20 jemalloc内存分配器是什么

更新时间：2020-08-05 09:42:59

> 天才免不了有障碍，因为障碍会创造天才。——罗曼·罗兰

前言

你好，我是彤哥。

前面的章节，我们一起学习了 Netty 中非池化的四种 ByteBuf，今天，我们本来是要学习池化的 ByteBuf 的，但是，直接看着代码讲池化，特别是内存池，可能无法理解，所以，本节，我们先讲讲 Netty 使用的内存分配器 —— jemalloc，理解了底层原理，我们才能如虎添翼，飞速奔跑。

好了，进入今天的学习吧。

jemalloc

基础知识

所谓内存池，是指应用程序向操作系统（或 JVM）申请一块内存，自己管理这一块内存，对象的创建和销毁都从这块内存中分配和回收，这么一块内存就可以称作内存池，对应地，管理这块内存的工具就称作内存分配器。同时，对于申请对象的不同又可以分为堆内存池和直接内存池，如果是向 JVM 申请的内存，那就是堆内存池，如果是向操作系统申请的内存，那就是直接内存池。

那么，有哪些内存分配器呢？

业界比较著名的有三个内存分配器：

1. ptmalloc，Doug Lea 编写的分配器，支持每个线程（per-thread，简称 pt）的 arena，glibc 的默认分配器。

> Doug Lea 大神还有个分配器叫作 dlmalloc，dl 即其名之缩写。

   tcmalloc，Google 的分配器，它加入了线程缓存（thread cache，简称 tc），Google 声称其比 ptmalloc 快 6 倍。

   jemalloc，Jason Evans 的分配器，je 即其名之缩写，借鉴了很 tcmalloc 的优秀设计，声称比 tcmalloc 更快，且 CPU 核数越多优势越大，当然，算法也更复杂。

其它的分配器还有 nedmalloc、Hoard、TLSF 等。

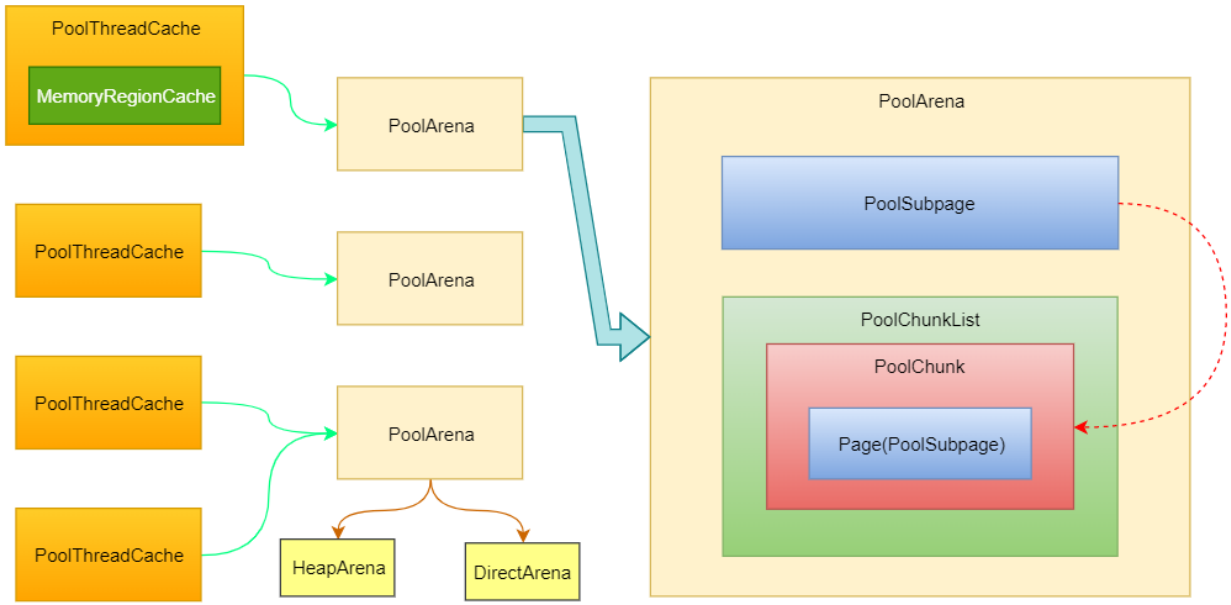
目前，jemalloc 已经广泛运用在 facebook、Mozilla、FreeBSD 等公司的产品上，那么，它有怎样的优势呢？

