Java魔法类：Unsafe应用解析 (https://tech.meituan.com/2019/02/14/talk-about-java-magic-class-unsafe.html)

2019年02月14日 作者: 璐璐 文章链接 (https://tech.meituan.com/2019/02/14/talk-about-java-magic-class-unsafe.html) 11828字 24分钟阅读

前言

Unsafe是位于sun.misc包下的一个类，主要提供一些用于执行低级别、不安全操作的方法，如直接访问系统内存资源、自主管理内存资源等，这些方法在提升Java运行效率、增强Java语言底层资源操作能力方面起到了很大的作用。但由于Unsafe类使Java语言拥有了类似C语言指针一样操作内存空间的能力，这无疑也增加了程序发生相关指针问题的风险。在程序中过度、不正确使用Unsafe类会使得程序出错的概率变大，使得Java这种安全的语言变得不再“安全”，因此对Unsafe的使用一定要慎重。

注：本文对sun.misc.Unsafe公共API功能及相关应用场景进行介绍。

基本介绍

如下Unsafe源码所示，Unsafe类为一单例实现，提供静态方法getUnsafe获取Unsafe实例，当且仅当调用getUnsafe方法的类为引导类加载器所加载时才合法，否则抛出SecurityException异常。

```
public final class Unsafe {
  // 单例对象
  private static final Unsafe theUnsafe;

  private Unsafe() {
  }
  @CallerSensitive
  public static Unsafe getUnsafe() {
    Class var0 = Reflection.getCallerClass();
    // 仅在引导类加载器`BootstrapClassLoader`加载时才合法
    if(!VM.isSystemDomainLoader(var0.getClassLoader())) {
      throw new SecurityException("Unsafe");
    } else {
      return theUnsafe;
    }
  }
}
```

那如若想使用这个类，该如何获取其实例？有如下两个可行方案。

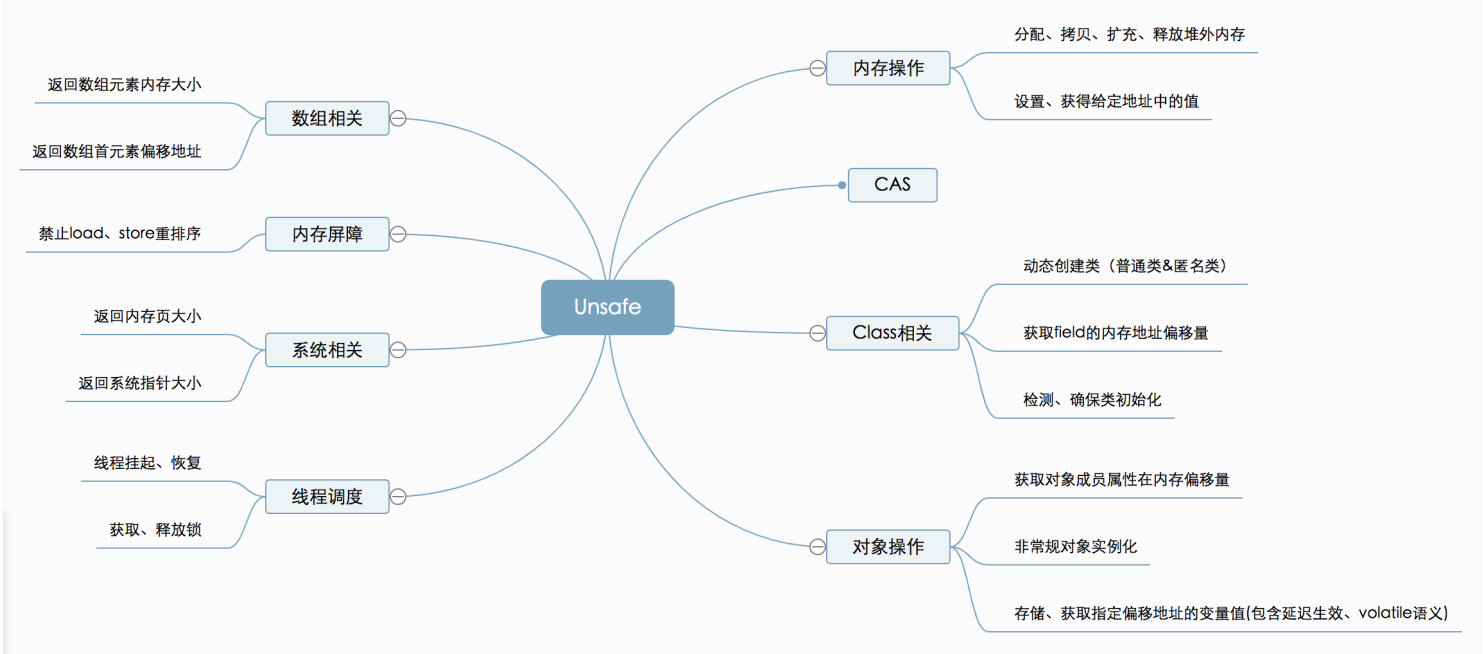
其一，从`getUnsafe`方法的使用限制条件出发，通过Java命令行命令`-Xbootclasspath/a`把调用Unsafe相关方法的类A所在jar包路径追加到默认的bootstrap路径中，使得A被引导类加载器加载，从而通过`Unsafe.getUnsafe`方法安全的获取Unsafe实例。

```
java -Xbootclasspath/a: ${path}   // 其中path为调用Unsafe相关方法的类所在jar包路径
```

