
mississippi

a collection of useful stream utility modules. learn how the modules work using this and then pick the ones you want and use them individually

the goal of the modules included in mississippi is to make working with streams easy without sacrificing speed, error handling or composability.

usage

```
1 var miss = require('mississippi')
```

methods

- pipe
- each
- pipeline
- duplex
- through
- from
- to
- concat
- finished
- parallel

pipe

miss.pipe(stream1, stream2, stream3, ..., cb) Pipes streams together and destroys all of them if one of them closes. Calls **cb** with (**error**) if there was an error in any of the streams.

When using standard **source.pipe(destination)** the source will *not* be destroyed if the destination emits close or error. You are also not able to provide a callback to tell when the pipe has finished.

miss.pipe does these two things for you, ensuring you handle stream errors 100% of the time (unhandled errors are probably the most common bug in most node streams code)

original module `miss.pipe` is provided by `require('pump')`

example

```
1 // lets do a simple file copy
2 var fs = require('fs')
3
4 var read = fs.createReadStream('./original.zip')
5 var write = fs.createWriteStream('./copy.zip')
6
7 // use miss.pipe instead of read.pipe(write)
8 miss.pipe(read, write, function (err) {
9   if (err) return console.error('Copy error!', err)
10  console.log('Copied successfully')
11 })
```

each

miss.each(stream, each, [done]) Iterate the data in `stream` one chunk at a time. Your `each` function will be called with (`data`, `next`) where `data` is a data chunk and `next` is a callback. Call `next` when you are ready to consume the next chunk.

Optionally you can call `next` with an error to destroy the stream. You can also pass the optional third argument, `done`, which is a function that will be called with (`err`) when the stream ends. The `err` argument will be populated with an error if the stream emitted an error.

original module `miss.each` is provided by `require('stream-each')`

example

```
1 var fs = require('fs')
2 var split = require('binary-split') // require('split2') would work
   here as well
3
4 var newLineSeparatedNumbers = fs.createReadStream('numbers.txt')
5
6 var pipeline = miss.pipeline(newLineSeparatedNumbers, split())
7 miss.each(pipeline, eachLine, done)
8 var sum = 0
9
10 function eachLine (line, next) {
11   sum += parseInt(line.toString())
12   next()
13 }
14
15 function done (err) {
16   if (err) throw err
```

```
17 console.log('sum is', sum)
18 }
```

pipeline

var pipeline = miss.pipeline(stream1, stream2, stream3, ...) Builds a pipeline from all the transform streams passed in as arguments by piping them together and returning a single stream object that lets you write to the first stream and read from the last stream.

If you are pumping object streams together use `pipeline = miss.pipeline.obj(s1, s2, ...)`.

If any of the streams in the pipeline emits an error or gets destroyed, or you destroy the stream it returns, all of the streams will be destroyed and cleaned up for you.

original module `miss.pipeline` is provided by `require('pumpify')`

example

```
1 // first create some transform streams (note: these two modules are
  fictional)
2 var imageResize = require('image-resizer-stream')({width: 400})
3 var pngOptimizer = require('png-optimizer-stream')({quality: 60})
4
5 // instead of doing a.pipe(b), use pipeline
6 var resizeAndOptimize = miss.pipeline(imageResize, pngOptimizer)
7 // `resizeAndOptimize` is a transform stream. when you write to it, it
  writes
8 // to `imageResize`. when you read from it, it reads from `pngOptimizer`
  .
9 // it handles piping all the streams together for you
10
11 // use it like any other transform stream
12 var fs = require('fs')
13
14 var read = fs.createReadStream('./image.png')
15 var write = fs.createWriteStream('./resized-and-optimized.png')
16
17 miss.pipe(read, resizeAndOptimize, write, function (err) {
18   if (err) return console.error('Image processing error!', err)
19   console.log('Image processed successfully')
20 })
```

duplex

var duplex = miss.duplex([writable, readable, opts]) Take two separate streams, a writable and a readable, and turn them into a single duplex (readable and writable) stream.

