22 Netty的对象池又是如何实现的

更新时间：2020-08-10 10:15:43

> 人的差异在于业余时间。——爱因斯坦

前言

你好，我是彤哥。

上一节，我们一起学习了 Netty 内存池的知识，它主要采用 jemalloc 内存分配器来实现，不过，在 Netty 中，还有另外一种池化手段 —— 对象池，相比于内存池，对象池要简单不少，你知道它的实现原理吗？

今天，我们就来学习 Netty 对象池的相关概念及源码阅读。

好了，开始今天的学习吧。

问题

关于 Netty 对象池，我想问你几个问题：

1. Netty 对象池与内存池的区别与联系？
2. Netty 内存池中是否使用到了对象池？
3. Netty 对象池中是否使用到了内存池？
4. Netty 对象池实现的基本逻辑？

Netty 对象池详解

调试用例

其实，在上一节，我们就见过了 Netty 对象池，只 是当时我们跳过去了，所以，本节，我们就简单一点，直接使用上一节的调试用例，然后跟踪源码到上一节跳过的地方：

```
public class ByteBufTest {
    public static void main(String[] args) {
        // 1. 创建池化的分配器
        ByteBufAllocator allocator = new PooledByteBufAllocator(false);
        // 2. 分配一个40B的ByteBuf
        ByteBuf byteBuf = allocator.heapBuffer(40);
        // 3. 写入数据
        byteBuf.writeInt(4);
        // 4. 读取数据
        System.out.println(byteBuf.readInt());
        // 5. 回收内存
        ReferenceCountUtil.release(byteBuf);
        // 6. 分配一个30B的ByteBuf
        ByteBuf byteBuf2 = allocator.heapBuffer(30);
        // 7. 再次分配一个40B的ByteBuf
        ByteBuf byteBuf3 = allocator.heapBuffer(40);
    }
}
```

通过上一节的学习，我们知道，当分配 30B 内存的 ByteBuf 的时候是不会使用到内存池的，那么，它会使用到对象池吗？

源码剖析

创建 40B 的 ByteBuf 对象

我们把断点打在 `ByteBuf byteBuf = allocator.heapBuffer(40);` 这一行，跟踪进去，看看第一次分配 ByteBuf 的时候发生了什么：

```
// ...
// PoolArena#allocate(PoolThreadCache, int, int)
PooledByteBuf<T> allocate(PoolThreadCache cache, int reqCapacity, int maxCapacity) {
    // key，创建一个ByteBuf
    PooledByteBuf<T> buf = newByteBuf(maxCapacity);
    // 通过内存池给它分配内存，上一节的内容，本节不展开
    allocate(cache, buf, reqCapacity);
    return buf;
}
```

本节，我们的重点是学习这个 buf 到底是如何创建出来的，所以，下面那行关于内存池的内容我们就不展开了，继续跟进：

```
// PoolArena.HeapArena#newByteBuf
@Override
protected PooledByteBuf<byte[]> newByteBuf(int maxCapacity) {
    return HAS_UNSAFE ? PooledUnsafeHeapByteBuf.newUnsafeInstance(maxCapacity)
        : PooledHeapByteBuf.newInstance(maxCapacity);
}
// PooledUnsafeHeapByteBuf#newUnsafeInstance
static PooledUnsafeHeapByteBuf newUnsafeInstance(int maxCapacity) {
    // key，通过RECYCLER得到一个buf，所以秘密在这个RECYCLE中
    PooledUnsafeHeapByteBuf buf = RECYCLER.get();
    // 重新使用，重置属性
    buf.reuse(maxCapacity);
    return buf;
}
final void reuse(int maxCapacity) {
    // 设置最大容量
    maxCapacity(maxCapacity);
    // 重置引用计数
    resetRefCnt();
    // 重置readIndex和writeIndex为0
    setIndex0(0, 0);
    // 重置mark为0
    discardMarks();
}
```

