只会一点java

掌控自己的生命轨迹，身心自由！

jdk源码剖析二: 对象内存布局、synchronized终极原理

Posted on 2017-04-20 19:00 只会一点java 阅读(2319) 评论(12) 编辑 收藏

目录

正文

很多人一提到锁，自然第一个想到了synchronized，但一直不懂源码实现，现特地追踪到C++层来剥开synchronized的面纱。

网上的很多描述大都不全，让人看了不够爽，看完本章，你将彻底了解synchronized的核心原理。

一、启蒙知识预热

开启本文之前先介绍2个概念

1.1.cas操作

为了提高性能，JVM很多操作都依赖CAS实现，一种乐观锁的实现。本文锁优化中大量用到了CAS，故有必要先分析一下CAS的实现。

CAS：Compare and Swap。

JNI来完成CPU指令的操作：

unsafe.compareAndSwapInt(this, valueOffset, expect, update);

CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。如果A=V，那么把B赋值给V，返回V；如果A！=V，直接返回V。

打开源码：openjdk\hotspot\src\os*cpu\windows*x86\vm\ atomic*windows*x86.inline.hpp，如下图：0

```
inline jint     Atomic::cmpxchg    (jint     exchange_value, volatile jint*     dest, jint     compare_
  // alternative for InterlockedCompareExchange
  int mp = os::is_MP();
  __asm {
    mov edx, dest
    mov ecx, exchange_value
    mov eax, compare_value
    LOCK_IF_MP(mp)
    cmpxchg dword ptr [edx], ecx
  }
}

inline jlong    Atomic::cmpxchg    (jlong    exchange_value, volatile jlong*    dest, jlong    compare_
  int mp = os::is_MP();
  jint ex_lo  = (jint)exchange_value;
  jint ex_hi  = *( ((jint*)&exchange_value) + 1 );
  jint cmp_lo = (jint)compare_value;
  jint cmp_hi = *( ((jint*)&compare_value) + 1 );
  __asm {
    push ebx
    push edi
    mov eax, cmp_lo
    mov edx, cmp_hi
    mov edi, dest
    mov ebx, ex_lo
    mov ecx, ex_hi
    LOCK_IF_MP(mp)
    cmpxchg8b qword ptr [edi]
    pop edi
    pop ebx
  }
}
```

os::is_MP() 这个是runtime/os.hpp，实际就是返回是否多处理器，源码如下：

```
// Interface for detecting multiprocessor system
static inline bool is_MP() {
  assert(_processor_count > 0, "invalid processor count");
  return _processor_count > 1 || AssumeMP;
}
```

如上面源代码所示（看第一个int参数即可），LOCK_IF_MP:会根据当前处理器的类型来决定是否为cmpxchg指令添加lock前缀。如果程序是在多处理
cmpxchg指令加上lock前缀（lock cmpxchg）。反之，如果程序是在单处理器上运行，就省略lock前缀（单处理器自身会维护单处理器内的顺序一致
缀提供的内存屏障效果）。

1.2.对象头

HotSpot虚拟机中，对象在内存中存储的布局可以分为三块区域：对象头（Header）、实例数据（Instance Data）和对齐填充（Padding）。

HotSpot虚拟机的对象头(Object Header)包括两部分信息:

第一部分"Mark Word":用于存储对象自身的运行时数据， 如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向

第二部分"Klass Pointer"：对象指向它的类的元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。(数组，对象头中还必须有一块用
据，因为虚拟机可以通过普通Java对象的元数据信息确定Java对象的大小，但是从数组的元数据中无法确定数组的大小。)

32位的HotSpot虚拟机对象头存储结构：（下图摘自网络）

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

图1 32位的HotSpot虚拟机对象头Mark Word组成

为了证实上图的正确性，这里我们看openJDK--》hotspot源码markOop.hpp，虚拟机对象头存储结构：

```
new 1  new 2  new 3  markOop.hpp
 * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
 * or visit www.oracle.com if you need additional information or have any
 * questions.
 *
 */

#ifndef SHARE_VM_OOPS_MARKOOP_HPP
#define SHARE_VM_OOPS_MARKOOP_HPP

#include "oops/oop.hpp"

// The markOop describes the header of an object.
//
// Note that the mark is not a real oop but just a word.
// It is placed in the oop hierarchy for historical reasons.
//
// Bit-format of an object header (most significant first, big endian layout below):
//
//  32 bits:
//  --------
//             hash:25 ------------>| age:4    biased_lock:1 lock:2 (normal object)
//             JavaThread*:23 epoch:2 age:4    biased_lock:1 lock:2 (biased object)
//             size:32 ------------------------------------------>| (CMS free block)
//             PromotedObject*:29 ---------->| promo_bits:3 ----->| (CMS promoted object)
//
//  64 bits:
//  --------
//  unused:25 hash:31 -->| unused:1   age:4    biased_lock:1 lock:2 (normal object)
//  JavaThread*:54 epoch:2 unused:1   age:4    biased_lock:1 lock:2 (biased object)
//  PromotedObject*:61 --------------------->| promo_bits:3 ----->| (CMS promoted object)
//  size:64 ----------------------------------------------------->| (CMS free block)
//
//  unused:25 hash:31 -->| cms_free:1 age:4    biased_lock:1 lock:2 (COOPs && normal object)
//  JavaThread*:54 epoch:2 cms_free:1 age:4    biased_lock:1 lock:2 (COOPs && biased object)
//  narrowOop:32 unused:24 cms_free:1 unused:4 promo_bits:3 ----->| (COOPs && CMS promoted obj
//  unused:21 size:35 -->| cms_free:1 unused:7 ------------------>| (COOPs && CMS free block)
//
```

图2 HotSpot源码markOop.hpp中注释

单词解释：

hash： 保存对象的哈希码

age： 保存对象的分代年龄

biased_lock： 偏向锁标识位

lock： 锁状态标识位

JavaThread*： 保存持有偏向锁的线程ID

epoch： 保存偏向时间戳

2017年3月 (2)
2017年1月 (1)
2016年12月 (1)
2016年11月 (3)
2016年10月 (1)
2016年9月 (2)
2016年8月 (1)
2016年7月 (2)
2016年6月 (1)
2016年5月 (1)

积分与排名

积分 - 67164
排名 - 5260

最新评论

1. Re:kafka原理和实践（二）spring-kafka简单实践
不是大神，能帮到你很开心!
--只会一点java

2. Re:kafka原理和实践（二）spring-kafka简单实践
@只会一点java谢谢大神~搞定了~...
--奥巴马说我长得丑

3. Re:kafka原理和实践（二）spring-kafka简单实践
@奥巴马说我长得丑 org.springframework.kafka.listener.config.ContainerProperties看一下这个类的构造，注意MessageListener是自动......
--只会一点java

