((JavaThread*) THREAD)->set_cached_monitor_info(NULL);
        assert(cond == BIAS_REVOKED, "why not?");
        return cond;
      } else {
        VM_RevokeBias revoke(&obj, (JavaThread*) THREAD);
        VMThread::execute(&revoke);
        return revoke.status_code();
      }
    }

    assert((heuristics == HR_BULK_REVOKE) ||
           (heuristics == HR_BULK_REBIAS), "?");
    VM_BulkRevokeBias bulk_revoke(&obj, (JavaThread*) THREAD,
                                  (heuristics == HR_BULK_REBIAS),
                                  attempt_rebias);
    VMThread::execute(&bulk_revoke);
    return bulk_revoke.status_code();
  }
```

> The metadata for each class includes a counter that is incremented each time a biased lock is revoked for an object of the given class. Also a timestamp is used to indicate the last time a bulk rebias operation was performed on the class. The revocation count does not include revocations on objects that are anonymously biased or rebiasable. These revocations are very cheap, requiring only a single CAS, and should therefore not cause bulk operations on the class. The current heuristics has two thresholds for the revocation counter: the bulk rebias threshold and the bulk revocation threshold. Initially, the heuristics will choose to revoke or rebias locks

> individually. Once the bulk rebias threshold is reached the bulk rebias operation is performed, revoking all biases and leaving the locks in a rebiasable state.
> A time threshold (decay time) is used to reset the revocation counter if enough time has passed since the last bulk rebias operation. This means that there has not been many revocations since the last bulk rebias operation was performed, which suggests that biasing could still be beneficial for the class. Typically this is caused by a scenario where one thread has been working on objects of a certain class and then handed them all over to other threads. This behaviour can be fine, as the epoch based bulk rebias operation should be cheap enough to allow the repeated rebiasing if it does not occur too often.
> If revocations keep occurring after a bulk rebias operation, within the decay time threshold, the counter will keep incrementing. Once the bulk revocation threshold is reached, a bulk revocation is performed and biased locking is no longer allowed on objects of the class. The heuristics will at this point stop tracking revocation count and decay time.

如之前描述，为了适时进行批量操作(包括重新偏向和禁止偏向)，需要记录统计竞争数据，这些数据就存在的 class 中：

```
class Klass : public Metadata {
  // Biased locking implementation and statistics
  // (the 64-bit chunk goes first, to avoid some fragmentation)
  jlong    _last_biased_lock_bulk_revocation_time;
  // Used when biased locking is both enabled and disabled for this type
  markOop  _prototype_header;
  jint     _biased_lock_revocation_count;
}

static HeuristicsResult update_heuristics(oop o, bool allow_rebias) {
  markOop mark = o->mark();
  if (!mark->has_bias_pattern()) {
    return HR_NOT_BIASED;
  }

  // Heuristics to attempt to throttle the number of revocations.
  // Stages:
  // 1. Revoke the biases of all objects in the heap of this type,
  //    but allow rebiasing of those objects if unlocked.
  // 2. Revoke the biases of all objects in the heap of this type
  //    and don't allow rebiasing of these objects. Disable
  //    allocation of objects of that type with the bias bit set.
  Klass* k = o->klass();
  jlong cur_time = os::javaTimeMillis();
  jlong last_bulk_revocation_time = k->last_biased_lock_bulk_revocation_time();
```

```
    int revocation_count = k->biased_lock_revocation_count();
    if ((revocation_count >= BiasedLockingBulkRebiasThreshold) &&
        (revocation_count <  BiasedLockingBulkRevokeThreshold) &&
        (last_bulk_revocation_time != 0) &&
        (cur_time - last_bulk_revocation_time >= BiasedLockingDecayTime)) {
      // This is the first revocation we've seen in a while of an
      // object of this type since the last time we performed a bulk
      // rebiasing operation. The application is allocating objects in
      // bulk which are biased toward a thread and then handing them
      // off to another thread. We can cope with this allocation
      // pattern via the bulk rebiasing mechanism so we reset the
      // klass's revocation count rather than allow it to increase
      // monotonically. If we see the need to perform another bulk
      // rebias operation later, we will, and if subsequently we see
      // many more revocation operations in a short period of time we
      // will completely disable biasing for this type.
      k->set_biased_lock_revocation_count(0);
      revocation_count = 0;
    }

