# sun.misc.Unsafe操作手册

原创

朱小厮 于 2018-11-15 23:58:12 发布 

分类专栏： java JAVA相关技术



本文链接：https://blog.csdn.net/u013256816/article/details/84112878



java 同时被 2 个专栏收录

134 篇文章 12 订阅

订阅专栏


JAVA相关技术

55 篇文章 60 订阅

订阅专栏

---

欢迎支持笔者新作：《深入理解Kafka:核心设计与实践原理》和《RabbitMQ实战指南》，同时欢迎关注笔者的微信公众号：朱小厮的博客。


欢迎跳转到本文的原文链接：https://honeypps.com/java/sun-misc-unsafe-operation-manual/

---

Java是一个安全的开发工具，它阻止开发人员犯很多低级的错误，而大部份的错误都是基于内存管理方面的。如果你想搞破坏，可以使用Unsafe这个类。这个类是属于sun.* API中的类，并且它不是J2SE中真正的一部份，因此你可能找不到任何的官方文档，更可悲的是，它也没有比较好的代码文档。

## 实例化sun.misc.Unsafe

如果你尝试创建Unsafe类的实例，基于以下两种原因是不被允许的：

1. Unsafe类的构造函数是私有的；
2. 虽然它有静态的getUnsafe()方法，但是如果你尝试调用Unsafe.getUnsafe()，会得到一个SecutiryException。这个类只有被JDK信任的类实例化。
   但是这总会是有变通的解决办法的，一个简单的方式就是使用反射进行实例化：

```
Field f = Unsafe.class.getDeclaredField("theUnsafe");
f.setAccessible(true);
Unsafe unsafe = (Unsafe) f.get(null);
```

## 使用sun.misc.Unsafe

### 1. 创建实例

通过allocateInstance()方法，你可以创建一个类的实例，但是却不需要调用它的构造函数、初始化代码、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。

```
public class UnsafePlayer {
    public static void main(String[] args) throws Exception {
        //通过反射实例化Unsafe
        Field f = Unsafe.class.getDeclaredField("theUnsafe");
        f.setAccessible(true);
        Unsafe unsafe = (Unsafe) f.get(null);

        //实例化私有的构造函数
        Player player = (Player) unsafe.allocateInstance(Player.class);
        player.setName("jack");
        System.out.println(player.getName());
    }

    public static class Player{
        @Getter @Setter private String name;
        private Player(){}
    }
}
```

### 2. 可以分配内存和释放内存

类中提供的3个本地方法allocateMemory、reallocateMemory、freeMemory分别用于分配内存，扩充内存和释放内存，与C语言中的3个方法对应。

### 3. 通过内存偏移地址修改变量值

**public native long objectFieldOffset(Field field);**
返回指定静态field的内存地址偏移量，在这个类的其他方法中这个值只是被用作一个访问特定field的一个方式。这个值对于给定的field是唯一的，并且后续对该方法的调用都应该返回相同的值。

**public native int arrayBaseOffset(Class arrayClass);**
获取给定数组中第一个元素的偏移地址。为了存取数组中的元素，这个偏移地址与arrayIndexScale方法的非0返回值一起被使用。
**public native int arrayIndexScale(Class arrayClass)**
获取用户给定数组寻址的换算因子。如果不能返回一个合适的换算因子的时候就会返回0。这个返回值能够与arrayBaseOffset一起使用去存取这个数组class中的元素

**public native boolean compareAndSwapInt(Object obj, long offset,int expect, int update);**
在obj的offset位置比较integer field和期望的值，如果相同则更新。这个方法的操作应该是原子的，因此提供了一种不可中断的方式更新integer field。当然还有与Object、Long对应的compareAndSwapObject和compareAndSwapLong方法。

**public native void putOrderedInt(Object obj, long offset, int value);**
设置obj对象中offset偏移地址对应的整型field的值为指定值。这是一个有序或者有延迟的putIntVolatile方法，并且不保证值的改变被其他线程立即看到。只有在field被volatile修饰并且期望被意外修改的时候使用才有用。当然还有与Object、Long对应的putOrderedObject和putOrderedLong方法。

**public native void putObjectVolatile(Object obj, long offset, Object value);**
设置obj对象中offset偏移地址对应的object型field的值为指定值。支持volatile store语义。
与这个方法对应的get方法为：
public native Object getObjectVolatile(Object obj, long offset);
获取obj对象中offset偏移地址对应的object型field的值,支持volatile load语义
这两个方法还有与Int、Boolean、Byte、Short、Char、Long、Float、Double等类型对应的相关方法.

**public native void putObject(Object obj, long offset, Object value);**
设置obj对象中offset偏移地址对应的object型field的值为指定值。与putObject方法对应的是getObject方法。Int、Boolean、Byte、Short、Char、Long、Float、Double等类型都有getXXX和putXXX形式的方法。

