Node Stream 流（一）流的基本介绍及流方式下载文件

什么是流，为什么要使用流？之前对流的概念一直比较模糊。最近有时间好好的看了看关于流的知识，总算有了一个比较清晰的认识。

首先我们来看一下什么是流。

就我自己的观点来看，其实流就是在两个设备之间建立一个管道，然后通过管道将数据以流动的方式传输。如何来理解这个以流动的方式呢？

举个例子来说吧，当我们读取文件的时候，如果不使用流的方式读取的话，我们会将整个文件的内容先通过I/O设备写进内存，然后再由消费者去内存中读取。而使用流的方式是边将文件内容写入缓存边由消费者去读取，不用将整个文件先写进内存，从而节省了内存的空间。

不使用流的方式

**内存**

将整个文件写入内存

从内存中读取文件内容

500M大小文件

500M数据

**生产者**

**消费者**

当一个文件非常大的时候我们看到，不使用流的方式的话，占用的内存也是相当大的。这样就影响了响应的速度。对用户的体验也是有一定的影响的。

var http = require('http');

var fs = require('fs');

var server = http.createServer(function (req, res) {

fs.readFile(\_\_dirname + '/data.txt', function (err, data) {  
 res.writeHead(200,{  
 'Content-Type': 'application/octet-stream',  
 'Content-Disposition': 'attachment; filename=data.txt',  
 'Accept-Length': 1024,  
 });  
 res.end(data);  
});

}).listen(8000)

上面是一段文件下载的功能。要下载的文件假设500M的话，其内存占用如下

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

5601 root 15 0 1212m 516m 9460 R 0.7 51.7 0:10.09 node

5643 root 15 0 12764 1116 836 R 0.3 0.1 0:00.47 top

下载一个500M的文件，上述方法会占用500多兆的内存。所以说我们应该尽量避免上述方式出现在我们代码中。

下面我们看流的方式读取文件

500M大小文件

50M数据1

**生产者**

**消费者**

写入部分数据1

写入部分数据1

50M数据2

**生产者**

**消费者**

写入部分数据2

写入部分数据2

50M数据3

**生产者**

**消费者**

写入部分数据3

写入部分数据3

我们看使用流的方式的话，内存占用非常小，当然上面只是假定每次都是50M。经过测试内存使用情况流的方式平均是前者的十分之一。

其实上面流的过程我们可以这样理解

500M文件

**消费者**

**生产者**

数据块3

数据块2

数据块1

**消费者**

**生产者**

这就是我理解的流。

**var *http*** = require('http');  
**var *fs*** = require('fs');

**var *server*** = ***http***.createServer(**function** (req, res) {

*res.writeHead(200,{  
 'Content-Type': 'application/octet-stream',  
 'Content-Disposition': 'attachment; filename=data.txt',  
 'Accept-Length': 1024,  
});*

**var** stream = ***fs***.createReadStream(\_\_dirname + '/data1.txt' );

stream.pipe(res)

}).listen(8000);

上面代码是使用流的方式来下载文件，其内存占用情况如下。

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

5673 root 15 0 776m 49m 9992 S 7.3 5.0 0:04.47 node

5683 root 15 0 12764 1112 836 R 0.3 0.1 0:00.18 top

我们看到占用内存再50M左右。是不使用流的方式的十分之一。

至于为什么使用流，流的好处通过我们上面的例子我们也能看到，其最大的一个好处就是节省内存，提高程序的运行速度。通过上面两段文件下载的代码我们也能得到结论。除此之外node中的管道函数.pipe()（管道是流中一个非常重要的概念）还能根据消费者读取的速度通过阀门来控制写入的速度。

代码中在合适的地方加入流的使用，确能大大提高程序的性能。

Node Stream 流（二）流的四种基本类型

在Node Stream 流（一）中对流的机制以及使用流的优点做了一个简单的介绍，在这篇文章中我们继续介绍流，对流的四种基本类型——Readable、Writable、Duplex和Transform——进行一个简单的介绍。

在对这几种类型进行介绍之前，我们先来介绍一个非常重要的函数——**.pipe()**

**pipe**

不同类型流之间都可以使用pipe函数来对输入和输出进行匹配。

什么意思呢？pipe我们知道是管道的意思。其实理解起来也非常容易，无非就是在Readable和Writable之间使用.pipe()函数建立一个管道，将Readable流中的数据传输给Writable流。

没错，.pipe()就是一个函数，其调用方式如下

Readable.pipe(Writeable);

.pipe()前面必须是Readable或者Transform，括号内是Writeable。也就是说这个管道是单向的。但是，.pipe()函数的返回值也可以看做是一个资源流，也就是说.pipe()函数可以链式调用。

