STREAMS

PAST.pipe(PRESENT).pipe(FUTURE)

Isaac Z. Schlueter

STREAMS

PAST.pipe(PRESENT).pipe(FUTURE)

Isaac Z. Schlueter

hi, I'm Isaac, I'm here to talk about streams in node.js, and about how they're changing in the next version of Node.

In node, we >> stream all the things.

STREAM ALL THE THINGS

stream all the things.

Any time we have data flowing, it uses a "stream" interface

stream things (core)

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

Just a quick non-definitive list, fs streams, tcp sockets, tls sockets, http request and response objects, zlib, stdio, and a few others, and that's just in node core.

stream things (npm)

- request, tar, muxdemux, shoe, browserify, etc.
- almost any popular module
- Anything with a .pipe() method, write() method, or 'data'/'end' events

In npm, most modules that work with data use this interface.

After experimenting with node, we've found it's just easier to get more done if we have consistent interfaces.

Thursday, November 22, 12

Streams are like lego blocks.
They're very easy to use, because they're consistent.
You can plug one thing into the other, even if they aren't related originally.

JUST WORKS

It just works.

But it wasn't this easy when node started.

I want you to understand what's coming, and how we got here, so that you can understand the problem that we're trying to solve.

So, this is a story about the history of node and streams.

the road to streaming

Once upon a time, Node had a lot of different interfaces for all these things.

pre-streams

- "Evented I/O for V8 JavaScript"
- http, fs, tcp: all different interfaces
- Different event names, methods
- Each made sense, but they didn't match, which was bad

Node is "evented IO for V8 JavaScript".

The goal was to re-invent the minimum necessary,
and stick to conventions when possible.

But these conventions didn't always match.

For example, http requests had "body" events instead of "data"

streams0

- Readable
  r.emit('data', chunk) ; r.emit('end')
  r.pause() ; r.resume()
- Writable
  w.write(chunk) ; w.end()
  w.on('drain', writeMore)

So, we had our first streaming interfaces in node in 0.1. This was a powerful change.

If a function operated on file system or tcp or http streams, then it could be reused in other places.

The write() method returned true if it was flushed, or false if it wasn't,
so you could manage back-pressure effectively,
which makes it easy to manage memory overhead.

If you're sending a big file to a client, you don't want to buffer the whole thing.

BUT WAIT!

THERE'S MORE!

So that was great, but even better was the >> util.pump method

util.pump(readable, writable);

>> util.pump method.

You specify a reader, and a writer, and it manages backpressure, sets event handlers, and does the right things.

This was awesome

STOP!

Don't use util.pump()

but it falls short

util.pump fail

- Real streams have many other events: 'close', 'error', etc.
- No feature detection
- No customization
- No signal that pumping has started

util.pump didn't handle all the events that are important, because even though the interface was now mostly consistent, there were still some edge cases.

Also, since it's an extra method that lives outside the Stream objects themselves, there was no way to customize it.

So, if you have some custom userland stream that needs to do custom things when it's piped, you can't do that, because you don't know what you're writing to.

That's simple, which is nice, but it's also very limiting.

>> And the API just didn't feel very much like javascript

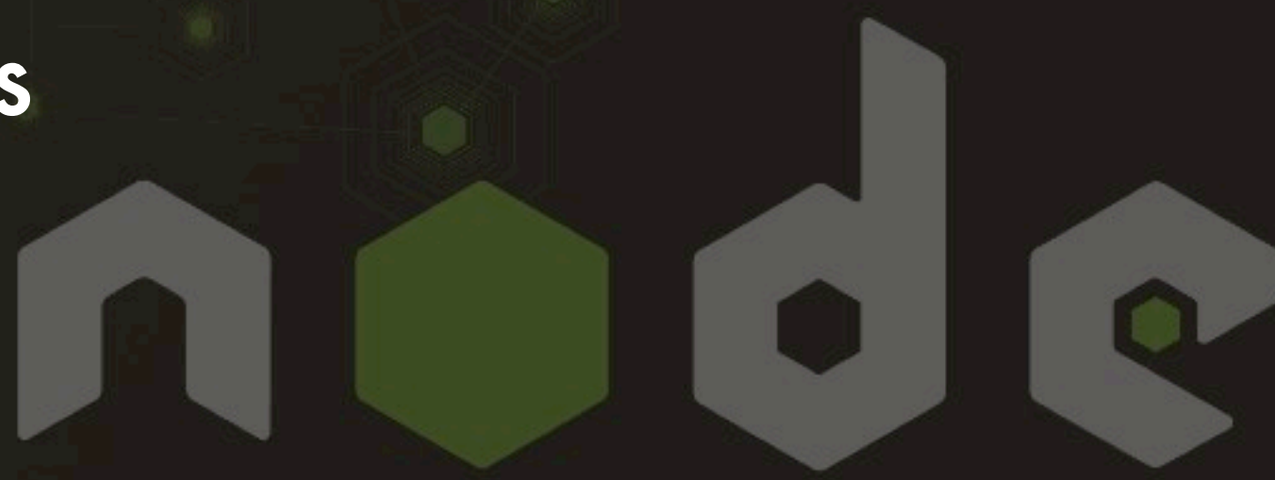
```
util.pump(a,b)
util.pump(b,c)
util.pump(c,d)
```

