t và quản lý module trong NodeJS213Cài đặt và quản lý module trong NodeJSchapter.2
n NodeJS317Lập trình cơ bản NodeJSchapter.3

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

BÀI TẬP CHUYÊN ĐỀ
WEB VÀ DỊCH VỤ TRỰC TUYẾN

Tìm hiểu và lập trình NodeJS

Bạch Văn Hải- CNTT1 20101464
Nguyễn Đức Tuấn-CNTT2 20102430

Giáo viên hướng dẫn :

Tạ Tuấn Anh

HÀ NỘI
Ngày 27 tháng 3 năm 2013

Mục lục

Lời nói đầu

Công nghệ web đang có những bước phát triển mạnh mẽ. Sự phát triển bùng nổ của mạng xã hội và các dịch vụ thời gian thực thúc đẩy nhiều công nghệ mới ra đời, giúp tăng cường khả năng tương tác. Trong phần nghiên cứu này, chúng em xin đề cập đến một công nghệ rất mới, NodeJS. Được giới thiệu từ năm 2010,đây là một công nghệ mạnh mẽ giúp xây dựng các ứng dụng hiệu năng cao, có khả năng tương tác người dùng mạnh mẽ.

Chương 1

Công nghệ NodeJS

1.1 Mô hình WebServer truyền thống

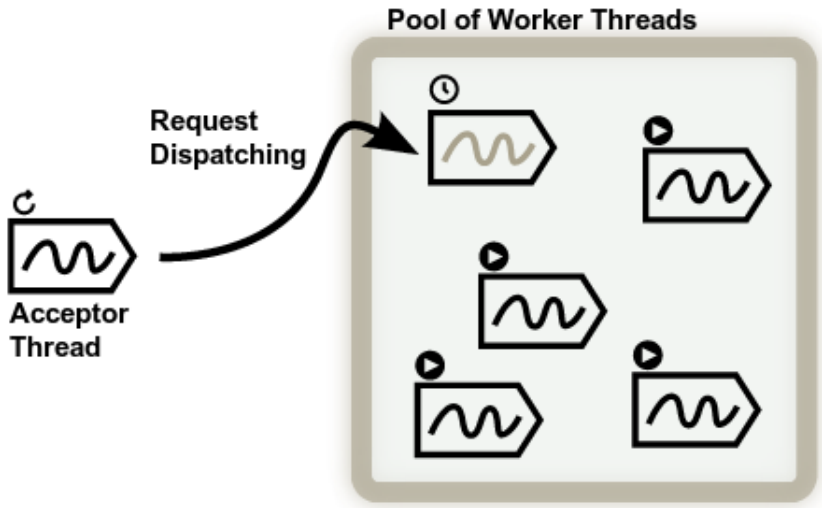
Trước hết ta tìm hiểu hoạt động của server. Server truyền thống hoạt động theo cơ chế `synchronize IO`. Khi client gửi đến server một yêu cầu, server sẽ dựa trên url yêu cầu để tìm tài nguyên tương ứng. Server sẽ đợi vào ra dữ liệu.Tùy vào tài nguyên đó, server có thể gọi một chương trình khác để tiền xử lý (ví dụ php) và trả lại kết quả cho client. Để đáp ứng được yêu cầu cho nhiều client, server sẽ sử dụng thread để quản lý mỗi yêu cầu của client. Để tăng cường hiệu quả sử dụng thread, các server thường sử dụng cơ chế spool thread. Mô hình này có thể mô hình bởi hình vẽ dưới đây.
Như có thể thấy trong hình 1.1, khi có một yêu cầu được chấp nhận. Bộ phân phối yêu cầu sẽ tìm thread rảnh rỗi và phân phối yêu cầu cho thread đó. Tuy nhiên, mô hình này vẫn chứa những hạn chế nhất định. Đó là:

- Hạn chế của việc sử dụng luồng: Việc sử dụng luồng có nhưng hạn chế nhất định. Trước hết, số lượng luồng tối đa có thể tạo ra bị giới hạn bởi khả năng của CPU. Tiếp đến là chi phí quản lý luồng cực kỳ tốn kém đặc biệt khi số lượng luồng lớn(khởi tạo luồng, lập lịch và chuyển đổi luồng, hủy bỏ luồng).
- Lãng phí CPU do thời gian vào ra IO. Khi một yêu cầu từ client gửi tới, server sẽ phải đọc dữ liệu được định danh trong url, và chuyển nó tới bộ tiền xử lý. Trong thời gian vào ra, thread đó hoàn toàn không có tính toán. Do vậy hiệu xuất sử dụng CPU rất thấp.
- Việc đồng bộ giữa các thread: Trong trường hợp muốn đồng bộ giữa các thread, việc này cực kỳ khó.