简单总结一下，主要有三大优势：

- 快速分配和回收
- 内存碎片少
- 支持性能分析

当然了，以上说的都是原生的 jemalloc，我们今天要讲的是 Netty 中的 jemalloc，它是原生 jemalloc 在 Java 中的一种实现方式，并根据 Java 语言自身的特点做了一些删减和优化，关于原生的 jemalloc 可以点击文后的 原生jemalloc相关链接 学习了解。

Netty 中的 jemalloc



我们先从宏观方面对 Netty 中的内存池有个全面的了解，在 Netty 中，主要包含上面这些组件：

- PoolArena
- PoolChunkList
- PoolChunk

- PoolSubpage
- PoolThreadCache

我们一一来分析。

PoolArena

根据内存方式的不同，PoolArena 分成 HeapArena 和 DirectArena 两个子类，在创建 PooledByteBufAllocator 的时候会分别初始化这两种类型的 PoolArena 数组，数组默认大小为核数的 2 倍，同时也会根据可以使用的内存大小动态调整。

```
public class PooledByteBufAllocator extends AbstractByteBufAllocator implements ByteBufAllocatorMetricProvider {
 private final PoolArena<byte[]>[] heapArenas;
   private final PoolArena<ByteBuffer>[] directArenas;
}
```

> 核数也可以通过 JVM 启动参数 `io.netty.availableProcessors` 配置，因为如果使用低版本的 JDK 且部署在 docker 容器中，获取的是主机的核数，而不是 docker 容器分配的核数。

PoolArena 中存储着 2 种类型的数据结构，分别为 2 个 PoolSubPage [] 数组和 6 个 PoolChunkList：

```
abstract class PoolArena<T> implements PoolArenaMetric {
 private final PoolSubpage<T>[] tinySubpagePools;
   private final PoolSubpage<T>[] smallSubpagePools;

   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;
}
```

为什么这么复杂呢？一切都是为了更好地利用内存。

实际上，所有的数据都存储在叫作 PoolChunk 的对象中，默认每个 PoolChunk 可以存储 16MB 的数据（chunkSize），每个 PoolChunk 内部又使用伙伴算法将这 16MB 拆分成 2048 个 Page，每个 Page 的大小（pageSize）为 `16MB/2048=8KB`。

如果分配的内存（规范化后的内存）小于 8KB，则把 Page 拆分成更小的内存块，并使用 PoolSubpage 管理这些更小的内存，每个 Page 的拆分标准根据这个 Page 首次被分配时的请求的大小决定：

