

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,但是它对应V8堆内存之外的一块原始内存

v6.0之前创建Buffer对象,直接用new Buffer()构造函数来创建,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。

在v6.0以后,官方文档建议使用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。通过使用显示的字符编码,就可以在Buffer实例与普通的JavaScript字符串之间进行相互转换。

createBuffer.js

```
const buf = Buffer.from('Hello world', 'ascii');
console.log(buf.toString('hex'));
console.log(buf.toString('base64'));
console.log(buf.toString('UTF-8'));
```

```
C:\Users\Node>node createBuffer.js
48656c6c6f776f726c64
SGVsbG93b3JsZA==
Hello world
C:\Users\Node>
```

Node.js目前支持的字符编码包括:

ascii: 仅支持7位ASCII数据。如果去掉高位的话,这种编码非常快。

utf-8: 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

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 (创建缓冲区)

Buffer提供了以下API来创建Buffer类:

Buffer.alloc(size[, fill[, encoding]]): 返回指定大小的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 (Node缓冲区) :

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

参数:

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'utf-8'

根据encoding的字符编码写入String 到buf中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.另一部分解码的字符不会被写入

返回值

返回世界写入的大小,如果buffer空间不足,则只会写入部分字符串

英文字母和中文汉字在不同字符集编码下的字节数

英文字母 :

字节数 : 1;编码 : GB2312

字节数 : 1;编码 : GBK

字节数 : 1;编码 : GB18030

字节数 : 1;编码 : ISO-8859-1

字节数 : 1;编码 : UTF-8

字节数 : 4;编码 : UTF-16

字节数 : 2;编码 : UTF-16BE

字节数 : 2;编码 : UTF-16LE

中文汉字 :

字节数 : 2;编码 : GB2312

字节数 : 2;编码 : GBK

字节数 : 2;编码 : GB18030

字节数 : 1;编码 : ISO-8859-1

字节数 : 3;编码 : UTF-8

字节数 : 4;编码 : UTF-16

字节数 : 2;编码 : UTF-16BE

字节数 : 2;编码 : UTF-16LE

案列:

BufferAlloc.js

```
buf = Buffer.alloc(256);  
len = buf.write("你好");  
console.log("写入字节数: "+len);
```

```
C:\Users\Node>node BufferAlloc.js  
写入字节数: 6
```

从缓冲区中读取数据:

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

encoding: -使用的编码. 默认为 'Utf-8'

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: 缓冲区数据并使用指定的编码返回字符串

案例: **readBuffer.js**

```
buf = Buffer.alloc(256);  
len = buf.write("你好,我来自上海兴宁区");  
console.log(buf.toString('UTF-8'));  
console.log(buf.toString('UTF-8',0,6));  
console.log(buf.toString('Base64',0,6));  
console.log(buf.toString(undefined,0,6));
```

如下图:

```
C:\Users\Node>node readBuffer.js  
你好,我来自上海兴宁区  
你好  
5L2g5aW9  
你好
```

从图中可以看到:

`console.log(buf.toString('UTF-8'))`: 按照 'UTF' 编码读取缓冲区的全部数据

`console.log(buf.toString('UTF-8',0,6))`: 按照 'UTF' 编码读取缓冲区的前6个字节, 以0开始

`console.log(buf.toString('Base64', 0, 6))`:按照'Base64'编码
读取缓冲区的前6个字节,以0开始

`console.log(buf.toString(undefined, 0, 6))`:如果没有设置编码
格式,默认是UTF-8

将Buffer转换为JSON对象

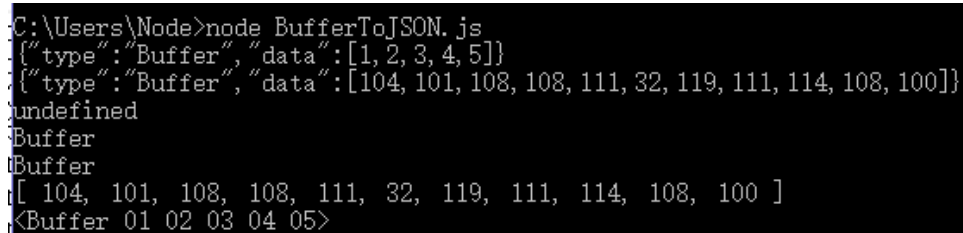
`buf.toJSON`,当字符串化一个Buffer实例时,`JSON.stringify`会隐士地
调用该`toJSON()`。

返回值:JSON对象

`BufferToJSON.js`

```
const buf = Buffer.from([0x1,0x2,0x3,0x4,0x5]);
const buff = Buffer.from("hello world");
const json = JSON.stringify(buf);
const json2 = JSON.stringify(buff);
console.log(json);
console.log(json2);
console.log(json.type);
var jsont = buf.toJSON();
var jsonf = buff.toJSON();
console.log(jsont.type);
console.log(jsonf.type);
console.log(jsonf.data);
const copy = JSON.parse(json,(key,value)=>{
return value && value.type === 'Buffer'?
Buffer.from(value.data):value;
})
console.log(copy);
```

输出如下图:



```
C:\Users\Node>node BufferToJSON.js
{"type":"Buffer","data":[1,2,3,4,5]}
{"type":"Buffer","data":[104,101,108,108,111,32,119,111,114,108,100]}
undefined
Buffer
Buffer
[ 104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100 ]
<Buffer 01 02 03 04 05>
```

从图中可以看到把Buffer转换为JSON对象的时候需要先将字符串转换成JSON字符串,然后再用toJSON正式转换为JSON对象.当我们还没有用toJSON把JSON字符串转换成JSON对象的时候,以键值对的形式输出就会显示undefined了.转换成JSON后就能输出确切的值了.后JSON是以键值对的数据格式存在的,Buffer转换称JSON字符串的时候,键统一为type,data,然后再进行存值.

