3

第 3 篇

Netty 核心篇

10

第 10 章

Netty 内存分配 ByteBuf

课程目标

1、了解 Netty 内存分配的类别。

2、如何减少多线程内存分配之间的竞争？

3、不同大小的内存是如何分配的？

内容定位

1、提前阅读了预习资料且完成前面章节学习的人群。

2、希望深入了解 Netty 源码的人群。

3、未来可能参与中间件开发的人群。

10.1 初识 ByteBuf

ByteBuf 是 Netty 整个结构里面最为底层的模块，主要负责把数据从底层 IO 里面读到 ByteBuf，然后传递给应用程序，应用程序处理完成之后再把数据封装成 ByteBuf 写回到 IO。所以，ByteBuf 是直接与底层打交道的一层抽象。这块内容，相对于 Netty 其他模块来说，是非常复杂的。但是没关系，我会把这部分内容分别拆解，从不同角度来分析 ByteBuf 的分配和回收。本章主要从内存与内存管理器的抽象、不同规格大小和不同类别的内存的分配策略以及内存的回收过程来展开。

10.1.1 ByteBuf 的基本结构

我们可以首先来看一下 Netty 官方对 ByteBuf 的描述如下：

```
+-------------------+------------------+----------------------+
| discardable bytes |  readable bytes  |    writable bytes    |
|                   |     (CONTENT)    |                      |
+-------------------+------------------+----------------------+
|                   |                  |                      |
0      <=      readerIndex   <=   writerIndex     <=     capacity
```

从上面 ByteBuf 的结构来看，我们发现 ByteBuf 有三个非常重要的指针，分别是 readerIndex(记录读指针的开始位置)、writerIndex(记录写指针的开始位置)和 capacity(缓冲区的总长度)，三者的关系是 readerIndex<=writerIndex<=capacity。然后，从 0 到 readerIndex 为 discardable bytes 表示是无效的，从 readerIndex 到 writerIndex 为 readable bytes 表示可读数据区，从 writerIndex 到 capacity 为 writable bytes 表示这段区间空闲可以往里面写数据。除了这三个指针，其实 ByteBuf 里面还有一个 maxCapacity，这就相当于是 ByteBuf 扩容的最大阈值，我们看它的源码中有定义：

```
    /**
     * Returns the maximum allowed capacity of this buffer.  If a user attempts to increase the
     * capacity of this buffer beyond the maximum capacity using {@link #capacity(int)} or
     * {@link #ensureWritable(int)}, those methods will raise an
     * {@link IllegalArgumentException}.
     */
    public abstract int maxCapacity();
```

这个指针可以看做是 capactiy 之后的这段，当 Netty 发现 writable bytes 写数据超出空间大小时，ByteBuf 会提前帮我们自动扩容，扩容之后，就有了足够的空间来写数据，同时 capactiy 也会同步更新，maxCapacity 就是扩容后 capactiy 的最大值。

10.1.2 ByteBuf的重要API

接下来我们来看 ByteBuf 的基本 API，主要包括 read()、write()、set()以及 mark()、reset()方法。我们用下面的表格对 ByteBuf 最重要的 API 做一个详细说明：

方法	解释
readByte()	从当前 readerIndex 指针开始往后读 1 个字节的数据并移动 readerIndex ,将数据转化为 byte。
readUnsignedByte()	读取一个无符号的 byte 数据。
readShort()	从当前 readerIndex 指针开始往后读 2 个字节的数据并移动 readerIndex ,将数据转化为 short。
readInt()	从当前 readerIndex 指针开始往后读 4 个字节的数据并移动 readerIndex，将数据转化为 int。
readLong()	从当前 readerIndex 指针开始往后读 8 个字节的数据并移动 readerIndex ,将数据转化为 long。
writeByte()	从当前 writerIndex 指针开始往后写 1 个字节的数据并移动 writerIndex。
setByte()	将 byte 数据写入到指定位置，不移动 writerIndex。
markReaderIndex()	在读数据之前，将 readerIndex 的状态保存起来，方便在读完数据之后将 readerIndex 复原。
resetReaderIndex()	将 readerIndex 复原到调用 markReaderIndex()之后的状态。
markWriterIndex()	在写数据之前，将 writerIndex 的状态保存起来，方便在读完数据之后将 writerIndex 复原。
resetWriterIndex()	将 writerIndex 复原到调用 markWriterIndex()之后的状态。
readableBytes()	获取可读数据区大小，相当于获取当前 writerIndex 减去 readerIndex 的值。
writableBytes()	获取可写数据区大小，相当于获取当前 capactiy 减去 writerIndex 的值。
maxWritableBytes()	获取最大可写数据区的大小，相当于获取当前 maxCapactiy 减去 writerIndex 的值。

在 Netty 中，ByteBuf 的大部分功能是在 AbstractByteBuf 中来实现的，我们可以先进入 AbstractByteBuf 的源码看看：

```
public abstract class AbstractByteBuf extends ByteBuf {

    ...

    int readerIndex;    //读指针
    int writerIndex;    //写指针
    private int markedReaderIndex;   //mark 之后的读指针
```

```
    private int markedWriterIndex;   //mark之后的写指针
    private int maxCapacity;    //最大容量

    ...

}
```

最重要的几个属性 readerIndex、writerIndex、markedReaderIndex、markedWriterIndex、maxCapacity 被定义在 AbstractByteBuf 这个抽象类中，下面我们可以来看看基本读写的骨架代码实现。例如，几个基本的判断读写区间的 API，我们来看一下它的具体实现：

```
public abstract class AbstractByteBuf extends ByteBuf {

    ...

    @Override
    public boolean isReadable() {
        return writerIndex > readerIndex;
    }

    @Override
    public boolean isReadable(int numBytes) {
        return writerIndex - readerIndex >= numBytes;
    }

    @Override
    public boolean isWritable() {
        return capacity() > writerIndex;
    }

    @Override
    public boolean isWritable(int numBytes) {
        return capacity() - writerIndex >= numBytes;
    }

    @Override
    public int readableBytes() {
        return writerIndex - readerIndex;
    }

    @Override
    public int writableBytes() {
        return capacity() - writerIndex;
    }

    @Override
    public int maxWritableBytes() {
        return maxCapacity() - writerIndex;
    }

     @Override
     public ByteBuf markReaderIndex() {
         markedReaderIndex = readerIndex;
         return this;
     }

     @Override
     public ByteBuf resetReaderIndex() {
         readerIndex(markedReaderIndex);
```

```
        return this;
        }

    @Override
    public ByteBuf markWriterIndex() {
        markedWriterIndex = writerIndex;
        return this;
    }

    @Override
    public ByteBuf resetWriterIndex() {
        writerIndex = markedWriterIndex;
        return this;
    }

    ...

}
```

上面代码中这些 API 的功能我们已经介绍过了。下面我们再来看几个读写操作的 API，具体源码如下：

```
public abstract class AbstractByteBuf extends ByteBuf {

    ...

    @Override
    public byte readByte() {
        checkReadableBytes0(1);
        int i = readerIndex;
        byte b = _getByte(i);
        readerIndex = i + 1;
        return b;
    }

    @Override
    public ByteBuf writeByte(int value) {
        ensureAccessible();
        ensureWritable0(1);
        _setByte(writerIndex++, value);
        return this;
    }

    @Override
    public byte getByte(int index) {
        checkIndex(index);
        return _getByte(index);
    }


    protected abstract void _setByte(int index, int value);

    protected abstract byte _getByte(int index);

    ...
}
```

我们看到，上面的代码中 readByte()方法和 getByte()方法都调用了一个抽象的_getByte()，这个方法在 AbstractByteBuf 的子类中实现。在 writeByte()方法中有调用一个抽象的_setByte()方法，这个方法同样也是在子类中实现。

10.1.2 ByteBuf 的基本分类

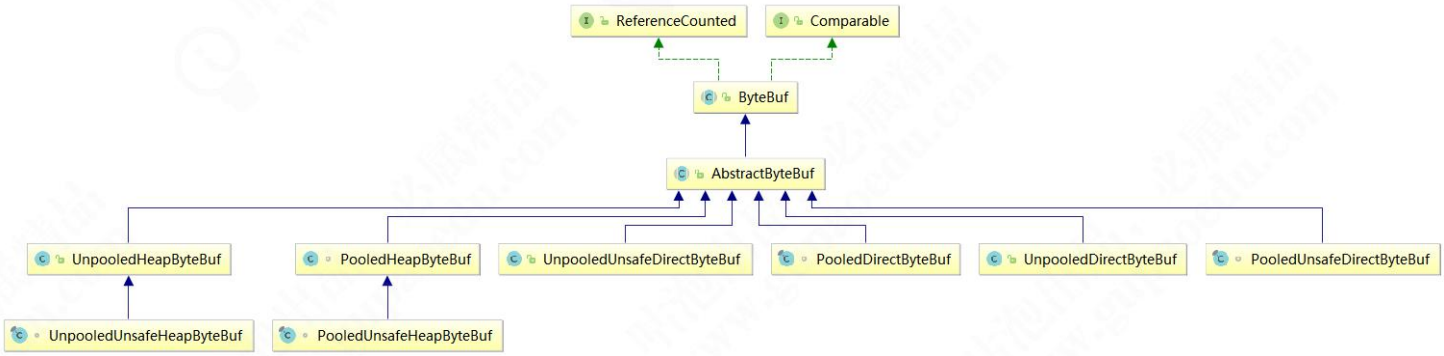
AbstractByteBuf 之下有众多子类，大致可以从三个维度来进行分类，分别如下：

Pooled：池化内存，就是从预先分配好的内存空间中提取一段连续内存封装成一个 ByteBuf 分给应用程序使用。

Unsafe：是 JDK 底层的一个负责 IO 操作的对象，可以直接拿到对象的内存地址，基于内存地址进行读写操作。

Direct：堆外内存，是直接调用 JDK 的底层 API 进行物理内存分配，不在 JVM 的堆内存中，需要手动释放。

综上所述，其实 ByteBuf 一共会有六种组合：Pooled 池化内存和 Unpooled 非池化内存；Unsafe 和非 Unsafe；Heap 堆内存和 Direct 堆外内存。下图是 ByteBuf 最重要的继承关系类结构图，通过命名就能一目了然：



ByteBuf 最基本的读写 API 操作在 AbstractByteBuf 中已经实现了，其众多子类采用不同的策略来分配内存空间，下面对重要的几个子类总结如下：

类	解释
PooledHeapByteBuf	池化的堆内缓冲区
PooledUnsafeHeapByteBuf	池化的 Unsafe 堆内缓冲区
PooledDirectByteBuf	池化的直接(堆外)缓冲区
PooledUnsafeDirectByteBuf	池化的 Unsafe 直接(堆外)缓冲区
UnpooledHeapByteBuf	非池化的堆内缓冲区
UnpooledUnsafeHeapByteBuf	非池化的 Unsafe 堆内缓冲区
UnpooledDirectByteBuf	非池化的直接(堆外)缓冲区

UnpooledUnsafeDirectByteBuf	非池化的 Unsafe 直接(堆外)缓冲区

下面我们来看几个类的基本结构，对比一下：

10.2 ByteBufAllocator 内存管理器

Netty 中内存分配有一个最顶层的抽象就是 ByteBufAllocator，负责分配所有 ByteBuf 类型的内存。功能其实不是很多，主要有以下几个重要的 API：

方法	解释
buffer()	分配一块内存，自动判断是否分配堆内内存或者堆外内存。
ioBuffer()	尽可能地分配一块堆外直接内存，如果系统不支持则分配堆内内存。
heapBuffer()	分配一块堆内内存。
directBuffer()	分配一块堆外内存。
compositeBuffer()	组合分配，把多个 ByteBuf 组合到一起变成一个整体。

到这里有些小伙伴可能会有疑问，以上 API 中为什么没有前面提到的 8 中类型的内存分配 API？下面我们来看 ByteBufAllocator 的基本实现类 AbstractByteBufAllocator ,重点分析主要 API 的基本实现 ,比如 buffer()方法源码如下：

```
public abstract class AbstractByteBufAllocator implements ByteBufAllocator {
    ...

    public ByteBuf buffer() {
       if (directByDefault) {
           return directBuffer();
       }
       return heapBuffer();
    }

    ...
}
```

我们发现 buffer()方法中做了判断 ,是否默认支持 directBuffer ,如果支持则分配 directBuffer ,否则分配 heapBuffer。

下面分别来看一下 directBuffer()方法和 heapBuffer()方法的实现，先来看 directBuffer()方法：

```
public abstract class AbstractByteBufAllocator implements ByteBufAllocator {
    ...

    @Override
    public ByteBuf directBuffer() {
```

```
        return directBuffer(DEFAULT_INITIAL_CAPACITY, Integer.MAX_VALUE);
    }

    @Override
    public ByteBuf directBuffer(int initialCapacity) {
        return directBuffer(initialCapacity, Integer.MAX_VALUE);
    }

    @Override
    public ByteBuf directBuffer(int initialCapacity, int maxCapacity) {
        if (initialCapacity == 0 && maxCapacity == 0) {
            return emptyBuf;
        }
        validate(initialCapacity, maxCapacity);
        return newDirectBuffer(initialCapacity, maxCapacity);
    }

    ...
}
```

directBuffer()方法有多个重载方法，最终会调用 newDirectBuffer()方法，我继续跟进到 newDirectBuffer()方法中：

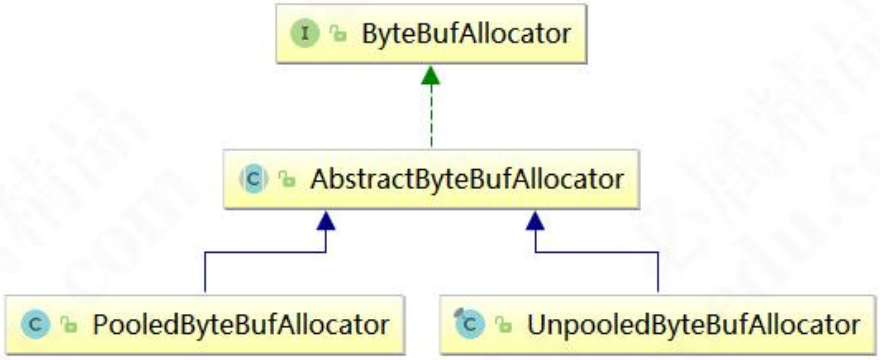
```
public abstract class AbstractByteBufAllocator implements ByteBufAllocator {
    ...

    protected abstract ByteBuf newDirectBuffer(int initialCapacity, int maxCapacity);
    ...
}
```

我们发现 newDirectBuffer()方法其实是一个抽象方法，最终，交给 AbstractByteBufAllocator 的子类来实现。同理，我们再来看 heapBuffer()方法的源码：

```
public abstract class AbstractByteBufAllocator implements ByteBufAllocator {
    ...

        @Override
        public ByteBuf heapBuffer() {
            return heapBuffer(DEFAULT_INITIAL_CAPACITY, Integer.MAX_VALUE);
        }

        @Override
        public ByteBuf heapBuffer(int initialCapacity) {
            return heapBuffer(initialCapacity, Integer.MAX_VALUE);
        }

        @Override
        public ByteBuf heapBuffer(int initialCapacity, int maxCapacity) {
            if (initialCapacity == 0 && maxCapacity == 0) {
                return emptyBuf;
            }
            validate(initialCapacity, maxCapacity);
            return newHeapBuffer(initialCapacity, maxCapacity);
        }

       protected abstract ByteBuf newHeapBuffer(int initialCapacity, int maxCapacity);
    ...
}
```

我们发现 heapBuffer()方法最终是调用 newHeapBuffer()方法，而 newHeapBuffer()方法也是抽象方法，具体交给

AbstractByteBufAllocator 的子类实现。AbstractByteBufAllocator 的子类主要有两个：PooledByteBufAllocator 和 UnpooledByteBufAllocator，下面我们来看 AbstractByteBufAllocator 子类实现的类结构图：



分析到这里，其实我们还只知道 directBuffer、heapBuffer 和 pooled、unpooled 的分配规则，那 unsafe 和非 unsafe 是如何判别的呢？其实，是 Netty 自动帮我们判别的，如果操作系统底层支持 unsafe 那就采用 unsafe 读写，否则采用非 unsafe 读写。我们可以从 UnpooledByteBufAllocator 的源码中验证一下，来看源码：

```
public final class UnpooledByteBufAllocator extends AbstractByteBufAllocator {

    ...

    @Override
    protected ByteBuf newHeapBuffer(int initialCapacity, int maxCapacity) {
        return PlatformDependent.hasUnsafe() ? new UnpooledUnsafeHeapByteBuf(this, initialCapacity, maxCapacity)
                : new UnpooledHeapByteBuf(this, initialCapacity, maxCapacity);
    }

    @Override
    protected ByteBuf newDirectBuffer(int initialCapacity, int maxCapacity) {
        ByteBuf buf = PlatformDependent.hasUnsafe() ?
                UnsafeByteBufUtil.newUnsafeDirectByteBuf(this, initialCapacity, maxCapacity) :
                new UnpooledDirectByteBuf(this, initialCapacity, maxCapacity);

        return disableLeakDetector ? buf : toLeakAwareBuffer(buf);
    }

    ...
}
```

我们发现在 newHeapBuffer()方法和 newDirectBuffer()方法中，分配内存判断 PlatformDependent 是否支持 Unsafa，如果支持则创建 Unsafe 类型的 Buffer，否则创建非 Unsafe 类型的 Buffer。由 Netty 帮我们自动判断了。

10.3 Unpooled 非池化内存分配

10.3.1 堆内内存的分配

现在我们来看 UnpooledByteBufAllocator 的内存分配原理。首先 ,来看 heapBuffer 的分配逻辑 ,进入 newHeapBuffer() 方法源码：

```
public final class UnpooledByteBufAllocator extends AbstractByteBufAllocator {

    ...

    @Override
    protected ByteBuf newHeapBuffer(int initialCapacity, int maxCapacity) {
        return PlatformDependent.hasUnsafe() ? new UnpooledUnsafeHeapByteBuf(this, initialCapacity, maxCapacity)
                : new UnpooledHeapByteBuf(this, initialCapacity, maxCapacity);
    }

    ...
}
```

通过调用 PlatformDependent.hasUnsafe() 方法来判断操作系统是否支持 Unsafe，如果支持 Unsafe 则创建 UnpooledUnsafeHeapByteBuf 类，否则创建 UnpooledHeapByteBuf 类。我们先进入 UnpooledUnsafeHeapByteBuf 的构造器看看会进行哪些操作？

```
final class UnpooledUnsafeHeapByteBuf extends UnpooledHeapByteBuf {

    UnpooledUnsafeHeapByteBuf(ByteBufAllocator alloc, int initialCapacity, int maxCapacity) {
        super(alloc, initialCapacity, maxCapacity);
    }
}
```

我们发现 UnpooledUnsafeHeapByteBuf 继承了 UnpooledHeapByte，并且在 UnpooledUnsafeHeapByteBuf 的构造器中直接调用了 super 也就是 UnpooledHeapByte 的构造方法，我们进入 UnpooledHeapByte 的构造器代码：

```
public class UnpooledHeapByteBuf extends AbstractReferenceCountedByteBuf {

    private final ByteBufAllocator alloc;
    byte[] array;
    private ByteBuffer tmpNioBuf;
    protected UnpooledHeapByteBuf(ByteBufAllocator alloc, int initialCapacity, int maxCapacity) {
        this(alloc, new byte[initialCapacity], 0, 0, maxCapacity);
    }

    protected UnpooledHeapByteBuf(ByteBufAllocator alloc, byte[] initialArray, int maxCapacity) {
        this(alloc, initialArray, 0, initialArray.length, maxCapacity);
    }

    private UnpooledHeapByteBuf(
            ByteBufAllocator alloc, byte[] initialArray, int readerIndex, int writerIndex, int maxCapacity) {

        super(maxCapacity);

        ...

        this.alloc = alloc;
```

```
        setArray(initialArray);
        setIndex(readerIndex, writerIndex);
    }

    ...
}
```

