原子操作

原子（atom）本意是“不能被进一步分割的最小粒子”，而原子操作（atomic operation）意为”不可被中断的一个或一系列操作” 。在多处理器上实现原子操作就变得有点复杂。本文让我们一起来聊一聊在Inter处理器和Java里是如何实现原子操作的。

1、相关术语

术语名称	英文	解释
缓存行	Cache line	缓存的最小操作单位
比较并交换	Compare and Swap	CAS操作需要输入两个数值，一个旧值（期望操作 生变化，如果没有发生变化，才**交换**成新值，发生
CPU流水线	CPU pipeline	CPU流水线的工作方式就象工业生产上的装配流水 理流水线，然后将一条X86指令分成5~6步后再由 完成一条指令，因此提高CPU的运算速度。
内存顺序冲突	**Memory order violation**	内存顺序冲突一般是由假共享引起，假共享是指多 CPU的操作无效，当出现这个内存顺序冲突时，CP

2、处理器如何实现原子操作

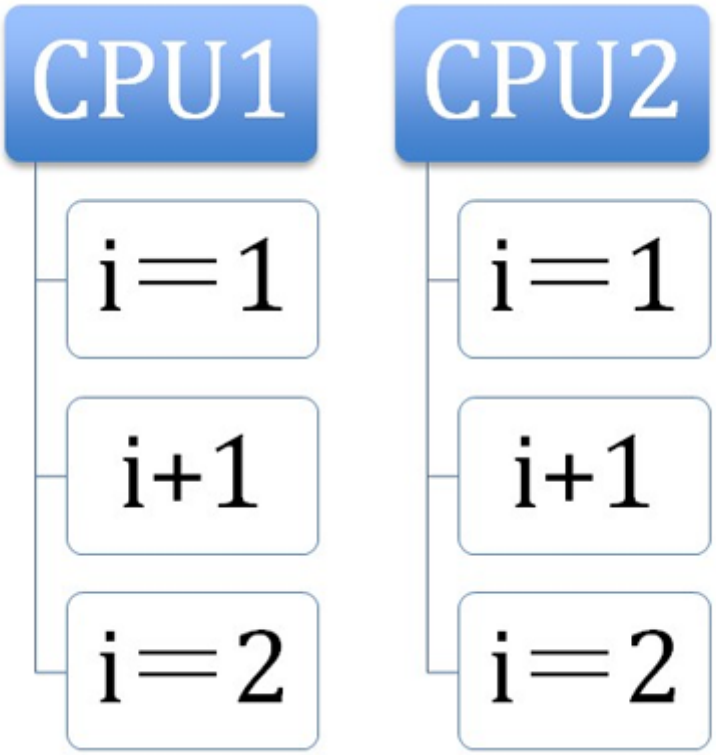
32位IA-32处理器使用**基于对缓存加锁或总线加锁**的方式来实现多处理器之间的原子操作。

2.1 处理器自动保证基本内存操作的原子性

首先处理器会自动保证基本的内存操作的原子性。处理器保证从系统内存当中读取或者写入一个字节是原子的，意思是当一个处理器读取一个字节时，其他处理器不能访问这个字节的内存地址。奔腾6和最新的处理器能自动保证单处理器对同一个缓存行里进行16/32/64位的操作是原子的，但是复杂的内存操作处理器不能自动保证其原子性，比如跨总线宽度，跨多个缓存行，跨页表的访问。但是处理器提供总线锁定和缓存锁定两个机制来保证复杂内存操作的原子性。

2.2 使用总线锁保证原子性

第一个机制是通过总线锁保证原子性。如果多个处理器同时对共享变量进行读改写（i++就是经典的读改写操作）操作，那么共享变量就会被多个处理器同时进行操作，这样读改写操作就不是原子的，操作完之后共享变量的值会和期望的不一致，举个例子：如果i=1,我们进行两次i++操作，我们期望的结果是3，但是有可能结果是2。如下图



原因是有可能多个处理器同时从各自的缓存中读取变量i，分别进行加一操作，然后分别写入系统内存当中。那么想要保证读改写共享变量的操作是原子的，就必须保证CPU1读改写共享变量的时候，CPU2不能操作缓存了该共享变量内存地址的缓存。