其二，通过反射获取单例对象theUnsafe。

```
private static Unsafe reflectGetUnsafe() {
    try {
      Field field = Unsafe.class.getDeclaredField("theUnsafe");
      field.setAccessible(true);
      return (Unsafe) field.get(null);
    } catch (Exception e) {
      log.error(e.getMessage(), e);
      return null;
    }
}
```

功能介绍



如上图所示，Unsafe提供的API大致可分为内存操作、CAS、Class相关、对象操作、线程调度、系统信息获取、内存屏障、数组操作等几类，下面将对其相关方法和应用场景进行详细介绍。

内存操作

这部分主要包含堆外内存的分配、拷贝、释放、给定地址值操作等方法。

```
//分配内存，相当于C++的malloc函数
public native long allocateMemory(long bytes);
//扩充内存
public native long reallocateMemory(long address, 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);
//内存拷贝
public native void copyMemory(Object srcBase, long srcOffset, Object destBase, long destOffset, long bytes);
//获取给定地址值，忽略修饰限定符的访问限制。与此类似操作还有：getInt，getDouble，getLong，getChar等
public native Object getObject(Object o, long offset);
//为给定地址设置值，忽略修饰限定符的访问限制，与此类似操作还有：putInt,putDouble，putLong，putChar等
public native void putObject(Object o, long offset, Object x);
//获取给定地址的byte类型的值（当且仅当该内存地址为allocateMemory分配时，此方法结果为确定的）
public native byte getByte(long address);
//为给定地址设置byte类型的值（当且仅当该内存地址为allocateMemory分配时，此方法结果才是确定的）
public native void putByte(long address, byte x);
```

通常，我们在Java中创建的对象都处于堆内内存（heap）中，堆内内存是由JVM所管控的Java进程内存，并且它们遵循JVM的内存管理机制，JVM会采用垃圾回收机制统一管理堆内存。与之相对的是堆外内存，存在于JVM管控之外的内存区域，Java中对堆外内存的操作，依赖于Unsafe提供的操作堆外内存的native方法。

使用堆外内存的原因

- 对垃圾回收停顿的改善。由于堆外内存是直接受操作系统管理而不是JVM，所以当我们使用堆外内存时，即可保持较小的堆内内存规模。从而在GC时减少回收停顿对于应用的影响。
- 提升程序I/O操作的性能。通常在I/O通信过程中，会存在堆内内存到堆外内存的数据拷贝操作，对于需要频繁进行内存间数据拷贝且生命周期较短的暂存数据，都建议存储到堆外内存。

典型应用

DirectByteBuffer是Java用于实现堆外内存的一个重要类，通常用在通信过程中做缓冲池，如在Netty、MINA等NIO框架中应用广泛。DirectByteBuffer对于堆外内存的创建、使用、销毁等逻辑均由Unsafe提供的堆外内存API来实现。

下图为DirectByteBuffer构造函数，创建DirectByteBuffer的时候，通过Unsafe.allocateMemory分配内存、Unsafe.setMemory进行内存初始化，而后构建Cleaner对象用于跟踪DirectByteBuffer对象的垃圾回收，以实现当DirectByteBuffer被垃圾回收时，分配的堆外内存一起被释放。

```
//
DirectByteBuffer(int cap) {                   // package-private

    super( mark: -1,  pos: 0, cap, cap);
    boolean pa = VM.isDirectMemoryPageAligned();
    int ps = Bits.pageSize();
    long size = Math.max(1L, (long)cap + (pa ? ps : 0));
    Bits.reserveMemory(size, cap);

    long base = 0;
    try {
        base = unsafe.allocateMemory(size);
    } catch (OutOfMemoryError x) {
        Bits.unreserveMemory(size, cap);
        throw x;
    }
    unsafe.setMemory(base, size, (byte) 0);
    if (pa && (base % ps != 0)) {
        // Round up to page boundary
        address = base + ps - (base & (ps - 1));
    } else {
        address = base;
    }
    cleaner = Cleaner.create( o: this, new Deallocator(base, size, cap));
    att = null;
```

