

master ▾

...

[mailparser](#) / [lib](#) / [mail-parser.js](#) / <> Jump to ▾

andris9 v3.4.0

[History](#)

12 contributors



1151 lines (1021 sloc) | 39.3 KB

...

```
1  'use strict';
2
3  const mailsplit = require('mailsplit');
4  const libmime = require('libmime');
5  const addressparser = require('nodemailer/lib/addressparser');
6  const Transform = require('stream').Transform;
7  const Splitter = mailsplit.Splitter;
8  const punycode = require('punycode');
9  const FlowedDecoder = require('mailsplit/lib/flowed-decoder');
10 const StreamHash = require('./stream-hash');
11 const iconv = require('iconv-lite');
12 const { htmlToText } = require('html-to-text');
13 const he = require('he');
14 const linkify = require('linkify-it')();
15 const tlds = require('tlds');
16 const encodingJapanese = require('encoding-japanese');
17
18 linkify
19   .tlds(tlds) // Reload with full tlds list
20   .tlds('onion', true) // Add unofficial `.onion` domain
21   .add('git:', 'http:') // Add `git:` protocol as "alias"
22   .add('ftp:', null) // Disable `ftp:` protocol
23   .set({ fuzzyIP: true, fuzzyLink: true, fuzzyEmail: true });
24
25 // twitter linkifier from
26 // https://github.com/markdown-it/linkify-it#example-2-add-twitter-mentions-handler
27 linkify.add('@', {
28   validate(text, pos, self) {
29     let tail = text.slice(pos);
```

```

30
31     if (!self.re.twitter) {
32         self.re.twitter = new RegExp('^([a-zA-Z0-9_]){1,15}(?!_)(?=$|' + self.re.src_Z
33     }
34     if (self.re.twitter.test(tail)) {
35         // Linkifier allows punctuation chars before prefix,
36         // but we additionally disable `@` ("@" is invalid)
37         if (pos >= 2 && tail[pos - 2] === '@') {
38             return false;
39         }
40         return tail.match(self.re.twitter)[0].length;
41     }
42     return 0;
43 },
44 normalize(match) {
45     match.url = 'https://twitter.com/' + match.url.replace(/^@/, '');
46 }
47 });
48
49 class IconvDecoder extends Transform {
50     constructor(Iconv, charset) {
51         super();
52
53         // Iconv throws error on ks_c_5601-1987 when it is mapped to EUC-KR
54         // https://github.com/bnoordhuis/node-iconv/issues/169
55         if (charset.toLowerCase() === 'ks_c_5601-1987') {
56             charset = 'CP949';
57         }
58         this.stream = new Iconv(charset, 'UTF-8//TRANSLIT//IGNORE');
59
60         this.inputEnded = false;
61         this.endCb = false;
62
63         this.stream.on('error', err => this.emit('error', err));
64         this.stream.on('data', chunk => this.push(chunk));
65         this.stream.on('end', () => {
66             this.inputEnded = true;
67             if (typeof this.endCb === 'function') {
68                 this.endCb();
69             }
70         });
71     }
72
73     _transform(chunk, encoding, done) {
74         this.stream.write(chunk);
75         done();
76     }
77
78     _flush(done) {

```

```

79     this.endCb = done;
80     this.stream.end();
81   }
82 }
83
84 class JPDecoder extends Transform {
85   constructor(charset) {
86     super();
87
88     this.charset = charset;
89     this.chunks = [];
90     this.chunklen = 0;
91   }
92
93   _transform(chunk, encoding, done) {
94     if (typeof chunk === 'string') {
95       chunk = Buffer.from(chunk, encoding);
96     }
97
98     this.chunks.push(chunk);
99     this.chunklen += chunk.length;
100    done();
101  }
102
103  _flush(done) {
104    let input = Buffer.concat(this.chunks, this.chunklen);
105    try {
106      let output = encodingJapanese.convert(input, {
107        to: 'UNICODE', // to_encoding
108        from: this.charset, // from_encoding
109        type: 'string'
110      });
111      if (typeof output === 'string') {
112        output = Buffer.from(output);
113      }
114      this.push(output);
115    } catch (err) {
116      // keep as is on errors
117      this.push(input);
118    }
119
120    done();
121  }
122 }
123
124 class MailParser extends Transform {
125   constructor(config) {
126     super({
127       readableObjectMode: true,

