19 Netty的ByteBuf是如何支持直接内存非池化实现的

更新时间：2020-08-04 09:45:58

> 对自己不满是任何真正有才能的人的根本特征之一。——契诃夫

前言

你好，我是彤哥。

上一节，我们一起从宏观上对 ByteBuf 做了一个全面的剖析，并从微观上深入剖析了其一个子类 UnpooledUnsafeHeapByteBuf，可以发现，其底层使用的是 Java 原生的数组，并使用 Unsafe 直接更新数组中的内容。

那么，如果是 UnpooledDirectByteBuf 和 UnpooledUnsafeDirectByteBuf，它们底层又是怎样的呢？针对 UnpooledDirectByteBuf 这个类，难道不使用 Unsafe 也能操作直接内存？

本节，我们就来将这些问题解决。

问题

1. UnpooledDirectByteBuf 不使用 Unsafe 吗？
2. UnpooledUnsafeDirectByteBuf 的实现跟 DirectByteBuffer 是否完全一致？
3. 上面两者如何释放内存？

微观剖析 UnpooledDirectByteBuf

我们先从带 Unsafe 的开始，因为在 Netty 中默认就是开启 Unsafe 的。

调试用例

调试用例还是延用上一节的，略作修改：

```
public class ByteBufTest {
    public static void main(String[] args) {
        // 参数是preferDirect，即是否偏向于使用直接内存
        ByteBufAllocator allocator = new UnpooledByteBufAllocator(true);

        // 创建一个非池化基于直接内存的ByteBuf
        ByteBuf byteBuf = allocator.directBuffer();

        // 写入数据
        byteBuf.writeInt(1);
        byteBuf.writeInt(2);
        byteBuf.writeInt(3);

        // 读取数据
        System.out.println(byteBuf.readInt());
        System.out.println(byteBuf.readInt());
        System.out.println(byteBuf.readInt());

        // 记得释放内存哦
        ReferenceCountUtil.release(byteBuf);
    }
}
```

在这个调试用例中，我们分配了一个直接内存实现的 ByteBuf，并进行了写入数据和读取数据的操作，另外，还多了一步操作，即释放内存。至于为什么要手动释放内存呢？我们待会儿说。

> 其实在学习 Netty核心组件剖析 这一节，我们就说过，入站事件的处理最好继承自 SimpleChannelInboundHandler 而不是 ChannelInboundHandlerAdapter，因为 SimpleChannelInboundHandler 内部调用了 ReferenceCountUtil.release(byteBuf) 方法，会帮我们自动清理垃圾。

创建分配器

在创建分配器这里，我们把 preferDirect 这个参数由 false 改为了 true，当然了，不改也没有关系，因为后面使用的是 allocator.directBuffer () 这个方法，它同样会创建直接内存实现的 ByteBuf。

不过，这里有个要注意的点，即下面这个构造方法：

```
public UnpooledByteBufAllocator(boolean preferDirect, boolean disableLeakDetector, boolean tryNoCleaner) {
    super(preferDirect);
    this.disableLeakDetector = disableLeakDetector;
    noCleaner = tryNoCleaner && PlatformDependent.hasUnsafe()
            && PlatformDependent.hasDirectBufferNoCleanerConstructor();
}
```

> noCleaner 的判断可以参考上一节 hasUnsafe () 分析，它们最终是在同一个静态块中判断的。

这个方法中尝试去获取 Java 原生的 DirectByteBuffer 中是否包含没有 Cleaner（NoCleaner）的构造方法，即下面这个方法：

```
private DirectByteBuffer(long addr, int cap) {
    super(-1, 0, cap, cap);
    address = addr;
    cleaner = null;
    att = null;
}
```

> 我是 win8 OracleJDK，所以是有这个方法的，因此，noCleaner 等于 true。

为什么要看有没有这个方法呢？因为 Netty 想干一件大事 —— 自己控制内存的释放。

在前面的章节，我们分析过，原生 DirectByteBuffer 是依赖于 Cleaner 来做垃圾清理的，本质是将直接内存的释放与 JVM 本身的 GC 关联起来，通过虚引用的清理间接地控制直接内存的释放，那么，Netty 为什么还要多此一举，自己去控制直接内存的释放呢？其实，Netty 这么做是有道理的，把所有类型的 ByteBuf 抽象到同一个高度，主要是为了与后面所说的池化的 ByteBuf 保持一致。通过观察，可以发现，所有的 ByteBuf 都继承自 AbstractReferenceCountedByteBuf 类，所以，都可以调用其 release () 方法释放内存，包括前面说过的 UnpooledUnsafeHeapByteBuf。

