深入理解sun.misc.Unsafe原理

原创

前言

Unsafe类在JDK源码中被广泛使用，在Spark使用off-heap memory时也会使用到，该类功能很强大，涉及到类加载机制（深入理解ClassLoader工作机制），其实例一般情况是获取不到的，源码中的设计是采用单例模式，不是系统加载初始化就会抛出SecurityException异常。

这个类的提供了一些绕开JVM的更底层功能，基于它的实现可以提高效率。但是，它是一把双刃剑：正如它的名字所预示的那样，它是Unsafe的，它所分配的内存需要手动free（不被GC回收）。如果对Unsafe类理解的不够透彻，就进行使用的话，就等于给自己挖了无形之坑，最为致命。

由于sun并没有将其开源，也没给出官方的Document，所以笔者只能参考一些博客（如Java Magic. Part 4: sun.misc.Unsafe）以及Unsafe在JDK源码中的一些使用，来加深理解。

获取Unsafe类的实例

1. 必须是Bootstrap ClassLoader加载的类

   getUnsafe方法源码:

```
private Unsafe() {
}

@CallerSensitive
public static Unsafe getUnsafe() {
    Class var0 = Reflection.getCallerClass();
    if (!VM.isSystemDomainLoader(var0.getClassLoader())) {
        throw new SecurityException("Unsafe");
    } else {
        return theUnsafe;
    }
}
```


   isSystemDomainLoader:

```
public static boolean isSystemDomainLoader(ClassLoader var0) {
    return var0 == null;
}
```

2. 通过反射暴力获取

```
// 通过反射实例化Unsafe
Field f = Unsafe.class.getDeclaredField("theUnsafe");
f.setAccessible(true);
Unsafe unsafe = (Unsafe) f.get(null);
```

Unsafe类中的核心方法

```
    //重新分配内存
    public native long reallocateMemory(long address, long bytes);

    //分配内存
    public native long allocateMemory(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);

    //从一个内存块拷贝到另一个内存块
    public native void copyMemory(Object srcBase, long srcOffset, Object destBase, long destOffset, long bytes)
;

    //获取值，不管java的访问限制，其他有类似的getInt，getDouble，getLong，getChar等等
    public native Object getObject(Object o, long offset);

    //设置值，不管java的访问限制，其他有类似的putInt,putDouble，putLong，putChar等等
    public native void putObject(Object o, long offset);
```

```
    //从一个给定的内存地址获取本地指针，如果不是allocateMemory方法的，结果将不确定
    public native long getAddress(long address);

    //存储一个本地指针到一个给定的内存地址,如果地址不是allocateMemory方法的，结果将不确定
    public native void putAddress(long address, long x);

    //该方法返回给定field的内存地址偏移量，这个值对于给定的filed是唯一的且是固定不变的
    public native long staticFieldOffset(Field f);

    //报告一个给定的字段的位置，不管这个字段是private，public还是保护类型，和staticFieldBase结合使用
    public native long objectFieldOffset(Field f);

    //获取一个给定字段的位置
    public native Object staticFieldBase(Field f);

    //确保给定class被初始化，这往往需要结合基类的静态域（field）
    public native void ensureClassInitialized(Class c);

    //可以获取数组第一个元素的偏移地址
    public native int arrayBaseOffset(Class arrayClass);

    //可以获取数组的转换因子，也就是数组中元素的增量地址。将arrayBaseOffset与arrayIndexScale配合使用， 可以定位数组中每个元素在内存中的位置
    public native int arrayIndexScale(Class arrayClass);

    //获取本机内存的页数，这个值永远都是2的幂次方
    public native int pageSize();

    //告诉虚拟机定义了一个没有安全检查的类，默认情况下这个类加载器和保护域来着调用者类
    public native Class defineClass(String name, byte[] b, int off, int len, ClassLoader loader, ProtectionDomain protectionDomain);

    //定义一个类，但是不让它知道类加载器和系统字典
    public native Class defineAnonymousClass(Class hostClass, byte[] data, Object[] cpPatches);

    //锁定对象，必须是没有被锁的
    public native void monitorEnter(Object o);

    //解锁对象
    public native void monitorExit(Object o);

    //试图锁定对象，返回true或false是否锁定成功，如果锁定，必须用monitorExit解锁
    public native boolean tryMonitorEnter(Object o);

    //引发异常，没有通知
    public native void throwException(Throwable ee);

    //CAS，如果对象偏移量上的值=期待值，更新为x,返回true.否则false.类似的有compareAndSwapInt,compareAndSwapLong,compareAndSwapBoolean,compareAndSwapChar等等。
    public final native boolean compareAndSwapObject(Object o, long offset,  Object expected, Object x);

    // 该方法获取对象中offset偏移地址对应的整型field的值,支持volatile load语义。类似的方法有getIntVolatile，getBooleanVolatile等等
    public native Object getObjectVolatile(Object o, long offset);

    //线程调用该方法，线程将一直阻塞直到超时，或者是中断条件出现。
    public native void park(boolean isAbsolute, long time);

    //终止挂起的线程，恢复正常.java.util.concurrent包中挂起操作都是在LockSupport类实现的，也正是使用这两个方法
```

```
    public native void unpark(Object thread);

    //获取系统在不同时间系统的负载情况
    public native int getLoadAverage(double[] loadavg, int nelems);

    //创建一个类的实例，不需要调用它的构造函数、初使化代码、各种JVM安全检查以及其它的一些底层的东西。即使构造函数是私有，我们也可以通过这个方法创建它的实例,对于单例模式，简直是噩梦。
    public native Object allocateInstance(Class cls) throws InstantiationException;
```

