Unsafe 源码解析

Unsafe是java里非常核心的类，无论对于(J.U.C)并发包的线程调度、CAS操作、还是内存操作、内存屏障，系统相关、加载器相关、类相关、数组相关等都是使用该类；

- Unsafe源码解析：

```
package com.bfxy.mix.unsafe;
import java.security.*;
import java.lang.reflect.*;

import sun.misc.VM;
import sun.reflect.CallerSensitive;
import sun.reflect.Reflection;

/**
 *  $Unsafe
 * @author hezhuo.bai-JiFeng
 * @since 2017年11月16日 下午8:02:28
 */
@SuppressWarnings("restriction")
public final class Unsafe {

    private static native void registerNatives();
    static {
        registerNatives();
        sun.reflect.Reflection.registerMethodsToFilter(Unsafe.class, "getUnsafe");
    }

    private Unsafe() {}

    private static final Unsafe theUnsafe = new Unsafe();

    @CallerSensitive
    public static Unsafe getUnsafe() {
        Class<?> caller = Reflection.getCallerClass();
        /**
         *  检查调用者的类加载器是否是启动类加载器；
         *  抛出异常就是因为我们自己创建的类不是由启动类加载器加载；
         *  而是默认由系统类加载器加载的，故抛出异常；
         *  只能利用反射来获取Unsafe实例对象，做法：com.bfxy.thread.unsafe.UnsafeUtil类；
         *  发现使用反射获取Unsafe实例是可以获取到的，打印的类加载器的信息为null，说明是由启动
类加载加载的；
         */
        if (!VM.isSystemDomainLoader(caller.getClassLoader()))
            throw new SecurityException("Unsafe");
        return theUnsafe;
    }
```

```java
    /** ------------------------------ peek and poke 取/存指令 ---------------------
-------  */

    //  peek: 获取对象o中给定偏移地址(offset)的值，以下相关get方法作用相同
    public native int getInt(Object o, long offset);
    //  poke: 在对象o的给定偏移地址存储数值x，以下set方法作用相同
    public native void putInt(Object o, long offset, int x);

    public native Object getObject(Object o, long offset);

    public native void putObject(Object o, long offset, Object x);

    public native boolean getBoolean(Object o, long offset);

    public native void putBoolean(Object o, long offset, boolean x);

    public native byte getByte(Object o, long offset);

    public native void putByte(Object o, long offset, byte x);

    public native short getShort(Object o, long offset);

    public native void putShort(Object o, long offset, short x);

    public native char getChar(Object o, long offset);

    public native void putChar(Object o, long offset, char x);

    public native long getLong(Object o, long offset);

    public native void putLong(Object o, long offset, long x);

    public native float getFloat(Object o, long offset);

    public native void putFloat(Object o, long offset, float x);

    public native double getDouble(Object o, long offset);

    public native void putDouble(Object o, long offset, double x);

    /**
     * This method, like all others with 32-bit offsets, was native
     * in a previous release but is now a wrapper which simply casts
     * the offset to a long value.  It provides backward compatibility
     * with bytecodes compiled against 1.4.
     * @deprecated As of 1.4.1, cast the 32-bit offset argument to a long.
     * See {@link #staticFieldOffset}.
     */
    @Deprecated
    public int getInt(Object o, int offset) {
        return getInt(o, (long)offset);
    }
    @Deprecated
    public void putInt(Object o, int offset, int x) {
        putInt(o, (long)offset, x);
    }

    @Deprecated
    public Object getObject(Object o, int offset) {
        return getObject(o, (long)offset);
    }

    @Deprecated
    public void putObject(Object o, int offset, Object x) {
        putObject(o, (long)offset, x);
    }

    @Deprecated
    public boolean getBoolean(Object o, int offset) {
        return getBoolean(o, (long)offset);
    }

    @Deprecated
    public void putBoolean(Object o, int offset, boolean x) {
        putBoolean(o, (long)offset, x);
    }

    @Deprecated
    public byte getByte(Object o, int offset) {
        return getByte(o, (long)offset);
    }

    @Deprecated
    public void putByte(Object o, int offset, byte x) {
        putByte(o, (long)offset, x);
    }

    @Deprecated
    public short getShort(Object o, int offset) {
        return getShort(o, (long)offset);
    }

    @Deprecated
    public void putShort(Object o, int offset, short x) {
        putShort(o, (long)offset, x);
    }

    @Deprecated
    public char getChar(Object o, int offset) {
        return getChar(o, (long)offset);
    }

    @Deprecated
    public void putChar(Object o, int offset, char x) {
        putChar(o, (long)offset, x);
```

```java
    }

    @Deprecated
    public long getLong(Object o, int offset) {
        return getLong(o, (long)offset);
    }

    @Deprecated
    public void putLong(Object o, int offset, long x) {
        putLong(o, (long)offset, x);
    }

    @Deprecated
    public float getFloat(Object o, int offset) {
        return getFloat(o, (long)offset);
    }

    @Deprecated
    public void putFloat(Object o, int offset, float x) {
        putFloat(o, (long)offset, x);
    }

    @Deprecated
    public double getDouble(Object o, int offset) {
        return getDouble(o, (long)offset);
    }

    @Deprecated
    public void putDouble(Object o, int offset, double x) {
        putDouble(o, (long)offset, x);
    }

    // These work on values in the C heap.