4. Re:kafka原理和实践（二）spring-kafka简单实践
参数中加Acknowledgment ack，消费完毕ack.acknowledge();
--只会一点java

5. Re:jdk源码剖析二: 对象内存布局、synchronized终极原理
哈哈，不用拜师，加粉就可以...
--只会一点java

阅读排行榜

1. PowerDesigner连接mysql逆向生成pdm(14322)
2. JDK8-十大新特性-附demo(13276)
3. eclipse下SVN同步时忽略target文件夹(9214)
4. maven常用插件pom配置(9161)
5. Eclipse Memory Analyzer，内存泄漏插件，安装使用一条龙(8274)

评论排行榜

1. jdk源码剖析二: 对象内存布局、synchronized终极原理(12)
2. spring boot容器启动详解(5)
3. kafka原理和实践（三）spring-kafka生产者源码(5)
4. kafka原理和实践（二）spring-kafka简单实践(5)
5. jdk源码剖析一：OpenJDK-Hotspot源码包目录结构(4)

推荐排行榜

1. spring boot容器启动详解(7)
2. JDK8-废弃永久代（PermGen）迎来元空间（Metaspace）(5)

上图中有源码中对锁标志位这样枚举：

```
enum {  locked_value             = 0,//00 轻量级锁
        unlocked_value           = 1,//01 无锁
        monitor_value            = 2,//10 监视器锁，也叫膨胀锁，也叫重量级锁
        marked_value             = 3,//11 GC标记
        biased_lock_pattern      = 5 //101 偏向锁
 };
```

下面是源码注释：

```
  [JavaThread* | epoch | age | 1 | 01]       lock is biased toward given thread
  [0           | epoch | age | 1 | 01]       lock is anonymously biased

 - the two lock bits are used to describe three states: locked/unlocked and monitor.

  [ptr             | 00]  locked             ptr points to real header on stack
  [header      | 0 | 01]  unlocked           regular object header
  [ptr             | 10]  monitor            inflated lock (header is wapped out)
  [ptr             | 11]  marked             used by markSweep to mark an object
                                             not valid at any other time
```

图3 HotSpot源码markOop.hpp中锁标志位注释

看图3，不管是32/64位JVM，都是1bit偏向锁+2bit锁标志位。上面红框是偏向锁（第一行是指向线程的显示偏向锁，第二行是匿名偏向锁）对应枚举biased_lock_pattern，下面红框是轻量级锁、无锁、监视器锁、GC标记，分别对应上面的前4种枚举。我们甚至能看见锁标志11时，是GC的markS法)使用的。（这里就不再拓展了）

对象头中的Mark Word，synchronized源码实现就用了Mark Word来标识对象加锁状态。

二、JVM中synchronized锁实现原理（优化）

大家都知道java中锁synchronized性能较差，线程会阻塞。本节将以图文形式来描述JVM的synchronized锁优化。

在jdk1.6中对锁的实现引入了大量的优化来减少锁操作的开销：

锁粗化（Lock Coarsening）：将多个连续的锁扩展成一个范围更大的锁，用以减少频繁互斥同步导致的性能损耗。

锁消除（Lock Elimination）：JVM及时编译器在运行时，通过逃逸分析，如果判断一段代码中，堆上的所有数据不会逃逸出去从来被其他线程除这些锁。

轻量级锁（Lightweight Locking）：JDK1.6引入。在没有多线程竞争的情况下避免重量级互斥锁，只需要依靠一条CAS原子指令就可以完成锁

偏向锁（Biased Locking）：JDK1.6引入。目的是消除数据再无竞争情况下的同步原语。使用CAS记录获取它的线程。下一次同一个线程进入则任何同步操作。

适应性自旋（Adaptive Spinning）：为了避免线程频繁挂起、恢复的状态切换消耗。产生了忙循环（循环时间固定），即自旋。JDK1.6引入了时间根据之前锁自旋时间和线程状态，动态变化，用以期望能减少阻塞的时间。

锁升级：**偏向锁--》轻量级锁--》重量级锁**

2.1.偏向锁

按照之前的HotSpot设计，每次加锁/解锁都会涉及到一些CAS操作（比如对等待队列的CAS操作），CAS操作会延迟本地调用，因此偏向锁的一次获得了监视对象，之后让监视对象"偏向"这个线程，之后的多次调用则可以避免CAS操作。

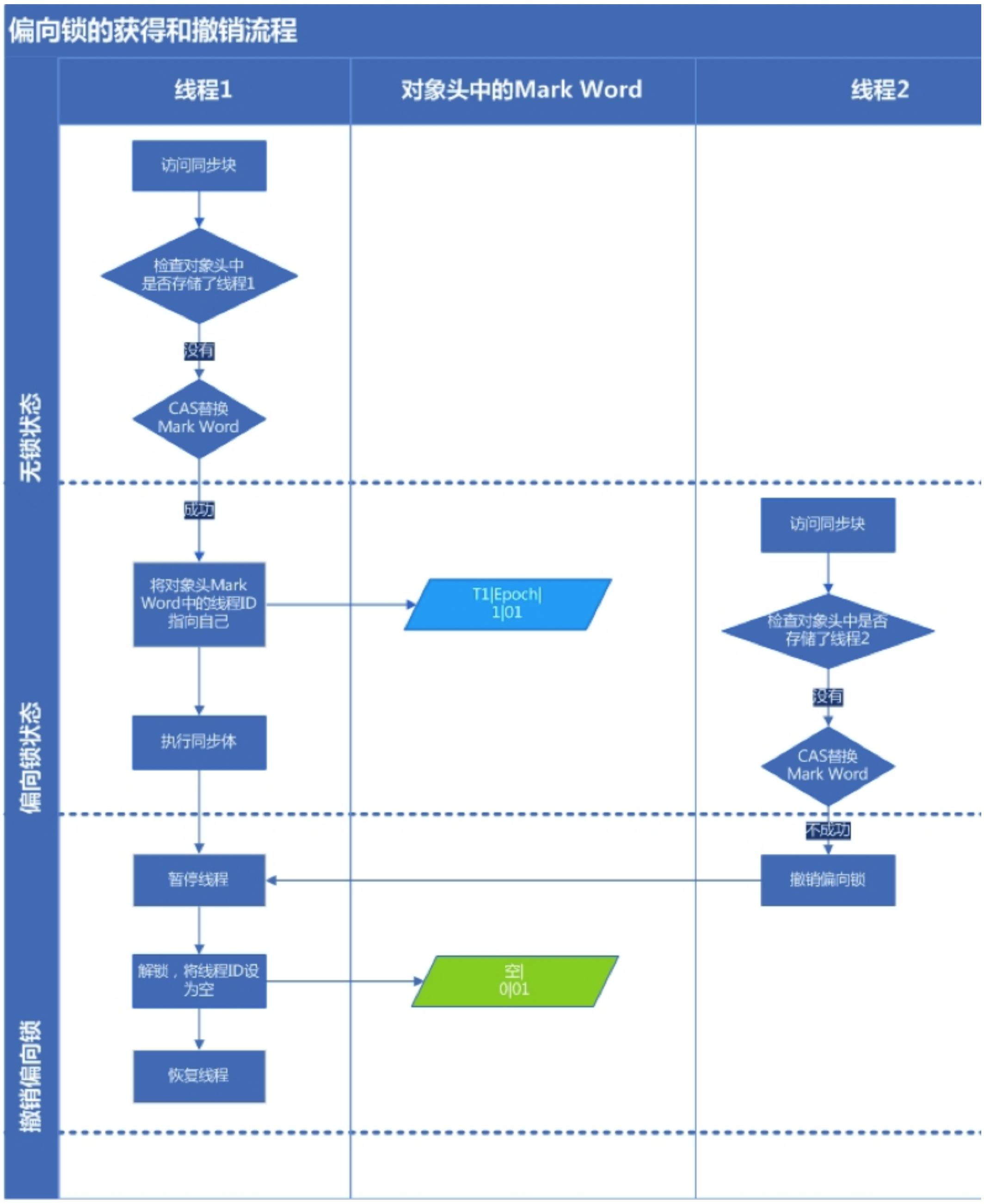
简单的讲，就是在锁对象的对象头（开篇讲的对象头数据存储结构）中有个ThreadId字段，这个字段如果是空的，第一次获取锁的时候，就ThreadId写入到锁的ThreadId字段内，将锁头内的是否偏向锁的状态位置1.这样下次获取锁的时候，直接检查ThreadId是否和自身线程Id一致为当前线程已经获取了锁，因此不需再次获取锁，略过了轻量级锁和重量级锁的加锁阶段。提高了效率。

