虽然JavaScript对Unicode支持的很友好，但是对于二进制数据却没有操作的公开方法。比如当你处理TCP流或文件流时，就需要处理一些OCTET流，在Nodejs中就提供了处理这些二进制流的方案。在nodejs中将原始数据存储在一个Buffer类中，这个有点像一个保存了数字类型的的数组，但有单独于V8堆，Buffer不能够重新分配内存大小，如果要使用他将需要通过require('buffer')引入。

将Buffer和JavaScript的字符串之间进行转换，还需要进行字符串编码，以下就是在nodejs中能够支持的编码类型：

* 'ascii' - ASCII，7字节保存，运算速度最快。

注：如果将一个字符串转换为Buffer，这个编码将空字符('\0'或'\u0000')转换为0x20（空格符），如果想把他转换为0x00，就需要使用UTF8编码。

* 'utf8' - 多字节Unicode字符编码 ，文档常用UTF-8.
* 'utf16le'（或'ucs2'） - 2 or 4 bytes, little endian encoded Unicode characters. Surrogate pairs (U+10000 to U+10FFFF) are supported.
* 'base64' - Base64编码。
* 'binary' - 二进制数据，如果直接转换将根据前面8字节进行编码。
* 'hex' - 16进制字符，将每个字节转换为两位16进制的字符串。

A Buffer object can also be used with typed arrays. The buffer object is cloned to an ArrayBuffer that is used as the backing store for the typed array. The memory of the buffer and the ArrayBuffer is not shared.

NOTE: Node.js v0.8 simply retained a reference to the buffer in array.buffer instead of cloning it.

While more efficient, it introduces subtle incompatibilities with the typed arrays specification.ArrayBuffer#slice() makes a copy of the slice while Buffer#slice() creates a view.

Class: Buffer

The Buffer class is a global type for dealing with binary data directly. It can be constructed in a variety of ways.

new Buffer(size)

* size Number

Allocates a new buffer of size octets.

new Buffer(array)

* array Array

Allocates a new buffer using an array of octets.

new Buffer(str, [encoding])

* str String - string to encode.
* encoding String - encoding to use, Optional.

Allocates a new buffer containing the given str. encoding defaults to 'utf8'.

Class Method: Buffer.isEncoding(encoding)

* encoding String The encoding string to test

Returns true if the encoding is a valid encoding argument, or false otherwise.

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

* string String - data to be written to buffer
* offset Number, Optional, Default: 0
* length Number, Optional, Default: buffer.length - offset
* encoding String, Optional, Default: 'utf8'

Writes string to the buffer at offset using the given encoding. offset defaults to 0, encoding defaults to'utf8'. length is the number of bytes to write. Returns number of octets written. If buffer did not contain enough space to fit the entire string, it will write a partial amount of the string. length defaults tobuffer.length - offset. The method will not write partial characters.

buf = new Buffer(256); len = buf.write('\u00bd + \u00bc = \u00be', 0); console.log(len + " bytes: " + buf.toString('utf8', 0, len));

The number of characters written (which may be different than the number of bytes written) is set inBuffer.\_charsWritten and will be overwritten the next time buf.write() is called.

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

* encoding String, Optional, Default: 'utf8'
* start Number, Optional, Default: 0
* end Number, Optional, Default: buffer.length

Decodes and returns a string from buffer data encoded with encoding (defaults to 'utf8') beginning at start(defaults to 0) and ending at end (defaults to buffer.length).

See buffer.write() example, above.

buf.toJSON()

Returns a JSON-representation of the Buffer instance, which is identical to the output for JSON Arrays.JSON.stringify implicitly calls this function when stringifying a Buffer instance.

Example:

var buf = new Buffer('test'); var json = JSON.stringify(buf); console.log(json); // '[116,101,115,116]' var copy = new Buffer(JSON.parse(json)); console.log(copy); // <Buffer 74 65 73 74>

buf[index]

Get and set the octet at index. The values refer to individual bytes, so the legal range is between 0x00 and0xFF hex or 0 and 255.

