Node.js GET/POST请求 Node.js Stream(流)

Node.js Buffer(缓冲区)

JavaScript 语言自身只有字符串数据类型，没有二进制数据类型。

但在处理像TCP流或文件流时，必须使用到二进制数据。因此在 Node.js中，定义了一个 Buffer 类，该类用来创建一个专门存放二进制数据的缓存区。

在 Node.js 中，Buffer 类是随 Node 内核一起发布的核心库。Buffer 库为 Node.js 带来了一种存储原始数据的方法，可以让 Node.js 处理二进制数据，每当需要在 Node.js 中处理I/O操作中移动的数据时，就有可能使用 Buffer 库。原始数据存储在 Buffer 类的实例中。一个 Buffer 类似于一个整数数组，但它对应于 V8 堆内存之外的一块原始内存。

> *在v6.0之前创建Buffer对象直接使用new Buffer()构造函数来创建对象实例，但是Buffer对内存的权限操作相比很大，可以直接捕获一些敏感信息，所以在v6.0以后，官方文档里面建议使用* ***`Buffer.from()`*** *接口去创建Buffer对象。*

Buffer 与字符编码

Buffer 实例一般用于表示编码字符的序列，比如 UTF-8 、 UCS2 、 Base64 、或十六进制编码的数据。 通过使用显式的字符编码，就可以在 Buffer 实例与普通的 JavaScript 字符串之间进行相互转换。

```
const buf = Buffer.from('runoob', 'ascii');

// 输出 72756e6f6f62
console.log(buf.toString('hex'));

// 输出 cnVub29i
console.log(buf.toString('base64'));
```

Node.js 目前支持的字符编码包括：

- **ascii** - 仅支持 7 位 ASCII 数据。如果设置去掉高位的话，这种编码是非常快的。
- **utf8** - 多字节编码的 Unicode 字符。许多网页和其他文档格式都使用 UTF-8 。
- **utf16le** - 2 或 4 个字节，小字节序编码的 Unicode 字符。支持代理对（U+10000 至 U+10FFFF）。
- **ucs2** - **utf16le** 的别名。
- **base64** - Base64 编码。
- **latin1** - 一种把 **Buffer** 编码成一字节编码的字符串的方式。
- **binary** - **latin1** 的别名。

- **hex** - 将每个字节编码为两个十六进制字符。

创建 Buffer 类

Buffer 提供了以下 API 来创建 Buffer 类：

- **Buffer.alloc(size[, fill[, encoding]])：** 返回一个指定大小的 Buffer 实例，如果没有设置 fill，则默认填满 0
- **Buffer.allocUnsafe(size)：** 返回一个指定大小的 Buffer 实例，但是它不会被初始化，所以它可能包含敏感的数据
- **Buffer.allocUnsafeSlow(size)**
- **Buffer.from(array)：** 返回一个被 array 的值初始化的新的 Buffer 实例（传入的 array 的元素只能是数字，不然就会自动被 0 覆盖）
- **Buffer.from(arrayBuffer[, byteOffset[, length]])：** 返回一个新建的与给定的 ArrayBuffer 共享同一内存的 Buffer。
- **Buffer.from(buffer)：** 复制传入的 Buffer 实例的数据，并返回一个新的 Buffer 实例
- **Buffer.from(string[, encoding])：** 返回一个被 string 的值初始化的新的 Buffer 实例

```
// 创建一个长度为 10、且用 0 填充的 Buffer。
const buf1 = Buffer.alloc(10);

// 创建一个长度为 10、且用 0x1 填充的 Buffer。
const buf2 = Buffer.alloc(10, 1);

// 创建一个长度为 10、且未初始化的 Buffer。
// 这个方法比调用 Buffer.alloc() 更快，
// 但返回的 Buffer 实例可能包含旧数据，
// 因此需要使用 fill() 或 write() 重写。
const buf3 = Buffer.allocUnsafe(10);

// 创建一个包含 [0x1, 0x2, 0x3] 的 Buffer。
const buf4 = Buffer.from([1, 2, 3]);

// 创建一个包含 UTF-8 字节 [0x74, 0xc3, 0xa9, 0x73, 0x74] 的 Buffer。
const buf5 = Buffer.from('tést');

// 创建一个包含 Latin-1 字节 [0x74, 0xe9, 0x73, 0x74] 的 Buffer。
const buf6 = Buffer.from('tést', 'latin1');
```

