你不懂流就别说会Node.js!

前提

本文较深入的探讨了 Node.js Stream（流） 原理，对于 Stream 存在的意义和基础用法，不做讨论。

前言

Stream 可以说是 node.js，最最最重要的api之一，理解并熟练使用它，对于node.js开发来说是必必必须的。但是网上很多文章都存在一些问题（后面会列举）。

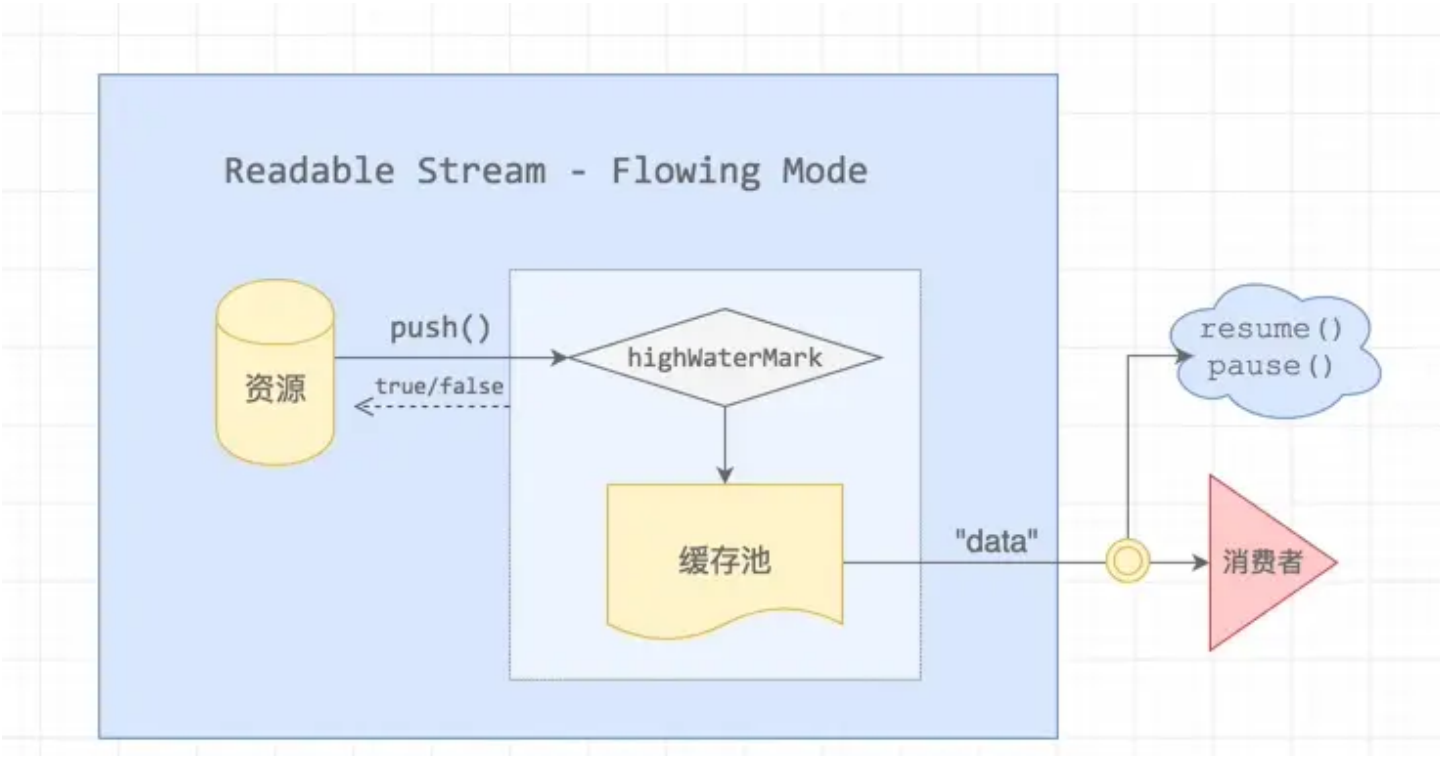
最终目的，文章会得出一些简单的结论。在常规情况下，这些结论会帮助我们可以放心大胆的使用可读流，可写流和 Transform 流，为你在生产环境对于流的处理增加信心。后面会有node.js的 js 部分源码调试过程来证明，但是由于极其枯燥，所以开头写结论，最后写过程，方便想深入的同学!

这是我总结的 Node.js 系列文章的第 5 篇。欢迎加入 node.js 技术讨论群，目前和我的前端组件库讨论群是在一起的。个人微信：a2298613245。

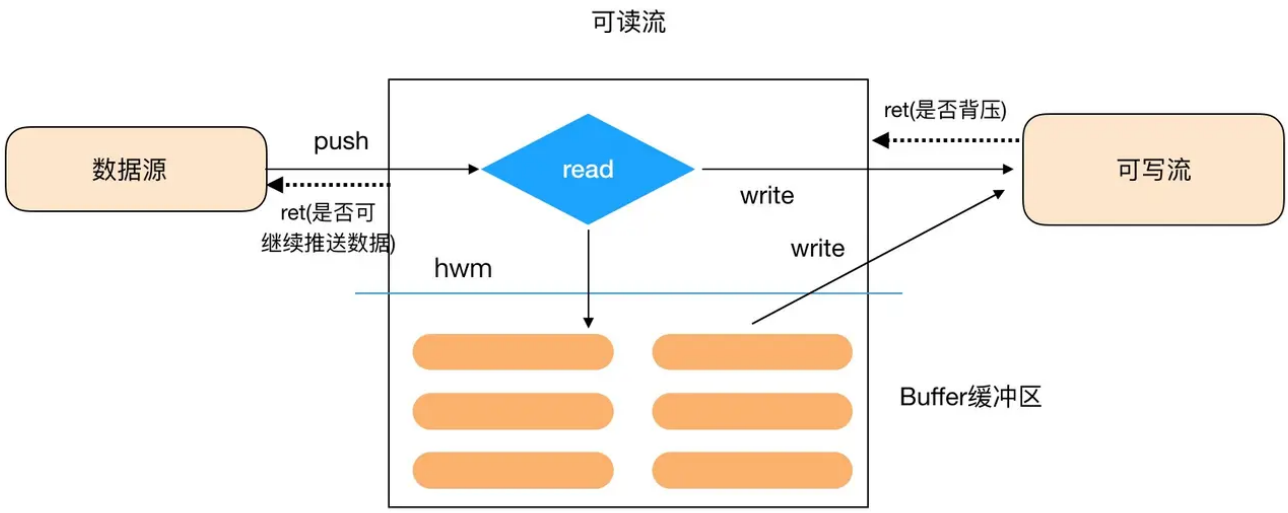
- 基础知识
  - Node.js 架构知识
  - 关于 V8 引擎你必须知道的基础知识
- 异步管理
  - 你可能没有听说过的一种异步管理模式 - Generator 异步管理器
  - 对不起，你之前学的 promise 可能是错的！（从 ecma 标准看 promise）