Example: copy an ASCII string into a buffer, one byte at a time:

str = "node.js"; buf = new Buffer(str.length); for (var i = 0; i < str.length ; i++) { buf[i] = str.charCodeAt(i); } console.log(buf); // node.js

Class Method: Buffer.isBuffer(obj)

* obj Object
* Return: Boolean

Tests if obj is a Buffer.

Class Method: Buffer.byteLength(string, [encoding])

* string String
* encoding String, Optional, Default: 'utf8'
* Return: Number

Gives the actual byte length of a string. encoding defaults to 'utf8'. This is not the same asString.prototype.length since that returns the number of *characters* in a string.

Example:

str = '\u00bd + \u00bc = \u00be'; console.log(str + ": " + str.length + " characters, " + Buffer.byteLength(str, 'utf8') + " bytes"); // ½ + ¼ = ¾: 9 characters, 12 bytes

Class Method: Buffer.concat(list, [totalLength])

* list Array List of Buffer objects to concat
* totalLength Number Total length of the buffers when concatenated

Returns a buffer which is the result of concatenating all the buffers in the list together.

If the list has no items, or if the totalLength is 0, then it returns a zero-length buffer.

If the list has exactly one item, then the first item of the list is returned.

If the list has more than one item, then a new Buffer is created.

If totalLength is not provided, it is read from the buffers in the list. However, this adds an additional loop to the function, so it is faster to provide the length explicitly.

buf.length

* Number

The size of the buffer in bytes. Note that this is not necessarily the size of the contents. length refers to the amount of memory allocated for the buffer object. It does not change when the contents of the buffer are changed.

buf = new Buffer(1234); console.log(buf.length); buf.write("some string", 0, "ascii"); console.log(buf.length); // 1234 // 1234

buf.copy(targetBuffer, [targetStart], [sourceStart], [sourceEnd])

* targetBuffer Buffer object - Buffer to copy into
* targetStart Number, Optional, Default: 0
* sourceStart Number, Optional, Default: 0
* sourceEnd Number, Optional, Default: buffer.length

Does copy between buffers. The source and target regions can be overlapped. targetStart and sourceStartdefault to 0. sourceEnd defaults to buffer.length.

All values passed that are undefined/NaN or are out of bounds are set equal to their respective defaults.

Example: build two Buffers, then copy buf1 from byte 16 through byte 19 into buf2, starting at the 8th byte inbuf2.

buf1 = new Buffer(26); buf2 = new Buffer(26); for (var i = 0 ; i < 26 ; i++) { buf1[i] = i + 97; // 97 is ASCII a buf2[i] = 33; // ASCII ! } buf1.copy(buf2, 8, 16, 20); console.log(buf2.toString('ascii', 0, 25)); // !!!!!!!!qrst!!!!!!!!!!!!!

buf.slice([start], [end])

* start Number, Optional, Default: 0
* end Number, Optional, Default: buffer.length

Returns a new buffer which references the same memory as the old, but offset and cropped by the start(defaults to 0) and end (defaults to buffer.length) indexes. Negative indexes start from the end of the buffer.

**Modifying the new buffer slice will modify memory in the original buffer!**

Example: build a Buffer with the ASCII alphabet, take a slice, then modify one byte from the original Buffer.

var buf1 = new Buffer(26); for (var i = 0 ; i < 26 ; i++) { buf1[i] = i + 97; // 97 is ASCII a } var buf2 = buf1.slice(0, 3); console.log(buf2.toString('ascii', 0, buf2.length)); buf1[0] = 33; console.log(buf2.toString('ascii', 0, buf2.length)); // abc // !bc

buf.readUInt8(offset, [noAssert])

* offset Number
* noAssert Boolean, Optional, Default: false
* Return: Number

Reads an unsigned 8 bit integer from the buffer at the specified offset.

Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.

Example:

var buf = new Buffer(4); buf[0] = 0x3; buf[1] = 0x4; buf[2] = 0x23; buf[3] = 0x42; for (ii = 0; ii < buf.length; ii++) { console.log(buf.readUInt8(ii)); } // 0x3 // 0x4 // 0x23 // 0x42

buf.readUInt16LE(offset, [noAssert])

buf.readUInt16BE(offset, [noAssert])

* offset Number
* noAssert Boolean, Optional, Default: false
* Return: Number

Reads an unsigned 16 bit integer from the buffer at the specified offset with specified endian format.

Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.

Example:

var buf = new Buffer(4); buf[0] = 0x3; buf[1] = 0x4; buf[2] = 0x23; buf[3] = 0x42; console.log(buf.readUInt16BE(0)); console.log(buf.readUInt16LE(0)); console.log(buf.readUInt16BE(1)); console.log(buf.readUInt16LE(1)); console.log(buf.readUInt16BE(2)); console.log(buf.readUInt16LE(2)); // 0x0304 // 0x0403 // 0x0423 // 0x2304 // 0x2342 // 0x4223

buf.readUInt32LE(offset, [noAssert])

buf.readUInt32BE(offset, [noAssert])

* offset Number
* noAssert Boolean, Optional, Default: false
* Return: Number

Reads an unsigned 32 bit integer from the buffer at the specified offset with specified endian format.

Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.

Example:

var buf = new Buffer(4); buf[0] = 0x3; buf[1] = 0x4; buf[2] = 0x23; buf[3] = 0x42; console.log(buf.readUInt32BE(0)); console.log(buf.readUInt32LE(0)); // 0x03042342 // 0x42230403

buf.readInt8(offset, [noAssert])

* offset Number
* noAssert Boolean, Optional, Default: false
* Return: Number

Reads a signed 8 bit integer from the buffer at the specified offset.

Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.

Works as buffer.readUInt8, except buffer contents are treated as two's complement signed values.

buf.readInt16LE(offset, [noAssert])

buf.readInt16BE(offset, [noAssert])

* offset Number
* noAssert Boolean, Optional, Default: false
* Return: Number

Reads a signed 16 bit integer from the buffer at the specified offset with specified endian format.

Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.

Works as buffer.readUInt16\*, except buffer contents are treated as two's complement signed values.

buf.readInt32LE(offset, [noAssert])

buf.readInt32BE(offset, [noAssert])

* offset Number
* noAssert Boolean, Optional, Default: false
* Return: Number

Reads a signed 32 bit integer from the buffer at the specified offset with specified endian format.

Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.

Works as buffer.readUInt32\*, except buffer contents are treated as two's complement signed values.

buf.readFloatLE(offset, [noAssert])

buf.readFloatBE(offset, [noAssert])

* offset Number
* noAssert Boolean, Optional, Default: false
* Return: Number

Reads a 32 bit float from the buffer at the specified offset with specified endian format.

Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.

Example:

var buf = new Buffer(4); buf[0] = 0x00; buf[1] = 0x00; buf[2] = 0x80; buf[3] = 0x3f; console.log(buf.readFloatLE(0)); // 0x01

buf.readDoubleLE(offset, [noAssert])

buf.readDoubleBE(offset, [noAssert])

* offset Number
* noAssert Boolean, Optional, Default: false
* Return: Number

Reads a 64 bit double from the buffer at the specified offset with specified endian format.

Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.

Example:

var buf = new Buffer(8); buf[0] = 0x55; buf[1] = 0x55; buf[2] = 0x55; buf[3] = 0x55; buf[4] = 0x55; buf[5] = 0x55; buf[6] = 0xd5; buf[7] = 0x3f; console.log(buf.readDoubleLE(0)); // 0.3333333333333333

buf.writeUInt8(value, offset, [noAssert])

* value Number
* offset Number
* noAssert Boolean, Optional, Default: false

Writes value to the buffer at the specified offset. Note, value must be a valid unsigned 8 bit integer.

Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.

Example:

var buf = new Buffer(4); buf.writeUInt8(0x3, 0); buf.writeUInt8(0x4, 1); buf.writeUInt8(0x23, 2); buf.writeUInt8(0x42, 3); console.log(buf); // <Buffer 03 04 23 42>

buf.writeUInt16LE(value, offset, [noAssert])

buf.writeUInt16BE(value, offset, [noAssert])

* value Number
* offset Number
* noAssert Boolean, Optional, Default: false

Writes value to the buffer at the specified offset with specified endian format. Note, value must be a valid unsigned 16 bit integer.

Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.

