- NETTY

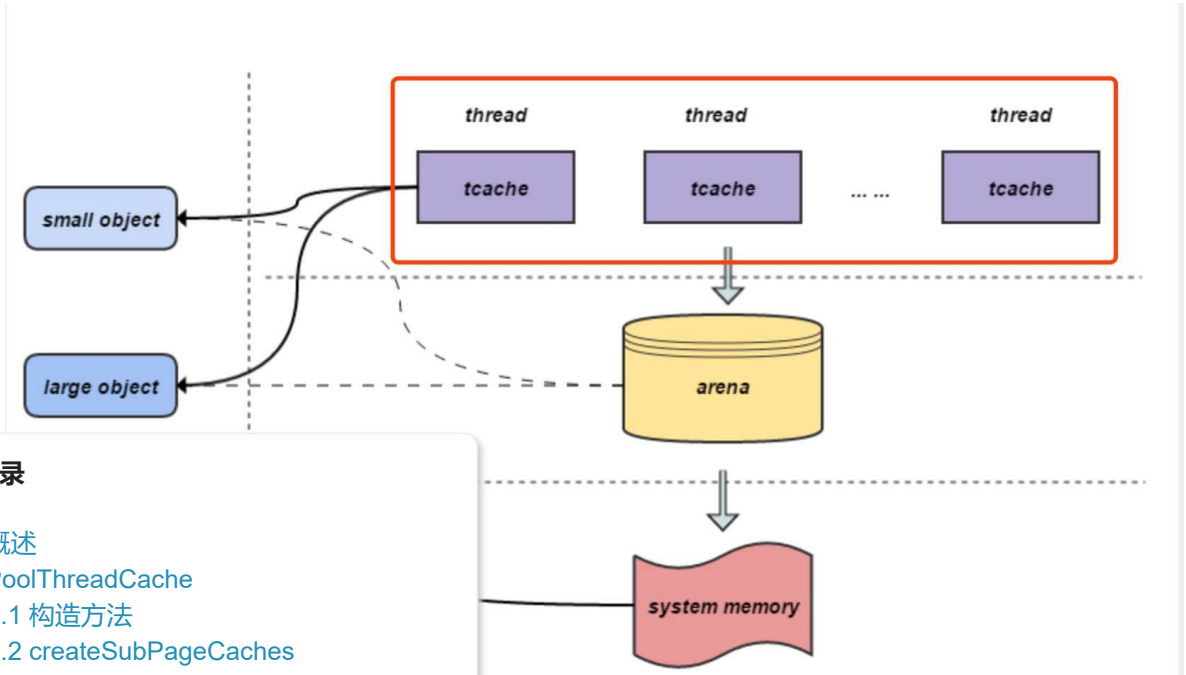
精尽 Netty 源码解析 —— Buffer 之 Jemalloc（六）PoolThreadCache

1. 概述

在《精尽 Netty 源码解析 —— Buffer 之 Jemalloc（五）PoolArena》一文中，我们看到 PoolArena 在分配(`#allocate(...)`)和释放(`#free(...)`)内存的过程中，无可避免会出现 `synchronized` 的身影。虽然锁的粒度不是很大，但是如果一个 PoolArena 如果被**多个**线程引用，带来的线程锁的同步和竞争。并且，如果在锁竞争的过程中，申请 Direct ByteBuffer ，那么带来的线程等待就可能是**几百毫秒**的时间。

