

# NETTY 的 BYTEBUF 源码分析

作者：淋雨

<b>NETTY 的 BYTEBUF 源码分析 .....</b>	<b>1</b>
<b>一、 Java Buffer 的相关基础知识.....</b>	<b>3</b>
1. Java 基本数据类型.....	3
2. Big-Endian 和 Little-Endian.....	4
3. 对象池技术 .....	6
4. 对象引用 .....	6
5. buddy allocation 和 slab allocation 内存分配算法.....	7
<b>二、 ByteBuf 整体结构的分析 .....</b>	<b>8</b>
1. ByteBuf 整体结构.....	9
<b>三、 ByteBuf 抽象层分析 .....</b>	<b>9</b>
1. ReferenceCounted 接口.....	9
2. ByteBuf 抽象类.....	10
3. AbstractByteBuf 类 .....	13
4. AbstractReferenceCountedByteBuf 类 .....	17
<b>四、 Buf 的 Unpooled 的实现.....</b>	<b>18</b>
1. UnpooledHeapByteBuf .....	19
2. UnpooledDirectByteBuf .....	21
3. UnpooledUnsafeDirectByteBuf .....	22
<b>五、 Pooled buffer 的实现.....</b>	<b>22</b>
1. Pooled Buffer 整体结构 .....	23
2. PoolChunk 的结构.....	24
3. PoolChunkList 的结构.....	27
4. PoolArena 的结构.....	28
5. 从对象池中获取 buffer .....	29
6. Pooled Buffer 内存监控 .....	34

## 一、Java Buffer 的相关基础知识

### 1. Java 基本数据类型

Java 中有 8 种基本类型: byte,char,short,int,long,float,double,boolean 如下图:

原始类型	字节数	取值范围	包装类	运算类型	JVM 运算指令
byte	1	-128 ~ 127	Byte	int	iadd、isub、 idiv、irem、ineg
boolean	1	0、1	Boolean	int	
short	2	-32768 ~ 32767	Short	int	
int	4	-2147483648 ~ 2147483647	Integer	int	
float	4		Float	float	fadd、fsub、 fmul、fdiv、frem 、fneg
long	8		Long	long	ladd、lsub、 lmul、ldiv、lrem 、lneg
double	8		Double	double	dadd、dsub、 dmul、ddiv、drem 、dneg
char	2	持 65536 个字符	Character	char	

JVM 对于这 8 种基本类型的存储、访问、运算都直接在 java 栈内存中进行, 在方法调用时存储在栈的本地局部变量中, 运算完成后或者方法退出时可以立即清除。这样提供了快速、高效的访问、运算和回收的方式。

从上图中可以看出 8 种基本类型在存储的时候占用的是 1-8 个字节不等存储空间, 假如给你这些字节的数组你该如何存储进入这些值呢, 下面我来做一个简单演示。

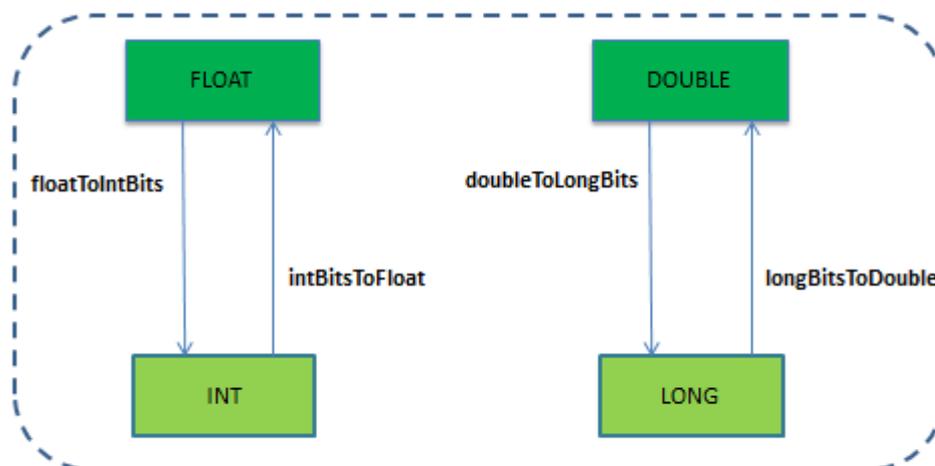
```
//存储short类型的值
public byte[] setShort(int value) {
    byte[] array = new byte[2];
    array[0] = (byte) (value >>> 8);
    array[1] = (byte) value;
    return array;
}

//存储int类型的值
protected byte[] setInt(int index, int value) {
    byte[] array = new byte[4];
    array[0] = (byte) (value >>> 24);
    array[1] = (byte) (value >>> 16);
    array[2] = (byte) (value >>> 8);
    array[3] = (byte) value;
    return array;
}
```

```
//存储long类型的值
protected byte[] setLong(int index, long value) {
    byte[] array = new byte[8];
    array[0] = (byte) (value >>> 56);
    array[1] = (byte) (value >>> 48);
    array[2] = (byte) (value >>> 40);
    array[3] = (byte) (value >>> 32);
    array[4] = (byte) (value >>> 24);
    array[5] = (byte) (value >>> 16);
    array[6] = (byte) (value >>> 8);
    array[7] = (byte) value;
    return array;
}
```

通过上面的代码演示，我们通过位移的运算分别将 short、int、long 类型的值存储到 2、4、8 个字节中，每个字节存储的是数值在这个字节内的表示。

那对于 float 和 double 又该如何存储呢，JDK 的 API 给我们提供了 float 的包装类 Float 提供了一个将 float 转换成 int 值的方法 floatToIntBits，并提供了将转换后的 int 再转换成 float 逆向解析方法 intBitsToFloat。这样我们就可以先将 float 转换成 int 进行存储了。double 类型同理可以转换成 long。



## 2. Big-Endian 和 Little-Endian

### ◆ 字节序定义

字节序，顾名思义字节的顺序，再多说两句就是大于一个字节类型的数据在内存中的存放顺序。在单个平台内部编程由于采用的是统一的编码排序，其实开发人员一般不用考虑这个问题，唯有在跨平台以及网络程序中才需要考虑这个问题。

在所有操作系统中，字节排序分总共为两类：。引用标准的 Big-Endian 和 Little-Endian 的定义如下：

- a) Little-Endian 就是低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。例子如下：

```
// 存储short类型的值
public byte[] setShort(int value) {
    byte[] array = new byte[2];
    array[0] = (byte) (value >>> 8); //高位在前
    array[1] = (byte) value;         //低位在后
    return array;
}
```

- b) Big-Endian 就是高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。例子如下：

```
// 存储short类型的值
public byte[] setShort(int value) {
    byte[] array = new byte[2];
    array[0] = (byte) value;         //低位在前
    array[1] = (byte) (value >>> 8); //高位在后
    return array;
}
```

- c) 网络字节序：在网络传输中，4个字节的32 bit 值以下面的次序传输：首先是0~7bit，其次8~15bit，然后16~23bit，最后是24~31bit。这种传输次序称作大端字节序。由于TCP/IP首部中所有的二进制整数在网络中传输时都要求以这种次序，因此它又称作网络字节序。这种只是按照字节的前后顺序去传输，并不涉及你用字符编码的顺序。

#### ◆ ByteOrder

在java的ByteOrder类中定义BIG\_ENDIAN和LITTLE\_ENDIAN两个字段，来控制数值在字节缓冲区中排序的规则，

#### ◆ 系统相关性

在/usr/include/中(包括子目录)查找字符串BYTE\_ORDER(或\_BYTE\_ORDER, \_\_BYTE\_ORDER)，确定其值。这个值一般在endian.h或machine/endian.h文件中可以找到，有时在feature.h中，不同的操作系统可能有所不同。一般来说，Little Endian系统BYTE\_ORDER(或\_BYTE\_ORDER, \_\_BYTE\_ORDER)为1234，Big Endian系统为4321。大部分用户的操作系统(如windows, Linux)是Little Endian的。少部分如MAC OS,是Big Endian的。本质上说，Little Endian还是Big Endian与操作系统和芯片类型都有关系。

下面是各种芯片操作系统对应的编码顺序参照表：

操作系统	编码排序方式
x86 系列(Intel, AMD, ...)	little-endian
DEC Alpha	little-endian
HP-PA NT	little-endian
HP-PA UNIX	big-endian
SUN SPARC All	little-endian
MIPS NT	little-endian
MIPS UNIX	big-endian
PowerPC NT	little-endian
PowerPC non-NT	big-endian

RS/6000 UNIX	big-endian
Motorola m68k	big-endian

### 3. 对象池技术

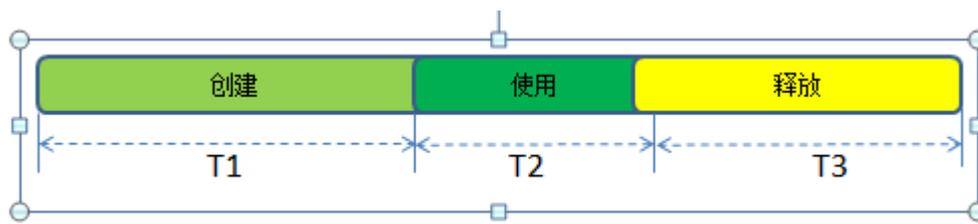
在 Java 对象的生命周期大致包括三个阶段：对象的创建，对象的使用，对象的清除。



因此对象的生命周期长度可用如下的表达式表示： $T = T1 + T2 + T3$ 。其中  $T1$  表示对象的创建时间， $T2$  表示对象的使用时间，而  $T3$  则表示其清除时间。其实能真正对我们产生作用的只有  $T2$  周期的时间，而  $T1$ 、 $T3$  则是对象本身的开销。

整个对象的使用率： $OP = T2 / (T1 + T2 + T3)$

对于有的对象创建周期  $T1$ 、释放周期  $T3$  都比较大的情况下，例如数据库连接，网络连接等。我们的对象使用率就会很低，也就是意味着计算机的系统资源大部分都用来对象的创建和释放了。如下图：



对象池技术是将用过的对象保存起来，等下一次需要这种对象的时候，再拿出来重复使用，从而在一定程度上减少频繁创建对象所造成的开销。用于充当保存对象的“容器”的对象，被称为“对象池”（Object Pool，或简称 Pool）。

对于没有状态的对象（例如 String），在重复使用之前，无需进行任何处理；对于有状态的对象（例如 StringBuffer），在重复使用之前，就需要把它们恢复到等同于刚刚生成时的状态。由于条件的限制，恢复某个对象的状态的操作不可能实现了的话，就得把这个对象抛弃，改用新创建的实例了。

并非所有对象都适合放在对象池中，因为维护对象池也要造成一定开销。对生成时开销不大的对象进行池化，反而可能会出现“维护对象池的开销”大于“生成新对象的开销”，从而使性能降低的情况。对于有的对象创建周期  $T1$ 、释放周期  $T3$  都比较大的情况下，对象池技术就是提高性能的有效策略了。

### 4. 对象引用

在 JDK 1.2 以前的版本中，若一个对象不被任何变量引用，那么程序就无法再使用这个对象。也就是说，只有对象处于可触及（reachable）状态，程序才能使用它。从 JDK 1.2 版本开始，把对象的引用分为 4 种级别，从而使程序能更加灵活地控制对象的生命周期。这 4 种级别由高到低依次为：强引用、软引用、弱引用和虚引用。

#### ◆ 强引用（StrongReference）

强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。当内存空间不足，Java 虚拟机宁愿抛出 `OutOfMemoryError` 错误，使程序异常终止，也不会靠随意回收具有强引用的对象。 `String name=new String ("icy");`

#### ◆ 软引用（SoftReference）

如果一个对象只具有软引用，则内存空间足够，GC 不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要 GC 没有回收它，该对象就可以被程序使用。软引用可用于来实现内存敏感的高速缓存。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被 GC 回收，Java 虚拟机就会把这个软引用加入到与之关联的引用队列中。

#### ◆ 弱引用（WeakReference）

弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在 GC 线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java 虚拟机就会把这个弱引用加入到与之关联的引用队列中。

#### ◆ 虚引用（PhantomReference）

“虚引用”顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，他并不会对 GC 的执行策略产生任何影响。

虚引用主要用来跟踪对象被 GC 回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列（ReferenceQueue）联合使用。当 GC 准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之 关联的引用队列中。

常可以用它来检测对象池中的对象是否存在内存泄漏的问题，例如某个对象直接从对象池中取出对象，我们可以将它维护一个虚引用记录下来，如果这个对象被回收了，没有对象池的信息更新的话就可以通过这个监控来得到，是否存在泄漏。

