二、Tomcat响应数据过程

当我们在Servlet中调用如下方法

```
OutputStream outputStream = resp.getOutputStream();
outputStream.write("test".getBytes());
```

resp对应的类型为ResponseFacade，得到的outputStream的类型为**CoyoteOutputStream**。
所以响应数据是通过CoyoteOutputStream这个类处理的。
当调用outputStream的write方法写数据时，实际调用的就是CoyoteOutputStream类的write(**byte**[] b)方法。

```
@Override
public void write(byte[] b)
    throws IOException {
    write(b, 0, b.length);
}
@Override
public void write(byte[] b, int off, int len)
    throws IOException {
    ob.write(b, off, len);
}
```

在CoyoteOutputStream类中有一个属性是ob，类型为org.apache.catalina.connector.OutputBuffer，该属性是在构造CoyoteOutputStream对象时初始化的。先注意OutputBuffer所在的包。

我们在调用write方法时，实际就是调用OutputBuffer的write方法，而write方法实际调用的就是该类中的writeBytes(byte b[], int off, int len)：

```
private void writeBytes(byte b[], int off, int len)
      throws IOException {

    if (closed) {
        return;
    }
```

```
7
8       bb.append(b, off, len);
9       bytesWritten += len;
10
11      // if called from within flush(), then immediately flush
12      // remaining bytes
13
14      if (doFlush) {
15          // 那么每次write都把缓冲中的数据发送出去
16          bb.flushBuffer();
17      }
18  }
```

在OutputBuffer中有一个属性叫做bb，类型是ByteChunk。在Tomcat响应流程中，可以把ByteChunk类当作一个缓冲区的实现，该类中有一个字节数组，名字叫做**buff**，默认大小为8192。

当我们在write字节数据时，就是把数据添加到ByteChunk对应的缓冲区buff中。当把数据添加到缓冲区后，如果有其他线程在执行outputSteam的flush()方法，则doFlush为true，那么则会调用bb.flushBuffer()。

这里就要考虑一个问题，我们把数据都写到了缓冲区buff中，那么buff中的数据是何时传递给socket中的呢?

在ByteChunk中有一个属性**out**，类型是ByteOutputChannel，它表示缓冲区中的数据该向流向哪个渠道，为了方便理解，可以先理解为渠道就是socket，表示把缓冲区中的数据发送给socket，当实际情况并不是，暂且这么理解。

ByteOutputChannel类中有一个方法realWriteBytes(**byte** buf[], **int** off, **int** len)，当调用**out**.realWriteBytes(src, off, len)方法时，就会把src数据发送给对应驱动

在当前这个ByteChunk中，它的out对应的仍然还是org.apache.catalina.connector.OutputBuffer，在这个类中存在该方法：

```
1  public void realWriteBytes(byte buf[], int off, int cnt)
2          throws IOException {
3      if (closed) {
4          return;
5      }
6      if (coyoteResponse == null) {
```

```
            return;
        }
        // If we really have something to write
        if (cnt > 0) {
            // real write to the adapter
            outputChunk.setBytes(buf, off, cnt);
            try {
                coyoteResponse.doWrite(outputChunk);
            } catch (IOException e) {
                // An IOException on a write is almost always due to
                // the remote client aborting the request.  Wrap this
                // so that it can be handled better by the error disp
atcher.
                throw new ClientAbortException(e);
            }
        }
    }
```

该方法中通过一个outputChunk来标记数据，表示标记的这些数据是要发送给socket的。而真正的发送逻辑交给了coyoteResponse.doWrite(**outputChunk**)来进行处理，coyoteResponse的类型为org.apache.coyote.Response。

具体怎么将缓冲区中所标记的数据怎么发送出去的，我们等会再看，我们先来看到底何时会触发这个发送动作。

ByteChunk中有一个方法append，表示向缓冲区buff中添加数据，其中有一个逻辑，当缓冲区满了之后就会调用**out**.realWriteBytes(src, off, len)，表示把缓冲区中的数据发送出去。缓存区的大小有一个限制，可以修改，默认为8192。

还有一种情况就算缓冲区没有满，但是在write之前调用用过flush方法，那么本次write的数据会先放入缓冲区，然后再把缓冲区中的数据发送出去。

当我们调用outputStream的flush方法时：

1. 先判断是否发送过响应头，没有发送则先发送响应头
2. 再调用ByteChunk的flushBuffer方法，把缓冲区中剩余的数据发送出去
3. 因为上文中我们所理解的将缓冲区的数据发送出去，是直接发送给socket，但实际情况是把数据发送给另外一个缓冲区，这个缓冲区也是用ByteChunk类实现的，名字叫做socketBuffer。所以当我们在使用flush方法时就需要把socketBuffer中的数据真正发送给socket。

接下来我们来看看coyoteResponse.doWrite(outputChunk);的具体实现细节。
该发方法实际调用的是**outputBuffer**.doWrite(chunk, **this**);这里的outputBuffer的类型是InternalOutputBuffer，在执行doWrite方法时，调用的是父类AbstractOutputBuffer的doWrite方法。

该doWrite方法中，首先会判断响应头是否已经发送，如果没有发送，则会构造响应头，并发响应头发送给socketBuffer，发送完响应头，会调用响应的output的**activeFilters**，对于不同的响应体需要使用不同的发送逻辑。比如ChunkedOutputFilter是用来发送分块响应体的，IdentityOutputFilter是用来发送Content-length响应体的，VoidOutputFilter不会真正的把数据发送出去。

在构造响应头时，会识别响应体应该通过什么OutputFilter来发送，如果响应中存在content-length那么则使用IdentityOutputFilter来发送响应体，否则使用ChunkedOutputFilter，当然还有一些异常情况下会使用VoidOutputFilter，表示不会发送响应体。

那现在的问题的，响应体的Content-length是在什么时候确定的?
答案是：当请求在servlet中执行完成后，会调用response.finishResponse()方法，该方法会调用**outputBuffer**.close()，该outputBuffer就是org.apache.catalina.connector.OutputBuffer，该方法会判断响应体是否已发送，如果在调用这个close时响应头还没有发送，则表示响应体的数据在之前一直没有发送过，一直存在了第一层缓冲区中，并且一直没有塞满该缓冲区，因为该缓冲区如果被塞满了，则会发送响应头，所以当执行到close方法是，响应头还没发送过，那么缓冲区中的数据就是响应体全部的数据，即，缓冲区数据的长度就是content-length。
反之，在调用close方法之前，就已经发送过数据了，那么响应头中就没有content-length，就会用ChunkedOutputFilter来发送数据。

并且在执行close方法时，会先将响应头的数据发送给socketbuffer，然后将第一层缓冲区的数据通过对应的OutputFilter发送给socketbuffer，然后调用OutputFilter的end方法，IdentityOutputFilter的end方法实现很简单，而ChunkedOutputFilter的end方法则相对做的事情更多一点，因为ChunkedOutputFilter的doWrite一次只会发送一块数据，所以end要负责循环调用doWrite方法，把全部的数据库发送完。

最后将socketbuffer中的数据发送给socket。