18 Netty的ByteBuf是如何支持堆内存非池化实现的

更新时间：2020-08-03 09:47:07

> 虚心使人进步，骄傲使人落后。——毛泽东

前言

你好，我是彤哥。

上一节，我们一起再次深入学习了 Java 原生的 ByteBuffer，并从源码级别对其进行了完整的剖析，ByteBuffer 从实现方式上分成 HeapByteBuffer 和 DirectByteBuffer 两种内存实现方式，HeapByteBuffer 底层使用 byte 数组存储数据，DirectByteBuffer 底层使用 unsafe 操作直接内存。通过源码的学习，我们还进一步掌握了很多 Java 中的高阶知识，有了很大的收获。

如果说 Java 中 ByteBuffer 的实现方式用精巧来形容，那么，Netty 中的 ByteBuf 可以用精妙来形式，它有哪些妙之处呢？我想用一词概括它的精妙之处 —— 高效易用。

问题

通过上一节的学习，我们知道，Java 原生的 Buffer 有三种不同的分类方式：按数据类型、按内存实现方式、按读写方式，可以分成不同的 Buffer。那么，今天，我想问：

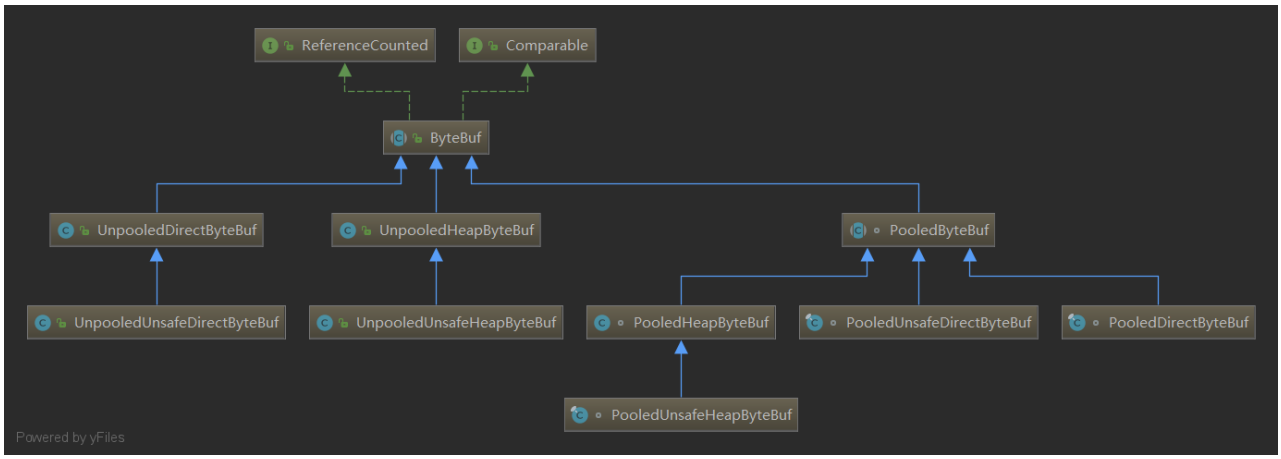
1. Netty 中的 ByteBuf 有没有兄弟类呢，即 CharBuf 等？
2. Netty 中的 ByteBuf 有哪些分类方式？
3. Netty 中的 ByteBuf 各种实现方式的底层原理是什么？

掌握这几个问题，相信你一定对 ByteBuf 有一个全面的认识。

今天的学习，我们还是遵循上一节的方法 —— 从宏观到微观，一步一步深入剖析，好了，让我们进入今天的学习吧。

宏观分析 ByteBuf

继承体系



从继承体系上看，ByteBuf 有八个主要实现类，这八个实现类又可以按三个维度来划分：

1. 内存实现方式：Heap 和 Direct；
2. 池化与否：Pooled 和 Unpooled；
3. Unsafe 与否：Unsafe 和 Safe（类名不带 Unsafe 的即为 Safe）；

> ByteBuf 并没有类似于 CharBuf、IntBuf 这样的兄弟类。

经过上一节的学习，内存实现方式这个维度我们都比较熟悉了，另外两个维度对于我们可能就比较懵了：

1. 池化，这里的池跟线程池、数据库连接池是一样的概念吗？ByteBuf 如何池化？
2. Unsafe，这里的 Unsafe 跟 Java 原生的 Unsafe 有关系吗？上一节不是说 DirectByteBuffer 底层就是通过 Unsafe 操作直接内存实现的吗？这里的 Unsafe 又是几个意思？

