臭名昭著的sun.misc.Unsafe解释

翻译

dnc8371 于 2020-05-19 01:33:03 发布 1484 收藏 1
文章标签： jvm java python jdk 安卓

Java虚拟机的最大竞争对手可能是托管C＃等语言的Microsoft CLR 。 CLR允许编写不安全的代码作为低级编程的入口，这在JVM上很难实现。 如果您需要Java中的此类高级功能，则可能会被迫使用JNI，这需要您了解一些C并Swift导致代码紧密耦合到特定平台。 使用sun.misc.Unsafe ，尽管不鼓励使用Java API在Java plarform上进行低级编程，但还有另一种选择。 尽管如此，仍有一些应用程序依赖于sun.misc.Unsafe （例如objenesis）以及所有基于后者的库（例如kryo），例如kryo，该库再次用于Twitter的Storm中 。 因此，现在该看看一下了，特别是因为sun.misc.Unsafe的功能被认为已成为Java 9中Java公共API的一部分 。

取得sun.misc.Unsafe实例

sun.misc.Unsafe类旨在仅由核心Java类使用，因此其作者将其唯一的构造函数设为私有，并且仅添加了一个同样私有的singleton实例。 此实例的公共获取程序将执行安全检查，以避免其公共使用：

```
public static Unsafe getUnsafe() {
  Class cc = sun.reflect.Reflection.getCallerClass(2);
  if (cc.getClassLoader() != null)
    throw new SecurityException("Unsafe");
  return theUnsafe;
}
```

此方法首先从当前线程的方法堆栈中查找调用的Class 。 该查找由另一个名为sun.reflection.Reflection内部类实现，该内部类基本上是浏览给定数量的调用堆栈帧，然后返回此方法的定义类。 但是，此安全检查可能会在将来的版本中更改 。 浏览堆栈时，第一个找到的类（索引0 ）显然是Reflection类本身，第二个（索引1 ）类将是Unsafe类，这样索引2将保存正在调用Unsafe#getUnsafe() 。

然后检查此查找的类的ClassLoader ，其中使用null引用表示HotSpot虚拟机上的引导程序类加载器。 （这在Class#getClassLoader()中有说明，其中说“ *某些实现可能使用null来表示引导类加载器* ”。）由于通常不会在该类加载器中加载非核心Java类，因此您永远不会能够直接调用此方法，但收到抛出的SecurityException作为答案。 （从技术上讲，您可以通过将引导程序类加载器添加到-Xbootclasspath来强制VM使用引导程序类加载器来加载应用程序类，但这需要在应用程序代码之外进行一些设置，您可能希望避免这种设置。）因此，进行以下测试将会成功：

```
@Test(expected = SecurityException.class)
public void testSingletonGetter() throws Exception {
  Unsafe.getUnsafe();
}
```

但是，安全检查的设计不当，应视为对单例反模式的警告。 只要不禁止使用反射 （这很困难，因为在许多框架中广泛使用反射 ），您总是可以通过检查类的私有成员来获得实例。 从Unsafe类的源代码中，您可以了解到单例实例存储在名为theUnsafe的私有静态字段中。 对于HotSpot虚拟机至少是这样。 对于我们来说不幸的是，其他虚拟机实现有时对此字段使用其他名称。 例如，Android的Unsafe类将其单例实例存储在名为THE_ONE的字段中。 这使得很难提供一种“兼容”的方式来接收实例。 但是，由于我们已经通过使用Unsafe类离开了兼容性的保存范围，因此我们不必为此担心，而应该比完全使用该类还要担心。 为了掌握单例实例，您只需读取单例字段的值即可：

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

或者，您可以调用私人教练。 我个人更喜欢这种方式，因为它可以与Android一起使用，而提取字段却不能：

```
Constructor<Unsafe> unsafeConstructor = Unsafe.class.getDeclaredConstructor();
unsafeConstructor.setAccessible(true);
Unsafe unsafe = unsafeConstructor.newInstance();
```