Example:

var buf = new Buffer(4); buf.writeUInt16BE(0xdead, 0); buf.writeUInt16BE(0xbeef, 2); console.log(buf); buf.writeUInt16LE(0xdead, 0); buf.writeUInt16LE(0xbeef, 2); console.log(buf); // <Buffer de ad be ef> // <Buffer ad de ef be>

buf.writeUInt32LE(value, offset, [noAssert])

buf.writeUInt32BE(value, offset, [noAssert])

* value Number
* offset Number
* noAssert Boolean, Optional, Default: false

Writes value to the buffer at the specified offset with specified endian format. Note, value must be a valid unsigned 32 bit integer.

Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.

Example:

var buf = new Buffer(4); buf.writeUInt32BE(0xfeedface, 0); console.log(buf); buf.writeUInt32LE(0xfeedface, 0); console.log(buf); // <Buffer fe ed fa ce> // <Buffer ce fa ed fe>

buf.writeInt8(value, offset, [noAssert])

* value Number
* offset Number
* noAssert Boolean, Optional, Default: false

Writes value to the buffer at the specified offset. Note, value must be a valid signed 8 bit integer.

Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.

Works as buffer.writeUInt8, except value is written out as a two's complement signed integer into buffer.

buf.writeInt16LE(value, offset, [noAssert])

buf.writeInt16BE(value, offset, [noAssert])

* value Number
* offset Number
* noAssert Boolean, Optional, Default: false

Writes value to the buffer at the specified offset with specified endian format. Note, value must be a valid signed 16 bit integer.

Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.

Works as buffer.writeUInt16\*, except value is written out as a two's complement signed integer intobuffer.

buf.writeInt32LE(value, offset, [noAssert])

buf.writeInt32BE(value, offset, [noAssert])

* value Number
* offset Number
* noAssert Boolean, Optional, Default: false

Writes value to the buffer at the specified offset with specified endian format. Note, value must be a valid signed 32 bit integer.

Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.

Works as buffer.writeUInt32\*, except value is written out as a two's complement signed integer intobuffer.

buf.writeFloatLE(value, offset, [noAssert])

buf.writeFloatBE(value, offset, [noAssert])

* value Number
* offset Number
* noAssert Boolean, Optional, Default: false

Writes value to the buffer at the specified offset with specified endian format. Note, behavior is unspecified ifvalue is not a 32 bit float.

Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.

Example:

var buf = new Buffer(4); buf.writeFloatBE(0xcafebabe, 0); console.log(buf); buf.writeFloatLE(0xcafebabe, 0); console.log(buf); // <Buffer 4f 4a fe bb> // <Buffer bb fe 4a 4f>

buf.writeDoubleLE(value, offset, [noAssert])

buf.writeDoubleBE(value, offset, [noAssert])

* value Number
* offset Number
* noAssert Boolean, Optional, Default: false

Writes value to the buffer at the specified offset with specified endian format. Note, value must be a valid 64 bit double.

Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.

Example:

var buf = new Buffer(8); buf.writeDoubleBE(0xdeadbeefcafebabe, 0); console.log(buf); buf.writeDoubleLE(0xdeadbeefcafebabe, 0); console.log(buf); // <Buffer 43 eb d5 b7 dd f9 5f d7> // <Buffer d7 5f f9 dd b7 d5 eb 43>

buf.fill(value, [offset], [end])

* value
* offset Number, Optional
* end Number, Optional

Fills the buffer with the specified value. If the offset (defaults to 0) and end (defaults to buffer.length) are not given it will fill the entire buffer.

var b = new Buffer(50); b.fill("h");

buffer.INSPECT\_MAX\_BYTES

* Number, Default: 50

How many bytes will be returned when buffer.inspect() is called. This can be overridden by user modules.

Note that this is a property on the buffer module returned by require('buffer'), not on the Buffer global, or a buffer instance.

Class: SlowBuffer

This class is primarily for internal use. JavaScript programs should use Buffer instead of using SlowBuffer.

In order to avoid the overhead of allocating many C++ Buffer objects for small blocks of memory in the lifetime of a server, Node allocates memory in 8Kb (8192 byte) chunks. If a buffer is smaller than this size, then it will be backed by a parent SlowBuffer object. If it is larger than this, then Node will allocate a SlowBuffer slab for it directly.