简述

JDK中rt.jar中的Unsafe类提供了硬件级别的原子性操作，Unsafe类中的方法都是native方法，它们使用JNI的方式访问本地C++实现库。

主要提供了以下功能：

1、通过Unsafe类可以分配内存，可以释放内存；
　　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

2、可以定位对象某字段的内存位置，也可以修改对象的字段值，即使它是私有的

3、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

4、CAS操作
　　是通过compareAndSwapXXX方法实现的

提供的一些方法

- long objectFieldOffset(Field field) 方法：
　　返回指定的变量在所属类的内存偏移地址，偏移地址仅仅在该 Unsafe 函数中访问指定字段时候使用。如下代码使用 unsafe 获取A
AtomicLong 对象中的内存偏移

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

- int arrayBaseOffset(Class arrayClass) 方法：获取数组中第一个元素的地址

- int arrayIndexScale(Class arrayClass) 方法：获取数组中单个元素占用的字节数

- boolean compareAndSwapLong(Object obj, long offset, long expect, long update) 方法：
　　比较对象 obj 中偏移量为 offset 的变量的值是不是和 expect 相等，相等则使用 update 值更新，然后返回 true，否者返回 false

- public native long getLongVolatile(Object obj, long offset) 方法：
　　获取对象 obj 中偏移量为 offset 的变量对应的 volatile 内存语义的值。

- void putLongVolatile(Object obj, long offset, long value) 方法：
　　设置 obj 对象中内存偏移为 offset 的 long 型变量的值为 value，支持 volatile 内存语义。

- void putOrderedLong(Object obj, long offset, long value) 方法：
　　设置 obj 对象中 offset 偏移地址对应的 long 型 field 的值为 value。这是有延迟的 putLongVolatile 方法，并不保证值修改对其它线
饰并且期望被意外修改的时候使用才有用。

- void park(boolean isAbsolute, long time)
　　阻塞当前线程，其中参数 isAbsolute 等于 false 时候，time 等于 0 表示一直阻塞，time 大于 0 表示等待指定的 time 后阻塞线程会
增量值，也就是相对当前时间累加 time 后当前线程就会被唤醒。

　　如果 isAbsolute 等于 true，并且 time 大于 0 表示阻塞后到指定的时间点后会被唤醒，这里 time 是个绝对的时间，是某一个时间点

另外当其它线程调用了当前阻塞线程的 interrupt 方法中断了当前线程时候，当前线程也会返回，当其它线程调用了 unpark 方法并会返回。

- void unpark(Object thread)：唤醒调用 park 后阻塞的线程，参数为需要唤醒的线程

下面是 Jdk8 新增的方法，这里简单的列出 Long 类型操作的方法

- long getAndSetLong(Object obj, long offset, long update) 方法
  获取对象 obj 中偏移量为 offset 的变量 volatile 语义的值，并设置变量 volatile 语义的值为 update

```
public final long getAndSetLong(Object obj, long offset, long update){
  long l;
    do{
      l = getLongVolatile(obj, offset);//(1)
    } while (!compareAndSwapLong(obj, offset, l, update));   {
       return l;
    }
  }
```

从代码可知内部代码 (1) 处使用 getLongVolatile 获取当前变量的值，然后使用 CAS 原子操作进行设置新值，这里使用 while 循环是考虑到要自旋重试

- long getAndAddLong(Object obj, long offset, long addValue) 方法
  获取对象 obj 中偏移量为 offset 的变量 volatile 语义的值，并设置变量值为原始值 +addValue

```
public final long getAndAddLong(Object obj, long offset, long addValue){
    long l;
    do{
      l = getLongVolatile(obj, offset);
    } while (!compareAndSwapLong(obj, offset, l, l + addValue));   {
         return l;
    }
  }
```