废话不多说，开车!

- 可读流在调用自定义可读流 this.push() 方法时，一定会把数据会放入缓存区吗？毕竟很多网络上的关于可写流的图都是如下类似：

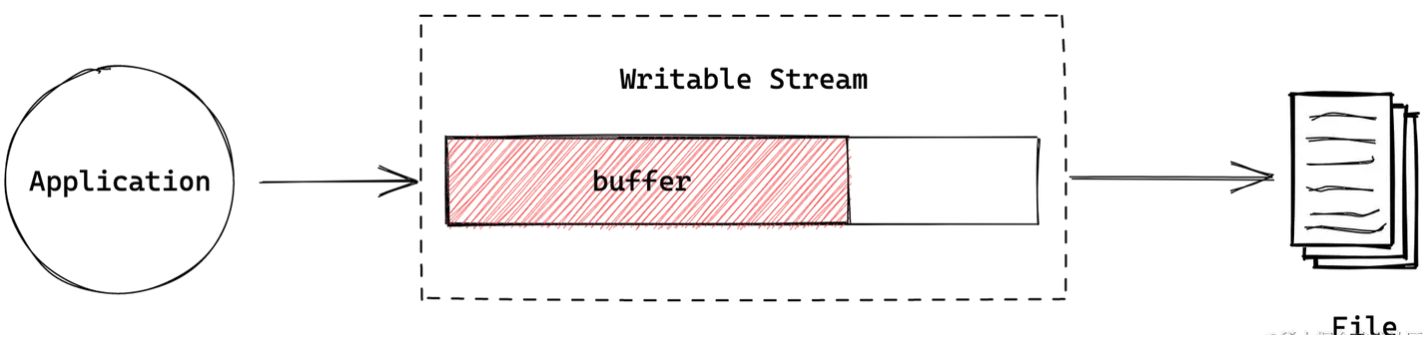


注意上图，可读流使用 push 方法是一定将输入放入缓存池（Buffer）里，可这对于你理解 stream 原理是很大的障碍，因为正确的图应该如下:

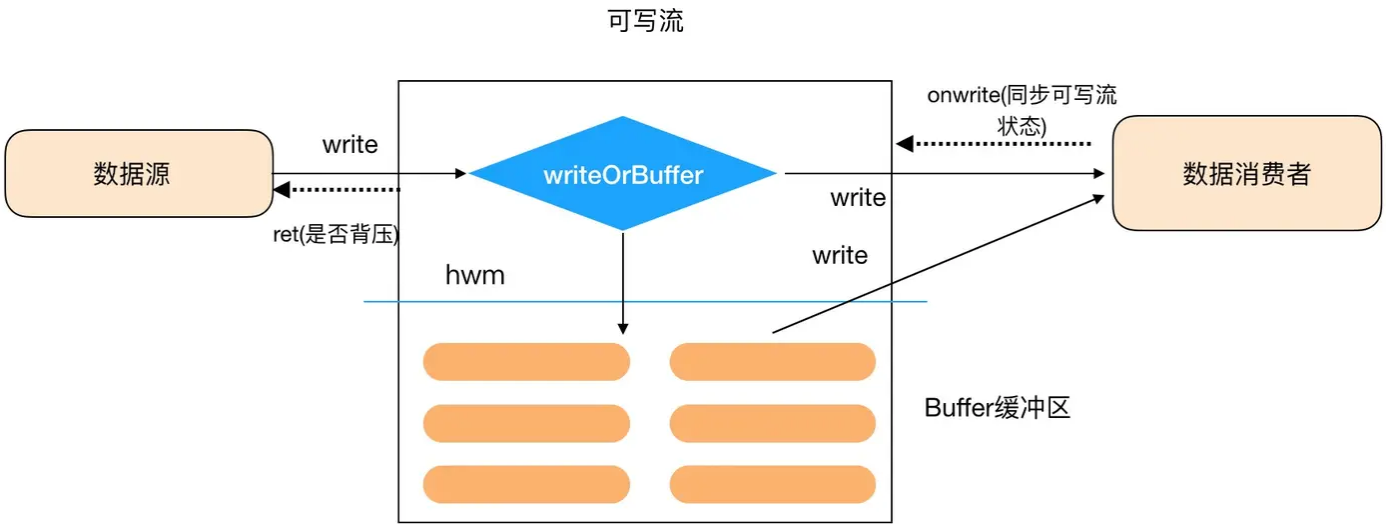


上需要注意的是，在 push 的过程中会有一个判断，有些情况是不进入 Buffer 缓冲区的，直接传递给了下游。

接着，我看看可写流，通常网上的文章有什么问题，如下图，自定义可写流调用 write 方法一定会把数据放入缓存池，如下图所示：



其实，这也是有问题的，可写流也可能不写入缓冲区，而是直接消费掉，真正的可写流示意图如下：



注意上图，自定义可写流的 write 方法传入了 writeOrBuffer 里，这个单词已经很明显了，直接 write 或者 buffer，就是给下游消费者数据，或者存入到 buffer。

