17 如何从源码的角度深入剖析ByteBuffer

更新时间：2020-07-31 09:42:39

> 生活的理想，就是为了理想的生活。——张闻天

前言

你好，我是彤哥。

前面的章节，我们一起从数据流向的角度剖析了 Netty 的源码，包括服务的启动、接收新的连接或数据、写出数据、关闭服务等。

然而，从数据流向的角度只能窥见 Netty 很小一部分的源码，比如，你只看见了调用一个方法就能创建符合要求的 ByteBuf 却不知为何如此简单，你只看见了 Netty 使用了线程池却不知线程池用的是什么队列，你只看见了 Netty 到处都在使用 Promise 却不知 Promise 为何物。

所以，从本节开始，我们将从 Netty 核心知识的角度来剖析源码，同时，我们也会一起学习很多 Java 中的高阶技巧。

在前面的章节，我们多多少少地介绍过一些 ByteBuffer、ByteBuf 相关的知识，但是它们还不全面，所以，本节，我想先从 Java 原生的 ByteBuffer 入手，来更好地过渡到 Netty 的源码设计。

问题

我们知道，Netty 之所以如此高效，很大一部分原因得益于其对直接内存的高效使用，所以，今天，我想问：

1. Java 中的 ByteBuffer 有直接内存的实现吗？
2. Java 中如何使用直接内存？又如何释放直接内存呢？

3. Java 中 ByteBuffer 的直接内存实现又是如何管理直接内存的？

好了，让我们带着这些问题进入今天的探索吧。

Buffer 的分类

经过前面的学习，我们知道，Buffer 按照不同的维度有不同的分类，大体上有两种主要的维度，按照数据类型分为 ByteBuffer、CharBuffer、ShortBuffer、IntBuffer、LongBuffer、FloatBuffer、DoubleBuffer 等，按照内存实现可以分为堆内存实现和直接内存实现。其实，还有另一种不太常见的维度，按照读写的维度分为只读和可读写，一般来说，只读的 Buffer 后面以 R 结尾，比如 HeapByteBufferR，这种比较少见，有兴趣的同学可以自己看看相关的源码。

今天我们的主角是堆内存实现和直接内存实现，它们分别是怎么实现的呢？有什么区别吗？

不过，在正式介绍之前，我想讲另外一个非常有意思的类，我把它称作 Java 中的魔法类 ——Unsafe。

不安全的 Unsafe

看过并发集合或者原子类源码的同学，应该对 Unsafe 这个类印象比较深刻，像我们经常使用的 CAS 操作，底层就是使用 Unsafe 来实现的，比如，AtomicInteger 中的 compareAndSet () 方法：

```
public final boolean compareAndSet(int expect, int update) {
    return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
}
```

然而，Unsafe 的功能远远不止 CAS 这一种操作，有兴趣的同学可以看看这篇文章【死磕 java 魔法类之 Unsafe 解析】，今天，我们再介绍一种 Unsafe 不太常见的功能 —— 操作直接内存。

Unsafe 操作直接内存是通过下面几个方法实现的：

```
// 分配内存
public native long allocateMemory(long var1);
// 释放内存
public native void freeMemory(long var1);
// 设置内存值
public native void setMemory(Object var1, long var2, long var4, byte var6);
// 设置某种类型的值，比如putInt()
public native void putXxx(long var1, xxx var3);
// 获取某种类型的值，比如getInt()
public native xxx getXxx(long var1);
```

比如，我们可以使用 Unsafe 来实现一个直接内存实现的 int 数组。

```
public class DirectIntArray {
    // 一个int等于4个字节
    private static final int INT = 4;
    private long size;
    private long address;

    private static Unsafe unsafe;
    static {
        try {
            // Unsafe类有权限访问控制，只能通过反射获取其实例
            Field f = Unsafe.class.getDeclaredField("theUnsafe");
            f.setAccessible(true);
            unsafe = (Unsafe) f.get(null);
        } catch (NoSuchFieldException e) {
```

```
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    public DirectIntArray(long size) {
        this.size = size;
        // 参数字节数
        address = unsafe.allocateMemory(size * INT);
    }

    // 获取某位置的值
    public int get(long i) {
        if (i >= size) {
            throw new ArrayIndexOutOfBoundsException();
        }
        return unsafe.getInt(address + i * INT);
    }

    // 设置某位置的值
    public void set(long i, int value) {
        if (i >= size) {
            throw new ArrayIndexOutOfBoundsException();
        }
        unsafe.putInt(address + i * INT, value);
    }

    // 数组大小
    public long size() {
        return size;
    }

    // 释放内存
    public void freeMemory() {
        unsafe.freeMemory(address);
    }

    public static void main(String[] args) {
        // 创建数组并赋值
        DirectIntArray array = new DirectIntArray(4);
        array.set(0, 1);
        array.set(1, 2);
        array.set(2, 3);
        array.set(3, 4);
        // 下面这行数组越界了
//        array.set(5, 5);

        int sum = 0;
        for (int i = 0; i < array.size(); i++) {
            sum += array.get(i);
        }
        // 打印10
        System.out.println(sum);

        // 最后别忘记释放内存
        array.freeMemory();
    }
}
```

