TCP 中的半包、粘包问题，网络应用程序是必须要进行处理的，否则会出现问题，通常情况下有如下几种处理方式：

- 基于分隔符，很常见的，比如基于换行
- 定长分隔符，每次报文读取固定大小的字节
- 消息头-消息体，消息头中指明消息体的长度
- 其它的方式

HTTP 请求中如何处理半包、粘包问题

HTTP 的请求格式大家都很清楚：

```
1.  请求方法 请求URI HTTP版本
2.  请求头
3.
4.  请求体
```

下面来讨论该如何使用上面常见的方法来处理 HTTP 请求的信息.其实是混合了上面的几种方式.

- 请求行：分隔符(空行 、 换行)
- 请求头： : 或 : (有一个空格) 来分隔请求头中的 name 和 value，多个请求头，以 换行 分隔，两次换行 意味着请求头解析结束.请求行和请求体都是基于 分隔符 来进行处理,还有就是得对一些非法的字符要进行处理.另外需要注意的一点就是请求头也是可以换行的，换行后跟着一个 tab .
- 请求体：如果要解析请求体中的数据，要么首先是 POST 或 PUT 方法，其次，请求头要包含 Content-Type (值要包含 application/x-www-form-urlencoded 或 application/x-www-form-urlencoded),另外要使用请求头 Content-Length 指明消息体的长度.那么 HTTP 服务器可以以此来读取多少数据算是解析完了请求体中的数据.这类似于 消息头(Content-Length)-消息体 的方式.对于参数名称和值,使用分隔符 = ,多个使用 & 分隔.

上面的处理对于 HTTP/1.0 与 HTTP/1.1 基本都适用.不过也有一些细节问题,比如 HTTP/1.0 是可以没有 host 请求中,那么就意味着,可能 HTTP请求行 + 两个换行 就意味着一次完整的 HTTP 请求.

好了,接下来就看看 Jetty 是如何处理本文所要讨论的问题的.

Jetty 半包、粘包的处理

Jetty 中连接的数据读取入口为: HttpConnection#onFillable() 方法.先从整体来看,然后再看一些细节.

从整体脉络来看

假设我们使用如下的半包消息和粘包消息：

半包消息

```
1.  // 先发送这部分的数据
2.  GET /my-web-test/index.jsp HTTP/1.1
3.
4.  // 休眠一段时间
5.
6.  // 接着发送余下的数据
7.  Host: localhost
8.  \r\n
```

注意，**休眠的时间不能太长**,在 Jetty 中,一个连接,默认情况下 30s 如果没有操作(比如从客户端读取数据),那么超过这个时间之后就会关闭和客户端的连接.

粘包消息

```
// 第一个请求
POST /my-web-test/getPostParams.jsp HTTP/1.1
Host: localhost:8080
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
// name=Tom&age=22 为 15 个字节. 这里指明的长度与其一致
Content-Length: 15

name=Tom&age=22
// 第二个请求
GET /my-web-test/index.jsp HTTP/1.1
Host: localhost:8080
\r\n
```

注意，`HttpConnection` 关联了一个 `HttpParser`，这个 `HttpParser` 中会保存解析到 HTTP 请求的哪个阶段,并保存解析到的相关数据.另外，`HttpConnection` 还有一个 `ByteBuffer` 类型的 `_requestBuffer`，从 channel 中读取到的数据将会保存到这个 `buffer` 中.然后会解析 `buffer` 中的数据.

半包的情况

如果读取到的数据已经构成了一个完整的请求,那么会调用 `_channel.handle()` 进行请求处理.如果没有构成一个完整的请求,那么会继续读取(已经读取到数据会被 `HttpParser` 解析，并进行相关部分的存储，`HttpParser` 中有很多变量来存储已经解析到的数据，如 `_method`、`_version` 等),如果此时读取不到数据,从 channel 中读取到的字节为 `0`，那么会结束读取，释放缓冲区，并重新注册 `OP_READ` 事件,等待数据到来,如果有数据,那么又会继续调用 `HttpConnection#onFillable()` 方法.如果读取到的字节数返回的是 `-1`，会关闭对应的 channel.

