bmpvieira.com

#bionodehack

BIONODE.IO

Streams

Slides mostly based on: http://brycebaril.github.io/streams2-presentation

STREAMS

Streams are a first-class construct in Node.js for handling data.

STREAM COMPONENTS

There are essentially three major concepts with Streams:

- The Source
- The Pipeline
- The Sink

STREAM BENEFITS

- Lazily produce or consume data in buffered chunks.
- Evented and non-blocking
- Low memory footprint
- Automatically handle back-pressure
- Buffers allow you to work around the V8 heap memory limit
- Most core Node.js content sources/sinks are streams already!

STREAMS CLASSES

- Readable -- Data Sources
- Writable -- Data Sinks
- Duplex -- Both a Source and a Sink
- Transform -- In-flight stream operations
- Passthrough -- Stream spy

HOW TO IMPLEMENT

- Use handy abstractions like mississippi module (easy way)
- Subclass appropriate Stream Class and implement required methods, i.e., _read(), _write(), etc (hard way)

github.com/maxogden/mississippi

a collection of useful stream utility modules

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

- **from** - Make a custom readable stream
- **to** - Make a custom writable stream
- **through** - Make a custom transform stream.
- **duplex** - Take two separate streams, a writable and a readable, and turn them into a single duplex (readable and writable) stream.
- **pipeline** - Combine streams together

check github for code examples

HOW TO IMPLEMENT READABLE

Subclass stream.Readable

Implement a `_read(size)` method.

THE `_READ(SIZE)` METHOD:

`size` is in bytes, but can be ignored (especially for objectMode streams)

`_read(size)` must call this.push(chunk) to send a chunk to the consumer

READABLE OPTIONS

`highWaterMark` Number: The maximum number of bytes to store in the internal buffer before ceasing to read. Default: 16kb

`encoding` String: If set, buffers will be decoded to strings instead of passing buffers. Default: null

`objectMode` Boolean: Instead of using buffers/strings, use Javascript objects. Default: false

HOW TO USE A READABLE STREAM

use `readable.pipe(target)`

use `readable.read(size)`

`readable.on("data", /* ... */)`

```javascript
var Readable = require("stream").Readable
  || require("readable-stream/readable")
var inherits = require("util").inherits

function Source(options) {
  Readable.call(this, options)
  this.content = "The quick brown fox jumps over the
lazy dog."
}
inherits(Source, Readable)
Source.prototype._read = function (size) {
  if (!this.content) this.push(null)
  else {
    this.push(this.content.slice(0, size))
    this.content = this.content.slice(size)
  }
}
```

```
var s = new Source()
console.log(s.read(10).toString())
console.log(s.read(10).toString())
console.log(s.read(10).toString())
console.log(s.read(10).toString())
console.log(s.read(10).toString())

// The quick
// brown fox
// jumps over
//  the lazy
// dog.
```

WRITABLE

Use a Writable stream when collecting data from a stream.

Think: Drain/Collect.

WRITABLE OPTIONS

`highWaterMark` Number: The maximum number of bytes to store in the internal buffer before ceasing to read. Default: 16kb

`decodeStrings` Boolean: Whether to decode strings to Buffers before passing them to _write(). Default: true

HOW TO IMPLEMENT WRITABLE

Subclass stream.Writable

Implement a `_write(chunk, encoding, callback)` method.

THE _WRITE() METHOD

chunk is the content to write

Call callback() when you're done with this chunk

HOW TO USE A WRITABLE STREAM

```
source.pipe(writable)

writable.write(chunk [,encoding] [,callback])
```

A SIMPLE WRITABLE STREAM

```
var Writable = require("stream").Writable
  || require("readable-stream/writable")
var inherits = require("util").inherits

function Drain(options) {
  Writable.call(this, options)
}
inherits(Drain, Writable)
Drain.prototype._write = function (chunk, encoding,
callback) {
  console.log(chunk.toString())
  callback()
}
```

USING OUR EXAMPLES SO FAR:

```
var s = new Source()
var d = new Drain()
s.pipe(d)

// The quick brown fox jumps over the lazy dog.
```

DUPLEX

Use a Duplex stream when you accept input OR output, but as different streams. It is simply both a Readable and a Writable stream.