创建直接内存实现的 ByteBuf

通过前面的操作，已经创建好了一个分配器，接下来，我们就使用这个分配器来创建一个直接内存实现的 ByteBuf，同样地，我们还是使用调试大法直接跟踪到 `allocator.directBuffer()` 代码内部：

```
@Override
public ByteBuf directBuffer() {
    // 调用重载方法，传入默认的初始容量和最大容量
    return directBuffer(DEFAULT_INITIAL_CAPACITY, DEFAULT_MAX_CAPACITY);
}
@Override
public ByteBuf directBuffer(int initialCapacity, int maxCapacity) {
    if (initialCapacity == 0 && maxCapacity == 0) {
        return emptyBuf;
    }
    // 检查初始容量和最大容量
    validate(initialCapacity, maxCapacity);
    // 调用另一个方法新建
    return newDirectBuffer(initialCapacity, maxCapacity);
}
@Override
protected ByteBuf newDirectBuffer(int initialCapacity, int maxCapacity) {
    final ByteBuf buf;
    // 如果支持Unsafe
    if (PlatformDependent.hasUnsafe()) {
        // 根据noCleaner的值判断使用哪个子类
        // 我的电脑noCleaner=true，所以使用的是InstrumentedUnpooledUnsafeNoCleanerDirectByteBuf
        buf = noCleaner ? new InstrumentedUnpooledUnsafeNoCleanerDirectByteBuf(this, initialCapacity, maxCapacity) :
        new InstrumentedUnpooledUnsafeDirectByteBuf(this, initialCapacity, maxCapacity);
    } else {
        // 不支持Unsafe的情况使用这个子类
        buf = new InstrumentedUnpooledDirectByteBuf(this, initialCapacity, maxCapacity);
    }
    return disableLeakDetector ? buf : toLeakAwareBuffer(buf);
}
// 省略N层构造方法
public UnpooledDirectByteBuf(ByteBufAllocator alloc, int initialCapacity, int maxCapacity) {
    super(maxCapacity);
    // 省略参数检查
```

```
    this.alloc = alloc;
    // 调用allocateDirect()方法
    setByteBuffer(allocateDirect(initialCapacity), false);
}
// InstrumentedUnpooledUnsafeNoCleanerDirectByteBuf#allocateDirect
@Override
protected ByteBuffer allocateDirect(int initialCapacity) {
    // 调用父类的
    ByteBuffer buffer = super.allocateDirect(initialCapacity);
    // 监控增强
    ((UnpooledByteBufAllocator) alloc()).incrementDirect(buffer.capacity());
    return buffer;
}
// UnpooledUnsafeNoCleanerDirectByteBuf#allocateDirect
@Override
protected ByteBuffer allocateDirect(int initialCapacity) {
    // 调用到PlatformDependent
    return PlatformDependent.allocateDirectNoCleaner(initialCapacity);
}
public static ByteBuffer allocateDirectNoCleaner(int capacity) {
    assert USE_DIRECT_BUFFER_NO_CLEANER;
    // 记录内存使用情况
    incrementMemoryCounter(capacity);
    try {
        // 调用到PlatformDependent0
        return PlatformDependent0.allocateDirectNoCleaner(capacity);
    } catch (Throwable e) {
        decrementMemoryCounter(capacity);
        throwException(e);
        return null;
    }
}
static ByteBuffer allocateDirectNoCleaner(int capacity) {
    // 调用UNSAFE分配内存，并把地址传给newDirectBuffer()方法
    return newDirectBuffer(UNSAFE.allocateMemory(Math.max(1, capacity)), capacity);
}
static ByteBuffer newDirectBuffer(long address, int capacity) {
    ObjectUtil.checkPositiveOrZero(capacity, "capacity");
    try {
        // 调用DirectByteBuffer的DirectByteBuffer(long addr, int cap)构造方法
        return (ByteBuffer) DIRECT_BUFFER_CONSTRUCTOR.newInstance(address, capacity);
    } catch (Throwable cause) {
        if (cause instanceof Error) {
            throw (Error) cause;
        }
        throw new Error(cause);
    }
}
```

所以，针对非池化且使用直接内存的情况下，Netty 默认创建的就是一个不使用 Cleaner 清理垃圾的 DirectByteBuffer，没错，Netty 底层使用的就是 Java 原生的 DirectByteBuffer，这个也可以在 UnpooledDirectByteBuf 的属性中找到：

```
public class UnpooledDirectByteBuf extends AbstractReferenceCountedByteBuf {
    ByteBuffer buffer;
}
```

既然使用的是 Java 原生的 DirectByteBuffer，那么写入数据和读取数据肯定是与其保持一致的，这两个方法我们就不再分析了，有兴趣地同学可以使用调试大法跟踪一下。