### 5. buddy allocation 和 slab allocation 内存分配算法

#### ◆ buddy allocation

buddy 算法是用来做内存管理的经典算法，是为了解决内存的外碎片。buddy 算法将所有空闲页框分组为 11 个块链表，每个块链表的每个块元素分别包含

1,2,4,8,16,32,64,128,256,512,1024 个连续的页框，每个块的第一个页框的物理地址是该块大小的整数倍。如，大小为 16 个页框的块，其起始地址是  $16 * 2^{12}$  (一个页框的大小为  $4k, 16$  个页框的大小为  $16 * 4k, 1k=1024=2$  的 10 次方， $4k=2$  的 12 次方) 的倍数。

例，假设要请求一个 128 个页框的块，算法先检查 128 个页框的链表是否有空闲块，如果没有则查 256 个页框的链表，有则将 256 个页框的块分裂两份，一份使用，一份插入 128 个页框的链表。如果还没有，就查 512 个页框的链表，有的话就分裂为 128, 128, 256, 一个 128 使用，剩余两个插入对应链表。如果在 512 还没查到，则返回出错信号。

回收过程相反，内核试图把大小为  $b$  的空闲伙伴合并为一个大小为  $2b$  的单独快，满足以下条件的两个块称为伙伴：1, 两个块具有相同的大小，记做  $b$ ；2, 它们的物理地址是连续的，3, 第一个块的第一个页框的物理地址是  $2 * b * 2^{12}$  的倍数，该算法迭代，如果成功合并所释放的块，会试图合并  $2b$  的块来形成更大的块。

◆ slab allocation

Linux 所使用的 slab 分配器的基础是 Jeff Bonwick 为 SunOS 操作系统首次引入的一种算法。Jeff 的分配器是围绕对象缓存进行的。在内核中，会为有限的对象集（例如文件描述符和其他常见结构）分配大量内存。Jeff 发现对内核中普通对象进行初始化所需的时间超过了对其进行分配和释放所需的时间。因此他的结论是不应该将内存释放回一个全局的内存池，而是将内存保持为针对特定目而初始化的状态。例如，如果内存被分配给了一个互斥锁，那么只需在为互斥锁首次分配内存时执行一次互斥锁初始化函数（mutex\_init）即可。后续的内存分配不需要执行这个初始化函数，因为从上次释放和析构函数之后，它已经处于所需的状态中了。

Linux slab 分配器使用了这种思想和其他一些思想来构建一个在空间和时间上都具有高效性的内存分配器。

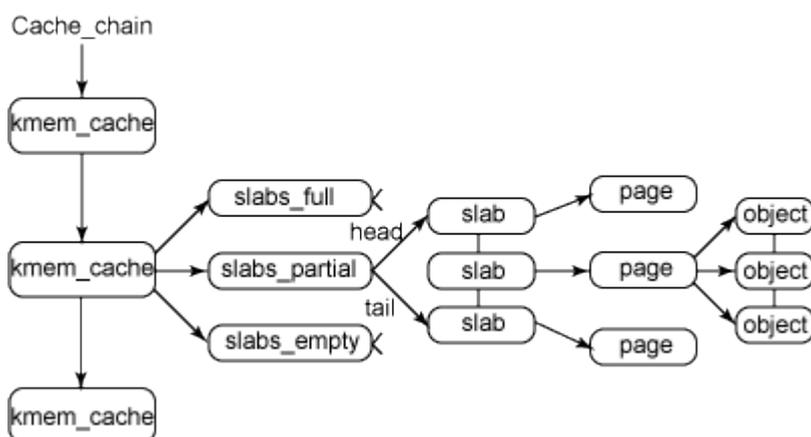
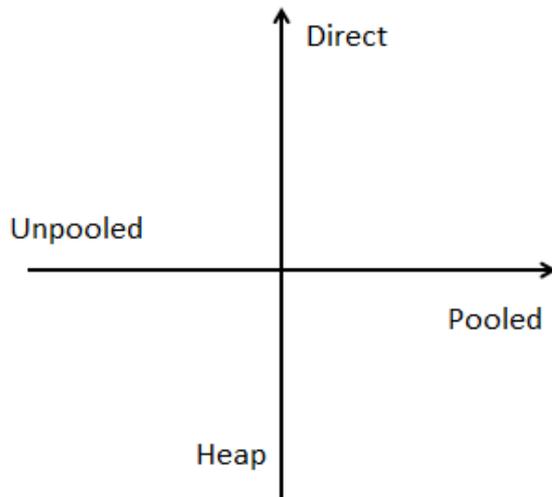


图 1 给出了 slab 结构的高层组织结构。在最高层是 cache\_chain，这是一个 slab 缓存的链接列表。这对于 best-fit 算法非常有用，可以用来查找最适合所需要的分配大小的缓存（遍历列表）。cache\_chain 的每个元素都是一个 kmem\_cache 结构的引用（称为一个 cache）。它定义了一个要管理的给定大小的对象池。

二、ByteBuf 整体结构的分析

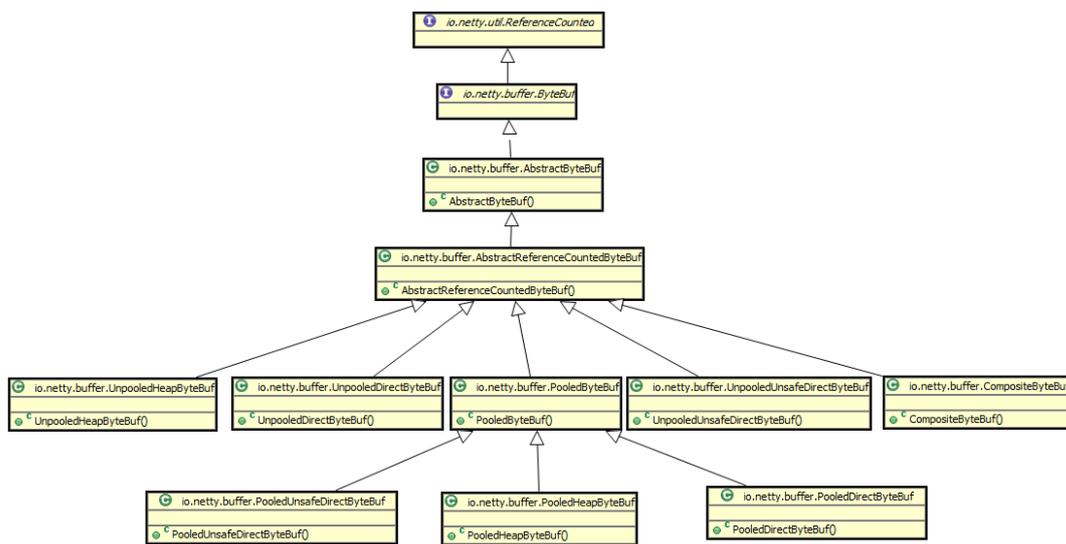
ByteBuf 提供了多种方式的 buffer 实现，按照内存分配的方式可以分为 heap 和 direct 两种实现方式。按照 buffer 的重用性可以分为 Unpooled 和 Pooled 两种实现方式。



从上面两种维度的组合，提供了多种实现方式的 buffer,具体如下：

重用性 \ 内存	非对象池	对象池
Heap 内存	UnpooledHeapByteBuf	PooledHeapByteBuf
Direct 内存	UnpooledDirectByteBuf	PooledDirectByteBuf
	UnpooledUnsafeDirectByteBuf	PooledUnsafeDirectByteBuf

### 1. ByteBuf 整体结构

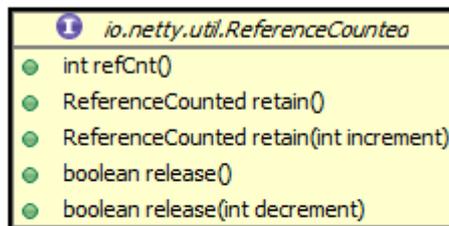


## 三、ByteBuf 抽象层分析

### 1. ReferenceCounted 接口

在整个 UML 中最上层的接口是 ReferenceCounted，这个接口从名称上我们就知道他是用来做引用计数的。当一个对象被使用的时候我们就+1，当我们一个对象不用的时候就-1。就是一个对象的引用计数器，当对象引用计数为 0 的时候就表示当前对象不再有用，可以分配

给其他线程。这也是对象池计数的一个标准实现方式。



接口中的 `retain()`和 `retain(int increment)`方法使用标记当前对象被使用几次的。`release()`和 `release(int decrement)`方法是用来释放引用计数的。在对象从对象池中取出会调用 `retain()`方法打上该对象已经使用，当对象不在使用的时候会调用 `release()`方法释放对象，以供其他线程使用。`refCnt()`方法是用来获取该对象是否在使用的标志。

## 2. ByteBuf 抽象类

`ByteBuf` 是整个 NETTY 的 `buffer` 的总接口，其中主要定义了缓冲区读写、容量控制，数据的刷新与清除等功能。完全实现了 JDK 自带的 `ByteBuffer` 全部功能，并根据 NETTY 自身的系统架构的需要，扩充了对 `InputStream` 和 `ScatteringByteChannel` 直接读写等。

我们可以将 `ByteBuf` 接口定义的功能进行以下的归类：

- ◆ 提供了对 8 种基本类型的读写功能

ByteBuf

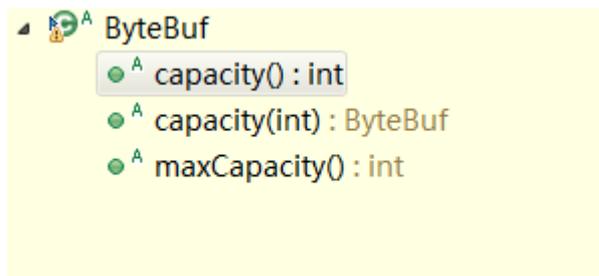
- readerIndex() : int
- readerIndex(int) : ByteBuf
- readableBytes() : int
- readBoolean() : boolean
- readByte() : byte
- readUnsignedByte() : short
- readShort() : short
- readUnsignedShort() : int
- readMedium() : int
- readUnsignedMedium() : int
- readInt() : int
- readUnsignedInt() : long
- readLong() : long
- readChar() : char
- readFloat() : float
- readDouble() : double
- readBytes(int) : ByteBuf
- readSlice(int) : ByteBuf
- readBytes(ByteBuf) : ByteBuf
- readBytes(ByteBuf, int) : ByteBuf
- readBytes(ByteBuf, int, int) : ByteBuf
- readBytes(byte[]) : ByteBuf
- readBytes(byte[], int, int) : ByteBuf

```
ByteBuf  
  ● ^ writerIndex() : int  
  ● ^ writerIndex(int) : ByteBuf  
  ● ^ writeBoolean(boolean) : ByteBuf  
  ● ^ writeByte(int) : ByteBuf  
  ● ^ writeShort(int) : ByteBuf  
  ● ^ writeMedium(int) : ByteBuf  
  ● ^ writeInt(int) : ByteBuf  
  ● ^ writeLong(long) : ByteBuf  
  ● ^ writeChar(int) : ByteBuf  
  ● ^ writeFloat(float) : ByteBuf  
  ● ^ writeDouble(double) : ByteBuf  
  ● ^ writeBytes(ByteBuf) : ByteBuf  
  ● ^ writeBytes(ByteBuf, int) : ByteBuf  
  ● ^ writeBytes(ByteBuf, int, int) : ByteBuf  
  ● ^ writeBytes(byte[]) : ByteBuf  
  ● ^ writeBytes(byte[], int, int) : ByteBuf  
  ● ^ writeBytes(ByteBuffer) : ByteBuf  
  ● ^ writeBytes(InputStream, int) : int  
  ● ^ writeBytes(ScatteringByteChannel, int) : int  
  ● ^ writeZero(int) : ByteBuf
```

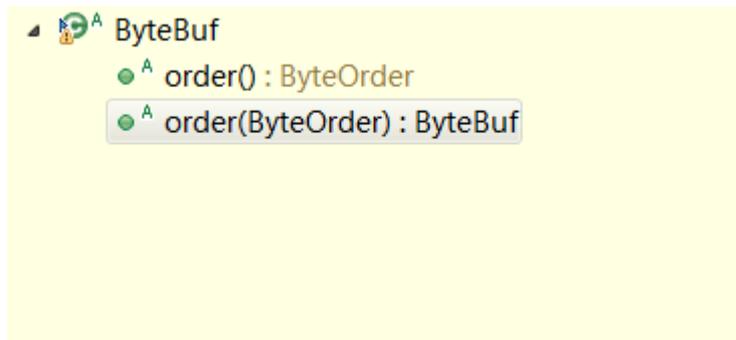
◆ 定义了对 buffer 的当前读写状态的判断, 主要用来判断当前的 buffer 是否可以读写的功能。

```
ByteBuf  
  ● ^ readableBytes() : int  
  ● ^ writableBytes() : int  
  ● ^ maxWritableBytes() : int  
  ● ^ isReadable() : boolean  
  ● ^ isReadable(int) : boolean  
  ● ^ isWritable() : boolean  
  ● ^ isWritable(int) : boolean  
  ● ^ ensureWritable(int) : ByteBuf  
  ● ^ ensureWritable(int, boolean) : int
```

