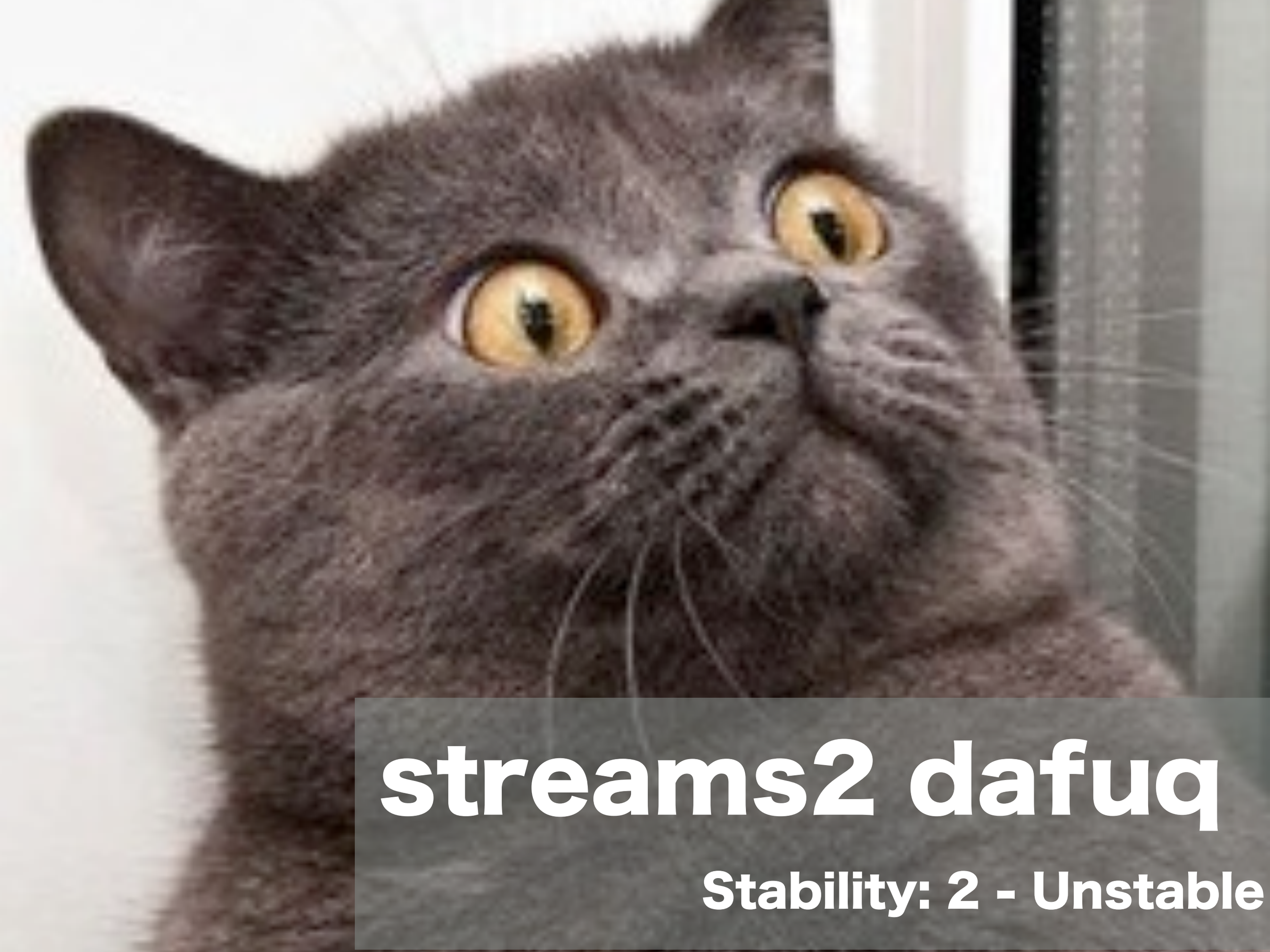
Node.js
0.10.x

@indexzero
www.github.com/indexzero

THE LOUD VERSION



streams2 dafuq

Stability: 2 - Unstable

Did you say streams?

- fs.ReadStream, fs.WriteStream
- tcp/tls Socket objects
- http request/response
- zlib, stdio, etc

.pause() is unreliable

push model for data isn't practical

Inconsistency in emitted events

Implementing your own stream is too hard

npm view readable-stream

The Streams2 API was developed while using it for modules in the npm registry. At the time of this writing, 37 published Node modules already are using the readable-stream library as a dependency.

"The readable-stream npm package allows you to use the new Stream interface in your legacy v0.8 codebase."


“Old programs will almost always work without modification, but streams start out in a paused state, and need to be read from to be consumed.”

Streams are paused by default

```
// WARNING!  BROKEN!
net.createServer(function(socket) {

  // we add an 'end' method, but never consume the data
  socket.on('end', function() {
    // It will never get here.
    socket.end('I got your message (but didnt read it)\n');
  });

}).listen(1337);
```

So you need to resume them

```
// Workaround
net.createServer(function(socket) {

  socket.on('end', function() {
    socket.end('I got your message (but didnt read it)\n');
  });

  // start the flow of data, discarding it.
  socket.resume();

}).listen(1337);
```

Push model for data?