处理器使用总线锁就是来解决这个问题的。所谓总线锁就是使用处理器提供的一个LOCK＃信号，当一个处理器在总线上输出此信号时，其他处理器的请求将被阻塞住,那么该处理器可以独占使用共享内存。

2.3 使用缓存锁保证原子性

第二个机制是通过缓存锁定保证原子性。在同一时刻我们只需保证对某个内存地址的操作是原子性即可，但总线锁定把CPU和内存之间通信锁住了，这使得锁定期间，其他处理器不能操作其他内存地址的数据，所以总线锁定的开销比较大，最近的处理器在某些场合下使用缓存锁定代替总线锁定来进行优化。

频繁使用的内存会缓存在处理器的L1，L2和L3高速缓存里，那么原子操作就可以直接在处理器内部缓存中进行，并不需要声明总线锁，在奔腾6和最近的处理器中可以使用“缓存锁定”的方式来实现复杂的原子性。所谓“缓存锁定”就是如果缓存在处理器缓存行中内

存区域在LOCK操作期间被锁定，当它执行锁操作回写内存时，处理器不在总线上声言LOCK＃信号，而是修改内部的内存地址，并允许它的缓存一致性机制来保证操作的原子性，因为缓存一致性机制会阻止同时修改被两个以上处理器缓存的内存区域数据，当其他处理器回写已被锁定的缓存行的数据时会起缓存行无效，在例1中，当CPU1修改缓存行中的i时使用缓存锁定，那么CPU2就不能同时缓存了i的缓存行。

但是有两种情况下处理器不会使用缓存锁定。第一种情况是：当操作的数据不能被缓存在处理器内部，或操作的数据跨多个缓存行（cache line），则处理器会调用总线锁定。第二种情况是：有些处理器不支持缓存锁定。对于Inter486和奔腾处理器,就算锁定的内存区域在处理器的缓存行中也会调用总线锁定。

以上两个机制我们可以通过Inter处理器提供了很多LOCK前缀的指令来实现。比如位测试和修改指令BTS，BTR，BTC，交换指令XADD，CMPXCHG和其他一些操作数和逻辑指令，比如ADD（加），OR（或）等，被这些指令操作的内存区域就会加锁，导致其他处理器不能同时访问它。

2.4Java当中如何实现原子操作

在java中可以通过**锁**和**循环CAS**的方式来实现原子操作。

JVM中的CAS操作正是利用了上文中提到的处理器提供的CMPXCHG指令实现的。自旋CAS实现的基本思路就是循环进行CAS操作直到成功为止，具体的类可以参见juc下的atomic包内的原子类。

Atomic

在Atomic包里一共有12个类，四种原子更新方式，分别是原子更新基本类型，原子更新数组，原子更新引用和原子更新字段。Atomic包里的类基本都是使用Unsafe实现的包装类。

基本类：AtomicInteger、AtomicLong、AtomicBoolean；

引用类型：AtomicReference、AtomicReference的ABA实例、AtomicStampedRerence、AtomicMarkableReference；
数组类型：AtomicIntegerArray、AtomicLongArray、AtomicReferenceArray
属性原子修改器（Updater）：AtomicIntegerFieldUpdater、AtomicLongFieldUpdater、AtomicReferenceFieldUpdater

1、原子更新基本类型类

用于通过原子的方式更新基本类型，Atomic包提供了以下三个类：

- AtomicBoolean：原子更新布尔类型。
- AtomicInteger：原子更新整型。
- AtomicLong：原子更新长整型。

AtomicInteger的常用方法如下：

- int addAndGet(int delta)：以原子方式将输入的数值与实例中的值（AtomicInteger里的value）相加，并返回结果
- boolean compareAndSet(int expect, int update)：如果输入的数值等于预期值，则以原子方式将该值设置为输入的值。
- int getAndIncrement()：以原子方式将当前值加1，注意：这里返回的是自增前的值。
- void lazySet(int newValue)：最终会设置成newValue，使用lazySet设置值后，可能导致其他线程在之后的一小段时间内还是可以读到旧的值。
- int getAndSet(int newValue)：以原子方式设置为newValue的值，并返回旧值。

