链滴

Unsafe 类的介绍和使用

作者：Lord-X
原文链接：https://ld246.com/article/1565922847637
来源网站：链滴

□rose□rose 如果您觉得我的文章对您有帮助的话，记得在GitHub上star一波哈□ose□rose

□rose□rose GitHub_awesome-it-blog □rose□rose

近期在看JDK8的ConcurrentHashMap源码时，发现里面大量用到了Unsafe类的API，这里来深入研一下。

0 简介

Java是一个安全的面向对象的编程语言。这里的安全指的是什么呢？不妨从什么是不安全的角度来看。

啥是不安全呢？这里以C语言为例，在C语言中：

- 当编写了开辟内存的代码之后，需要想着手动去回收这块内存，否则会造成内存泄漏；
- 指针的操作提供了很多的便利，但如果出现了例如计算指针指向地址错误的问题，就会产生很多意不到的结果；

其他的不安全的情况这里不再一一列举。在Java中，很好的解决了C语言中诸多的不安全问题。例如GC解决了内存回收的问题，而Java中本身没有指针的概念，只是提供了引用类型，而引用类型是无直接修改其引用的内存地址的，所以指针误操作的问题也得到了有效的解决。

那么，在Java中有没有突破这些限制的方法呢？答案是肯定的，他就是今天要聊的sum.misc.Unsafe。

使用Unsafe的API，你可以：

- 开辟内存：allocateMemory

- 扩充内存：reallocateMemory
- 释放内存：freeMemory
- 在指定的内存块中设置值：setMemory
- 未经安全检查的加载Class：defineClass
- 原子性的更新实例对象指定偏移内存地址的值：compareAndSwapObject
- 获取系统的负载情况：getLoadAverage，等同于linux中的uptime
- 不调用构造函数来创建一个类的实例：allocateInstance

本文会介绍几个API的使用方式，但主要关注可用于处理多线程并发问题的几个API：

- compareAndSwapInt
- getAndAddInt
- getIntVolatile

1 在OpenJDK中查看Unsafe源码

想要了解Unsafe，最直接的一个方式就是看源码啦。

但是从Oracle官方下载的JDK中，Unsafe类是没有注释的。而OpenJDK中是有的，我们可以从OpenJDK源码入手。

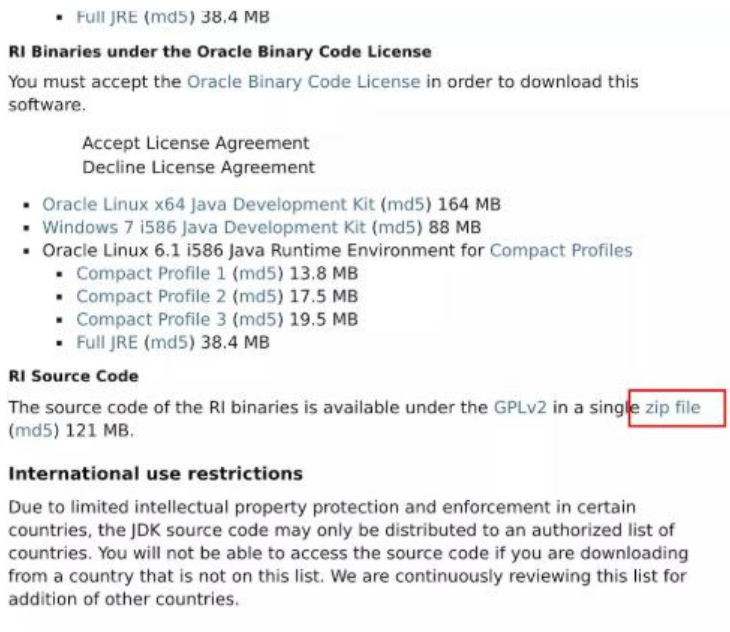
下面介绍一下如何通过OpenJDK查看Unsafe源码（同样适用与查看其它类的源码以及查看native实现）。