Rather than waiting for you to call the read() method, 'data' events would start emitting immediately.

Cannonical example - http-proxy

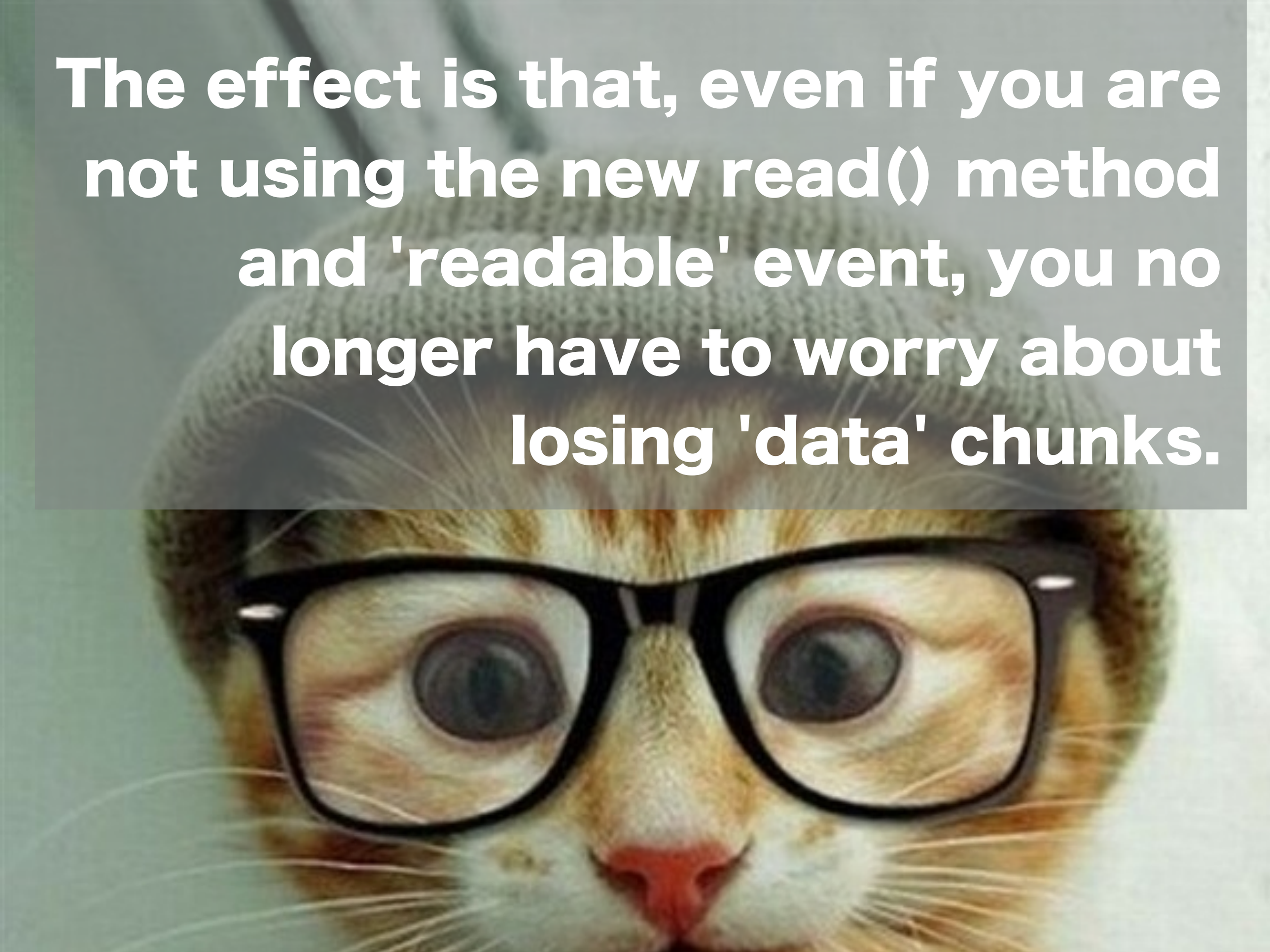
```
var fs = require('fs'),
    httpProxy = require('http-proxy');

// Create a proxy server with custom application logic
httpProxy.createServer(function (req, res, proxy) {
  fs.readFile('some-file', function (err, data) {
    // ALL REQUEST DATA IS LOST
    proxy.proxyRequest(req, res, {
      host: 'localhost',
      port: 9000
    });
  })
}).listen(8000);
```

If you needed to do some I/O to decide how to handle data, then you had to store the chunks in some kind of buffer so that they would not be lost.

http-proxy - buffered requests

```
var fs = require('fs'),
    httpProxy = require('http-proxy');

// Create a proxy server with custom application logic
httpProxy.createServer(function (req, res, proxy) {
  // Buffer the request data so it is retained
  var buffer = httpProxy.buffer(req);
  fs.readFile('some-file', function (err, data) {
    proxy.proxyRequest(req, res, {
      buffer: buffer,
      host: 'localhost',
      port: 9000
    });
  })
}).listen(8000);
```


IT’S TOO MUCH WORK!