这段代码的关键是通过一个叫作 `RECYCLER` 的东西获得了一个 buf，这玩意是个什么东西呢？或者它是不是东西呢？其实，它是定义在 PooledUnsafeHeapByteBuf 类中的一个常量：

```
final class PooledUnsafeHeapByteBuf extends PooledHeapByteBuf {
    // 第一个核心类——ObjectPool
    private static final ObjectPool<PooledUnsafeHeapByteBuf> RECYCLER = ObjectPool.newPool(
            // 第二个核心类——ObjectCreator
            new ObjectCreator<PooledUnsafeHeapByteBuf>() {
        @Override
        public PooledUnsafeHeapByteBuf newObject(Handle<PooledUnsafeHeapByteBuf> handle) {
            // 实际创建buf的地方
            return new PooledUnsafeHeapByteBuf(handle, 0);
        }
    });
}
```

可以看到，RECYCLER 是 ObjectPool 的一个对象，它里面封装了一个 ObjectCreator 对象，用来创建对象，当然，这里创建的就是 PooledUnsafeHeapByteBuf 对象了，且它的容量为 0，同时还有一个 handle 属性，这个 handle 不是我们上一节说的那个 handle 哈，上一节的 handle 是一个 long 类型的，里面存储了节点编号及 bitmapIdx 等相关信息，那么，今天这个 handle 是个什么东西呢？不急，我们慢慢来分析。

我们先不纠结于这个类，接着上面的代码继续跟踪：

```
private static final class RecyclerObjectPool<T> extends ObjectPool<T> {
    // 第三个核心类——Recycler
    private final Recycler<T> recycler;

    RecyclerObjectPool(final ObjectCreator<T> creator) {
        // Recycler对象，包装creator
        recycler = new Recycler<T>() {
            // 第四个核心类——Handle
            // 注意这个方法，下面会用到
            @Override
            protected T newObject(Handle<T> handle) {
                // 通过creator创建对象
                return creator.newObject(handle);
            }
        };
    }

    @Override
    public T get() {
        // 调用recycler的get()方法
        return recycler.get();
    }
}
```

可以看到，这个 get () 方法最后是调用了 Recycler 的 get () 方法，继续跟进：

```
// Recycler#get
public final T get() {
    // 每个线程最多可缓存的容量大小，默认为4KB
    if (maxCapacityPerThread == 0) {
        return newObject((Handle<T>) NOOP_HANDLE);
    }
    // 第五个核心类——Stack
    // 从threadLocal中获取一个对象，这个对象是Stack的对象
    Stack<T> stack = threadLocal.get();
    // 从栈中弹出一个元素，这个元素是DefaultHandle，它实现了Handle接口
    DefaultHandle<T> handle = stack.pop();
    if (handle == null) {
        // handle为空则创建之
        handle = stack.newHandle();
        // 并创建它里面的对象
        // newObject()方法即上面Recycler实现中的方法
        handle.value = newObject(handle);
    }
    // 返回创建的对象
    return (T) handle.value;
}
```