您为这种次要的兼容性优势所付出的代价是最小的堆空间。 但是，在字段或构造函数上使用反射时执行的安全性检查类似。

创建类的实例而不调用构造函数

我第一次使用Unsafe类是为了创建类的实例而不调用任何类的构造函数。 我需要代理整个类，该类只具有一个嘈杂的构造函数，但我只想将所有方法调用委托给一个实际实例，但是在构造时我还不知道。 创建子类很容易，如果该类已由接口表示，则创建代理将是一件简单的事情。 但是，对于昂贵的构造函数，我陷入了困境。 通过使用Unsafe类，我得以解决该问题。 考虑一个带有虚假的构造函数的类：

```
class ClassWithExpensiveConstructor {

  private final int value;

  private ClassWithExpensiveConstructor() {
    value = doExpensiveLookup();
  }

  private int doExpensiveLookup() {
    try {
      Thread.sleep(2000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    return 1;
  }

  public int getValue() {
    return value;
  }
}
```

使用Unsafe，我们可以创建ClassWithExpensiveConstructor （或其任何子类）的实例，而不必调用上述构造函数，只需直接在堆上分配实例即可：

```
@Test
public void testObjectCreation() throws Exception {
  ClassWithExpensiveConstructor instance = (ClassWithExpensiveConstructor)
  unsafe.allocateInstance(ClassWithExpensiveConstructor.class);
  assertEquals(0, instance.getValue());
}
```

请注意，final字段尚未由构造方法初始化，但使用其类型的默认值进行设置 。 除此之外，构造的实例的行为类似于普通的Java对象。 例如，当它变得不可访问时，将被垃圾回收。

Java运行时本身在创建对象（例如反序列化）时无需调用构造函数即可创建对象。 因此，`ReflectionFactory`提供了更多访问单个对象的权限：

```
@Test
public void testReflectionFactory() throws Exception {
  @SuppressWarnings("unchecked")
  Constructor<ClassWithExpensiveConstructor> silentConstructor = ReflectionFactory.getReflectionFactory()
      .newConstructorForSerialization(ClassWithExpensiveConstructor.class, Object.class.getConstructor());
  silentConstructor.setAccessible(true);
  assertEquals(10, silentConstructor.newInstance().getValue());
}
```

请注意， `ReflectionFactory`类只需要一个`RuntimePermission`呼吁`reflectionFactoryAccess`接收的单一实例，因此没有反映这里需要。 收到的`ReflectionFactory`实例允许您定义任何构造函数以*成为*给定类型的构造函数。 在上面的示例中，我为此使用了默认的`java.lang.Object`构造函数。 但是，您可以使用任何构造函数：

```
class OtherClass {

  private final int value;
  private final int unknownValue;

  private OtherClass() {
    System.out.println("test");
    this.value = 10;
    this.unknownValue = 20;
  }
}

@Test
public void testStrangeReflectionFactory() throws Exception {
  @SuppressWarnings("unchecked")
  Constructor<ClassWithExpensiveConstructor> silentConstructor = ReflectionFactory.getReflectionFactory()
      .newConstructorForSerialization(ClassWithExpensiveConstructor.class,
            OtherClass.class.getDeclaredConstructor());
  silentConstructor.setAccessible(true);
  ClassWithExpensiveConstructor instance = silentConstructor.newInstance();
  assertEquals(10, instance.getValue());
  assertEquals(ClassWithExpensiveConstructor.class, instance.getClass());
  assertEquals(Object.class, instance.getClass().getSuperclass());
}
```

请注意，即使调用了完全不同的类的构造函数，也已在此构造函数中设置了value 。 然而，目标类中不存在的字段也会被忽略，从上面的示例中也可以明显看出。 请注意， OtherClass不会成为构造的实例类型层次结构的一部分，只需为“序列化”类型借用OtherClass的构造函数。

此博客条目中未提及的是其他方法，例如Unsafe#defineClass ， Unsafe#defineAnonymousClass或Unsafe#ensureClassInitialized 。 但是，公共API的ClassLoader也定义了类似的功能。

本机内存分配

您是否曾经想过用Java分配一个数组，该数组应该具有多个Integer.MAX_VALUE条目？ 可能不是因为这不是一项常见任务，但是如果您曾经需要此功能，则可以实现。 您可以通过分配**本机内存**来创建这样的数组。 本机内存分配例如由Java的NIO包中提供的直接字节缓冲区使用。 除了堆内存之外，本机内存不是堆区域的一部分，可以非排他性地用于例如与其他进程进行通信。 结果，Java的堆空间与本机空间竞争：分配给JVM的内存越多，剩余的本机内存就越少。