分配内存，并返回基地址

内存初始化

跟踪DirectByteBuffer对象的垃圾回收，以实现堆外内存释放

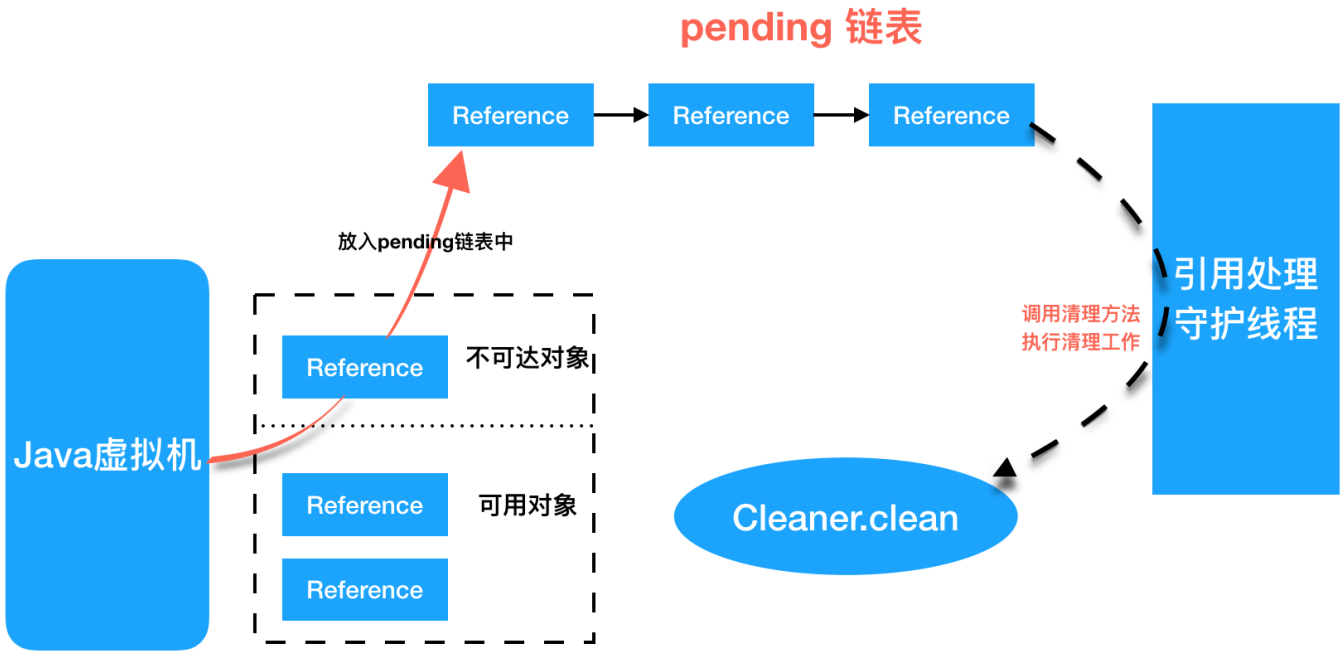
那么如何通过构建垃圾回收追踪对象Cleaner实现堆外内存释放呢?

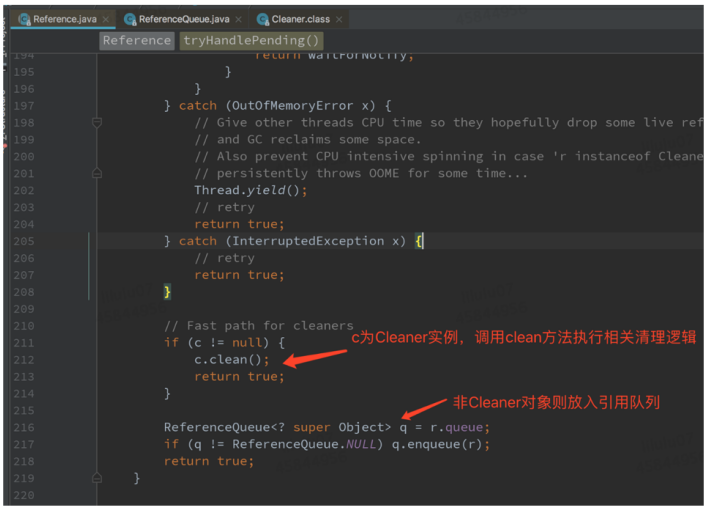
Cleaner继承自Java四大引用类型之一的虚引用PhantomReference（众所周知，无法通过虚引用获取与之关联的对象实例，且当对象仅被虚引用引用时，在任何发生GC的时候，其均可被回收），通常PhantomReference与引用队列ReferenceQueue结合使用，可以实现虚引用关联对象被垃圾回收时能够进行系统通知、资源清理等功能。如下图所示，当某个被Cleaner引用的对象将被回收时，JVM垃圾收集器会将此对象的引用放入到对象引用中的pending链表中，等待Reference-Handler进行相关处理。其中，Reference-Handler为一个拥有最高优先级的守护线程，会循环不断的处理pending链表中的对象引用，执行Cleaner的clean方法进行相关清理工作。

所以当DirectByteBuffer仅被Cleaner引用（即为虚引用）时，其可以在任意GC时段被回收。当DirectByteBuffer实例对象被回收时，在Reference-Handler线程操作中，会调用Cleaner的clean方法根据创建Cleaner时传入的Deallocator来进行堆外内存的释放。