And the API just didn't feel very much like javascript

Very heavy. Not expressive.

streams1

- Readable
  r.pipe(w) // pipes to the writer
- Writable
  w.emit('pipe', r) // when piped into
- Stream base class



In node 0.4, we got a "Stream" base class. This provides a 'pipe' method.

pipe() does everything that util.pump() did, but in a better way.

It can be overridden in the Stream classes. Also, it emits 'pipe' on the destination, so you can have writers know what their reader is.

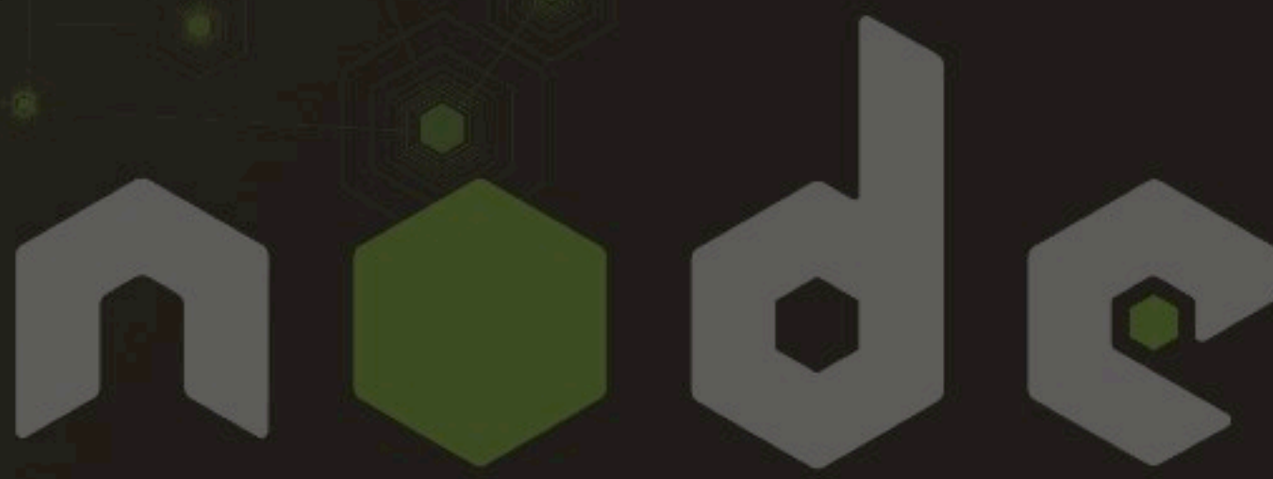
This allowed a >> much prettier api

```
a.pipe(b);
b.pipe(c);
c.pipe(d);
```

It's much more like JavaScript, much more expressive.

In v0.6, it got even better >> because pipe returns the destination

```
a.pipe(b)
 .pipe(c)
 .pipe(d)
;
```



In v0.6, it got a little nicer, because pipe() returns the destination, >>

so if you can chain them like this.

AND STREAMS WERE GOOD.

So, we've had these streams, and they're great

AND MANY PEOPLE TOLD YOU TO USE STREAMS

and a lot of people have been saying that you should write your libraries using streams,

and we've gone to conferences and given talks about how to go about doing that.

And you STREAMED ALL THE THINGS!

And it's great, because these lego blocks are fun.

They make programming simple. You don't have to think about as many things at once.

So now, we have all these streaming libraries.

