Nhóm 3

Nguyễn Trung Hiếu

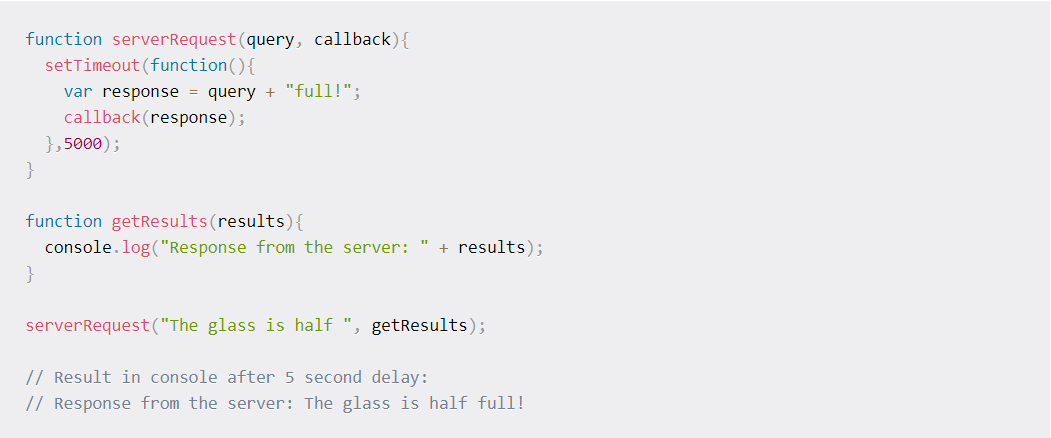
Phạm Văn Tuyên

1. Callback trong Node.js
2. Callback là gì?

* Callback có tính chất không đồng bộ. Một hàm callback được gọi khi hoàn thành một nhiệm vụ cụ thể. Tất cả các API của Node đều được viết theo các cách của hàm callback. Xem ví dụ để hiểu callback trong node.js
* Ví dụ, với một hàm đọc file bắt đầu với việc đọc file và hệ thống tiếp tục thực thi lệnh tiếp theo. Khi phần đọc file được hoàn thành, nó sẽ gọi về một hàm callback, với nội dung của file là tham số. Do đó sẽ không có blocking hoặc chờ khi đọc File mà hệ thống vẫn tiếp tục thực thi các lệnh tiếp theo. Nó làm cho Node.js có hiệu năng cao hơn, như có số lượng request cao hơn mà không cần phải chờ kết quả trả về.

1. Tại sao lại cần Callback?

* Phần lớn thời gian chúng ta đang tạo ra các chương trình và ứng dụng hoạt động một cách **đồng bộ** . Nói cách khác, một số hoạt động của chúng tôi chỉ được bắt đầu sau khi những hoạt động trước đó đã hoàn thành. Thông thường, khi chúng tôi yêu cầu dữ liệu từ các nguồn khác, chẳng hạn như API bên ngoài, chúng tôi không phải lúc nào cũng biết khi nào dữ liệu của mình sẽ được phục vụ trở lại. Trong những trường hợp này, chúng tôi muốn đợi phản hồi, nhưng không phải lúc nào chúng tôi cũng muốn toàn bộ ứng dụng của mình tạm dừng trong khi dữ liệu của chúng tôi đang được tìm nạp. Những tình huống này là nơi các hàm gọi lại có ích.
* Hãy xem một ví dụ mô phỏng một yêu cầu đến máy chủ:



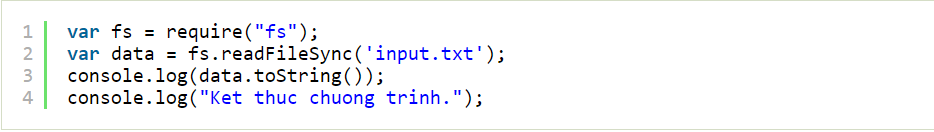
* Trong ví dụ trên, chúng tôi thực hiện một yêu cầu giả tới một máy chủ. Sau 5 giây trôi qua, phản hồi được sửa đổi và sau đó hàm gọi lại của chúng tôi **getResults** được thực thi. Để xem điều này đang hoạt động, bạn có thể sao chép / dán đoạn mã trên vào công cụ dành cho nhà phát triển của trình duyệt và thực thi nó.
* Ngoài ra, nếu bạn đã quen thuộc với **setTimeout**, thì bạn đã sử dụng các hàm gọi lại. Đối số hàm ẩn danh được truyền vào **setTimeout** lệnh gọi hàm của ví dụ trên cũng là một lệnh gọi lại! Vì vậy, lệnh gọi lại ban đầu của ví dụ thực sự được thực thi bởi một lệnh gọi lại khác. Hãy cẩn thận không lồng quá nhiều lệnh gọi lại nếu bạn có thể giúp được, vì điều này có thể dẫn đến một thứ gọi là "địa ngục gọi lại"! Như tên của nó, nó không phải là một niềm vui để giải quyết.

