- NETTY

精尽 Netty 源码解析 —— Buffer 之 Jemalloc（二）PoolChunk

1. 概述

> 老芳芳：如下阐释的内容，参考 Hypercube 《自顶向下深入分析Netty（十）–JEMalloc分配算法》 。

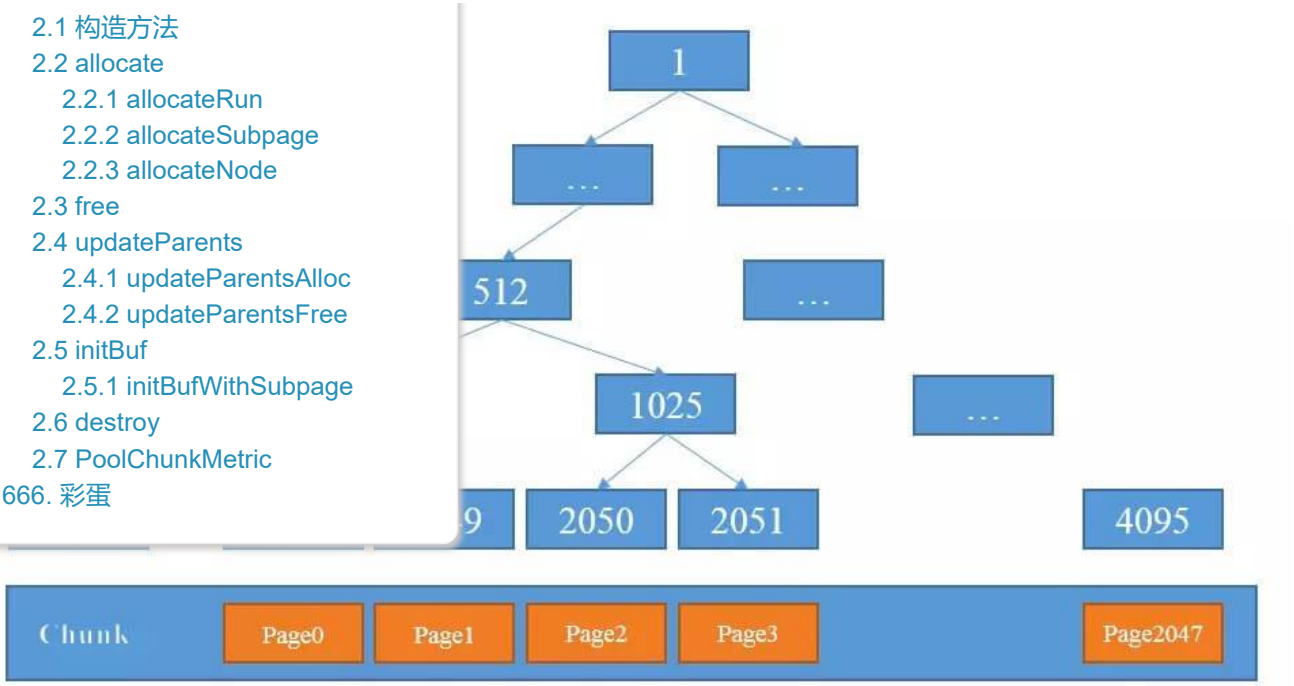
为了提高内存**分配效率**并减少**内存碎片**，Jemalloc 算法将每个 Arena 切分成多个**小块** Chunk 。但是实际上，每个 Chunk 依然是**相当大**的内存块。因为在 Jemalloc 建议为 4MB ， Netty 默认使用为 16MB 。

为了进一步提供提高内存**分配效率**并减少**内存碎片**，Jemalloc 算法将每个 Chunk 切分成多个**小块** Page 。一个典型的切分是将 Chunk 切分为 2048 块 Page ， Netty 也是如此，因此 Page 的大小为： `16MB / 2048 = 8KB` 。

一个好的内存分配算法，应使得已分配内存块尽可能保持连续，这将大大减少内部碎片，由此 Jemalloc 使用伙伴分配算法尽可能提高连续性。**伙伴分配算法**的示意图如下：

> 可能很多胖友不了解【伙伴分配算法】，感兴趣的话，可以看看 《伙伴分配器的一个极简实现》 了解了解。
>
> 当然，Netty PoolChunk 也是基于【伙伴分配算法】实现。

文章目录



图中**最底层**表示一个被切分为 2048 个 Page 的 Chunk 块。自底向上，每一层节点作为上一层的子节点构造出一棵满二叉树，然后按层分配满足要求的内存块。以待分配序列 8KB、16KB、8KB 为例分析分配过程(假设每个 Page 大小 8KB):

1. 8KB —— 需要一个 Page ， 第 11 层满足要求，故分配 *2048* 节点即 **Page0** 。
2. 16KB —— 需要两个Page ， 故需要在第 10 层进行分配，而 *1024* 的子节点 *2048* 已分配，从左到右找到满足要求的 *1025* 节点，故分配节点 *1025* 即**Page2** 和 **Page3** 。
3. 8KB —— 需要一个 Page ， 第 11 层满足要求，但是 *2048* 已分配，从左到右找到 *2049* 节点即 **Page1** 进行分配。