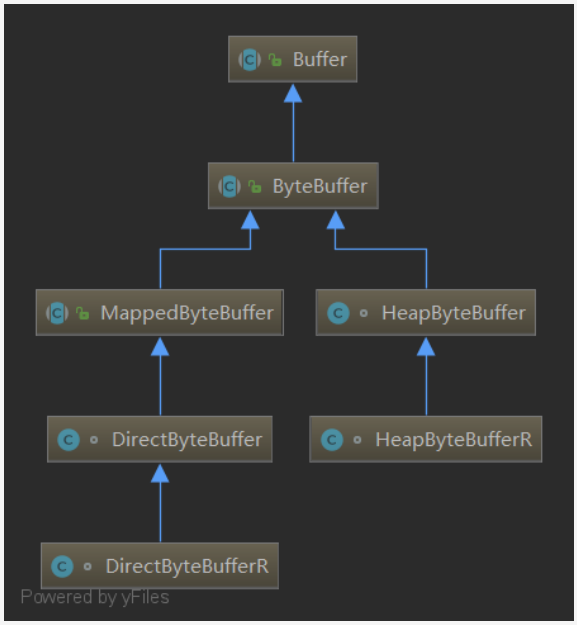
最后，一定别忘了释放内存，这是操作直接内存的一个非常“不安全的”地方，谨记！

好了，有了上面的介绍，相信你对 Java 中如何操作直接内存一定有了非常清晰的认识，其实，也是非常简单的。

下面，我们就正式地来揭开 Java 中堆内存和直接内存两种方式实现的 ByteBuffer 的神秘面纱。

宏观分析 ByteBuffer

> 学习源码一般遵循着先宏观再微观的原则。宏观上，一般先看继承体系，类的基本结构等，通过这种方式一般能找到一到两个突破口。微观上，一般根据宏观找到的突破口，进入调试，调试，调试。很多同学看源码喜欢干看，其实是不对的，一定要调试，不调试就无法掌握细节。



从继承体系上，ByteBuffer 是一个抽象类，它继承自 Buffer 抽象类，它有两个主要实现类 ——HeapByteBuffer 和 DirectByteBuffer，其中 DirectByteBuffer 和 ByteBuffer 之间还有一个 MappedByteBuffer，另外，这两个实现类分别还有自己的只读模式，即 HeapByteBufferR 和 DirectByteBufferR。

> MappedByteBuffer 是做什么的呢？本节的知识点不涉及到这个类，有兴趣的同学可以自己探索。

从类的基本结构上，ByteBuffer 包含两个非常重要的方法：

```
// 创建一个直接内存实现的ByteBuffer
public static ByteBuffer allocateDirect(int capacity) {
    return new DirectByteBuffer(capacity);
}
// 创建一个堆内存实现的ByteBuffer
public static ByteBuffer allocate(int capacity) {
    if (capacity < 0)
        throw new IllegalArgumentException();
    return new HeapByteBuffer(capacity, capacity);
}
```

这两个方法就是我们微观分析时需要使用到的突破口。

另外，还包含一些操作 ByteBuffer 的方法，主要是 put () 和 get ()，以及切片、子集等，与数组或者集合 list 的操作方法比较类似，这里就不一一列举了。

有了上面的突破口，我们就可以正式进入微观分析阶段了，即调试。

微观分析 ByteBuffer

堆内存实现的 ByteBuffer——HeapByteBuffer

既然要调试，当然要写调试用例啦，我这里也准备了一个调试用例：

```
public class ByteBufferTest {
    public static void main(String[] args) {
        // 1. 创建一个堆内存实现的ByteBuffer
        ByteBuffer buffer = ByteBuffer.allocate(12);
        // 2. 写入值
        buffer.putInt(1);
        buffer.putInt(2);
        buffer.putInt(3);

        // 3. 切换为读模式
        buffer.flip();

        // 4. 读取值
        System.out.println(buffer.getInt());
        System.out.println(buffer.getInt());
        System.out.println(buffer.getInt());
    }
}
```

OK，我们要开始调试了，你准备好了么？！Let's go!