那么，问题来了：Netty 既然不使用 Cleaner 来自动清理垃圾，那它怎么实现垃圾清理呢？

释放内存

这就是 `ReferenceCountUtil.release(byteBuf);` 这行代码的作用了，同样地，我们跟踪进去瞧一瞧：

```
// ReferenceCountUtil#release
public static boolean release(Object msg) {
    if (msg instanceof ReferenceCounted) {
        // 转换成ReferenceCounted
        return ((ReferenceCounted) msg).release();
    }
    return false;
}
// AbstractReferenceCountedByteBuf#release()
@Override
public boolean release() {
    // updater.release(this)将其引用计数减一
    // 默认的引用计数为2
    // 如果减到1的话表示可以清理掉了，此时会返回true
    return handleRelease(updater.release(this));
}
private boolean handleRelease(boolean result) {
    // 如果减到1的话，就把它清理掉
    if (result) {
        deallocate();
    }
    return result;
}
@Override
protected void deallocate() {
    ByteBuffer buffer = this.buffer;
    if (buffer == null) {
        return;
    }
    // 将对DirectByteBuffer的引用置为空，方便GC清理this.buffer引用的对象
    this.buffer = null;

    if (!doNotFree) {
        // 调用freeDirect
        freeDirect(buffer);
    }
}
// InstrumentedUnpooledUnsafeNoCleanerDirectByteBuf#freeDirect
@Override
protected void freeDirect(ByteBuffer buffer) {
    int capacity = buffer.capacity();
    // 调用父类
    super.freeDirect(buffer);
    // 监控增强
    ((UnpooledByteBufAllocator) alloc()).decrementDirect(capacity);
}
@Override
protected void freeDirect(ByteBuffer buffer) {
    // 调用到PlatformDependent
    PlatformDependent.freeDirectNoCleaner(buffer);
}
public static void freeDirectNoCleaner(ByteBuffer buffer) {
    assert USE_DIRECT_BUFFER_NO_CLEANER;

    int capacity = buffer.capacity();
    // 调用到PlatformDependent
    PlatformDependent0.freeMemory(PlatformDependent0.directBufferAddress(buffer));
    decrementMemoryCounter(capacity);
}
static void freeMemory(long address) {
    // 调用Unsafe释放直接内存
    UNSAFE.freeMemory(address);
}
```

可以看到，最终是调用的 Unsafe 的 freeMemory () 方法来释放内存的，这与 DirectByteBuffer 的最终结果又是一致的（Deallocator 中调用了 Unsafe 的 freeMemory () 方法）。

所以，我们总结归纳一下整个过程：

1. 在创建分配器 UnpooledByteBufAllocator 的时候会查找 DirectByteBuffer 中是否包含 NoCleaner 的构造方法；
2. 在创建 UnpooledDirectByteBuf 的时候默认创建的是其子类 InstrumentedUnpooledUnsafeNoCleanerDirectByteBuf，这个类将不使用 Cleaner 来清理垃圾，同时，它的底层使用的是 Java 原生的 DirectByteBuffer，且包含 Unsafe；
3. 在清理内存的时候需要手动调用 `ReferenceCountUtil.release(byteBuf);` 方法；
4. 最终也是调用 Unsafe 的 freeMemory () 来释放内存；

其它

在上面分析的过程中，我们发现 InstrumentedUnpooledUnsafeNoCleanerDirectByteBuf 类还有两个兄弟类：

- InstrumentedUnpooledUnsafeDirectByteBuf
- InstrumentedUnpooledDirectByteBuf

从名字上也可以看出，前者不包含 `NoCleaner`，那可能就是使用的 Cleaner 来清理垃圾，后者不包含 `Unsafe`，那可能就是不使用 Unsafe 来操作直接内存了吧，具体是不是这样呢？我们简单快速地过一下源码。

InstrumentedUnpooledUnsafeDirectByteBuf

直接打开这个类：

```
private static final class InstrumentedUnpooledUnsafeDirectByteBuf extends UnpooledUnsafeDirectByteBuf {
    InstrumentedUnpooledUnsafeDirectByteBuf(
        UnpooledByteBufAllocator alloc, int initialCapacity, int maxCapacity) {
        super(alloc, initialCapacity, maxCapacity);
    }

    @Override
    protected ByteBuffer allocateDirect(int initialCapacity) {
        ByteBuffer buffer = super.allocateDirect(initialCapacity);
        ((UnpooledByteBufAllocator) alloc()).incrementDirect(buffer.capacity());
        return buffer;
    }

    @Override
    protected void freeDirect(ByteBuffer buffer) {
        int capacity = buffer.capacity();
        super.freeDirect(buffer);
        ((UnpooledByteBufAllocator) alloc()).decrementDirect(capacity);
    }
}
```