注意：当锁有竞争关系的时候，需要解除偏向锁，进入轻量级锁。

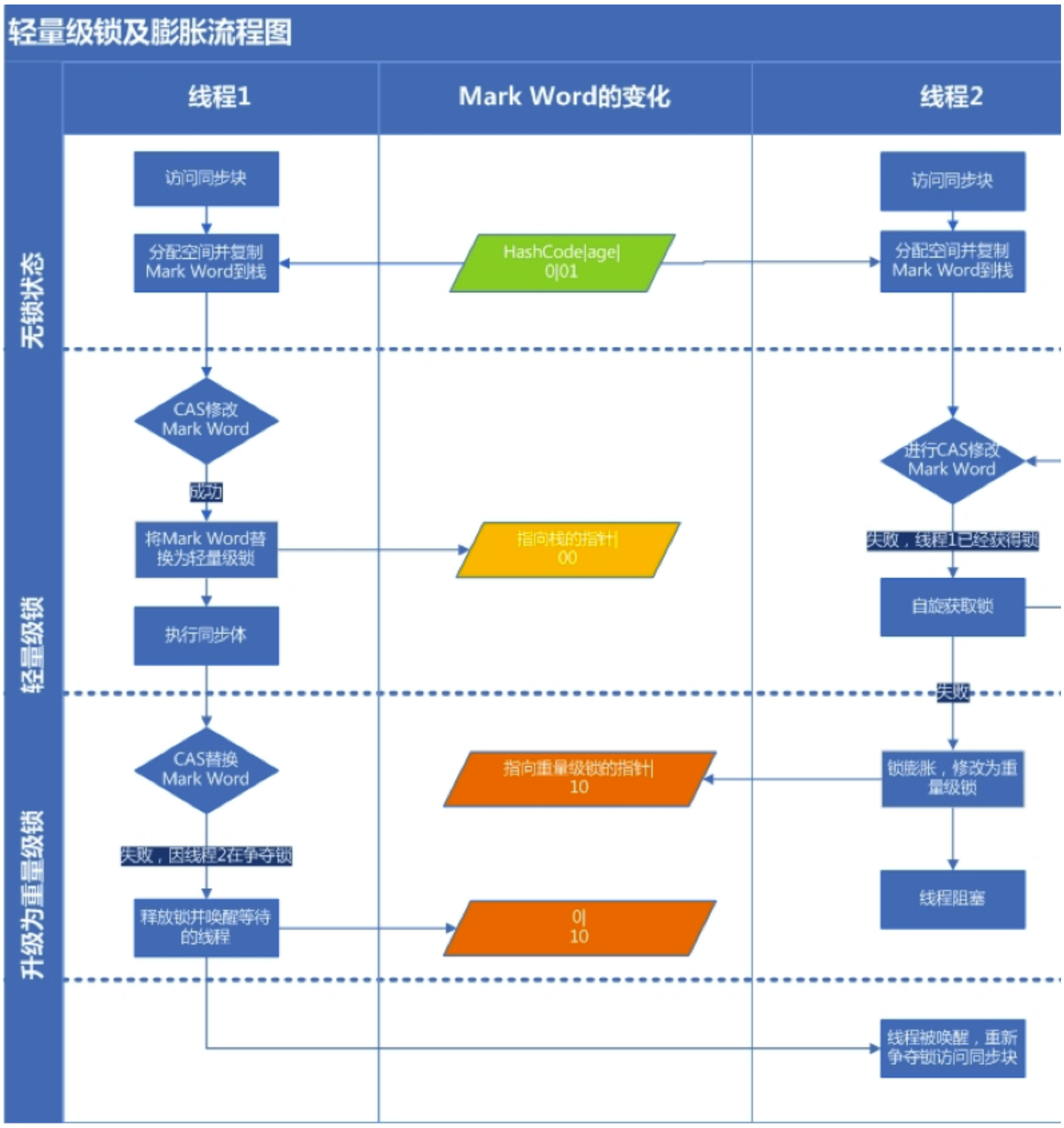
每一个线程在准备获取共享资源时：

第一步，检查MarkWord里面是不是放的自己的ThreadId ,如果是，表示当前线程是处于 "偏向锁".跳过轻量级锁直接执行同

获得偏向锁如下图：



2.2.轻量级锁和重量级锁



如上图所示：

第二步，如果MarkWord不是自己的ThreadId,锁升级，这时候，用CAS来执行切换，新的线程根据MarkWord里面现有的T知之前线程暂停，之前线程将Markword的内容置为空。

第三步，两个线程都把对象的HashCode复制到自己新建的用于存储锁的记录空间，接着开始通过CAS操作，把共享对象的Ma修改为自己新建的记录空间的地址的方式竞争MarkWord.

第四步，第三步中成功执行CAS的获得资源，失败的则进入自旋.