Atomic包提供了三种基本类型的原子更新，但是Java的基本类型里还有char，float和double等。那么问题来了，如何原子的更新其他的基本类型呢？Atomic包里的类基本都是使用Unsafe实现的，Unsafe只提供了三种CAS方法，compareAndSwapObject，compareAndSwapInt和compareAndSwapLong，再看AtomicBoolean源码，发现其是先把Boolean转换成整型，再使用compareAndSwapInt进行CAS，所以原子更新double也可以用类似的思路来实现。

2、原子更新数组类

通过原子的方式更新数组里的某个元素，Atomic包提供了以下三个类：

- AtomicIntegerArray：原子更新整型数组里的元素。

- AtomicLongArray：原子更新长整型数组里的元素。
- AtomicReferenceArray：原子更新引用类型数组里的元素。

AtomicIntegerArray类主要是提供原子的方式更新数组里的整型，其常用方法如下

- int addAndGet(int i, int delta)：以原子方式将输入值与数组中索引i的元素相加。
- boolean compareAndSet(int i, int expect, int update)：如果当前值等于预期值，则以原子方式将数组位置i的元素设置成update值。

3、原子更新引用类型

原子更新基本类型的AtomicInteger，只能更新一个变量，如果要原子的更新多个变量，就需要使用这个原子更新引用类型提供的类。Atomic包提供了以下三个类：

- AtomicReference：原子更新引用类型。
- AtomicReferenceFieldUpdater：原子更新引用类型里的字段。
- AtomicMarkableReference：原子更新带有标记位的引用类型。可以原子的更新一个布尔类型的标记位和引用类型。构造方法是AtomicMarkableReference(V initialRef, boolean initialMark)

4、原子更新字段类

如果我们只需要某个类里的某个字段，那么就需要使用原子更新字段类，Atomic包提供了以下三个类：

- AtomicIntegerFieldUpdater：原子更新整型的字段的更新器。
- AtomicLongFieldUpdater：原子更新长整型字段的更新器。
- AtomicStampedReference：原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于原子的更数据和数据的版本号，可以解决使用CAS进行原子更新时，可能出现的ABA问题。

原子更新字段类都是抽象类，每次使用都时候必须使用静态方法newUpdater创建一个更新器。原子更新类的字段的必须使用public volatile修饰符。

Unsafe应用解析

Unsafe是位于sun.misc包下的一个类，主要提供一些用于执行低级别、不安全操作的方法，如直接访问系统内存资源、自主管理内存资源等，这些方法在提升Java运行效率、增

强Java语言底层资源操作能力方面起到了很大的作用。但由于Unsafe类使Java语言拥有了类似C语言指针一样操作内存空间的能力，这无疑也增加了程序发生相关指针问题的风险。在程序中过度、不正确使用Unsafe类会使得程序出错的概率变大，使得Java这种安全的语言变得不再“安全”，因此对Unsafe的使用一定要慎重。

Unsafe类为一单例实现，提供静态方法getUnsafe获取Unsafe实例，当且仅当调用getUnsafe方法的类为引导类加载器所加载时才合法，否则抛出SecurityException异常。

```
public class Unsafe {
    // 单例对象
    private static final Unsafe theUnsafe;
    private Unsafe() {
    }
    @CallerSensitive
    public static Unsafe getUnsafe() {
        Class var0 = Reflection.getCallerClass();
        // 仅在引导类加载器`BootstrapClassLoader`加载时才合法
        if(!VM.isSystemDomainLoader(var0.getClassLoader())) {
            throw new SecurityException("Unsafe");
        } else {
            return theUnsafe;
        }
    }
}
```

如何获取Unsafe实例?

