Java中的Unsafe 硬件级别的原子性操作

zhong0316 关注
2 2019.02.21 10:01:08 字数 1,195 阅读 32,468

Java和C++语言的一个重要区别就是Java中我们无法直接操作一块内存区域，不能像C++中那样可以自己申请内存和释放内存。Java中的Unsafe类为我们提供了类似C++手动管理内存的能力。

Unsafe类，全限定名是 `sun.misc.Unsafe` ，从名字中我们可以看出来这个类对普通程序员来说是"危险"的，一般应用开发者不会用到这个类。

Unsafe类是"final"的，不允许继承。且构造函数是private的:

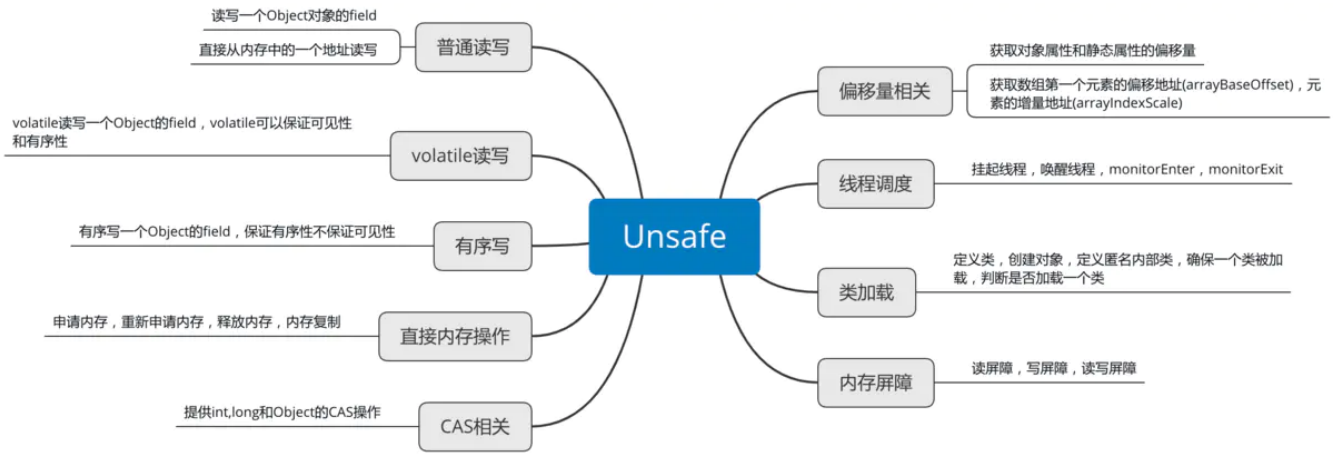
```
public final class Unsafe {
    private static final Unsafe theUnsafe;
    public static final int INVALID_FIELD_OFFSET = -1;

    private static native void registerNatives();
    // 构造函数是private的，不允许外部实例化
    private Unsafe() {
    }
    ...
}
```

Unsafe无法实例化，那么怎么获取Unsafe呢？答案就是通过反射来获取Unsafe：

```
public Unsafe getUnsafe() throws IllegalAccessException {
    Field unsafeField = Unsafe.class.getDeclaredFields()[0];
    unsafeField.setAccessible(true);
    Unsafe unsafe = (Unsafe) unsafeField.get(null);
    return unsafe;
}
```

主要功能

Unsafe的功能如下图：



Unsafe-xmind

Java并发 --- volatile关键字
阅读 485

ReentrantLock源码分析
阅读 703

通过Unsafe可以读写一个类的属性，即使这个属性是私有的，也可以对这个属性进行读写。

读写一个Object属性的相关方法

普通读

```
public native int getInt(Object var1, long var2);

public native void putInt(Object var1, long var2, int var4);
```

getInt用于从对象的指定偏移地址处读取一个int。putInt用于在对象指定偏移地址处写入一个int。其他的primitive type也有对应的方法。

Unsafe**还可以直接在一个地址上读写**

```
public native byte getByte(long var1);

public native void putByte(long var1, byte var3);
```

getByte用于从指定内存地址处开始读取一个byte。putByte用于从指定内存地址写入一个byte。其他的primitive type也有对应的方法。

volatile读写

普通的读写无法保证可见性和有序性，而volatile读写就可以保证可见性和有序性。

```
3 public native void putIntVolatile(Object var1, long var2, int var4);
```

```
public native void putIntVolatile(Object var1, long var2, int var4);
```

getIntVolatile方法用于在对象指定偏移地址处volatile读取一个int。putIntVolatile方法用于在对象指定偏移地址处volatile写入一个int。

volatile读写相对普通读写是更加昂贵的，因为需要保证可见性和有序性，而与volatile写入相比putOrderedXX写入代价相对较低，putOrderedXX写入不保证可见性，但是保证有序性，所谓有序性，就是保证指令不会重排序。