先把这些问题记录下来，一步一步来，我们再来看看 ByteBuf 这个类的结构。

类结构

类结构主要包含字段和方法，ByteBuf 作为一个抽象类，没有任何字段，大致浏览一下它的方法，主要可以分成三大类：

1. writeXxx ()/readXxx ()：写入、读取数据
2. setXxx ()/getXxx ()：根据索引写入、读取数据
3. 其它：返回 ByteBuf 的状态、遍历等方法，比如返回容量、是否可读等方法

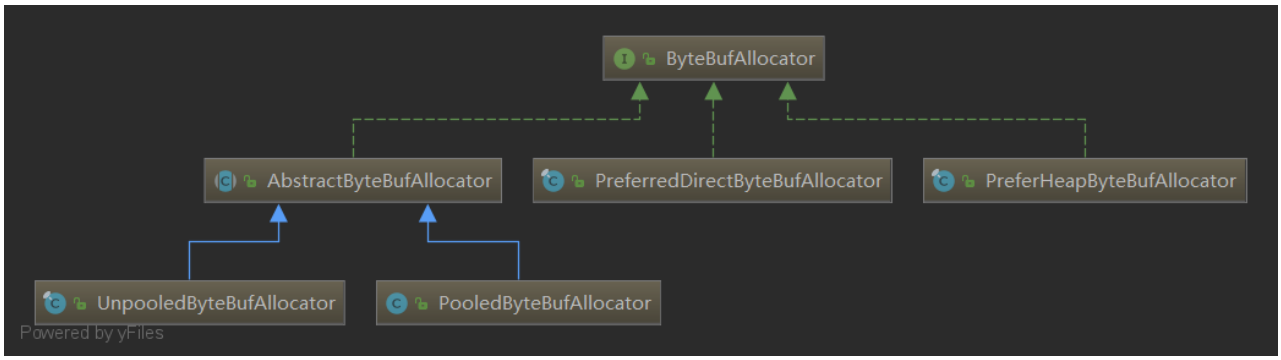
在这些方法中，有一个方法比较亮眼，即：

```
public abstract ByteBufAllocator alloc();
```

OK，这是我们找到的一个突破口，从类名 ByteBufAllocator 可以看出它是 ByteBuf 的分配器，什么是分配器呢？它类似于我们常说的工厂模式，用来创建 ByteBuf 的地方。

此时，我们需要简单浏览一下 ByteBufAllocator 的继承体系和类结构，即宏观分析一下。

ByteBufAllocator 的继承体系



我们可以把上图分解成左右两个部分来看，左半边是以 Pooled/Unpooled 为划分标准，右半边是以 Heap/Direct 为划分标准，这是跟 ByteBuf 本身的分类方式差不多嘛？实际上是不是这样呢？

很遗憾，并不是这样，再仔细观察一下右边两个类的名字，一个是以 Preferred 开头，一个是以 Prefer 开头，可见这两个类不是出自同一人之手，或者不是同一人同一时间写的。

通过这个切入点，再仔细观察一下，会发现 UnpooledByteBufAllocator 和 PooledByteBufAllocator 位于核心库 netty-buffer 中，而 PreferHeapByteBufAllocator 位于核心库 netty-transport 中，而 PreferredDirectByteBufAllocator 位于库 transport-native-unix-common 中。

