21 Netty的内存池是如何实现的

更新时间：2020-08-06 09:49:54

> 人的影响短暂而微弱，书的影响则广泛而深远。——普希金

前言

你好，我是彤哥。

上一节，我们一起学习了Netty中jemalloc的基本概念，并从原理层面对jemalloc做了一定的解析，本节，我们将从源码层面对Netty的内存池做一个全面的剖析。

因为内存池跟对象池往往纠缠在一起，所以，关于对象池相关的代码本节一律先跳过，免得把你弄晕。

好了，让我们开始今天的学习吧。

问题

对于今天的源码剖析，你可以带着下面这么几个问题：

1. PoolArena中的PoolSubpage数组和PoolChunk中的PoolSubpage数组有什么关联？
2. PoolThreadCache中的MemoryRegionCache数组与PoolSubpage是否有联系？
3. Netty内存池的整体结构是什么样的？
4. 如果让你来介绍Netty内存池，你如何来描述？

内存池源码剖析

调试用例

上一节，我们说了Netty根据请求的大小将其分成四类：Tiny、Small、Normal、Huge，这四类请求的分界线分别为512B、8KB、16MB，针对这四类请求，Netty的处理逻辑并不一样，但是，本节，我们并不打算把这四种全部讲解一遍，我们只讲解其中的一个——Tiny。

另外，为了加快分配内存的速度，Netty还使用了线程缓存，而线程缓存实际上是在有回收内存的情况下才有效，所以，我们设计的调试用例里面还应该包括回收内存的部分，即 `ReferenceCountUtil.release(byteBuf)` 。

好了，让我们看看今天的调试用例吧：

```
public class ByteBufTest {
    public static void main(String[] args) {
        // 1. 创建池化的分配器
        ByteBufAllocator allocator = new PooledByteBufAllocator(false);
        // 2. 分配一个40B的ByteBuf
        ByteBuf byteBuf = allocator.heapBuffer(40);
        // 3. 写入数据
        byteBuf.writeInt(4);
        // 4. 读取数据
        System.out.println(byteBuf.readInt());
        // 5. 回收内存
        ReferenceCountUtil.release(byteBuf);
        // 6. 分配一个30B的ByteBuf
        ByteBuf byteBuf2 = allocator.heapBuffer(30);
        // 7. 再次分配一个40B的ByteBuf
        ByteBuf byteBuf3 = allocator.heapBuffer(40);
    }
}
```


我们先创建一个池化的分配器，使用其分配一个40B的ByteBuf，接着写入数据并读取数据，然后回收，理论上来说，这个40B的ByteBuf会进入线程缓存，接着再分配一个30B的ByteBuf，观察其是否会从线程缓存中获取，最后再分配一个40B的ByteBuf，观察其是否会从线程缓存中获取。

源码调试

本节的代码有点多，有些不太重要的我就直接跳过去了，希望大家能够亲自动手调试一下整个过程。

1. 创建池化的分配器

让我们先在创建分配器那行打个断点，跟踪进去：

```
public PooledByteBufAllocator(boolean preferDirect, int nHeapArena, int nDirectArena,
                              int pageSize, int maxOrder,
                              int tinyCacheSize, int smallCacheSize, int normalCacheSize,
                              boolean useCacheForAllThreads, int directMemoryCacheAlignment) {
    super(preferDirect);
    // 这是一个ThreadLocal，用于存放线程缓存PoolThreadCache对象
    threadCache = new PoolThreadLocalCache(useCacheForAllThreads);
    // Tiny的缓存个数，默认512
    this.tinyCacheSize = tinyCacheSize;
    // Small的缓存个数，默认256
    this.smallCacheSize = smallCacheSize;
    // Normal的缓存个数，默认64
    this.normalCacheSize = normalCacheSize;
    // pageSize默认为8KB
    // maxOrder表示树的最大高度，默认为11
    // 计算每个Chunk的大小，默认为16MB
    chunkSize = validateAndCalculateChunkSize(pageSize, maxOrder);

    // ...

    // 计算每页大小的左移位数
    // pageSize=8KB=10 0000 0000 0000=1<<13
    // pageShifts表示pageSize是1左移13位得来的
    // 所以pageShifts为13
    int pageShifts = validateAndCalculatePageShifts(pageSize);

    // 初始化堆内存池heapArenas
    // 默认为CPU核数*2，同时根据堆内存大小调整
    if (nHeapArena > 0) {
        heapArenas = newArenaArray(nHeapArena);
        List<PoolArenaMetric> metrics = new ArrayList<PoolArenaMetric>(heapArenas.length);
        for (int i = 0; i < heapArenas.length; i ++) {
            // 创建HeapArena并加入到数组中
            PoolArena.HeapArena arena = new PoolArena.HeapArena(this,
                                                                pageSize, maxOrder, pageShifts, chunkSize,
                                                                directMemoryCacheAlignment);
            heapArenas[i] = arena;
            // 同时加入监控
            metrics.add(arena);
        }
        heapArenaMetrics = Collections.unmodifiableList(metrics);
    } else {
        heapArenas = null;
        heapArenaMetrics = Collections.emptyList();
    }

    // 初始化直接内存池directArenas
    // 默认为CPU核数*2，同时根据直接内存大小调整
    if (nDirectArena > 0) {
        directArenas = newArenaArray(nDirectArena);
        List<PoolArenaMetric> metrics = new ArrayList<PoolArenaMetric>(directArenas.length);
        for (int i = 0; i < directArenas.length; i ++) {
            // 创建DirectArena并加入到数组中
            PoolArena.DirectArena arena = new PoolArena.DirectArena(
                this, pageSize, maxOrder, pageShifts, chunkSize, directMemoryCacheAlignment);
            directArenas[i] = arena;
            // 同时加入监控
            metrics.add(arena);
        }
        directArenaMetrics = Collections.unmodifiableList(metrics);
    } else {
        directArenas = null;
        directArenaMetrics = Collections.emptyList();
    }
    // 监控
    metric = new PooledByteBufAllocatorMetric(this);
}
```