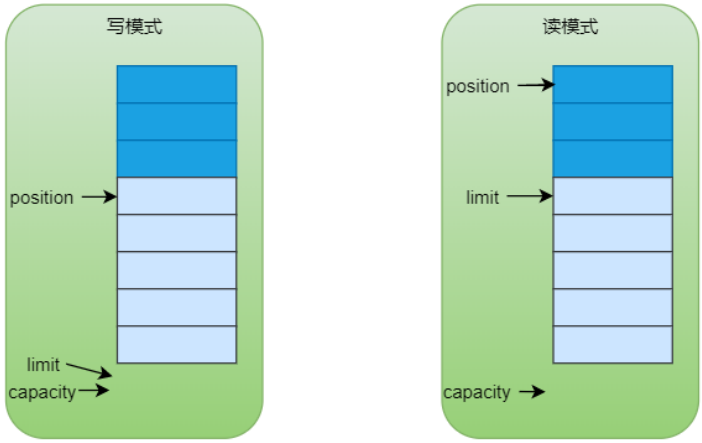
让我们把断点打在 `ByteBuffer buffer = ByteBuffer.allocate(12);` 这行，跟踪进去：

```
// 1. 创建堆内存实现的ByteBuffer
public static ByteBuffer allocate(int capacity) {
    if (capacity < 0)
        throw new IllegalArgumentException();
    return new HeapByteBuffer(capacity, capacity);
}
// 2. HeapByteBuffer的构造方法
HeapByteBuffer(int cap, int lim) {           // package-private
    // lim = cap = 12
    // 创建了一个12大小的byte数组
    // 调用父构造方法
    super(-1, 0, lim, cap, new byte[cap], 0);
}
// 3. ByteBuffer的构造方法
ByteBuffer(int mark, int pos, int lim, int cap,   // package-private
           byte[] hb, int offset)
{
    // 调用父构造方法
    // pos = 0，默认创建的就是写模式
    // lim = cap = 12
    super(mark, pos, lim, cap);
    // byte数组hb（heap buffer），为上面传过来的new byte[cap]
    this.hb = hb;
    this.offset = offset;
}
// 4. Buffer的构造方法
Buffer(int mark, int pos, int lim, int cap) {       // package-private
    if (cap < 0)
        throw new IllegalArgumentException("Negative capacity: " + cap);
    // 三个非常重要的变量：capacity、limit、position
    this.capacity = cap;
    limit(lim);
    position(pos);
    if (mark >= 0) {
        if (mark > pos)
            throw new IllegalArgumentException("mark > position: ("
                                               + mark + " > " + pos + ")");
        this.mark = mark;
    }
}
```

整个创建的过程非常简单，主要包含以下逻辑：

1. 创建了一个 byte 数组保存在 hb 这个变量中；
2. 给几个重要的变量赋值，比如 capacity、limit、position，还有一个 mark，感兴趣的同学可以自己看看这个变量的作用；
3. 默认创建的 ByteBuffer 为写模式，因为 position 从 0 开始且 capacity=limit = 数组大小；

还记得写模式吗？还记得 capacity、limit、position 这三个重要的属性吗？让我们把前面的图再拿出来回顾一下。



> 在上面的左图中，limit=capacity=8，表示的是数组的大小，而数组的下标是从 0 开始的，所以这里指向了数组最大位置的下一个位置。position 表示的是下一次写入的位置，从 0 开始，与数组的下标保持一致，所以，这里的 position 最大只能等于 8，也就是 limit 的值，当等于 8 时再写入，就溢出了（8 这个位置不会写入数据）。