再大概浏览一下四个类中的代码，可以发现，UnpooledByteBufAllocator 和 PooledByteBufAllocator 类中默认都是偏向于使用 Direct 的方式创建 ByteBuf，而 PreferHeapByteBufAllocator 和 PreferredDirectByteBufAllocator 中保存了 ByteBufAllocator 的实例，其实真正干活还是交给 UnpooledByteBufAllocator 或者 PooledByteBufAllocator 类来实现的，这是设计模式中的装饰器模式的用法。

```
// 使用池化技术
public class PooledByteBufAllocator extends AbstractByteBufAllocator implements ByteBufAllocatorMetricProvider {
    public static final PooledByteBufAllocator DEFAULT =
            new PooledByteBufAllocator(PlatformDependent.directBufferPreferred());
    // 省略其它代码
}
// 不使用池化技术
public final class UnpooledByteBufAllocator extends AbstractByteBufAllocator implements ByteBufAllocatorMetricProvider {
    public static final UnpooledByteBufAllocator DEFAULT =
            new UnpooledByteBufAllocator(PlatformDependent.directBufferPreferred());
    // 省略其它代码
}
// 更偏向于使用堆内存
public final class PreferHeapByteBufAllocator implements ByteBufAllocator {
    private final ByteBufAllocator allocator;
    // 省略其它代码
}
// 更偏向于使用直接内存
public final class PreferredDirectByteBufAllocator implements ByteBufAllocator {
    private ByteBufAllocator allocator;
    // 省略其它代码
}
```

所以，我大胆猜测，一开始 Netty 的作者只提供了 UnpooledByteBufAllocator 和 PooledByteBufAllocator 这两种创建 ByteBuf 的分配器，而 PreferHeapByteBufAllocator 和 PreferredDirectByteBufAllocator 是后面写的，且不是同一个作者写的，或者不是同一时间写的。

好了，我们再来看看 ByteBufAllocator 这个接口的结构。

ByteBufAllocator 的结构

这里我列出几个比较重要的方法，还有一个常量：

```
public interface ByteBufAllocator {
    // 默认的分配器，除非显式地配成unpooled，否则使用pooled
    ByteBufAllocator DEFAULT = ByteBufUtil.DEFAULT_ALLOCATOR;

    // 创建一个ByteBuf，看具体的实现方式决定创建的是direct的还是heap的
    ByteBuf buffer();
    ByteBuf buffer(int initialCapacity);
    ByteBuf buffer(int initialCapacity, int maxCapacity);

    // 创建一个heap类型的ByteBuf
    ByteBuf heapBuffer();
    ByteBuf heapBuffer(int initialCapacity);
    ByteBuf heapBuffer(int initialCapacity, int maxCapacity);

    // 创建一个direct的ByteBuf
    ByteBuf directBuffer();
    ByteBuf directBuffer(int initialCapacity);
    ByteBuf directBuffer(int initialCapacity, int maxCapacity);

  // 省略其它方法
  }
```

有这几个方法，完全够我们使用了，通过观察可以发现，每种方法都提供了三种重载方式，且参数的变化是跟容量相关的，所以，我们是否可以大胆猜测：其实，ByteBuf 是支持扩容的呢？

支不支持扩容呢？下面进入我们的微观分析阶段。

微观分析 ByteBuf

通过宏观分析，我们知道 ByteBuf 有 8 个主要实现类，我们又知道，可以通过 ByteBufAllocator 来创建不同类型的 ByteBuf，但是，并没有看到跟 Unsafe 相关的代码，这是一个问题，等待我们去挖掘。

所以，我们先从简单的 Unpooled 和 Heap 入手，这样怎么写调试用例呢？其实很简单，请看：

```
public class ByteBufTest {
    public static void main(String[] args) {
        // 1. 参数是preferDirect，即是否偏向于使用直接内存
        UnpooledByteBufAllocator allocator = new UnpooledByteBufAllocator(false);

        // 2. 创建一个非池化基于堆内存的ByteBuf
        ByteBuf byteBuf = allocator.heapBuffer();

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

        // 4. 读取数据
        System.out.println(byteBuf.readInt());
        System.out.println(byteBuf.readInt());
        System.out.println(byteBuf.readInt());
    }
}
```