总结来说:

- 分配结束后，已分配连续的 **Page0 - Page3** 。这样的连续内存块，大大减少内部碎片并提高**内存使用率**。
- 通过使用**满二叉树**这样的树结构，提升检索到可用 Page 的速度，从而提高内存**分配效率**。

2. PoolChunk

`io.netty.buffer.PoolChunk` ，实现 PoolChunkMetric 接口，Netty 对 Jemalloc Chunk 的实现类。

2.1 构造方法

```
/**
 * 所属 Arena 对象
 */
final PoolArena<T> arena;
/**
 * 内存空间。
 *
 * @see PooledByteBuf#memory
 */
final T memory;
/**
```


`int, int)` 非池化。当申请的内存大小为 `Huge` 类型时，创建一整块 `Chunk` ，`int, int, int, int, int)` 池化

```
 * 高度信息满二叉树
 *
 * index 为节点编号
 */
private final byte[] depthMap;
/**
 * PoolSubpage 数组
 */
private final PoolSubpage<T>[] subpages;
/**
 * 判断分配请求内存是否为 Tiny/Small ，即分配 Subpage 内存块。
 *
 * Used to determine if the requested capacity is equal to or greater than pageSize.
 */
private final int subpageOverflowMask;
/**
 * Page 大小，默认 8KB = 8192B
 */
private final int pageSize;
/**
 * 从 1 开始左移到 {@link #pageSize} 的位数。默认 13 ，1 << 13 = 8192 。
 *
 * 具体用途，见 {@link #allocateRun(int)} 方法，计算指定容量所在满二叉树的层级。
 */
private final int pageShifts;
/**
 * 满二叉树的高度。默认为 11 。
 */
private final int maxOrder;
/**
 * Chunk 内存块占用大小。默认为 16M = 16 * 1024  。
 */
private final int chunkSize;
/**
 * log2 {@link #chunkSize} 的结果。默认为 log2( 16M ) = 24 。
 */
private final int log2ChunkSize;
```


数组大小。默认为 1 << maxOrder = 1 << 11 = 2048 。

= 12 。

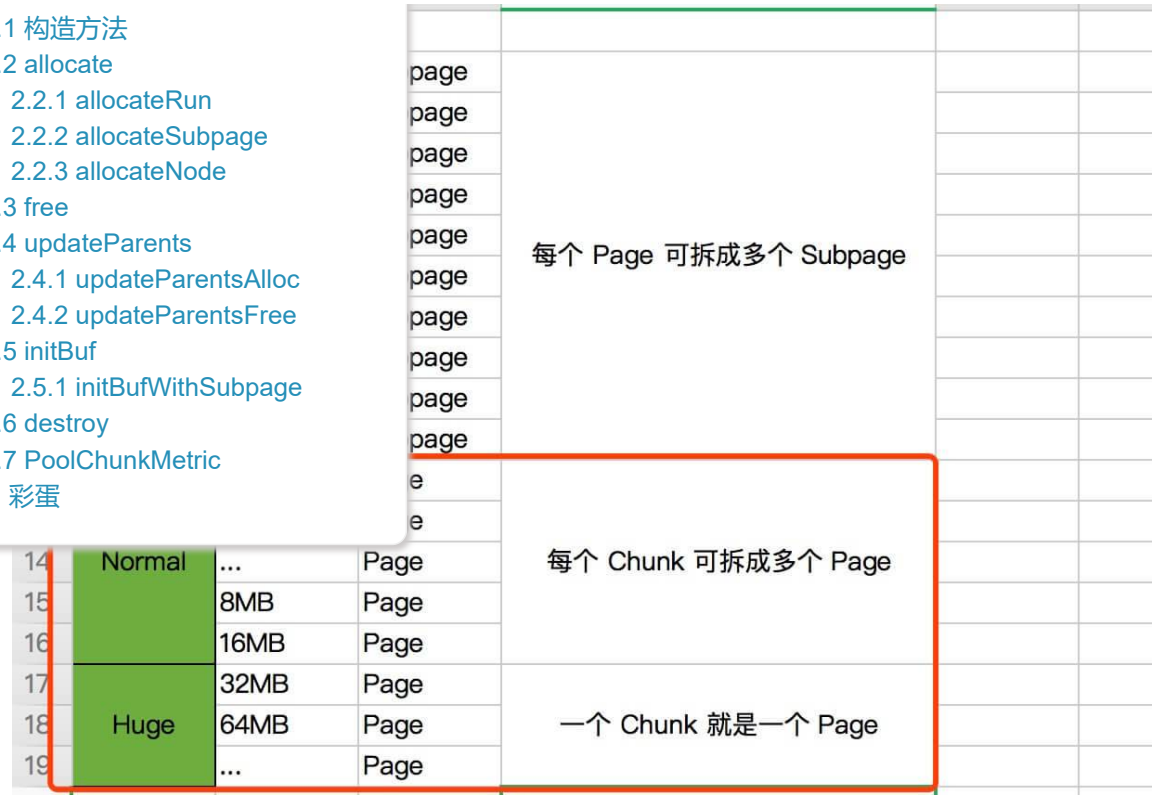
```
 * 所属 PoolChunkList 对象
 */
PoolChunkList<T> parent;
/**
 * 上一个 Chunk 对象
 */
PoolChunk<T> prev;
/**
 * 下一个 Chunk 对象
 */
PoolChunk<T> next;

// 构造方法一：
 1: PoolChunk(PoolArena<T> arena, T memory, int pageSize, int maxOrder, int pageShifts, int chunkSize
 2:     // 池化
 3:     unpooled = false;
 4:     this.arena = arena;
 5:     this.memory = memory;
 6:     this.pageSize = pageSize;
 7:     this.pageShifts = pageShifts;
 8:     this.maxOrder = maxOrder;
 9:     this.chunkSize = chunkSize;
10:     this.offset = offset;
11:     unusable = (byte) (maxOrder + 1);
12:     log2ChunkSize = log2(chunkSize);
13:     subpageOverflowMask = ~(pageSize - 1);
14:     freeBytes = chunkSize;
15:
16:     assert maxOrder < 30 : "maxOrder should be < 30, but is: " + maxOrder;
17:     maxSubpageAllocs = 1 << maxOrder;
18:
19:     // 初始化 memoryMap 和 depthMap
20:     // Generate the memory map.
21:     memoryMap = new byte[maxSubpageAllocs << 1];
22:     depthMap = new byte[memoryMap.length];
23:     int memoryMapIndex = 1;
24:     for (int d = 0; d <= maxOrder; ++ d) { // move down the tree one level at a time
```