有一段关键方法就是 setArray()方法，里面的实现也非常简单，就是把默认分配的数组 new byte[initialCapacity]赋值给全局变量 array。

```
    private void setArray(byte[] initialArray) {
        array = initialArray;
        tmpNioBuf = null;
    }
```

紧接着就是调用了 setIndex()方法：

```
    public ByteBuf setIndex(int readerIndex, int writerIndex) {
            if (readerIndex < 0 || readerIndex > writerIndex || writerIndex > capacity()) {
                throw new IndexOutOfBoundsException(String.format(
                        "readerIndex: %d, writerIndex: %d (expected: 0 <= readerIndex <= writerIndex <= capacity(%d))",
                        readerIndex, writerIndex, capacity()));
            }
            setIndex0(readerIndex, writerIndex);
            return this;
     }

     final void setIndex0(int readerIndex, int writerIndex) {
            this.readerIndex = readerIndex;
            this.writerIndex = writerIndex;
      }
```

最终在 setIndex0()方法中初始化 readerIndex 和 writerIndex。

既然，UnpooledUnsafeHeapByteBuf 和 UnpooledHeapByteBuf 调用的都是 UnpooledHeapByteBuf 的构造方法，那么它们之间到底有什么区别呢？其实根本区别在于 IO 的读写，我们可以分别来看一下它们的 getByte()方法了解二者的区别。先来看 UnpooledHeapByteBuf 的 getByte()方法实现：

```
    public byte getByte(int index) {
        ensureAccessible();
        return _getByte(index);
    }

    @Override
    protected byte _getByte(int index) {
        return HeapByteBufUtil.getByte(array, index);
    }
```

我们可以看到最终调用的是 HeapByteBufUtil 的 getByte()方法，继续跟进去：

```
    final class HeapByteBufUtil {

        static byte getByte(byte[] memory, int index) {
            return memory[index];
        }
        ...
    }
```

这个操作就非常简单，直接用数组索引取值。再来看 UnpooledUnsafeHeapByteBuf 的 getByte()方法实现：

```
      public byte getByte(int index) {
          checkIndex(index);
          return _getByte(index);
       }

       @Override
       protected byte _getByte(int index) {
          return UnsafeByteBufUtil.getByte(array, index);
       }
```

我们可以看到最终调用的是 UnsafeByteBufUtil 的 getByte()方法，继续跟进去：

```
   final class UnsafeByteBufUtil{

       static byte getByte(byte[] array, int index) {
          return PlatformDependent.getByte(array, index);
       }
       ...
   }
```

通过这样一对比我们基本已经了解 UnpooledUnsafeHeapByteBuf 和 UnpooledHeapByteBuf 的区别了。

10.3.2 堆外内存的分配

我还是回到 UnpooledByteBufAllocator 的 newDirectBuffer()方法：

```
public final class UnpooledByteBufAllocator extends AbstractByteBufAllocator {

    ...

   @Override
   protected ByteBuf newDirectBuffer(int initialCapacity, int maxCapacity) {
       ByteBuf buf = PlatformDependent.hasUnsafe() ?
              UnsafeByteBufUtil.newUnsafeDirectByteBuf(this, initialCapacity, maxCapacity) :
              new UnpooledDirectByteBuf(this, initialCapacity, maxCapacity);

       return disableLeakDetector ? buf : toLeakAwareBuffer(buf);
   }

    ...
}
```

从上面代码可以看出，如果支持 Unsafe 则调用 UnsafeByteBufUtil.newUnsafeDirectByteBuf()方法，否则创建 UnpooledDirectByteBuf 类。我们先看一下 UnpooledDirectByteBuf 的构造器：

```
public class UnpooledDirectByteBuf extends AbstractReferenceCountedByteBuf {

   private final ByteBufAllocator alloc;

   private ByteBuffer buffer;
   private ByteBuffer tmpNioBuf;
   private int capacity;
   private boolean doNotFree;
```

```
    protected UnpooledDirectByteBuf(ByteBufAllocator alloc, int initialCapacity, int maxCapacity) {
        super(maxCapacity);
        ...
        this.alloc = alloc;
        setByteBuffer(ByteBuffer.allocateDirect(initialCapacity));
    }
    ...
}
```

首先调用了 ByteBuffer.allocateDirect.allocateDirect()通过 JDK 底层分配一个直接缓冲区，然后传给 setByteBuffer()方法，

我们继续跟进：

```
    private void setByteBuffer(ByteBuffer buffer) {
        ByteBuffer oldBuffer = this.buffer;
        if (oldBuffer != null) {
            if (doNotFree) {
                doNotFree = false;
            } else {
                freeDirect(oldBuffer);
            }
        }

        this.buffer = buffer;
        tmpNioBuf = null;
        capacity = buffer.remaining();
    }
```

我们可以看到 setByteBuffer()方法中主要就是做了一次赋值。

下面我们继续来 UnsafeByteBufUtil.newUnsafeDirectByteBuf()方法，看它的逻辑：

```
final class UnsafeByteBufUtil {
    ...
    static UnpooledUnsafeDirectByteBuf newUnsafeDirectByteBuf(
            ByteBufAllocator alloc, int initialCapacity, int maxCapacity) {
        if (PlatformDependent.useDirectBufferNoCleaner()) {
            return new UnpooledUnsafeNoCleanerDirectByteBuf(alloc, initialCapacity, maxCapacity);
        }
        return new UnpooledUnsafeDirectByteBuf(alloc, initialCapacity, maxCapacity);
    }
    ...
}
```

在这个方法中返回了一个 UnpooledUnsafeDirectByteBuf 对象，关于 UnpooledUnsafeNoCleanerDirectByteBuf 我们

再后续章节再详细分析。下面进入 UnpooledUnsafeDirectByteBuf 的构造器：

```
public class UnpooledUnsafeDirectByteBuf extends AbstractReferenceCountedByteBuf {
        protected UnpooledUnsafeDirectByteBuf(ByteBufAllocator alloc, int initialCapacity, int maxCapacity) {
            super(maxCapacity);
            ...

            this.alloc = alloc;
            setByteBuffer(allocateDirect(initialCapacity), false);
        }

        ...
}
```

它的逻辑和 UnpooledDirectByteBuf 构造器的逻辑是相似的，所以我们关注 setByteBuffer()方法，看跟

UnpooledDirectByteBuf 的 setByteBuffer()方法有什么区别，我跟进去：

```
final void setByteBuffer(ByteBuffer buffer, boolean tryFree) {
        if (tryFree) {
            ByteBuffer oldBuffer = this.buffer;
            if (oldBuffer != null) {
                if (doNotFree) {
                    doNotFree = false;
                } else {
                    freeDirect(oldBuffer);
                }
            }
        }
        this.buffer = buffer;
        memoryAddress = PlatformDependent.directBufferAddress(buffer);
        tmpNioBuf = null;
        capacity = buffer.remaining();
    }
```

同样还是先把创建从 JDK 底层创建好的 buffer 保存，接下来有个很重要的操作就是调用了 PlatformDependent.directBufferAddress()方法获取的 buffer 真实的内存地址，并保存到 memoryAddress 变量中。

我们进入 PlatformDependent.directBufferAddress()方法一探究竟。

```
    public static long directBufferAddress(ByteBuffer buffer) {
        return PlatformDependent0.directBufferAddress(buffer);
    }
```

继续进入 PlatformDependent0 的 directBufferAddress()方法：

```
    static long directBufferAddress(ByteBuffer buffer) {
        return getLong(buffer, ADDRESS_FIELD_OFFSET);
    }
```

我们继续看 getLong()方法：

```
    private static long getLong(Object object, long fieldOffset) {
        return UNSAFE.getLong(object, fieldOffset);
    }
```

可以看到，调用了 UNSAFE 的 getLong()方法,这个方法是一个 native 方法。它是直接通过 buffer 的内存地址加上一个偏移量去取数据，。到这里我们已经基本清楚 UnpooledUnsafeDirectByteBuf 和 UnpooledDirectByteBuf 的区别，非 unsafe 是通过数组的下标取数据，而 unsafe 是直接操作内存地址，相对于非 unsafe 来说效率当然要更高。

10.4 Pooled 池化内存分配

10.4.1 PooledByteBufAllocator 简述

现在开始，我们来分析 Pooled 池化内存的分配原理。我们首先还是找到 AbstractByteBufAllocator 的子类

PooledByteBufAllocator 实现的分配内存的两个方法 newDirectBuffer()方法和 newHeapBuffer()方法：

```
public class PooledByteBufAllocator extends AbstractByteBufAllocator {
    ...

    @Override
    protected ByteBuf newHeapBuffer(int initialCapacity, int maxCapacity) {
        PoolThreadCache cache = threadCache.get();
        PoolArena<byte[]> heapArena = cache.heapArena;

        ByteBuf buf;
        if (heapArena != null) {
            buf = heapArena.allocate(cache, initialCapacity, maxCapacity);
        } else {
            buf = new UnpooledHeapByteBuf(this, initialCapacity, maxCapacity);
        }

        return toLeakAwareBuffer(buf);
    }

    @Override
    protected ByteBuf newDirectBuffer(int initialCapacity, int maxCapacity) {
        PoolThreadCache cache = threadCache.get();
        PoolArena<ByteBuffer> directArena = cache.directArena;

        ByteBuf buf;
        if (directArena != null) {
            buf = directArena.allocate(cache, initialCapacity, maxCapacity);
        } else {
            if (PlatformDependent.hasUnsafe()) {
                buf = UnsafeByteBufUtil.newUnsafeDirectByteBuf(this, initialCapacity, maxCapacity);
            } else {
                buf = new UnpooledDirectByteBuf(this, initialCapacity, maxCapacity);
            }
        }

        return toLeakAwareBuffer(buf);
    }

    ...
}
```

我们发现这两个方法大体结构上是一样的，我们以 newDirectBuffer()方法为例，简单地分析一下：

```
    @Override
    protected ByteBuf newDirectBuffer(int initialCapacity, int maxCapacity) {
        PoolThreadCache cache = threadCache.get();
        PoolArena<ByteBuffer> directArena = cache.directArena;

        ByteBuf buf;
        if (directArena != null) {
            buf = directArena.allocate(cache, initialCapacity, maxCapacity);
        } else {
            if (PlatformDependent.hasUnsafe()) {
                buf = UnsafeByteBufUtil.newUnsafeDirectByteBuf(this, initialCapacity, maxCapacity);
            } else {
                buf = new UnpooledDirectByteBuf(this, initialCapacity, maxCapacity);
            }
        }

        return toLeakAwareBuffer(buf);
    }
```

首先，我们看到它是通过 threadCache.get()拿到一个类型为 PoolThreadCache 的 cache 对象，然后，通过 cache 拿到 directArena 对象，最终会调用 directArena.allocate()方法分配 ByteBuf。这个地方大家可能会看得有点懵，我们来详细分析一下。我们跟进到 threadCache 对象其实是 PoolThreadLocalCache 类型的变量，跟进到 PoolThreadLocalCache 的源码：

```
final class PoolThreadLocalCache extends FastThreadLocal<PoolThreadCache> {

        @Override
        protected synchronized PoolThreadCache initialValue() {
            //从 heapArenas 中获得一个使用率最少的 Arena
            final PoolArena<byte[]> heapArena = leastUsedArena(heapArenas);
            //从 directArenas 中获得一个使用率最少的 Arena
            final PoolArena<ByteBuffer> directArena = leastUsedArena(directArenas);

            return new PoolThreadCache(
                    heapArena, directArena, tinyCacheSize, smallCacheSize, normalCacheSize,
                    DEFAULT_MAX_CACHED_BUFFER_CAPACITY, DEFAULT_CACHE_TRIM_INTERVAL);
        }

         ...
}
```

从名字来看，我们发现 PoolThreadLocalCache 的 initialValue()方法就是用来初始化 PoolThreadLocalCache 的。首先就调用了 leastUsedArena()方法分别获得类型为 PoolArena 的 heapArena 和 directArena 对象。然后把 heapArena 和 directArena 对象作为参数传递到了 PoolThreadCache 的构造器中。那么 heapArena 和 directArena 对象是在哪里初始化的呢？我们查找一下发现在 PooledByteBufAllocator 的构造方法中调用 newArenaArray()方法给 heapArenas 和 directArenas 赋值了。

```
public PooledByteBufAllocator(boolean preferDirect, int nHeapArena, int nDirectArena, int pageSize, int maxOrder,
                              int tinyCacheSize, int smallCacheSize, int normalCacheSize) {

        ...

        if (nHeapArena > 0) {
            heapArenas = newArenaArray(nHeapArena);

            ...

        } else {
            heapArenas = null;
            heapArenaMetrics = Collections.emptyList();
        }

        if (nDirectArena > 0) {
            directArenas = newArenaArray(nDirectArena);

            ...

        }
```

```
        ...
}
```

进入到 newArenaArray()方法：

```
    private static <T> PoolArena<T>[] newArenaArray(int size) {
        return new PoolArena[size];
    }
```

其实就是创建了一个固定大小的 PoolArena 数组，数组大小由传入的参数 nHeapArena 和 nDirectArena 来决定。我们再回到 PooledByteBufAllocator 的构造器源码，看 nHeapArena 和 nDirectArena 是怎么初始化的，我们找到 PooledByteBufAllocator 的重载构造器：

```
    public PooledByteBufAllocator(boolean preferDirect) {
        this(preferDirect, DEFAULT_NUM_HEAP_ARENA, DEFAULT_NUM_DIRECT_ARENA, DEFAULT_PAGE_SIZE, DEFAULT_MAX_ORDER);
    }
```

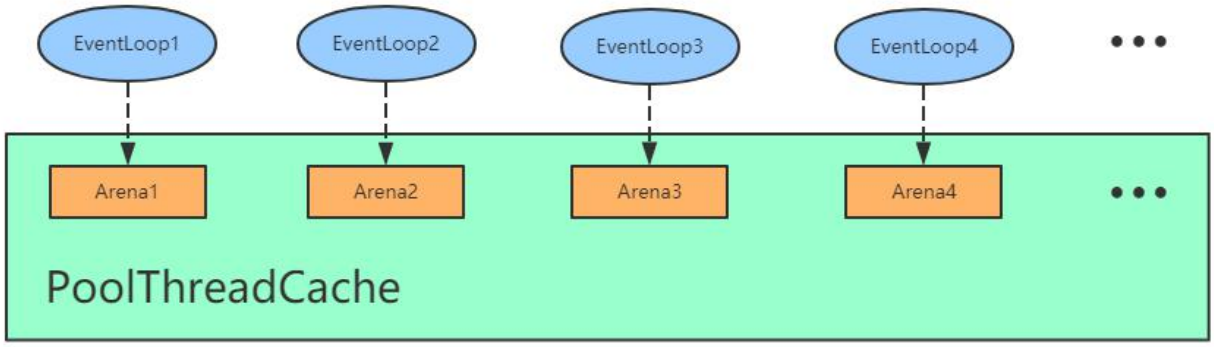
我没发现，nHeapArena 和 nDirectArena 是通过 DEFAULT_NUM_HEAP_ARENA 和 DEFAULT_NUM_DIRECT_ARENA 这两个常量默认赋值的。再继续跟进常量的定义：

```
    final int defaultMinNumArena = runtime.availableProcessors() * 2;

    DEFAULT_NUM_HEAP_ARENA = Math.max(0,
                SystemPropertyUtil.getInt(
                        "io.netty.allocator.numHeapArenas",
                        (int) Math.min(
                                defaultMinNumArena,
                                runtime.maxMemory() / defaultChunkSize / 2 / 3)));
    DEFAULT_NUM_DIRECT_ARENA = Math.max(0,
                SystemPropertyUtil.getInt(
                        "io.netty.allocator.numDirectArenas",
                        (int) Math.min(
                                defaultMinNumArena,
                                PlatformDependent.maxDirectMemory() / defaultChunkSize / 2 / 3)));
```

到这里为止，我们才知道 nHeapArena 和 nDirectArena 的默认赋值。默认是分配 CPU 核数*2，也就是把 defaultMinNumArena 的值赋值给 nHeapArena 和 nDirectArena。对于 CPU 核数*2 大家应该有印象，我们在前面的章节中，介绍 EventLoopGroup 给分配线程时默认线程数也是 CPU 核数*2,。那么，Netty 为什么要这样设计呢？其实，主要目的就是保证 Netty 中的每一个任务线程都可以有一个独享的 Arena，保证在每个线程分配内存的时候是不用加锁的。

基于上面的分析，我们知道 Arena 有 heapArean 和 DirectArena，这里我们统称为 Arena。假设我们有四个线程，那么对应会分配四个 Arean。在创建 BtyeBuf 的时候，首先通过 PoolThreadCache 获取 Arena 对象并赋值给其成员变量，然后，每个线程通过 PoolThreadCache 调用 get 方法的时候会拿到它底层的 Arena，也就是说 EventLoop1 拿到 Arena1，

EventLoop2 拿到 Arena2，以此类推，如下图所示：



那么 PoolThreadCache 除了可以旨在 Arena 上进行内存分配，还可以在它底层维护的 ByteBuf 缓存列表进行分配。举个例子：我们通过 PooledByteBufAllocator 去创建了一个 1024 字节大小的 ByteBuf，当我们用完释放之后，我们可能在其他地方会继续分配 1024 字节大小的 ByteBuf。这个时候，其实不需要在 Arena 上进行内存分配，而是直接通过 PoolThreadCache 中维护的 ByteBuf 的缓存列表直接拿过来返回。那么，在 PooledByteBufAllocator 中维护三种规格大小的缓存列表，分别是三个值 tinyCacheSize、smallCacheSize、normalCacheSize：

```
public class PooledByteBufAllocator extends AbstractByteBufAllocator {

      ...

      DEFAULT_TINY_CACHE_SIZE = SystemPropertyUtil.getInt("io.netty.allocator.tinyCacheSize", 512);
      DEFAULT_SMALL_CACHE_SIZE = SystemPropertyUtil.getInt("io.netty.allocator.smallCacheSize", 256);
      DEFAULT_NORMAL_CACHE_SIZE = SystemPropertyUtil.getInt("io.netty.allocator.normalCacheSize", 64);


   private final int tinyCacheSize;
   private final int smallCacheSize;
   private final int normalCacheSize;
      ...


 public PooledByteBufAllocator(boolean preferDirect, int nHeapArena, int nDirectArena, int pageSize, int maxOrder) {
      this(preferDirect, nHeapArena, nDirectArena, pageSize, maxOrder,
             DEFAULT_TINY_CACHE_SIZE, DEFAULT_SMALL_CACHE_SIZE, DEFAULT_NORMAL_CACHE_SIZE);
   }

   public PooledByteBufAllocator(boolean preferDirect, int nHeapArena, int nDirectArena, int pageSize, int maxOrder,
                                 int tinyCacheSize, int smallCacheSize, int normalCacheSize) {
      super(preferDirect);
      threadCache = new PoolThreadLocalCache();
      this.tinyCacheSize = tinyCacheSize;
      this.smallCacheSize = smallCacheSize;
      this.normalCacheSize = normalCacheSize;
      final int chunkSize = validateAndCalculateChunkSize(pageSize, maxOrder);

      ...
   }
```

```
}
```

我们看到，在 PooledByteBufAllocator 的构造器中，分别赋值了 tinyCacheSize=512，smallCacheSize=256，normalCacheSize=64。通过这样一种方式，Netty 中给我们预创建固定规格的内存池，大大提高了内存分配的性能。

10.4.2 DirectArena 内存分配流程

Arena 分配内存的基本流程有三个步骤：

1、从对象池里拿到 PooledByteBuf 进行复用；

2、从缓存中进行内存分配；

3、从内存堆里进行内存分配。

我们以 directBuff 为例，首先来看从对象池里拿到 PooledByteBuf 进行复用的情况，我们依旧跟进到 PooledByteBufAllocator 的 newDirectBuffer()方法:

```
    @Override
    protected ByteBuf newDirectBuffer(int initialCapacity, int maxCapacity) {
        PoolThreadCache cache = threadCache.get();
        PoolArena<ByteBuffer> directArena = cache.directArena;

        ByteBuf buf;
        if (directArena != null) {
            buf = directArena.allocate(cache, initialCapacity, maxCapacity);
        } else {
            ...
        }

        return toLeakAwareBuffer(buf);
    }
```

