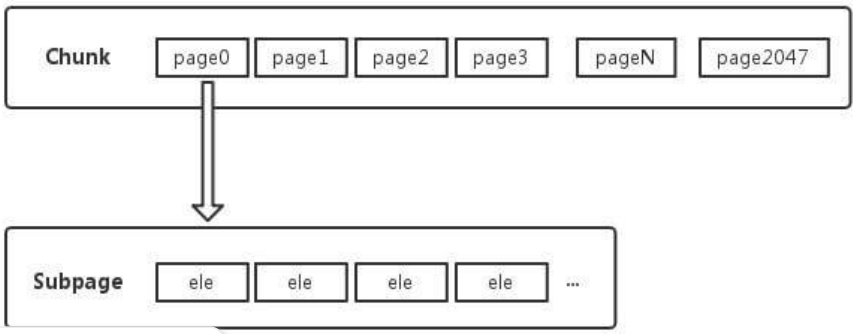
- NETTY

精尽 Netty 源码解析 —— Buffer 之 Jemalloc（三）PoolSubpage

1. 概述

在《精尽 Netty 源码解析 —— Buffer 之 Jemalloc（一）PoolChunk》一文中，我们已经看到，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 。

但是实际应用中，Page 也是**比较大**的内存块，如果直接使用，明显是很浪费的。因此，Jemalloc 算法将每个 Page 更进一步的切分为**多个** Subpage 内存块。Page 切分成**多个** Subpage 内存块，并未采用相对复杂的算法和数据结构，而是直接基于**数组**，通过数组来**标记**每个 Subpage 内存块是否已经分配。如下图所示：



存块**大小均等**。
不同，以 Page 第一次拆分为 Subpage 内存块时请求分配的内存大小为准。例

么 Page0 被拆成成 512(8KB / 16B)个 Subpage 块，使用第 0 块。
Page1 被拆分成 256(8KB / 32B)个 Subpage 块，使用第 0 块。
重用 Page0 ，使用第 1 块。
先去找**大小匹配**，且有可分配 Subpage 内存块的 Page： 1）如果有，则使
有，则选择一个新的 Page 拆分成多个 Subpage 内存块，使用第 0 块

ll 两类，并且每类有多种大小，如下图所示：

	分类	大小		
2	Tiny	16B	Subpage	每个 Page 可拆成多个 Subpage
3		32B	Subpage	
4		...	Subpage	
5		480B	Subpage	
6		496B	Subpage	
7	Small	512B	Subpage	
8		1KB	Subpage	
9		...	Subpage	
10		2KB	Subpage	
11		4KB	Subpage	
12	Normal	8KB	Page	每个 Chunk 可拆成多个 Page
13		16KB	Page	
14		...	Page	
15		8MB	Page	
16		16MB	Page	
17	Huge	32MB	Page	一个 Chunk 就是一个 Page
18		64MB	Page	
19		...	Page	
20				

- 为了方便描述，下文我们会继续将 `ele` 小块，描述成“Subpage 内存块”，简称“Subpage” 。

2. PoolSubpage

实现 PoolSubpageMetric 接口，Netty 对 Jemalloc Subpage 的实现类。

，实际描述的是，Page 切分为**多个** Subpage 内存块的分配情况。那么为什么不直
析 —— Buffer 之 Jemalloc（一）PoolChunk》 一文中，我们可以看到，当申请分
的是一块或多块 Page 内存块。如果 PoolSubpage 命名成 PoolPage 后，和这块的
，**只是 Page 分配内存的一种形式**。

