**1. Intro to ByteBuf**

io.netty.buffer.ByteBuf interface is netty’s byte container.

Depending on where ByteBuf object stores the data, there are mainly two categories of ByteBuf.

1. **Heap-based ByteBuf:** Data is backed by heap (byte[]).
2. **Native memory-based ByteBuf:** Data is backed by native memory. Memory is allocated using native calls.

#1 are sometime referred as “HeapByteBuf” and #2 referred as “DirectByteBuf”, note that there is no type in the *io.netty.buffer* name-space with exactly same name.

The native IO operation requires data it is operating on in native memory not JVM managed memory.

If the data is residing in JVM heap, then in order to send using native IO calls it needs to be copied to native memory. By using direct buffer, we can avoid this extra copying.

**2. How to allocate ByteBuf instance**

An implementation of io.netty.buffer.ByteBuf interface can be created (allocated) using one of the allocators represented by the interface io.netty.buffer.ByteBufAllocator.

interface ByteBufAllocator {

ByteBuf directBuffer();

ByteBuf heapBuffer();

ByteBuf buffer();

}

There are mainly two implementation of io.netty.buffer.ByteBufAllocator

io.netty.buffer.PooledByteBufAllocator

io.netty.buffer.UnpooledByteBufAllocator

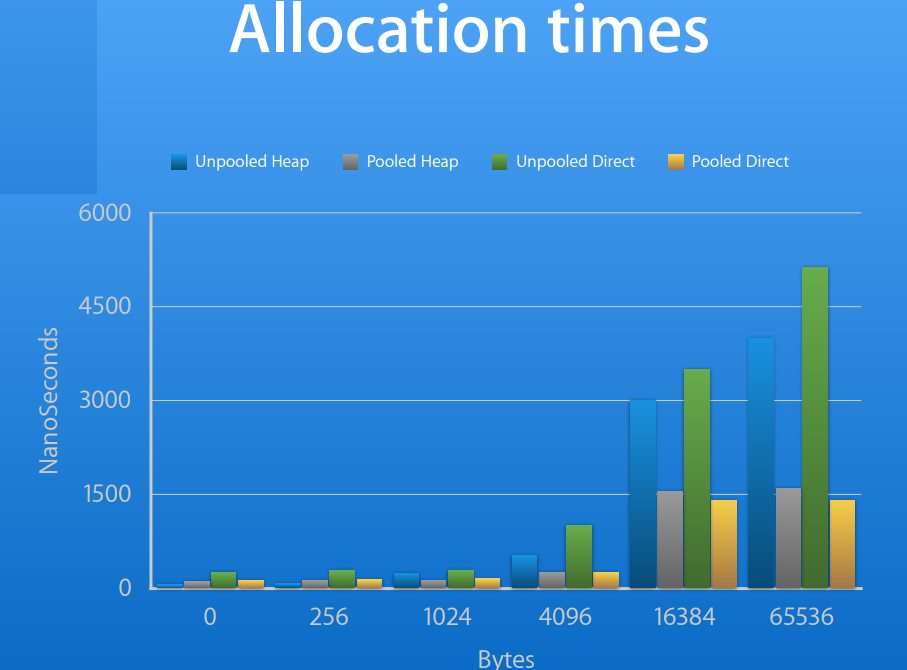
**3. How to get Allocator**

There are couple of ways to get access to one of the concrete allocators.

1. io.netty.buffer.PooledByteBufAllocator.DEFAULT
2. io.netty.buffer.UnpooledByteBufAllocator.DEFAULT
3. io.netty.buffer.ByteBufAllocator.DEFAULT
4. Return io.netty.buffer.PooledByteBufAllocator by default.
5. Return io.netty.buffer.UnpooledByteBufAllocator if the system property with key io.netty.allocator.type set to unpooled. [Ref: [setting system properties](https://stackoverflow.com/questions/7054972/java-system-properties-and-environment-variables)]
6. io.netty.buffer.Unpooled

This type wraps io.netty.buffer.UnpooledByteBufAllocator.DEFAULT allocator and exposes static methods that simply redirect calls to methods in the wrapped allocator.

**4. Prefer PooledByteBufAllocator over UnpooledByteBufAllocator**

****

Source: <https://blog.twitter.com/engineering/en_us/a/2013/netty-4-at-twitter-reduced-gc-overhead.html>

Using pooled direct buffer reduces the allocation time by a factor of 5.

