nodejs-post文件上传原理详解

技术笔记 node formidable webuploader

基础知识

浅谈HTTP中Get与Post的区别
HTTP请求报文格式 简单介绍下，如下图：

其中请求报文中的请求行和请求头部包含了常见的各种信息，比如http协议版本，请求方法（GET/POST），accept-language，cookie等等。 而'请求数据'一般在post中使用，比如我们用表单上传文件，文件数据就是在这个'请求数据'当中。

引子

写这篇教程的起因是因为在学习nodejs的过程中，想要自己实现一些文件上传的功能，于是不得不去研究POST。如果你写过一点PHP，那么你肯定记得，在PHP里面，进行文件上传的时候，我们可以直接使用全局变量 $_FILE['name']来获取已经被临时存储的文件信息。但是实际上，POST数据实体，会根据数据量的大小进行分包传送，然后再从这些数据包里面分析出哪些是文件的元数据，那些是文件本身的数据。PHP是底层做了封装，但是在nodejs里面，这个看似常见的功能却是需要自己来实现的。这篇文章主要就是介绍如何使用nodejs来解析post数据。

正文

总体上来说，对于post文件上传这样的过程，主要有以下几个部分：

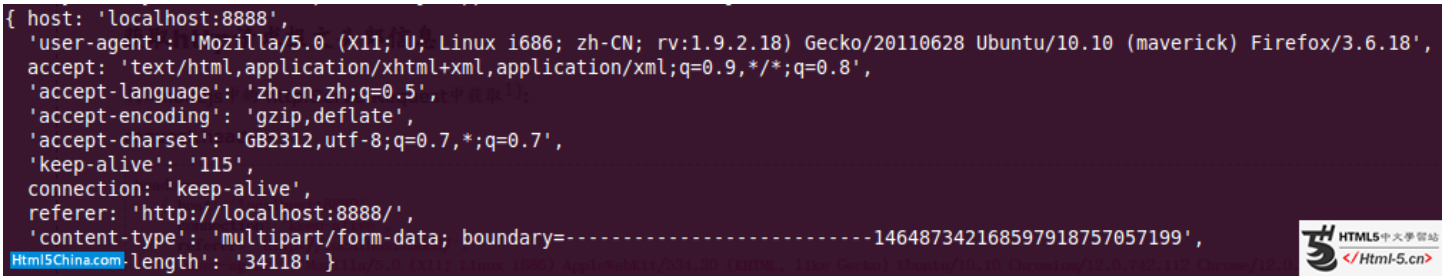
- 获取http请求报文中的头部信息，我们可以从中获得是否为POST方法，实体主体的总大小，边界字符串等，这些对于实体主体数据的解析都是非常重要的
- 获取POST数据（实体主体）
- 对POST数据进行解析
- 将数据写入文件

获取http请求报文头部信息

利用nodejs中的 http.ServerRequest 中获取

> request.method　用来标识请求类型（GET/POST）

request.headers



其中我们关心两个字段：

- content-type 包含了表单类型和边界字符串（下面会介绍）信息
- content-length post数据的长度

关于content-type

get请求的headers中没有content-type这个字段，post 的 content-type 有两种：

- application/x-www-Form-urlencoded 这种就是一般的文本表单用post传地数据，只要将得到的data用querystring解析下就可以了
- multipart/form-data 文件表单的传输，也是本文介绍的重点

获取POST数据

前面已经说过，post数据的传输是可能分包的，因此必然是异步的。post数据的接受过程如下：

```
var postData = '';
request.addListener("data", function(postDataChunk) { // 有新的数据包到达就执行
    postData += postDataChunk;
    console.log("Received POST data chunk '" +
    postDataChunk + "'.");
});
request.addListener("end", function() { // 数据传输完毕
  console.log('post data finish receiving: ' + postData);
});
```

注意

> 对于非文件post数据，上面以字符串接收是没问题的，但其实 postDataChunk 是一个 buffer 类型数据，在遇到二进制时，这样的接受方式存在问题。

POST数据的解析（multipart/form-data）

在解析POST数据之前，先介绍一下post数据的格式：

multipart/form-data类型的post数据

例如我们有表单如下

```
<form action="#" enctype="multipart/form-data" method="post">
```

```
  What is your name?
  <input type="text" name="submit-name">
  <BR>
  What files are you sending?
  <input type="file" name="files">
  <BR>
  <input type="submit" value="Send">
  <input type="reset">
</form>
```

若用户在text字段中输入‘Neekey’，并且在file字段中选择文件‘text.txt’,那么服务器端收到的post数据如下：

```
  --AaB03x
  Content-Disposition: form-data; name="submit-name"

  Neekey
  --AaB03x
  Content-Disposition: form-data; name="files"; filename="file1.txt"
  Content-Type: text/plain

  ... contents of file1.txt ...
  --AaB03x--
```

若file字段为空：

```
  --AaB03x
  Content-Disposition: form-data; name="submit-name"

  Neekey
  --AaB03x
  Content-Disposition: form-data; name="files"; filename=""
  Content-Type: text/plain

  --AaB03x--
```

若将file 的 input修改为可以多个文件一起上传：type=file 添加 multiple="multiple"

```
<form action="#" enctype="multipart/form-data" method="post">
  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">
  <input type="reset">
</form>
```

那么在text中输入‘Neekey’,并在file字段中选中两个文件’a.jpg’和’b.jpg’后：

```
  --AaB03x
  Content-Disposition: form-data; name="submit-name"

  Neekey
  --AaB03x
  Content-Disposition: form-data; name="files"; filename="a.jpg"
  Content-Type: image/jpeg

  /* data of a.jpg */
  --AaB03x
  Content-Disposition: form-data; name="files"; filename="b.jpg"
  Content-Type: image/jpeg

  /* data of b.jpg */
  --AaB03x--

   // 可以发现 两个文件数据部分，他们的name值是一样的
```