上面的 PoolArena 我们已经清楚，现在我们直接跟进到 PoolArena 的 allocate()方法：

```
    PooledByteBuf<T> allocate(PoolThreadCache cache, int reqCapacity, int maxCapacity) {
        PooledByteBuf<T> buf = newByteBuf(maxCapacity);
        allocate(cache, buf, reqCapacity);
        return buf;
    }
```

在这个地方其实就非常清晰了，首先调用了 newByteBuf()方法去拿到一个 PooledByteBuf 对象，接下来通过 allocate()方法在线程私有的 PoolThreadCache 中分配一块内存，然后 buf 里面的内存地址之类的值进行初始化。我们可以跟进到 newByteBuf()看一下，选择 DirectArena 对象：

```
protected PooledByteBuf<ByteBuffer> newByteBuf(int maxCapacity) {
    if (HAS_UNSAFE) {
        return PooledUnsafeDirectByteBuf.newInstance(maxCapacity);
    } else {
```

```
        return PooledDirectByteBuf.newInstance(maxCapacity);
    }
}
```

我们发现首先判断是否支持 UnSafe，默认情况下一般是支持 Unsafe 的，所以我们继续进入到 PooledUnsafeDirectByteBuf 的 newInstance()方法：

```
final class PooledUnsafeDirectByteBuf extends PooledByteBuf<ByteBuffer> {
   private static final Recycler<PooledUnsafeDirectByteBuf> RECYCLER = new Recycler<PooledUnsafeDirectByteBuf>() {
       @Override
       protected PooledUnsafeDirectByteBuf newObject(Handle<PooledUnsafeDirectByteBuf> handle) {
           return new PooledUnsafeDirectByteBuf(handle, 0);
       }
   };

   static PooledUnsafeDirectByteBuf newInstance(int maxCapacity) {
       PooledUnsafeDirectByteBuf buf = RECYCLER.get();
       buf.reuse(maxCapacity);
       return buf;
   }
    ...
}
```

顾名思义，我看到首先就是从 RECYCLER（也就是内存回收站）对象的 get()方法拿到一个 buf。从上面的代码片段来看，RECYCLER 对象实现了一个 newObject()方法，当回收站里面没有可用的 buf 时就会创建一个新的 buf。因为获取到的 buf 可能是回收站里面拿出来的，复用前需要重置。因此，继续往下看就会调用 buf 的 reuse()方法：

```
       final void reuse(int maxCapacity) {
           maxCapacity(maxCapacity);
           setRefCnt(1);
           setIndex0(0, 0);
           discardMarks();
       }
```

我们发现 reuse()就是对所有的参数重新归为初始状态。到这里我们应该已经清楚从内存池获取 buf 对象的全过程。那么接下来，再回到 PoolArena 的 allocate()方法，看看真实的内存是如何分配出来的？buf 的内存分配主要有两种情况，分别是：从缓存中进行内存分配和从内存堆里进行内存分配。我们来看代码，进入 allocate()方法具体逻辑：

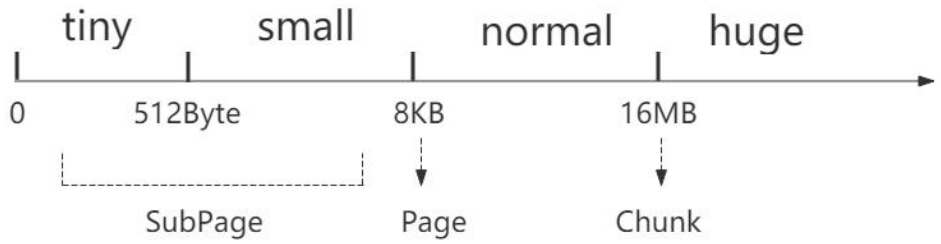
```
       private void allocate(PoolThreadCache cache, PooledByteBuf<T> buf, final int reqCapacity) {
           ...
           if (normCapacity <= chunkSize) {
               if (cache.allocateNormal(this, buf, reqCapacity, normCapacity)) {
                   // was able to allocate out of the cache so move on
                   return;
               }
               allocateNormal(buf, reqCapacity, normCapacity);
           } else {
               // Huge allocations are never served via the cache so just call allocateHuge
               allocateHuge(buf, reqCapacity);
           }
       }
```

这段代码逻辑看上去非常复杂，其实我们省略掉的逻辑基本上都是判断不同规格大小，从其对应的缓存中获取内存。

如果所有规格都不满足，那就直接调用 allocateHuge()方法进行真实的内存分配。

10.4.3 内存池的内存规格

在前面的源码分析过程中，关于内存规格大小我们应该还有些印象。其实在 Netty 内存池中主要设置了四种规格大小的内存：tiny 是指 0-512Byte 之间的规格大小，small 是指 512Byte-8KB 之间的规格大小，normal 是指 8KB-16MB 之间的规格大小，huge 是指 16MB 以上。为什么 Netty 会选择这些值作为一个分界点呢？其实在 Netty 底层还有一个内存单位的封装，为了更高效地管理内存，避免内存浪费，把每一个区间的内存规格由做了细分。默认情况下，Netty 将内存规格划分为 4 个部分。Netty 中所有的内存申请是以 Chunk 为单位向内存申请的，大小为 16M，后续的所有内存分配都是在这个 Chunk 里面的操作。8K 对应的是一个 Page，一个 Chunk 会以 Page 为单位进行切分，8K 对应 Chunk 被划分为 2048 个 Page。小于 8K 的对应的是 SubPage。例如：我们申请的一段内存空间只有 1K，却给我们分配了一个 Page，显然另外 7K 就会被浪费，所以就继续把 Page 进行划分，来节省空间。如下图所示：



至此，小伙伴们应该已经非常清楚 Netty 的内存池缓存管理机制了。

10.4.4 命中缓存的分配

前面我们简单分析了 directArena 内存分配大概流程，知道其先命中缓存，如果命中不到，则区分配一款连续内存。现在开始带大家剖析命中缓存的相关逻辑。前面我们也讲到 PoolThreadCache 中维护了三个缓存数组(实际上是六个，这里仅仅以 Direct 为例，Heap 类型的逻辑是一样的)：tinySubPageDirectCaches，smallSubPageDirectCaches，和 normalDirectCaches 分别代表 tiny 类型，small 类型和 normal 类型的缓存数组)。这三个数组保存在 PoolThreadCache 的成员变量中:

```
final class PoolThreadCache {
    ...
    private final MemoryRegionCache<ByteBuffer>[] tinySubPageDirectCaches;
    private final MemoryRegionCache<ByteBuffer>[] smallSubPageDirectCaches;
    private final MemoryRegionCache<ByteBuffer>[] normalDirectCaches;
     ...
}
```

其实是在构造方法中进行了初始化:

```
final class PoolThreadCache {
    ...
    PoolThreadCache(PoolArena<byte[]> heapArena, PoolArena<ByteBuffer> directArena,
                    int tinyCacheSize, int smallCacheSize, int normalCacheSize,
                    int maxCachedBufferCapacity, int freeSweepAllocationThreshold) {
        ...

        if (directArena != null) {
            tinySubPageDirectCaches = createSubPageCaches(
                    tinyCacheSize, PoolArena.numTinySubpagePools, SizeClass.Tiny);
            smallSubPageDirectCaches = createSubPageCaches(
                    smallCacheSize, directArena.numSmallSubpagePools, SizeClass.Small);

            numShiftsNormalDirect = log2(directArena.pageSize);
            normalDirectCaches = createNormalCaches(
                    normalCacheSize, maxCachedBufferCapacity, directArena);

            directArena.numThreadCaches.getAndIncrement();
         }
         ...
    }
    ...
}
```

我们以 tiny 类型为例跟到 createSubPageCaches 方法中:

```
    private static <T> MemoryRegionCache<T>[] createSubPageCaches(
            int cacheSize, int numCaches, SizeClass sizeClass) {
        if (cacheSize > 0) {
            @SuppressWarnings("unchecked")
            MemoryRegionCache<T>[] cache = new MemoryRegionCache[numCaches];
            for (int i = 0; i < cache.length; i++) {
                cache[i] = new SubPageMemoryRegionCache<T>(cacheSize, sizeClass);
            }
            return cache;
        } else {
            return null;
        }
    }
```

从代码中看出，其实就是创建了一个缓存数组，这个缓存数组的长度，也就是 numCaches，在不同的类型，这个长度不一样，tiny 类型长度是 32，small 类型长度为 4，normal 类型长度为 3。我们知道，缓存数组中每个节点代表一个缓存对象，里面维护了一个队列，队列大小由 PooledByteBufAllocator 类中的 tinyCacheSize，smallCacheSize，normalCacheSize 属性决定的。其中每个缓存对象，队列中缓存的 ByteBuf 大小是固定的，netty 将每种缓冲区类型分成了不同长度规格，而每个缓存中的队列缓存的 ByteBuf 的长度，都是同一个规格的长度，而缓冲区数组的长度，就是规

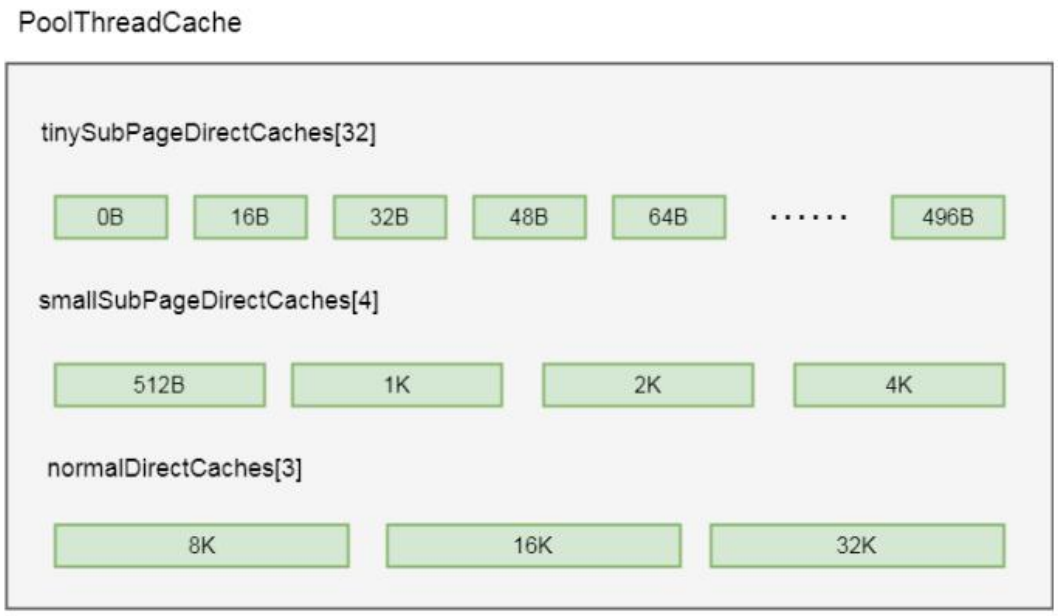
格的数量。比如 :在 tiny 类型中 ,Netty 将其长度分成 32 个规格，每个规格都是 16 的整数倍，也就是包含 0Byte, 16Byte, 32Byte, 48Byte, 64Byte, 80Byte, 96Byte……496Byte 总共 32 种规格，而在其缓存数组 tinySubPageDirectCaches 中，这每一种规格代表数组中的一个缓存对象缓存的 ByteBuf 的大小，我们以 tinySubPageDirectCaches[1]为例(这里下标选择 1 是因为下标为 0 代表的规格是 0Byte，其实就代表一个空的缓存，这里不进行举例)，在 tinySubPageDirectCaches[1]的缓存对象中所缓存的 ByteBuf 的缓冲区长度是 16Byte，在 tinySubPageDirectCaches[2]中缓存的 ByteBuf 长度都为 32Byte，以此类推, tinySubPageDirectCaches[31]中缓存的 ByteBuf 长度为 496Byte。其具体类型规则的配置如下:

tiny:总共 32 个规格，均是 16 的整数倍, 0Byte, 16Byte, 32Byte, 48Byte, 64Byte, 80Byte, 96Byte……496Byte ；

small:4 种规格, 512Byte, 1KB, 2KB, 4KB ；

nomal:3 种规格, 8KB, 16KB ，32KB。

如此，我们得出结论 PoolThreadCache 中缓存数组的数据结构如下图所示：



在基本了解缓存数组的数据结构之后，我们再继续剖析在缓冲中分配内存的逻辑，回到 PoolArena 的 allocate()方法中:

```
private void allocate(PoolThreadCache cache, PooledByteBuf<T> buf, final int reqCapacity) {
    //规格化
    final int normCapacity = normalizeCapacity(reqCapacity);
    if (isTinyOrSmall(normCapacity)) {
        int tableIdx;
        PoolSubpage<T>[] table;
        //判断是不是 tinty
        boolean tiny = isTiny(normCapacity);
        if (tiny) { // < 512
```

```
            //缓存分配
            if (cache.allocateTiny(this, buf, reqCapacity, normCapacity)) {
                return;
            }
            //通过 tinyIdx 拿到 tableIdx
            tableIdx = tinyIdx(normCapacity);
            //subpage 的数组
            table = tinySubpagePools;
        } else {
            if (cache.allocateSmall(this, buf, reqCapacity, normCapacity)) {
                return;
            }
            tableIdx = smallIdx(normCapacity);
            table = smallSubpagePools;
        }

        //拿到对应的节点
        final PoolSubpage<T> head = table[tableIdx];

        synchronized (head) {
            final PoolSubpage<T> s = head.next;
            //默认情况下，head 的 next 也是自身
            if (s != head) {
                assert s.doNotDestroy && s.elemSize == normCapacity;
                long handle = s.allocate();
                assert handle >= 0;
                s.chunk.initBufWithSubpage(buf, handle, reqCapacity);

                if (tiny) {
                    allocationsTiny.increment();
                } else {
                    allocationsSmall.increment();
                }
                return;
            }
        }
        allocateNormal(buf, reqCapacity, normCapacity);
        return;
    }
    if (normCapacity <= chunkSize) {
        //首先在缓存上进行内存分配
        if (cache.allocateNormal(this, buf, reqCapacity, normCapacity)) {
            //分配成功，返回
            return;
        }
        //分配不成功，做实际的内存分配
        allocateNormal(buf, reqCapacity, normCapacity);
    } else {
        //大于这个值，就不在缓存上分配
        allocateHuge(buf, reqCapacity);
    }
}
```

首先通过 normalizeCapacity 方法进行内存规格化，我们跟到 normalizeCapacity()方法中:

```
int normalizeCapacity(int reqCapacity) {
    if (reqCapacity < 0) {
        throw new IllegalArgumentException("capacity: " + reqCapacity + " (expected: 0+)");
    }
    if (reqCapacity >= chunkSize) {
        return reqCapacity;
    }
    //如果>tiny
```

```
    if (!isTiny(reqCapacity)) { // >= 512
        //找一个 2 的幂次方的数值，确保数值大于等于 reqCapacity
        int normalizedCapacity = reqCapacity;
        normalizedCapacity --;
        normalizedCapacity |= normalizedCapacity >>>  1;
        normalizedCapacity |= normalizedCapacity >>>  2;
        normalizedCapacity |= normalizedCapacity >>>  4;
        normalizedCapacity |= normalizedCapacity >>>  8;
        normalizedCapacity |= normalizedCapacity >>> 16;
        normalizedCapacity ++;

        if (normalizedCapacity < 0) {
            normalizedCapacity >>>= 1;
        }

        return normalizedCapacity;
    }
    //如果是 16 的倍数
    if ((reqCapacity & 15) == 0) {
        return reqCapacity;
    }
    //不是 16 的倍数，变成最大小于当前值的值+16
    return (reqCapacity & ~15) + 16;
}
```

上面代码中 if (!isTiny(reqCapacity)) 代表如果大于 tiny 类型的大小，也就是 512，则会找一个 2 的幂次方的数值，确保这个数值大于等于 reqCapacity。如果是 tiny，则继续往下 if ((reqCapacity & 15) == 0) 这里判断如果是 16 的倍数，则直接返回。如果不是 16 的倍数，则返回 (reqCapacity & ~15) + 16，也就是变成最小大于当前值的 16 的倍数值。从上面规格化逻辑看出，这里将缓存大小规格化成固定大小，确保每个缓存对象缓存的 ByteBuf 容量统一。回到 allocate() 方法： if(isTinyOrSmall(normCapacity)) 这里是根据规格化后的大小判断是否 tiny 或者 small 类型，我们跟进去：

```
boolean isTinyOrSmall(int normCapacity) {
    return (normCapacity & subpageOverflowMask) == 0;
}
```

这个方法是判断如果 normCapacity 小于一个 page 的大小，也就是 8k 代表其实 tiny 或者 small。

继续看 allocate()方法，如果当前大小是 tiny 或者 small，则 isTiny(normCapacity)判断是否是 tiny 类型，跟进去：

```
static boolean isTiny(int normCapacity) {
    return (normCapacity & 0xFFFFFE00) == 0;
}
```

这个方法是判断如果小于 512，则认为是 tiny。

再继续看 allocate()方法：如果是 tiny，则通过 cache.allocateTiny(this, buf, reqCapacity, normCapacity)在缓存上进行分配。我们就以 tiny 类型为例，分析在缓存上分配 ByteBuf 的流：allocateTiny 是缓存分配的入口。我们跟进去，进入到了 PoolThreadCache 的 allocateTiny()方法中：

```
boolean allocateTiny(PoolArena<?> area, PooledByteBuf<?> buf, int reqCapacity, int normCapacity) {
    return allocate(cacheForTiny(area, normCapacity), buf, reqCapacity);
```

```
}
```

这里有个方法 cacheForTiny(area, normCapacity)，这个方法的作用是根据 normCapacity 找到 tiny 类型缓存数组中的一个缓存对象。我们跟进到 cacheForTiny()方法：

```
private MemoryRegionCache<?> cacheForTiny(PoolArena<?> area, int normCapacity) {
    int idx = PoolArena.tinyIdx(normCapacity);
    if (area.isDirect()) {
        return cache(tinySubPageDirectCaches, idx);
    }
    return cache(tinySubPageHeapCaches, idx);
}
```

PoolArena.tinyIdx(normCapacity)是找到 tiny 类型缓存数组的下标。继续跟 tinyIdx()方法：

```
static int tinyIdx(int normCapacity) {
    return normCapacity >>> 4;
}
```

这里相当于直接将 normCapacity 除以 16，通过前面的内容我们知道, tiny 类型缓存数组中每个元素规格化的数据都是 16 的倍数，所以通过这种方式可以找到其下标，参考图 5-2，如果是 16Byte 会拿到下标为 1 的元素，如果是 32Byte 则会拿到下标为 2 的元素。

回到 acheForTiny()方法中： if (area.isDirect()) 这里判断是否是分配堆外内存，因为我们是按照堆外内存进行举例，所以这里为 true。再继续跟到 cache(tinySubPageDirectCaches, idx)方法：

```
private static <T> MemoryRegionCache<T> cache(MemoryRegionCache<T>[] cache, int idx) {
    if (cache == null || idx > cache.length - 1) {
        return null;
    }
    return cache[idx];
}
```

这里我们看到直接通过下标的方式拿到了缓存数组中的对象，回到 PoolThreadCache 的 allocateTiny()方法中：

```
boolean allocateTiny(PoolArena<?> area, PooledByteBuf<?> buf, int reqCapacity, int normCapacity) {
    return allocate(cacheForTiny(area, normCapacity), buf, reqCapacity);
}
```

拿到了缓存对象之后，我们跟到 allocate(cacheForTiny(area, normCapacity), buf, reqCapacity)方法中：

```
private boolean allocate(MemoryRegionCache<?> cache, PooledByteBuf buf, int reqCapacity) {
    if (cache == null) {
        return false;
    }
    boolean allocated = cache.allocate(buf, reqCapacity);
    if (++ allocations >= freeSweepAllocationThreshold) {
        allocations = 0;
        trim();
    }
    return allocated;
}
```