1. Ví dụ Blocking Code

* Tạo một dòng text với tên input.txt với nội dung sau đây:



* Tạo một file js với tên callback1.js với nội dung sau đây:



* Bây giờ chạy lệnh sau để xem kết quả:



* Kết quả:

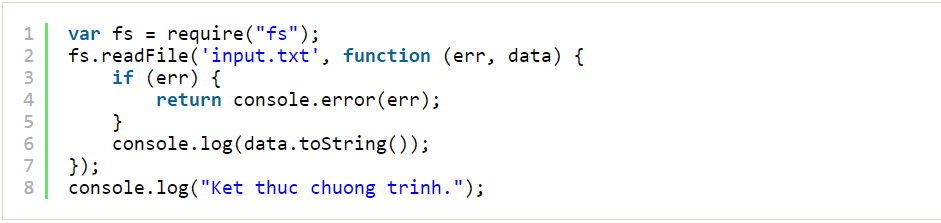


1. Ví dụ Non-Blocking Code

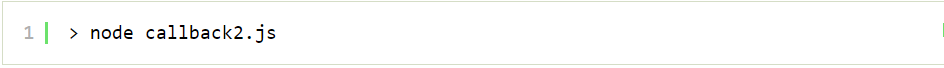
* Tạo một file với tên input.txt với nội dung sau đây:



* Cập nhật callback2.js với dòng code sau:



* Bây giờ chạy callback2.js để xem kết quả:



* Kết quả:



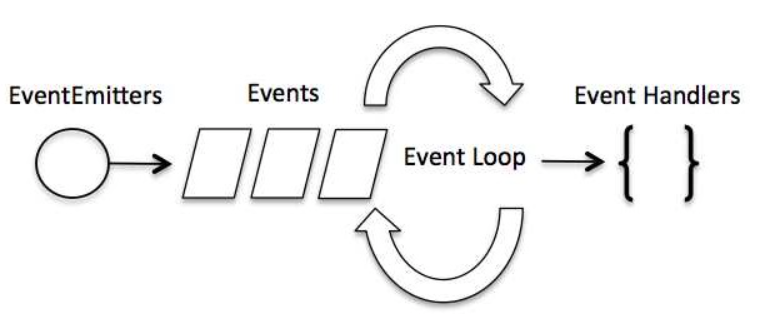
* Hai ví dụ trên giải thích định nghĩa cách gọi blocking và non-blocking. Ví dụ đầu tiên chỉ ra rằng chương trình bị khóa cho đến khi nó hoàn thành nhiệm vụ đọc file và chỉ tiếp tục chạy vài giây sau đó, chương trình thứ 2 không đợi cho việc đọc file và tiếp tục in "Ket thuc chuong trinh." cùng thời điểm thực hiện chương trình.

1. Event Loop trong Node.js

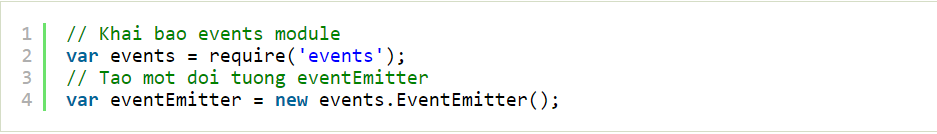
Node.js là ứng dụng đơn luồng nhưng có hỗ trợ việc xử lí đồng thời thông qua các định nghĩa về event (sự kiện) và callback. Tất cả các API của Node.js có tính chất không đồng bộ và được xử lí đơn luồng, nó sử dụng hàm **async** để duy trì sự đồng thời. Node.js sử dụng Observer Pattern. Các Thread trong Node.js có một Event Loop và bất cứ khi nào có tác vụ nào hoàn thành, nó sẽ kích hoạt sự kiện tương ứng để báo cho Event Listener sẵn sàng thực hiện.

1. Khái niệm Event Driven trong Node.js

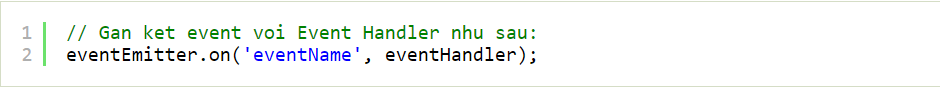
* Node.js sử dụng rất nhiều sự kiện, đó là lý do tại sao Node.js khá nhanh so với các công nghệ khác. Ngay khi Node khởi động Server của nó, nó sẽ nhanh chóng khởi tạo các biến, khai báo các hàm và sau đó đơn giản là chờ đợi các sự kiện xảy ra.
* Trong ứng dụng xử lý sự kiện, nhìn chung vòng lặp chính lắng nghe các sự kiện, và sau đó trigger đến hàm callback khi một trong những sự kiện được phát hiện.