The returned stream will emit data from the readable. When you write to it it writes to the writable.

You can either choose to supply the writable and the readable at the time you create the stream, or you can do it later using the `.setWritable` and `.setReadable` methods and data written to the stream in the meantime will be buffered for you.

original module `miss.duplex` is provided by `require('duplexify')`

example

```
1 // lets spawn a process and take its stdout and stdin and combine them
  into 1 stream
2 var child = require('child_process')
3
4 // @- tells it to read from stdin, --data-binary sets 'raw' binary mode
5 var curl = child.spawn('curl -X POST --data-binary @- http://foo.com')
6
7 // duplexCurl will write to stdin and read from stdout
8 var duplexCurl = miss.duplex(curl.stdin, curl.stdout)
```

through

var transformer = miss.through([options, transformFunction, flushFunction]) Make a custom transform stream.

The `options` object is passed to the internal transform stream and can be used to create an `objectMode` stream (or use the shortcut `miss.through.obj([...])`)

The `transformFunction` is called when data is available for the writable side and has the signature `(chunk, encoding, cb)`. Within the function, add data to the readable side any number of times with `this.push(data)`. Call `cb()` to indicate processing of the `chunk` is complete. Or to easily emit a single error or chunk, call `cb(err, chunk)`

The `flushFunction`, with signature `(cb)`, is called just before the stream is complete and should be used to wrap up stream processing.

original module `miss.through` is provided by `require('through2')`

example

```
1 var fs = require('fs')
2
3 var read = fs.createReadStream('./boring_lowercase.txt')
4 var write = fs.createWriteStream('./AWESOMECASE.TXT')
5
6 // Leaving out the options object
7 var upercaser = miss.through(
8   function (chunk, enc, cb) {
9     cb(null, chunk.toString().toUpperCase())
10  },
11  function (cb) {
12    cb(null, 'ONE LAST BIT OF UPPERCASE')
13  }
14 )
15
16 miss.pipe(read, upercaser, write, function (err) {
17   if (err) return console.error('Trouble uppercasing!')
18   console.log('Splendid uppercasing!')
19 })
```

from

miss.from([opts], read) Make a custom readable stream.

opts contains the options to pass on to the ReadableStream constructor e.g. for creating a readable object stream (or use the shortcut `miss.from.obj([...])`).

Returns a readable stream that calls `read(size, next)` when data is requested from the stream.

- `size` is the recommended amount of data (in bytes) to retrieve.
- `next(err, chunk)` should be called when you're ready to emit more data.

original module `miss.from` is provided by `require('from2')`

example

```
1
2
3 function fromString(string) {
4   return miss.from(function(size, next) {
5     // if there's no more content
6     // left in the string, close the stream.
7     if (string.length <= 0) return next(null, null)
8
9     // Pull in a new chunk of text,
10    // removing it from the string.
```

```
11     var chunk = string.slice(0, size)
12     string = string.slice(size)
13
14     // Emit "chunk" from the stream.
15     next(null, chunk)
16   })
17 }
18
19 // pipe "hello world" out
20 // to stdout.
21 fromString('hello world').pipe(process.stdout)
```

to

miss.to([options], write, [flush]) Make a custom writable stream.

opts contains the options to pass on to the WritableStream constructor e.g. for creating a writable object stream (or use the shortcut `miss.to.obj([...])`).

Returns a writable stream that calls `write(data, enc, cb)` when data is written to the stream.