那么该如何解决呢？如下图红框所示：

FROM 《jemalloc源码解析-内存管理》



大体结构

文章目录

而是缓存到 tcache 中。
到，再从 Chunk 中分配。

Jemalloc tcache 的实现类，内存分配的线程缓存。

2.1 构造方法

```
/**
 * 对应的 Heap PoolArena 对象
 */
final PoolArena<byte[]> heapArena;
/**
 * 对应的 Direct PoolArena 对象
 */
final PoolArena<ByteBuffer> directArena;

// Hold the caches for the different size classes, which are tiny, small and normal.
/**
 * Heap 类型的 tiny Subpage 内存块缓存数组
 */
private final MemoryRegionCache<byte[]>[] tinySubPageHeapCaches;
/**
 * Heap 类型的 small Subpage 内存块缓存数组
 */
private final MemoryRegionCache<byte[]>[] smallSubPageHeapCaches;
/**
 * Heap 类型的 normal 内存块缓存数组
 */
private final MemoryRegionCache<byte[]>[] normalHeapCaches;
/**
 * Direct 类型的 tiny Subpage 内存块缓存数组
 */
private final MemoryRegionCache<ByteBuffer>[] tinySubPageDirectCaches;
/**
 * Direct 类型的 small Subpage 内存块缓存数组
 */
```



```
>[] smallSubPageDirectCaches;


>[] normalDirectCaches;

 index of normal caches later

 {@link #normalDirectCaches} 数组中的位置



 {@link #normalHeapCaches} 数组中的位置
```

```
     * 分配次数
     */
    private int allocations;
    /**
     * {@link #allocations} 到达该阀值，释放缓存
     *
     * 默认为 8192 次
     *
     * @see #free()
     */
    private final int freeSweepAllocationThreshold;
```

```
 1: PoolThreadCache(PoolArena<byte[]> heapArena, PoolArena<ByteBuffer> directArena,
 2:                 int tinyCacheSize, int smallCacheSize, int normalCacheSize,
 3:                 int maxCachedBufferCapacity, int freeSweepAllocationThreshold) {
 4:     if (maxCachedBufferCapacity < 0) {
 5:         throw new IllegalArgumentException("maxCachedBufferCapacity: "
 6:                 + maxCachedBufferCapacity + " (expected: >= 0)");
 7:     }
 8:     this.freeSweepAllocationThreshold = freeSweepAllocationThreshold;
 9:     this.heapArena = heapArena;
10:     this.directArena = directArena;
11:
12:     // 初始化 Direct 类型的内存块缓存
13:     if (directArena != null) {
14:         // 创建 tinySubPageDirectCaches
15:         tinySubPageDirectCaches = createSubPageCaches(tinyCacheSize, PoolArena.numTinySubpagePool
16:         // 创建 smallSubPageDirectCaches
17:         smallSubPageDirectCaches = createSubPageCaches(smallCacheSize, directArena.numSmallSubpag
18:
19:         // 计算 numShiftsNormalDirect
20:         numShiftsNormalDirect = log2(directArena.pageSize);
21:         // 创建 normalDirectCaches
```



```
rmalCaches(normalCacheSize, maxCachedBufferCapacity, directA

计数
etAndIncrement();

d so just null out all caches
l;
ll;



上面部分。

heap allocations
eSubPageCaches(tinyCacheSize, PoolArena.numTinySubpagePools,
```

```
teSubPageCaches(smallCacheSize, heapArena.numSmallSubpagePoo

apArena.pageSize);
alCaches(normalCacheSize, maxCachedBufferCapacity, heapArena

AndIncrement();
```

```
43:     } else {
44:         // No heapArea is configured so just null out all caches
45:         tinySubPageHeapCaches = null;
46:         smallSubPageHeapCaches = null;
47:         normalHeapCaches = null;
48:         numShiftsNormalHeap = -1;
49:     }
50:
51:     // 校验参数，保证 PoolThreadCache 可缓存内存块。
52:     // Only check if there are caches in use.
53:     if ((tinySubPageDirectCaches != null || smallSubPageDirectCaches != null || normalDirectCache
54:             || tinySubPageHeapCaches != null || smallSubPageHeapCaches != null || normalHeapCache
55:             && freeSweepAllocationThreshold < 1) {
56:         throw new IllegalArgumentException("freeSweepAllocationThreshold: " + freeSweepAllocation
57:     }
58: }
```

- 虽然代码比较多，主要分为 Heap 和 Direct 两种内存。
- Direct 相关
  - `directArena` 属性，对应的 Heap PoolArena 对象。
  - `tinySubPageDirectCaches` 属性，Direct 类型的 tiny Subpage 内存块缓存数组。
    - 默认情况下，数组大小为 512 。
    - 在【第 15 行】的代码，调用 `#createSubPageCaches(int cacheSize, int numCaches, SizeClass sizeClass)` 方法，创建 MemoryRegionCache 数组。详细解析，见「2.2 createSubPageCaches」。
  - `smallSubPageDirectCaches` 属性，Direct 类型的 small Subpage 内存块缓存数组。
    - 默认情况下，数组大小为 256 。
    - 在【第 17 行】的代码，调用 `#createSubPageCaches(int cacheSize, int numCaches, SizeClass sizeClass)` 方法，创建 MemoryRegionCache 数组。详细解析，见「2.2 createSubPageCaches」。
  - `normalDirectCaches` 属性，Direct 类型的 normal Page 内存块缓存数组。
    - 默认情况下，数组大小为 64 。
    - 在【第 22 行】的代码，调用 `#createNormalCaches(int cacheSize, int T> area)` 方法，创建 MemoryRegionCache 数组。详细解