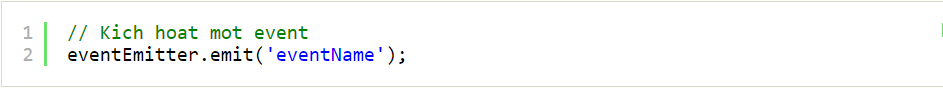
* Nhìn chung các Event là tương tự như các hàm callback. Điểm khác nhau nằm ở chỗ, hàm callback gọi khi một hàm không đồng bộ và trả về kết quả của nó trong khi phần xử lí sự kiện làm việc trên Observer Pattern. Hàm này sẽ lắng nghe các sự kiện, đóng vai trò như một Observers (Người quan sát). Bất cứ khi nào một sự kiện phát sinh, các hàm Listener của nó sẽ bắt đầu thực thi. Node.js có nhiều sự kiện có sẵn thông qua **events** Module và lớp **EventEmitter** có thể dựa vào để bind sự kiện và lắng nghe sự kiện.
* Trước khi sử dụng event Module, bạn sử dụng phương thức require() để khai báo như sau:



* Sau đó, để gắn kết Event Handler với một sự kiện, bạn sử dụng cú pháp sau:

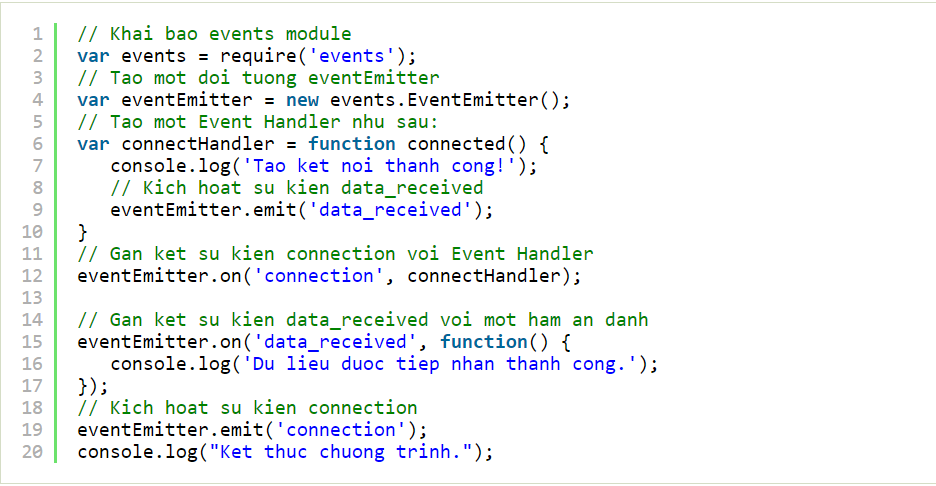


* Bạn có thể kích hoạt một sự kiện bởi sử dụng phương thức emit() của EventEmitter:



1. Ví dụ sử dụng Event Loop trong Node.js

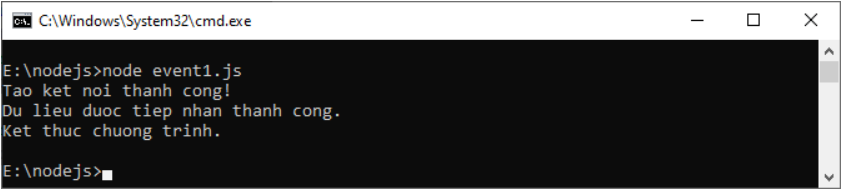
* Tạo một file js với tên là event1.js có đoạn code sau:



* Chạy chương trình trên như sau:

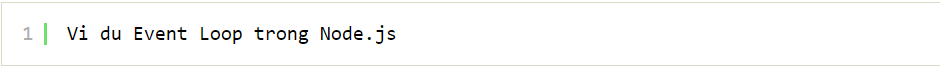


* Kết quả là:

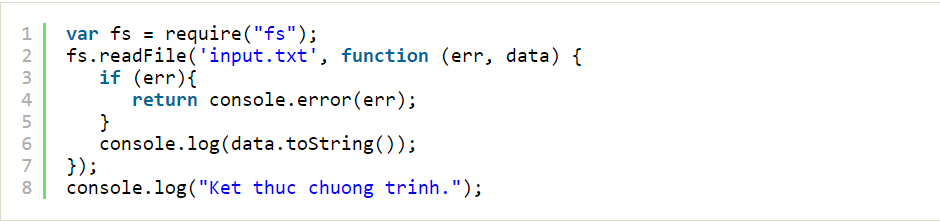


1. Ứng dụng Node hoạt động như thế nào?

* Trong ứng dụng Node.js, một hàm không đồng bộ chấp nhận một callback như tham số cuối cùng và hàm callback chấp nhận error như tham số đầu tiên. Cùng xem lại ví dụ trước. Tạo một text file với tên input.txt với nội dung sau đây:



* Trong ví dụ này, chúng ta sử dụng Module **fs** để xử lý các hoạt động File I/O. Đầu tiên, bạn tạo một file js với tên là event2.js như sau:



* Ở đây, fs.readFile() là một hàm không đồng bộ với mục đích để đọc file. Nếu có một lỗi xảy ra trong quá trình đọc file, đối tượng err sẽ chứa lỗi đó, nếu không thì data sẽ chứa các phần nội dung của file đó. Hàm readFile truyền err và data đến hàm callback sau khi quá trình đọc file đã hoàn thành, và cuối cùng sẽ in ra nội dung.
* Kết quả sẽ là:

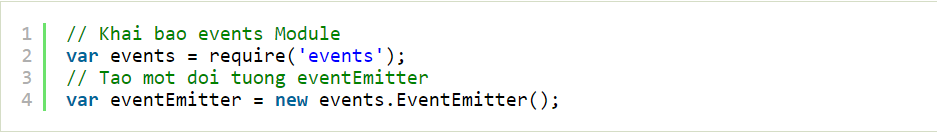


1. Lớp Event Emitter trong Node.js

Nhiều đối tượng trong Node.js sinh ra các sự kiện, ví dụ **net.Server** sinh ra một sự kiện mỗi khi có một kết nối ngang hàng đến nó, hay **fs.readStream** sinh ra sự kiện khi một file được mở. Tất cả các đối tượng này đều là sự thể hiện của lớp **EventEmitter** trong Node.js.