◆ 对整个 buffer 容量的设置和获取



◆ 对 buffer 的存储数据是采用什么编码方式的设定



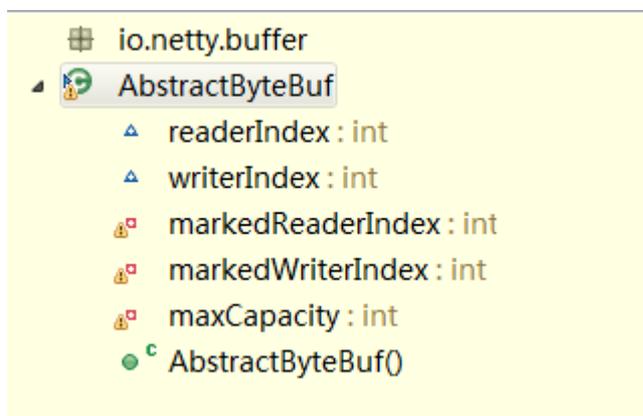
◆ 定义了对缓冲区是 heap 的还是 direct,并定义获取内存地址的方法:

```
public abstract boolean isDirect();
public abstract boolean hasMemoryAddress();
public abstract long memoryAddress();
```

### 3. AbstractByteBuffer 类

AbstractByteBuffer 作为 ByteBuffer 的直接的抽象类，主要完成了 ByteBuffer 接口定义的对 buffer 的当前读写状态的判断，主要用来判断当前的 buffer 是否可以读写的功能。

在 AbstractByteBuffer 中设置了 readerIndex（当前读的位置），writerIndex（当前写入的位置），markedReaderIndex(备份的读的位置)，markedWriterIndex（备份写入的位置），maxCapacity（系统最大容量）等五个字段，并通过五个字段的位置来判断当前 buffer 可读、可写入的状态。



下面给出一些方法的源码;

- 1 对 readerIndex 和 writerIndex 的获取和设定

```
@Override
public int readerIndex() { //获取当前读取的位置
    return readerIndex;
}

@Override
public ByteBuf readerIndex(int readerIndex) {
    //当读取的位置小于0或者读取的位置已经大于写入的位置了, 直接抛出异常
    if (readerIndex < 0 || readerIndex > writerIndex) {
        throw new IndexOutOfBoundsException(String.format(
            "readerIndex: %d (expected: 0 <= readerIndex <= writerIndex(%d))", readerIndex, writerIndex));
    }
    this.readerIndex = readerIndex;
    return this;
}

@Override
public int writerIndex() { //获取当前写入的位置
    return writerIndex;
}

@Override
public ByteBuf writerIndex(int writerIndex) {
    //当写入的位置小于读取的位置, 或者写入的索引位置大于当前的容量, 抛出异常
    if (writerIndex < readerIndex || writerIndex > capacity()) {
        throw new IndexOutOfBoundsException(String.format(
            "writerIndex: %d (expected: readerIndex(%d) <= writerIndex <= capacity(%d))",
            writerIndex, readerIndex, capacity()));
    }
    this.writerIndex = writerIndex;
    return this;
}
```

## 2 对当前 buffer 的可读、可写状态的判断

```
public boolean isReadable() {
    return writerIndex > readerIndex; //当写入的位置大于读取的位置 表示可以读取
}
public boolean isReadable(int numBytes) {
    return writerIndex - readerIndex >= numBytes; //当写入的位置 > 当前读取的位置+这次需要读取的字节数 返回可以读取
}
public boolean isWritable() {
    return capacity() > writerIndex; //当buffer的容量 > 当前写入的位置 可以写入
}
public boolean isWritable(int numBytes) {
    return capacity() - writerIndex >= numBytes; //当buffer的容量 > 当前写入的位置 + 需要写入的字节数, 表示可以写入
}
public int readableBytes() {
    return writerIndex - readerIndex;
}
public int writableBytes() {
    return capacity() - writerIndex; //获取可以写入的字节数
}
public int maxWritableBytes() {
    return maxCapacity() - writerIndex; //获取最大写入的字节数
}
```

## 3 确保 buffer 写入的支持方法, 确保 buffer 在容量不够的时候也能自动扩容以达到对写入的支持。

```
@Override
public ByteBuf ensureWritable(int minWritableBytes) {
    if (minWritableBytes < 0) { // 当需要写入的字节小于0直接抛出异常
        throw new IllegalArgumentException(String.format(
            "minWritableBytes: %d (expected: >= 0)", minWritableBytes));
    }
    // 当需要写入的字节数小于buffer容量，当然可以直接写入，
    if (minWritableBytes <= writableBytes()) {
        return this;
    }
    // 需要写入的字节数大于 最大容量 - 已经写入的字节数，说明超出预期了直接抛出错误
    if (minWritableBytes > maxCapacity - writerIndex) {
        throw new IndexOutOfBoundsException(String.format(
            "writerIndex(%d) + minWritableBytes(%d) exceeds maxCapacity(%d): %s",
            writerIndex, minWritableBytes, maxCapacity, this));
    }
    // 当前容量不够，需要计算出buffer需要扩容的大小
    int newCapacity = calculateNewCapacity(writerIndex + minWritableBytes);
    // 将已有的buffer进行扩容
    capacity(newCapacity);
    return this;
}
```

Buffer 的扩容算法，这个对于整个 buffer 的动态扩充的算法

```
/**
 * 功能：系统扩容容量计算的方法：当要扩容的容量等于默认阈值4M 直接返回 当要扩容的容量大于默认阈值4M的时候，防止倍数扩容，以免申请过大的扩容
 * 当新的容量小于默认阈值4M的时候，以64进行倍数扩容，最大可以大4M
 * @param minNewCapacity
 * @return
 */
private int calculateNewCapacity(int minNewCapacity) {
    final int maxCapacity = this.maxCapacity;
    final int threshold = 1048576 * 4; // 4M大小
    if (minNewCapacity == threshold) { // 当要扩容的容量等于4M 直接返回
        return threshold;
    }
    if (minNewCapacity > threshold) { // 当要扩容的容量大于默认容量4M的时候，防止倍数扩容，以免申请过大的扩容
        int newCapacity = minNewCapacity / threshold * threshold;
        if (newCapacity > maxCapacity - threshold) {
            newCapacity = maxCapacity;
        } else {
            newCapacity += threshold;
        }
        return newCapacity;
    }
    int newCapacity = 64;
    while (newCapacity < minNewCapacity) { // 当新的容量小于默认阈值4M的时候，以64进行倍数扩容，最大可以大4M
        newCapacity <<= 1;
    }
    return Math.min(newCapacity, maxCapacity);
}
```

4 对 8 种基本数据类型的读、写操作给了初步实现。为什么说是初步实现呢，是因为他提供的是初步的包装，最终读取字节是由具体的各种 buffer 子类实现的。

对 8 种类型的读取操作：

```
// 读取boolean类型
public boolean getBoolean(int index) {
    return getByte(index) != 0;
}

// 读取boolean类型
public short getShort(int index) {
    checkIndex(index, 2);
    return _getShort(index);
}

// 读取字符类型
public char getChar(int index) {
    return (char) getShort(index);
}

// 读取int类型
public int getInt(int index) {
    checkIndex(index, 4);
    return _getInt(index);
}

// 读取long类型
public long getLong(int index) {
    checkIndex(index, 8); //判断是否可以读取8个字节
    return _getLong(index);
}

// 获取4个字节，然后将四个字节转换成float
public float getFloat(int index) {
    return Float.intBitsToFloat(getInt(index)); //参照我前面讲的准备知识
}

// 获取8个字节，然后将四个字节转换成float
public double getDouble(int index) {
    return Double.longBitsToDouble(getLong(index)); //参照我前面讲的准备知识
}

// 从原来数组中获取到具体字节，到目标字节数组中
@Override
public ByteBuf getBytes(int index, byte[] dst) {
    getBytes(index, dst, 0, dst.length);
    return this;
}
```

对 8 种类型的数据写入操作：

```
/**
 * 功能: 判断当前的buffer是否可以访问的, 主要通过对象引用的计数的, 当引用计数为0的时候表示buffer已经不再使用。
 * 背景: 在pool buffer中, 每次从池中取出对象, 都将计数设置为1 当对象不在使用的释放回线程池的时候设置为0
 */
protected final void ensureAccessible() {
    if (refCnt() == 0) {
        throw new IllegalReferenceCountException(0);
    }
}

public ByteBuf writeByte(int value) {
    ensureAccessible();// 判断对象是否可用
    ensureWritable(1);// 判断是否可以写入个字节
    _setByte(writerIndex++, value);// 写入int值, 写入索引+1
    return this;
}

public ByteBuf writeBoolean(boolean value) { // 写入boolean类型
    writeByte(value ? 1 : 0);
    return this;
}

public ByteBuf writeShort(int value) {
    ensureAccessible();// 判断对象是否可用
    ensureWritable(2);// 判断是否可以写入2个字节
    _setShort(writerIndex, value);// 写入short值
    writerIndex += 2;// 写入索引+2
    return this;
}

public ByteBuf writeInt(int value) {
    ensureAccessible();// 判断对象是否可用
    ensureWritable(4);// 判断是否可以写入4个字节
    _setInt(writerIndex, value);// 写入int 值
    writerIndex += 4;// 写入索引+4
    return this;
}

public ByteBuf writeLong(long value) {
    ensureAccessible();// 判断对象是否可用
    ensureWritable(8);// 判断是否可以写入8个字节
    _setLong(writerIndex, value);
    writerIndex += 8;// 写入索引+8
    return this;
}

public ByteBuf writeChar(int value) {
    writeShort(value);// 转换成short写入
    return this;
}

public ByteBuf writeFloat(float value) {
    writeInt(Float.floatToRawIntBits(value));// 将float转换成int表示, 然后存储
    return this;
}

public ByteBuf writeDouble(double value) {
    writeLong(Double.doubleToRawLongBits(value));// 将double转换成long表示, 然后存储
    return this;
}
}
```

#### 4. AbstractReferenceCountedByteBuf 类

AbstractReferenceCountedByteBuf 是个抽象类, 是 ReferenceCounted 接口的直接实现。主要完成了对 buffer 对象引用计数的统计。在 AbstractReferenceCountedByteBuf 中定义了 refCnt 对象应用计算字段, 该字段是 refCnt 保证了多线程的可见性。同时定义了 AtomicIntegerFieldUpdater 类型的变量 refCntUpdater, 通过 CAS 原理, 可以确保多线程修改的正确性。

```
//对象引用进行计数
private volatile int refCnt = 1;
//定义了对refCnt的修改器，主要是调用java的unsafe类来实现
private static final AtomicIntegerFieldUpdater<AbstractReferenceCountedByteBuf> refCntUpdater;
static {
    AtomicIntegerFieldUpdater<AbstractReferenceCountedByteBuf> updater =
        PlatformDependent.newAtomicIntegerFieldUpdater(AbstractReferenceCountedByteBuf.class, "refCnt");
    if (updater == null) {
        updater = AtomicIntegerFieldUpdater.newUpdater(AbstractReferenceCountedByteBuf.class, "refCnt");
    }
    refCntUpdater = updater;
}
```

实现了类对对象使用的引用计数，引用计数为默认值 1，进行引用计数+1。当对象释放的时候将引用计数-1。当引用计数为默认值 1 的时候，就说明没有对象使用进行对象释放。

Retain 方法:

```
/**
 * 功能：当创建一个对象的时候，将引用计数+1，默认值是1
 */
@Override
public ByteBuf retain() {
    for (;;) {
        int refCnt = this.refCnt;
        if (refCnt == 0) { //当引用的值为0的时候，说明出现了错误
            throw new IllegalReferenceCountException(0, 1);
        }
        if (refCnt == Integer.MAX_VALUE) { //当引用的值超出限制的时候，说明出现了错误
            throw new IllegalReferenceCountException(Integer.MAX_VALUE, 1);
        }
        if (refCntUpdater.compareAndSet(this, refCnt, refCnt + 1)) { //对引用计数+1
            break;
        }
    }
    return this;
}
```

Release 方法:

```
//当释放减一 当这个对象的引用计数为0的时候表示可以释放
@Override
public final boolean release() {
    for (;;) {
        int refCnt = this.refCnt;
        if (refCnt == 0) { //当引用的值为0的时候，说明出现了错误
            throw new IllegalReferenceCountException(0, -1);
        }

        if (refCntUpdater.compareAndSet(this, refCnt, refCnt - 1)) { //对引用计数-1
            if (refCnt == 1) {
                deallocate(); //释放内存，会调用其子类进行设置
                return true;
            }
            return false;
        }
    }
}
```

#### 四、Buf 的 Unpooled 的实现

Buf 的 Unpooled 直接实现，主要有 heap 和 direct 两种实现方式。Heap 是用字节数组的方式实现，主要实现类 UnpooledHeapByteBuf。Direct 使用内存的方式实现，主要实现类有 UnpooledDirectByteBuf 和 UnpooledUnsafeDirectByteBuf。