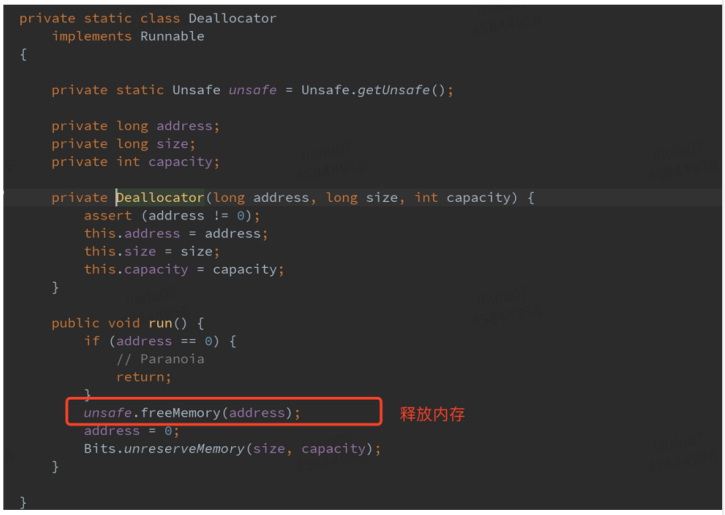
CAS相关

如下源代码释义所示，这部分主要为CAS相关操作的方法。





图一



图二

```
/**
         *  CAS
   * @param o           包含要修改field的对象
   * @param offset      对象中某field的偏移量
   * @param expected    期望值
   * @param update      更新值
   * @return            true | false
   */
public final native boolean compareAndSwapObject(Object o, long offset,  Object expected, Object update);

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

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

什么是CAS? 即比较并替换，实现并发算法时常用到的一种技术。CAS操作包含三个操作数——内存位置、预期原值及新值。执行CAS操作的时候，将内存位置的值与预期原值比较，如果相匹配，那么处理器会自动将该位置值更新为新值，否则，处理器不做任何操作。我们都知道，CAS是一条CPU的原子指令（cmpxchg指令），不会造成所谓的数据不一致问题，Unsafe提供的CAS方法（如compareAndSwapXXX）底层实现即为CPU指令cmpxchg。

典型应用

CAS在java.util.concurrent.atomic相关类、Java AQS、CurrentHashMap等实现上有非常广泛的应用。如下图所示，AtomicInteger的实现中，静态字段valueOffset即为字段value的内存偏移地址，valueOffset的值在AtomicInteger初始化时，在静态代码块中通过Unsafe的objectFieldOffset方法获取。在AtomicInteger中提供的线程安全方法中，通过字段valueOffset的值可以定位到AtomicInteger对象中value的内存地址，从而可以根据CAS实现对value字段的原子操作。

下图为某个AtomicInteger对象自增操作前后的内存示意图，对象的基地址baseAddress=“0x110000”，通过baseAddress+valueOffset得到value的内存地址valueAddress=“0x11000c”；然后通过CAS进行原子性的更新操作，成功则返回，否则继续重试，直到更新成功为止。

线程调度

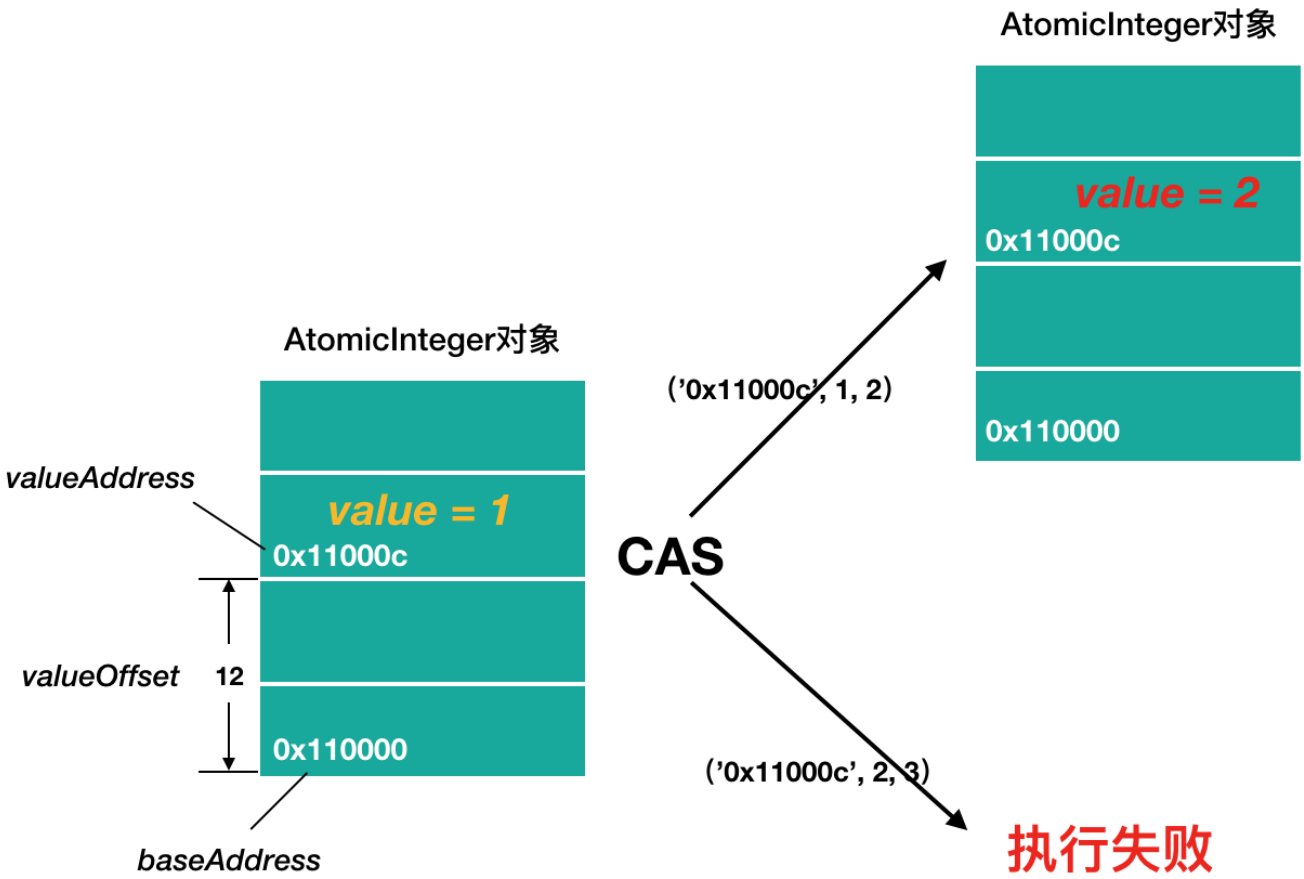
这部分，包括线程挂起、恢复、锁机制等方法。