1. Lớp EventEmitter trong Node.js

* Lớp events.EventEmitter nằm trong events Module. Lớp này được truy cập qua cú pháp sau:



* Khi một EventEmitter gặp bất kì lỗi nào, nó sẽ sinh ra một Error Event. Khi một Listener mới được thêm, sự kiện 'newListener' sẽ được kích hoạt và một Listener sẽ bị loại bỏ, sự kiện 'removeListener' sẽ được kích hoạt.
* Event Emitter cung cấp nhiều thuộc tính như **on** hay **emit**. Thuộc tính *on* được sử dụng để gắn kết một hàm với sự kiện, và *emit* dược sử dụng để kích hoạt một sự kiện.

1. Các phương thức của lớp EventEmitter trong Node.js

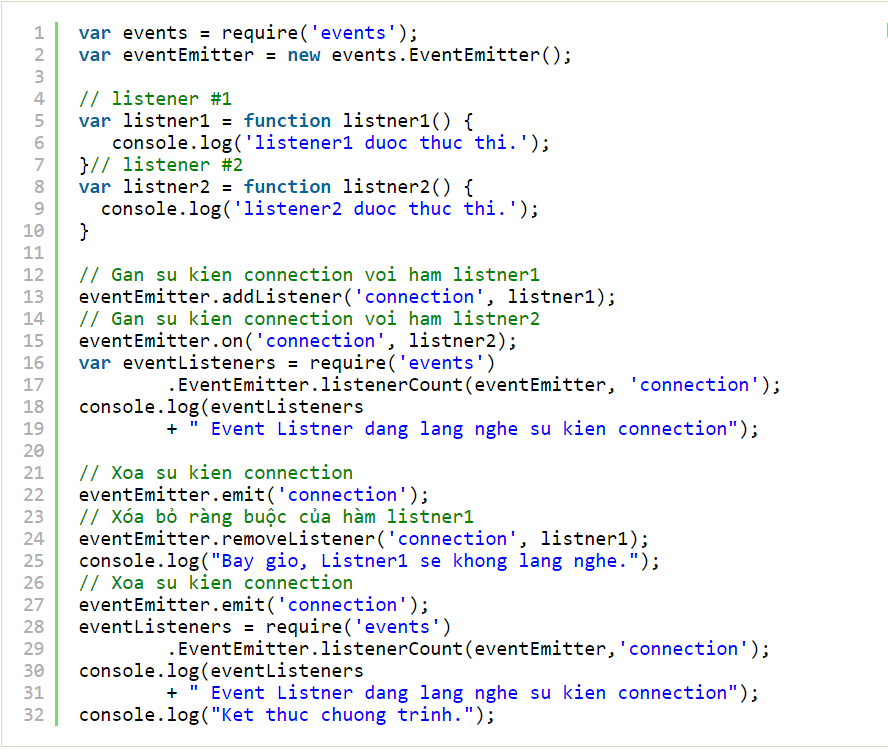
|  |  |
| --- | --- |
| **Stt Phương thức & Mô tả** | |
| 1 | **addListener(event,listener)** Thêm một Listener vào phần cuối của mảng các Listener cho một sự kiện cụ thể |
| 2 | **on(event,listener)** Thêm một Listener vào phần cuối của mảng các Listener cho một sự kiện cụ thể |
| 3 | **once(event,listener)** Thêm một One-Time Listener cho sự kiện. Listener dạng này sẽ chỉ được gọi khi sự kiện được kích hoạt, sau đó nó sẽ bị xóa |
| 4 | **removeListener(event,listener)** Xóa một Listener ra khỏi mảng các Listener cho một sự kiện nào đó. |
| 5 | **removeAllListeners([event])** Xóa tất cả Listener của một sự kiện |
| 6 | **setMaxListeners(n)** Theo mặc định, lớp EventEmitters sẽ in một lời cảnh báo nếu bạn thêm nhiều hơn 10 Listener cho một sự kiện cụ thể. Việc này khá hữu ích, bởi vì nó sẽ giúp tìm ra các lỗi gây rò rỉ bộ nhớ. Tất nhiên, không phải tất cả các Emitters đều cần được giới hạn với con số là 10. Hàm này cho phép bạn tăng con số đó. Thiết lập nó về 0 để không giới hạn lượng Listener cần thêm |
| 7 | **listeners(event)** Trả về một mảng bao gồm các Listener cho một sự kiện cụ thể nào đó |
| 8 | **emit(event,[arg1],[arg2],[...])** Thực thi từng Listener với các tham số đã cho. Trả về true nếu sự kiện có các Listener, và false nếu không có |