最后，我们我们还会介绍

- pipe的原理是什么？为什么它可以做到防止背压（backpressure）？
- 大名鼎鼎的 gulp 库，依赖了 through2 这个库，这个库本质使用的是 transform 流，transform 流能解决背压问题吗？

先说结论，证明过程，有兴趣的同学可以看，很枯燥的 debug 过程

关于自定义可读流

首先，可读流什么时候会不经过 可读流缓冲区 直接给把数据传递给下游呢？关键函数在于可读流源码的 addChunk函数(this.push 最终调用的就是它)，大概扫一下就行，我只是想证明这里有一个 if 函数判断，也证明了 this. push 是有两种走向的，不是很多网络文章说的只会放入到可读流缓冲区：

```
function addChunk(stream, state, chunk, addToFront) {
  if (state.flowing && state.length === 0 && !state.sync &&
      stream.listenerCount('data') > 0) {
    // Use the guard to avoid creating `Set()` repeatedly
    // when we have multiple pipes.
    if (state.multiAwaitDrain) {
      state.awaitDrainWriters.clear();
    } else {
      state.awaitDrainWriters = null;
    }

```

```
    state.dataEmitted = true;
    stream.emit('data', chunk);
  } else {
    // Update the buffer info.
    state.length += state.objectMode ? 1 : chunk.length;
    if (addToFront)
      state.buffer.unshift(chunk);
    else
      state.buffer.push(chunk);

    if (state.needReadable)
      emitReadable(stream);
  }
  maybeReadMore(stream, state);
}
```

注意，我们这里探讨的内容的前期是可读流必须要注册 data 事件。

然后，这个 addChunk 的逻辑主要分为两种：

- 将数据直接给 data 事件注册的回调函数消费
- 将数据放入到可读流的缓冲池（bufferlist）中，bufferlist是一个自定义的链表结构，每次从队列头部取出一个数据给下游。

然后我们再举一个实际的例子，来说明：

```
const { Readable } = require('stream');
const data = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'];
const readableStream = new Readable({
  highWaterMark: 3,
  read() {
    const chunk = data.shift();
    this.push(chunk)
  },
});
readableStream.on('data',(data)=>{
  console.log('data: ', data);
})
```

直接消费 data 的情况是，将 this.push(chunk) 不同步调用，例如

```
setTimeout(() => {
   this.push(chunk)
}, 0)
```

这样数据会直接给下游消费，如果同步调用 this.push(chunk)，这个数据是会放入到缓冲区的，并且消费方式是：

- 现将第一个数据读入缓冲区（对应源码 read(0) 函数）
- 消费第一个数据给 data 事件注册的回调函数，然后将下一个数据读入缓冲区(对应源码的 read() 函数)
- 然后直到读完数据

然后我们接着看可写流的结论：

- 如果同步调用 this.push(chunk)，数据将会存入缓冲区，但是消费方式，可以说基本不咋占用缓冲区，只要你每次push的数据不大，基本就保持在每次 push 数据占用的缓冲区内存大小
- 非同步调用，会直接把数据给下游，不经过缓冲区

关于自定义可写流

我们同样拿一个案例说明：

```
const Stream = require('stream');
const writableStream = Stream.Writable({ highWaterMark: 3, encoding: 'utf8' });
writableStream._write = function (data, encoding, next) {
   next();
}
writableStream.on('finish', () => console.log('done~'));
writableStream.write('123456');
writableStream.write('2123456');
writableStream.end();
```

关键在于 next()

- 是同步调用，此时不会经过可写流缓冲区，直接给下游消费
- 是异步调用，例如 setTimeout 包裹，第一次会不经过缓冲区直接输出，第二次开始，会写入到可写流缓冲区。而且调用过程会让你完全想不到，我们用一个例子来说明异步调用 next 的奇特景观！

```
const Stream = require("stream");
const writableStream = Stream.Writable({ highWaterMark: 3, encoding: "utf8" });
let outerIndex = 1;
let innerIndex = 1;
writableStream._write = function (data, encoding, next) {
  console.log("outerIndex", outerIndex++);
```

```
  setTimeout(() => {
    console.log("innerIndex", outerIndex++, innerIndex++);
    next();
  }, 0);
};
writableStream.on("finish", () => console.log("done~"));
writableStream.write("1");
writableStream.write("2");
writableStream.write("3");
writableStream.write("4");
writableStream.end();
```

我们先看看会打印什么，然后告诉你运行过程：

```
outerIndex 1
innerIndex 2 1
outerIndex 3
innerIndex 4 2
outerIndex 5
innerIndex 6 3
outerIndex 7
innerIndex 8 4
done~
```

我来简述一下运行过程：

- 首先第一次不会 next 虽然是异步调用，但仍然不会走缓存而是直接输出 writableStream.write("1") 直接调用了

```
writableStream._write = function (data, encoding, next) {
  console.log("outerIndex", outerIndex++);
  setTimeout(() => {
    console.log("innerIndex", outerIndex++, innerIndex++);
    next();
  }, 0);
};
```

然后打印 console.log("outerIndex", 1); 再把 setTimeout 的回调函数放入宏任务队列。 接着，所有的

```
writableStream.write("2");
writableStream.write("3");
```

```
3 writableStream.write("4");
```

都会同步放入到可写流缓冲区。此时缓冲区Bufferlist 有三个元素，分别是 [2、3、4]。 接着执行宏任务，取出最开始调用 writableStream.write("1") -> 调用 writableStream._write -> setTimeout 产生的宏任务。