1.1 下载OpenJDK8

从下面链接下载

OpenJDK8

点下面这里下载即可



下载后是zip包，解压到一个地方就好。

1.2 下载NetBeans

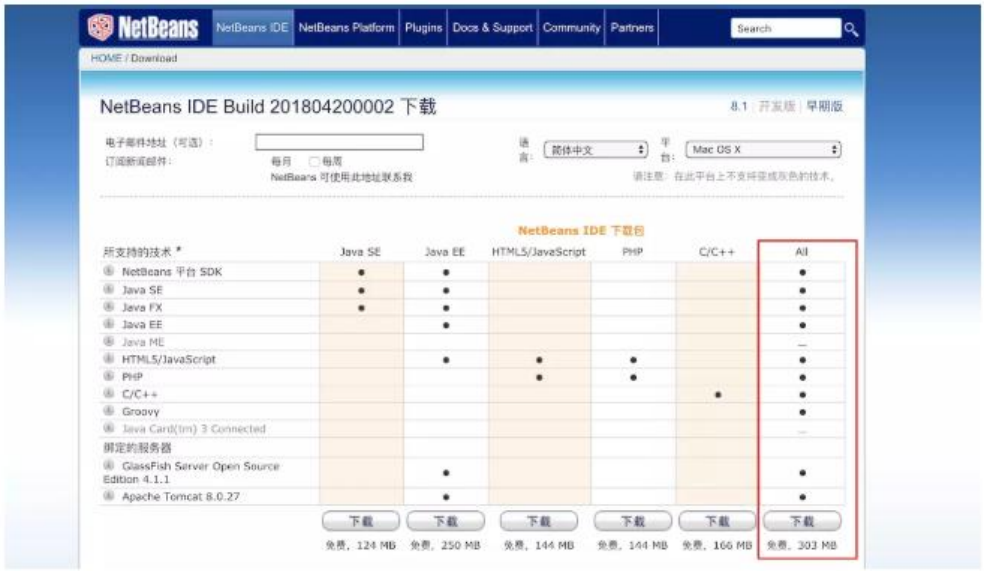
因为我们接下来要同时看C++和Java的源码，NetBeans是同时支持这两种语言的，所以这里通过NetBeans来看OpenJDK的源码。

下载地址为：

NetBeans下载

http://137.254.56.27/download/trunk/nightly/latest

注意要下载这个ALL版本的，只有这个才能同时支持Java和C++。



1.3 导入OpenJDK源码

- 文件 -> 新建项目
- 按下面这样选择，然后下一步

- 选择刚才解压出来的openjdk根目录

- 点击“完成”

Unsafe的源码在jdk/src/share/classes/sun/misc/Unsafe.java

2 Unsafe的获取

通过源码发现，Unsafe在JVM中是一个单例对象，我们不能直接去new它。

```
private Unsafe() {}

private static final Unsafe theUnsafe = new Unsafe();
```

继续往下看，然后可以发现有这样一个方法

```
/**
 * Provides the caller with the capability of performing unsafe
 * operations.
 *
 * <p> The returned <code>Unsafe</code> object should be carefully guarded
 * by the caller, since it can be used to read and write data at arbitrary
 * memory addresses.  It must never be passed to untrusted code.
 *
 * <p> Most methods in this class are very low-level, and correspond to a
 * small number of hardware instructions (on typical machines).  Compilers
 * are encouraged to optimize these methods accordingly.
 *
 * <p> Here is a suggested idiom for using unsafe operations:
 *
 * <blockquote><pre>
 * class MyTrustedClass {
 *   private static final Unsafe unsafe = Unsafe.getUnsafe();
 *   ...
```

```
 *   private long myCountAddress = ...;
 *   public int getCount() { return unsafe.getByte(myCountAddress); }
 * }
 * </pre></blockquote>
 *
 * (It may assist compilers to make the local variable be
 * <code>final</code>.)
 *
 * @exception  SecurityException  if a security manager exists and its
 *             <code>checkPropertiesAccess</code> method doesn't allow
 *             access to the system properties.
 */
@CallerSensitive
public static Unsafe getUnsafe() {
    Class<?> caller = Reflection.getCallerClass();
    if (!VM.isSystemDomainLoader(caller.getClassLoader()))
        throw new SecurityException("Unsafe");
    return theUnsafe;
}
```

看似我们可以通过Unsafe.getUnsafe()来获取Unsafe的实例。

其实不然，这个方法在return之前做了一个校验，他会通过VM.isSystemDomainLoader方法校验调者的ClassLoader，此方法的实现如下

```
/**
 * Returns true if the given class loader is in the system domain
 * in which all permissions are granted.
 */
public static boolean isSystemDomainLoader(ClassLoader loader) {
    return loader == null;
}
```

如果调用者的ClassLoader==null，在getUnsafe方法中才可以成功返回实例，否则会抛出SecurityException("Unsafe")异常。

啥时候是null呢？可以想到只有由启动类加载器（BootstrapClassLoader）加载的class才是null。

PS：关于类加载器可以参考笔者的另一篇文章【深入分析Java类加载器原理】

所以在我们自己的代码中是不能直接通过这个方法获取Unsafe实例的。

还有啥别的办法么？有的！**反射大法好！**

在源码中可以发现，它是用theUnsafe字段来引用unsafe实例的，那我们可以尝试通过反射获取theUsafe字段，进而获取Unsafe实例。代码如下：

```
import sun.misc.Unsafe;

import java.lang.reflect.Field;

public class UnsafeTest1 {
    public static void main(String[] args) throws NoSuchFieldException, IllegalAccessException {
        Class klass = Unsafe.class;
        Field field = klass.getDeclaredField("theUnsafe");
```

```
        field.setAccessible(true);
        Unsafe unsafe = (Unsafe) field.get(null);
        System.out.println(unsafe.toString());
    }
}
```