```

```
128         writableObjectMode: false
129     });
130
131     this.options = config || {};
132     this.splitter = new Splitter(config);
133     this.finished = false;
134     this.waitingEnd = false;
135
136     this.headers = false;
137     this.headerLines = false;
138
139     this.endReceived = false;
140     this.reading = false;
141     this.errorred = false;
142
143     this.tree = false;
144     this.curnode = false;
145     this.waitUntilAttachmentEnd = false;
146     this.attachmentCallback = false;
147
148     this.hasHtml = false;
149     this.hasText = false;
150
151     this.text = false;
152     this.html = false;
153     this.textAsHtml = false;
154
155     this.attachmentList = [];
156
157     this.boundaries = [];
158
159     this.textTypes = ['text/plain', 'text/html'].concat(!this.options.keepDeliveryStat
160
161     this.decoder = this.getDecoder();
162
163     this.splitter.on('readable', () => {
164         if (this.reading) {
165             return false;
166         }
167         this.readData();
168     });
169
170     this.splitter.on('end', () => {
171         this.endReceived = true;
172         if (!this.reading) {
173             this.endStream();
174         }
175     });
176
```

```

177     this.splitter.on('error', err => {
178         this.errorred = true;
179         if (typeof this.waitingEnd === 'function') {
180             return this.waitingEnd(err);
181         }
182         this.emit('error', err);
183     });
184
185     this.libmime = new libmime.Libmime({ Iconv: this.options.Iconv });
186 }
187
188 getDecoder() {
189     if (this.options.Iconv) {
190         const Iconv = this.options.Iconv;
191         // create wrapper
192         return {
193             decodeStream(charset) {
194                 return new IconvDecoder(Iconv, charset);
195             }
196         };
197     } else {
198         return {
199             decodeStream(charset) {
200                 charset = (charset || 'ascii').toString().trim().toLowerCase();
201                 if (/^jis|^iso-?2022-?jp|^EUCJP/i.test(charset)) {
202                     // special case not supported by iconv-lite
203                     return new JPDecoder(charset);
204                 }
205
206                 return iconv.decodeStream(charset);
207             }
208         };
209     }
210 }
211
212 readData() {
213     if (this.errorred) {
214         return false;
215     }
216     this.reading = true;
217     let data = this.splitter.read();
218     if (data === null) {
219         this.reading = false;
220         if (this.endReceived) {
221             this.endStream();
222         }
223         return;
224     }
225

```

```

226     this.processChunk(data, err => {
227         if (err) {
228             if (typeof this.waitingEnd === 'function') {
229                 return this.waitingEnd(err);
230             }
231             return this.emit('error', err);
232         }
233         setImmediate(() => this.readData());
234     });
235 }
236
237 endStream() {
238     this.finished = true;
239
240     if (this.curnode && this.curnode.decoder) {
241         this.curnode.decoder.end();
242     }
243     if (typeof this.waitingEnd === 'function') {
244         this.waitingEnd();
245     }
246 }
247
248 _transform(chunk, encoding, done) {
249     if (!chunk || !chunk.length) {
250         return done();
251     }
252
253     if (this.splitter.write(chunk) === false) {
254         return this.splitter.once('drain', () => {
255             done();
256         });
257     } else {
258         return done();
259     }
260 }
261
262 _flush(done) {
263     setImmediate(() => this.splitter.end());
264     if (this.finished) {
265         return this.cleanup(done);
266     }
267     this.waitingEnd = () => {
268         this.cleanup(() => {
269             done();
270         });
271     };
272 }
273
274 cleanup(done) {