分析上面代码，看到 cache.allocate(buf, reqCapacity)进行继续进行分配。再继续往里跟，来到内部类

MemoryRegionCache 的 allocate(PooledByteBuf<T> buf, int reqCapacity)方法：

```
public final boolean allocate(PooledByteBuf<T> buf, int reqCapacity) {
    Entry<T> entry = queue.poll();
    if (entry == null) {
        return false;
    }
    initBuf(entry.chunk, entry.handle, buf, reqCapacity);
    entry.recycle();
    ++ allocations;
    return true;
}
```

在这个方法中，首先通过 queue.poll()这种方式弹出一个 entry，我们之前的小节分析过, MemoryRegionCache 维护着一个队列，而队列中的每一个值是一个 entry。我们简单看下 Entry 这个类:

```
static final class Entry<T> {
    final Handle<Entry<?>> recyclerHandle;
    PoolChunk<T> chunk;
    long handle = -1;

    //代码省略
}
```

我们重点关注 chunk 和 handle 的这两个属性, chunk 代表一块连续的内存，我们之前简单介绍过, netty 是通过 chunk 为单位进行内存分配的，我们后面会对 chunk 进行详细剖析。handle 相当于一个指针，可以唯一定位到 chunk 里面的一块连续的内存，之后也会详细分析。这样，通过 chunk 和 handle 就可以定位 ByteBuf 中指定一块连续内存，有关 ByteBuf 相关的读写，都会在这块内存中进行。现在再回到 MemoryRegionCache 的 allocate(PooledByteBuf<T> buf, int reqCapacity)方法：

```
public final boolean allocate(PooledByteBuf<T> buf, int reqCapacity) {
    Entry<T> entry = queue.poll();
    if (entry == null) {
        return false;
    }
    initBuf(entry.chunk, entry.handle, buf, reqCapacity);
    entry.recycle();
    ++ allocations;
    return true;
}
```

弹出 entry 之后，通过 initBuf(entry.chunk, entry.handle, buf, reqCapacity)这种方式给 ByteBuf 初始化，这里参数传入我们刚才分析过的当前 Entry 的 chunk 和 hanle。因为我们分析的 tiny 类型的缓存对象是 SubPageMemoryRegionCache 类型,所以我们继续跟到 SubPageMemoryRegionCache 类的 initBuf(entry.chunk, entry.handle, buf, reqCapacity)方法中：

```
protected void initBuf(
        PoolChunk<T> chunk, long handle, PooledByteBuf<T> buf, int reqCapacity) {
```

```
        chunk.initBufWithSubpage(buf, handle, reqCapacity);
    }
```

这里的 chunk 调用了 initBufWithSubpage(buf, handle, reqCapacity)方法，其实就是 PoolChunk 类中的方法。我们继续跟 initBufWithSubpage()方法：

```
    void initBufWithSubpage(PooledByteBuf<T> buf, long handle, int reqCapacity) {
        initBufWithSubpage(buf, handle, bitmapIdx(handle), reqCapacity);
    }
```

上面代码中，调用了 bitmapIdx()方法，有关 bitmapIdx(handle)相关的逻辑，会在后续的章节进行剖析，这里继续往里跟，看 initBufWithSubpage()的逻辑：

```
    private void initBufWithSubpage(PooledByteBuf<T> buf, long handle, int bitmapIdx, int reqCapacity) {
        assert bitmapIdx != 0;
        int memoryMapIdx = memoryMapIdx(handle);
        PoolSubpage<T> subpage = subpages[subpageIdx(memoryMapIdx)];
        assert subpage.doNotDestroy;
        assert reqCapacity <= subpage.elemSize;
        buf.init(
            this, handle,
            runOffset(memoryMapIdx) + (bitmapIdx & 0x3FFFFFFF) * subpage.elemSize, reqCapacity, subpage.elemSize,
            arena.parent.threadCache());
    }
```

我们先关注 init 方法，因为我们是以 PooledUnsafeDirectByteBuf 为例，所以这里走的是 PooledUnsafeDirectByteBuf 的 init()方法。进入 init()方法：

```
    void init(PoolChunk<ByteBuffer> chunk, long handle, int offset, int length, int maxLength,
              PoolThreadCache cache) {
        super.init(chunk, handle, offset, length, maxLength, cache);
        initMemoryAddress();
    }
```

首先调用了父类的 init 方法，继续跟进去：

```
    void init(PoolChunk<T> chunk, long handle, int offset, int length, int maxLength, PoolThreadCache cache) {
        //初始化
        assert handle >= 0;
        assert chunk != null;
        //在哪一块内存上进行分配的
        this.chunk = chunk;
        //这一块内存上的哪一块连续内存
        this.handle = handle;
        memory = chunk.memory;
        this.offset = offset;
        this.length = length;
        this.maxLength = maxLength;
        tmpNioBuf = null;
        this.cache = cache;
    }
```

上面的代码就是将 PooledUnsafeDirectByteBuf 的各个属性进行了初始化。

this.chunk = chunk 这里初始化了 chunk，代表当前的 ByteBuf 是在哪一块内存中分配的。

this.handle = handle 这里初始化了 handle，代表当前的 ByteBuf 是这块内存的哪个连续内存。

有关 offset 和 length，我们会在之后再分析，在这里我们只需要知道，通过缓存分配 ByteBuf，我们只需要通过一个 chunk 和 handle，就可以确定一块内存，以上就是通过缓存分配 ByteBuf 对象的全过程。

现在，我们回到 MemoryRegionCache 的 allocate(PooledByteBuf<T> buf, int reqCapacity)方法：

```
public final boolean allocate(PooledByteBuf<T> buf, int reqCapacity) {
    Entry<T> entry = queue.poll();
    if (entry == null) {
        return false;
    }
    initBuf(entry.chunk, entry.handle, buf, reqCapacity);
    entry.recycle();
    ++ allocations;
    return true;
}
```

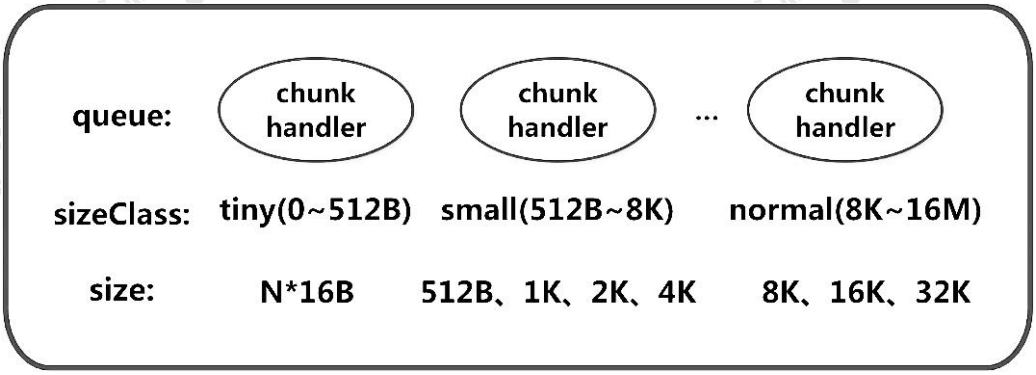
分析完了 initBuf()方法，再继续往下看：entry.recycle()这步是将 entry 对象进行回收，因为 entry 对象弹出之后没有再被引用，可能 gc 会将 entry 对象回收，netty 为了将对象进行循环利用，就将其放在对象回收站进行回收。我们跟进 recycle()方法：

```
void recycle() {
    chunk = null;
    handle = -1;
    recyclerHandle.recycle(this);
}
```

chunk = null 和 handle = -1 表示当前 Entry 不指向任何一块内存。recyclerHandle.recycle(this) 将当前 entry 回收。

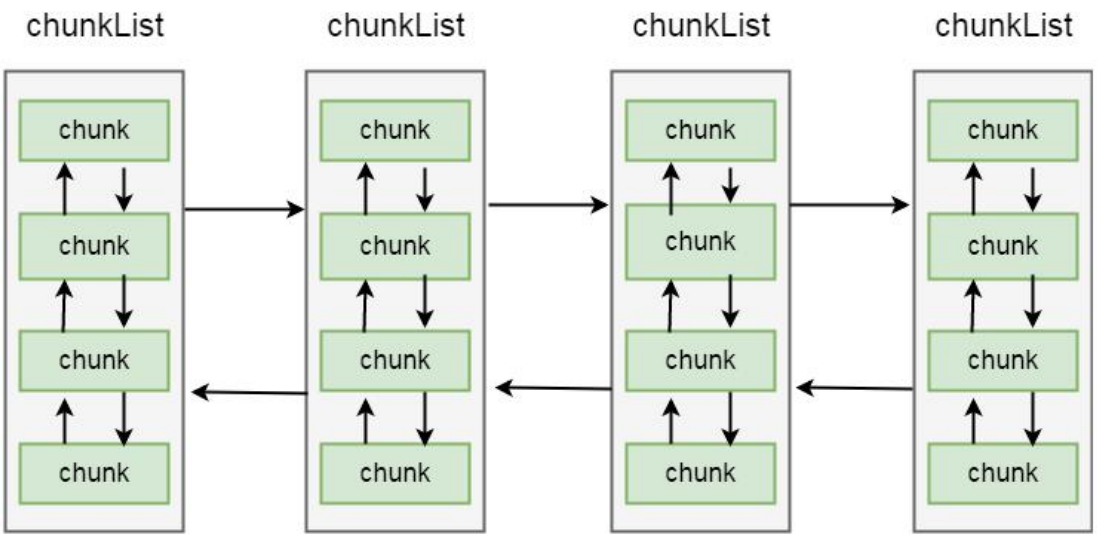
以上就是命中缓存的流程，因为这里我们是假设缓中有值的情况下进行分配的，如果第一次分配，缓存中是没有值的，那么在缓存中没有值的情况下，netty 是如何进行分配的呢?我们在后面再详细分析。

最后，我们简单总结一下 MemoryRegionCache 对象的基本结构，如下图所示：



10.4.5 Page 级别的内存分配

之前我们介绍过，netty 内存分配的单位是 chunk，一个 chunk 的大小是 16MB，实际上每个 chunk，都以双向链表的形式保存在一个 chunkList 中，而多个 chunkList，同样也是双向链表进行关联的，大概结构如下所示：



在 chunkList 中，是根据 chunk 的内存使用率归到一个 chunkList 中，这样，在内存分配时，会根据百分比找到相应的 chunkList，在 chunkList 中选择一个 chunk 进行内存分配。我们来看 PoolArena 中有关 chunkList 的成员变量：

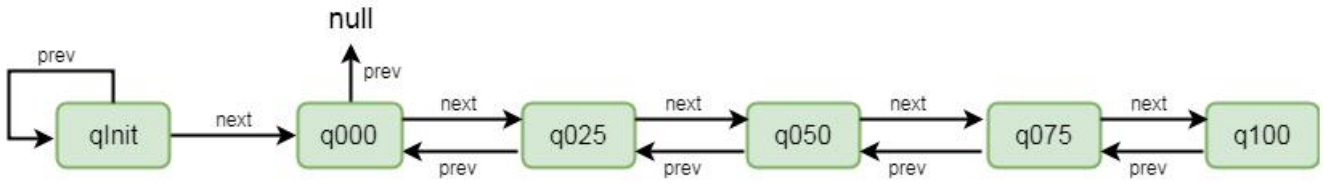
```
private final PoolChunkList<T> q050;
private final PoolChunkList<T> q025;
private final PoolChunkList<T> q000;
private final PoolChunkList<T> qInit;
private final PoolChunkList<T> q075;
private final PoolChunkList<T> q100;
```

这里总共定义了 6 个 chunkList，并在构造方法将其进行初始化，我们跟到其构造方法中：

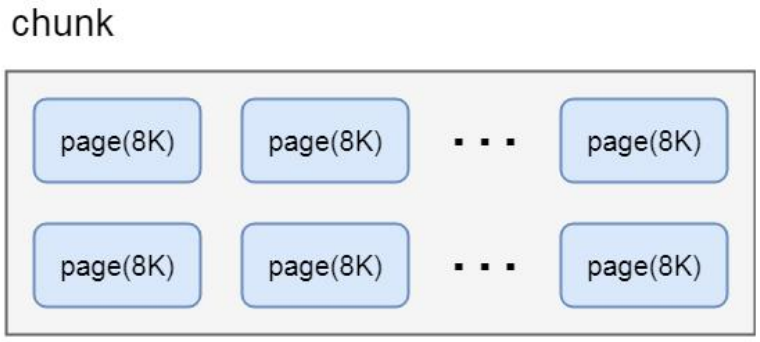
```
protected PoolArena(PooledByteBufAllocator parent, int pageSize, int maxOrder, int pageShifts, int chunkSize) {
    //代码省略
    q100 = new PoolChunkList<T>(null, 100, Integer.MAX_VALUE, chunkSize);
    q075 = new PoolChunkList<T>(q100, 75, 100, chunkSize);
    q050 = new PoolChunkList<T>(q075, 50, 100, chunkSize);
    q025 = new PoolChunkList<T>(q050, 25, 75, chunkSize);
    q000 = new PoolChunkList<T>(q025, 1, 50, chunkSize);
    qInit = new PoolChunkList<T>(q000, Integer.MIN_VALUE, 25, chunkSize);

    //用双向链表的方式进行连接
    q100.prevList(q075);
    q075.prevList(q050);
    q050.prevList(q025);
    q025.prevList(q000);
    q000.prevList(null);
```

```
        qInit.prevList(qInit);
        //代码省略
    }
```

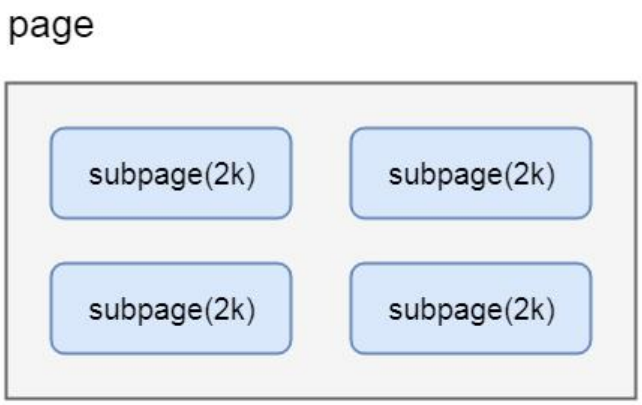
首先通过 new PoolChunkList()这种方式将每个 chunkList 进行创建，我们以 q050 = new PoolChunkList<T>(q075, 50, 100, chunkSize) 为例进行简单的介绍。q075 表示当前 q50 的下一个节点是 q075，刚才我们讲过 ChunkList 是通过双向链表进行关联的，所以这里不难理解。参数 50 和 100 表示当前 chunkList 中存储的 chunk 的内存使用率都在 50%到 100%之间，最后 chunkSize 为其设置大小。创建完 ChunkList 之后，再设置其上一个节点，q050.prevList(q025)为例，这里代表当前 chunkList 的上一个节点是 q025。以这种方式创建完成之后，chunkList 的节点关系变成了如下图所示：



Netty 中，chunk 又包含了多个 page，每个 page 的大小为 8KB，如果要分配 16KB 的内存，则在在 chunk 中找到连续的两个 page 就可以分配，对应关系如下：



很多场景下，为缓冲区分配 8KB 的内存也是一种浪费，比如只需要分配 2KB 的缓冲区，如果使用 8KB 会造成 6KB 的浪费，这种情况，netty 又会将 page 切分成多个 subpage，每个 subpage 大小要根据分配的缓冲区大小而指定，比如要分配 2KB 的内存，就会将一个 page 切分成 4 个 subpage，每个 subpage 的大小为 2KB，如下图：



来看看 PoolSubpage 的基本结构：

```
final PoolChunk<T> chunk;
private final int memoryMapIdx;
private final int runOffset;
private final int pageSize;
private final long[] bitmap;
PoolSubpage<T> prev;
PoolSubpage<T> next;
boolean doNotDestroy;
int elemSize;
```

chunk 代表其子页属于哪个 chunk；bitmap 用于记录子页的内存分配情况；prev 和 next，代表子页是按照双向链表进行关联的，这里分别指向上一个和下一个节点；elemSize 属性，代表的就是这个子页是按照多大内存进行划分的，如果按照 1KB 划分，则可以划分出 8 个子页；简单介绍了内存分配的数据结构，

我们开始剖析 Netty 在 page 级别上分配内存的流程，还是回到 PoolArena 的 allocate 方法：

```
private void allocate(PoolThreadCache cache, PooledByteBuf<T> buf, final int reqCapacity) {
    //规格化
    final int normCapacity = normalizeCapacity(reqCapacity);
    if (isTinyOrSmall(normCapacity)) {
        int tableIdx;
        PoolSubpage<T>[] table;
        //判断是不是 tinty
        boolean tiny = isTiny(normCapacity);
        if (tiny) { // < 512
            //缓存分配
            if (cache.allocateTiny(this, buf, reqCapacity, normCapacity)) {
                return;
            }
            //通过 tinyIdx 拿到 tableIdx
            tableIdx = tinyIdx(normCapacity);
            //subpage 的数组
            table = tinySubpagePools;
        } else {
            if (cache.allocateSmall(this, buf, reqCapacity, normCapacity)) {
                return;
```

```
            }
            tableIdx = smallIdx(normCapacity);
            table = smallSubpagePools;
        }

        //拿到对应的节点
        final PoolSubpage<T> head = table[tableIdx];

        synchronized (head) {
            final PoolSubpage<T> s = head.next;
            //默认情况下，head 的 next 也是自身
            if (s != head) {
                assert s.doNotDestroy && s.elemSize == normCapacity;
                long handle = s.allocate();
                assert handle >= 0;
                s.chunk.initBufWithSubpage(buf, handle, reqCapacity);

                if (tiny) {
                    allocationsTiny.increment();
                } else {
                    allocationsSmall.increment();
                }
                return;
            }
        }
        allocateNormal(buf, reqCapacity, normCapacity);
        return;
    }
    if (normCapacity <= chunkSize) {
        //首先在缓存上进行内存分配
        if (cache.allocateNormal(this, buf, reqCapacity, normCapacity)) {
            //分配成功，返回
            return;
        }
        //分配不成功，做实际的内存分配
        allocateNormal(buf, reqCapacity, normCapacity);
    } else {
        //大于这个值，就不在缓存上分配
        allocateHuge(buf, reqCapacity);
    }
}
```

我们之前讲过，如果在缓存中分配不成功，则会开辟一块连续的内存进行缓冲区分配，这里我们先跳过 isTinyOrSmall(normCapacity)往后的代码，之后再来分析。

首先 if (normCapacity <= chunkSize) 说明其小于 16MB，然后首先在缓存中分配，因为最初缓存中没有值，所以会走到 allocateNormal(buf, reqCapacity, normCapacity)，这里实际上就是在 page 级别上进行分配，分配一个或者多个 page 的空间。我们跟进到 allocateNormal()方法：

```
private synchronized void allocateNormal(PooledByteBuf<T> buf, int reqCapacity, int normCapacity) {
    //首先在原来的 chunk 上进行内存分配(1)
    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)) {
        ++allocationsNormal;
        return;
    }
```

```
        //创建 chunk 进行内存分配(2)
        PoolChunk<T> c = newChunk(pageSize, maxOrder, pageShifts, chunkSize);
        long handle = c.allocate(normCapacity);
        ++allocationsNormal;
        assert handle > 0;
        //初始化 byteBuf(3)
        c.initBuf(buf, handle, reqCapacity);
        qInit.add(c);
    }
```

这里主要拆解了如下步骤：

1. 在原有的 chunk 中进行分配；

2. 创建 chunk 进行分配；

3. 初始化 ByteBuf。

首先我们看第一步，在原有的 chunk 中进行分配：

```
    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)) {
        ++allocationsNormal;
        return;
    }
```

