- NETTY

精尽 Netty 源码解析 —— Buffer 之 ByteBufAllocator（三）PooledByteBufAllocator

1. 概述

本文，我们来分享 PooledByteBufAllocator ，基于**内存池**的 ByteBuf 的分配器。而 PooledByteBufAllocator 的内存池，是基于 **Jemalloc** 算法进行分配管理，所以在看本文之前，胖友先跳到《精尽 Netty 源码解析 —— Buffer 之 Jemalloc（一）简介》，将 Jemalloc 相关的**几篇**文章看完，在回到此处。

2. PooledByteBufAllocatorMetric

文章目录

ric ，实现 ByteBufAllocatorMetric 接口，PooledByteBufAllocator

```
Metric implements ByteBufAllocatorMetric {

allocator;

teBufAllocator allocator) {

     */
    public int numHeapArenas() {
        return allocator.numHeapArenas();
    }
    /**
     * Return the number of direct arenas.
     */
    public int numDirectArenas() {
        return allocator.numDirectArenas();
    }

    /**
     * Return a {@link List} of all heap {@link PoolArenaMetric}s that are provided by this pool.
     */
```

```java
    public List<PoolArenaMetric> heapArenas() {
        return allocator.heapArenas();
    }
    /**
     * Return a {@link List} of all direct {@link PoolArenaMetric}s that are provided by this pool.
     */
    public List<PoolArenaMetric> directArenas() {
        return allocator.directArenas();
    }

    /**
     * Return the number of thread local caches used by this {@link PooledByteBufAllocator}.
     */
    public int numThreadLocalCaches() {
        return allocator.numThreadLocalCaches();
    }

    /**
     * Return the size of the tiny cache.
     */
    public int tinyCacheSize() {
        return allocator.tinyCacheSize();
```

文章目录


```java
        return allocator.chunkSize();
    }

    @Override
    public long usedHeapMemory() {
        return allocator.usedHeapMemory();
    }
    @Override
    public long usedDirectMemory() {
        return allocator.usedDirectMemory();
    }

    @Override
    public String toString() {
```

```
        StringBuilder sb = new StringBuilder(256);
        sb.append(StringUtil.simpleClassName(this))
                .append("(usedHeapMemory: ").append(usedHeapMemory())
                .append("; usedDirectMemory: ").append(usedDirectMemory())
                .append("; numHeapArenas: ").append(numHeapArenas())
                .append("; numDirectArenas: ").append(numDirectArenas())
                .append("; tinyCacheSize: ").append(tinyCacheSize())
                .append("; smallCacheSize: ").append(smallCacheSize())
                .append("; normalCacheSize: ").append(normalCacheSize())
                .append("; numThreadLocalCaches: ").append(numThreadLocalCaches())
                .append("; chunkSize: ").append(chunkSize()).append(')');
        return sb.toString();
    }

}
```

- 每个实现方法，都是调用 `allocator` 对应的方法。通过 PooledByteBufAllocatorMetric 的封装，可以统一获得 PooledByteBufAllocator Metric 相关的信息。

3. PooledByteBufAllocator

文章目录

实现 ByteBufAllocatorMetricProvider 接口，实现
teBuf 的分配器。

```
_ARENA;

CT_ARENA;

 *
 * 默认配置，8192 B = 8 KB
 */
private static final int DEFAULT_PAGE_SIZE;
/**
 * {@link PoolChunk} 满二叉树的高度，默认为 11 。
 */
private static final int DEFAULT_MAX_ORDER; // 8192 << 11 = 16 MiB per chunk
/**
 * 默认 {@link PoolThreadCache} 的 tiny 类型的内存块的缓存数量。默认为 512 。
 *
 * @see #tinyCacheSize
 */
private static final int DEFAULT_TINY_CACHE_SIZE;
```

```java
/**
 * 默认 {@link PoolThreadCache} 的 small 类型的内存块的缓存数量。默认为 256 。
 *
 * @see #smallCacheSize
 */
private static final int DEFAULT_SMALL_CACHE_SIZE;
/**
 * 默认 {@link PoolThreadCache} 的 normal 类型的内存块的缓存数量。默认为 64 。
 *
 * @see #normalCacheSize
 */
private static final int DEFAULT_NORMAL_CACHE_SIZE;
/**
 * 默认 {@link PoolThreadCache} 缓存的内存块的最大字节数
 */
private static final int DEFAULT_MAX_CACHED_BUFFER_CAPACITY;
/**
 * 默认 {@link PoolThreadCache}
 */
private static final int DEFAULT_CACHE_TRIM_INTERVAL;
/**
 * 默认是否使用 {@link PoolThreadCache}
CACHE_FOR_ALL_THREADS;
EMORY_CACHE_ALIGNMENT;
4096;
 (int) (((long) Integer.MAX_VALUE + 1) / 2);
    // 初始化 DEFAULT_PAGE_SIZE
    int defaultPageSize = SystemPropertyUtil.getInt("io.netty.allocator.pageSize", 8192);
    Throwable pageSizeFallbackCause = null;
    try {
        validateAndCalculatePageShifts(defaultPageSize);
    } catch (Throwable t) {
        pageSizeFallbackCause = t;
        defaultPageSize = 8192;
    }
    DEFAULT_PAGE_SIZE = defaultPageSize;

    // 初始化 DEFAULT_MAX_ORDER
    int defaultMaxOrder = SystemPropertyUtil.getInt("io.netty.allocator.maxOrder", 11);
    Throwable maxOrderFallbackCause = null;
```

