链滴

Java 高并发与多线程（三）—— synchronized 关键字的实现原理

作者：xiaoyao2102
原文链接：https://ld246.com/article/1582982506027
来源网站：链滴

synchronzied 关键字应该是 Java 并发编程中最重要的内容了，甚至没有之一。在 JDK6 以前，synchonized 关键字还代表着一把重量级锁，因此在 JUC 包里还推出了 Lock 类来替代 synchronized，不 JDK6 以后的 synchronized 经过优化，引入了偏向锁、轻量级锁和重量级锁的概念，从效率层面来已经和 Lock 类不相上下了（当然 Lock 类有公平锁非公平锁和定时功能等）。本文将从**内存对象、节码和JVM层面**去剖析synchronized关键字。

1. synchronized 在内存对象中的表现

上一篇文章中我们介绍了一个JOL工具，并且用这个工具解释了对象头中的内容。本节中我们将验证同锁级别对对象头的操作。在这里我们先回顾一下前一篇文章中的表格

```
<table>
    <tr style="background-color:#bfbfbf">
        <td>锁状态</td>
        <td colspan="2" >56 bit</td>
        <td>1 bit</td>
        <td>4 bit</td>
        <td>1 bit</td>
        <td>2 bit</td>
    </tr>
    <tr>
        <td>无锁</td>
        <td>31 bit 未用</td>
        <td>25 bit hash</td>
        <td>未用</td>
        <td>分代年龄</td>
        <td>是否偏向锁</td>
        <td>01</td>
    </tr>
    <tr>
        <td>偏向锁</td>
        <td>54 bit 线程ID</td>
        <td>2 bit Epoch</td>
        <td>未用</td>
        <td>分代年龄</td>
        <td>是否偏向锁</td>
        <td>01</td>
    </tr>
    <tr>
        <td>轻量级锁</td>
        <td colspan="5">指向栈中锁记录的指针</td>
        <td>00</td>
    </tr>
    <tr>
        <td>重量级锁</td>
        <td colspan="5">指向系统互斥量Mutex的指针 (Linux实现)</td>
        <td>10</td>
    </tr>
    <tr>
        <td>GC标记</td>
        <td colspan="5">空</td>
        <td>11</td>
    </tr>
```

</table>

1.1 偏向锁状态下的对象头

偏向锁是最轻量级的锁，是第一个申请锁的线程获得的锁类型。它的特点为**在不存在锁竞争的情况下持有锁的线程不需要重复获取锁资源**。这就像小明同学在图书馆占座，拿了本书往桌子上一放就代表了个位置，那么在没有其它同学觊觎这个位子的时候，小明同学离开位置再回来的时候并不需要重新虑这个位置是不是有人用了，直接坐下就好。

在Java中，偏向锁的获取就是这么一个过程，我们用JOL看一下。

首先我们需要设置JVM参数

```
-XX:BiasedLockingStartupDelay=0
```

这个设置原因是JVM在启动的时候，会启动很多带有synchronized关键字的线程，JVM非常明确地道这些线程之间一定会存在锁竞争的情况。在启动的如果还是使用偏向锁的话，会导致许多锁膨胀的况，因此JVM会直接使用轻量级锁，然后再恢复偏向锁的使用（默认5s）。

代码如下：

```
import org.openjdk.jol.info.ClassLayout;

public class Test {

    public static void main(String[] args) {
        Object o = new Object();

        System.out.println(ClassLayout.parseInstance(o).toPrintable());

        synchronized (o) {
            System.out.println(ClassLayout.parseInstance(o).toPrintable());
        }
    }
}
```

输出为

```
java.lang.Object object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           05 00 00 00 (00000101 00000000 00000000 00
00000) (5)
      4     4        (object header)                           00 00 00 00 (00000000 00000000 00000000 00
00000) (0)
      8     4        (object header)                           e5 01 00 20 (11100101 00000001 00000000 00
00000) (536871397)
     12     4        (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

java.lang.Object object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           05 e8 ea 00 (00000101 11101000 11101010 00
00000) (15394821)
```

```
     4     4        (object header)                           00 00 00 00 (00000000 00000000 00000000 00
00000) (0)
     8     4        (object header)                           e5 01 00 20 (11100101 00000001 00000000 00
00000) (536871397)
    12     4        (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

我们可以看到，在synchronized锁定对象后输出的对象头内容，第一组bit的最后三位变成了101，并其它位也有变化（其实就是线程ID），表示此时该线程获取到了偏向锁。

当然肯定有同学也发现了一些异象，就是第一次输出对象头中也有101的情况但线程ID为0。这里其是JVM的一种优化，叫做**匿名偏向锁**，即对象创建了就默认为偏向锁状态，但是并没有线程ID写入，时第一个来请求锁的线程只需要把自己的线程ID写入就好了。

1.2 轻量级锁状态下的对象头

轻量级锁也叫自旋锁，当偏向锁遇到竞争情况的时候，对象的锁级别便会升级为轻量级锁。

首先，为了排除JVM偏向锁延迟生效而产生轻量级锁情况，我们还是设置JVM参数：

```
-XX:BiasedLockingStartupDelay=0
```

同时，我们需要创建一个**轻微竞争**的场景，避免锁膨胀为重量级锁。

```
import org.openjdk.jol.info.ClassLayout;

public class Test {

    public static void main(String[] args) {
        Object o = new Object();

        Thread t1 = new Thread(() -> {
            synchronized (o) {
                // The body has to be easy enough
                // Printing object header will raise competition level for lock
            }
        });

        Thread t2 = new Thread(() -> {
            synchronized (o) {
                System.out.println(ClassLayout.parseInstance(o).toPrintable());
            }
        });

        t1.start();
        t2.start();
    }
}
```

输出为：

```
java.lang.Object object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
```

```
     0     4        (object header)                           48 f7 19 1b (01001000 11110111 00011001 000
1011) (454686536)
     4     4        (object header)                           00 00 00 00 (00000000 00000000 00000000 00
00000) (0)
     8     4        (object header)                           e5 01 00 20 (11100101 00000001 00000000 00
00000) (536871397)
    12     4        (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

可以看到，输出中第一组bit的最后三位是000，查表可知现在锁状态为轻量级锁。其余56bit，是一指向自己线程栈帧内一个叫**Lock Record**的记录的指针。当一个线程去获取轻量级锁的时候，主要会以下几步：

1. 在线程内创建一个叫Lock Record的空间，并且将对象的Mark Word复制到这块空间内，被称为Displaced Mark Word。
2. 使用CAS操作，尝试将对象原Mark Word替换为指向自己栈帧内Lock Record的指针。（这也是被为自旋锁的原因）
3. 如果替换成功，则获取到了对象的轻量级锁。

1.3 重量级锁状态下的对象头

当同步资源竞争加剧的时候，轻量级锁会膨胀（inflate）为重量级锁，申请重量级锁的线程首先会阻并释放CPU资源，直到获取到锁之后再重新运行。

至于什么时候属于竞争加剧，有以下两个条件，达成其一即可：

1. 轻量级锁超过10次自旋，这个也可以通过-XX:PreLockSpin来指定。在JDK6以后，有自适应自旋（daptive Self Spin）来控制。
2. 自旋线程数超过CPU核数的一半。

重量级锁很容易复现，只要synchronized块中代码运行足够的时间即可

```
import org.openjdk.jol.info.ClassLayout;

public class Test {


    public static void main(String[] args) {
        Object o = new Object();

        Thread t1 = new Thread(() -> {
            synchronized (o) {
                try {
                    Thread.sleep(10L);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        Thread t2 = new Thread(() -> {
```

```
            synchronized (o) {
                System.out.println(ClassLayout.parseInstance(o).toPrintable());
            }
        });

        t1.start();
        t2.start();

    }
}
```

输出为：

```
java.lang.Object object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           2a f5 f5 17 (00101010 11110101 11110101 000
0111) (401995050)
      4     4        (object header)                           00 00 00 00 (00000000 00000000 00000000 00
00000) (0)
      8     4        (object header)                           e5 01 00 20 (11100101 00000001 00000000 00
00000) (536871397)
     12     4        (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

第一组bit值的最后三位变成了010，即当前对象锁等级为重量级锁。而其余的bit值部分，则是指向系互斥量（mutex）的指针。

这里的互斥量（Mutex），可以简单理解为一把系统级别的锁，需要程序向操作系统申请。因此互斥的资源十分有限，每次申请都需要在系统内的一个队列中排队等待。这就是重量级锁效率较低的原因。

2. synchronized关键字在字节码中的表现

Java源码在经过javac的编译后变为字节码文件，然后虚拟机解释执行字节码文件。对于synchronize关键字来说，此时虚拟机还不知道对象的锁级别，因此编译后的字节码文件应该是一样的。我们来看下如下代码的字节码文件：