第五步，自旋的线程在自旋过程中，成功获得资源(即之前获的资源的线程执行完成并释放了共享资源)，则整个状态依然处于轻量果自旋失败 第六步，进入重量级锁的状态，这个时候，自旋的线程进行阻塞，等待之前线程执行完成并唤醒自己.

注意点：JVM加锁流程

偏向锁--》轻量级锁--》重量级锁

从左往右可以升级，从右往左不能降级

三、从C++源码看synchronized

前两节讲了synchronized锁实现原理，这一节我们从C++源码来剖析synchronized。

3.1 同步和互斥

同步：多个线程并发访问共享资源时，保证同一时刻只有一个（信号量可以多个）线程使用。

实现同步的方法有很多，常见四种如下：

1）临界区（CriticalSection，又叫关键段）：通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。进程内可用。

2）互斥量：**互斥量用于线程的互斥。只能为0/1。**一个互斥量只能用于一个资源的互斥访问，可跨进程使用。

3）信号量：**信号线用于线程的同步。可以为非负整数，**可实现多个同类资源的多线程互斥和同步。当信号量为单值信号量是，也可以完成一问。可跨进程使用。

4）事件：用来通知线程有一些事件已发生，从而启动后继任务的开始,可跨进程使用。

synchronized的底层实现就用到了临界区和互斥锁（重量级锁的情况下）这两个概念。

3.2 synchronized C++源码

重点来了，之前在第一节中的图1，看过了对象头Mark Word。现在我们从C++源码来剖析具体的数据结构和获取释放锁的过程。

2.2.1 C++中的监视器锁数据结构

oopDesc--继承-->markOopDesc--方法monitor()-->ObjectMonitor-->enter、exit 获取、释放锁

1.oopDesc类

openjdk\hotspot\src\share\vm\oops\oop.hpp下oopDesc类是JVM对象的顶级基类,故每个object都包含markOop。如下图所示：

```
class oopDesc {
  friend class VMStructs;
 private:
  volatile markOop  _mark;//markOop:Mark Word标记字段
  union _metadata {
    Klass*      _klass;//对象类型元数据的指针
    narrowKlass _compressed_klass;
  } _metadata;

  // Fast access to barrier set.  Must be initialized.
  static BarrierSet* _bs;

 public:
  markOop  mark() const         { return _mark; }
  markOop* mark_addr() const    { return (markOop*) &_mark; }

  void set_mark(volatile markOop m)      { _mark = m;   }

  void    release_set_mark(markOop m);
  markOop cas_set_mark(markOop new_mark, markOop old_mark);

  // Used only to re-initialize the mark word (e.g., of promoted
  // objects during a GC) -- requires a valid klass pointer
  void init_mark();

  Klass* klass() const;
  Klass* klass_or_null() const volatile;
  Klass** klass_addr();
  narrowKlass* compressed_klass_addr();
....省略...
}
```

2.markOopDesc类

openjdk\hotspot\src\share\vm\oops\markOop.hpp下markOopDesc继承自oopDesc，并拓展了自己的方法monitor(),如下图

```
ObjectMonitor* monitor() const {
    assert(has_monitor(), "check");
    // Use xor instead of &~ to provide one extra tag-bit check.
    return (ObjectMonitor*) (value() ^ monitor_value);
  }
```

该方法返回一个ObjectMonitor*对象指针。

其中value()这样定义：

```
uintptr_t value() const { return (uintptr_t) this; }
```

可知：将this转换成一个指针宽度的整数（uintptr_t），然后进行"异或"位操作。

monitor_value是常量

```
enum {   locked_value             = 0,//00偏向锁
         unlocked_value           = 1,//01无锁
         monitor_value            = 2,//10监视器锁，又叫重量级锁
         marked_value             = 3,//11GC标记
         biased_lock_pattern      = 5 //101偏向锁
  };
```

指针低2位00，异或10，结果还是10.（拿一个模板为00的数，异或一个二位数=数本身。因为异或："相同为0，不同为1".操作）

3.ObjectMonitor类

在HotSpot虚拟机中，最终采用ObjectMonitor类实现monitor。

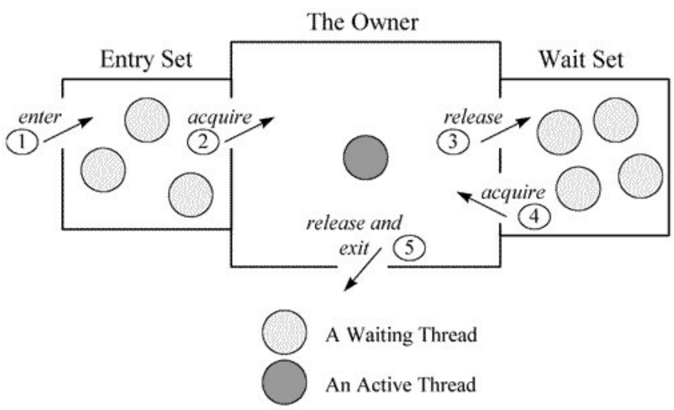
openjdk\hotspot\src\share\vm\runtime\objectMonitor.hpp源码如下：

```
ObjectMonitor() {
    _header       = NULL;//markOop对象头
    _count        = 0;
    _waiters      = 0,//等待线程数
    _recursions   = 0;//重入次数
    _object       = NULL;//监视器锁寄生的对象。锁不是平白出现的，而是寄托存储于对象中。
    _owner        = NULL;//指向获得ObjectMonitor对象的线程或基础锁
```

```
  _WaitSet      = NULL;//处于wait状态的线程，会被加入到wait set;
  _WaitSetLock  = 0 ;
  _Responsible  = NULL ;
  _succ         = NULL ;
  _cxq          = NULL ;
  FreeNext      = NULL ;
  _EntryList    = NULL ;//处于等待锁block状态的线程，会被加入到entry set;
  _SpinFreq     = 0 ;
  _SpinClock    = 0 ;
  OwnerIsThread = 0 ;// _owner is (Thread *) vs SP/BasicLock
  _previous_owner_tid = 0;// 监视器前一个拥有者线程的ID
}
```

每个线程都有两个ObjectMonitor对象列表，分别为free和used列表，如果当前free列表为空，线程将向全局global list请求分配ObjectMonitor。