有序写入

有序写入只保证写入的有序性，不保证可见性，就是说一个线程的写入不保证其他线程立马可见。

```
1 public native void putOrderedObject(Object var1, long var2, Object var4);
2
3 public native void putOrderedInt(Object var1, long var2, int var4);
4
5 public native void putOrderedLong(Object var1, long var2, long var4);
```

直接内存操作

我们都知道Java不可以直接对内存进行操作，对象内存的分配和回收都是由JVM帮助我们实现的。但是Unsafe为我们在Java中提供了直接操作内存的能力。

```
// 重新分配内存
public native long reallocateMemory(long var1, long var3);
// 内存初始化
public native void setMemory(long var1, long var3, byte var5);
// 内存复制
public native void copyMemory(Object var1, long var2, Object var4, long var5, long var7);
// 清除内存
public native void freeMemory(long var1);
```

CAS相关

JUC中大量运用了CAS操作，可以说CAS操作是JUC的基础，因此CAS操作是非常重要的。
Unsafe中提供了int,long和Object的CAS操作：

```
public final native boolean compareAndSwapObject(Object var1, long var2, Object var4, Object v

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);
```

CAS一般用于乐观锁，它在Java中有广泛的应用，ConcurrentHashMap，
ConcurrentLinkedQueue中都有用到CAS来实现乐观锁。

偏移量相关

```
public native long staticFieldOffset(Field var1);
```



```
3  public native long objectFieldOffset(Field var1);
4
5  public native Object staticFieldBase(Field var1);
6
7  public native int arrayBaseOffset(Class<?> var1);
8
9  public native int arrayIndexScale(Class<?> var1);
```

staticFieldOffset方法用于获取静态属性Field在对象中的偏移量，读写静态属性时必须获取其偏移量。objectFieldOffset方法用于获取非静态属性Field在对象实例中的偏移量，读写对象的非静态属性时会用到这个偏移量。staticFieldBase方法用于返回Field所在的对象。arrayBaseOffset方法用于返回数组中第一个元素实际地址相对整个数组对象的地址的偏移量。arrayIndexScale方法用于计算数组中第一个元素所占用的内存空间。

线程调度

```
1  public native void unpark(Object var1);
2
3  public native void park(boolean var1, long var2);
4
5  public native void monitorEnter(Object var1);
6
7  public native void monitorExit(Object var1);
8
9  public native boolean tryMonitorEnter(Object var1);
```

park方法和unpark方法相信看过LockSupport类的都不会陌生，这两个方法主要用来挂起和唤醒线程。
LockSupport中的park和unpark方法正是通过Unsafe来实现的：

通过Unsafe的park方法来阻塞当前线程，注意此方法将当前线程阻塞后，当前线程就不会继续往下走了，直到其他线程unpark此线程

```
// 挂起线程
public static void park(Object blocker) {
    Thread t = Thread.currentThread();
    setBlocker(t, blocker); // 通过Unsafe的putObject方法设置阻塞阻塞当前线程的blocker
    UNSAFE.park(false, 0L); // 通过Unsafe的park方法来阻塞当前线程，注意此方法将当前线程阻塞后，当前
    setBlocker(t, null); // 清除blocker
}

// 唤醒线程
public static void unpark(Thread thread) {
    if (thread != null)
        UNSAFE.unpark(thread);
}
```

monitorEnter方法和monitorExit方法用于加锁，Java中的synchronized锁就是通过这两个指令来实现的。

类加载

```
public native Class<?> defineClass(String var1, byte[] var2, int var3, int var4, ClassLoader v

public native Class<?> defineAnonymousClass(Class<?> var1, byte[] var2, Object[] var3);

public native Object allocateInstance(Class<?> var1) throws InstantiationException;

public native boolean shouldBeInitialized(Class<?> var1);
```

defineAnonymousClass用于动态的创建一个匿名内部类。
allocateInstance方法用于创建一个类的实例，但是不会调用这个实例的构造方法，如果这个类还未被初始化，则初始化这个类。
shouldBeInitialized方法用于判断是否需要初始化一个类。
ensureClassInitialized方法用于保证已经初始化过一个类。

内存屏障

```
1  public native void loadFence();
2
3  public native void storeFence();
4
5  public native void fullFence();
```

loadFence：保证在这个屏障之前的所有读操作都已经完成。
storeFence：保证在这个屏障之前的所有写操作都已经完成。
fullFence：保证在这个屏障之前的所有读写操作都已经完成。