文章目录

的节点编号

```
private final int memoryMapIdx;
/**
 * 在 Chunk 中，偏移字节量
 *
 * @see PoolChunk#runOffset(int)
 */
```

```
private final int runOffset;
/**
 * Page 大小 {@link PoolChunk#pageSize}
 */
private final int pageSize;

/**
 * Subpage 分配信息数组
 *
 * 每个 long 的 bits 位代表一个 Subpage 是否分配。
 * 因为 PoolSubpage 可能会超过 64 个( long 的 bits 位数 )，所以使用数组。
 *   例如：Page 默认大小为 8KB ，Subpage 默认最小为 16 B ，所以一个 Page 最多可包含 8 * 1024 / 16 = 512 个
 *        因此，bitmap 数组大小为 512 / 64 = 8 。
 * 另外，bitmap 的数组大小，使用 {@link #bitmapLength} 来标记。或者说，bitmap 数组，默认按照 Subpage 的大小
 *    为什么是这样的设定呢？因为 PoolSubpage 可重用，通过 {@link #init(PoolSubpage, int)} 进行重新初始化。
 */
private final long[] bitmap;

/**
 * 双向链表，前一个 PoolSubpage 对象
 */
PoolSubpage<T> prev;
/**
 * 双向链表，后一个 PoolSubpage 对象
 */
PoolSubpage<T> next;

/**
 * 是否未销毁
 */
boolean doNotDestroy;
/**
```

文章目录


```
 * 剩余可用 Subpage 的数量
 */
private int numAvail;

  1: // 【构造方法 1】 双向链表，头节点
  2: /** Special constructor that creates a linked list head */
```

```
3: PoolSubpage(int pageSize) {
4:     chunk = null;
5:     memoryMapIdx = -1;
6:     runOffset = -1;
7:     elemSize = -1;
8:     this.pageSize = pageSize;
9:     bitmap = null;
10: }
11:
12: // 【构造方法 2】 双向链表，Page 节点
13: PoolSubpage(PoolSubpage<T> head, PoolChunk<T> chunk, int memoryMapIdx, int runOffset, int pageSiz
14:     this.chunk = chunk;
15:     this.memoryMapIdx = memoryMapIdx;
16:     this.runOffset = runOffset;
17:     this.pageSize = pageSize;
18:     // 创建 bitmap 数组
19:     bitmap = new long[pageSize >>> 10]; // pageSize / 16 / 64
20:     // 初始化
21:     init(head, elemSize);
22: }
```

- Chunk 相关
  - `chunk` 属性，所属 PoolChunk 对象。
  - `memoryMapIdx` 属性，在 `PoolChunk.memoryMap` 的节点编号，例如节点编号 2048 。
  - `runOffset` 属性，在 Chunk 中，偏移字节量，通过 `PoolChunk#runOffset(id)` 方法计算。在 PoolSubpage 中，无相关的逻辑，仅用于 `#toString()` 方法，打印信息。
  - `pageSize` 属性，Page 大小。
- Subpage 相关
  - `bitmap` 属性，Subpage **分配信息**数组。
    - 1、每个 `long` 的 bits 位代表一个 Subpage 是否分配。因为 PoolSubpage 可能会超过 64 个(`long` 的 bits 位数)，所以使用数组。例如：Page 默认大小为 `8KB` ，Subpage 默认最小为 `16B` ，所以一个 Page 2 个 Subpage 。
    - itmap 数组。我们可以看到， `bitmap` 数组的大小为 8(`pageSize` = 512 / 64) 个。
    - Subpage **可重用**，通过 `#init(PoolSubpage, int)` 进行重新初始

    - itmapLength 属性来标记**真正**使用的数组大小。
    - 组的**真正**使用的数组大小。
    - 占用内存大小，例如 `16B` 、 `32B` 等等。
    - 的数量。例如 `16B` 为 512 个， `32b` 为 256 个。
    - e 的数量。
    - bpage 的数组(`bitmap`)位置。可能会有胖友有疑问， `bitmap` 又是数 在 「2.6 getNextAvail」 见分晓。
    - 详细解析，见 「2.5 free」 中。
    - Subpage 对象。
    - `next` 属性，双向链表，后一个 PoolSubpage 对象。
  - 详细解析，见 「2.3 双向链表」 。
- 构造方法 **1** ，用于创建双向链表的头(head)节点。
- 构造方法 **2** ，用于创建双向链表的 Page 节点。
  - 第 21 行：调用 `#init(PoolSubpage<T> head, int elemSize)` 方法，初始化。详细解析，见 「2.2 init」 。

