crazyhzm 265

Dubbo源码解析（十一）远程通信——Buffer

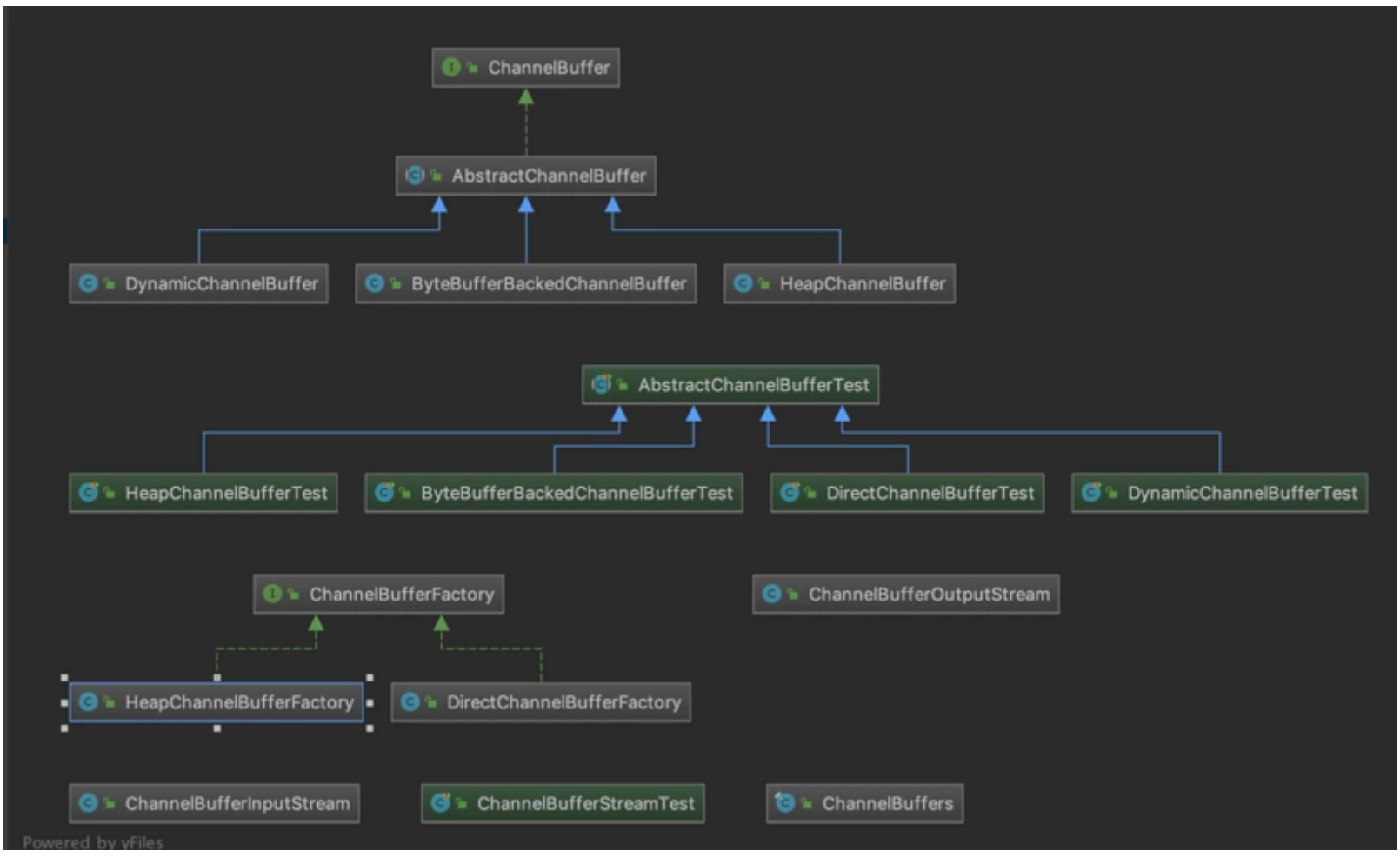
java dubbo 阅读约 28 分钟

远程通讯——Buffer

> 目标：介绍Buffer的相关实现逻辑、介绍dubbo-remoting-api中的buffer包内的源码解析。

前言

缓存区在NIO框架中非常重要，它作为字节容器，每个NIO框架都有自己的相应的设计实现。比如Java NIO有ByteBuffer的设计，Mina有IoBuffer的设计，Netty4有ByteBuf的设计。 那么在本文讲到的内容是dubbo对于缓冲区做的一些接口定义，并且做了不同的框架实现缓冲区公共的逻辑。下面是本文要讲到的类图：



接下来我就按照类图上的一个一个分析，忽略test类。

源码分析

（一）ChannelBuffer

该接口继承了Comparable接口，该接口是通道缓存接口，是字节容器，在netty中也有通道缓存的设计，也就是io.netty.buffer.ByteBuf，该接口的方法定义和设计跟ByteBuf几乎一样，连注释都一样，所以我就不再细说了。

（二）AbstractChannelBuffer

该类实现了ChannelBuffer接口，是通道缓存的抽象类，它实现了ChannelBuffer所有方法，但是它实现的方法都是需要被重写的方法，具体的实现都是需要子类来实现。现在我们来恶补一下这个通道缓存的原理，当然这个原理跟netty的ByteBuf原理是分不开的。AbstractChannelBuffer维护了两个索引，一个用于读取，另一个用于写入当你从通道缓存中读取时，readerIndex将会被递增已经被读取的字节数，同样的当你写入的时候writerIndex也会被递增。

```
/**
 * 读索引
 */
private int readerIndex;

/**
 * 写索引
 */
private int writerIndex;

/**
 * 标记读索引
 */
private int markedReaderIndex;

/**
 * 标记写索引
 */
private int markedWriterIndex;
```

可以看到该类有四个属性，读索引和写索引的作用就是我上述介绍的，读索引和写索引的起始位置都为索引位置0。而标记读索引和标记写索引是为了做备份回滚，当对缓冲区进行读写操作时，可能需要对之前的操作进行回滚，我们就需要将当前的读写索引备份到相应的标记索引中。