    // Make revocation count saturate just beyond BiasedLockingBulkRevokeThreshold
    if (revocation_count <= BiasedLockingBulkRevokeThreshold) {
      revocation_count = k->atomic_incr_biased_lock_revocation_count();
    }

    if (revocation_count == BiasedLockingBulkRevokeThreshold) {
      return HR_BULK_REVOKE;
    }

    if (revocation_count == BiasedLockingBulkRebiasThreshold) {
      return HR_BULK_REBIAS;
    }

    return HR_SINGLE_REVOKE;
  }
```

Table 3.1. Current heuristics threshold values in HotSpot.

Threshold	Value
Startup delay	4s
Bulk rebias threshold	20
Bulk revoke threshold	40
Decay time	25s

pasted-image.png

实验数据表示，在 JVM 启动前几秒 (startup delay) 使用偏向锁并没有带来好的性能；而倘若一段时间(decay time)都没有发生竞争，则说明可以竞争并不激烈，可以重新 reset threshold。

场景

从上面的描述，其实我们也可以看到，由于偏向锁在竞争处理上需要 suspend thread，批量操作还会 STW，因此不应适合竞争激烈的场景，但当大部分时候只有同一个线程持有锁的时候，性能确实非常高效。尽管 Hotspot 提供统计数据用于偏向的调整，但往往应用场景也是变幻莫测，这个时候为了稳定我们也可以通过启动参数关闭偏向锁。

轻量级锁

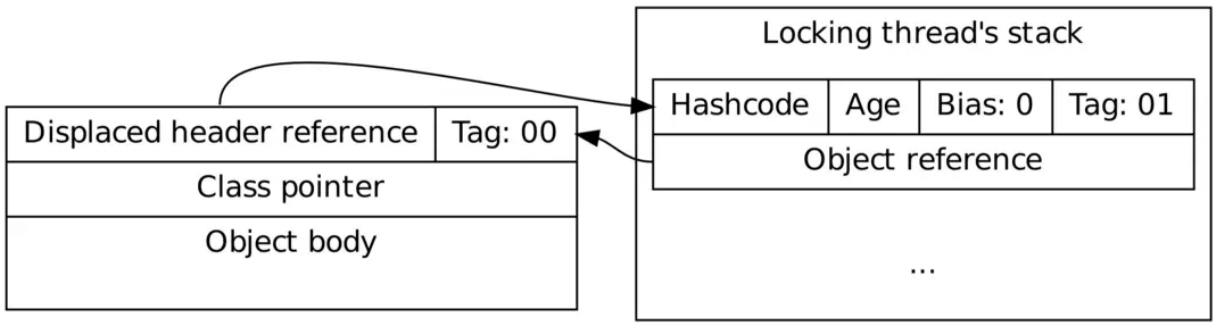


Figure 3.3. Example of a thin locked object.

pasted-image.png

相对于偏向锁，轻量级锁没有那么获取流程，通过一个简单的 CAS 即可，失败进行锁膨胀。由于持有锁的线程在退出同步区域的时候会释放锁(就不需要像看向锁那样 suspend thread 进行逻辑处理)，因此非常适合在大部分时间只有一个线程持有锁的场景。

> When a thread tries to acquire a thin lock already held by some other thread (the CAS fails), the non-owner thread will attempt to inflate the lock. During inflation the heavyweight object

> monitor structure is allocated and initialized for the object. The inflating thread will attempt to inflate the lock in a loop until it is successful, or until some other thread has successfully inflated the lock. Inflation also occurs if wait or notify is called, even on objects that are not locked.
> In order to inflate a lock that is thin locked, the inflating thread will first CAS the mark word to INFLATING (0), after which it will read the displaced mark word and copy it to the new object monitor structure. Threads that encounter an INFLATING mark word will wait for the inflating thread to complete the inflation action. This includes the thread that currently holds the lock, making it unable to release the lock until the inflation is complete. The use of this temporary inflation state is necessary, because otherwise the inflating thread might read an outdated version of the mark word. By putting the lock in the inflating state, the inflating thread guarantees that the displaced mark word cannot change, and it is safe to read and copy it.
> Depending on the state of the lock in each iteration of the inflation loop, the following actions are taken:

1. The lock is already inflated (tag is 10)
   Some other thread successfully inflated the lock. Break out of the loop.
2. The lock is being inflated (mark word is INFLATING)
   Some other thread is currently inflating the lock. Wait until inflated.
3. The lock is thin locked (tag is 00)
   Allocate an ObjectMonitor, then CAS INFLATING to the mark word. If the CAS fails, deallocate the monitor and retry the inflation loop. If the CAS succeeds, install the monitor by setting up the appropriate fields, copying the displaced mark word to the object monitor, and finally store the monitor pointer in the mark word (replacing the INFLATING).
4. The lock is unlocked (tag is 01)
   Allocate an ObjectMonitor, set it up, and try to CAS its reference into the mark word. On failure, deallocate the monitor and retry the inflation loop, otherwise break out of the loop. Once the lock is inflated any acquiring threads will simply use the underlying monitor mechanism to acquire the lock.

