深入理解Java内存模型（五）——锁

作者 程晓明 程晓明 关注 82 他的粉丝 发布于 2013年3月6日. 估计阅读时间: 23 分钟 来 ArchSummit北京2018 共同探讨机器学习、信息安全、微服务治理的关键点

锁的释放-获取建立的happens before 关系

锁是java并发编程中最重要的同步机制。锁除了让临界区互斥执行外，还可以让释放锁的线程向获取同一个锁的线程发送消息。

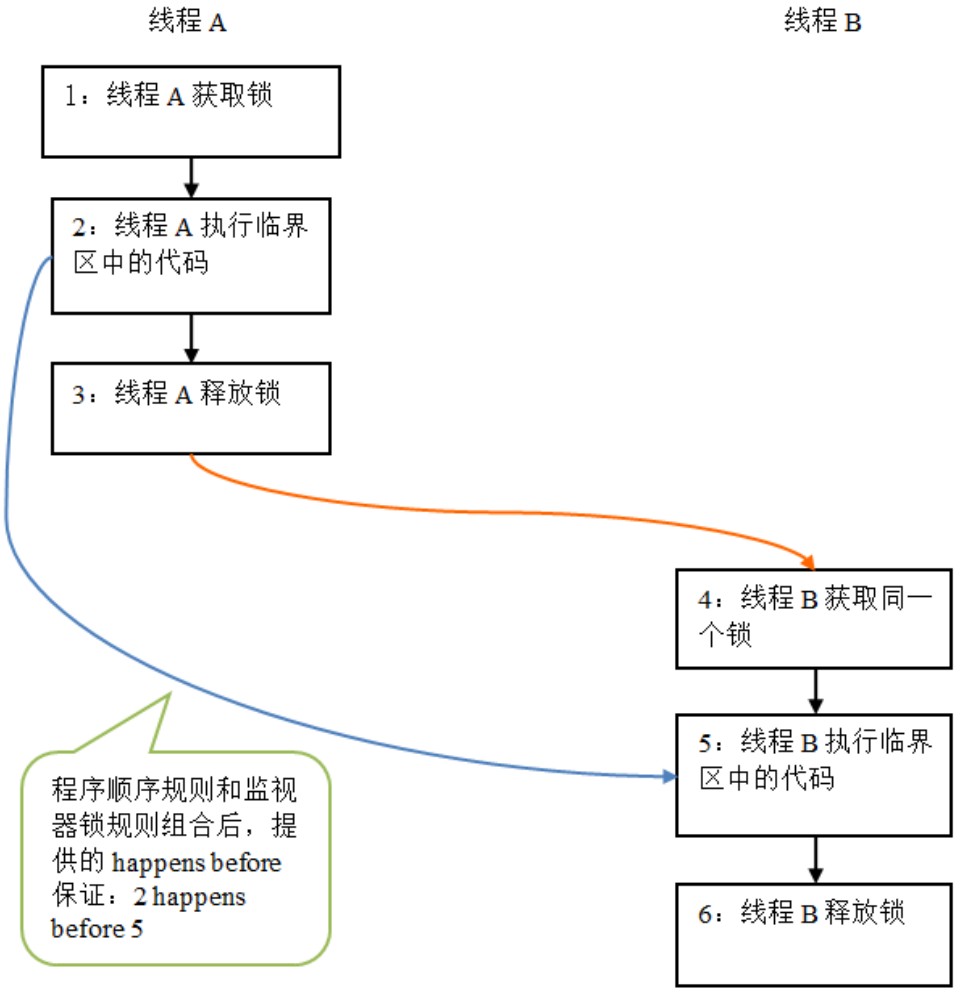
下面是锁释放-获取的示例代码：

```
class MonitorExample {
    int a = 0;

    public synchronized void writer() {  //1
        a++;                             //2
    }                                    //3

    public synchronized void reader() {  //4
        int i = a;                       //5
        ……
    }                                    //6
}
```

假设线程A执行writer()方法，随后线程B执行reader()方法。根据happens before规则，这个过程包含的happens before 关系可以分为两类：

1. 根据程序次序规则，1 happens before 2, 2 happens before 3; 4 happens before 5, 5 happens before 6。
2. 根据监视器锁规则，3 happens before 4。
3. 根据happens before 的传递性，2 happens before 5。