**5. io.netty.buffer.ByteBuf concrete types**

Depending on the allocator and specific methods in it, there are 4 different concrete type of ByteBuf we can allocate.

|  |  |  |  |
| --- | --- | --- | --- |
| **ALLOCATOR (Pooled vs UnPooled)** | **MEHOD** | **BYTEBUF** | **ALLOCATED FROM** |
|
| io.netty.buffer.PooledByteBufAllocator | directBuffer() | io.netty.buffer.PooledDirectByteBuf | Native\_Memory |
| heapBuffer() | io.netty.buffer.PooledHeapByteBuf | Heap\_Memory |
| io.netty.buffer.UnpooledByteBufAllocator | directBuffer() | io.netty.buffer.UnpooledDirectByteBuf | Native\_Memory |
| heapBuffer() | io.netty.buffer.UnpooledHeapByteBuf | Heap\_Memory |

Reference to the allocator from which a ByteBuf is allocated can be retrieved using ByteBuf::alloc() method.

**6. Note on netty’s Native memory based ByteBuf (a.k.a.s DirectByteBuf)**

The native memory based ByteBuf implementations in Netty (namely io.netty.buffer.UnpooledDirectByteBuf and io.netty.buffer.PooledDirectByteBuf) wraps JDK java.nio.DirectByteBuffer.

**6.A - How java.nio.DirectByteBuffer is GC-ed**

During the course of garbage collecting (GC-ing) on-heap java.nio.DirectByteBuffer object, the off-heap (native) memory it refers can be released. Below technique is used to achieve this:

java.nio.DirectByteBuffer has a field of type sun.misc.Cleaner. When this sun.misc.Cleaner instance gets GC-ed (as part of GC-ing its parent), the reference handler thread handles it in a special way

if (r instanceof Cleaner) {

   ((Cleaner)r).clean();

   ….

}

This will make a call to java.nio.DirectByteBuffer$Deallocator#run() method where the native memory gets freed with a call to unsafe#freeMemory(long).

Refer: <https://stackoverflow.com/questions/6697709/are-java-directbytebuffer-wrappers-garbage-collected>

**6.B - Why  java.nio.DirectByteBuffer GC collection is not optimal**

Relying on GC to free up the native memory hold by java.no.DirectByteBuffer is not a good idea. GC monitors heap and kicks off once we short on the heap memory. GC will not monitor the native memory, this means couple of non-referenced java.nio.DirectByteBuffer instances in heap may not cause GC to run, but they will be holding releasable native memory.

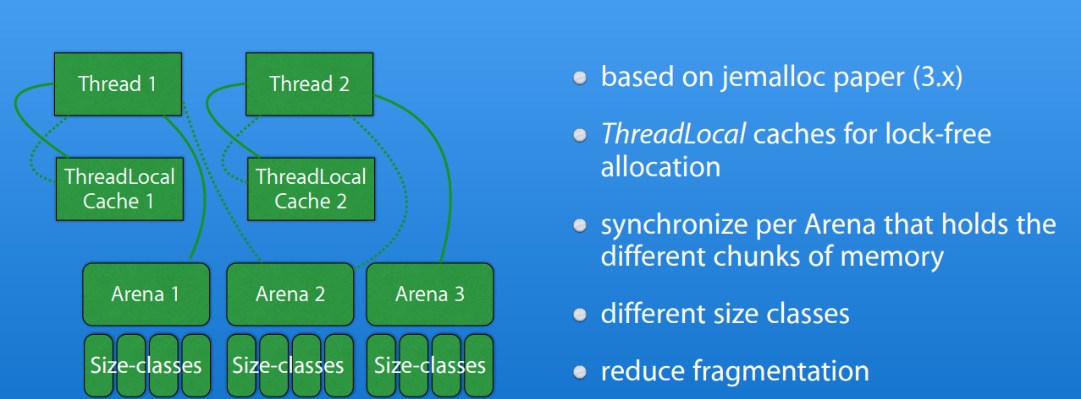
**6.C - io.netty.buffer.[Pooled|Unpooled]DirectByteBuf::release()**

The io.netty.buffer.[Pooled|Unpooled]DirectByteBuf::release() enables deallocating the native memory explicitly. If release() method finds that CERTAIN CONDITIONS are met then it get access to the cleaner and invoke clean.

Double clicking on CERTAIN CONDITIONS: If ByteBuf is allocated from a PooledByteBufAllocator then calling release() returns the buffer to the pool if its reference count is 0, so no call to cleaner.clean(). If ByteBuf is allocated from a UnpooledByteBufAllocator and its reference count is 0 then calling release() invokes cleaner.clean() to release the native memory.