写入缓冲区

语法

写入 Node 缓冲区的语法如下所示：

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

参数

参数描述如下：

- **string** - 写入缓冲区的字符串。
- **offset** - 缓冲区开始写入的索引值，默认为 0 。
- **length** - 写入的字节数，默认为 buffer.length
- **encoding** - 使用的编码。默认为 'utf8' 。

根据 encoding 的字符编码写入 string 到 buf 中的 offset 位置。 length 参数是写入的字节数。 如果 buf 没有足够的空间保存整个字符串，则只会写入 string 的一部分。 只部分解码的字符不会被写入。

返回值

返回实际写入的大小。如果 buffer 空间不足， 则只会写入部分字符串。

实例

```
buf = Buffer.alloc(256);
len = buf.write("www.runoob.com");

console.log("写入字节数 : "+  len);
```

执行以上代码，输出结果为：

```
$node main.js
写入字节数 : 14
```

从缓冲区读取数据

语法

读取 Node 缓冲区数据的语法如下所示：

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

参数

参数描述如下：

- **encoding** - 使用的编码。默认为 'utf8' 。
- **start** - 指定开始读取的索引位置，默认为 0。
- **end** - 结束位置，默认为缓冲区的末尾。

返回值

解码缓冲区数据并使用指定的编码返回字符串。

实例

```
buf = Buffer.alloc(26);
for (var i = 0 ; i < 26 ; i++) {
  buf[i] = i + 97;
}

console.log( buf.toString('ascii'));       // 输出: abcdefghijklmnopqrstuvwxyz
console.log( buf.toString('ascii',0,5));   // 输出: abcde
console.log( buf.toString('utf8',0,5));    // 输出: abcde
console.log( buf.toString(undefined,0,5)); // 使用 'utf8' 编码, 并输出: abcde
```

执行以上代码，输出结果为：

```
$ node main.js
abcdefghijklmnopqrstuvwxyz
abcde
abcde
abcde
```

将 Buffer 转换为 JSON 对象

语法

将 Node Buffer 转换为 JSON 对象的函数语法格式如下：

```
buf.toJSON()
```

当字符串化一个 Buffer 实例时，JSON.stringify() 会隐式地调用该 `toJSON()`。

返回值

返回 JSON 对象。

实例

```
const buf = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5]);
const json = JSON.stringify(buf);

// 输出: {"type":"Buffer","data":[1,2,3,4,5]}
console.log(json);

const copy = JSON.parse(json, (key, value) => {
```

```
  return value && value.type === 'Buffer' ?
    Buffer.from(value.data) :
    value;
});

// 输出: <Buffer 01 02 03 04 05>
console.log(copy);
```

执行以上代码，输出结果为：

```
{"type":"Buffer","data":[1,2,3,4,5]}
<Buffer 01 02 03 04 05>
```

缓冲区合并

语法

Node 缓冲区合并的语法如下所示：

```
Buffer.concat(list[, totalLength])
```

参数

参数描述如下：

- **list** - 用于合并的 Buffer 对象数组列表。
- **totalLength** - 指定合并后Buffer对象的总长度。

返回值

返回一个多个成员合并的新 Buffer 对象。

实例

```
var buffer1 = Buffer.from(('菜鸟教程'));
var buffer2 = Buffer.from(('www.runoob.com'));
var buffer3 = Buffer.concat([buffer1,buffer2]);
console.log("buffer3 内容: " + buffer3.toString());
```

执行以上代码，输出结果为：

```
buffer3 内容: 菜鸟教程www.runoob.com
```

缓冲区比较

语法

Node Buffer 比较的函数语法如下所示, 该方法在 Node.js v0.12.2 版本引入：

```
buf.compare(otherBuffer);
```

参数

参数描述如下：

- **otherBuffer** - 与 **buf** 对象比较的另外一个 Buffer 对象。

返回值

返回一个数字，表示 **buf** 在 **otherBuffer** 之前，之后或相同。

实例

```
var buffer1 = Buffer.from('ABC');
var buffer2 = Buffer.from('ABCD');
var result = buffer1.compare(buffer2);

if(result < 0) {
   console.log(buffer1 + " 在 " + buffer2 + "之前");
}else if(result == 0){
   console.log(buffer1 + " 与 " + buffer2 + "相同");
}else {
   console.log(buffer1 + " 在 " + buffer2 + "之后");
}
```

执行以上代码，输出结果为：

```
ABC在ABCD之前
```

拷贝缓冲区

语法

Node 缓冲区拷贝语法如下所示：

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

参数

参数描述如下：

- **targetBuffer** - 要拷贝的 Buffer 对象。

- **targetStart** - 数字, 可选, 默认: 0
- **sourceStart** - 数字, 可选, 默认: 0
- **sourceEnd** - 数字, 可选, 默认: buffer.length