文章目录

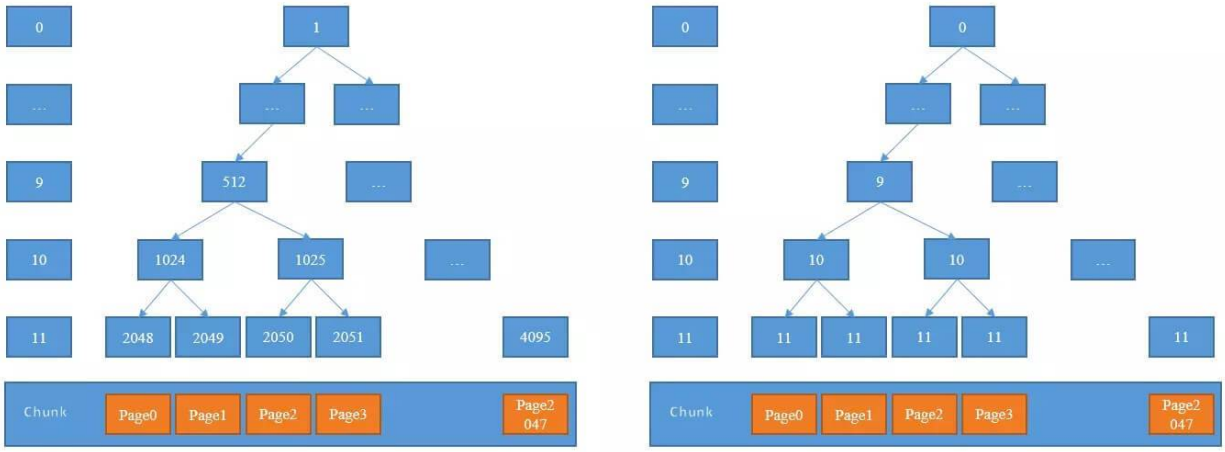
```
ndex] = (byte) d;
dex] = (byte) d;

axSubpageAllocs);

 is not pooled. */
memory, int size, int offset) {
    // 非池化
    unpooled = true;
    this.arena = arena;
    this.memory = memory;
    this.offset = offset;
    memoryMap = null;
    depthMap = null;
    subpages = null;
    subpageOverflowMask = 0;
    pageSize = 0;
    pageShifts = 0;
    maxOrder = 0;
    unusable = (byte) (maxOrder + 1);
    chunkSize = size;
    log2ChunkSize = log2(chunkSize);
    maxSubpageAllocs = 0;
}
```

- `arena` 属性，所属 Arena 对象。详细解析，见《精尽 Netty 源码解析 —— Buffer 之 Jemalloc（五）PoolArena》。
- `memory` 属性，内存空间。即**用于** `PooledByteBuf.memory` 属性，有 Direct ByteBuffer 和 `byte[]` 字节数组。
- `unpooled` 属性，是否非池化。
  - `unpooled = false` ，池化，对应构造方法**一**。默认情况下，对于 分配 16M **以内**的内存空间时，Netty 会分配一个 Normal 类型的 Chunk 块。并且，该 Chunk 块在使用完后，进行池化缓存，重复使用。
  - `unpooled = true` ，非池化，对应构造方法**二**。默认情况下，对于分配 16M **以上**的内存空间时，Netty 会分配一个 Huge 类型的**特殊**的 Chunk 块。并且，由于 Huge 类型的 Chunk 占用内存空间较大，比较特殊，所以该 Chunk 块在使用完后，立即释放，不进行重复使用。
  - 笔者对 Netty 对 Jemalloc 不同类型的内存块的整理，如下图所示：