**7. Implementation of PooledByteBufAllocator**

Pooling is implemented based on jemalloc 3.x paper.



Ref: jemalloc <https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919/>

**8. Intro: Reference counting in ByteBuf**

io.netty.buffer.ByteBuf extends io.netty.util.ReferenceCounted.

io.netty.buffer.ByteBuf::retain() increment the reference count by 1.

io.netty.buffer.ByteBuf::release() decrement the reference count by 1.

**8.A Behavior of release()**

**8.A.a Behavior of release() when ByteBuf::refCnt() > 1:**

Invoking io.netty.buffer.ByteBuf::release() when reference count is > 1 will simply decrement the reference count by 1.

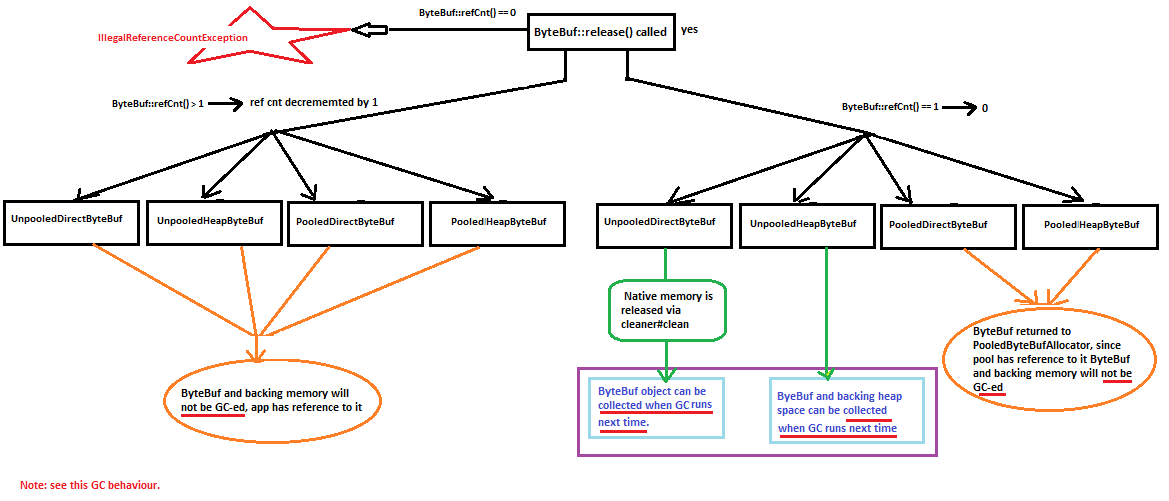
**8.A.b Behavior of release() when ByteBuf::refCnt() == 0**

If reference count is zero then trying to call retain() or release() will throw IllegalReferenceCountException.

**8.A.c Behavior of release() when ByteBuf::refCnt() == 1**

When reference count of a ByteBuf is 1, behavior of release() method upon it invocation differ depending on the concrete type of ByteBuf.

|  |  |  |  |
| --- | --- | --- | --- |
| **ByteBuf variable;** | **refCount()** | **release()** | **Behavior** |
| io.netty.buffer.PooledDirectByteBuf pdbf =  PooledByteBufAllocator.DEFAULT.directBuffer(); | pdbf.refCount() == 1 | pdbf.release() | DirectByteBuf instance returned to pool maintained by PooledByteBufAllocator instance from which it is allocated |
| io.netty.buffer.UnpooledDirectByteBuf updbf =  UnpooledByteBufAllocator.DEFAULT.directBuffer(); | updbf.refCount() == 1 | updbf.release() | The native memory referenced by the DirectByteBuf released  ByteBuf instance is eligible for collection when GC run next time. |
| io.netty.buffer.PooledHeapByteBuf phbf = PooledByteBufAllocator.DEFAULT.heapBuffer(); | phbf.refCount() == 1 | phbf.release() | HeapByteBuf instance returned to pool maintained by PooledByteBufAllocator instance from which it is allocated |
| io.netty.buffer.UnpooledHeapByteBuf uphbf = UnpooledByteBufAllocator.DEFAULT.heapBuffer(); | uphbf.refCount() == 1 | uphbf.release() | HeapByteBuf instance eligible for collection when GC run next time.  This bytebuf is unpooled and allocated from heap, hence even if we don’t call release() still it can be GC-ed if collector found that this object is not reachable from any other object [i.e. the application has no reference to it] |

****

Note: The ByteBuf will not be GC-ed if one of the following conditions are true.