1.2 NodeJS, mô hình server mới

1.2.1 Mô hình NodeJS

Khác với cơ chế truyền thống, NodeJS hoạt động theo cơ chế `asynchronous IO`. Trong NodeJS, có duy nhất một tiến trình hoạt động. Khi yêu cầu gửi từ client đến, server sẽ gửi yêu cầu cho bộ phận vào ra về yêu cầu đó và tiếp tục lắng nghe các yêu cầu khác từ client. Khi đã kết thúc vào ra, bộ phận vào ra gửi tín hiệu ngắt tới server. Server sẽ nhận dữ liệu này và tiếp tục xử lý. Như vậy, hiệu xuất sử dụng CPU đã tăng nên. Hơn nữa, trong trường hợp này, do chỉ có một tiến trình duy nhất, ta không còn gặp phải vấn đề về thread và đồng



Hình 1.1: Mô hình Server truyền thống

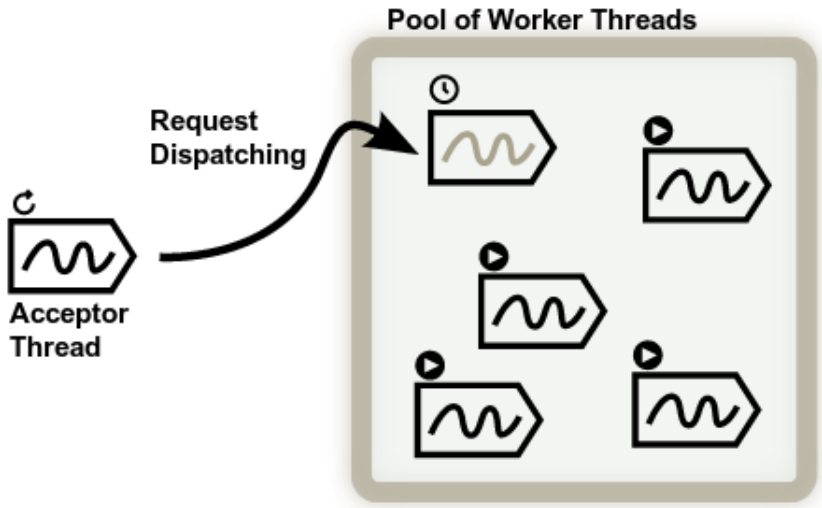
bộ giữa các Thread như trong mô hình cũ gặp phải nữa. Mô hình hoạt động của NodeJS được theer hiện trong hình dưới đây: Theo cơ chế này, khi có ngắt xuất hiện, server sẽ tiếp tục thực hiện xử lý với dữ liệu được đọc. Vậy làm thế nào server có thể xác định được công việc cần phải làm? Giả sử đây là một công việc cần thực hiện của server:

```
var post = db.query('SELECT * FROM posts where id = 1');

// processing from this line onward cannot execute

// until the line above completes

doSomethingWithPost(post);

doSomethingElse();
```

Trong ví dụ này server phải thực hiện truy vấn cơ sở dữ liệu. Sau khi thực hiện song truy vấn, ta muốn server thực hiện hàm `doSomethingWithPost(post)` Trong mô hình `Asynchronous IO`, làm thế nào để khi truy vấn thực hiện xong, server biết và thực hiện hàm `ioSomethingWithPost(post)` NodeJS quản lý theo cơ chế `Event callback`. `Event callback` là khái niệm chỉ hàm sẽ được gọi khi có một sự kiện xảy ra. Code trên có thể được viết lại như sau:

```
callback = function(post) {
        doSomethingWithPost(post);
};

db.query('SELECT * FROM posts where id = 1', callback);
doSomethingElse();
```