文章目录



- Jemalloc 基于【伙伴分配算法】分配 Chunk 中的 Page 节点。Netty 实现的伙伴分配算法中，构造了**两颗**满二叉树。因为满二叉树非常适合数组存储，Netty 使用两个字节数组 `memoryMap` 和 `depthMap` 来分别表示**分配信息**满二叉树、**高度信息**满二叉树。如下图所示：



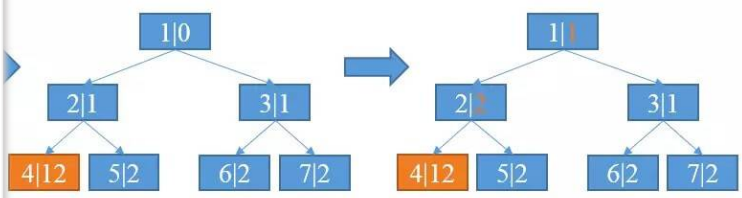
  - `maxOrder` 属性，满二叉树的高度。默认为 11 。注意，层高是从 0 开始。
  - `maxSubpageAllocs` 属性，可分配的 Page 的数量。默认为 2048 ，在【第 17 行】的代码进行初始化。在第 11 层，可以看到 Page0 - Page2047 这 2048 个节点，也也符合 `1 << maxOrder = 11 << 11 = 2048` 的计算。
  - 在【第 19 至 32 行】的代码， `memoryMap` 和 `depthMap` 进行满二叉树的初始化。
    - 数组大小为 `maxSubpageAllocs << 1 = 2048 << 1 = 4096` 。
    - 数组下标为**左图**对应的节点编号。在【第 23 行】的代码，从 `memoryMapIndex = 1` 代码可以看出，满二叉树的节点编号是**从 1 开始**。省略 0 是因为这样更容易计算父子关系：子节点加倍，父节点减半，例如：512 的子节点为 1024(`512 * 2`)和 1025(`512 * 2 + 1`)。
    - 初始时， `memoryMap` 和 `depthMap` 相等，值为**节点高度**。例如：

```
memoryMap[1024] = depthMap[1024] = 10;
```

文章目录

变(因为，节点的高度没发生变化)， `memoryMap` 的值发生**变化**(因个节点被分配后，该节点的值设为 `unusable` (标记节点不可用。默，**会更新祖先节点的值为其子节点较小的值**(因为，祖先节点共用该节

两个子节点，一个节点不可用后，另一个子节点可能可用，所以更新
下图表示随着节点 4 分配而更新祖先节点的过程，其中每个节点的第
示**节点高度**：



为 12 表示不可用。
为两个子节点的较小值。其他祖先节点亦然，直到高度值更新至根节

情况：

[id] ，该节点没有被分配。

- 2、 最大高度 >= memoryMap[id] > depthMap[id] ，至少有一个子节点被分配，不能再分配该高度满足的内存，但可以根据实际分配较小一些的内存。比如，上图中父节点 2 分配了子节点 4，值从 1 更新为 2，表示该节点不能再分配 8MB 的只能最大分配 4MB 内存，即只剩下节点 5 可用。
- 3、 memoryMap[id] = 最大高度 + 1 ，该节点及其子节点已被**完全**分配，没有剩余空间。

- Chunk 相关字段
  - chunkSize 属性，Chunk 内存块占用大小。默认为 16M = 16 * 1024KB 。
  - log2ChunkSize 属性， log2(chunkSize) 的结果。默认为 log2(16M) = 24 。代码如下：

```
private static final int INTEGER_SIZE_MINUS_ONE = Integer.SIZE - 1; // 32 - 1 = 31

private static int log2(int val) {
    // compute the (0-based, with lsb = 0) position of highest set bit i.e, log2
    return INTEGER_SIZE_MINUS_ONE - Integer.numberOfLeadingZeros(val);
}
```

  - x
  - freeBytes 属性，剩余可用字节数。
- Page 相关字段
  - pageSize 属性，每个 Page 的大小。默认为 8KB = 8192B 。
  - pageShifts 属性，从 1 开始左移到 pageSize 的位数。默认 13， 1 << 13 = 8192 。具体用于计算指定容量所在满二叉树的层级，详细解析，见「2.2.1 allocateRun」。
- SubPage 相关字段
  - 详细解析，见《精尽 Netty 源码解析 —— Buffer 之 Jemalloc（三）PoolSubpage》。
  - subpages 属性，PoolSubpage 数组。每个节点对应一个 PoolSubpage 对象。因为实际上，每个 Page 还是**比较大**的内存块，可以进一步切分成小块 SubPage 。在【第 35 行】的代码，调用 #newSubpageArray(int size) 方法，进行初始化。代码如下：

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

  - 默认情况下，数组大小为 maxSubpageAllocs = 2048 。
  - subpageOverflowMask 属性，判断分配请求内存是否为 **Tiny/Small** ，即分配 Subpage 内存块。默认，-8192
    在【13 行】的代码进行初始化。对于 -8192 的二进制，除了首 bits 为 1 ，其它都为 0 。这样，对于小于 8K 字
    & length 都等于 0 ；对于大于 8K 字节的申请，求
    不等于 0 。相当于说，做了 if (length < pageSize) 的计算优