返回值

没有返回值。

实例

```
var buf1 = Buffer.from('abcdefghijkl');
var buf2 = Buffer.from('RUNOOB');

//将 buf2 插入到 buf1 指定位置上
buf2.copy(buf1, 2);

console.log(buf1.toString());
```

执行以上代码，输出结果为：

```
abRUNOOBijkl
```

缓冲区裁剪

Node 缓冲区裁剪语法如下所示：

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

参数

参数描述如下：

- **start** - 数字, 可选, 默认: 0
- **end** - 数字, 可选, 默认: buffer.length

返回值

返回一个新的缓冲区，它和旧缓冲区指向同一块内存，但是从索引 start 到 end 的位置剪切。

实例

```
var buffer1 = Buffer.from('runoob');
// 剪切缓冲区
var buffer2 = buffer1.slice(0,2);
console.log("buffer2 content: " + buffer2.toString());
```

执行以上代码，输出结果为：

```
buffer2 content: ru
```

缓冲区长度

语法

Node 缓冲区长度计算语法如下所示：

```
buf.length;
```

返回值

返回 Buffer 对象所占据的内存长度。

实例

```
var buffer = Buffer.from('www.runoob.com');
//  缓冲区长度
console.log("buffer length: " + buffer.length);
```

执行以上代码，输出结果为：

```
buffer length: 14
```

方法参考手册

以下列出了 Node.js Buffer 模块常用的方法（注意有些方法在旧版本是没有的）：

序号	方法 & 描述
1	**new Buffer(size)** 分配一个新的 size 大小单位为8位字节的 buffer。 注意, size 必须小于 kMaxLength，否则，将会抛出异常 RangeError。废弃的: 使用 Buffer.alloc() 代替（或 Buffer.allocUnsafe()）。
2	**new Buffer(buffer)** 拷贝参数 buffer 的数据到 Buffer 实例。废弃的: 使用 Buffer.from(buffer) 代替。
3	**new Buffer(str[, encoding])** 分配一个新的 buffer ，其中包含着传入的 str 字符串。 encoding 编码方式默认为 'utf8'。 废弃的: 使用 Buffer.from(string[, encoding]) 代替。

4 **buf.length**

返回这个 buffer 的 bytes 数。注意这未必是 buffer 里面内容的大小。length 是 buffer 对象所分配的内存数，它不会随着这个 buffer 对象内容的改变而改变。

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

根据参数 offset 偏移量和指定的 encoding 编码方式，将参数 string 数据写入buffer。 offset 偏移量默认值是 0, encoding 编码方式默认是 utf8。 length 长度是将要写入的字符串的 bytes 大小。 返回 number 类型，表示写入了多少 8 位字节流。如果 buffer 没有足够的空间来放整个 string，它将只会只写入部分字符串。 length 默认是 buffer.length - offset。 这个方法不会出现写入部分字符。

6 **buf.writeUIntLE(value, offset, byteLength[, noAssert])**

将 value 写入到 buffer 里， 它由 offset 和 byteLength 决定，最高支持 48 位无符号整数，小端对齐，例如：

```
const buf = Buffer.allocUnsafe(6);

buf.writeUIntLE(0x1234567890ab, 0, 6);

// 输出: <Buffer ab 90 78 56 34 12>
console.log(buf);
```

noAssert 值为 true 时，不再验证 value 和 offset 的有效性。 默认是 false。

7 **buf.writeUIntBE(value, offset, byteLength[, noAssert])**

将 value 写入到 buffer 里， 它由 offset 和 byteLength 决定，最高支持 48 位无符号整数，大端对齐。noAssert 值为 true 时，不再验证 value 和 offset 的有效性。 默认是 false。

```
const buf = Buffer.allocUnsafe(6);

buf.writeUIntBE(0x1234567890ab, 0, 6);

// 输出: <Buffer 12 34 56 78 90 ab>
console.log(buf);
```

8 **buf.writeIntLE(value, offset, byteLength[, noAssert])**

将value 写入到 buffer 里， 它由offset 和 byteLength 决定，最高支持48位有符号整数，小端对齐。noAssert 值为 true 时，不再验证 value 和 offset 的有效性。 默认是 false。

9 **buf.writeIntBE(value, offset, byteLength[, noAssert])**

将value 写入到 buffer 里， 它由offset 和 byteLength 决定，最高支持48位有符号整数，大端对齐。noAssert 值为 true 时，不再验证 value 和 offset 的有效性。 默认是 false。

