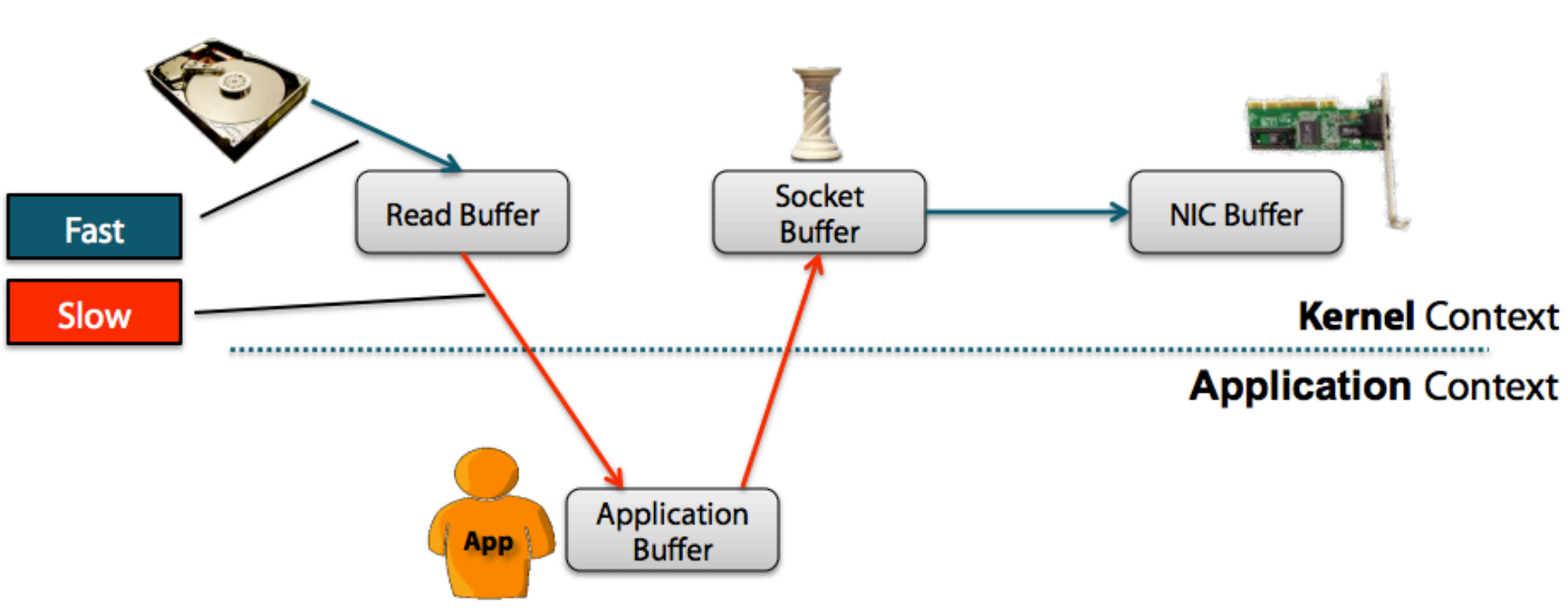
理解Netty中的零拷贝（Zero-Copy）机制 - plucury的个人空间

