先提出三个问题：
1、synchronized 如何实现锁?
2、为什么任何对象都可以成为锁?
3、锁存在哪个地方?

每个对象在内存中都存储为**对象头、实例数据、对齐填充**（非必须，HotSpot VM 的自动内存管理系统要求对象的起始地址必须是8字节的整数倍，也就是说对象占用的内存大小必须为8字节的整数倍，如果实例数据没有对齐，则需要进行对齐填充来补全）
注意：数组对象与普通Java对象的区别。

任何对象在 JVM 层面都有一个 oop/oopDesc 对应。
在 oop.hpp 中有一个 oopDesc class，代码如下：

```
class oopDesc {
    friend class VMStructs;
    private:
        volatile markOop _mark; // 此处是对象的对象头
        union _metadata {
            Klass* _klass;
            narrowKlass _compressed_klass;
        } _metadata;

    // 省略。。。
}

// 其中 markOop 在 markOop.hpp 中，注释中解释了保存了什么样的数据
// 该文件中定义了 markOopDesc 继承 oopDesc，其中扩展了一个monitor()方法，返回了 ObjectMonitor 对象
// ObjectMonitor：对象监视器
ObjectMonitor* monitor() const {
    assert(has_monitor(), "check");
    // Use xor instead of & to provide ont extra tag-bit check
    return (ObjectMonitor*) (value() ^ monitor_value);
}

// 在 objectMonitor.hpp 中可以找到 ObjectMonitor 的定义
ObjectMonitor() {
    _header       = NULL; // markOop 对象头
    _count        = 0; //记录个数
    _waiters      = 0,
    _recursions   = 0; // 重入次数
    _object       = NULL;
    _owner        = NULL; // 指向获得 ObjectMonitor 对象的线程
    _WaitSet      = NULL; // 处于wait状态的线程，会被加入到_WaitSet
    _WaitSetLock  = 0 ;
    _Responsible  = NULL ;
    _succ         = NULL ;
    _cxq          = NULL ; // JVM 为每个尝试进入 synchronized 代码块的 JavaThread 创建一个 ObjectWaiter
    FreeNext      = NULL ;
    _EntryList    = NULL ; //处于等待锁block状态的线程，由 ObjectWaiter 组成的双向链表。
    _SpinFreq     = 0 ;
    _SpinClock    = 0 ;
    OwnerIsThread = 0 ;
    _previous_owner_tid = 0; // 监视器前一个拥有者的线程 id
}
```

如下是 32 位 HotSpot 虚拟机中 Mark Word 在不同状态下存储的信息：

锁状态	25 bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
无锁	对象的hashCode		对象分代年龄	0	01
偏向锁	线程ID	Epoch	对象分代年龄	1	01
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥量（重量级锁）的指针				10
GC标记	空				11