数据规则

简单总结下post数据的规则:

- 不同字段数据之间以边界字符串分隔：--boundary\r\n
- 每一行数据用”CR LF”（\r\n)分隔
- 数据以 边界分割符 后面加上 -- 结尾
- 每个字段数据的header信息（content-disposition/content-type）和字段数据以一个空行分隔 \r\n\r\n

更加详细的信息可以参考W3C的文档 Forms，不过文档中对于 multiple=“multiple” 的文件表单的post数据格式使用了二级边界字符串，但是在实际测试中，multiple类型的表单和多个单文件表单上传数据的格式一致，有更加清楚的可以交流下：

> If the user selected a second (image) file "file2.gif", the user agent might construct the parts as follows:

```
Content-Type: multipart/form-data; boundary=AaB03x

--AaB03x
```

```
Content-Disposition: form-data; name="submit-name"

Larry
--AaB03x
Content-Disposition: form-data; name="files"
Content-Type: multipart/mixed; boundary=BbC04y

--BbC04y
Content-Disposition: file; filename="file1.txt"
Content-Type: text/plain

... contents of file1.txt ...
--BbC04y
Content-Disposition: file; filename="file2.gif"
Content-Type: image/gif
Content-Transfer-Encoding: binary

...contents of file2.gif...
--BbC04y--
--AaB03x--
```

数据解析基本思路

必须使用buffer来进行post数据的解析

利用文章一开始的方法（data += chunk， data为字符串 ），可以利用字符串的操作，轻易地解析出各自端的信息，但是这样有两个问题：

- **文件的写入需要buffer类型的数据** 二进制buffer转化为string，并做字符串操作后，起索引和字符串是不一致的（若原始数据就是字符串，一致），因此是先将不总的buffer数据的toString()复制给一个字符串，再利用字符串解析出个数据的start，end位置这样的方案也是不可取的。
- **利用边界字符串来分割各字段数据** 每个字段数据中，使用空行（\r\n\r\n)来分割字段信息和字段数据，所有的数据都是以\r\n分割，利用上面的方法，我们以某种方式确定了数据在buffer中的start和end，利用buffer.splice(start, end) 便可以进行文件写入了

文件写入

比较简单，使用 File System 模块（nodejs的文件处理，我很弱很弱….)

```
var fs = new require( 'fs' ).writeStream,
file = new fs( filename );
fs.write( buffer, function(){
    fs.end();
});
```

node-formidable模块源码分析

> node-formidable是比较流行的处理表单的nodejs模块 github主页

项目中的lib目录

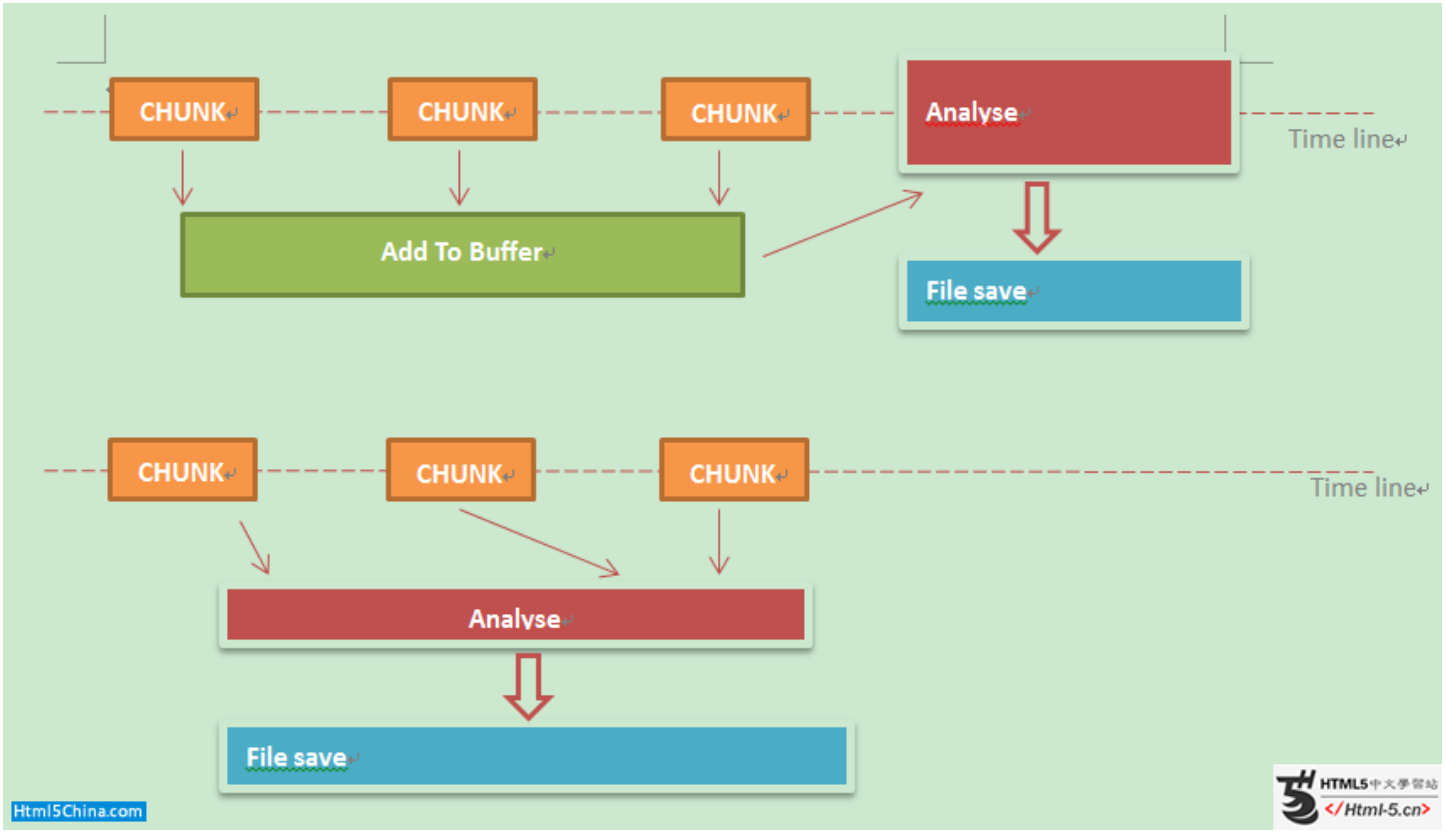
```
lib
  |-file.js
  |-incoming_form.js
  |-index.js
  |-multipart_parser.js
  |-querystring_parser.js
  |-util.js
```