ObjectMonitor对象中有两个队列：_WaitSet 和 _EntryList，用来保存ObjectWaiter对象列表；



2.获取锁流程

synchronized关键字修饰的代码段，在JVM被编译为monitorenter、monitorexit指令来获取和释放互斥锁.。

解释器执行monitorenter时会进入到`InterpreterRuntime.cpp`的`InterpreterRuntime::monitorenter`函数，具体实现如下：

```
IRT_ENTRY_NO_ASYNC(void, InterpreterRuntime::monitorenter(JavaThread* thread, BasicObjectLock* elem))
#ifdef ASSERT
  thread->last_frame().interpreter_frame_verify_monitor(elem);
#endif
  if (PrintBiasedLockingStatistics) {
    Atomic::inc(BiasedLocking::slow_path_entry_count_addr());
  }
  Handle h_obj(thread, elem->obj());
  assert(Universe::heap()->is_in_reserved_or_null(h_obj()),
         "must be NULL or an object");
  if (UseBiasedLocking) {//标识虚拟机是否开启偏向锁功能,默认开启
    // 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);
  }
  assert(Universe::heap()->is_in_reserved_or_null(elem->obj()),
         "must be NULL or an object");
#ifdef ASSERT
  thread->last_frame().interpreter_frame_verify_monitor(elem);
#endif
IRT_END
```

先看一下入参：

1、JavaThread thread指向java中的当前线程；
2、BasicObjectLock基础对象锁：包含一个BasicLock和一个指向Object对象的指针oop。

`openjdk\hotspot\src\share\vm\runtime\basicLock.hpp`中`BasicObjectLock`类源码如下：

```
class BasicObjectLock VALUE_OBJ_CLASS_SPEC {
  friend class VMStructs;
 private:
  BasicLock _lock;                                    // the lock, must be double word aligned
  oop       _obj;                                     // object holds the lock;

 public:
  // Manipulation
  oop      obj() const                                { return _obj;  }
  void set_obj(oop obj)                               { _obj = obj; }
```

```
  BasicLock* lock()                                { return &_lock; }

  // Note: Use frame::interpreter_frame_monitor_size() for the size of BasicObjectLocks
  //       in interpreter activation frames since it includes machine-specific padding.
  static int size()                                { return sizeof(BasicObjectLock)/wordSize; }

  // GC support
  void oops_do(OopClosure* f) { f->do_oop(&_obj); }

  static int obj_offset_in_bytes()                 { return offset_of(BasicObjectLock, _obj);  }
  static int lock_offset_in_bytes()                { return offset_of(BasicObjectLock, _lock); }
};
```

3、BasicLock类型_lock对象主要用来保存：指向Object对象的对象头数据；

basicLock.hpp中BasicLock源码如下：

```
class BasicLock VALUE_OBJ_CLASS_SPEC {
  friend class VMStructs;
 private:
  volatile markOop _displaced_header;//markOop是不是很熟悉？1.2节中讲解对象头时就是分析的markOop源码
 public:
  markOop      displaced_header() const               { return _displaced_header; }
  void         set_displaced_header(markOop header)   { _displaced_header = header; }

  void print_on(outputStream* st) const;

  // move a basic lock (used during deoptimization
  void move_to(oop obj, BasicLock* dest);

  static int displaced_header_offset_in_bytes()       { return offset_of(BasicLock, _displaced_header);
};
```

偏向锁的获取ObjectSynchronizer::fast_enter

在HotSpot中，偏向锁的入口位于openjdk\hotspot\src\share\vm\runtime\synchronizer.cpp文件的ObjectSynchronizer::fast_enter函

```
void ObjectSynchronizer::fast_enter(Handle obj, BasicLock* lock, bool attempt_rebias, TRAPS) {
 if (UseBiasedLocking) {
    if (!SafepointSynchronize::is_at_safepoint()) {
      BiasedLocking::Condition cond = BiasedLocking::revoke_and_rebias(obj, attempt_rebias, THREAD);
      if (cond == BiasedLocking::BIAS_REVOKED_AND_REBIASED) {
        return;
      }
    } else {
      assert(!attempt_rebias, "can not rebias toward VM thread");
      BiasedLocking::revoke_at_safepoint(obj);
    }
    assert(!obj->mark()->has_bias_pattern(), "biases should be revoked by now");
 }
 //轻量级锁
 slow_enter (obj, lock, THREAD) ;
}
```

偏向锁的获取由BiasedLocking::revoke_and_rebias方法实现，由于实现比较长，就不贴代码了，实现逻辑如下：
1、通过markOop mark = obj->mark()获取对象的markOop数据mark，即对象头的Mark Word；
2、判断mark是否为可偏向状态，即mark的偏向锁标志位为 **1**，锁标志位为 **01**；
3、判断mark中JavaThread的状态：如果为空，则进入步骤（4）；如果指向当前线程，则执行同步代码块；如果指向其它线程，进入步骤（5）；
4、通过CAS原子指令设置mark中JavaThread为当前线程ID，如果执行CAS成功，则执行同步代码块，否则进入步骤（5）；
5、如果执行CAS失败，表示当前存在多个线程竞争锁，当达到全局安全点（safepoint），获得偏向锁的线程被挂起，撤销偏向锁，并升级为轻量级，在安全点的线程继续执行同步代码块；

偏向锁的撤销

只有当其它线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁，偏向锁的撤销由BiasedLocking::revoke_at_safepoint方法实现：

```
void BiasedLocking::revoke_at_safepoint(Handle h_obj) {
  assert(SafepointSynchronize::is_at_safepoint(), "must only be called while at safepoint");//校验全局安全
  oop obj = h_obj();
  HeuristicsResult heuristics = update_heuristics(obj, false);
  if (heuristics == HR_SINGLE_REVOKE) {
    revoke_bias(obj, false, false, NULL);
  } else if ((heuristics == HR_BULK_REBIAS) ||
             (heuristics == HR_BULK_REVOKE)) {
    bulk_revoke_or_rebias_at_safepoint(obj, (heuristics == HR_BULK_REBIAS), false, NULL);
  }
  clean_up_cached_monitor_info();
}
```

1、偏向锁的撤销动作必须等待全局安全点；
2、暂停拥有偏向锁的线程，判断锁对象是否处于被锁定状态；
3、撤销偏向锁，恢复到无锁（标志位为 **01**）或轻量级锁（标志位为 **00**）的状态；

偏向锁在Java 1.6之后是默认启用的，但在应用程序启动几秒钟之后才激活，可以使用`-XX:BiasedLockingStartupDelay=0`参数关闭延迟，如果
锁通常情况下处于竞争状态，可以通过`XX:-UseBiasedLocking=false`参数关闭偏向锁。

轻量级锁的获取

当关闭偏向锁功能，或多个线程竞争偏向锁导致偏向锁升级为轻量级锁，会尝试获取轻量级锁，其入口位于`ObjectSynchronizer::slow_enter`

```
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()) {//是否为无锁状态001
    // 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)) {//CAS成功，释放栈锁
      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);
}
```