The effect is that, even if you are not using the new read() method and 'readable' event, you no longer have to worry about losing 'data' chunks.

http-proxy - streams2

```
var fs = require('fs'),
    httpProxy = require('http-proxy');

// Create a proxy server with custom application logic
httpProxy.createServer(function (req, res, proxy) {
  fs.readFile('some-file', function (err, data) {
    // THIS ACTUALLY WORKS!
    proxy.proxyRequest(req, res, {
      host: 'localhost',
      port: 9000
    });
  })
}).listen(8000);
```

A Readable Stream has the following methods, members, and events.

Note that stream.Readable is an abstract class designed to be extended with an underlying implementation of the _read(size) method. (See below.)

NEW STREAM.READABLE([OPTIONS])

options {Object}

• **highWaterMark {Number}** The maximum number of bytes to store in the internal buffer before ceasing to read from the underlying resource. Default=16kb

• **encoding {String}** If specified, then buffers will be decoded to strings using the specified encoding. Default=null

• **objectMode {Boolean}** Whether this stream should behave as a stream of objects. Meaning that stream.read(n) returns a single value instead of a Buffer of size n

READABLE._READ(SIZE)

- **size {Number}** Number of bytes to read asynchronously

Note: This function should NOT be called directly. It should be implemented by child classes, and called by the internal Readable class methods only.

All Readable stream implementations must provide a _read method to fetch data from the underlying resource.

READABLE.PUSH/UNSHIFT(CHUNK)

- **chunk {Buffer | null | String}** Chunk of data to push (or unshift) into the read queue
- **return {Boolean}** Whether or not more pushes should be performed

Note: This function should be called by Readable implementors, NOT by consumers of Readable subclasses. The _read() function will not be called again until at least one push(chunk) call is made. If no data is available, then you MAY call push('') (an empty string) to allow a future _read call, without adding any data to the queue.

The Readable class works by putting data into a read queue to be pulled out later by calling the read() method when the 'readable' event fires.

The push() method will explicitly insert some data into the read queue. If it is called with null then it will signal the end of the data.

READABLE.WRAP(STREAM)

- **stream {Stream}** An "old style" readable stream

If you are using an older Node library that emits 'data' events and has a pause() method that is advisory only, then you can use the wrap() method to create a Readable stream that uses the old stream as its data source.

```
var OldReader = require('./old-module.js').OldReader;
var oreader = new OldReader;
var Readable = require('stream').Readable;
var myReader = new Readable().wrap(oreader);

myReader.on('readable', function() {
  myReader.read(); // etc.
});
```

READABLESTREAM : EVENTS

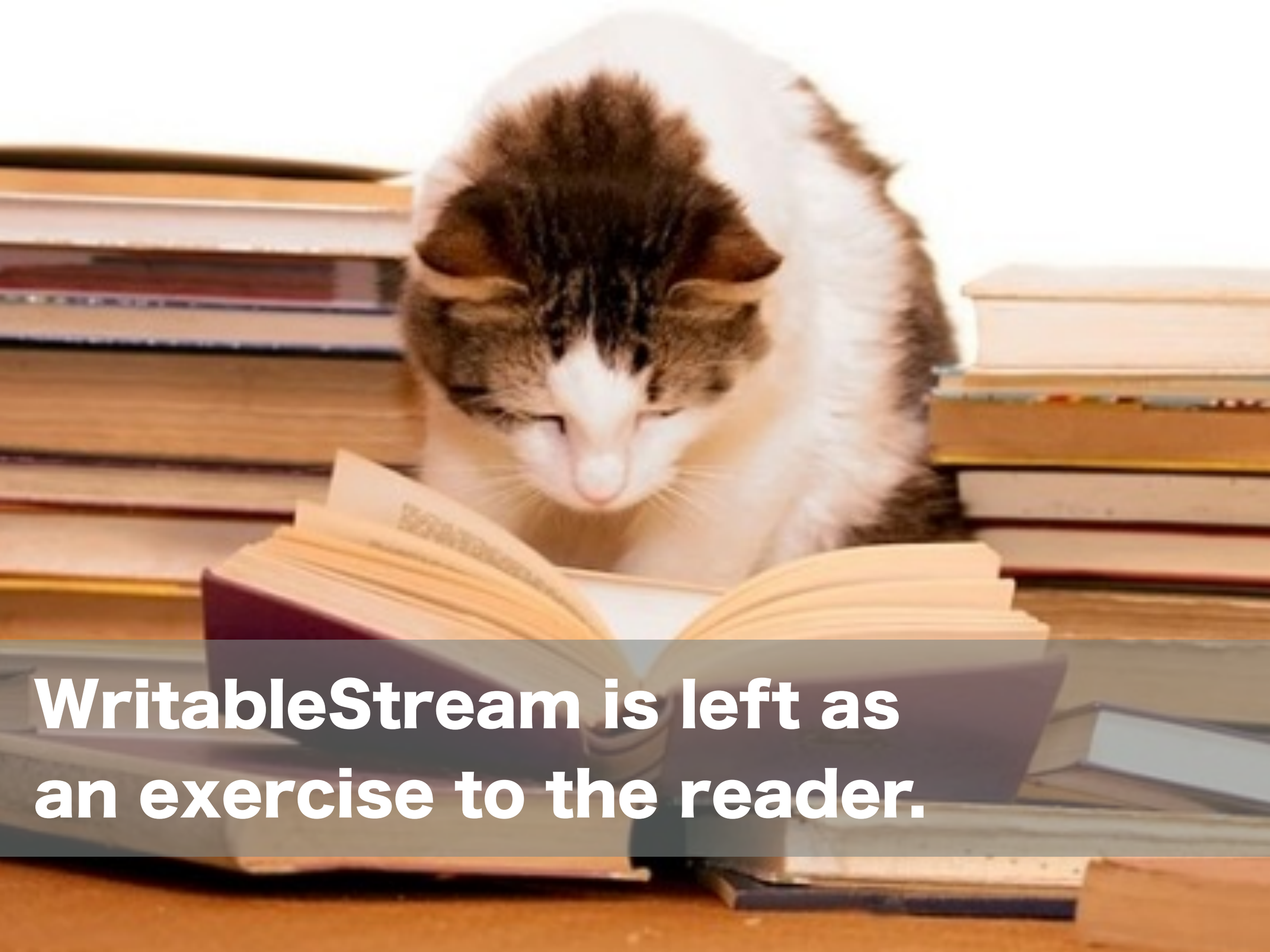
• **'READABLE':** When there is data ready to be consumed, this event will fire. When this event emits, call the read() method to consume the data.