算请求分配的 normal 类型的内存块，在 `normalDirectCaches`

`pageSize)` 方法， `log2(pageSize) = log2(8192) =`

的线程引用计数。通过这样的方式，我们能够知道，一个

记时，该计数器 + 1 。

`allocations` 到达该阀值时，调用 `#free()` 方法，释放缓

。

umCaches, SizeClass sizeClass) 方法，创建 Subpage 内存块缓存

```
fAllocator.DEFAULT_TINY_CACHE_SIZE = 512 , numCaches = PoolA
ufAllocator.DEFAULT_SMALL_CACHE_SIZE = 256 , numCaches = pag
createSubPageCaches(int cacheSize, int numCaches, SizeClass
    if (cacheSize > 0 && numCaches > 0) {
        @SuppressWarnings("unchecked")
        MemoryRegionCache<T>[] cache = new MemoryRegionCache[numCaches];
        for (int i = 0; i < cache.length; i++) {
            // TODO: maybe use cacheSize / cache.length
            cache[i] = new SubPageMemoryRegionCache<T>(cacheSize, sizeClass);
        }
        return cache;
    } else {
        return null;
    }
}
```

- 创建的 Subpage 内存块缓存数组，实际和 PoolArena.tinySubpagePools 和 PoolArena.smallSubpagePools 数组**大小保持一致**。从而实现，相同大小的内存，能对应相同的数组下标。
  - sizeClass = tiny 时，默认 cacheSize = PooledByteBufAllocator.DEFAULT_TINY_CACHE_SIZE = 512 ，numCaches = PoolArena.numTinySubpagePools = 512 >>> 4 = 32 。
  - sizeClass = small 时，默认 cacheSize = PooledByteBufAllocator.DEFAULT_SMALL_CACHE_SIZE = 256 ，numCaches = pageSize - 9 = 13 - 9 = 4 。
- 创建的数组，每个元素的类型为 SubPageMemoryRegionCache 。详细解析，见「3.X.1 SubPageMemoryRegionCache」 。

2.3 createNormalCaches

#createSubPageCaches(int cacheSize, int numCaches, SizeClass sizeClass) 方法，创建 Normal Page 内存块缓存数组。代码如下：

```
// normal 类型，默认 cacheSize = PooledByteBufAllocator.DEFAULT_NORMAL_CACHE_SIZE = 64 , maxCachedBuff
private static <T> MemoryRegionCache<T>[] createNormalCaches(int cacheSize, int maxCachedBufferCapacit
    if (cacheSize > 0 && maxCachedBufferCapacity > 0) {
        // <1> 计算数组大小
 maxCachedBufferCapacity);
ax / area.pageSize) + 1);

 MemoryRegionCache[arraySize];
i++) {
ionCache<T>(cacheSize);
```

mal 内存块的最大容量，避免过大的 Normal 内存块被缓存，占用过多
= PoolArena.DEFAULT_MAX_CACHED_BUFFER_CAPACITY = 32 *
ize 的计算**数组大小**的结果为 3 。刚好是 cache[0] = 8KB 、
么，如果申请的 Normal 内存块大小为 64KB ，超过了数组大小，所
CachedBufferCapacity 实现最大容量的想法，有点不同。
RegionCache 。详细解析，见「3.X.2 NormalMemoryRegionCache」 。