##理解零拷贝 零拷贝是Netty的重要特性之一，而究竟什么是零拷贝呢？WIKI中对其有如下定义：

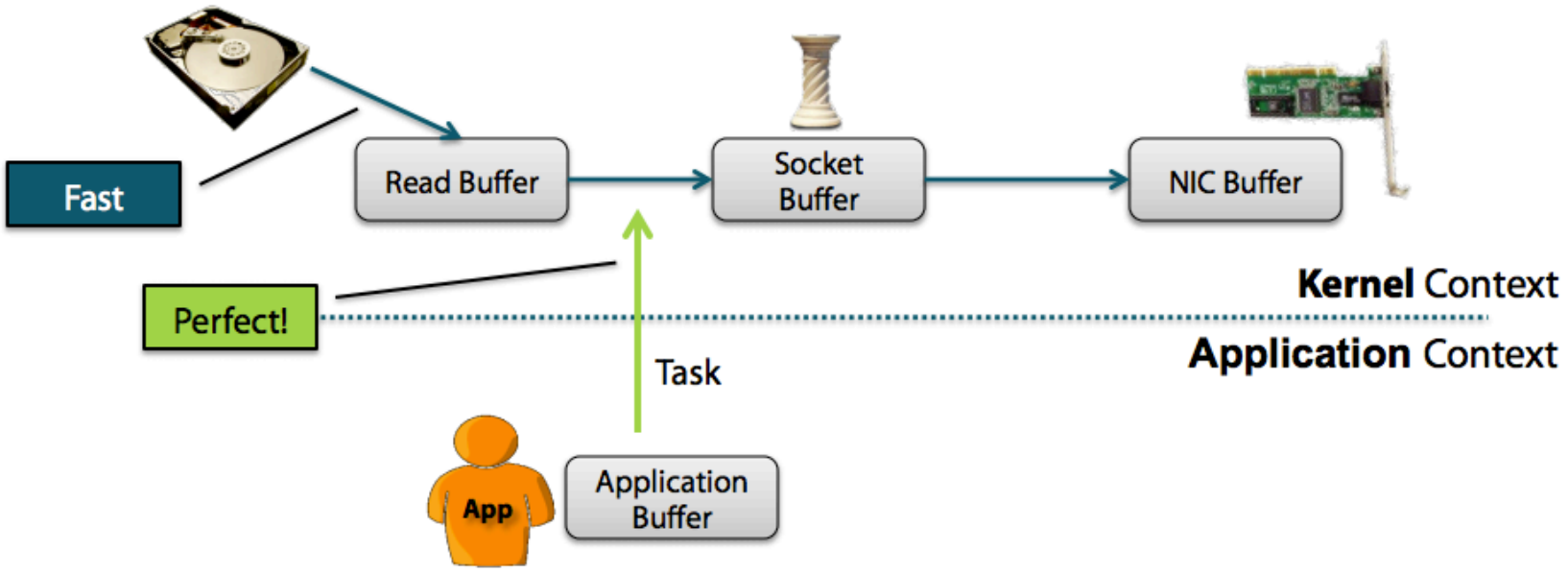
> "Zero-copy" describes computer operations in which the CPU does not perform the task of copying data from one memory area to another.

从WIKI的定义中，我们看到“零拷贝”是指计算机操作的过程中，CPU不需要为数据在内存之间的拷贝消耗资源。而它通常是指计算机在网络上发送文件时，不需要将文件内容拷贝到用户空间（User Space）而直接在内核空间（Kernel Space）中传输到网络的方式。

Non-Zero Copy方式：



Zero Copy方式：



从上图中可以清楚的看到，Zero Copy的模式中，避免了数据在用户空间和内存空间之间的拷贝，从而提高了系统的整体性能。Linux中的`sendfile()`以及Java NIO中的`FileChannel.transferTo()`方法都实现了零拷贝的功能，而在Netty中也通过在`FileRegion`中包装了NIO的`FileChannel.transferTo()`方法实现了零拷贝。

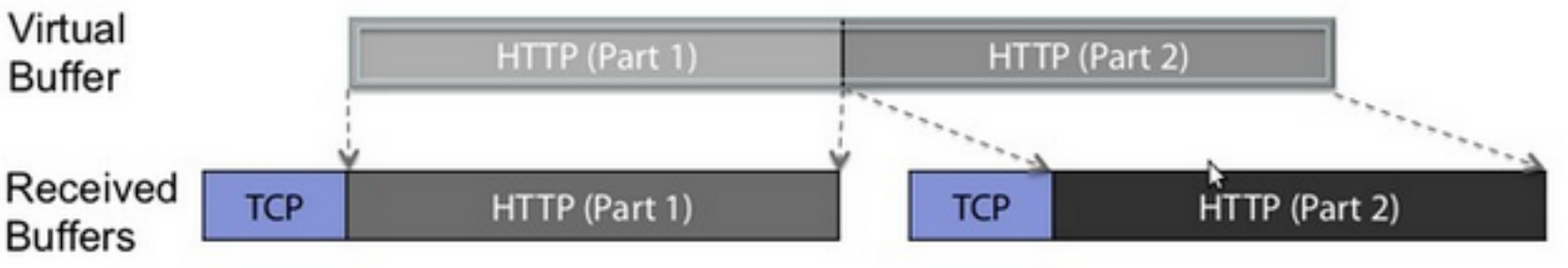
而在Netty中还有另一种形式的零拷贝，即Netty允许我们将多段数据合并为一整段虚拟数据供用户使用，而过程中不需要对数据进行拷贝操作，这也是我们今天要讲的重点。我们都知道在stream-based transport（如TCP/IP）的传输过程中，数据包有可能会被重新封装在不同的数据包中，例如当你发送如下数据时：

```
+-----+-----+-----+
| ABC | DEF | GHI |
+-----+-----+-----+
```