2.2 init

#init(PoolSubpage<T> head, int elemSize) 方法，初始化。代码如下：

```
 1: void init(PoolSubpage<T> head, int elemSize) {
 2:     // 未销毁
 3:     doNotDestroy = true;
 4:     // 初始化 elemSize
 5:     this.elemSize = elemSize;
 6:     if (elemSize != 0) {
 7:         // 初始化 maxNumElems
 8:         maxNumElems = numAvail = pageSize / elemSize;
 9:         // 初始化 nextAvail
10:         nextAvail = 0;
11:         // 计算 bitmapLength 的大小
12:         bitmapLength = maxNumElems >>> 6;
13:         if ((maxNumElems & 63) != 0) { // 未整除，补 1.
14:             bitmapLength ++;
15:         }
16:
17:         // 初始化 bitmap
18:         for (int i = 0; i < bitmapLength; i ++) {
19:             bitmap[i] = 0;
20:         }
21:     }
22:     // 添加到 Arena 的双向链表中。
23:     addToPool(head);
24: }
```

- 第 3 行：未销毁。
- 第 5 行：初始化 `elemSize` 。
  - 第 8 行：初始化 `maxNumElems` 。
- 第 10 行：初始化 `nextAvail` 。
- 第 11 至 15 行：初始化 `bitmapLength` 。

age<T> head) 方法中，添加到 Arena 的双向链表中。详细解析，见「2.3.1

和 `smallSubpagePools` 属性，分别表示 **tiny** 和 **small** 类型的 PoolSubpage

```
 */
private final PoolSubpage<T>[] tinySubpagePools;
/**
 * small 类型的 SubpagePools 数组
 *
 * 数组的每个元素，都是双向链表
```

```
     */
    private final PoolSubpage<T>[] smallSubpagePools;
```

- 数组的每个元素，通过 `prev` 和 `next` 属性，形成**双向**链表。并且，每个元素，表示对应的 Subpage 内存规格的**双向**链表，例如： `tinySubpagePools[0]` 表示 `16B` ， `tinySubpagePools[1]` 表示 `32B` 。
- 通过 `tinySubpagePools` 和 `smallSubpagePools` 属性，可以从中查找，是否已经有符合分配内存规格的 Subpage 节点可分配。
- 初始时，每个双向链表，会创建对应的 `head` 节点，代码如下：

```
// PoolArena.java

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

- 比较神奇的是， `head` 的上下节点都是**自己**。也就说，这是个双向环形(循环)链表。

2.3.1 addToPool

`#addToPool(PoolSubpage<T> head)` 方法中，添加到 Arena 的双向链表中。代码如下：

```
private void addToPool(PoolSubpage<T> head) {
    assert prev == null && next == null;
    // 将当前节点，插入到 head 和 head.next 中间
    prev = head;
    next = head.next;
    next.prev = this;
    head.next = this;
```

文章目录

ext 中间。如下图所示：

	A	B	C	D	E
1	分类	大小			递增规律
2	Tiny	16B	Subpage	每个 Page 可拆成多个 Subpage	16B 递增
3		32B	Subpage		
4		...	Subpage		
5		480B	Subpage		
6		496B	Subpage		
7	Small	512B	Subpage		2 倍递增
8		1KB	Subpage		
9		...	Subpage		
10		2KB	Subpage		
11		4KB	Subpage		
12	Normal	8KB	Page	每个 Chunk 可拆成多个 Page	2 倍递增
13		16KB	Page		
14		...	Page		
15		8MB	Page		
16		16MB	Page		
17	Huge	> 16MB	Page	一个 Chunk 就是一个 Page	无规律
18		> 16MB	Page		
19		> 16MB	Page		
20					

- 注意，是在 `head` 和 `head.next` **中间**插入节点噢。

2.3.2 removeFromPool

`#removeFromPool()` 方法中，从双向链表中移除。代码如下：