– Buffer 之 Jemalloc（四）PoolChunkList》 。
象。

一遍，再看下面的代码具体实现。

配内存空间。代码如下：

```
 内存块
    if ((normCapacity & subpageOverflowMask) != 0) { // >= pageSize
        return allocateRun(normCapacity);
    // 小于 Page 大小，分配 Subpage 内存块
    } else {
        return allocateSubpage(normCapacity);
    }
}
```

- 第 2 至 4 行：当申请的 `normCapacity` 大于等于 Page 大小时，调用 `#allocateRun(int normCapacity)` 方法，分配 Page 内存块。详细解析，见 「2.2.1 allocateRun」 中。
- 第 5 至 8 行：调用 `#allocateSubpage(int normCapacity)` 方法，分配 Subpage 内存块。详细解析，见 「2.2.1 allocateSubpage」 中。

2.2.1 allocateRun

`#allocateRun(int normCapacity)` 方法，分配 Page 内存块。代码如下：

```
/**
 * Allocate a run of pages (>=1)
 *
 * @param normCapacity normalized capacity
 * @return index in memoryMap
 */
private long allocateRun(int normCapacity) {
    // 获得层级
    int d = maxOrder - (log2(normCapacity) - pageShifts);
    // 获得节点
    int id = allocateNode(d);
    // 未获得到节点，直接返回
    if (id < 0) {
        return id;
    }
    // 减少剩余可用字节数
    freeBytes -= runLength(id);
    return id;
}
```


Capacity) 方法，分配节点。详细解析，见 「2.2.3 allocateNode」 中。

法，计算使用节点的字节数，并减少剩余可用字节数。代码如下：

```
es supported by node 'id' in the tree
oth(id);
```

> 老艿艿：本小节，胖友先看完 《精尽 Netty 源码解析 —— Buffer 之 Jemalloc（三）PoolSubpage》 。

#allocateSubpage(int normCapacity) 方法，分配 Subpage 内存块。代码如下：

```
    /**
     * Create/ initialize a new PoolSubpage of normCapacity
     * Any PoolSubpage created/ initialized here is added to subpage pool in the PoolArena that owns th
     *
     * @param normCapacity normalized capacity
     * @return index in memoryMap
     */
 1: private long allocateSubpage(int normCapacity) {
 2:     // 获得对应内存规格的 Subpage 双向链表的 head 节点
 3:     // Obtain the head of the PoolSubPage pool that is owned by the PoolArena and synchronize on i
 4:     // This is need as we may add it back and so alter the linked-list structure.
 5:     PoolSubpage<T> head = arena.findSubpagePoolHead(normCapacity);
 6:     // 加锁，分配过程会修改双向链表的结构，会存在多线程的情况。
 7:     synchronized (head) {
 8:         // 获得最底层的一个节点。Subpage 只能使用二叉树的最底层的节点。
 9:         int d = maxOrder; // subpages are only be allocated from pages i.e., leaves
10:         int id = allocateNode(d);
11:         // 获取失败，直接返回
12:         if (id < 0) {
13:             return id;
14:         }
15:
16:         final PoolSubpage<T>[] subpages = this.subpages;
17:         final int pageSize = this.pageSize;
18:
19:         // 减少剩余可用字节数
```


数组的编号
dx(id);

```
数组的 PoolSubpage 对象
subpages[subpageIdx];

不存在，则进行创建 PoolSubpage 对象
page<T>(head, this, id, runOffset(id), pageSize, normCapacity);
= subpage;
始化 PoolSubpage 对象
rmCapacity);

;
```

- 第 5 行：调用 `PoolArena#findSubpagePoolHead(int normCapacity)` 方法，获得对应内存规格的 Subpage 双向链表的 `head` 节点。详细解析，见《精尽 Netty 源码解析 —— Buffer 之 Jemalloc（五）PoolArena》。
- 第 7 行：`synchronized` 加锁，分配过程会修改双向链表的结构，会存在**多线程**的情况。
- 第 8 至 10 行：调用 `#allocateNode(int d)` 方法，获得最底层的一个节点。**Subpage 只能使用二叉树的最底层的节点。**
  - 第 11 至 14 行：获取失败，直接返回。
  - 第 20 行：减少剩余可用字节数。
- 第 23 至 34 行：分配 PoolSubpage 内存块。
  - 第 23 行：调用 `#subpageIdx(int id)` 方法，获得节点对应的 `subpages` 数组的编号。代码如下：

```
private int subpageIdx(int memoryMapIdx) {
    return memoryMapIdx ^ maxSubpageAllocs; // remove highest set bit, to get offset
}
```

    - 去掉最高位(bit)。例如节点 2048 计算后的结果为 0 。
  - 第 25 行：获得节点对应的 `subpages` 数组的 PoolSubpage 对象。
  - 第 26 至 32 行：初始化 PoolSubpage 对象。
  - 第 34 行：调用 `PoolSubpage#allocate()` 方法，分配 PoolSubpage 内存块。