    /**
     * Fetches a value from a given memory address.  If the address is zero, or
     * does not point into a block obtained from {@link #allocateMemory}, the
     * results are undefined.
     *
     * @see #allocateMemory
     */
    //  从给定内存地址获取一个byte
    public native byte getByte(long address);

    //  在给定内存地址放置一个x
    public native void putByte(long address, byte x);

    /** @see #getByte(long) */
    public native short   getShort(long address);
    /** @see #putByte(long, byte) */
    public native void    putShort(long address, short x);
    /** @see #getByte(long) */
    public native char    getChar(long address);
    /** @see #putByte(long, byte) */
    public native void    putChar(long address, char x);
    /** @see #getByte(long) */
    public native int     getInt(long address);
    /** @see #putByte(long, byte) */
    public native void    putInt(long address, int x);
    /** @see #getByte(long) */
    public native long    getLong(long address);
    /** @see #putByte(long, byte) */
    public native void    putLong(long address, long x);
    /** @see #getByte(long) */
    public native float   getFloat(long address);
    /** @see #putByte(long, byte) */
    public native void    putFloat(long address, float x);
    /** @see #getByte(long) */
    public native double  getDouble(long address);
    /** @see #putByte(long, byte) */
    public native void    putDouble(long address, double x);

    /**
     * Fetches a native pointer from a given memory address.  If the address is
     * zero, or does not point into a block obtained from {@link
     * #allocateMemory}, the results are undefined.
     *
     * <p> If the native pointer is less than 64 bits wide, it is extended as
     * an unsigned number to a Java long.  The pointer may be indexed by any
     * given byte offset, simply by adding that offset (as a simple integer) to
     * the long representing the pointer.  The number of bytes actually read
     * from the target address maybe determined by consulting {@link
     * #addressSize}.
     *
     * @see #allocateMemory
     */
    //  获取给定内存地址的一个本地指针 内存地址->本地指针(offset)
    public native long getAddress(long address);

    /**
     * Stores a native pointer into a given memory address.  If the address is
     * zero, or does not point into a block obtained from {@link
     * #allocateMemory}, the results are undefined.
     *
     * <p> The number of bytes actually written at the target address maybe
     * determined by consulting {@link #addressSize}.
     *
     * @see #getAddress(long)
     */
    //  在给定的内存地址处存放一个本地指针x
    public native void putAddress(long address, long x);

    /// wrappers for malloc, realloc, free:

    /** ------------------------------------ 内存操作 -------------------------------
----------  */
```

```
    /**
     * Allocates a new block of native memory, of the given size in bytes.  The
     * contents of the memory are uninitialized; they will generally be
     * garbage.  The resulting native pointer will never be zero, and will be
     * aligned for all value types.  Dispose of this memory by calling {@link
     * #freeMemory}, or resize it with {@link #reallocateMemory}.
     *
     * @throws IllegalArgumentException if the size is negative or too large
     *         for the native size_t type
     *
     * @throws OutOfMemoryError if the allocation is refused by the system
     *
     * @see #getByte(long)
     * @see #putByte(long, byte)
     */

    //  在本地内存分配一块指定大小的新内存，内存的内容未初始化；它们通常被当做垃圾回收
    public native long allocateMemory(long bytes);

    /**
     * Resizes a new block of native memory, to the given size in bytes.  The
     * contents of the new block past the size of the old block are
     * uninitialized; they will generally be garbage.  The resulting native
     * pointer will be zero if and only if the requested size is zero.  The
     * resulting native pointer will be aligned for all value types.  Dispose
     * of this memory by calling {@link #freeMemory}, or resize it with {@link
     * #reallocateMemory}.  The address passed to this method may be null, in
     * which case an allocation will be performed.