OK，到这里 ByteBuffer 我们就创建好了，所谓堆内存的实现方式，就是使用的 Java 自带的 byte 数组来实现的，让我们再来看看写入 putInt () 这个方法是如何实现的。

```
// 写入一个int类型的数值
public ByteBuffer putInt(int x) {
    // 调用Bits工具类的putInt()方法，Bits是位的意思
    // 堆内存的实现中使用大端法来存储数据
    Bits.putInt(this, ix(nextPutIndex(4)), x, bigEndian);
    return this;
}
// 移动position到下一个位置
// 因为一个int占4个字节，所以这里往后移动4位
final int nextPutIndex(int nb) {                    // package-private
    // 判断有没有越界
    if (limit - position < nb)
        throw new BufferOverflowException();
    int p = position;
    position += nb;
    // 注意，这里返回的是移动前的位置，初始值为0
    return p;
}
// 计算写入的偏移量，初始值为0
protected int ix(int i) {
    return i + offset;
}
// java.nio.Bits#putInt(java.nio.ByteBuffer, int, int, boolean)
static void putInt(ByteBuffer bb, int bi, int x, boolean bigEndian) {
    // 堆内存使用的是大端法，更符合人们的习惯
    if (bigEndian)
        // 大端法
        putIntB(bb, bi, x);
    else
        putIntL(bb, bi, x);
}
// java.nio.Bits#putIntB(java.nio.ByteBuffer, int, int)
static void putIntB(ByteBuffer bb, int bi, int x) {
    // 把一个int拆分成4个byte，分别写入
    // int3(int x) { return (byte)(x >> 24); }
    bb._put(bi    , int3(x));
    // int2(int x) { return (byte)(x >> 16); }
    bb._put(bi + 1, int2(x));
    // int1(int x) { return (byte)(x >>  8); }
    bb._put(bi + 2, int1(x));
    // int0(int x) { return (byte)(x      ); }
    bb._put(bi + 3, int0(x));
}
// java.nio.HeapByteBuffer#_put
void _put(int i, byte b) {                  // package-private
    // 最终变成了修改byte数组
    hb[i] = b;
}
```

写入方法无非就是根据当前 position 的位置往后写入一个 int 大小的数据，写入的时候会把 int 拆分成 4 个 byte 分别写入，而最终其实就是修改前面创建的 byte 数组。

OK，同样地，读取方法应该就是先根据当前 position 计算读取的偏移量，再从数组中读取 4 个字节的数据，最后再拼装成一个 int 类型返回。这块的代码相对来说都比较简单，我们就不一一细看了。

综上所述，HeapByteBuffer 内部使用 byte 数组来存储数据，并根据 position 来写入或者读取数据，既然使用的是 Java 中的类型，自然使用的是堆内存。

直接内存实现的 ByteBuffer——DirectByteBuffer

上面我们简单介绍了 HeapByteBuffer，它主要是使用 byte 数组来实现的，那么，DirectByteBuffer 是如何实现的呢？通过上面我们讲解的 Unsafe，结合 HeapByteBuffer 的实现，你能自己实现一个 DirectByteBuffer 吗，就像直接内存实现的 DirectIntArray 一样？

建议先想好这几个问题，再接着看下面的源码分析。

同样地，我们还是使用上面的调试用例，只不过改动一个方法：

```
public class ByteBufferTest {
    public static void main(String[] args) {
        // 创建一个直接内存实现的ByteBuffer
        ByteBuffer buffer = ByteBuffer.allocateDirect(12);
        // 写入值
        buffer.putInt(1);
        buffer.putInt(2);
        buffer.putInt(3);

        // 切换为读模式
        buffer.flip();

        // 读取值
        System.out.println(buffer.getInt());
        System.out.println(buffer.getInt());
        System.out.println(buffer.getInt());
    }
}
```

问题无处不在，在 DirectIntArray 的使用中，我们是手动调用 freeMemory () 来释放内存的，DirectByteBuffer 的使用过程中如何释放内存，保证内存不泄漏？

在 `ByteBuffer buffer = ByteBuffer.allocateDirect(12);` 这行打一个断点，一步一步跟踪进去：

```
public static ByteBuffer allocateDirect(int capacity) {
    // 创建直接内存实现的ByteBuffer
    return new DirectByteBuffer(capacity);
}
DirectByteBuffer(int cap) {                   // package-private
    // 调用父构造方法，设置position/limit/capacity/mark这几个值
    // 与HeapByteBuffer类似，只不过没有创建hb那个数组
    super(-1, 0, cap, cap);
    // 是否页对齐，默认为否
    boolean pa = VM.isDirectMemoryPageAligned();
    // 每页大小
    int ps = Bits.pageSize();
    long size = Math.max(1L, (long)cap + (pa ? ps : 0));
    // 先预订内存，如果内存不够，会进行清理，并尝试几次
    Bits.reserveMemory(size, cap);

    long base = 0;
    try {
        // key1，重点来了，调用unsafe的allocateMemory()方法来分配内存
        base = unsafe.allocateMemory(size);
    } catch (OutOfMemoryError x) {
        Bits.unreserveMemory(size, cap);
        throw x;
    }
    // key2，初始化这片内存的值为0
    unsafe.setMemory(base, size, (byte) 0);
    // 根据是否页对齐计算实际的地址
    if (pa && (base % ps != 0)) {
        // Round up to page boundary
        address = base + ps - (base & (ps - 1));
    } else {
        // 默认不页对齐，所以地址就等于allocateMemory()返回的地址
        address = base;
    }
    // key3，Cleaner是什么？干什么的？有什么作用？
    cleaner = Cleaner.create(this, new Deallocator(base, size, cap));
    att = null;
}
```

