minipass

A *very* minimal implementation of a PassThrough stream

It's very fast for objects, strings, and buffers.

Supports `pipe()`ing (including multi-`pipe()` and backpressure transmission), buffering data until either a `data` event handler or `pipe()` is added (so you don't lose the first chunk), and most other cases where PassThrough is a good idea.

There is a `read()` method, but it's much more efficient to consume data from this stream via `'data'` events or by calling `pipe()` into some other stream. Calling `read()` requires the buffer to be flattened in some cases, which requires copying memory.

If you set `objectMode: true` in the options, then whatever is written will be emitted. Otherwise, it'll do a minimal amount of Buffer copying to ensure proper Streams semantics when `read(n)` is called.

`objectMode` can only be set at instantiation. Attempting to write something other than a String or Buffer without having set `objectMode` in the options will throw an error.

This is not a `through` or `through2` stream. It doesn't transform the data, it just passes it right through. If you want to transform the data, extend the class, and override the `write()` method. Once you're done transforming the data however you want, call `super.write()` with the transform output.

For some examples of streams that extend Minipass in various ways, check out:

- minizlib
- fs-minipass
- tar
- minipass-collect
- minipass-flush
- minipass-pipeline
- tap
- tap-parser
- treport
- minipass-fetch
- pacote
- make-fetch-happen
- cacache
- ssri
- npm-registry-fetch
- minipass-json-stream
- minipass-sized

Usage in TypeScript

The `Minipass` class takes three type template definitions:

- `RType` the type being read, which defaults to `Buffer`. If `RType` is `string`, then the constructor *must* get an options object specifying either an `encoding` or `objectMode`:

`true`. If it's anything other than `string` or `Buffer`, then it *must* get an options object specifying `objectMode: true`.

- `WType` the type being written. If `RType` is `Buffer` or `string`, then this defaults to `ContiguousData` (Buffer, string, ArrayBuffer, or ArrayBufferView). Otherwise, it defaults to `RType`.
- `Events` type mapping event names to the arguments emitted with that event, which extends `Minipass.Events`.

To declare types for custom events in subclasses, extend the third parameter with your own event signatures. For example:

```
import { Minipass } from 'minipass'

// a NDJSON stream that emits
         'jsonError' when it can't
         stringify
export interface Events extends
         Minipass.Events {
  jsonError: [e: Error]
}

export class NDJSONStream extends
         Minipass<string, any, Events> {
  constructor() {
    super({ objectMode: true })
  }

  // data is type `any` because that's
         WType
  write(data, encoding, cb) {
    try {
      const json = JSON.stringify(data)
```

```
      return super.write(json + '\n',
        encoding, cb)
    } catch (er) {
      if (!er instanceof Error) {
        er = Object.assign(new
        Error('json stringify failed'),
        {
          cause: er,
        })
      }
      // trying to emit with something
        OTHER than an error will
      // fail, because we declared the
        event arguments type.
      this.emit('jsonError', er)
    }
  }
}

const s = new NDJSONStream()
s.on('jsonError', e => {
  // here, TS knows that e is an Error
})
```

Emitting/handling events that aren't declared in this way is fine, but the arguments will be typed as `unknown`.


```
      } else if (Buffer.isBuffer(chunk)) {
        chunk = chunk.toString()
      }
      if (typeof encoding === 'function')
          {
        cb = encoding
      }
      const jsonData = (this._jsonBuffer
          + chunk).split('\n')
      this._jsonBuffer = jsonData.pop()
      for (let i = 0; i <
          jsonData.length; i++) {
        try {
          // JSON.parse can throw, emit an
          error on that
          super.write(JSON.parse(jsonData[i]))
        } catch (er) {
          this.emit('error', er)
          continue
        }
      }
      if (cb) cb()
    }
  }
```

```
      this.emit('error', er)
    }
  }
  end(obj, cb) {
    if (typeof obj === 'function') {
      cb = obj
      obj = undefined
    }
    if (obj !== undefined) {
      this.write(obj)
    }
    return super.end(cb)
  }
}
```

transform that parses newline-delimited JSON

```
class NDJSONDecode extends Minipass {
  constructor(options) {
    // always be in object mode, as far
         as Minipass is concerned
    super({ objectMode: true })
    this._jsonBuffer = ''
  }
  write(chunk, encoding, cb) {
    if (
      typeof chunk === 'string' &&
      typeof encoding === 'string' &&
      encoding !== 'utf8'
    ) {
      chunk = Buffer.from(chunk,
        encoding).toString()
```


Differences from Node.js Streams

There are several things that make Minipass streams different from (and in some ways superior to) Node.js core streams.