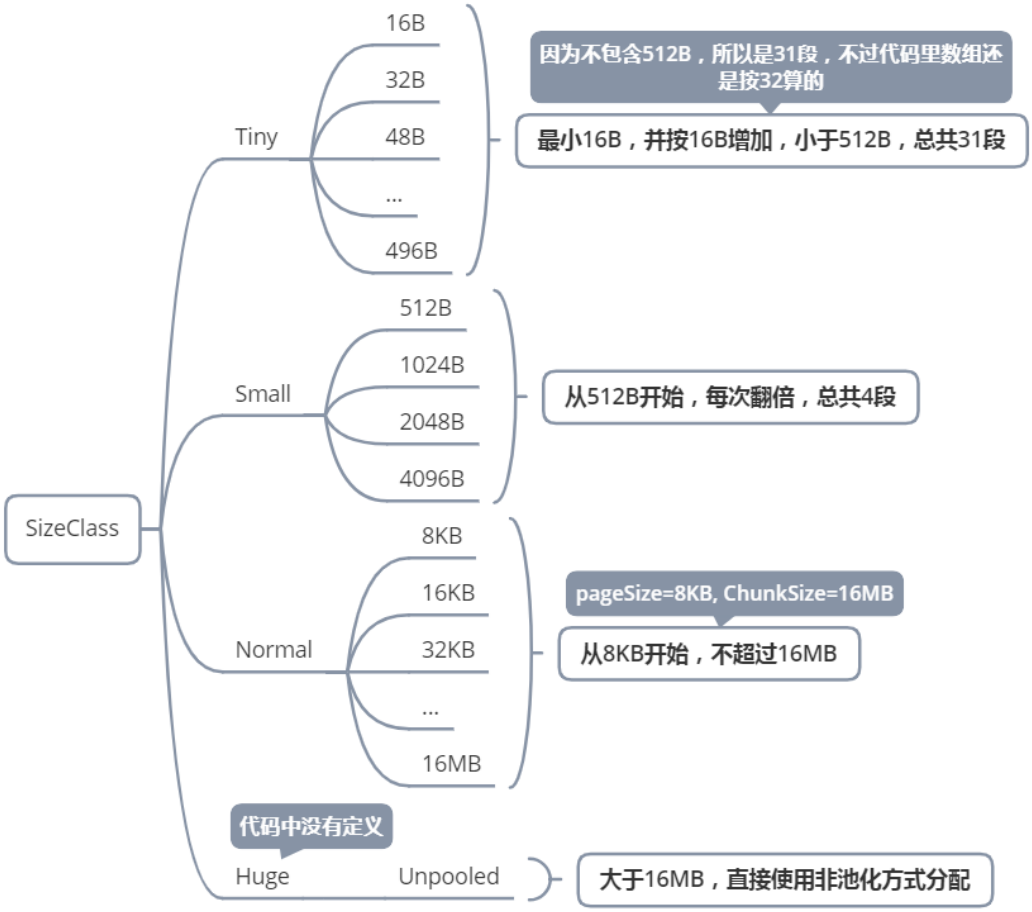
- 如果小于 512B，则按照 16B 规范化，比如请求的大小为 30B，则规范化到 32B，然后 PoolSubpage 中的元素大小就是 32B，那么，这个 Page 就被拆分成了 `8KB/32B=256` 个更小的内存块。
- 如果大于等于 512B，则按照 `512B*(2^n)` 规范化，比如请求的大小为 996B，那就规范化到 1024B，也就是 1KB，然后这个 Page 就被拆分成了 `8KB/1KB=8` 个更小的内存块。

如果分配的内存大于等于 8KB，且小于等于 16MB，则按照 Page 的大小，也就是 8KB，进行规范化，然后再根据伙伴算法的规则进行内存的分配，什么是伙伴算法呢？我们待会讲。

如果分配的内存大于 16MB，则按照非池化的方式分配内存。

所以，为了区分以上几种情况，Netty 中定义了一个 SizeClass 类型的枚举，把这几种情况分别叫作 Tiny、Small、Normal、Huge，其中 Huge 不在这个枚举中。

如果要用一张图来表示的话，我觉得下面这张比较合适：



对于 Tiny 和 Small 类型，Netty 为了快速定位，定义了两个数组放在 PoolArena 中，分别是 tinySubpagePools 和 smallSubpagePools，它们的大小分别为 32 和 4，如果这两个数组对应的位置有值，说明之前出现过相同大小的内存块，那就快速定位到那个 PoolSubpage，使用它直接分配内存，而不用再从头查找，加快分配内存的速度。

前面我们说了，实际上，所有的数据都位于 PoolChunk 中，为了更好地管理这些 PoolChunk，Netty 将它们以双向链表的形式存储在 PoolChunkList 中，同时 PoolChunkList 本身也以双向链表的形式呈现。

