JAVA堆外内存回收机制

> 注意：本文档基于jdk1.8，在高版本的jdk中，实现与jdk1.8略有不同（如reference对象中的引用状态）

一、堆外内存概述

JVM启动时分配的内存，称为堆内存，与之相对的，在代码中还可以使用堆外内存，比如Netty，广泛使用了堆外内存，但是这部分内存不归JVM管理，GC算法并不会对它们进行回收，所以使用堆外内存是需要格外小心，以防出现内存泄露。堆外内存对象里维护了一个引用address指向了数据，从而操作数据。

堆外内存的使用场景及优势

在进行IO操作时，当我们把一个堆内存写出的时候，实际上底层实现会先构建一个临时的堆外内存，然后把堆内存的内容复制到这个临时的堆外内存上，再把这个堆外内存写出去。之所以要把jvm堆里的数据copy出来再操作，不是因为操作系统不能直接操作jvm内存，而是因为jvm在进行gc时，会进行一个标记整理的过程，会更改对象存储的内存位置，一旦出现这种问题，IO就会出现数据错乱的情况。

而使用堆外内存，就避免了数据从堆内存向堆外内存拷贝的过程，减少了一次内存拷贝，加快了读写速率。 而由于堆外内存是在JVM之外的，故不受GC管控，堆外内存的回收就成了问题。那么堆外内存是怎样进行内存回收的呢？在介绍解释这个问题前，我们先了解一些必须要知道的前置知识。

二、Reference对象概览

1.1 四类引用：

- 强引用：

> 正常的引用，生命周期最长，例如 `Object obj = new Object();` 当JVM内存不足时，宁可抛出OutOfMemoryError，也不愿回收。

- 软引用：

> 生命周期比强引用短，通过SoftReference类实现，可以通过get方法获取对象。当JVM内存不足时会先回收软引用指向的对象，即抛出OutOfMemoryError之前，会去清理软引用对象。

- 弱引用：

> 弱引用是通过WeakReference类实现的，它的生命周期比软引用还要短，也是通过get()方法获取对象。JVM内存回收时，不论是否内存不足，都会回收弱引用的对象。

- 虚引用

> 通过PhantomReference类实现，任何时候都可以被回收，可以看作没有引用。必须和引用队列联用。无法通过get方法获取对象。

综上，我们可以看出，通过软、弱、虚引用虽然可以找到被引用对象，但其并不能作为可达性分析算法的gc root。

1.2 引用对象的四个状态：

- Active：能够感知到垃圾收集器的行为，在垃圾收集器检测到referent的可达性状态已经变更后的一段时间，会将实例的状态更改为Pending或者Inactive。这取决与引用对象是否与某一个引用队列关联，如果关联则会进入Pending状态，如果没有关联则会进入Inactive状态
- Pending：实例如果处于此状态，表明它是pending-Reference列表中的一个元素，等待被Reference-handler线程加入到它关联的引用队列。未注册引用队列的实例永远不会处于该状态。
- Enqueued：实例如果处于此状态，表明它已经是它注册的引用队列中的一个元素，当它被从引用队列中移除时，它的状态将会变为Inactive，未注册引用队列的实例永远不会处于该状态。
- Inactive：实例如果处于此状态，那么它就是个废实例了，一旦一个实例变成Inactive状态永远不会再改变。

1.3 Reference Queue（引用队列）介绍

当GC（垃圾回收线程）准备回收一个对象时，如果发现它还仅有软引用(或弱引用，或虚引用)指向它，就会在回收该对象之前，把这个软引用（或弱引用，或虚引用）加入到与之关联的引用队列（Reference Queue）中。如果一个软引用（或弱引用，或虚引用）对象本身在引用队列中，就说明该引用对象所指向的对象被回收了。

> 当软引用（或弱引用，或虚引用）对象所指向的对象被回收了，那么这个引用对象本身就没有价值了，如果程序中存在大量的这类对象（注意，我们创建的软引用、弱引用、虚引用对象本身是个强引用，不会自动被gc回收），就会浪费内存。因此我们这就可以手动回收位于引用队列中的引用对象本身。

1.4 pending-Reference列表

如果一个引用对象关联了引用列表，那么在它被gc后，会被gc线程加入到pending-reference列表，等待被Reference-handler线程加入到它关联的引用队列。

