`OP_READ` 注册的过程没啥太多可说的了，这里不再关注，

## 从 Channel 中读取数据并存储

入口代码如下：

**IoSocketDispatcher.run()**

```
1.  public void run() {
2.      while(isOpen.get()) {
3.          if (eventCount > 0) {
4.              // XXX 处理读写事件
5.              handleReadWriteKeys();
6.          }
7.      }
8.  }
```

**IoSocketDispatcher.handleReadWriteKeys()** 方法：

```
1.  private void handleReadWriteKeys() {
2.      ...
3.
4.      if (eventKey.isValid() && eventKey.isReadable()) {
5.          onReadableEvent(socketHandler);
6.      }
7.
8.  }
```

**IoSocketHandler.onReadableEvent()** 方法：

```
1.  // 从 channel 中读取数据, 数组中就单个的 ByteBuffer, 存放此次读取到的数据
2.  ByteBuffer[] received  = readSocket();
3.
4.  // 调用 NonBlockingConnection.IoHandlerCallback#onData() 方法
5.  // 将数据添加到 ReadQueue 队列
6.  getPreviousCallback().onData(received, size);
```

`NonBlockingConnection.IoHandlerCallback#onData() 方法` 方法中会调用 `AbstractNonBlockingStream.appendDataToReadBuffer()`：

```
1.  // 读取到的数据队列
2.  private final ReadQueue readQueue = new ReadQueue();
3.
4.  protected final void appendDataToReadBuffer(ByteBuffer[] data, int size) {
5.      readQueue.append(data, size);
6.      onPostAppend();
7.  }
```

`readQueue.append()` 最终会调用 `ReadQueue.Queue.append()` 方法：

```
1.  private static final class Queue implements ISource {
2.      // queue
3.      private ByteBuffer[] buffers = null;    // 所有读取的数据
4.
```

```
        public synchronized void append(ByteBuffer[] bufs, int size) {
            isAppended = true;
            version++;

            if (buffers == null) {
                buffers = bufs;
                currentSize = size;
            }  else {
                currentSize = null;

                // 数据复制
                // 上一次和本次
                ByteBuffer[] newBuffers = new ByteBuffer[buffers.length + bufs.length];
                // 将 buffers 中的数据复制到 newBuffers 中.
                // buffers 存放的是上一次为止的数据.
                // 现在 newBuffers 中的最后 bufs.length 个元素为null
                System.arraycopy(buffers, 0, newBuffers, 0, buffers.length);
                // 填充 newBuffers 中的最后 bufs.length 个元素为本次读取的 bufs
                System.arraycopy(bufs, 0, newBuffers, buffers.length, bufs.length);
                buffers = newBuffers;
            }
        }

    }
```

从上面可以看到所有的数据都被存放到了 `Queue` 的 `buffers` 中.值得考虑一下的是，每次都进行数组的拷贝，性能是否有些低效，使用 `List` 的方式是否更好些.

## 读取到的数据过多，暂停读取

暂停读取数据其实也就是先注销 `OP_READ` 事件.

这里只要回过头来看看之前的 `AbstractNonBlockingStream.appendDataToReadBuffer()` 方法:

```
    protected final void appendDataToReadBuffer(ByteBuffer[] data, int size) {
        readQueue.append(data, size);
        onPostAppend();
    }
```

`readQueue.append(data, size)` 之前已经说过，关注一下 `onPostAppend()`:

**NonBlockingConnection.onPostAppend() :**

```
    protected void onPostAppend() {
        if (getReadQueueSize() >= maxReadBufferSize) {
            // 读取的数据太多，暂停读取
            ioHandler.suspendRead();
        }
    }
```

**IoSocketHandler.suspendRead()** 方法:

```
    public void suspendRead() throws IOException {
        dispatcher.suspendRead(this);
    }

    public void suspendRead(final IoSocketHandler socketHandler) throws IOException {
```

```
        addKeyUpdateTask(new UpdateReadSelectionKeyTask(socketHandler, false));
    }
```

可以看到 `suspendRead()` 会添加一个 `UpdateReadSelectionKeyTask` 任务到队列中去:

**UpdateReadSelectionKeyTask**

```
    private final class UpdateReadSelectionKeyTask implements Runnable {

        public void run() {
            try {
                if (isSet) {
                    setReadSelectionKeyNow(socketHandler);
                } else {
                    // 因此传入的 isSet 为 false, so 执行这里
                    unsetReadSelectionKeyNow(socketHandler);
                }

            }
        }
    }
```

**unsetReadSelectionKeyNow()**

```
    private void unsetReadSelectionKeyNow(final IoSocketHandler socketHandler) throws IOExcepti
    on {
        SelectionKey key = getSelectionKey(socketHandler);
        if (key != null) {
            if (isReadable(key)) {
                // 注销 OP_READ 事件
                key.interestOps(key.interestOps() & ~SelectionKey.OP_READ);
            }
        }
    }
```

后续的操作也就是回调我们的 `handler` 的 `onData()` 方法，读取队列中的数据，进行业务逻辑的处理.但是我们发现并没有进行 `OP_READ` 注销的操作.

## 总结

xSocket 中对读事件的处理和 Tomcat、Jetty 所不同的，不会再读取到数据之后就注销 `OP_READ` 事件，而且将读取到的数据放到一个队列中，只有在读取到数据超过限制的情况下才会先注销 `OP_READ` 事件.

貌似应该写下 xSocket 和 Tomcat、Jetty 处理方式各自的利弊，后面再写吧.-_-#