这个调试用例，我将分成 4 个部分来进行源码级别的剖析：

1. 创建 ByteBufAllocator 的过程；
2. 创建 heapByteBuf 的过程；
3. 写入数据的过程；
4. 读取数据的过程；

首先，让我们看看创建 ByteBufAllocator 的过程：

```
// 参数1preferDirect表示是否偏向于使用直接内存
// 这里我们传的是false
public UnpooledByteBufAllocator(boolean preferDirect) {
    this(preferDirect, false);
}
// 参数2表示是否禁用内存泄漏检测，先跳过
public UnpooledByteBufAllocator(boolean preferDirect, boolean disableLeakDetector) {
    this(preferDirect, disableLeakDetector, PlatformDependent.useDirectBufferNoCleaner());
}
// 参数3表示是否尝试没有Cleaner的构造方法，这个是什么意思呢？
public UnpooledByteBufAllocator(boolean preferDirect, boolean disableLeakDetector, boolean tryNoCleaner) {
    super(preferDirect);
    this.disableLeakDetector = disableLeakDetector;
    noCleaner = tryNoCleaner && PlatformDependent.hasUnsafe()
        && PlatformDependent.hasDirectBufferNoCleanerConstructor();
}
// 调用父类的构造方法
protected AbstractByteBufAllocator(boolean preferDirect) {
    // 如果偏向于直接内存且平台有Unsafe，则默认使用直接内存
    // 这里的Unsafe是Java原生的Unsafe吗？
    directByDefault = preferDirect && PlatformDependent.hasUnsafe();
    emptyBuf = new EmptyByteBuf(this);
}
```

这段代码看着是不是很费劲？那就对了。这段代码主要是用来判断当前运行的 JDK 平台是否支持相关的功能，比如直接内存，并不是说这里传了 `preferDirect=true`，最后就一定会创建直接内存形式的 ByteBuf 给到我们。说句不好听的，你喜欢是你的事，支不支持是我的事。

为什么会有平台是否支持的判断呢？归纳起来主要有三个原因：

1. JDK 各版本之间并不完全兼容。比如低版本的 JDK 可能就不支持使用直接内存。

2. 不同的设备支持的功能不一样。我们知道，Java 程序是 write once, run anywhere，这其实是各个平台级别的 JDK 底层给我们做了很多兼容，有的平台可能就不支持直接内存，比如安卓可能就不支持直接内存。
3. 不同类型的 JDK 支持的功能不一样。目前比较主流的有两大阵营 ——OracleJDK 和 OpenJDK，OracleJDK 是功能最齐全的，OpenJDK 相对来说会缺失一些功能。

所以，Netty 把这些细节都给我们考虑在内了，它会检测当前运行的平台是否支持某些功能，然后做出最好的优化让程序可以继续运行。比如，我们如果直接使用 Java 原生的方法创建了一个 DirectByteBuffer，可能换一个平台就无法运行，但是，使用 Netty 不会出现这种情况，它可能会退化成 HeapByteBuffer，让程序可以继续运行。如果这部分判断让我们自己来写呢，那肯定是不现实的，我们不可能把每个平台不同设备不同类型的 JDK 全部研究一遍。

然而，它是怎么检测的呢？比如 PlatformDependent.hasUnsafe() 这个方法内部到底做了哪些判断呢？让我们一起来看一看：

```
public static boolean hasUnsafe() {
    return UNSAFE_UNAVAILABILITY_CAUSE == null;
}
```

这里只是简单地判断了一下 UNSAFE_UNAVAILABILITY_CAUSE 是否为 null，这个变量又是什么？

在 IDEA 中，按住 CTRL 键鼠标点击一下，可以定位到这个变量声明的地方：

```
public final class PlatformDependent {
    private static final Throwable UNSAFE_UNAVAILABILITY_CAUSE = unsafeUnavailabilityCause0();
 // 省略其它代码
}
```

OK，可以看到，这里是通过一个方法返回的，跟踪进去：

```
// 返回不支持Unsafe的原因
private static Throwable unsafeUnavailabilityCause0() {
    // 如果是安卓，直接抛出不支持的异常
    if (isAndroid()) {
        logger.debug("sun.misc.Unsafe: unavailable (Android)");
        return new UnsupportedOperationException("sun.misc.Unsafe: unavailable (Android)");
    }

    // 如果是IKVM.NET，直接抛出不支持的异常
    if (isIkvmDotNet()) {
        logger.debug("sun.misc.Unsafe: unavailable (IKVM.NET)");
        return new UnsupportedOperationException("sun.misc.Unsafe: unavailable (IKVM.NET)");
    }

    // 通过PlatformDependent0.getUnsafeUnavailabilityCause()判断是否支持Unsafe
    Throwable cause = PlatformDependent0.getUnsafeUnavailabilityCause();
    // 如果有原因，直接返回
    if (cause != null) {
        return cause;
    }

    try {
        // 再判断是否有Unsafe
        boolean hasUnsafe = PlatformDependent0.hasUnsafe();
        logger.debug("sun.misc.Unsafe: {}", hasUnsafe ? "available" : "unavailable");
        return hasUnsafe ? null : PlatformDependent0.getUnsafeUnavailabilityCause();
    } catch (Throwable t) {
        logger.trace("Could not determine if Unsafe is available", t);
        // Probably failed to initialize PlatformDependent0.
        return new UnsupportedOperationException("Could not determine if Unsafe is available", t);
    }
}
```