     *
     * @throws IllegalArgumentException if the size is negative or too large
     *         for the native size_t type
     *
     * @throws OutOfMemoryError if the allocation is refused by the system
     *
     * @see #allocateMemory
     */

    //  扩容内存
    public native long reallocateMemory(long address, long bytes);

    /**
     * Sets all bytes in a given block of memory to a fixed value
     * (usually zero).
     *
     * <p>This method determines a block's base address by means of two parameters,
     * and so it provides (in effect) a <em>double-register</em> addressing mode,
     * as discussed in {@link #getInt(Object,long)}.  When the object reference is
null,
     * the offset supplies an absolute base address.
     *
     * <p>The stores are in coherent (atomic) units of a size determined
     * by the address and length parameters.  If the effective address and
     * length are all even modulo 8, the stores take place in 'long' units.
     * If the effective address and length are (resp.) even modulo 4 or 2,
     * the stores take place in units of 'int' or 'short'.
     *
     * @since 1.7
     */

    //  将给定内存块中的所有字节设置为固定值（通常是0）
    public native void setMemory(Object o, long offset, long bytes, byte value);

    /**
     * Sets all bytes in a given block of memory to a fixed value
     * (usually zero).  This provides a <em>single-register</em> addressing mode,
     * as discussed in {@link #getInt(Object,long)}.
     *
     * <p>Equivalent to <code>setMemory(null, address, bytes, value)</code>.
     */
    public void setMemory(long address, long bytes, byte value) {
        setMemory(null, address, bytes, value);
    }

    /**
     * Sets all bytes in a given block of memory to a copy of another
     * block.
     *
     * <p>This method determines each block's base address by means of two
parameters,
     * and so it provides (in effect) a <em>double-register</em> addressing mode,
     * as discussed in {@link #getInt(Object,long)}.  When the object reference is
null,
     * the offset supplies an absolute base address.
     *
     * <p>The transfers are in coherent (atomic) units of a size determined
     * by the address and length parameters.  If the effective addresses and
     * length are all even modulo 8, the transfer takes place in 'long' units.
     * If the effective addresses and length are (resp.) even modulo 4 or 2,
     * the transfer takes place in units of 'int' or 'short'.
     *
     * @since 1.7
     */

    //  内存拷贝：复制一块内存，double-register模型
    public native void copyMemory(Object srcBase, long srcOffset,
                                  Object destBase, long destOffset,
                                  long bytes);
    /**
     * Sets all bytes in a given block of memory to a copy of another
     * block.  This provides a <em>single-register</em> addressing mode,
     * as discussed in {@link #getInt(Object,long)}.
     *
     * Equivalent to <code>copyMemory(null, srcAddress, null, destAddress, bytes)
</code>.
     */
```

```java
public void copyMemory(long srcAddress, long destAddress, long bytes) {
    copyMemory(null, srcAddress, null, destAddress, bytes);
}

/**
 * Disposes of a block of native memory, as obtained from {@link
 * #allocateMemory} or {@link #reallocateMemory}.  The address passed to
 * this method may be null, in which case no action is taken.
 *
 * @see #allocateMemory
 */

//  释放给定地址的内存
public native void freeMemory(long address);

/// random queries

/**
 * This constant differs from all results that will ever be returned from
 * {@link #staticFieldOffset}, {@link #objectFieldOffset},
 * or {@link #arrayBaseOffset}.
 */
public static final int INVALID_FIELD_OFFSET   = -1;

@Deprecated
public int fieldOffset(Field f) {
    if (Modifier.isStatic(f.getModifiers()))
        return (int) staticFieldOffset(f);
    else
        return (int) objectFieldOffset(f);
}

@Deprecated
public Object staticFieldBase(Class<?> c) {
    Field[] fields = c.getDeclaredFields();
    for (int i = 0; i < fields.length; i++) {
        if (Modifier.isStatic(fields[i].getModifiers())) {
            return staticFieldBase(fields[i]);
        }
    }
    return null;
}

/**
 * Report the location of a given field in the storage allocation of its
 * class.  Do not expect to perform any sort of arithmetic on this offset;
 * it is just a cookie which is passed to the unsafe heap memory accessors.