总的来说就是这么几类

（1）Info相关。主要返回某些低级别的内存信息：addressSize(), pageSize()

（2）Objects相关。主要提供Object和它的域操纵方法：allocateInstance(),objectFieldOffset()

（3）Class相关。主要提供Class和它的静态域操纵方法：staticFieldOffset(),defineClass(),defineAnonymousClass(),ensureClassInitialized()

（4）Arrays相关。数组操纵方法：arrayBaseOffset(),arrayIndexScale()

（5）Synchronization相关。主要提供低级别同步原语（如基于CPU的CAS（Compare-And-Swap）原语）：monitorEnter(),tryMonitorEnter(),monitorExit(),compareAndSwapInt(),putOrderedInt()

（6）Memory相关。直接内存访问方法（绕过JVM堆直接操纵本地内存）：allocateMemory(),copyMemory(),freeMemory(),getAddress(),getInt(),putInt()

Unsafe该怎么用？举些例子

1、Unsafe.allocateInstance

```
public static void main(String[] args) throws Exception {
        Field f = Unsafe.class.getDeclaredField("theUnsafe");
        f.setAccessible(true);
        Unsafe unsafe = (Unsafe) f.get(null);

        A o1 = new A(); // constructor
        o1.a(); // prints 1
        A o2 = A.class.newInstance(); // reflection
        o2.a(); // prints 1
        A o3 = (A) unsafe.allocateInstance(A.class); // unsafe
        o3.a(); // prints 0
    }

    static class A {
        private long a; // not initialized value, default 0
        public A() {
            this.a = 1; // initialization
        }
        public long a() {
            return this.a;
        }
    }
```

allocateInstance()根本没有进入构造方法，对于单例模式，简直是噩梦。

2、内存修改，绕过安全检查器（Unsafe.objectFieldOffset）

```
public static void main(String[] args) throws Exception {
        Field f = Unsafe.class.getDeclaredField("theUnsafe");
        f.setAccessible(true);
        Unsafe unsafe = (Unsafe) f.get(null);

        Guard guard = new Guard();
        guard.giveAccess();   // false, no access

        // bypass
        Field field = guard.getClass().getDeclaredField("ACCESS_ALLOWED");
        unsafe.putInt(guard, unsafe.objectFieldOffset(field), 42); // memory corruption
        guard.giveAccess(); // true, access granted
    }

    static class Guard {
        private int ACCESS_ALLOWED = 1;
        public boolean giveAccess() {
            return 42 == ACCESS_ALLOWED;
        }
    }
```

通过获取目标对象的字段在内存中的offset，并使用putInt()方法，类的ACCESS_ALLOWED被修改。在已知类结构的时候，数据的偏移总是可以获得的（与c++中的类中数据的偏移计算是一致的）。