10 **buf.readUIntLE(offset, byteLength[, noAssert])**

	支持读取 48 位以下的无符号数字，小端对齐。noAssert 值为 true 时， offset 不再验证是否超过 buffer 的长度，默认为 false。
11	**buf.readUIntBE(offset, byteLength[, noAssert])** 支持读取 48 位以下的无符号数字，大端对齐。noAssert 值为 true 时， offset 不再验证是否超过 buffer 的长度，默认为 false。
12	**buf.readIntLE(offset, byteLength[, noAssert])** 支持读取 48 位以下的有符号数字，小端对齐。noAssert 值为 true 时， offset 不再验证是否超过 buffer 的长度，默认为 false。
13	**buf.readIntBE(offset, byteLength[, noAssert])** 支持读取 48 位以下的有符号数字，大端对齐。noAssert 值为 true 时， offset 不再验证是否超过 buffer 的长度，默认为 false。
14	**buf.toString([encoding[, start[, end]]])** 根据 encoding 参数（默认是 'utf8'）返回一个解码过的 string 类型。还会根据传入的参数 start (默认是 0) 和 end (默认是 buffer.length)作为取值范围。
15	**buf.toJSON()** 将 Buffer 实例转换为 JSON 对象。
16	**buf[index]** 获取或设置指定的字节。返回值代表一个字节，所以返回值的合法范围是十六进制0x00到0xFF 或者十进制0至 255。
17	**buf.equals(otherBuffer)** 比较两个缓冲区是否相等，如果是返回 true，否则返回 false。
18	**buf.compare(otherBuffer)** 比较两个 Buffer 对象，返回一个数字，表示 buf 在 otherBuffer 之前，之后或相同。
19	**buf.copy(targetBuffer[, targetStart[, sourceStart[, sourceEnd]]])** buffer 拷贝，源和目标可以相同。 targetStart 目标开始偏移和 sourceStart 源开始偏移默认都是 0。 sourceEnd 源结束位置偏移默认是源的长度 buffer.length 。
20	**buf.slice([start[, end]])** 剪切 Buffer 对象，根据 start(默认是 0) 和 end (默认是 buffer.length) 偏移和裁剪了索引。 负的索引是从 buffer 尾部开始计算的。
21	**buf.readUInt8(offset[, noAssert])** 根据指定的偏移量，读取一个无符号 8 位整数。若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 如果这样 offset 可能会超出buffer 的末尾。默认是 false。

22	**buf.readUInt16LE(offset[, noAssert])** 根据指定的偏移量，使用特殊的 endian 字节序格式读取一个无符号 16 位整数。若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出 buffer 的末尾。默认是 false。
23	**buf.readUInt16BE(offset[, noAssert])** 根据指定的偏移量，使用特殊的 endian 字节序格式读取一个无符号 16 位整数，大端对齐。若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出 buffer 的末尾。默认是 false。
24	**buf.readUInt32LE(offset[, noAssert])** 根据指定的偏移量，使用指定的 endian 字节序格式读取一个无符号 32 位整数，小端对齐。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出buffer 的末尾。默认是 false。
25	**buf.readUInt32BE(offset[, noAssert])** 根据指定的偏移量，使用指定的 endian 字节序格式读取一个无符号 32 位整数，大端对齐。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出buffer 的末尾。默认是 false。
26	**buf.readInt8(offset[, noAssert])** 根据指定的偏移量，读取一个有符号 8 位整数。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出 buffer 的末尾。默认是 false。
27	**buf.readInt16LE(offset[, noAssert])** 根据指定的偏移量，使用特殊的 endian 格式读取一个 有符号 16 位整数，小端对齐。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出 buffer 的末尾。默认是 false。
28	**buf.readInt16BE(offset[, noAssert])** 根据指定的偏移量，使用特殊的 endian 格式读取一个 有符号 16 位整数，大端对齐。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出 buffer 的末尾。默认是 false。
29	**buf.readInt32LE(offset[, noAssert])** 根据指定的偏移量，使用指定的 endian 字节序格式读取一个有符号 32 位整数，小端对齐。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出buffer 的末尾。默认是 false。
30	**buf.readInt32BE(offset[, noAssert])** 根据指定的偏移量，使用指定的 endian 字节序格式读取一个有符号 32 位整数，大端对齐。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出buffer 的末尾。默认是 false。
31	**buf.readFloatLE(offset[, noAssert])** 根据指定的偏移量，使用指定的 endian 字节序格式读取一个 32 位双浮点数，小端对齐。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出buffer的末尾。默认是 false。
32	**buf.readFloatBE(offset[, noAssert])**