 *
 * <p>Any given field will always have the same offset and base, and no
 * two distinct fields of the same class will ever have the same offset
 * and base.
 *
 * <p>As of 1.4.1, offsets for fields are represented as long values,
     * although the Sun JVM does not use the most significant 32 bits.
     * However, JVM implementations which store static fields at absolute
     * addresses can use long offsets and null base pointers to express
     * the field locations in a form usable by {@link #getInt(Object,long)}.
     * Therefore, code which will be ported to such JVMs on 64-bit platforms
     * must preserve all bits of static field offsets.
     * @see #getInt(Object, long)
     */

    //  获取给定<静态>对象的偏移地址，通过反射类的属性：Field f
    public native long staticFieldOffset(Field f);

    /**
     * Report the location of a given static field, in conjunction with {@link
     * #staticFieldBase}.
     * <p>Do not expect to perform any sort of arithmetic on this offset;
     * it is just a cookie which is passed to the unsafe heap memory accessors.
     *
     * <p>Any given field will always have the same offset, and no two distinct
     * fields of the same class will ever have the same offset.
     *
     * <p>As of 1.4.1, offsets for fields are represented as long values,
     * although the Sun JVM does not use the most significant 32 bits.
     * It is hard to imagine a JVM technology which needs more than
     * a few bits to encode an offset within a non-array object,
     * However, for consistency with other methods in this class,
     * this method reports its result as a long value.
     * @see #getInt(Object, long)
     */
    //  获取给定<非静态>对象的偏移地址，通过反射类的属性：Field f
    public native long objectFieldOffset(Field f);

    /**
     * Report the location of a given static field, in conjunction with {@link
     * #staticFieldOffset}.
     * <p>Fetch the base "Object", if any, with which static fields of the
     * given class can be accessed via methods like {@link #getInt(Object,
     * long)}.  This value may be null.  This value may refer to an object
     * which is a "cookie", not guaranteed to be a real Object, and it should
     * not be used in any way except as argument to the get and put routines in
     * this class.
     */

    //  获取静态变量所属的类在方法区的首地址
    public native Object staticFieldBase(Field f);

    /**
     * Detect if the given class may need to be initialized. This is often
     * needed in conjunction with obtaining the static field base of a
     * class.
     * @return false only if a call to {@code ensureClassInitialized} would have no
effect
     */
    public native boolean shouldBeInitialized(Class<?> c);

    /**
     * Ensure the given class has been initialized. This is often
     * needed in conjunction with obtaining the static field base of a
     * class.
     */
    public native void ensureClassInitialized(Class<?> c);

    /**
     * Report the offset of the first element in the storage allocation of a
     * given array class.  If {@link #arrayIndexScale} returns a non-zero value
     * for the same class, you may use that scale factor, together with this
     * base offset, to form new offsets to access elements of arrays of the
     * given class.
     *
     * @see #getInt(Object, long)
     * @see #putInt(Object, long, int)
     */

    /** ------------------------------- 数组操作 ---------------------------------
*/

    //  获取给定数组的第一个元素的偏移地址
    public native int arrayBaseOffset(Class<?> arrayClass);

    /** The value of {@code arrayBaseOffset(boolean[].class)} */
    public static final int ARRAY_BOOLEAN_BASE_OFFSET
            = theUnsafe.arrayBaseOffset(boolean[].class);

    /** The value of {@code arrayBaseOffset(byte[].class)} */
    public static final int ARRAY_BYTE_BASE_OFFSET
            = theUnsafe.arrayBaseOffset(byte[].class);

    /** The value of {@code arrayBaseOffset(short[].class)} */
    public static final int ARRAY_SHORT_BASE_OFFSET
            = theUnsafe.arrayBaseOffset(short[].class);

    /** The value of {@code arrayBaseOffset(char[].class)} */
    public static final int ARRAY_CHAR_BASE_OFFSET
            = theUnsafe.arrayBaseOffset(char[].class);

    /** The value of {@code arrayBaseOffset(int[].class)} */
    public static final int ARRAY_INT_BASE_OFFSET
            = theUnsafe.arrayBaseOffset(int[].class);

    /** The value of {@code arrayBaseOffset(long[].class)} */
    public static final int ARRAY_LONG_BASE_OFFSET
            = theUnsafe.arrayBaseOffset(long[].class);

    /** The value of {@code arrayBaseOffset(float[].class)} */
    public static final int ARRAY_FLOAT_BASE_OFFSET
            = theUnsafe.arrayBaseOffset(float[].class);
    /** The value of {@code arrayBaseOffset(double[].class)} */
    public static final int ARRAY_DOUBLE_BASE_OFFSET
            = theUnsafe.arrayBaseOffset(double[].class);

    /** The value of {@code arrayBaseOffset(Object[].class)} */
    public static final int ARRAY_OBJECT_BASE_OFFSET
            = theUnsafe.arrayBaseOffset(Object[].class);

    /**
     * Report the scale factor for addressing elements in the storage
     * allocation of a given array class.  However, arrays of "narrow" types
     * will generally not work properly with accessors like {@link
     * #getByte(Object, int)}, so the scale factor for such classes is reported
     * as zero.