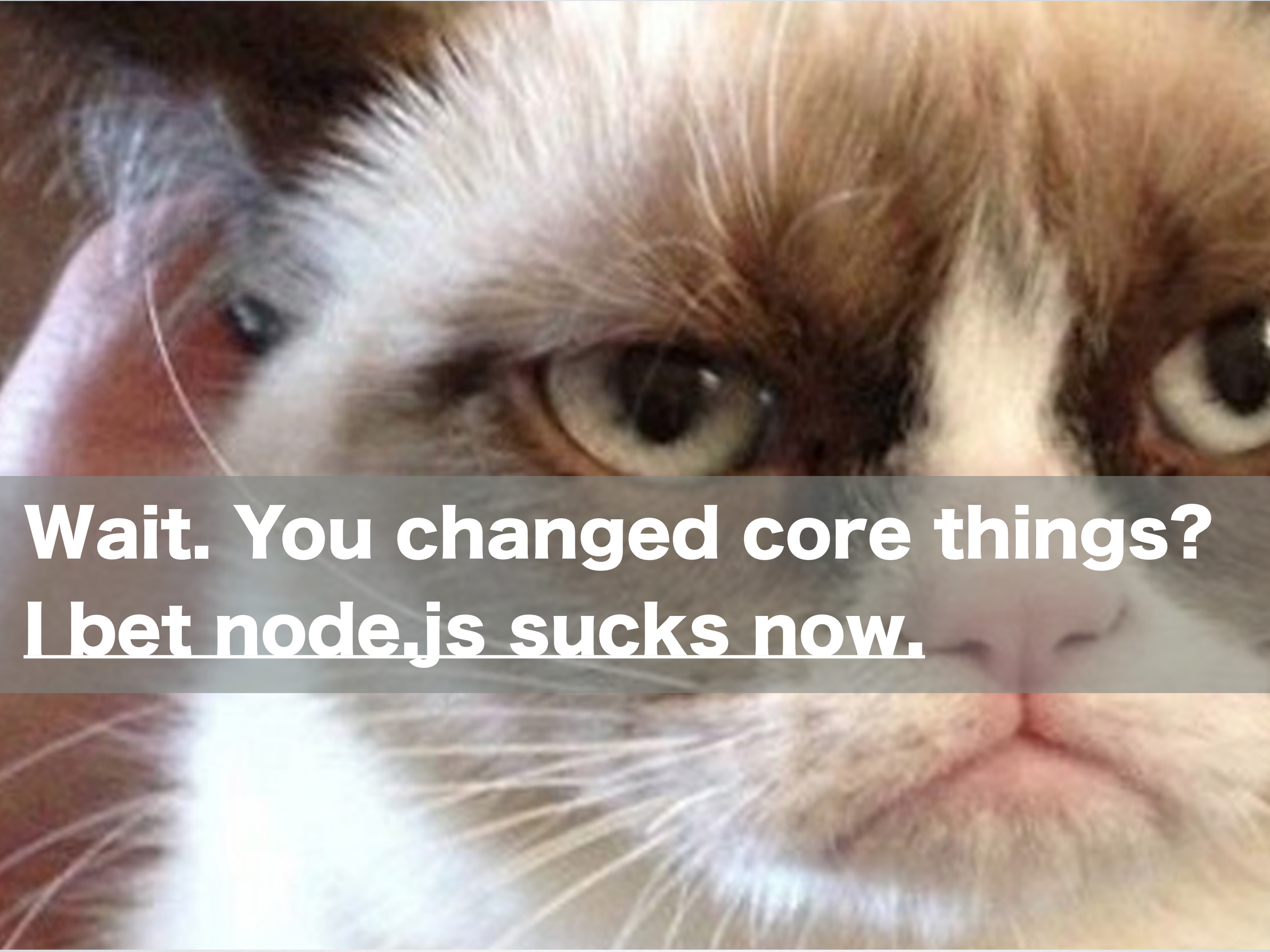
• **'END':** Emitted when the stream has received an EOF (FIN in TCP terminology). Indicates that no more 'data' events will happen. If the stream is also writable, it may be possible to continue writing.