## 1. UnpooledHeapByteBuffer

UnpooledHeapByteBuffer 定义字节数组 array 来存储的数据信息的，并定义了 Buf 向 ByteBuffer 转换的临时变量 tmpNioBuf。并完成了数据读、写的最终实现。

```
/**
 * Big endian Java heap buffer implementation.
 */
public class UnpooledHeapByteBuffer extends AbstractReferenceCountedByteBuffer {

    private final ByteBufferAllocator alloc;
    private byte[] array;
    private ByteBuffer tmpNioBuf;

    /**
```

数组扩容的方法 capacity

```
@Override
public ByteBuffer capacity(int newCapacity) {
    ensureAccessible();
    if (newCapacity < 0 || newCapacity > maxCapacity()) {
        throw new IllegalArgumentException("newCapacity: " + newCapacity);
    }
    int oldCapacity = array.length;
    if (newCapacity > oldCapacity) { //当新的容量大于老的容量时候
        byte[] newArray = new byte[newCapacity]; //构造新的数组将，老的数组复制进去
        System.arraycopy(array, 0, newArray, 0, array.length);
        setArray(newArray); //设置新的数组
    }
    else if (newCapacity < oldCapacity) { //当新的容量小于已有的容量，就是减小容量
        byte[] newArray = new byte[newCapacity];
        int readerIndex = readerIndex();
        if (readerIndex < newCapacity) {
            int writerIndex = writerIndex();
            if (writerIndex > newCapacity) {
                writerIndex(writerIndex = newCapacity);
            }
            //复制部分可读的数据进入新的数组
            System.arraycopy(array, readerIndex, newArray, readerIndex, writerIndex - readerIndex);
        } else {
            setIndex(newCapacity, newCapacity); //设置数组读写的索引
        }
        setArray(newArray);
    }
    return this;
}
```

数据具体读取方法的实现:

```
@Override
protected short _getShort(int index) {
    return (short) (array[index] << 8 | array[index + 1] & 0xFF);
}
//读取字节
protected byte _getBytes(int index) {
    return array[index];
}
//将4个字节转换成int
protected int _getInt(int index) {
    return (array[index] & 0xff) << 24 |
        (array[index + 1] & 0xff) << 16 |
        (array[index + 2] & 0xff) << 8 |
        array[index + 3] & 0xff;
}
//将8个字节转换成long
@Override
protected long _getLong(int index) {
    return ((long) array[index] & 0xff) << 56 |
        ((long) array[index + 1] & 0xff) << 48 |
        ((long) array[index + 2] & 0xff) << 40 |
        ((long) array[index + 3] & 0xff) << 32 |
        ((long) array[index + 4] & 0xff) << 24 |
        ((long) array[index + 5] & 0xff) << 16 |
        ((long) array[index + 6] & 0xff) << 8 |
        (long) array[index + 7] & 0xff;
}
```

数据具体写入方法的时:

```
protected void _setByte(int index, int value) {
    array[index] = (byte) value;
}
//将short类型转换成两个字节
protected void _setShort(int index, int value) {
    array[index] = (byte) (value >>> 8);
    array[index + 1] = (byte) value;
}
//将int转换成四个字节
protected void _setInt(int index, int value) {
    array[index] = (byte) (value >>> 24);
    array[index + 1] = (byte) (value >>> 16);
    array[index + 2] = (byte) (value >>> 8);
    array[index + 3] = (byte) value;
}
//将long转换成8个字
protected void _setLong(int index, long value) {
    array[index] = (byte) (value >>> 56);
    array[index + 1] = (byte) (value >>> 48);
    array[index + 2] = (byte) (value >>> 40);
    array[index + 3] = (byte) (value >>> 32);
    array[index + 4] = (byte) (value >>> 24);
    array[index + 5] = (byte) (value >>> 16);
    array[index + 6] = (byte) (value >>> 8);
    array[index + 7] = (byte) value;
}
```

ByteBuffer 转换功能:

```
/**
 * 功能: 将byte字节数组转换成ByteBuffer的方法
 * @return
 */
private ByteBuffer internalNioBuffer() {
    ByteBuffer tmpNioBuf = this.tmpNioBuf;
    if (tmpNioBuf == null) {
        this.tmpNioBuf = tmpNioBuf = ByteBuffer.wrap(array);
    }
    return tmpNioBuf;
}
```

## 2. UnpooledDirectByteBuf

UnpooledDirectByteBuf 类是将 JDK 自带的 ByteBuffer 的 direct buffer 做了一层封装。实现的功能和 UnpooledHeapByteBuf 基本一致，只是在具体的实现方式上有所差别。

```
/**
 * Creates a new direct buffer.
 *
 * @param initialCapacity the initial capacity of the underlying direct buffer
 * @param maxCapacity the maximum capacity of the underlying direct buffer
 */
protected UnpooledDirectByteBuf(ByteBufAllocator alloc, int initialCapacity, int maxCapacity) {
    super(maxCapacity);
    if (alloc == null) {
        throw new NullPointerException("alloc");
    }
    if (initialCapacity < 0) {
        throw new IllegalArgumentException("initialCapacity: " + initialCapacity);
    }
    if (maxCapacity < 0) {
        throw new IllegalArgumentException("maxCapacity: " + maxCapacity);
    }
    if (initialCapacity > maxCapacity) {
        throw new IllegalArgumentException(String.format(
            "initialCapacity(%d) > maxCapacity(%d)", initialCapacity, maxCapacity));
    }
    this.alloc = alloc;
    setByteBuffer(ByteBuffer.allocateDirect(initialCapacity)); //通过ByteBuffer.allocateDirect来获取缓冲区
}
```

数据的读取是通过 ByteBuffer 的读取方法来间接实现的:

```
//byte的读取
protected byte _getBytes(int index) {
    return buffer.get(index);
}
//short的读取
protected short _getShort(int index) {
    return buffer.getShort(index);
}
//int的读取
protected int _getInt(int index) {
    return buffer.getInt(index);
}
//long的读取
protected long _getLong(int index) {
    return buffer.getLong(index);
}
```

数据的写入是通过 ByteBuffer 的写入方法来间接实现的:

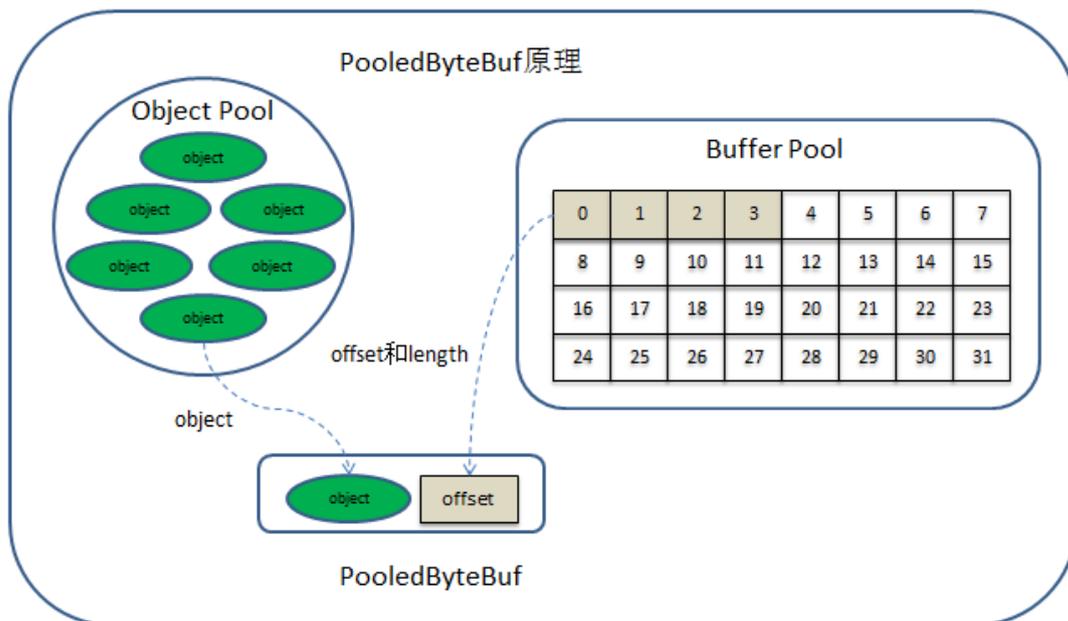
```
//写入字节
protected void _setByte(int index, int value) {
    buffer.put(index, (byte) value);
}
//写入short
protected void _setShort(int index, int value) {
    buffer.putShort(index, (short) value);
}
//写入int
protected void _setInt(int index, int value) {
    buffer.putInt(index, value);
}
//写入long
protected void _setLong(int index, long value) {
    buffer.putLong(index, value);
}
```

### 3. UnpooledUnsafeDirectByteBuf

UnpooledUnsafeDirectByteBuf 是通过 unsafe 来操作数据的读写, 功能和 UnpooledDirectByteBuf 是相似的。当系统支持 unsafe 的时候, 默认 direct buffer 就是 UnpooledUnsafeDirectByteBuf 的。

## 五、Pooled buffer 的实现

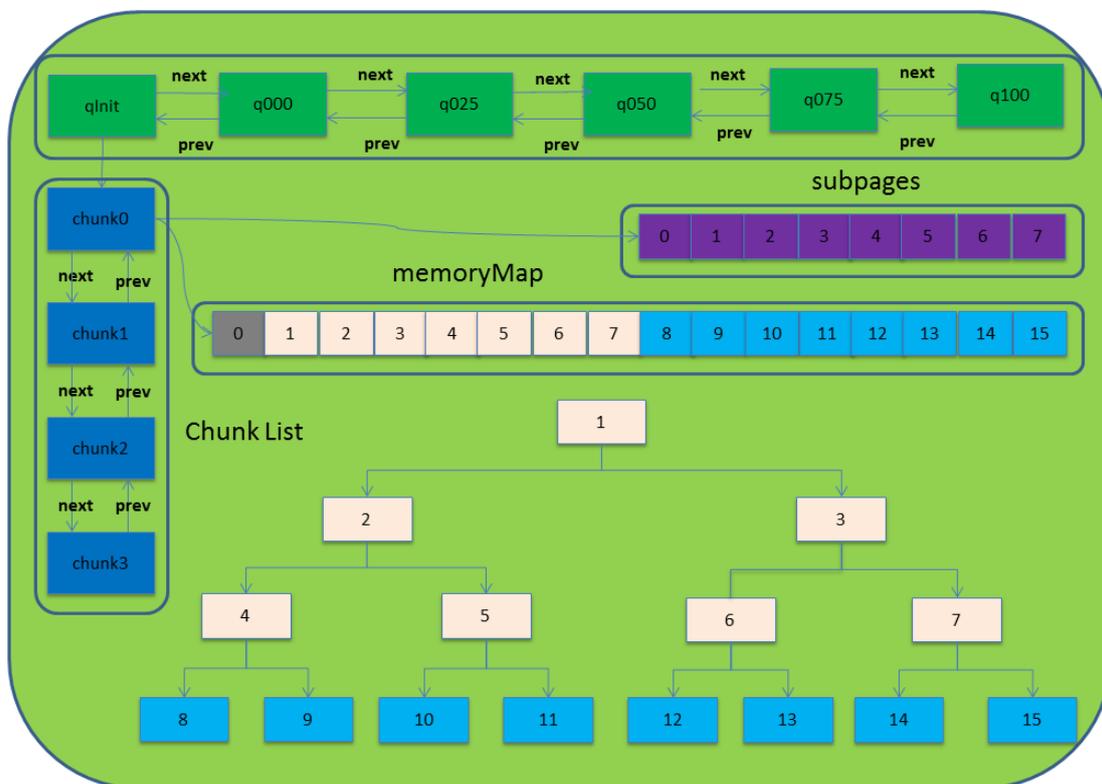
Netty 4.x 开发了 Pooled Buffer, 采用对象池技术实现了一个高性能的 buffer 池, 分配策略则是结合了 buddy allocation 和 slab allocation 的 jemalloc 变种。在网络编程中提供了高效的 buffer 创建方式, 同时减少了 buffer 的创建和垃圾回收的消耗。