各文件说明

- **file.js** 主要是封装了文件的写操作
- **incoming_from.js** 模块的主体部分
- **multipart_parser.js** 封装了对于POST数据的分段读取与解析的方法
- **querystring_parser.js** 封装了对于GET数据的解析

总体思路

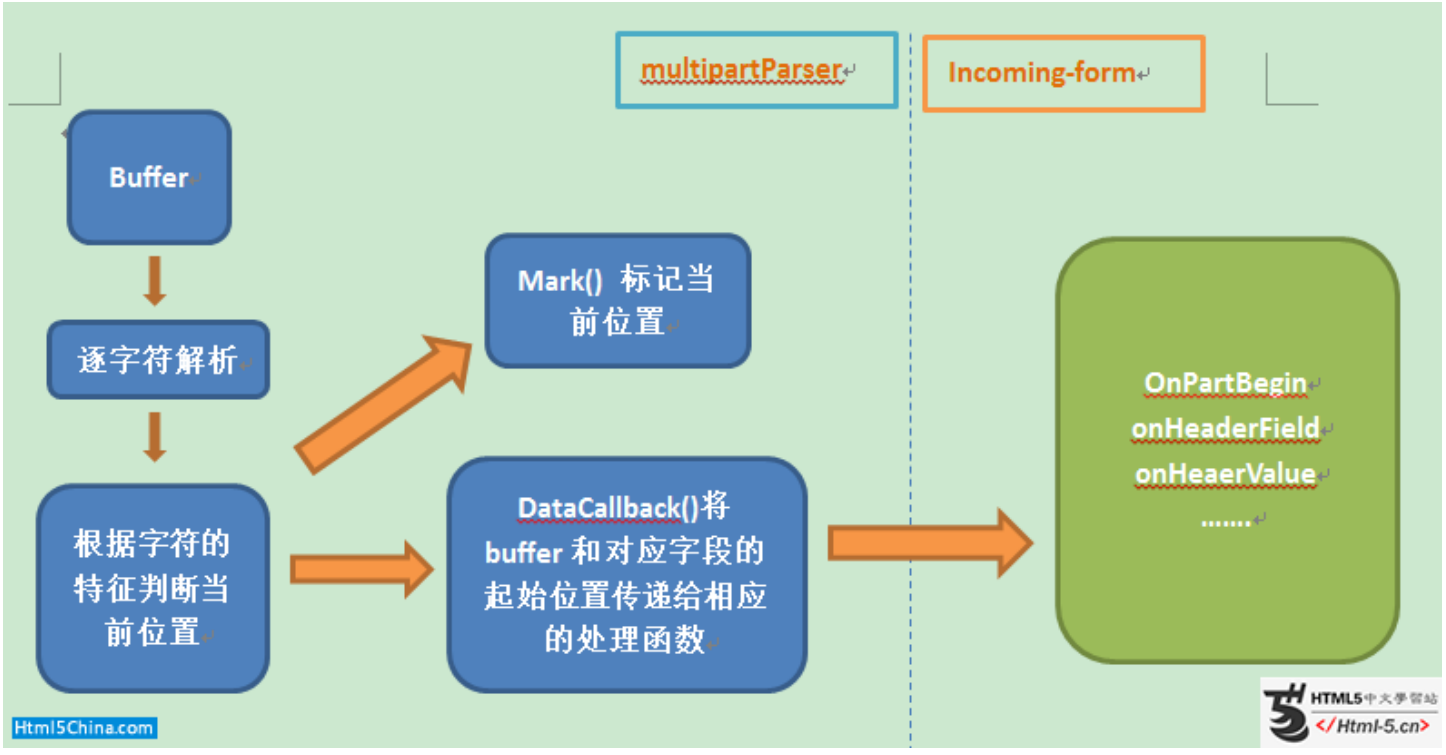
> 与我上面提到的思路不一样，node-formidable是边接受数据边进行解析。



上面那种方式是每一次有数据包到达后, 添加到buffer中,等所有数据都到齐后,再对数据进行解析.这种方式,在每次数据包到达的间隙是空闲的.

第二种方式使用边接收边解析的方式,对于大文件来说,能大大提升效率.