1、`markOop mark = obj->mark()`方法获取对象的markOop数据mark；
2、`mark->is_neutral()`方法判断mark是否为无锁状态：mark的偏向锁标志位为 **0**，锁标志位为 **01**；
3、如果mark处于无锁状态，则进入步骤（4），否则执行步骤（6）；
4、把mark保存到BasicLock对象的_displaced_header字段；
5、通过CAS尝试将Mark Word更新为指向BasicLock对象的指针，如果更新成功，表示竞争到锁，则执行同步代码，否则执行步骤（6）；
6、如果当前mark处于加锁状态，且mark中的ptr指针指向当前线程的栈帧，则执行同步代码，否则说明有多个线程竞争轻量级锁，轻量级锁需要膨胀

假设线程A和B同时执行到临界区`if (mark->is_neutral())`：
1、线程AB都把Mark Word复制到各自的_displaced_header字段，该数据保存在线程的栈帧上，是线程私有的；
2、`Atomic::cmpxchg_ptr`原子操作保证只有一个线程可以把指向栈帧的指针复制到Mark Word，假设此时线程A执行成功，并返回继续执行同步代
3、线程B执行失败，退出临界区，通过`ObjectSynchronizer::inflate`方法开始膨胀锁；

轻量级锁的释放

轻量级锁的释放通过ObjectSynchronizer::slow_exit--->调用`ObjectSynchronizer::fast_exit`完成。

```
void ObjectSynchronizer::fast_exit(oop object, BasicLock* lock, TRAPS) {
  assert(!object->mark()->has_bias_pattern(), "should not see bias pattern here");
  // if displaced header is null, the previous enter is recursive enter, no-op
  markOop dhw = lock->displaced_header();
  markOop mark ;
  if (dhw == NULL) {
     // Recursive stack-lock.
     // Diagnostics -- Could be: stack-locked, inflating, inflated.
     mark = object->mark() ;
     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 ;
```

```
  }

  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") ;
     if ((markOop) Atomic::cmpxchg_ptr (dhw, object->mark_addr(), mark) == mark) {//成功的释放了锁
        TEVENT (fast_exit: release stacklock) ;
        return;
     }
  }

  ObjectSynchronizer::inflate(THREAD, object)->exit (true, THREAD) ;//锁膨胀升级
}
```

1、确保处于偏向锁状态时不会执行这段逻辑；
2、取出在获取轻量级锁时保存在BasicLock对象的mark数据dhw；
3、通过CAS尝试把dhw替换到当前的Mark Word，如果CAS成功，说明成功的释放了锁，否则执行步骤（4）；
4、如果CAS失败，说明有其它线程在尝试获取该锁，这时需要将该锁升级为重量级锁，并释放；

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的切换，切换成本非常高。

锁膨胀过程

锁的膨胀过程通过`ObjectSynchronizer::inflate`函数实现

```
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 (;;) {//自旋
      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已膨胀，即重量级锁
      if (mark->has_monitor()) {//判断当前是否为重量级锁
          ObjectMonitor * inf = mark->monitor() ;//获取指向ObjectMonitor的指针
          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.膨胀等待（其他线程正在从轻量级锁转为膨胀锁
      // 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栈锁（轻量级锁）
      // 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) ;//获取一个可用的ObjectMonitor
          // 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(), r
          if (cmp != mark) {//CAS失败//CAS失败，说明冲突了，自旋等待//CAS失败，说明冲突了，自旋等待//CAS失败，说明
待
             omRelease (Self, m, true) ;//释放监视器锁
             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") ;
          //CAS成功，设置ObjectMonitor的_header、_owner和_object等
          // 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 无锁
```

```
141         // TODO-FIXME: for entry we currently inflate and then try to CAS _owner.
142         // If we know we're inflating for entry it's better to inflate by swinging a
143         // pre-locked objectMonitor pointer into the object header.   A successful
144         // CAS inflates the object *and* confers ownership to the inflating thread.
145         // In the current implementation we use a 2-step mechanism where we CAS()
146         // to inflate and then CAS() again to try to swing _owner from NULL to Self.
147         // An inflateTry() method that we could call from fast_enter() and slow_enter()
148         // would be useful.
149 
150         assert (mark->is_neutral(), "invariant");
151         ObjectMonitor * m = omAlloc (Self) ;
152         // prepare m for installation - set monitor to initial state
153         m->Recycle();
154         m->set_header(mark);
155         m->set_owner(NULL);
156         m->set_object(object);
157         m->OwnerIsThread = 1 ;
158         m->_recursions   = 0 ;
159         m->_Responsible  = NULL ;
160         m->_SpinDuration = ObjectMonitor::Knob_SpinLimit ;       // consider: keep metastats by type/cla
161 
162         if (Atomic::cmpxchg_ptr (markOopDesc::encode(m), object->mark_addr(), mark) != mark) {
163             m->set_object (NULL) ;
164             m->set_owner  (NULL) ;
165             m->OwnerIsThread = 0 ;
166             m->Recycle() ;
167             omRelease (Self, m, true) ;
168             m = NULL ;
169             continue ;
170             // interference - the markword changed - just retry.
171             // The state-transitions are one-way, so there's no chance of
172             // live-lock -- "Inflated" is an absorbing state.
173         }
174 
175         // Hopefully the performance counters are allocated on distinct
176         // cache lines to avoid false sharing on MP systems ...
177         if (ObjectMonitor::_sync_Inflations != NULL) ObjectMonitor::_sync_Inflations->inc() ;
178         TEVENT(Inflate: overwrite neutral) ;
179         if (TraceMonitorInflation) {
180           if (object->is_instance()) {
181             ResourceMark rm;
182             tty->print_cr("Inflating object " INTPTR_FORMAT " , mark " INTPTR_FORMAT " , type %s",
183               (void *) object, (intptr_t) object->mark(),
184               object->klass()->external_name());
185           }
186         }
187         return m ;
188     }
189 }
```

膨胀过程的实现比较复杂，大概实现过程如下：

1、整个膨胀过程在自旋下完成；

2、`mark->has_monitor()`方法判断当前是否为重量级锁（上图18-25行），即Mark Word的锁标识位为 **10**，如果当前状态为重量级锁，执行步骤骤（4）；

3、`mark->monitor()`方法获取指向ObjectMonitor的指针，并返回，说明膨胀过程已经完成；

4、如果当前锁处于膨胀中（上图33-37行），说明该锁正在被其它线程执行膨胀操作，则当前线程就进行自旋等待锁膨胀完成，这里需要注意一点，是不会一直占用cpu资源，每隔一段时间会通过os::NakedYield方法放弃cpu资源，或通过park方法挂起；如果其他线程完成锁的膨胀操作，则退出自旋