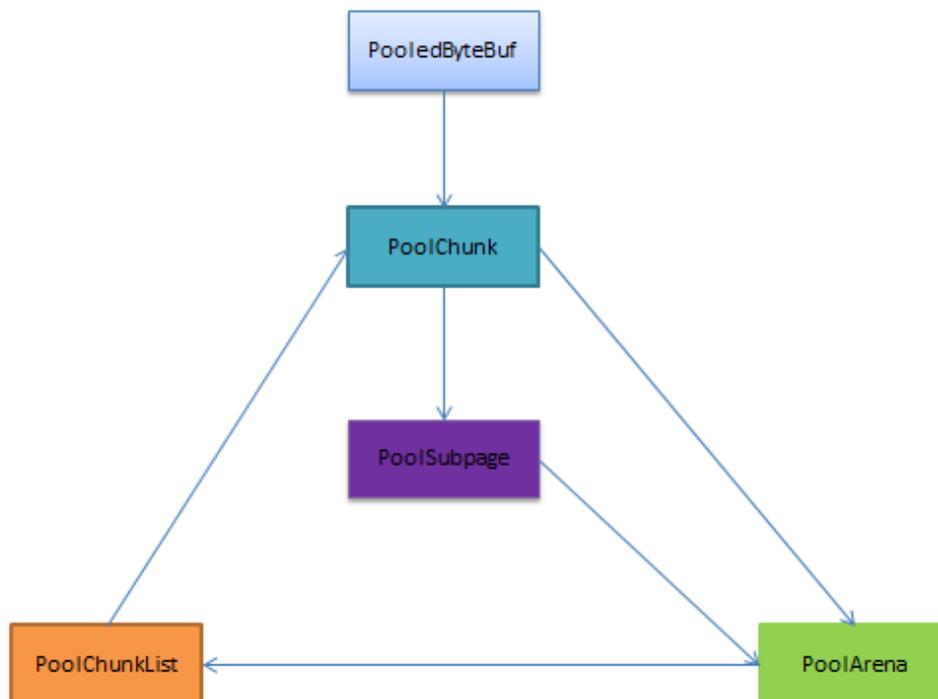
#### 1. Pooled Buffer 整体结构

Pooled Buffer 主要由 PoolArena 来规划一大片的 buffer 缓冲区（字节数组、direct 内存区域）。PoolArena 又由一系列的 PoolChunkList 组成双向指针链表，来将具体的每个 PoolChunk 单元（相当操作系统内存的 page）合并成一个大的 buffer 缓冲区。并将 PoolChunk 划分为大小相同的多个 PoolSubpage，以方便进行具体 buffer 分配。

整体的逻辑结构图如下：



类之间的引用关系如下：

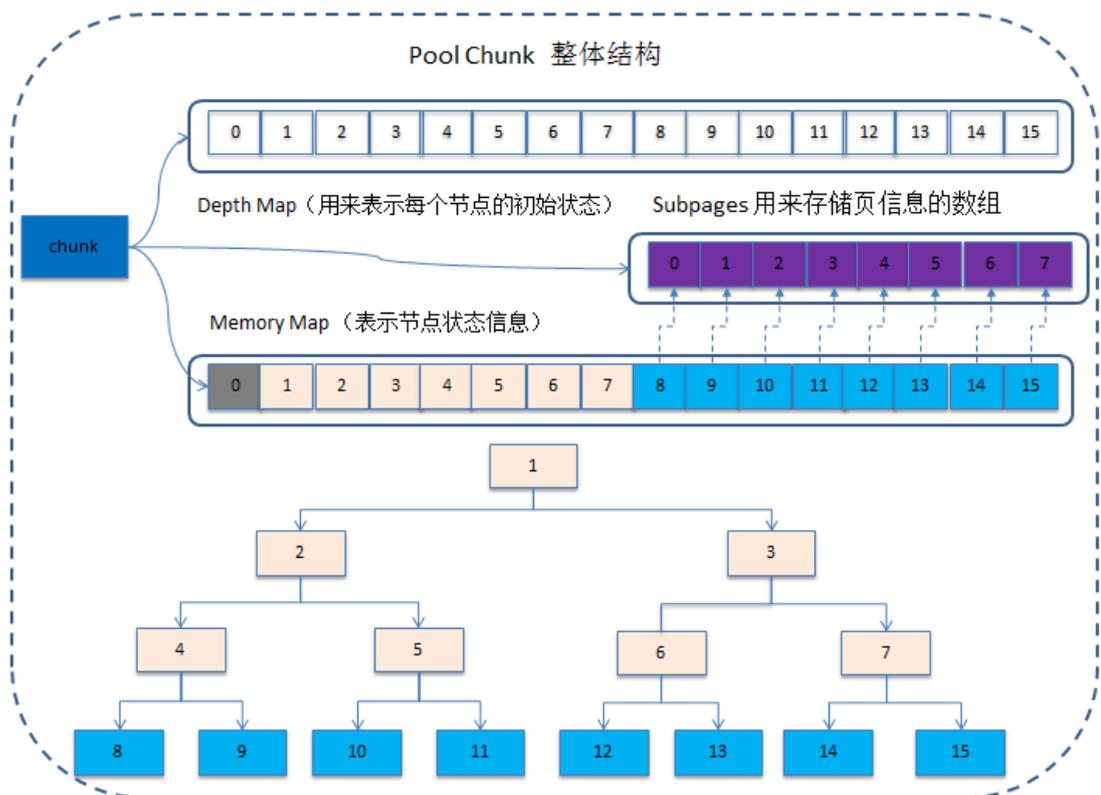


## 2. PoolChunk 的结构

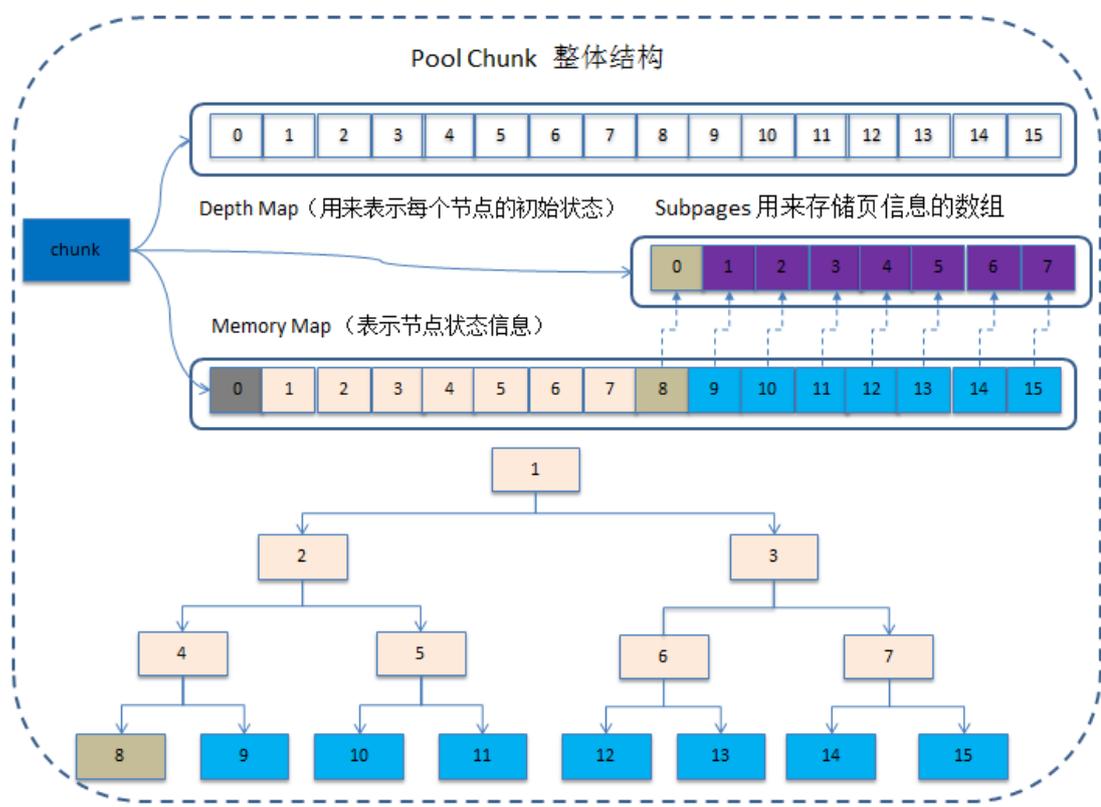
**PoolChunk** 是对象池中进行分配的主要管理单元,他管理了多个内存页(**PoolSubpage**)。每次从**PoolChunk** 申请 **buffer** 的时候会 **subpages** 数组中获取到一个未分配的内存页, 然后进行 **buffer** 的分配。

在 **PoolChunk** 创建的时候,同时构造了 **Memory Map**(表示节点状态信息), **Depth Map** (用来表示每个节点的初始状态),**Subpages** (存储内存页数组)。

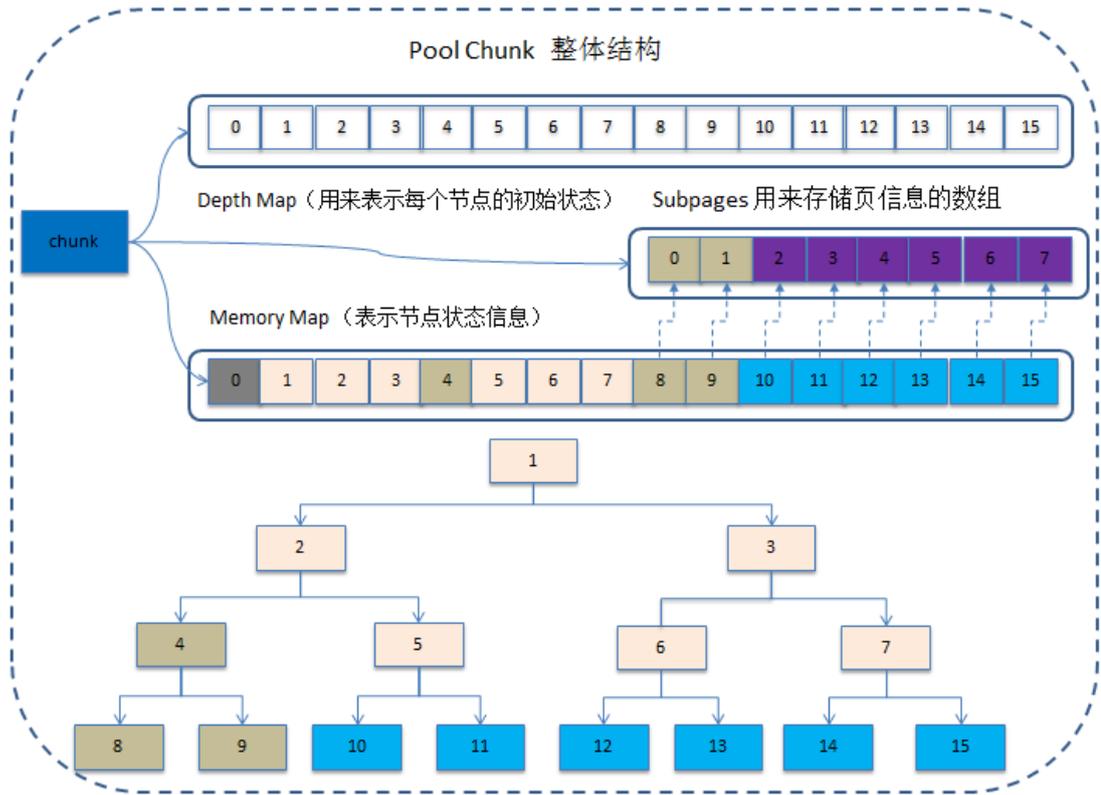
**Subpages** 为  $2N$  字方的数组, **Memory Map** 为  $2(N+1)$  字方的数组, 其实是一个二叉树的数组方式实现。**Depth Map** 为二叉树的初始化状态值。**Memory Map** 只有叶子节点才表示存储信息,非叶子节点只是用来表示叶子节点的状态信息。没次取节点的时候通过深度遍历边节点开始取



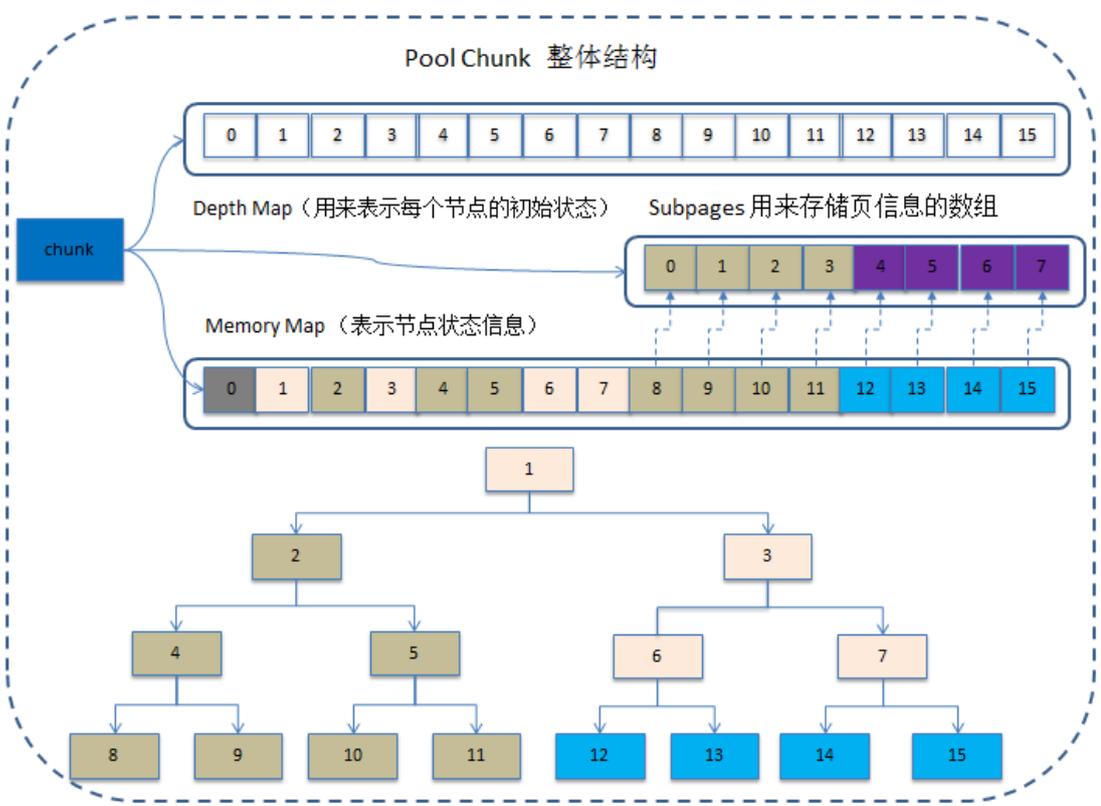
◆ 一次性取一页的逻辑  
第一次取分页后数组状态:



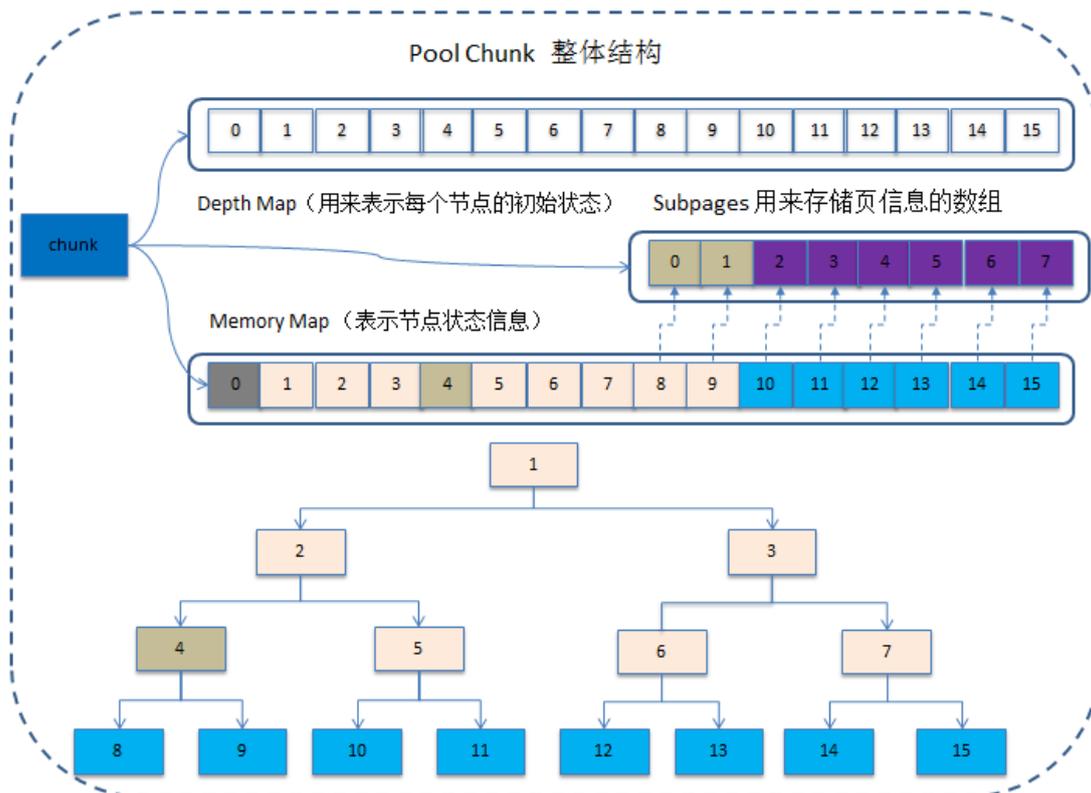
第二次取分页后数组状态:



第四次取分页之后状态

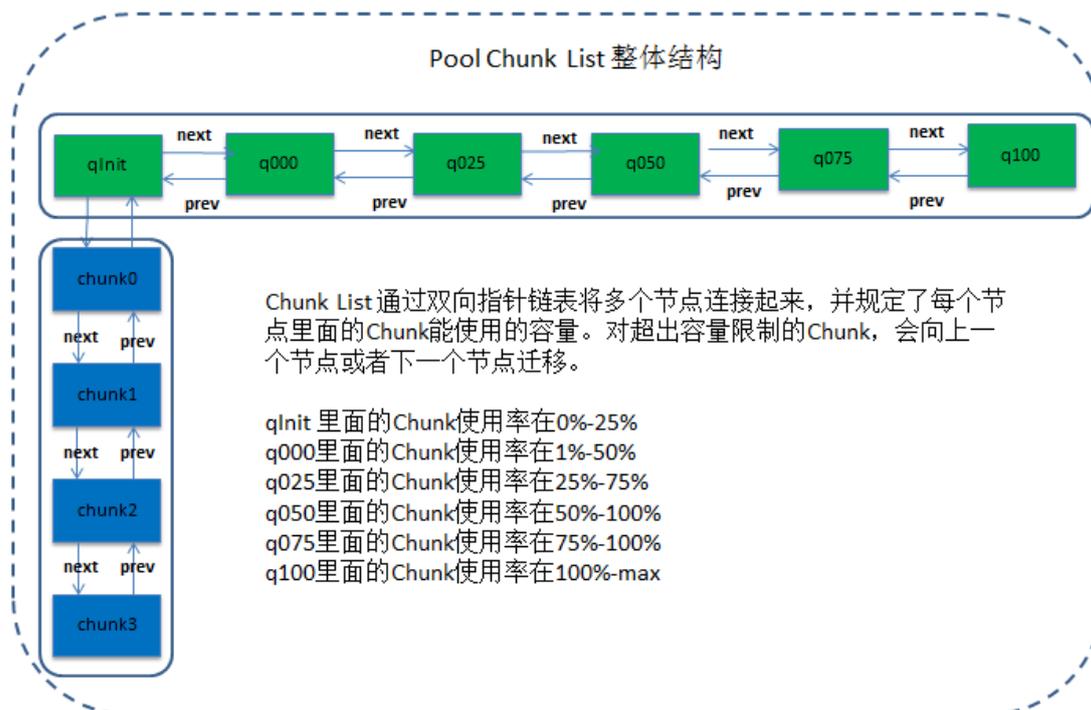


◆ 一次性取两页的逻辑

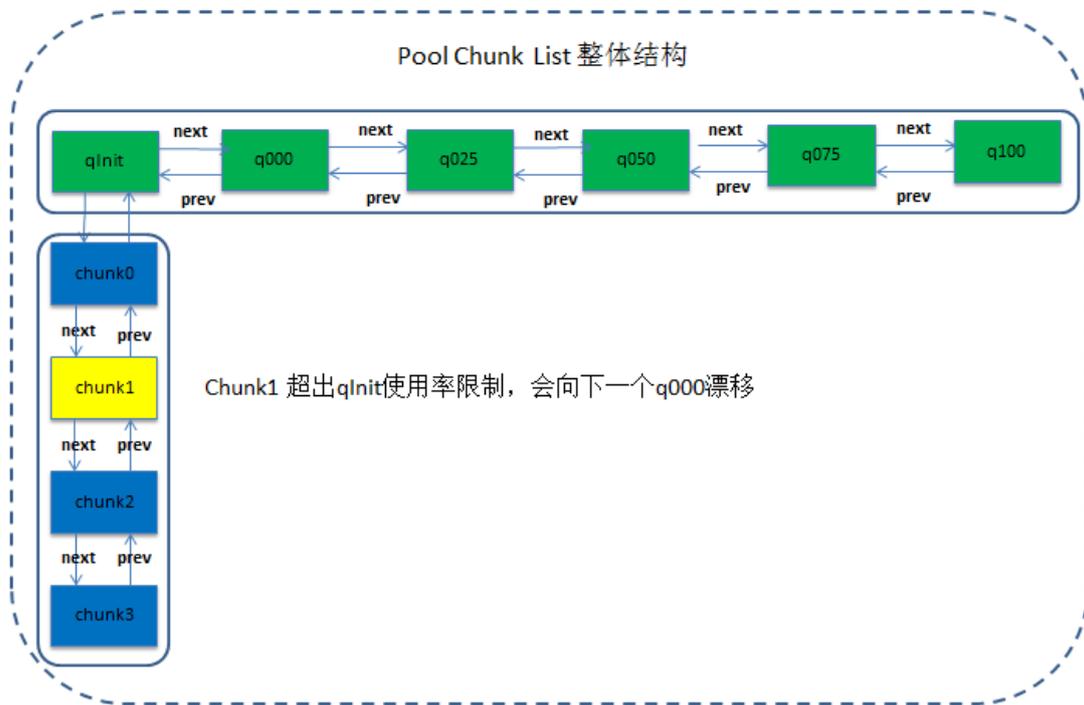


### 3. PoolChunkList 的结构

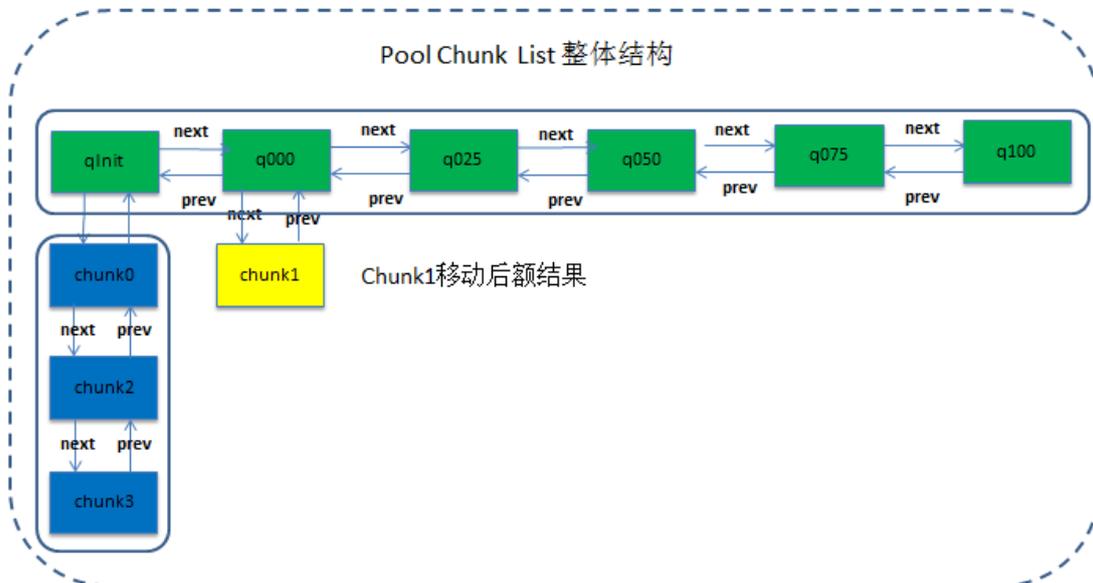
PoolChunkList 通过双向指针链表将多个节点连接起来，并对每个节点里面存储的 Chunk 能使用的使用率做出限制。对超出使用率限制的 Chunk，会向上一个节点或者下一个节点迁移。这样就能起到优先从使用率较低的 chunk 中分配内存页，以减少内存页分配时二叉树遍历的次数。