1. Các sự kiện của lớp EventEmitter trong Node.js

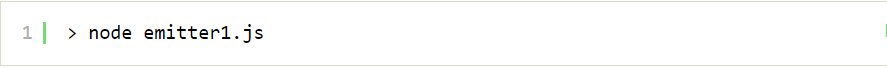
|  |  |
| --- | --- |
| **Stt Sự kiện & Mô tả** | |
| 1 | **newListener**   * **event** - Dạng chuỗi, biểu diễn tên sự kiện * **listener** - Tên hàm xử lý sự kiện   Sự kiện này được sinh bất cứ khi nào bạn thêm một Listener. Khi sự kiện này được kích hoạt, Listener có thể sẽ chưa được thêm vào mảng Listener của sự kiện |
| 2 | **removeListener**   * **event** - Dạng chuỗi, biểu diễn tên sự kiện * **listener** - Tên hàm xử lý sự kiện   Sự kiện này xảy ra bất cứ khi nào có ai đó xóa một Listener. Khi một sự kiện được kích hoạt, Listener này chưa được xóa khỏi mảng Listener của sự kiện |

1. Ví dụ về lớp EventEmitter trong Node.js

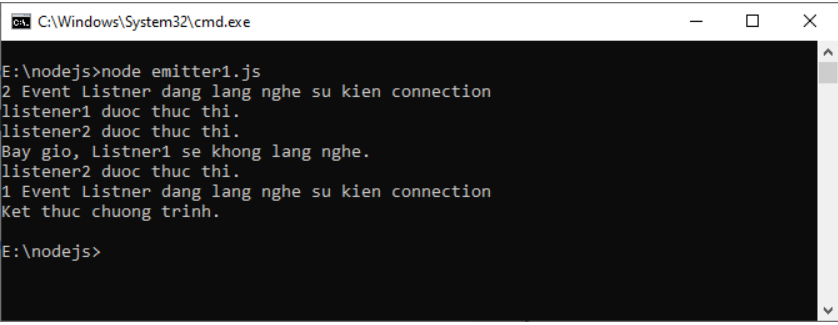
* Tạo một file js với tên là emitter1.js với nội dụng Node.js như dưới đây. Trong emitter1.js, đầu tiên bạn khai báo Module **events**. Tiếp đó, bạn sử dụng phương thức addListener() để thêm một Listener cho một sự kiện nào đó, và sử dụng các thuộc tính on và emit để thực hiện các tính năng đã trình bày ở trên.



* Chạy emitter1.js để xem kết quả:



* Kết quả:



1. Buffer trong Node.js

Javascript thuần được mã hóa Unicode, nhưng đối với các dữ liệu nhị phân thì không thật sự tốt. Khi làm việc với các luồng TCP hoặc file dữ liệu, cần thiết phải xử lý các luồng dữ liệu bát phân. Node.js cung cấp các lớp Buffer cho phép lưu trữ các dữ liệu thô như một mảng các số nguyên tương ứng với phần cấp phát bộ nhớ thô bên ngoài V8 heap.

**Lớp Buffer trong Node.js** là các lớp toàn cục và có thể được truy cập trong ứng dụng mà không cần khai báo các Module Buffer bởi phương thức require() như các Module khác.

* 1. Tạo các Buffer trong Node.js

Buffer trong Node.js có thể được xây dựng theo nhiều cách khác nhau.

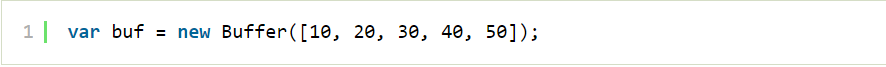
**Cách 1**

Cú pháp cho việc tạo một Buffer cỡ 10:



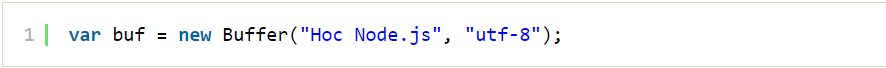
**Cách 2**

Cú pháp tạo một Buffer từ một mảng cho trước:



**Cách 3**

Cú pháp tạo một Buffer từ một chuỗi cho trước và với kiểu mã hóa tùy ý:

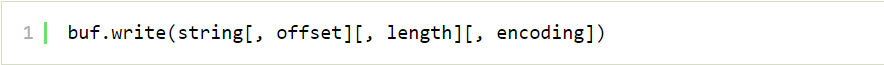


Mặc dù "utf8" là cách mã hóa mặc định nhưng bạn có thể sử dụng các cách mã hóa khác như "ascii", "utf8","base64", …

* 1. Ghi dữ liệu vào Buffer trong Node.js

**Cú pháp**

Cú pháp để ghi một Buffer trong Node.js là:



**Chi tiết về tham số**

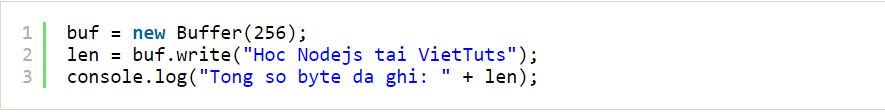
* **string**: Đây là một dữ liệu dạng chuỗi được ghi tới buffer.
* **offset**: Đây là chỉ mục để buffer bắt đầu ghi tại đó. Giá trị mặc định là số 0.
* **length**: Số lượng các byte được ghi. Mặc định là buffer.length.
* **encoding**vMã hóa được sử dụng. "utf8" là mã hóa mặc định.