在PooledByteBufAllocator的构造方法中，主要是初始化一些属性，比如chunkSize、heapArenas和directArenas数组等，默认地，chunkSize为16M，heapArenas和directArenas数组大小为2倍的CPU核数，同时根据内存大小动态调整。

2. 分配一个40B的ByteBuf

好了，上面我们已经创建好了一个池化的分配器，下面就来看看它是如何为我们分配一个40B的ByteBuf的，继续调试：

```
// PooledByteBufAllocator#newHeapBuffer
@Override
protected ByteBuf newHeapBuffer(int initialCapacity, int maxCapacity) {
    // 先从ThreadLocal中获取一个PoolThreadCache线程缓存对象
    PoolThreadCache cache = threadCache.get();
    // 因为我们创建的是基于堆内存的ByteBuf，所以使用heapArena
    // 另外，在PoolThreadCache初始化的时候会绑定一个最少使用的heapArena
    // 所以这里是可以获取到的
    // 具体可参考PooledByteBufAllocator.PoolThreadLocalCache#initialValue()方法
    PoolArena<byte[]> heapArena = cache.heapArena;

    final ByteBuf buf;
    if (heapArena != null) {
        // 使用heapArena来分配
        buf = heapArena.allocate(cache, initialCapacity, maxCapacity);
    } else {
        buf = PlatformDependent.hasUnsafe() ?
            new UnpooledUnsafeHeapByteBuf(this, initialCapacity, maxCapacity) :
            new UnpooledHeapByteBuf(this, initialCapacity, maxCapacity);
    }

    return toLeakAwareBuffer(buf);
}
```


在这段代码中，首先从threadCache中获取了一个PoolThreadCache对象cache，再从这个cache中获取一个heapArena，这个heapArena是从allocator的heapArenas数组中取的一个最少使用的，在cache初始化的时候绑定进去的，为什么绑定的是最少使用的呢？

我们知道，heapArenas数组的大小是固定的，而线程是可能无限多的，每个线程都要绑定一个heapArena，那么，怎么绑定才能减少线程之间的竞争呢？答案很明显，绑定最少使用的那个heapArena。比如，heapArenas数组大小为16，而线程也正好为16，这样的话，线程之间完全没竞争关系，如果按照数组的顺序绑定，最后这16个线程都会绑定到同一个heapArena，竞争非常剧烈。

OK，我们继续跟踪到`heapArena.allocate()`中：

```
// PoolArena#allocate(PoolThreadCache, int, int)
PooledByteBuf<T> allocate(PoolThreadCache cache, int reqCapacity, int maxCapacity) {
    // 使用对象池创建一个池化的ByteBuf，先不展开，跳过
    // PooledByteBuf中有一个memory字段用来存放数据
    // 从对象池中返回的这个buf，它的memory大小为0
    // 内存池的作用就是为了给这个memory分配空间
    PooledByteBuf<T> buf = newByteBuf(maxCapacity);
    // 继续深入
    allocate(cache, buf, reqCapacity);
    return buf;
}
// PoolArena#allocate(PoolThreadCache, PooledByteBuf<T>, int)
private void allocate(PoolThreadCache cache, PooledByteBuf<T> buf, final int reqCapacity) {
    // 将请求的大小规范化，这里40B会被规范化到48B
    final int normCapacity = normalizeCapacity(reqCapacity);
    // 判断是否小于8KB
```

```
    if (isTinyOrSmall(normCapacity)) { // capacity < pageSize
        int tableIdx;
        PoolSubpage<T>[] table;
        // 是否为Tiny，即小于512B
        boolean tiny = isTiny(normCapacity);
        if (tiny) { // < 512
            // 先尝试从线程缓存中分配内存
            if (cache.allocateTiny(this, buf, reqCapacity, normCapacity)) {
                // 分配成功则返回
                return;
            }
            // 计算48B在tinySubpagePools数组中的索引
            // 16、32、48，所以这里的索引是3
            tableIdx = tinyIdx(normCapacity);
            // PoolArena的tinySubpagePools数组，大小为32
            table = tinySubpagePools;
        } else {
            // 如果为Small，从这里走
            if (cache.allocateSmall(this, buf, reqCapacity, normCapacity)) {
                return;
            }
            tableIdx = smallIdx(normCapacity);
            table = smallSubpagePools;
        }

        // 获取头节点，head中不存放任何数据，仅用来加锁使用
        final PoolSubpage<T> head = table[tableIdx];

        // 分段锁的用法，减少锁竞争
        synchronized (head) {
            // 因为还没有分配过任何内存，所以next为空，先跳过这一段
            final PoolSubpage<T> s = head.next;
            // ...
            // 分配成功会返回
        }
        // 按PoolArena加锁，又是分段锁的用法
        // 这里就体会到了上面按最少使用去绑定PoolArena的魅力了吧
        // 减少了锁竞争
        synchronized (this) {
            // 这个方法名有点歧义
            // 并不是说按Normal的请求去处理
            allocateNormal(buf, reqCapacity, normCapacity);
        }

        incTinySmallAllocation(tiny);
        return;
    }
    // 如果是Normal请求
    if (normCapacity <= chunkSize) {
        // ...
    } else {
        // 如果是Huge请求
        allocateHuge(buf, reqCapacity);
    }
}
```


这段代码中运用了大量的分段锁技巧：

- PoolArena，每个线程绑定一个最少使用的PoolArena，充分减少锁竞争。
- tinySubpagePools，按16B分成32段，只有分配相同大小（规范后的大小）的请求时且还是两个线程绑定到同一个PoolArena的情况下才有锁竞争。

结合ConcurrentHashMap中分段锁的用法，我们归纳出来一个规律：**分段锁一般都是通过数组来实现的，通过某种规则将多个线程的操作分离到数组中不同的位置上，以便降低线程之间修改同一段数组的竞争，所以，有时候也叫作锁分离。**

比如，在ConcurrentHashMap中，是根据key值hash到不同的桶中进行处理。在PoolArena中，是通过线程绑定最小使用PoolArena来达到锁分离的目的。在tinySubpagePools中，是通过将请求分割成不同的段进行处理减少锁竞争。