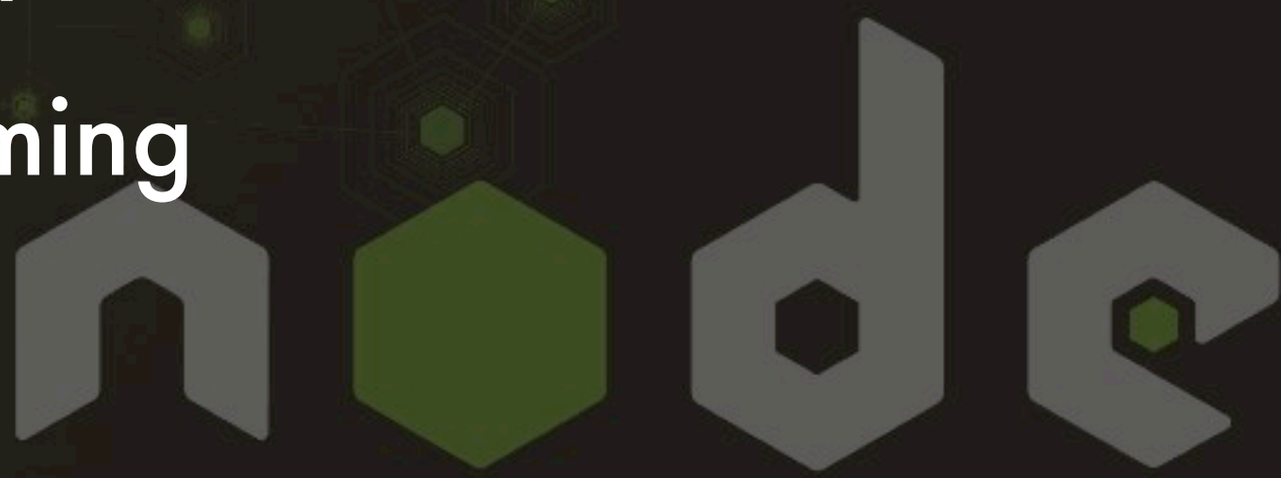
I've written some, and I'm here to tell you. It's way too difficult!

node streams are terrible

It really really sucks right now. It's terrible

stream badness

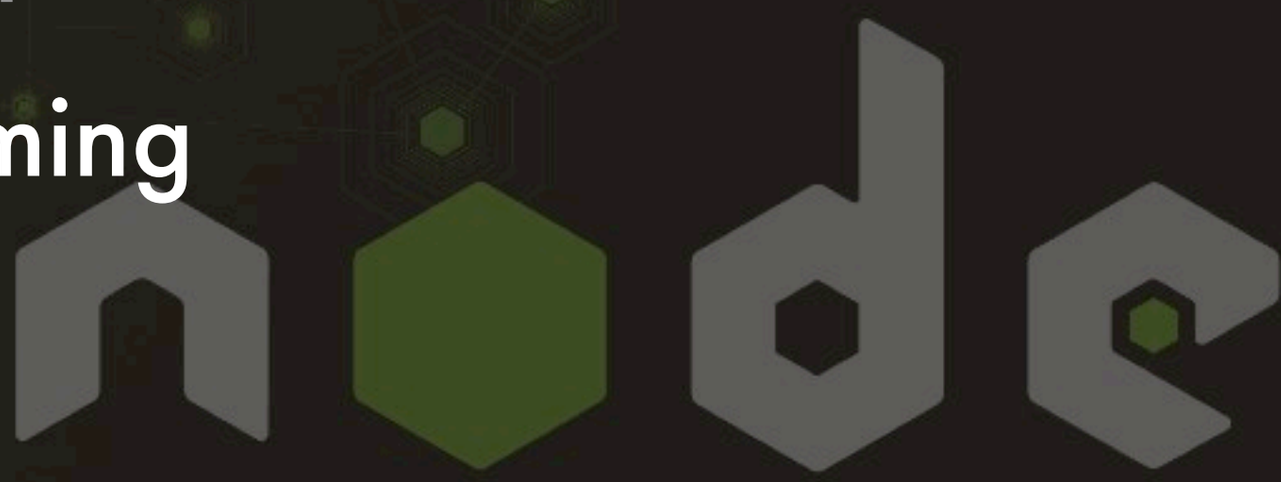
- immediate 'data' events
- pause() doesn't
- buffering is too hard to get right
- hyperactive backpressure
- crypto isn't streaming



There are a few things that make streams pretty rotten. I'm going to go through each of them.

stream badness

- immediate 'data' events
- pause() doesn't
- buffering is too hard to get right
- hyperactive backpressure
- crypto isn't streaming



So, first: the last one.

The crypto module doesn't use streams.

Actually, as of v0.8, it doesn't even use buffers properly.

That's a pretty obvious win, so >> we're changing that in v0.10.

v0.10: crypto streams

v(^_^)v

This was a bit tedious to get just right, but it makes the crypto library more useful.

immediate 'data'

- Surprise!

```
createServer(function(q,s) {
    // do some I/O
  session(q, function(ses) {
    // even nextTick is
    // too late!
    q.on('data', handler)
  })
})
```

How many of you have done this?

Data events can come right away, even on this current tick.

That means that if you have to hit a database to look up a session, and then decide what to do with the request, it's already too late.

That sucks. It means that you have to add the data handler onto the stream *before* you know what to do with the data, which usually means that you need to save up chunks.

So, you think, "I'll use pause()"? Nope.

pause() doesn't.

- Surprise!

```
createServer(function(q,s) {
    // ADVISORY only!
    q.pause()
    session(q, function(ses) {
      q.on('data', handler)
      q.resume()
    })
})
```

pause() will prevent the stream from reading any more from the file descriptor, but if anything is already in the queue, you'll get it right away.