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

它就相当于

a.pipe(b);

b.pipe(c);

c.pipe(d);

但是这里需要注意的是，中间的b和c必须是Transform流。

**var *fs*** = require('fs');  
**var *stream*** = ***fs***.createReadStream('./data.txt');  
***stream***.pipe(process.stdout);

上面的一段程序就是创建data.txt文件的读取流，然后将其传输给process.stdout（标准输出）

**Readable Stream**

Readable Stream其实可以理解为生产者，将生产的数据通过.pipe()函数传输给Writeable 、Transform或者Duplex流，从而进行相应的处理。

Readable.pipe(destination);

首先看一个简单的例子

var Readable = require('stream').Readable;

var rs = new Readable;

rs.push('beep ');

rs.push('boop\n');

rs.push(null);

rs.pipe(process.stdout);

rs.push()是向Readable stream中放入数据。其中rs.push(null)是告诉消费者数据已经生产完了。

但是看上面的例子，在消费者读取数据之前，还是需要将生产的数据缓存到内存中。那是不是有一种更好的方式是当消费者请求数据的时候再去生产数据呢。这个还真可以有，可以通过.\_read函数来实现

**var *stream*** = require('stream');  
**var *c*** = 97;  
**function** *readableStream*(num){  
 **var** rs = ***stream***.Readable();  
 **var** count = 0;  
 rs.\_read = **function**(){  
 **if**(count > num || ***c*** > 'z'.charCodeAt(0)){  
 **return** rs.push(**null**);  
 }  
 setTimeout(**function**(){  
 rs.push(String.fromCharCode(***c***++));  
 },100)  
 count++;  
 }  
 **return** rs;  
}

**rws** = *readableStream*();

rws.pipe(process.stdout);

这个例子就是当有消费者准备好读取数据的时候我们才将a-z放入Readable stream中。默认情况下只能向流中放入string或者buffer类型的数据，如果想让如任意合法类型的数据，可以在Readable()函数中加入以下选项

**var** rs = ***stream***.Readable({ **objectMode**: **true** });

**Writable Stream**

Writable stream相对于Readable stream就很好理解了，前者只能作为数据的目的地，不能作为数据的来源。相反后者只能作为数据的来源不能作为数据的目的地。

src.pipe(Writable);

在上面案例的基础上我们来扩展Writable stream

**var *stream*** = require('stream');

...

**function** *writeableStream*(){  
 **var** ws = ***stream***.Writable();  
 ws.\_write = **function**(chunk,enc,next){  
 **console**.log(chunk.toString());  
 next();  
 }  
 **return** ws;  
}

rws = *readableStream*();  
**wws** = *writeableStream*();

rws.pipe(**wws**);

这里我们是将放入Readable stream中的a-z传输给Writable stream，由其来进行输出。

其中.\_write 函数我们看到有三个参数 ，chunk就是由Readable stream写入的数据——默认情况下是经过编码的buffer类型的数据；enc是编码的类型，它是一个字符串，默认情况下是“buffer”；最后next是一个回调函数——next()，该函数的作用就是告诉消费者可以写入更多的数据。

默认情况下Writable stream写入的数据都是经过编码的buffer数据。当然我们可以通过以下代码来改变这个默认设置。

**var** ws = ***stream***.Writable({ **decodeStrings**: **false** });

**Duplex Stream**

Duplex stream既可以作为Readable stream也可以作为Writable stream。也就是说该类型的数据流是可以双向流动的。

a.pipe(*b*).pipe(a)

其实理解起来很简单，就相当于我们现实中的电话，我们既可以发出声音也可以接收声音。

**Transform Stream**

Transform stream可以认为是Duplex stream的一种特定的类型。首先它可以作为Writable stream接收Readable stream中的数据进行特定的处理然后再作为Readable stream将经过特定处理的数据通过.pipe函数传输给Writable stream。

我们继续扩展上面的应用

**var *stream*** = require('stream');

...

**function** *transformStream\_ToUper*(){  
 **var** ts = ***stream***.Transform();  
 ts.\_transform = **function**(chunk,enc,next){  
 chunk = chunk.toString().toUpperCase();  
 ts.push(chunk);  
 next();  
 }  
 **return** ts;  
}

**rws** = *readableStream*();  
**wws** = *writeableStream*();  
**tws** = *transformStream\_ToUper*();

**rws**.pipe(**tws**).pipe(**wws**);

该程序的功能就是将Readable stream 生产的a-z转换为大写的A-Z然后再输出。

我们看上面的最后一句代码就相当于

**rws**.pipe(**tws**);  
**tws**.pipe(**wws**);

其中 tws就是Transform stream，既可以作为Readable stream也可以作为Writable stream。

到此，关于node stream的介绍就告一段落。希望对大家有所帮助。