宏任务 打印 console.log("innerIndex", outerIndex++, innerIndex++); 并执行next, next 会清除缓冲区的一个第一个元素，此时缓冲区为 [null, 3, 4]。

接着调用内部的 dowrite 函数，这个函数会继续调用 writableStream._write ，所以又会打印 console 内容，并把 setTimeout 回调放入宏任务队列，依次类推，直到可写流的 buffer 清空。

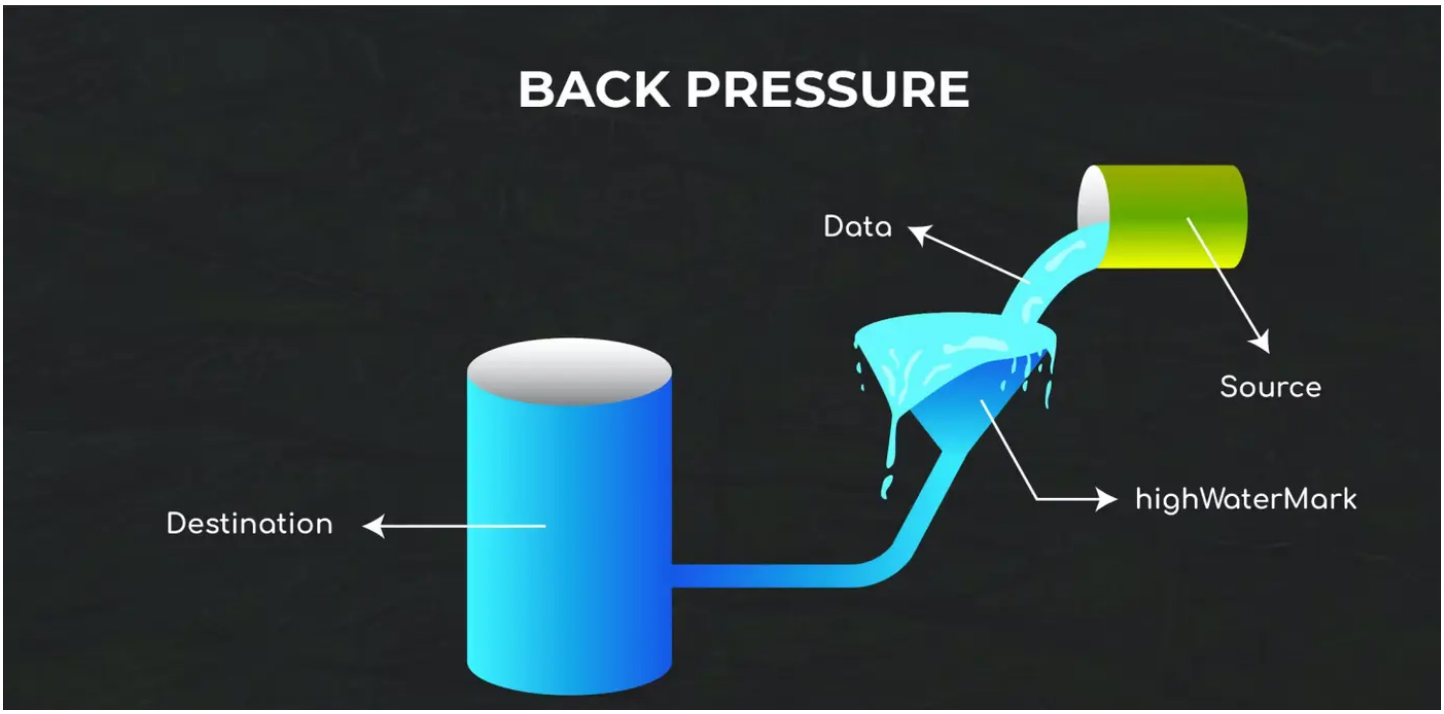
关于 pipe 函数的原理

简单实现如下：

```
1  pipe(ws){
2    // pipe的时候就已经开始读数据了，读数据的同时还会写数据
3    // 如果读的太快
4    this.on('data',(chunk)=>{
5      let flag = ws.write(chunk);
6      if(!flag){
7        this.pause();
8      }
9    });
10   ws.on('drain',()=>{
11     this.resume();
12   })
13 }
```

原理很简单，就是可写流写的太快，就暂停可读流继续传送数据。

但是，你有没有想过，为啥没有控制可读流的速度，比如我可读流数据一下子就溢出可读流缓冲区了，为什么没有暂停可读流？ 如下图，source 太快，水也会溢出。



其实从我们之前的结论也可以看出，你在使用 pipe 的时候，this.push 一定要注意最好同步调用，这样基本上不怎么占用缓冲区（ push 的 data 一次性不要太多）。也就不存在我们说的 可读流数据一下子就溢出可读流缓冲区了。

关于自定义 Transform 流

我在 github 的 node.js 的 issue 里看到一个问题，就是他们不知道如何使用 transform 流，主要疑问是，是否需要再 transform 流里控制背压？如下：

```
export class MyTransform extends stream.Transform {
  constructor() {
    super({ objectMode: true });
  }

  async _transform(chunk: any, encoding: string, callback: TransformCallback) {
    const arrayOfStrings = extractStrings(chunk);
    for (const string of arrayOfStrings) {
      if (!this.push(string)) {
        await new Promise((res) => this.once("drain", res));
        callback();
      }
    }
  }
}
```

其实，按照我们之前对可读流和可写流的结论，是不需要做背压处理的，pipe 函数自动会处理。我们只需要这样使用 Transform 流即可：

```
export class MyTransform extends stream.Transform {
  constructor() {
    super({ objectMode: true });
  }

  async _transform(chunk: any, encoding: string, callback: TransformCallback) {
    const arrayOfStrings = extractStrings(chunk);
    for (const string of arrayOfStrings) {
        this.push(string)
        callback();
    }
  }
}
```