该类的其他方法都是利用四个属性来操作，无非就是检测是否有数据可读或者还是否有空间可写等方法，做一些前置条件的校验以及索引的设置，具体的实现都是需要子类来实现，所以我就不贴代码，因为逻辑比较简单。

（三）DynamicChannelBuffer

该类继承了AbstractChannelBuffer类，该类是动态的通道缓存区类，也就是该类是从ChannelBufferFactory工厂中动态的生成缓冲区，默认使用的工厂是HeapChannelBufferFactory。

1.属性和构造方法

```
/**
 * 通道缓存区工厂
 */
private final ChannelBufferFactory factory;

/**
 * 通道缓存区
 */
private ChannelBuffer buffer;

public DynamicChannelBuffer(int estimatedLength) {
    // 默认是HeapChannelBufferFactory
    this(estimatedLength, HeapChannelBufferFactory.getInstance());
}

public DynamicChannelBuffer(int estimatedLength, ChannelBufferFactory factory) {
    // 如果预计长度小于0 则抛出异常
    if (estimatedLength < 0) {
        throw new IllegalArgumentException("estimatedLength: " + estimatedLength);
    }
    // 如果工厂为空，则抛出空指针异常
    if (factory == null) {
        throw new NullPointerException("factory");
    }
    // 设置工厂
    this.factory = factory;
```

可以看到，该类有两个属性，所有的实现方法都是调用了buffer的方法，不过该buffer产生是通过工厂动态生成的。并且从构造方法来看，默认使用HeapChannelBufferFactory。

