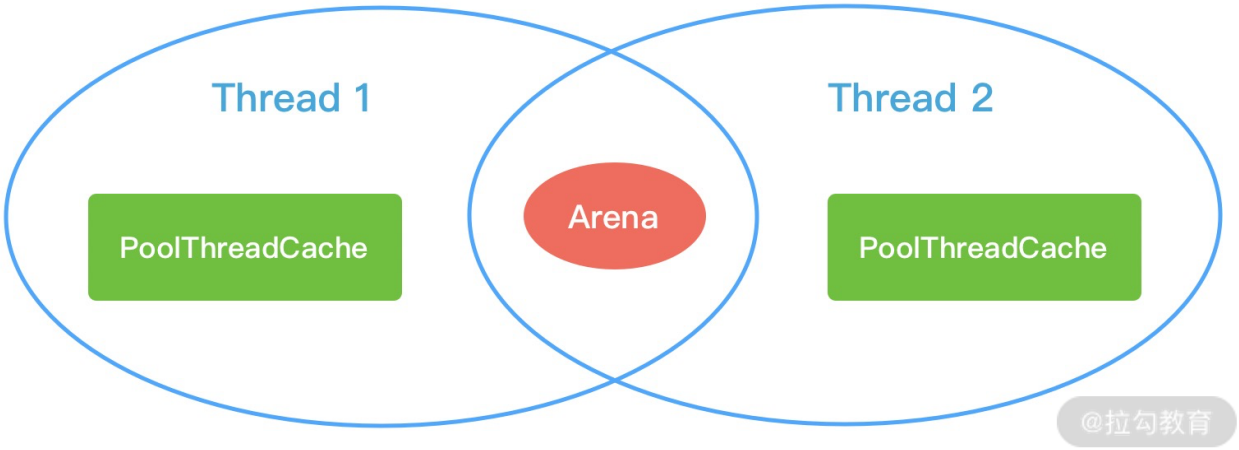
14 举一反三：Netty 高性能内存管理设计（下）

在上一节课，我们学习了 Netty 的内存规格分类以及内存管理的核心组件，今天这节课我们继续介绍 Netty 内存分配与回收的实现原理。有了上节课的基础，相信接下来的学习过程会事半功倍。

本节课会侧重于详细分析不同场景下 Netty 内存分配和回收的实现过程，让你对 Netty 内存池的整体设计有一个更加清晰的认识。

内存分配实现原理

Netty 中负责线程分配的组件有两个：**PoolArena**和**PoolThreadCache**。PoolArena 是多个线程共享的，每个线程会固定绑定一个 PoolArena，PoolThreadCache 是每个线程私有的缓存空间，如下图所示。



在上节课中，我们介绍了 PoolChunk、PoolSubpage、PoolChunkList，它们都是 PoolArena 中所用到的概念。PoolArena 中管理的内存单位为 PoolChunk，每个 PoolChunk 会被划分为 2048 个 8K 的 Page。在申请的内存大于 8K 时，PoolChunk 会以 Page 为单位进行内存分配。当申请的内存大小小于 8K 时，会由 PoolSubpage 管理更小粒度的内存分配。

PoolArena 分配的内存被释放后，不会立即会还给 PoolChunk，而且会缓存在本地私有缓存 PoolThreadCache 中，在下一次进行内存分配时，会优先从 PoolThreadCache 中查找