OK，让我们继续跟踪到`allocateNormal()`内部：

```
// PoolArena#allocateNormal
private void allocateNormal(PooledByteBuf<T> buf, int reqCapacity, int normCapacity) {
    // 先尝试从已有的PoolChunk中分配内存
    // 此时是第一次请求，所以，没有任何可用的PoolChunk
    if (q050.allocate(buf, reqCapacity, normCapacity)
        || q025.allocate(buf, reqCapacity, normCapacity)
        || q000.allocate(buf, reqCapacity, normCapacity)
        || qInit.allocate(buf, reqCapacity, normCapacity)
        || q075.allocate(buf, reqCapacity, normCapacity)) {
        return;
    }

    // key1, 创建一个新的PoolChunk
    PoolChunk<T> c = newChunk(pageSize, maxOrder, pageShifts, chunkSize);
    // key2, 使用这个PoolChunk分配内存
    boolean success = c.allocate(buf, reqCapacity, normCapacity);
    assert success;
    // 这新的PoolChunk加入到qInit中
    qInit.add(c);
}
```


这个方法中，有两个关键的地方：

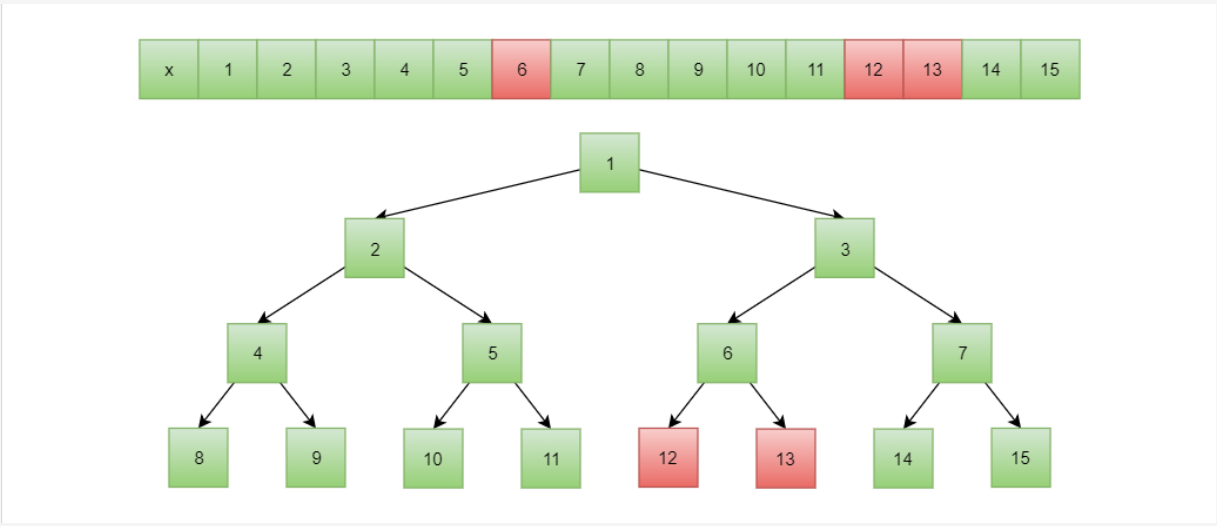
- 创建PoolChunk，通过上一节的分析，我们知道，数据全部都是存储在PoolChunk的memory字段中的，那么，这个memory是如何创建的呢？
- 使用PoolChunk分配内存，如何分配？

我们先来看第一个问题：

```
// PoolArena.HeapArena#newChunk
@Override
protected PoolChunk<byte[]> newChunk(int pageSize, int maxOrder, int pageShifts, int chunkSize) {
    return new PoolChunk<byte[]>(this, newByteArray(chunkSize), pageSize, maxOrder, pageShifts, chunkSize, 0);
}
// PoolArena.HeapArena#newByteArray
private static byte[] newByteArray(int size) {
    // 调用PlatformDependent
    return PlatformDependent.allocateUninitializedArray(size);
}
public static byte[] allocateUninitializedArray(int size) {
    // 默认地，UNINITIALIZED_ARRAY_ALLOCATION_THRESHOLD=-1
    // 创建了一个大小为16MB的byte[]数组，在堆内存中
    return UNINITIALIZED_ARRAY_ALLOCATION_THRESHOLD < 0 || UNINITIALIZED_ARRAY_ALLOCATION_THRESHOLD > size ?
        new byte[size] : PlatformDependent0.allocateUninitializedArray(size);
}
// PoolChunk的构造方法
PoolChunk(PoolArena<T> arena, T memory, int pageSize, int maxOrder, int pageShifts, int chunkSize, int offset) {
    unpooled = false;
    this.arena = arena;
    // memory就是上面创建的byte[]数组
    this.memory = memory;
    // 8KB
    this.pageSize = pageSize;
    // 13
    this.pageShifts = pageShifts;
    // 11
    this.maxOrder = maxOrder;
    // 16MB
    this.chunkSize = chunkSize;
    // 0
    this.offset = offset;
    // 12，表示如果满二叉树中的节点更新到了12，则表示此节点已经全分配了
    unusable = (byte) (maxOrder + 1);
    // 24，用于计算满二叉树中的节点占用的空间大小，比如2048占用8K，而1024占用16K
    log2ChunkSize = log2(chunkSize);
    // -8192，表示subpage溢出的掩码
    subpageOverflowMask = ~(pageSize - 1);
    // 初始时可使用的内存等于PoolChunk的整个空间，即16MB
    freeBytes = chunkSize;

    assert maxOrder < 30 : "maxOrder should be < 30, but is: " + maxOrder;
    // 一个PoolChunk可以被分成多少页，1<<11=2048
    maxSubpageAllocs = 1 << maxOrder;

    // 创建memoryMap和depthMap，满二叉树，最后一层的节点数正好等于上面所有层的节点数
    // 所以它们的大小为2048*2=4096
    memoryMap = new byte[maxSubpageAllocs << 1];
    depthMap = new byte[memoryMap.length];
    int memoryMapIndex = 1;
    // 初始化memoryMap和depthMap中的元素为每个节点的高度
    for (int d = 0; d <= maxOrder; ++ d) {
        int depth = 1 << d;
        for (int p = 0; p < depth; ++ p) {
            memoryMap[memoryMapIndex] = (byte) d;
            depthMap[memoryMapIndex] = (byte) d;
            memoryMapIndex ++;
        }
    }
    // 一个PoolChunk可以被分割成2048个Page
    // Page在Netty中同样使用PoolSubpage来表示
    subpages = newSubpageArray(maxSubpageAllocs);
    cachedNioBuffers = new ArrayDeque<ByteBuffer>(8);
}
```