因为任何对象在 JVM 中都会存在一个对象头，而对象头中存储了锁标志，所以每个对象都可以成为锁。
至此解决了 2、3 两个问题。

synchronized 实现锁的原理看这篇文章：https://keanu96.github.io/deep-in-synchronized/

锁的优化

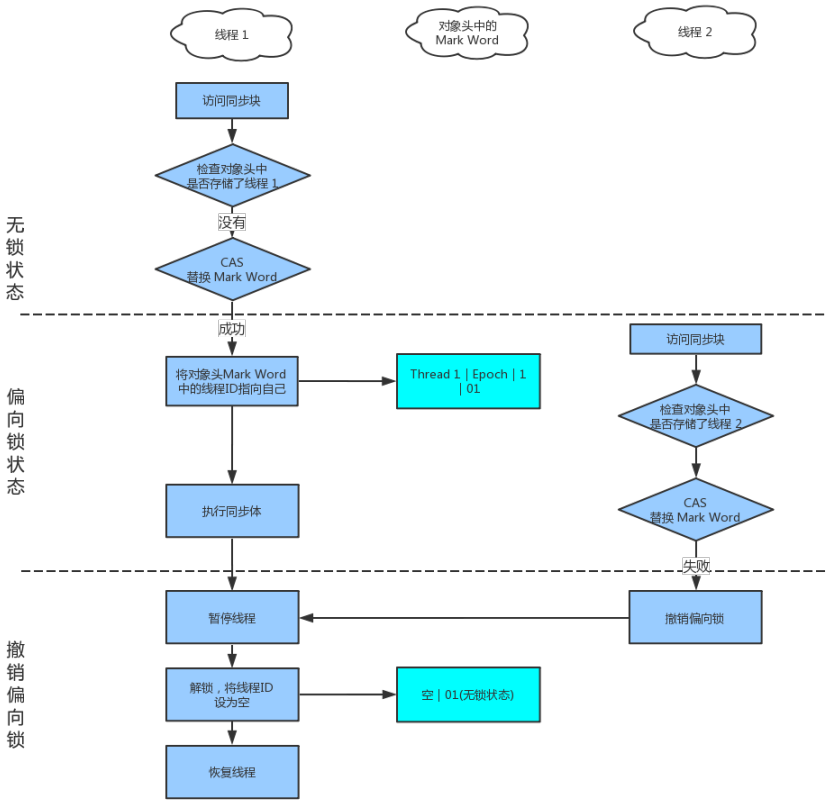
自旋锁

当尝试获取锁的时候，会进行（有限次数的）无意义的循环。这样避免了线程阻塞、唤醒的性能开销。如果在指定时间内或者次数内没有获取到锁，还是会进入到阻塞状态。

偏向锁

在锁不存在竞争，并且都是由同一个线程获得时，采用偏向锁。但是这个时候有另一个线程尝试获取锁，将会先撤销偏向锁，升级为轻量级锁。

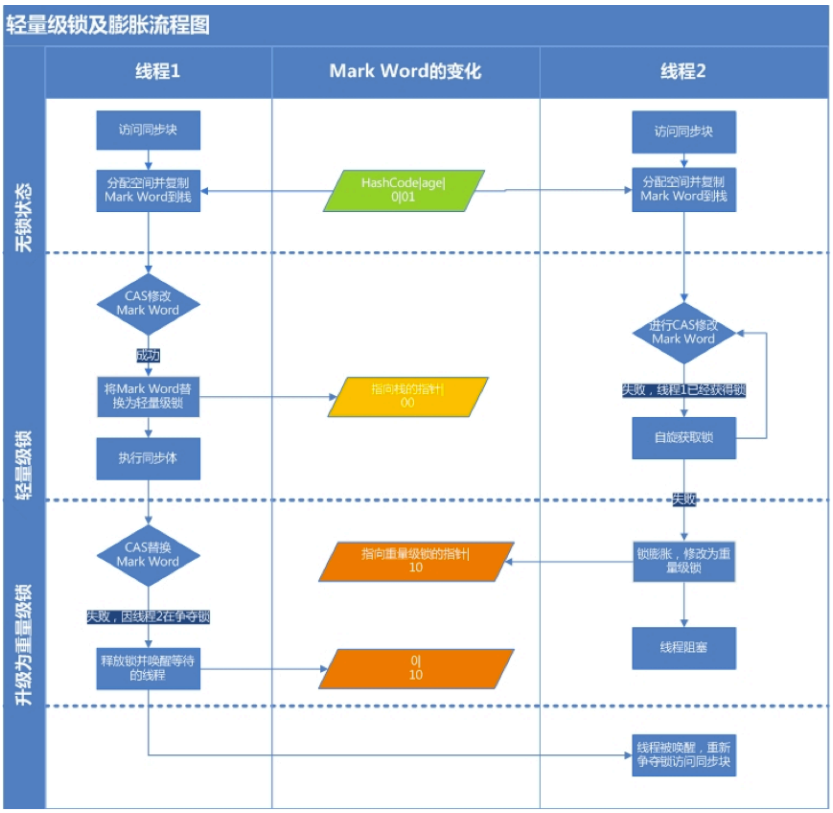
偏向锁的获得和撤销流程



轻量级锁

在没有多线程竞争的情况下，减少重量级锁导致的性能损耗。

加锁以及锁膨胀流程图



加锁的条件：无锁状态

无锁状态下，锁状态标志位为01，偏向锁标志位为0，则可尝试加锁。

Mark Word 初始状态：同步对象处于无锁状态，锁标志位为01，偏向锁标志位为0。

建立锁记录

1、在加锁前，虚拟机需要在当前线程的栈桢中建立锁记录（Lock Record）的空间。Lock Record 中包含一个 _displaced_header 属性，用于存储 锁对象 Mark Word 的拷贝。