Cách tiếp cận này khá đơn giản và hợp lý. Ta sẽ truyền hàm cần thực thi như một tham số của truy vấn. Khi quá trình IO kết thúc, callback sẽ được gọi. Vấn đề còn lại duy nhất là làm sao quản lý ngữ cảnh của hàm. Khi hàm trên được gọi, các biến môi trường đang ở trạng thái nào? Ryan Dahl , tác giả của NodeJS ban đầu viết sử dụng ngôn ngữ C, tuy nhiên do



Hình 1.2: Mô hình hoạt động của NodeJS

việc quản lý ngữ cảnh giữa các callback rất phức tạp, ông đã chuyển sang dùng ngôn ngữ Lua. Lua có cả các thư viện non IO blocking và I/O blocking có thể gây phức tạp cho các nhà lập trình vì vậ Lua không phải là một giải pháp hay. Sau đó, ông nghĩ tới javascript. Trong javascript, có một đặc trưng thú vị là closure function. Khi một hàm được gọi, trước hết bộ thông dịch sẽ tìm các biến cục bộ của hàm sẽ được tham chiếu. Nếu biến đó không tồn tại, bộ thông dịch tiếp tục tìm kiếm ra toàn cục. Ngữ cảnh của hàm được xác định khi hàm được khai báo. Để hiểu kỹ vấn đề này hơn, ta cùng xem ví dụ đơn giản sau:

```
var clickCount = 0;

document.getElementById('mybutton').onclick = function() {
        clickCount ++;
}
```

Trong ví dụ này, hàm callback ở đây thực hiện đếm số lần click vào một nút. Biến clickCount ở đây không được khai báo trong hàm. Do vậy nó chính là biến toàn cục. Do tại thời điểm khai báo, biến clickCount đã tồn tại, nên đây chính là biến được tham chiếu đến khi callback được gọi.

1.2.2 Một kết quả benchmark

Trong phần này ta cùng làm một ví dụ benchmark đơn giản để so sánh hiệu năng của 2 mô hình

NodeJs vs Apache PHP benchmark

Tất cả các bài so sánh đều được thực hiện trong điều kiện:

- Server Apache 2.2.22

- Apache Bench 2.3
- Node.js v0.8.12
- Laptop:
  - Ubuntu: Release 12.04(precise) 32-bit
  - KernelLinux 3.2.0-38-generic-pae
  - GNOME 3.4.2
- Hardware:
  - Memory: 3.8 GB
  - Processor: IntelCore i3-2330M CPU @ 2.20GHz x 4

Chương trình này có 2 tùy chọn quan trọng là -n số yêu cầu và -c số lượng yêu cầu đồng thời và cuối cùng là url.
Test với Node.js:

```
/usr/bin/ab -n 100000 -c 1000 http://localhost:8080/
```

Test với Apache:

```
/usr/bin/ab -n 100000 -c 1000 http://localhost/
```

Test với chương trình HelloWorld

Chương trình đơn giản dùng để test với node.js server và Apache:
`hello.js`

```
var sys = require('sys'),
http = require('http');

http.createServer(function(req, res) {
  res.writeHead(200, {
    'Server': 'nodeJS',
    'Content-Type': 'text/html',
    'Content-Length': '18',
    'Connection': 'close',
  });
  res.write('<p>Hello World</p>');
  res.end();
}).listen(8080);
```