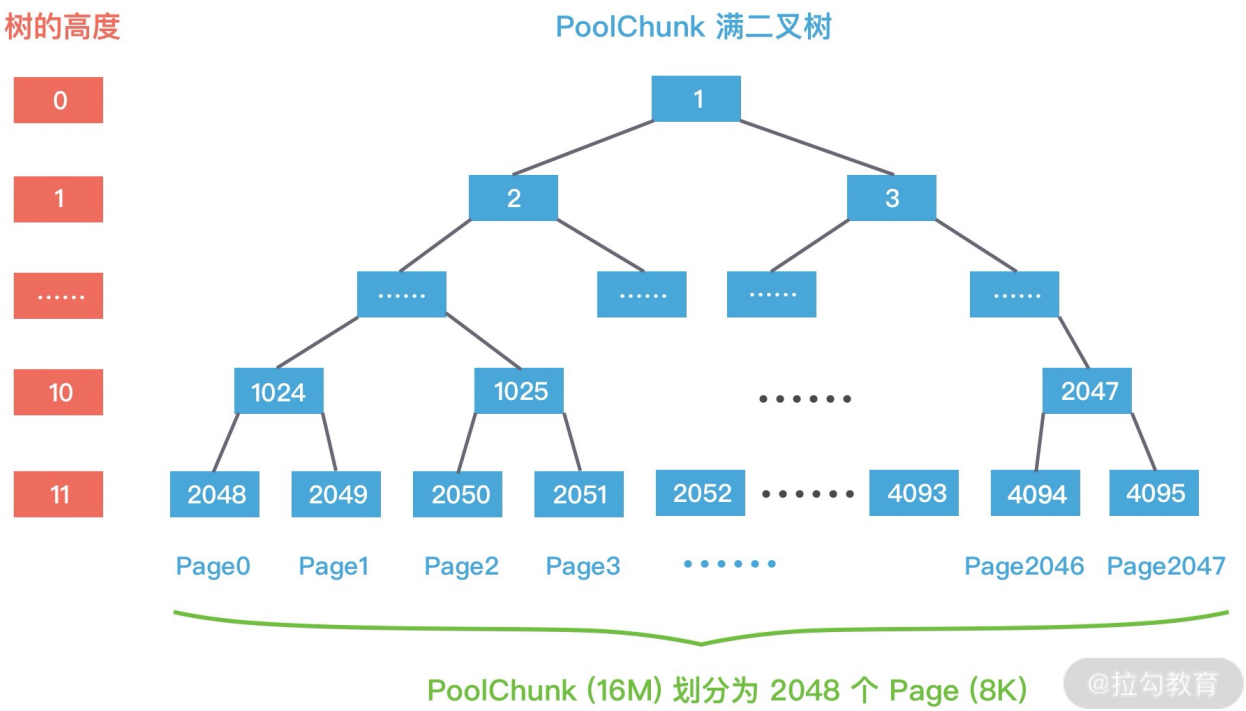
匹配的内存块。

由此可见，Netty 中不同的内存规格采用的分配策略是不同的，我们主要分为以下三个场景逐一进行分析。

- 分配内存大于 8K 时，PoolChunk 中采用的 Page 级别的内存分配策略。
- 分配内存小于 8K 时，由 PoolSubpage 负责管理的内存分配策略。
- 分配内存小于 8K 时，为了提高内存分配效率，由 PoolThreadCache 本地线程缓存提供的内存分配。

PoolChunk 中 Page 级别的内存分配

每个 PoolChunk 默认大小为 16M，PoolChunk 是通过伙伴算法管理多个 Page，每个 PoolChunk 被划分为 2048 个 Page，最终通过一颗满二叉树实现，我们再一起回顾下 PoolChunk 的二叉树结构，如下图所示。



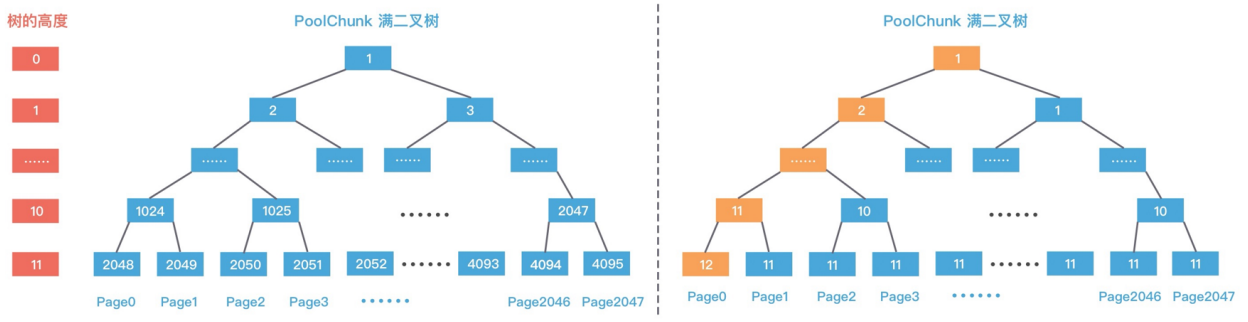
假如用户需要依次申请 8K、16K、8K 的内存，通过这里例子我们详细描述下 PoolChunk 如何分配 Page 级别的内存，方便大家理解伙伴算法的原理。

