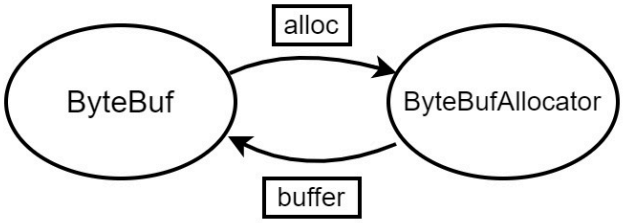
面向对象源码阅读：Netty Btyebuf 模块——高级设计意图分析

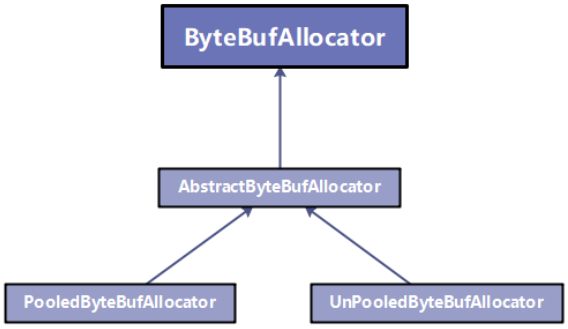
一、抽象工厂模式

意图：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。
主要解决：接口选择问题
应用场景：系统的产品有多于一个的产品族，而系统只消费其中某一族的产品。
如何解决：在一个产品族里面，定义多个产品。

在具体的 ByteBuf 应用场景中，如果要使用一个 ByteBuf，首先要获取一段内存空间，之前在第一部分和第二部分的报告中已经提到，ByteBuf 主要有池化分配方法和非池化分配方法两种分配方法，我们首先来梳理一下它们的类间关系：



ByteBuf 和 ByteBufAllocator 是两个接口，ByteBufAllocator 可以创建一个 ByteBuf，而 ByteBuf 可以返回创建他的 Allocator。
但是，要注意 ByteBufAllocator 只是接口，具体的实现是如下图的类间关系实现的：



ByteBufAllocator 为接口，AbstractByteBufAllocator 是抽象类，而 PooledByteBufAllocator，UnPooledByteBufAllocator 是继承 AbstractByteBufAllocator 的具体实现类。

在 Netty 中，ByteBuf 需要 allocator“生产”出不同种类（池化和非池化）的可用内存空间，而无论是池化还是非池化，都需要它们能在堆外内存和直接内存上进行分配。所以这两类“工厂”生产出的产品功能是基本一致的，只是在池化问题上的特性不一样。

这种情况下就可以使用抽象工厂模型，ByteBufAllocator 是抽象工厂接口，这两种内存空间的“生产”对应 PooledByteBufAllocator 和 UnPooledByteBufAllocator 这两个“工厂”。

如果要简单的概括一下这里体现出来的抽象工厂模型的特性，那么就是：抽象类需要一组方法的实现，而子类都有这些方法，只是实现不同。

在 Netty 的源码中可以看到，Netty 给 ByteBufAllocator 设置了一个默认工厂：

```
public interface ByteBufAllocator {
    ByteBufAllocator DEFAULT = ByteBufUtil.DEFAULT_ALLOCATOR;
}
```

而 ByteBufUtil 会根据 allocType 得到具体的工厂

```
public final class ByteBufUtil {
    static final ByteBufAllocator DEFAULT_ALLOCATOR;

    static {
        String allocType = SystemPropertyUtil.get(
                "io.netty.allocator.type",
PlatformDependent.isAndroid() ? "unpooled" : "pooled");
        allocType = allocType.toLowerCase(Locale.US).trim();

        ByteBufAllocator alloc;
        if ("unpooled".equals(allocType)) {
            alloc = UnpooledByteBufAllocator.DEFAULT;
            logger.debug("-Dio.netty.allocator.type: {}", allocType);
        } else if ("pooled".equals(allocType)) {
            alloc = PooledByteBufAllocator.DEFAULT;
            logger.debug("-Dio.netty.allocator.type: {}", allocType);
        } else {
            alloc = PooledByteBufAllocator.DEFAULT;
            logger.debug("-Dio.netty.allocator.type: pooled (unknown:
{})", allocType);
        }
    }
```