```

```

275     let finish = () => {
276         let t = this.getTextContent();
277         this.push(t);
278         done();
279     };
280
281     if (this.curnode && this.curnode.decoder && this.curnode.decoder.readable && !this
282         (this.curnode.contentStream || this.curnode.decoder).once('end', () => {
283             finish();
284         }));
285     this.curnode.decoder.end();
286 } else {
287     setImmediate(() => {
288         finish();
289     });
290 }
291 }
292
293 processHeaders(lines) {
294     let headers = new Map();
295     (lines || []).forEach(line => {
296         let key = line.key;
297         let value = ((this.libmime.decodeHeader(line.line) || {}).value || '').toString
298         value = Buffer.from(value, 'binary').toString();
299         switch (key) {
300             case 'content-type':
301             case 'content-disposition':
302             case 'dkim-signature':
303                 value = this.libmime.parseHeaderValue(value);
304                 if (value.value) {
305                     value.value = this.libmime.decodeWords(value.value);
306                 }
307                 Object.keys((value && value.params) || {}).forEach(key => {
308                     try {
309                         value.params[key] = this.libmime.decodeWords(value.params[key]
310                     } catch (E) {
311                         // ignore, keep as is
312                     }
313                 });
314                 break;
315             case 'date': {
316                 let dateValue = new Date(value);
317                 if (isNaN(dateValue)) {
318                     // date parsing failed :S
319                     dateValue = new Date();
320                 }
321                 value = dateValue;
322                 break;
323             }

```

```
324     case 'subject':
325         try {
326             value = this.libmime.decodeWords(value);
327         } catch (E) {
328             // ignore, keep as is
329         }
330         break;
331     case 'references':
332         try {
333             value = this.libmime.decodeWords(value);
334         } catch (E) {
335             // ignore
336         }
337         value = value.split(/\s+/).map(this.ensureMessageIDFormat);
338         break;
339     case 'message-id':
340     case 'in-reply-to':
341         try {
342             value = this.libmime.decodeWords(value);
343         } catch (E) {
344             // ignore
345         }
346         value = this.ensureMessageIDFormat(value);
347         break;
348     case 'priority':
349     case 'x-priority':
350     case 'x-msmail-priority':
351     case 'importance':
352         key = 'priority';
353         value = this.parsePriority(value);
354         break;
355     case 'from':
356     case 'to':
357     case 'cc':
358     case 'bcc':
359     case 'sender':
360     case 'reply-to':
361     case 'delivered-to':
362     case 'return-path':
363         value = addressparser(value);
364         this.decodeAddresses(value);
365         value = {
366             value,
367             html: this.getAddressesHTML(value),
368             text: this.getAddressesText(value)
369         };
370         break;
371 }
372
```



```
373 // handle list-* keys
374 if (key.substr(0, 5) === 'list-') {
375     value = this.parseListHeader(key.substr(5), value);
376     key = 'list';
377 }
378
379 if (value) {
380     if (!headers.has(key)) {
381         headers.set(key, [].concat(value || []));
382     } else if (Array.isArray(value)) {
383         headers.set(key, headers.get(key).concat(value));
384     } else {
385         headers.get(key).push(value);
386     }
387 }
388 });
389
390 // keep only the first value
391 let singleKeys = [
392     'message-id',
393     'content-id',
394     'from',
395     'sender',
396     'in-reply-to',
397     'reply-to',
398     'subject',
399     'date',
400     'content-disposition',
401     'content-type',
402     'content-transfer-encoding',
403     'priority',
404     'mime-version',
405     'content-description',
406     'precedence',
407     'errors-to'
408 ];
409
410 headers.forEach((value, key) => {
411     if (Array.isArray(value)) {
412         if (singleKeys.includes(key) && value.length) {
413             headers.set(key, value[value.length - 1]);
414         } else if (value.length === 1) {
415             headers.set(key, value[0]);
416         }
417     }
418
419     if (key === 'list') {
420         // normalize List-* headers
421         let listValue = {};
```

```

422         [].concat(value || []).forEach(val => {
423             Object.keys(val || {}).forEach(listKey => {
424                 listValue[listKey] = val[listKey];
425             });
426         });
427         headers.set(key, listValue);
428     }
429 });
430
431     return headers;
432 }
433
434 parseListHeader(key, value) {
435     let addresses = addressparser(value);
436     let response = {};
437     let data = addresses
438         .map(address => {
439             if (/^https?:/i.test(address.name)) {
440                 response.url = address.name;
441             } else if (address.name) {
442                 response.name = address.name;
443             }
444             if (/^mailto:/.test(address.address)) {
445                 response.mail = address.address.substr(7);
446             } else if (address.address && address.address.indexOf('@') < 0) {
447                 response.id = address.address;
448             } else if (address.address) {
449                 response.mail = address.address;
450             }
451             if (Object.keys(response).length) {
452                 return response;
453             }
454             return false;
455         })
456         .filter(address => address);
457     if (data.length) {
458         return {
459             [key]: response
460         };
461     }
462     return false;
463 }
464
465 parsePriority(value) {
466     value = value.toLowerCase().trim();
467     if (!isNaN(parseInt(value, 10))) {
468         // support "X-Priority: 1 (Highest)"
469         value = parseInt(value, 10) || 0;
470         if (value === 3) {