让我们看一个示例，该示例在Java中使用本地（堆外）内存创建上述超大数组：

```
class DirectIntArray {

  private final static long INT_SIZE_IN_BYTES = 4;

  private final long startIndex;

  public DirectIntArray(long size) {
    startIndex = unsafe.allocateMemory(size * INT_SIZE_IN_BYTES);
    unsafe.setMemory(startIndex, size * INT_SIZE_IN_BYTES, (byte) 0);
    }
  }

  public void setValue(long index, int value) {
    unsafe.putInt(index(index), value);
  }

  public int getValue(long index) {
    return unsafe.getInt(index(index));
  }

  private long index(long offset) {
    return startIndex + offset * INT_SIZE_IN_BYTES;
  }

  public void destroy() {
    unsafe.freeMemory(startIndex);
  }
}

@Test
public void testDirectIntArray() throws Exception {
  long maximum = Integer.MAX_VALUE + 1L;
  DirectIntArray directIntArray = new DirectIntArray(maximum);
  directIntArray.setValue(0L, 10);
  directIntArray.setValue(maximum, 20);
  assertEquals(10, directIntArray.getValue(0L));
  assertEquals(20, directIntArray.getValue(maximum));
  directIntArray.destroy();
}
```

首先，请确保您的计算机有足够的内存来运行此示例！ 您至少需要`(2147483647 + 1) * 4 byte = 8192 MB`本机内存才能运行代码。 如果您使用过其他编程语言（例如C），则每天都要进行直接内存分配。 通过调用`Unsafe#allocateMemory(long)`，虚拟机将为您分配请求的本机内存量。 之后，您有责任正确处理此内存。

存储特定值所需的内存量取决于类型的大小。 在上面的示例中，我使用了一个`int`类型，该类型表示32位整数。 因此，单个`int`值消耗4个字节。 对于基本类型， 大小有据可查 。 但是，计算对象类型的大小更为复杂，因为它们取决于类型层次结构中任何地方声明的非静态字段的数量。 计算对象大小的最典型方法是使用Java的Attach API中的`Instrumented`类，该类为此目的提供了一种专用方法，称为`getObjectSize` 。 但是，在本节的最后，我将评估处理对象的另一种（hacky）方式。

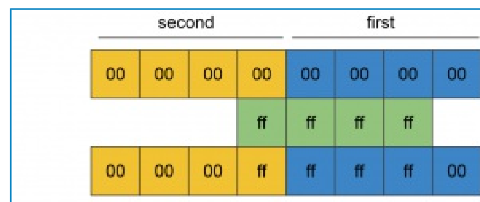
请注意，直接分配的内存始终是本机内存，因此不会进行垃圾回收。 因此，如上例所示，您必须通过调用`Unsafe#freeMemory(long)`显式释放内存。 否则，您将保留一些内存，只要JVM实例正在运行，就无法将其用于其他用途，这是内存泄漏和非垃圾收集语言中的常见问题。 或者，您也可以通过调用`Unsafe#reallocateMemory(long, long)`直接在某个地址重新分配内存，其中第二个参数描述了JVM在给定地址保留的新字节数。

另外，请注意，直接分配的内存未使用特定值初始化。 通常，您会从该内存区域的旧用法中发现垃圾，因此如果需要默认值，则必须显式初始化分配的内存。 当您让Java运行时为您分配内存时，通常会为您完成此操作。 在上面的示例中，借助于`Unsafe#setMemory`方法，整个区域被零覆盖。

使用直接分配的内存时，JVM都不会为您执行范围检查。 因此，如以下示例所示，可能会破坏您的内存：

```
@Test
public void testMallaciousAllocation() throws Exception {
  long address = unsafe.allocateMemory(2L * 4);
  unsafe.setMemory(address, 8L, (byte) 0);
  assertEquals(0, unsafe.getInt(address));
  assertEquals(0, unsafe.getInt(address + 4));
  unsafe.putInt(address + 1, 0xffffffff);
  assertEquals(0xffffff00, unsafe.getInt(address));
  assertEquals(0x000000ff, unsafe.getInt(address + 4));
}
```

请注意，我们在空间中写入了一个值，该值分别部分保留给第一个和第二个数字。 这张照片可能会清除一切。 请注意，内存中的值是从“右向左”运行的（但这可能取决于计算机）。



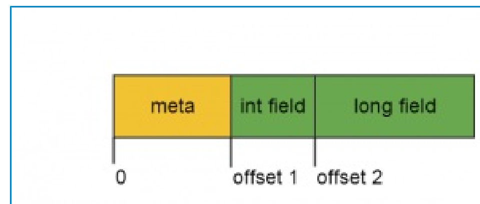
第一行显示将零写入整个分配的本机内存区域后的初始状态。 然后，我们使用32个字节覆盖4个字节，并以单个字节的偏移量覆盖。 最后一行显示此写入操作后的结果。