有可能实际收到的数据如下：

```
+----+-------+---+---+
| AB | CDEFG | H | I |
+----+-------+---+---+
```

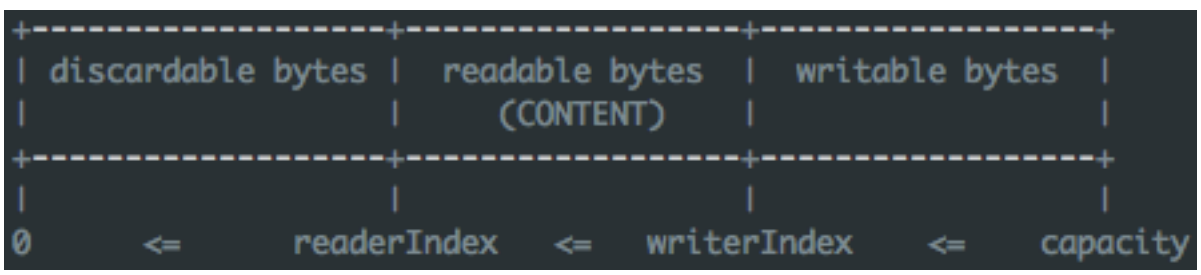
因此在实际应用中，很有可能一条完整的消息被分割为多个数据包进行网络传输，而单个的数据包对你而言是没有意义的，只有当这些数据包组成一条完整的消息时你才能做出正确的处理，而Netty可以通过零拷贝的方式将这些数据包组合成一条完整的消息供你来使用。而此时，零拷贝的作用范围仅在用户空间中。



##Netty3中零拷贝的实现机制 *以下以Netty 3.8.0.Final的源代码来进行说明*

###ChannelBuffer接口 Netty为需要传输的数据制定了统一的`ChannelBuffer`接口。该接口的主要设计思路如下：

- 使用`getByte(int index)`方法来实现随机访问
- 使用双指针的方式实现顺序访问
  - 每个Buffer都有一个读指针（readIndex）和写指针（writeIndex）
  - 在读取数据时读指针后移，在写入数据时写指针后移



定义了统一的接口之后，就是来做各种实现了。Netty主要实现了`HeapChannelBuffer`,`ByteBufferBackedChannelBuffer`等等，下面我们就来讲讲与Zero Copy直接相关的`CompositeChannelBuffer`类。

###CompositeChannelBuffer类 `CompositeChannelBuffer`类的作用是将多个`ChannelBuffer`组成一个虚拟的`ChannelBuffer`来进行操作。为什么说是虚拟的呢，因为`CompositeChannelBuffer`并没有将多个`ChannelBuffer`真正的组合起来，而只是保存了他们的引用，这样就避免了数据的拷贝，实现了Zero Copy。 下面我们来看看具体的代码实现，首先是成员变量

```
private int readerIndex;
private int writerIndex;
private ChannelBuffer[] components;
private int[] indices;
private int lastAccessedComponentId;
```

以上这里列出了几个比较重要的成员变量。其中`readerIndex`既读指针和`writerIndex`既写指针是从`AbstractChannelBuffer`继承而来的；然后`components`是一个`ChannelBuffer`的数组，他保存了组成这个虚拟Buffer

