

淘宝前端团队 (FED)

用技术为体验提供无限可能

主页 Web开发 Node.js 无线开发 工具&平台 团队生活 关于我们

Q 搜索

关注我们：



下一篇

机器学习, Hello World from Javascript!



上一篇

GCanvas 渲染引擎介绍



开源产品

Rax

ICE

Pandora.js

BindingX

GCanvas

G3D

最新文章

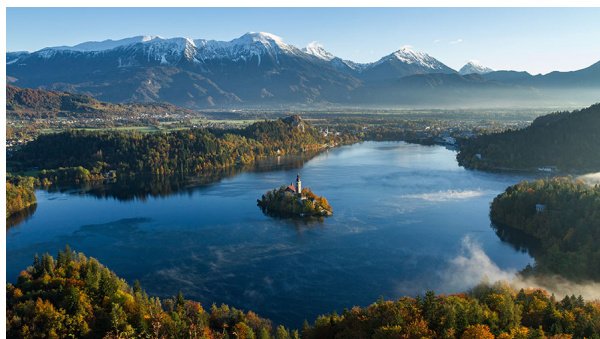
Web开发

G3D —— Hybrid 环境下的
WebGL 3D 渲染引擎

Node.js

深入理解 Node.js Stream 内部机制

作者: 阎王 发表于: 2017-08-31



相信很多人对 Node.js 的 Stream 已经不陌生了, 不论是请求流、响应流、文件流还是 socket 流的底层都是使用 stream 的, 甚至我们平时用的最多的 console.log 打印日志也使用了它, 不信你打开 Node.js runtime 的源码, 看看 lib/console.js :

by 叶斋
at 2018-03-05

团队生活
淘宝技术部 2018 实习生内部推荐启动啦
by 浩睿
at 2018-02-27

Web开发
Rax 系列教程 (native 扫盲)
by 亚城
at 2018-02-06

无线开发
实现一个 JavaScriptCore 的 debugger —— iOS 篇
by 寒泉
at 2018-01-23

Web开发
Rax 系列教程 (上手)
by 亚城
at 2018-01-18

微信公众号



分类

► Node.js (52)

```
function write(ignoreErrors, stream,
  // ...
  stream.once('error', noop);
  stream.write(string, errorHandler).
  //...
}
```

```
Console.prototype.log = function log
  write(this._ignoreErrors,
    this._stdout,
    `${util.format.apply(null, a
    this._stdoutErrorHandler);
};
```



Stream 模块做了很多事情，了解了 Stream，那么 Node.js 中其他很多模块理解起来就顺畅多了。

stream 模块

如果你了解 生产者和消费者问题 的解法，那理解 stream 就基本没有压力了，它不仅仅是资料的起点和落点，还包含了一系列状态控制，可以说一个 stream 就是一个状态管理单元。了解内部机制的最佳方式除了看 Node.js 官方文档，还可以去看看 Node.js 的 源码：

- lib/module.js
- lib/_stream_readab
- lib/_stream_writable.js
- lib/_stream_tranform.js
- lib/_stream_duplex.js



- ▶ Web开发 (62)
- ▶ 团队生活 (7)
- ▶ 工具&平台 (13)
- ▶ 无线开发 (22)

归档

- ▶ 2018 (7)
- ▶ 2017 (18)
- ▶ 2016 (57)
- ▶ 2015 (51)
- ▶ 2014 (7)
- ▶ 2013 (3)
- ▶ 2012 (4)
- ▶ 2010 (5)
- ▶ 2009 (1)
- ▶ 2008 (2)
- ▶ 2007 (1)

链接

- ▶ Node 地下铁
- ▶ alinode
- ▶ 百度 FEX
- ▶ 奇舞团
- ▶ 凹凸实验室

把 Readable 和 Writable 看明白，
Transform 和 Duplex 就不难理解了。

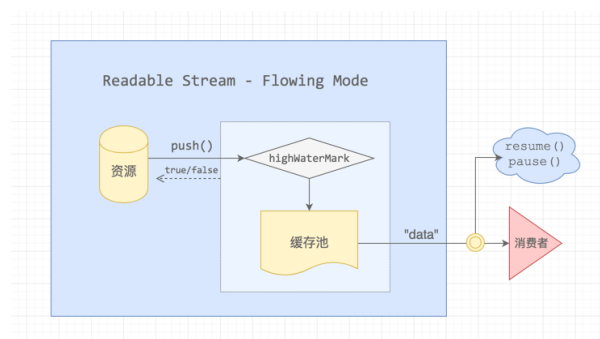
Readable Stream

Readable Stream 存在两种模式，一种是叫做 Flowing Mode，流动模式，在 Stream 上绑定 ondata 方法就会自动触发这个模式，比如：

```
const readable = getReadableStreamSomehow()
readable.on('data', (chunk) => {
  console.log(`Received ${chunk.length} bytes of data.`);
});
```