Think: Server

HOW TO IMPLEMENT DUPLEX

Subclass stream.Duplex

Implement a `_read(size)` method.

Implement a `_write(chunk, encoding, callback)` method.

DUPLEX OPTIONS

Superset of Readable and Writable options.

HOW TO USE A DUPLEX STREAM

```
input.pipe(duplex)

duplex.pipe(output)

duplex.on("data", /* ... */)

duplex.write()

duplex.read()
```

A SIMPLE DUPLEX STREAM

```
var Duplex = require("stream").Duplex
  || require("readable-stream/duplex")
var inherits = require("util").inherits

function Server(options) {
  Duplex.call(this, options)
  this.queue = []
}
inherits(Server, Duplex)
Server.prototype. read = function (size) {
  this.push(this.queue.shift())
}
Server.prototype. write = function (chunk, encoding, callback) {
  this.queue.push(Buffer.concat([new Buffer("REC: "), chunk, new
Buffer("\n")]))
  callback()
}
```

USING OUR EXAMPLE:

```
var s = new Server()
s.write("HI THERE")
s.write("HOW ARE YOU?")
s.pipe(process.stdout)

// REC: HI THERE
// REC: HOW ARE YOU?
```

TRANSFORM

Use a Transform stream when you want to operate on a stream in transit. This is a special kind of Duplex stream where the input and output stream are the same stream.

Think: Filter/Map

HOW TO IMPLEMENT TRANSFORM

Subclass stream.Transform

Implement a `_transform(chunk, encoding, callback)` method.

Optionally implement a `_flush(callback)` method.

THE `_TRANSFORM(CHUNK, ENCODING, CALLBACK)` METHOD:

Call `this.push(something)` to forward it to the next consumer.

You don't have to push anything, this will skip a chunk.

You *must* call `callback` one time per _transform call.

THE `_FLUSH(CALLBACK)` METHOD:

When the stream ends, this is your chance to do any cleanup or last-minute `this.push()` calls to clear any buffers or work. Call `callback()` when done.

TRANSFORM OPTIONS

Superset of Readable and Writable options.

HOW TO USE A TRANSFORM STREAM

```
source.pipe(transform).pipe(drain)

transform.on("data", /* ... */)
```

A SIMPLE TRANSFORM STREAM

```
var Transform = require("stream").Transform
  || require("readable-stream/transform")
var inherits = require("util").inherits

function ToUpper (options) {
  Transform.call(this, options)
}
inherits(ToUpper, Transform)
ToUpper.prototype._transform = function (chunk, encoding, cb) {
  var str = chunk.toString().toUpperCase()
  this.push(str)
  cb()
}
```

USING OUR EXAMPLE:

```
source.pipe(transform).pipe(drain)

transform.on("data", /* ... */)
```

```
var s = new Source()
var d = new Drain()
var tx = new ToUpper()

s.pipe(tx).pipe(d)

// THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
```

PASSTHROUGH

Most commonly Passthrough streams are used for testing. They are exactly a Transform stream that does no transformations.

Think: spy

HOW TO USE A PASSTHROUGH STREAM

Short answer: Don't

Use through2-spy instead.