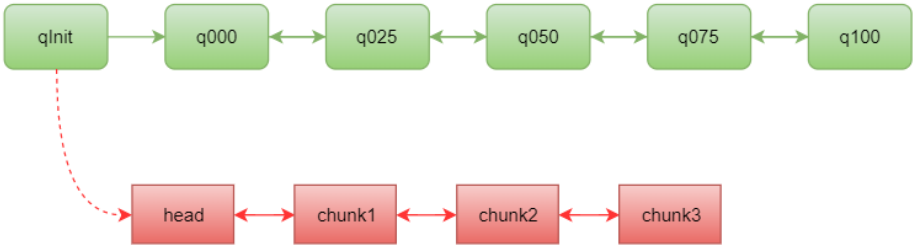
在 PoolArena 中，定义了 6 个 PoolChunkList，分别是 qInit、q000、q025、q050、q075、q100，Netty 根据 PoolChunk 的使用率将它们放到不同类型的 PoolChunkList 中，它们代表的使用率分别为：

- qInit，内存使用率为 Integer.MIN_VALUE ~ 25%，当然不可能有负的使用率，所以最小应该是 0
- q000，内存使用率为 0 ~ 50%
- q025，内存使用率为 25% ~ 75%
- q050，内存使用率为 50% ~ 100%
- q075，内存使用率为 75% ~ 100%
- q100，内存使用率为 100% ~ Integer.MAX_VALUE，当然不可能有超过 100% 的使用率，所以最大应该是 100%

举个例子来说明，比如一个 Chunk 首次分配了大小为 512B 的内存，那么它的内存使用率就是 512B/16MB 不足 1%，向上取整为 1%，初始时放在 qInit 中，当其分配的总内存超过了 4MB 的时候，也就是达到 25% 了，这个 PoolChunk 就被移动到 q000 中，同样地，当其分配的内存超过 8MB 的时候，就移动到了 q025 中。反过来也是一样，当有对象释放内存时，这部分内存又会被回收到 PoolChunk 中待分配，这时候内存使用会降低，当降低到 4MB 时，也就是 q025 的下限，则会将这个 PoolChunk 移动到 q000 中。

PoolChunkList

正如前面所说，PoolChunkList 就是相近内存使用率的 PoolChunk 的集合，这些 PoolChunk 以双链表的形式存储在 PoolChunkList 中，而 PoolChunkList 本身也以双向链表的形式连在一起，为什么要以双向链表的形式存在呢？



其实，这包含两个问题：

1. PoolChunk 以双向链表的形式存在，是为了删除元素（移动 PoolChunk）的时候更快，比如，要删除 chunk2，只要把它的 prev 和 next 连一起就行了，时间复杂度更低；
2. PoolChunkList 以双向链表的形式存在，是为了让 PoolChunk 在 PoolChunkList 之间移动更快，比如，一个 PoolChunk 不管是从 q025 到 q050，还是从 q050 回到 q025，都很快，时间复杂度都很低；

另外，在 Netty 中，当分配内存时，优先从 q050 中寻找合适的 PoolChunk 来分配内存，为什么先从 q050 开始呢？

```
private void allocateNormal(PooledByteBuf<T> buf, int reqCapacity, int normCapacity) {
    if (q050.allocate(buf, reqCapacity, normCapacity)
        || q025.allocate(buf, reqCapacity, normCapacity)
        || q000.allocate(buf, reqCapacity, normCapacity)
        || qInit.allocate(buf, reqCapacity, normCapacity)
        || q075.allocate(buf, reqCapacity, normCapacity)) {
        return;
    }
    // 省略其它代码
}
```

因为 q050 中的 PoolChunk 的内存使用率都比 50% 多一点，这样更容易找到符合条件的 PoolChunk，又不至于使 PoolChunk 的利用率偏低。

我们举个例子，假如从 q075 中先寻找，如果要分配 4M 以上的内存就无法找到合适的 PoolChunk；假如从 q025 中先寻找，可能正好有内存使用率在 25% 以上的 PoolChunk，这时候就直接使用了，那么 q050 中的 PoolChunk 就很难被利用起来，也就是 q050 中的 PoolChunk 的剩余空间很难被利用起来，进而导致整体的利用率偏低，也就是内存碎片会变高。

那么，如果先从 q050 寻找合适的 PoolChunk 呢？这时 q025 和 q075 中的 PoolChunk 可能永远都不会被使用到，不过没关系，对于 q025 中的 PoolChunk 的内存使用率变为 0 的时候，它们自然就被释放了，而 q075 中的 PoolChunk 本身内存使用率就已经很高了，不用到它们反而更好，等它们的内存使用率降低的时候就又回到 q050 中了，此时就又来很容易地被利用起来。