这个模式的流程图如下：



资源的数据流并不是直接流向消费者，而是先 push 到缓存池。缓存池有一个水位标记 highWaterMark，超过这个标记阈值，push 的时候会返回 false，什么场景下会出现这种情况呢？

- 消费者主动执行了 .pause()

► 腾讯 AlloyTeam

- 消费速度比数据 push 到缓存池的生产速度慢

有个专有名词来形成这种情况，叫做「背压」，**Writable Stream** 也存在类似的情况。

流动模式，这个名词还是很形象的，缓存池就像一个水桶，消费者通过管口接水，同时，资源池就像一个水泵，不断地往水桶中泵水，而 **highWaterMark** 是水桶的浮标，达到阈值就停止蓄水。

下面是一个简单的 Demo：

```
const Readable = require('stream').Readable;

// Stream 实现
class MyReadable extends Readable {
  constructor(dataSource, options) {
    super(options);
    this.dataSource = dataSource;
  }
  // 继承了 Readable 的类必须实现这个函数
  // 触发系统底层对流的读取
  _read() {
    const data = this.dataSource.makeData();
    this.push(data);
  }
}

// 模拟资源池
const dataSource = {
  data: new Array(10).fill('-'),
  // 每次读取时 pop 一个数据
  makeData() {
    if (!dataSource.data.length) return;
    return dataSource.data.pop();
  }
}
```



```
};
```

```
const myReadable = new MyReadable(data);  
myReadable.setEncoding('utf8');  
myReadable.on('data', (chunk) => {  
  console.log(chunk);  
});
```



另外一种模式是 Non-Flowing Mode ,
没流动, 也就是暂停模式, 这是
Stream 的预设模式, Stream 实例的
_readableState.flow 有三个状态, 分
别是:

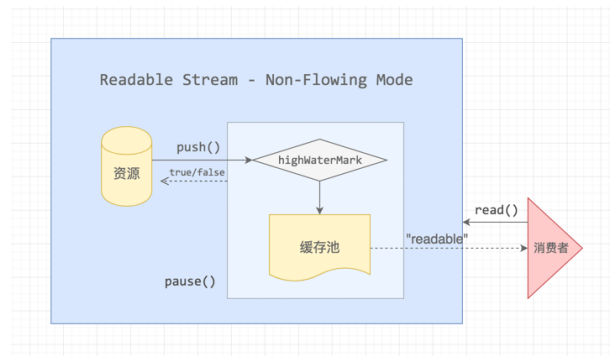
- `_readableState.flow = null` , 暂时
没有消费者过来
- `_readableState.flow = false` , 主
动触发了 `.pause()`
- `_readableState.flow = true` , 流
动模式

当我们监听了 `onreadable` 事件后,
会进入这种模式, 比如:

```
const myReadable = new MyReadable(data);  
myReadable.setEncoding('utf8');  
myReadable.on('readable', () => {});
```



监听 `readable` 的回调函数第一个参数
不会传递内容, 需要我们通过
`myReadable.read()` 主动读取, 为啥
呢, 可以看看下面这张图:



资源池会不断地往缓存池输送数据，直到 `highWaterMark` 阈值，消费者监听了 `readable` 事件并不会消费数据，需要主动调用 `.read([size])` 函数才会从缓存池取出，并且可以带上 `size` 参数，用多少就取多少：

```
const myReadable = new MyReadable(data);
myReadable.setEncoding('utf8');
myReadable.on('readable', () => {
  let chunk;
  while (null !== (chunk = myReadable.read())) {
    console.log(`Received ${chunk.length} bytes of data`);
  }
});
```


这里需要注意一点，只要数据达到缓存池都会触发一次 `readable` 事件，有可能出现「消费者正在消费数据的时候，又触发了一次 `readable` 事件，那么下次回调中 `read()` 数据可能为空」的情况。我们 ↑ 过 `_readableState.buffer` 来看看缓存池到底缓存了多少资源：

```
let once = false;
myReadable.on('readable', (chunk) =>
```

```
console.log(myReadable._readableSt;
if (once) return;
once = true;
console.log(myReadable.read());
});
```