三、Reference对象结构分析

3.1 Reference对象重要属性

```
    // Reference对象实际引用的对象
    private T referent;

    // Reference对象关联的引用队列
    volatile ReferenceQueue<? super T> queue;

    /* 可以理解为引用队列中的下一个值
     * When active:   NULL
     *     pending:   this
     *    Enqueued:   next reference in queue (or this if last)
     *    Inactive:   this
     */
    @SuppressWarnings("rawtypes")
    volatile Reference next;

    /* 可以理解为pending-Reference列表中的下一个值
     * When active:   next element in a discovered reference list maintained by
GC (or this if last)
     *     pending:   next element in the pending list (or null if last)
     *   otherwise:   NULL
     */
```

```
    transient private Reference<T> discovered;  /* used by VM */


    /* 锁对象，避免gc线程一方面在往pending-Reference列表在添加对象，而ReferenceHandler线程又在向pending-Reference列表取对象
     * Object used to synchronize with the garbage collector.  The collector
     * must acquire this lock at the beginning of each collection cycle.  It is
     * therefore critical that any code holding this lock complete as quickly
     * as possible, allocate no new objects, and avoid calling user code.
     */
    static private class Lock { }
    private static Lock lock = new Lock();


    /* 可以理解为pending-Reference列表的头指针
     * 此对象为static对象，注册了引用队列的对象，在被gc后，会进入pending状态，pending对象指向下一个需要enqueue进引用队列的值。
     * List of References waiting to be enqueued.  The collector adds
     * References to this list, while the Reference-handler thread removes
     * them.  This list is protected by the above lock object. The
     * list uses the discovered field to link its elements.
     */
    private static Reference<Object> pending = null;
```

3.2 Reference Handler 线程

我们再来回顾一下四种引用状态中关于pending状态的描述：

> Pending：实例如果处于此状态，表明它是pending-Reference列表中的一个元素，等待被Reference-handler线程做入队处理。未注册引用队列的实例永远不会处于该状态。

由上可知，Reference-handler线程的工作就是负责，把引用从pending-Reference列表取出，enqueue到它关联的引用队列中去。Reference-handler线程执行的代码如下：

```
 static boolean tryHandlePending(boolean waitForNotify) {
        Reference<Object> r;
        Cleaner c;
        try {
            synchronized (lock) {
                // pending-Reference列表的出队逻辑
                if (pending != null) {
                    r = pending;
                    c = r instanceof Cleaner ? (Cleaner) r : null;
                    pending = r.discovered;
                    r.discovered = null;
                } else {
                    // pending-Reference 列表中没有对象了就释放锁并阻塞，等待gc线程往pending-Reference添加值后唤醒此线程
                    // The waiting on the lock may cause an OutOfMemoryError
                    // because it may try to allocate exception objects.
                    if (waitForNotify) {
                        lock.wait();
                    }
                    // retry if waited
                    return waitForNotify;
                }
```

```
            }
        } catch (OutOfMemoryError x) {
            // Give other threads CPU time so they hopefully drop some live
references
            // and GC reclaims some space.
            // Also prevent CPU intensive spinning in case 'r instanceof
Cleaner' above
            // persistently throws OOME for some time...
            Thread.yield();
            // retry
            return true;
        } catch (InterruptedException x) {
            // retry
            return true;
        }

        // Fast path for cleaners
        if (c != null) {
            c.clean();
            return true;
        }

        ReferenceQueue<? super Object> q = r.queue;
        if (q != ReferenceQueue.NULL) q.enqueue(r);
        return true;
    }
```

那么Reference-handler线程是如何启动的呢？在Reference类中，我们可以找到如下静态代码块：

```
static {
    ThreadGroup tg = Thread.currentThread().getThreadGroup();
    for (ThreadGroup tgn = tg;
         tgn != null;
         tg = tgn, tgn = tg.getParent());
    Thread handler = new ReferenceHandler(tg, "Reference Handler");
    /* If there were a special system-only priority greater than
     * MAX_PRIORITY, it would be used here
     */
    handler.setPriority(Thread.MAX_PRIORITY);
    handler.setDaemon(true);
    handler.start();

    // provide access in SharedSecrets
    SharedSecrets.setJavaLangRefAccess(new JavaLangRefAccess() {
        @Override
        public boolean tryHandlePendingReference() {
            return tryHandlePending(false);
        }
    });
}
```

可见此线程是jvm启动时作为守护线程运行在jvm中。

四、DirectByteBuffer对象详解

4.1 DirectByteBuffer重要属性

```
    // unsafe对象，用来分配、读写、回收 DirectByteBuffer实际指向的内存
    // Cached unsafe-access object
    protected static final Unsafe unsafe = Bits.unsafe();

    // DirectByteBuffer实际指向的内存的起始地址
    // Cached array base offset
    private static final long arrayBaseOffset =
(long)unsafe.arrayBaseOffset(byte[].class);

    // 堆外内存对象的垃圾回收器
    private final Cleaner cleaner;
```

4.2 DirectByteBuffer构造方法

```
DirectByteBuffer(int cap) {
    ......
    long base = 0;
    try {
        //unsafe对象调用native方法分配一块堆外内存，并将堆外内存的起始地址复制给base
        base = unsafe.allocateMemory(size);
    } catch (OutOfMemoryError x) {
        Bits.unreserveMemory(size, cap);
        throw x;
    }
    // 将申请的堆外内存清空
    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;
    }
    // 初始化cleaner对象
    cleaner = Cleaner.create(this, new Deallocator(base, size, cap));
    ......
}
```