So, everyone wrote a module to buffer data, and we found that >> buffering is difficult

buffering is hard

buffering is difficult.

Many bugs in tar and fstream were a result of buffering data while paused.

Occasionally things would just deadlock in some unusual state, and run out of memory.

Other times, it wouldn't try to read any more, and the program would just exit.

Very tricky, very annoying.

hyperactive backpressure

- write() returns true if data flushed
- fs.write() ALWAYS takes some time
- socket.pipe(file) pause();resume();pause();resume();...
- Particularly annoying for mobile

The write() method returns false if the data can't be flushed immediately, or true if it is all consumed.

But, file writes are an asynchronous operation; they *always* take some time to complete.

So, it always returns false, causing a pause/resume/pause/resume kind of dance.

This is unnecessary work.

In general, if you have a stream that can vary in speed, like a connection to a mobile client, you want to buffer a bit of data when it is blocked, but with a configurable limit.

With all these edges, and events, and buffering, and pausing, and resuming, and the fact that you have to build it all from scratch,

it means that it's just HARD to build streams and get it all exactly correct.

You need to do a lot of work, and it's not clear how to do it properly.

Streams are usually easy to use, as long as you don't dig into the details, but they're VERY hard to build without making mistakes.

BUT WAIT!

THERE'S MORE!

And if that was all, it'd be easy, because we can just fix those things.

But, we have many many modules on npm, and they all depend on streams behaving a certain way.

If we change it too drastically, then they'll break, and that just means that no one will use 0.10.

If no one uses the new version, then what's the use in fixing streams at all?

In other words,

backwards compatibility ruins everything.

But that's just not satisfying. We can't just say, "Oh well, I guess node is terrible. I guess I'll go do something else now."

At least, I'm not very happy with that. We're supposed to be better than that!

streams2

so that's where streams2 comes in. This is the new API which is coming very soon.

streams2

- "suck streams"
- Instead of 'data' events spewing, call read() to pull data from source
- Solves all problems (that we know of)

Dominic Tarr had a very insightful comment that what we're doing is "suck streams", where as what we had before were "spew streams".

Instead of data events spewing out at you, you call read() to pull some more data out of the stream.

If you don't read(), the data sits there waiting for you.

When the internal buffer is full, pushes back on the underlying system. There's no need for pause() or resume() methods.

streams2

- Readable
  r.read(size) → buffer or null
  r.emit('readable') → time to read()
- Easy configurable buffering:
  highWaterMark (default=1024)
  lowWaterMark (default=0)

When there isn't any data to consume, then read() will return null.

When it returns null, the "readable" event will let you know when there's more data for you to read().

To handle the hyperactive back pressure problem, we have a highWaterMark and lowWaterMark setting for each stream. These are limits that let you configure how different streams exert backpressure.

With this change, there is a nice symmetry between readable and writable streams

symmetry!

Readable	Writable
read() → null/buffer	write() → true/false
"readable" after null	"drain" after false
"end" event	end() method

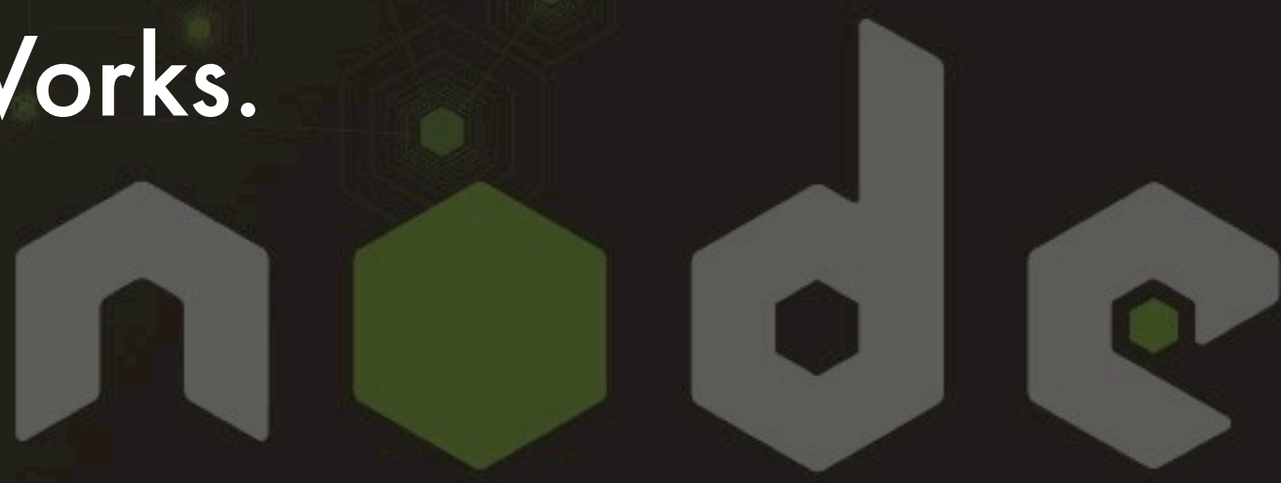
so, it's easy to see how readable and writable streams fit together.