```
private void removeFromPool() {
    assert prev != null && next != null;
```

文章目录

内存块，并返回该内存块的位置 `handle` 。代码如下：

解释好呢？笔者暂时没想好，官方的定义是 "Returns the bitmap index of the subpage allocation." 。

```
1: long allocate() {
2:     // 防御性编程，不存在这种情况。
```

```
3:     if (elemSize == 0) {
4:         return toHandle(0);
5:     }
6:
7:     // 可用数量为 0 ，或者已销毁，返回 -1 ，即不可分配。
8:     if (numAvail == 0 || !doNotDestroy) {
9:         return -1;
10:    }
11:
12:    // 获得下一个可用的 Subpage 在 bitmap 中的总体位置
13:    final int bitmapIdx = getNextAvail();
14:    // 获得下一个可用的 Subpage 在 bitmap 中数组的位置
15:    int q = bitmapIdx >>> 6;
16:    // 获得下一个可用的 Subpage 在 bitmap 中数组的位置的第几 bits
17:    int r = bitmapIdx & 63;
18:    assert (bitmap[q] >>> r & 1) == 0;
19:    // 修改 Subpage 在 bitmap 中不可分配。
20:    bitmap[q] |= 1L << r;
21:
22:    // 可用 Subpage 内存块的计数减一
23:    if (-- numAvail == 0) { // 无可用 Subpage 内存块
24:        // 从双向链表中移除
25:        removeFromPool();
26:    }
27:
28:    // 计算 handle
29:    return toHandle(bitmapIdx);
30: }
```

- 第 2 至 5 行：防御性编程，不存在这种情况。
- 第 7 至 10 行：可用数量为 0 ，或者已销毁，返回 -1 ，即**不可分配。**
- 第 12 至 20 行：分配一个 Subpage 内存块。

方法，获得下一个可用的 Subpage 在 bitmap 中的**总体**位置。详细解析，见

mapIdx / 64 操作，获得下一个可用的 Subpage 在 bitmap 中**数组的位**

apIdx % 64 操作， 获得下一个可用的 Subpage 在 bitmap 中数组的位置

Subpage 在 bitmap 中不可分配。

一。

表示无可用 Subpage 内存块。所以，调用 #removeFromPool() 方法，从

.2 removeFromPool」 。

) 方法，计算 handle 值。代码如下：

```
Idx) {
 (long) bitmapIdx << 32 | memoryMapIdx;
}
```

- 低 32 bits： memoryMapIdx ，可以判断所属 Chunk 的哪个 Page 节点，即 memoryMap[memoryMapIdx] 。
- 高 32 bits： bitmapIdx ，可以判断 Page 节点中的哪个 Subpage 的内存块，即 bitmap[bitmapIdx] 。
  - 那么为什么会有 0x4000000000000000L 呢？因为在 PoolChunk#allocate(int normCapacity) 中：
    - 如果分配的是 Page 内存块，返回的是 memoryMapIdx 。

- 如果分配的是 Subpage 内存块，返回的是 `handle` 。**但但是**，如果说 `bitmapIdx = 0` ，那么没有 `0x4000000000000000L` 情况下，就会和【分配 Page 内存块】冲突。因此，需要有 `0x4000000000000000L` 。
- 因为有了 `0x4000000000000000L` (最高两位为 `01` ，其它位为 `0`)，所以获取 `bitmapIdx` 时，通过 `handle >>> 32 & 0x3FFFFFFF` 操作。使用 `0x3FFFFFFF` (最高两位为 `00` ，其它位为 `1`) 进行消除 `0x4000000000000000L` 带来的影响。

2.5 free

`#free(PoolSubpage<T> head, int bitmapIdx)` 方法，释放指定位置的 Subpage 内存块，并返回当前 Page **是否正在使用中**(`true`)。代码如下：