2.2.3 allocateNode

`#allocateNode(int normCapacity)` 方法，分配节点。代码如下：

```
/**
 * Algorithm to allocate an index in memoryMap when we query for a free node
 * at depth d
 *
 * @param d depth
 * @return index in memoryMap
 */
1: private int allocateNode(int d) {
2:     int id = 1;
3:     int initial = - (1 << d); // has last d bits = 0 and rest all = 1
4:     // 获得根节点的指值。
5:     // 如果根节点的值 大于 d ，说明，第 d 层没有符合的节点，也就是说 [0, d-1] 层也没有符合的节点。即，当
```

于 d 会继续循环

al) == 0) { // id & initial == 1 << d for all ids at depth d, for

左节点作为根节点形成虚拟的虚拟满二叉树，没有符合的节点。

```
23:             val = value(id);
24:         }
25:     }
26:
27:     // 校验获得的节点值合理
28:     byte value = value(id);
29:     assert value == d && (id & initial) == 1 << d : String.format("val = %d, id & initial = %d, d
30:             value, id & initial, d);
31:
32:     // 更新获得的节点不可用
33:     setValue(id, unusable); // mark as unusable
34:     // 更新获得的节点的祖先都不可用
35:     updateParentsAlloc(id);
36:
37:     // 返回节点编号
38:     return id;
39: }
```

- 第 3 行：通过 `-(1 << d)` 计算，获得 `initial` 。用于【第 12 行】的代码， `id & initial` ，来保证，高度小于 `d` 会继续**循环**。
- 第 6 行：获得根节点(`id = 1`)的指值。代码如下：

```
private byte value(int id) {
    return memoryMap[id];
}
```

  - 第 7 至 9 行：如果根节点的值，大于 `d` ，说明，第 `d` 层没有符合的节点，也就是说 `[1, d-1]` 层也没有符合的节点。即，当前 Chunk 没有符合的节点。
- 第 10 至 25 行：获得第 `d` 层，匹配的节点。因为 `val < d` 难以保证是第 `d` 层， `[0, d-1]` 层也可以满足 `val < d` ，所以才有 `id & initial` 来保证，高度小于 `d` 会继续循环。
  - ← 第 15 行： `<< 1` 操作，进入下一层。获得**左节点**的编号。
  - ← 第 17 行：获得左节点的值。
  - → 第 19 行：如果值大于 `d` ，说明，以左节点作为根节点形成虚拟的虚拟满二叉树，没有符合的节点。此时，需要跳到**右节点**。
  - → 第 21 行： `^ 1` 操作，获得**右节点**的编号。

会通过【第 12 行】的代码，结束循环。也就说，获得第 `d` 层，匹配的节

`e val)` 方法，设置获得的节点的值为 `unusable` ，表示不可用。代码如

```
val) {
```

`nt id)` 方法，更新获得的节点的祖先都不可用。详细解析，见 「2.4.1

> 老艿艿：本小节，胖友先看完 《精尽 Netty 源码解析 —— Buffer 之 Jemalloc（三）PoolSubpage》 。

`#free(long handle)` 方法，释放指定位置的内存块。根据情况，内存块可能是 SubPage ，也可能是 Page ，也可能是释放 SubPage 并且释放对应的 Page 。代码如下：

```
/**
 * Free a subpage or a run of pages
 * When a subpage is freed from PoolSubpage, it might be added back to subpage pool of the owning P
 * If the subpage pool in PoolArena has at least one other PoolSubpage of given elemSize, we can
 * completely free the owning Page so it is available for subsequent allocations
 *
 * @param handle handle to free
 */
1: void free(long handle) {
2:     // 获得 memoryMap 数组的编号( 下标 )
3:     int memoryMapIdx = memoryMapIdx(handle);
4:     // 获得 bitmap 数组的编号( 下标 )。注意，此时获得的还不是真正的 bitmapIdx 值，需要经过 `bitmapIdx &
5:     int bitmapIdx = bitmapIdx(handle);
6:
7:     // 释放 Subpage begin ~
8:
9:     if (bitmapIdx != 0) { // free a subpage bitmapIdx 非空，说明释放的是 Subpage
10:         // 获得 PoolSubpage 对象
11:         PoolSubpage<T> subpage = subpages[subpageIdx(memoryMapIdx)];
12:         assert subpage != null && subpage.doNotDestroy;
13:
14:         // 获得对应内存规格的 Subpage 双向链表的 head 节点
15:         // Obtain the head of the PoolSubPage pool that is owned by the PoolArena and synchronize
16:         // This is need as we may add it back and so alter the linked-list structure.
17:         PoolSubpage<T> head = arena.findSubpagePoolHead(subpage.elemSize);
18:         // 加锁，分配过程会修改双向链表的结构，会存在多线程的情况。
19:         synchronized (head) {
                , bitmapIdx & 0x3FFFFFFF)) {
```

明 Page 中无切分正在使用的 Subpage 内存块，所以可以继续向下执行，释放

MapIdx);

memoryMapIdx));
用
x);