注意，this.push 和 callback 是同步调用的。然后结合 pipe 或者 pipeline，再切记 push 的数据一次性不要太多，就不会有背压的问题了。

推理过程（debug 过程）

先以下的可读流的示意图：

可以看到可读流是通过内部的 this.push 方法把数据放到缓存池或者直接送给下游可写流的，然后可写流可以监听data 事件来消费这些数据。举个例子，我们自定义一个可读流：

```
const { Readable } = require('stream');
const data = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'];
const readableStream = new Readable({
  highWaterMark: 3,
  read() {
    const chunk = data.shift();
    this.push(chunk)
  },
});
readableStream.on('data',(data)=>{
  console.log('data: ', data);
})
```

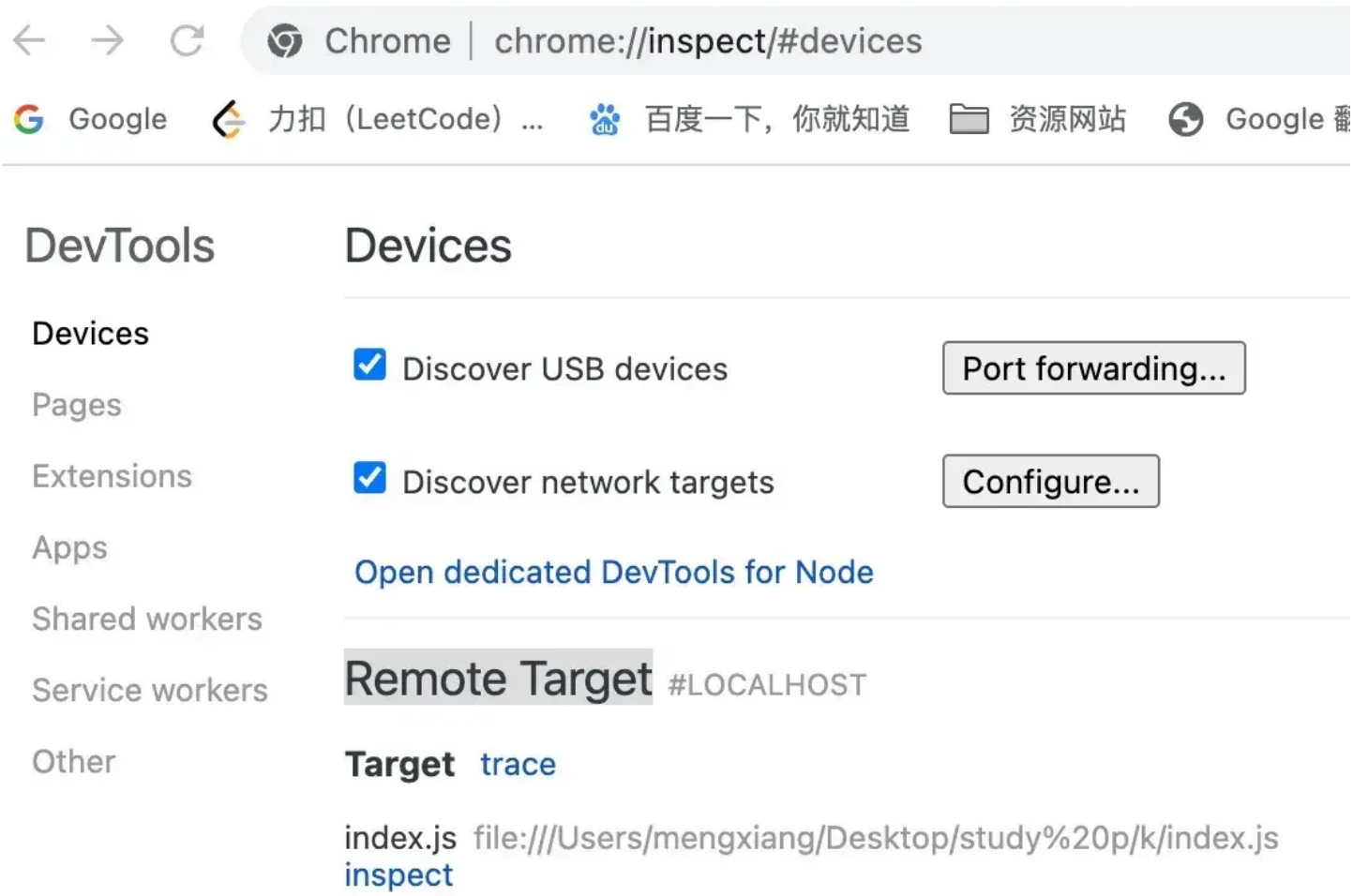
当readableStream注册data事件的时候，流就会源源不断的调用read方法，把数据拿来。

调试方法

我用的是chrome浏览器来协助看源码的方式（js代码，如果要看c++的话这种方式不适合）

```
node --inspect-brk index.js
```

然后在chrome://inspect/#devices中，能看到一个Remote Target的一个列表，点击inspect即可进入调试页面。



然后进去 点击右上角的调试按钮即可一步一步的看代码了，走到readableStream.on这里，我们进入函数，就可以看到node源码了

readable的js源码如下：

正式调试

我们用的案例如下

```
const { Readable } = require('stream');
const data = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'];
const readableStream = new Readable({
  highWaterMark: 3,
  read() {
    const chunk = data.shift();
    this.push(chunk)
  },
});
readableStream.on('data',(data)=>{
```

```
  console.log('data: ', data);
})
```