```java
public class AtomicInteger extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 6214790243416807050L;

    // setup to use Unsafe.compareAndSwapInt for updates
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

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

    private volatile int value;
```




```java
//取消阻塞线程
public native void unpark(Object thread);
//阻塞线程
public native void park(boolean isAbsolute, long time);
//获得对象锁（可重入锁）
@Deprecated
public native void monitorEnter(Object o);
//释放对象锁
@Deprecated
public native void monitorExit(Object o);
//尝试获取对象锁
@Deprecated
public native boolean tryMonitorEnter(Object o);
```

如上源码说明中，方法park、unpark即可实现线程的挂起与恢复，将一个线程进行挂起是通过park方法实现的，调用park方法后，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。

典型应用

Java锁和同步器框架的核心类AbstractQueuedSynchronizer，就是通过调用 `LockSupport.park()` 和 `LockSupport.unpark()` 实现线程的阻塞和唤醒的，而LockSupport的park、unpark方法实际是调用Unsafe的park、unpark方式来实现。

Class相关

此部分主要提供Class和它的静态字段的操作相关方法，包含静态字段内存定位、定义类、定义匿名类、检验&确保初始化等。

```
//获取给定静态字段的内存地址偏移量，这个值对于给定的字段是唯一且固定不变的
public native long staticFieldOffset(Field f);
//获取一个静态类中给定字段的对象指针
public native Object staticFieldBase(Field f);
//判断是否需要初始化一个类，通常在获取一个类的静态属性的时候（因为一个类如果没初始化，它的静态属性也不会初始化）使用。 当且仅当ensureClassInitialized方法不生效时返回false。
public native boolean shouldBeInitialized(Class<?> c);
//检测给定的类是否已经初始化。通常在获取一个类的静态属性的时候（因为一个类如果没初始化，它的静态属性也不会初始化）使用。
public native void ensureClassInitialized(Class<?> c);
//定义一个类，此方法会跳过JVM的所有安全检查，默认情况下，ClassLoader（类加载器）和ProtectionDomain（保护域）实例来源于调用者
public native Class<?> defineClass(String name, byte[] b, int off, int len, ClassLoader loader, ProtectionDomain protectionDomain);
//定义一个匿名类
public native Class<?> defineAnonymousClass(Class<?> hostClass, byte[] data, Object[] cpPatches);
```

典型应用

从Java 8开始，JDK使用invokedynamic及VM Anonymous Class结合来实现Java语言层面上的Lambda表达式。

- **invokedynamic**：invokedynamic是Java 7为了实现在JVM上运行动态语言而引入的一条新的虚拟机指令，它可以实现在运行期动态解析出调用点限定符所引用的方法，然后再执行该方法，invokedynamic指令的分派逻辑是由用户设定的引导方法决定。
- **VM Anonymous Class**：可以看做是一种模板机制，针对于程序动态生成很多结构相同、仅若干常量不同的类时，可以先创建包含常量占位符的模板类，而后通过Unsafe.defineAnonymousClass方法定义具体类时填充模板的占位符生成具体的匿名类。生成的匿名类不显式挂在任何ClassLoader下面，只要当该类没有存在的实例对象、且没有强引用来引用该类的Class对象时，该类就会被GC回收。故而VM Anonymous Class相比于Java语言层面的匿名内部类无需通过ClassClassLoader进行类加载且更易回收。

在Lambda表达式实现中，通过invokedynamic指令调用引导方法生成调用点，在此过程中，会通过ASM动态生成字节码，而后利用Unsafe的defineAnonymousClass方法定义实现相应的函数式接口的匿名类，然后再实例化此匿名类，并返回与此匿名类中函数式方法的方法句柄关联的调用点；而后可以通过此调用点实现调用相应Lambda表达式定义逻辑的功能。下面以如下图所示的Test类来举例说明。

```
import java.util.function.Consumer;


public class Test {

    public static void main(String[] args) throws Exception {
        Consumer<String> consumer = s -> System.out.println(s);
        consumer.accept( t: "lambda");
    }
}
```

Test类编译后的class文件反编译后的结果如下图一所示（删除了对本文说明无意义的部分），我们可以从中看到main方法的指令实现、invokedynamic指令调用的引导方法BootstrapMethods、及静态方法 `lambda$main$0` （实现了Lambda表达式中字符串打印逻辑）等。在引导方法执行过程中，会通过Unsafe.defineAnonymousClass生成如下图二所示的实现Consumer接口的匿名类。其中，accept方法通过调用Test类中的静态方法 `lambda$main$0` 来实现Lambda表达式中定义的逻辑。而后执行语句 `consumer.accept（"lambda"）` 其实就是调用下图二所示的匿名类的accept方法。