1、从getUnsafe方法的使用限制条件出发，通过Java命令行命令-Xbootclasspath/a把调用Unsafe相关方法的类A所在jar包路径追加到默认的bootstrap路径中，使得A被引导类加载器加载，从而通过Unsafe.getUnsafe方法安全的获取Unsafe实例。

```
java -Xbootclasspath/a:${path}   // 其中path为调用Unsafe相关方法的类所在jar包路径
```

2、通过反射获取单例对象theUnsafe。

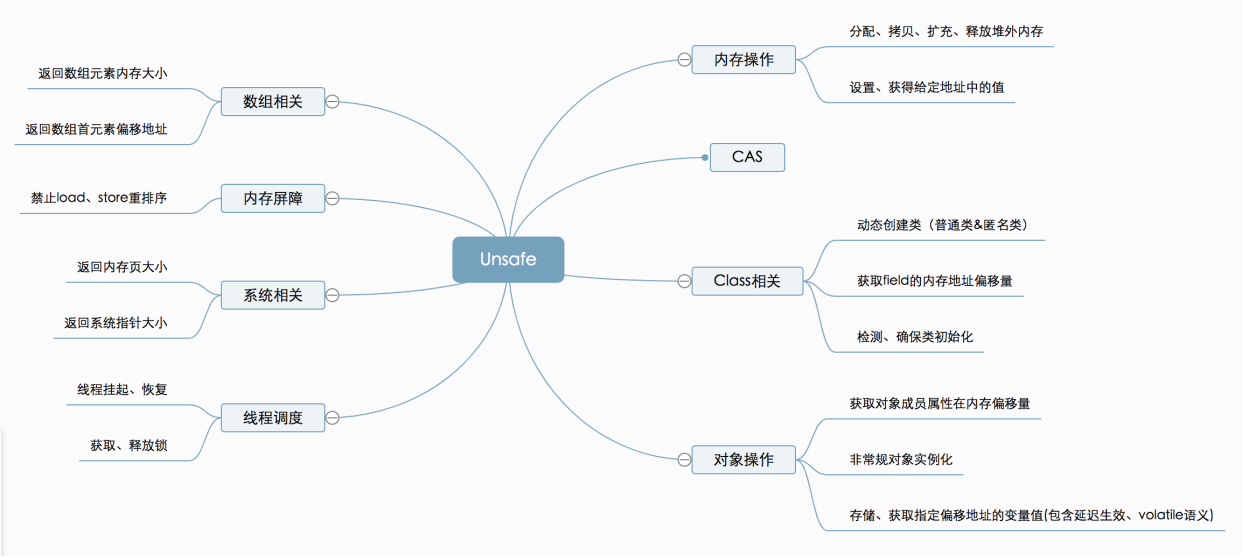
```
public class UnsafeInstance {

    public static Unsafe reflectGetUnsafe() {
        try {
```

```
            Field field =
Unsafe.class.getDeclaredField("theUnsafe");
            field.setAccessible(true);
            return (Unsafe) field.get(null);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

Unsafe功能介绍

Unsafe提供的API大致可分为内存操作、CAS、Class相关、对象操作、线程调度、系统信息获取、内存屏障、数组操作等几类，下面将对其相关方法和应用场景进行详细介绍。



1、内存操作

这部分主要包含堆外内存的分配、拷贝、释放、给定地址值操作等方法。

```
    //分配内存，相当于C++的malloc函数
    public native long allocateMemory(long bytes);
    //扩充内存
    public native long reallocateMemory(long address, long bytes);
    //释放内存
    public native void freeMemory(long address);
```