粘包的情况

上面已经说过了，如果构成了一个完整的请求，那么会调用 `_channel.handle()` 方法来处理，通常情况下，`suspended = !_channel.handle() = false`，那么此时还会继续进入 `while` 循环,因为缓冲区中还有数据，那么会继续解析这部分的数据(也就是后面的请求中的数据).如此反复.

其实,可以看到,半包、粘包可能是混合在一起出现的.

从细节来看

数据的读取和解析

这里将 `HttpConnection#onFillable()` 方法简化一下:

```
@Override
public void onFillable() {
    boolean suspended = false;
    try {
        while (!suspended && getEndPoint().getConnection() == this) {
            // buffer 为空 -> 可以填充数据
                        // 不为空，说明有数据，可能是由于粘包导致缓冲区中留有接下来的请求的数据,
那么就直接解析这部分的数据
```

```
                if (BufferUtil.isEmpty(_requestBuffer)) {
                    // 如果之前的迭代填充了 0 字节或遇到了 close, 在这里 break, 结束 while
                    if (filled <= 0)
                        break;

                    // channel 可能已经关闭
                    if (getEndPoint().isInputShutdown()) {
                        filled = -1;
                        _parser.atEOF();    // 结束
                    } else {
                        // 分配缓冲区
                        _requestBuffer = getRequestBuffer();

                        // 读取到数据(>0),不算超时;-1,则关闭 channel
                        filled = getEndPoint().fill(_requestBuffer);
                        if (filled == 0)
                            filled = getEndPoint().fill(_requestBuffer);

                        // tell parser
                        if (filled < 0)
                            _parser.atEOF();
                    }
                }

                // 解析缓冲区
                if (_parser.parseNext(_requestBuffer == null ? BufferUtil.EMPTY_BUFFER : _reque
    stBuffer)) {
                    // 请求处理,已经构成了一次完整的请求
                    // 通常情况下返回 false(suspended=false), 继续进入 while 进行处理
                    suspended = !_channel.handle();
                } else {
                    // 释放缓冲区
                    releaseRequestBuffer();

                    if (_parser.isClose())
                        close();
                }
            }
        } catch (EofException e) {
        } catch (Exception e) {
            close();
        } finally {
            setCurrentConnection(last);
            if (!suspended && getEndPoint().isOpen() && getEndPoint().getConnection() == this)
    {
                // 最后会重新注册 OP_READ 事件
                fillInterested();
            }
        }
    }
```

ChannelEndPoint#fill()

```
    @Override
    public int fill(ByteBuffer buffer) throws IOException {
        if (_ishut)
            return -1;

```

```
        int pos = BufferUtil.flipToFill(buffer);
        try {
            int filled = _channel.read(buffer);

            if (filled > 0)
                // 读取到了数据，不能算超时
                notIdle();
            else if (filled == -1)
                shutdownInput();    // 关闭 channel

            return filled;
        } catch (IOException e) {
            shutdownInput();    // 关闭 channel
            return -1;
        } finally {
            BufferUtil.flipToFlush(buffer, pos);
        }
    }
```

请求行

这里的解析方式基本就是按照我们之前所说的，基于 `空格` 、 `换行` ，但是要注意这个 `换行` ，是使用 `\r` 、 `\n` ，还是两者一起使用.在 Jetty 中，这部分是处理逻辑是，可以使用 `\n` 或 `\r\n` ，算作换行，但是单个的 `\r` 是不算的， `\r` 之后必须有 `\n` .以下是相关的代码：

HttpParser#next()

```
    private byte next(ByteBuffer buffer) {
        byte ch = buffer.get();

        // 遇到了 CR
        if (_cr) {
            if (ch != LINE_FEED)    // CR 之后必须是 LF
                throw new BadMessageException("Bad EOL");
            _cr = false;
            return ch;
        }

        ...

        return ch;
    }
```