4.3 Cleaner 对象详解

仅截取部分重要代码：

```
/**
 * Cleaner对象是一个虚引用
 */
public class Cleaner extends PhantomReference<Object> {
    /**
     * cleaner对象关联的引用队列，但实际并未使用到，由它的命名也可以知道，dummyQueue，虚假的
queue
     * 但因为虚引用构造方法一定要传引用队列，且只有关联了引用队列的引用对象才会被Refrence-
Handler线程处理
     */
    private static final ReferenceQueue<Object> dummyQueue = new
ReferenceQueue();
```

```
    // 进行堆外内存回收的线程，实际是DirectByteBuffer的内部类Deallocator
    private final Runnable thunk;

    private Cleaner(Object var1, Runnable var2) {
        // 调用虚引用的构造方法
        super(var1, dummyQueue);
        this.thunk = var2;
    }

   // Object var0 是DirectByteBuffer的引用， Runnable var1是DirectByteBuffer的内部类
Deallocator
    public static Cleaner create(Object var0, Runnable var1) {
        return var1 == null ? null : add(new Cleaner(var0, var1));
    }

    // 执行堆外垃圾回收
    public void clean() {
        // 当前cleaner对象移除cleaner列表（不是特别重要所以不细讲）
        if (remove(this)) {
            try {
                // 执行Deallocator的run方法
                this.thunk.run();
            } catch (final Throwable var2) {
                AccessController.doPrivileged(new PrivilegedAction<Void>() {
                    public Void run() {
                        if (System.err != null) {
                            (new Error("Cleaner terminated abnormally",
var2)).printStackTrace();
                        }
                        System.exit(1);
                        return null;
                    }
                });
            }
        }
    }
}
```

再让我们来看一下Deallocator方法到底干了啥：

```
private static class Deallocator
    implements Runnable
{

    private static Unsafe unsafe = Unsafe.getUnsafe();

    private long address;
    private long size;
    private int capacity;

    private Deallocator(long address, long size, int capacity) {
        assert (address != 0);
        this.address = address;
        this.size = size;
        this.capacity = capacity;
    }
```

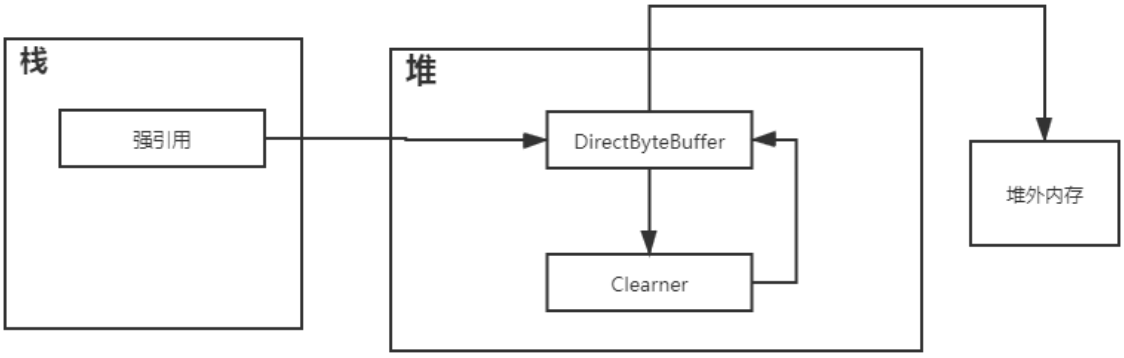
```
    public void run() {
        if (address == 0) {
            // Paranoia
            return;
        }
        // 通过unsafe对象释放堆外内存
        unsafe.freeMemory(address);
        address = 0;
        Bits.unreserveMemory(size, capacity);
    }

}
```

由此可见，cleaner对象是一个虚引用，其虚引用指向的对象就是DirectByteBuffer对象。在cleaner执行clean方法时候，会执行Deallocator线程的run方法，run方法中会通过unsafe对象释放堆外内存。那什么时候cleaner的clean方法会被调用呢？

4.4 堆外内存的释放过程

在初始状态下，内存中各部分结构如下：



栈中持有的强引用，指向了DirectByteBuffer，而Cleaner与DirectByteBuffer相互引用，且DirectByteBuffer指向了一块堆外内存。而当强引用断开时，虚引用Cleaner就进入了pending队列中等待Reference-handler线程处理。

回顾一下Reference-handler线程的工作，有如下几行代码

```
static boolean tryHandlePending(boolean waitForNotify) {
        Cleaner c;
        // 从pending队列中取引用对象，如果它是cleaner类型，则复制给cleaner c
        c = r instanceof Cleaner ? (Cleaner) r : null;
        ......
        // 如果cleaner对象部位null，则执行clean方法并返回
        if (c != null) {
            c.clean();
            return true;
        }
        ......
}
```

综上，在Reference-handler线程中，会执行Cleaner的clean方法，进行堆外内存的回收工作。