- 第 3 行：调用 `#memoryMapIdx(handle)` 方法，获得 `memoryMap` 数组的编号(下标)。代码如下：

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

- 第 5 行：调用 `#bitmapIdx(handle)` 方法，获得 `bitmap` 数组的编号(下标)。代码如下：

```
private static int bitmapIdx(long handle) {
    return (int) (handle >>> Integer.SIZE);
}
```

  - 注意，此时获得的还不是真正的 bitmapIdx 值，需要经过 `bitmapIdx & 0x3FFFFFFF` 运算，即【第 21 行】的代码。
- 第 9 至 26 行：释放 Subpage 内存块。
  - 第 9 行：通过 `bitmapIdx !=0` 判断，说明释放的是 Subpage 内存块。
  - 第 11 行：获得 PoolSubpage 对象。
  - 第 17 行：调用 `PoolArena#findSubpagePoolHead(int normCapacity)` 方法，获得对应内存规格的 Subpage 双向链表的 `head` 节点。详细解析，见《精尽 Netty 源码解析 —— Buffer 之 Jemalloc（五） PoolArena》 。
  - 第 19 行： `synchronized` 加锁，分配过程会修改双向链表的结构，会存在多线程的情况。
  - 第 21 行：调用 `SubPage#free(PoolSubpage<T> head, int bitmapIdx)` 方法，释放 Subpage 内存块。
    - 如果返回 `false` ，说明 Page 中**无切分正在使用**的 Subpage 内存块，所以可以继续向下执行，释放 Page 内存块。
- 第 30 至 35 行：释放 Page 内存块。
  - 第 31 行：增加剩余可用字节数。
  - 第 33 行：调用 `#setValue(int id, byte val)` 方法，设置 Page 对应的节点**可用**。
  - 第 35 行：调用 `#updateParentsAlloc(int id)` 方法，更新获得的节点的祖先**可用**。

2.4 updateParents

2.4.1 updateParentsAlloc

`#updateParentsAlloc(int id)` 方法，更新获得的节点的祖先都不可用。代码如下：

uccessor is allocated and all its predecessors

```
ree rooted at id has some free space

nt id) {

;
到父节点
10:         byte val = val1 < val2 ? val1 : val2;
11:         setValue(parentId, val);
12:         // 跳到父节点
13:         id = parentId;
14:     }
15: }
```

- 😈 注意，调用此方法时，节点 `id` 已经更新为**不可用**。
- 第 2 行：循环，直到**根**节点。
- 第 4 行： `>>> 1` 操作，获得父节点的编号。
- 第 5 至 11 行：获得子节点较小值，并调用 `#setValue(int id, int value)` 方法，设置到父节点。
- 第 13 行：跳到父节点。

2.4.2 updateParentsFree

`#updateParentsAlloc(int id)` 方法，更新获得的节点的祖先可用。代码如下：

```
/**
 * Update method used by free
 * This needs to handle the special case when both children are completely free
 * in which case parent be directly allocated on request of size = child-size * 2
 *
 * @param id id
 */
1: private void updateParentsFree(int id) {
2:     // 获得当前节点的子节点的层级
3:     int logChild = depth(id) + 1;
4:     while (id > 1) {
5:         // 获得父节点的编号
6:         int parentId = id >>> 1;
7:         // 获得子节点的值
8:         byte val1 = value(id);
9:         // 获得另外一个子节点的值
10:        byte val2 = value(id ^ 1);
11:        // 获得当前节点的层级
t iteration equals log, subsequently reduce 1 from logChild as we

设置父节点的层级
al2 == logChild) {
```

```
yte) (logChild - 1));
取子节点较小值，并设置到父节点

2 ? val1 : val2;
1);
```

新为可用。

- 第 3 行：获得当前节点的子节点的层级。
- 第 4 行：循环，直到**根**节点。
- 第 6 行： `>>> 1` 操作，获得**父**节点的编号。
- 第 7 至 10 行：获得两个**子**节点的值。
- 第 12 行：获得当前节点的层级。
- 第 14 至 16 行：两个子节点都可用，则调用 `#setValue(id, value)` 方法，直接设置父节点的层级(注意，是 `logChild - 1`)。
- 第 17 至 21 行：两个子节点任一不可用，则 `#setValue(id, value)` 方法，取子节点较小值，并设置到父节点。
- 第 24 行：跳到父节点。

2.5 initBuf

`#initBuf(PooledByteBuf<T> buf, long handle, int reqCapacity)` 方法，初始化分配的内存块到 PooledByteBuf 中。代码如下：

```
 1: void initBuf(PooledByteBuf<T> buf, long handle, int reqCapacity) {
 2:     // 获得 memoryMap 数组的编号( 下标 )
 3:     int memoryMapIdx = memoryMapIdx(handle);
 4:     // 获得 bitmap 数组的编号( 下标 )。注意，此时获得的还不是真正的 bitmapIdx 值，需要经过 `bitmapIdx &
 5:     int bitmapIdx = bitmapIdx(handle);
 6:     // 内存块为 Page
 7:     if (bitmapIdx == 0) {
 8:         byte val = value(memoryMapIdx);
 9:         assert val == unusable : String.valueOf(val);
10:         // 初始化 Page 内存块到 PooledByteBuf 中
11:         buf.init(this, handle, runOffset(memoryMapIdx) + offset, reqCapacity, runLength(memoryMapI
12:     // 内存块为 SubPage
13:     } else {
14:         // 初始化 SubPage 内存块到 PooledByteBuf 中
15:         initBufWithSubpage(buf, handle, bitmapIdx, reqCapacity);
16:     }
17: }
```