类似 getAndSetLong 的实现，只是这里使用CAS的时候使用了原始值+传递的增量参数 addValue 的值

Unsafe类的使用

```
public class TestUnSafe {
    //获取Unsafe的实例 (2.2.1)
    static final Unsafe unsafe = Unsafe.getUnsafe();

    //记录变量state在类TestUnSafe中的偏移值 (2.2.2)
    static final long stateOffset;

    //变量(2.2.3)
    private volatile long state = 0;

     static {
         try {
             //获取state变量在类TestUnSafe中的偏移值(2.2.4)
             stateOffset = unsafe.objectFieldOffset(TestUnSafe.class.getDeclaredField("state"));
         } catch (Exception ex) {
             System.out.println(ex.getLocalizedMessage());
```

```
            throw new Error(ex);
        }
    }

    public static void main(String[] args) {
        //创建实例，并且设置state值为1(2.2.5)
        TestUnSafe test = new TestUnSafe();
        //(2.2.6)
        Boolean sucess = unsafe.compareAndSwapInt(test, stateOffset, 0, 1);
        System.out.println(sucess);
        System.out.println(test.state);
    }
}
```

如上代码（2.2.1）获取了 Unsafe 的一个实例，代码（2.2.3）创建了一个变量 state 初始化为 0。

代码（2.2.4）使用 unsafe.objectFieldOffset 获取 TestUnSafe 类里面的 state 变量在 TestUnSafe 对象里面的内存偏移量地址并保存到 stat

代码（2.2.6）调用创建的 unsafe 实例的 compareAndSwapInt 方法，设置 test 对象的 state 变量的值，具体意思是如果 test 对象内存偏移 0，则更新该值为 1。

运行上面代码我们期望会输出 true，并且输出改变后的state为1。然而执行后会输出如下结果：

```
Exception in thread "main" java.lang.ExceptionInInitializerError
Caused by: java.lang.SecurityException: Unsafe
    at sun.misc.Unsafe.getUnsafe(Unsafe.java:90)
    at threadlocal.TestUnSafe.<clinit>(TestUnSafe.java:7)
```

既然异常信息在Unsafe.getUnsafe那么查看源码

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

代码判断了 Unsafe类如果是由Bootstrap类加载器加载的就返回，否则抛出异常。
这里很明显是由用户定义的，也就是用到默认AppClassLoader加载的

Unsafe 类是在 rt.jar 里面提供的，而 rt.jar 里面的类是使用 Bootstrap 类加载器加载的，而我们启动 main 函数所在的类是使用 AppClassLo

所以在 main 函数里面加载 Unsafe 类时候鉴于委托机制会委托给 Bootstrap 去加载 Unsafe 类。

如果没有上面这段代码，那么我们应用程序就可以随意使用 Unsafe 做事情了，而 Unsafe 类可以直接操作内存，是不安全的。

所以 JDK 开发组特意做了这个限制，不让开发人员在正规渠道下使用 Unsafe 类，而是在 rt.jar 里面的核心类里面使用 Unsafe 功能

通过反射使用Unsafe类

```
public class TestUnSafe {
    //获取Unsafe的实例（2.2.1）
    static Unsafe unsafe = null;

    //记录变量state在类TestUnSafe中的偏移值（2.2.2）
```

```
    static final long stateOffset;

    //变量(2.2.3)
    private volatile long state = 0;

     static {
         try {
             Field field = Unsafe.class.getDeclaredField("theUnsafe");

             field.setAccessible(true);

             unsafe = (Unsafe) field.get(null);

             //获取state变量在类TestUnSafe中的偏移值(2.2.4)
             stateOffset = unsafe.objectFieldOffset(TestUnSafe.class.getDeclaredField("state"));
         } catch (Exception ex) {
             System.out.println(ex.getLocalizedMessage());
             throw new Error(ex);
         }
     }

     public static void main(String[] args) {
         //创建实例，并且设置state值为1(2.2.5)
         TestUnSafe test = new TestUnSafe();
         //(2.2.6)
         Boolean sucess = unsafe.compareAndSwapInt(test, stateOffset, 0, 1);
         System.out.println(sucess);
         System.out.println(test.state);
     }
}
```