最后，到 `PlatformDependent0` 这个类中，实际上，是在它的静态块中进行判断的（删减了许多代码）：

```
// io.netty.util.internal.PlatformDependent0#getUnsafeUnavailabilityCause
static Throwable getUnsafeUnavailabilityCause() {
    // 这个变量是在下面这个静态块中赋值的
    return UNSAFE_UNAVAILABILITY_CAUSE;
}
static {
    // 是否显式地关闭unsafe，通过参数io.netty.noUnsafe控制
    if ((unsafeUnavailabilityCause = EXPLICIT_NO_UNSAFE_CAUSE) != null) {
        direct = null;
        addressField = null;
        unsafe = null;
        internalUnsafe = null;
    } else {
        // 创建一个Java原生的DirectByteBuffer后面使用
        direct = ByteBuffer.allocateDirect(1);

        // 尝试获取Unsafe的字段theUnsafe
        final Field unsafeField = Unsafe.class.getDeclaredField("theUnsafe");

        // 尝试获取copyMemory()方法
        finalUnsafe.getClass().getDeclaredMethod(
            "copyMemory", Object.class, long.class, Object.class, long.class, long.class);

        // 尝试获取DirectByteBuffer中的address字段（实际是在其父类Buffer中）
        // 这个address是专门用来存储直接内存的地址的
        final Field field = Buffer.class.getDeclaredField("address");
        final long offset = finalUnsafe.objectFieldOffset(field);
        final long address = finalUnsafe.getLong(direct, offset);

        // 上面方法的返回值
        UNSAFE_UNAVAILABILITY_CAUSE = unsafeUnavailabilityCause;
    }
    // 省略其它代码
}
```

首先，会检测我们是否显式地关闭了 Unsafe，即通过参数 `io.netty.noUnsafe` 控制的，其实，Netty 中很多场景都是可以通过参数显式地控制的，但是，一般我们也没有必要去修改默认配置，因为 Netty 给我们的默认配置已经足够好了。

然后，会尝试反射访问 Unsafe 中的属性 `theUnsafe` 和方法 `copyMemory`，其中，`theUnsafe` 是我们获得 Unsafe 实例的唯一方法，因为这个类是 Java 核心类，有非常严格的权限控制，我们只能通过这种方式获得其实例。

最后，会反射获取 DirectByteBuffer 中的 `address` 属性，这个 `address` 是定义在其父类 Buffer 中的。

如果上面三步都成功了，才能宣判我们可以正确地使用 Unsafe 了，当然了，这个 Unsafe 就是 Java 原生的那个 Unsafe。

另外，前面看到的 `PlatformDependent.hasDirectBufferNoCleanerConstructor()` 最终也是在这个静态块中判断的，原理都差不多，无非是通过反射判断某个方法或者属性存不存在，有兴趣的同学可以研究下。

好了，接下来，我们来看看创建 HeapByteBuffer 的过程吧，即 `ByteBuf byteBuf = allocator.heapBuffer();`：

```
@Override
public ByteBuf heapBuffer() {
    // DEFAULT_INITIAL_CAPACITY = 256
    // DEFAULT_MAX_CAPACITY = Integer.MAX_VALUE = 2147483647
    return heapBuffer(DEFAULT_INITIAL_CAPACITY, DEFAULT_MAX_CAPACITY);
}
@Override
public ByteBuf heapBuffer(int initialCapacity, int maxCapacity) {
    if (initialCapacity == 0 && maxCapacity == 0) {
        return emptyBuf;
    }
    // 检查这两个容量是否合规
    validate(initialCapacity, maxCapacity);
    return newHeapBuffer(initialCapacity, maxCapacity);
}
@Override
protected ByteBuf newHeapBuffer(int initialCapacity, int maxCapacity) {
    // 根据是否可以使用Unsafe创建不同的类型
    return PlatformDependent.hasUnsafe() ?
        new InstrumentedUnpooledUnsafeHeapByteBuf(this, initialCapacity, maxCapacity) :
    new InstrumentedUnpooledHeapByteBuf(this, initialCapacity, maxCapacity);
}
```