Please read these caveats if you are familiar with node-core streams and intend to use Minipass streams in your programs.

You can avoid most of these differences entirely (for a very small performance penalty) by setting `{async: true}` in the constructor options.

Timing

Minipass streams are designed to support synchronous use-cases. Thus, data is emitted as soon as it is available, always. It is buffered until read, but no longer. Another way to look at it is that Minipass streams are exactly as synchronous as the logic that writes into them.

This can be surprising if your code relies on `PassThrough.write()` always providing data on the next tick rather than the current one, or being able to call `resume()` and not have the entire buffer disappear immediately.

However, without this synchronicity guarantee, there would be no way for Minipass to achieve the speeds it does, or support the synchronous use cases that it does. Simply put, waiting takes time.

This non-deferring approach makes Minipass streams much easier to reason about, especially in the context of Promises and other flow-control mechanisms.

Example:

```
// hybrid module, either works
import { Minipass } from 'minipass'
// or:
const { Minipass } = require('minipass')

const stream = new Minipass()
stream.on('data', () =>
        console.log('data event'))
console.log('before write')
stream.write('hello')
console.log('after write')
// output:
// before write
// data event
// after write
```

Exception: Async Opt-In

If you wish to have a Minipass stream with behavior that more closely mimics Node.js core streams, you can set the stream in async mode either by setting `async: true` in the constructor options, or by setting `stream.async = true` later on.

```
// hybrid module, either works
import { Minipass } from 'minipass'
// or:
const { Minipass } = require('minipass')
```


subclass that defers 'end' for some reason

```
class SlowEnd extends Minipass {
  emit(ev, ...args) {
    if (ev === 'end') {

        console.log('going to end, hold
        on a sec')
      setTimeout(() => {
        console.log('ok, ready to end
        now')
        super.emit('end', ...args)
      }, 100)
      return true
    } else {
      return super.emit(ev, ...args)
    }
  }
}
```

transform that creates newline-delimited JSON

```
class NDJSONEncode extends Minipass {
  write(obj, cb) {
    try {
      // JSON.stringify can throw, emit
        an error on that
      return
        super.write(JSON.stringify(obj)
        + '\n', 'utf8', cb)
    } catch (er) {
```

same thing, but using an inline anonymous class

```
// js classes are fun
someSource
  .pipe(
    new (class extends Minipass {
      emit(ev, ...data) {
        // let's also log events,
        because debugging some weird
        thing
        console.log('EMIT', ev)
        return super.emit(ev, ...data)
      }
      write(chunk, encoding, callback) {
        console.log('WRITE', chunk,
        encoding)
        return super.write(chunk,
        encoding, callback)
      }
      end(chunk, encoding, callback) {
        console.log('END', chunk,
        encoding)
        return super.end(chunk,
        encoding, callback)
      }
    })()
  )
  .pipe(someDest)
```



```
const asyncStream = new Minipass({
        async: true })
asyncStream.on('data', () =>
        console.log('data event'))
console.log('before write')
asyncStream.write('hello')
console.log('after write')
// output:
// before write
// after write
// data event <-- this is deferred until
        the next tick
```

Switching *out* of async mode is unsafe, as it could cause data corruption, and so is not enabled. Example:

```
import { Minipass } from 'minipass'
const stream = new Minipass({ encoding:
        'utf8' })
stream.on('data', chunk =>
        console.log(chunk))
stream.async = true
console.log('before writes')
stream.write('hello')
setStreamSyncAgainSomehow(stream) // <--
        this doesn't actually exist!
stream.write('world')
console.log('after writes')
// hypothetical output would be:
// before writes
// world
// after writes
// hello
// NOT GOOD!
```