缓冲区合并

语法:buffer.concat(list[,totalLenth])

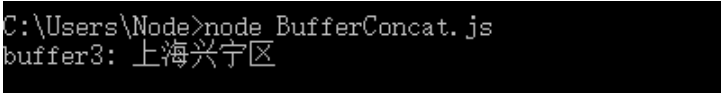
list -用于合并的Buffer对象数组列表

titalLength-指定合并后Buffer对象的总长度

案例: BufferConcat.js

```
var buffer1 = Buffer.from(('上海'));
var buffer2 = Buffer.from(('兴宁区'));
var buffer3 = Buffer.concat([buffer1,buffer2]);
console.log("buffer3: " + buffer3.toString());
```

输出如下图:



```
C:\Users\Node>node BufferConcat.js
buffer3: 上海兴宁区
```

从图中可以看到buffer1,buffer2两个缓冲区的内容合并在一起了.

缓冲区比较

buf.compare(otherBuffer);

参数如下:

otherBuffer 与buf对像比较的另一个Buffer对象

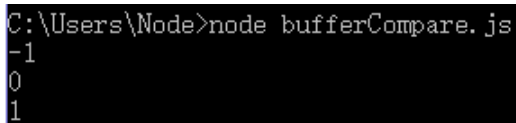
返回值:表示buf再otherBuffer之前,之后或者相同

案例:bufferCompare.js

```
var buffer1 = Buffer.from('ABC');
var buffer2 = Buffer.from('ABDC');
var buffer3 = Buffer.from('ABC');
var buffer4 = Buffer.from('AB');
```

```
var result = buffer1.compare(buffer2);
var result2 = buffer1.compare(buffer3);
var result3 = buffer1.compare(buffer4);
console.log(result);
console.log(result2);
console.log(result3);
```

输出如下图:



```
C:\Users\Node>node bufferCompare.js
-1
0
1
```

从图中你可以发现当两个缓冲区相等时返回值为0,大于时返回1,小于时返回-1

拷贝缓冲区:

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

- 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

案例:

```
var buf1 = Buffer.from("hello world");
var buf2 = Buffer.from("cili");
var buf3 = Buffer.from(buf1);
buf2.copy(buf1, 3, 2);
```

```
console.log(buf1.toString());
console.log(buf3.toString());
```

输出如下图:

```
C:\Users\Node>node bufferCopy.js
hello world
hello world
C:\Users\Node>
```

`var buf3 = Buffer.from(buf1)`:表示把buf1整个缓冲区复制给buf3.

`buf2.copy(buf1, 3, 2)`:表示把buf2这个缓冲区的位置2 (包括) 开始, 插入buf1的位置3. 如果不写第三个参数, 那么全插入buf1的位置3 (包括) 后面

缓冲区裁剪

`buf.slice(start, end)`

`start`: 数字, 可选, 默认值为0, 表示裁剪的起始位置

`end`-数字, 可选, 默认值为`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`的位置裁剪

案列: `bufferSlice.js`

```
var buf1 = Buffer.from("hello world");
console.log(buf1.slice(0, 2).toString());
```

如下图:

```
C:\Users\Node>node bufferSlice.js
he
C:\Users\Node>
```

从图中可以看到截取了下标为0, 1的值, 不包括2

缓冲区长度:

`buf.length`;

返回值:

返回Buffer对象所占据的内存长度

案列: `bufferLength.js`

```
var buf1 = Buffer.from("hello world");
console.log(buf1.length);
```

如下图:


```
C:\Users\Node>node bufferLength.js
11
```

这个缓冲区默认是UTF-8编码的, UTF-8编码下, 单个字符占据一个字节, 空格也占据1个字节, 所以就11个字节.

`buf[index]`: 获取或设置指定的字节. 返回值代表一个字节, 所以返回值的合法返回是十六进制0x00到0xFF或者十进制0至255

案例: `bufferIndex.js`

```
var buf = Buffer.from("hello world");
console.log(buf[2]);
```

如下图 :

```
C:\Users\Node>node bufferIndex.js
108
```

`buf.fill(value, start, end)`

`value` : 一个字符串

`start`: 填充的初始位置, 默认0

`end`: 填充的结束位置, 默认`Buffer.length`

案列: `bufferFill.js`

```
const buf = Buffer.alloc(20);
var Str = "you are good boy!";
buf.fill(Str, 0, 17);
console.log(buf.toString());
```

如下图:

```
C:\Users\Node>node bufferFill.js
you are good boy!
```

相当于把`Str`的值放进缓存区`Buf`中