**Giá trị trả về**

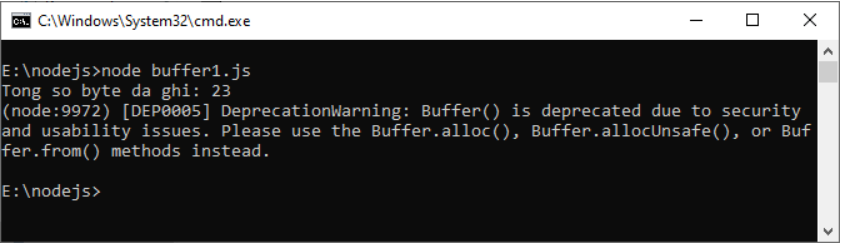
Phương thức sẽ trả về số lượng các byte được ghi. Nếu bộ nhớ trong buffer là không đủ để đáp ứng cho toàn bộ chuỗi, nó sẽ ghi một phần của chuỗi đó.

**Ví dụ cách ghi dữ liệu vào Buffer trong Node.js**

Ở ví dụ này, mình sử dụng phương thức write() nhận tham số là dữ liệu dạng chuỗi để ghi dữ liệu đó tới Buffer.



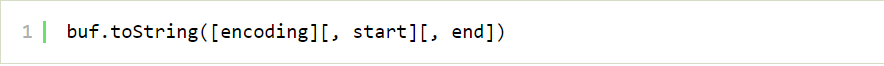
Khi chương trình trên được thực thi sẽ cho kết quả:



* 1. Đọc dữ liệu từ Buffer trong Node.js

**Cú pháp**

Cú pháp để đọc dữ liệu từ Buffer trong Node.js như sau:



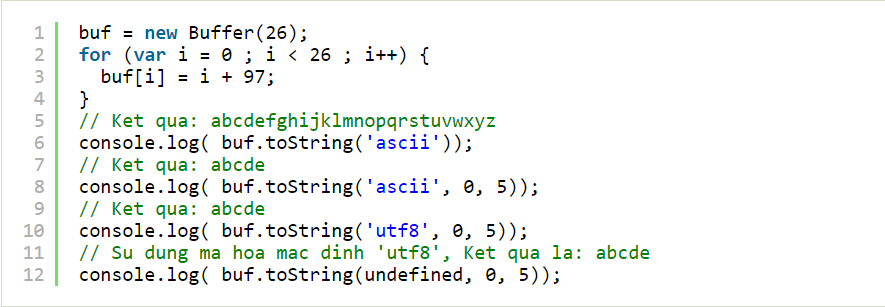
**Chi tiết về tham số**

* **encoding**: Là mã hóa để sử dụng. Mã hóa mặc định là 'utf8'.
* **start**: Chỉ mục để bắt đầu hoạt động đọc, giá trị mặc định là 0.
* **end**: Chỉ mục để kết thúc hoạt động đọc, giá trị mặc định là độ dài của Buffer.

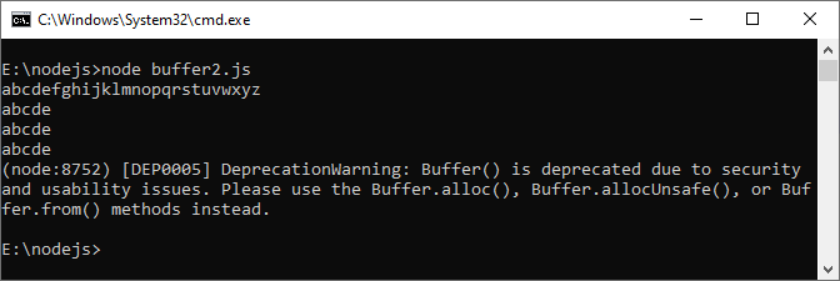
**Giá trị trả về**

Phương thức này giải mã và trả về một chuỗi từ dữ liệu đã được mã hóa trong Buffer bởi sử dụng bộ mã hóa cụ thể.