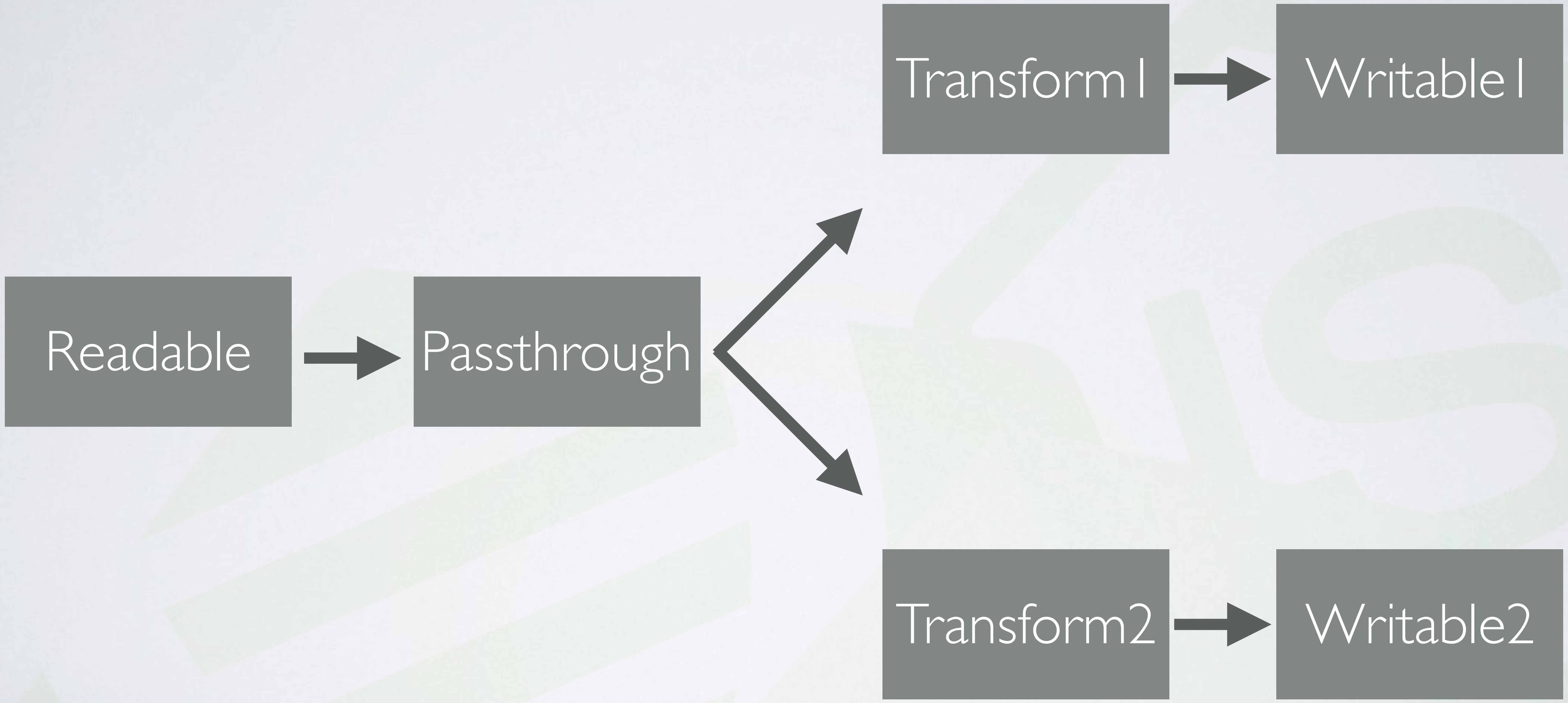
A THROUGH2-SPY

```
var spy = require("through2-spy")

var bytes = 0
// Spy on a pipeline, counting the number of bytes it has passed
var counter = spy(function (chunk) {bytes += chunk.length})

source.pipe(counter).pipe(drain)
```

FORKING STREAMS



FORKING STREAMS

```
readable
.pipe(passthrough)

passthrough
.pipe(transform1)
.pipe(writable1)

passthrough
.pipe(transform2)
.pipe(writable2)
```

BUFFERING

Streams handle buffering and backpressure automatically.

READABLE BUFFERING

Readable streams (Readable/Duplex/Transform) buffer when you call `this.push(chunk)` internally until the stream is read.

WRITABLE BUFFERING

Writable streams (Writable/Duplex/Transform) buffer when written to, draining as they are read or processed.

STREAM.READ(0)

You can trigger a refresh of the system without consuming any data by calling `.read(0)` on a readable stream. You probably won't need to do this.

STREAM.PUSH('') OR STREAM.PUSH(NULL)

Pushing a zero-byte string, or null for Object mode will terminate the pipeline.

ERRORS

Streams are EventEmitters, so they get traditional EventEmitter error handling.

I.e. Either add an 'error' listener to catch errors or let them bubble as exceptions.

PASSING ERRORS

Either Emit an 'error' event, or put an Error in the first argument of the `callback` in _write or _transform to signal an error and abort the stream.

Example:

stream-meter

ACKNOWLEDGMENTS

Organisers

DNA digest

Community

bionode.io

Sponsor

repositive.io

Research group

WurmLab .github.io

Queen Mary University of London

Venue

CAMPUS LONDON

Friends

dat DATA