我们之前讲过，chunkList 是存储不同内存使用量的 chunk 集合，每个 chunkList 通过双向链表的形式进行关联，这里的 q050.allocate(buf, reqCapacity, normCapacity)就代表首先在 q050 这个 chunkList 上进行内存分配。我们以 q050 为例进行分析，跟到 q050.allocate(buf, reqCapacity, normCapacity)方法中：

```
    boolean allocate(PooledByteBuf<T> buf, int reqCapacity, int normCapacity) {
        if (head == null || normCapacity > maxCapacity) {
            return false;
        }
        //从 head 节点往下遍历
        for (PoolChunk<T> cur = head;;) {
            long handle = cur.allocate(normCapacity);
            if (handle < 0) {
                cur = cur.next;
                if (cur == null) {
                    return false;
                }
            } else {
                cur.initBuf(buf, handle, reqCapacity);
                if (cur.usage() >= maxUsage) {
                    remove(cur);
                    nextList.add(cur);
                }
                return true;
            }
        }
    }
```

首先会从 head 节点往下遍历：long handle = cur.allocate(normCapacity) 表示对于每个 chunk，都尝试去分配；

if (handle < 0) 说明没有分配到，则通过 cur = cur.next 找到下一个节点继续进行分配，我们讲过 chunk 也是通过双向链表进行关联的，所以对这块逻辑应该不会陌生。如果 handle 大于 0 说明已经分配到了内存，则通过 cur.initBuf(buf, handle, reqCapacity)对 byteBuf 进行初始化；if (cur.usage() >= maxUsage) 代表当前 chunk 的内存使用率大于其最大使用率，则通过 remove(cur)从当前的 chunkList 中移除，再通过 nextList.add(cur)添加到下一个 chunkList 中。

我们再回到 PoolArena 的 allocateNormal()方法中：

看第二步 PoolChunk<T> c = newChunk(pageSize, maxOrder, pageShifts, chunkSize) ,这里的参数 pageSize 是 8192，也就是 8KB。

maxOrder 为 11；

pageShifts 为 13，2 的 13 次方正好是 8192，也就是 8KB；

chunkSize 为 16777216，也就是 16MB。

因为我们分析的是堆外内存，newChunk(pageSize, maxOrder, pageShifts, chunkSize)所以会走到 DirectArena 的 newChunk()方法：

```
protected PoolChunk<ByteBuffer> newChunk(int pageSize, int maxOrder, int pageShifts, int chunkSize) {
    return new PoolChunk<ByteBuffer>(
            this, allocateDirect(chunkSize),
            pageSize, maxOrder, pageShifts, chunkSize);
}
```

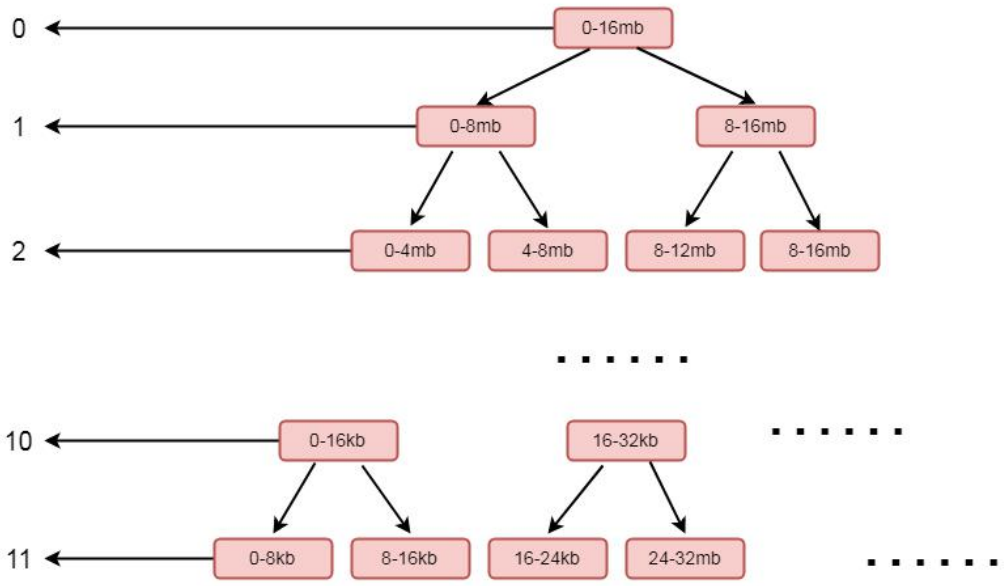
这里直接通过构造函数创建了一个 chunk。allocateDirect(chunkSize)这里是通过 jdk 的 api 的申请了一块直接内存，我们跟到 PoolChunk 的构造函数中：

```
PoolChunk(PoolArena<T> arena, T memory, int pageSize, int maxOrder, int pageShifts, int chunkSize) {
    unpooled = false;
    this.arena = arena;
    //memeory 为一个 ByteBuf
    this.memory = memory;
    //8k
    this.pageSize = pageSize;
    //13
    this.pageShifts = pageShifts;
    //11
    this.maxOrder = maxOrder;
    this.chunkSize = chunkSize;
    unusable = (byte) (maxOrder + 1);
    log2ChunkSize = log2(chunkSize);
    subpageOverflowMask = ~(pageSize - 1);
```

```
        freeBytes = chunkSize;

        assert maxOrder < 30 : "maxOrder should be < 30, but is: " + maxOrder;
        maxSubpageAllocs = 1 << maxOrder;

        //节点数量为 4096
        memoryMap = new byte[maxSubpageAllocs << 1];
        //也是 4096 个节点
        depthMap = new byte[memoryMap.length];
        int memoryMapIndex = 1;
        //d 相当于一个深度，赋值的内容代表当前节点的深度
        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 ++;
            }
        }

        subpages = newSubpageArray(maxSubpageAllocs);
    }
```

首先将参数传入的值进行赋值 this.memory = memory 就是将参数中创建的堆外内存进行保存，就是 chunk 所指向的那块连续的内存，在这个 chunk 中所分配的 ByteBuf，都会在这块内存中进行读写。

我们重点关注 memoryMap = new byte[maxSubpageAllocs << 1] 和 depthMap = new byte[memoryMap.length] 这两步：首先看 memoryMap = new byte[maxSubpageAllocs << 1]；这里初始化了一个字节数组 memoryMap，大小为 maxSubpageAllocs << 1，也就是 4096；depthMap = new byte[memoryMap.length] 同样也是初始化了一个字节数组，大小为 memoryMap 的大小，也就是 4096。继续往下分析之前，我们看 chunk 的一个层级关系。



这是一个二叉树的结构，左侧的数字代表层级，右侧代表一块连续的内存，每个父节点下又拆分成多个子节点，最顶层表示的内存范围为 0-16MB，其又下分为两层，范围为 0-8MB, 8-16MB，以此类推，最后到 11 层，以 8k 的大小划分，也就是一个 page 的大小。

如果我们分配一个 8mb 的缓冲区，则会将第二层的第一个节点，也就是 0-8 这个连续的内存进行分配，分配完成之后，会将这个节点设置为不可用。结合上面的图，我们再看构造方法中的 for 循环：

```
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 ++;
    }
}
```

实际上这个 for 循环就是将上面的结构包装成一个字节数组 memoryMap，外层循环用于控制层数，内层循环用于控制里面每层的节点，这里经过循环之后，memoryMap 和 depthMap 内容为以下表现形式：

[0, 0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4.....]

这里注意一下，因为程序中数组的下标是从 1 开始设置的，所以第零个节点元素为默认值 0。

这里数字代表层级，同时也代表了当前层级的节点，相同的数字个数就是这一层级的节点数。

其中 0 为 2 个(因为这里分配时下标是从 1 开始的，所以第 0 个位置是默认值 0，实际上第零层元素只有一个，就是头结点)，1 为 2 个，2 为 4 个，3 为 8 个，4 为 16 个，n 为 2 的 n 次方个，直到 11，也就是 11 有 2 的 11 次方个。

我们再回到 PoolArena 的 allocateNormal()方法：

```
private synchronized void allocateNormal(PooledByteBuf<T> buf, int reqCapacity, int normCapacity) {
    //首先在原来的 chunk 上进行内存分配(1)
    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)) {
        ++allocationsNormal;
        return;
    }

    //创建 chunk 进行内存分配(2)
    PoolChunk<T> c = newChunk(pageSize, maxOrder, pageShifts, chunkSize);
    long handle = c.allocate(normCapacity);
    ++allocationsNormal;
    assert handle > 0;
    //初始化 byteBuf(3)
    c.initBuf(buf, handle, reqCapacity);
    qInit.add(c);
}
```

我们继续剖析 long handle = c.allocate(normCapacity) 这步，跟到 allocate(normCapacity)中：

```
long allocate(int normCapacity) {
    if ((normCapacity & subpageOverflowMask) != 0) {
        return allocateRun(normCapacity);
    } else {
        return allocateSubpage(normCapacity);
    }
}
```

如果分配是以 page 为单位，则走到 allocateRun(normCapacity)方法中，跟进去：

```
private long allocateRun(int normCapacity) {
    int d = maxOrder - (log2(normCapacity) - pageShifts);
    int id = allocateNode(d);
    if (id < 0) {
        return id;
    }
    freeBytes -= runLength(id);
    return id;
}
```

int d = maxOrder - (log2(normCapacity) - pageShifts) 表示根据 normCapacity 计算出第几层；

int id = allocateNode(d) 表示根据层级关系，去分配一个节点，其中 id 代表 memoryMap 中的下标。

我们跟到 allocateNode()方法中：

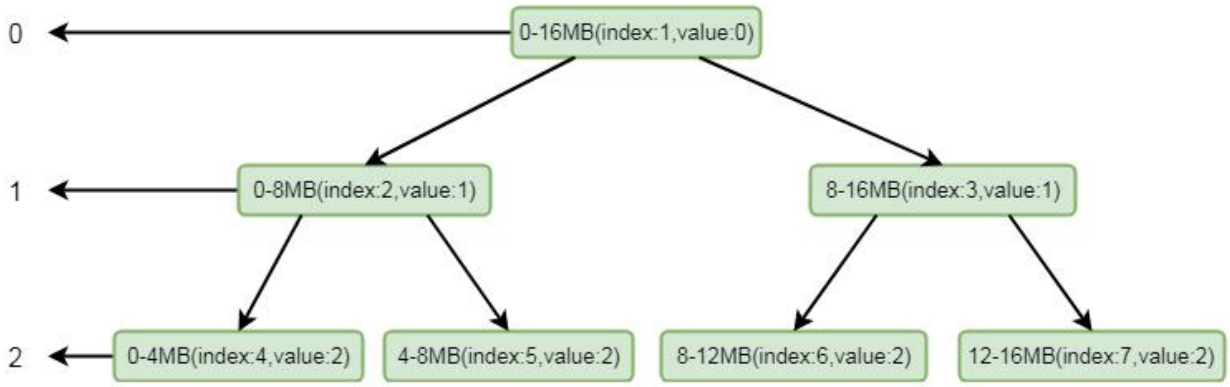
```
private int allocateNode(int d) {
    //下标初始值为 1
    int id = 1;
    //代表当前层级第一个节点的初始下标
    int initial = - (1 << d);
    //获取第一个节点的值
    byte val = value(id);
    //如果值大于层级，说明 chunk 不可用
    if (val > d) {
        return -1;
    }
    //当前下标对应的节点值如果小于层级，或者当前下标小于层级的初始下标
    while (val < d || (id & initial) == 0) {
        //当前下标乘以 2，代表下当前节点的子节点的起始位置
        id <<= 1;
        //获得 id 位置的值
        val = value(id);
        //如果当前节点值大于层数(节点不可用)
        if (val > d) {
            //id 为偶数则+1，id 为奇数则-1(拿的是其兄弟节点)
            id ^= 1;
            //获取 id 的值
            val = value(id);
        }
    }
    byte value = value(id);
    assert value == d && (id & initial) == 1 << d : String.format("val = %d, id & initial = %d, d = %d",
            value, id & initial, d);
    //将找到的节点设置为不可用
    setValue(id, unusable);
    //逐层往上标记被使用
    updateParentsAlloc(id);
    return id;
```

```
    }
```

这里是实际上是从第一个节点往下找，找到层级为 d 未被使用的节点，我们可以通过注释体会其逻辑。找到相关节点后通过 setValue 将当前节点设置为不可用，其中 id 是当前节点的下标，unusable 代表一个不可用的值，这里是 12，因为我们的层级只有 12 层，所以设置为 12 之后就相当于标记不可用。设置成不可用之后，通过 updateParentsAlloc(id)逐层设置为被使用。我们跟进 updateParentsAlloc()方法：

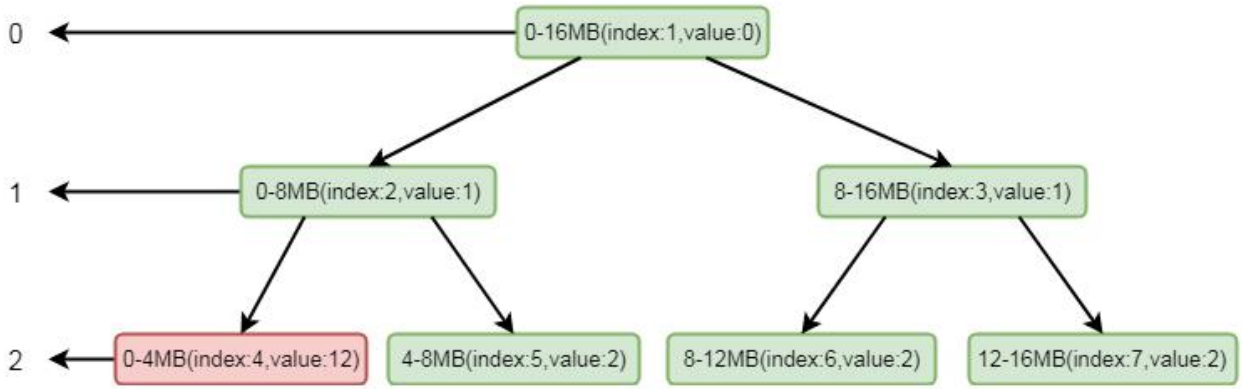
```
    private void updateParentsAlloc(int id) {
        while (id > 1) {
            //取到当前节点的父节点的 id
            int parentId = id >>> 1;
            //获取当前节点的值
            byte val1 = value(id);
            //找到当前节点的兄弟节点
            byte val2 = value(id ^ 1);
            //如果当前节点值小于兄弟节点，则保存当前节点值到 val，否则，保存兄弟节点值到 val
            //如果当前节点是不可用，则当前节点值是 12，大于兄弟节点的值，所以这里将兄弟节点的值进行保存
            byte val = val1 < val2 ? val1 : val2;
            //将 val 的值设置为父节点下标所对应的值
            setValue(parentId, val);
            //id 设置为父节点 id，继续循环
            id = parentId;
        }
    }
```

这里其实是将循环将兄弟节点的值替换成父节点的值，我们可以通过注释仔细的进行逻辑分析。如果实在理解有困难，我通过画图帮助大家理解，简单起见，我们这里只设置三层：

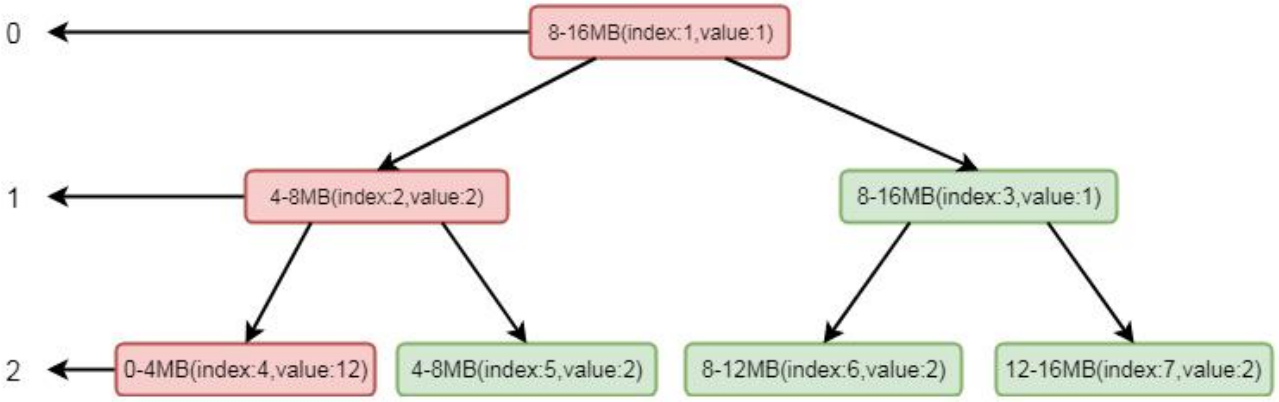


我们模拟其分配场景，假设只有三层，其中 index 代表数组 memoryMap 的下标，value 代表其值，memoryMap 中的值就为[0, 0, 1, 1, 2, 2, 2, 2]。我们要分配一个 4MB 的 byteBuf，在我们调用 allocateNode(int d)中传入的 d 是 2，也就是第二层。根据我们上面分分析的逻辑这里会找到第二层的第一个节点，也就是 0-4mb 这个节点，找到之后将其设置为

不可用，这样 memoryMap 中的值就为[0, 0, 1, 1, 12, 2, 2, 2]，二叉树的结构就会变为：

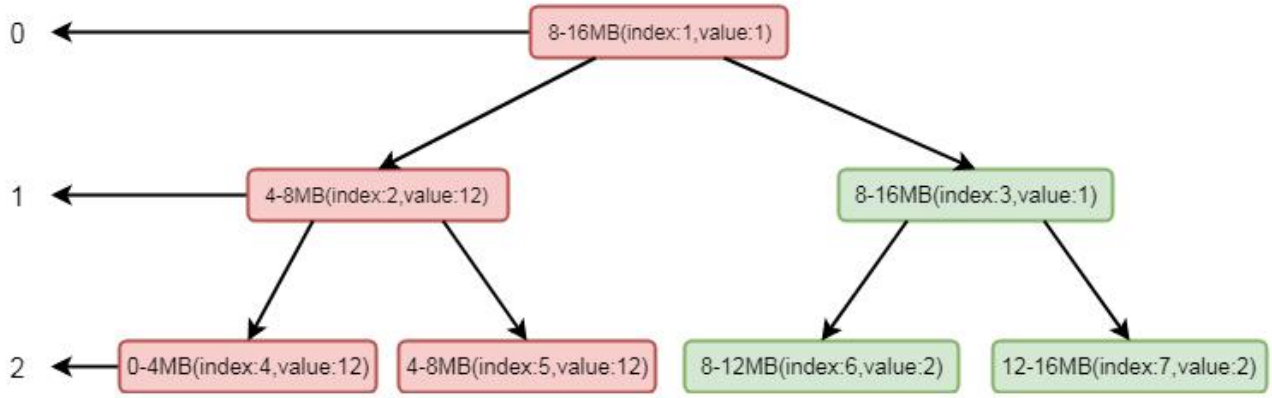


注意标红部分，将 index 为 4 的节点设置为了不可用。将这个节点设置为不可用之后，则会将进行向上设置不可用，循环将兄弟节点数值较小的节点替换到父节点，也就是将 index 为 2 的节点的值替换成了 index 的为 5 的节点的值，这样数组的值就会变为[0, 1, 2, 1, 12, 2, 2, 2]，二叉树的结构变为：



注意：这里节点标红仅仅代表节点变化，并不是当前节点为不可用状态，真正不可用状态的判断依据是 value 值为 12。

这样，如果再次分配一个 4MB 内存的 ByteBuf，根据其逻辑，则会找到第二层的第二个节点，也就是 4-8MB。再根据我们的逻辑，通过向上设置不可用，index 为 2 就会设置成不可用状态，将 value 的值设置为 12，数组数值变为[0, 1, 12, 1, 12, 12, 2, 2]二叉树如下图所示：