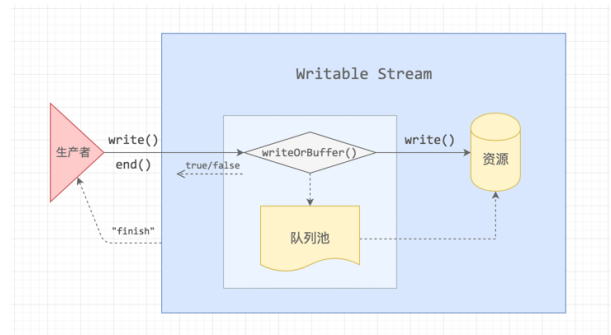
上面的代码我们只消费一次缓存池的数据，那么在消费后，缓存池又收到了一次资源池的 `push` 操作，此时还会触发一次 `readable` 事件，我们可以看看这次存了多大的 `buffer`。

需要注意的是，`buffer` 大小也是有上限的，默认设置为 `16kb`，也就是 `16384` 个字节长度，它最大可设置为 `8Mb`，没记错的话，这个值好像是 `Node` 的 `new space memory` 的大小。

上面介绍了 `Readable Stream` 大概的机制，还有很多细节部分没有提到，比如 `Flowing Mode` 在不同 `Node` 版本中的 `Stream` 实现不太一样，实际上，它有三个版本，上面提到的是第 2 和第 3 个版本的实现；再比如 `Mixins Mode` 模式，一般我们只推荐（允许）使用 `ondata` 和 `onreadable` 的一种来源 `Readable Stream`，但是如果要求  `Flowing Mode` 的情况下使用 `ondata` 如何实现呢？那么就可以考虑 `Mixins Mode` 了。

Writable Stream

原理与 **Readable Stream** 是比较相似的，数据流过来的时候，会直接写入到资源池，当写入速度比较缓慢或者写入暂停时，数据流会进入队列池缓存起来，如下图所示：



当生产者写入速度过快，把队列池装满了之后，就会出现「背压」，这个时候是需要告诉生产者暂停生产的，当队列释放之后，**Writable Stream** 会给生产者发送一个 **drain** 消息，让它恢复生产。下面是一个写入一百万条数据的 **Demo**：

```

function writeOneMillionTimes(writer,
  let i = 10000;
  write();
  function write() {
    let ok = true;
    while(i-- > 0 && ok) {
      // 写入结束时回调
      ok = writer.write(data, encoding);
    }
    if (i > 0) {
      // 这里提前停下了，等待 'drain' 事件
      console.log('drain', i);
      writer.once('drain', write);
    }
  }
}
  
```


我们构造一个 **Writable Stream**，在写入到资源池的时候，我们稍作处理，让它效率低一点：


```
const Writable = require('stream').Writable
const writer = new Writable({
  write(chunk, encoding, callback) {
    // 比 process.nextTick() 稍慢
    setTimeout(() => {
      callback && callback();
    });
  }
});

writeOneMillionTimes(writer, 'simple',
  console.log('end'));
});
```

最后执行的结果是：

```
drain 7268
drain 4536
drain 1804
end
```

说明程序遇到了三次「背压」，如果我们没有在上面绑定

`writer.once('drain')`，用  的结果就是 **Stream** 将第一次数据消耗完变结束了程序。

pipe

了解了 `Readable` 和 `Writable`, `pipe` 这个常用的函数应该就很好理解了,

```
readable.pipe(writable);
```

这句代码的语意性很强, `readable` 通过 `pipe` (管道) 传输给 `writable`, `pipe` 的实现大致如下 (伪代码):

```
Readable.prototype.pipe = function(writable) {
  this.on('data', (chunk) => {
    let ok = writable.write(chunk);
    // 背压, 暂停
    !ok && this.pause();
  });
  writable.on('drain', () => {
    // 恢复
    this.resume();
  });
  // 告诉 writable 有流要导入
  writable.emit('pipe', this);
  // 支持链式调用
  return writable;
};
```

