

[首页](#) [HTML](#) [CSS](#) [JAVASCRIPT](#) [JQUERY](#) [BOOTSTRAP](#) [SQL](#) [MYSQL](#) [PHP](#) [PYTHON](#) [C](#) [C++](#)

Node.js 教程

- [Node.js 教程](#)
- [Node.js 安装配置](#)
- [Node.js 创建第一个应用](#)
- [NPM 使用介绍](#)
- [Node.js REPL](#)
- [Node.js 回调函数](#)
- [Node.js 事件循环](#)
- [Node.js EventEmitter](#)
- [Node.js Buffer](#)
- [Node.js Stream](#)
- [Node.js 模块系统](#)
- [Node.js 函数](#)
- [Node.js 路由](#)
- [Node.js 全局对象](#)
- [Node.js 常用工具](#)
- [Node.js 文件系统](#)
- [Node.js GET/POST 请求](#)
- [Node.js 工具模块](#)
- [Node.js Web 模块](#)
- [Node.js Express 框架](#)
- [Node.js RESTful API](#)
- [Node.js 多进程](#)
- [Node.js JXcore 打包](#)

[← Node.js EventEmitter](#)
[Node.js Stream →](#)

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。

创建 Buffer 类

Node Buffer 类可以通过多种方式来创建。

方法 1

创建长度为 10 字节的 Buffer 实例：

```
var buf = new Buffer(10);
```

方法 2

通过给定的数组创建 Buffer 实例：

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

方法 3

通过一个字符串来创建 Buffer 实例：

```
var buf = new Buffer("www.runoob.com", "utf-8");
```

utf-8 是默认的编码方式，此外它同样支持以下编码：“ascii”，“utf8”，“utf16le”，“ucs2”，“base64” 和 “hex”。

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])
```

关注微信



分类导航

[HTML / CSS](#)

[JavaScript](#)

[服务端](#)

[数据库](#)

[移动端](#)

[XML 教程](#)

[ASP.NET](#)

[Web Services](#)

[开发工具](#)

[网站建设](#)

Advertisement



反馈

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'。

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

console.log(buf.toString('ascii'));
```

// 输出: abcdefghi

```
jklmnopqrstuvwxyz
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()
```

返回值

返回 JSON 对象。

实例

```
var buf = new Buffer('www.runoob.com');
var json = buf.toJSON(buf);

console.log(json);
```

执行以上代码，输出结果为：

```
[ 119, 119, 119, 46, 114, 117, 110, 111, 111, 98, 46, 99, 11
1, 109 ]
```

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 = new Buffer('菜鸟教程');
var buffer2 = new Buffer('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 = new Buffer('ABC');
var buffer2 = new Buffer('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 buffer1 = new Buffer('ABC');
// 拷贝一个缓冲区
var buffer2 = new Buffer(3);
buffer1.copy(buffer2);
console.log("buffer2 content: " + buffer2.toString());
```

执行以上代码，输出结果为：

```
buffer2 content: ABC
```

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 = new Buffer('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 = new Buffer('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。
2	new Buffer(buffer) 拷贝参数 buffer 的数据到 Buffer 实例。
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'。
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.writeIntLE(value, offset, byteLength[, noAssert]) 将value 写入到 buffer 里, 它由offset 和 byteLength 决定, 支持 48 位计算, 例如:

```
var b = new Buffer(6);
b.writeUIntBE(0x1234567890ab, 0, 6);
// <Buffer 12 34 56 78 90 ab>
```

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，支持 48 位计算。 noAssert 值为 true 时，不再验证 value 和 offset 的有效性。默认是 false。
8	buf.writeIntLE(value, offset, byteLength[, noAssert]) 将value 写入到 buffer 里，它由offset 和 byteLength 决定，支持 48 位计算。 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，支持 48 位计算。 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 位以下的数字。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 位以下的数字。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 位以下的数字。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 位以下的数字。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，读取一个有符号 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有符号 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有符号 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。

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有符号 32 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。
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有符号 32 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。
26	buf.readInt8(offset[, noAssert])	根据指定的偏移量，读取一个 signed 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 signed 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。
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 signed 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。
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 signed 32 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。
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 signed 32 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。
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 位浮点数。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。
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 位浮点数。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。
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 double。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 double。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 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 必须是一个合法的有符号 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。
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 必须是一个合法的有符号 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。
38	buf.writeUInt32LE(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。

39	buf.writeUInt32BE(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。