> 满二叉树非常适合用数组来存储，下层节点的个数正好是上层节点个数的两倍，所以，从数组下标为1的位置开始存储元素，那么它的两个子节点的下标正好是父节点下标的两倍及两倍加1，举个例子，节点6的子节点分别为节点12和节点13，反之，一个节点的父节点正好为其下标的一半，比如节点13的父节点为`13/2=6`。
>
> 
>

这段代码主要创建了一个基于堆内存的byte[]数组并将其赋值给了PoolChunk的memory字段。同时，构建了两棵满二叉树memoryMap和depthMap，memoryMap用于保存节点分配的情况，depthMap用于存储节点的原始高度，当一个节点被完全分配后，它在memoryMap中的值将变成12（unusable）。并且，新建了一个大小为2048的PoolSubpage数组，当然了，这个数组暂未初始化，因为如果分配的内存大于等于一个Page大小（8KB）时，这个数组其实并没有什么用。

OK，到此，我们已经从堆内存拿到了一个16MB的byte[]数组，那么，PoolChunk是怎么把它分配给具体的PoolByteBuf的呢？继续跟踪`c.allocate()`方法:

```
// buffer.PoolChunk#allocate
boolean allocate(PooledByteBuf<T> buf, int reqCapacity, int normCapacity) {
    final long handle;
    // 判断是否大于等于一个Page的大小，即8KB
    if ((normCapacity & subpageOverflowMask) != 0) { // >= pageSize
        // 调用allocateRun()处理
        handle =  allocateRun(normCapacity);
    } else {
        // key1，调用allocateSubpage()处理
        handle = allocateSubpage(normCapacity);
    }

    // 处理失败
    if (handle < 0) {
        return false;
    }
    ByteBuffer nioBuffer = cachedNioBuffers != null ? cachedNioBuffers.pollLast() : null;
    // key2，分配完毕，初始化前面创建的PooledByteBuf
    // 所有分配的信息都保存在handle中
    initBuf(buf, nioBuffer, handle, reqCapacity);
    return true;
}
```

这个方法主要包含两部分内容：

- 根据请求的大小调用不同的方法处理，如果请求大于8KB，交给allocateRun()方法处理，否则交给

allocateSubpage()方法处理，它们都会返回一个叫作handle的long类型变量，这个handle里面就保存着分配内存的结果，这两个handle有什么区别呢？

- 使用上述返回的handle初始化前面创建的PooledByteBuf对象；

我们接着来看看两个分配的方法，先来一个简单点的分配方法——allocateRun()：

```
// PoolChunk#allocateRun
// 这个方法仅用于分配大于等于8KB的内存
private long allocateRun(int normCapacity) {
    // 计算在满二叉树的哪层寻找节点
    // 假设normCapacity=8KB
    // d=11-(log2(8192)-13)=11-(13-13)=11
    int d = maxOrder - (log2(normCapacity) - pageShifts);
    // 在d层寻找到一个节点，这个节点的id
    // 比如，在11层找到了2048这个节点
    int id = allocateNode(d);
    if (id < 0) {
        return id;
    }
    // 整个PoolChunk的可用容量减去8192
    // runLength()用于计算节点占用的数据量
    // 比如2048这个节点，注意位运算与减法运算的优先级
    // runLength(2048)=1<<log2ChunkSize-depth(id)=1<<(24-11)=1<<13=8192
    freeBytes -= runLength(id);
    // 返回这个节点的id
    return id;
}
```

allocateRun()方法用于分配大于等于8KB的内存，最后返回的是找到的那个节点的id，当然，这里在allocateNode()方法中也会调用updateParentsAlloc(id)方法更新其及父节点在memoryMap中的值，这一块就不细说了。