2.ensureWritableBytes

```
@Override
public void ensureWritableBytes(int minWritableBytes) {
    // 如果最小写入的字节数不大于可写的字节数，则结束
    if (minWritableBytes <= writableBytes()) {
        return;
    }

    // 新增容量
    int newCapacity;
    // 此缓冲区可包含的字节数等于0。
    if (capacity() == 0) {
        // 新增容量设置为1
        newCapacity = 1;
    } else {
        // 新增容量设置为缓冲区可包含的字节数
        newCapacity = capacity();
    }
    // 最小新增容量 = 当前的写索引+最小写入的字节数
    int minNewCapacity = writerIndex() + minWritableBytes;
    // 当新增容量比最小新增容量小
    while (newCapacity < minNewCapacity) {
        // 新增容量左移1位，也就是加倍
        newCapacity <<= 1;
    }

    // 通过工厂创建该容量大小当缓冲区
```

该方法是确保数组有可写的容量，该方法是重写了父类的方法，通过传入一个最小写入的字节数，来对缓冲区进行扩容，可以看到，当现有的缓冲区不够大的时候，会对缓冲区进行加倍对扩容，直到buffer的大小大于传入的最小可写字节数。

3.copy

```
@Override
public ChannelBuffer copy(int index, int length) {
    // 创建缓冲区，预计长度最小为64，或者更大
    DynamicChannelBuffer copiedBuffer = new DynamicChannelBuffer(Math.max(length, 64), factory());
    // 复制数据
    copiedBuffer.buffer = buffer.copy(index, length);
    // 设置索引，读索引设置为0，写索引设置为copy的数据长度
    copiedBuffer.setIndex(0, length);
    // 返回缓存区
    return copiedBuffer;
}
```

该方法是复制数据，在创建缓冲区的时候，预计长度最小是64，，然后重新设置读索引写索引。

其他方法都调用了buffer的方法或者调用了父类的方法，所以不再这里多说。

（四）ByteBufferBackedChannelBuffer

该方法继承AbstractChannelBuffer，该类是基于 Java NIO中的ByteBuffer来实现相关的读写数据等操作。

```
/**
 * ByteBuffer实例
 */
private final ByteBuffer buffer;

/**
 * 容量
 */
private final int capacity;

public ByteBufferBackedChannelBuffer(ByteBuffer buffer) {
    if (buffer == null) {
        throw new NullPointerException("buffer");
    }

    // 创建一个新的字节缓冲区，新缓冲区的大小将是此缓冲区的剩余容量
    this.buffer = buffer.slice();
    // 返回buffer的剩余容量
    capacity = buffer.remaining();
    // 设置写索引
    writerIndex(capacity);
}
```

上述就是该类的属性和构造函数，可以看到它有一个ByteBuffer类型的实例，并且capacity是buffer的剩余容量。

还有其他的方法比如getByte方法是从buffer中读取数据方法，setBytes方法是把数据写入buffer，它们都有很多重载方法，为就不一一讲解了，它们都是调用了ByteBuffer中的一些方法，如果对于Java NIO中的ByteBuffer方法不是很熟悉的朋友，需要先了解一下Java NIO中的ByteBuffer。

（五）HeapChannelBuffer

该方法继承了AbstractChannelBuffer，该类中buffer是基于字节数组实现

```
/**
 * The underlying heap byte array that this buffer is wrapping.
 * 此缓冲区包装的基础堆字节数组。
 */
protected final byte[] array;

/**
 * Creates a new heap buffer with a newly allocated byte array.
 * 使用新分配的字节数组创建新的堆缓冲区。
 *
 * @param length the length of the new byte array
 */
public HeapChannelBuffer(int length) {
    this(new byte[length], 0, 0);
}

/**
 * Creates a new heap buffer with an existing byte array.
 * 使用现有字节数组创建新的堆缓冲区。
 *
 * @param array the byte array to wrap
 */
public HeapChannelBuffer(byte[] array) {
    this(array, 0, array.length);
}
```