这样我们看到，通过分配两个 4mb 的 byteBuf 之后，当前节点和其父节点都会设置成不可用状态，当 index=2 的节点设置为不可用之后，将不会再找这个节点下的子节点。以此类推，直到所有的内存分配完毕的时候，index 为 1 的节点，也会变成不可用状态，这样所有的 page 就分配完毕，chunk 中再无可用节点。现在再回到 PoolArena 的 allocateNormal() 方法：

```
private synchronized void allocateNormal(PooledByteBuf<T> buf, int reqCapacity, int normCapacity) {
    //首先在原来的 chunk 上进行内存分配(1)
    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)) {
        ++allocationsNormal;
        return;
    }

    //创建 chunk 进行内存分配(2)
    PoolChunk<T> c = newChunk(pageSize, maxOrder, pageShifts, chunkSize);
    long handle = c.allocate(normCapacity);
    ++allocationsNormal;
    assert handle > 0;
    //初始化 byteBuf(3)
    c.initBuf(buf, handle, reqCapacity);
    qInit.add(c);
}
```

通过以上逻辑我们知道，long handle = c.allocate(normCapacity)这一步，其实返回的就是 memoryMap 的一个下标，通过这个下标，我们能唯一的定位一块内存。继续往下跟，通过 c.initBuf(buf, handle, reqCapacity)初始化 ByteBuf 之后，通过 qInit.add(c)将新创建的 chunk 添加到 chunkList 中，我们跟到 initBuf 方法中去：

```
void initBuf(PooledByteBuf<T> buf, long handle, int reqCapacity) {
    int memoryMapIdx = memoryMapIdx(handle);
    int bitmapIdx = bitmapIdx(handle);
    if (bitmapIdx == 0) {
        byte val = value(memoryMapIdx);
```

```
            assert val == unusable : String.valueOf(val);
            buf.init(this, handle, runOffset(memoryMapIdx), reqCapacity, runLength(memoryMapIdx),
                    arena.parent.threadCache());
        } else {
            initBufWithSubpage(buf, handle, bitmapIdx, reqCapacity);
        }
    }
```

从上面代码中，看出通过 memoryMapIdx(handle)找到 memoryMap 的下标，其实就是 handle 的值。bitmapIdx(handle)是有关 subPage 中使用到的逻辑，如果是 page 级别的分配，这里只返回 0，所以进入到 if 块中。if 中首先断言当前节点是不是不可用状态，然后通过 init 方法进行初始化。其中 runOffset(memoryMapIdx)表示偏移量，偏移量相当于分配给缓冲区的这块内存相对于 chunk 中申请的内存的首地址偏移了多少。参数 runLength(memoryMapIdx)，表示根据下标获取可分配的最大长度。我们跟到 init()方法中，这里会走到 PooledByteBuf 的 init()方法：

```
    void init(PoolChunk<T> chunk, long handle, int offset, int length, int maxLength, PoolThreadCache cache) {
        //初始化
        assert handle >= 0;
        assert chunk != null;

        //在哪一块内存上进行分配的
        this.chunk = chunk;
        //这一块内存上的哪一块连续内存
        this.handle = handle;
        memory = chunk.memory;
        this.offset = offset;
        this.length = length;
        this.maxLength = maxLength;
        tmpNioBuf = null;
        this.cache = cache;
    }
```

这段代码又是我们熟悉的部分，将属性进行了初始化。以上就是完整的 DirectUnsafePooledByteBuf 在 Page 级别的完整分配的流程，逻辑也是非常的复杂，想真正的掌握熟练，还需要小伙伴们多下功夫进行调试和剖析。

10.4.6 SubPage 级别的内存分配

通过之前的学习我们知道，如果我们分配一个缓冲区大小远小于 page，则直接在一个 page 上进行分配则会造成内存浪费，所以需要将 page 继续进行切分成多个子块进行分配，子块分配的个数根据你要分配的缓冲区大小而定，比如只需要分配 1KB 的内存，就会将一个 page 分成 8 等分。简单起见，我们这里仅仅以 16 字节为例，讲解其分配逻辑。在分析其逻辑前，首先看 PoolArean 的一个属性：

```
    private final PoolSubpage<T>[] tinySubpagePools;
```

这个属性是一个 PoolSubpage 的数组，有点类似于一个 subpage 的缓存，我们创建一个 subpage 之后，会将创建的

subpage 与该属性其中每个关联，下次在分配的时候可以直接通过该属性的元素去找关联的 subpage。我们其中是在构造方法中初始化的，看构造方法中其初始化代码：

```
tinySubpagePools = newSubpagePoolArray(numTinySubpagePools);
```

这里为 numTinySubpagePools 为 32，跟到 newSubpagePoolArray(numTinySubpagePools)方法中：

```
private PoolSubpage<T>[] newSubpagePoolArray(int size) {
    return new PoolSubpage[size];
}
```

这里直接创建了一个 PoolSubpage 数组，长度为 32，在构造方法中创建完毕之后，会通过循环为其赋值：

```
for (int i = 0; i < tinySubpagePools.length; i ++) {
    tinySubpagePools[i] = newSubpagePoolHead(pageSize);
}
```

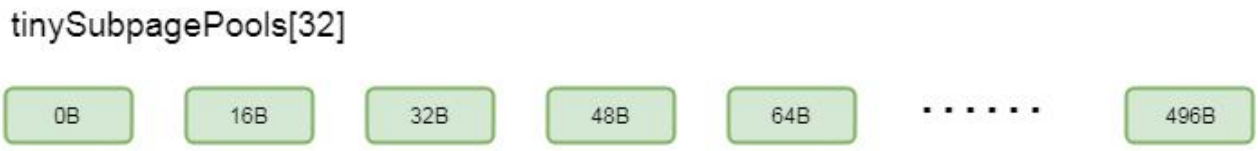
继续跟到 newSubpagePoolHead()方法中：

```
private PoolSubpage<T> newSubpagePoolHead(int pageSize) {
    PoolSubpage<T> head = new PoolSubpage<T>(pageSize);
    head.prev = head;
    head.next = head;
    return head;
}
```

在 newSubpagePoolHead()方法中创建了一个 PoolSubpage 对象 head。

```
head.prev = head;
head.next = head;
```

这种写法我们知道 Subpage 其实也是个双向链表，这里的将 head 的上一个节点和下一个节点都设置为自身，有关 PoolSubpage 的关联关系，我们稍后分析。这样通过循环创建 PoolSubpage，总共会创建出 32 个 subpage，其中每个 subpage 实际代表一块内存大小：



tinySubPagePools 的结构就有点类之前小节的缓存数组 tinySubPageDirectCaches 的结构。了解了 tinySubpagePools 属性，我们看 PoolArean 的 allocate 方法，也就是缓冲区的入口方法：

```
private void allocate(PoolThreadCache cache, PooledByteBuf<T> buf, final int reqCapacity) {
    //规格化
    final int normCapacity = normalizeCapacity(reqCapacity);
    if (isTinyOrSmall(normCapacity)) {
        int tableIdx;
```

```
        PoolSubpage<T>[] table;
        //判断是不是 tinty
        boolean tiny = isTiny(normCapacity);
        if (tiny) { // < 512
            //缓存分配
            if (cache.allocateTiny(this, buf, reqCapacity, normCapacity)) {
                return;
            }
            //通过 tinyIdx 拿到 tableIdx
            tableIdx = tinyIdx(normCapacity);
            //subpage 的数组
            table = tinySubpagePools;
        } else {
            if (cache.allocateSmall(this, buf, reqCapacity, normCapacity)) {
                return;
            }
            tableIdx = smallIdx(normCapacity);
            table = smallSubpagePools;
        }

        //拿到对应的节点
        final PoolSubpage<T> head = table[tableIdx];

        synchronized (head) {
            final PoolSubpage<T> s = head.next;
            //默认情况下，head 的 next 也是自身
            if (s != head) {
                assert s.doNotDestroy && s.elemSize == normCapacity;
                long handle = s.allocate();
                assert handle >= 0;
                s.chunk.initBufWithSubpage(buf, handle, reqCapacity);

                if (tiny) {
                    allocationsTiny.increment();
                } else {
                    allocationsSmall.increment();
                }
                return;
            }
        }
        allocateNormal(buf, reqCapacity, normCapacity);
        return;
    }
    if (normCapacity <= chunkSize) {
        //首先在缓存上进行内存分配
        if (cache.allocateNormal(this, buf, reqCapacity, normCapacity)) {
            //分配成功，返回
            return;
        }
        //分配不成功，做实际的内存分配
        allocateNormal(buf, reqCapacity, normCapacity);
    } else {
        //大于这个值，就不在缓存上分配
        allocateHuge(buf, reqCapacity);
    }
}
```

之前我们最这个方法剖析过在 page 级别相关内存分配逻辑，先在我们来看 subpage 级别分配的相关逻辑。假设我们分配 16 字节的缓冲区，isTinyOrSmall(normCapacity)就会返回 true，进入 if 块，同样 if (tiny)这里会返回 true，继续跟

到 if (tiny)中的逻辑。首先会在缓存中分配缓冲区，如果分配不到，就开辟一块内存进行内存分配，先看这一步:

```
tableIdx = tinyIdx(normCapacity);
```

这里通过 normCapacity 拿到 tableIdx，我们跟进去：

```
static int tinyIdx(int normCapacity) {
    return normCapacity >>> 4;
}
```

这里将 normCapacity 除以 16，其实也就是 1。我们回到 PoolArena 的 allocate()方法继续看：

```
table = tinySubpagePools
```

这里将 tinySubpagePools 赋值到局部变量 table 中，继续往下看：

final PoolSubpage<T> head = table[tableIdx] 这步时通过下标拿到一个 PoolSubpage，因为我们以 16 字节为例，所以我们拿到下标为 1 的 PoolSubpage，对应的内存大小也就是 16Byte。再看 final PoolSubpage<T> s = head.next 这一步，跟我们刚才了解的的 tinySubpagePools 属性，默认情况下 head.next 也是自身，所以 if (s != head)会返回 false，

我们继续往下看，会走到 allocateNormal(buf, reqCapacity, normCapacity)这个方法：

```
private synchronized void allocateNormal(PooledByteBuf<T> buf, int reqCapacity, int normCapacity) {
    //首先在原来的 chunk 上进行内存分配(1)
    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)) {
        ++allocationsNormal;
        return;
    }

    //创建 chunk 进行内存分配(2)
    PoolChunk<T> c = newChunk(pageSize, maxOrder, pageShifts, chunkSize);
    long handle = c.allocate(normCapacity);
    ++allocationsNormal;
    assert handle > 0;
    //初始化 byteBuf(3)
    c.initBuf(buf, handle, reqCapacity);
    qInit.add(c);
}
```

这里的逻辑我们之前的小节已经剖析过，首先在原来的 chunk 中分配，如果分配不成功，则会创建 chunk 进行分配。

我们看这一步 long handle = c.allocate(normCapacity) ，跟到 allocate(normCapacity)方法中：

```
long allocate(int normCapacity) {
    if ((normCapacity & subpageOverflowMask) != 0) {
        return allocateRun(normCapacity);
    } else {
        return allocateSubpage(normCapacity);
    }
}
```

上一小节我们分析 page 级别分配的时候，剖析的是 allocateRun(normCapacity)方法。因为这里我们是以 16 字节举例，所以这次我们剖析 allocateSubpage(normCapacity)方法，也就是在 subpage 级别进行内存分配。

```
private long allocateSubpage(int normCapacity) {
    PoolSubpage<T> head = arena.findSubpagePoolHead(normCapacity);
    synchronized (head) {
        int d = maxOrder;
        //表示在第 11 层分配节点
        int id = allocateNode(d);
        if (id < 0) {
            return id;
        }

        //获取初始化的 subpage
        final PoolSubpage<T>[] subpages = this.subpages;
        final int pageSize = this.pageSize;

        freeBytes -= pageSize;
        //表示第几个 subpageIdx
        int subpageIdx = subpageIdx(id);
        PoolSubpage<T> subpage = subpages[subpageIdx];
        if (subpage == null) {
            //如果 subpage 为空
            subpage = new PoolSubpage<T>(head, this, id, runOffset(id), pageSize, normCapacity);
            //则将当前的下标赋值为 subpage
            subpages[subpageIdx] = subpage;
        } else {
            subpage.init(head, normCapacity);
        }
        //取出一个子 page
        return subpage.allocate();
    }
}
```

首先，通过 PoolSubpage<T> head = arena.findSubpagePoolHead(normCapacity) 这种方式找到 head 节点，实际上这里 head，就是我们刚才分析的 tinySubpagePools 属性的第一个节点，也就是对应 16B 的那个节点。int d = maxOrder 是将 11 赋值给 d，也就是在内存树的第 11 层取节点，这部分上一小节剖析过了。int id = allocateNode(d) 这里获取的是上一小节我们分析过的，字节数组 memoryMap 的下标，这里指向一个 page，如果第一次分配，指向的是 0-8k 的那个 page，上一小节对此进行详细的剖析这里不再赘述。final PoolSubpage<T>[] subpages = this.subpages 这一步，是拿到 PoolChunk 中成员变量 subpages 的值，也是个 PoolSubpage 的数组，在 PoolChunk 进行初始化的时候，也会初始化该数组，长度为 2048。也就是说每个 chunk 都维护着一个 subpage 的列表，如果每一个 page 级别的内存都需要被切分成子 page，则会将这个这个 page 放入该列表中，专门用于分配子 page，所以这个列表中的 subpage，其实就是一个用于切分的 page。

int subpageIdx = subpageIdx(id) 这一步是通过 id 拿到这个 PoolSubpage 数组的下标，如果 id 对应的 page 是 0-8k 的节点，这里拿到的下标就是 0。在 if (subpage == null) 中，因为默认 subpages 只是创建一个数组，并没有往数组中赋值，所以第一次走到这里会返回 true，跟到 if 块中：

```
subpage = new PoolSubpage<T>(head, this, id, runOffset(id), pageSize, normCapacity);
```

这里通过 new PoolSubpage 创建一个新的 subpage 之后，通过 subpages[subpageIdx] = subpage 这种方式将新创建的 subpage 根据下标赋值到 subpages 中的元素中。在 new PoolSubpage 的构造方法中，传入 head，就是我们刚才提到过的 tinySubpagePools 属性中的节点，如果我们分配的 16 字节的缓冲区，则这里对应的就是第一个节点，我们跟到 PoolSubpage 的构造方法中：

```
PoolSubpage(PoolSubpage<T> head, PoolChunk<T> chunk, int memoryMapIdx, int runOffset, int pageSize, int elemSize) {
    this.chunk = chunk;
    this.memoryMapIdx = memoryMapIdx;
    this.runOffset = runOffset;
    this.pageSize = pageSize;
    bitmap = new long[pageSize >>> 10];
    init(head, elemSize);
}
```

这里重点关注属性 bitmap，这是一个 long 类型的数组，初始大小为 8，这里只是初始化的大小，真正的大小要根据将 page 切分多少块而确定，这里将属性进行了赋值，我们跟到 init()方法中：

```
void init(PoolSubpage<T> head, int elemSize) {
    doNotDestroy = true;
    this.elemSize = elemSize;
    if (elemSize != 0) {
        maxNumElems = numAvail = pageSize / elemSize;
        nextAvail = 0;
        bitmapLength = maxNumElems >>> 6;
        if ((maxNumElems & 63) != 0) {
            bitmapLength ++;
```

```
        }

        for (int i = 0; i < bitmapLength; i ++) {
            //bitmap 标识哪个子 page 被分配
            //0 标识未分配，1 表示已分配
            bitmap [i] = 0;
        }
    }
    //加到 arena 里面
    addToPool(head);
}
```

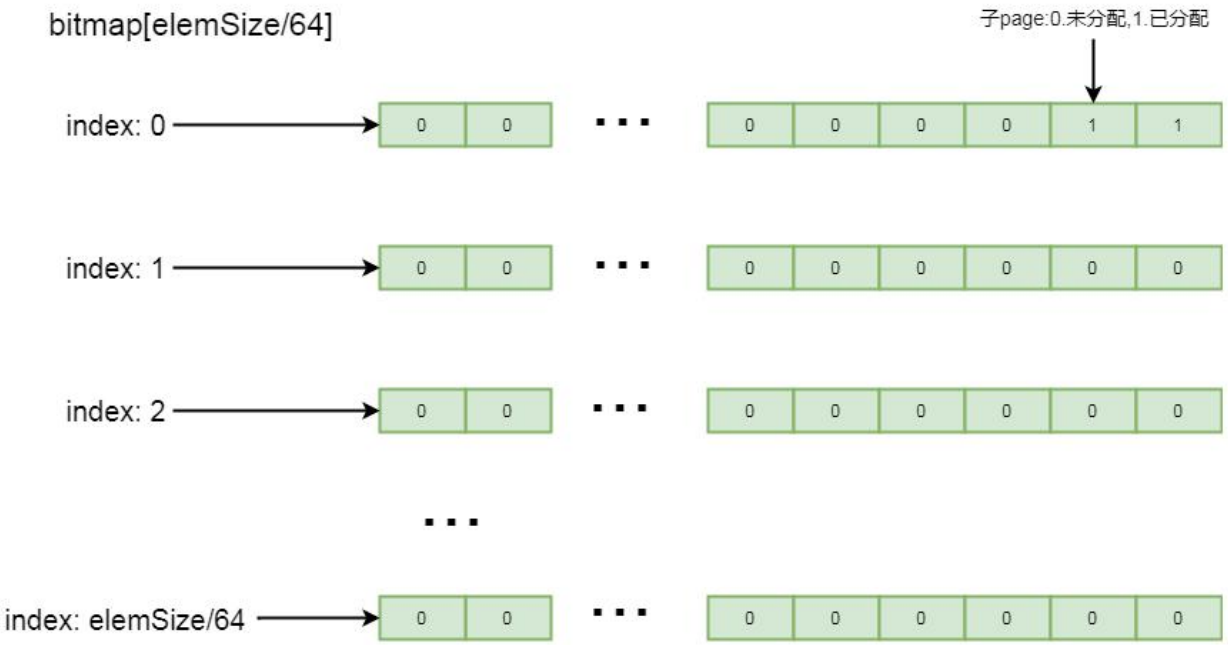
this.elemSize = elemSize 表示保存当前分配的缓冲区大小，这里我们以 16 字节举例，所以这里是 16。maxNumElems = numAvail = pageSize / elemSize 这里初始化了两个属性 maxNumElems, numAvail，值都为 pageSize / elemSize，表示一个 page 大小除以分配的缓冲区大小，也就是表示当前 page 被划分了多少分。

numAvail 则表示剩余可用的块数，由于第一次分配都是可用的，所以 numAvail=maxNumElems；

bitmapLength 表示 bitmap 的实际大小，刚才我们分析过，bitmap 初始化的大小为 8，但实际上并不一定需要 8 个元素，元素个数要根据 page 切分的子块而定，这里的大小是所切分的子块数除以 64。

再往下看， if ((maxNumElems & 63) != 0) 判断 maxNumElems 也就是当前配置所切分的子块是不是 64 的倍数，如果不是，则 bitmapLength 加 1,最后通过循环，将其分配的大小中的元素赋值为 0。

这里详细分析一下 bitmap，这里是个 long 类型的数组，long 数组中的每一个值，也就是 long 类型的数字，其中的每一个比特位，都标记着 page 中每一个子块的内存是否已分配，如果比特位是 1，表示该子块已分配，如果比特位是 0，表示该子块未分配，标记顺序是其二进制数从低位到高位进行排列。我们应该知道为什么 bitmap 大小要设置为子块数量除以，64，因为 long 类型的数字是 64 位，每一个元素能记录 64 个子块的数量，这样就可以通过子 page 个数除以 64 的方式决定 bitmap 中元素的数量。如果子块不能整除 64，则通过元素数量+1 方式，除以 64 之后剩余的子块通过 long 中比特位由低到高进行排列记录，其逻辑结构如下图所示：