	根据指定的偏移量，使用指定的 endian 字节序格式读取一个 32 位双浮点数，大端对齐。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出buffer的末尾。默认是 false。
33	**buf.readDoubleLE(offset[, noAssert])** 根据指定的偏移量，使用指定的 endian字节序格式读取一个 64 位双精度数，小端对齐。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出buffer 的末尾。默认是 false。
34	**buf.readDoubleBE(offset[, noAssert])** 根据指定的偏移量，使用指定的 endian字节序格式读取一个 64 位双精度数，大端对齐。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出buffer 的末尾。默认是 false。
35	**buf.writeUInt8(value, offset[, noAssert])** 根据传入的 offset 偏移量将 value 写入 buffer。注意：value 必须是一个合法的无符号 8 位整数。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则不要使用。默认是 false。
36	**buf.writeUInt16LE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个合法的无符号 16 位整数，小端对齐。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value 可能过大，或者 offset 可能会超出buffer的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。
37	**buf.writeUInt16BE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个合法的无符号 16 位整数，大端对齐。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value 可能过大，或者 offset 可能会超出buffer的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。
38	**buf.writeUInt32LE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式(LITTLE-ENDIAN:小字节序)将 value 写入buffer。注意：value 必须是一个合法的无符号 32 位整数，小端对齐。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着value 可能过大，或者offset可能会超出buffer的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。
39	**buf.writeUInt32BE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式(Big-Endian:大字节序)将 value 写入buffer。注意：value 必须是一个合法的有符号 32 位整数。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value 可能过大，或者offset可能会超出buffer的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。
40	**buf.writeInt8(value, offset[, noAssert])**

41	**buf.writeInt16LE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个合法的 signed 16 位整数。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false 。
42	**buf.writeInt16BE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个合法的 signed 16 位整数。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false 。
43	**buf.writeInt32LE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个合法的 signed 32 位整数。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。
44	**buf.writeInt32BE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个合法的 signed 32 位整数。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。
45	**buf.writeFloatLE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer 。注意：当 value 不是一个 32 位浮点数类型的值时，结果将是不确定的。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value可能过大，或者 offset 可能会超出 buffer 的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。
46	**buf.writeFloatBE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer 。注意：当 value 不是一个 32 位浮点数类型的值时，结果将是不确定的。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value可能过大，或者 offset 可能会超出 buffer 的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。
47	**buf.writeDoubleLE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个有效的 64 位double 类型的值。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾从而造成value被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。
48	**buf.writeDoubleBE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个有效的 64 位double 类型

	的值。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾从而造成value被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。
49	**buf.fill(value[, offset][, end])** 使用指定的 value 来填充这个 buffer。如果没有指定 offset (默认是 0) 并且 end (默认是 buffer.length) ，将会填充整个 buffer。

← Node.js GET/POST请求 Node.js Stream(流) →

2 篇笔记

写笔记

```
buf.compare(otherBuffer);
```

这个方法是按位比较的。`buffer1.compare(buffer2)`，这个方法是按位比较的。**buffer1** 的第一位比较 **buffer2** 的第一位，相等的话比较第二位以此类推直到得出结果。

举例：

```
var buffer1 = Buffer.from('ABCDEF99');
var buffer2 = Buffer.from('ABCDEF98765');
```

上面那个 **buffer1** 和 **buffer2** 比较的话，**result** 的结果是 **1**，前面的结果都是相等，直到比较 **9>8** 的时候出结果。按位逐一比较直到出结果。

那么说如果把 **buffer1** 设为 **12313213121** 这种纯数字呢，数字和字母比较的结果就是 **-1**。

曾经被狼咬 8个月前 (07-20)

```
buffer.slice();
```

裁剪功能返回的实际是原始缓存区 buffer 或者一部分，操作的是与原始 buffer 同一块内存区域。

```
// 裁剪
var buffer_origin = Buffer.from('runoob');
var buffer_slice = buffer_origin.slice(0,2);
console.log("buffer slice content: "+buffer_slice.toString());
console.log("buffer origin content: "+buffer_origin.toString());
buffer_slice.write("wirte"); // Write buffer slice

// 裁剪前与原始字符串的改变
console.log("buffer slice content: "+buffer_slice.toString());
console.log("buffer origin content: "+buffer_origin.toString());
```

输出：

```
buffer slice content: ru
buffer origin content: runoob
```

```
buffer slice content: wi
buffer origin content: winoob
```

可以看到对裁剪返回的 buffer 进行写操作同时，也对原始 buffer 进行了写操作。

martin 8个月前 (07-23)