3、sizeOf 计算内存大小（Unsafe.getDeclaredFields和Unsafe.objectFieldOffset）

```
public static void main(String[] args) throws Exception {
        Guard guard = new Guard();
        sizeOf(guard); // 16, the size of guard
    }

    static class Guard {
        private int ACCESS_ALLOWED = 1;
        public boolean giveAccess() {
            return 42 == ACCESS_ALLOWED;
        }
    }

    public static long sizeOf(Object o) throws Exception {
        Field f = Unsafe.class.getDeclaredField("theUnsafe");
        f.setAccessible(true);
        Unsafe unsafe = (Unsafe) f.get(null);

        HashSet<Field> fields = new HashSet();
        Class c = o.getClass();
        while (c != Object.class) {
            for (Field field : c.getDeclaredFields()) {
                if ((field.getModifiers() & Modifier.STATIC) == 0) {
                    fields.add(field);
                }
            }
            c = c.getSuperclass();
        }

        // get offset
        long maxSize = 0;
        for (Field field : fields) {
            long offset = unsafe.objectFieldOffset(field);
            if (offset > maxSize) {
                maxSize = offset;
            }
        }
        return ((maxSize/8) + 1) * 8;   // padding
    }
```

算法的思路非常清晰：从底层子类开始，依次取出它自己和它的所有超类的非静态域，放置到一个HashSet中（重复的只计算一次，Java是单继承），然后使用objectFieldOffset()获得一个最大偏移，最后还考虑了对齐。

4、实现Java浅复制

```
public static void main(String[] args) throws Exception {
        Field f = Unsafe.class.getDeclaredField("theUnsafe");
        f.setAccessible(true);
        unsafe = (Unsafe) f.get(null);
        Guard guard = new Guard();
        shallowCopy(guard);
    }

    private static Unsafe getUnsafe() {
        return unsafe;
    }

    static Object shallowCopy(Object obj) throws Exception {
        long size = sizeOf(obj);
        long start = toAddress(obj);
        long address = getUnsafe().allocateMemory(size);
```

```
        long address = getUnsafe().allocateMemory(size);
        getUnsafe().copyMemory(start, address, size);
        return fromAddress(address);
    }

    static long toAddress(Object obj) {
        Object[] array = new Object[]{obj};
        long baseOffset = getUnsafe().arrayBaseOffset(Object[].class);
        return normalize(getUnsafe().getLong(array, baseOffset));
    }

    static Object fromAddress(long address) {
        Object[] array = new Object[] {null};
        long baseOffset = getUnsafe().arrayBaseOffset(Object[].class);
        getUnsafe().putLong(array, baseOffset, address);
        return array[0];
    }

    static class Guard {
        private int ACCESS_ALLOWED = 1;
        public boolean giveAccess() {
            return 42 == ACCESS_ALLOWED;
        }
    }

    public static long sizeOf(Object o) throws Exception {
        Field f = Unsafe.class.getDeclaredField("theUnsafe");
        f.setAccessible(true);
        Unsafe unsafe = (Unsafe) f.get(null);

        HashSet<Field> fields = new HashSet();
        Class c = o.getClass();
        while (c != Object.class) {
            for (Field field : c.getDeclaredFields()) {
                if ((field.getModifiers() & Modifier.STATIC) == 0) {
                    fields.add(field);
                }
            }
            c = c.getSuperclass();
        }

        // get offset
        long maxSize = 0;
        for (Field field : fields) {
            long offset = unsafe.objectFieldOffset(field);
            if (offset > maxSize) {
                maxSize = offset;
            }
        }
        return ((maxSize / 8) + 1) * 8;   // padding
    }
```

思路很简单，利用Unsafe.copyMemory()，将老地址及其指向的对象的size，拷贝到新的内存地址上。
并且浅复制函数可以应用于任意java对象，它的尺寸是动态计算的。
（在实际测试的时候，执行unsafe.copyMemory时，JVM会输出hs_err_pid.log日志然后挂掉，该问题还有待排查）

5、隐藏密码

一般密码都要存成byte[]或者char[]数组，为什么呢？？
因为我们使用完了，可以直接将他们设为null。但如果密码存在String中，将其设为null，密码实际任然存在在内存中，等到GC后，才能被释放，就很不安全。
所以当我们把密码字段存储在String中时，在密码字段使用完之后，最安全的做法是：将它的值覆盖。
很多不再需要的，但是又是比较机密的对象，想快点消灭证据，都可以通过这种方法来消除。

```
Field stringValue = String.class.getDeclaredField("value");
    stringValue.setAccessible(true);
    char[] mem = (char[]) stringValue.get(password);
    for (int i = 0; i < mem.length; i++) {
        mem[i] = '?';
    }
```

6、多继承

在java中没有多继承，除非我们能在这些不同的类互相强制转换。

```
long intClassAddress = normalize(getUnsafe().getInt(new Integer(0), 4L));
long strClassAddress = normalize(getUnsafe().getInt("", 4L));
getUnsafe().putAddress(intClassAddress + 36, strClassAddress);
```