`index.php`

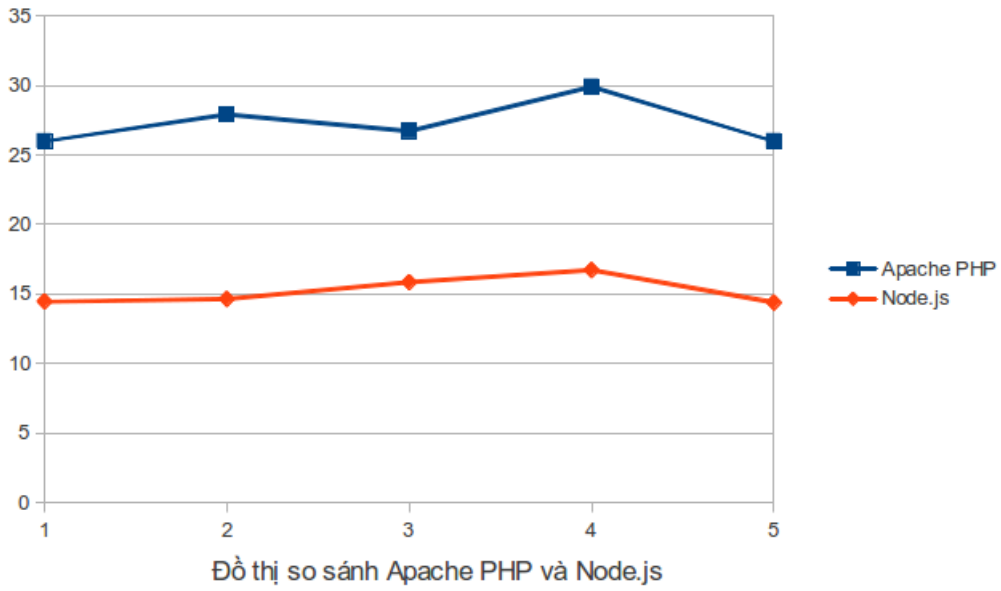
```php
<?php
    echo '<p>Hello World</p>'
?>
```

Sau quá trình test ta thu được các kết quả dưới bảng sau:(Thời gian tính theo đơn vị giây)

100,000 **yêu cầu,** 1000 **yêu cầu đồng thời**

x	1	2	3	4	5
Apache	25.972	27.908	26.708	29.893	25.980
NodeJs	14.467	14.643	15.854	16.724	14.389

Đồ thị minh họa:



1.2.3 Test với chương trình tính số PI

Chương trình sử dụng để test:
`testPI.js`

```
var sys = require('sys'),
http = require('http');

http.createServer(function(req, res) {
  res.writeHead(200, {
    'Server': 'nodeJS',
    'Content-Type': 'text/html',
    'Content-Length': '18',
    'Connection': 'close',
  });

  var x = 3;
  var steps = 1000000;
  var pi = (1 - 1 / x);

  for (var i = 0; i < steps; i++) {
    x += 2;
    if (i % 2) {
      pi -= 1/x;
    } else {
      pi += 1/x;
```

```
    }
  }
  pi *= 4;
  res.write('Pi = '+ pi);
  res.end();
}).listen(8080);
```

testPI.php

```
<?php

$x = 3;
$steps = 1000000;
$pi = (1 - 1 / $x);

for ($i = 0; $i < $steps; $i++) {
        $x += 2;
        if ($i % 2) {
                $pi -= 1/$x;
        } else {
                $pi += 1/$x;
        }
}

$pi *= 4;
echo "Pi = ". $pi;

?>
```

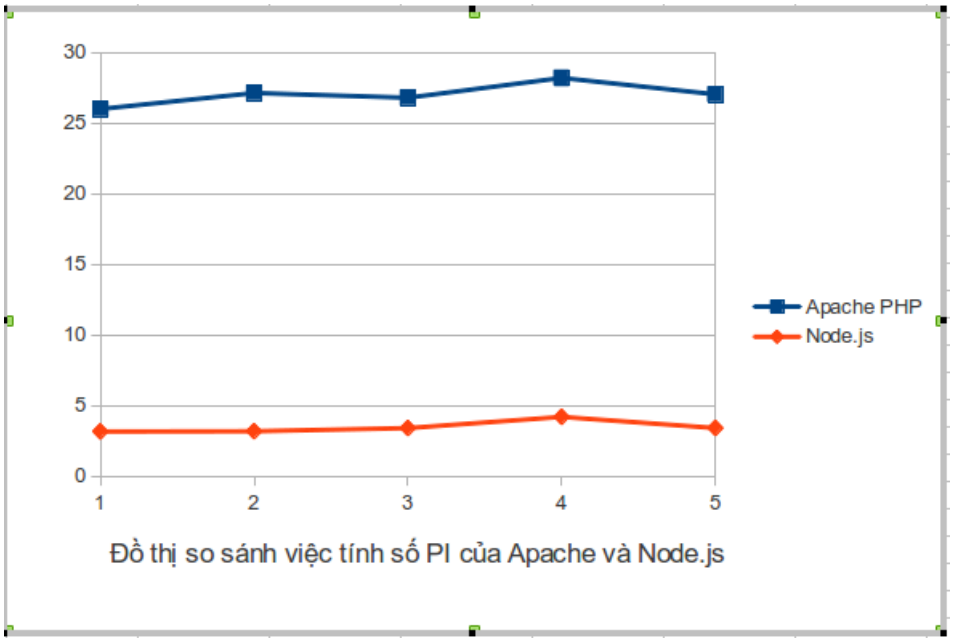
Sau quá trình test ta thu được kết quả trong bảng dưới đây:

x	1	2	3	4	5
Apache	26.001	27.134	26.801	28.201	27.040
NodeJs	3.181	3.198	3.423	4.214	3.426

Đồ thị minh họa:

1.2.4 Lập trình hướng sự kiện trên server

Với việc ra đời NodeJS, cũng dẫn đến một mô hình lập trình mới trên server. Đó là lập trình hướng sự kiện. Phía client, khái niệm này có thể không quá mới mẻ. Trong ngôn ngữ javascript, việc lập trình gắn liền với các sự kiện. Các lập trình viên sẽ tạo các sự kiện DOM và bắt các sự kiện này và sử lý. Với NodeJS, cách lập trình cũng như vậy. Ta cũng bắt các sự kiện từ phía client hoặc các sự kiện từ hệ thống và phản hồi. So với cách lập trình truyền thống, đây là một cách lập trình khá thú vị, mới mẻ.Để hiểu rõ vấn đề này hơn ta sẽ tìm hiểu các module được trình bày trong chương 3 của bài báo



Đồ thị so sánh việc tính số PI của Apache và Node.js

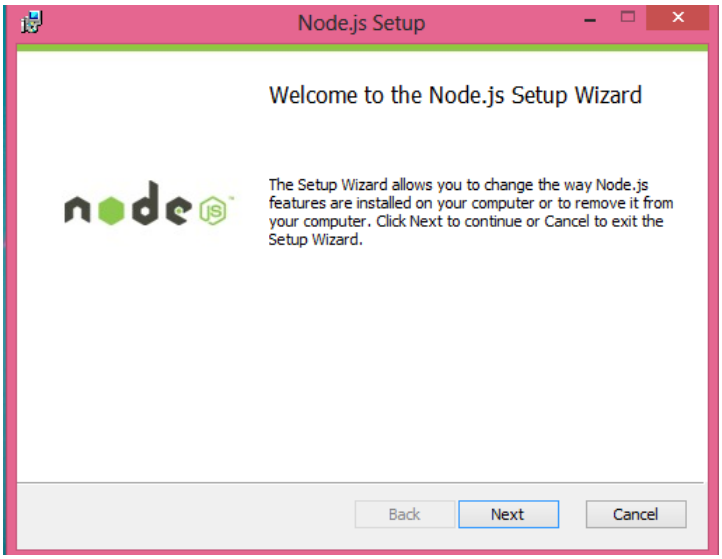
Chương 2

Cài đặt và quản lý module trong NodeJS

2.1 Cài đặt NodeJS

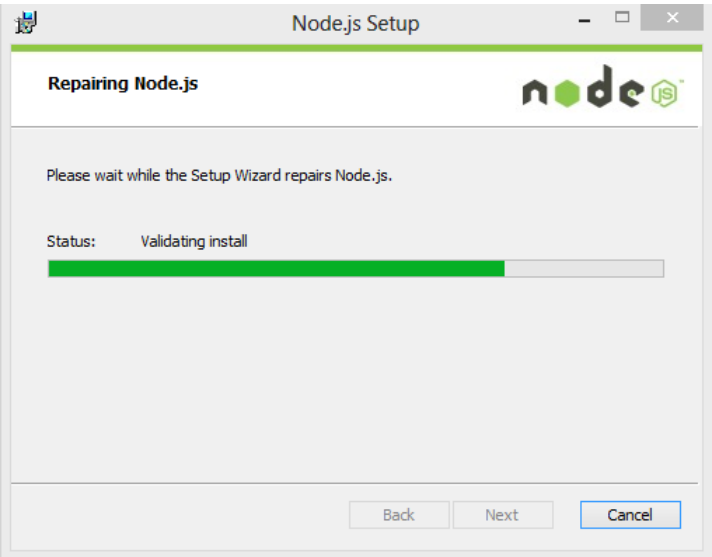
2.1.1 Cài đặt NodeJS

Để cài đặt NodeJS bạn có thể truy cập và trang http://nodejs.org/download/ để tải phiên bản phù hợp với hệ điều hành của bạn xuống. **Với Windows:**
Bạn chỉ cần chạy file *node-v0.8.18-x84.msi* hoặc *node-v0.8.18-x64.msi* tùy theo hệ điều hành đang sử dụng và tiến hành cài đặt bình thường.



Với Linux:
Sau khi tải về gói package của NodeJs ta tiến hành thực hiện các bước sau:

1. $ `xzf node-v0.8.18.tar.gz`
2. $ `cd node -v0.8.18`
3. $ `./configure`
4. $ `make`



5. `$ sudo make install`

Ngoài ra một số bản phân phối linux cho phép bản tải NodeJS thông qua package. Để thực hiện điều đó, bạn chỉ cần vào terminal và gõ lệnh:

```
$ apt-get install nodejs
```

Để kiểm tra máy đã cài nodeJS chưa ta thực hiện lệnh:

```
$ node -v
```

2.1.2 Sử dụng NodeJs

Sau khi tiến hành cài đặt xong NodeJs chúng ta bắt đầu sử dụng Node. Để khởi động giao diện dòng lệnh của Node ta sử dụng câu lệnh:

```
$ node
```