2.4 cache

```
private MemoryRegionCache<?> cacheForTiny(PoolArena<?> area, int normCapacity) {
    // 获得数组下标
    int idx = PoolArena.tinyIdx(normCapacity);
    if (area.isDirect()) {
        return cache(tinySubPageDirectCaches, idx);
    }
    return cache(tinySubPageHeapCaches, idx);
}

private MemoryRegionCache<?> cacheForSmall(PoolArena<?> area, int normCapacity) {
    // 获得数组下标
    int idx = PoolArena.smallIdx(normCapacity);
    if (area.isDirect()) {
        return cache(smallSubPageDirectCaches, idx);
    }
    return cache(smallSubPageHeapCaches, idx);
}

private MemoryRegionCache<?> cacheForNormal(PoolArena<?> area, int normCapacity) {
    if (area.isDirect()) {
        // 获得数组下标
        int idx = log2(normCapacity >> numShiftsNormalDirect);
        return cache(normalDirectCaches, idx);
    }
    // 获得数组下标
    int idx = log2(normCapacity >> numShiftsNormalHeap);
    return cache(normalHeapCaches, idx);
}
```


RegionCache 对象。通过调用 #cache(MemoryRegionCache<T>[]

```
cache(MemoryRegionCache<T>[] cache, int idx) {

gth - 1) {
```

a<?> area, int normCapacity, SizeClass sizeClass) 方法，支
代码如下：

```
na<?> area, int normCapacity, SizeClass sizeClass) {

        return cacheForNormal(area, normCapacity);
    case Small:
        return cacheForSmall(area, normCapacity);
    case Tiny:
        return cacheForTiny(area, normCapacity);
    default:
        throw new Error();
    }
}
```

2.5 add

#add(PoolArena<?> area, PoolChunk chunk, long handle, int normCapacity, SizeClass sizeClass) 方法，添加内存块到 PoolThreadCache 的指定 MemoryRegionCache 的队列中，进行缓存。并且，返回是否添加成功。代码如下：

```
/**
 * Add {@link PoolChunk} and {@code handle} to the cache if there is enough room.
 * Returns {@code true} if it fit into the cache {@code false} otherwise.
 */
@SuppressWarnings({ "unchecked", "rawtypes" })
boolean add(PoolArena<?> area, PoolChunk chunk, long handle, int normCapacity, SizeClass sizeClass) {
    // 获得对应的 MemoryRegionCache 对象
    MemoryRegionCache<?> cache = cache(area, normCapacity, sizeClass);
    if (cache == null) {
        return false;
    }
    // 添加到 MemoryRegionCache 内存块中
    return cache.add(chunk, handle);
}
```


g handle, int normCapacity, PoolThreadCache cache) 中，
析 —— Buffer 之 Jemalloc（五）PoolArena》 的 「2.6 free」 一起

```
e cache. Returns {@code true} if successful {@code false} ot

oledByteBuf<?> buf, int reqCapacity, int normCapacity) {
mCapacity), buf, reqCapacity);
```

```
he cache. Returns {@code true} if successful {@code false} o

ooledByteBuf<?> buf, int reqCapacity, int normCapacity) {
rmCapacity), buf, reqCapacity);
```

```
/**
 * Try to allocate a small buffer out of the cache. Returns {@code true} if successful {@code false} o
 */
boolean allocateNormal(PoolArena<?> area, PooledByteBuf<?> buf, int reqCapacity, int normCapacity) {
    return allocate(cacheForNormal(area, normCapacity), buf, reqCapacity);
}
```

- 三个方法，从缓存中分别获取不同容量大小的内存块，初始化到 PooledByteBuf 对象中。通过调用 `#allocate(MemoryRegionCache<?> cache, PooledByteBuf buf, int reqCapacity)` 方法，代码如下：

```
 1: private boolean allocate(MemoryRegionCache<?> cache, PooledByteBuf buf, int reqCapacity) {
 2:     if (cache == null) {
 3:         // no cache found so just return false here
 4:         return false;
 5:     }
 6:     // 分配内存块，并初始化到 MemoryRegionCache 中
 7:     boolean allocated = cache.allocate(buf, reqCapacity);
 8:     // 到达阀值，整理缓存
 9:     if (++ allocations >= freeSweepAllocationThreshold) {
10:         allocations = 0;
11:         trim();
12:     }
13:     // 返回是否分配成功
14:     return allocated;
15: }
```