上面这段代码将String添加为int的父类，所以我们转换的时候就不会报错了。

```
(String) (Object) (new Integer(666))
```

7、动态加载类

标准的动态加载类的方法是Class.forName()(在编写jdbc程序时，记忆深刻)，使用Unsafe也可以动态加载java 的class文件。
我们可以在程序运行是动态加载编译好的.class文件，不明白ClassLoader的可以参考<深入理解ClassLoader工作机制>
具体是怎么实现的呢？？

```
// 我们首先读取一个class文件到byte数组中
byte[] classContents = getClassContent();
// 然后通过Unsafe.defineClass()来加载对应的Class
Class c = getUnsafe().defineClass(
              null, classContents, 0, classContents.length);
// 最后调用
c.getMethod("a").invoke(c.newInstance(), null); // 1

private static byte[] getClassContent() throws Exception {
    File f = new File("/home/mishadoff/tmp/A.class");
    FileInputStream input = new FileInputStream(f);
    byte[] content = new byte[(int)f.length()];
    input.read(content);
    input.close();
    return content;
}
```

动态加载、代理、切片等功能中可以应用。

8、快速序列化

我们都知道标准的Java Serializable速度很慢，它还限制类必须有public无参构造函数。
Externalizable相对好些，但它需要为序列化的类指定schema。
更流行的如kyro，在小内存的情况下不适用。
序列化过程：

1. 用反射构建对象的schema
2. 用Unsafe中的getLong, getInt, getObject等方法来检索对象中字段的值。
3. 增加对象对应的类的标示符，来标记序列化结果。
4. 将结果写入文件或者输出流。(可以增加压缩来减小序列化结果)

反序列化过程：

1. 使用Unsafe.allocateInstance()来实例化一个被序列化的对象。(不需要执行构造函数)
2. 构建schema，同序列化过程中的第一步。
3. 从文件或者输入流中读取所有的字段。
4. 用Unsafe中的putLong, putInt, putObject等方法来填充该对象。

在kyro序列化中，也有一些使用Unsafe的尝试：https://code.google.com/archive/p/kryo/issues/75
（笔者还没仔细看，有兴趣的可以阅读一下。。估计能看到这儿的也挺累了）

9、在非Java堆中分配内存

使用java 的new会在堆中为对象分配内存，并且对象的生命周期内，会被JVM GC管理。
Unsafe分配的内存，不受Integer.MAX_VALUE的限制，并且分配在非堆内存，使用它时，需要非常谨慎：忘记手动回收时，会产生内存泄露；非法的地址访问时，会导致JVM崩溃。在需要分配大的连续区域、实时编程（不能容忍JVM延迟）时，可以使用它。java.nio使用这一技术。
Spark中的Netty也使用了这个技术。
在Spark UnsafeMemoryAllocator源码中我们可以看到其使用了Unsafe.allocateMemory()并会抛出OOM异常：

```
@Override
public MemoryBlock allocate(long size) throws OutOfMemoryError {
  long address = Platform.allocateMemory(size);
  MemoryBlock memory = new MemoryBlock(null, address, size);
  if (MemoryAllocator.MEMORY_DEBUG_FILL_ENABLED) {
    memory.fill(MemoryAllocator.MEMORY_DEBUG_FILL_CLEAN_VALUE);
  }
  return memory;
}
```


10、大数组

Java的数组最大容量受常量Integer.MAX_VALUE的限制，如果我们用直接申请内存的方式去创建数组，那么数组大小只会收到堆的大小的限制。

```
    private static Unsafe unsafe;

    public static void main(String[] args) throws Exception {
        Field f = Unsafe.class.getDeclaredField("theUnsafe");
        f.setAccessible(true);
        unsafe = (Unsafe) f.get(null);
        // 设置数组大小为Integer.MAX_VALUE的2倍
        long SUPER_SIZE = (long) Integer.MAX_VALUE * 2;
        SuperArray array = new SuperArray(SUPER_SIZE);
        System.out.println("Array size:" + array.size()); // 4294967294
        int sum = 0;
        for (int i = 0; i < 100; i++) {
            array.set((long) Integer.MAX_VALUE + i, (byte) 3);
            sum += array.get((long) Integer.MAX_VALUE + i);
        }
        System.out.println("Sum of 100 elements:" + sum);  // 300
    }

    private static Unsafe getUnsafe() {
        return unsafe;
    }

    static class SuperArray {
        private final static int BYTE = 1;
        private long size;
        private long address;

        public SuperArray(long size) {
            this.size = size;
            address = getUnsafe().allocateMemory(size * BYTE);
        }
        public void set(long i, byte value) {
            getUnsafe().putByte(address + i * BYTE, value);
        }
        public int get(long idx) {
            return getUnsafe().getByte(address + idx * BYTE);
        }
        public long size() {
            return size;
        }
    }
```