> 模块的核心文件主要是 multipart_parser.js 和 incoming_from.js 两个文件, 宏观上, multipartParser 用于解析数据, 比如给定一个buffer, 将在解析的过程中调用相应的回调函数.比如解析到字段数据的元信息(文件名,文件类型等), 将会使用 this.onHeaderField(buffer, start, end) 这样的形式来传输信息. 而这些方法的具体实现则是在 incoming_form.js 文件中实现的. 下面着重对这两个文件的源码进行分析



multipart_form.js
这个模块是POST数据接受的核心。起核心思想是对每个接受到的partData进行解析，并触发相应事件，由于每次write方法的调用都将产生内部私有方法，所以partData将会被传送到各个触发事件当中，而触发事件（即对于partData的具体处理）的具体实现则是在incoming_form中实现，从这一点来说，两个模块是高度耦合的。

multipart_form 的源码读起来会比较吃力。必须在对post数据结构比较清楚的情况下，在看源码。源码主要是四个部分：

- 全局变量（闭包内）
- 构造函数
- 初始化函数（initWithBoundary）
- 解析函数（write）

其中**全局变量**，**构造函数**比较简单。**初始化函数**用 用传进的 边界字符串 构造boundary的buffer，主要用于在**解析函数**中做比较。

下面主要介绍下解析函数，几个容易迷惑的私有方法

- **make(name)** 将当前索引（对buffer的遍历）复制给 this[name]. 这个方法就是做标记，用于记录一个数据段在buffer中的开始位置
- **callback(name , buffer, start, end)** 调用this的onName方法，并传入buffer和start以及end三个参数。 比如当文件post数据中的文件部分的数据解析完毕，则通过callback('partData', buffer, start, end) 将该数据段的首尾位置和buffer传递给 this.onPartData 方法，做进一步处理。
- **dataCallback(name, clear)** 前面的callback，如果不看后面的三个参数，其本质不过是一个调用某个方法的桥接函数。而dataCallback则是对callback的一个封装，他将start和end传递给callback。

从源码中可以看到，start是通过mark(name)的返回值获得，而end则可能是当前遍历到的索引或者是buffer的末尾。因此dataCallback被调用有二种情况：

- 在解析的数据部分的末尾在当前buffer的内部，这个时候mark记录的开始点和当前遍历到的i这个区段就是需要的数据，因此start = mark(name), end = i， 并且由于解析结束，需要将mark清除掉。
- 在当前buffer内，解析的数据部分尚未解析完毕（剩下的内容在下一个buffer里），因此start = mark(name), end = buffer.length

解析的主要部分

> 解析的主要部分是对buffer进行遍历，然后对于每一个字符，都根据当前的状态进行switch的各个case进行操作。switch的每一个case都是一个解析状态。具体看源码和注释，然后对照post的数据结构就会比较清楚。