- 第 7 行：调用 `MemoryRegionCache#allocate(buf, reqCapacity)` 方法，从缓存中分配内存块，并初始化到 MemoryRegionCache 中。
- 第 8 至 12 行：增加 `allocations` 计数。若到达阀值(`freeSweepAllocationThreshold`)，重置计数，并调用 `#trim()` 方法，整理缓存。详细解析，见「2.7 trim」。
- 第 14 行：返回是否分配成功。如果从缓存中分配失败，后续就从 PoolArena 中获取内存块。

存块缓存。代码如下：

```
?> cache) {
```

```
<?>[] caches) {
    if (caches == null) {
        return;
    }
    for (MemoryRegionCache<?> c: caches) {
        trim(c);
    }
}

private static void trim(MemoryRegionCache<?> cache) {
    if (cache == null) {
        return;
    }
    cache.trim();
}
```

- 会调用所有 MemoryRegionCache 的 `#trim()` 方法，整理每个内存块缓存。详细解析，见「3.6 trim」。

2.8 finalize

`#finalize()` 方法，对象销毁时，清空缓存等等。代码如下：

```
/// TODO: In the future when we move to Java9+ we should use java.lang.ref.Cleaner.
@Override
protected void finalize() throws Throwable {
    try {
        // <1> 调用父 finalize
        super.finalize();
    } finally {
        // 清空缓存
        free();
    }
}
```



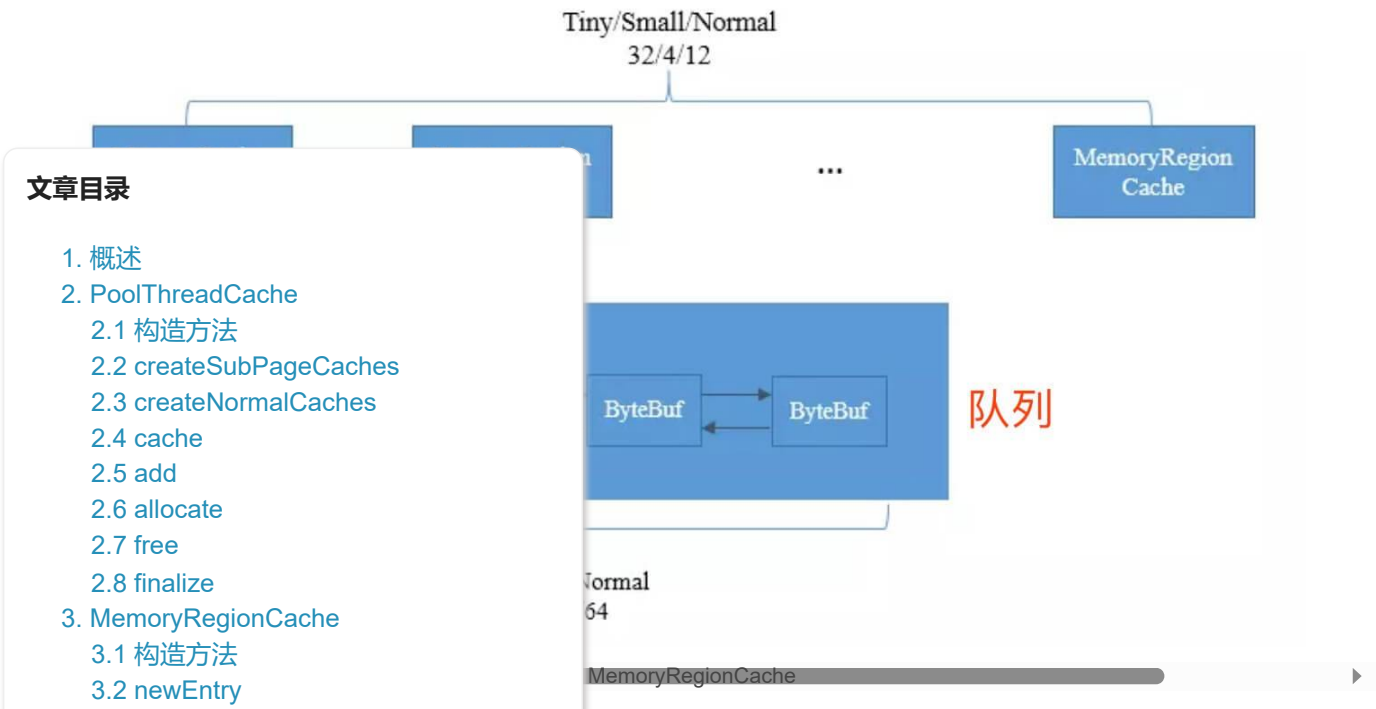
```
es this cache is about to exist to release resources out of
aches) +
 +
led()) {
l buffer(s) from thread: {}", numFreed, Thread.currentThread
```

```
Decrement();

    // <3.2> 减小 heapArena 的线程引用计数
    if (heapArena != null) {
        heapArena.numThreadCaches.getAndDecrement();
    }
}

private static int free(MemoryRegionCache<?>[] caches) {
    if (caches == null) {
        return 0;
    }

    int numFreed = 0;
    for (MemoryRegionCache<?> c: caches) {
        numFreed += free(c);
    }
    return numFreed;
}
```