     *
     * @see #arrayBaseOffset
     * @see #getInt(Object, long)
     * @see #putInt(Object, long, int)
     */

    //  获取给定数组的元素增量地址，也就是说每个元素的占位数
    public native int arrayIndexScale(Class<?> arrayClass);

    /** The value of {@code arrayIndexScale(boolean[].class)} */
    public static final int ARRAY_BOOLEAN_INDEX_SCALE
            = theUnsafe.arrayIndexScale(boolean[].class);

    /** The value of {@code arrayIndexScale(byte[].class)} */
    public static final int ARRAY_BYTE_INDEX_SCALE
            = theUnsafe.arrayIndexScale(byte[].class);

    /** The value of {@code arrayIndexScale(short[].class)} */
    public static final int ARRAY_SHORT_INDEX_SCALE
            = theUnsafe.arrayIndexScale(short[].class);

    /** The value of {@code arrayIndexScale(char[].class)} */
    public static final int ARRAY_CHAR_INDEX_SCALE
            = theUnsafe.arrayIndexScale(char[].class);

    /** The value of {@code arrayIndexScale(int[].class)} */
    public static final int ARRAY_INT_INDEX_SCALE
            = theUnsafe.arrayIndexScale(int[].class);

    /** The value of {@code arrayIndexScale(long[].class)} */
    public static final int ARRAY_LONG_INDEX_SCALE
            = theUnsafe.arrayIndexScale(long[].class);

    /** The value of {@code arrayIndexScale(float[].class)} */
    public static final int ARRAY_FLOAT_INDEX_SCALE
            = theUnsafe.arrayIndexScale(float[].class);

    /** The value of {@code arrayIndexScale(double[].class)} */
    public static final int ARRAY_DOUBLE_INDEX_SCALE
            = theUnsafe.arrayIndexScale(double[].class);

    /** The value of {@code arrayIndexScale(Object[].class)} */
    public static final int ARRAY_OBJECT_INDEX_SCALE
            = theUnsafe.arrayIndexScale(Object[].class);

    /**
     * Report the size in bytes of a native pointer, as stored via {@link
     * #putAddress}.  This value will be either 4 or 8.  Note that the sizes of
     * other primitive types (as stored in native memory blocks) is determined
     * fully by their information content.
     */
    //  获取系统指针的大小，64位系统是8
    public native int addressSize();

    /** The value of {@code addressSize()} */
    public static final int ADDRESS_SIZE = theUnsafe.addressSize();

    /**
     * Report the size in bytes of a native memory page (whatever that is).
     * This value will always be a power of two.
     */
    //  获取内存页大小，2的幂次方，我本机测试是4096
    public native int pageSize();


    /// random trusted operations from JNI:

    /**
     * Tell the VM to define a class, without security checks.  By default, the
     * class loader and protection domain come from the caller's class.
     */

    //  告诉虚拟机去定义一个类；    默认情况下，类加载器和保护域都来自这个方法(都会调用此方法)
    public native Class<?> defineClass(String name, byte[] b, int off, int len,
                                       ClassLoader loader,
                                       ProtectionDomain protectionDomain);

    /**
     * Define a class but do not make it known to the class loader or system
dictionary.
     * <p>
     * For each CP entry, the corresponding CP patch must either be null or have
     * the a format that matches its tag:
     * <ul>
     * <li>Integer, Long, Float, Double: the corresponding wrapper object type from
java.lang
     * <li>Utf8: a string (must have suitable syntax if used as signature or name)
     * <li>Class: any java.lang.Class object
     * <li>String: any object (not just a java.lang.String)
     * <li>InterfaceMethodRef: (NYI) a method handle to invoke on that call site's
arguments
```

```
     * </ul>
     * @params hostClass context for linkage, access control, protection domain,
and class loader
     * @params data      bytes of a class file
     * @params cpPatches where non-null entries exist, they replace corresponding
CP entries in data
     */

    //  定义匿名内部类
    public native Class<?> defineAnonymousClass(Class<?> hostClass, byte[] data,
Object[] cpPatches);


    /** Allocate an instance but do not run any constructor.