该类有好几个构造函数，都是基于字节数组的，也就是在该类中包装了一个字节数组，把构造函数传入的字节数组传入到该属性中。其他方法逻辑比较简单。

（六）ChannelBufferFactory

```java
public interface ChannelBufferFactory {

    /**
     * 获得缓冲区实例
     * @param capacity
     * @return
     */
    ChannelBuffer getBuffer(int capacity);

    ChannelBuffer getBuffer(byte[] array, int offset, int length);

    ChannelBuffer getBuffer(ByteBuffer nioBuffer);

}
```

该接口是通道缓冲区工厂，其中就只定义了获得通道缓冲区的方法，比较好理解，它有两个实现类，我后续会讲到。

（七）HeapChannelBufferFactory

该类实现了ChannelBufferFactory，该类就是基于字节数组来创建缓冲区的工厂。

```java
public class HeapChannelBufferFactory implements ChannelBufferFactory {

    /**
     * 单例
     */
    private static final HeapChannelBufferFactory INSTANCE = new HeapChannelBufferFactory();

    public HeapChannelBufferFactory() {
        super();
    }

    public static ChannelBufferFactory getInstance() {
        return INSTANCE;
    }

    @Override
    public ChannelBuffer getBuffer(int capacity) {
        // 创建一个capacity容量的缓冲区
        return ChannelBuffers.buffer(capacity);
    }

    @Override
    public ChannelBuffer getBuffer(byte[] array, int offset, int length) {
        return ChannelBuffers.wrappedBuffer(array, offset, length);
    }
```

该类利用了单例模式，其中的方法比较简单，就是调用了ChannelBuffers中的方法，调用的方法实际上还是使用了HeapChannelBuffer中创建缓冲区的方法。

（八）DirectChannelBufferFactory

该类实现了ChannelBufferFactory接口，是直接缓冲区工厂，用来创建直接缓冲区。

```java
public class DirectChannelBufferFactory implements ChannelBufferFactory {

    /**
     * 单例
     */
    private static final DirectChannelBufferFactory INSTANCE = new DirectChannelBufferFactory();

    public DirectChannelBufferFactory() {
        super();
    }

    public static ChannelBufferFactory getInstance() {
        return INSTANCE;
    }

    @Override
    public ChannelBuffer getBuffer(int capacity) {
        if (capacity < 0) {
            throw new IllegalArgumentException("capacity: " + capacity);
        }
        if (capacity == 0) {
            return ChannelBuffers.EMPTY_BUFFER;
        }
        // 生成直接缓冲区
        return ChannelBuffers.directBuffer(capacity);
    }
```

该类中的实现方式与HeapChannelBufferFactory中的实现方式差不多，唯一的区别就是它创建的是一个直接缓冲区。

（九）ChannelBuffers

该类是缓冲区的工具类，提供创建、比较 ChannelBuffer 等公用方法。我在这里举两个方法来讲：

```java
public static ChannelBuffer wrappedBuffer(ByteBuffer buffer) {
    // 如果缓冲区没有剩余容量
    if (!buffer.hasRemaining()) {
        return EMPTY_BUFFER;
    }
    // 如果是字节数组生成的缓冲区
    if (buffer.hasArray()) {
        // 使用buffer的字节数组生成一个新的缓冲区
        return wrappedBuffer(buffer.array(), buffer.arrayOffset() + buffer.position(), buffer.remaining());
    } else {
        // 基于ByteBuffer创建一个缓冲区（利用buffer的剩余容量创建）
        return new ByteBufferBackedChannelBuffer(buffer);
    }
}
```

该方法通过buffer来创建一个新的缓冲区。可以看出来调用的就是上述生成缓冲区的三个类中的方法，ChannelBuffers中很多方法都是这样去实现的，逻辑比较简单。

```
public static boolean equals(ChannelBuffer bufferA, ChannelBuffer bufferB) {
    // 获得bufferA的可读数据
    final int aLen = bufferA.readableBytes();
    // 如果两个缓冲区的可读数据大小不一样，则不是同一个
    if (aLen != bufferB.readableBytes()) {
        return false;
    }

    final int byteCount = aLen & 7;

    // 获得两个比较的缓冲区的读索引
    int aIndex = bufferA.readerIndex();
    int bIndex = bufferB.readerIndex();

    // 最多比较缓冲区中的7个数据
    for (int i = byteCount; i > 0; i--) {
        // 一旦有一个数据不一样，则不是同一个
        if (bufferA.getByte(aIndex) != bufferB.getByte(bIndex)) {
            return false;
        }
        aIndex++;
        bIndex++;
    }

    return true;
}
```