下面列出非池化方法的实现代码：

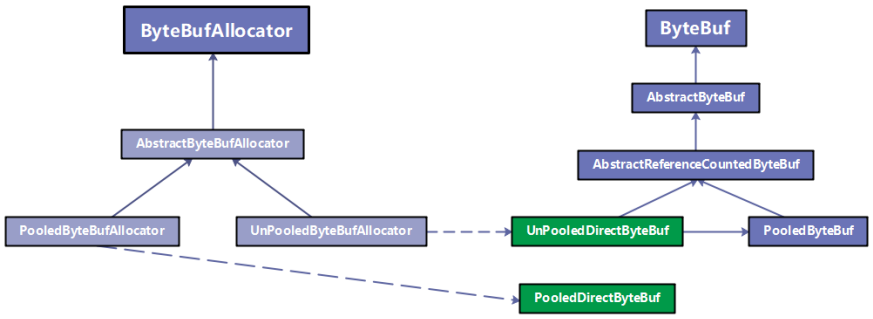
```
@Override
    protected ByteBuf newHeapBuffer(int initialCapacity, int
maxCapacity) {
        return PlatformDependent.hasUnsafe() ?
```

```
                new InstrumentedUnpooledUnsafeHeapByteBuf(this,
initialCapacity, maxCapacity) :
                new InstrumentedUnpooledHeapByteBuf(this,
initialCapacity, maxCapacity);
    }

    @Override
    protected ByteBuf newDirectBuffer(int initialCapacity, int
maxCapacity) {
        final ByteBuf buf;
        if (PlatformDependent.hasUnsafe()) {
            buf = noCleaner ? new
InstrumentedUnpooledUnsafeNoCleanerDirectByteBuf(this,
initialCapacity, maxCapacity) :
                    new InstrumentedUnpooledUnsafeDirectByteBuf(this,
initialCapacity, maxCapacity);
        } else {
            buf = new InstrumentedUnpooledDirectByteBuf(this,
initialCapacity, maxCapacity);
        }
        return disableLeakDetector ? buf : toLeakAwareBuffer(buf);
    }
```

池化方法和非池化方法的结构是类似的，这里就不重复列举代码了。

至此我们梳理了抽象工厂接口到具体工厂实现的结构，一开始我们展示了 ByteBuf 和 ByteBufAllocator 的关系，现在可以把关系图更加清楚地展现出来：



其中深紫色为接口，紫色为抽象类，淡紫色和绿色为具体的实现类。ByteBuf 通过 alloc 方法获得创建它的 allocator，从而由具体的池化或非池化方法完成地址空间的处理和分配。

在抽象工厂模式分析的最后部分，笔者想提出一点质疑，抽象工厂模式往往强调具体的工厂生产出的产品之间有相关性或者依赖关系，但是在 ByteBuf 中使用的抽象工厂设计模式中，池化和非池化工厂生产出的空间之间似乎没有依赖关系或者紧密的相关性。换言之，如果使用工厂模式，而不是抽象工厂模式进行设计，似乎也是可行的。

二、组合模式

意图：模糊简单元素和复杂元素的概念，允许程序像处理简单元素一样来处理复杂元素，从而使得程序与复杂元素的内部结构解耦。

应用场景：需要忽略组合对象与单个对象的不同时、需要统一地使用组合结构中的所有对象时。

如何解决：树枝和叶子实现统一接口，树枝内部组合该接口

在 Netty 的应用场景中，常常需要将多个 ByteBuf 结构进行合并或者重组，以进一步处理或者向下传递。具体的一个例子是，在网络中，包头，包体的拼接。

在将两个或者多个 ByteBuf 合并为一个 ByteBuf 处理时，一般的思路是将两个或者多个 ByteBuf 拷贝到一个新的 ByteBuf 里。JDK 中的处理办法就是如此，代码摘录如下：

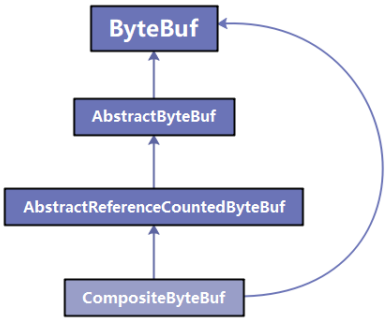
```
//Use an array to hold the message parts
ByteBuffer[] message = new ByteBuffer[] { header, body };
// Create a new ByteBuffer to merge the header and body
ByteBuffer message2 =ByteBuffer.allocate(header.remaining()+ body
.remaining()）;
message2.put(header);
message2.put (body);
message2.flip();
```