```
    //在给定的内存块中设置值
    public native void setMemory(Object o, long offset, long bytes,
    byte value);
    //内存拷贝
    public native void copyMemory(Object srcBase, long srcOffset,
    Object destBase, long destOffset, long bytes);
    //获取给定地址值，忽略修饰限定符的访问限制。与此类似操作还有：getInt,
    getDouble, getLong, getChar等
    public native Object getObject(Object o, long offset);
    //为给定地址设置值，忽略修饰限定符的访问限制，与此类似操作还有:
    putInt,putDouble, putLong, putChar等
    public native void putObject(Object o, long offset, Object x);
    public native byte getByte(long address);
    //为给定地址设置byte类型的值（当且仅当该内存地址为
allocateMemory分配          时，此方法结果才是确定的）
    public native void putByte(long address, byte x);
```

通常，我们在Java中创建的对象都处于堆内内存（heap）中，堆内内存是由JVM所管控的Java进程内存，并且它们遵循JVM的内存管理机制，JVM会采用垃圾回收机制统一管理堆内存。与之相对的是堆外内存，存在于JVM管控之外的内存区域，Java中对堆外内存的操作，依赖于Unsafe提供的操作堆外内存的native方法。

使用堆外内存的原因

- 对垃圾回收停顿的改善。由于堆外内存是直接受操作系统管理而不是JVM，所以当我们使用堆外内存时，即可保持较小的堆内内存规模。从而在GC时减少回收停顿对于应用的影响。
- 提升程序I/O操作的性能。通常在I/O通信过程中，会存在堆内内存到堆外内存的数据拷贝操作，对于需要频繁进行内存间数据拷贝且生命周期较短的暂存数据，都建议存储到堆外内存。

典型应用

DirectByteBuffer是Java用于实现堆外内存的一个重要类，通常用在通信过程中做缓冲池，如在Netty、MINA等NIO框架中应用广泛。DirectByteBuffer对于堆外内存的创建、使用、销毁等逻辑均由Unsafe提供的堆外内存API来实现。

下图为DirectByteBuffer构造函数，创建DirectByteBuffer的时候，通过Unsafe.allocateMemory分配内存、Unsafe.setMemory进行内存初始化，而后构建Cleaner对象用于跟踪DirectByteBuffer对象的垃圾回收，以实现当DirectByteBuffer被垃圾回收时，分配的堆外内存一起被释放。

```
//
DirectByteBuffer(int cap) {                   // package-private

    super( mark: -1,  pos: 0, cap, cap);
    boolean pa = VM.isDirectMemoryPageAligned();
    int ps = Bits.pageSize();
    long size = Math.max(1L, (long)cap + (pa ? ps : 0));
    Bits.reserveMemory(size, cap);

    long base = 0;
    try {
        base = unsafe.allocateMemory(size);
    } catch (OutOfMemoryError x) {
        Bits.unreserveMemory(size, cap);
        throw x;
    }
    unsafe.setMemory(base, size, (byte) 0);
    if (pa && (base % ps != 0)) {
        // Round up to page boundary
        address = base + ps - (base & (ps - 1));
    } else {
        address = base;
    }
    cleaner = Cleaner.create( o: this, new Deallocator(base, size, cap));
    att = null;
```

分配内存，并返回基地址

内存初始化

跟踪DirectByteBuffer对象的垃圾回收，以实现堆外内存释放

2、CAS相关

如下源代码释义所示，这部分主要为CAS相关操作的方法。

```
/**
 *  CAS
 * @param o         包含要修改field的对象
 * @param offset    对象中某field的偏移量
 * @param expected  期望值
 * @param update    更新值
 * @return          true | false
 */
public final native boolean compareAndSwapObject(Object var1, long var2, Object var4, Object var5);