```
Compiled from "Test.java"
public class com.sankuai.meituan.trippackage.api.Test {

  public static void main(java.lang.String[]) throws java.lang.Exception;
    Code:
       0: invokedynamic #2,  0       // InvokeDynamic #0:accept:()Ljava/util/function/Consumer;
       5: astore_1
       6: aload_1
       7: ldc           #3         // String lambda
       9: invokeinterface #4,  2    // InterfaceMethod java/util/function/Consumer.accept:(Ljava/lang/Object;)V
      14: return

  private static void lambda$main$0(java.lang.String);
    Code:
       0: getstatic     #5         // Field java/lang/System.out:Ljava/io/PrintStream;
       3: aload_0
       4: invokevirtual #6         // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       7: return

  SourceFile: "Test.java"
  InnerClasses:
       public static final #75= #74 of #78; //Lookup=class java/lang/invoke/MethodHandles$Lookup
                of class java/lang/invoke/MethodHandles
  BootstrapMethods:
    0: #38 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(Ljava/lang/invoke/MethodHandles$Lookup;
           Ljava/lang/String;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;
           Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;
      Method arguments:
        #39 (Ljava/lang/Object;)V
        #40 invokestatic com/sankuai/meituan/trippackage/api/Test.lambda$main$0:(Ljava/lang/String;)V
        #41 (Ljava/lang/String;)V
}
```

引导方法

图一

```
import java.lang.invoke.LambdaForm.Hidden;
import java.util.function.Consumer;

// $FF: synthetic class
final class Test$$Lambda$1 implements Consumer {
    private Test$$Lambda$1() {
    }

    @Hidden
    public void accept(Object var1) {
        Test.lambda$main$0((String)var1);
    }
}
```

图二

对象操作

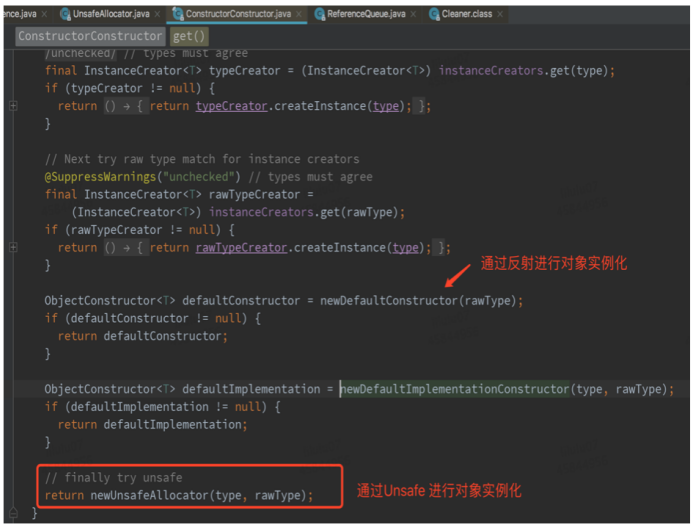
此部分主要包含对象成员属性相关操作及非常规的对象实例化方式等相关方法。

```
//返回对象成员属性在内存地址相对于此对象的内存地址的偏移量
public native long objectFieldOffset(Field f);
//获得给定对象的指定地址偏移量的值，与此类似操作还有：getInt，getDouble，getLong，getChar等
public native Object getObject(Object o, long offset);
//给定对象的指定地址偏移量设值，与此类似操作还有：putInt，putDouble，putLong，putChar等
public native void putObject(Object o, long offset, Object x);
//从对象的指定偏移量处获取变量的引用，使用volatile的加载语义
public native Object getObjectVolatile(Object o, long offset);
//存储变量的引用到对象的指定的偏移量处，使用volatile的存储语义
public native void putObjectVolatile(Object o, long offset, Object x);
//有序、延迟版本的putObjectVolatile方法，不保证值的改变被其他线程立即看到。只有在field被volatile修饰符修饰时有效
public native void putOrderedObject(Object o, long offset, Object x);
//绕过构造方法、初始化代码来创建对象
public native Object allocateInstance(Class<?> cls) throws InstantiationException;
```

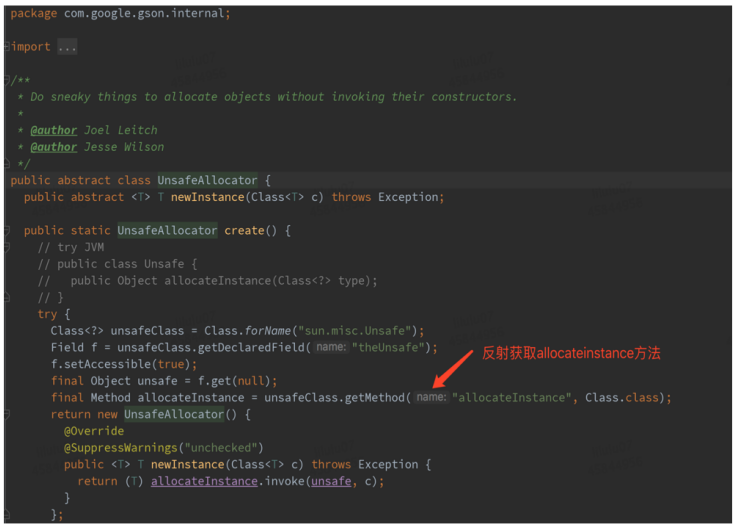
典型应用

- **常规对象实例化方式**：我们通常所用到的创建对象的方式，从本质上来讲，都是通过new机制来实现对象的创建。但是，new机制有个特点就是当类只提供有参的构造函数且无显示声明无参构造函数时，则必须使用有参构造函数进行对象构造，而使用有参构造函数时，必须传递相应个数的参数才能完成对象实例化。
- **非常规的实例化方式**：而Unsafe中提供allocateInstance方法，仅通过Class对象就可以创建此类的实例对象，而且不需要调用其构造函数、初始化代码、JVM安全检查等。它抑制修饰符检测，也就是即使构造器是private修饰的也能通过此方法实例化，只需提类对象即可创建相应的对象。由于这种特性，allocateInstance在java.lang.invoke、Objenesis（提供绕过类构造器的对象生成方式）、Gson（反序列化时用到）中都有相应的应用。

如下图所示，在Gson反序列化时，如果类有默认构造函数，则通过反射调用默认构造函数创建实例，否则通过UnsafeAllocator来实现对象实例的构造，UnsafeAllocator通过调用Unsafe的allocateInstance实现对象的实例化，保证在目标类无默认构造函数时，反序列化不够影响。



图一



图二

数组相关

这部分主要介绍与数据操作相关的arrayBaseOffset与arrayIndexScale这两个方法，两者配合起来使用，即可定位数组中每个元素在内存中的位置。