输出结果：

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_161.jdk/Contents/Home/bin/java ...
Connected to the target VM, address: '127.0.0.1:59973', transport: 'socket'
Array size:4294967294
Disconnected from the target VM, address: '127.0.0.1:59973', transport: 'socket'
Sum of 100 elements:300

Process finished with exit code 0
```


Unsafe类的底层源码实现

CAS

终于等到CAS出场了。CAS在AQS、ConcurrentHashmap、ForkJoinPool、FutureTask、StampedLock等都有大量的应用。甚至可以说很多Java程序员觉得CAS就是他们所了解的最小单元了。确实，也包括我。。
但是总觉得源码看到这儿了，不往下看就像一件事做了一半。
natUnsafe.cc

```
// natUnsafe.cc - Implementation of sun.misc.Unsafe native methods.

/** Copyright (C) 2006, 2007
   Free Software Foundation
   This file is part of libgcj.
This software is copyrighted work licensed under the terms of the
Libgcj License.  Please consult the file "LIBGCJ_LICENSE" for
details.  */

#include <gcj/cni.h>
#include <gcj/field.h>
#include <gcj/javaprims.h>
#include <jvm.h>
#include <sun/misc/Unsafe.h>
#include <java/lang/System.h>
#include <java/lang/InterruptedException.h>

#include <java/lang/Thread.h>
#include <java/lang/Long.h>

#include "sysdep/locks.h"

// Use a spinlock for multi-word accesses
class spinlock
{
  static volatile obj_addr_t lock;

public:

spinlock ()
  {
    while (! compare_and_swap (&lock, 0, 1))
      _Jv_ThreadYield ();
  }
  ~spinlock ()
  {
    release_set (&lock, 0);
  }
};

// This is a single lock that is used for all synchronized accesses if
// the compiler can't generate inline compare-and-swap operations.  In
// most cases it'll never be used, but the i386 needs it for 64-bit
// locked accesses and so does PPC32.  It's worth building libgcj with
// target=i486 (or above) to get the inlines.
volatile obj_addr_t spinlock::lock;


static inline bool
compareAndSwap (volatile jint *addr, jint old, jint new_val)
{
  jboolean result = false;
  spinlock lock;
  if ((result = (*addr == old)))
```

```
    *addr = new_val;
  return result;
}

static inline bool
compareAndSwap (volatile jlong *addr, jlong old, jlong new_val)
{
  jboolean result = false;
  spinlock lock;
  if ((result = (*addr == old)))
    *addr = new_val;
  return result;
}

static inline bool
compareAndSwap (volatile jobject *addr, jobject old, jobject new_val)
{
  jboolean result = false;
  spinlock lock;
  if ((result = (*addr == old)))
    *addr = new_val;
  return result;
}



jlong
sun::misc::Unsafe::objectFieldOffset (::java::lang::reflect::Field *field)
{
  _Jv_Field *fld = _Jv_FromReflectedField (field);
  // FIXME: what if it is not an instance field?
  return fld->getOffset();
}

jint
sun::misc::Unsafe::arrayBaseOffset (jclass arrayClass)
{
  // FIXME: assert that arrayClass is array.
  jclass eltClass = arrayClass->getComponentType();
  return (jint)(jlong) _Jv_GetArrayElementFromElementType (NULL, eltClass);
}

jint
sun::misc::Unsafe::arrayIndexScale (jclass arrayClass)
{
  // FIXME: assert that arrayClass is array.
  jclass eltClass = arrayClass->getComponentType();
  if (eltClass->isPrimitive())
    return eltClass->size();
  return sizeof (void *);
}

// These methods are used when the compiler fails to generate inline
// versions of the compare-and-swap primitives.

jboolean
sun::misc::Unsafe::compareAndSwapInt (jobject obj, jlong offset,
                      jint expect, jint update)
{
  jint *addr = (jint *)((char *)obj + offset);
  return compareAndSwap (addr, expect, update);
```