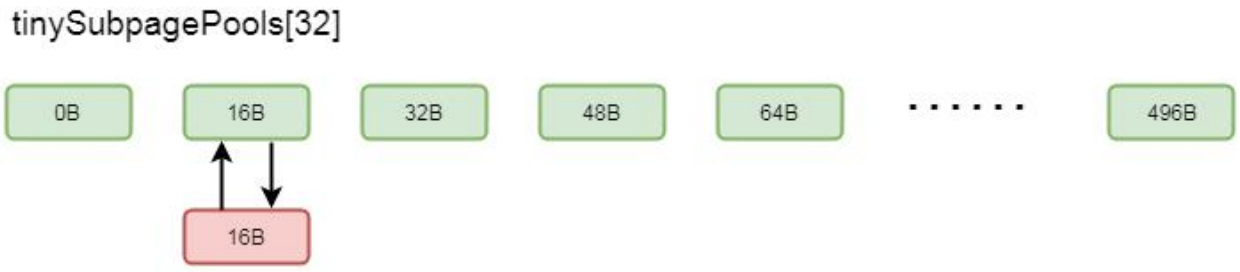


进入 PoolSubpage 的 addToPool(head)方法：

```
private void addToPool(PoolSubpage<T> head) {
    assert prev == null && next == null;
    prev = head;
    next = head.next;
    next.prev = this;
    head.next = this;
}
```

这里的 head 我们刚才讲过，是 Arena 中数组 tinySubpagePools 中的元素，通过以上逻辑，就会将新创建的 Subpage

通过双向链表的方式关联到 tinySubpagePools 中的元素，我们以 16 字节为例，关联关系如图所示：



这样，下次如果还需要分配 16 字节的内存，就可以通过 tinySubpagePools 找到其元素关联的 subpage 进行分配了。

我们再回到 PoolChunk 的 allocateSubpage()方法：

```
private long allocateSubpage(int normCapacity) {
    PoolSubpage<T> head = arena.findSubpagePoolHead(normCapacity);
    synchronized (head) {
        int d = maxOrder;
        //表示在第 11 层分配节点
        int id = allocateNode(d);
        if (id < 0) {
            return id;
        }

        //获取初始化的 subpage
        final PoolSubpage<T>[] subpages = this.subpages;
        final int pageSize = this.pageSize;

        freeBytes -= pageSize;
        //表示第几个 subpageIdx
        int subpageIdx = subpageIdx(id);
        PoolSubpage<T> subpage = subpages[subpageIdx];
        if (subpage == null) {
            //如果 subpage 为空
            subpage = new PoolSubpage<T>(head, this, id, runOffset(id), pageSize, normCapacity);
            //则将当前的下标赋值为 subpage
            subpages[subpageIdx] = subpage;
        } else {
            subpage.init(head, normCapacity);
        }
        //取出一个子 page
        return subpage.allocate();
    }
}
```

创建完了一个 subpage，我们就可以通过 subpage.allocate()方法进行内存分配了。我们跟到 allocate()方法中:

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

    if (numAvail == 0 || !doNotDestroy) {
        return -1;
    }
    //取一个 bitmap 中可用的 id(绝对 id)
    final int bitmapIdx = getNextAvail();
    //除以 64(bitmap 的相对下标)
    int q = bitmapIdx >>> 6;
    //除以 64 取余，其实就是当前绝对 id 的偏移量
    int r = bitmapIdx & 63;
    assert (bitmap[q] >>> r & 1) == 0;

    //当前位标记为 1
    bitmap[q] |= 1L << r;
    //如果可用的子 page 为 0
    //可用的子 page-1
    if (-- numAvail == 0) {
        //则移除相关子 page
        removeFromPool();
    }
    //bitmapIdx 转换成 handler
    return toHandle(bitmapIdx);
}
```

这里的逻辑看起来比较复杂，我带大家一点点来剖析，首先看：

```
final int bitmapIdx = getNextAvail();
```

其中 bitmapIdx 表示从 bitmap 中找到一个可用的 bit 位的下标，注意，这里是 bit 的下标，并不是数组的下标，我们之前分析过，因为每一比特位代表一个子块的内存分配情况，通过这个下标就可以知道那个比特位是未分配状态，我们跟进去：

```
private int getNextAvail() {
  //nextAvail=0
  int nextAvail = this.nextAvail;
  if (nextAvail >= 0) {
      //一个子 page 被释放之后，会记录当前子 page 的 bitmapIdx 的位置，下次分配可以直接通过 bitmapIdx 拿到一个子 page
      this.nextAvail = -1;
      return nextAvail;
  }
  return findNextAvail();
}
```

上述代码片段中的 nextAvail，表示下一个可用的 bitmapIdx，在释放的时候的会被标记，标记被释放的子块对应 bitmapIdx 的下标，如果<0 则代表没有被释放的子块，则通过 findNextAvail 方法进行查找，继续跟进 findNextAvail() 方法:

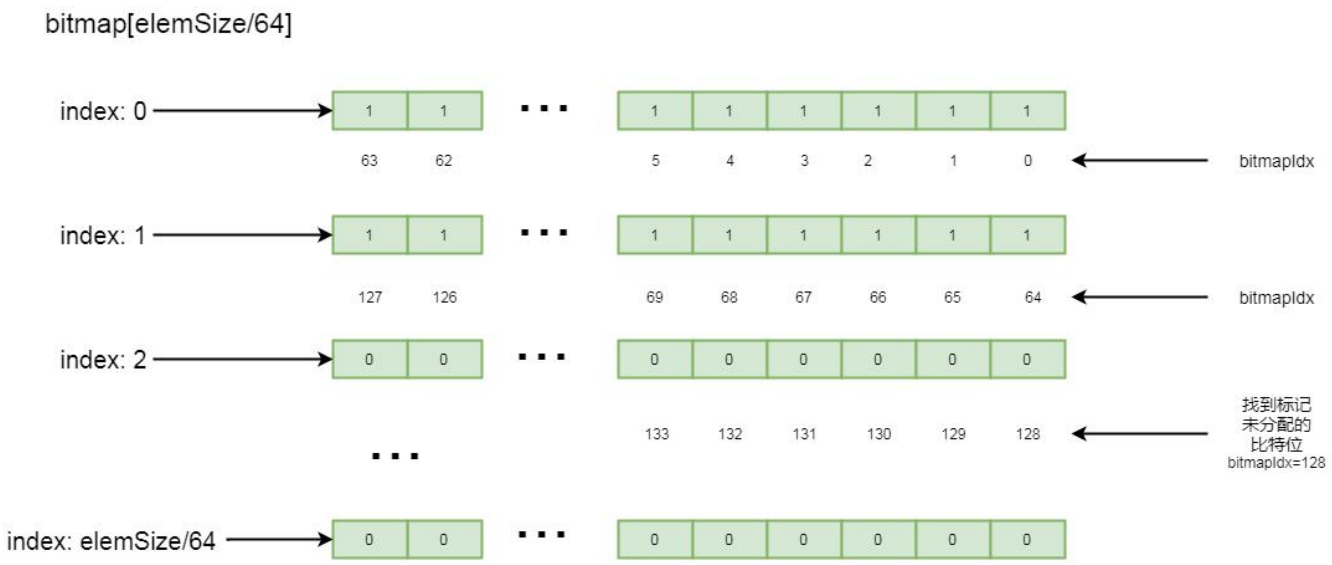
```
  private int findNextAvail() {
    //当前 long 数组
    final long[] bitmap = this.bitmap;
    //获取其长度
    final int bitmapLength = this.bitmapLength;
    for (int i = 0; i < bitmapLength; i ++) {
        //第 i 个
        long bits = bitmap[i];
        //!=-1 说明 64 位没有全部占满
        if (~bits != 0) {
            //找下一个节点
            return findNextAvail0(i, bits);
        }
    }
    return -1;
}
```

这里会遍历 bitmap 中的每一个元素，如果当前元素中所有的比特位并没有全部标记被使用，则通过 findNextAvail0(i, bits)方法一个一个往后找标记未使用的比特位。再继续跟 findNextAvail0()：

```
  private int findNextAvail0(int i, long bits) {
    //多少份
    final int maxNumElems = this.maxNumElems;
    //乘以 64，代表当前 long 的第一个下标
    final int baseVal = i << 6;
    //循环 64 次(指代当前的下标)
    for (int j = 0; j < 64; j ++) {
        //第一位为 0(如果是 2 的倍数，则第一位就是 0)
        if ((bits & 1) == 0) {
            //这里相当于加，将 i*64 之后加上 j，获取绝对下标
            int val = baseVal | j;
            //小于块数(不能越界)
            if (val < maxNumElems) {
```

```
                return val;
            } else {
                break;
            }
        }
        //当前下标不为0
        //右移一位
        bits >>>= 1;
    }
    return -1;
}
```

这里从当前元素的第一个比特位开始找，直到找到一个标记为 0 的比特位，并返回当前比特位的下标，大致流程如下图所示：



我们回到 allocate()方法中：

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

    if (numAvail == 0 || !doNotDestroy) {
        return -1;
    }
    //取一个 bitmap 中可用的 id(绝对 id)
    final int bitmapIdx = getNextAvail();
    //除以 64(bitmap 的相对下标)
    int q = bitmapIdx >>> 6;
    //除以 64 取余，其实就是当前绝对 id 的偏移量
    int r = bitmapIdx & 63;
    assert (bitmap[q] >>> r & 1) == 0;

    //当前位标记为 1
    bitmap[q] |= 1L << r;
```

```
    //如果可用的子 page 为 0
    //可用的子 page-1
    if (-- numAvail == 0) {
        //则移除相关子 page
        removeFromPool();
    }
    //bitmapIdx 转换成 handler
    return toHandle(bitmapIdx);
}
```

找到可用的 bitmapIdx 之后，通过 int q = bitmapIdx >>> 6 获取 bitmap 中 bitmapIdx 所属元素的数组下标。int r = bitmapIdx & 63 表示获取 bitmapIdx 的位置是从当前元素最低位开始的第几个比特位。bitmap[q] |= 1L << r 是将 bitmap 的位置设置为不可用，也就是比特位设置为 1，表示已占用。然后将可用子配置的数量 numAvail 减 1。如果没有可用子 page 的数量，则会将 PoolArena 中的数组 tinySubpagePools 所关联的 subpage 进行移除。最后通过 toHandle(bitmapIdx)获取当前子块的 handle，上一小节我们知道 handle 指向的是当前 chunk 中的唯一的一块内存，我们跟进 toHandle(bitmapIdx)中：

```
private long toHandle(int bitmapIdx) {
    return 0x4000000000000000L | (long) bitmapIdx << 32 | memoryMapIdx;
}
```

(long) bitmapIdx << 32 是将 bitmapIdx 右移 32 位，而 32 位正好是一个 int 的长度，这样，通过 (long) bitmapIdx << 32 | memoryMapIdx 计算，就可以将 memoryMapIdx，也就是 page 所属的下标的二进制数保存在 (long) bitmapIdx << 32 的低 32 位中。0x4000000000000000L 是一个最高位是 1 并且所有低位都是 0 的二进制数，这样通过按位或的方式可以将 (long) bitmapIdx << 32 | memoryMapIdx 计算出来的结果保存在 0x4000000000000000L 的所有低位中，这样，返回对的数字就可以指向 chunk 中唯一的一块内存，我们回到 PoolArena 的 allocateNormal 方法中：

```
private synchronized void allocateNormal(PooledByteBuf<T> buf, int reqCapacity, int normCapacity) {
    //首先在原来的 chunk 上进行内存分配(1)
    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)) {
        ++allocationsNormal;
        return;
    }

    //创建 chunk 进行内存分配(2)
    PoolChunk<T> c = newChunk(pageSize, maxOrder, pageShifts, chunkSize);
    long handle = c.allocate(normCapacity);
    ++allocationsNormal;
    assert handle > 0;
    //初始化 byteBuf(3)
    c.initBuf(buf, handle, reqCapacity);
    qInit.add(c);
}
```

我们分析完了 long handle = c.allocate(normCapacity)这步，这里返回的 handle 就指向 chunk 中的某个 page 中的某

个子块所对应的连续内存。最后，通过 iniBuf 初始化之后，将创建的 chunk 加到 ChunkList 里面，我们跟到 initBuf 方法中：

```
void initBuf(PooledByteBuf<T> buf, long handle, int reqCapacity) {
    int memoryMapIdx = memoryMapIdx(handle);
    //bitmapIdx 是后面分配 subpage 时候使用到的
    int bitmapIdx = bitmapIdx(handle);
    if (bitmapIdx == 0) {
        byte val = value(memoryMapIdx);
        assert val == unusable : String.valueOf(val);
        //runOffset(memoryMapIdx):偏移量
        //runLength(memoryMapIdx):当前节点的长度
        buf.init(this, handle, runOffset(memoryMapIdx), reqCapacity, runLength(memoryMapIdx),
                arena.parent.threadCache());
    } else {
        initBufWithSubpage(buf, handle, bitmapIdx, reqCapacity);
    }
}
```

这部分在前面我们剖析过，相信大家不会陌生，这里有区别的是 if (bitmapIdx == 0) 的判断，这里的 bitmapIdx 不会是 0，这样，就会走到 initBufWithSubpage(buf, handle, bitmapIdx, reqCapacity)方法中 ,跟到 initBufWithSubpage()方法：

```
private void initBufWithSubpage(PooledByteBuf<T> buf, long handle, int bitmapIdx, int reqCapacity) {
    assert bitmapIdx != 0;
    int memoryMapIdx = memoryMapIdx(handle);
    PoolSubpage<T> subpage = subpages[subpageIdx(memoryMapIdx)];
    assert subpage.doNotDestroy;
    assert reqCapacity <= subpage.elemSize;
    buf.init(
        this, handle,
        runOffset(memoryMapIdx) + (bitmapIdx & 0x3FFFFFFF) * subpage.elemSize, reqCapacity, subpage.elemSize,
        arena.parent.threadCache());
}
```

首先拿到 memoryMapIdx，这里会将我们之前计算 handle 传入，跟进去：

```
private static int memoryMapIdx(long handle) {
    return (int) handle;
}
```

这里将其强制转化为 int 类型，也就是去掉高 32 位，这样就得到 memoryMapIdx，回到 initBufWithSubpage 方法中：我们注意在 buf 调用 init 方法中的一个参数：runOffset(memoryMapIdx) + (bitmapIdx & 0x3FFFFFFF) * subpage.elemSize，这里的偏移量就是，原来 page 的偏移量+子块的偏移量：bitmapIdx & 0x3FFFFFFF 代表当前分配的子 page 是属于第几个子 page。(bitmapIdx & 0x3FFFFFFF) * subpage.elemSize 表示在当前 page 的偏移量。这样，分配的 ByteBuf 在内存读写的时候，就会根据偏移量进行读写。最后，我们跟到 init()方法中：

```
void init(PoolChunk<T> chunk, long handle, int offset, int length, int maxLength, PoolThreadCache cache) {

    //初始化
    assert handle >= 0;
    assert chunk != null;
```

```
    //在哪一块内存上进行分配的
    this.chunk = chunk;

    //这一块内存上的哪一块连续内存
    this.handle = handle;
    memory = chunk.memory;

    //偏移量
    this.offset = offset;
    this.length = length;
    this.maxLength = maxLength;
    tmpNioBuf = null;
    this.cache = cache;
}
```

这里又是我们熟悉的逻辑，初始化了属性之后，一个缓冲区分配完成，以上就是 Subpage 级别的缓冲区分配逻辑。

10.4.7 内存池 ByteBuf 内存回收

在前面的章节中我们有提到，堆外内存是不受JVM垃圾回收机制控制的，所以我们分配一块堆外内存进行ByteBuf操作时，使用完毕要对对象进行回收，本节就以 PooledUnsafeDirectByteBuf 为例讲解有关内存分配的相关逻辑。

PooledUnsafeDirectByteBuf 中内存释放的入口方法是其父类 AbstractReferenceCountedByteBuf 中的 release()方法：

```
@Override
public boolean release() {
    return release0(1);
}
```

这里调用了 release0()，跟进去：

```
private boolean release0(int decrement) {
   for (;;) {
       int refCnt = this.refCnt;
       if (refCnt < decrement) {
           throw new IllegalReferenceCountException(refCnt, -decrement);
       }
       if (refCnt == decrement) {
           deallocate();
           return true;
       }
       return false;
   }
}
```

if (refCnt == decrement) 中判断当前 byteBuf 是否没有被引用了，如果没有被引用，则通过 deallocate()方法进行释放。因为我们是以 PooledUnsafeDirectByteBuf 为例，所以这里会调用其父类 PooledByteBuf 的 deallocate()方法：

```
protected final void deallocate() {
   if (handle >= 0) {
       final long handle = this.handle;
      //表示当前的 ByteBuf 不再指向任何一块内存
```

```
        this.handle = -1;
       //这里将 memory 也设置为 null
        memory = null;
       //这一步是将 ByteBuf 的内存进行释放
        chunk.arena.free(chunk, handle, maxLength, cache);
       //将对象放入的对象回收站，循环利用
        recycle();
    }
}
```

我们首先来分析 free()方法：

```
 void free(PoolChunk<T> chunk, long handle, int normCapacity, PoolThreadCache cache) {
    //是否为 unpooled
    if (chunk.unpooled) {
        int size = chunk.chunkSize();
        destroyChunk(chunk);
        activeBytesHuge.add(-size);
        deallocationsHuge.increment();
    } else {
        //那种级别的 Size
        SizeClass sizeClass = sizeClass(normCapacity);
        //加到缓存里
        if (cache != null && cache.add(this, chunk, handle, normCapacity, sizeClass)) {
            return;
        }
        //将缓存对象标记为未使用
        freeChunk(chunk, handle, sizeClass);
    }
}
```

首先判断是不是 unpooled，我们这里是 Pooled，所以会走到 else 块中：

sizeClass(normCapacity)计算是哪种级别的 size，我们按照 tiny 级别进行分析；

cache.add(this, chunk, handle, normCapacity, sizeClass)是将当前当前 ByteBuf 进行缓存；

我们之前讲过，再分配 ByteBuf 时首先在缓存上分配，而这步，就是将其缓存的过程，继续跟进去：

```
boolean add(PoolArena<?> area, PoolChunk chunk, long handle, int normCapacity, SizeClass sizeClass) {
    //拿到 MemoryRegionCache 节点
    MemoryRegionCache<?> cache = cache(area, normCapacity, sizeClass);
    if (cache == null) {
        return false;
    }
    //将 chunk，和 handle 封装成实体加到 queue 里面
    return cache.add(chunk, handle);
}
```

首先根据根据类型拿到相关类型缓存节点，这里会根据不同的内存规格去找不同的对象，我们简单回顾一下，每个缓存对象都包含一个 queue，queue 中每个节点是 entry，每一个 entry 中包含一个 chunk 和 handle，可以指向唯一的连续的内存，我们跟到 cache 中：

```
private MemoryRegionCache<?> cache(PoolArena<?> area, int normCapacity, SizeClass sizeClass) {
    switch (sizeClass) {
    case Normal:
```

```
        return cacheForNormal(area, normCapacity);
    case Small:
        return cacheForSmall(area, normCapacity);
    case Tiny:
        return cacheForTiny(area, normCapacity);
    default:
        throw new Error();
    }
}
```

假设我们是 tiny 类型，这里就会走到 cacheForTiny(area, normCapacity)方法中，跟进去：

```
private MemoryRegionCache<?> cacheForTiny(PoolArena<?> area, int normCapacity) {
    int idx = PoolArena.tinyIdx(normCapacity);
    if (area.isDirect()) {
        return cache(tinySubPageDirectCaches, idx);
    }
    return cache(tinySubPageHeapCaches, idx);
}
```

这个方法我们之前剖析过，就是根据大小找到第几个缓存中的第几个缓存，拿到下标之后，通过 cache 去超相对应的缓存对象：

```
private static <T>  MemoryRegionCache<T> cache(MemoryRegionCache<T>[] cache, int idx) {
    if (cache == null || idx > cache.length - 1) {
        return null;
    }
    return cache[idx];
}
```

我们这里看到，是直接通过下标拿的缓存对象，回到 add()方法中：

```
boolean add(PoolArena<?> area, PoolChunk chunk, long handle, int normCapacity, SizeClass sizeClass) {
    //拿到 MemoryRegionCache 节点
    MemoryRegionCache<?> cache = cache(area, normCapacity, sizeClass);
    if (cache == null) {
        return false;
    }
    //将 chunk，和 handle 封装成实体加到 queue 里面
    return cache.add(chunk, handle);
}
```