上述happens before 关系的图形化表现形式如下：

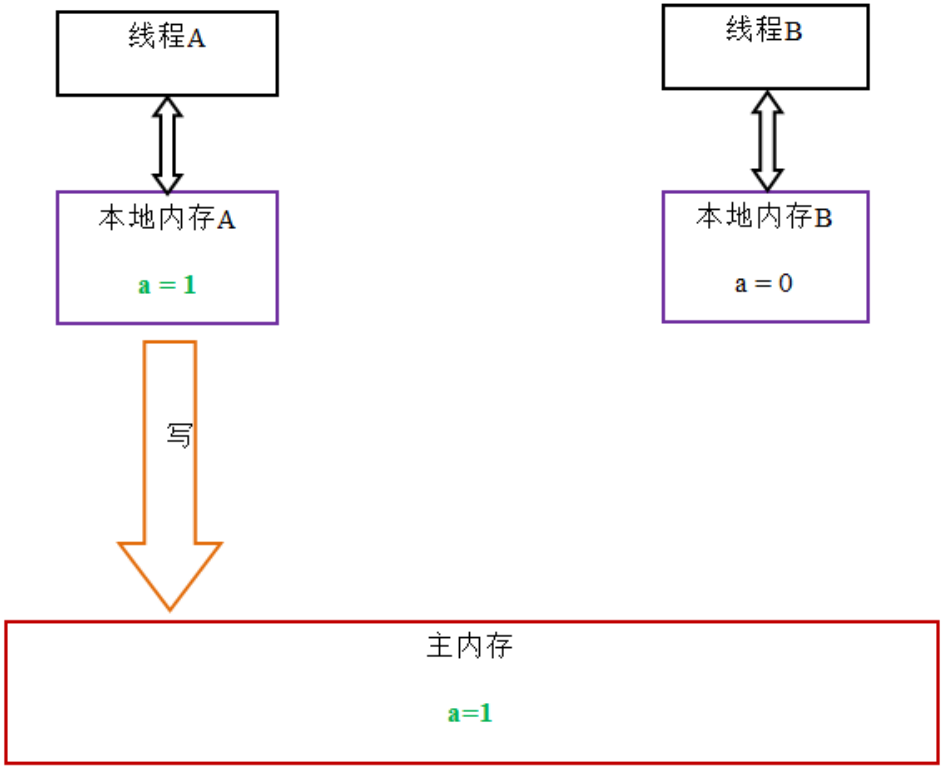


在上图中，每一个箭头链接的两个节点，代表了一个happens before 关系。黑色箭头表示程序顺序规则；橙色箭头表示监视器锁规则；蓝色箭头表示组合这些规则后提供的happens before保证。

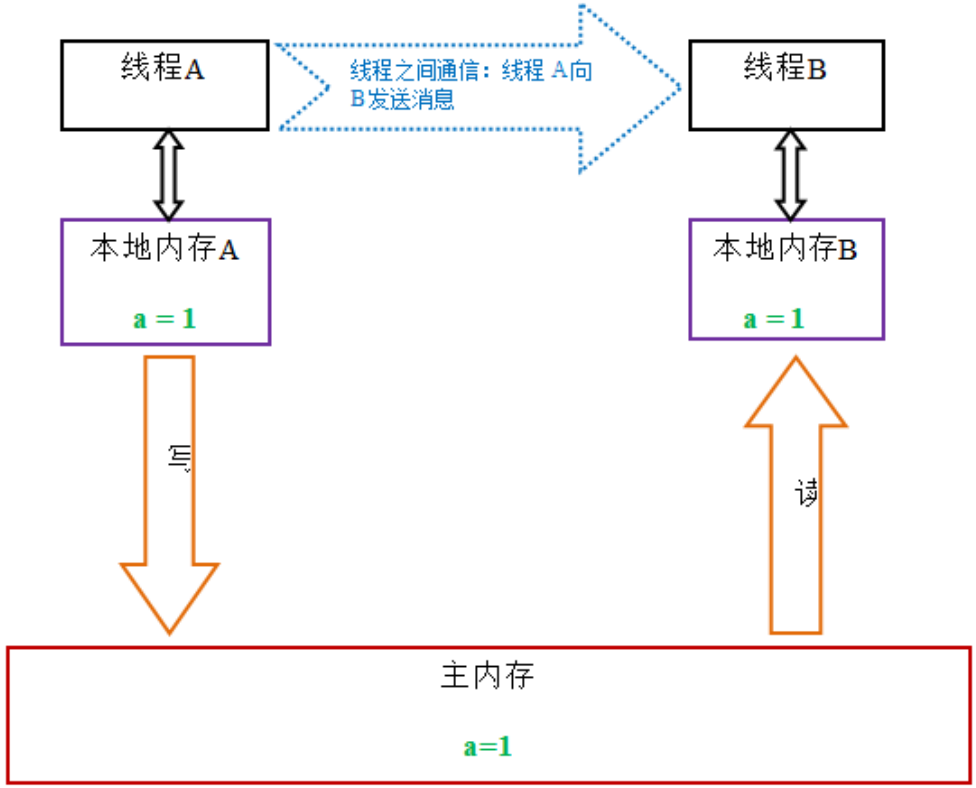
上图表示在线程A释放了锁之后，随后线程B获取同一个锁。在上图中，2 happens before 5。因此，线程A在释放锁之前所有可见的共享变量，在线程B获取同一个锁之后，将立刻变得对B线程可见。