To avoid this problem, once set into async mode, any attempt to make the stream sync again will be ignored.

```
const { Minipass } = require('minipass')
const stream = new Minipass({ encoding:
        'utf8' })
stream.on('data', chunk =>
        console.log(chunk))
stream.async = true
console.log('before writes')
stream.write('hello')
stream.async = false // <-- no-op,
        stream already async
stream.write('world')
console.log('after writes')
// actual output:
// before writes
// after writes
// hello
// world
```

No High/Low Water Marks

Node.js core streams will optimistically fill up a buffer, returning `true` on all writes until the limit is hit, even if the data has nowhere to go. Then, they will not attempt to draw more data in until the buffer size dips below a minimum value.

Minipass streams are much simpler. The `write()` method will return `true` if the data has somewhere to go (which is to say,


```
  }
  return 'ok'
}

consume().then(res => console.log(res))
// logs `foo\n` 5 times, and then `ok`
```

subclass that `console.log()`s everything written into it

```
class Logger extends Minipass {
  write(chunk, encoding, callback) {
    console.log('WRITE', chunk,
        encoding)
    return super.write(chunk, encoding,
        callback)
  }
  end(chunk, encoding, callback) {
    console.log('END', chunk, encoding)
    return super.end(chunk, encoding,
        callback)
  }
}

someSource.pipe(new
        Logger()).pipe(someDest)
```

```
  console.log(letter) // c, d
}
mp.write('e')
mp.end()
for (let letter of mp) {
  console.log(letter) // e
}
for (let letter of mp) {
  console.log(letter) // nothing
}
```

Asynchronous iteration will continue until the end event is reached, consuming all of the data.

```
const mp = new Minipass({ encoding:
        'utf8' })

// some source of some data
let i = 5
const inter = setInterval(() => {
  if (i-- > 0)
        mp.write(Buffer.from('foo\n',
        'utf8'))
  else {
    mp.end()
    clearInterval(inter)
  }
}, 100)

// consume the data with asynchronous
        iteration
async function consume() {
  for await (let chunk of mp) {
    console.log(chunk)
```


given the timing guarantees, that the data is already there by the time `write()` returns).

If the data has nowhere to go, then `write()` returns false, and the data sits in a buffer, to be drained out immediately as soon as anyone consumes it.

Since nothing is ever buffered unnecessarily, there is much less copying data, and less bookkeeping about buffer capacity levels.

Hazards of Buffering (or: Why Minipass Is So Fast)

Since data written to a Minipass stream is immediately written all the way through the pipeline, and `write()` always returns true/false based on whether the data was fully flushed, backpressure is communicated immediately to the upstream caller. This minimizes buffering.

Consider this case:

```
const { PassThrough } =
        require('stream')
const p1 = new PassThrough({
        highWaterMark: 1024 })
const p2 = new PassThrough({
        highWaterMark: 1024 })
const p3 = new PassThrough({
        highWaterMark: 1024 })
const p4 = new PassThrough({
        highWaterMark: 1024 })
```

```
p1.pipe(p2).pipe(p3).pipe(p4)
p4.on('data', () =>
        console.log('made it through'))

// this returns false and buffers, then
        writes to p2 on next tick (1)
// p2 returns false and buffers, pausing
        p1, then writes to p3 on next
        tick (2)
// p3 returns false and buffers, pausing
        p2, then writes to p4 on next
        tick (3)
// p4 returns false and buffers, pausing
        p3, then emits 'data' and
        'drain'
// on next tick (4)
// p3 sees p4's 'drain' event, and calls
        resume(), emitting 'resume' and
// 'drain' on next tick (5)
// p2 sees p3's 'drain', calls resume(),
        emits 'resume' and 'drain' on
        next tick (6)
// p1 sees p2's 'drain', calls resume(),
        emits 'resume' and 'drain' on
        next
// tick (7)

p1.write(Buffer.alloc(2048)) // returns
        false
```



```
mp.concat().then(onebigchunk => {
  // onebigchunk is a string if the
        stream
  // had an encoding set, or a buffer
        otherwise.
})
```

iteration

You can iterate over streams synchronously or asynchronously in platforms that support it.