重点是highWaterMark为3，我们每次往里面放一个字节。后面我们还会举例，如果一次性放5个字节，超过highWaterMark又会怎么样。

首先进入了on方法，注册data事件，一旦注册data事件，就会调用 resume 方法（开启流动模式）

```
// Ensure readable listeners eventually get something.
Readable.prototype.on = function(ev, fn) {
  const res = Stream.prototype.on.call(this, ev, fn);
  const state = this._readableState;
  if (ev === 'data') {
    // Update readableListening so that resume() may be a no-op
    // a few lines down. This is needed to support once('readable').
    state.readableListening = this.listenerCount('readable') > 0;
    // Try start flowing on next tick if stream isn't explicitly paused.
    if (state.flowing !== false)
      this.resume();
  } else if (ev === 'readable') {
    if (!state.endEmitted && !state.readableListening) {
      state.readableListening = state.needReadable = true;
      state.flowing = false;
      state.emittedReadable = false;
      debug('on readable', state.length, state.reading);
      if (state.length) {
        emitReadable(this);
      } else if (!state.reading) {
        process.nextTick(nReadingNextTick, this);
      }
    }
  }
  return res;
};
```

res其实就是继承的Event模块，所以返回的res可以调用on方法来注册data事件 state返回的是Readable，标记的是当前可读流的一些属性，例如初始化时：

- buffer: 这是缓冲区的对象，是一个链表结构，BufferList {head: null, tail: null, length: 0}
- flowing: null，表示是否是流动状态，因为我们这里只看流动模式，这个变量比较重要
- highWaterMark: 3，表示缓冲区大小，单位为字节
- reading: false，是否正在读数据

- sync: true，是否是同步读取数据

这里可以看到，因为state.flowing !== false，所以直接进入了 this.resume(); 我们接着看resume

```
Readable.prototype.resume = function() {
  const state = this._readableState;
  if (!state.flowing) {
    debug('resume');
    // We flow only if there is no one listening
    // for readable, but we still have to call
    // resume().
    state.flowing = !state.readableListening;
    resume(this, state);
  }
  state[kPaused] = false;
  return this;
};
```

因为state.flowing是null，所以 state.flowing = ture（state.readableListening初始化为false），继续调用resume

```
function resume(stream, state) {
  if (!state.resumeScheduled) {
    state.resumeScheduled = true;
    process.nextTick(resume_, stream, state);
  }
}
```

state.resumeScheduled初始化也是false，调用了process.nextTick，在本轮事件循环末尾执行resume_。我们接着等待执行process.nextTick。

```
function resume_(stream, state) {
  debug('resume', state.reading);
  if (!state.reading) {
    stream.read(0);
  }
  state.resumeScheduled = false;
  stream.emit('resume');
  flow(stream);
  if (state.flowing && !state.reading)
    stream.read(0);
}
```

因为state.reading初始化是false，所以走到 stream.read(0);我们接着看read方法

```
Readable.prototype.read = function(n) {
  const nOrig = n;
  n = howMuchToRead(n, state);
  let doRead = state.needReadable;
  if (state.length === 0 || state.length - n < state.highWaterMark) {
    doRead = true;
  }
  if (state.ended || state.reading || state.destroyed || state.errored ||
      !state.constructed) {
    doRead = false;
    debug('reading, ended or constructing', doRead);
  } else if (doRead) {
    debug('do read');
    state.reading = true;
    state.sync = true;
    // If the length is currently zero, then we
    need
    a readable event.
      if (state.length === 0)
    state.needReadable = true;
    // Call internal read method
    this._read(state.highWaterMark);
    state.sync = false;
    // If _read pushed data synchronously, then
    reading
    will be false,
      // and we need to re-evaluate how much data we can return to the user.
      if (!state.reading)
      n = howMuchToRead(nOrig, state);
  }
  let ret;
  if (n > 0)
    ret = fromList(n, state);
  else
    ret = null;
  if (ret === null) {
    state.needReadable = state.length <= state.highWaterMark;
    n = 0;
  } else {
    state.length -= n;
    if (state.multiAwaitDrain) {
      state.awaitDrainWriters.clear();
    } else {
```

```
      state.awaitDrainWriters = null;
    }
  }
  if (state.length === 0) {
    // If we have nothing in the buffer, then we want to know
    // as soon as we
    do
      get something into the buffer.
      if (!state.ended)
    state.needReadable = true;
    // If we tried to read() past the EOF, then emit end on the next tick.
    if (nOrig !== n && state.ended)
      endReadable(this);
  }
  if (ret !== null) {
    state.dataEmitted = true;
    this.emit('data', ret);
  }

  return ret;
};
```

因为刚开始，我们的缓存区肯定是没有数据的，所以state.length === 0 是true, 首先会走到

```
if (state.length === 0 || state.length - n < state.highWaterMark) {
  doRead = true;
}
```

然后走到

```
if(doRead) {
  state.reading = true;
  state.sync = true;
  // If the length is currently zero, then we
  need
  a readable event.
    if (state.length === 0)
  state.needReadable = true;
  // Call internal read method
  this._read(state.highWaterMark);
}
```

this._read 也就是触发我们之前在readableStream上自定义的read方法，(this._read(state.highWaterMark))

```
read() {
  const chunk = data.shift();
  this.push(chunk)
},
```

也就是最终调用了push方法，push方法最终调用了addChunk方法，因为判断条件是：

```
if(chunk && chunk.length > 0){
  addChunk...
}
```

addChunk方法最终在这里判断是否是直接把数据给data事件的回调函数，还是写入到 buffer 缓冲区里(在初始化的时候，顺便把state.reading改为了false)

```
state.flowing && state.length === 0 && !state.sync &&
  stream.listenerCount('data') > 0
```