* 1. Application holds a reference to a ByteBuf, it won’t be GC-ed.
  2. Once the application return ByteBuf to pool (by calling release()), pool’s DS holds a reference to ByteBuf hence it won’t be GC-ed.

**8.B Memory leak and PooledByteBuf::release()**

In case of pooled ByteBufs (io.netty.buffer.PooledDirectByteBuf, io.netty.buffer.PooledHeapByteBuf ) not calling release() will cause memory leak.

If app has no reference to the pooled ByteBuf (e.g. variable holding the reference is not in the scope anymore) and if app missed to invoke release() on it (hence pool is not referring the ByteBuf instance) then such ByteBuf instance become ELIGIBLE for collection. When collected, the underlying buffer will be reclaimed by GC, hence there is no memory leak w.r.t underlying buffer. The memory leak in this case is, Netty’s pooled byteBuf allocator won’t know that the ByteBuf allocated from it is GC-ed. Allocator thinks application is still using pool which cause the slots in the internal data structure that allocator uses to keep track of the ByteBuf metadata to never get released (GC-ed).

**8.C OutOfDirectMemory and DirectByteBuf::release()**

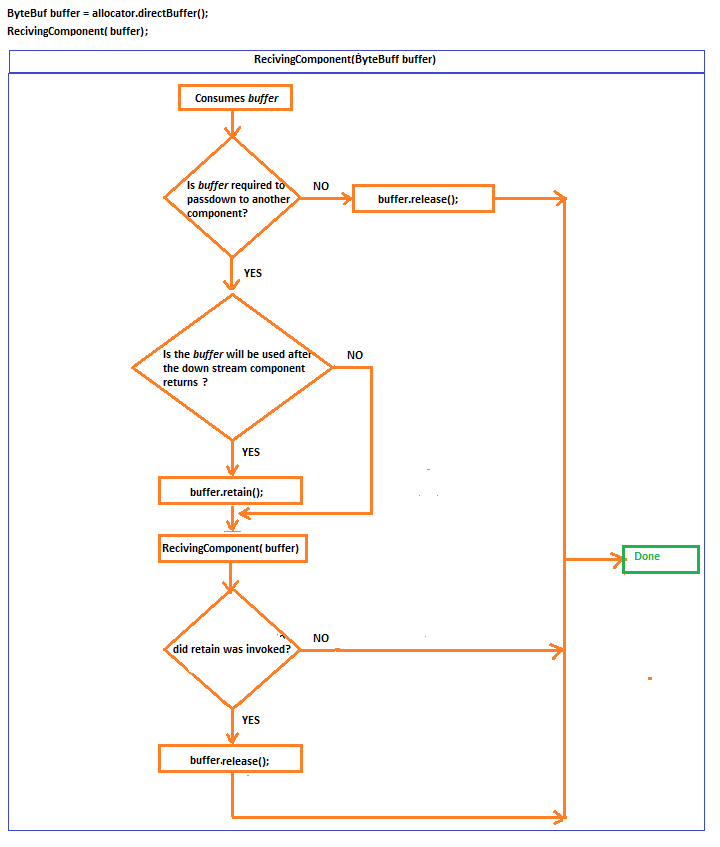
In case of io.netty.buffer.PooledDirectByteBuf and  io.netty.buffer.UnPooledDirectByteBuf not calling release() can cause OutOfDirectMemory Exception.

Even though as part of GC-ing this object releases the referenced native memory [Cleaner#clean()], relying on GC to free up the native memory hold by java.no.DirectByteBuffer is not a good idea. GC monitors heap and kicks off once app is short on the heap memory. GC will not monitor the native memory, this means couple of non-referenced java.nio.DirectByteBuffer instances in heap may not cause GC to run, but they will be holding releasable native memory. This may cause OutOfDirectMemoryException.

**9. How a method decide whether it is responsible for invoking ByteBuf::release()**

The general rule of thumb is, the party that access a ByteBuf instance lastly is responsible for calling calling release() on it.

1. If a sending component is supposed to pass ByteBuf instance to another receiving component, the sending component usually does not need to invoke release() but that decision is deferred to receiving component.
2. If a component consumes a ByteBuf and knows nothing else will access it anymore (i.e. does not pass along the ByteBuf instance to another component), the component should invoke release().



Ref: <http://netty.io/wiki/reference-counted-objects.html>