复制锁对象的 Mark Word

2、将锁对象的 Mark Word 复制到锁记录中，复制过来的记录叫做 Displaced Mark Word。就是将锁对象的 Mark Word 放到锁记录的 _displaced_header 属性中。

```
// 将 mark word 设置到 _displaced_header 中
lock -> set_displaced_header(mark);

class BasicLock VALUE_OBJ_CLASS_SPEC {
    friend class VMStructs;
    private:
        volatile markOop _displaced_header;
    public:
        void set_displaced_header(markOop header) { _displaced_header = header; }
    // ......
}
```

CAS 更新锁对象的 Mark Word

3、虚拟机使用 CAS 操作尝试将锁对象的 Mark Word 更新为指向锁记录的指针。如果更新成功，这个线程就获取到该对象的锁。

```
// lock 指向 Lock Record 的指针
// obj() -> mark_addr() 锁对象的 Mark Word 地址
// mark 锁对象的 Mark Word
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 ;
        }
    } 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;
    }

    // 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.
    // 当锁膨胀为重量级锁时，将不会把锁对象头替换为指向锁的指针，
    // 所以 _displaced_header 是什么值都无关紧要，但是它必须要是
    // 非空值避免与可重入锁一样，并且也不能是锁住的。
    lock->set_displaced_header(markOopDesc::unused_mark());
    ObjectSynchronizer::inflate(THREAD, obj())->enter(THREAD);
}
```

轻量级锁加锁（有锁状态）

当同步对象处于有锁状态时，如果是当前线程持有的轻量级锁，则说明是重入，不需要争抢锁。否则，说明有多个线程竞争锁，轻量级锁需要升级为重量级锁。

当前线程持有锁

锁对象处于加锁状态，并且锁对象的 Mark Word 指向当前线程的栈桢范围内，就表示当前线程持有该轻量级锁，可在此获取到该锁，也就是重入。

此时就不需要争抢锁，直接执行同步代码。

```
// Mark Word 处于加锁状态，当前线程持有锁（锁对象的 Mark Word 指向当前线程的栈桢范围内）
if (mark->has_lock() && 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 re-lock with same BasicLock");
    // 锁重入，将 Displaced Mark Word 设置为 null
    lock->set_displaced_header(NULL);
    return;
}
```

当不是当前线程持有锁

上一种条件不满足，说明存在多个线程竞争锁，轻量级锁要膨胀了。膨胀成重量级锁后再加锁。

```
// 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);
```

轻量级锁解锁

解锁的思路是使用 CAS 操作把当前线程的栈帧中的 Displaced Mark Word 替换回锁对象中去，如果替换成功，则解锁成功。

CAS 替换回 Mark Word

```
// ...
mark = object->mark() ;

// If the object is stack-locked by the current thread, try to
// swing the displaced header from the box back to the mark.
if (mark == (markOop) lock) {
  assert (dhw->is_neutral(), "invariant") ;
  // 将 Displaced Mark Word 替换回去
  if ((markOop) Atomic::cmpxchg_ptr (dhw, object->mark_addr(), mark) == mark) {
    TEVENT (fast_exit: release stacklock) ;
    return;
   }
}
```

替换失败

替换失败，轻量级锁膨胀成重量级锁后再解锁

```
ObjectSynchronizer::inflate(THREAD, object)->exit (true, THREAD) ;
```

锁重入

如上文所讲：加锁时，如果是锁重入，会将 Displaced Mark Word 设置为 null。相对应地，解锁时，如果判断 Displaced Mark Word 为 null 则说明是锁重入，不做替换操作。

```
markOop dhw = lock->displaced_header(); // 获取锁记录的 _displaced_header 对象头
markOop mark ;
if (dhw == NULL) { // 如果 dhw 是空，则说明是锁重入
    // Recursive stack-lock.
    // Diagnostics -- Could be: stack-locked, inflating, inflated.
    mark = object->mark() ; // 获取锁对象的 Mark Word
    assert (!mark->is_neutral(), "invariant") ;
    if (mark->has_locker() && mark != markOopDesc::INFLATING()) {
      assert(THREAD->is_lock_owned((address)mark->locker()), "invariant") ;
     }
    if (mark->has_monitor()) {
      ObjectMonitor * m = mark->monitor() ;
      assert(((oop)(m->object()))->mark() == mark, "invariant") ;
      assert(m->is_entered(THREAD), "invariant") ;
    }
    return ;
}
```