文章目录

```
    try {
        validateAndCalculateChunkSize(DEFAULT_PAGE_SIZE, defaultMaxOrder);
    } catch (Throwable t) {
        maxOrderFallbackCause = t;
        defaultMaxOrder = 11;
    }
    DEFAULT_MAX_ORDER = defaultMaxOrder;

    // Determine reasonable default for nHeapArena and nDirectArena.
    // Assuming each arena has 3 chunks, the pool should not consume more than 50% of max memory.
    final Runtime runtime = Runtime.getRuntime();

    /*
     * We use 2 * available processors by default to reduce contention as we use 2 * available process
     * number of EventLoops in NIO and EPOLL as well. If we choose a smaller number we will run into h
     * allocation and de-allocation needs to be synchronized on the PoolArena.
     *
     * See https://github.com/netty/netty/issues/3888.
     */
    // 默认最小 Arena 个数。为什么这样计算，见上面的英文注释，大体的思路是，一个 EventLoop 一个 Arena ，避免多
    final int defaultMinNumArena = NettyRuntime.availableProcessors() * 2;
    // 初始化默认 Chunk 的内存大小。默认值为 8192 << 11 = 16 MiB per chunk
                                               PAGE_SIZE << DEFAULT_MAX_ORDER;
```

文章目录


```
                                               umHeapArenas",

                                               Arena,
                                               mory() / defaultChunkSize / 2 / 3))); // `/ 2` 是为了不超过内存

                                               ,

                                               umDirectArenas",

                                               Arena,
                                               dent.maxDirectMemory() / defaultChunkSize / 2 / 3)));

    DEFAULT_TINY_CACHE_SIZE = SystemPropertyUtil.getInt("io.netty.allocator.tinyCacheSize", 512);
    // 初始化 DEFAULT_SMALL_CACHE_SIZE
    DEFAULT_SMALL_CACHE_SIZE = SystemPropertyUtil.getInt("io.netty.allocator.smallCacheSize", 256);
    // 初始化 DEFAULT_NORMAL_CACHE_SIZE
    DEFAULT_NORMAL_CACHE_SIZE = SystemPropertyUtil.getInt("io.netty.allocator.normalCacheSize", 64);

    // 初始化 DEFAULT_MAX_CACHED_BUFFER_CAPACITY
    // 32 kb is the default maximum capacity of the cached buffer. Similar to what is explained in
    // 'Scalable memory allocation using jemalloc'
    DEFAULT_MAX_CACHED_BUFFER_CAPACITY = SystemPropertyUtil.getInt("io.netty.allocator.maxCachedBuffer

    // 初始化 DEFAULT_CACHE_TRIM_INTERVAL
    // the number of threshold of allocations when cached entries will be freed up if not frequently u
    DEFAULT_CACHE_TRIM_INTERVAL = SystemPropertyUtil.getInt("io.netty.allocator.cacheTrimInterval", 81
```

```java
    // 初始化 DEFAULT_USE_CACHE_FOR_ALL_THREADS
    DEFAULT_USE_CACHE_FOR_ALL_THREADS = SystemPropertyUtil.getBoolean("io.netty.allocator.useCacheForA

    // 初始化 DEFAULT_DIRECT_MEMORY_CACHE_ALIGNMENT
    DEFAULT_DIRECT_MEMORY_CACHE_ALIGNMENT = SystemPropertyUtil.getInt("io.netty.allocator.directMemory

    // 打印调试日志( 省略... )
}
```