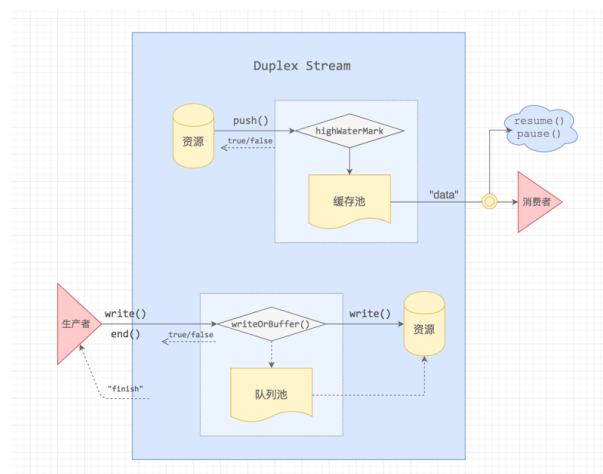
上面做了五件事情:

- `emit(pipe)`, 通知写入
- `.write()`, 新数据写入
- `.pause()`, 消费者变慢, 暂停写入
- `.resume()`, 消费者完成消费, 继续写入
- `return writable`, 支持链式调用

当然，上面只是最简单的逻辑，还有很多异常和临界判断没有加入，具体可以去看看 Node.js 的代码（[/lib/_stream_readable.js](#)）。


Duplex Stream

Duplex，双工的意思，它的输入和输出可以没有任何关系，



Duplex Stream 实现特别简单，不到一百行代码，它继承了 Readable Stream，并拥有 Writable Stream 的方法（源码地址）：

```
const util = require('util');
const Readable = require('_stream_readable');
const Writable = require('_stream_writable');

util.inherits(Duplex,  );

var keys = Object.keys(Writable.prototype);
for (var v = 0; v < keys.length; v++) {
  var method = keys[v];
  if (!Duplex.prototype[method])
    Duplex.prototype[method] = Writable.prototype[method];
}
```



我们可以通过 `options` 参数来配置它为只可读、只可写或者半工模式，一个简单的 Demo：

```
var Duplex = require('stream').Duplex;

const duplex = Duplex();

// readable
let i = 2;
duplex._read = function () {
  this.push(i-- ? 'read ' + i : null);
};
duplex.on('data', data => console.log(data));

// writable
duplex._write = function (chunk, encoding, callback) {
  console.log(chunk.toString());
  callback();
};
duplex.write('write');
```



输出的结果为：

```
write
read 1
read 0
```

可以看出，两个管道是相

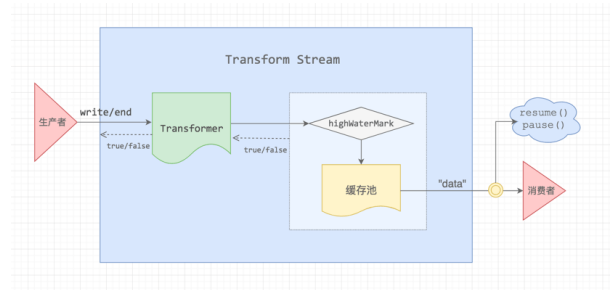


不干

扰的。

Transform Stream

Transform Stream 集成了 Duplex Stream，它同样具备 Readable 和 Writable 的能力，只不过它的输入和输出是存在相互关联的，中间做了一次转换处理。常见的处理有 Gzip 压缩、解压等。



Transform 的处理就是通过 `_transform` 函数将 Duplex 的 Readable 连接到 Writable，由于 Readable 的生产效率与 Writable 的消费效率是一样的，所以这里 Transform 内部不存在「背压」问题，背压问题的源头是外部的生产者和消费者速度差造成的。

关于 Transform Stream，我写了一个简单的 Demo：

```
const Transform = require('stream').Transform;
const MAP = {
  'Barret': '靖',
  'Lee': '李'
};

class Translate extends Transform {
  constructor(dataSource, options) {
    super(options);
  }
  _transform(buf, enc, next) {
```



```
const key = buf.toString();
const data = MAP[key];
this.push(data);
next();
}
}

var transform = new Translate();
transform.on('data', data => console
transform.write('Lee');
transform.write('Barret');
transform.end();
```

小结

本文主要参考和查阅 **Node.js** 官网的文档和源码，细节问题都是从源码中找到的答案，如有理解不准确之处，还请斧正。关于 **Stream**，这篇文章只是讲述了基础的原理，还有很多细节之处没有讲到，要真正理解它，还是需要多读读文档，写写代码。

了解了这些 **Stream** 的内部机制，对我们后续深入理解上层代码有很大的促进作用，特别希望初学 **Node.js** 的同学花点时间进来看看。

题图：

<https://unsplash>

By @Neven Krcm



photo

#Node.js #Stream

评论

分享到