看到这里你可能会有一些疑问：

1. 原来 Unsafe 是在最后一步这里才真正用到，但是，Unsafe 不是用来操作直接内存的吗？我们这里创建的是基于堆内存的，它有什么意义呢？
2. 为什么创建的不是我们上面介绍的 8 种类型呢，而是它们的子类？
3. 这里有两个容量，难道 Netty 中的 ByteBuf 可以支持扩容？

OK，我们先回答第二个问题，为什么创建的是 InstrumentedUnpooledHeapByteBuf 或者 InstrumentedUnpooledUnsafeHeapByteBuf 类的对象呢？

首先，我们要理解 Instrumented 这个单词的含义，直接查它的意思是指乐器、仪器、工具，这个词更偏向于指法律、艺术、科学方面的工具或者器械，不过，这些可能还是跟我们这里的意思没有太大关系，最后，我找到了一段对这个单词的英文解释：

> the semantic role of the entity (usually inanimate) that the agent uses to perform an action or start a process

正所谓是，每个单词都认识，连一起就都不认识了，其实，也不用全认识，认识一个单词即可 ——agent—— 代理的意思，这个代理跟 Proxy 比较类似，不过有一点区别，像我们常用的代理（Proxy）都是方法级别的，而这个代理（Agent/Instrument）一般是指类级别的代理，它会代理整个类。好吧，还是比较懵，那就当成一样的吧也无所谓。

> 其实，在 Java 和 Spring 中也分别有 Instrument 一说，在 Spring 中，也可以把它看成是 AOP 的一种方式，有兴趣的同学可以面向搜索引擎学习一下。

问题又来了，为什么要使用 Instrument 做一层代理呢？那肯定是对原有功能进行了一些增强，至于做了哪些增强，我们接着看。

因为我使用的是 win8 JDK8，天生就是支持 Unsafe 的，所以，在我的电脑上创建的就是 InstrumentedUnpooledUnsafeHeapByteBuf 的对象，让我们继续跟下去：

```
InstrumentedUnpooledUnsafeHeapByteBuf(UnpooledByteBufAllocator alloc, int initialCapacity, int maxCapacity) {
    // 调用父类的构造方法
    super(alloc, initialCapacity, maxCapacity);
}
public UnpooledUnsafeHeapByteBuf(ByteBufAllocator alloc, int initialCapacity, int maxCapacity) {
    // 调用父类的构造方法
    super(alloc, initialCapacity, maxCapacity);
}
public UnpooledHeapByteBuf(ByteBufAllocator alloc, int initialCapacity, int maxCapacity) {
    // 这个父类构造方法里没什么重要的东西，不断续往里跟了
    super(maxCapacity);

    // 检查两个容量
    if (initialCapacity > maxCapacity) {
        throw new IllegalArgumentException(String.format(
            "initialCapacity(%d) > maxCapacity(%d)", initialCapacity, maxCapacity));
    }

    this.alloc = checkNotNull(alloc, "alloc");
    // 调用allocateArray创建了一个byte数组，并保存起来
    setArray(allocateArray(initialCapacity));
    // 初始化readIndex和writeIndex为0
    setIndex(0, 0);
}
// InstrumentedUnpooledUnsafeHeapByteBuf#allocateArray
// 这个方法就是做了增强的方法
@Override
protected byte[] allocateArray(int initialCapacity) {
    byte[] bytes = super.allocateArray(initialCapacity);
    // 增强的地方
    ((UnpooledByteBufAllocator) alloc()).incrementHeap(bytes.length);
    return bytes;
}
```

到这里就比较清楚了，UnpooledUnsafeHeapByteBuf 底层还是使用的 Java 原生的 byte 数组来实现的，至于这里增强的方法，其实是加入了监控，打开 InstrumentedUnpooledUnsafeHeapByteBuf 这个类，你会发现，它里面还有另一个方法叫作 freeArray ()，它做的增强即：当创建 byte 数组的时候记录下来分配的堆内存大小，当释放 byte 数组的时候将其占用的堆内存大小相应减少。这样就起来了监控的目的，即 Netty 可以监控进程中所有的 UnpooledUnsafeHeapByteBuf 到底占用了多少堆内存，方便出问题时进行排查，当然，也可以用于提前发现内存泄漏等问题。