The other problem was that you need deep understanding to implement your own streams correctly. There was a base Stream class, but it wasn't very useful.

That led to a lot of duplicated code, sometimes with slightly different behavior in different streams. This is not good.

extending

- inherit your stream class from Readable or Writable
- Override _read(size, callback) or _write(chunk, callback)
- All the rest Just Works.



With streams2, we have base classes for Readable and Writable streams.

If your code inherits from Readable, you implement the asynchronous _read(size,callback) method.

For Writable streams, implement the asynchronous _write(chunk, callback) method.

This makes it so that all the streams in node use the same implementation, and have the same behavior. Also, it's now trivial to implement your own streams.

Solves all problems!

- Data events are never lost
- No need for buffering
  just don't call read() until ready
- Easy extension points in base classes
- Low/high configs calm hyperactive backpressure

This solves every failure that was brought up. It might make some new ones, but we'll find that out later.

It's definitely an improvement, and it's one that Node needs

It's also very similar to the unix async behavior, and streaming interfaces in Dart and Java and Python, so that's another vote of confidence.

BUT WAIT!

THERE'S MORE!

Also, there are a lot of cases where we've got something that's a reader, and also a writer. It takes the written data, does something to it, and that produces some output.

For example, zlib, crypto, or computing hash digests.

We were all writing modules that do almost same exact thing, so why not make that easier?

Transform Class

- provide _transform(chunk, output, cb)
  (also _flush(output, cb) if relevant)
- call output(chunk) with output
- call cb() when done with that chunk
- Just Works.

The Transform class, so that you can override one method and get a thing that behaves appropriately.

It's pretty easy to use.

Zlib and crypto have already been ported to it.

consistency!

- All Readables have setEncoding()
- All Writables emit 'finish' when ended and flushed
- More overlapping of hidden classes, makes the magic V8 fairies happy

The idea of Streams is that we have a consistent API for dealing with data. With this refactor, it's much more consistent than before.

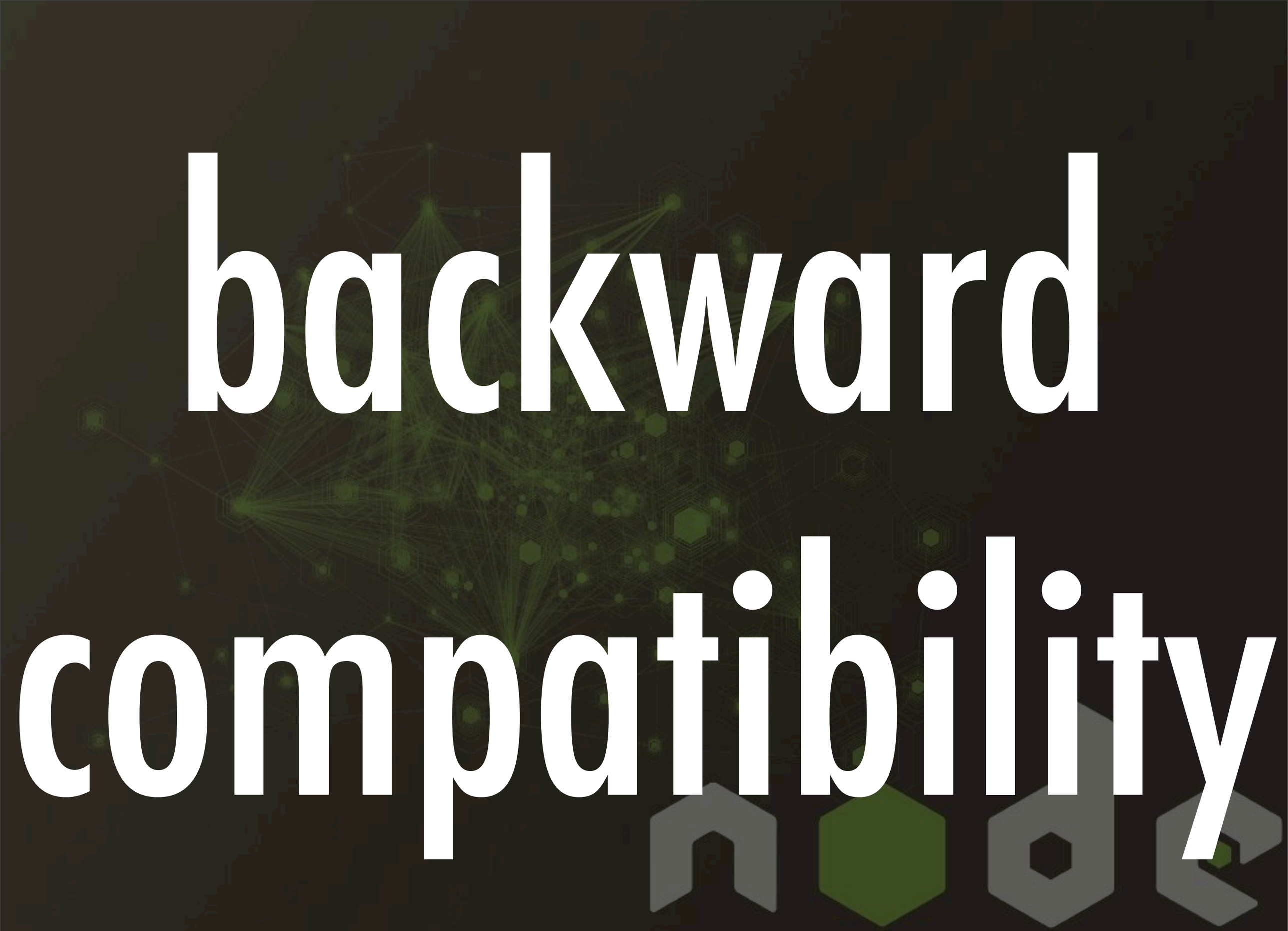
Now, things that used to be only on http streams, or only on fs streams, or implemented in many different places, are now blessed and official.