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

public final native boolean compareAndSwapLong(Object var1, long var2, long var4, long var6);
```

典型应用

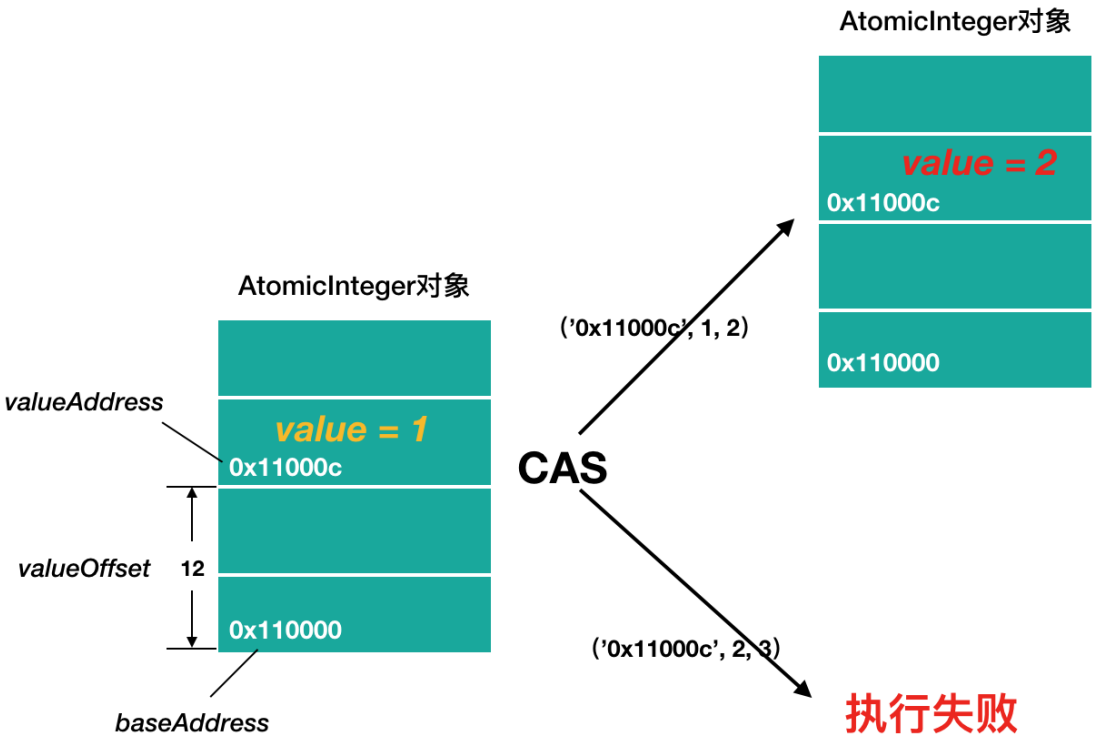
如下图所示，AtomicInteger的实现中，静态字段valueOffset即为字段value的内存偏移地址，valueOffset的值在AtomicInteger初始化时，在静态代码块中通过Unsafe的objectFieldOffset方法获取。在AtomicInteger中提供的线程安全方法中，通过字段valueOffset的值可以定位到AtomicInteger对象中value的内存地址，从而可以根据CAS实现对value字段的原子操作。

```
public class AtomicInteger extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 6214790243416807050L;

    // setup to use Unsafe.compareAndSwapInt for updates
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

    static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField( name: "value"));
        } catch (Exception ex) { throw new Error(ex); }
    }
```

下图为某个AtomicInteger对象自增操作前后的内存示意图，对象的基地址baseAddress=“0x110000”，通过baseAddress+valueOffset得到value的内存地址valueAddress=“0x11000c”；然后通过CAS进行原子性的更新操作，成功则返回，否则继续重试，直到更新成功为止。



3、线程调度

包括线程挂起、恢复、锁机制等方法。

```
//取消阻塞线程
public native void unpark(Object thread);
//阻塞线程
public native void park(boolean isAbsolute, long time);
//获得对象锁（可重入锁）
@Deprecated
public native void monitorEnter(Object o);
//释放对象锁
@Deprecated
public native void monitorExit(Object o);
//尝试获取对象锁
@Deprecated
public native boolean tryMonitorEnter(Object o);
```

方法park、unpark即可实现线程的挂起与恢复，将一个线程进行挂起是通过park方法实现的，调用park方法后，线程将一直阻塞直到超时或者中断等条件出现；unpark可以终止一个挂起的线程，使其恢复正常。

典型应用

Java锁和同步器框架的核心类AbstractQueuedSynchronizer，就是通过调用LockSupport.park()和LockSupport.unpark()实现线程的阻塞和唤醒的，而LockSupport的park、unpark方法实际是调用Unsafe的park、unpark方式来实现。

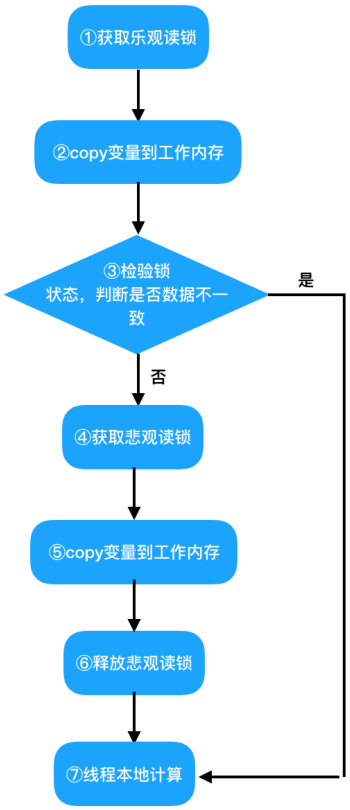
4、内存屏障

在Java 8中引入，用于定义内存屏障（也称内存栅栏，内存栅障，屏障指令等，是一类同步屏障指令，是CPU或编译器在对内存随机访问的操作中的一个同步点，使得此点之前的所有读写操作都执行后才可以开始执行此点之后的操作），避免代码重排序。