首先看下分配逻辑 allocateRun 的源码，如下所示。PoolChunk 分配 Page 主要分为三步：首先根据分配内存大小计算二叉树所在节点的高度，然后查找对应高度中是否存在可用节点，如果分配成功则减去已分配的内存大小得到剩余可用空间。

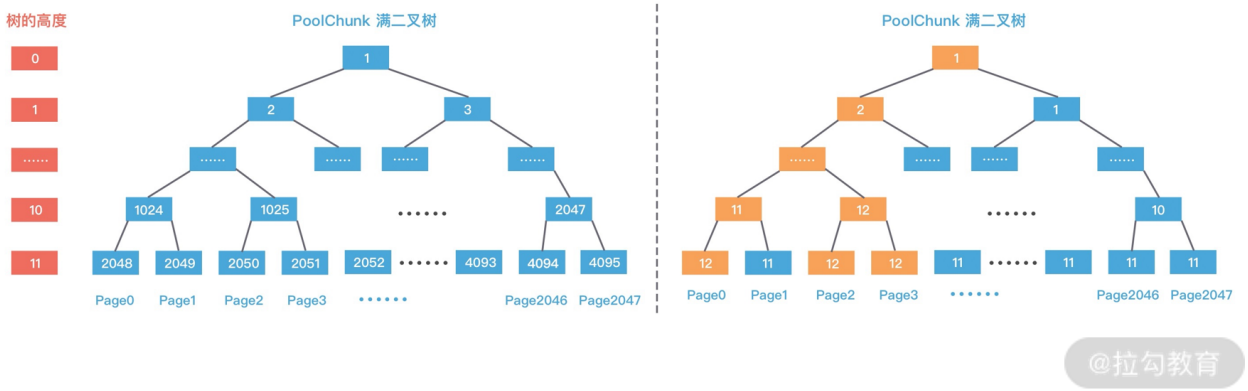
```
private long allocateRun(int normCapacity) {
    // 根据分配内存大小计算二叉树对应的节点高度
    int d = maxOrder - (log2(normCapacity) - pageShifts);
    // 查找对应高度中是否存在可用节点
    int id = allocateNode(d);
    if (id < 0) {
        return id;
    }
    // 减去已分配的内存大小
    freeBytes -= runLength(id);
    return id;
}
```