可读流分水岭（证明文章开头第一个观点）

这里我们详细看下 addChunk 方法, 简单来说就分为两块，

- 将数据直接给监听 data 事件的回调函数 （流动模式）
- 将数据存入缓存中 （暂停模式）

```
function addChunk(stream, state, chunk, addToFront) {
  if (state.flowing && state.length === 0 && !state.sync &&
      stream.listenerCount('data') > 0) {
    // Use the guard to avoid creating `Set()` repeatedly
    // when we have multiple pipes.
    if (state.multiAwaitDrain) {
      state.awaitDrainWriters.clear();
    } else {
      state.awaitDrainWriters = null;
    }

    state.dataEmitted = true;
    stream.emit('data', chunk);
```

```
  } else {
    // Update the buffer info.
    state.length += state.objectMode ? 1 : chunk.length;
    if (addToFront)
      state.buffer.unshift(chunk);
    else
      state.buffer.push(chunk);

    if (state.needReadable)
      emitReadable(stream);
  }
  maybeReadMore(stream, state);
}
```

这里最关键的就是 state.sync 函数是否为 false，正常情况下，进入 addChunk 之前，在 read 函数中，会提前把 state.sync = true，所以同步调用 this.push 总是会把数据放入到缓冲区。

但是，在调用完毕 addChunk ， 在 read 函数中，会把 state.sync = false，所以吗，如果 addChunk 是异步调用，那么意味着 state.sync = false 会比 addChunk 先执行！

所以 addChunk 在判断的时候， state.sync = false ，加上我们本身探讨的就是流动模式，所以其它条件也会促成 addChunk 的第一个 if 判断是 true，从而直接把数据给下游！

我们接着调试之前的案例，因为我们的案例其实就是流动模式，所以将输入放入缓存中：

```
state.flowing && state.length === 0 && !state.sync &&
  stream.listenerCount('data') > 0
```

换一个案例：如果push比highWaterMark大的数据会怎样

我们换一个上来就push比highWaterMark大的数据

```
const { Readable } = require('stream');
const readableStream = new Readable({
  highWaterMark: 3,
  read() {
    const chunk = 'abcdefg';
    this.push(chunk)
  },
});
readableStream.on('data',(data)=>{
  console.log('data: ', data);
})
```

其实调用过程没啥区别，也就是先存到缓冲区，然后消费，再存到缓冲区，然后消费。。。

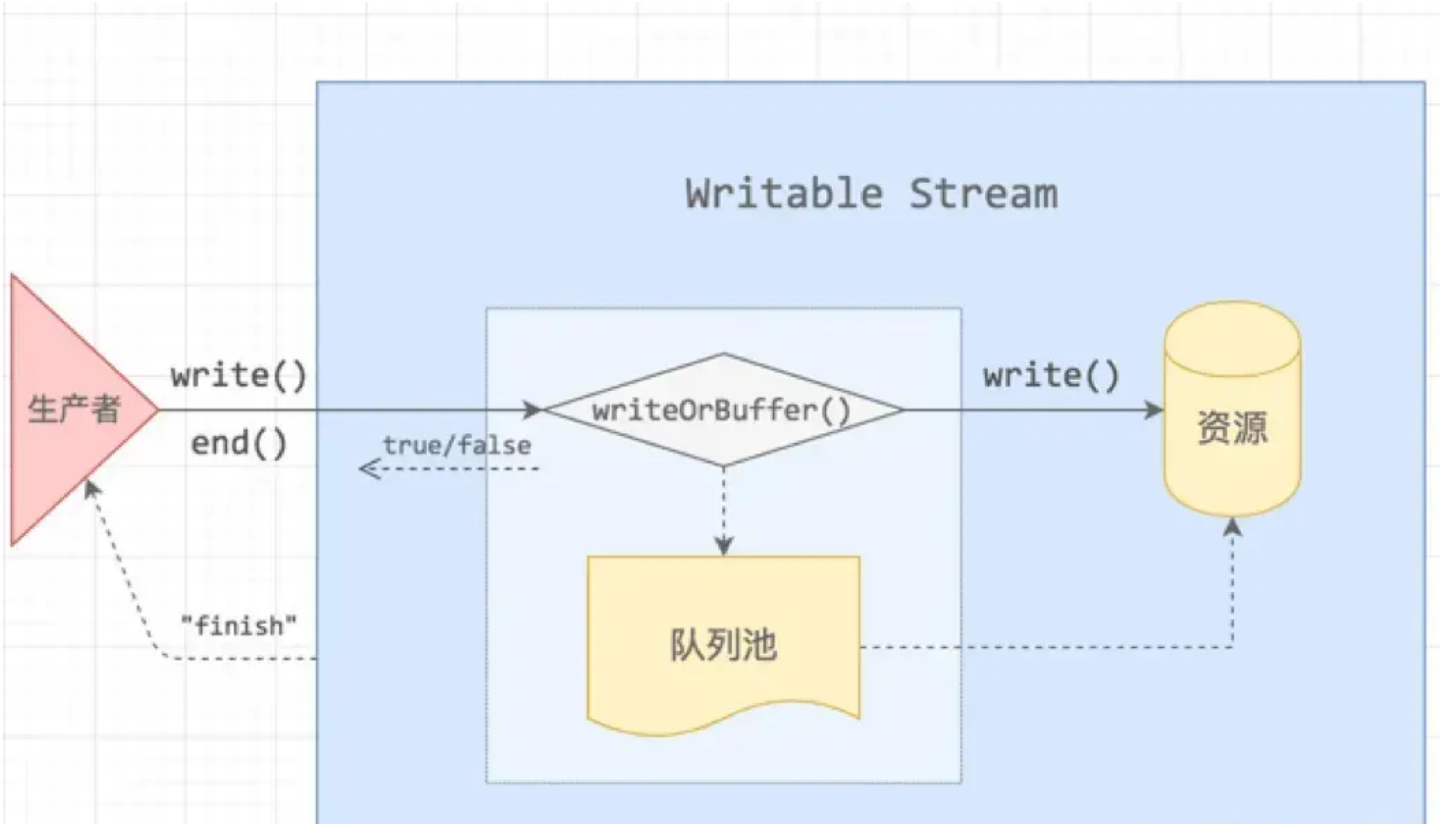
但是你可以看到，如果 push 的数据特别大，内存会处于一个长期很高的情况，并不建议这么做