显然，这种办法会导致额外的性能开销和空间占用，只是满足了基本功能，在效率上是不尽如人意的。

在 Netty 中，由于 CompositeByteBuf 类使用了组合模式的设计方法，使得它可以直接将这些 ByteBuf 形成一个统一的视图，对外提供和单个 ByteBuf 一样的操作。

在第一部分中，我们曾经提到过 CompositeByteBuf，它可以让我们把多个 ByteBuf 当成一个 ByteBuf 进行处理，这里我们再对它做详细的介绍。

首先给出这部分的类间关系：



ByteBuf 是抽象接口，AbstractByteBuf 和 AbstractReferenceCountesByteBuf 是依次继承关系的抽象类，CompositeByte 是一个具体的实现类，继承 AbstractReferenceCountesByteBuf 而来。

我们可以从源码中看出这样的继承关系：

```
public class CompositeByteBuf extends AbstractReferenceCountedByteBuf
implements Iterable<ByteBuf> {
    ...
}
```

在 CompositeByteBuf 中定义了一个包装类的集合：Component，Component 是 ByteBuf

的包装类：

```
private Component[] components;
```

component 的类定义如下：

```
private static final class Component {
        ...
         final ByteBuf buf;
         Component(ByteBuf srcBuf, int srcOffset, ByteBuf buf, int
bufOffset,
                 int offset, int len, ByteBuf slice) {
             this.srcBuf = srcBuf;
             this.srcAdjustment = srcOffset - offset;
             this.buf = buf;
             this.adjustment = bufOffset - offset;
             this.offset = offset;
             this.endOffset = offset + len;
             this.slice = slice;
         }
         ...
}
```

如果要新建一个由多个 ByteBuf 组成的 ByteBuf，则可以使用 CompositeByteBuf 的 addComponent 方法：

```
public CompositeByteBuf addComponent(boolean increaseWriterIndex, int
cIndex, ByteBuf buffer) {
    checkNotNull(buffer, "buffer");
    addComponent0(increaseWriterIndex, cIndex, buffer);
    consolidateIfNeeded();
    return this;
}
```

其中 addComponent0 方法会逐层调用方法，最终使用上面提及的 Component 集合：

```
private void addComp(int i, Component c) {
    shiftComps(i, 1);
    components[i] = c;
}
```

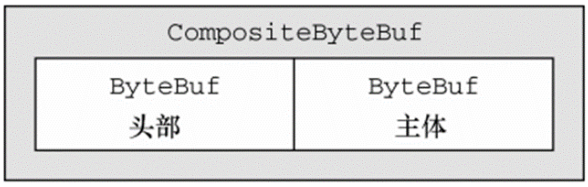
值得注意的是，CompositeByteBuf 是 AbstractReferenceCountesByteBuf 的实现类，所以它也应该可以提供 ByteBuf 其他实现类可以提供的方法（区别在于前者是将多个 ByteBuf 合并，对外表现为操作一个 ByteBuf）。

```
public CompositeByteBuf getBytes(int index, OutputStream out, int
length) throws IOException {
    checkIndex(index, length);
    if (length == 0) {
        return this;
    }
    int i = toComponentIndex0(index);
    while (length > 0) {
```

```
        Component c = components[i];
        ......
    }
    return this;
}
```

例如上述部分的代码就是 CompositeByteBuf 提供的重写的 getBytes 方法。

关于该种组合模式可以实现的网络环境下的组合包头包体的案例说明，可以在报告的第一部分看到，这里再次引用图示：



组合模式可以视作面向对象编程中“隐藏”思想的一种体现。用户程序不需要关心一个包内的多个 ByteBuf 的具体结构，也不用关心它们的关系，只需要用 CompositeByteBuf 提供的方法，像操作单个 ByteBuf 那样进行操作即可，简化了外部视角下的操作。

CompositeByteBuf 的另一个重要意义在于，它可以省去拷贝数据带来的性能消耗，提高效率。与之相对的就是 JDK 的 ByteBuffer，它在合并多个缓冲区时，需要将原先的缓冲区写入到新创建的缓冲区里，效率不尽如人意。