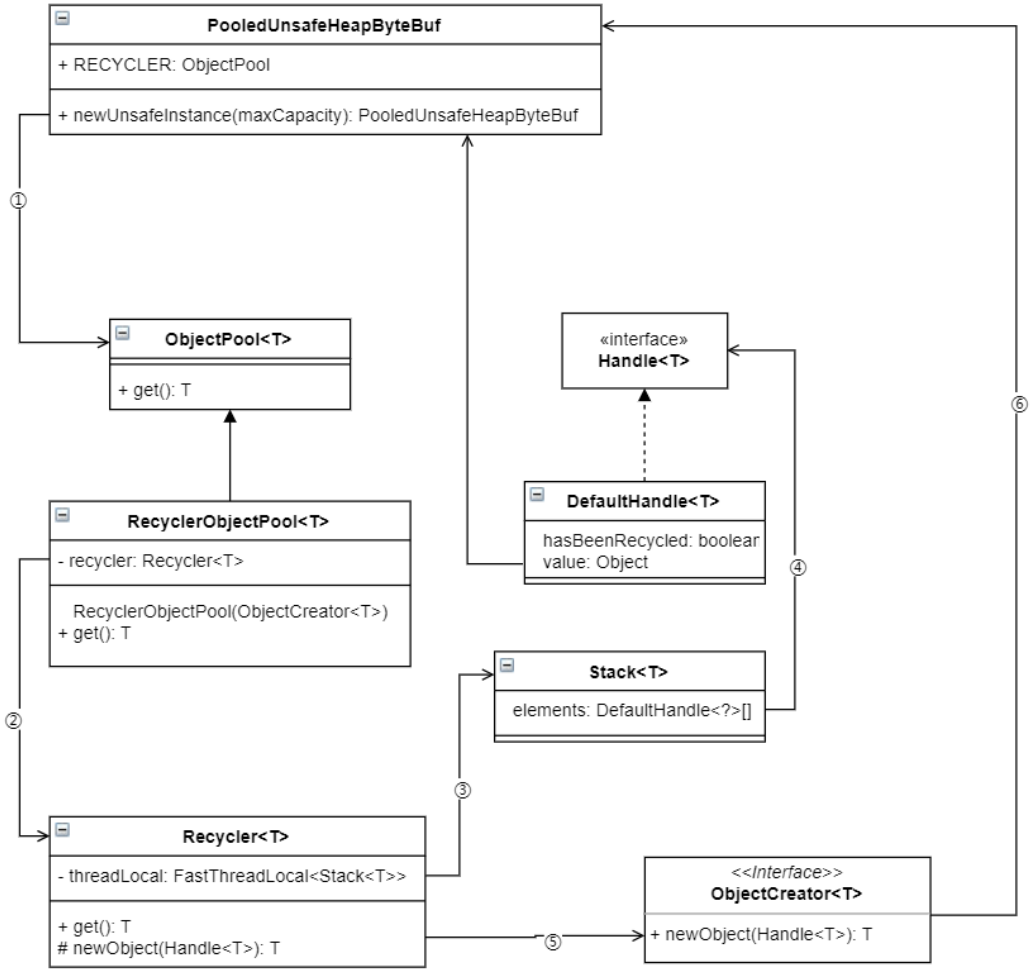
OK，到这里整个逻辑都比较清晰了，Netty 对象池底层是通过 ThreadLocal + Stack（栈）实现的，把 Stack 存储在线程本地变量中，这个栈里面存储的是 Handle，Handle 里面关联了 PooledByteBuf，当我们获取 PooledByteBuf 时，先尝试从线程本地变量的栈中取出一个 Handle，如果没有缓存，则新建一个 Handle 并初始化它里面的 PooledByteBuf，即 value。

> 确切地来说，这个 ThreadLocal 是 FastThreadLocal，它在 Java 原生 ThreadLocal 之上做了一些改进，使其性能更优越，我们后面再单独分析这个类。

直接在 Stack 里面存储 PooledByteBuf 它不香吗？为什么要存储 Handle，再用 Handle 来关联 PooledByteBuf 呢？

其实，Stack 直接存储 PooledByteBuf 作为元素也是可以的，中间加一层 Handle，是因为 Handle 需要存储一些状态信息，比如，是否被回收等，如果把这些信息放到 PooledByteBuf 中，显然会对 PooledByteBuf 本身造成侵略，而且，如果对象池中放到不是 PooledByteBuf，而是换成另一个类，那个类也要加上诸如是否被回收等状态信息，显然也是不合适的，所以，抽象出来一层 Handle 专门用来存储这些状态信息，再用 Handle 去关联真正缓存的对象，这样设计扩展性更好，代码解耦更彻底。

如果上面这样一大段文字看着比较费劲，那我们就把上面标记的那些核心类连起来看看：



这是一个非标准的类图，我把组合、聚合、关联、依赖等几种关系统一弱化成了关联关系，即图中标有数字的那种箭头，而且，我并没有把所有的关联关系全部画出来，看着这张图，我们梳理一下整个流程：

1. 创建 buf 的入口是 PooledUnsafeHeapByteBuf 的 newUnsafeInstance () 方法；
2. newUnsafeInstance () 方法中调用了 RECYCLER（ObjectPool 的对象）的 get () 方法，而 RECYCLER 实际上是 RecyclerObjectPool 的对象；
3. RecyclerObjectPool 的 get () 方法中调用了 recycler（Recycler 的对象）的 get () 方法，而 recycler 实际上是 Recycler 的匿名实现类的对象，这个匿名实现类中实现了 newObject () 方法，newObject () 方法在下面会用到；
4. Recycler 的 get () 方法是真正干活的地方，内部主要做了四件事：