Tiến hành kiểm tra việc cài đặt và quan sát Node đang làm gì, ta sử dụng:

```
> console.log('Hello World!')
Hello World!
> undefined
```

Cũng có thể chạy một JavaScript từ một file. Nếu tạo chúng ta tạo một file `hello_world.js` với nội dung.

```
console.log('Hello World!')
```

Tiến hành file trên với câu lệnh:

```
$ node hello_world.js
Hello World
```

2.2 Quản lý module trong NodeJS

2.2.1 Quản lý module trong NodejS

Với NodeJs ta có thể mở rộng các chức năng hông qua các module. Để quản lý chúng, ta sử dụng một phần mềm của bên thứ 3 là *npm* (Node Package control management). Thông tin về *npm* có thể tìm thấy ở trang chủ : http://npmjs.org. Npm được cài sẵn khi cài Nodejs.

1. Để liệt kê các gói ta sử dụng lệnh
   `$ node ls [filter]`
   - Liệt kê tất cả các gói
     `$ node ls`
   - Liệt kê tất cả các gói được cài đặt
     `$ node ls installed`
   - Liệt kê tất cả các gói đang ở trạng thái ổn định(stable)
     `$ node ls stable`
   - Liệt kê tất cả các gói theo tên
     `$ node ls "tên hoặc pattern"`
2. Để cài đặt gói ta sử dụng lệnh:
   `$ npm install package[@filters]`
   - Cài gói express
     `$ npm install express`
   - Cài gói express phiên bản 2.0.0beta
     `$ npm install express@2.0.0beta`
   - Cài gói phiên bản lớn hơn 0.1.0
     `$ npm install express@">=0.1.0`
3. Để gỡ một gói , ta dùng lệnh
   `$ npm rm [-g] <package name>[@version]`
4. Để xem thông tin về module nào đó ta sử dụng lệnh.
   `$ npm view [@] [[.]...]`
5. Để update các module bằng việc sử dụng câu lệnh:
   `$ npm update [-g] <package name>`

2.3 Xây dựng và sử dụng các module

Trong phần này, ta sẽ tìm hiểu cách xây dựng và sử dụng một module cho nodeJS.

```
var module = require(path_to_module| module_name);
```

Tham số truyền vào là tên module nếu đó là một core module được quản lý qua npm. Trong trường hợp module đó là một module custom của người dùng, ta chỉ cần chi ra đường dẫn đến file đó.