运行此代码，没有报错，大功告成。

3 Unsafe几个API的使用

3.1 通过内存偏移量原子性的更新成员变量的值

```
import sun.misc.Unsafe;

import java.lang.reflect.Field;

/**
 * Description:
 *
 * @author zhiminxu
 * @package com.lordx.sprintbootdemo
 * @create_time 2019-03-22
 */
public class UnsafeTest {

    public static void main(String[] args) throws NoSuchFieldException, IllegalAccessException,
nterruptedException {
        Test test = new Test();
        test.test();
    }
}

class Test {

    private int count = 0;

    public void test() throws NoSuchFieldException, IllegalAccessException {
        // 获取unsafe实例
        Class klass = Unsafe.class;
        Field field = klass.getDeclaredField("theUnsafe");
        field.setAccessible(true);
        Unsafe unsafe = (Unsafe) field.get(null);

        // 获取count域的Field
        Class unsafeTestClass = Test.class;
        Field fieldCount = unsafeTestClass.getDeclaredField("count");
        fieldCount.setAccessible(true);

        // 计算count的内存偏移量
        long countOffset = (int) unsafe.objectFieldOffset(fieldCount);
        System.out.println(countOffset);

        // 原子性的更新指定偏移量的值（将count的值修改为3）
```

```
        unsafe.compareAndSwapInt(this, countOffset, count, 3);

        // 获取指定偏移量的int值
        System.out.println(unsafe.getInt(this, countOffset));
    }
}
```

3.2 用Unsafe模拟synchronized

用到了Unsafe中的monitorEnter和monitorExit方法，但monitorEnter后一定要记着monitorExit。

```
import sun.misc.Unsafe;

import java.lang.reflect.Field;

/**
 * Description:
 *
 * @author zhiminxu
 * @package com.lordx.sprintbootdemo
 * @create_time 2019-03-22
 */
public class UnsafeTest {

    public static void main(String[] args) throws NoSuchFieldException, IllegalAccessException,
nterruptedException {
        final Test test = new Test();
        // 模拟两个线程并发给Test.count递增的场景
        new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 1000000; i++) {
                    test.addCount();
                }
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 1000000; i++) {
                    test.addCount();
                }
            }
        }).start();
        Thread.sleep(5000);
        System.out.println(test.getCount());
    }
}

class Test {

    private int count = 0;

    public int getCount() {
```

```
        return this.count;
    }

    private Unsafe unsafe;
    public Test() {
        try {
            Class klass = Unsafe.class;
            Field field = klass.getDeclaredField("theUnsafe");
            field.setAccessible(true);
            unsafe = (Unsafe) field.get(null);
        }catch (Exception e) {
            e.printStackTrace();
        }
    }

    private Object lock = new Object();

    public void addCount() {
        // 给lock对象设置锁
        unsafe.monitorEnter(lock);
        count++;
        // 给lock对象解锁
        unsafe.monitorExit(lock);
    }


}
```

4 Unsafe中几个线程安全API的实现原理

4.1 compareAndSwapInt

此方法在Unsafe中的源码为

```
/**
 * Atomically update Java variable to <tt>x</tt> if it is currently
 * holding <tt>expected</tt>.
 * 如果对象o指定offset所持有的值是expected，那么将它原子性的改为值x。
 * @return <tt>true</tt> if successful
 */
public final native boolean compareAndSwapInt(Object o, long offset,
                                              int expected,
                                              int x);
```

在OpenJDK中可以看到这个方法的native实现，在unsafe.cpp中。

```
UNSAFE_ENTRY(jboolean, Unsafe_CompareAndSwapInt(JNIEnv *env, jobject unsafe, jobject ob
, jlong offset, jint e, jint x))
  UnsafeWrapper("Unsafe_CompareAndSwapInt");
  // #1
  oop p = JNIHandles::resolve(obj);
  // #2
  jint* addr = (jint *) index_oop_from_field_offset_long(p, offset);
```

```
  // #3
  return (jint)(Atomic::cmpxchg(x, addr, e)) == e;
UNSAFE_END
```

代码#1将目标对象转换为oop，oop是本地实现中oopDesc类的实现，其定义在oop.hpp中。oopDec是所有class的顶层baseClass，它描述了Java object的格式，使Java object中的field可以被C++访。

代码#2负责获取oop中指定offset的内存地址，指针变量addr记录的就是这个地址中存储的int值。

代码#3调用Atomic::cmpxchg来原子性的完成值得替换。

4.2 getAndAddInt

此方法的源码如下

```
/**
 * Atomically adds the given value to the current value of a field
 * or array element within the given object <code>o</code>
 * at the given <code>offset</code>.
 *
 * @param o object/array to update the field/element in
 * @param offset field/element offset
 * @param delta the value to add
 * @return the previous value
 * @since 1.8
 */
public final int getAndAddInt(Object o, long offset, int delta) {
    int v;
    do {
        v = getIntVolatile(o, offset);
    } while (!compareAndSwapInt(o, offset, v, v + delta));
    return v;
}
```

用于原子性的将值delta加到对象o的offset上。

getIntVolatile方法用于获取对象o指定偏移量的int值，此操作具有volatile内存语义，也就是说，即对象o指定offset的变量不是volatile的，次操作也会使用volatile语义，会强制从主存获取值。

然后通过compareAndSwapInt来替换值，直到替换成功后，退出循环。