这里的 cache 对象调用了一个 add 方法，这个方法就是将 chunk 和 handle 封装成一个 entry 加到 queue 里面，我们跟到 add()方法中：

```
public final boolean add(PoolChunk<T> chunk, long handle) {
    Entry<T> entry = newEntry(chunk, handle);
    boolean queued = queue.offer(entry);
    if (!queued) {
        entry.recycle();
    }
    return queued;
}
```

我们之前介绍过，从在缓存中分配的时候从 queue 弹出一个 entry，会放到一个对象池里面，而这里 Entry<T> entry = newEntry(chunk, handle)就是从对象池里去取一个 entry 对象，然后将 chunk 和 handle 进行赋值，然后通过

queue.offer(entry)加到 queue，我们回到 free()方法中：

```
void free(PoolChunk<T> chunk, long handle, int normCapacity, PoolThreadCache cache) {
    //是否为 unpooled
    if (chunk.unpooled) {
        int size = chunk.chunkSize();
        destroyChunk(chunk);
        activeBytesHuge.add(-size);
        deallocationsHuge.increment();
    } else {
        //那种级别的 Size
        SizeClass sizeClass = sizeClass(normCapacity);
        //加到缓存里
        if (cache != null && cache.add(this, chunk, handle, normCapacity, sizeClass)) {
            return;
        }
        freeChunk(chunk, handle, sizeClass);
    }
}
```

这里加到缓存之后，如果成功，就会 return，如果不成功，就会调用 freeChunk(chunk, handle, sizeClass)方法，这个方法的意义是，将原先给 ByteBuf 分配的内存区段标记为未使用，跟进 freeChunk()简单分析下：

```
void freeChunk(PoolChunk<T> chunk, long handle, SizeClass sizeClass) {
    final boolean destroyChunk;
    synchronized (this) {
        switch (sizeClass) {
        case Normal:
            ++deallocationsNormal;
            break;
        case Small:
            ++deallocationsSmall;
            break;
        case Tiny:
            ++deallocationsTiny;
            break;
        default:
            throw new Error();
        }
        destroyChunk = !chunk.parent.free(chunk, handle);
    }
    if (destroyChunk) {
        destroyChunk(chunk);
    }
}
```

我们再跟到 free()方法中：

```
boolean free(PoolChunk<T> chunk, long handle) {
    chunk.free(handle);
    if (chunk.usage() < minUsage) {
        remove(chunk);
        return move0(chunk);
    }
    return true;
}
```

chunk.free(handle)的意思是通过 chunk 释放一段连续的内存，再跟到 free()方法中：

```
void free(long handle) {
    int memoryMapIdx = memoryMapIdx(handle);
```

```
    int bitmapIdx = bitmapIdx(handle);

    if (bitmapIdx != 0) {
        PoolSubpage<T> subpage = subpages[subpageIdx(memoryMapIdx)];
        assert subpage != null && subpage.doNotDestroy;
        PoolSubpage<T> head = arena.findSubpagePoolHead(subpage.elemSize);
        synchronized (head) {
            if (subpage.free(head, bitmapIdx & 0x3FFFFFFF)) {
                return;
            }
        }
    }
    freeBytes += runLength(memoryMapIdx);
    setValue(memoryMapIdx, depth(memoryMapIdx));
    updateParentsFree(memoryMapIdx);
}
```

if (bitmapIdx != 0)这 里判断是当前缓冲区分配的级别是 Page 还是 Subpage，如果是 Subpage，则会找到相关的 Subpage 将其位图标记为 0，如果不是 subpage，这里通过分配内存的反向标记，将该内存标记为未使用。这段逻辑大家可以自行分析，如果之前分配相关的知识掌握扎实的话，这里的逻辑也不是很难。回到 PooledByteBuf 的 deallocate 方法中：

```
protected final void deallocate() {
    if (handle >= 0) {
        final long handle = this.handle;
        this.handle = -1;
        memory = null;
        chunk.arena.free(chunk, handle, maxLength, cache);
        recycle();
    }
}
```

最后，通过 recycle()将释放的 ByteBuf 放入对象回收站，有关对象回收站的知识，会在以后的章节进行剖析，以上就是内存回收的大概逻辑。

10.4.8 SocketChannel 读取 ByteBuf 的过程

本节知识和之前的分析过很多知识有关联，我们不再重复介绍。因此，学习本节之前，小伙伴们可以再恶补一下最前面的几章内容，如客户端接入的流程，客户端发送数据，Server 读取数据的流程等。我们首先看 NioEventLoop 的 processSelectedKey 方法：

```
private void processSelectedKey(SelectionKey k, AbstractNioChannel ch) {
    //获取到 channel 中的 unsafe
    final AbstractNioChannel.NioUnsafe unsafe = ch.unsafe();
    //如果这个 key 不是合法的，说明这个 channel 可能有问题
    if (!k.isValid()) {
        //代码省略
    }
```

```
    try {
        //如果是合法的，拿到 key 的 io 事件
        int readyOps = k.readyOps();
        //链接事件
        if ((readyOps & SelectionKey.OP_CONNECT) != 0) {
            int ops = k.interestOps();
            ops &= ~SelectionKey.OP_CONNECT;
            k.interestOps(ops);
            unsafe.finishConnect();
        }
        //写事件
        if ((readyOps & SelectionKey.OP_WRITE) != 0) {
            ch.unsafe().forceFlush();
        }
        //读事件和接受链接事件
        //如果当前 NioEventLoop 是 work 线程的话，这里就是 op_read 事件
        //如果是当前 NioEventLoop 是 boss 线程的话，这里就是 op_accept 事件
        if ((readyOps & (SelectionKey.OP_READ | SelectionKey.OP_ACCEPT)) != 0 || readyOps == 0) {
            unsafe.read();
            if (!ch.isOpen()) {
                return;
            }
        }
    } catch (CancelledKeyException ignored) {
        unsafe.close(unsafe.voidPromise());
    }
}
```

if ((readyOps & (SelectionKey.OP_READ | SelectionKey.OP_ACCEPT)) != 0 || readyOps == 0) 这里的判断表示轮询到大事件是 OP_READ 或者 OP_ACCEPT 事件。之前我们分析过，如果当前 NioEventLoop 是 work 线程的话，那么这里就是 OP_READ 事件，也就是读事件，表示客户端发来了数据流，这里会调用 unsafe 的 redis()方法进行读取。如果是 work 线程，那么这里的 channel 是 NioServerSocketChannel，其绑定的 unsafe 是 NioByteUnsafe，这里会走进 NioByteUnsafe 的 read()方法中：

```
public final void read() {
        final ChannelConfig config = config();
        final ChannelPipeline pipeline = pipeline();
        final ByteBufAllocator allocator = config.getAllocator();
        final RecvByteBufAllocator.Handle allocHandle = recvBufAllocHandle();
        allocHandle.reset(config);

        ByteBuf byteBuf = null;
        boolean close = false;
        try {
            do {
                byteBuf = allocHandle.allocate(allocator);
                allocHandle.lastBytesRead(doReadBytes(byteBuf));
                if (allocHandle.lastBytesRead() <= 0) {
                    byteBuf.release();
                    byteBuf = null;
                    close = allocHandle.lastBytesRead() < 0;
                    break;
                }

                allocHandle.incMessagesRead(1);
                readPending = false;
```

```
                pipeline.fireChannelRead(byteBuf);
                byteBuf = null;
            } while (allocHandle.continueReading());

            allocHandle.readComplete();
            pipeline.fireChannelReadComplete();

            if (close) {
                closeOnRead(pipeline);
            }
        } catch (Throwable t) {
            handleReadException(pipeline, byteBuf, t, close, allocHandle);
        } finally {
            if (!readPending && !config.isAutoRead()) {
                removeReadOp();
            }
        }
    }
}
```

首先获取 SocketChannel 的 config, pipeline 等相关属性，final ByteBufAllocator allocator = config.getAllocator(); 这一步是获取一个 ByteBuf 的内存分配器，用于分配 ByteBuf。这里会走到 DefaultChannelConfig 的 getAllocator 方法中:

```
public ByteBufAllocator getAllocator() {
    return allocator;
}
```

这里返回的 DefualtChannelConfig 的成员变量，我们看这个成员变量：

```
private volatile ByteBufAllocator allocator = ByteBufAllocator.DEFAULT;
```

这里调用 ByteBufAllocator 的属性 DEFAULT，跟进去：

```
ByteBufAllocator DEFAULT = ByteBufUtil.DEFAULT_ALLOCATOR;
```

我们看到这里又调用了 ByteBufUtil 的静态属性 DEFAULT_ALLOCATOR，再跟进去：

```
static final ByteBufAllocator DEFAULT_ALLOCATOR;
```

DEFAULT_ALLOCATOR 这个属性是在 static 块中初始化的，我们跟到 static 块中：

```
static {
    String allocType = SystemPropertyUtil.get(
            "io.netty.allocator.type", PlatformDependent.isAndroid() ? "unpooled" : "pooled");
    allocType = allocType.toLowerCase(Locale.US).trim();

    ByteBufAllocator alloc;
    if ("unpooled".equals(allocType)) {
        alloc = UnpooledByteBufAllocator.DEFAULT;
        logger.debug("-Dio.netty.allocator.type: {}", allocType);
    } else if ("pooled".equals(allocType)) {
        alloc = PooledByteBufAllocator.DEFAULT;
        logger.debug("-Dio.netty.allocator.type: {}", allocType);
    } else {
        alloc = PooledByteBufAllocator.DEFAULT;
        logger.debug("-Dio.netty.allocator.type: pooled (unknown: {})", allocType);
    }
    DEFAULT_ALLOCATOR = alloc;
    //代码省略
}
```

首先判断运行环境是不是安卓，如果不是安卓，在返回"pooled"字符串保存在 allocType 中 ,然后通过 if 判断，最后局部

变量 alloc = PooledByteBufAllocator.DEFAULT，最后将 alloc 赋值到成员变量 DEFAULT_ALLOCATOR，我们跟到 PooledByteBufAllocator 的 DEFAULT 属性中：

```
public static final PooledByteBufAllocator DEFAULT =
        new PooledByteBufAllocator(PlatformDependent.directBufferPreferred());
```

我们看到这里直接通过 new 的方式，创建了一个 PooledByteBufAllocator 对象，也就是基于申请一块连续内存进行缓冲区分配的缓冲区分配器。缓冲区分配器的知识，我们在前面的章节进行了详细的剖析，这里就不再赘述。回到 NioByteUnsafe 的 read()方法中：

```
public final void read() {
        final ChannelConfig config = config();
        final ChannelPipeline pipeline = pipeline();
        final ByteBufAllocator allocator = config.getAllocator();
        final RecvByteBufAllocator.Handle allocHandle = recvBufAllocHandle();
        allocHandle.reset(config);

        ByteBuf byteBuf = null;
        boolean close = false;
        try {
            do {
                byteBuf = allocHandle.allocate(allocator);
                allocHandle.lastBytesRead(doReadBytes(byteBuf));
                if (allocHandle.lastBytesRead() <= 0) {
                    byteBuf.release();
                    byteBuf = null;
                    close = allocHandle.lastBytesRead() < 0;
                    break;
                }

                allocHandle.incMessagesRead(1);
                readPending = false;
                pipeline.fireChannelRead(byteBuf);
                byteBuf = null;
            } while (allocHandle.continueReading());

            allocHandle.readComplete();
            pipeline.fireChannelReadComplete();

            if (close) {
                closeOnRead(pipeline);
            }
        } catch (Throwable t) {
            handleReadException(pipeline, byteBuf, t, close, allocHandle);
        } finally {
            if (!readPending && !config.isAutoRead()) {
                removeReadOp();
            }
        }
    }
}
```

这里 ByteBufAllocator allocator = config.getAllocator()中的 allocator，就是 PooledByteBufAllocator。

final RecvByteBufAllocator.Handle allocHandle = recvBufAllocHandle() 是创建一个 handle，我们之前的章节讲过，

handle 是对 RecvByteBufAllocator 进行实际操作的对象，我们跟进 recvBufAllocHandle：

```
public RecvByteBufAllocator.Handle recvBufAllocHandle() {
    //如果不存在，则创建一个 handle 的实例
    if (recvHandle == null) {
        recvHandle = config().getRecvByteBufAllocator().newHandle();
    }
    return recvHandle;
}
```

这里是我们之前剖析过的逻辑，如果不存在，则创建 handle 的实例，具体创建过程我们可以回顾前面的章节，这里就不再赘述。同样 allocHandle.reset(config)是将配置重置，前面章节也对其进行过剖析。重置完配置之后，进行 do-while 循环，有关循环终止条件 allocHandle.continueReading()，之前小节也有过详细剖析，这里也不再赘述。在 do-while 循环中，首先看 byteBuf = allocHandle.allocate(allocator) 这一步，这里传入了刚才创建的 allocate 对象，也就是 PooledByteBufAllocator，这里会进入 DefaultMaxMessagesRecvByteBufAllocator 类的 allocate()方法中：

```
public ByteBuf allocate(ByteBufAllocator alloc) {
    return alloc.ioBuffer(guess());
}
```

这里的 guess 方法，会调用 AdaptiveRecvByteBufAllocator 的 guess 方法：

```
public int guess() {
    return nextReceiveBufferSize;
}
```

这里会返回 AdaptiveRecvByteBufAllocator 的成员变量 nextReceiveBufferSize，也就是下次所分配缓冲区的大小，根据我们之前学习的内容，第一次分配的时候会分配初始大小，也就是 1024 字节。这样，alloc.ioBuffer(guess())就会分配一个 PooledByteBuf，我们跟到 AbstractByteBufAllocator 的 ioBuffer 方法中：

```
public ByteBuf ioBuffer(int initialCapacity) {
    if (PlatformDependent.hasUnsafe()) {
        return directBuffer(initialCapacity);
    }
    return heapBuffer(initialCapacity);
}
```

这里首先判断是否能获取 jdk 的 unsafe 对象，默认为 true，所以会走到 directBuffer(initialCapacity)中，这里最终会分配一个 PooledUnsafeDirectByteBuf 对象，具体分配流程我们再之前小节做过详细剖析。回到 NioByteUnsafe 的 read()方法中，分配完了 ByteBuf 之后，再看这一步 allocHandle.lastBytesRead(doReadBytes(byteBuf))。

首先看参数 doReadBytes(byteBuf)方法，这步是将 channel 中的数据读取到我们刚分配的 ByteBuf 中，并返回读取到的字节数，这里会调用到 NioSocketChannel 的 doReadBytes()方法：

```
protected int doReadBytes(ByteBuf byteBuf) throws Exception {
```

```
    final RecvByteBufAllocator.Handle allocHandle = unsafe().recvBufAllocHandle();
    allocHandle.attemptedBytesRead(byteBuf.writableBytes());
    return byteBuf.writeBytes(javaChannel(), allocHandle.attemptedBytesRead());
}
```

首先拿到绑定在 channel 中的 handler，因为我们已经创建了 handle，所以这里会直接拿到。再看 allocHandle.attemptedBytesRead(byteBuf.writableBytes())这步，byteBuf.writableBytes()返回 byteBuf 的可写字节数，也就是最多能从 channel 中读取多少字节写到 ByteBuf，allocate 的 attemptedBytesRead 会把可写字节数设置到 DefaultMaxMessagesRecvByteBufAllocator 类的 attemptedBytesRead 属性中，跟到 DefaultMaxMessagesRecvByteBufAllocator 中的 attemptedBytesRead 我们会看到：

```
public void attemptedBytesRead(int bytes) {
    attemptedBytesRead = bytes;
}
```

继续看 doReadBytes()方法。往下看最后，通过 byteBuf.writeBytes(javaChannel(), allocHandle.attemptedBytesRead())将 jdk 底层的 channel 中的数据写入到我们创建的 ByteBuf 中，并返回实际写入的字节数。回到 NioByteUnsafe 的 read()方法中继续看 allocHandle.lastBytesRead(doReadBytes(byteBuf))这步，刚才我们剖析过 doReadBytes(byteBuf)返回的是世界写入 ByteBuf 的字节数，再看 lastBytesRead()方法，跟到 DefaultMaxMessagesRecvByteBufAllocator 的 lastBytesRead()方法中：

```
public final void lastBytesRead(int bytes) {
    lastBytesRead = bytes;
    totalBytesRead += bytes;
    if (totalBytesRead < 0) {
        totalBytesRead = Integer.MAX_VALUE;
    }
}
```

这里会赋值两个属性，lastBytesRead 代表最后读取的字节数，这里赋值为我们刚才写入 ByteBuf 的字节数，totalBytesRead 表示总共读取的字节数，这里将写入的字节数追加。继续来到 NioByteUnsafe 的 read()方法，如果最后一次读取数据为 0，说明已经将 channel 中的数据全部读取完毕，将新创建的 ByteBuf 释放循环利用，并跳出循环。allocHandle.incMessagesRead(1)这步是增加消息的读取次数，因为我们循环最多 16 次，所以当增加消息次数增加到 16 会结束循环。读取完毕之后，会通过 pipeline.fireChannelRead(byteBuf)将传递 channelRead 事件，有关 channelRead 事件，我们在前面的章节也进行了详细的剖析。

至此，小伙伴们应该有个疑问，如果一次读取不完，就传递 channelRead 事件，那么 server 接收到的数据有可能就是不

完整的，其实关于这点，Netty 也做了相应的处理，我们会在之后的章节详细剖析 Netty 的半包处理机制。循环结束后，会执行到 allocHandle.readComplete()这一步。

我们知道第一次分配 ByteBuf 的初始容量是 1024，但是初始容量不一定一定满足所有的业务场景，netty 中，将每次读取数据的字节数进行记录，然后之后次分配 ByteBuf 的时候，容量会尽可能的符合业务场景所需要大小，具体实现方式，就是在 readComplete()这一步体现的。我们跟到 AdaptiveRecvByteBufAllocator 的 readComplete()方法中：

```
public void readComplete() {
    record(totalBytesRead());
}
```

这里调用了 record()方法，并且传入了这一次所读取的字节总数，跟到 record()方法中：

```
private void record(int actualReadBytes) {
    if (actualReadBytes <= SIZE_TABLE[Math.max(0, index - INDEX_DECREMENT - 1)]) {
        if (decreaseNow) {
            index = Math.max(index - INDEX_DECREMENT, minIndex);
            nextReceiveBufferSize = SIZE_TABLE[index];
            decreaseNow = false;
        } else {
            decreaseNow = true;
        }
    } else if (actualReadBytes >= nextReceiveBufferSize) {
        index = Math.min(index + INDEX_INCREMENT, maxIndex);
        nextReceiveBufferSize = SIZE_TABLE[index];
        decreaseNow = false;
    }
}
```

首先看判断条件 if (actualReadBytes <= SIZE_TABLE[Math.max(0, index - INDEX_DECREMENT - 1)]) 。这里 index 是当前分配的缓冲区大小所在的 SIZE_TABLE 中的索引，将这个索引进行缩进，然后根据缩进后的所以找出 SIZE_TABLE 中所存储的内存值，再判断是否大于等于这次读取的最大字节数，如果条件成立，说明分配的内存过大，需要缩容操作，我们看 if 块中缩容相关的逻辑。首先 if (decreaseNow) 会判断是否立刻进行收缩操作，通常第一次不会进行收缩操作，然后会将 decreaseNow 设置为 true，代表下一次直接进行收缩操作。假设需要立刻进行收缩操作，我们看收缩操作的相关逻辑：

index = Math.max(index - INDEX_DECREMENT, minIndex) 这一步将索引缩进一步，但不能小于最小索引值；

然后通过 nextReceiveBufferSize = SIZE_TABLE[index] 获取设置索引之后的内存，赋值在 nextReceiveBufferSize，也就是下次需要分配的大小，下次就会根据这个大小分配 ByteBuf 了，这样就实现了缩容操作。

再看 else if (actualReadBytes >= nextReceiveBufferSize) ，这里判断这次读取字节的总量比上次分配的大小还要大，

则进行扩容操作。扩容操作也很简单，索引步进，然后拿到步进后的索引所对应的内存值，作为下次所需要分配的大小

在 NioByteUnsafe 的 read()方法中，经过了缩容或者扩容操作之后，通过 pipeline.fireChannelReadComplete()传播 ChannelReadComplete()事件，以上就是读取客户端消息的相关流程。