```

```

471         return 'normal';
472     } else if (value > 3) {
473         return 'low';
474     } else {
475         return 'high';
476     }
477 } else {
478     switch (value) {
479         case 'non-urgent':
480             case 'low':
481                 return 'low';
482             case 'urgent':
483             case 'high':
484                 return 'high';
485         }
486     }
487     return 'normal';
488 }
489
490 ensureMessageIDFormat(value) {
491     if (!value.length) {
492         return false;
493     }
494
495     if (value.charAt(0) !== '<') {
496         value = '<' + value;
497     }
498
499     if (value.charAt(value.length - 1) !== '>') {
500         value += '>';
501     }
502
503     return value;
504 }
505
506 decodeAddresses(addresses) {
507     for (let i = 0; i < addresses.length; i++) {
508         let address = addresses[i];
509         address.name = (address.name || '').toString().trim();
510
511         if (!address.address && /^(=\\?([^?]+)\\?[Bb]\\?[^?]*\\?)(\\s*=\\?([^?]+)\\?[Bb]\\?[^
512             let parsed = addressparser(this.libmime.decodeWords(address.name));
513             if (parsed.length) {
514                 parsed.forEach(entry => addresses.push(entry));
515             }
516
517             // remove current element
518             addresses.splice(i, 1);
519             i--;

```

```

520         continue;
521     }
522
523     if (address.name) {
524         try {
525             address.name = this.libmime.decodeWords(address.name);
526         } catch (E) {
527             //ignore, keep as is
528         }
529     }
530     if (/@xn--/.test(address.address)) {
531         try {
532             address.address =
533                 address.address.substr(0, address.address.lastIndexOf('@') + 1) +
534                 punycode.toUnicode(address.address.substr(address.address.lastInde
535         } catch (E) {
536             // Not a valid punycode string; keep as is
537         }
538     }
539     if (address.group) {
540         this.decodeAddresses(address.group);
541     }
542 }
543 }
544
545 createNode(node) {
546     let contentType = node.contentType;
547     let disposition = node.disposition;
548     let encoding = node.encoding;
549     let charset = node.charset;
550
551     if (!contentType && node.root) {
552         contentType = 'text/plain';
553     }
554
555     let newNode = {
556         node,
557         headerLines: node.headers.lines,
558         headers: this.processHeaders(node.headers.getList()),
559         contentType,
560         children: []
561     };
562
563     if (/^multipart\/i.test(contentType)) {
564         if (disposition && !['attachment', 'inline'].includes(disposition)) {
565             disposition = 'attachment';
566         }
567
568         if (!disposition && !this.textTypes.includes(contentType)) {