的所有子Buffer，indices是一个int类型的数组，它保存的是各个Buffer的索引值；最后的lastAccessedComponentId是一个int值，它记录了最后一次访问时的子Buffer ID。从这个数据结构，我们不难发现所谓的CompositeChannelBuffer实际上就是将一系列的Buffer通过数组保存起来，然后实现了ChannelBuffer 的接口，使得在上层看来，操作这些Buffer就像是操作一个单独的Buffer一样。

####创建 接下来，我们再看一下CompositeChannelBuffer.setComponents方法，它会在初始化CompositeChannelBuffer时被调用。

```
/**
 * Setup this ChannelBuffer from the list
 */
private void setComponents(List<ChannelBuffer> newComponents) {
    assert !newComponents.isEmpty();

    // Clear the cache.
    lastAccessedComponentId = 0;

    // Build the component array.
    components = new ChannelBuffer[newComponents.size()];
    for (int i = 0; i < components.length; i ++) {
        ChannelBuffer c = newComponents.get(i);
        if (c.order() != order()) {
            throw new IllegalArgumentException(
                    "All buffers must have the same endianness.");
        }

        assert c.readerIndex() == 0;
        assert c.writerIndex() == c.capacity();

        components[i] = c;
    }

    // Build the component lookup table.
    indices = new int[components.length + 1];
    indices[0] = 0;
    for (int i = 1; i <= components.length; i ++) {
        indices[i] = indices[i - 1] + components[i - 1].capacity();
    }

    // Reset the indexes.
    setIndex(0, capacity());
}
```

通过代码可以看到该方法的功能就是将一个ChannelBuffer的List给组合起来。它首先将List中得元素放入到components数组中，然后创建indices用于数据的查找，最后使用setIndex来重置指针。这里需要注意的是setIndex(0, capacity())会将读指针设置为0，写指针设置为当前Buffer的长度，这也就是前面需要做assert c.readerIndex() == 0和assert c.writerIndex() == c.capacity()这两个判断的原因，否则很容易会造成数据重复读写的问题，所以Netty推荐我们使用ChannelBuffers.wrappedBuffer方法来进行Buffer的合并，因为在该方法中Netty会通过slice()方法来确保构建CompositeChannelBuffer是传入的所有子Buffer都是符合要求的。

####数据访问 CompositeChannelBuffer.getByte(int index)的实现如下：

```
public byte getByte(int index) {
    int componentId = componentId(index);
    return components[componentId].getByte(index - indices[componentId]);
}
```

从代码我们可以看到，在随机查找时会首先通过index获取这个字节所在的componentId既字节所在的子Buffer序列，然后通过index - indices[componentId]计算出它在这个子Buffer中的第几个字节，然后返回结果。

下面再来看一下componentId(int index)的实现：

```
private int componentId(int index) {
    int lastComponentId = lastAccessedComponentId;
    if (index >= indices[lastComponentId]) {
        if (index < indices[lastComponentId + 1]) {
            return lastComponentId;
        }

        // Search right
        for (int i = lastComponentId + 1; i < components.length; i ++) {
            if (index < indices[i + 1]) {
                lastAccessedComponentId = i;
```

```
                return i;
            }
        }
    } else {
        // Search left
        for (int i = lastComponentId - 1; i >= 0; i --) {
            if (index >= indices[i]) {
                lastAccessedComponentId = i;
                return i;
            }
        }
    }

    throw new IndexOutOfBoundsException("Invalid index: " + index + ", maxi
}
```

从代码中我们发现，Netty以`lastComponentId`既上次访问的子Buffer序号为中心，向左右两边进行搜索，这样做的目的是，当我们两次随机查找的字符序列相近时（大部分情况下都是这样），可以最快的搜索到目标索引的`componentId`。

##参考资料

1. http://my.oschina.net/flashsword/blog/164237
2. http://en.wikipedia.org/wiki/Zero-copy
3. http://stackoverflow.com/questions/20727615/is-nettys-zero-copy-different-from-os-level-zero-copy
4. http://www-old.itm.uni-luebeck.de/teaching/ws1112/vs/Uebung/GrossUebungNetty/VS-WS1112-xx-Zero-Copy_Event-Driven_Servers_with_Netty.pdf?lang=de

标签： Java Netty 零拷贝 源码分析