```
void ObjectSynchronizer::slow_enter(Handle obj, BasicLock* lock, TRAPS) {
  markOop mark = obj->mark();
  assert(!mark->has_bias_pattern(), "should not see bias pattern here");

  if (mark->is_neutral()) {
```

```
    // Anticipate successful CAS -- the ST of the displaced mark must
    // be visible <= the ST performed by the CAS.
    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
  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;
  }

#if 0
  // The following optimization isn't particularly useful.
  if (mark->has_monitor() && mark->monitor()->is_entered(THREAD)) {
    lock->set_displaced_header (NULL) ;
    return ;
  }
#endif

  // The object header will never be displaced to this lock,
  // so it does not matter what the value is, except that it
  // must be non-zero to avoid looking like a re-entrant lock,
  // and must not look locked either.
  lock->set_displaced_header(markOopDesc::unused_mark());
  ObjectSynchronizer::inflate(THREAD, obj())->enter(THREAD);
}

ObjectMonitor * ATTR ObjectSynchronizer::inflate (Thread * Self, oop object) {
  // Inflate mutates the heap ...
  // Relaxing assertion for bug 6320749.
  assert (Universe::verify_in_progress() ||
          !SafepointSynchronize::is_at_safepoint(), "invariant") ;

  for (;;) {
      const markOop mark = object->mark() ;
      assert (!mark->has_bias_pattern(), "invariant") ;

      // The mark can be in one of the following states:
      // *  Inflated     - just return
      // *  Stack-locked - coerce it to inflated
      // *  INFLATING    - busy wait for conversion to complete
      // *  Neutral      - aggressively inflate the object.
      // *  BIASED       - Illegal.  We should never see this

      // CASE: inflated
      if (mark->has_monitor()) {
          ObjectMonitor * inf = mark->monitor() ;
          assert (inf->header()->is_neutral(), "invariant");
          assert (inf->object() == object, "invariant") ;
```

```
    assert (ObjectSynchronizer::verify_objmon_isinpool(inf), "monitor is invalid");
    return inf ;
}

// CASE: inflation in progress - inflating over a stack-lock.
// Some other thread is converting from stack-locked to inflated.
// Only that thread can complete inflation -- other threads must wait.
// The INFLATING value is transient.
// Currently, we spin/yield/park and poll the markword, waiting for inflation to finish.
// We could always eliminate polling by parking the thread on some auxiliary list.
if (mark == markOopDesc::INFLATING()) {
   TEVENT (Inflate: spin while INFLATING) ;
   ReadStableMark(object) ;
   continue ;
}

// CASE: stack-locked
// Could be stack-locked either by this thread or by some other thread.
//
// Note that we allocate the objectmonitor speculatively, _before_ attempting
// to install INFLATING into the mark word.  We originally installed INFLATING,
// allocated the objectmonitor, and then finally STed the address of the
// objectmonitor into the mark.  This was correct, but artificially lengthened
// the interval in which INFLATED appeared in the mark, thus increasing
// the odds of inflation contention.
//
// We now use per-thread private objectmonitor free lists.
// These list are reprovisioned from the global free list outside the
// critical INFLATING...ST interval.  A thread can transfer
// multiple objectmonitors en-mass from the global free list to its local free list.
// This reduces coherency traffic and lock contention on the global free list.
// Using such local free lists, it doesn't matter if the omAlloc() call appears
// before or after the CAS(INFLATING) operation.
// See the comments in omAlloc().