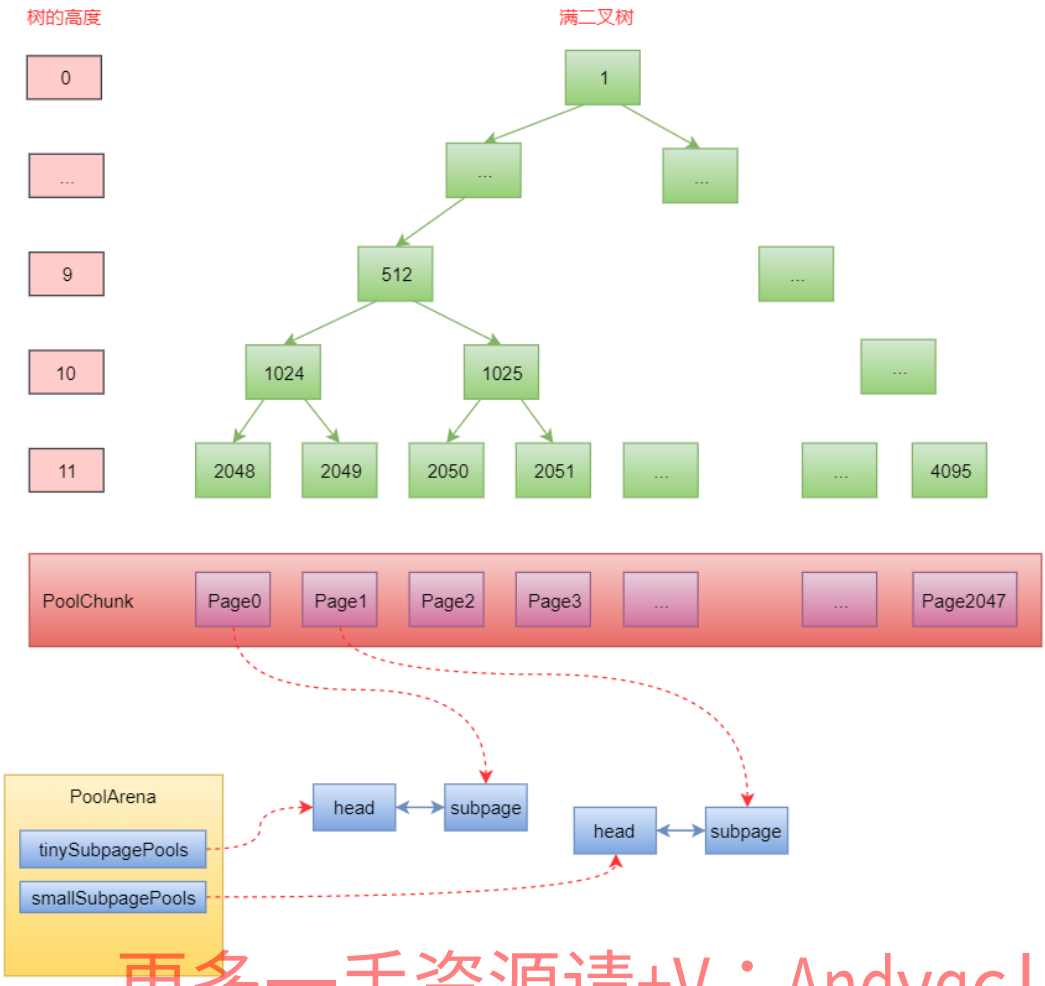
再来看看第二个分配方法——allocateSubpage()：

```
// PoolChunk#allocateSubpage
// 用于分配小于8KB的内存
private long allocateSubpage(int normCapacity) {
    // 根据规范化后的大小从PoolArena的tinySubpagePools或者smallSubpagePools中找到一个合适的元素作为head
    PoolSubpage<T> head = arena.findSubpagePoolHead(normCapacity);
    // 从第11层寻找一个合适的节点
    // 因为小于8KB，所以肯定是从叶子节点中寻找合适的节点
    int d = maxOrder;
    // 分段锁
    synchronized (head) {
        // 找到的节点id，比如2048
        int id = allocateNode(d);
        if (id < 0) {
            return id;
        }

        // 这个数组是PoolChunk中的数组，其实是Page数组，大小为2048
        final PoolSubpage<T>[] subpages = this.subpages;
        // 一页的大小，8192
        final int pageSize = this.pageSize;

        // 可用容量减去1页的大小
        // 这一页只要分配出去了内存，则表示这一页都被占用了
        // 后续只能分配跟第一次请求大小（规范后的大小）一样的内存
        freeBytes -= pageSize;

        // Page的索引，比如2048对应Page0
        int subpageIdx = subpageIdx(id);
        // 取出这个Page
        PoolSubpage<T> subpage = subpages[subpageIdx];
        if (subpage == null) {
            // 创建一个Page，并将其与PoolArena的head节点形成双向循环链表
            // runOffset()表示的是这个Page在PoolChunk中的偏移量
            // elemSize=normCapacity
            subpage = new PoolSubpage<T>(head, this, id, runOffset(id), pageSize, normCapacity);
            subpages[subpageIdx] = subpage;
        } else {
            subpage.init(head, normCapacity);
        }
        // 使用Page来分配内存
        return subpage.allocate();
    }
}
```



跟PoolSubpage这块相关的内容可以结合上一节的图来学习，一定要时刻记住，PoolSubpage有两重身份，一重代表着Page，一重代表着更小的内存块。

可以发现，PoolChunk对于分配小于8KB的内存，最后还是交给PoolSubpage来干的，所以，我们继续往里跟踪：

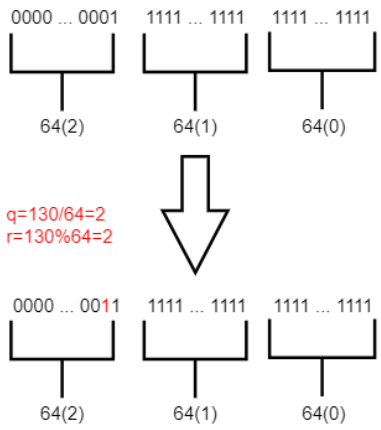
```
// PoolSubpage#allocate
long allocate() {
    if (elemSize == 0) {
        return toHandle(0);
    }

    if (numAvail == 0 || !doNotDestroy) {
        return -1;
    }
    // bitmap的每一bit（位）用于保存每一小块内存是否被分配了
    final int bitmapIdx = getNextAvail();
    // q表示在第几个long中，2的6次方=64，q=bimapIdx>>>6=bitmapIdx/64
    // 比如，130位对应于下标为130/64=2的long的第2位
    int q = bitmapIdx >>> 6;
    // r表示在这个long类型中的偏移量
    // 相当于bitmapIdx%64
    // 比如，130%64=2
    int r = bitmapIdx & 63;
    assert (bitmap[q] >>> r & 1) == 0;
    // 更新第q个long的第r位为1，表示已分配
    bitmap[q] |= 1L << r;

    // 如果可用内存为0了，就从PoorArena的tinySubpagePools或者smallSubpagePools对应的链表中移除
    if (-- numAvail == 0) {
        removeFromPool();
    }

    // 转换成handle
    return toHandle(bitmapIdx);
}
private long toHandle(int bitmapIdx) {
    // 为了与Normal类型的区分，加一个掩码
    // 比如，在Page中分得了一小块内存，且是第一个元素
    // 那么handle=0x4000000000000000L|0|2048
    return 0x4000000000000000L | (long) bitmapIdx << 32 | memoryMapIdx;
}
```


关于bitmap，我举个例子助你来理解，比如，按48B分割一个Page，那么，可以分割成 8KB/48B=170.666=170 个小内存块，剩余那零点几就变成了内存碎片，一个long类型可以存储64个小内存块的状态信息，那么，170个就需要 170/64=2.656=3 个long类型来存储所有小内存块的状态信息，此时，要分配第130个48B，那么，就标记下标为2的long的第2位为1，当然，在此之前前面129位肯定都已经被分配了，所以它们对应的位都是1：