```

```
569         newNode.disposition = 'attachment';
570     } else {
571         newNode.disposition = disposition || 'inline';
572     }
573
574     newNode.isAttachment = !this.textTypes.includes(contentType) || newNode.dispos
575
576     newNode.encoding = ['quoted-printable', 'base64'].includes(encoding) ? encodin
577
578     if (charset) {
579         newNode.charset = charset;
580     }
581
582     let decoder = node.getDecoder();
583     decoder.on('end', () => {
584         this.decoderEnded = true;
585     });
586     newNode.decoder = decoder;
587 }
588
589 if (node.root) {
590     this.headers = newNode.headers;
591     this.headerLines = newNode.headerLines;
592 }
593
594 // find location in tree
595
596 if (!this.tree) {
597     newNode.root = true;
598     this.curnode = this.tree = newNode;
599     return newNode;
600 }
601
602 // immediate child of root node
603 if (!this.curnode.parent) {
604     newNode.parent = this.curnode;
605     this.curnode.children.push(newNode);
606     this.curnode = newNode;
607     return newNode;
608 }
609
610 // siblings
611 if (this.curnode.parent.node === node.parentNode) {
612     newNode.parent = this.curnode.parent;
613     this.curnode.parent.children.push(newNode);
614     this.curnode = newNode;
615     return newNode;
616 }
617
```

```

618 // first child
619 if (this.curnode.node === node.parentNode) {
620     newNode.parent = this.curnode;
621     this.curnode.children.push(newNode);
622     this.curnode = newNode;
623     return newNode;
624 }
625
626 // move up
627 let parentNode = this.curnode;
628 while ((parentNode = parentNode.parent)) {
629     if (parentNode.node === node.parentNode) {
630         newNode.parent = parentNode;
631         parentNode.children.push(newNode);
632         this.curnode = newNode;
633         return newNode;
634     }
635 }
636
637 // should never happen, can't detect parent
638 this.curnode = newNode;
639 return newNode;
640 }
641
642 getTextContent() {
643     let text = [];
644     let html = [];
645     let processNode = (alternative, level, node) => {
646         if (node.showMeta) {
647             let meta = ['From', 'Subject', 'Date', 'To', 'Cc', 'Bcc']
648                 .map(fkey => {
649                     let key = fkey.toLowerCase();
650                     if (!node.headers.has(key)) {
651                         return false;
652                     }
653                     let value = node.headers.get(key);
654                     if (!value) {
655                         return false;
656                     }
657                     return {
658                         key: fkey,
659                         value: Array.isArray(value) ? value[value.length - 1] : value
660                     };
661                 })
662                 .filter(entry => entry);
663             if (this.hasHtml) {
664                 html.push(
665                     '<table class="mp_head">' +
666                     meta

```

```

667         .map(entry => {
668             let value = entry.value;
669             switch (entry.key) {
670                 case 'From':
671                 case 'To':
672                 case 'Cc':
673                 case 'Bcc':
674                     value = value.html;
675                     break;
676                 case 'Date':
677                     value = this.options.formatDateString ? this.o
678                     break;
679                 case 'Subject':
680                     value = '<strong>' + he.encode(value) + '</str
681                     break;
682                 default:
683                     value = he.encode(value);
684             }
685
686             return '<tr><td class="mp_head_key">' + he.encode(entr
687         })
688         .join('\n') +
689         '<table>'
690     );
691 }
692 if (this.hasText) {
693     text.push(
694         '\n' +
695         meta
696         .map(entry => {
697             let value = entry.value;
698             switch (entry.key) {
699                 case 'From':
700                 case 'To':
701                 case 'Cc':
702                 case 'Bcc':
703                     value = value.text;
704                     break;
705                 case 'Date':
706                     value = this.options.formatDateString ? this.o
707                     break;
708             }
709             return entry.key + ': ' + value;
710         })
711         .join('\n') +
712         '\n'
713     );
714 }
715 }

```

```

716     if (node.textContent) {
717         if (node.contentType === 'text/plain') {
718             text.push(node.textContent);
719             if (!alternative && this.hasHtml) {
720                 html.push(this.textToHtml(node.textContent));
721             }
722         } else if (node.contentType === 'message/delivery-status' && !this.options
723             text.push(node.textContent);
724             if (!alternative && this.hasHtml) {
725                 html.push(this.textToHtml(node.textContent));
726             }
727         } else if (node.contentType === 'text/html') {
728             let failedToParseHtml = false;
729             if ((!alternative && this.hasText) || (node.root && !this.hasText)) {
730                 if (this.options.skipHtmlToText) {
731                     text.push('');
732                 } else if (node.textContent.length > this.options.maxHtmlLengthToP
733                     this.emit('error', new Error(`HTML too long for parsing ${node
734                     text.push('Invalid HTML content (too long)');
735                     failedToParseHtml = true;
736                 } else {
737                     try {
738                         text.push(htmlToText(node.textContent));
739                     } catch (err) {
740                         this.emit('error', new Error('Failed to parse HTML'));
741                         text.push('Invalid HTML content');
742                         failedToParseHtml = true;
743                     }
744                 }
745             }
746             if (!failedToParseHtml) {
747                 html.push(node.textContent);
748             }
749         }
750     }
751     alternative = alternative || node.contentType === 'multipart/alternative';
752     node.children.forEach(subNode => {
753         processNode(alternative, level + 1, subNode);
754     });
755 };
756
757 processNode(false, 0, this.tree);
758
759 let response = {
760     type: 'text'
761 };
762 if (html.length) {
763     this.html = response.html = html.join('<br/>\n');
764 }