> 看源码有个准则，一定要学会抓重点，对于看不懂的东西可以先记下并跳过，比如，DirectByteBuffer 的构造方法中其实牵涉到很多高阶知识，像页对齐、弱引用 / 虚引用（在 reserveMemory () 方法中）等相关的东西，这部分东西非常复杂且难以理解，先记下来，等把整体流程理清楚了，再回头深究这一块的东西，其实也是遵循着从宏观到微观的方法论，宏观使你了解整体流程，微观才能使你的知识体系得到升华。

好了，针对 DirectByteBuffer 的构造方法，整体流程与 HeapByteBuffer 是比较类似的，只不过不是创建一个 byte 数组来保存数据，而是调用 unsafe 来分配内存并保存数据，总结下来有三个非常重要的地方：

1. `base = unsafe.allocateMemory(size);`，调用 unsafe 的 allocateMemory () 方法来分配内存
2. `unsafe.setMemory(base, size, (byte) 0);`，初始化这片内存的值为 0，为什么要进行初始化？如果不初始化，之前这块内存可能被别的程序使用过，会残留一些数据，对当前的数据造成影响，这是我们写 DirectIntArray 没有考虑到的。
3. `cleaner = Cleaner.create(this, new Deallocator(base, size, cap));`，这行代码是干什么的？看着似乎跟清理内存有关，这个我们等会再看，先来看看如何写入数据和读取数据。

经过上面的折腾，我们终于创建好了一个 DirectByteBuffer，接下来，我们来一起看看如何写入数据和读取数据吧。

写入数据调用的是 buffer.putInt(1); 这个方法，同样地，调试跟踪进去：

```
// 写入一个int类型的数值
public ByteBuffer putInt(int x) {
    // 1 << 2 = 4，一个int占4个字节
    putInt(ix(nextPutIndex((1 << 2))), x);
    return this;
}
// 计算下一个position的位置并返回当前position的值
final int nextPutIndex(int nb) {                    // package-private
    if (limit - position < nb)
        throw new BufferOverflowException();
    int p = position;
    position += nb;
    // 返回移动前的值
    return p;
}
// 计算偏移量，在address的基础上加上position的值
private long ix(int i) {
    return address + ((long)i << 0);
}
private ByteBuffer putInt(long a, int x) {
    // unaligned不是之前讲的那个页对齐
    // 这里是跟CPU架构相关的一个参数
    if (unaligned) {
        int y = (x);
        // 在windows系统中内存值使用的是小端法，所以直接内存使用的是小端法
        // 因此，这里要转换一下
        // 调用unsafe的putInt()方法修改直接内存中对应地址的值
        unsafe.putInt(a, (nativeByteOrder ? y : Bits.swap(y)));
    } else {
        Bits.putInt(a, x, bigEndian);
    }
    return this;
}
```

写入方法无非就是根据当前 position 的位置往后写入一个 int 大小的数据，写入的时候会调用 unsafe 的 putInt () 方法在内存中对应地址的位置直接写入值，而不是像 HeapByteBuffer 那样修改 byte 数组对应位置的值。

OK，同样地，读取方法应该就是先根据当前 position 计算读取的偏移地址，再调用 unsafe 的 getInt () 方法在内存中对应地址的位置读取一个 int 大小的数据，这块的代码相对来说都比较简单，我们就不一一细看了。

综上所述，DirectByteBuffer 底层使用的是 Unsafe 来分配一块直接内存，并在写入数据和读取数据的时候使用 Unsafe 对应的方法来操作直接内存，了解了其原理，是不是也很简单呢？

好了，到这里 DirectByteBuffer 的基本原理我们已经看透 80% 了，为什么是 80% 呢？

还记得前面我们使用 DirectIntArray 的时候最后要调用 freeMemory () 来清理内存吗？

是的，在 DirectByteBuffer 这里，我们并没有看到清理内存的相关代码，那肯定是有问题的，哎不对，上面好像有个 Cleaner，看着像是清理什么东西，那么，它是不是清理内存的呢？让我们再仔细研究一下。

```
cleaner = Cleaner.create(this, new Deallocator(base, size, cap));
```

这里新建了一个叫作 Deallocator 的对象，所谓 Deallocator，它等于 De + allocator，在英语中，De 前缀一般表示相反的意思，比如，increse 是升高的意思，而 decrease 是下降的意思，所以，allocate 是分配的意思，deallocate 应该是解除分配的意思，也就是清理的意思，变成名词就是 deallocator，可以理解为清理器的意思。