• **'ERROR':** Emitted if there was an error receiving data.

• **'CLOSE':** Emitted when the underlying resource (for example, the backing file descriptor) has been closed. *Not all streams will emit this*.

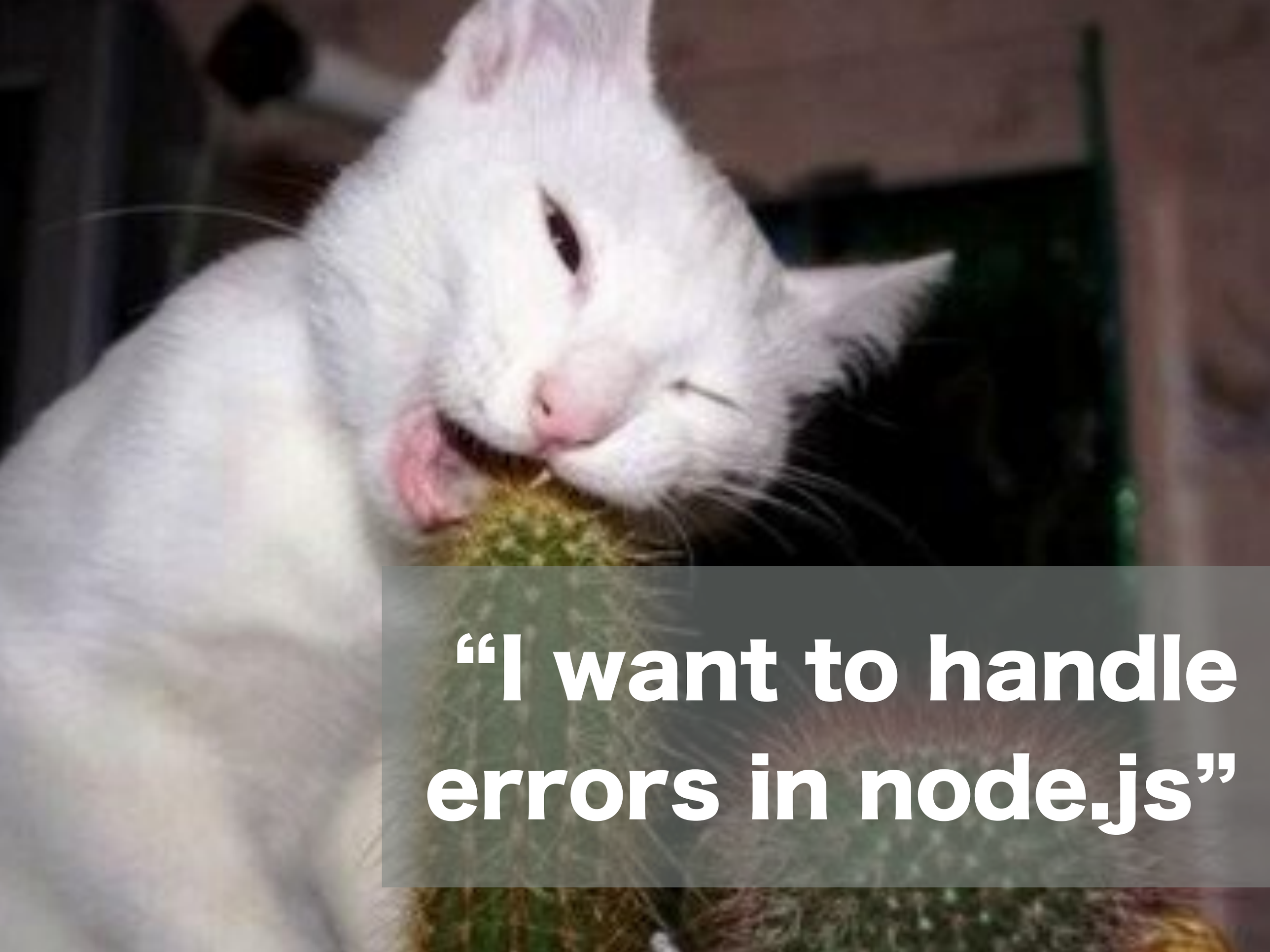
• **'DATA':** The 'data' event emits either a Buffer (by default) or a string if setEncoding() was used. *Note that adding a 'data' event listener will switch the Readable stream into "old mode", where data is emitted as soon as it is available, rather than waiting for you to call read() to consume it.*


WritableStream is left as
an exercise to the reader.


Wait. You changed core things?
I bet node.js sucks now.

“We are fanatical about maintaining performance in Node.js, so of course this will have to be fixed before the v0.10 stable release.”

http://blog.nodejs.org/2013/03/11/node-v0-10-0-stable/


“I want to handle
errors in node.js”

Read a file and handle errors

```
var fs = require('fs');

fs.readFile('some/file.txt', 'utf8', function (err, data) {
  if (err) {
    console.log('Error reading some/file.txt');
    console.dir(err);
    return;
  }

  console.dir(data);
});
```


Sure ok. Now handle a bunch of unrelated errors.

