node-formidable详解

前言

最近在研究nodejs如何实现文件上传功能，偶然读到《nodejs-post文件上传原理详解》这篇教程，感觉非常给力。教程中详细解读了post数据格式，以及通过分析node-formidable模块的源代码来解读文件上传的原理。

可是，在源码分析的过程中，作者并没有解释得很到位，从而导致在看完整篇教程后某些地方还有些云里雾里，在潜心研究一番后，终于明了，然而这也催生了我写这篇文章的想法。

由于该文章是在上文所提教程的基础之上补充的，因此，建议读之前先看以上教程。

详解post数据的格式

使用post提交表单后，传递过来的数据到底是什么样的呢？

上篇教程中已经讲解过，可是，为了使之能更好地与后面的数据解析部分的代码相结合，我对Post数据进行了一个详细的标注

HTML

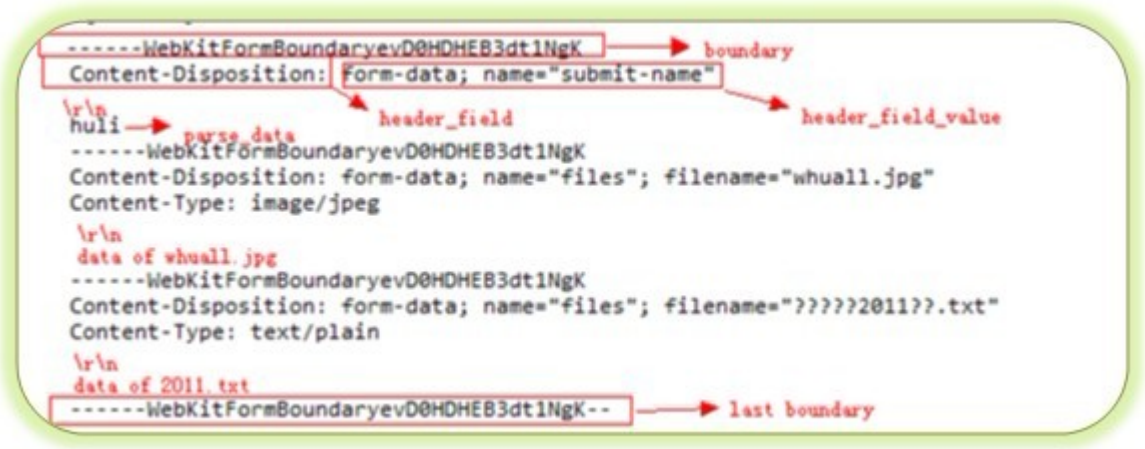
```
<body>
  <form method="post" action="test.php" enctype="multipart/form-data">
   What is your name? <input type="text" name="submit-name" /><br />
   What files are you sending? <input type="file" name="files" multiple="multiple" /><br />
   <input type="submit" value="send" />
  </form>
</body>
```

> 该html中有一个表单，其中包含了一个text文本框和一个file多文件上传控件。

POST HEADER

这是Post的header信息，请注意Content-Type后面的boundary数据，这就是边界字符串，用于分隔每个字段数据。

POST DATA



上图便是传递过来的POST数据部分，图中对各部分已经作了详细的表示，每一行其实隐含着一个\r\n（CR LF）。上面提到的教程中其实已经总结过POST数据的规则，这里再总结一次：

- 每一部分数据包括字段部分和数据部分，比如文本框数据，字段部分是ContentPosition，数据部分是huli;字段部分和数据部分使用一个空行分隔； 每一行后面实际隐含这一个\r\n（CR LF）
- 每一部分数据以边界字符串分隔，假设post_header中定义的边界为Aa03x，那么一般的边界字符串为–Aa03x\r\n（程序中称这样的边界字符串为PART_BOUNDARY,表示后面还有数据需要解析），最后面的边界字符串为–Aa03x–（程序中称这样的边界字符串为LAST_BOUNDARY，表示所有的数据都解析完毕）
- 在每一部分数据中，数据部分（记得前面提到过的，每一部分数据包括字段部分和数据部分吗？）是和接下来的边界字符串紧挨在一起的！

