**Nodejs Buffer**

Pure JavaScript is Unicode friendly, but it is not so for binary data. While dealing with TCP streams or the file system, it's necessary to handle octet streams. Node provides Buffer class which provides instances to store raw data similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap.

Buffer class is a global class that can be accessed in an application without importing the buffer module.

**Creating Buffers**

Node Buffer can be constructed in a variety of ways.

**Method 1**

Following is the syntax to create an uninitiated Buffer of 10 octets −

var buf = new Buffer(10);

**Method 2**

Following is the syntax to create a Buffer from a given array −

var buf = new Buffer([10, 20, 30, 40, 50]);

**Method 3**

Following is the syntax to create a Buffer from a given string and optionally encoding type −

var buf = new Buffer("Simply Easy Learning", "utf-8");

Though "utf8" is the default encoding, you can use any of the following encodings "ascii", "utf8", "utf16le", "ucs2", "base64" or "hex".

**Writing to Buffers**

Syntax

Following is the syntax of the method to write into a Node Buffer −

buf.write(string[, offset][, length][, encoding])

**Parameters**

Here is the description of the parameters used −

string − This is the string data to be written to buffer.

offset − This is the index of the buffer to start writing at. Default value is 0.

length − This is the number of bytes to write. Defaults to buffer.length.

encoding − Encoding to use. 'utf8' is the default encoding.

**Return Value**

This method returns the number of octets written. If there is not enough space in the buffer to fit the entire string, it will write a part of the string.

Example

Live Demo

buf = new Buffer(256);

len = buf.write("Simply Easy Learning");

console.log("Octets written : "+ len);

When the above program is executed, it produces the following result −

Octets written : 20

**Reading from Buffers**

Syntax

Following is the syntax of the method to read data from a Node Buffer −

buf.toString([encoding][, start][, end])

Parameters

Here is the description of the parameters used −

encoding − Encoding to use. 'utf8' is the default encoding.

start − Beginning index to start reading, defaults to 0.

end − End index to end reading, defaults is complete buffer.

**Return Value**

This method decodes and returns a string from buffer data encoded using the specified character set encoding.

**Example**

Live Demo

buf = new Buffer(26);

for (var i = 0 ; i < 26 ; i++) {

buf[i] = i + 97;

}

console.log( buf.toString('ascii')); // outputs: abcdefghijklmnopqrstuvwxyz

console.log( buf.toString('ascii',0,5)); // outputs: abcde

console.log( buf.toString('utf8',0,5)); // outputs: abcde

console.log( buf.toString(undefined,0,5)); // encoding defaults to 'utf8', outputs abcde

When the above program is executed, it produces the following result −

abcdefghijklmnopqrstuvwxyz

abcde

abcde

abcde

**Convert Buffer to JSON**

Syntax

Following is the syntax of the method to convert a Node Buffer into JSON object −

buf.toJSON()

Return Value

This method returns a JSON-representation of the Buffer instance.

Example

Live Demo

var buf = new Buffer('Simply Easy Learning');

var json = buf.toJSON(buf);

console.log(json);

When the above program is executed, it produces the following result −

{ type: 'Buffer',

data:

[

83,

105,

109,

112,

108,

121,

32,

69,

97,

115,

121,

32,

76,

101,

97,

114,

110,

105,

110,

103

]

}

**Node js Streams**

**What are Streams?**

Streams are objects that let you read data from a source or write data to a destination in continuous fashion. In Node.js, there are four types of streams −

Readable − Stream which is used for read operation.

Writable − Stream which is used for write operation.

Duplex − Stream which can be used for both read and write operation.

Transform − A type of duplex stream where the output is computed based on input.

Each type of Stream is an EventEmitter instance and throws several events at different instance of times. For example, some of the commonly used events are −

data − This event is fired when there is data is available to read.

end − This event is fired when there is no more data to read.

error − This event is fired when there is any error receiving or writing data.

finish − This event is fired when all the data has been flushed to underlying system.

This tutorial provides a basic understanding of the commonly used operations on Streams.

**Reading from a Stream**

Create a text file named input.txt having the following content −

Tutorials Point is giving self learning content

to teach the world in simple and easy way!!!!!

Create a js file named main.js with the following code −

var fs = require("fs");

var data = '';

// Create a readable stream

var readerStream = fs.createReadStream('input.txt');

// Set the encoding to be utf8.

readerStream.setEncoding('UTF8');

// Handle stream events --> data, end, and error

readerStream.on('data', function(chunk) {

data += chunk;

});

readerStream.on('end',function() {

console.log(data);

});

readerStream.on('error', function(err) {

console.log(err.stack);

});

console.log("Program Ended");

Now run the main.js to see the result −

$ node main.js

Verify the Output.

Program Ended

Tutorials Point is giving self learning content

to teach the world in simple and easy way!!!!!