当超出使用率限制的 Chunk 进行位移

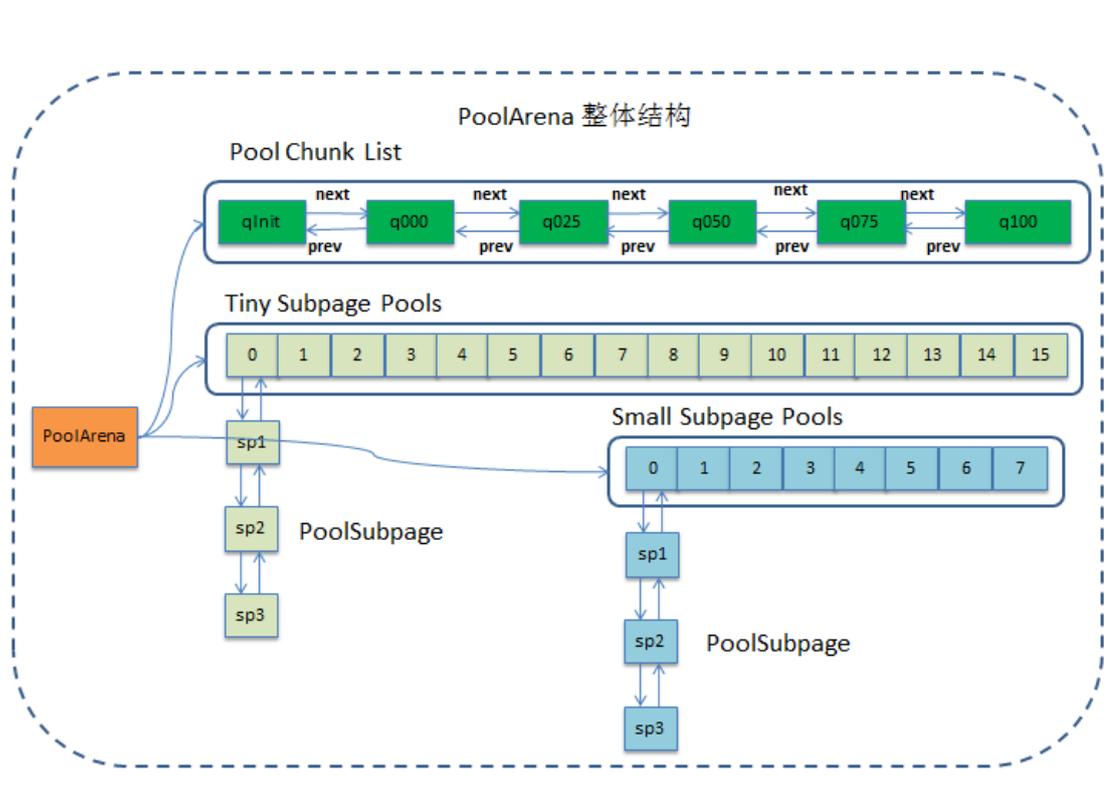


移动后的结果

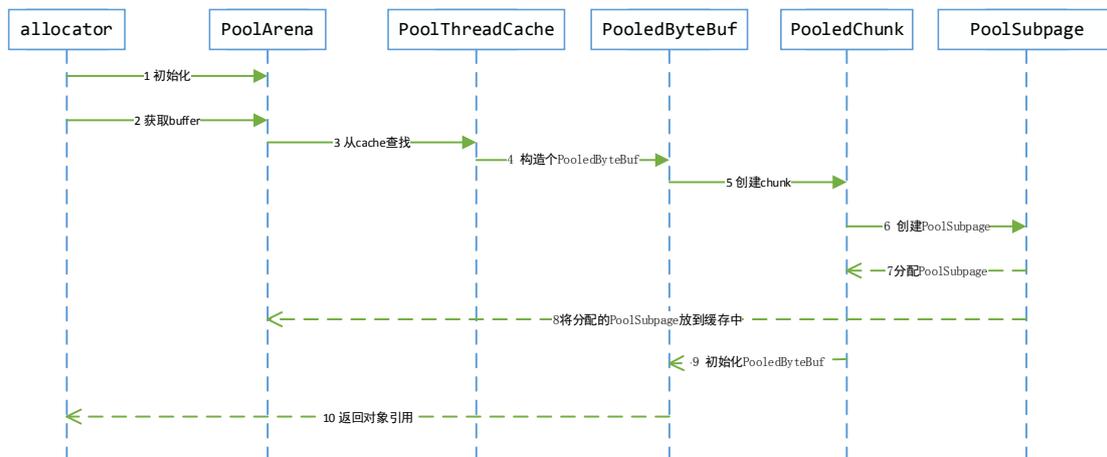


#### 4. PoolArena 的结构

PoolArena 主要是包含了 Pool Chunk List、Tiny Subpage Pools 和 Small Subpage Pools 这三个组成部分。Pool Chunk List 维护了整个 Chunk 的链表组合结构。Tiny Subpage Pools 和 Small Subpage Pools 存储了从 Chunk 中分配出来的 PoolSubpage。



## 5. 从对象池中获取 buffer



### 1 初始化 PoolArena

主要是构造 **PoolChunkList** 的双向指针链表，够一个整体的 **Chunk List**。同时初始化了 **tiny page pool** 和 **small page pool**，这两个 **pool** 是 **Pooled Buffer** 主要缓存实现。**Tiny** 是指小于 512 个字节的 **buffer**，**small** 是指大于 512 并且小于 **page size** 的 **buffer**。

```
protected PoolArena(PooledByteBufAllocator parent, int pageSize, int maxOrder, int pageShifts, int chunkSize) {
    this.parent = parent; // 设置了PooledByteBufAllocator
    this.pageSize = pageSize; // 设置了一页的大小
    this.maxOrder = maxOrder; // 设置chunk队列的深度
    this.pageShifts = pageShifts;
    this.chunkSize = chunkSize; // 设置了真个chunk的大小
    subpageOverflowMask = ~(pageSize - 1);
    // 设置了tiny pagepool
    tinySubpagePools = newSubpagePoolArray(numTinySubpagePools);
    for (int i = 0; i < tinySubpagePools.length; i++) {
        tinySubpagePools[i] = newSubpagePoolHead(pageSize);
    }

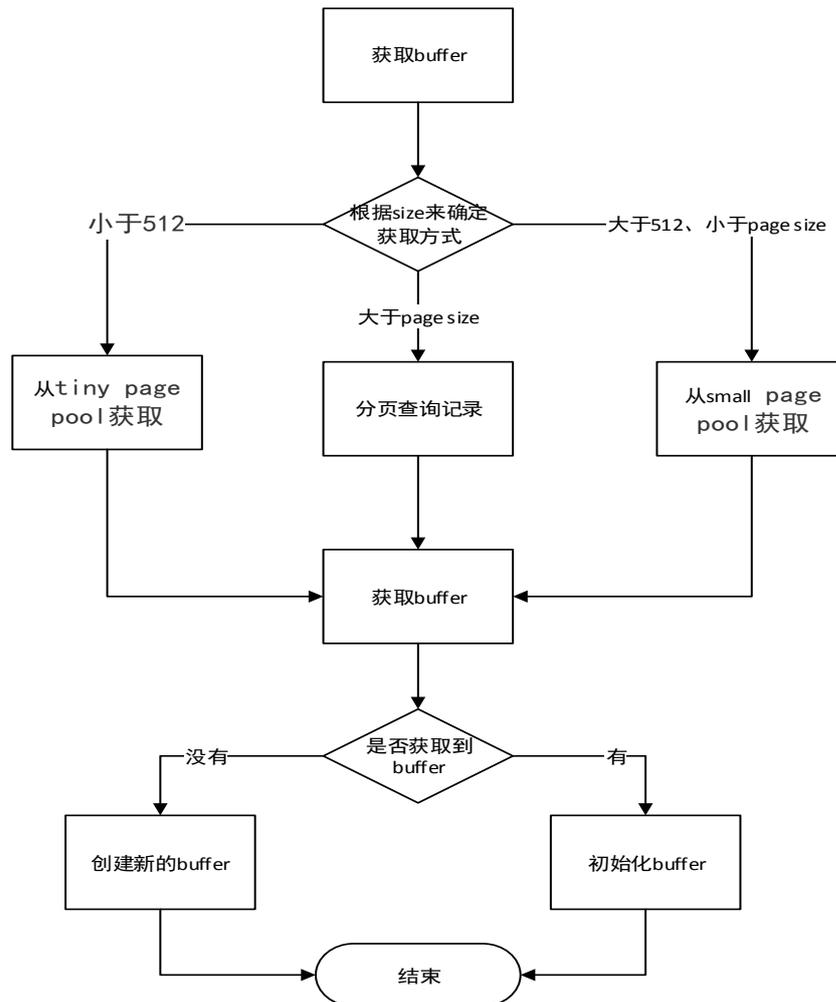
    // 设置了small pagepool
    numSmallSubpagePools = pageShifts - 9;
    smallSubpagePools = newSubpagePoolArray(numSmallSubpagePools);
    for (int i = 0; i < smallSubpagePools.length; i++) {
        smallSubpagePools[i] = newSubpagePoolHead(pageSize);
    }

    q100 = new PoolChunkList<T>(this, null, 100, Integer.MAX_VALUE);
    q075 = new PoolChunkList<T>(this, q100, 75, 100);
    q050 = new PoolChunkList<T>(this, q075, 50, 100);
    q025 = new PoolChunkList<T>(this, q050, 25, 75);
    q000 = new PoolChunkList<T>(this, q025, 1, 50);
    qInit = new PoolChunkList<T>(this, q000, Integer.MIN_VALUE, 25);

    q100.prevList = q075;
    q075.prevList = q050;
    q050.prevList = q025;
    q025.prevList = q000;
    q000.prevList = null;
    qInit.prevList = qInit;
}
}
```

## 2 从 PoolArena 中获取 buffer

先从当前线程的缓存中查找到对应 PoolArena，然后从 PoolArena 中获取 buffer。根据需要申请 buffer 大小来判断获取 buffer 的方式。具体的流程图如下：



具体的代码如下：

```
private void allocate(PoolThreadCache cache, PooledByteBuf<T> buf, final int reqCapacity) {
    final int normCapacity = normalizeCapacity(reqCapacity);
    if (isTinyOrSmall(normCapacity)) { // capacity < pageSize
        int tableIdx;
        PoolSubpage<T>[] table;
        if (isTiny(normCapacity)) { // < 512 从tiny buffer池中获取buffer
            if (cache.allocateTiny(this, buf, reqCapacity, normCapacity)) {
                return;
            }
            tableIdx = tinyIdx(normCapacity);
            table = tinySubpagePools;
        } else { //从small池中获取buffer
            if (cache.allocateSmall(this, buf, reqCapacity, normCapacity)) {
                return;
            }
            tableIdx = smallIdx(normCapacity);
            table = smallSubpagePools;
        }

        synchronized (this) { //线程同步获取buffer
            final PoolSubpage<T> head = table[tableIdx];
            final PoolSubpage<T> s = head.next;
            if (s != head) {
                assert s.doNotDestroy && s.elemSize == normCapacity;
                long handle = s.allocate();
                assert handle >= 0;
                s.chunk.initBufWithSubpage(buf, handle, reqCapacity); //初始化buffer
                return;
            }
        }
    } else if (normCapacity <= chunkSize) { //当请求的大小大于1页的情况，需要从缓存中查找有空闲的chunk
        if (cache.allocateNormal(this, buf, reqCapacity, normCapacity)) {
            return;
        }
    } else {
        allocateHuge(buf, reqCapacity);
        return;
    }
    allocateNormal(buf, reqCapacity, normCapacity); //在上面所有方式都无法获取到buffer的，需要新建一个chunk了
}
```

### 3 构造新的 chunk

当查找从 cache 中查找无法得到 buffer 的时候，创建一个新的 chunk 来扩充缓存的大小。并在新的 chunk 中划分出 PooledByteBuf。

```
private synchronized void allocateNormal(PooledByteBuf<T> buf, int reqCapacity, int normCapacity) {
    //从数组链表中划分buffer，得到直接返回
    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) || q100.allocate(buf, reqCapacity, normCapacity)) {
        return;
    }

    //当现有的链表查询不到对象时，创建一个新的
    PoolChunk<T> c = newChunk(pageSize, maxOrder, pageShifts, chunkSize);
    long handle = c.allocate(normCapacity); //从chunk中划分对象
    assert handle > 0;
    c.initBuf(buf, handle, reqCapacity); //初始化buffer
    qInit.add(c); //赶羊策略，从队列的头部想后赶
}
```

newChunk 方法在 PoolArena 是一个抽象的方法，最终的实现时由他的子类 HeapArena 和 DirectArena 来实现的。最终调用的是 PoolChunk 构造方法。

```
.....
@Override
protected PoolChunk<byte[]> newChunk(int pageSize, int maxOrder, int pageShifts, int chunkSize) {
    return new PoolChunk<byte[]>(this, new byte[chunkSize], pageSize, maxOrder, pageShifts, chunkSize);
}

//用direct buffer来构造了缓冲区
@Override
protected PoolChunk<ByteBuffer> newChunk(int pageSize, int maxOrder, int pageShifts, int chunkSize) {
    return new PoolChunk<ByteBuffer>(
        this, ByteBuffer.allocateDirect(chunkSize), pageSize, maxOrder, pageShifts, chunkSize);
}
```