- 代码比较简单，胖友自己看。主要是 `<1>` 、 `<2>` 、 `<3.1>/<3.2>` 三个点。

3. MemoryRegionCache

MemoryRegionCache ，是 PoolThreadCache 的内部静态类，**内存块缓存**。在其内部，有一个**队列**，存储缓存的内存块。如下图所示：



```
Cache<T> {

     * {@link #queue} 队列大小
     */
    private final int size;
    /**
     * 队列。里面存储内存块
     */
    private final Queue<Entry<T>> queue;
    /**
     * 内存类型
     */
    private final SizeClass sizeClass;
    /**
     * 分配次数计数器
     */
    private int allocations;

    MemoryRegionCache(int size, SizeClass sizeClass) {
        this.size = MathUtil.safeFindNextPositivePowerOfTwo(size);
        queue = PlatformDependent.newFixedMpscQueue(this.size); // <1> MPSC
        this.sizeClass = sizeClass;
    }

    // ... 省略其它方法
}
```

- `sizeClass` 属性，内存类型。
- `queue` 属性，队列，里面存储内存块。每个元素为 Entry 对象，对应一个内存块。代码如下：

```
static final class Entry<T> {

    /**
     * Recycler 处理器，用于回收 Entry 对象
```

```
    }
}
```

- 通过 `chunk` 和 `handle` 属性，可以唯一确认一个内存块。
- `recyclerHandle` 属性，用于回收 Entry 对象，用于 `#recycle()` 方法中。

- `size` 属性，队列大小。
- `allocations` 属性，分配次数计数器。
- 在 `<1>` 处理，我们可以看到创建的 `queue` 属性，类型为 MPSC(Multiple Producer Single Consumer) 队列，即**多个**生产者**单一**消费者。为什么使用 MPSC 队列呢?
  - 多个生产者，指的是多个线程，移除(释放)内存块出队列。
  - 单个消费者，指的是单个线程，添加(缓存)内存块到队列。

3.2 newEntry

`#newEntry(PoolChunk<?> chunk, long handle)` 方法，创建 Entry 对象。代码如下：

```
@SuppressWarnings("rawtypes")
private static Entry newEntry(PoolChunk<?> chunk, long handle) {
    // 从 Recycler 对象中，获得 Entry 对象
    Entry entry = RECYCLER.get();
    // 初始化属性
    entry.chunk = chunk;
    entry.handle = handle;
    return entry;
}

@SuppressWarnings("rawtypes")
private static final Recycler<Entry> RECYCLER = new Recycler<Entry>() {

    @SuppressWarnings("unchecked")
    @Override
    protected Entry newObject(Handle<Entry> handle) {
        return new Entry(handle); // 创建 Entry 对象
```


法，添加(缓存)内存块到队列，并返回是否添加成功。代码如下：

```
k, long handle) {
```

```
e);

对象

 the chunk, immediately recycle the entry
        entry.recycle();
    }

    return queued; // 是否添加成功
}
```