其中 在状态：S.PART_DATA 这边，node-formidable做了一些处理，应该是针对 文章一开始介绍post数据格式中提到的 二级边界字符串 类型的数据处理。我没有深究，有兴趣的可以再研究下。

```
var Buffer = require('buffer').Buffer,
    s = 0,
    S = {
      PARSER_UNINITIALIZED: s++, // 解析尚未初始化
```

```
      START: s++, // 开始解析
      START_BOUNDARY: s++, // 开始找到边界字符串
      HEADER_FIELD_START: s++, // 开始解析到header field
      HEADER_FIELD: s++,
      HEADER_VALUE_START: s++, // 开始解析到header value
      HEADER_VALUE: s++,
      HEADER_VALUE_ALMOST_DONE: s++, // header value 解析完毕
      HEADERS_ALMOST_DONE: s++, // header 部分 解析完毕
      PART_DATA_START: s++, // 开始解析 数据段
      PART_DATA: s++,
      PART_END: s++,
      END: s++,
    },
    f = 1,
    F = {
      PART_BOUNDARY: f,
      LAST_BOUNDARY: f *= 2,
    },
    /* 一些字符的ASCII值 */
    LF = 10,
    CR = 13,
    SPACE = 32,
    HYPHEN = 45,
    COLON = 58,
    A = 97,
    Z = 122,

    /* 将所有大写小写字母的ascii一律转化为小写的ascii值 */
    lower = function(c) {
      return c | 0x20;
    };

    for (var s in S) {
      exports[s] = S[s];
    }

/* 构造函数 */
function MultipartParser() {
  this.boundary = null;
  this.boundaryChars = null;
  this.lookbehind = null;
  this.state = S.PARSER_UNINITIALIZED;

  this.index = null;
```

```
  this.flags = 0;
};
exports.MultipartParser = MultipartParser;
/* 给定边界字符串以初始化 */
MultipartParser.prototype.initWithBoundary = function(str) {
  this.boundary = new Buffer(str.length + 4);
  this.boundary.write('\r\n--', 'ascii', 0);
  this.boundary.write(str, 'ascii', 4);
  this.lookbehind = new Buffer(this.boundary.length + 8);
  this.state = S.START;
  this.boundaryChars = {};
  for (var i = 0; i < this.boundary.length; i++) {
    this.boundaryChars[this.boundary[i]] = true;
  }
};
/* 每个数据段到达时的处理函数 */
MultipartParser.prototype.write = function(buffer) {
  var self = this,
      i = 0,
      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.boundary.length,
      boundaryEnd = boundaryLength - 1,
      bufferLength = buffer.length,
      c,
      cl,
    /* 标记了name这个标记点的buffer偏移 */
    mark = function(name) {
      self[name + 'Mark'] = i;
    },
    /* 清除标记 */
    clear = function(name) {
      delete self[name + 'Mark'];
    },
    /* 回调函数，将调用onName,并传入对应的buffer和对应的offset区间，可知这些回调都在 incoming_form 模块中被具体实现 */
    callback = function(name, buffer, start, end) {
      if (start !== undefined && start === end) {
```

```
        return;
      }
      var callbackSymbol = 'on' + name.substr(0, 1).toUpperCase() + name.substr(1);
      if (callbackSymbol in self) {
        self[callbackSymbol](buffer, start, end);
      }
    },
    /* 数据回调 */
    dataCallback = function(name, clear) {
      var markSymbol = name + 'Mark';
      if (!(markSymbol in self)) {
        return;
      }
      if (!clear) {
        /* 传入回调的名称，buffer，buffer的开始位置（可见mark方法就是用来存储offset的）
，end为数据的重点 */
        callback(name, buffer, self[markSymbol], buffer.length);
        self[markSymbol] = 0;
      } else {
        /* 区别是 end 的值为i，在一个数据已经判断到达其结束位置时，就删除这个mark点，因为
这个时候已经知道了这个数据的起始位置 */
        callback(name, buffer, self[markSymbol], i);
        delete self[markSymbol];
      }
    };
  /* 对buffer逐个字节遍历，进行解析判断，并触发相应事件 */
  for (i = 0; i < len; i++) {
    c = buffer[i];
    switch (state) {
      case S.PARSER_UNINITIALIZED:
        return i;
      case S.START:
        index = 0;
        state = S.START_BOUNDARY;
      case S.START_BOUNDARY:
        /**
         * 对于边界的判断 ====
         * 这里看了很多次，一直很迷惑
         * 因为首先：除了最后一个边界字符串，其他（包括第一个边界字符串）都是这个样子：
         * --boundary\r\t
         * 但是在初始化函数中，对于this.boundary的赋值是这样的：
         * \r\n--boundary
         *
         * 但是仔细看下面部分的代码，作者只是为了初始化方便，或者出于其他的考虑
```

```
         */
        /**
         * 判断是否到了边界字符串的结尾\r处
         * 如果当前字符与 \r不匹配，则return ，算是解析出错了
         * 注意，虽然this.boundary != --boundary\r\n 但是长度是一致的，因此这里的判断是没有问题的
         */
        if (index == boundary.length - 2) {
          if (c != CR) {
            return i;
          }
          index++;
          break;
        }
        /**
         * 判断是否到了边界字符串的结尾\n处
         * 如果是，则置index = 0
         * 回调 partBegin，并将状态设置为还是header 的 field 信息的读取状态
         */
        else if (index - 1 == boundary.length - 2) {
          if (c != LF) {
            return i;
          }
          index = 0;
          callback('partBegin');
          state = S.HEADER_FIELD_START;
          break;
        }
        /**
         * 除了boundary的最后的\r\n外，其他字符都要进行检查
         * 注意这里用的是 index+2 进行匹配，证实了
         * 作者是因为某种意图将 boundary设置成\r\n--boundary的形式
         * （其实这种形式也没有错，对处第一个边界字符串外的其他边界字符串，这个形式都是适用的）
         */
        if (c != boundary[index + 2]) {
          return i;
        }
        index++;
        break;
        /* 对于header field的扫描开始，这里记录了标记了开始点在buffer中的位置 */
      case S.HEADER_FIELD_START:
        state = S.HEADER_FIELD;
        mark('headerField');
```

```
        index = 0;
        /* header field的扫描过程 */
      case S.HEADER_FIELD:
        /* 这里是header和data中间的那个空行，所以第一个字符就是\r */
        if (c == CR) {
          clear('headerField');
          state = S.HEADERS_ALMOST_DONE;
          break;
        }
        index++;
        /* 如果是小横线 '-' 比如在 Content-Disposition */
        if (c == HYPHEN) {
          break;
        }
        /**
         *  如果是冒号，那么说明 field结束了
         *  dataCallback，注意第二个参数为true，则它将调用this.onHeaderField,并且将buff
er，start(之前mark(headerField)记录的位置)，end(当前的i)传递过去，最后将这个mark清理掉
         *  之后进入 header value 的开始阶段
         */
        if (c == COLON) {
          if (index == 1) {
            // empty header field
            return i;
          }
          dataCallback('headerField', true);
          state = S.HEADER_VALUE_START;
          break;
        }
        /**
         * 对于所有其他不是冒号和小横线的字符，必须为字母，否则解析结束
         */
        cl = lower(c);
        if (cl < A || cl > Z) {
          return i;
        }
        break;
        /**
         * value 的读取开始
         * 做记号，设置state
         */
      case S.HEADER_VALUE_START:
        if (c == SPACE) {
          break;
```

```
        }
        mark('headerValue');
        state = S.HEADER_VALUE;
        /**
         * value 的分析阶段
         */
      case S.HEADER_VALUE:
        /**
         * 如果是 \r，则value结束
         * 同样是调用 dataCallback，参数为true
         * 注意这里还 callback了 headerEnd，没有给定任何参数，这里是作为一个trigger抛出一行header结束的事件
         */
        if (c == CR) {
          dataCallback('headerValue', true);
          callback('headerEnd');
          state = S.HEADER_VALUE_ALMOST_DONE;
        }
        break;
        /**
         * value 结束的检查，之前是检查到了 \r ，如果下一个字符不是 \n 肯定问题
         * 一个value结束，可能下面还是一行header，所以不会直接 header done，而是重新进入扫描 fields的阶段
         */
      case S.HEADER_VALUE_ALMOST_DONE:
        if (c != LF) {
          return i;
        }
        state = S.HEADER_FIELD_START;
        break;
        /**
         * 同样是先检查一下字符是否有误（看一下 case S.HEADER_FIELD 的第一个if ）
         * 抛出headers解析完毕的事件
         */
      case S.HEADERS_ALMOST_DONE:
        if (c != LF) {
          return i;
        }
        callback('headersEnd');
        state = S.PART_DATA_START;
        break;
        /**
         * 开始解析post数据
         * 设置状态，做记号
```

```
     */
    case S.PART_DATA_START:
      state = S.PART_DATA
      mark('partData');
      /**
       * 进入post数据的解析状态
       * 上一次设置index是在 HEADER_FIELD_START中设置为0
       */
    case S.PART_DATA:
      prevIndex = index;
      if (index == 0) {
        // boyer-moore derrived algorithm to safely skip non-boundary data
        i += boundaryEnd;
        while (i < bufferLength && !(buffer[i] in boundaryChars)) {
          i += boundaryLength;
        }
        i -= boundaryEnd;
        c = buffer[i];
      }
      if (index < boundary.length) {
        if (boundary[index] == c) {
          if (index == 0) {
            dataCallback('partData', true);
          }
          index++;
        } else {
          index = 0;
        }
      } else if (index == boundary.length) {
        index++;
        if (c == CR) {
          // CR = part boundary
          flags |= F.PART_BOUNDARY;
        } else if (c == HYPHEN) {
          // HYPHEN = end boundary
          flags |= F.LAST_BOUNDARY;
        } else {
          index = 0;
        }
      } else if (index - 1 == boundary.length) {
        if (flags & F.PART_BOUNDARY) {
          index = 0;
          if (c == LF) {
            // unset the PART_BOUNDARY flag
```