在 PoolChunk 的构造方法中对整个二叉树的结构进行了初始化。并初始化了

PoolSubpage 的数组对象。也就是在这个构造方法中真正实现了 buffer 的定义。

```
/**
 * 功能: chunk的构造函数
 * @param arena
 * @param memory 要存储的buffer对象
 * @param pageSize 每个page存储的大小
 * @param maxOrder 整个存储二叉树的深度 最终能存储的节点有2的maxOrder子方
 * @param pageShifts
 * @param chunkSize
 */
PoolChunk(PoolArena<T> arena, T memory, int pageSize, int maxOrder, int pageShifts, int chunkSize) {
    unpooled = false;
    this.arena = arena;
    this.memory = memory;
    this.pageSize = pageSize;
    this.pageShifts = pageShifts;
    this.maxOrder = maxOrder;
    this.chunkSize = chunkSize;
    unusable = (byte) (maxOrder + 1);
    log2ChunkSize = log2(chunkSize);
    subpageOverflowMask = ~(pageSize - 1);
    freeBytes = chunkSize;
    maxSubpageAllocs = 1 << maxOrder; //2的maxOrder方
    memoryMap = new byte[maxSubpageAllocs << 1]; //2的(maxOrder+1)方
    depthMap = new byte[memoryMap.length]; //用来存储每个节点的初始值
    int memoryMapIndex = 1;
    //需要对数组附上初始值
    for (int d = 0; d <= maxOrder; ++ d) { // move down the tree one level at a time
        int depth = 1 << d;
        for (int p = 0; p < depth; ++ p) {
            memoryMap[memoryMapIndex] = (byte) d;
            depthMap[memoryMapIndex] = (byte) d;
            memoryMapIndex ++;
        }
    }
    subpages = newSubpageArray(maxSubpageAllocs); //构造分页数组
}
```

## 5 从 chunk 中分配 Subpage 对象

根据需要申请 buffer 的大小判断需要申请的是一个或者多个 Subpage 所表示的 buffer 大小。

```
//获取某个容量的对象 当大于每页的容量值
long allocate(int normCapacity) {
    if ((normCapacity & subpageOverflowMask) != 0) { // >= pageSize
        return allocateRun(normCapacity); //获取多个节点
    } else {
        return allocateSubpage(normCapacity); //获取到单个节点
    }
}
```

获取一个空闲页的大小 buffer 实现方法如下:

```

/**
 * 功能: 按深度遍历二叉树获取到空闲的PoolSubpage叶子节点, 然后PoolSubpage几次那个初始化
 *
 * @param normCapacity normalized capacity
 * @return index in memoryMap
 */
private long allocateSubpage(int normCapacity) {
    int d = maxOrder;
    int id = allocateNode(d); //按深度遍历二叉树, 获取到当前页的空闲节点, 当没有空闲的节点直接返回
    if (id < 0) {
        return id;
    }

    final PoolSubpage<T>[] subpages = this.subpages;
    final int pageSize = this.pageSize;
    //将空闲的容量调小
    freeBytes -= pageSize;
    int subpageIdx = subpageIdx(id);
    PoolSubpage<T> subpage = subpages[subpageIdx]; //根据索引获取到对应PoolSubpage
    //当这一页没有值得时候 创建新的页
    if (subpage == null) {
        subpage = new PoolSubpage<T>(this, id, runOffset(id), pageSize, normCapacity);
        subpages[subpageIdx] = subpage; //设置新页
    } else {
        //重新初始化新页
        subpage.init(normCapacity);
    }
    return subpage.allocate();
}

```

从根节点开始 逐层向下查找各个节点获取到 空闲的节点算法实现如下:

```

/**
 *
 * 从根节点开始 逐层向下查找各个节点获取到 空闲的节点id 当没有空闲节点返回 -1
 * @param d depth
 * @return index in memoryMap
 */
private int allocateNode(int d) {
    int id = 1;
    int initial = - (1 << d); //设置初始值
    byte val = value(id);
    if (val > d) { // 判断根节点是否是空闲的, 如果不是说明这一页已经全部使用了
        return -1;
    }
    while (val < d || (id & initial) == 0) { // 按深度, 逐层向下遍历整个二叉树节点
        id <<= 1;
        val = value(id);
        if (val > d) {
            id ^= 1;
            val = value(id);
        }
    }
    byte value = value(id); //找到空闲节点
    assert value == d && (id & initial) == 1 << d : String.format("val = %d, id & initial = %d, d = %d",
        value, id & initial, d);
    setValue(id, unusable); // 标记这个空闲节点已经不可用
    updateParentsAlloc(id); //从当前节点逐层向上更新父节点的状态
    return id;
}

```

## 7 分配 PoolSubpage

因为请求的容量不足一个 subpage 的容量, 会对 PoolSubpage 按照请求的大小进行等分, 分成 N 个小格子, 以充分使用内存大小。同时将当前的 PoolSubpage 加入到 PoolArena 缓存中, 以方便使用。

```

}

void init(int elemSize) {
    doNotDestroy = true;
    this.elemSize = elemSize;
    if (elemSize != 0) {
        //设置多少个元素
        maxNumElems = numAvail = pageSize / elemSize;
        nextAvail = 0;
        //位图的长度
        bitmapLength = maxNumElems >>> 6;
        if ((maxNumElems & 63) != 0) {
            bitmapLength ++;
        }

        for (int i = 0; i < bitmapLength; i ++) {
            bitmap[i] = 0;
        }
    }
    //并将当前的PoolSubpage添加到PoolArena中
    addToPool();
}

```

## 9 PooledByteBuf 的初始化

主要是获取到 PooledByteBuf 对象在整个对象缓存区中的开始位置和结束位置。

```

/**
 * 功能: 主要是设置buffer的write Index 和read Index
 * @param buf
 * @param handle
 * @param reqCapacity
 */
void initBuf(PooledByteBuf<T> buf, long handle, int reqCapacity) {
    int memoryMapIdx = (int) handle;
    int bitmapIdx = (int) (handle >>> Integer.SIZE);
    if (bitmapIdx == 0) {
        byte val = value(memoryMapIdx);
        assert val == unusable : String.valueOf(val);
        buf.init(this, handle, runOffset(memoryMapIdx), reqCapacity, runLength(memoryMapIdx));
    } else {
        initBufWithSubpage(buf, handle, bitmapIdx, reqCapacity);
    }
}

void initBufWithSubpage(PooledByteBuf<T> buf, long handle, int reqCapacity) {
    initBufWithSubpage(buf, handle, (int) (handle >>> Integer.SIZE), reqCapacity);
}

private void initBufWithSubpage(PooledByteBuf<T> buf, long handle, int bitmapIdx, int reqCapacity) {
    assert bitmapIdx != 0;
    int memoryMapIdx = (int) handle;
    PoolSubpage<T> subpage = subpages[subpageIdx(memoryMapIdx)];
    assert subpage.doNotDestroy;
    assert reqCapacity <= subpage.elemSize;

    //每页的开始位置+elemSize (每页等分格子的大小) *第几格子
    buf.init(
        this, handle,
        runOffset(memoryMapIdx) + (bitmapIdx & 0x3FFFFFFF) * subpage.elemSize, reqCapacity, subpage.elemSize);
}

```

## 6. Pooled Buffer 内存监控

### ◆ 引用计数

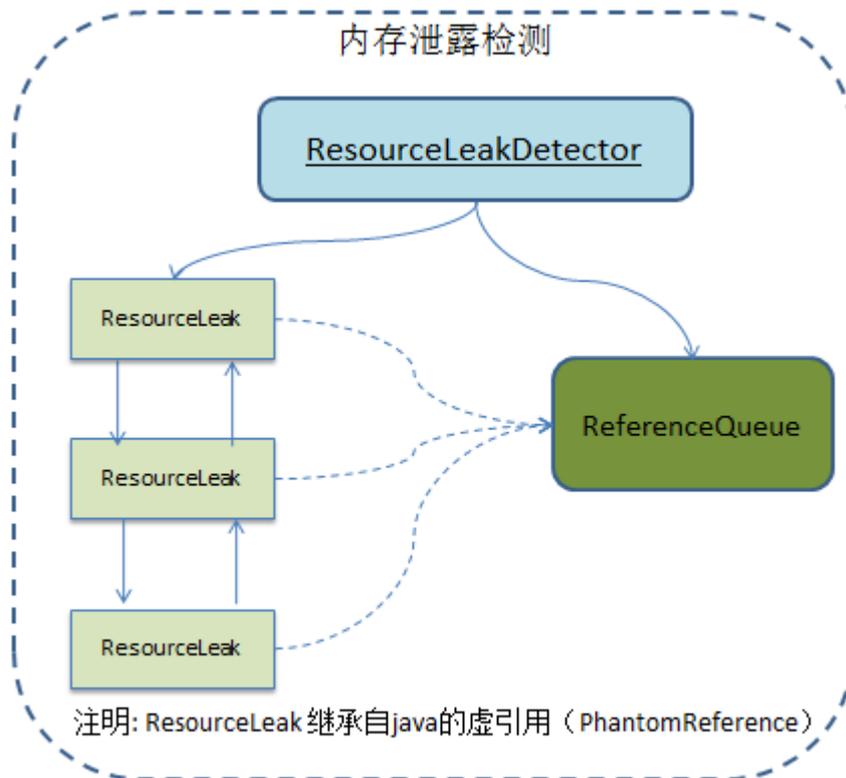
NETTY 中使用引用计数机制来管理资源,当一个实现 ReferenceCounted 的对象实例化时,引用计数置 1. 客户代码中需要保持一个该对象的引用时需要调用接口的 retain 方法将计数增 1.对象使用完毕时调用 release 将计数减 1. 当引用计数变为 0 时,对象将释放所持有的底层资源或将资源返回资源池.

### ◆ 内存泄露

按上述规则使用 `Direct` 和 `Pooled` 的 `ByteBuf` 时，内存使用情况的监控非常重要。`DirectBuf` 其内存不受 VM 垃圾回收控制只有在调用 `release` 导致计数为 0 时才会主动释放内存，而 `PooledByteBuf` 只有在 `release` 后才能被回收到池中，以便以循环利用。如果用户只使用不去 `release`，将会导致内存泄露。最终会导致整个系统 OOM。

◆ 内存泄露检测

NETTY 通过 java 的虚引用（`PhantomReference`）来实现了内存使用情况的跟踪。在从 `Pool` 中取出 `buffer` 会被包装成一个虚引用的对象，并通过门面模式封装成 `SimpleLeakAwareByteBuf`、`AdvancedLeakAwareByteBuf` 对象来对外提供功能。具体实现模型如下：



通过判断 `ReferenceQueue` 中在垃圾回收的对象是否还是左边链表中的节点就可以确定是否存在内存泄露。因为在调用 `release` 方法时候会将 `ResourceLeak` 从链表中删除。

1 通过门面模式服装检测资源

```
@SuppressWarnings("incomplete-switch")
protected static ByteBuf toLeakAwareBuffer(ByteBuf buf) {
    ResourceLeak leak;
    switch (ResourceLeakDetector.getLevel()) { // 获取到检测级别
        case SIMPLE:
            leak = AbstractByteBuf.LeakDetector.open(buf); // 获取到泄露检测资源
            if (leak != null) {
                buf = new SimpleLeakAwareByteBuf(buf, leak); // 通过门面模式对外提供功能，屏蔽了内存泄露检测
            }
            break;
        case ADVANCED:
        case PARANOID:
            leak = AbstractByteBuf.LeakDetector.open(buf); // 获取到泄露检测资源
            if (leak != null) {
                buf = new AdvancedLeakAwareByteBuf(buf, leak); // 通过门面模式对外提供功能，屏蔽了内存泄露检测
            }
            break;
    }
    return buf;
}
```

## 2 体获取到检测资源的方法

```
public ResourceLeak open(T obj) {
    Level level = ResourceLeakDetector.level; // 获取到检测级别
    if (level == Level.DISABLED) {
        return null;
    }
    if (level.ordinal() < Level.PARANOID.ordinal()) {
        if (leakCheckCnt ++ % samplingInterval == 0) {
            reportLeak(level); // 检测内存泄露情况
            return new DefaultResourceLeak(obj); // 构造虚引用
        } else {
            return null;
        }
    } else {
        reportLeak(level); // 检测内存泄露情况
        return new DefaultResourceLeak(obj); // 构造虚引用
    }
}
```

## 3 具体构造虚引用的对象的实现

```
private final class DefaultResourceLeak extends PhantomReference<Object> implements ResourceLeak {  
  
    private static final int MAX_RECORDS = 4;  
  
    private final String creationRecord;  
    private final Deque<String> lastRecords = new ArrayDeque<String>();  
    private final AtomicBoolean freed;  
    private DefaultResourceLeak prev;  
    private DefaultResourceLeak next;  
  
    DefaultResourceLeak(Object referent) {  
        // super(referent, referent != null? refQueue : null);  
        super(referent, refQueue );//通过调用PhantomReference构造方法生成弱引用  
        if (referent != null) {  
            Level level = getLevel();  
            if (level.ordinal() >= Level.ADVANCED.ordinal()) {  
                creationRecord = newRecord(3);  
            } else {  
                creationRecord = null;  
            }  
            synchronized (head) {  
                prev = head;  
                next = head.next;  
                head.next.prev = this;  
                head.next = this;  
                active ++;  
            }  
            freed = new AtomicBoolean();  
        } else {  
            creationRecord = null;  
            freed = new AtomicBoolean(true);  
        }  
    }  
}
```