锁释放和获取的内存语义

当线程释放锁时，JMM会把该线程对应的本地内存中的共享变量刷新到主内存中。以上面的MonitorExample程序为例，A线程释放锁后，共享数据的状态示意图如下：



当线程获取锁时，JMM会把该线程对应的本地内存置为无效。从而使得被监视器保护的临界区代码必须要从主内存中去读取共享变量。下面是锁获取的状态示意图：



对比锁释放-获取的内存语义与volatile写-读的内存语义，可以看出：锁释放与volatile写有相同的内存语义；锁获取与volatile读有相同的内存语义。

下面对锁释放和锁获取的内存语义做个总结：

- 线程A释放一个锁，实质上是线程A向接下来将要获取这个锁的某个线程发出了（线程A对共享变量所做修改的）消息。

- 线程B获取一个锁，实质上是线程B接收了之前某个线程发出的（在释放这个锁之前对共享变量所做修改的）消息。
- 线程A释放锁，随后线程B获取这个锁，这个过程实质上是线程A通过主内存向线程B发送消息。

锁内存语义的实现

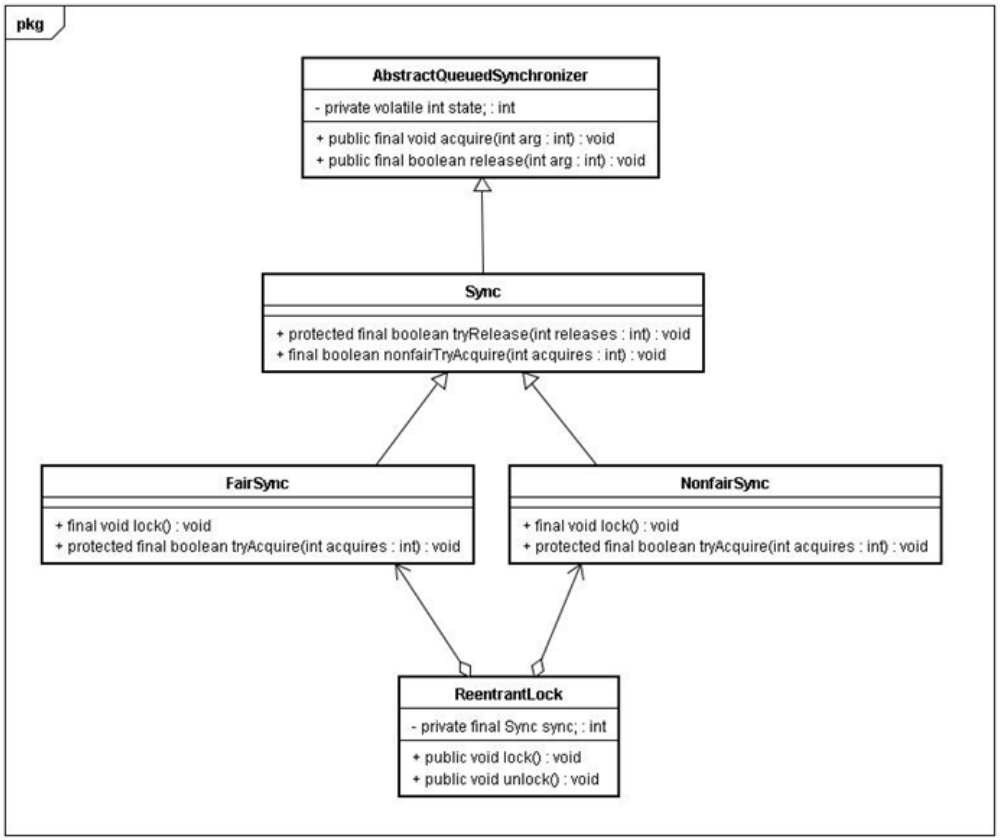
本文将借助ReentrantLock的源代码，来分析锁内存语义的具体实现机制。

请看下面的示例代码：

```
class ReentrantLockExample {
int a = 0;
ReentrantLock lock = new ReentrantLock();

public void writer() {
    lock.lock();         //获取锁
    try {
        a++;
    } finally {
        lock.unlock();  //释放锁
    }
}

public void reader () {
    lock.lock();        //获取锁
    try {
        int i = a;
        ……
    } finally {
        lock.unlock();  //释放锁
    }
}
}
```

在ReentrantLock中，调用lock()方法获取锁；调用unlock()方法释放锁。

ReentrantLock的实现依赖于java同步器框架AbstractQueuedSynchronizer（本文简称之为AQS）。AQS使用一个整型的volatile变量（命名为state）来维护同步状态，马上我们会看到，这个volatile变量是ReentrantLock内存语义实现的关键。 下面是ReentrantLock的类图（仅画出与本文相关的部分）：



ReentrantLock分为公平锁和非公平锁，我们首先分析公平锁。

使用公平锁时，加锁方法lock()的方法调用轨迹如下：

1. ReentrantLock : lock()
2. FairSync : lock()
3. AbstractQueuedSynchronizer : acquire(int arg)
4. ReentrantLock : tryAcquire(int acquires)

在第4步真正开始加锁，下面是该方法的源代码：

```
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();   //获取锁的开始，首先读volatile变量state
    if (c == 0) {
        if (isFirst(current) &&
            compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

从上面源代码中我们可以看出，加锁方法首先读volatile变量state。

在使用公平锁时，解锁方法unlock()的方法调用轨迹如下：

1. ReentrantLock : unlock()
2. AbstractQueuedSynchronizer : release(int arg)
3. Sync : tryRelease(int releases)

在第3步真正开始释放锁，下面是该方法的源代码：

```
protected final boolean tryRelease(int releases) {
    int c = getState() - releases;
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);           //释放锁的最后，写volatile变量state
    return free;
}
```

从上面的源代码我们可以看出，在释放锁的最后写volatile变量state。

公平锁在释放锁的最后写volatile变量state；在获取锁时首先读这个volatile变量。根据volatile的happens-before规则，释放锁的线程在写volatile变量之前可见的共享变量，在获取锁的线程读取同一个volatile变量后将立即变的对获取锁的线程可见。

现在我们分析非公平锁的内存语义的实现。

非公平锁的释放和公平锁完全一样，所以这里仅仅分析非公平锁的获取。

使用公平锁时，加锁方法lock()的方法调用轨迹如下：

1. ReentrantLock : lock()
2. NonfairSync : lock()
3. AbstractQueuedSynchronizer : compareAndSetState(int expect, int update)

在第3步真正开始加锁，下面是该方法的源代码：