    - 从 threadLocal 中获取 Stack 对象；
    - 从 Stack 中弹出 DefaultHandle 对象；
    - 如果没有弹出任何元素，则创建一个 DefaultHandle，并调用 Recycler 的 newObject () 方法创建实际要缓存的对象；

- 返回 DefaultHandle 中实际缓存的对象，即它的 value 值；

5. 在匿名 Recycler 的 newObject () 内部，又调用了了 ObjectCreator 的 newObject () 方法；
6. ObjectCreator 的 newObject () 方法的实现实际上是在 PooledUnsafeHeapByteBuf 初始化 RECYCLER 时实现的；
7. ObjectCreator 的 newObject () 方法中调用了 PooledUnsafeHeapByteBuf 的构造方法初始化了一个容量为 0 的 buf，且绑定了 DefaultHandle。

OK，到此从对象池中拿 buf 的过程剖析完毕，也就是说，第一次创建 buf 的时候实际上最后还是使用 buf 自己的构造方法创建了一个 0 容量的 buf，跟上一节的内容结合在一起，后面就是给这个 buf 分配内存了，我们就跳过啦，搞不清楚的同学可以把这两节的内容串在一起调试一遍。

回收 ByteBuf 对象

对象池的作用无疑是重复利用资源，所以，如果没有回收对象，就体现不出它的价值了，所以，让我们看看回收对象的时候这个对象是怎么加入到对象池中的。

OK，让我们在 `ReferenceCountUtil.release(byteBuf);` 打一个断点，并跟踪进去（跳过了内存池相关的代码）：

```
// PooledByteBuf#deallocate
@Override
protected final void deallocate() {
    if (handle >= 0) {
        final long handle = this.handle;
        this.handle = -1;
        memory = null;
        chunk.arena.free(chunk, tmpNioBuf, handle, maxLength, cache);
        tmpNioBuf = null;
        chunk = null;
        // 关键之处
        recycle();
    }
}
// PooledByteBuf#recycle
private void recycle() {
    recyclerHandle.recycle(this);
}
// Recycler.DefaultHandle#recycle
@Override
public void recycle(Object object) {
    if (object != value) {
        throw new IllegalArgumentException("object does not belong to handle");
    }

    Stack<?> stack = this.stack;
    if (lastRecycledId != recycleId || stack == null) {
        throw new IllegalStateException("recycled already");
    }

    stack.push(this);
}
```

OK，这一块的逻辑比较简单，最后就是把 buf 中绑定的 handle 入到 Stack 中，因为 handle 里面同样保存了 buf，所以，下一次再创建 buf 的时候就可以重复利用这个 buf 了。

创建 30B 的 ByteBuf 对象

相信通过上面的源码分析，到这里就很简单了，我们直接脑补一下流程：

1. 通过 PooledUnsafeHeapByteBuf 的 newUnsafeInstance () 方法入口一直调到 Recycler 的 get () 方法；
2. 从 ThreadLocal 中获取 Stack；
3. 从 Stack 中弹出 Handle，正好有一个刚放进去的 Handle；
4. 直接返回 Handle 中的 value，即 buf；
5. 重置 buf 的属性，为我们重新使用；
6. 调用内存池相关逻辑为其分配内存；

可以看到，对象池跟分配多少字节的数据完全没有关系，那是内存池的事儿，所以，这里创建 30B 的 ByteBuf 对象一样可以重复利用前面缓存下来的 ByteBuf 对象。

到这里，对象池的源码剖析基本就结束了，既然这个对象池这么好用，我们能不能使用呢？答案当然是可以啦。

对象池的使用

其实，使用方法也非常简单，就像你看到的 PooledUnsafeHeapByteBuf 中的用法一样，只需要简单的三步就可以很容易使用 Netty 的对象池了：

1. 需要使用的类要与 handle 绑定；
2. 这个类的对象回收的时候调用 handler 的 recycle () 方法；
3. 定义一个 ObjectPool 对象，用它来创建这个类的对象；