但是也要注意，组合模式非常依赖继承关系，容易导致后续的扩展的便利性降低，也容易出现过度设计，在使用时需要仔细考虑。在笔者看来，Netty 这里使用组合模式是比较合理的处理，规避了组合模式的这些缺点。

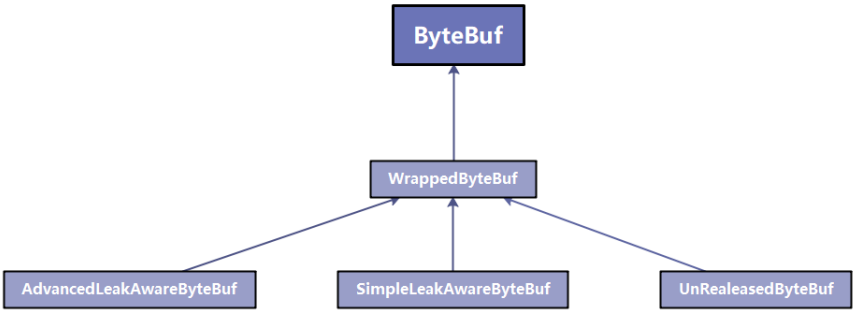
三、装饰器模式

意图：动态地给一个对象添加一些额外的职责。

应用场景：在不想增加很多子类的情况下扩展类。

如何解决：将具体功能职责划分，同时继承装饰者模式。

在 Netty 的具体应用场景中，为了对一些 ByteBuf 进行保护或者标记，抑或是出于类似防止内存泄露的初衷，需要对 ByteBuf 的功能进行增强或者增加特征，这个时候就需要用到装饰器模式：



在 Netty 中，WrappedByteBuf 是所有 ByteBuf 装饰器的基类，在它的构造函数里传入了原始的 ByteBuf 实例作为被装饰者。

```
class WrappedByteBuf extends ByteBuf {
    protected final ByteBuf buf;
    protected WrappedByteBuf(ByteBuf buf) {
        if (buf == null) {
            throw new NullPointerException("buf");
        }
        this.buf = buf;
    }
    ......
    public final boolean hasMemoryAddress() {
        return buf.hasMemoryAddress();
    }
    ......
    public final long memoryAddress() {
        return buf.memoryAddress();
    }
    ......
}
```

WrappedByteBuf 有两个子类 UnreleasableByteBuf 和 SimpleLeakAwareByteBuf，它们是真正实现对 ByteBuf 的功能增强，例如 UnreleasableByteBuf 类的 release() 方法是直接返回 false 表示不可被释放，源码实现如下所示。

```
final class UnreleasableByteBuf extends WrappedByteBuf {
    private SwappedByteBuf swappedBuf;
    UnreleasableByteBuf(ByteBuf buf) {
        super(buf instanceof UnreleasableByteBuf ? buf.unwrap() :
buf);
    }
    ......
    public boolean release() {
        return false;
    }
    ......
}
```

可以看到，通过继承关系，UnreleasableByteBuf 继承了 WrappedByteBuf，使得凡是用 UnreleasableByteBuf 实现的对象都不可以通过 release 方法释放。实现了对原有功能的更改，为 ByteBuf 增添了不可被释放的特性。

四、总结

通过上面的设计模式的分析介绍，我们可以发现，ByteBuf 中使用了大量的继承关系，广泛地运用了接口—抽象类—实现类的关系。

这样的策略有利有弊，一方面，在第二部分展示过的类间关系图已经体现了这样设计的问题：由于大量地使用继承，使得类间的层级较高，最深的层级达到了六层，这对于源码阅读者其实造成了一定困难，想要找到一个方法的具体实现并不容易。
但是另一方面，从功能开发的角度来看，作为一个开源项目，使用继承关系可以有效地复用代码，而不必去更改父类的具体代码。此外，在装饰器和抽象工厂模式的介绍中，我们可以看到继承使得对 ByteBuf 的生成方法选择、功能拓展变得容易并且清晰。一旦理清了它们的类间关系，就会发现继承在这里的使用是合适的。
总而言之，ByteBuf 在结构上是一个重继承轻组合的工程，尽管这与面向对象的基本设计建议不完全符合，但是在它的开发和应用中体现出来的效果是令人满意的。