**Ví dụ cách đọc dữ liệu từ Buffer trong Node.js**



Khi chương trình trên được thực thi sẽ cho kết quả:



* 1. Chuyển đổi Buffer thành JSON trong Node.js

**Cú pháp**

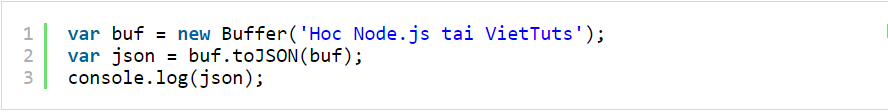
Để chuyển đổi một Buffer trong Node.js thành đối tượng JSON, bạn sử dụng phương thức toJSON() có cú pháp như sau:



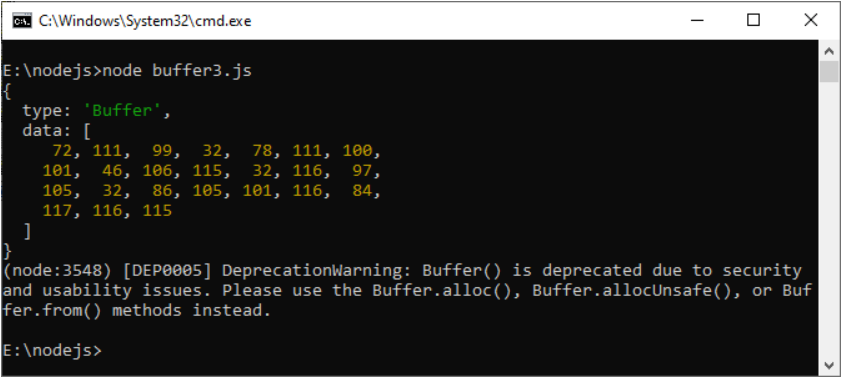
**Giá trị trả về**

Phương thức này trả về một biểu diễn JSON cho đối tượng Buffer đã cho.

**Ví dụ cách chuyển đổi Buffer thành JSON**



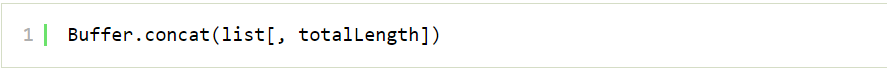
Khi chương trình trên được thực thi sẽ cho kết quả:



* 1. Ghép nối các Buffer trong Node.js

**Cú pháp**

Để nối ghép hai hoặc nhiều Buffer thành một Buffer trong Node.js, bạn sử dụng phương thức concat() như sau:



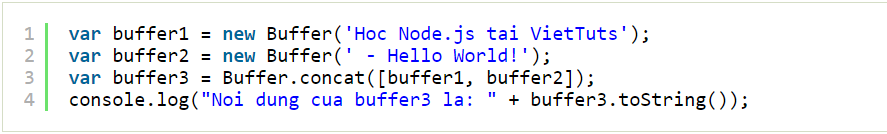
**Chi tiết về tham số**

* **list** - Xác định một mảng các Buffer được sử dụng để ghép nối thành một Buffer.
* **totalLength** - Là tổng độ dài của các Buffer sau khi đã được ghép nối.

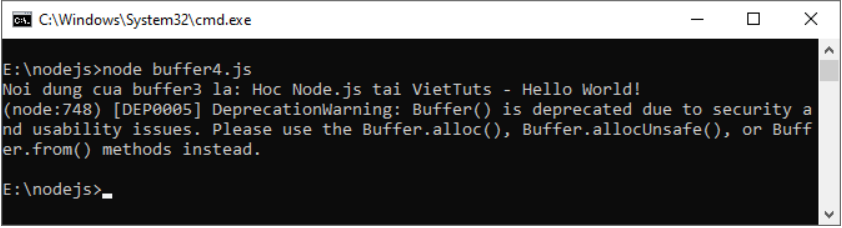
**Giá trị trả về**

Phương thức này trả về một Buffer mới.

**Ví dụ cách ghép nối các Buffer**



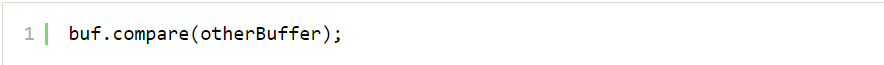
Khi chương trình trên được thực thi sẽ cho kết quả:



* 1. So sánh các Buffer trong Node.js

**Cú pháp**

Để so sánh hai Buffer trong Node.js, bạn sử dụng phương thức compare() như sau:



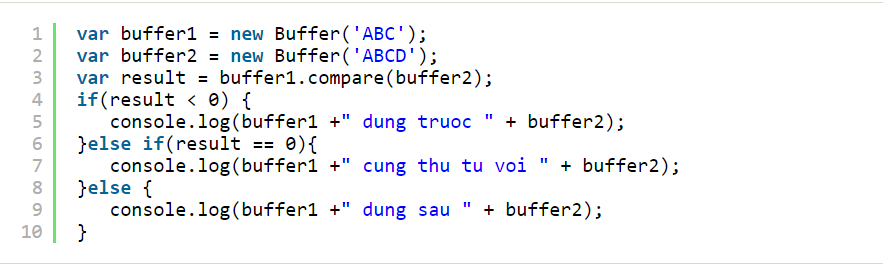
**Chi tiết về tham số**

* **otherBuffer**: Một Buffer khác để được so sánh với Buffer có tên là **buf**.