**Writing to a Stream**

Create a js file named main.js with the following code −

Live Demo

var fs = require("fs");

var data = 'Simply Easy Learning';

// Create a writable stream

var writerStream = fs.createWriteStream('output.txt');

// Write the data to stream with encoding to be utf8

writerStream.write(data,'UTF8');

// Mark the end of file

writerStream.end();

// Handle stream events --> finish, and error

writerStream.on('finish', function() {

console.log("Write completed.");

});

writerStream.on('error', function(err) {

console.log(err.stack);

});

console.log("Program Ended");

Now run the main.js to see the result −

$ node main.js

Verify the Output.

Program Ended

Write completed.

Now open output.txt created in your current directory; it should contain the following

Simply Easy Learning

**Piping the Streams**

Piping is a mechanism where we provide the output of one stream as the input to another stream. It is normally used to get data from one stream and to pass the output of that stream to another stream. There is no limit on piping operations. Now we'll show a piping example for reading from one file and writing it to another file.

Create a js file named main.js with the following code −

var fs = require("fs");

// Create a readable stream

var readerStream = fs.createReadStream('input.txt');

// Create a writable stream

var writerStream = fs.createWriteStream('output.txt');

// Pipe the read and write operations

// read input.txt and write data to output.txt

readerStream.pipe(writerStream);

console.log("Program Ended");

Now run the main.js to see the result −

$ node main.js

Verify the Output.

Program Ended

Open output.txt created in your current directory; it should contain the following −

Tutorials Point is giving self learning content

to teach the world in simple and easy way!!!!!

**Chaining the Streams**

Chaining is a mechanism to connect the output of one stream to another stream and create a chain of multiple stream operations. It is normally used with piping operations. Now we'll use piping and chaining to first compress a file and then decompress the same.

Create a js file named main.js with the following code −

var fs = require("fs");

var zlib = require('zlib');

// Compress the file input.txt to input.txt.gz

fs.createReadStream('input.txt')

.pipe(zlib.createGzip())

.pipe(fs.createWriteStream('input.txt.gz'));

console.log("File Compressed.");

Now run the main.js to see the result −

$ node main.js

Verify the Output.

File Compressed.

You will find that input.txt has been compressed and it created a file input.txt.gz in the current directory. Now let's try to decompress the same file using the following code −

var fs = require("fs");

var zlib = require('zlib');

// Decompress the file input.txt.gz to input.txt

fs.createReadStream('input.txt.gz')

.pipe(zlib.createGunzip())

.pipe(fs.createWriteStream('input.txt'));

console.log("File Decompressed.");

Now run the main.js to see the result −

$ node main.js

Verify the Output.

File Decompressed.

**Nodejs File system**

**Create a Directory**

Syntax

Following is the syntax of the method to create a directory −

fs.mkdir(path[, mode], callback)

Parameters

Here is the description of the parameters used −

path − This is the directory name including path.

mode − This is the directory permission to be set. Defaults to 0777.

callback − This is the callback function No arguments other than a possible exception are given to the completion callback.

Example

Let us create a js file named main.js having the following code −

var fs = require("fs");

console.log("Going to create directory /tmp/test");

fs.mkdir('/tmp/test',function(err) {

if (err) {

return console.error(err);

}

console.log("Directory created successfully!");

});

Now run the main.js to see the result −

$ node main.js

Verify the Output.

Going to create directory /tmp/test

Directory created successfully!

**Read a Directory**

Syntax

Following is the syntax of the method to read a directory −

fs.readdir(path, callback)

Parameters

Here is the description of the parameters used −

path − This is the directory name including path.

callback − This is the callback function which gets two arguments (err, files) where files is an array of the names of the files in the directory excluding '.' and '..'.

Example

Let us create a js file named main.js having the following code −

var fs = require("fs");

console.log("Going to read directory /tmp");

fs.readdir("/tmp/",function(err, files) {

if (err) {

return console.error(err);

}

files.forEach( function (file) {

console.log( file );

});

});

Now run the main.js to see the result −

$ node main.js

Verify the Output.

Going to read directory /tmp

ccmzx99o.out

ccyCSbkF.out

employee.ser

hsperfdata\_apache

test

test.txt

**Remove a Directory**

Syntax

Following is the syntax of the method to remove a directory −

fs.rmdir(path, callback)

Parameters

Here is the description of the parameters used −

path − This is the directory name including path.

callback − This is the callback function No arguments other than a possible exception are given to the completion callback.

Example

Let us create a js file named main.js having the following code −

var fs = require("fs");

console.log("Going to delete directory /tmp/test");

fs.rmdir("/tmp/test",function(err) {

if (err) {

return console.error(err);

}

console.log("Going to read directory /tmp");

fs.readdir("/tmp/",function(err, files) {

if (err) {

return console.error(err);

}

files.forEach( function (file) {

console.log( file );

});

});

});

Now run the main.js to see the result −

$ node main.js

Verify the Output.

Going to read directory /tmp

ccmzx99o.out

ccyCSbkF.out

employee.ser

hsperfdata\_apache

test.txt

--------------------------------------------------------------------------------------------------