Ngoài ra ta cũng có thể load các module trong một thư mục. Đầu tiên Nodejs sẽ kiểm tra xem đó có phải là một packge hay không thông qua kiểm tra một file package.json. Cấu trúc file json sẽ chứa thông tin về file chính của package. Ví dụ:

```
{
    "name": "myModule",
    "main": "./lib/myModule.js"
}
```

Khi đó, Node sẽ thử load file path_to_module/lib/myModule.js

Một điểm quan trọng cần lưu tâm là khi thực hiện thao tác này, ta đã cache module đó. Do vậy khi viết nhiều lần nạp module, thực chất chỉ có một hàm đã được thực hiện. Để truy cập vào các thành phần của module, module cần phải xuất thông tin đó ra thông qua lệnh module.exports.

Ví dụ:

```
1  //test.js
2  var level = 0;
3
4  function loadGame(){
5           //code in here
6  }
7
8  function playGame(){
9          //code in here
10 }
11
12 module.exports.loadGame = loadGame;
13 module.exports.playGame = payGame;
14 module.exports.level = level;
15
16 //app.js
17 var test = require('./test.js');
18 test.loadGame();
19 test.playGame();
20 console.log(test.level);
```

Trong ví dụ này, trong module test, ta đã xuất ra thông tin gồm 2 hàm và một biến. Chính vì vậy khi load module này vào file app.js, ta có thể sử dụng chúng.

Chương 3

Lập trình cơ bản NodeJS

3.1 Một số module làm việc vào ra

Node cung cấp một số module làm việc trực tiếp với file.

3.1.1 Module fs

Đây là module chính giúp ta có thể thao tác với tệp.Module này cung cấp các thao tác tệp bao gồm:

- Tạo lập, đổi tên, xóa tệp
- Thao tác trên tệp: mở tệp, đọc, ghi, đóng tệp
- Cập nhật thời gian truy cập
- Thay đổi quyền sở hữu và quyền truy cập (trên hệ thống linux)
- ...

Một điểm chú ý là NodeJS hỗ trợ asynchronous IO và synchronize IO với thao tác tệp. Ta dễ dàng nhận biết các lệnh này thông qua tên gọi. Các lệnh đọc file đồng bộ sẽ có 'Sync' trong tên. Ví dụ:

- readFile(filename: string, data: mixed, [encoding], [callback:function])
- readFileSync(filename: string, data: mixed, [encoding], [callback: function])

Ta cũng có thể kiểm tra thông tin về trạng thái tệp, thư mục nhờ module này. Một số hàm kiểm tra trạng thái.

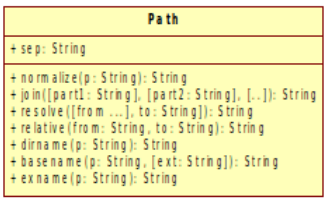
- fs.stat(filename: string, callback: function) : đọc thông tin trạng thái, khi đọc xong gọi hàm callback. Tham số hàm callback bao gồm lỗi (nếu có) và object chứa thông tin về file
- Các hàm kiểm tra kiểu file: Kiểm tra các kiểu file . Các kiểu file này tương ứng trong hệ thống POSIX:
  - isFile();
  - isDirrectory(): kiểm tra loại file là thư mục.
  - isBlockDevice(): kiểm tra loại file là thiết bị block.

- isCharacterDevice(): kiểm tra thiết bị thuộc loại đọc ký tự.
- isSymbolicLink(): file là link.
- isFIFO(): thiết bị đọc dạng FIFO.
- isSocket(): socket.