Synchronous iteration will end when the currently available data is consumed, even if the `end` event has not been reached. In string and buffer mode, the data is concatenated, so unless multiple writes are occurring in the same tick as the `read()`, sync iteration loops will generally only have a single iteration.

To consume chunks in this way exactly as they have been written, with no flattening, create the stream with the `{ objectMode: true }` option.

```
const mp = new Minipass({ objectMode:
        true })
mp.write('a')
mp.write('b')
for (let letter of mp) {
  console.log(letter) // a, b
}
mp.write('c')
mp.write('d')
for (let letter of mp) {
```

```
  },
  er => {
    // stream emitted an error
  }
)
```

collecting

```
mp.collect().then(all => {
  // all is an array of all the data
       emitted
  // encoding is supported in this case,
       so
  // so the result will be a collection
       of strings if
  // an encoding is specified, or
       buffers/objects if not.
  //
  // In an async function, you may do
  // const data = await stream.collect()
})
```

collecting into a single blob

This is a bit slower because it concatenates the data into one chunk for you, but if you're going to do it yourself anyway, it's convenient this way:

Along the way, the data was buffered and deferred at each stage, and multiple event deferrals happened, for an unblocked pipeline where it was perfectly safe to write all the way through!

Furthermore, setting a `highWaterMark` of `1024` might lead someone reading the code to think an advisory maximum of 1KiB is being set for the pipeline. However, the actual advisory buffering level is the *sum* of `highWaterMark` values, since each one has its own bucket.

Consider the Minipass case:

```
const m1 = new Minipass()
const m2 = new Minipass()
const m3 = new Minipass()
const m4 = new Minipass()

m1.pipe(m2).pipe(m3).pipe(m4)
m4.on('data', () =>
        console.log('made it through'))

// m1 is flowing, so it writes the data
       to m2 immediately
// m2 is flowing, so it writes the data
       to m3 immediately
// m3 is flowing, so it writes the data
       to m4 immediately
// m4 is flowing, so it fires the 'data'
       event immediately, returns true
// m4's write returned true, so m3 is
       still flowing, returns true
// m3's write returned true, so m2 is
       still flowing, returns true
```

```
// m2's write returned true, so m1 is
        still flowing, returns true
// No event deferrals or buffering along
        the way!

m1.write(Buffer.alloc(2048)) // returns
        true
```

It is extremely unlikely that you *don't* want to buffer any data written, or *ever* buffer data that can be flushed all the way through. Neither node-core streams nor Minipass ever fail to buffer written data, but node-core streams do a lot of unnecessary buffering and pausing.

As always, the faster implementation is the one that does less stuff and waits less time to do it.

Immediately emit `end` for empty streams (when not paused)

If a stream is not paused, and `end()` is called before writing any data into it, then it will emit `end` immediately.

If you have logic that occurs on the `end` event which you don't want to potentially happen immediately (for example, closing file descriptors, moving on to the next entry in an archive parse stream, etc.) then be sure to call `stream.pause()` on creation, and then `stream.resume()` once you are ready to respond to the `end` event.

However, this is *usually* not a problem because:

- `readable` Emitted when data is buffered and ready to be read by a consumer.
- `resume` Emitted when stream changes state from buffering to flowing mode. (Ie, when `resume` is called, `pipe` is called, or a `data` event listener is added.)

Static Methods

- `Minipass.isStream(stream)` Returns `true` if the argument is a stream, and false otherwise. To be considered a stream, the object must be either an instance of Minipass, or an EventEmitter that has either a `pipe()` method, or both `write()` and `end()` methods. (Pretty much any stream in node-land will return `true` for this.)

EXAMPLES

Here are some examples of things you can do with Minipass streams.

simple "are you done yet" promise

```
mp.promise().then(
  () => {
    // stream is finished
```

- `objectMode` Indicates whether the stream is in `objectMode`.
- `aborted` Readonly property set when the `AbortSignal` dispatches an `abort` event.