Multiple unrelated errors

```
var domain = require('domain').create(),
    fs = require('fs');

fs.readFile('some/file.txt', 'utf8',
domain.intercept(function (data) {
  console.dir(data);
});

domain.on('error', function (err) {
  console.log('Error in this domain');
  console.dir(err);
});
```

0.8.x - domains

```
process.on('uncaughtException',
function (err) { });
```

Stability: 3 - Experimental

0.10.x - domains

Domain error handler no longer relies on process.on('uncaughtException') being raised, and the C++ code in Node is domain-aware.

Stability: 2 - Unstable

Complex domains example

```
// create a top-level domain for the server
var serverDomain = domain.create();

serverDomain.run(function() {
  // server is created in the scope of serverDomain
  http.createServer(function(req, res) {
    // req and res are also created in the scope of serverDomain
    // however, we'd prefer to have a separate domain for each request.
    // create it first thing, and add req and res to it.
    var reqd = domain.create();
    reqd.add(req);
    reqd.add(res);
    reqd.on('error', function(er) {
      console.error('Error', er, req.url);
      try {
        res.writeHead(500);
        res.end('Error occurred, sorry.');
        res.on('close', function() {
          // forcibly shut down any other things added to this domain
          reqd.dispose();
        });
      } catch (er) {
        console.error('Error sending 500', er, req.url);
        // tried our best.  clean up anything remaining.
        reqd.dispose();
      }
    });
  }).listen(1337);
});
```


This is not automatic. You have to write your code to be **domain aware.**

“Do something in the next event loop tick”

0.8.x - process.nextTick()

"The process.nextTick() function scheduled its callback using a spinner on the event loop. This usually caused the callback to be fired before any other I/O. However, it was not guaranteed."

"As a result, a lot of programs (including some parts of Node's internals) began using process.nextTick as a "do later, but before any actual I/O is performed" interface. Since it usually works that way, it seemed fine."

process.nextTick()

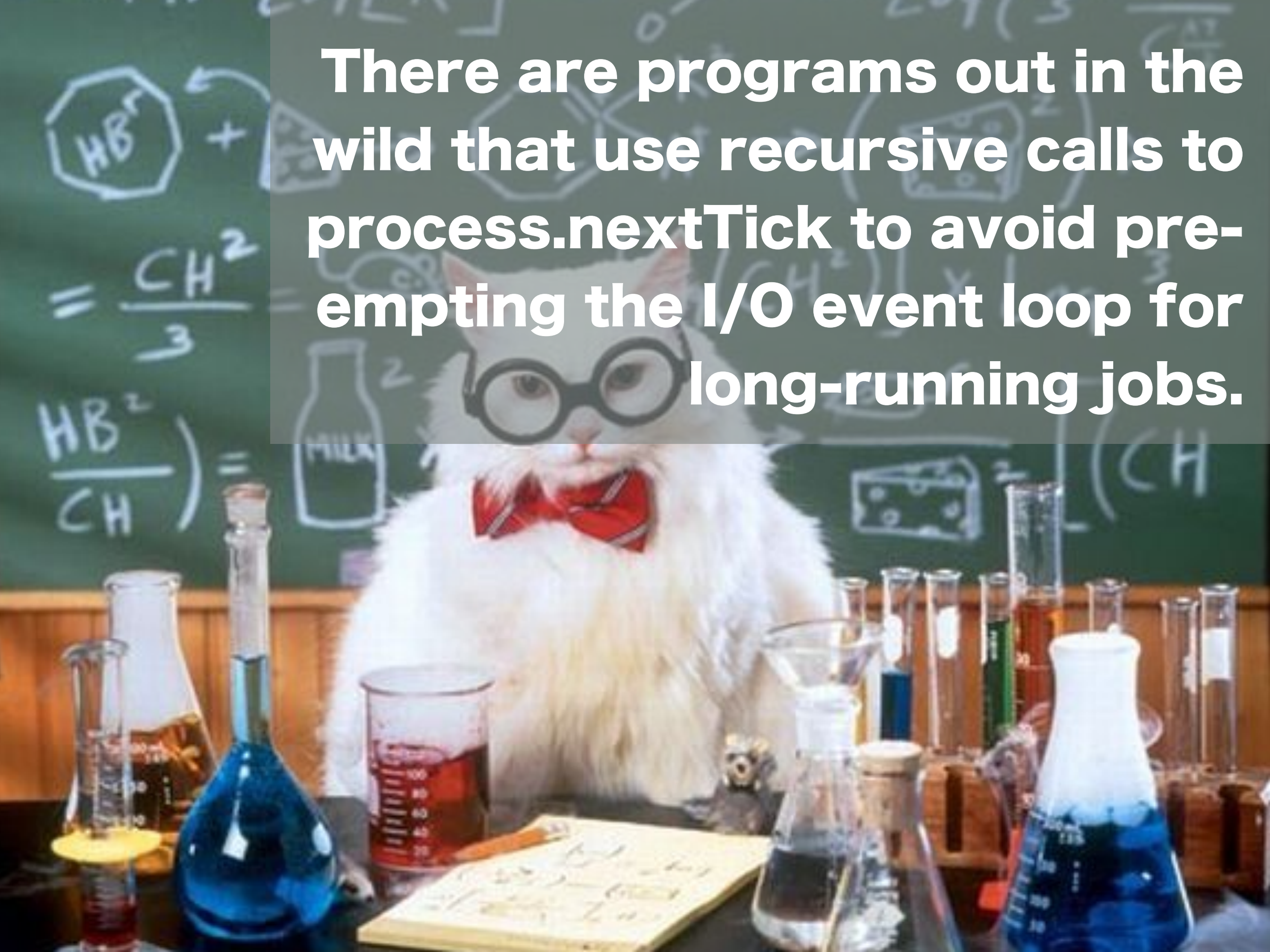
```
var events = require('events');

module.exports = function () {
  // Return an EventEmitter immediately and
  // emit an event in the nextTick.
  var emitter = new events.EventEmitter();

  process.nextTick(function () {
    emitter.emit('ready');
  });

  return emitter;
};
```