与其它 `Instrumented` 开头的增强类几乎没有任何区别，其实，真正区别的地方在于它们的父类，通过观察可以发现，InstrumentedUnpooledUnsafeNoCleanerDirectByteBuf 继承自 UnpooledUnsafeNoCleanerDirectByteBuf ，而 InstrumentedUnpooledUnsafeDirectByteBuf 继承自 UnpooledUnsafeDirectByteBuf，所以，真正的区别还是位于它们的父类中，我们以 `allocateDirect()` 这个方法为例，跟踪到它的父类中：

```
// UnpooledDirectByteBuf#allocateDirect
protected ByteBuffer allocateDirect(int initialCapacity) {
    // 这不是`再谈ByteBuffer`中我们的调试用例使用的方法么？！
    return ByteBuffer.allocateDirect(initialCapacity);
}
// ByteBuffer#allocateDirect
public static ByteBuffer allocateDirect(int capacity) {
    // 直接创建了DirectByteBuffer，包含Cleaner的
    return new DirectByteBuffer(capacity);
}
```

OK，到这里就很清晰了，也就是说，如果 DirectByteBuffer 中不包含 NoCleaner 的构造方法，就使用与 Java 原生一样的方式，创建一个包含 Cleaner 的 DirectByteBuffer。

InstrumentedUnpooledDirectByteBuf

同样地，如果没有 Unsafe，直接看其父类 UnpooledDirectByteBuf 的 allocateDirect () 方法：

```
// UnpooledDirectByteBuf#allocateDirect
protected ByteBuffer allocateDirect(int initialCapacity) {
    return ByteBuffer.allocateDirect(initialCapacity);
}
// ByteBuffer#allocateDirect
public static ByteBuffer allocateDirect(int capacity) {
    return new DirectByteBuffer(capacity);
}
```

纳尼，与上面的那个 InstrumentedUnpooledUnsafeDirectByteBuf 一样的处理逻辑，是的，你没有看错，就是一样的。

其实，也不完全一样，仔细观察一下，InstrumentedUnpooledUnsafeDirectByteBuf 的父类 UnpooledUnsafeDirectByteBuf 是继承自 InstrumentedUnpooledDirectByteBuf 的父类 UnpooledDirectByteBuf 的，它里面加了一个属性 memoryAddress：

```
public class UnpooledUnsafeDirectByteBuf extends UnpooledDirectByteBuf {
    long memoryAddress;
}
```

Netty 这里的意思并不是说不支持 Unsafe 就甩锅给 JDK，而是说 Netty 支不支持使用 Unsafe，如果可以使用，那么，它就可以使用 Unsafe 来操作直接内存，如果不支持使用 Unsafe，那它也只能交给 JDK 自己来处理了。

> Netty 支不支持使用 Unsafe 是通过参数 io.netty.noUnsafe 来控制，默认为 false，也就是可以使用 Unsafe。

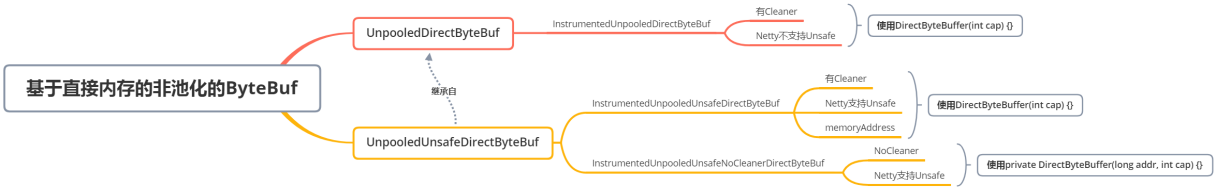
好了，到这里，UnpooledDirectByteBuf 和 UnpooledUnsafeDirectByteBuf 的分析就结束了。

后记

今天，我们一起学习了 UnpooledDirectByteBuf 和 UnpooledUnsafeDirectByteBuf 的实现细节，总的来说，它们底层都是使用 Java 原生的 DirectByteBuffer，不同之处在于，默认情况下，Netty 是自己控制直接内存的释放，即使用 DirectByteBuffer 的 NoCleaner 构造方法，且支持使用 Unsafe。如果 DirectByteBuffer 没有 NoCleaner 方法，则使用包含 Cleaner 的构造方法，同时也支持使用 Unsafe。如果显式地说明了不支持使用 Unsafe 的话，那么最后只能交给 JDK 自己来处理了。

后面，我们将学习池化相关的概念，简单地说，Netty 中包含两种池化 —— 内存池和对象池，它们有怎样的区别和联系呢，敬请期待。

思维导图



}

18 Netty的ByteBuf是如何支持堆内存非池化实现的

20 jemalloc内存分配器是什么