        Initializes the class if it has not yet been. */

    //  定位一个实例，但不运行构造函数
    public native Object allocateInstance(Class<?> cls) throws
InstantiationException;


    /** -----------------------------锁指令 (synchronized) -------------------------
------ */


    /** Lock the object.  It must get unlocked via {@link #monitorExit}. */
    //  对象加锁方法
    public native void monitorEnter(Object o);

    /**
     * Unlock the object.  It must have been locked via {@link
     * #monitorEnter}.
     */
    //  对象释放锁方法
    public native void monitorExit(Object o);

    /**
     * Tries to lock the object.  Returns true or false to indicate
     * whether the lock succeeded.  If it did, the object must be
     * unlocked via {@link #monitorExit}.
     */
    //  尝试对象加锁方法
    public native boolean tryMonitorEnter(Object o);

    /** Throw the exception without telling the verifier. */
    public native void throwException(Throwable ee);


    /** ----------------------------------- CAS ----------------------------- */

    //  根据指定对象，地址，期望值，更新值
    public final native boolean compareAndSwapObject(Object o, long offset,
                                                     Object expected,
```

```java
                                                 Object x);

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

public final native boolean compareAndSwapLong(Object o, long offset,
                                               long expected,
                                               long x);

/**
 * Fetches a reference value from a given Java variable, with volatile
 * load semantics. Otherwise identical to {@link #getObject(Object, long)}
 */

//  获取对象o的给定偏移地址的引用值（volatile方式）
public native Object getObjectVolatile(Object o, long offset);

public native void    putObjectVolatile(Object o, long offset, Object x);

/** Volatile version of {@link #getInt(Object, long)}  */
public native int     getIntVolatile(Object o, long offset);

/** Volatile version of {@link #putInt(Object, long, int)}  */
public native void    putIntVolatile(Object o, long offset, int x);

/** Volatile version of {@link #getBoolean(Object, long)}  */
public native boolean getBooleanVolatile(Object o, long offset);

/** Volatile version of {@link #putBoolean(Object, long, boolean)}  */
public native void    putBooleanVolatile(Object o, long offset, boolean x);

/** Volatile version of {@link #getByte(Object, long)}  */
public native byte    getByteVolatile(Object o, long offset);

/** Volatile version of {@link #putByte(Object, long, byte)}  */
public native void    putByteVolatile(Object o, long offset, byte x);

/** Volatile version of {@link #getShort(Object, long)}  */
public native short   getShortVolatile(Object o, long offset);

/** Volatile version of {@link #putShort(Object, long, short)}  */
public native void    putShortVolatile(Object o, long offset, short x);

/** Volatile version of {@link #getChar(Object, long)}  */
public native char    getCharVolatile(Object o, long offset);

/** Volatile version of {@link #putChar(Object, long, char)}  */
public native void    putCharVolatile(Object o, long offset, char x);

/** Volatile version of {@link #getLong(Object, long)}  */
public native long    getLongVolatile(Object o, long offset);
```

```java
    /** Volatile version of {@link #putLong(Object, long, long)}  */
    public native void    putLongVolatile(Object o, long offset, long x);

    /** Volatile version of {@link #getFloat(Object, long)}  */
    public native float   getFloatVolatile(Object o, long offset);

    /** Volatile version of {@link #putFloat(Object, long, float)}  */
    public native void    putFloatVolatile(Object o, long offset, float x);

    /** Volatile version of {@link #getDouble(Object, long)}  */
    public native double  getDoubleVolatile(Object o, long offset);

    /** Volatile version of {@link #putDouble(Object, long, double)}  */
    public native void    putDoubleVolatile(Object o, long offset, double x);

    /**
     * Version of {@link #putObjectVolatile(Object, long, Object)}
     * that does not guarantee immediate visibility of the store to
     * other threads. This method is generally only useful if the
     * underlying field is a Java volatile (or if an array cell, one
     * that is otherwise only accessed using volatile accesses).