更加详细的信息可参考W3C文档的 FORMS

POST数据解析流程

模块中将POST数据的解析分成多个不同的状态，接着它会遍历buffer中每个字符的状态，进行相应的处理。 这些状态分为：

1. PARSER_UNINITIALIZED：系统尚未初始化
2. TART：开始解析
3. START_BOUNDARY：开始解析边界字符串
4. HEADER_FIELD_START：解析HEADER_FIELD前的预处理
5. HEADER_FIELD：开始解析HEADER_FIELD
6. HEADER_VALUE_START：解析HEADER_VALUE前的预处理
7. HEADER_VALUE：开始解析HEADER_VALUE
8. HEADER_VALUE_ALMOST_DONE:header_value部分解析完毕
9. HEADERS_ALMOST_DONE：所有的header解析完毕
10. PART_DATA_START：解析数据段前的预处理
11. PART_DATA：开始解析数据段
12. ART_END：数据段解析完毕
13. END:所有的POST_DATA解析完

结合**POST_DATA**中的图示来看这些状态， 注意：状态的流转是从上到下的

源码详解

下面，我会按照文件上传的原理把整个流程重新走一遍，以便大家更加理解。 其中，我会省略一些源码中对本篇文章来说并不太需要的部分，比如容错处理等等。 首要工作是进行一些基本模块的引入

```
var http = require('http');
var fs = require('fs');
var path = require('path');
var StringDecoder = require('string_decoder').StringDecoder;
var EventEmitter = require('events').EventEmitter;
var util = require('util');
// 首先创建一个服务器
http.createServer(function (request, response) {
 // 处理文件上传:TODO
}).listen(8080);

// 处理文件上传，首先定义一个用于文件上传的类,用于一些初始化设置
function IncomingForm() {
  if (!(this instanceof IncomingForm)){
    return new IncomingForm();
  }
  EventEmitter.call(this); // 使对象可以调用事件函数
  this.maxFieldsSize = 210241024; // 设定最大文件限制
  this.upLoadDir = '/tmp'; // 设定上传文件路径
  this.encoding = "utf-8"; // 设定文件的编码
  this.headers = null; // post数据的头部信息，可以使用request.headers获得
  this.type = null;
  this.bytesReceived = null; // 收到的字节数
  this.bytesExpected = null; // 预计的字节数，一般只有最终收到的字节数等于预计的字节数，传输才是正确的，否则，传输出错
  this._parser = null;  // 用于初始化处理post数据的类，等下会定义，类名为multipart_parser.js
  this.filedsSize = 0;
}
util.inherits(IncomingForm,EventEmitter);
exports.IncomingForm=IncomingForm; // 导出

// 我想用过node-formidable的童鞋都应该知道parser这个方法的重要性，没错，他正是上传文件模块的入口它有两个参数，第一个是request对象（大家都懂的），另一个就是一个自定义的回调函数cb
IncomingForm.prototype.parser = function (req, cb) {
  //定义两个函数，用于请求的暂定和继续。由于前一篇文章已经提到过，node-formidable是采用边接收边解析的方式，而接收到的文件数据往往先存放到buffer缓冲区，然后再写入指定路径的文件之中。因此，在写文件的过程中，最后先暂定请求（req.pause），等文件写入完毕，再继续请求（req.resume）,这是为了避免文件数据传输速度快于写入速度而导致缓冲区溢出，从而使得数据丢失
  this.pause = function () {
    req.pause();
  };
  this.resume = function() {
    req.resume();
  };
  this.writeHeaders(req.headers); //存储Post头部信息，因为要使用其中的content-type字段来判断传输的是字符串还是文件，稍后会定义
  var self=this;

  // 对这个我想大家不会陌生，由于传输的数据是分包的，而并非所有的数据一齐全部传过来，因此，没接收到一部分数据，就会触发一次data事件
  req.on('data', function (buffer) {
    self.write(buffer); // 用于处理传过来的数据

  });
```

```
  //如果定义了回调
  if (cb) {
    var fields = {}, files = {};
    // 定义一个field事件，当接收到一个完整字段时手动触发，该字段是指非文件字段数据。
    // 比如一个文本框中的值
    this.on('field', function (name, value) {
      fields[name] = value;
    });

    // 与上面的事件类似，区别是它只有接受完一个文件数据才手动触发，文件数据都是二进制数据，而字段数据往往是字符串
    this.on('file', function (name,file) {
      files[name]=file;
    });

    // 由事件名我们可以看出来，end事件是在接受完所有的数据后触发，例：
    // (new IncomingForm()).parse(request, function (error, field, files) {
    //       fs.renameSync(files.upload.path,/tmp);  用于重命名上传的文件
    // })
    this.on('end', function () {
      cb(null, field, files);
    });
  }

};

// 下面就定义刚刚提到的self.write函数，该函数用于处理传过来的数据
IncomingForm.prototype.write = function (buffer) {
  this.bytesReceived += buffer.length;

  // 由于处理post数据是个非常复杂的过程，因此，我们把它放在另个模块里来处理，模块名为multipart_parser，而这里的this._parser就是new mulipartParser()的一个实例化对象
  var bytesParsed=this._parser.write(buffer);
};
```

