我是一段不羁的公告！
记得给芳芳这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro

文章目录

—— Buffer 之 Jemalloc（四）

Jemalloc（二）PoolChunk》，我们看到 PoolChunk 有如下三个属性：

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

- 通过 `prev` 和 `next` 两个属性，形成一个**双向** Chunk 链表 `parent`（PoolChunkList）。

那么为什么需要有 PoolChunkList 这样一个链表呢？直接开始撸代码。

2. PoolChunkList

`io.netty.buffer.PoolChunkList` ，实现 PoolChunkListMetric 接口，负责管理多个 Chunk 的生命周期，**在此基础上对内存分配进行进一步的优化。**

2.1 构造方法

```
/**
 * 所属 PoolArena 对象
 */
private final PoolArena<T> arena;
/**
 * 下一个 PoolChunkList 对象
 */
private final PoolChunkList<T> nextList;
```

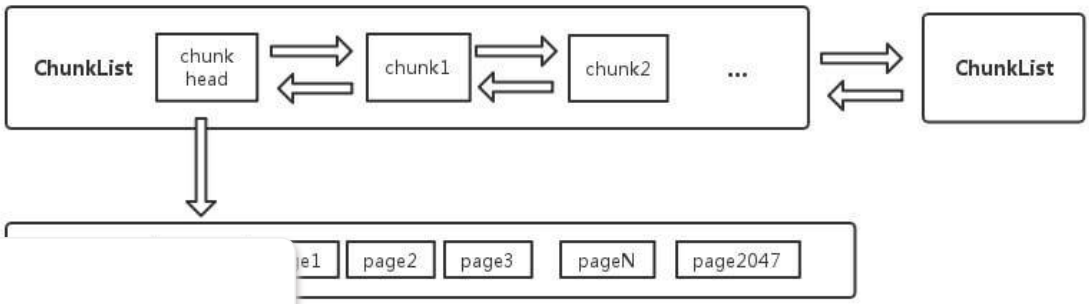
```
/**
 * Chunk 最小内存使用率
 */
private final int minUsage;
/**
 * Chunk 最大内存使用率
 */
```

文章目录


```
nt, int) 方法

n create the linked like list of PoolChunkList in PoolArena constructor
private PoolChunkList<T> prevList;

// TODO: Test if adding padding helps under contention
//private long pad0, pad1, pad2, pad3, pad4, pad5, pad6, pad7;

PoolChunkList(PoolArena<T> arena, PoolChunkList<T> nextList, int minUsage, int maxUsage, int chunkSize
    assert minUsage <= maxUsage;
    this.arena = arena;
    this.nextList = nextList;
    this.minUsage = minUsage;
    this.maxUsage = maxUsage;
    // 计算 maxUsage 属性
    maxCapacity = calculateMaxCapacity(minUsage, chunkSize);
}
```

- `arena` 属性，所属 PoolArena 对象。
- `prevList` + `nextList` 属性，上一个和下一个 PoolChunkList 对象。也就是说，PoolChunkList 除了**自身**有一条双向链表外，PoolChunkList 和 PoolChunkList **之间**也形成了一条双向链表。如下图所示：

> FROM 《深入浅出Netty内存管理 PoolChunkList》



文章目录

双向链表的**头节点**。

olChunkList 管理的 Chunk 们的内存使用率。

Usage 时，从当前 PoolChunkList 节点移除，添加到下一个 PoolChunkList 节
析。

Usage 时，从当前 PoolChunkList 节点移除，添加到上一个 PoolChunkList 节
析。

最大可分配的容量。通过 #calculateMaxCapacity(int minUsage, int
下：

```
acity of a buffer that will ever be possible to allocate out of the {
 * that belong to the {@link PoolChunkList} with the given {@code minUsage} and {@code maxUsage} s
 */
private static int calculateMaxCapacity(int minUsage, int chunkSize) {
    // 计算 minUsage 值
    minUsage = minUsage0(minUsage);

    if (minUsage == 100) {
        // If the minUsage is 100 we can not allocate anything out of this list.
        return 0;
    }

    // Calculate the maximum amount of bytes that can be allocated from a PoolChunk in this PoolCh
    //
    // As an example:
    // - If a PoolChunkList has minUsage == 25 we are allowed to allocate at most 75% of the chunk
    //   this is the maximum amount available in any PoolChunk in this PoolChunkList.
    return  (int) (chunkSize * (100L - minUsage) / 100L);
}

// 保证最小 >= 1
private static int minUsage0(int value) {
    return max(1, value);
}
```