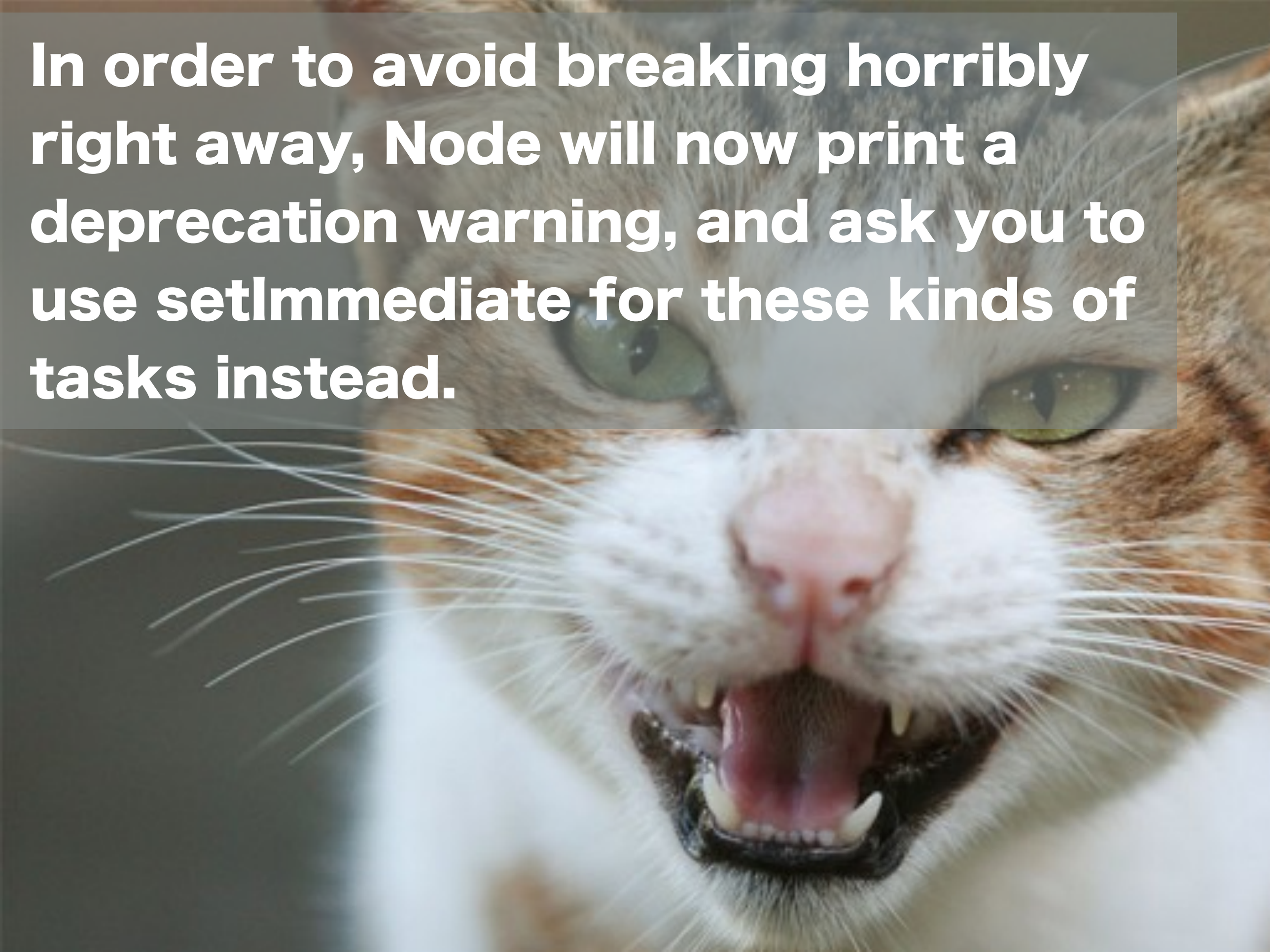
0.10.x - process.nextTick()

"nextTick handlers are run right after each call from C++ into JavaScript. That means that, if your JavaScript code calls process.nextTick, then the callback will fire as soon as the code runs to completion, but before going back to the event loop. The race is over, and all is good."