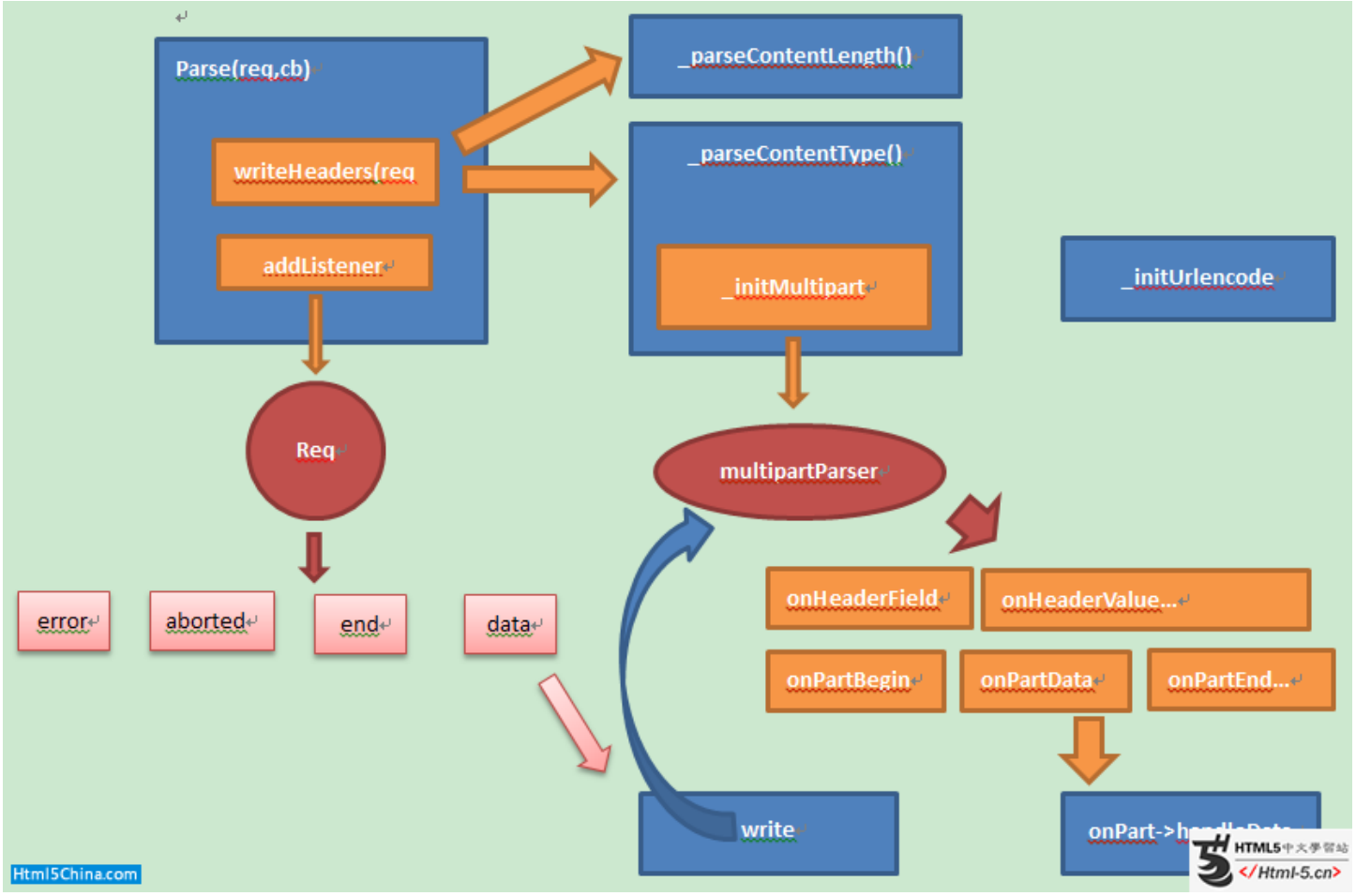
```
              flags &= ~F.PART_BOUNDARY;
              callback('partEnd');
              callback('partBegin');
              state = S.HEADER_FIELD_START;
              break;
            }
          } else if (flags & F.LAST_BOUNDARY) {
            if (c == HYPHEN) {
              callback('partEnd');
              callback('end');
              state = S.END;
            } else {
              index = 0;
            }
          } else {
            index = 0;
          }
        }
        if (index > 0) {
          // when matching a possible boundary, keep a lookbehind reference
          // in case it turns out to be a false lead
          lookbehind[index - 1] = c;
        } else if (prevIndex > 0) {
          // if our boundary turned out to be rubbish, the captured lookbehind
          // belongs to partData
          callback('partData', lookbehind, 0, prevIndex);
          prevIndex = 0;
          mark('partData');
          // reconsider the current character even so it interrupted the sequence
          // it could be the beginning of a new sequence
          i--;
        }
        break;
      case S.END:
        break;
      default:
        return i;
    }
  }
  /**
   * 下面这三个是在当前数据段解析完全后调用的。
   * 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
```

```
    */
    dataCallback('headerField');
    dataCallback('headerValue');
    dataCallback('partData');

    this.index = index;
    this.state = state;
    this.flags = flags;

    return len;
};
MultipartParser.prototype.end = function() {
    if (this.state != S.END) {
        return new Error('MultipartParser.end(): stream ended unexpectedly');
    }
};
```