```
//返回数组中第一个元素的偏移地址
public native int arrayBaseOffset(Class<?> arrayClass);
//返回数组中一个元素占用的大小
public native int arrayIndexScale(Class<?> arrayClass);
```

典型应用

这两个与数据操作相关的方法，在java.util.concurrent.atomic 包下的AtomicIntegerArray（可以实现对Integer数组中每个元素的原子性操作）中有典型的应用，如下图AtomicIntegerArray源码所示，通过Unsafe的arrayBaseOffset、arrayIndexScale分别获取数组首元素的偏移地址base及单个元素大小因子scale。后续相关原子性操作，均依赖于这两个值进行数组中元素的定位，如下图二所示的getAndAdd方法即通过checkedByteOffset方法获取某数组元素的偏移地址，而后通过CAS实现原子性操作。

内存屏障

在Java 8中引入，用于定义内存屏障（也称内存栅栏，内存栅障，屏障指令等，是一类同步屏障指令，是CPU或编译器在对内存随机访问的操作中的一个同步点，使得此点之前的所有读写操作都执行后才可以开始执行此点之后的操作），避免代码重排序。

```
//内存屏障，禁止load操作重排序。屏障前的load操作不能被重排序到屏障后，屏障后的load操作不能被重排序到屏障前
public native void loadFence();
//内存屏障，禁止store操作重排序。屏障前的store操作不能被重排序到屏障后，屏障后的store操作不能被重排序到屏障前
public native void storeFence();
//内存屏障，禁止load、store操作重排序
public native void fullFence();
```

典型应用

在Java 8中引入了一种锁的新机制——StampedLock，它可以看成是读写锁的一个改进版本。StampedLock提供了一种乐观读锁的实现，这种乐观读锁类似于无锁的操作，完全不会阻塞写线程获取写锁，从而缓解读多写少时写线程“饥饿”现象。由于StampedLock提供的乐观读锁不阻塞写线程获取读锁，当线程共享变量从主内存load到线程工作内存时，会存在数据不一致问题，所以当使用StampedLock的乐观读锁时，需要遵从如下图用例中使用的模式来确保数据的一致性。

如上图用例所示计算坐标点Point对象，包含点移动方法move及计算此点到原点的距离的方法distanceFromOrigin。在方法distanceFromOrigin中，首先，通过tryOptimisticRead方法获取乐观读标记；然后从主内存中加载点的坐标值 (x,y)；而后通过StampedLock的validate方法校验锁状态，判断坐标点(x,y)从主内存加载到线程工作内存过程中，主内存的值是否已被其他线程通过move方法修改，如果validate返回值为true，证明(x, y)的值未被修改，可参与后续计算；否则，需加悲观读锁，再次从主内存加载(x,y)的最新值，然后再进行距离计算。其中，校验锁状态这步操作至关重要，需要判断锁状态是否发生改变，从而判断之前copy到线程工作内存中的值是否与主内存的值存在不一致。

下图为StampedLock.validate方法的源码实现，通过锁标记与相关常量进行位运算、比较来校验锁状态，在校验逻辑之前，会通过Unsafe的loadFence方法加入一个load内存屏障，目的

```java
public class AtomicIntegerArray implements java.io.Serializable {
    private static final long serialVersionUID = 2862133569453604235L;

    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final int base = unsafe.arrayBaseOffset(int[].class);
    private static final int shift;
    private final int[] array;

    static {
        int scale = unsafe.arrayIndexScale(int[].class);
        if ((scale & (scale - 1)) != 0)
            throw new Error("data type scale not a power of two");
        shift = 31 - Integer.numberOfLeadingZeros(scale);
    }

    private long checkedByteOffset(int i) {
        if (i < 0 || i >= array.length)
            throw new IndexOutOfBoundsException("index " + i);