- 为什么使用 `(int) (chunkSize * (100L - minUsage) / 100L)` 来计算呢？因为 Chunk 进入当前 PoolChunkList 节点，意味着 Chunk 内存已经分配了 `minUsage` 比率，所以 Chunk 剩余的容量是 `chunkSize * (100L - minUsage) / 100L` 。😈 是不是豁然开朗噢？！

2.2 allocate

> 随着 Chunk 中 Page 的不断分配和释放，会导致很多碎片内存段，大大增加了之后分配一段连续内存的失败率。针对这种情况，可以把内存使用率较大的 Chunk 放到PoolChunkList 链表更后面。

文章目录

int reqCapacity, int normCapacity) 方法，给 PooledByteBuf 对象分配内存如下：

```
Buf<T> buf, int reqCapacity, int normCapacity) {

nkList 的每个 Chunk 最大可分配的容量
nCapacity > maxCapacity) {
ChunkList is empty or the requested capacity is larger then the capacity
e PoolChunks that are contained in this PoolChunkList.



历的是 ChunkList 的内部双向链表。
 head;;) {

13:         long handle = cur.allocate(normCapacity);
14:         // 分配失败
15:         if (handle < 0) {
16:             // 进入下一节点
17:             cur = cur.next;
18:             // 若下一个节点不存在，返回 false ，结束循环
19:             if (cur == null) {
20:                 return false; // 分配失败
21:             }
22:         // 分配成功
23:         } else {
24:             // 初始化内存块到 PooledByteBuf 对象中
25:             cur.initBuf(buf, handle, reqCapacity);
26:             // 超过当前 ChunkList 管理的 Chunk 的内存使用率上限
27:             if (cur.usage() >= maxUsage) {
28:                 // 从当前 ChunkList 节点移除
29:                 remove(cur);
30:                 // 添加到下一个 ChunkList 节点
31:                 nextList.add(cur);
32:             }
33:             return true; // 分配成功
34:         }
35:     }
36: }
```

- 第 2 至 8 行：双向链表中无 Chunk，或者申请分配的内存超过 ChunkList 的每个 Chunk 最大可分配的容量，返回 `false` ，分配失败。
- 第 11 行：遍历双向链表。**注意，遍历的是 ChunkList 的内部双向链表。**
- 第 13 行：调用 `PoolChunk#allocate(normCapacity)` 方法，分配内存块。这块，可以结合《精尽 Netty 源码解析 —— Buffer 之 Jemalloc（二）PoolChunk》「2.2 allocate」 在复习下。

- 第 15 至 17 行：分配失败，进入下一个节点。
  - 第 18 至 21 行：若下一个节点不存在，返回 `false` ，分配失败。
- 第 22 至 25 行：分配成功，调用 `PooledByteBuf##initBuf(PooledByteBuf<T> buf, long handle, int reqCapacity)` 方法，初始化分配的内存块到 PooledByteBuf 中。这块，可以结合《精尽 Netty 源码解析 —— Buffer 之 Jemalloc（二）PoolChunk》「2.5 initBuf」 在复习下。
  - 第 26 至 32 行：超过当前 ChunkList 管理的 Chunk 的内存使用率上限，从当前 ChunkList 节点移除，并添加到“**下**”一个 ChunkList 节点。

lChunk<T> cur) 方法，解析见「2.4.2 remove」 。

lChunk<T> cur) 方法，解析见「2.4.1 add」 。

力。

handle) 方法，释放 PoolChunk 的指定位置(handle)的内存块。代码如下：

```
nunk, long handle) {
位置( handle )的内存块

理的 Chunk 的内存使用率下限
Usage) {
节点移除

kList 节点
        // Move the PoolChunk down the PoolChunkList linked-list.
        return move0(chunk);
    }
    // 释放成功
    return true;
}
```

- 第 3 行：调用 `PoolChunk#free(long handle)` 方法，释放指定位置的内存块。这块，可以结合《精尽 Netty 源码解析 —— Buffer 之 Jemalloc（二）PoolChunk》「2.3 free」 在复习下。
- 第 5 行：小于当前 ChunkList 管理的 Chunk 的内存使用率下限：
  - 第 7 行：调用 `#remove(PoolChunk<T> cur)` 方法，从当前 ChunkList 节点移除。
  - 第 10 行：调用 `#move(PoolChunk<T> chunk)` 方法， 添加到“上”一个 ChunkList 节点。详细解析，见「2.4.3 move」 。
- 第 13 行：返回 `true` ，释放成功。