```
//内存屏障，禁止load操作重排序。屏障前的load操作不能被重排序到屏障后，屏障后的load操作不能被重排序到屏障前
public native void loadFence();
//内存屏障，禁止store操作重排序。屏障前的store操作不能被重排序到屏障后，屏障后的store操作不能被重排序到屏障前
public native void storeFence();
//内存屏障，禁止load、store操作重排序
public native void fullFence();
```

典型应用

在Java 8中引入了一种锁的新机制——StampedLock，它可以看成是读写锁的一个改进版本。StampedLock提供了一种乐观读锁的实现，这种乐观读锁类似于无锁的操作，完全不会阻塞写线程获取写锁，从而缓解读多写少时写线程“饥饿”现象。由于StampedLock提供的乐观读锁不阻塞写线程获取读锁，当线程共享变量从主内存load到线程工作内存时，会存在数据不一致问题，所以当使用StampedLock的乐观读锁时，需要遵从如下图用例中使用的模式来确保数据的一致性。




```java
class Point {

    private double x, y;

    private final StampedLock sl = new StampedLock();

    void move(double deltaX, double deltaY) {
        long stamp = sl.writeLock();     // 使用写锁-独占操作
        try {
            x += deltaX;
            y += deltaY;
        } finally {
            sl.unlockWrite(stamp);
        }
    }
    double distanceFromOrigin() {
        long stamp = sl.tryOptimisticRead();         // ①
        double currentX = x, currentY = y;           // ②
        if (!sl.validate(stamp)) {                   // ③
            stamp = sl.readLock();                   // ④
            try {
                currentX = x;                        // ⑤
                currentY = y;
            } finally {
                sl.unlockRead(stamp);                // ⑥
            }
        }
        return Math.sqrt(currentX * currentX + currentY * currentY); // ⑦
    }
}
```

如上图用例所示计算坐标点Point对象，包含点移动方法move及计算此点到原点的距离的方法distanceFromOrigin。在方法distanceFromOrigin中，首先，通过tryOptimisticRead方法获取乐观读标记；然后从主内存中加载点的坐标值 (x,y)；而后通过StampedLock的validate方法校验锁状态，判断坐标点(x,y)从主内存加载到线程工作内存过程中，主内存的值是否已被其他线程通过move方法修改，如果validate返回值为true，证明(x, y)的值未被修改，可参与后续计算；否则，需加悲观读锁，再次从主内存加载(x,y)的最新值，然后再进行距离计算。其中，校验锁状态这步操作至关重要，需要判断锁状态是否发生改变，从而判断之前copy到线程工作内存中的值是否与主内存的值存在不一致。

下图为StampedLock.validate方法的源码实现，通过锁标记与相关常量进行位运算、比较来校验锁状态，在校验逻辑之前，会通过Unsafe的loadFence方法加入一个load内存屏障，目的是避免上图用例中步骤②和StampedLock.validate中锁状态校验运算发生重排序导致锁状态校验不准确的问题。

```java
/**
 * Returns true if the lock has not been exclusively acquired
 * since issuance of the given stamp. Always returns false if the
 * stamp is zero. Always returns true if the stamp represents a
 * currently held lock. Invoking this method with a value not
 * obtained from {@link #tryOptimisticRead} or a locking method
 * for this lock has no defined effect or result.
 *
 * @param stamp a stamp
 * @return {@code true} if the lock has not been exclusively acquired
 * since issuance of the given stamp; else false
 */
public boolean validate(long stamp) {
    U.loadFence();
    return (stamp & SBITS) == (state & SBITS);
}
```

load内存屏障

其它方面，大家可以自行查阅...

> 有道云笔记：http://note.youdao.com/noteshare?id=3224c156c25f8efcc118d5492d8fcffe&sub=CC5F38B599C048D4B9B7623F2F474AE3