Events

- `data` Emitted when there's data to read. Argument is the data to read. This is never emitted while not flowing. If a listener is attached, that will resume the stream.
- `end` Emitted when there's no more data to read. This will be emitted immediately for empty streams when `end()` is called. If a listener is attached, and `end` was already emitted, then it will be emitted again. All listeners are removed when `end` is emitted.
- `prefinish` An end-ish event that follows the same logic as `end` and is emitted in the same conditions where `end` is emitted. Emitted after `'end'`.
- `finish` An end-ish event that follows the same logic as `end` and is emitted in the same conditions where `end` is emitted. Emitted after `'prefinish'`.
- `close` An indication that an underlying resource has been released. Minipass does not emit this event, but will defer it until after `end` has been emitted, since it throws off some stream libraries otherwise.
- `drain` Emitted when the internal buffer empties, and it is again suitable to `write()` into the stream.

Emit `end` When Asked

One hazard of immediately emitting `'end'` is that you may not yet have had a chance to add a listener. In order to avoid this hazard, Minipass streams safely re-emit the `'end'` event if a new listener is added after `'end'` has been emitted.

Ie, if you do `stream.on('end', someFunction)`, and the stream has already emitted `end`, then it will call the handler right away. (You can think of this somewhat like attaching a new `.then(fn)` to a previously-resolved Promise.)

To prevent calling handlers multiple times who would not expect multiple ends to occur, all listeners are removed from the `'end'` event whenever it is emitted.

Emit `error` When Asked

The most recent error object passed to the `'error'` event is stored on the stream. If a new `'error'` event handler is added, and an error was previously emitted, then the event handler will be called immediately (or on `process.nextTick` in the case of async streams).

This makes it much more difficult to end up trying to interact with a broken stream, if the error handler is added after an error was previously emitted.

Impact of “immediate flow” on Tee-streams

A “tee stream” is a stream piping to multiple destinations:

```
const tee = new Minipass()
t.pipe(dest1)
t.pipe(dest2)
t.write('foo') // goes to both
        destinations
```

Since Minipass streams *immediately* process any pending data through the pipeline when a new pipe destination is added, this can have surprising effects, especially when a stream comes in from some other function and may or may not have data in its buffer.

```
// WARNING! WILL LOSE DATA!
const src = new Minipass()
src.write('foo')
src.pipe(dest1) // 'foo' chunk flows to
        dest1 immediately, and is gone
src.pipe(dest2) // gets nothing!
```

One solution is to create a dedicated tee-stream junction that pipes to both locations, and then pipe to *that* instead.

```
// Safe example: tee to both places
const src = new Minipass()
src.write('foo')
const tee = new Minipass()
tee.pipe(dest1)
```


`close()` method, and has not emitted a `'close'` event yet, then `stream.close()` will be called. Any Promises returned by `.promise()`, `.collect()` or `.concat()` will be rejected. After being destroyed, writing to the stream will emit an error. No more data will be emitted if the stream is destroyed, even if it was previously buffered.

Properties

- `bufferLength` Read-only. Total number of bytes buffered, or in the case of objectMode, the total number of objects.
- `encoding` Read-only. The encoding that has been set.
- `flowing` Read-only. Boolean indicating whether a chunk written to the stream will be immediately emitted.
- `emittedEnd` Read-only. Boolean indicating whether the end-ish events (ie, `end`, `prefinish`, `finish`) have been emitted. Note that listening on any end-ish event will immediateyl re-emit it if it has already been emitted.
- `writable` Whether the stream is writable. Default `true`. Set to `false` when `end()`
- `readable` Whether the stream is readable. Default `true`.
- `pipes` An array of Pipe objects referencing streams that this stream is piping into.
- `destroyed` A getter that indicates whether the stream was destroyed.
- `paused` True if the stream has been explicitly paused, otherwise false.