5、如果当前是轻量级锁状态（上图58-138行），即锁标识位为 **00**，膨胀过程如下：

1. 通过omAlloc方法，获取一个可用的ObjectMonitor monitor，并重置monitor数据；
2. 通过CAS尝试将Mark Word设置为markOopDesc:INFLATING，标识当前锁正在膨胀中，如果CAS失败，说明同一时刻其它线程已经将Mark markOopDesc:INFLATING，当前线程进行自旋等待膨胀完成；
3. 如果CAS成功，设置monitor的各个字段：_header、_owner和_object等，并返回；

6、如果是无锁（中立，上图150-186行），重置监视器值；

monitor竞争

当锁膨胀完成并返回对应的monitor时，并不表示该线程竞争到了锁，真正的锁竞争发生在`ObjectMonitor::enter`方法中。

```
void ATTR ObjectMonitor::enter(TRAPS) {
  // The following code is ordered to check the most common cases first
  // and to reduce RTS->RTO cache line upgrades on SPARC and IA32 processors.
  Thread * const Self = THREAD ;
  void * cur ;

  cur = Atomic::cmpxchg_ptr (Self, &_owner, NULL) ;
  if (cur == NULL) {//CAS成功
     // Either ASSERT _recursions == 0 or explicitly set _recursions = 0.
     assert (_recursions == 0   , "invariant") ;
     assert (_owner      == Self, "invariant") ;
     // CONSIDER: set or assert OwnerIsThread == 1
     return ;
  }

  if (cur == Self) {//重入锁
```

```
    // TODO-FIXME: check for integer overflow!  BUGID 6557169.
    _recursions ++ ;
    return ;
 }

 if (Self->is_lock_owned ((address)cur)) {
   assert (_recursions == 0, "internal state error");
   _recursions = 1 ;
   // Commute owner from a thread-specific on-stack BasicLockObject address to
   // a full-fledged "Thread *".
   _owner = Self ;
   OwnerIsThread = 1 ;
   return ;
 }

 // We've encountered genuine contention.
 assert (Self->_Stalled == 0, "invariant") ;
 Self->_Stalled = intptr_t(this) ;

 // Try one round of spinning *before* enqueueing Self
 // and before going through the awkward and expensive state
 // transitions.  The following spin is strictly optional ...
 // Note that if we acquire the monitor from an initial spin
 // we forgo posting JVMTI events and firing DTRACE probes.
 if (Knob_SpinEarly && TrySpin (Self) > 0) {
    assert (_owner == Self      , "invariant") ;
    assert (_recursions == 0    , "invariant") ;
    assert (((oop)(object()))->mark() == markOopDesc::encode(this), "invariant") ;
    Self->_Stalled = 0 ;
    return ;
 }

 assert (_owner != Self          , "invariant") ;
 assert (_succ  != Self          , "invariant") ;
 assert (Self->is_Java_thread()  , "invariant") ;
 JavaThread * jt = (JavaThread *) Self ;
 assert (!SafepointSynchronize::is_at_safepoint(), "invariant") ;
 assert (jt->thread_state() != _thread_blocked   , "invariant") ;
 assert (this->object() != NULL  , "invariant") ;
 assert (_count >= 0, "invariant") ;

 // Prevent deflation at STW-time.  See deflate_idle_monitors() and is_busy().
 // Ensure the object-monitor relationship remains stable while there's contention.
 Atomic::inc_ptr(&_count);

 EventJavaMonitorEnter event;

 { // Change java thread status to indicate blocked on monitor enter.
   JavaThreadBlockedOnMonitorEnterState jtbmes(jt, this);

   DTRACE_MONITOR_PROBE(contended__enter, this, object(), jt);
   if (JvmtiExport::should_post_monitor_contended_enter()) {
     JvmtiExport::post_monitor_contended_enter(jt, this);
   }

   OSThreadContendState osts(Self->osthread());
   ThreadBlockInVM tbivm(jt);

   Self->set_current_pending_monitor(this);

   // TODO-FIXME: change the following for(;;) loop to straight-line code.
   for (;;) {
     jt->set_suspend_equivalent();
     // cleared by handle_special_suspend_equivalent_condition()
     // or java_suspend_self()

     EnterI (THREAD) ;

...省略... }
```

1、通过CAS尝试把monitor的_owner字段设置为当前线程；
2、如果设置之前的_owner指向当前线程，说明当前线程再次进入monitor，即重入锁，执行_recursions ++ ，记录重入的次数；
3、如果之前的_owner指向的地址在当前线程中，这种描述有点拗口，换一种说法：之前_owner指向的BasicLock在当前线程栈上，说明当前线程是
monitor，设置_recursions为1，_owner为当前线程，该线程成功获得锁并返回；
4、如果获取锁失败，则等待锁的释放；

monitor等待

monitor竞争失败的线程，通过自旋执行`ObjectMonitor::EnterI`方法等待锁的释放，EnterI方法的部分逻辑实现如下：

```
ObjectWaiter node(Self) ;
    Self->_ParkEvent->reset() ;
    node._prev   = (ObjectWaiter *) 0xBAD ;
    node.TState  = ObjectWaiter::TS_CXQ ;

```

```
    // Push "Self" onto the front of the _cxq.
    // Once on cxq/EntryList, Self stays on-queue until it acquires the lock.
    // Note that spinning tends to reduce the rate at which threads
    // enqueue and dequeue on EntryList|cxq.
    ObjectWaiter * nxt ;
    for (;;) {
        node._next = nxt = _cxq ;
        if (Atomic::cmpxchg_ptr (&node, &_cxq, nxt) == nxt) break ;

        // Interference - the CAS failed because _cxq changed.  Just retry.
        // As an optional optimization we retry the lock.
        if (TryLock (Self) > 0) {
            assert (_succ != Self         , "invariant") ;
            assert (_owner == Self        , "invariant") ;
            assert (_Responsible != Self  , "invariant") ;
            return ;
        }
    }
```