incoming_form.js



上图是incoming_form解析的主要过程（文件类型），其中:

- **parse** 根据传入的requeset对象开始启动整个解析的过程
- **writeHeaders** 从request对象中获取post数据长度，解析出边界字符串，用来初始化multipartParser，为request对象添加监听事件

- **write** request对象的 ‘data’事件到达会调用该方法，而write方法实质上是调用multipartParser.write
- **_initMultipart** 利用边界字符串初始化multipartParser，并实现在multipart_form.js中write解析方法中会触发的事件回调函数

具体细节看源码会比较清楚。

```
if (global.GENTLY) require = GENTLY.hijack(require);
var util = require('./util'),
    path = require('path'),
    File = require('./file'),
    MultipartParser = require('./multipart_parser').MultipartParser,
    QuerystringParser = require('./querystring_parser').QuerystringParser,
    StringDecoder = require('string_decoder').StringDecoder,
    EventEmitter = require('events').EventEmitter;
function IncomingForm() {
  if (!(this instanceof IncomingForm)) return new IncomingForm;
  EventEmitter.call(this);
  this.error = null;
  this.ended = false;
  this.maxFieldsSize = 2 * 1024 * 1024; // 设置最大文件限制
  this.keepExtensions = false;
  this.uploadDir = '/tmp'; // 设置文件存放目录
  this.encoding = 'utf-8';
  this.headers = null; // post请求的headers信息
  // 收到的post数据类型（一般的字符串数据，还是文件）
  this.type = null;
  this.bytesReceived = null; // 已经接受的字节
  this.bytesExpected = null; // 预期接受的字节
  this._parser = null;
  this._flushing = 0;
  this._fieldsSize = 0;
};
util.inherits(IncomingForm, EventEmitter);
exports.IncomingForm = IncomingForm;
IncomingForm.prototype.parse = function(req, cb) {
  // 每次调用都重新建立方法，用于对req和cb的闭包使用
  this.pause = function() {
    try {
      req.pause();
    } catch (err) {
      // the stream was destroyed
      if (!this.ended) {
        // before it was completed, crash & burn
        this._error(err);
      }
```

```
      return false;
    }
    return true;
  };
  this.resume = function() {
    try {
      req.resume();
    } catch (err) {
      // the stream was destroyed
      if (!this.ended) {
        // before it was completed, crash & burn
        this._error(err);
      }
      return false;
    }
    return true;
  };
  // 记录下headers信息
  this.writeHeaders(req.headers);
  var self = this;
  req
    .on('error', function(err) {
      self._error(err);
    })
    .on('aborted', function() {
      self.emit('aborted');
    })
  // 接受数据
  .on('data', function(buffer) {
    self.write(buffer);
  })
  // 数据传送结束
  .on('end', function() {
    if (self.error) {
      return;
    }
    var err = self._parser.end();
    if (err) {
      self._error(err);
    }
  });
  // 若回调函数存在
  if (cb) {
    var fields = {},
```

```
      files = {};
    this
    // 一个字段解析完毕，触发事件
    .on('field', function(name, value) {
      fields[name] = value;
    })
    // 一个文件解析完毕，处罚事件
    .on('file', function(name, file) {
      files[name] = file;
    })
      .on('error', function(err) {
        cb(err, fields, files);
      })
    // 所有数据接收完毕，执行回调函数
    .on('end', function() {
      cb(null, fields, files);
    });
  }
  return this;
};
// 保存header信息
IncomingForm.prototype.writeHeaders = function(headers) {
  this.headers = headers;
  // 从头部中解析数据的长度和form类型
  this._parseContentLength();
  this._parseContentType();
};
IncomingForm.prototype.write = function(buffer) {
  if (!this._parser) {
    this._error(new Error('unintialized parser'));
    return;
  }
  /* 累加接收到的信息 */
  this.bytesReceived += buffer.length;
  this.emit('progress', this.bytesReceived, this.bytesExpected);
  // 解析数据
  var bytesParsed = this._parser.write(buffer);
  if (bytesParsed !== buffer.length) {
    this._error(new Error('parser error, ' + bytesParsed + ' of ' + buffer.length +
 ' bytes parsed'));
  }
  return bytesParsed;
};
IncomingForm.prototype.pause = function() {
```

```
  // this does nothing, unless overwritten in IncomingForm.parse
  return false;
};
IncomingForm.prototype.resume = function() {
  // this does nothing, unless overwritten in IncomingForm.parse
  return false;
};
/**
 * 开始接受数据（这个函数在headers被分析完成后调用，这个时候剩下的data还没有解析过来
 */
IncomingForm.prototype.onPart = function(part) {
  // this method can be overwritten by the user
  this.handlePart(part);
};
IncomingForm.prototype.handlePart = function(part) {
  var self = this;
  /* post数据不是文件的情况 */
  if (!part.filename) {
    var value = '',
      decoder = new StringDecoder(this.encoding);
    /* 有数据过来时 */
    part.on('data', function(buffer) {
      self._fieldsSize += buffer.length;
      if (self._fieldsSize > self.maxFieldsSize) {
        self._error(new Error('maxFieldsSize exceeded, received ' + self._fieldsSiz
e + ' bytes of field data'));
        return;
      }
      value += decoder.write(buffer);