结合 PoolChunk 的二叉树结构以及 allocateRun 源码我们开始分析模拟的示例：

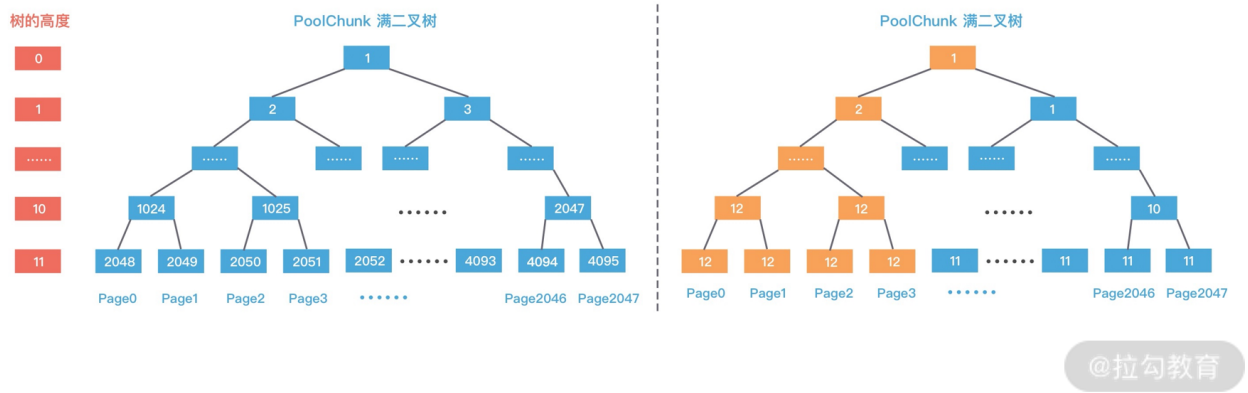
第一次分配 8K 大小的内存时，通过 d = maxOrder - (log2(normCapacity) - pageShifts) 计算得到二叉树所在节点高度为 11，其中 maxOrder 为二叉树的最大高度，normCapacity 为 8K，pageShifts 默认值为 13，因为只有当申请内存大小大于 2^13 = 8K 时才会使用 allocateRun 分配内存。然后从第 11 层查找可用的 Page，下标为 2048 的节点可以被用于分配内存，即 Page[0] 被分配使用，此时赋值 memoryMap[2048] = 12，表示该节点已经不可用，然后递归更新父节点的值，父节点的值取两个子节点的最小值，memoryMap[1024] = 11，memoryMap[512] = 10，以此类推直至 memoryMap[1] = 1，更新后的二叉树分配结果如下图所示。



第二次分配 16K 大小内存时，计算得到所需节点的高度为 10。此时 1024 节点已经分配了一个 8K 内存，不再满足条件，继续寻找到 1025 节点。1025 节点并未使用过，满足分配条件，于是将 1025 节点的两个子节点 2050 和 2051 全部分配出去，并赋值 memoryMap[2050] = 12，memoryMap[2051] = 12，再次递归更新父节点的值，更新后的二叉树分配结果如下图所示。



第三次再次分配 8K 大小的内存时，依然从二叉树第 11 层开始查找，2048 已经被使用，2049 可以被分配，赋值 memoryMap[2049] = 12，并递归更新父节点值，memoryMap[1024] = 12，memoryMap[512] = 12，以此类推直至 memoryMap[1] = 1，最终的二叉树分配结果如下图所示。



至此，PoolChunk 中 Page 级别的内存分配已经介绍完了，可以看出伙伴算法尽可能保证了分配内存地址的连续性，有效地降低了内存碎片。

Subpage 级别的内存分配

为了提高内存分配的利用率，在分配小于 8K 的内存时，PoolChunk 不在分配单独的 Page，而是将 Page 划分为更小的内存块，由 PoolSubpage 进行管理。

首先我们看下 PoolSubpage 的创建过程，由于分配的内存小于 8K，所以走到了 allocateSubpage 源码中：

```
private long allocateSubpage(int normCapacity) {
    // 根据内存大小找到 PoolArena 中 subpage 数组对应的头结点
    PoolSubpage<T> head = arena.findSubpagePoolHead(normCapacity);
    int d = maxOrder; // 因为分配内存小于 8K，所以从满二叉树最底层开始查找
    synchronized (head) {
        int id = allocateNode(d); // 在满二叉树中找到一个可用的节点
        if (id < 0) {
            return id;
        }
        final PoolSubpage<T>[] subpages = this.subpages; // 记录哪些 Page 被转化为 Su
        final int pageSize = this.pageSize;
        freeBytes -= pageSize;
        int subpageIdx = subpageIdx(id); // pageId 到 subpageId 的转化，例如 pageId=
        PoolSubpage<T> subpage = subpages[subpageIdx];
        if (subpage == null) {
            // 创建 PoolSubpage，并切分为相同大小的子内存块，然后加入 PoolArena 对应的双
            subpage = new PoolSubpage<T>(head, this, id, runOffset(id), pageSize, n
            subpages[subpageIdx] = subpage;
        } else {
            subpage.init(head, normCapacity);
        }
        return subpage.allocate(); // 执行内存分配并返回内存地址
    }
}
```