2.4 双向链表操作

2.4.1 add

`#add(PoolChunk<T> chunk)` 方法，将 PoolChunk 添加到 ChunkList 节点中。代码如下：

```
1: void add(PoolChunk<T> chunk) {
2:     // 超过当前 ChunkList 管理的 Chunk 的内存使用率上限，继续递归到下一个 ChunkList 节点进行添加。
3:     if (chunk.usage() >= maxUsage) {
4:         nextList.add(chunk);
5:         return;
6:     }
```

文章目录

理的 Chunk 的内存使用率上限，调用 `nextList` 的 `#add(PoolChunk<T>` hunkList 节点进行添加。

> chunk) 方法，执行真正的添加。代码如下：

```
 to this {@link PoolChunkList}.

 {

节点

        head = chunk;
        chunk.prev = null;
        chunk.next = null;
    // <2> 有头节点，自己成为头节点，原头节点成为自己的下一个节点
    } else {
        chunk.prev = null;
        chunk.next = head;
        head.prev = chunk;
        head = chunk;
    }
}
```

- `<1>` 处，比较好理解，胖友自己看。
- `<2>` 处，因为 `chunk` **新**进入下一个 ChunkList 节点，一般来说，内存使用率相对较低，分配内存块成功率相对较高，所以变成新的首节点。

2.4.2 remove

`#remove(PoolChunk<T> chunk)` 方法，从当前 ChunkList 节点移除。代码如下：

```
private void remove(PoolChunk<T> cur) {
    // 当前节点为首节点，将下一个节点设置为头节点
    if (cur == head) {
        head = cur.next;
        if (head != null) {
            head.prev = null;
        }
    // 当前节点非首节点，将节点的上一个节点指向节点的下一个节点
    } else {
        PoolChunk<T> next = cur.next;
        cur.prev.next = next;
        if (next != null) {
```

```
            next.prev = cur.prev;
        }
    }
}
```

- 代码比较简单，胖友自己研究。

文章目录

添加到“上”一个 ChunkList 节点。代码如下：

```
k} down the {@link PoolChunkList} linked-list so it will end up in the
t has the correct minUsage / maxUsage in respect to {@link PoolChunk#us

unk<T> chunk) {
maxUsage;

理的 Chunk 的内存使用率下限，继续递归到上一个 ChunkList 节点进行添加。
Usage) {
nk down the PoolChunkList linked-list.
;

 9:
10:     // 执行真正的添加
11:     // PoolChunk fits into this PoolChunkList, adding it here.
12:     add0(chunk);
13:     return true;
14: }
```

- 第 4 至 8 行：小于当前 ChunkList 管理的 Chunk 的内存使用率下限，调用 `#move0(PoolChunk<T> chunk)` 方法，继续递归到上一个 ChunkList 节点进行添加。代码如下：

```
private boolean move(PoolChunk<T> chunk) {
    assert chunk.usage() < maxUsage;

    // 小于当前 ChunkList 管理的 Chunk 的内存使用率下限，继续递归到上一个 ChunkList 节点进行添加。
    if (chunk.usage() < minUsage) {
        // Move the PoolChunk down the PoolChunkList linked-list.
        return move0(chunk);
    }

    // 执行真正的添加
    // PoolChunk fits into this PoolChunkList, adding it here.
    add0(chunk);
    return true;
}
```

- 第 12 行：调用 `#add0(PoolChunk<T> chunk)` 方法，执行真正的添加。
- 第 13 行：返回 `true` ，移动成功。

2.5 iterator

#iterator() 方法，创建 Iterator 对象。代码如下：

```
private static final Iterator<PoolChunkMetric> EMPTY_METRICS = Collections.<PoolChunkMetric>emptyList(

@Override
 iterator() {

S

;

tor
etrics = new ArrayList<PoolChunkMetric>();
 head;;) {

();
    }
}
```


2.6 destroy

#destroy() 方法，销毁。代码如下：

```
void destroy(PoolArena<T> arena) {
    // 循环，销毁 ChunkList 管理的所有 Chunk
    PoolChunk<T> chunk = head;
    while (chunk != null) {
        arena.destroyChunk(chunk);
        chunk = chunk.next;
    }
    // 置空
    head = null;
}
```

2.7 PoolChunkListMetric

io.netty.buffer.PoolChunkListMetric ，继承 Iterable 接口，PoolChunkList Metric 接口。代码如下：

```
public interface PoolChunkListMetric extends Iterable<PoolChunkMetric> {

    /**
     * Return the minimum usage of the chunk list before which chunks are promoted to the previous lis
     */
    int minUsage();
```

```
    /**
     * Return the maximum usage of the chunk list after which chunks are promoted to the next list.
     */
    int maxUsage();
}
```