```



```

765     if (text.length) {
766         this.text = response.text = text.join('\n');
767         this.textAsHtml = response.textAsHtml = text.map(part => this.textToHtml(part)
768     }
769     return response;
770 }
771
772 processChunk(data, done) {
773     let partId = null;
774     if (data._parentBoundary) {
775         partId = this._getPartId(data._parentBoundary);
776     }
777     switch (data.type) {
778         case 'node': {
779             let node = this.createNode(data);
780             if (node === this.tree) {
781                 ['subject', 'references', 'date', 'to', 'from', 'to', 'cc', 'bcc', 'me
782                 if (node.headers.has(key)) {
783                     this[key.replace(/-([a-z])/g, (m, c) => c.toUpperCase())] = no
784                 }
785             });
786             this.emit('headers', node.headers);
787         }
788
789         if (data.contentType === 'message/rfc822' && data.messageNode) {
790             break;
791         }
792
793         if (data.parentNode && data.parentNode.contentType === 'message/rfc822') {
794             node.showMeta = true;
795         }
796
797         if (node.isAttachment) {
798             let contentType = node.contentType;
799             if (node.contentType === 'application/octet-stream' && data.filename)
800                 contentType = this.libmime.detectMimeType(data.filename) || 'appli
801         }
802
803         let attachment = {
804             type: 'attachment',
805             content: null,
806             contentType,
807             partId,
808             release: () => {
809                 attachment.release = null;
810                 if (this.waitUntilAttachmentEnd && typeof this.attachmentCallb
811                     setImmediate(this.attachmentCallback);
812             }
813             this.attachmentCallback = false;

```

```

814         this.waitUntilAttachmentEnd = false;
815     }
816 };
817
818 let algo = this.options.checksumAlgo || 'md5';
819 let hasher = new StreamHash(attachment, algo);
820 node.decoder.on('error', err => {
821     hasher.emit('error', err);
822 });
823
824 node.decoder.on('readable', () => {
825     let chunk;
826
827     while ((chunk = node.decoder.read()) !== null) {
828         hasher.write(chunk);
829     }
830 });
831
832 node.decoder.once('end', () => {
833     hasher.end();
834 });
835
836 //node.decoder.pipe(hasher);
837 attachment.content = hasher;
838
839 this.waitUntilAttachmentEnd = true;
840 if (data.disposition) {
841     attachment.contentDisposition = data.disposition;
842 }
843
844 if (data.filename) {
845     attachment.filename = data.filename;
846 }
847
848 if (node.headers.has('content-id')) {
849     attachment.contentId = [].concat(node.headers.get('content-id') ||
850 attachment.cid = attachment.contentId.trim().replace(/^<|>$/, '')
851 // check if the attachment is "related" to text content like an em
852 let parentNode = node;
853 while ((parentNode = parentNode.parent)) {
854     if (parentNode.contentType === 'multipart/related') {
855         attachment.related = true;
856     }
857 }
858 }
859
860 attachment.headers = node.headers;
861 this.push(attachment);
862 this.attachmentList.push(attachment);