好了，到这里内存就已经分配完毕了，那么，怎么把它跟PooledByteBuf关联起来呢？其实也很简单，只要把上面我们求得的那个handle跟PooledByteBuf绑定在一起我们就能准确地定位到是在哪个Page的哪一小块分配了多少的内存，这也就是初始化PooledByteBuf的功能，即： initBuf() ：

```
// PoolChunk#initBuf
void initBuf(PooledByteBuf<T> buf, ByteBuffer nioBuffer, long handle, int reqCapacity) {
    // 第几号节点
    int memoryMapIdx = memoryMapIdx(handle);
    // bitmapIdx索引
    int bitmapIdx = bitmapIdx(handle);
    // bitmapIdx=0表示的是Normal
    // 因为subpage方式的handle前面有个掩码，所以不会为0
    if (bitmapIdx == 0) {
        byte val = value(memoryMapIdx);
        assert val == unusable : String.valueOf(val);
        buf.init(this, nioBuffer, handle, runOffset(memoryMapIdx) + offset,
                reqCapacity, runLength(memoryMapIdx), arena.parent.threadCache());
    } else {
        // 使用subpage初始化
        initBufWithSubpage(buf, nioBuffer, handle, bitmapIdx, reqCapacity);
    }
}
// PoolChunk#initBufWithSubpage(PooledByteBuf<T>, ByteBuffer, long, int, int)
private void initBufWithSubpage(PooledByteBuf<T> buf, ByteBuffer nioBuffer,
                                long handle, int bitmapIdx, int reqCapacity) {
    assert bitmapIdx != 0;

    // 第几号节点
    int memoryMapIdx = memoryMapIdx(handle);

    // 第几个Page
    PoolSubpage<T> subpage = subpages[subpageIdx(memoryMapIdx)];
    assert subpage.doNotDestroy;
    assert reqCapacity <= subpage.elemSize;

    // 调用PooledByteBuf自己的init方法
    // 第四个参数用于计算这一个块内存在整个PoolChunk中的偏移量
    buf.init(
        this, nioBuffer, handle,
        runOffset(memoryMapIdx) + (bitmapIdx & 0x3FFFFFFF) * subpage.elemSize + offset,
        reqCapacity, subpage.elemSize, arena.parent.threadCache());
}
// PooledByteBuf#init
void init(PoolChunk<T> chunk, ByteBuffer nioBuffer,
          long handle, int offset, int length, int maxLength, PoolThreadCache cache) {
    init0(chunk, nioBuffer, handle, offset, length, maxLength, cache);
}
// PooledByteBuf#init0
private void init0(PoolChunk<T> chunk, ByteBuffer nioBuffer,
                   long handle, int offset, int length, int maxLength, PoolThreadCache cache) {
    assert handle >= 0;
    assert chunk != null;

    // 属于哪个PoolChunk
    this.chunk = chunk;
    // 将chunk的memory赋值给buf
    memory = chunk.memory;
    tmpNioBuf = nioBuffer;
    // 哪个分配器
    allocator = chunk.arena.parent;
    // 绑定的PoolThreadCache，在释放内存的时候使用
    this.cache = cache;
    // handle，记录了哪个节点哪个位
    this.handle = handle;
    // 在整个chunk中的偏移量
    this.offset = offset;
    // 实际请求的大小，即40B
    this.length = length;
    // 最大长度，为规范化后的大小，即48B
    this.maxLength = maxLength;
}
```

OK，到这里整个PooledByteBuf就创建完毕了，既然PooledByteBuf中保存了PoolChunk的整个memory及操作的偏移量，那么，写入数据和读取数据的时候是不是只要按这个偏移量来操作就可以了呢？

3. 写入数据

关于写入数据，我们快速的过一下源码：

```
// 写入一个int类型
@Override
public ByteBuf writeInt(int value) {
    ensureWritable0(4);
    _setInt(writerIndex, value);
    writerIndex += 4;
    return this;
}
@Override
protected void _setInt(int index, int value) {
    // 调用Unsafe来写入数据
    // memory即chunk的memory
    UnsafeByteBufUtil.setInt(memory, idx(index), value);
}
protected final int idx(int index) {
    // 计算写入偏移量，等于memory的偏移量+writeIndex
    return offset + index;
}
```

可以看到，跟我们预期的完全一样，这里我再提一个问题：通过前面我们分析非池化的ByteBuf，我们知道，Netty中的ByteBuf是可以扩容的，而上面我们创建PooledByteBuf有个maxLength限制了最大可以使用的内存，那么，对于PooledByteBuf，它是如何扩容的呢？留给你自己来探索喽^^

4. 读取数据

读取数据肯定是跟写入数据一样的处理方式，这里就不再赘述了。

5. 回收内存

到目前为止，PoolThreadCache都没有派上用场，那么，它是没用吗？当然不是，只有回收内存的时候才考虑要不要把这部分内存放到PoolThreadCache中缓存起来，下面我们就来看看回收内存的整个过程，同样地，还是直接跟踪到源码里面：

```
// ReferenceCountUtil#release(Object)
// 回收内存的工具方法
public static boolean release(Object msg) {
    if (msg instanceof ReferenceCounted) {
        return ((ReferenceCounted) msg).release();
    }
    return false;
}
// ...
// PooledByteBuf#deallocate
@Override
protected final void deallocate() {
    if (handle >= 0) {
        final long handle = this.handle;
        // 将handle置为-1
        this.handle = -1;
        // 将memory置为空
        memory = null;
        // 调用PoolArena的free()方法回收内存
        chunk.arena.free(chunk, tmpNioBuf, handle, maxLength, cache);
        tmpNioBuf = null;
        // 将chunk置为空
        chunk = null;
        // 对象池的方法，本节暂不涉及
        recycle();
    }
}
```