假如我们需要分配 20B 大小的内存，一起分析下上述源码的执行过程：

1. 因为 20B 小于 512B，属于 Tiny 场景，按照内存规格的分类 20B 需要向上取整到 32B。
2. 根据内存规格的大小找到 PoolArena 中 tinySubpagePools 数组对应的头结点，32B 对应的 tinySubpagePools[1]。
3. 在满二叉树中寻找可用的节点用于内存分配，因为我们分配的内存小于 8K，所以直接从二叉树的最底层开始查找。假如 2049 节点是可用的，那么返回的 id = 2049。
4. 找到可用节点后，因为 pageIdx 是从叶子节点 2048 开始记录索引，而 subpageIdx 需要从 0 开始的，所以需要将 pageIdx 转化为 subpageIdx，例如 2048 对应的 subpageIdx = 0，2049 对应的 subpageIdx = 1，以此类推。
5. 如果 PoolChunk 中 subpages 数组的 subpageIdx 下标对应的 PoolSubpage 不存在，那么将创建一个新的 PoolSubpage，并将 PoolSubpage 切分为相同大小的子内存块，示例对应的子内存块大小为 32B，最后将新创建的 PoolSubpage 节点与 tinySubpagePools[1] 对应的 head 节点连接成双向链表。
6. 最后 PoolSubpage 执行内存分配并返回内存地址。

接下来我们跟进一下 subpage.allocate() 源码，看下 PoolSubpage 是如何执行内存分配的，源码如下：

```
long allocate() {
    if (elemSize == 0) {
        return toHandle(0);
    }
    if (numAvail == 0 || !doNotDestroy) {
        return -1;
    }
    final int bitmapIdx = getNextAvail(); // 在 bitmap 中找到第一个索引段，然后将该 bi
    int q = bitmapIdx >>> 6; // 定位到 bitmap 的数组下标
    int r = bitmapIdx & 63; // 取到节点对应一个 long 类型中的二进制位
    assert (bitmap[q] >>> r & 1) == 0;
    bitmap[q] |= 1L << r;
    if (-- numAvail == 0) {
        removeFromPool(); // 如果 PoolSubpage 没有可分配的内存块，从 PoolArena 双向链表
```

```
        }
        return toHandle(bitmapIdx);
    }
```

PoolSubpage 通过位图 bitmap 记录每个内存块是否已经被使用。在上述的示例中，8K/32B = 256，因为每个 long 有 64 位，所以需要 256/64 = 4 个 long 类型的即可描述全部的内存块分配状态，因此 bitmap 数组的长度为 4，从 bitmap[0] 开始记录，每分配一个内存块，就会移动到 bitmap[0] 中的下一个二进制位，直至 bitmap[0] 的所有二进制位都赋值为 1，然后继续分配 bitmap[1]，以此类推。当我们使用 2049 节点进行内存分配时，bitmap[0] 中的二进制位如下图所示：

当 bitmap 分成成功后，PoolSubpage 会将可用节点的个数 numAvail 减 1，当 numAvail 降为 0 时，表示 PoolSubpage 已经没有可分配的内存块，此时需要从 PoolArena 中 tinySubpagePools[1] 的双向链表中删除。

至此，整个 PoolChunk 中 Subpage 的内存分配过程已经完成了，可见 PoolChunk 的伙伴算法几乎贯穿了整个流程，位图 bitmap 的设计也是非常巧妙的，不仅节省了内存空间，而且加快了定位内存块的速度。