- 第 3 行：调用 `#memoryMapIdx(handle)` 方法，获得 `memoryMap` 数组的编号(下标)。
- 第 5 行：调用 `#bitmapIdx(handle)` 方法，获得 `bitmap` 数组的编号(下标)。

断出，内存块是 Page 。所以，调用

unk, long handle, int offset, int length, int maxLength,

Page 内存块到 PooledByteBuf 中。其中， `runOffset(memoryMapIdx) +` `ory` 中的开始位置。 `runOffset(int id)` 方法，代码如下：

```
et in #bytes from start of the byte-array chunk
);

es supported by node 'id' in the tree
oth(id);
```

- 第12 至 16 行：通过 `bitmapIdx != 0` 判断出，内存块是 SubPage 。所以，调用 `#initBufWithSubpage(PooledByteBuf<T> buf, long handle, int reqCapacity)` 方法，初始化 SubPage 内存块到 PooledByteBuf 中。详细解析，见 「2.5.1 initBufWithSubpage」 。

2.5.1 initBufWithSubpage

`#initBufWithSubpage(PooledByteBuf<T> buf, long handle, int reqCapacity)` 方法，初始化 SubPage 内存块到 PooledByteBuf 中。代码如下：

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

```
 1: private void initBufWithSubpage(PooledByteBuf<T> buf, long handle, int bitmapIdx, int reqCapacity)
 2:     assert bitmapIdx != 0;
 3:
 4:     // 获得 memoryMap 数组的编号( 下标 )
 5:     int memoryMapIdx = memoryMapIdx(handle);
 6:     // 获得 SubPage 对象
 7:     PoolSubpage<T> subpage = subpages[subpageIdx(memoryMapIdx)];
 8:     assert subpage.doNotDestroy;
 9:     assert reqCapacity <= subpage.elemSize;
10:
11:     // 初始化 SubPage 内存块到 PooledByteBuf 中
12:     buf.init(
13:         this, handle,
14:         runOffset(memoryMapIdx) + (bitmapIdx & 0x3FFFFFFF) * subpage.elemSize + offset,
15:             reqCapacity, subpage.elemSize, arena.parent.threadCache());
16: }
```

- 第 3 至 7 行：获得 SubPage 对象。
- 第 11 至于 15 行：调用 `PooledByteBuf#init(PoolChunk<T> chunk, long handle, int offset, int length, int maxLength, PoolThreadCache cache)` 方法，初始化 SubPage 内存块到 PooledByteBuf 中。其中， `runOffset(memoryMapIdx) + (bitmapIdx & 0x3FFFFFFF) * subpage.elemSize + offset` 代码块，计算 SubPage 内存块在 `memory` 中的开始位置。

文章目录

unk 。代码如下：

uffer 之 Jemalloc（五）PoolArena》 。

olChunk Metric 接口。代码如下：

```
    /**
     * Return the percentage of the current usage of the chunk.
     */
    int usage();

    /**
     * Return the size of the chunk in bytes, this is the maximum of bytes that can be served out of t
     */
    int chunkSize();

    /**
     * Return the number of free bytes in the chunk.
     */
    int freeBytes();

}
```

PoolChunk 对 PoolChunkMetric 接口的实现，代码如下：

```
@Override
public int usage() {
    final int freeBytes;
    synchronized (arena) {
        freeBytes = this.freeBytes;
    }
    return usage(freeBytes);
}

private int usage(int freeBytes) {
    // 全部使用，100%
    if (freeBytes == 0) {
        return 100;
    }
```

文章目录

ytes * 100L / chunkSize);

```
}
```

- `synchronized` 的原因是，保证 `freeBytes` 对其它线程的可见性。对应 Github 提交为 a7fe6c01539d3ad92d7cd94a25daff9e10851088 。

> **Motivation**:
>
> As we may access the metrics exposed of PooledByteBufAllocator from another thread then the allocations happen we need to ensure we synchronize on the PoolArena to ensure correct visibility.
>
> **Modifications**:
>
> Synchronize on the PoolArena to ensure correct visibility.
>
> **Result**:
>
> Fix multi-thread issues on the metrics

666. 彩蛋

老艿艿有点二，在 `#allocateNode(int normCapacity)` 方法卡了很久。因为没看到 `memoryMap` 和 `depthMap` 数组，下标是从 1 开始的！！！我恨那。

参考如下文章：

- 占小狼 《深入浅出Netty内存管理 PoolChunk》
- Hypercube 《自顶向下深入分析Netty（十）–PoolChunk》