比如，在游戏中有这么一个类叫作 Player，也就是玩家，假设玩家这个类的对象的创建过程是非常耗费资源的，那么，我们完全就可以把下线的玩家缓存到对象池中，等下次有上线的玩家，我们直接从对象池中拿一个“玩家”，并重置它的属性给这次新上线的玩家复用是完全可以的，下面我们就来简单实现这么个逻辑。

定义玩家类

```
public class Player {

    // 定义对象池
    private static final ObjectPool<Player> RECYCLER = ObjectPool.newPool(
            handle -> new Player(handle));

    // 玩家属性
    private Long id;
    private String name;
    // ...

    // 绑定handle
    private ObjectPool.Handle<Player> handle;

    // 构造方法私有化
    private Player(ObjectPool.Handle<Player> handle) {
        this.handle = handle;
    }

    // 创建实例
    public static Player newInstance(Long id, String name) {
        Player player = RECYCLER.get();
        player.id = id;
        player.name = name;
        return player;
    }

    // 玩家下线
    public void offline() {
        handle.recycle(this);
    }

    // 打印
    @Override
    public String toString() {
        return String.format("id=%d, name=%s, classId=%s", id, name, super.toString());
    }
}
```

这个 Player 类比较简单，我们来测试一下。

测试玩家类

```
public class PlayerTest {
    public static void main(String[] args) {
        // 创建player1
        Player player1 = Player.newInstance(1L, "alan");
        // 打印player1
        System.out.println(player1);
        // player1下线
        player1.offline();

        // 创建player2
        Player player2 = Player.newInstance(2L, "alex");
        // 打印player2
        System.out.println(player2);
        // 二者是否相等
        System.out.println(player1 == player2);

        // 假设player1又上线了呢
        player1 = Player.newInstance(1L, "alan");
        // 打印player1
        System.out.println(player1);
        // 二者是否相等
        System.out.println(player1 == player2);
    }
}
```

你猜这个结果会如何？我直接给出我的运行结果了：

```
id=1, name=alan, classId=com.imooc.netty.core.$21.Player@300ffa5d
id=2, name=alex, classId=com.imooc.netty.core.$21.Player@300ffa5d
true
id=1, name=alan, classId=com.imooc.netty.core.$21.Player@37f8bb67
false
```

是不是很简单？！^^

其它

其实，Netty 中对象池可不只是用在池化 ByteBuf 这里，在很多地方都有用到，比如，内存池里面的 PoolThreadCache 中用来做缓存的 MemoryRegionCache 中的队列中的元素 Entry：

```
// io.netty.buffer.PoolThreadCache.MemoryRegionCache.Entry
static final class Entry<T> {
    final Handle<Entry<?>> recyclerHandle;
    PoolChunk<T> chunk;
    ByteBuffer nioBuffer;
    long handle = -1;

    Entry(Handle<Entry<?>> recyclerHandle) {
        this.recyclerHandle = recyclerHandle;
    }

    void recycle() {
        chunk = null;
        nioBuffer = null;
        handle = -1;
        recyclerHandle.recycle(this);
    }
}

@SuppressWarnings("rawtypes")
private static Entry newEntry(PoolChunk<?> chunk, ByteBuffer nioBuffer, long handle) {
    Entry entry = RECYCLER.get();
    entry.chunk = chunk;
    entry.nioBuffer = nioBuffer;
    entry.handle = handle;
    return entry;
}

@SuppressWarnings("rawtypes")
private static final ObjectPool<Entry> RECYCLER = ObjectPool.newPool(new ObjectCreator<Entry>() {
    @SuppressWarnings("unchecked")
    @Override
    public Entry newObject(Handle<Entry> handle) {
        return new Entry(handle);
    }
});
```

另外，还有 `io.netty.channel.AbstractChannelHandlerContext.WriteTask`，等等，有兴趣的同学可以翻翻源码看看。