到底是不是我们理解的意思呢？直接上代码：

```
private static class Deallocator implements Runnable {

    private static Unsafe unsafe = Unsafe.getUnsafe();

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

    // 构造方法传入allocate的时候返回的地址，以及容量等参数
    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;
        }
        // 调用unsafe的freeMemory释放内存
        unsafe.freeMemory(address);
        address = 0;
        // 取消预订的内存
        Bits.unreserveMemory(size, capacity);
    }

}
```

Deallocator 实现了 Runnable 接口，Runnable 接口是什么？大家都比较熟悉了，它是线程执行的任务。那么，这个任务是在什么时候执行的呢？又干了什么呢？我们先来看第二个问题，从上面的代码中可以看到它调用了 unsafe 的 freeMemory () 方法来释放内存，所以，这个任务的作用就是清理内存。

但是，这个任务又是在什么时候执行的呢？或者说，在哪里执行的呢？还是回到创建的地方，也就是下面这行代码：

```
cleaner = Cleaner.create(this, new Deallocator(base, size, cap));
```

这里创建了一个 Cleaner 的对象，跟踪进去看看 Cleaner 类（前方高能，请系好安全带！）：

```
// 虚引用
public class Cleaner extends PhantomReference<Object> {
    private static final ReferenceQueue<Object> dummyQueue = new ReferenceQueue();
    private static Cleaner first = null;
    private Cleaner next = null;
    private Cleaner prev = null;
    private final Runnable thunk;

    private static synchronized Cleaner add(Cleaner var0) {
        // 省略部分代码，将var0添加到Cleaner链表中
        return var0;
    }

    private static synchronized boolean remove(Cleaner var0) {
        // 省略部分代码，将var0从链表中移除
    }

    private Cleaner(Object var1, Runnable var2) {
        // 调用父类的构造方法
        // ★Cleaner这个虚引用引用的对象是var1，也就是Deallocaotr对象
        // 先记住上面这句话！！！
        super(var1, dummyQueue);
        // var2即上面创建的Deallocator对象
        this.thunk = var2;
    }

    public static Cleaner create(Object var0, Runnable var1) {
        // 创建一个Cleaner对象，并返回这个对象
        // 它里面封装了一个任务
        return var1 == null ? null : add(new Cleaner(var0, var1));
    }

    public void clean() {
        // 从链表中移除当前对象
        if (remove(this)) {
            try {
                // 执行任务
                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;
                    }
                });
            }

        }
    }
}
```

可见，Cleaner 继承自一个叫作 PhantomReference 的类，PhantomReference 是什么呢？

PhantomReference 翻译过来叫作虚引用，它还有三个兄弟，一个叫作强引用，一个叫作软引用，还有一个叫作弱引用。

强引用，使用最普遍的引用。如果一个对象具有强引用，它绝对不会被 gc 回收。如果内存空间不足了，gc 宁愿抛出 OutOfMemoryError，也不是会回收具有强引用的对象。

软引用（SoftReference），如果一个对象只具有软引用，则内存空间足够时不会回收它，但内存空间不够时就会回收这部分对象。只要这个具有软引用对象没有被回收，程序就可以正常使用。因此，可以使用软引用来做缓存使用，有效减少 OOM 的出现。

弱引用（WeakReference），如果一个对象只具有弱引用，则不管内存空间够不够，当 gc 扫描到它时就会回收它。因此，弱引用也可用来作为缓存使用。

虚引用（PhantomReference），如果一个对象只具有虚引用，那么它就和没有任何引用一样，任何时候都可能被 gc 回收。虚引用主要用来跟踪对象被垃圾回收的活动。

让我们总结一下这四个兄弟的区别：

类型	被回收时间	用途	举例
强引用	不会被回收	对象正常状态	Object object = new Object();
软引用	内存不足时	缓存	很少使用
弱引用	执行垃圾回收时	缓存	WeakHashMap、TheadLocal
虚引用	任何时候	跟踪对象被垃圾回收的活动	Cleaner

软（弱、虚）引用通常和一个引用队列（ReferenceQueue）一起使用，当 gc 回收这个软（弱、虚）引用引用的对象时，会把这个软（弱、虚）引用本身放到这个引用队列中。（先记住这句话）