好的，这里先暂定`IncomingForm`模块的编写，让我们来探究一下神秘的post数据是到底是如何处理的呢

```
//multipart_parser.js
var s = 0;
var S = { //定义数据解析的各个阶段
  PARSER_UNINITIALIZED: s++,
  START: s++,
  START_BOUNDARY: s++,
  HEADER_FIELD_START: s++,
  HEADER_FIELD: s++,
  HEADER_VALUE_START: s++,
  HEADER_VALUE: s++,
  HEADER_VALUE_ALMOST_DONE: s++,
  HEADERS_ALMOST_DONE: s++,
  PART_DATA_START: s++,
  PART_DATA: s++,
  PART_END: s++,
  END: s++
};

// 由上图可知，不同的字段数据之间是以边界字符串分隔开的，而中间的边界字符串和最终的边界字符串不一样,最终的边界字符串的最后面多了两个’-’，因此，F对象的两个属性是为了识别当前的边界是中间的边界（PART_BOUNDARY）还是最后的边界（LAST_BOUNDARY）
var f = 1;
var F = {
  PART_BOUNDARY: f, // 用二进制表示为01
  LAST_BOUNDARY: f*=2 // 用二进制表示为10
},

//每个特殊字符对应的ascaii码
var LF = 10,
    CR = 13,
    SPACE = 32,
    HYPHEN = 45, // 短横杠’-’
    COLON = 58 // 冒号’:’
;
//定义的MultipartParser类，用于一些初始化设置
function MultipartParser () {
  this.boundary = null; // 边界字符串
  this.boundaryChars = null; // 一个边界字符串对，字符串中每个字母为它的一个键，值都是true
  this.lookbehind = null; // 稍后会讲到，是为了防止错误地识别边界字符串
  this.state = S.PARSER_UNINITIALIZED;
  this.index = null;
  this.flags = 0;
}

exports.MultipartParser=MultipartParser;

// 该函数用于初始化边界字符串，一般在刚开始的时候调用一次
MultipartParser.prototype.initWithBoundary = function (str) {
  // 一般，真正的边界字符串会在boundary字段值后另外加四个字母’-\r\n’, 例如，boundary=AaB03x，那么中间的边界字符串将会是–AaBo3x\r\n,结尾的边界字符串是–AaBo3x–,如上图
  this.boundary = new Buffer(str.length+4);


  // 可以看到，代码中把boundary写成\r\n-str,与事实-str\r\n不相符，这是出于什么考虑呢，在write函数中会有
```

```
介绍
  this.boundary.write('\r\n-','ascaii',0);
  this.boundary.write(str, 'ascii', 4);
  this.lookbehind = new Buffer(this.boundary.length+8);
  // 初始化为边界字符串后，buffer的解析状态流转为START
  this.state = S.START;
}

// 这是今天的主角，也是最重要的处理逻辑，还记得IncomingForm模块中的 self._write(buffer) 吗，没错它实际调用的正是这个函数。
MultipartParser.prototype.write = function (buffer) {
  // 首先进行一些初始化设置,定义一些局部变量
  var self = this,
      len = buffer.length,
      prevIndex = this.index,
      index = this.index,
      state = this.state,
      flags = this.flags,
      lookbehind = this.lookbehind,
      boundary = this.boundary,
      boundaryChars = this.boundaryChars,
      boundaryLength = this.boundaryLength,
      // 刚开始一直不明白为什么定义一个boundaryEnd,不过看到后面的代码，再自己琢磨一会，终于弄明白鸟
      boundaryEnd = boundaryLength - 1,
      bufferLength = buffer.length,
      c,

      // 这个函数在上面提到的文章中也有讲过，实际上就是作一个起点标识的，比如，我现在字段数据（headerField）读取完毕，要开始读字段值（headerValue）数据,那么我便使用一个mark('headerValue')来作一个字段值数据的起点标识,存储我是从哪一个字符开始读取字段值的
      mark = function (name) {
        self[name + 'Mark'] = i;
      },

      //顾名思义，该函数的作用是为了删除起点标识的，那何时该删除标识呢？ 当解析的数据部分的末尾在buffer内部时，这个时候，在读取完该部分数据后，需将起点标识删除，例如我现在读取的是字段数据部分，headerField,如果headerField部分在当前的buffer内部结束时，就应该删除该headerField的起点标识
      clear = function (name) {
        delete self[name + 'Mark'];
      },

      // 每解析完一段数据（我说的一段数据指的是一段有意义的数据，例如 headerField, headerValue, partData 都算是一段数据）就要调用相应的回调进行处理，该函数会传递四个参数，第一个参数为该段数据名（headerField等），第二个参数为当前的buffer，最后两个参数分别为该数据段的起始位置和结束位置。。这里又要分为两种情况：

      // 1. 该段数据在当前的buffer内结束，那么 start = mark(name), end = i;
      // 2. 当前的buffer内，解析的该段数据尚未解析完毕，例如当前是个headerField字段数据，它有一部分在当前的 buffer 内，剩余的部分在下一个buffer内，这时候，start = mark(name), end = buffer.length
      callback = function (name, buffer, start, end) {
        if(start != undefined && start === end) {
          return;

        }
```

```
        // 这是形成回调函数名
        var callbackSymbol = 'on' + name.substr(0, 1).toUpperCase() + name.substr(1);
        if (callbackSymbol in self) {
          //这些回调函数是在哪定义的呢？别急，等我们再次回到IncomingForm中时你就会明白了
          self[callbackSymbol](buffer, start, end);
        }
      },

      // 这是对callback的一个封装，目的是为了给callback传递start和end，这个clear代表什么呢？，没错，它代表我在讲callback时提到的两种情况，可以看下源码对照一下这两种情况。