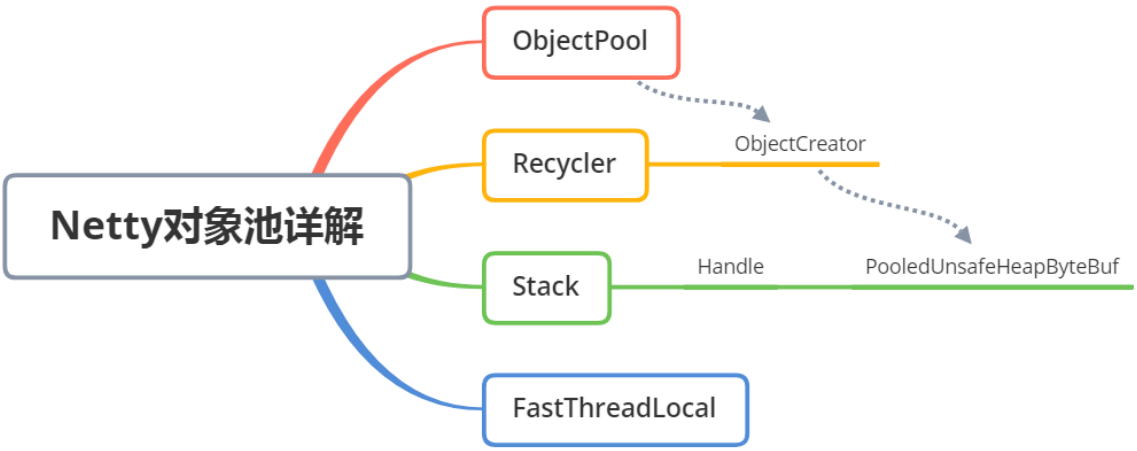
后记

今天，我们一起学习了 Netty 中的对象池，可以发现，对象池的实现相比于内存池的实现不要太简单，是吧？！

但是，它真的如此简单么？我留了两道思考题，你能解答出来吗？见文后。

在今天的内容中，我们提到了对象池底层使用到了一个叫作 `FastThreadLocal` 的类，它是比 Java 原生的 ThreadLocal 效率更高的实现，它有哪些神奇之处呢？我们下一节一起来剖析之，敬请期待。

思维导图



思考题一

```
public class ByteBufTest {
    public static void main(String[] args) throws InterruptedException {
        // 参数是preferDirect，即是否偏向于使用直接内存
        ByteBufAllocator allocator = new PooledByteBufAllocator(false);
        // 分配一个40B的ByteBuf
        ByteBuf byteBuf = allocator.heapBuffer(40);
        // 写入数据
        byteBuf.writeInt(4);
        // 读取数据
        System.out.println(byteBuf.readInt());
        // 回收内存
        new Thread(()->ReferenceCountUtil.release(byteBuf)).start();
        // 休息1秒
        Thread.sleep(1000);
        // 再次分配一个40B的ByteBuf
        ByteBuf byteBuf2 = allocator.heapBuffer(30);
    }
}
```

如上代码，创建 30B 的 ByteBuf 时是否可以使用到对象池中的缓存？

思考题二

```
public class ByteBufTest {
    public static void main(String[] args) throws InterruptedException {
        // 参数是preferDirect，即是否偏向于使用直接内存
        ByteBufAllocator allocator = new PooledByteBufAllocator(false);
        // 分配一个40B的ByteBuf
        ByteBuf byteBuf = allocator.heapBuffer(40);
        // 写入数据
        byteBuf.writeInt(4);
        // 读取数据
        System.out.println(byteBuf.readInt());
        // 回收内存
        ReferenceCountUtil.release(byteBuf);
        // 休息1秒，保证完全释放
        Thread.sleep(1000);
        new Thread(() -> {
            // 再次分配一个40B的ByteBuf
            ByteBuf byteBuf2 = allocator.heapBuffer(30);
            // 其它处理
        }).start();
    }
}
```

如上代码，创建 30B 的 ByteBuf 时是否可以使用到对象池中的缓存？

}

21 Netty的内存池是如何实现的

23 Netty的FastThreadLocal到底快在哪里