```
1: boolean free(PoolSubpage<T> head, int bitmapIdx) {
2:     // 防御性编程，不存在这种情况。
3:     if (elemSize == 0) {
4:         return true;
5:     }
6:     // 获得 Subpage 在 bitmap 中数组的位置
7:     int q = bitmapIdx >>> 6;
8:     // 获得 Subpage 在 bitmap 中数组的位置的第几 bits
9:     int r = bitmapIdx & 63;
10:    assert (bitmap[q] >>> r & 1) != 0;
11:    // 修改 Subpage 在 bitmap 中可分配。
12:    bitmap[q] ^= 1L << r;
13:
14:    // 设置下一个可用为当前 Subpage
15:    setNextAvail(bitmapIdx);
16:
17:    // 可用 Subpage 内存块的计数加一
18:    if (numAvail ++ == 0) {
19:        // 添加到 Arena 的双向链表中。
20:        addToPool(head);
                                                    ) {
                                                    , 不进行移除
                                                    (numAvail == maxNumElems)
                                                    this subpage is the only one left in the pool.
36:        // 标记为已销毁
37:        // Remove this subpage from the pool if there are other subpages left in the pool.
38:        doNotDestroy = false;
39:        // 从双向链表中移除
40:        removeFromPool();
41:        return false;
```

文章目录

```
42:     }
43: }
```

- 第 2 至 5 行：防御性编程，不存在这种情况。
- 第 6 至 12 行：释放指定位置的 Subpage 内存块。
  - 第 7 行： `bitmapIdx >>> 6 = bitmapIdx / 64` 操作，获得下一个可用的 Subpage 在 bitmap 中**数组的位置**。
  - 第 9 行： `bitmapIdx & 63 = bitmapIdx % 64` 操作， 获得下一个可用的 Subpage 在 bitmap 中数组的位置的**第几 bit** 。
  - 第 12 行： `^ (1L << r)` 操作，修改 Subpage 在 bitmap 中可分配。
- 第 15 行：调用 `#setNextAvail(int bitmapIdx)` 方法，设置下一个可用为当前 Subpage 的位置。这样，就能避免下次分配 Subpage 时，再去找位置。代码如下：

```
private void setNextAvail(int bitmapIdx) {
    nextAvail = bitmapIdx;
}
```

- 第 18 行：可用 Subpage 内存块的计数加一。
  - 第 20 行：当之前 `numAvail == 0` 时，表示**又有**可用 Subpage 内存块。所以，调用 `#addToPool(PoolSubpage<T> head)` 方法，添加到 Arena 的双向链表中。详细解析，见 「2.3.1 addToPool」 。
  - 第 21 行：返回 `true` ，正在使用中。
- 第 24 至 26 行：返回 `true` ，因为还有其它在使用的 Subpage 内存块。
- 第 27 至 42 行：没有 Subpage 在使用。
  - 第 29 至 34 行：返回 `true` ，因为通过 `prev == next` 可判断，当前节点为双向链表中的唯一节点，不进行移除。也就说，该节点后续，继续使用。
  - 第 36 至 41 行：返回 `false` ，不在使用中。
    - 第 38 行：标记为已销毁。
    - 第 40 行：调用 `#removeFromPool()` 方法，从双向链表中移除。因为此时双向链表中，还有其它节点可使用，**没必要保持多个相同规格的节点**。

d, int bitmapIdx) 方法，需要返回 true 或 false 呢? 胖友再看看就能明白。答案是，如果不再使用，可以将该节点(Page)从 Chunk 中释放，标记

的 Subpage 在 bitmap 中的**总体**位置。代码如下：

已经“缓存”好下一个可用的位置，直接返回即可。