      dataCallback = function (name, clear) {
        var markSymbol = name + 'Mark';
        if (!(markSymbol in self)) {
          return;
        }

        if (!clear) {
          callback(name, buffer, self[markSymbol], buffer.length);
          self[markSymbol] = 0;
        } else {
          callback(name, buffer, self[markSymbol], i);
          delete self[markSymbol];
        }
      }
  ;

  // 该进入主题部分了，在当前buffer内，逐个字符地判断，然后根据每个字符地当前状态进行相应地操作。这里提到的每个字符的状态是指上就是指当前字符出于哪一个数据段中
  for (i = 0; i < len; i++ ) {
    c = buffer[i]; // 当前字符
    switch (state) {

      case S.PARSER_UNINITIALIZED :
        return i; // 注意，该case并没有break语句，说明可以直接执行下一个case,在该case中，只是起到一个初始化作用，不涉及到字符的处理

      case S.START : // 数据开始进入解析阶段，一般是在初始化完边界字符串后，可以看initWithBoundary代码
        index = 0;
        state = S.START_BOUNDARY; // 状态流转，开始解析解析数据区域的边界字符串

      case S.START_BOUNDARY:
        if (index == boundary.length-2) {
          if (c != CR) { // 边界的倒数第二个字符串应该为CR（最后两个字符为\r\n）,如果不是，则说明解析出错
            return i;
          }
          index++; // 若正确，则继续解析下一个字符
          break;
        } else if (index-1 == boundary.length-2) {
          if (c != LF) { // 如果边界的最后一个字符部位LF,则说明解析出错
            return i;
          }

          //否则，说明边界字符串解析完毕，开始解析下面的字段数据
```

```
          index = 0;
          // 触发onPartBegin事件（在IncomingForm中定义）,目的是为了初始化一些变量，方便接下来的解析过程，可以看到，它没有传递buffer,start,end，因此不涉及传输数据的处理
          callback('partBegin');
          state=S.HEADER_FIELD_START; // 状态流转
          break;
        }

        // 为什么index要加2呢，还记得initWithBounday中如何初始化边界字符串吗？没错，它解析成\r\n-boundary,与实际的字符串-boundary\r\n对比一下，明白为什么要加上2了吧，是为了跳过CR和LF
        if (c != boundary[index+2]) {
          return i;
        }
        index++;
        break;

      case S.HEADER_FIELD_START: // 可以看到，该case同样没有break,因此，可以看出，凡是START区域都是起到一个起始点标识和状态流转的作用
        state = S.HEADER_FIELD; // 状态流转
        mark('headerField'); // 标识字段数据的起始点
        index=0;

      case S.HEADER_FIELD:
        //这里的CR指的是所有的HEADER结束后于DATA区域之间的那一个空行，因为有些数据有两个header,比如文件数据，它不仅有Content-Disposition，还有Content-Type，如上图。可能这里还不太明白，等到下面的S.HEADER_VALUE_ALMOST_DONE状态，再回过头来看这一段代码
        if (c == CR) {
          clear('headerField');
          state=S.HEADERS_ALMOST_DONE;
          break;
        }
        index++;
        // 如果字符是'-',例如Content-Diposition:中间的'-',则继续检测下一个字符的状态
        if (c == HYPHTN) {
          break;
        }
        // 如果字符是':'则表明header_field部分结束，可看图示
        if (c == COLON) {
          if (index == 1) { // 是个空字段
            return i;
          }
          // header_field 部分结束，调用相关的回调处理该部分数据，注意，它是在同一个buffer内部结束的，所有第二个参数为true
          dataCallback('headerField', true);
          state=S.HEADER_VALUE_START; // 状态流转，开始处理field_value的数据
          break;
        }
        break;

      case S.HEADER_VALUE_START:  // 不用解释了吧，呵呵
        mark('headerValue');
        state = S.HEADER_VALUE;
```

```
      case S.HEADER_VALUE:
        // 如果当前字符是CR，则HEADER_VALUE解析完毕，还记得每一行的最后是\r\n（CR LF）吗?
        if (c == CR) {
          dataCallback('headerValue', true);
          callback('headerEnd'); // 改行header结束后进行回调
          state = S.HEADER_VALUE_ALMOST_DONE;
        }
        break;

      case S.HEADER_VALUE_ALMOST_DONE:
        // 由于上一状态最后的字符是CR，所以当前字符应该是LF，否则就是解析出错
        if (c != LF) {
          return i;
        }
        // 为什么这里又要重新回滚状态呢？ 前面已经提到过，有的数据段可能有两行header，比如文件和图片，现在再讲前面 HEADER_FIELD 中的解释联系起来，是不是懂了呢
        state=S.HEADER_FIELD_START;
        break;

      case S.HEADERS_ALMOST_DONE: // 这是在某个数据段的所有Header全部解析完毕时的状态
        // 回过头去看header_field状态，在header都解析完成时的字符是CR，所以当前字符应该是LF，否则就是解析错误