Every writable stream emits 'finish' when it's fully done.

Every Readable has a setEncoding() method, and so on.

There's more overlapping of the hidden classes, which V8 loves.

>> I know what you're thinking



I know what you're thinking,

But what about EVERY MODULE EVER?

what about all those many thousands of modules that use the old streams interface?

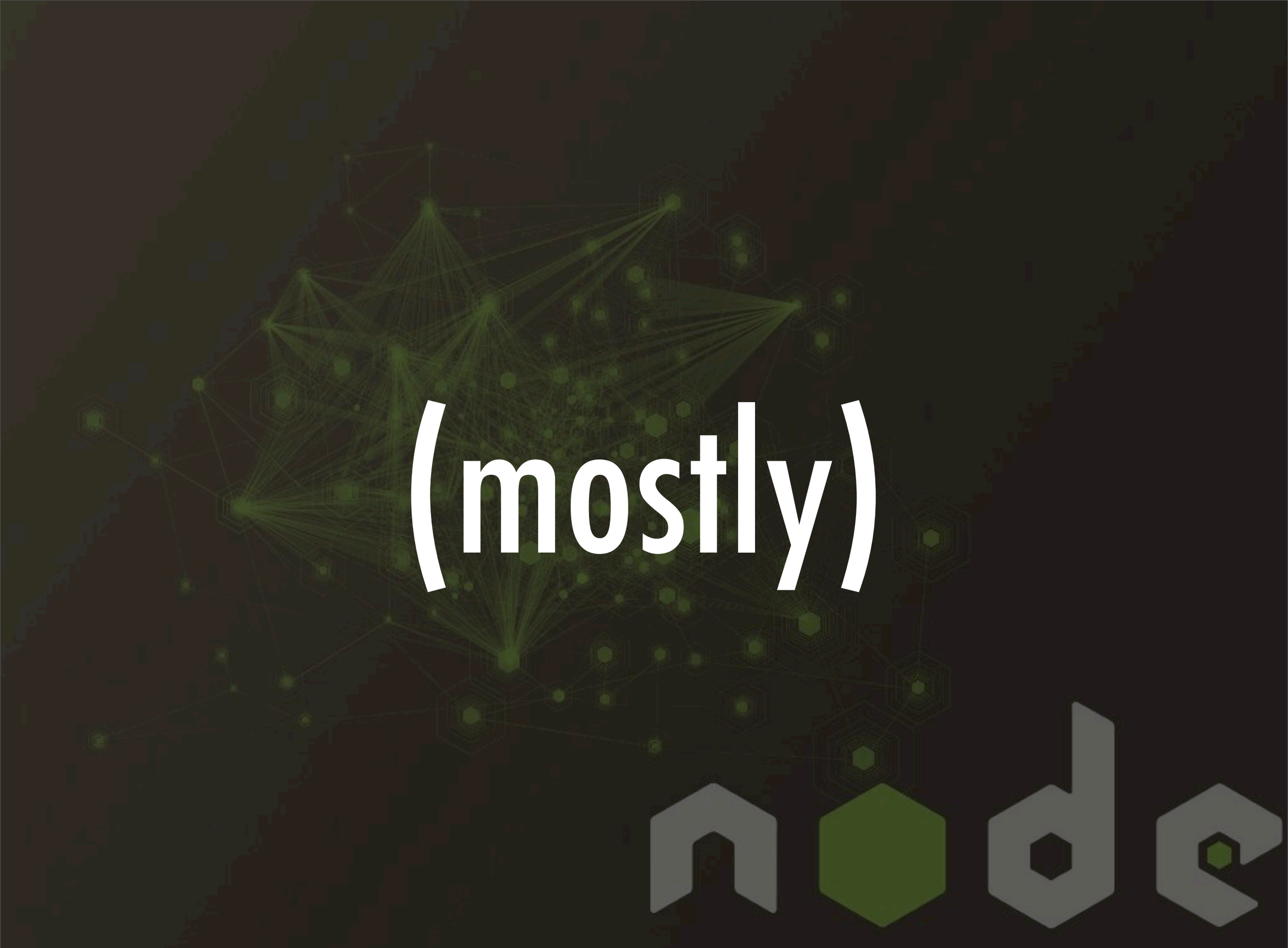
Won't they all BREAK?