最后，我们想将整个对象写入本地内存。 如上所述，这是一项艰巨的任务，因为我们首先需要计算对象的大小才能知道我们需要保留的大小。 但是，Unsafe类不提供此类功能。 至少不是直接地，因为我们至少可以使用Unsafe类来查找实例字段的偏移量，JVM自身在堆上分配对象时会使用该实例字段的偏移量。 这使我们能够找到对象的近似大小：

```
public long sizeOf(Class<?> clazz)
  long maximumOffset = 0;
  do {
    for (Field f : clazz.getDeclaredFields()) {
      if (!Modifier.isStatic(f.getModifiers())) {
        maximumOffset = Math.max(maximumOffset, unsafe.objectFieldOffset(f));
      }
    }
  } while ((clazz = clazz.getSuperclass()) != null);
  return maximumOffset + 8;
}
```

乍一看，这似乎很神秘，但是此代码背后没有什么大秘密。 我们只是简单地遍历在类本身或其任何超类中声明的所有非静态字段。 我们不必担心接口，因为它们无法定义字段，因此永远不会更改对象的内存布局。 这些字段中的任何一个都有一个偏移量，该偏移量表示当JVM将此类实例存储在内存中时，该字段的值相对于用于该对象的第一个字节占用该字段的第一个字节。 我们只需找到最大偏移量即可找到除最后一个字段以外的所有字段所需的空间。 由于在64位计算机上运行时，字段对于`long`值或`double`值或对象引用的占用空间永远不会超过64位（8字节），因此我们至少找到了用于存储空间的上限。宾语。 因此，我们只需将这8个字节添加到最大索引中，就不会遇到保留很小空间的危险。 这个想法当然会浪费一些字节，并且应该在生产代码中使用更好的算法。

在这种情况下，最好将类定义视为异构数组的一种形式。 请注意，最小字段偏移量不是`0`而是正值。 前几个字节包含元信息。 下图通过一个`int`和一个`long`字段（其中两个字段都有偏移量）的示例对象形象化了该原理。 请注意，在将对象的副本写入本机内存时，我们通常不会写入元信息，因此我们可以进一步减少使用的本机便笺的数量。 还要注意，此内存布局可能高度依赖于Java虚拟机的实现。



通过这种过度仔细的估计，我们现在可以实现一些存根方法，将对象的浅表副本直接写入本机内存。 请注意，本机内存并不真正了解对象的概念。 我们基本上只是将给定的字节数设置为反映对象当前值的值。 只要我们记住此类型的内存布局，这些字节就包含了足以重构此对象的信息。

```
public void place(Object o, long address) throws Exception {
  Class clazz = o.getClass();
  do {
    for (Field f : clazz.getDeclaredFields()) {
      if (!Modifier.isStatic(f.getModifiers())) {
        long offset = unsafe.objectFieldOffset(f);
        if (f.getType() == long.class) {
          unsafe.putLong(address + offset, unsafe.getLong(o, offset));
        } else if (f.getType() == int.class) {
          unsafe.putInt(address + offset, unsafe.getInt(o, offset));
        } else {
          throw new UnsupportedOperationException();
        }
      }
    }
  } while ((clazz = clazz.getSuperclass()) != null);
}

public Object read(Class clazz, long address) throws Exception {
  Object instance = unsafe.allocateInstance(clazz);
  do {
    for (Field f : clazz.getDeclaredFields()) {
      if (!Modifier.isStatic(f.getModifiers())) {
        long offset = unsafe.objectFieldOffset(f);
        if (f.getType() == long.class) {
          unsafe.putLong(instance, offset, unsafe.getLong(address + offset));
        } else if (f.getType() == int.class) {
          unsafe.putLong(instance, offset, unsafe.getInt(address + offset));
        } else {
          throw new UnsupportedOperationException();
        }
      }
    }
  } while ((clazz = clazz.getSuperclass()) != null);
  return instance;
}

@Test
public void testObjectAllocation() throws Exception {
  long containerSize = sizeOf(Container.class);
  long address = unsafe.allocateMemory(containerSize);
  Container c1 = new Container(10, 1000L);
  Container c2 = new Container(5, -10L);
  place(c1, address);
  place(c2, address + containerSize);
  Container newC1 = (Container) read(Container.class, address);
  Container newC2 = (Container) read(Container.class, address + containerSize);
  assertEquals(c1, newC1);
  assertEquals(c2, newC2);
}
```