        if (c != LF) {
          return i;
        }
        callback('headersEnd');
        state = S.PART_DATA_START;
        break;

      case S.PART_DATA_START:
        state=S.PART_DATA;
        mark('partData');

      case S.PART_DATA: // 数据段的解析，这是最难懂的一部分，当初也是看了很久
        prevIndex = index;
        //index上一次的设置是在header_field_start中，设置为0
        if (index == 0) {
          //这一段代码有些难懂，它的目的是尽快地跳过数据区域。因为数据区域一般有很多地字符，而我们所需要的仅仅是该数据区域的起始位置和结束位置。起始位置我们在PART_DATA_START中已经设置了。该如何找到数据区域的结束位置呢？如果像前面的headerField,headerValue一样，逐个字符进行遍历，那太影响效率，因此这里采用边界跳跃法使我们能更快地接近数据区域的结束位置。如何跳跃呢？此时i指向的是数据其余的第一个字符，而数据区域和下一个字段其余的边界是连在一起的。这时候，我们用i加上一个boundaryEnd(它比真实的边界少一个字符)可以上我们往前跳一部分，但是它又同时保证了我们不会跳出边界之外，就算数据区域只有一个字符，我加上一个boundaryEnd,也只会跳到边界字符串的最尾部。大家可以按照post数据格式，假设一段数据来照着流程走一遍
          i+=boundaryEnd;
          while (i < bufferLength && !(buffer[i] in boundaryChars)) {
            i+=boundaryLength;
          }
          i -= boundaryEnd;
          c=buffer[i];
        }


        if (index < boundary.length) {
```

```
          if (boundary[index] == c) {
            // 表示数据区域解析完毕，这里同时也说明了作者当时在初始化边界时为什么要把\r\n放在首部，在数据区域结尾的部分正好是\r\n，因此，如果两者相等，则正好说明了当前指针走到了数据区域的尾部
            if (index == 0) {
              dataCallback('partData',true);
            }
            // 这里开始解析下一个boundary，也就是紧接着data区域之后的boundary,可以边看代码边看图示
            index++;
          } else {
            index=0;
          }
        }
        else if (index == boundary.length) {
          // 当解析到边界字符串的倒数第二个字符：注意，因为index是从1开始的，所以当它等于boundary.length时应该是倒数第二个字符
          index++;
          if (c == CR) {
            // 如果倒数第二个字符为CR,表名下面还有数据需要解析，前面已经解释过，边界字符串有两种情况，可以看上面的图示
            flags = F.PART_BOUNDARY; // flag为F.PART_BOUNDARY
          } else if (c == HYPHEN) {
            //如果倒数第二个字符为'-',表明所有的post数据解析完毕
          } else {
            index=0;
          }
        }
        else if (index-1 == boundary.length) {
          if (flags & F.PART_BOUNDARY) { // 如果当前的边界字符串为PART_BOUNDARY
            index = 0;
            if (c == LF) {
              // 因为下面还有数据需要解析，所以重新初始化flags的值
              flags &= ~F.PART_BOUNDARY;
              callback('partEnd');
              callback('partBegin');
              // 状态重新流转到header_field_start进行下一个字段的解析
              state = S.HEADER_FIELD_START;
              break;
            }
          }
          else if (flags & F.LAST_BOUNDARY) { // 如果当前边界字符串为LAST_BOUNDARY
              if (c == HYPHEN) {
                callback('partEnd');
                callback('end');
                state = S.END; // 状态流转，解析结束
              } else {
                index=0;
              }
          } else {
            index=0;
          }
        }


        if (index > 0) {
```

```
            // 这里是在干吗呢？实际上它就是个日志功能，在解析遇到错误的时候，可以通过它来回滚错误。例如，在跳跃边界那一部分，当数据区域中有一段字符和边界字符串中的某一段字符相同时，就会错误地把它认为是边界字符串，等到解析到后来发现了错误时怎么办呢？就是用这里保存的数据来进行回滚操作
            lookbehind[index-1] = c
          } else if (prevIndex > 0) {
            // 可以看前面part_data部分的代码，在发现解析错误时，总是把index置0，如果这时候prevIndex不为0,那么说明有数据需要回滚
            callback('partData', lookbehind, 0, prevIndex);
            prevIndex = 0;
            mark('partData');
            i-;
          }
          break;

        case S.END:
          break;
        default
          return i;
      }
      // 这里是在当前buffer完全解析完后调用的,如果一个数据部分（比如如field信息）已经在上面的解析过程中解析完毕，那么自然已经调用过clear方法，那下面的dataCallback将什么也不做,否则，下面的调用将会把这次数据段中的数据部分传递到回调函数中，看看上面提到的调用dataCallback的两种情况
    dataCallback('headerField');
    dataCallback('headerValue');
    dataCallback('partData');
    this.index = index;
    this.state = state;
    this.flags = flags;
    return len;
  }
```