won't they all break?

The answer is:

NO!

Nope. They won't all break with this change. >> mostly



Nope. They won't all break with this change. >> probably, mostly.

The way that node makes the old mode work is by detecting when you're using the old API style, and

>> shimming everything into the old mode

old-mode streams1 shim

- New streams can switch into old-mode, where they spew 'data'
- If you add a 'data' event handler, or call pause() or resume(), then switch
- Making minimal changes to existing tests to keep us honest

>> shimming everything into the old mode

When you make a new API, it is like you are discovering a new place.
But if there's no way to get to the new place, you leave your users behind.
Even if it's better, no one will use it, because it's too much work to get there.

If you add a data event handler, or if you call pause() or resume(), then we know that you're using the old API, and we present the old interface.

In order to verify this works, we're keeping all the old tests, and making sure that they all still pass.

There is one edge case that unfortunately is not fixable

Unavoidable Edge

- If you add a 'end' listener,
but **don't** add a 'data' listener,
and **don't** ever read() or pipe(),
it'll **never** emit 'end'

If you never cause it to switch into the old mode, then it won't know to do that

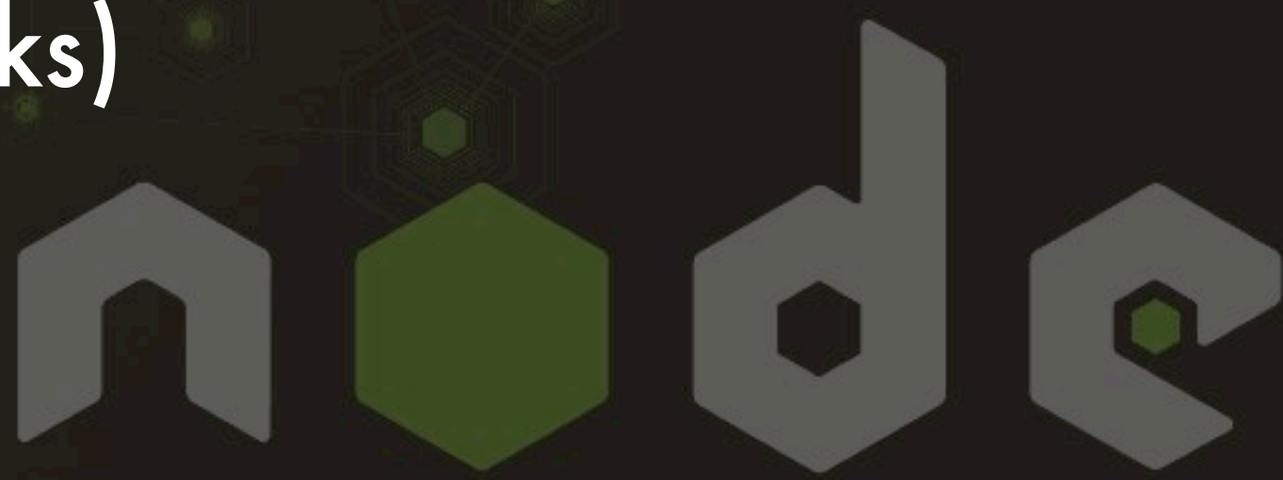
So, the stream will just sit there in a paused state forever. If you're waiting for the 'end' event, it'll never come.

This is usually only relevant in tests, but it is a semantic change.

Thankfully it's easy to avoid

Solution

- To trigger streams1 style behavior, without adding a 'data' listener, call stream.resume()
- Semantically, streams start out paused (and pause() works)



Call the resume() method, and it'll start flowing in old mode.

You can think about the stream as starting out paused.

We'll see how big of an issue this turns out to be.

This appears to be a very rare case where you care about the 'end' event, but you don't care about the data.