文章目录

口的实现，代码如下：

始化

在 PoolChunkArena 中，初始化 PoolChunkList 代码如下：

```
// PoolChunkList 之间的双向链表

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;

/**
 * PoolChunkListMetric 数组
 */
private final List<PoolChunkListMetric> chunkListMetrics;
```

```
 1: protected PoolArena(PooledByteBufAllocator parent, int pageSize,
 2:       int maxOrder, int pageShifts, int chunkSize, int cacheAlignment) {
 3:
 4:     // ... 省略其它无关代码
 5:
 6:     // PoolChunkList 之间的双向链表，初始化
 7:
 8:     q100 = new PoolChunkList<T>(this, null, 100, Integer.MAX_VALUE, chunkSize);
 9:     q075 = new PoolChunkList<T>(this, q100, 75, 100, chunkSize);
10:     q050 = new PoolChunkList<T>(this, q075, 50, 100, chunkSize);
11:     q025 = new PoolChunkList<T>(this, q050, 25, 75, chunkSize);
12:     q000 = new PoolChunkList<T>(this, q025, 1, 50, chunkSize);
13:     qInit = new PoolChunkList<T>(this, q000, Integer.MIN_VALUE, 25, chunkSize);
14:
```

```
15:     q100.prevList(q075);
16:     q075.prevList(q050);
17:     q050.prevList(q025);
18:     q025.prevList(q000);
19:     q000.prevList(null); // 无前置节点
20:     qInit.prevList(qInit); // 前置节点为自己
21:
```

文章目录


```
tric 数组
ic> metrics = new ArrayList<PoolChunkListMetric>(6);
```

```
lections.unmodifiableList(metrics);
```

Init 、 q000 、 q025 、 q050 、 q075 、 q100 有 6 个节点，在【第 6 至 20

```
qInit -> q000 -> q025 -> q050 -> q075 -> q100 -> null

// 逆向
null <- q000 <- q025 <- q050 <- q075 <- q100
qInit <- qInit
```

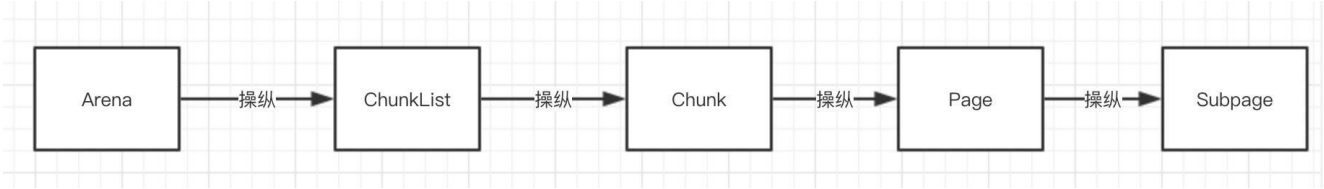
- 比较神奇的是， qInit 指向自己？！ qInit 用途是，新创建的 Chunk 内存块 chunk_new （这只是个代号，方便描述），添加到 qInit 后，不会被释放掉。
  - 为什么不会被释放掉？ qInit.minUsage = Integer.MIN_VALUE ，所以在 PoolChunkList#move(PoolChunk chunk) 方法中， chunk_new 的内存使用率最小值为 0 ，所以肯定不会被释放。
  - 那岂不是 chunk_new 无法被释放？随着 chunk_new 逐渐分配内存，内存使用率到达 25 (qInit.maxUsage)后，会移动到 q000 。再随着 chunk_new 逐渐释放内存，内存使用率降到 0 (q000.minUsage) 后，就可以被释放。
- 当然，如果新创建的 Chunk 内存块 chunk_new **第一次**分配的内存使用率超过 25 (qInit.maxUsage)，不会进入 qInit 中，而是进入后面的 PoolChunkList 节点。

- chunkListMetrics 属性，PoolChunkListMetric 数组。在【第 22 至 30 行】的代码，进行**初始化**。

666. 彩蛋

PoolChunList 相比 PoolSubpage 来说，又又又更加简单啦。

老艿艿整理了下 Arena、ChunkList、Chunk、Page、Subpage 的“操纵”关系如下图：



- 当然，这不是一幅严谨的图，仅仅表达“操纵”的关系。

参考如下文章:

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

文章目录