到这里，已经完成了整个数据解析部分的流程，如果大家有些地方还没弄明白，可以使用实例数据照着流程多走几次，这里，还有一个问题是，在解析的过程中，触发的这些回调都是在哪里定义的呢？ 让我么能再回到 IncomingForm 模块中。想一想，该何时定义这些回调..... 在post数据传过来时，会先传递头部信息，就是那些键值对，然后才是实际的数据部分（数据部分中又分header和data，上面的流程就是解析这两块）因此，我们可以在处理头部信息时来定义这些回调函数。

还记得parser中的`this.writeHeaders(req.headers)`这一句吗，不错，这正是处理Header信息的入口

```
IncomingForm.prototype.writeHeaders = function (headers) {
  this.headers=headers;
  // 调用一个私有方法
  this._parseContentType();
}

IncomingForm.prototype._parseContentType = function() {
  // headers中的content-type有application/x-www-form-urlencoded和multipart/form-data两种，由于这里专注讲文件上传，所以我们只讲multipart/form-data部分
  if (this.headers['content-type'].match(/multipart/i)) {
    var m;
    if (m = this.headers['content-type'].match(/boundary=(?:"("+)"|(;+))/i)) {
      // 获得boundary字段的指，也就是边界字符串
      this._initMultipart(m[1] || m[2]);
    }
    return
  }
}

IncomingForm.prototype._initMultipart = function (boundary) {
  this.type = 'multipart';
  // 初始化一个MultipartParser的实例，用于在后面定义multipart_parser中执行的回调
  var parser = new MultipartParser(),
      self = this,
      headerField,
      headerValue,
      part
  ;
  // 初始化边界字符串，initWithBoundary是在multipart_parser中定义的
  parser.initWithBoundary(boundary);

  //该函数是在状态流转到S.HEADER_FIELD_START时执行。
  parser.onPartBegin = function() {
    part = new EventEmitter();
    part.headers = {};
    part.name = null;
    part.filename = null;
    part.mime = null;
    headerField = '';
    headerValue = '';
  };

  // header_field解析完毕时执行
  parser.onHeaderField = function(b, start, end) {
    headerField += b.toString(self.encoding, start, end);
  };

  // header_value解析完毕时执行
  parser.onHeaderValue = function(b, start, end) {
    headerValue += b.toString(self.encoding, start, end);
  };

  //headr_value解析完毕时执行,用于存储该行header中的一些信息, 例如name,filename,filetype等等
```

```
  parser.onHeaderEnd = function() {
    headerField = headerField.toLowerCase();
    part.headers[headerField] = headerValue;

    var m;
    if (headerField == 'content-disposition') {
      if (m = headerValue.match(/name="([^"]+)"/i)) {
        part.name = m[1];
      }
      part.filename = self._fileName(headerValue);
    } else if (headerField == 'content-type') {
      part.mime = headerValue;
    }

    headerField = '';
    headerValue = '';
  };

  // 所有的header解析完毕时执行
  parser.onHeadersEnd = function() {
    self.onPart(part);
  };

  // 当找到data的结束位置时执行
  parser.onPartData = function(b, start, end) {
    //触发data事件，该事件是在哪侦听的呢，稍后会说明
    part.emit('data', b.slice(start, end));
  };

  // 当解析完紧接data区域之后的边界字符串时执行
  parser.onPartEnd = function() {
    // 同理，触发end事件
    part.emit('end');
  };

  //当解析完last_boundary时执行
  parser.onEnd = function() {
    self.ended = true;
    self._maybeEnd();
  };

  this._parser = parser;

}
```