OK，经过前面的过程，我们终于创建了一个 UnpooledUnsafeHeapByteBuf 对象，接下来我们再来看看写入数据的过程吧，即 `byteBuf.writeInt(1);` 这行代码，继续跟踪进去：

```
@Override
public ByteBuf writeInt(int value) {
    // 一个int等于4个字节
    // 检查是否可写，里面会做扩容相关的操作，
    // 最终会调用allocateArray()分配一个新的数组，并把旧数组的数据拷贝到新数组
    // 且调用freeArray()把旧数组释放，当然，此时也会改变上面提到的监控的数值
    ensureWritable0(4);
    // 在写索引的位置开始写入值
    _setInt(writerIndex, value);
    // 写索引加4
    writerIndex += 4;
    return this;
}
@Override
protected void _setInt(int index, int value) {
    // 调用UnsafeByteBufUtil工具类修改array数组index位置的值
    UnsafeByteBufUtil.setInt(array, index, value);
}
static void setInt(byte[] array, int index, int value) {
    // 我的电脑UNALIGNED=true
    if (UNALIGNED) {
        // 又到了PlatformDependent这个类中
        PlatformDependent.putInt(array, index, BIG_ENDIAN_NATIVE_ORDER ? value : Integer.reverseBytes(value));
    } else {
        PlatformDependent.putByte(array, index, (byte) (value >>> 24));
        PlatformDependent.putByte(array, index + 1, (byte) (value >>> 16));
        PlatformDependent.putByte(array, index + 2, (byte) (value >>> 8));
        PlatformDependent.putByte(array, index + 3, (byte) value);
    }
}
public static void putInt(byte[] data, int index, int value) {
    // 继续到PlatformDependent0这个类中
    PlatformDependent0.putInt(data, index, value);
}
static void putInt(byte[] data, int index, int value) {
    // 调用Unsafe的putInt()方法直接修改对象的属性
    // 数组本身就是一种特殊的对象，它也有对象头等属性
    // 所以，需要一个偏移量，BYTE_ARRAY_BASE_OFFSET=16
    UNSAFE.putInt(data, BYTE_ARRAY_BASE_OFFSET + index, value);
}
// native方法
// 这个putInt()跟上一节修改直接内存的putInt()不是同一个方法
// 上一节的putInt(long var1, int var3)第一个参数是内存地址
// 本节的putInt()第一个参数是对象，第二参数是在对象中的偏移量
public native void putInt(Object var1, long var2, int var4);
```

好了，到这里就比较清晰了，UnpooledUnsafeHeapByteBuf 底层是使用的 Unsafe 来修改数组中的值，为了与 Java 原生 HeapByteBuffer 对比，我们把上一节 HeapByteBuffer 写入数据的最终方法拿过来再对比一下：

```
// 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;
}
// 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;
}
```

可以看到，Java 原生的 HeapByteBuffer 并没有使用 Unsafe，而是把一个 int 类型拆分成了 4 个 byte 类型，再分别修改 byte 数组对应位置的值，而 Netty 中是直接使用 Unsafe 来修改 byte 数组的值，它是直接修改那一片内存区域的值，不需要拆分等操作，所以，相对来说，更高效一些。

> 其实呢，Netty 中也有 Java 这样的实现，即不带 Unsafe 的 UnpooledHeapByteBuf 类，它的底层就是像 Java 这样把一个 int 拆分成 4 个 byte，并修改数组下标的方式来实现的，有兴趣的同学可以自己看看相关的源码。