"As a result, a lot of programs (including some parts of Node's internals) began using process.nextTick as a "do later, but before any actual I/O is performed" interface. Since it usually works that way, it seemed fine."

There are programs out in the wild that use recursive calls to process.nextTick to avoid pre-empting the I/O event loop for long-running jobs.




In order to avoid breaking horribly right away, Node will now print a deprecation warning, and ask you to use setImmediate for these kinds of tasks instead.

“Node.js got beter at garbage collection”

0.8.x - V8 Gabage Collection

"Node used to try to tell V8 to collect some garbage whenever the event loop was idle."

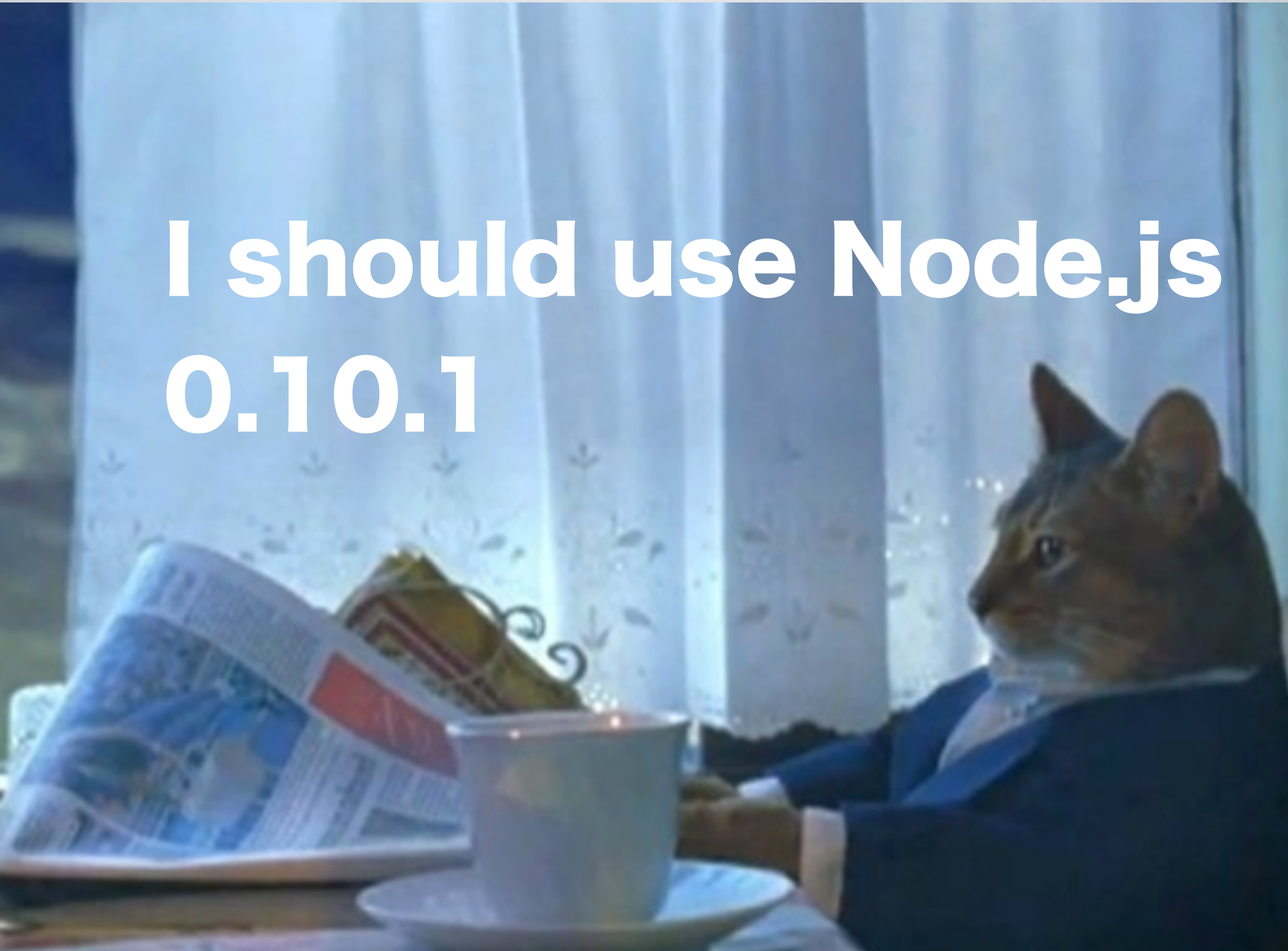
"In practice, disabling the IdleNotification call yields better performance without any excessive memory usage, because V8 is pretty good at knowing when it's the best time to run GC."

0.8.x - V8 Gabage Collection

"In v0.10, we just ripped that feature out. (According to another point of view, we fixed the bug that it was ever there in the first place.) As a result, latency is much more predictable and stable."

"You won't see a difference in the benchmarks as a result of this, but you'll probably find that your app's response times are more reliable."

I should use Node.js 0.10.1



FIN