- **data** is the received data to write the destination.
- **enc** encoding of the piece of data received.
- **cb(err, data)** should be called when you're ready to write more data, or encountered an error.

flush(cb) is called before **finish** is emitted and allows for cleanup steps to occur.

original module `miss.to` is provided by `require('flush-write-stream')`

example

```
1 var ws = miss.to(write, flush)
2
3 ws.on('finish', function () {
4   console.log('finished')
5 })
6
7 ws.write('hello')
8 ws.write('world')
9 ws.end()
10
11 function write (data, enc, cb) {
12   // i am your normal ._write method
13   console.log('writing', data.toString())
14   cb()
```

```
15 }
16
17 function flush (cb) {
18   // i am called before finish is emitted
19   setTimeout(cb, 1000) // wait 1 sec
20 }
```

If you run the above it will produce the following output

```
1 writing hello
2 writing world
3 (nothing happens for 1 sec)
4 finished
```

concat

var concat = miss.concat(cb) Returns a writable stream that concatenates all data written to the stream and calls a callback with the single result.

Calling `miss.concat(cb)` returns a writable stream. `cb` is called when the writable stream is finished, e.g. when all data is done being written to it. `cb` is called with a single argument, (`data`), which will contain the result of concatenating all the data written to the stream.

Note that `miss.concat` will not handle stream errors for you. To handle errors, use `miss.pipe` or handle the `error` event manually.

original module `miss.concat` is provided by `require('concat-stream')`

example

```
1 var fs = require('fs')
2
3 var readStream = fs.createReadStream('cat.png')
4 var concatStream = miss.concat(gotPicture)
5
6 function callback (err) {
7   if (err) {
8     console.error(err)
9     process.exit(1)
10  }
11 }
12
13 miss.pipe(readStream, concatStream, callback)
14
15 function gotPicture(imageBuffer) {
16   // imageBuffer is all of `cat.png` as a node.js Buffer
17 }
```

```
18
19 function handleError(err) {
20   // handle your error appropriately here, e.g.:
21   console.error(err) // print the error to STDERR
22   process.exit(1) // exit program with non-zero exit code
23 }
```

finished

miss.finished(stream, cb) Waits for `stream` to finish or error and then calls `cb` with (`err`). `cb` will only be called once. `err` will be null if the stream finished without error, or else it will be populated with the error from the streams `error` event.

This function is useful for simplifying stream handling code as it lets you handle success or error conditions in a single code path. It's used internally `miss.pipe`.

original module `miss.finished` is provided by `require('end-of-stream')`

example

```
1 var copySource = fs.createReadStream('./movie.mp4')
2 var copyDest = fs.createWriteStream('./movie-copy.mp4')
3
4 copySource.pipe(copyDest)
5
6 miss.finished(copyDest, function(err) {
7   if (err) return console.log('write failed', err)
8   console.log('write success')
9 })
```

parallel

miss.parallel(concurrency, each) This works like `through` except you can process items in parallel, while still preserving the original input order.

This is handy if you wanna take advantage of node's async I/O and process streams of items in batches. With this module you can build your very own streaming parallel job queue.

Note that `miss.parallel` preserves input ordering. Passing the option `{ordered:false}` will output the data as soon as it's processed by a transform, without waiting to respect the order (this previously required a separate module `through2-concurrent`).

original module `miss.parallel` is provided by `require('parallel-transform')`

example This example fetches the GET HTTP headers for a stream of input URLs 5 at a time in parallel.

```
1 function getResponse (item, cb) {
2   var r = request(item.url)
3   r.on('error', function (err) {
4     cb(err)
5   })
6   r.on('response', function (re) {
7     cb(null, {url: item.url, date: new Date(), status: re.statusCode,
8       headers: re.headers})
9     r.abort()
10  })
11 }
12 miss.pipe(
13   fs.createReadStream('./urls.txt'), // one url per line
14   split(),
15   miss.parallel(5, getResponse),
16   miss.through(function (row, enc, next) {
17     console.log(JSON.stringify(row))
18     next()
19   })
20 )
```

see also

- [substack/stream-handbook](#)
- [nodejs.org/api/stream.html](#)
- [awesome-nodejs-streams](#)

license

Licensed under the BSD 2-clause license.