PoolThreadCache 的内存分配

上节课已经介绍了 PoolThreadCache 的基本概念，我们知道 PoolArena 分配的内存被释放时，Netty 并没有将缓存归还给 PoolChunk，而是使用 PoolThreadCache 缓存起来，当下次有同样规格的内存分配时，直接从 PoolThreadCache 取出使用即可。所以下面我们从 PoolArena#allocate() 的源码中看下 PoolThreadCache 是如何使用的。

```
private void allocate(PoolThreadCache cache, PooledByteBuf<T> buf, final int reqCap
    final int normCapacity = normalizeCapacity(reqCapacity);
    if (isTinyOrSmall(normCapacity)) { // capacity < pageSize
        int tableIdx;
        PoolSubpage<T>[] table;
        boolean tiny = isTiny(normCapacity);
        if (tiny) { // < 512
            if (cache.allocateTiny(this, buf, reqCapacity, normCapacity)) {
                return;
            }
            tableIdx = tinyIdx(normCapacity);
            table = tinySubpagePools;
        } else {
            if (cache.allocateSmall(this, buf, reqCapacity, normCapacity)) {
                return;
            }
            tableIdx = smallIdx(normCapacity);
            table = smallSubpagePools;
        }
        // 省略其他代码
    }
    if (normCapacity <= chunkSize) {
        if (cache.allocateNormal(this, buf, reqCapacity, normCapacity)) {
            return;
        }
        synchronized (this) {
            allocateNormal(buf, reqCapacity, normCapacity);
            ++allocationsNormal;
```

```
        }
    } else {
        allocateHuge(buf, reqCapacity);
    }
}
```

从源码中可以看出在分配 Tiny、Small 和 Normal 类型的内存时，都会尝试先从 PoolThreadCache 中进行分配，源码结构比较清晰，我们整体梳理一遍流程：

1. 对申请的内存大小做向上取整，例如 20B 的内存大小会取整为 32B。
2. 当申请的内存大小小于 8K 时，分为 Tiny 和 Small 两种情况，分别都会优先尝试从 PoolThreadCache 分配内存，如果 PoolThreadCache 分配失败，才会走 PoolArena 的分配流程。
3. 当申请的内存大小大于 8K，但是小于 Chunk 的默认大小 16M，属于 Normal 的内存分配，也会优先尝试从 PoolThreadCache 分配内存，如果 PoolThreadCache 分配失败，才会走 PoolArena 的分配流程。
4. 当申请的内存大小大于 Chunk 的 16M，则不会经过 PoolThreadCache，直接进行分配。

PoolThreadCache 具体分配内存的过程使用到了一个重要的数据结构 MemoryRegionCache，关于 MemoryRegionCache 的概念你可以回顾下上节课的内容，在这里我就不再赘述了。假如我们现在需要分配 32B 大小的堆外内存，会从 MemoryRegionCache 数组 tinySubPageDirectCaches[1] 中取出对应的 MemoryRegionCache 节点，尝试从 MemoryRegionCache 的队列中取出可用的内存块。

内存回收实现原理

通过之前的介绍我们知道，当用户线程释放内存时会将内存块缓存到本地线程的私有缓存 PoolThreadCache 中，这样在下次分配内存时会提高分配效率，但是当内存块被用完一次后，再没有分配需求，那么一直驻留在内存中又会造成浪费。接下来我们就看下 Netty 是如何实现内存释放的呢？直接跟进下 PoolThreadCache 的源码。

```
private boolean allocate(MemoryRegionCache<?> cache, PooledByteBuf buf, int reqCapa
    if (cache == null) {
        return false;
    }
```

```
    // 默认每执行 8192 次 allocate()，就会调用一次 trim() 进行内存整理
    boolean allocated = cache.allocate(buf, reqCapacity);
    if (++ allocations >= freeSweepAllocationThreshold) {
        allocations = 0;
        trim();
    }
    return allocated;
}
void trim() {
    trim(tinySubPageDirectCaches);
    trim(smallSubPageDirectCaches);
    trim(normalDirectCaches);
    trim(tinySubPageHeapCaches);
    trim(smallSubPageHeapCaches);
    trim(normalHeapCaches);
}
```