```
protected final boolean compareAndSetState(int expect, int update)
{
    return unsafe.compareAndSwapInt(this, stateOffset, expect,
update);
}
```

该方法以原子操作的方式更新state变量，本文把java的compareAndSet()方法调用简称为CAS。JDK文档对该方法的说明如下：如果当前状态值等于预期值，则以原子方式将同步状态设置为给定的更新值。此操作具有 volatile 读和写的内存语义。

这里我们分别从编译器和处理器的角度来分析,CAS如何同时具有volatile读和volatile写的内存语义。

前文我们提到过，编译器不会对volatile读与volatile读后面的任意内存操作重排序；编译器不会对volatile写与volatile写前面的任意内存操作重排序。组合这两个条件，意味着为了同时实现volatile读和volatile写的内存语义，编译器不能对CAS与CAS前面和后面的任意内存操作重排序。

下面我们来分析在常见的intel x86处理器中，CAS是如何同时具有volatile读和volatile写的内存语义的。

下面是sun.misc.Unsafe类的compareAndSwapInt()方法的源代码：

```
public final native boolean compareAndSwapInt(Object o, long
offset,
                                               int expected,
                                               int x);
```

可以看到这是个本地方法调用。这个本地方法在openjdk中依次调用的c++代码为：unsafe.cpp，atomic.cpp和atomic*windows*x86.inline.hpp。这个本地方法的最终实现在openjdk的如下位置：openjdk-7-fcs-src-b147-27*jun*2011\openjdk\hotspot\src\os*cpu**windows*x86\vm\atomic*windows*x86.inline.hpp（对应于windows操作系统，X86处理器）。下面是对应于intel x86处理器的源代码的片段：

```
// Adding a lock prefix to an instruction on MP machine
// VC++ doesn't like the lock prefix to be on a single line
// so we can't insert a label after the lock prefix.
// By emitting a lock prefix, we can define a label after it.
#define LOCK_IF_MP(mp) __asm cmp mp, 0  \
                       __asm je L0      \
                       __asm _emit 0xF0 \
                       __asm L0:

inline jint     Atomic::cmpxchg    (jint     exchange_value,
volatile jint*     dest, jint     compare_value) {
  // 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
  }
}
```

如上面源代码所示，程序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器上运行，就为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性，不需要lock前缀提供的内存屏障效果）。

intel的手册对lock前缀的说明如下：

1. 确保对内存的读-改-写操作原子执行。在Pentium及Pentium之前的处理器中，带有lock前缀的指令在执行期间会锁住总线，使得其他处理器暂时无法通过总线访问内存。很显然，这会带来昂贵的开销。从Pentium 4，Intel Xeon及P6处理器开始，intel在原有总线锁的基础上做了一个很有意义的优化：如果要访问的内存区域（area of memory）在lock前缀指令执行期间已经在处理器内部的缓存中被锁定（即包含该内存区域的缓存行当前处于独占或以修改状态），并且该内存区域被完全包含在单个缓存行（cache line）中，那么处理器将直接执行该指令。由于在指令执行期间该缓存行会一直被锁定，其它处理器无法读/写该指令要访问的内存区域，因此能保证指令执行的原子性。这个操作过程叫做缓存锁定（cache locking），缓存锁定将大大降低lock前缀指令的执行开销，但是当多处理器之间的竞争程度很高或者指令访问的内存地址未对齐时，仍然会锁住总线。
2. 禁止该指令与之前和之后的读和写指令重排序。
3. 把写缓冲区中的所有数据刷新到内存中。

上面的第2点和第3点所具有的内存屏障效果，足以同时实现volatile读和volatile写的内存语义。

经过上面的这些分析，现在我们终于能明白为什么JDK文档说CAS同时具有volatile读和volatile写的内存语义了。

现在对公平锁和非公平锁的内存语义做个总结：

- 公平锁和非公平锁释放时，最后都要写一个volatile变量state。
- 公平锁获取时，首先会去读这个volatile变量。
- 非公平锁获取时，首先会用CAS更新这个volatile变量,这个操作同时具有volatile读和volatile写的内存语义。

从本文对ReentrantLock的分析可以看出，锁释放-获取的内存语义的实现至少有下面两种方式：

1. 利用volatile变量的写-读所具有的内存语义。
2. 利用CAS所附带的volatile读和volatile写的内存语义。

concurrent包的实现

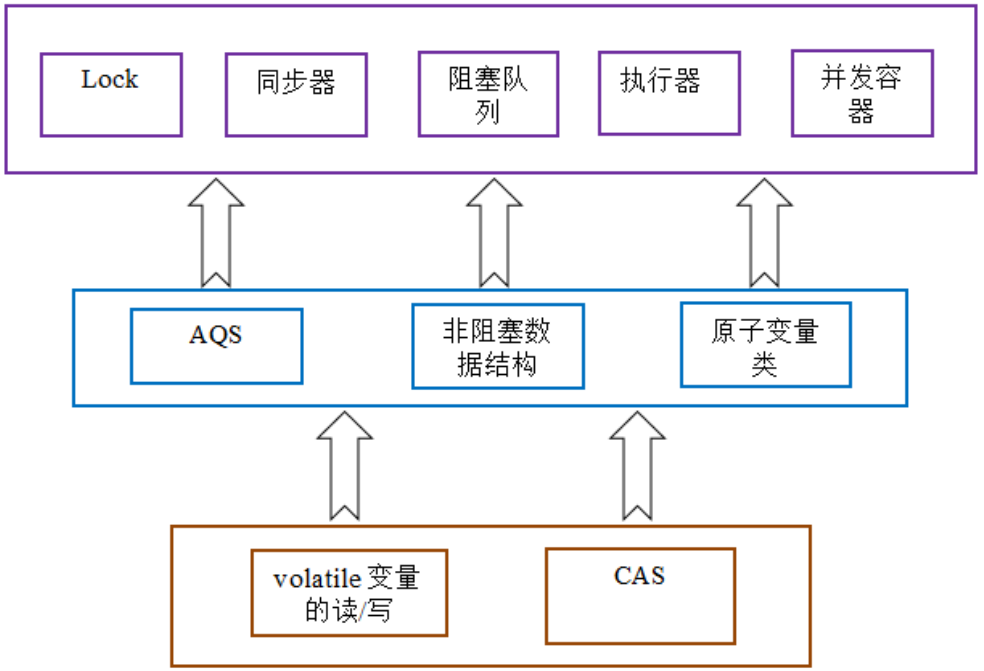