```

```
863     } else if (node.disposition === 'inline') {
864         let chunks = [];
865         let chunklen = 0;
866         node.contentStream = node.decoder;
867
868         if (node.contentType === 'text/plain') {
869             this.hasText = true;
870         } else if (node.contentType === 'text/html') {
871             this.hasHtml = true;
872         } else if (node.contentType === 'message/delivery-status' && !this.opt
873             this.hasText = true;
874         }
875
876         if (node.node.flowed) {
877             let contentStream = node.contentStream;
878             let flowDecoder = new FlowedDecoder({
879                 delSp: node.node.delSp
880             });
881             contentStream.on('error', err => {
882                 flowDecoder.emit('error', err);
883             });
884             contentStream.pipe(flowDecoder);
885             node.contentStream = flowDecoder;
886         }
887
888         let charset = node.charset || 'utf-8';
889         //charset = charset || 'windows-1257';
890
891         if (!['ascii', 'usascii', 'utf8'].includes(charset.toLowerCase()).repla
892             try {
893                 let contentStream = node.contentStream;
894                 let decodeStream = this.decoder.decodeStream(charset);
895                 contentStream.on('error', err => {
896                     decodeStream.emit('error', err);
897                 });
898                 contentStream.pipe(decodeStream);
899                 node.contentStream = decodeStream;
900             } catch (E) {
901                 // do not decode charset
902             }
903         }
904
905         node.contentStream.on('readable', () => {
906             let chunk;
907             while ((chunk = node.contentStream.read()) !== null) {
908                 if (typeof chunk === 'string') {
909                     chunk = Buffer.from(chunk);
910                 }
911                 chunks.push(chunk);
```

```

912         chunklen += chunk.length;
913     }
914     });
915
916     node.contentStream.once('end', () => {
917         node.textContent = Buffer.concat(chunks, chunklen).toString().repl
918     });
919
920     node.contentStream.once('error', err => {
921         this.emit('error', err);
922     });
923     }
924
925     break;
926 }
927
928 case 'data':
929     if (this.curnode && this.curnode.decoder) {
930         this.curnode.decoder.end();
931     }
932
933     if (this.waitUntilAttachmentEnd) {
934         this.attachmentCallback = done;
935         return;
936     }
937
938     // multipart message structure
939     // this is not related to any specific 'node' block as it includes
940     // everything between the end of some node body and between the next head
941     //process.stdout.write(data.value);
942     break;
943
944 case 'body':
945     if (this.curnode && this.curnode.decoder && this.curnode.decoder.writable)
946         if (this.curnode.decoder.write(data.value) === false) {
947             return this.curnode.decoder.once('drain', done);
948         }
949     }
950
951     // Leaf element body. Includes the body for the last 'node' block. You mig
952     // have several 'body' calls for a single 'node' block
953     //process.stdout.write(data.value);
954     break;
955 }
956
957     setImmediate(done);
958 }
959
960 _getPartId(parentBoundary) {

```

```

961     let boundaryIndex = this.boundaries.findIndex(item => item.name === parentBoundary
962     if (boundaryIndex === -1) {
963         this.boundaries.push({ name: parentBoundary, count: 1 });
964         boundaryIndex = this.boundaries.length - 1;
965     } else {
966         this.boundaries[boundaryIndex].count++;
967     }
968     let partId = '1';
969     for (let i = 0; i <= boundaryIndex; i++) {
970         if (i === 0) partId = this.boundaries[i].count.toString();
971         else partId += '.' + this.boundaries[i].count.toString();
972     }
973     return partId;
974 }
975
976 getAddressessHTML(value) {
977     let formatSingleLevel = addresses =>
978         addresses
979             .map(address => {
980                 let str = '<span class="mp_address_group">';
981                 if (address.name) {
982                     str += '<span class="mp_address_name">' + he.encode(address.name)
983                 }
984                 if (address.address) {
985                     let link = '<a href="mailto:' + he.encode(address.address) + '" cl
986                     if (address.name) {
987                         str += ' &lt;' + link + '&gt;';
988                     } else {
989                         str += link;
990                     }
991                 }
992                 if (address.group) {
993                     str += formatSingleLevel(address.group) + ' ';
994                 }
995                 return str + '</span>';
996             })
997             .join(', ');
998     return formatSingleLevel([], [].concat(value || []));
999 }
1000
1001 getAddressessText(value) {
1002     let formatSingleLevel = addresses =>
1003         addresses
1004             .map(address => {
1005                 let str = '';
1006                 if (address.name) {
1007                     str += address.name + (address.group ? ': ' : '');
1008                 }
1009                 if (address.address) {