```
  return compareAndSwap (addr, expect, update);
}

jboolean
sun::misc::Unsafe::compareAndSwapLong (jobject obj, jlong offset,
                       jlong expect, jlong update)
{
  volatile jlong *addr = (jlong*)((char *) obj + offset);
  return compareAndSwap (addr, expect, update);
}

jboolean
sun::misc::Unsafe::compareAndSwapObject (jobject obj, jlong offset,
                     jobject expect, jobject update)
{
  jobject *addr = (jobject*)((char *) obj + offset);
  return compareAndSwap (addr, expect, update);
}

void
sun::misc::Unsafe::putOrderedInt (jobject obj, jlong offset, jint value)
{
  volatile jint *addr = (jint *) ((char *) obj + offset);
  *addr = value;
}

void
sun::misc::Unsafe::putOrderedLong (jobject obj, jlong offset, jlong value)
{
  volatile jlong *addr = (jlong *) ((char *) obj + offset);
  spinlock lock;
  *addr = value;
}

void
sun::misc::Unsafe::putOrderedObject (jobject obj, jlong offset, jobject value)
{
  volatile jobject *addr = (jobject *) ((char *) obj + offset);
  *addr = value;
}

void
sun::misc::Unsafe::putIntVolatile (jobject obj, jlong offset, jint value)
{
  write_barrier ();
  volatile jint *addr = (jint *) ((char *) obj + offset);
  *addr = value;
}

void
sun::misc::Unsafe::putLongVolatile (jobject obj, jlong offset, jlong value)
{
  volatile jlong *addr = (jlong *) ((char *) obj + offset);
  spinlock lock;
  *addr = value;
}

void
sun::misc::Unsafe::putObjectVolatile (jobject obj, jlong offset, jobject value)
{
```

```
  write_barrier ();
  volatile jobject *addr = (jobject *) ((char *) obj + offset);
  *addr = value;
}

#if 0  // FIXME
void
sun::misc::Unsafe::putInt (jobject obj, jlong offset, jint value)
{
  jint *addr = (jint *) ((char *) obj + offset);
  *addr = value;
}
#endif

void
sun::misc::Unsafe::putLong (jobject obj, jlong offset, jlong value)
{
  jlong *addr = (jlong *) ((char *) obj + offset);
  spinlock lock;
  *addr = value;
}

void
sun::misc::Unsafe::putObject (jobject obj, jlong offset, jobject value)
{
  jobject *addr = (jobject *) ((char *) obj + offset);
  *addr = value;
}

jint
sun::misc::Unsafe::getIntVolatile (jobject obj, jlong offset)
{
  volatile jint *addr = (jint *) ((char *) obj + offset);
  jint result = *addr;
  read_barrier ();
  return result;
}

jobject
sun::misc::Unsafe::getObjectVolatile (jobject obj, jlong offset)
{
  volatile jobject *addr = (jobject *) ((char *) obj + offset);
  jobject result = *addr;
  read_barrier ();
  return result;
}

jlong
sun::misc::Unsafe::getLong (jobject obj, jlong offset)
{
  jlong *addr = (jlong *) ((char *) obj + offset);
  spinlock lock;
  return *addr;
}

jlong
sun::misc::Unsafe::getLongVolatile (jobject obj, jlong offset)
{
  volatile jlong *addr = (jlong *) ((char *) obj + offset);
  spinlock lock;
```

```
  spinlock lock;
  return *addr;
}

void
sun::misc::Unsafe::unpark (::java::lang::Thread *thread)
{
  natThread *nt = (natThread *) thread->data;
  nt->park_helper.unpark ();
}

void
sun::misc::Unsafe::park (jboolean isAbsolute, jlong time)
{
  using namespace ::java::lang;
  Thread *thread = Thread::currentThread();
  natThread *nt = (natThread *) thread->data;
  nt->park_helper.park (isAbsolute, time);
}
```

我们可以看到compareAndSwap就这么几行代码:

```
static inline bool
compareAndSwap (volatile jint *addr, jint old, jint new_val)
{
  jboolean result = false;
  spinlock lock;
  // result=原先指针指向的地址的值(*addr)是否与旧的值(old)相等
  if ((result = (*addr == old)))
    // 如果相等则把内存修改为新值
    *addr = new_val;
  return result;
}
```

好了，现在终于彻底知道了整个链路的原理了。

ps.写了N个晚上，总算是搞定了。。如果您觉得写的好，想要转载，请麻烦您注明出处，感谢！！