- `unpipe(dest)` - Stop piping to the destination stream. This is immediate, meaning that any asynchronously queued data will *not* make it to the destination when running in `async` mode.
  - `options.end` - Boolean, end the destination stream when the source stream ends. Default `true`.
  - `options.proxyErrors` - Boolean, proxy `error` events from the source stream to the destination stream. Note that errors are *not* proxied after the pipeline terminates, either due to the source emitting `'end'` or manually unpiping with `src.unpipe(dest)`. Default `false`.
- `on(ev, fn)`, `emit(ev, fn)` - Minipass streams are EventEmitters. Some events are given special treatment, however. (See below under "events".)
- `promise()` - Returns a Promise that resolves when the stream emits `end`, or rejects if the stream emits `error`.
- `collect()` - Return a Promise that resolves on `end` with an array containing each chunk of data that was emitted, or rejects if the stream emits `error`. Note that this consumes the stream data.
- `concat()` - Same as `collect()`, but concatenates the data into a single Buffer object. Will reject the returned promise if the stream is in objectMode, or if it goes into objectMode by the end of the data.
- `read(n)` - Consume `n` bytes of data out of the buffer. If `n` is not provided, then consume all of it. If `n` bytes are not available, then it returns null. **Note** consuming streams in this way is less efficient, and can lead to unnecessary Buffer copying.
- `destroy([er])` - Destroy the stream. If an error is provided, then an `'error'` event is emitted. If the stream has a


```
tee.pipe(dest2)
src.pipe(tee) // tee gets 'foo', pipes
        to both locations
```

The same caveat applies to `on('data')` event listeners. The first one added will *immediately* receive all of the data, leaving nothing for the second:

```
// WARNING! WILL LOSE DATA!
const src = new Minipass()
src.write('foo')
src.on('data', handler1) // receives
        'foo' right away
src.on('data', handler2) // nothing to
        see here!
```

Using a dedicated tee-stream can be used in this case as well:

```
// Safe example: tee to both data
        handlers
const src = new Minipass()
src.write('foo')
const tee = new Minipass()
tee.on('data', handler1)
tee.on('data', handler2)
src.pipe(tee)
```

All of the hazards in this section are avoided by setting `{ async: true }` in the Minipass constructor, or by setting `stream.async = true` afterwards. Note that this does add some overhead, so should only be done in cases where you are willing to lose a bit of performance in order to avoid having to refactor program logic.

USAGE

It's a stream! Use it like a stream and it'll most likely do what you want.

```
import { Minipass } from 'minipass'
const mp = new Minipass(options) //
        options is optional
mp.write('foo')
mp.pipe(someOtherStream)
mp.end('bar')
```

OPTIONS

- `encoding` How would you like the data coming *out* of the stream to be encoded? Accepts any values that can be passed to `Buffer.toString()`.
- `objectMode` Emit data exactly as it comes in. This will be flipped on by default if you write() something other than a string or Buffer at any point. Setting `objectMode: true` will prevent setting any encoding value.
- `async` Defaults to `false`. Set to `true` to defer data emission until next tick. This reduces performance slightly, but makes Minipass streams use timing behavior closer to Node core streams. See Timing for more details.
- `signal` An `AbortSignal` that will cause the stream to unhook itself from everything and become as inert as possible. Note that providing a `signal` parameter will make `'error'`

events no longer throw if they are unhandled, but they will still be emitted to handlers if any are attached.

API

Implements the user-facing portions of Node.js's `Readable` and `Writable` streams.

Methods

- `write(chunk, [encoding], [callback])` - Put data in. (Note that, in the base Minipass class, the same data will come out.) Returns `false` if the stream will buffer the next write, or true if it's still in "flowing" mode.
- `end([chunk, [encoding]], [callback])` - Signal that you have no more data to write. This will queue an `end` event to be fired when all the data has been consumed.
- `pause()` - No more data for a while, please. This also prevents `end` from being emitted for empty streams until the stream is resumed.
- `resume()` - Resume the stream. If there's data in the buffer, it is all discarded. Any buffered events are immediately emitted.
- `pipe(dest)` - Send all output to the stream provided. When data is emitted, it is immediately written to any and all pipe destinations. (Or written on next tick in `async` mode.)