下面通过一个组合示例来了解一下如何使用它们，详细如下：

```
public class UnsafePlayerCAS {
    public static void main(String[] args) throws Exception{
        //通过反射实例化Unsafe
        Field f = Unsafe.class.getDeclaredField("theUnsafe");
        f.setAccessible(true);
        Unsafe unsafe = (Unsafe) f.get(null);

        //实例化私有的构造函数
        Player player = (Player) unsafe.allocateInstance(Player.class);
        String name = "jack";
        int age = 19;
        player.setName(name);
        player.setAge(age);
        for (Field field : Player.class.getDeclaredFields()) {
            System.out.println(field.getName() + ":对应的内存偏移地址:"
                    + unsafe.objectFieldOffset(field));
        }
        //上面的输出为 name:对应的内存偏移地址:16
        //age:对应的内存偏移地址:12
        //修改内存偏移地址为12的值（age）,返回true,说明通过内存偏移地址修改age的值成功
        System.out.println(unsafe.compareAndSwapInt(player, 12, age, age + 1));
        System.out.println("age修改后的值:" + player.getAge());

        //修改内存偏移地址为12的值，但是修改后不保证立马能被其他的线程看到。
        unsafe.putOrderedInt(player, 12, age + 2);
        System.out.println("age修改后的值:" + player.getAge());

        //修改内存偏移地址为16的值，volatile修饰，修改能立马对其他线程可见
        unsafe.putObjectVolatile(player, 16, "tom");
        System.out.println("name:" + player.getName());
        System.out.println(unsafe.getObjectVolatile(player,16));
    }

    public static class Player{
        @Getter @Setter private String name;
        @Getter @Setter private int age;
        private Player(){}
    }
}
```

## 4. 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。整个并发框架中对线程的挂起操作被封装在 LockSupport类中，LockSupport类中有各种版本pack方法，但最终都调用了Unsafe.park()方法。

**public native void park(boolean isAbsolute, long timeout);**
阻塞一个线程直到unpark出现、线程被中断或者timeout时间到期。如果一个unpark调用已经出现了，这里只计数。timeout为0表示永不过期。当isAbsolute为true时，timeout是相对于新纪元之后的毫秒。否则这个值就是超时前的纳秒数。这个方法执行时也可能不合理地返回。

**public native void unpark(Thread thread);**
释放被park创建的在一个线程上的阻塞.这个方法也可以被使用来终止一个先前调用`park`导致的阻塞这个操作操作时不安全的,因此线程必须保证是活的.这是java代码不是native代码。参数thread指要解除阻塞的线程。

下面来看一下LockSupport类中关于Unsafe.park和Unsafe.unpark的使用：

```
private static void setBlocker(Thread t, Object arg) {
    // Even though volatile, hotspot doesn't need a write barrier here.
    UNSAFE.putObject(t, parkBlockerOffset, arg);
}
//恢复阻塞线程
public static void unpark(Thread thread) {
    if (thread != null)
        UNSAFE.unpark(thread);
}
//一直阻塞当前线程
public static void park(Object blocker) {
    Thread t = Thread.currentThread();
    setBlocker(t, blocker);
    UNSAFE.park(false, 0L);
    setBlocker(t, null);
}
//阻塞当前线程nanos纳秒
public static void parkNanos(Object blocker, long nanos) {
    if (nanos > 0) {
        Thread t = Thread.currentThread();
        setBlocker(t, blocker);
        UNSAFE.park(false, nanos);
        setBlocker(t, null);
    }
}

public static void parkUntil(Object blocker, long deadline) {
    Thread t = Thread.currentThread();
    setBlocker(t, blocker);
    UNSAFE.park(true, deadline);
    setBlocker(t, null);
}
//一直阻塞当前线程
public static void park() {
    UNSAFE.park(false, 0L);
}
//阻塞当前线程nanos纳秒
public static void parkNanos(long nanos) {
    if (nanos > 0)
        UNSAFE.park(false, nanos);
}

public static void parkUntil(long deadline) {
    UNSAFE.park(true, deadline);
}
```

下面是使用LockSupport的示例：

```
public class LockDemo {
    public static void main(String[] args) throws InterruptedException {
        ThreadPark threadPark = new ThreadPark();
        threadPark.start();
        ThreadUnpark threadUnPark = new ThreadUnpark(threadPark);
        threadUnPark.start();
        //等待threadUnPark执行成功
        threadUnPark.join();
        System.out.println("运行成功....");
    }

    static class ThreadPark extends Thread{
        @Override public void run(){
            System.out.println(Thread.currentThread() + "我将被阻塞在这了60s....");
            LockSupport.parkNanos(1000000000L * 60);
            System.out.println(Thread.currentThread() +"我被恢复正常了....");
        }
    }

    static class ThreadUnpark extends Thread{
        public Thread thread = null;

        public ThreadUnpark(Thread thread) {
            this.thread = thread;
        }

        public void run(){
            System.out.println("提前恢复阻塞线程ThreadPark");
            //恢复阻塞线程
            LockSupport.unpark(thread);
        }
    }
}
```

程序输出：

```
Thread[Thread-0,5,main]我将被阻塞在这了60s....
提前恢复阻塞线程ThreadPark
Thread[Thread-0,5,main]我被恢复正常了....
运行成功....
```

当然sun.misc.Unsafe中还有一些其它的功能，读者可以继续深挖。sun.misc.Unsafe提供了可以随意查看及修改JVM中运行时的数据结构，尽管这些功能在JAVA开发本身是不适用的，Unsafe是一个用于研究学习HotSpot虚拟机非常棒的工具，因为它不需要调用C++代码，或者需要创建即时分析的工具。

参考资料


1. https://blog.csdn.net/fenglibing/article/details/17138079
2. https://blog.csdn.net/dfdsggdgg/article/details/51538601
3. https://blog.csdn.net/aesop_wubo/article/details/7537278
4. https://blog.csdn.net/dfdsggdgg/article/details/51543545

欢迎支持笔者新作：《深入理解Kafka:核心设计与实践原理》和《RabbitMQ实战指南》，同时欢迎关注笔者的微信公众号：朱小厮的博客。