3.4 allocate

`#allocate(PooledByteBuf<T> buf, int reqCapacity)` 方法，从队列中获取缓存的内存块，初始化到 PooledByteBuf 对象中，并返回是否分配成功。代码如下：

```
/**
 * Allocate something out of the cache if possible and remove the entry from the cache.
 */
public final boolean allocate(PooledByteBuf<T> buf, int reqCapacity) {
    // 获取并移除队列首个 Entry 对象
    Entry<T> entry = queue.poll();
    // 获取失败，返回 false
    if (entry == null) {
        return false;
    }
    // <1> 初始化内存块到 PooledByteBuf 对象中
    initBuf(entry.chunk, entry.handle, buf, reqCapacity);
    // 回收 Entry 对象
    entry.recycle();

    // 增加 allocations 计数。因为分配总是在相同线程，所以不需要考虑线程安全的问题
    // allocations is not thread-safe which is fine as this is only called from the same thread all ti
    ++ allocations;
    return true; // 返回 true ，分配成功
}
```



```
unk, long handle, PooledByteBuf<T> buf, int
edByteBuf 对象中。代码如下：

 the provided chunk and handle with the capacity restrictio

k<T> chunk, long handle, PooledByteBuf<T> buf, int reqCapa
```

Cache 和 NormalMemoryRegionCache 来实现。并且，这也是

```
 *
 * Clear out this cache and free up all previous cached {@link PoolChunk}s and {@code handle}s.
 */
public final int free() {
    return free(Integer.MAX_VALUE);
}

// 清除队列中的指定数量元素
private int free(int max) {
    int numFreed = 0;
    for (; numFreed < max; numFreed++) {
        // 获取并移除首元素
        Entry<T> entry = queue.poll();
        if (entry != null) {
            // 释放缓存的内存块回 Chunk 中
            freeEntry(entry); <1>
        } else {
            // all cleared
            return numFreed;
        }
    }
    return numFreed;
}
```

- 代码比较简单，胖友自己看注释。
- `<1>` 处， 释放缓存的内存块回 Chunk 中。代码如下：

```
private  void freeEntry(Entry entry) {
    PoolChunk chunk = entry.chunk;
    long handle = entry.handle;
```


`C'ed.

sizeClass);

etty （十） –PoolThreadCache》 文章后，看懂了 `#trim()` 方法。引用

法，当分配操作达到一定阈值（Netty默认
存空间都要被释放，以防止内存泄漏，核心代

```
// 内部类MemoryRegionCache
public final void trim() {
    // allocations 表示已经重新分配出去的ByteBuf个数
    int free = size - allocations;
    allocations = 0;

    // 在一定阈值内还没被分配出去的空间将被释放
    if (free > 0) {
        free(free); // 释放队列中的节点
    }
}
```

> 也就是说，期望一个 MemoryRegionCache 频繁进行回收-分配，这样 `allocations > size` ，将不会释放队列中的任何一个节点表示的内存空间；
>
> 但如果长时间没有分配，则应该释放这一部分空间，防止内存占据过多。Tiny请求缓存512 个节点，由此可知当使用率超过 `512 / 8192 = 6.25%` 时就不会释放节点。

3.X1 SubPageMemoryRegionCache

SubPageMemoryRegionCache ，是 PoolThreadCache 的内部静态类，继承 MemoryRegionCache 抽象类，**Subpage** MemoryRegionCache 实现类。代码如下：


```
 by TINY or SMALL size.

gionCache<T> extends MemoryRegionCache<T> {

eClass sizeClass) {



unk, long handle, PooledByteBuf<T> buf, int reqCapacity) {
teBuf 对象中
le, reqCapacity);
```

NormalMemoryRegionCache ，是 PoolThreadCache 的内部静态类，继承 MemoryRegionCache 抽象类，**Page** MemoryRegionCache 实现类。代码如下：

```
/**
 * Cache used for buffers which are backed by NORMAL size.
 */
private static final class NormalMemoryRegionCache<T> extends MemoryRegionCache<T> {

    NormalMemoryRegionCache(int size) {
        super(size, SizeClass.Normal);
    }

    @Override
    protected void initBuf(PoolChunk<T> chunk, long handle, PooledByteBuf<T> buf, int reqCapacity) {
        // 初始化 Page 内存块到 PooledByteBuf 对象中
        chunk.initBuf(buf, handle, reqCapacity);
    }

}
```

666. 彩蛋

嘿嘿，比想象中简单蛮多的一篇文章。

推荐阅读文章：

- Hypercube 《自顶向下深入分析Netty（十）–PoolThreadCache》