    });
    part.on('end', function() {
      self.emit('field', part.name, value);
    });
    return;
  }
  this._flushing++;
  // 创建新的file实例
  var file = new File({
    path: this._uploadPath(part.filename),
    name: part.filename,
    type: part.mime,
  });
  this.emit('fileBegin', part.name, file);
  file.open();
```

```
  /* 当文件数据达到，一点一点写入文件 */
  part.on('data', function(buffer) {
    self.pause();
    file.write(buffer, function() {
      self.resume();
    });
  });
  // 一个文件的数据解析完毕，出发事件
  part.on('end', function() {
    file.end(function() {
      self._flushing--;
      self.emit('file', part.name, file);
      self._maybeEnd();
    });
  });
};
/**
 * 解析表单类型
 * 如果为文件表单，则解析出边界字串，初始化multipartParser
 */
IncomingForm.prototype._parseContentType = function() {
  if (!this.headers['content-type']) {
    this._error(new Error('bad content-type header, no content-type'));
    return;
  }
  // 如果是一般的post数据
  if (this.headers['content-type'].match(/urlencoded/i)) {
    this._initUrlencoded();
    return;
  }
  // 如果为文件类型
  if (this.headers['content-type'].match(/multipart/i)) {
    var m;
    if (m = this.headers['content-type'].match(/boundary=(?:"([^"]+)"|([^;]+))/i))
{
      // 解析出边界字符串，并利用边界字符串初始化multipart组件
      this._initMultipart(m[1] || m[2]);
    } else {
      this._error(new Error('bad content-type header, no multipart boundary'));
    }
    return;
  }
  this._error(new Error('bad content-type header, unknown content-type: ' + this.he
aders['content-type']));
```

```
};
IncomingForm.prototype._error = function(err) {
  if (this.error) {
    return;
  }
  this.error = err;
  this.pause();
  this.emit('error', err);
};
// 从 this.headers 中获取数据总长度
IncomingForm.prototype._parseContentLength = function() {
  if (this.headers['content-length']) {
    this.bytesReceived = 0;
    this.bytesExpected = parseInt(this.headers['content-length'], 10);
  }
};
IncomingForm.prototype._newParser = function() {
  return new MultipartParser();
};
// 初始化multipartParset 组件
IncomingForm.prototype._initMultipart = function(boundary) {
  this.type = 'multipart';
  // 实例化组件
  var parser = new MultipartParser(),
      self = this,
      headerField,
      headerValue,
      part;
  parser.initWithBoundary(boundary);
  /**
   * 下面这些方法便是multipartParser中的callback以及dataCallback调用的函书
   * 当开始解析一个数据段（比如一个文件..)
   * 并重置相关信息
   */
  parser.onPartBegin = function() {
    part = new EventEmitter();
    part.headers = {};
    part.name = null;
    part.filename = null;
    part.mime = null;
    headerField = '';
    headerValue = '';
  };
  /**
```

```
   * 数据段的头部信息解析完毕（或者数据段的头部信息在当前接受到的数据段的尾部，并且尚未结束
)
   * 下面的onHeaderValue和onPartData也是一样的道理
   */
  parser.onHeaderField = function(b, start, end) {
    headerField += b.toString(self.encoding, start, end);
  };
  /* 数据段的头部信息value的解析过程 */
  parser.onHeaderValue = function(b, start, end) {
    headerValue += b.toString(self.encoding, start, end);
  };
  /* header信息（一行）解析完毕，并储存起来 */
  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];
      }
      if (m = headerValue.match(/filename="([^;]+)"/i)) {
        part.filename = m[1].substr(m[1].lastIndexOf('\\') + 1);
      }
    } else if (headerField == 'content-type') {
      part.mime = headerValue;
    }
    /* 重置，准备解析下一个header信息 */
    headerField = '';
    headerValue = '';
  };
  /* 整个headers信息解析完毕 */
  parser.onHeadersEnd = function() {
    self.onPart(part);
  };
  /* 数据部分的解析 */
  parser.onPartData = function(b, start, end) {
    part.emit('data', b.slice(start, end));
  };
  /* 数据段解析完毕 */
  parser.onPartEnd = function() {
    part.emit('end');
  };
  parser.onEnd = function() {
    self.ended = true;
```

```
    self._maybeEnd();
  };
  this._parser = parser;
};
/* 初始化，处理application/x-www-form-urlencoded类型的表单 */
IncomingForm.prototype._initUrlencoded = function() {
  this.type = 'urlencoded';
  var parser = new QuerystringParser(),
    self = this;
  parser.onField = function(key, val) {
    self.emit('field', key, val);
  };
  parser.onEnd = function() {
    self.ended = true;
    self._maybeEnd();
  };
  this._parser = parser;
};
/**
 * 根据给定的文件名，构造出path
 */
IncomingForm.prototype._uploadPath = function(filename) {
  var name = '';
  for (var i = 0; i < 32; i++) {
    name += Math.floor(Math.random() * 16).toString(16);
  }
  if (this.keepExtensions) {
    name += path.extname(filename);
  }
  return path.join(this.uploadDir, name);
};
IncomingForm.prototype._maybeEnd = function() {
  if (!this.ended || this._flushing) {
    return;
  }
  this.emit('end');
};
```