可写流源码简单解读

有的同学可能不清楚自定义可写流如何实现，我们先简单了解下：

```
1 const Stream = require('stream');
2 const writableStream = Stream.Writable();
3 writableStream._write = function (data, encoding, next) {
4   next();
5 }
6 writableStream.on('finish', () => console.log('done~'));
7 writableStream.write('写入数据，');
8 writableStream.end();
```

如上，只要write方法会调用_write，_write接收写入的数据。



我们打断点进入到write方法中,案例就上面的ritableStream.write('写入数据，');。

```
1 Writable.prototype.write = function(chunk, encoding, cb) {
2   return _write(this, chunk, encoding, cb) === true;
```

```
};
```

此时只有chunk是有数据，encoding为undefined（会帮我们默认设为utf8，highwatermark会置为16384，cb为空） 我们看一下_write函数，主要就是初始化writeable的state，比如encoding, 然后调用了

```
return writeOrBuffer(stream, state, chunk, encoding, cb);
```

注意，注意，writeOrBuffer 可以看做是我们可写流的 write 方法！

writeOrBuffer 源码如下：

```
function writeOrBuffer(stream, state, chunk, encoding, callback) {
  // 我们这里的数据length是15
  const len = state.objectMode ? 1 : chunk.length;
  // 写缓存大小加上15
  state.length += len;
  // 此时因为highWaterMark是16384，所以ret是true,而且一般情况下都是true
  const ret = state.length < state.highWaterMark;
  // We must ensure that previous needDrain will not be reset to false.
  if (!ret)
    state.needDrain = true;

if (state.writing || state.corked || state.errored || !state.constructed) {
  state.buffered.push({ chunk, encoding, callback });
  if (state.allBuffers && encoding !== 'buffer') {
    state.allBuffers = false;
  }
  if (state.allNoop && callback !== nop) {
    state.allNoop = false;
  }
} else {
  state.writelen = len;
  state.writecb = callback;
  state.writing = true;
  state.sync = true;
  // stream._write就是我们外部写的_write函数
  stream._write(chunk, encoding, state.onwrite);
  state.sync = false;
  }
}
```

分水岭

因为 writeOrBuffer 可以看做是我们外面自定义的 write 方法，也就是写数据的方法。 然后，这里最重要的就是 state.writing 是否为 true，为 true 就表示写入的数据需要存入可读流缓存，否则不会。

还有另一个关键点就是 stream._write(chunk, encoding, state.onwrite);

- stream._write就是我们外部写的 _write 函数。
- state.onwrite 就是 next 回调函数

这里我们可以看到，第一次 state.writing 因为 默认是 false，所以会直接输出数据调用 stream._write 。

到这里，是很轻松的，但是问题就在如果 next 被异步包裹，就比较麻烦了。例如：

```
const Stream = require("stream");
const writableStream = Stream.Writable({ highWaterMark: 3, encoding: "utf8" });
let outerIndex = 1;
let innerIndex = 1;
writableStream._write = function (data, encoding, next) {
  console.log("outerIndex", outerIndex++);
  setTimeout(() => {
    console.log("innerIndex", outerIndex++, innerIndex++);
    next();
  }, 0);
};
writableStream.on("finish", () => console.log("done~"));
writableStream.write("1");
writableStream.write("2");
writableStream.write("3");
writableStream.write("4");
writableStream.end();
```

其中，第一次调用 writableStream.write("1"); 的时候，其实调用内部的 writeorBuffer，因为 state.writing 因为 默认是 false，所以会直接输出数据调用 stream._write ，stream._write 是我们外部自定义的 writableStream._write， writableStream._write 直接同步被调用。

这里的问题在于，writableStream._write 之前 state.writing 被设为了 true，但是 writableStream._write 中的 onwrite 又是被 setTimeout 包裹，调用更晚（onwrite 会把 state.writing 被设为了 false）。

然后第二次调用的时候，我们看 writeorBuffer 函数有,注意下面第一句的 state.writing

```
if (state.writing || state.corked || state.errored || !state.constructed) {
  state.buffered.push({ chunk, encoding, callback });
```

```
  if (state.allBuffers && encoding !== 'buffer') {
    state.allBuffers = false;
  }
  if (state.allNoop && callback !== nop) {
    state.allNoop = false;
  }
} else {
  state.writelen = len;
  state.writecb = callback;
  state.writing = true;
  state.sync = true;
  // stream._write就是我们外部写的_write函数
  stream._write(chunk, encoding, state.onwrite);
  state.sync = false;
  }

```

因为 state.writing 改为 true 了，所以会被放入到可写流缓冲区 -> state.buffered.push({ chunk, encoding, callback });

这下懂了吧，最后再看下 state.onwrite，也就是调用 next 函数的时候，是如何清空缓冲区的。

```
  }
} else {
  if (state.buffered.length > state.bufferedIndex) {
    clearBuffer(stream, state);
  }
```

这里会有一个判断，就是缓冲区的长度和已经更新的缓冲区的 bufferedIndex ，最开始 bufferedIndex 是 0，每次把缓冲区数据输出后会 +1，直到缓冲区全部清空。

clearBuffer 会： 首先 buffered[i++] = null，也就是清空一个缓冲区数据

```
  const { chunk, encoding, callback } = buffered[i];
  buffered[i++] = null;
  const len = objectMode ? 1 : chunk.length;
  doWrite(stream, state, false, len, chunk, encoding, callback);
```

然后，调用 stream._write，也就是我们外部自定义 _write

```
state.writecb = cb;  cb = () => nop()
state.writing = true;
state.sync = true;
if (state.destroyed)
  state.onwrite(new ERR_STREAM_DESTROYED('write'));
else if (writev)  writev = false
  stream._writev(chunk, state.onwrite);  stream = Writable {_w
else
  stream._write(chunk, encoding, state.onwrite);
```