这里将PooledByteBuf中跟内存池有关的变量全部置为初始值，并交给PoolArena来free内存，所以，PoolArena可以看作是整个内存池的管家，所有的入口都在它这里，继续跟进：

```
// PoolArena#free
// 注意，这个nioBuffer不是上面创建的那个PooledByteBuf
void free(PoolChunk<T> chunk, ByteBuffer nioBuffer, long handle, int normCapacity, PoolThreadCache cache) {
    // 非池化的，对于Huge来说使用的是非池化的方式
    if (chunk.unpooled) {
        int size = chunk.chunkSize();
        destroyChunk(chunk);
        activeBytesHuge.add(-size);
        deallocationsHuge.increment();
    } else {
        SizeClass sizeClass = sizeClass(normCapacity);
        // 使用PoolThreadCache来处理
        if (cache != null && cache.add(this, chunk, nioBuffer, handle, normCapacity, sizeClass)) {
            // 这里相当于线程缓存做了一层拦截，如果被拦截下来了，就不用释放了
            return;
        }

        freeChunk(chunk, handle, sizeClass, nioBuffer, false);
    }
}
// PoolThreadCache#add
boolean add(PoolArena<?> area, PoolChunk chunk, ByteBuffer nioBuffer,
            long handle, int normCapacity, SizeClass sizeClass) {
    // key1，PoolThreadCache中同样根据不同的请求大小分成不同的MemoryRegionCache
    MemoryRegionCache<?> cache = cache(area, normCapacity, sizeClass);
    if (cache == null) {
        return false;
    }
    // 存储在MemoryRegionCache中
    return cache.add(chunk, nioBuffer, handle);
}
// PoolThreadCache.MemoryRegionCache#add
public final boolean add(PoolChunk<T> chunk, ByteBuffer nioBuffer, long handle) {
    Entry<T> entry = newEntry(chunk, nioBuffer, handle);
    // key2，入队
    boolean queued = queue.offer(entry);
    if (!queued) {
        entry.recycle();
    }
    return queued;
}
```


整个流程很清晰，根据请求大小（规范化后的大小）从PoolThreadCache中找到一个对应的MemoryRegionCache，然后把这个PoolChunk的这个handle组成一个Entry放到队列中，这里有两个问题：

1. MemoryRegionCache的规则是什么？
2. 这个队列是阻塞队列吗？

我们先来看第一个问题，其实，MemoryRegionCache跟PoolArena中的tinySubpagePools和smallSubpagePools这两个数组有点类似，在PoolThreadCache中，存在三个类似的数组：tinySubPageHeapCaches、smallSubPageHeapCaches、normalHeapCaches，它们分别用来缓存Tiny、Small、Normal类型的请求，它们的大小分别是32、4、3，前两者跟PoolArena中的那两个数组大小完全一样，寻找的规则也是一模一样的，对于normalHeapCaches，有点区别，它只缓存Normal类型的三个段，即8KB、16KB、32KB，所以，PoolThreadCache最多只能缓存32KB大小的请求释放的内存，超过32KB的请求释放的内存则不再缓存。

接着来看看第二个问题，这个队列不是一个阻塞队列，它是一种叫作`MpscArrayQueue`，`mpsc`的全称叫作`Multiple producer single consumer`，即多生产者单消费者，它是Netty自己实现的一种队列，里面运用了很多Java中的高级技巧，效率比Java原生的阻塞队列或者并发安全的队列要高不少。

那么，Netty为什么要使用这种多生产者单消费者队列呢？

先埋个伏笔，我们后面详细介绍`MpscArrayQueue`队列。

6. 分配一个30B的ByteBuf

如果分配一个30B的ByteBuf，它会走上面加到PoolThreadPool中的缓存吗？

我不打算跟代码，让我们一起来脑补“分配一个30B的ByteBuf”的整个过程：

1. 30B规范化之后等于32B，属于Tiny类型；
2. 尝试使用PoolThreadCache分配内存，发现存放32B的缓存MemoryRegionCache中并没有相关缓存，所以，走缓存失败；
3. 尝试从PoolArena的tinySubpagePools[1]中找到符合的PoolSubpage，发现并没有，所以，只能对整个PoolArena加锁了；
4. 尝试从PoolArena的6个chunkList中分配内存，发现在qInit中存在可用的PoolChunk，使用它分配内存；
5. 在这个PoolChunk中找到第2049号节点可以使用，使用它分配内存（2048节点已经被前面的48B的请求占用了）；
6. 2049节点对应Page1，它首次接收请求，所以新建一个PoolSubpage，它的elemSize=32B，同时，把这个PoolSubpage与PoolArena的tinySubpagePools[1]这个head形成双向链表，另外，整个Page按32B分割成了`8192/32=256`个小块内存，需要`256/64=4`个long类型存储每个小块内存的状态；
7. 当前这个请求是Page1的第一个请求，所以，它在Page1中的偏移量是0，但是它在整个PoolChunk中的偏移量为8192，所以，分配完毕返回的handle=0x4000000000000000L | 0 | 2049；
8. 调用PooledByteBuf的init()方法初始化，它的memory与48B那个是同一个PoolChunk的memory，但是它的偏移量为8192，容量为30B，最大容量为32B；
9. 虽然分配的是30B的容量，且最大容量为32B，但是依然可以写入超过32B容量的数据，当写入超过32B时，将进行扩容，扩容的逻辑咱们就不脑补了。

通过这种脑补法，发现48B和32B关联的是同一个PoolChunk的memory，那么，它们并发地写会不会有线程安全的问题呢？其实，并不会有，因为它们是对memory这个数组不同的段进行操作，之间不存在线程安全的问题，这也是锁分离思想的体现（大锁化小锁，小锁化无锁）。

7. 再次分配一个40B的ByteBuf

让我们再一起使用脑补法看看再次分配一个40B的ByteBuf会怎样：

1. 40B规范化后为48B，属于Tiny类型；
2. 尝试使用PoolThreadCache分配内存，查看其tinySubPageHeapCaches[2]，发现它的队列里面有个可以使用的entry，将其出队（poll），这个entry里面存储了属于哪个PoolChunk，及handle，而handle里面存储了哪个节点（memoryMapIdx），及这小块内存的索引（bitmapIdx）；
3. 通过这个entry初始化PooledByteBuf。