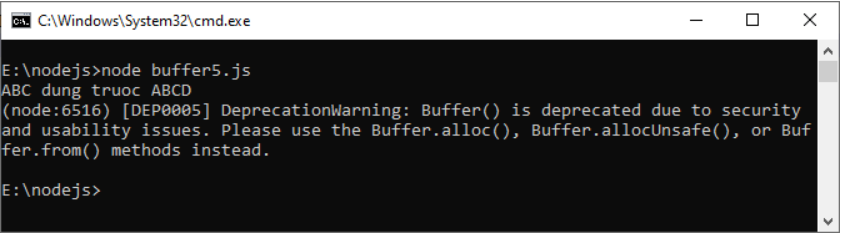
**Giá trị trả về**

Trả về một giá trị dạng số thể hiện Buffer này là đứng trước, sau hay cùng thứ tự với Buffer kia.

**Ví dụ minh họa cách so sánh hai Buffer trong Node.js**



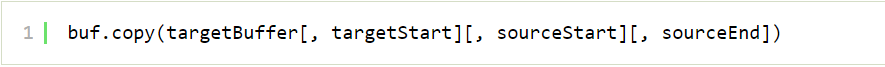
Khi chương trình trên được thực thi sẽ cho kết quả:



* 1. Sao chép Buffer trong Node.js

**Cú pháp**

Để sao chép Buffer trong Node.js, bạn sử dụng phương thức copy() như sau:



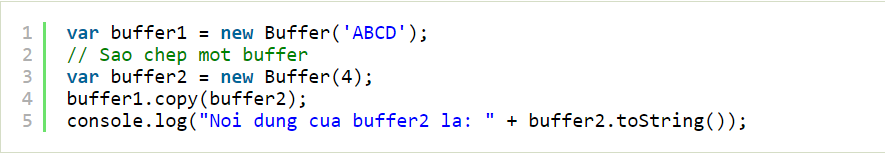
**Chi tiết về tham số**

* **targetBuffer**: Đối tượng Buffer, đây là nơi Buffer sẽ được sao chép.
* **targetStart**: Dạng số, mặc định là 0.
* **sourceStart**: Dạng số, mặc định là 0.
* **sourceEnd**: Dạng số, mặc định là độ dài của buffer.

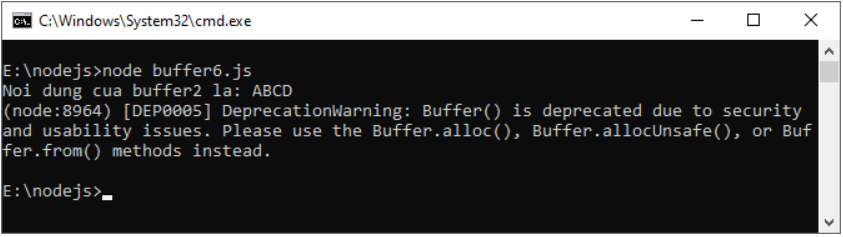
**Giá trị trả về**

Phương thức copy() này không trả về bất kỳ giá trị nào.

**Ví dụ minh họa cách sao chép Buffer**



Khi chương trình trên được thực thi sẽ cho kết quả:



* 1. Chia nhỏ Buffer trong Node.js

**Cú pháp**

Để lập một Buffer con của một Buffer trong Node.js, bạn sử dụng phương thức slice() như sau:



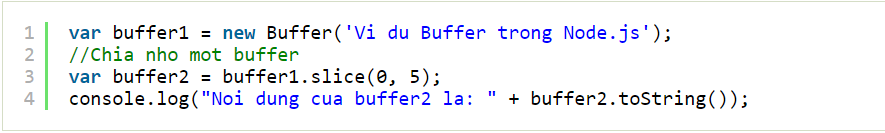
**Chi tiết về tham số**

* **start**: Dạng số, giá trị mặc định là 0.
* **end**: Dạng số, giá trị mặc định là buffer.length

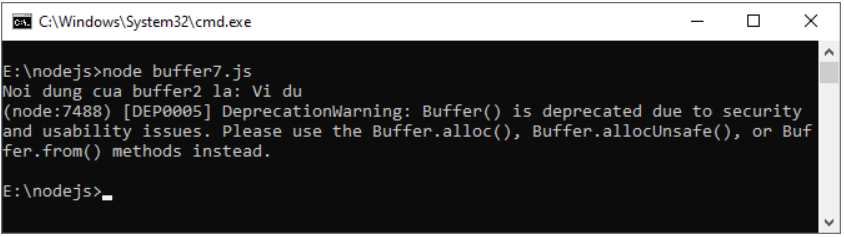
**Giá trị trả về**

Trả về một Buffer mới mà tham chiếu tới cùng vùng bộ nhớ như Buffer cũ.

**Ví dụ**



Khi chương trình trên được thực thi sẽ cho kết quả:



* 1. Độ dài Buffer trong Node.js

**Cú pháp**

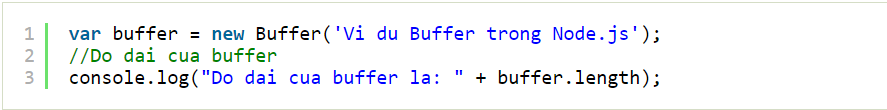
Để lấy độ dài (bằng giá trị byte) của một Buffer trong Node.js, bạn sử dụng thuộc tính length như sau:



**Giá trị trả về**

Trả về độ dài bằng byte của một Buffer.

**Ví dụ**



Khi chương trình trên được thực thi sẽ cho kết quả:

