CAS

Compare And Swap (Compare And Exchange) / 自旋 / 自旋锁 / 无锁

因为经常配合循环操作，直到完成为止，所以泛指一类操作

cas(v, a, b)，变量v，期待值a, 修改值b

ABA问题，你的女朋友在离开你的这段儿时间经历了别的人，自旋就是你空转等待，一直等到她接纳你为止

解决办法（版本号 AtomicStampedReference），基础类型简单值不需要版本号

Unsafe

AtomicInteger:

```
public final int incrementAndGet() {
        for (;;) {
            int current = get();
            int next = current + 1;
            if (compareAndSet(current, next))
                return next;
        }
    }

public final boolean compareAndSet(int expect, int update) {
        return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
    }
```

Unsafe:

```
public final native boolean compareAndSwapInt(Object var1, long var2, int
var4, int var5);
```

运用:

```
package com.mashibing.jol;

import sun.misc.Unsafe;

import java.lang.reflect.Field;

public class T02_TestUnsafe {

    int i = 0;
    private static T02_TestUnsafe t = new T02_TestUnsafe();

    public static void main(String[] args) throws Exception {
        //Unsafe unsafe = Unsafe.getUnsafe();

        Field unsafeField = Unsafe.class.getDeclaredFields()[0];
```

```
        unsafeField.setAccessible(true);
        Unsafe unsafe = (Unsafe) unsafeField.get(null);

        Field f = TO2_TestUnsafe.class.getDeclaredField("i");
        long offset = unsafe.objectFieldOffset(f);
        System.out.println(offset);

        boolean success = unsafe.compareAndSwapInt(t, offset, 0, 1);
        System.out.println(success);
        System.out.println(t.i);
        //unsafe.compareAndSwapInt()
    }
}
```

jdk8u: unsafe.cpp:

cmpxchg = compare and exchange

```
UNSAFE_ENTRY(jboolean, Unsafe_CompareAndSwapInt(JNIEnv *env, jobject unsafe,
jobject obj, jlong offset, jint e, jint x))
  UnsafeWrapper("Unsafe_CompareAndSwapInt");
  oop p = JNIHandles::resolve(obj);
  jint* addr = (jint *) index_oop_from_field_offset_long(p, offset);
  return (jint)(Atomic::cmpxchg(x, addr, e)) == e;
UNSAFE_END
```

jdk8u: atomic_linux_x86.inline.hpp

is_MP = Multi Processor

```
inline jint     Atomic::cmpxchg    (jint     exchange_value, volatile jint*
  dest, jint     compare_value) {
  int mp = os::is_MP();
  __asm__ volatile (LOCK_IF_MP(%4) "cmpxchgl %1,(%3)"
                    : "=a" (exchange_value)
                    : "r" (exchange_value), "a" (compare_value), "r" (dest),
"r" (mp)
                    : "cc", "memory");
  return exchange_value;
}
```

jdk8u: os.hpp is_MP()

```
  static inline bool is_MP() {
    // During bootstrap if _processor_count is not yet initialized
    // we claim to be MP as that is safest. If any platform has a
    // stub generator that might be triggered in this phase and for
    // which being declared MP when in fact not, is a problem - then
    // the bootstrap routine for the stub generator needs to check
    // the processor count directly and leave the bootstrap routine
    // in place until called after initialization has ocurred.
    return (_processor_count != 1) || AssumeMP;
  }
```

jdk8u: atomic_linux_x86.inline.hpp

```
#define LOCK_IF_MP(mp) "cmp $0, " #mp "; je 1f; lock; 1: "
```

最终实现：

cmpxchg = cas修改变量值

```
lock cmpxchg 指令
```

硬件：

lock指令在执行后面指令的时候锁定一个北桥信号

（不采用锁总线的方式）

markword

工具：JOL = Java Object Layout

```
<dependencies>
        <!-- https://mvnrepository.com/artifact/org.openjdk.jol/jol-core -->
        <dependency>
            <groupId>org.openjdk.jol</groupId>
            <artifactId>jol-core</artifactId>
            <version>0.9</version>
        </dependency>
    </dependencies>
```

jdk8u: markOop.hpp

```
// 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 object)
//  unused:21 size:35 -->| cms_free:1 unused:7 ------------------>| (COOPs && CMS free block)
```

synchronized的横切面详解

1. synchronized原理
2. 升级过程
3. 汇编实现
4. vs reentrantLock的区别

java源码层级

synchronized(o)

字节码层级

monitorenter moniterexit

JVM层级（Hotspot）

```
package com.mashibing.insidesync;

import org.openjdk.jol.info.ClassLayout;

public class T01_Sync1 {


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

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

```
com.mashibing.insidesync.T01_Sync1$Lock object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4   (object header)  05 00 00 00 (00000101 00000000 00000000
00000000) (5)
      4     4   (object header)  00 00 00 00 (00000000 00000000 00000000
00000000) (0)
      8     4   (object header)  49 ce 00 20 (01001001 11001110 00000000
00100000) (536923721)
     12     4        (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

```
com.mashibing.insidesync.T02_Sync2$Lock object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4   (object header)  05 90 2e 1e (00000101 10010000 00101110
00011110) (506368005)
      4     4   (object header)  1b 02 00 00 (00011011 00000010 00000000
00000000) (539)
      8     4   (object header)  49 ce 00 20 (01001001 11001110 00000000
00100000) (536923721)
     12     4        (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes tota
```

InterpreterRuntime:: monitorenter方法

```
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) {
    // 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
```

synchronizer.cpp

revoke_and_rebias

```

```

```
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");
 }

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

```
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());
```

```
35    ObjectSynchronizer::inflate(THREAD, obj())->enter(THREAD);
36  }
```

inflate方法：膨胀为重量级锁

锁升级过程

JDK8 markword实现表：

 markword

无锁 - 偏向锁 - 轻量级锁 （自旋锁，自适应自旋）- 重量级锁

synchronized优化的过程和markword息息相关

用markword中最低的三位代表锁状态 其中1位是偏向锁位 两位是普通锁位

1. Object o = new Object()
   锁 = 0 01 无锁态
2. o.hashCode()
   001 + hashcode

   ```
   00000001 10101101 00110100 00110110
   01011001 00000000 00000000 00000000
   ```

   little endian big endian

   00000000 00000000 00000000 01011001 00110110 00110100 10101101 00000000
3. 默认synchronized(o)
   00 -> 轻量级锁
   默认情况 偏向锁有个时延，默认是4秒
   why? 因为JVM虚拟机自己有一些默认启动的线程，里面有好多sync代码，这些sync代码启动时就知道肯定会有竞争，如果使用偏向锁，就会造成偏向锁不断的进行锁撤销和锁升级的操作，效率较低。

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

4. 如果设定上述参数
   new Object () - > 101 偏向锁 ->线程ID为0 -> Anonymous BiasedLock
   打开偏向锁，new出来的对象，默认就是一个可偏向匿名对象101
5. 如果有线程上锁
   上偏向锁，指的就是，把markword的线程ID改为自己线程ID的过程
   偏向锁不可重偏向 批量偏向 批量撤销
6. 如果有线程竞争
   撤销偏向锁，升级轻量级锁
   线程在自己的线程栈生成LockRecord ， 用CAS操作将markword设置为指向自己这个线程的LR的指针，设置成功者得到锁

7. 如果竞争加剧
   竞争加剧：有线程超过10次自旋， -XX:PreBlockSpin， 或者自旋线程数超过CPU核数的一半，1.6之后，加入自适应自旋 Adapative Self Spinning ， JVM自己控制
   升级重量级锁：-> 向操作系统申请资源，linux mutex , CPU从3级-0级系统调用，线程挂起，进入等待队列，等待操作系统的调度，然后再映射回用户空间

(以上实验环境是JDK11，打开就是偏向锁，而JDK8默认对象头是无锁)

偏向锁默认是打开的，但是有一个时延，如果要观察到偏向锁，应该设定参数

没错，我就是厕所所长

加锁，指的是锁定对象

锁升级的过程

JDK较早的版本 OS的资源 互斥量 用户态 -> 内核态的转换 重量级 效率比较低

现代版本进行了优化

无锁 - 偏向锁 -轻量级锁（自旋锁）-重量级锁

偏向锁 - markword 上记录当前线程指针，下次同一个线程加锁的时候，不需要争用，只需要判断线程指针是否同一个，所以，偏向锁，偏向加锁的第一个线程 。hashCode备份在线程栈上 线程销毁，锁降级为无锁

有争用 - 锁升级为轻量级锁 - 每个线程有自己的LockRecord在自己的线程栈上，用CAS去争用markword的LR的指针，指针指向哪个线程的LR，哪个线程就拥有锁

自旋超过10次，升级为重量级锁 - 如果太多线程自旋 CPU消耗过大，不如升级为重量级锁，进入等待队列（不消耗CPU）-XX:PreBlockSpin

自旋锁在 JDK1.4.2 中引入，使用 -XX:+UseSpinning 来开启。JDK 6 中变为默认开启，并且引入了自适应的自旋锁（适应性自旋锁）。

自适应自旋锁意味着自旋的时间（次数）不再固定，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。如果在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也是很有可能再次成功，进而它将允许自旋等待持续相对更长的时间。如果对于某个锁，自旋很少成功获得过，那在以后尝试获取这个锁时将可能省略掉自旋过程，直接阻塞线程，避免浪费处理器资源。

偏向锁由于有锁撤销的过程revoke，会消耗系统资源，所以，在锁争用特别激烈的时候，用偏向锁未必效率高。还不如直接使用轻量级锁。

synchronized最底层实现

```
1
2  public class T {
3      static volatile int i = 0;
4
5      public static void n() { i++; }
6
7      public static synchronized void m() {}
```

```
8
9       publics static void main(String[] args) {
10          for(int j=0; j<1000_000; j++) {
11              m();
12              n();
13          }
14      }
15  }
16
```

java -XX:+UnlockDiagonositicVMOptions -XX:+PrintAssembly T

C1 Compile Level 1 (一级优化)

C2 Compile Level 2 (二级优化)

找到m() n()方法的汇编码，会看到 lock comxchg .....指令

synchronized vs Lock (CAS)

```
1  在高争用 高耗时的环境下synchronized效率更高
2  在低争用 低耗时的环境下CAS效率更高
3  synchronized到重量级之后是等待队列（不消耗CPU）
4  CAS（等待期间消耗CPU）
5
6  一切以实测为准
```

锁消除 lock eliminate

```
1  public void add(String str1,String str2){
2           StringBuffer sb = new StringBuffer();
3           sb.append(str1).append(str2);
4  }
```

我们都知道 StringBuffer 是线程安全的，因为它的关键方法都是被 synchronized 修饰过的，但我们看上面这段代码，我们会发现，sb 这个引用只会在 add 方法中使用，不可能被其它线程引用（因为是局部变量，栈私有），因此 sb 是不可能共享的资源，JVM 会自动消除 StringBuffer 对象内部的锁。

锁粗化 lock coarsening

```
1  public String test(String str){
2
3          int i = 0;
4          StringBuffer sb = new StringBuffer():
5          while(i < 100){
6              sb.append(str);
7              i++;
8          }
9          return sb.toString():
10 }
```

JVM 会检测到这样一连串的操作都对同一个对象加锁（while 循环内 100 次执行 append，没有锁粗化的就要进行 100 次加锁/解锁），此时 JVM 就会将加锁的范围粗化到这一连串的操作的外部（比如 while 虚幻体外），使得这一连串操作只需要加一次锁即可。

锁降级（不重要）

https://www.zhihu.com/question/63859501

其实，只被VMThread访问，降级也就没啥意义了。所以可以简单认为锁降级不存在！

超线程

一个ALU + 两组Registers + PC

参考资料

http://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html