```
    }
    // <2> 寻找下一个 nextAvail
    return findNextAvail();
}
```

- `<1>` 处，如果 `nextAvail` 大于 0 ，意味着已经“缓存”好下一个可用的位置，直接返回即可。

- 获取好后，会将 `nextAvail` 置为 -1 。意味着，下次需要寻找下一个 `nextAvail` 。

- `<2>` 处，调用 `#findNextAvail()` 方法，寻找下一个 `nextAvail` 。代码如下：

```
private int findNextAvail() {
    final long[] bitmap = this.bitmap;
    final int bitmapLength = this.bitmapLength;
    // 循环 bitmap
    for (int i = 0; i < bitmapLength; i ++) {
        long bits = bitmap[i];
        // ~ 操作，如果不等于 0 ，说明有可用的 Subpage
        if (~bits != 0) {
            // 在这 bits 寻找可用 nextAvail
            return findNextAvail0(i, bits);
        }
    }
    // 未找到
    return -1;
}
```

- 代码比较简单，胖友直接看注释。

- 调用 `#findNextAvail0(int i, long bits)` 方法，在这 bits 寻找可用 `nextAvail` 。代码如下：

```
1: private int findNextAvail0(int i, long bits) {
2:     final int maxNumElems = this.maxNumElems;
3:     // 计算基础值，表示在 bitmap 的数组下标
4:     final int baseVal = i << 6; // 相当于 * 64
5:
6:     // 遍历 64 bits
7:     for (int j = 0; j < 64; j ++) {
8:         // 计算当前 bit 是否未分配
9:         if ((bits & 1) == 0) {
           最后一个元素，并没有 64 位，通过 baseVal | j < maxNumElems 来保证
           Val | j;
           umElems) {
           ;
```


  - 第 4 行：计算基础值，表示在 `bitmap` 的数组**下标**。通过 `i << 6 = i * 64` 的计算，我们可以通过 `i >>> 6 = i / 64` 的方式，知道是 `bitmap` 数组的第几个元素。
  - 第 7 行：循环 64 bits 。
    - 第 9 行：`(bits & 1) == 0` 操作，计算当前 bit 是否**未分配**。
      - 第 11 行：`baseVal | j` 操作，使用**低 64 bits** ，表示分配 `bitmap` 数组的元素的**第几 bit** 。

- 第 12 行：可能 `bitmap` 数组的最后一个元素，并没有 64 位，通过 `baseVal | j < maxNumElems` 来保证不超过上限。如果
  - 第 13 行：未超过，返回 `val` 。
  - 第 15 行：超过，结束循环，最终返回 `-1` 。
- 第 19 行：去掉当前 bit 。这样，下次循环就可以判断下一个 bit 是否**未分配。**
- 第 23 行：返回 `-1` ，表示未找到。

2.6 destroy

`#destroy()` 方法，销毁。代码如下：

```
void destroy() {
    if (chunk != null) {
        chunk.destroy();
    }
}
```

2.7 PoolSubpageMetric

`io.netty.buffer.PoolSubpageMetric` ，PoolSubpage Metric 接口。代码如下：

```
public interface PoolSubpageMetric {

    /**
     * Return the number of maximal elements that can be allocated out of the sub-page.
     */
    int maxNumElements();

    /**
     * Return the number of available elements to be allocated.
     */
```



```
    of the elements that will be allocated.

    of this page.
```

，代码如下：

```
@Override
public int maxNumElements() {
    synchronized (chunk.arena) {
        return maxNumElems;
    }
}
```

```
@Override
public int numAvailable() {
    synchronized (chunk.arena) {
        return numAvail;
    }
}

@Override
public int elementSize() {
    synchronized (chunk.arena) {
        return elemSize;
    }
}

@Override
public int pageSize() {
    return pageSize;
}
```

666. 彩蛋

PoolSubpage 相比 PoolChunk 来说，简单好多。嘿嘿。

参考如下文章：

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