该方法就是比较两个缓冲区是否为同一个，重写了equals。

（十）ChannelBufferOutputStream

该类继承了OutputStream

1.属性和构造方法

```
/**
 * 缓冲区
 */
private final ChannelBuffer buffer;
/**
 * 记录开始写入的索引
 */
private final int startIndex;

public ChannelBufferOutputStream(ChannelBuffer buffer) {
    if (buffer == null) {
        throw new NullPointerException("buffer");
    }
    this.buffer = buffer;
    // 把开始写入数据的索引记录下来
    startIndex = buffer.writerIndex();
}
```

该类中包装了一个缓冲区对象和startIndex，startIndex是记录开始写入的索引。

2.writtenBytes

```
public int writtenBytes() {
    return buffer.writerIndex() - startIndex;
}
```

该方法是返回写入了多少数据。

该类里面还有write方法，都是调用了buffer.writeBytes。

（十一）ChannelBufferInputStream

该类继承了InputStream

1.属性和构造函数

```
/**
 * 缓冲区
 */
private final ChannelBuffer buffer;
/**
 * 记录开始读数据的索引
 */
private final int startIndex;
/**
 * 结束读数据的索引
 */
private final int endIndex;

public ChannelBufferInputStream(ChannelBuffer buffer) {
    this(buffer, buffer.readableBytes());
}

public ChannelBufferInputStream(ChannelBuffer buffer, int length) {
    if (buffer == null) {
        throw new NullPointerException("buffer");
    }
    if (length < 0) {
        throw new IllegalArgumentException("length: " + length);
    }
    if (length > buffer.readableBytes()) {
        throw new IndexOutOfBoundsException();
```

该类里面包装了读开始索引和结束索引，并且在构造方法中初始化这些属性。

2.readBytes

```
public int readBytes() {
    return buffer.readerIndex() - startIndex;
}
```

该方法是返回读了多少数据。

3.available

```
@Override
public int available() throws IOException {
    return endIndex - buffer.readerIndex();
}
```

该方法是返回还剩多少数据没读

4.read

```
@Override
public int read() throws IOException {
    if (!buffer.readable()) {
        return -1;
    }
    return buffer.readByte() & 0xff;
}

@Override
public int read(byte[] b, int off, int len) throws IOException {
    // 判断是否还有数据可读
    int available = available();
    if (available == 0) {
        return -1;
    }

    // 获得需要读取的数据长度
    len = Math.min(available, len);
    buffer.readBytes(b, off, len);
    return len;
}
```

该方法是读数据，返回读了数据长度。

5.skip

```
@Override
public long skip(long n) throws IOException {
    if (n > Integer.MAX_VALUE) {
        return skipBytes(Integer.MAX_VALUE);
    } else {
        return skipBytes((int) n);
    }
}

private int skipBytes(int n) throws IOException {
    int nBytes = Math.min(available(), n);
    // 跳过一些数据
    buffer.skipBytes(nBytes);
    return nBytes;
}
```

该方法是跳过n长度来读数据。

后记

> 该部分相关的源码解析地址：https://github.com/CrazyHZM/i...

该文章讲解了Buffer的相关实现逻辑,我很多方法都没有贴出源码，因为很多都是基于Java NIO的ByteBuffer都设计实现，并且要注意AbstractChannelBuffer的三个子类，也就是生成缓冲区的三种形式，还有就是要注意两个创建缓冲区实例的工厂。下一篇我会讲解telnet部分。如果我在哪一部分写的不够到位或者写错了，欢迎给我提意见，我的私人微信号码：HUA799695226。

crazyhzm

265

关注作者