     */
//  用于lazySet，适用于低延迟代码
    public native void    putOrderedObject(Object o, long offset, Object x);

    /** Ordered/Lazy version of {@link #putIntVolatile(Object, long, int)}  */
    public native void    putOrderedInt(Object o, long offset, int x);

    /** Ordered/Lazy version of {@link #putLongVolatile(Object, long, long)} */
    public native void    putOrderedLong(Object o, long offset, long x);

    /**
     * Unblock the given thread blocked on <tt>park</tt>, or, if it is
     * not blocked, cause the subsequent call to <tt>park</tt> not to
     * block.  Note: this operation is "unsafe" solely because the
     * caller must somehow ensure that the thread has not been
     * destroyed. Nothing special is usually required to ensure this
     * when called from Java (in which there will ordinarily be a live
     * reference to the thread) but this is not nearly-automatically
     * so when calling from native code.
     * @param thread the thread to unpark.
     *
     */
//  解除给定线程的阻塞
    public native void unpark(Object thread);

    /**
     * Block current thread, returning when a balancing
     * <tt>unpark</tt> occurs, or a balancing <tt>unpark</tt> has
     * already occurred, or the thread is interrupted, or, if not
     * absolute and time is not zero, the given time nanoseconds have
     * elapsed, or if absolute, the given deadline in milliseconds
     * since Epoch has passed, or spuriously (i.e., returning for no
     * "reason"). Note: This operation is in the Unsafe class only
     * because <tt>unpark</tt> is, so it would be strange to place it
     * elsewhere.
     */
    //  阻塞当前线程
    public native void park(boolean isAbsolute, long time);

    /**
     * Gets the load average in the system run queue assigned
     * to the available processors averaged over various periods of time.
     * This method retrieves the given <tt>nelem</tt> samples and
     * assigns to the elements of the given <tt>loadavg</tt> array.
     * The system imposes a maximum of 3 samples, representing
     * averages over the last 1,  5,  and  15 minutes, respectively.
     *
     * @params loadavg an array of double of size nelems
     * @params nelems the number of samples to be retrieved and
     *         must be 1 to 3.
     *
     * @return the number of samples actually retrieved; or -1
     *         if the load average is unobtainable.
     */
    public native int getLoadAverage(double[] loadavg, int nelems);

    // The following contain CAS-based Java implementations used on
    // platforms not supporting native instructions

    public final int getAndAddInt(Object o, long offset, int delta) {
        int v;
        do {
            v = getIntVolatile(o, offset);
        } while (!compareAndSwapInt(o, offset, v, v + delta));
        return v;
    }

    public final long getAndAddLong(Object o, long offset, long delta) {
        long v;
        do {
            v = getLongVolatile(o, offset);
        } while (!compareAndSwapLong(o, offset, v, v + delta));
        return v;
    }

    public final int getAndSetInt(Object o, long offset, int newValue) {
        int v;
        do {
            v = getIntVolatile(o, offset);
        } while (!compareAndSwapInt(o, offset, v, newValue));
        return v;
    }

    public final long getAndSetLong(Object o, long offset, long newValue) {
        long v;
        do {
            v = getLongVolatile(o, offset);
        } while (!compareAndSwapLong(o, offset, v, newValue));
        return v;
    }

    public final Object getAndSetObject(Object o, long offset, Object newValue) {
        Object v;
        do {
            v = getObjectVolatile(o, offset);
        } while (!compareAndSwapObject(o, offset, v, newValue));
        return v;
    }

    /** ---------------------------- 内存屏障操作 ------------------------ */

    /**
     * Ensures lack of reordering of loads before the fence
     * with loads or stores after the fence.
     * @since 1.8
     */
    //  表示该方法之前的所有load操作在内存屏障之前完成
    public native void loadFence();

    /**
     * Ensures lack of reordering of stores before the fence
     * with loads or stores after the fence.
     * @since 1.8
     */
    //  表示该方法之前的所有store操作在内存屏障之前完成
    public native void storeFence();

    /**
     * Ensures lack of reordering of loads or stores before the fence
     * with loads or stores after the fence.
     * @since 1.8
     */
    //  表示该方法之前的所有load、store操作在内存屏障之前完成，这个相当于上面两个的合体功能
    public native void fullFence();

    /**
     * Throws IllegalAccessError; for use by the VM.
     * @since 1.8
     */
    private static void throwIllegalAccessError() {
       throw new IllegalAccessError();
    }

}
```