Current Status

- File system, tcp, tls, crypto, zlib, child_processes, and stdio all working
- http quite broken, last module to port
- Mild solvable performance issues
- v0.10 in December or January
- github.com/isaacs/readable-stream

Thursday, November 22, 12

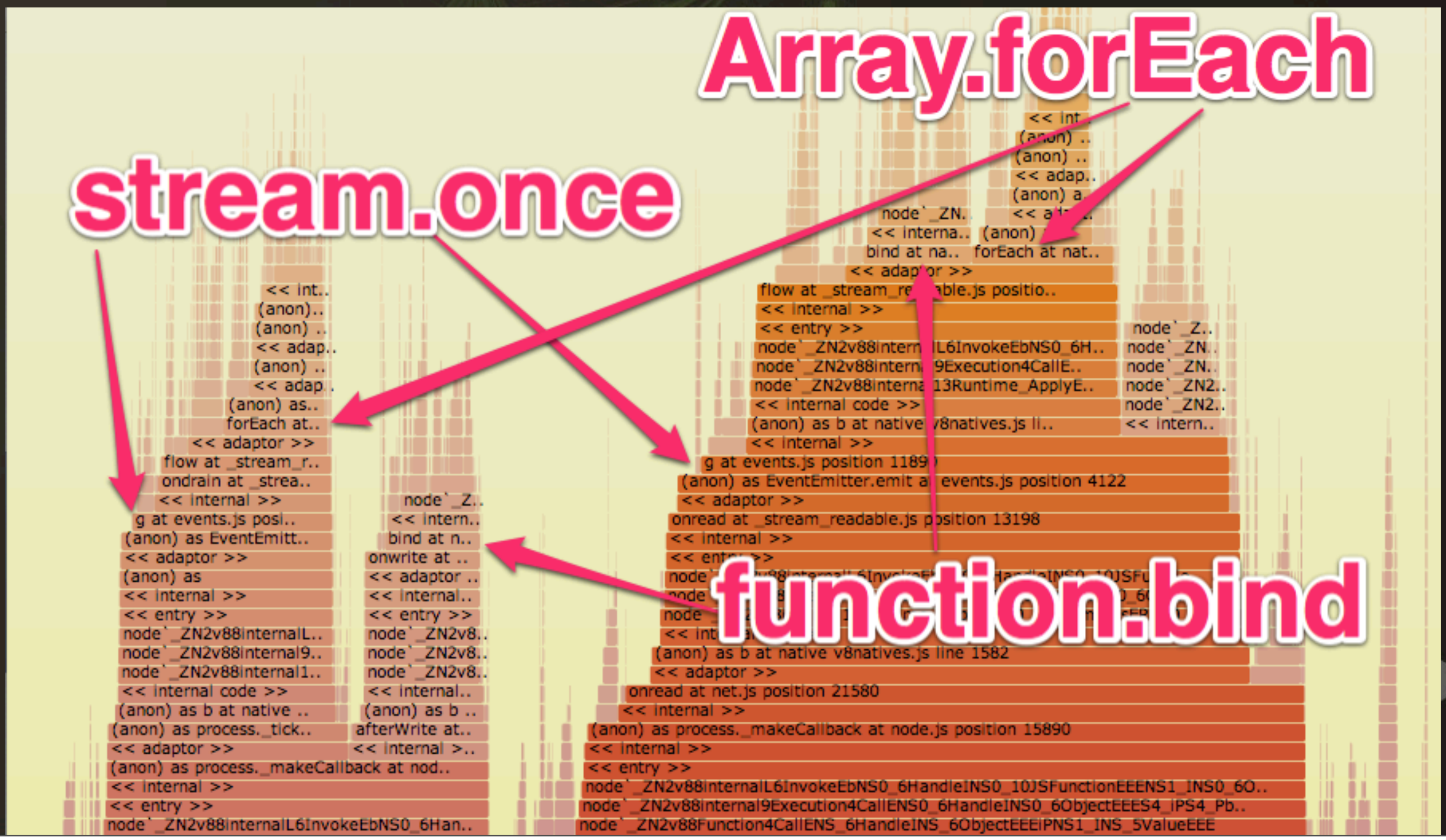
My goal was to get the new streams interface done in time for this trip, but it has turned out to be quite a lot of work. We're close.

The last module to be ported over is http, which is a big mess, because it's such an old part of node, and is very optimized.

I expect that we'll have v0.10 very soon. If not December, then very early in 2013.

There is about a 5% performance regression, but it's quite easy to track down using flame graphs

FLAME GRAPHS! \(^o^)/



If you have never seen these before, search the web for "node flame graph".

It's a great way to track down performance problems, using DTrace to see exactly what your program is spending its time doing.

In this case, there are a few functions that are a bit too slow to be used in such a sensitive feature. A 5% reduction in TCP speed is not at all acceptable.

I'm very confident that we can make it as good as the current node, or better.

isaacs.talk.

end('bye');

http://j.mp/streams2-ko
http://j.mp/streams2-ko-pdf

Thursday, November 22, 12