```
public class Test {

    public static void main(String[] args) {
        Object o = new Object();

        synchronized (o) {
        }
    }
}
```

读者可以使用IDEA的Bytecode Viewer工具或者用如下命令：

```
javap -c -p Test.class
```

字节码输出如下：

```
Compiled from "Test.java"
```

```
public class Test {
  public Test();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: new           #2                  // class java/lang/Object
       3: dup
       4: invokespecial #1                  // Method java/lang/Object."<init>":()V
       7: astore_1
       8: aload_1
       9: dup
      10: astore_2
      11: monitorenter
      12: aload_2
      13: monitorexit
      14: goto          22
      17: astore_3
      18: aload_2
      19: monitorexit
      20: aload_3
      21: athrow
      22: return
    Exception table:
       from    to  target type
          12    14    17   any
          17    20    17   any
}
```

我们可以看到，字节码中第11行的**monitorenter**就是进入同步块的指令，13行的**monitorexit**就是出同步块的指令。至于19行的monitorexit，笔者猜测是同步块中抛异常后的退出指令（错了不要打）。

3. synchronized关键字在JVM中的执行过程

本节将深入到JVM的C++源码中探究synchronized关键字是如何执行的。（当然笔者能力有限，也不到哪去。。。笑哭脸.jpg）

3.1 进入同步块

我们的探究起点还是monitorenter指令，在interpreterRuntime.cpp文件中，有一个monitorenter数，我截取了最关键的部分，是一个if-else判断，代码如下：

```
  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);
  }
```

可以看出，在使用偏向锁的情况下，JVM执行fast_enter函数，否则执行slow_enter函数。我们先从fast_enter入手，在synchronizer.cpp文件中可以找到对应函数。

```
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, T
READ);
      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");
 }

 slow_enter (obj, lock, THREAD) ;
}
```

这里面我们也只需要关注两部分，首先是return前的revoke_and_rebias函数。这个函数内容很长，此不贴出来了，大意为在某些条件下可以成功设置偏向锁。在里面有很多之前提到过的操作，比如CA操作和修改对象头等，有兴趣的同学可以查看biasedLocking.cpp文件。如果这个函数返回值为BIAS_EVOKED_AND_REBIASED，则直接返回，否则将调用slow_enter函数。

那么slow_enter又做了什么呢？我们继续看

```
// -----------------------------------------------------------------------------
// Interpreter/Compiler Slow Case
// This routine is used to handle interpreter/compiler slow case
// We don't need to use fast path here, because it must have been
// failed in the interpreter/compiler code.
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;
  }
```

```
  // 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);
}
```

记住，我们如果进行到slow_enter函数，说明我们没有使用偏向锁或者在fast_enter设置偏向锁失败，因此此时应该使用轻量级锁。所以在上面代码中，我们可以找到一些用CAS操作设置对象头等。如以上操作都没有成功的话，那代码的最后一行便提示了我们，轻量级锁膨胀（inflate）为重量级锁。

3.2 退出同步块

我们还是回到interpreterRuntime.cpp中，找到monitorexit函数。

```
//%note monitor_1
IRT_ENTRY_NO_ASYNC(void, InterpreterRuntime::monitorexit(JavaThread* thread, BasicObject
ock* elem))
#ifdef ASSERT
  thread->last_frame().interpreter_frame_verify_monitor(elem);
#endif
  Handle h_obj(thread, elem->obj());
  assert(Universe::heap()->is_in_reserved_or_null(h_obj()),
         "must be NULL or an object");
  if (elem == NULL || h_obj()->is_unlocked()) {
    THROW(vmSymbols::java_lang_IllegalMonitorStateException());
  }
  ObjectSynchronizer::slow_exit(h_obj(), elem->lock(), thread);
  // Free entry. This must be done here, since a pending exception might be installed on
  // exit. If it is not cleared, the exception handling code will try to unlock the monitor again.
  elem->set_obj(NULL);
#ifdef ASSERT
  thread->last_frame().interpreter_frame_verify_monitor(elem);
#endif
IRT_END
```

这里面核心的内容是在中间部分，调用了slow_exit函数

```
void ObjectSynchronizer::slow_exit(oop object, BasicLock* lock, TRAPS) {
  fast_exit (object, lock, THREAD) ;
}
```

slow_exit本质是调用fast_exit函数。

```
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) {
      TEVENT (fast_exit: release stacklock) ;
      return;
    }
  }

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

这段代码关键在两个if块内。第一个if块的情况为Displaced Mark Word为空，说明是偏向锁，则做些简单验证操作就可以返回。第二个if块内为把Displaced Mark Word写回对象头，如果成功就返回否则说明有竞争存在，将锁膨胀为重量级锁然后退出。

4. 总结一下

1. synchronized关键字会修改被锁对象的对象头

2. 锁级别从低到高为：偏向锁 -> 轻量级锁 -> 重量级锁。偏向锁适用于无竞争时候的对象同步，一发生竞争，则先升级为轻量级锁。如果竞争加剧，则再升级为重量级锁。

3. synchronized关键字在字节码层面的体现为monitorenter和monitorexit

4. JVM层面，synchronized关键字主要由fast_enter、slow_enter、fast_exit和inflate几个方法支撑而这几个方法，是由cmpxchg函数支撑的。