```

```

1010         let link = address.address;
1011         if (address.name) {
1012             str += ' <' + link + '>';
1013         } else {
1014             str += link;
1015         }
1016     }
1017     if (address.group) {
1018         str += formatSingleLevel(address.group) + ';';
1019     }
1020     return str;
1021 })
1022 .join(', ');
1023 return formatSingleLevel([].concat(value || []));
1024 }
1025
1026 updateImageLinks(replaceCallback, done) {
1027     if (!this.html) {
1028         return setImmediate(() => done(null, false));
1029     }
1030
1031     let cids = new Map();
1032     let html = (this.html || '').toString();
1033
1034     if (this.options.skipImageLinks) {
1035         return done(null, html);
1036     }
1037
1038     html.replace(/\bcid:([\^"\s]{1,256})/g, (match, cid) => {
1039         for (let i = 0, len = this.attachmentList.length; i < len; i++) {
1040             if (this.attachmentList[i].cid === cid && /^image\/[\w]+$/i.test(this.atta
1041                 if (/^image\/[\w]+$/i.test(this.attachmentList[i].contentType)) {
1042                 cids.set(cid, {
1043                     attachment: this.attachmentList[i]
1044                 });
1045             }
1046             break;
1047         }
1048     }
1049     return match;
1050 });
1051
1052 let cidList = [];
1053 cids.forEach(entry => {
1054     cidList.push(entry);
1055 });
1056
1057 let pos = 0;
1058 let processNext = () => {

```

```

1059         if (pos >= cidList.length) {
1060             html = html.replace(/\bcid:([\^'"\\s]{1,256})/g, (match, cid) => {
1061                 if (cids.has(cid) && cids.get(cid).url) {
1062                     return cids.get(cid).url;
1063                 }
1064                 return match;
1065             });
1066
1067             return done(null, html);
1068         }
1069         let entry = cidList[pos++];
1070         replaceCallback(entry.attachment, (err, url) => {
1071             if (err) {
1072                 return setImmediate(() => done(err));
1073             }
1074             entry.url = url;
1075             setImmediate(processNext);
1076         });
1077     };
1078
1079     setImmediate(processNext);
1080 }
1081
1082 textToHtml(str) {
1083     if (this.options.skipTextToHtml) {
1084         return '';
1085     }
1086     str = (str || '').toString();
1087     let encoded;
1088
1089     let linkified = false;
1090     if (!this.options.skipTextLinks) {
1091         try {
1092             if (linkify.pretest(str)) {
1093                 linkified = true;
1094                 let links = linkify.match(str) || [];
1095                 let result = [];
1096                 let last = 0;
1097
1098                 links.forEach(link => {
1099                     if (last < link.index) {
1100                         let textPart = he
1101                             // encode special chars
1102                             .encode(str.slice(last, link.index), {
1103                                 useNamedReferences: true
1104                             });
1105                         result.push(textPart);
1106                     }
1107

```

```

1108         result.push(``\);
1109
1110         last = link.lastIndex;
1111     }\);
1112
1113     let textPart = he
1114         // encode special chars
1115         .encode\(str.slice\(last\), {
1116             useNamedReferences: true
1117         }\);
1118     result.push\(textPart\);
1119
1120     encoded = result.join\(''\);
1121     }
1122 } catch \(E\) {
1123     // failed, don't linkify
1124 }
1125 }
1126
1127 if \(!linkified\) {
1128     encoded = he
1129         // encode special chars
1130         .encode\(str, {
1131             useNamedReferences: true
1132         }\);
1133 }
1134
1135 let text =
1136     '<p>' +
1137     encoded
1138         .replace\(/\r?\n/g, '\n'\)
1139         .trim\(\) // normalize line endings
1140         .replace\(/\[ \t\]+\$/gm, ''\)
1141         .trim\(\) // trim empty line endings
1142         .replace\(/\n\n+/g, '</p><p>'\)
1143         .trim\(\) // insert <p> to multiple linebreaks
1144         .replace\(/\n/g, '<br/>'\) + // insert <br> to single linebreaks
1145     '</p>';
1146
1147     return text;
1148 }
1149 }
1150
1151 module.exports = MailParser;

```