请求行完整的源码可见 `HttpParser#quickStart()` 和 `HttpParser#parseLine()` 方法.看阅读 `HttpParser#parseNext()` 的如下部分：

```
    public boolean parseNext(ByteBuffer buffer) {
        // Start a request/response
        if (_state == State.START) {
            _version = null;
            _method = null;
            _methodString = null;
            _endOfContent = EndOfContent.UNKNOWN_CONTENT;
            _header = null;
            if (quickStart(buffer)) // 解析请求的方法,不过这里没有返回 true 的情况
                return true;
        }
```

```
12.
13.         // Request/response line
14.         if (_state.ordinal() >= State.START.ordinal() && _state.ordinal() < State.HEADER.ordina
    l()) {
15.             if (parseLine(buffer))
16.                 return true;
17.         }
18.
19.     }
```

请求头

按照 `:` 和 `: `(后面有一个空格)来划分 name 和 value.value 中如果有多个值的话使用 `;` 进行分隔. 遇到两次换行,请求头算是结束了.如果请求头被分割为多行的话,也就是说如果有如下的请求头. `loca` 后面是一个换行, `lhost` 前面是一个 `tab` .

```
1.  Host: loca
2.   lhost
```

那么之后我们取出来的 `Host` 的值为 `loca lhost` , 可以看到是不会被连接到一起的.另外,这部分的换行规则和请求行是一致的.

这部分的代码可以详细阅读 `HttpParser#parseHeaders()` 方法.

请求体

请求头结束之后就可以开始对请求进行处理了, 如果是 `POST` 请求,那么后面可能还有请求体,但是在解析完请求头之后是不会立即进行解析的, 只有当需要里面的数据的时候才进行解析,比如第一次调用 `request.getParameter()` 方法等,这时候才会去解析请求体中的数据.看下如下的方法片段:

```
1.  if (_parser.parseNext(_requestBuffer == null ? BufferUtil.EMPTY_BUFFER : _requestBuffer)) {
2.      suspended = !_channel.handle();
3.  }
4.
5.  public boolean parseNext(ByteBuffer buffer) {
6.      // parse headers
7.      if (_state.ordinal() >= State.HEADER.ordinal() && _state.ordinal() < State.CONTENT.ordi
    nal()) {
8.          if (parseHeaders(buffer))   // 这里解析完 headers 之后就直接返回 true, 而不是直接就去
    解析 content
9.              return true;
10.     }
11. }
```

之前说过如果要解析请求体,那么是需要一定条件的, 这里可以从 `Request.getParameter()` 方法入手,它会调用 `Request#extractContentParameters()` 方法:

```
1.  private void extractContentParameters() {
2.      // Content-Length 中指定的长度
3.      int contentLength = getContentLength();
4.      if (contentLength != 0) {
5.          // 普通的表单提交: application/x-www-form-urlencoded
6.          if (MimeTypes.Type.FORM_ENCODED.is(contentType) && _inputState == __NONE &&
7.                  // POST 方法或 PUT 方法
8.                  (HttpMethod.POST.is(getMethod()) || HttpMethod.PUT.is(getMethod()))) {
```

```
                extractFormParameters(_contentParameters);
            } else if (contentType.startsWith("multipart/form-data") &&
                    getAttribute(__MULTIPART_CONFIG_ELEMENT) != null &&
                    _multiPartInputStream == null) {
                // 文件上传: multipart/form-data
                extractMultipartParameters(_contentParameters);
            }
        }
    }
```

可以看到对于解析请求体需要满足的条件，`extractFormParameters()` 方法最终会调用 `HttpParser#parseContent()`(中间的调用过程省略)：

```
    case CONTENT: {
        long content = _contentLength - _contentPosition;
        if (content == 0) {
            setState(State.END);
            if (_handler.messageComplete())
                return true;
        } else {
            _contentChunk = buffer.asReadOnlyBuffer();

            // 限制 content 为期望的大小 -> 有可能是下一个请求的数据, 本次不应该读取
            if (remaining > content) {
                _contentChunk.limit(_contentChunk.position() + (int) content);
            }

            _contentPosition += _contentChunk.remaining();
            buffer.position(buffer.position() + _contentChunk.remaining());
            // HttpChannelOverHttp#content -> 返回 true
            if (_handler.content(_contentChunk))
                return true;

            if (_contentPosition == _contentLength) {
                setState(State.END);
                if (_handler.messageComplete())
                    return true;
            }
        }
        break;
    }
```