因此，从 q050 开始寻找，能很大程度上增大整体的内存使用率，降低内存碎片的存在。

PoolChunk

前面我们说了，默认地，一个 PoolChunk 可以存储 16MB 的数据，PoolChunk 是真正存储数据的地方，何以见得？

```
final class PoolChunk<T> implements PoolChunkMetric {
    // 数据存储的地方
    final T memory;
    // 满二叉树对应节点是否被分配，数组大小为4096
    private final byte[] memoryMap;
    // 满二叉树原始节点高度，数组大小为4096
    private final byte[] depthMap;
    // 管理更小的内存，数组大小为2048
    private final PoolSubpage<T>[] subpages;
    // 剩余的内存
    private int freeBytes;
    PoolChunk<T> prev;
    PoolChunk<T> next;
}
```

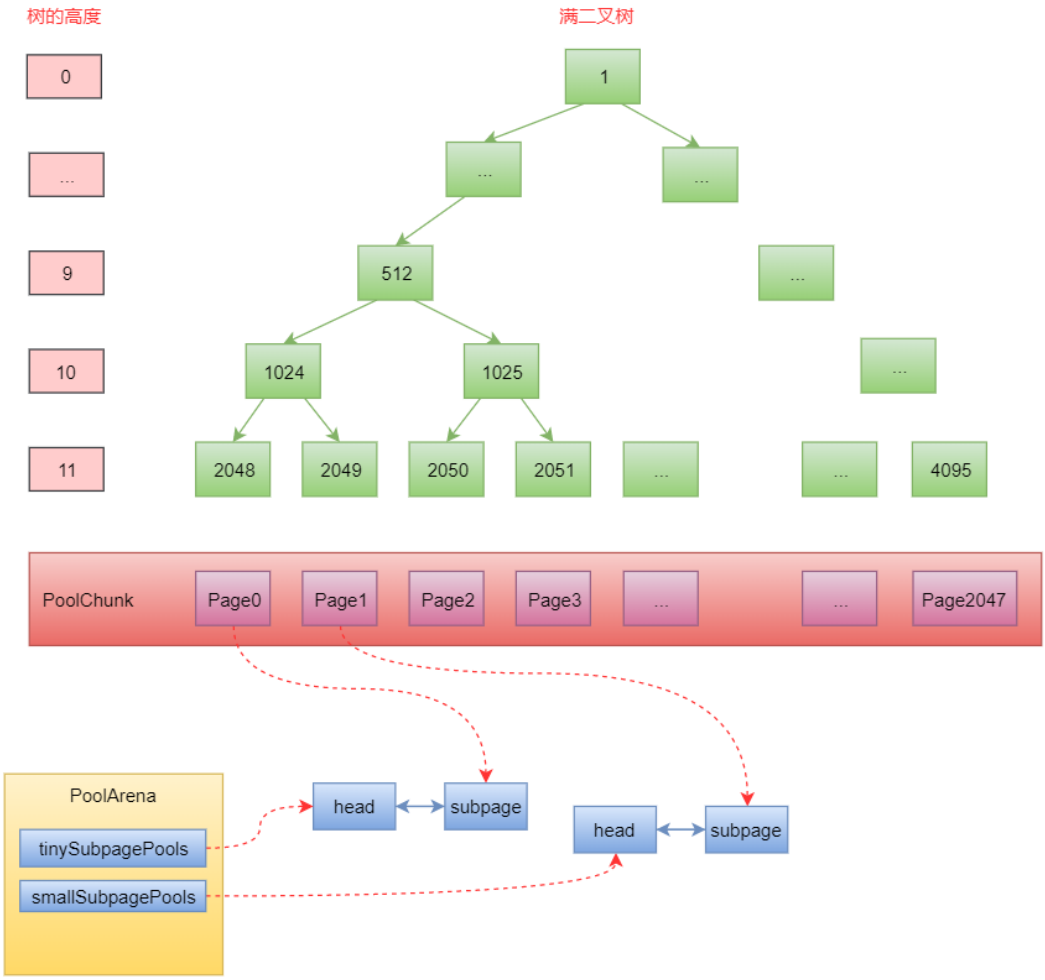
PoolChunk 本身是一个泛型类型，内部保存了一个叫作 memory 的变量，这个 memory 会根据分配的是堆内存还是直接内存而变换类型：