1、当前线程被封装成ObjectWaiter对象node，状态设置成ObjectWaiter::TS_CXQ；
2、在for循环中，通过CAS把node节点push到_cxq列表中，同一时刻可能有多个线程把自己的node节点push到_cxq列表中；
3、node节点push到_cxq列表之后，通过自旋尝试获取锁，如果还是没有获取到锁，则通过park将当前线程挂起，等待被唤醒，实现如下：

```
for (;;) {

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

        if ((SyncFlags & 2) && _Responsible == NULL) {
           Atomic::cmpxchg_ptr (Self, &_Responsible, NULL) ;
        }

        // park self
        if (_Responsible == Self || (SyncFlags & 1)) {
            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 {
            TEVENT (Inflated enter - park UNTIMED) ;
            Self->_ParkEvent->park() ;//当前线程挂起
        }

        if (TryLock(Self) > 0) break ;

        // The lock is still contested.
        // Keep a tally of the # of futile wakeups.
        // Note that the counter is not protected by a lock or updated by atomics.
        // That is by design - we trade "lossy" counters which are exposed to
        // races during updates for a lower probe effect.
        TEVENT (Inflated enter - Futile wakeup) ;
        if (ObjectMonitor::_sync_FutileWakeups != NULL) {
           ObjectMonitor::_sync_FutileWakeups->inc() ;
        }
        ++ nWakeups ;

        // Assuming this is not a spurious wakeup we'll normally find _succ == Self.
        // We can defer clearing _succ until after the spin completes
        // TrySpin() must tolerate being called with _succ == Self.
        // Try yet another round of adaptive spinning.
        if ((Knob_SpinAfterFutile & 1) && TrySpin (Self) > 0) break ;

        // We can find that we were unpark()ed and redesignated _succ while
        // we were spinning.  That's harmless.  If we iterate and call park(),
        // park() will consume the event and return immediately and we'll
        // just spin again.  This pattern can repeat, leaving _succ to simply
        // spin on a CPU.  Enable Knob_ResetEvent to clear pending unparks().
        // Alternately, we can sample fired() here, and if set, forgo spinning
        // in the next iteration.

        if ((Knob_ResetEvent & 1) && Self->_ParkEvent->fired()) {
           Self->_ParkEvent->reset() ;
           OrderAccess::fence() ;
        }
        if (_succ == Self) _succ = NULL ;

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

4、当该线程被唤醒时，会从挂起的点继续执行，通过`ObjectMonitor::TryLock`尝试获取锁，TryLock方法实现如下：

```
int ObjectMonitor::TryLock (Thread * Self) {
   for (;;) {
      void * own = _owner ;
      if (own != NULL) return 0 ;
      if (Atomic::cmpxchg_ptr (Self, &_owner, NULL) == NULL) {//CAS成功，获取锁
         // Either guarantee _recursions == 0 or set _recursions = 0.
         assert (_recursions == 0, "invariant") ;
         assert (_owner == Self, "invariant") ;
         // CONSIDER: set or assert that OwnerIsThread == 1
         return 1 ;
      }
      // The lock had been free momentarily, but we lost the race to the lock.
      // Interference -- the CAS failed.
      // We can either return -1 or retry.
      // Retry doesn't make as much sense because the lock was just acquired.
      if (true) return -1 ;
   }
}
```

其本质就是通过CAS设置monitor的_owner字段为当前线程，如果CAS成功，则表示该线程获取了锁，跳出自旋操作，执行同步代码，否则继续被挂

monitor释放

当某个持有锁的线程执行完同步代码块时，会进行锁的释放，给其它线程机会执行同步代码，在HotSpot中，通过退出monitor的方式实现锁的释放，
程，具体实现位于`ObjectMonitor::exit`方法中。

```
void ATTR ObjectMonitor::exit(bool not_suspended, TRAPS) {
   Thread * Self = THREAD ;
   if (THREAD != _owner) {
     if (THREAD->is_lock_owned((address) _owner)) {
       // Transmute _owner from a BasicLock pointer to a Thread address.
       // We don't need to hold _mutex for this transition.
       // Non-null to Non-null is safe as long as all readers can
       // tolerate either flavor.
       assert (_recursions == 0, "invariant") ;
       _owner = THREAD ;
       _recursions = 0 ;
       OwnerIsThread = 1 ;
     } else {
       // NOTE: we need to handle unbalanced monitor enter/exit
       // in native code by throwing an exception.
       // TODO: Throw an IllegalMonitorStateException ?
       TEVENT (Exit - Throw IMSX) ;
       assert(false, "Non-balanced monitor enter/exit!");
       if (false) {
          THROW(vmSymbols::java_lang_IllegalMonitorStateException());
       }
       return;
     }
   }

   if (_recursions != 0) {
     _recursions--;        // this is simple recursive enter
     TEVENT (Inflated exit - recursive) ;
     return ;
   }
...省略...
```

1、如果是重量级锁的释放，monitor中的_owner指向当前线程，即THREAD == _owner；
2、根据不同的策略（由QMode指定），从cxq或EntryList中获取头节点，通过`ObjectMonitor::ExitEpilog`方法唤醒该节点封装的线程，唤醒操
成，实现如下：

```
void ObjectMonitor::ExitEpilog (Thread * Self, ObjectWaiter * Wakee) {
   assert (_owner == Self, "invariant") ;

   // Exit protocol:
   // 1. ST _succ = wakee
   // 2. membar #loadstore|#storestore;
   // 2. ST _owner = NULL
   // 3. unpark(wakee)

   _succ = Knob_SuccEnabled ? Wakee->_thread : NULL ;
   ParkEvent * Trigger = Wakee->_event ;

   // Hygiene -- once we've set _owner = NULL we can't safely dereference Wakee again.
   // The thread associated with Wakee may have grabbed the lock and "Wakee" may be
   // out-of-scope (non-extant).
   Wakee  = NULL ;

```

```
    // Drop the lock
    OrderAccess::release_store_ptr (&_owner, NULL) ;
    OrderAccess::fence() ;                               // ST _owner vs LD in unpark()

    if (SafepointSynchronize::do_call_back()) {
       TEVENT (unpark before SAFEPOINT) ;
    }

    DTRACE_MONITOR_PROBE(contended__exit, this, object(), Self);
    Trigger->unpark() ;

    // Maintain stats and report events to JVMTI
    if (ObjectMonitor::_sync_Parks != NULL) {
       ObjectMonitor::_sync_Parks->inc() ;
    }
}
```

3、被唤醒的线程，继续执行monitor的竞争；

四.总结

本文重点介绍了Synchronized原理以及JVM对Synchronized的优化。简单来说解决三种场景：

1）只有一个线程进入临界区，偏向锁

2）多个线程交替进入临界区，轻量级锁

3）多线程同时进入临界区，重量级锁

==

参考：

《深入理解 Java 虚拟机》

JVM源码分析之synchronized实现

------------------ 掌控自己的生命轨迹，身心自由！ ------------------

标签: jdk源码剖析

好文要顶 关注我 收藏该文


只会一点java
关注 - 6
粉丝 - 46
+加关注

5

« 上一篇：jdk源码剖析一：OpenJDK-Hotspot源码包目录结构
» 下一篇：jdk源码剖析四：JDK1.7升级1.8 HashMap原理的变化

Feedback

#1楼
2017-11-30 00:19 by christoph

老哥牛逼啊，这么牛逼的文章居然没有然看啊，

#2楼[楼主]
2017-11-30 12:16 by 只会一点java

甚是欣慰

#3楼
2017-12-27 19:00 by Mr King~

感谢您的分享。对图1有些不太明白的地方。图一中，线程2，进行CAS MarkWord替换，什么时候会出现不成功的情况，什么时候会出现成功的情况