好了，关于强软弱虚引用的概念就介绍到这里，通过上面 Cleaner 的源码，我们发现，Deallocator 的 run () 方法实际上是在 Cleaner 的 clean () 方法中调用的，那么，这个 clean () 方法又是在哪里调用的呢？其实，它是在 Reference 中调用的，Reference 中有一个线程会一直扫描这些软弱虚引用，可以先看我下面删减后的代码，再看其完整代码：

```
public abstract class Reference<T> {
    // 引用的对象
    private T referent;         /* Treated specially by GC */
    // 引用队列
    volatile ReferenceQueue<? super T> queue;

    @SuppressWarnings("rawtypes")
    volatile Reference next;

    // JVM内部使用，当引用的对象被gc清理时，放到这里
    transient private Reference<T> discovered;  /* used by VM */

    static private class Lock { }
    private static Lock lock = new Lock();

    // 通过discover来为其赋值
    private static Reference<Object> pending = null;

    // 处理线程
    private static class ReferenceHandler extends Thread {
        // 省略部分代码

        public void run() {
            // 死循环
            while (true) {
                tryHandlePending(true);
            }
        }
    }

    static boolean tryHandlePending(boolean waitForNotify) {
        Reference<Object> r;
        Cleaner c;
```

```
            try {
                synchronized (lock) {
                    if (pending != null) {
                        r = pending;
                        // 判断是否为Cleaner对象
                        c = r instanceof Cleaner ? (Cleaner) r : null;
                        // pending从discovered赋值而来
                        pending = r.discovered;
                        r.discovered = null;
                    } else {
                        if (waitForNotify) {
                            lock.wait();
                        }
                        return waitForNotify;
                    }
                }
            } catch (OutOfMemoryError x) {
                Thread.yield();
                return true;
            } catch (InterruptedException x) {
                return true;
            }

            // 如果是Cleaner对象，则执行其clean()方法
            if (c != null) {
                c.clean();
                return true;
            }

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

        // 在静态块中创建上述线程
        static {
            ThreadGroup tg = Thread.currentThread().getThreadGroup();
            for (ThreadGroup tgn = tg;
                 tgn != null;
                 tg = tgn, tgn = tg.getParent());
            // 创建这个线程
            Thread handler = new ReferenceHandler(tg, "Reference Handler");
            handler.setPriority(Thread.MAX_PRIORITY);
            // 守护线程
            handler.setDaemon(true);
            // 启动线程
            handler.start();

            SharedSecrets.setJavaLangRefAccess(new JavaLangRefAccess() {
                @Override
                public boolean tryHandlePendingReference() {
                    return tryHandlePending(false);
                }
            });
        }

        // 省略其他代码

    }
```

通过上面的分析，我们总结一下虚引用及其引用的对象被清理的过程：

1. 引用的对象被 gc 清理；
2. 虚引用进入 discovered 队列；
3. discovered 队列中的虚引用被赋值给 pending 队列；

4. 如果虚引用是 Cleaner，则执行其 clean () 方法；
5. 如果引用队列不为空，则进入引用队列 ReferenceQueue；

比如，我们拿 DirectByteBuffer 为例，来描述一下整个过程：

1. 首先，DirectByteBuffer 本身是一个堆内存中的对象，它里面有一个属性叫作 address，address 保存的是直接内存的地址，操作 DirectByteBuffer 的时候实际上是对 address 指向地址的操作，当然，这种操作是通过 unsafe 来执行的；
2. 其次，Cleaner 是一个虚引用，它引用的对象是 DirectByteBuffer，并注册了 Deallocator 这个任务，是通过 `Cleaner.create(this, new Deallocator(base, size, cap))` 这行代码实现的；
3. 再次，当 DirectByteBuffer 不具有强引用时，随时都可能被 gc 从堆内存清理掉，此时，JVM 会把上面绑定的 Cleaner 对象放到 Reference 的 discovered 队列上；
4. 然后，Reference 中的线程 ReferenceHandler 不断轮循，把 discovered 队列中的虚引用赋值到 pending 队列中，并且，这个虚引用如果是 Cleaner 对象，会执行它的 clean () 方法，且会把这个虚引用加入到 ReferenceQueue 队列中；
5. 最后，执行 clean () 方法的时候将会执行到 Deallocator 的 run () 方法，在这里调用 unsafe 的 freeMemory () 清理掉直接内存；