- 对于堆内存，memory 的类型为 byte []
- 对于直接内存，memory 的类型为 ByteBuffer，实际上为 DirectByteBuffer

所有的数据都存储在 memory 中，至于更小粒度的划分，比如 PoolSubpage，它们使用各种偏移量对 memory 进行分段处理，数据本身并不会复制到这些细粒度的类中。

> 在 Netty 中，并没有 PoolPage 或者 Page 这个类，Page 是一种抽象的说法，它表示的是 PoolChunk 中每 8KB 的数据块，它同样使用 PoolSubpage 来表示。

默认地，Netty 使用伙伴算法将 PoolChunk 分成 2048 个 Page，这些 Page 又向上形成一颗满二叉树：



结合上图，我们先来简单介绍一下 PoolChunk 中的几个变量：

- depthMap，保存着满二叉树原始的高度信息，比如 depthMap [1024]=10
- memoryMap，初始值等于 depthMap，随着节点的被分配，它的值会不断变化，更新子节点的值时，会同时更新其父节点的值，其父节点的值等于两个子节点值中的最小者。
- subpages，对应于上图中的 Page0、Page1、…、Page2047，在 Netty 中并没有 Page 的具体代码实现，它同样使用 PoolSubpage 来表示。只有分配的内存小于 8KB，才会使用 PoolSubpage 进行管理，在 PoolSubpage 创建之后，会加入到 PoolArena 中 tinySubpagePools [] 或 smallSubpagePools [] 对应位置的链表中，同时，在 PoolSubpage 代表的内存被分配完之后，会从对应的链表中删除，也就是说，在同一时刻，head 最多只会与一个 PoolSubpage 形成双向链表。
- freeBytes，PoolChunk 中剩余的内存，即可被使用的内存。

> 如果分配的内存大于等于 8KB，由 PoolChunk 自己管理。

为了更好地理解伙伴分配算法，我们来假想一种分配内存的情况，如果分配内存的顺序分别为 8KB、16KB、8KB，则会按以下顺序进行：

- 8KB，符合一个 Page 大小，所以从第 11 层（12-8KB/8KB）寻找节点，这里找到了 2048 这个节点，发现其 memoryMap [2048]=11=depthMap [2048]，可以被分配，然后到其对应的 Page [0] 中分配内存，分配之后将其 memoryMap [2048]=12，memoryMap [1024]=11=（2048 和 2049 中的最小者 11），memoryMap

[512]=10=（1024 和 1025 中的最小者 10），…，memoryMap [1]=1；

- 16KB，符合两个 Page 大小，所以从第 10 层寻找节点（12-16KB/8KB），找到 1024 节点，发现其 memoryMap [1024]=11!=depthMap [1024]，不符合条件，继续寻找到 1025 节点，发现其 memoryMap [1025]=10=depthMap [1025]，符合条件，所以，到其对应的叶子节点 2050/2051 对应的 Page [2]/Page [3] 中分配内存，分配之后 memoryMap [2050]=12，memoryMap [2051]=12，memoryMap [1025]=12=（2050 和 2051 中的最小值 12），memoryMap [512]=11=（1024 和 1025 中的最小者 11），…，memoryMap [1]=1；
- 8KB，符合一个 Page 大小，所以从第 11 层（12-8KB/8KB）寻找节点，2048 已经不符合条件了，所以找到了 2049 这个节点，到其对应的 Page [1] 中分配内存，然后更新 memoryMap [2049]=12，memoryMap [1024]=12=（2048 和 2049 中的最小者 12），memoryMap [512]=12=（1024 和 1025 中的最小者 12），…，memoryMap [1]=1；