if (mark->has_locker()) {
    ObjectMonitor * m = omAlloc (Self) ;
    // Optimistically prepare the objectmonitor - anticipate successful CAS
    // We do this before the CAS in order to minimize the length of time
    // in which INFLATING appears in the mark.
    m->Recycle();
    m->_Responsible  = NULL ;
    m->OwnerIsThread = 0 ;
    m->_recursions   = 0 ;
    m->_SpinDuration = ObjectMonitor::Knob_SpinLimit ;   // Consider: maintain by type/class

    markOop cmp = (markOop) Atomic::cmpxchg_ptr (markOopDesc::INFLATING(), object->mark_addr(), m
    if (cmp != mark) {
       omRelease (Self, m, true) ;
       continue ;       // Interference -- just retry
    }

    // We've successfully installed INFLATING (0) into the mark-word.
    // This is the only case where 0 will appear in a mark-work.
```

```
// Only the singular thread that successfully swings the mark-word
// to 0 can perform (or more precisely, complete) inflation.
//
// 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.


// fetch the displaced mark from the owner's stack.
// The owner can't die or unwind past the lock while our INFLATING
// object is in the mark.  Furthermore the owner can't complete
// an unlock on the object, either.
markOop dmw = mark->displaced_mark_helper() ;
assert (dmw->is_neutral(), "invariant") ;

// Setup monitor fields to proper values -- prepare the monitor
m->set_header(dmw) ;

// Optimization: if the mark->locker stack address is associated
// with this thread we could simply set m->_owner = Self and
// m->OwnerIsThread = 1. Note that a thread can inflate an object
// that it has stack-locked -- as might happen in wait() -- directly
// with CAS.  That is, we can avoid the xchg-NULL .... ST idiom.
m->set_owner(mark->locker());
m->set_object(object);
// TODO-FIXME: assert BasicLock->dhw != 0.

// Must preserve store ordering. The monitor state must
// be stable at the time of publishing the monitor address.
guarantee (object->mark() == markOopDesc::INFLATING(), "invariant") ;
object->release_set_mark(markOopDesc::encode(m));

// Hopefully the performance counters are allocated on distinct cache lines
// to avoid false sharing on MP systems ...
if (ObjectMonitor::_sync_Inflations != NULL) ObjectMonitor::_sync_Inflations->inc() ;
TEVENT(Inflate: overwrite stacklock) ;
if (TraceMonitorInflation) {
  if (object->is_instance()) {
```

```
        ResourceMark rm;
        tty->print_cr("Inflating object " INTPTR_FORMAT " , mark " INTPTR_FORMAT " , type %s",
          (void *) object, (intptr_t) object->mark(),
          object->klass()->external_name());
      }
    }
    return m ;
}

// CASE: neutral
// TODO-FIXME: for entry we currently inflate and then try to CAS _owner.
// If we know we're inflating for entry it's better to inflate by swinging a
// pre-locked objectMonitor pointer into the object header.   A successful
// CAS inflates the object *and* confers ownership to the inflating thread.
// In the current implementation we use a 2-step mechanism where we CAS()
// to inflate and then CAS() again to try to swing _owner from NULL to Self.
// An inflateTry() method that we could call from fast_enter() and slow_enter()
// would be useful.

assert (mark->is_neutral(), "invariant");
ObjectMonitor * m = omAlloc (Self) ;
// prepare m for installation - set monitor to initial state
m->Recycle();
m->set_header(mark);
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 type/clas

if (Atomic::cmpxchg_ptr (markOopDesc::encode(m), object->mark_addr(), mark) != mark) {
    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.
    // The state-transitions are one-way, so there's no chance of
    // live-lock -- "Inflated" is an absorbing state.
}

// Hopefully the performance counters are allocated on distinct
// cache lines to avoid false sharing on MP systems ...
if (ObjectMonitor::_sync_Inflations != NULL) ObjectMonitor::_sync_Inflations->inc() ;
TEVENT(Inflate: overwrite neutral) ;
if (TraceMonitorInflation) {
  if (object->is_instance()) {
    ResourceMark rm;
    tty->print_cr("Inflating object " INTPTR_FORMAT " , mark " INTPTR_FORMAT " , type %s",
      (void *) object, (intptr_t) object->mark(),
      object->klass()->external_name());
```