请注意，这些用于在本机内存中写入和读取对象的存根方法仅支持`int`和`long`字段值。 当然， `Unsafe`支持所有原始值，甚至可以通过使用方法的易失性形式编写值，而无需访问线程本地缓存。 存根仅用于使示例简洁。 请注意，由于这些“实例”是直接分配其内存的，因此**永远**也不会垃圾回收。 （但是，也许这就是您想要的。） 此外，在计算大小时要小心，因为对象的内存布局可能取决于VM，并且与32位计算机相比，如果64位计算机运行您的代码，则该对象也会更改。 偏移甚至可能在JVM重新启动之间发生变化。

为了读取和编写原语或对象引用，`Unsafe`提供了以下类型相关的方法：

- `getXXX(Object target, long offset)`：将在指定的偏移量处从目标地址读取XXX类型的值。
- `putXXX(Object target, long offset, XXX value)`：将值放置在目标地址的指定偏移量处。
- `getXXXVolatile(Object target, long offset)`：将在指定的偏移量处从目标地址读取XXX类型的值，并且不会命中任何线程本地缓存。
- `putXXXVolatile(Object target, long offset, XXX value)`：将值放置在目标地址处的指定偏移量处，并且不会命中任何线程本地缓存。
- `putOrderedXXX(Object target, long offset, XXX value)`：将值放置在指定offet的目标地址上，并且可能不会访问所有线程本地缓存。
- `putXXX(long address, XXX value)`：将XXX类型的指定值直接放在指定地址。
- `getXXX(long address)`：将从指定地址读取XXX类型的值。
- `compareAndSwapXXX(Object target, long offset, long expectedValue, long value)`：将从目标地址的指定偏移量原子读取一个XXX类型的值，如果此偏移量的当前值等于预期值，则设置给定值。

请注意，使用`getObject(Object, long)`方法族在本地内存中写入或读取对象副本时，您正在复制引用。因此，在应用上述方法时，您仅创建实例的浅表副本。 但是，您始终可以递归读取对象大小和偏移量并创建深层副本。 但是，请注意循环对象引用，当不小心应用此原理时，循环引用会导致无限循环。

这里没有提到Unsafe类中的现有实用程序，这些实用程序允许处理诸如`staticFieldOffset`类的静态字段值并用于处理数组类型。 最后，两种名为`Unsafe#copyMemory`方法`Unsafe#copyMemory`可以指示相对于特定对象偏移量或绝对地址的直接内存复制，如以下示例所示：

```
@Test
public void testCopy() throws Exception {
  long address = unsafe.allocateMemory(4L);
  unsafe.putInt(address, 100);
  long otherAddress = unsafe.allocateMemory(4L);
  unsafe.copyMemory(address, otherAddress, 4L);
  assertEquals(100, unsafe.getInt(otherAddress));
}
```

抛出未经检查的检查异常

在`Unsafe`中还有其他有趣的方法可以找到。 您是否曾经想过要抛出特定的异常以在较低层中进行处理，但是您的高层接口类型并未声明此已检查异常？ `Unsafe#throwException`允许这样做：

```
@Test(expected = Exception.class)
public void testThrowChecked() throws Exception {
  throwChecked();
}

public void throwChecked() {
  unsafe.throwException(new Exception());
}
```

本机并发

使用`park`和`unpark`方法，您可以将线程暂停一定时间并恢复它：

```
@Test
public void testPark() throws Exception {
  final boolean[] run = new boolean[1];
  Thread thread = new Thread() {
    @Override
    public void run() {
      unsafe.park(true, 100000L);
      run[0] = true;
    }
  };
  thread.start();
  unsafe.unpark(thread);
  thread.join(100L);
  assertTrue(run[0]);
}
```

此外，可以通过使用`monitorEnter(Object)`，`monitorExit(Object)`和`tryMonitorEnter(Object)`使用Unsafe直接获取监视器。

- 包含此博客条目所有示例的文件的主旨是可用 。

参考： 臭名昭著的sun.misc.Unsafe在我的Java每日博客上由我们的JCG合作伙伴 Rafael Winterhalter 解释 。

> 翻译自: https://www.javacodegeeks.com/2013/12/the-infamous-sun-misc-unsafe-explained.html