至此，三次内存都分配完毕，总共分配了 Page0~Page3 共 4 个 Page，从分配结果也可以看出，使用伙伴分配算法，能极大地保证分配连续的内存空间，并减少内存碎片的诞生。

PoolSubpage

前面我们说过，只有当分配的内存小于一个 Page 大小，即 8KB 时，才会使用 PoolSubpage 来进行管理，那么它是怎么管理的呢？

让我们先来看看它的几个关键字段：

```
final class PoolSubpage<T> implements PoolSubpageMetric {
    // 对应满二叉树中的哪个节点
    private final int memoryMapIdx;
    // 在PoolChunk的memory中的偏移量
    private final int runOffset;
    // 表示每个小块的状态
    private final long[] bitmap;
    // 每个小块（元素）的大小
    int elemSize;
    // 最大的元素个数=8KB/elemSize
    private int maxNumElems;
    // 需要使用到几个long
    private int bitmapLength;
    // 可用的元素个数
    private int numAvail;
    // 双向链表的指针
    // 与PoolArena中的tinySubpagePoos或smallSubpagePools中的元素形成双向链表
    PoolSubpage<T> prev;
    PoolSubpage<T> next;
}
```

elemSize 表示每个元素的大小，这个大小是根据这个 Page 接收到的第一个请求的大小决定的。

比如，首次分配 30B 的内存，则会经历以下几个步骤：

1. 判断小于 512B，按 16B 向上规范化到 32B；
2. 在满二叉树的第 11 层寻找一个可用的节点，假如是 2049，即 memoryMapIdx=2049，它代表的是 Page1，Page1 这个节点在 PoolChunk 中对应到 memory 上的偏移量就是 8192（前面有个 Page0），所以，runOffset=8192；
3. 此时，会把 Page0 按 32B 分成（8KB/32B=256）个小块，所以，elemSize=32B，maxNumElems=256，numAvail=256；
4. 同时，这 256 个小块就需要 256 个 bit（位）来表示其每个小块的状态，也就是需要（256/64=4）个 long 类

型来表示，所以，bitmapLength=4；

5. 然后，把这个 PoolSubpage 与 PoolArena 的 tinySubpagePools [1]（相当于 head）形成双向链表，因为 tinySubpagePools [0] 代表的是 16B 的内存，tinySubpagePools [1] 代表的是 32B 的内存；
6. 当分配完这 32B 之后，可用节点数减一，所以，numAvail=255；

当再次分配规范为 32B 内存的时候，就看 PoolArena 的 tinySubpagePools [1] 的 next 中有没有值，有值，就直接使用其分配内存了，而不用再重新走一遍上面的过程，从而加快分配内存的速度。

PoolThreadCache

前面讲了这么多，分配内存的速度已经足够快了，但是，还可以更快，那就是加入线程缓存 PoolThreadCache，那么，PoolThreadCache 在何时使用呢？

其实，这要结合回收内存一起使用，当回收内存时，先不还给 PoolChunk，而是使用本地线程缓存起来，当下一次再分配同样大小（规范化后的大小）的内存时，先尝试从本地线程缓存里面取，如果取到了就可以直接使用了。

那么，PoolThreadCache 可以缓存哪些类型的缓存呢？

在 Netty 中，除了 Huge，其它类型的内存都可以缓存，即 Tiny、Small、Normal，当然，根据堆内存和直接内存的不同，PoolThreadCache 中又分成了两大类：

```
final class PoolThreadCache {
    // 堆内存的缓存
    private final MemoryRegionCache<byte[]>[] tinySubPageHeapCaches;
    private final MemoryRegionCache<byte[]>[] smallSubPageHeapCaches;
    private final MemoryRegionCache<byte[]>[] normalHeapCaches;
    // 直接内存的缓存
    private final MemoryRegionCache<ByteBuffer>[] tinySubPageDirectCaches;
    private final MemoryRegionCache<ByteBuffer>[] smallSubPageDirectCaches;
    private final MemoryRegionCache<ByteBuffer>[] normalDirectCaches;
}
```