```
            }
          }
          return m ;
        }
      }
```

重量级锁

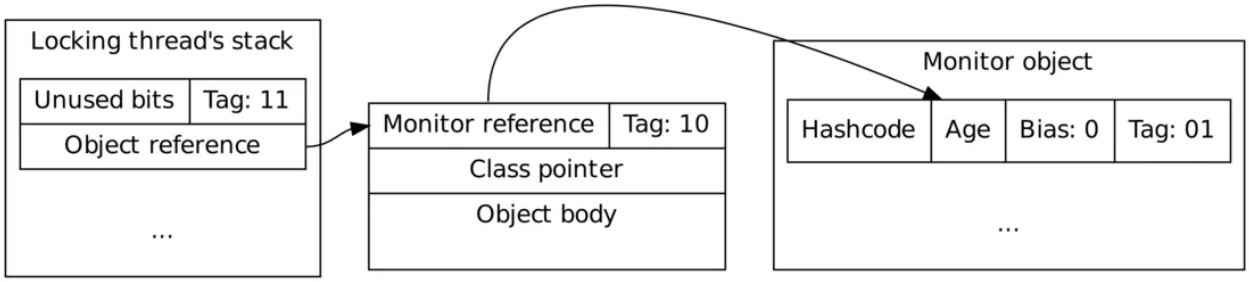
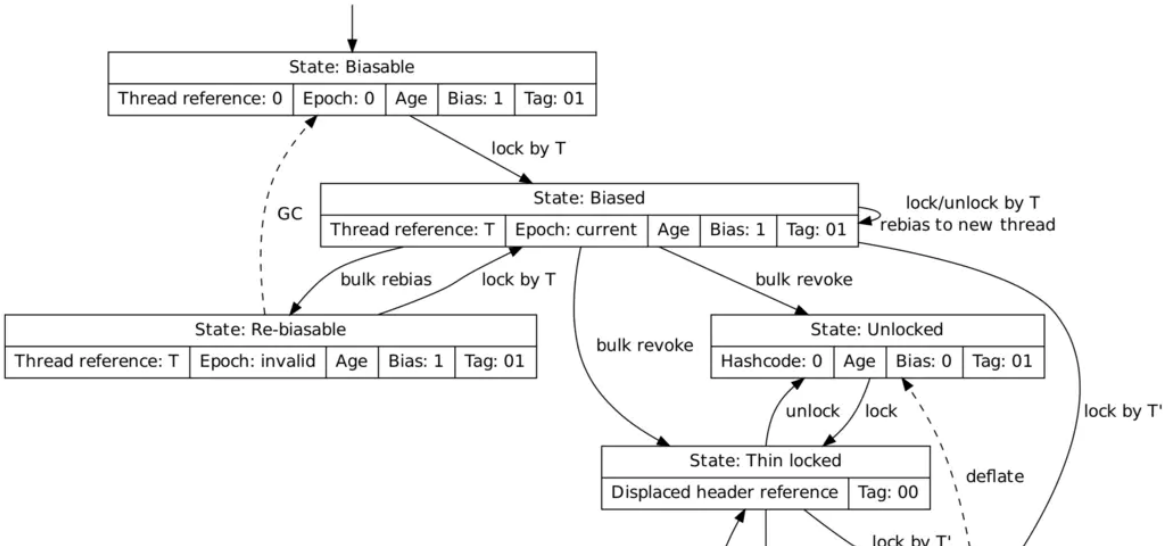


Figure 3.4. Example of a fat locked object.

pasted-image.png

重量级锁非常适合锁竞争激烈的场景，毕竟这个时候自旋往往只会造成无谓的空转。重量级锁相对于上面的两种锁，多了一个 monitor object，当你查看 monitor object 以及相关逻辑源码，你会发现其结构和处理逻辑上跟 AQS 非常相似 -- queue + cas + spin，实际上，synchronized (即使膨胀成重量级锁) 仍然可以满足性能需求，更别提一些列大大小小的优化，synchronized 已经完全可以满足性能需求，你也不难发现包括向 Spring / Jackson 等一些框架在一些方法上也仍然使用关键字来实现同步，其实 JAVA 并发库中的 AQS，更对的是提供丰富的应用场景，例如：ReadWriteLock / CountDownLatch / Semaphore 等。

最后我们来看个这几种锁之间的完整转化图：



pasted-image.png