        return byteOffset(i);
    }

    private static long byteOffset(int i) { return ((long) i << shift) + base; }
```

获取数组元素的首地址

获取每个元素所占大小

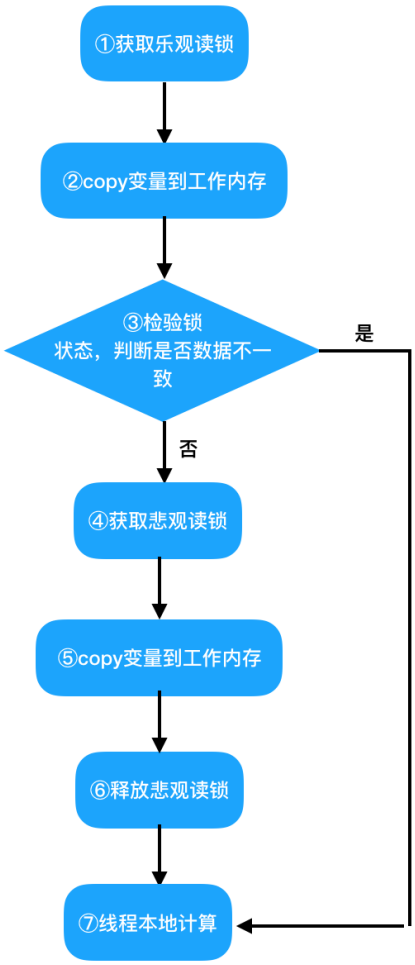
通过数据元素的位置计算偏移地址

图一

```java
/**
 * Atomically decrements by one the element at index {@code i}.
 *
 * @param i the index
 * @return the previous value
 */
public final int getAndDecrement(int i) {
    return getAndAdd(i, delta: -1);
}

/**
 * Atomically adds the given value to the element at index {@code i}.
 *
 * @param i the index
 * @param delta the value to add
 * @return the previous value
 */
public final int getAndAdd(int i, int delta) {
    return unsafe.getAndAddInt(array, checkedByteOffset(i), delta);
}
```

图二




```java
class Point {

    private double x, y;

    private final StampedLock sl = new StampedLock();

    void move(double deltaX, double deltaY) {
        long stamp = sl.writeLock();    // 使用写锁-独占操作
        try {
            x += deltaX;
            y += deltaY;
        } finally {
            sl.unlockWrite(stamp);
        }
    }
    double distanceFromOrigin() {
        long stamp = sl.tryOptimisticRead();         // ①
        double currentX = x, currentY = y;           // ②
        if (!sl.validate(stamp)) {                   // ③
            stamp = sl.readLock();                   // ④
            try {
                currentX = x;                        // ⑤
                currentY = y;
            } finally {
                sl.unlockRead(stamp);                // ⑥
            }
        }
        return Math.sqrt(currentX * currentX + currentY * currentY); // ⑦
    }
}
```

是避免上图用例中步骤②和StampedLock.validate中锁状态校验运算发生重排序导致锁状态校验不准确的问题。

系统相关

这部分包含两个获取系统相关信息的方法。

```java
//返回系统指针的大小。返回值为4（32位系统）或 8（64位系统）。
public native int addressSize();
//内存页的大小，此值为2的幂次方。
public native int pageSize();
```

典型应用

如下图所示的代码片段，为java.nio下的工具类Bits中计算待申请内存所需内存页数量的静态方法，其依赖于Unsafe中pageSize方法获取系统内存页大小实现后续计算逻辑。

结语

本文对Java中的sun.misc.Unsafe的用法及应用场景进行了基本介绍，我们可以看到Unsafe提供了很多便捷、有趣的API方法。即便如此，由于Unsafe中包含大量自主操作内存的方法，如若使用不当，会对程序带来许多不可控的灾难。因此对它的使用我们需要慎之又慎。

参考资料

```java
/**
 * Returns true if the lock has not been exclusively acquired
 * since issuance of the given stamp. Always returns false if the
 * stamp is zero. Always returns true if the stamp represents a
 * currently held lock. Invoking this method with a value not
 * obtained from {@link #tryOptimisticRead} or a locking method
 * for this lock has no defined effect or result.
 *
 * @param stamp a stamp
 * @return {@code true} if the lock has not been exclusively acquired
 * since issuance of the given stamp; else false
 */
public boolean validate(long stamp) {
    U.loadFence();
    return (stamp & SBITS) == (state & SBITS);
}

/**
 * If the lock state matches the given stamp, releases the
```

load 内存屏障

```java
    private static int pageSize = -1;

    static int pageSize() {
        if (pageSize == -1)
            pageSize = unsafe().pageSize();
        return pageSize;
    }

    static int pageCount(long size) {
        return (int)(size + (long)pageSize() - 1L) / pageSize();
    }
```

- OpenJDK Unsafe source (http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/9b8c96f96a0f/src/share/classes/sun/misc/Unsafe.java)
- Java Magic. Part 4: sun.misc.Unsafe (http://mishadoff.com/blog/java-magic-part-4-sun-dot-misc-dot-unsafe)
- JVM crashes at libjvm.so (https://www.zhihu.com/question/51132462)
- Java中神奇的双刃剑–Unsafe (https://www.cnblogs.com/throwable/p/9139947.html)
- JVM源码分析之堆外内存完全解读 (http://lovestblog.cn/blog/2015/05/12/direct-buffer/)
- 堆外内存 之 DirectByteBuffer 详解 (https://www.jianshu.com/p/007052ee3773)
- 《深入理解Java虚拟机（第2版）》

作者简介

- 璐璐，美团点评Java开发工程师。2017年加入美团点评，负责美团点评境内度假的后端开发。