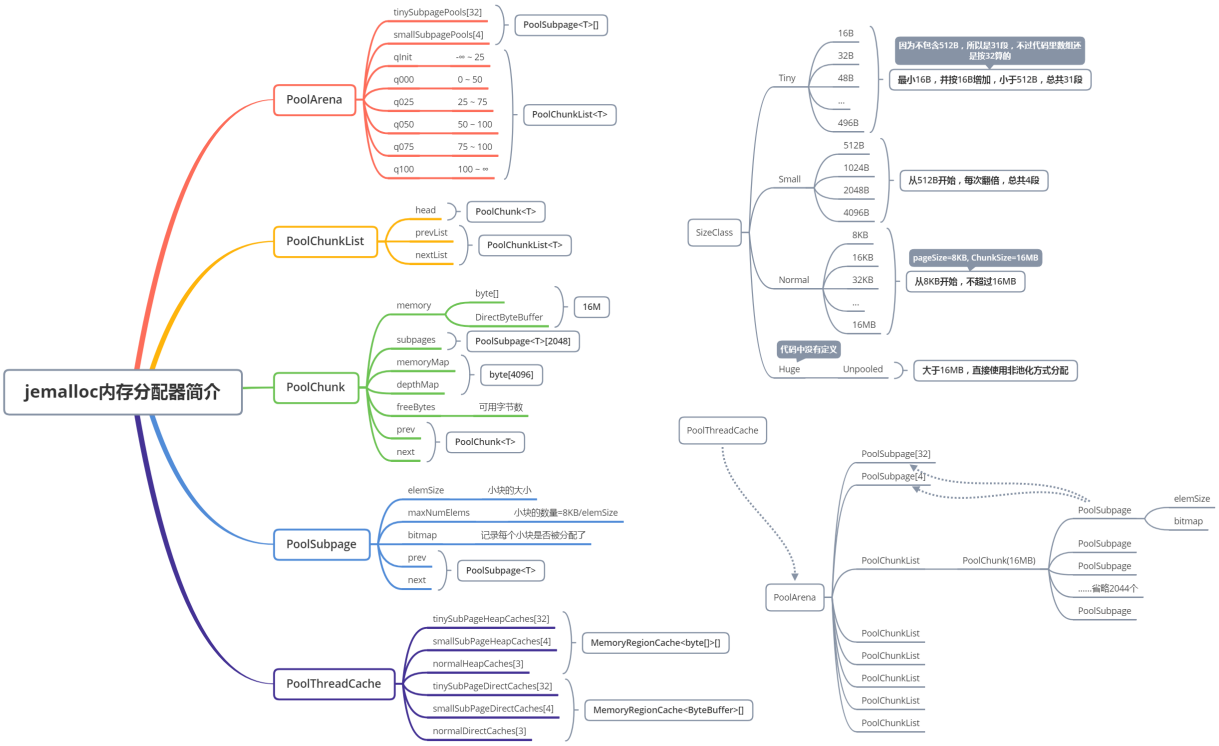
PoolThreadCache 中使用了一个叫作 MemoryRegionCache 的类来做缓存，它内部维护了一个队列，当回收内存时，这块内存进入到这个队列中，当下次再分配同样大小（规范化后的大小）的内存时，从这个队列中取，关于 PoolThreadCache 的使用，我们下一节结合代码一起学习。

后记

今天，我们一起学习了 Netty 中 jemalloc 内存分配器实现的一些基本概念，有了这些概念，我们再去学习 Netty 的内存池，相信一定可以事半功倍的。

下一节，我们将通过调试大法把今天讲的这些概念全部连成线，从根本上理解 Netty 中内存池的实现原理，敬请期待。

思维导图



原生 **jemalloc** 相关链接

[1] one malloc to rule them all

[2] Structures in jemalloc

[3] A Scalable Concurrent malloc(3) Implementation for FreeBSD

[4] Scalable memory allocation using jemalloc

[5] ptmalloc,tcmalloc 和 jemalloc 内存分配策略研究

}

19 Netty的ByteBuf是如何支持直接内存非池化实现的

21 Netty的内存池是如何实现的