- 静态变量有点多，主要是为 PoolThreadCache 做的**默认**配置项。读过《精尽 Netty 源码解析 —— Buffer 之 Jemalloc（六）PoolThreadCache》的胖友，是不是灰常熟悉。
- 比较有意思的是， `DEFAULT_NUM_HEAP_ARENA` 和 `DEFAULT_NUM_DIRECT_ARENA` 变量的初始化，在 `<1>` 处。
  - 默认情况下，最小值是 `NettyRuntime.availableProcessors() * 2` ，也就是 CPU 线程数。这样的好处是，一个 EventLoop 一个 Arena ，**避免多线程竞争**。更多的讨论，胖友可以看看 https://github.com/netty/netty/issues/3888 。
  - 比较有趣的一段是 `runtime.maxMemory() / defaultChunkSize / 2 / 3` 代码块。其中， `/ 2` 是为了保证 Arena 不超过内存的一半，而 `/ 3` 是为了每个 Arena 有三个 Chunk 。
  - 当然最终取值是上述两值的最小值。所以在推荐上，尽可能配置的内存，能够保证 `defaultMinNumArena` 。因为**要避免多线程竞争**。

文章目录

ts

ize) 方法，校验 `pageSize` 参数，并计算 `pageShift` 值。代码如

```java
geShifts(int pageSize) {

n("pageSize: " + pageSize + " (expected: " + MIN_PAGE_SIZE +

级

n("pageSize: " + pageSize + " (expected: power of 2)");



 know that pageSize is a power of two.
berOfLeadingZeros(pageSize);
}
```

- 默认情况下， `pageSize = 8KB = 8 * 1024= 8096` ， `pageShift = 8192` 。

3.3 validateAndCalculateChunkSize

`#validateAndCalculateChunkSize(int pageSize, int maxOrder)` 方法，校验 `maxOrder` 参数，并计算 `chunkSize` 值。代码如下：

```java
private static int validateAndCalculateChunkSize(int pageSize, int maxOrder) {
    if (maxOrder > 14) {
        throw new IllegalArgumentException("maxOrder: " + maxOrder + " (expected: 0-14)");
```

```
    }

    // 计算 chunkSize
    // Ensure the resulting chunkSize does not overflow.
    int chunkSize = pageSize;
    for (int i = maxOrder; i > 0; i --) {
        if (chunkSize > MAX_CHUNK_SIZE / 2) {
            throw new IllegalArgumentException(String.format(
                    "pageSize (%d) << maxOrder (%d) must not exceed %d", pageSize, maxOrder, MAX_CHUNK_
        }
        chunkSize <<= 1;
    }
    return chunkSize;
}
```

3.4 构造方法

```
/**
 * 单例
 */
r DEFAULT = new PooledByteBufAllocator(PlatformDependent.dire

nas;

ectArenas;

缓存数组的大小

缓存数组的大小

/**
 * {@link PoolThreadCache} 的 normal 内存块缓存数组的大小
 */
private final int normalCacheSize;
/**
 * PoolArenaMetric 数组
 */
private final List<PoolArenaMetric> heapArenaMetrics;
/**
 * PoolArenaMetric 数组
 */
private final List<PoolArenaMetric> directArenaMetrics;
/**
 * 线程变量，用于获得 PoolThreadCache 对象。
```

文章目录

```
 */
private final PoolThreadLocalCache threadCache;
/**
 * Chunk 大小
 */
private final int chunkSize;
/**
 * PooledByteBufAllocatorMetric 对象
 */
private final PooledByteBufAllocatorMetric metric;

public PooledByteBufAllocator() {
    this(false);
}

@SuppressWarnings("deprecation")
public PooledByteBufAllocator(boolean preferDirect) {
    this(preferDirect, DEFAULT_NUM_HEAP_ARENA, DEFAULT_NUM_DIRECT_ARENA, DEFAULT_PAGE_SIZE, DEFAULT_MA
}

@SuppressWarnings("deprecation")
public PooledByteBufAllocator(int nHeapArena, int nDirectArena, int pageSize, int maxOrder) {
                                              pageSize, maxOrder);
```

文章目录


```
teBufAllocator(boolean, int, int, int, int, int, int, int, bo

ferDirect, int nHeapArena, int nDirectArena, int pageSize, in
tArena, pageSize, maxOrder,
ULT_SMALL_CACHE_SIZE, DEFAULT_NORMAL_CACHE_SIZE);

teBufAllocator(boolean, int, int, int, int, int, int, int, bo

public PooledByteBufAllocator(boolean preferDirect, int nHeapArena, int nDirectArena, int pageSize, in
                              int tinyCacheSize, int smallCacheSize, int normalCacheSize) {
    this(preferDirect, nHeapArena, nDirectArena, pageSize, maxOrder, tinyCacheSize, smallCacheSize,
            normalCacheSize, DEFAULT_USE_CACHE_FOR_ALL_THREADS, DEFAULT_DIRECT_MEMORY_CACHE_ALIGNMENT)
}

public PooledByteBufAllocator(boolean preferDirect, int nHeapArena,
                              int nDirectArena, int pageSize, int maxOrder, int tinyCacheSize,
                              int smallCacheSize, int normalCacheSize,
                              boolean useCacheForAllThreads) {
    this(preferDirect, nHeapArena, nDirectArena, pageSize, maxOrder,
            tinyCacheSize, smallCacheSize, normalCacheSize,
            useCacheForAllThreads, DEFAULT_DIRECT_MEMORY_CACHE_ALIGNMENT);
}
```

```java
public PooledByteBufAllocator(boolean preferDirect, int nHeapArena, int nDirectArena, int pageSize, in
                              int tinyCacheSize, int smallCacheSize, int normalCacheSize,
                              boolean useCacheForAllThreads, int directMemoryCacheAlignment) {
    super(preferDirect);
    // 创建 PoolThreadLocalCache 对象
    threadCache = new PoolThreadLocalCache(useCacheForAllThreads);
    this.tinyCacheSize = tinyCacheSize;
    this.smallCacheSize = smallCacheSize;
    this.normalCacheSize = normalCacheSize;
    // 计算 chunkSize
    chunkSize = validateAndCalculateChunkSize(pageSize, maxOrder);

    if (nHeapArena < 0) {
        throw new IllegalArgumentException("nHeapArena: " + nHeapArena + " (expected: >= 0)");
    }
    if (nDirectArena < 0) {
        throw new IllegalArgumentException("nDirectArea: " + nDirectArena + " (expected: >= 0)");
    }

    if (directMemoryCacheAlignment < 0) {
        throw new IllegalArgumentException("directMemoryCacheAlignment: "
                                               ent + " (expected: >= 0)");

                                                !isDirectMemoryCacheAlignmentSupported()) {
                                               n("directMemoryCacheAlignment is not supported");


                                               rectMemoryCacheAlignment) != directMemoryCacheAlignment) {
                                               n("directMemoryCacheAlignment: "
                                               ent + " (expected: power of two)");


                                               PageShifts(pageSize);



                                               rena);

                                               ew ArrayList<PoolArenaMetric>(heapArenas.length);
        // 初始化 heapArenas 和 metrics 数组
        for (int i = 0; i < heapArenas.length; i ++) {
            // 创建 HeapArena 对象
            PoolArena.HeapArena arena = new PoolArena.HeapArena(this,
                    pageSize, maxOrder, pageShifts, chunkSize,
                    directMemoryCacheAlignment);
            heapArenas[i] = arena;
            metrics.add(arena);
        }
        heapArenaMetrics = Collections.unmodifiableList(metrics);
    } else {
        heapArenas = null;
        heapArenaMetrics = Collections.emptyList();
    }
```

文章目录

```
    if (nDirectArena > 0) {
        directArenas = newArenaArray(nDirectArena);
        List<PoolArenaMetric> metrics = new ArrayList<PoolArenaMetric>(directArenas.length);
        for (int i = 0; i < directArenas.length; i ++) {
            PoolArena.DirectArena arena = new PoolArena.DirectArena(
                    this, pageSize, maxOrder, pageShifts, chunkSize, directMemoryCacheAlignment);
            directArenas[i] = arena;
            metrics.add(arena);
        }
        directArenaMetrics = Collections.unmodifiableList(metrics);
    } else {
        directArenas = null;
        directArenaMetrics = Collections.emptyList();
    }
    // 创建 PooledByteBufAllocatorMetric
    metric = new PooledByteBufAllocatorMetric(this);
}
```

- orz 代码比较长，主要是构造方法和校验代码比较长。胖友自己耐心看下。笔者下面只重点讲几个属性。
- DEFAULT **静态**属性，PooledByteBufAllocator 单例。绝绝绝大多数情况下，我们不需要自己创建

文章目录

|即可。

ThreadCache 对象。通过该属性，不同线程虽然使用**相同**的

adCache 对象。关于 PoolThreadLocalCache 的详细解析，见 「4.

lArena 数组。代码如下：

```
enaArray(int size) {
```

axCapacity) 方法，创建 Heap ByteBuf 对象。代码如下：

```
alCapacity, int maxCapacity) {
    // <1> 获得线程的 PoolThreadCache 对象
    PoolThreadCache cache = threadCache.get();
    PoolArena<byte[]> heapArena = cache.heapArena;

    // <2.1> 从 heapArena 中，分配 Heap PooledByteBuf 对象，基于池化
    final ByteBuf buf;
    if (heapArena != null) {
        buf = heapArena.allocate(cache, initialCapacity, maxCapacity);
    // <2.2> 直接创建 Heap ByteBuf 对象，基于非池化
    } else {
        buf = PlatformDependent.hasUnsafe() ?
                new UnpooledUnsafeHeapByteBuf(this, initialCapacity, maxCapacity) :
                new UnpooledHeapByteBuf(this, initialCapacity, maxCapacity);
    }
```

```
    // <3> 将 ByteBuf 装饰成 LeakAware ( 可检测内存泄露 )的 ByteBuf 对象
    return toLeakAwareBuffer(buf);
}
```

- 代码比较易懂，胖友自己看代码注释。

3.6 newDirectBuffer

`#newDirectBuffer(int initialCapacity, int maxCapacity)` 方法，创建 Direct ByteBuf 对象。代码如下：

```
@Override
protected ByteBuf newDirectBuffer(int initialCapacity, int maxCapacity) {
    // <1> 获得线程的 PoolThreadCache 对象
    PoolThreadCache cache = threadCache.get();
    PoolArena<ByteBuffer> directArena = cache.directArena;

    final ByteBuf buf;
    // <2.1> 从 directArena 中，分配 Direct PooledByteBuf 对象，基于池化
    if (directArena != null) {
                                    initialCapacity, maxCapacity);
                                    基于非池化

                                   () ?
                                   feDirectByteBuf(this, initialCapacity, maxCapacity) :
                                   (this, initialCapacity, maxCapacity);


                                   可检测内存泄露 )的 ByteBuf 对象
```

文章目录

再多做哔哔啦，胖友自己感兴趣的话，可以翻翻噢。

4. PoolThreadLocalCache

PoolThreadLocalCache ，是 PooledByteBufAllocator 的内部类。继承 FastThreadLocal 抽象类，PoolThreadCache **ThreadLocal** 类。

4.1 构造方法

```
/**
 * 是否使用缓存
 */
private final boolean useCacheForAllThreads;
```

```java
PoolThreadLocalCache(boolean useCacheForAllThreads) {
    this.useCacheForAllThreads = useCacheForAllThreads;
}
```

4.2 leastUsedArena

`#leastUsedArena(PoolArena<T>[] arenas)` 方法，从 PoolArena 数组中，获取线程使用最少的 PoolArena 对象，基于 `PoolArena.numThreadCaches` 属性。通过这样的方式，尽可能让 PoolArena 平均分布在不同线程，从而尽肯能避免线程的**同步和竞争**问题。代码如下：

```java
private <T> PoolArena<T> leastUsedArena(PoolArena<T>[] arenas) {
    // 一个都没有，返回 null
    if (arenas == null || arenas.length == 0) {
        return null;
    }

    // 获得第零个 PoolArena 对象
    PoolArena<T> minArena = arenas[0];
    // 比较后面的 PoolArena 对象，选择线程使用最少的
    for (int i = 1; i < arenas.length; i++) {
        PoolArena<T> arena = arenas[i];
                                     minArena.numThreadCaches.get()) {
```

文章目录

adCache 对象。代码如下：

```java
itialValue() {
directArena 对象，基于 `PoolArena.numThreadCaches` 属性。
eastUsedArena(heapArenas);
na = leastUsedArena(directArenas);

    // 创建开启缓存的 PoolThreadCache 对象
    Thread current = Thread.currentThread();
    if (useCacheForAllThreads || current instanceof FastThreadLocalThread) {
        return new PoolThreadCache(
                heapArena, directArena, tinyCacheSize, smallCacheSize, normalCacheSize,
                DEFAULT_MAX_CACHED_BUFFER_CAPACITY, DEFAULT_CACHE_TRIM_INTERVAL);
    }

    // 创建不进行缓存的 PoolThreadCache 对象
    // No caching so just use 0 as sizes.
    return new PoolThreadCache(heapArena, directArena, 0, 0, 0, 0, 0);
}
```

4.4 onRemoval

`#onRemoval(PoolThreadCache threadCache)` 方法，释放 PoolThreadCache 对象的缓存。代码如下：

```
@Override
protected void onRemoval(PoolThreadCache threadCache) {
    // 释放缓存
    threadCache.free();
}
```

666. 彩蛋

推荐阅读文章：

- 杨武兵 《netty源码分析系列——PooledByteBuf&PooledByteBufAllocator》
- wojiushimogui 《Netty源码分析：PooledByteBufAllocator》
- RobertoHuang 《死磕Netty源码之内存分配详解(一)(PooledByteBufAllocator)》

文章目录