同样地，我们可以想像得到，读取数据的过程肯定也是使用 Unsafe 来操作的，我们直接给出代码，就不赘述了：

```
@Override
public int readInt() {
    // 检查可读
    checkReadableBytes0(4);
    // 在读索引的位置读取数据
    int v = _getInt(readerIndex);
    // 读索引加4
    readerIndex += 4;
    return v;
}
@Override
protected int _getInt(int index) {
    // 调用UnsafeByteBufUtil工具类
    return UnsafeByteBufUtil.getInt(array, index);
}
static int getInt(byte[] array, int index) {
    // 在我的电脑UNALIGNED=true
    if (UNALIGNED) {
        // 调用PlatformDependent
        int v = PlatformDependent.getInt(array, index);
        return BIG_ENDIAN_NATIVE_ORDER ? v : Integer.reverseBytes(v);
    }
    return PlatformDependent.getByte(array, index) << 24 |
        (PlatformDependent.getByte(array, index + 1) & 0xff) << 16 |
        (PlatformDependent.getByte(array, index + 2) & 0xff) <<  8 |
        PlatformDependent.getByte(array, index + 3) & 0xff;
}
public static int getInt(byte[] data, int index) {
    // 调用PlatformDependent0
    return PlatformDependent0.getInt(data, index);
}
static int getInt(byte[] data, int index) {
    // 调用Unsafe的getInt()方法
    return UNSAFE.getInt(data, BYTE_ARRAY_BASE_OFFSET + index);
}
// 第一个参数是对象，第二个参数是在对象中的偏移量
public native int getInt(Object var1, long var2);
```

好了，到这里对于 UnpooledUnsafeHeapByteBuf 的源码级别的剖析就结束了。

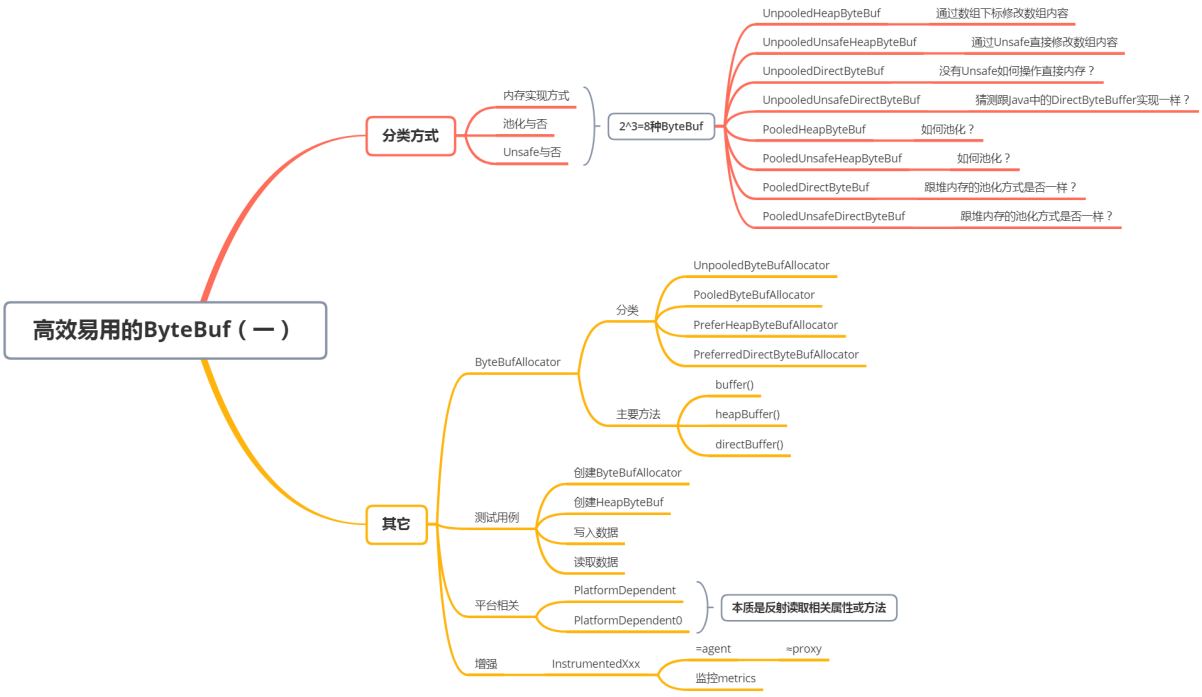
后记

本节，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，到这里，你可能还有一些疑问的，比如，Netty 是怎么利用直接内存的，等等。嗯，有问题是好事，别急，我们一步一步来，先从简单的入手，再慢慢过渡到更难的级别。

下一节，我们将一起学习 Netty是如何使用直接内存的 这个问题，不见不散。

思维导图

本次的思维导图和以往不太一样，大家可以在分类方式后面不断联想并提出一些问题。



}

17 如何从源码的角度深入剖析ByteBuffer

19 Netty的ByteBuf是如何支持直接内存非池化实现的