从源码中可以看出，Netty 记录了 allocate() 的执行次数，默认每执行 8192 次，就会触发 PoolThreadCache 调用一次 trim() 进行内存整理，会对 PoolThreadCache 中维护的六个 MemoryRegionCache 数组分别进行整理。我们继续跟进 trim 的源码，定位到核心逻辑。

```
public final void trim() {
    int free = size - allocations;
    allocations = 0;
    // We not even allocated all the number that are
    if (free > 0) {
        free(free, false);
    }
}
```

```
private int free(int max, boolean finalizer) {
    int numFreed = 0;
    for (; numFreed < max; numFreed++) {
        Entry<T> entry = queue.poll();
        if (entry != null) {
            freeEntry(entry, finalizer);
        } else {
            // all cleared
            return numFreed;
        }
    }
    return numFreed;
}
```

通过 size - allocations 衡量内存分配执行的频繁程度，其中 size 为该 MemoryRegionCache 对应的内存规格大小，size 为固定值，例如 Tiny 类型默认为 512。allocations 表示 MemoryRegionCache 距离上一次内存整理已经发生了多少次 allocate 调用，当调用次数小于 size 时，表示 MemoryRegionCache 中缓存的内存块并不常用，从队列中取出内存块依次释放。

此外 Netty 在线程退出的时候还会回收该线程的所有内存，PoolThreadCache 重载了 finalize() 方法，在销毁前执行缓存回收的逻辑，对应源码如下：

```
@Override
protected void finalize() throws Throwable {
    try {
        super.finalize();
    } finally {
        free(true);
    }
}
```

```
void free(boolean finalizer) {
    if (freed.compareAndSet(false, true)) {
        int numFreed = free(tinySubPageDirectCaches, finalizer) +
                free(smallSubPageDirectCaches, finalizer) +
                free(normalDirectCaches, finalizer) +
                free(tinySubPageHeapCaches, finalizer) +
                free(smallSubPageHeapCaches, finalizer) +
                free(normalHeapCaches, finalizer);
        if (numFreed > 0 && logger.isDebugEnabled()) {
            logger.debug("Freed {} thread-local buffer(s) from thread: {}", numFree
                    Thread.currentThread().getName());
        }
        if (directArena != null) {
            directArena.numThreadCaches.getAndDecrement();
        }
        if (heapArena != null) {
            heapArena.numThreadCaches.getAndDecrement();
        }
    }
}
```

线程销毁时 PoolThreadCache 会依次释放所有 MemoryRegionCache 中的内存数据，其中 free 方法的核心逻辑与之前内存整理 trim 中释放内存的过程是一致的，有兴趣的同学可以自行翻阅源码。

到此为止，整个 Netty 内存池的分配和释放原理我们已经分析完了，其中巧妙的设计思路以及源码细节的实现，都是非常值得我们学习的宝贵资源。

总结

最后，我们对 Netty 内存池的设计思想做一个知识点总结：

- 分四种内存规格管理内存，分别为 Tiny、Samll、Normal、Huge，PoolChunk 负责管理 8K 以上的内存分配，PoolSubpage 用于管理 8K 以下的内存分配。当申请内存大于 16M 时，不会经过内存池，直接分配。
- 设计了本地线程缓存机制 PoolThreadCache，用于提升内存分配时的并发性能。用于申请 Tiny、Samll、Normal 三种类型的内存时，会优先尝试从 PoolThreadCache 中分配。
- PoolChunk 使用伙伴算法管理 Page，以二叉树的数据结构实现，是整个内存池分配的核心所在。
- 每调用 PoolThreadCache 的 allocate() 方法到一定次数，会触发检查 PoolThreadCache 中缓存的使用频率，使用频率较低的内存块会被释放。
- 线程退出时，Netty 会回收该线程对应的所有内存。

Netty 中引入类似 jemalloc 的内存池管理技术可以说是一大突破，将 Netty 的性能又提升了一个台阶，而这种思想不仅可以用于 Netty，在很对缓存的场景下都可以借鉴学习，希望这些优秀的设计思想能够对你有所帮助，在实际工作中学以致用。

上一页 下一页