到此，已经定义了所有需要的回调函数,可是上面代码中提到的data和end事件是在哪定义的呢？

让我们按照刚才需找"回调函数定义之处"的方式思维来思考,由于在`onPartData`和`onPartEnd`函数执行之前会先执行`onHeadersEnd`,

而`onHeadersEnd`中只有一条语句`self.onPart(part)`，不难看到，这两个事件应该是在`onPart`中定义的。

```
IncomingForm.prototype.onPart = function (part) { //真纠结啊
  this.handlePart(part);
}

// 这个函数很重要，目的是处理header_field和header_value,以及定义上文提到的事件
IncomingForm.prototype.handlePart = function (part) {
  var self=this;
  if (part.filename==undefined) { // 如果当前的数据部分不是文件
    //这里可以看源代码,不讲解了
    return;
  }

  // 这里的File是另一个模块，用于写文件，很简单，这里就不讲了，可以看一下源码，我想应该都能看懂
  var file = new File({
   path: 'tmp/'+path.filename,
   name: part.filename,
   type: part.mime,
  });

  file.open();
  // 这里就是刚才提到的data事件，当文件数据解析完毕后，会触发这个事件
  part.on('data', function (buffer) {
    // 文件写入过程中，暂定request请求，写入完毕后，回复请求，原因前面已经将结果
    self.pause();
    file.write(buffer, function () {
      self.resume();
    });

    //end事件

   part.on('end', function () {
     file.end(function () {
       self.emit('file', part.name, file);
       self._maybeEnd();
     });
   });

  });
}
```