OK，到这里整个内存池部分就讲解完毕了。

后记

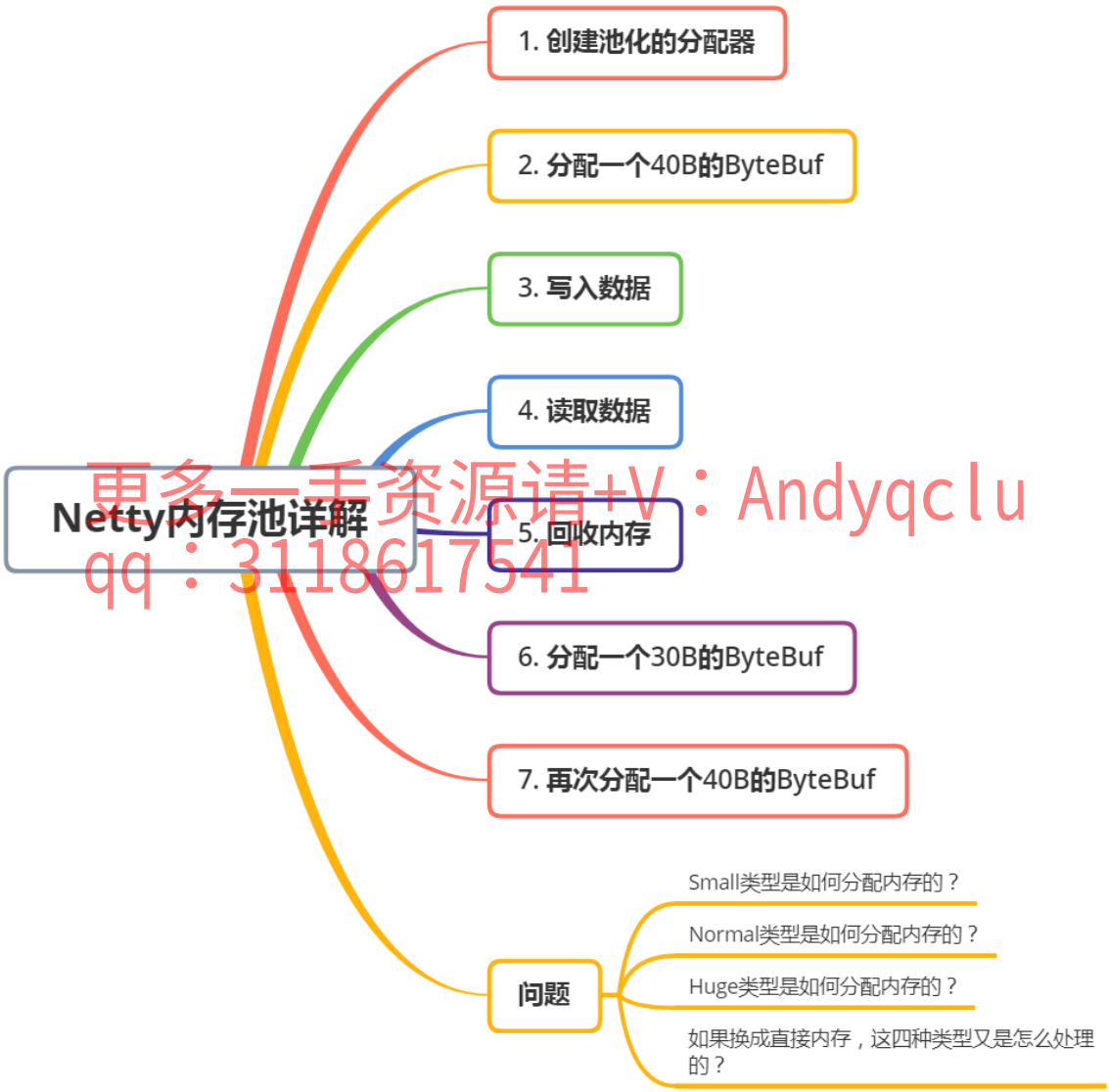
本节，我们基于堆内存把整个内存池学习了一遍，其实这一块可以学习的东西还有很多，比如：

1. Small类型是如何分配内存的？
2. Normal类型是如何分配内存的？
3. Huge类型是如何分配内存的？
4. 如果换成直接内存，这四种类型又是怎么处理的？

另外，我留了两个思考题，大家可以尝试解答一下，见文后。

思维导图

今天的思维导图，我希望你能根据这几个步骤，把Netty的内存池描述清楚。



思考题一

```
public class ByteBufTest {
    public static void main(String[] args) throws InterruptedException {
        // 参数是preferDirect，即是否偏向于使用直接内存
        ByteBufAllocator allocator = new PooledByteBufAllocator(false);
        // 分配一个40B的ByteBuf
        ByteBuf byteBuf = allocator.heapBuffer(40);
        // 写入数据
        byteBuf.writeInt(4);
        // 读取数据
        System.out.println(byteBuf.readInt());
        // 回收内存
        new Thread(()->ReferenceCountUtil.release(byteBuf)).start();
        // 休息1秒，保证完全释放
        Thread.sleep(1000);
        // 再次分配一个40B的ByteBuf
        ByteBuf byteBuf2 = allocator.heapBuffer(40);
    }
}
```

如上代码，第二次分配40B内存的时候是否可以使用到缓存？

思考题二

```
public class ByteBufTest {
    public static void main(String[] args) throws InterruptedException {
        // 参数是preferDirect，即是否偏向于使用直接内存
        ByteBufAllocator allocator = new PooledByteBufAllocator(false);
        // 分配一个40B的ByteBuf
        ByteBuf byteBuf = allocator.heapBuffer(40);
        // 写入数据
        byteBuf.writeInt(4);
        // 读取数据
        System.out.println(byteBuf.readInt());
        // 回收内存
        ReferenceCountUtil.release(byteBuf);
        // 休息1秒，保证完全释放
        Thread.sleep(1000);
        new Thread(() -> {
            // 再次分配一个40B的ByteBuf
            ByteBuf byteBuf2 = allocator.heapBuffer(40);
            // 其它处理
        }).start();
    }
}
```


如上代码，第二次分配40B内存的时候是否可以使用到缓存？

}

20 jemalloc内存分配器是什么

22 Netty的对象池又是如何实现的