整个过程就是这样，比较绕，且牵涉到很多虚（软弱）引用相关的知识点，望多多体会。

其实，总结来说，在 DirectByteBuffer 的使用过程中，直接内存的回收还是 gc 控制的，只不过是一种间接控制。

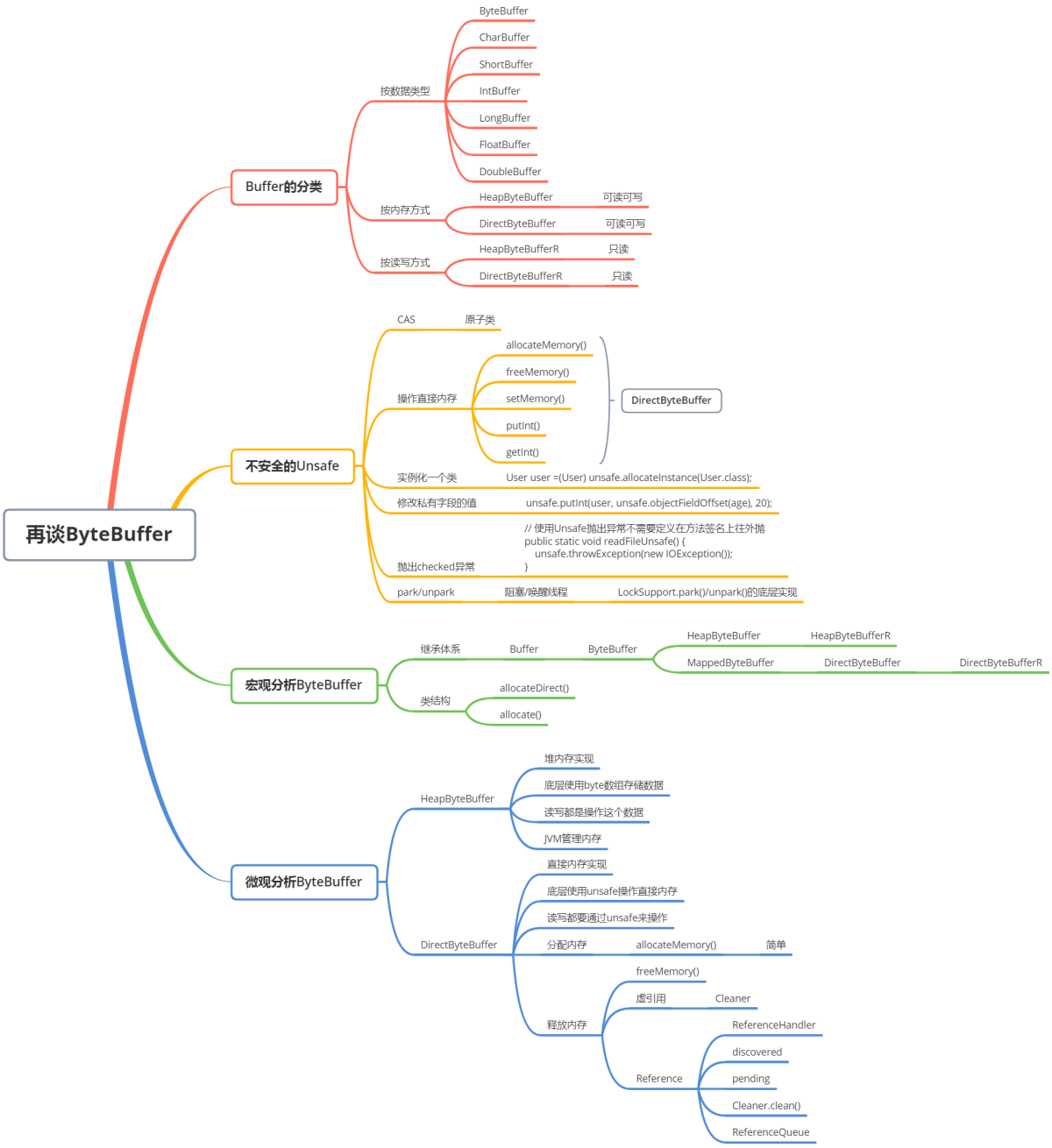
好了，到这里 DirectByteBuffer 的整个源码就剖析完成了，你有没有 Get 到呢？

后记

本节，我们从宏观和微观两个角度剖析了 ByteBuffer 在 Java 中的实现方式，并从源码层面对 HeapByteBuffer 和 DirectByteBuffer 做了非常深入的挖掘，特别是 DirectByteBuffer，它牵涉到很多 Java 中的高阶知识，相信通过本节的学习，你一定能够见识到很多未曾见过的知识，并且会发现很多自己感兴趣的点，比如大端法小端法、Unsafe、强软弱虚引用等，如果你对哪个点特别感兴趣，请死磕到底。

既然 Java 原生的 DirectByteBuffer 都已经这么牛 X 了，Netty 还能对它做出哪些更牛 X 的改造呢？下一节，我们不见不散。

思维导图



}

16 Netty服务如何优雅关闭

18 Netty的ByteBuf是如何支持堆内存非池化实现的