由于java的CAS同时具有 volatile 读和volatile写的内存语义，因此Java线程之间的通信现在有了下面四种方式：

1. A线程写volatile变量，随后B线程读这个volatile变量。
2. A线程写volatile变量，随后B线程用CAS更新这个volatile变量。
3. A线程用CAS更新一个volatile变量，随后B线程用CAS更新这个volatile变量。
4. A线程用CAS更新一个volatile变量，随后B线程读这个volatile变量。

Java的CAS会使用现代处理器上提供的高效机器级别原子指令，这些原子指令以原子方式对内存执行读-改-写操作，这是在多处理器中实现同步的关键（从本质上来说，能够支持原子性读-改-写指令的计算机器，是顺序计算图灵机的异步等价机器，因此任何现代的多处理器都会去支持某种能对内存执行原子性读-改-写操作的原子指令）。同时，volatile变量的读/写和CAS可以实现线程之间的通信。把这些特性整合在一起，就形成了整个concurrent包得以实现的基石。如果我们仔细分析concurrent包的源代码实现，会发现一个通用化的实现模式：

1. 首先，声明共享变量为volatile；
2. 然后，使用CAS的原子条件更新来实现线程之间的同步；
3. 同时，配合以volatile的读/写和CAS所具有的volatile读和写的内存语义来实现线程之间的通信。

AQS，非阻塞数据结构和原子变量类（java.util.concurrent.atomic包中的类），这些concurrent包中的基础类都是使用这种模式来实现的，而concurrent包中的高层类又是依赖于这些基础类来实现的。从整体来看，concurrent包的实现示意图如下：



参考文献

1. Concurrent Programming in Java: Design Principles and Pattern
2. JSR 133 (Java Memory Model) FAQ
3. JSR-133: Java Memory Model and Thread Specification

4. Java Concurrency in Practice

5. Java™ Platform, Standard Edition 6 API Specification

6. The JSR-133 Cookbook for Compiler Writers

7. Intel® 64 and IA-32 ArchitecturesvSoftware Developer's Manual Volume 3A: System Programming Guide, Part 1

8. The Art of Multiprocessor Programming

关于作者

程晓明，Java软件工程师，国家认证的系统分析师、信息项目管理师。专注于并发编程，就职于富士通南大。个人邮箱：asst2003@163.com。

11 讨论