3.1.2 Module path

Module này chứa các tiện ích giúp xác định và biến đổi đường dẫn của tệp. Một số phương thức chính:

- resolve: xác định địa chỉ tuyệt đối file dựa trên địa chỉ tương đối tệp đó với một địa chỉ tuyệt đối
- join: nối địa chỉ với nhau

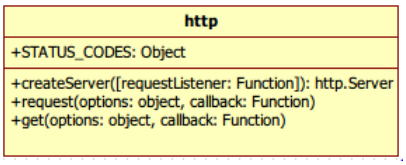


3.2 Một số module về mạng

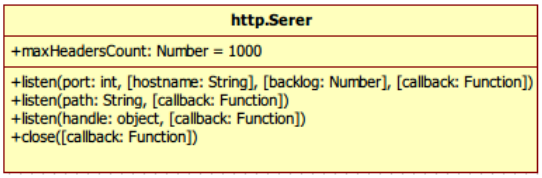
3.2.1 HTTP & HTTPS

Module HTTP là tập hợp các thao tác qua mạng . Module này gồm có các lớp:

- http.Server: lớp server
- http.ServerRequest : cho phép biểu diễn thông tin truy vấn tới server.
- http.ServerResponse : lớp biểu diễn các thông tin trả về từ server.
- http.ClientRequest: lớp biểu diễn yêu cầu dữ liệu.



HTTP là một trong những thành phần quan trọng nhất giúp Nodejs hoạt động như một Web Server. HTTP module cung cấp phương thức createServer giúp tạo ra một thể hiện của lớp HTTP Server. Ngoài ra có một số sự kiện và cấu trúc dữ liệu để thực hiện các callback.



http.Server

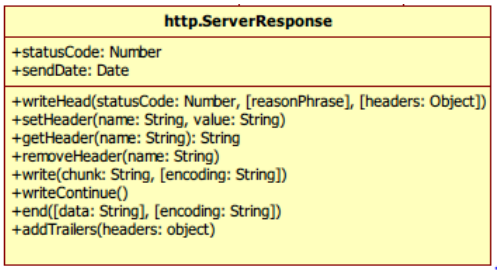
Phương thức listen trong lớp này thiết lập thông tin cho server như cổng, tên host, đường dẫn,.. Phương thức close nhằm đóng lại server. Ngoài ra một http Server sẽ có các sự kiện cơ bản sau:

- 'request' : Sự kiện sẽ được kích hoạt mỗi khi có một yêu cầu (request) tới server. Tham số của hàm callback gồm :
  - req : một thể hiện của lớp http.serverRequest, chứa thông tin yêu cầu từ máy client
  - res : một thể hiện của lớp http.serverResponse, chứa thông tin trả về do máy chủ tạo ra
- 'connection': Sự kiện được kích hoạt khi có một luồng dữ liệu TCP xuất hiện. Chú ý rằng với một liên kết TCP có thể gồm nhiều request.
- 'close': Khi server đóng
- 'checkContinue': Sự kiện được kích hoạt khi có 100 kết nối liên tiếp. Mục đích của sự kiện này giúp lập trình viên tăng khả năng điều khiển . Mặc định, nếu sự kiện này không được cài đặt, 100 kết nối này sẽ được đáp ứng.
- 'connect' : Sự kiện được kích hoạt khi có một kết nối http CONNECT .Nếu sự kiện này không được lắng nghe, mặc định các kết nối sẽ bị đóng lại. Tham số truyền vào callback gồm:
  - request: tham số của request.
  - socket: socket giữa server và client.
  - head: một thể hiện của lớp Buffer, gói đầu tiên của .., thường được để trống.
- 'upgrade' : sự kiện được kích hoạt khi client có yêu cầu http upgrade.
- 'clientError': sự kiện được kích hoạt khi kết nối server và client thất bại.

3.2.2 Lớp http.ServerRequest

Lớp này được tạo ngầm định bởi server, được truyền như một tham số như ta đã thấy trong sự kiện 'request' của một Server. Lớp này thực thi giao diện của Readable Stream kế thừa EventEmitter. Các sự kiện cơ bản của lớp này :

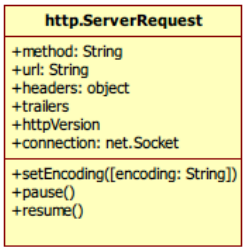
- 'data': sự kiện được kích hoạt khi nhận được các đoạn tin. Các đoạn tin này là một xâu nếu kiểu mã hóa xác định thông qua phương thức setEncoding hoặc một Buffer nếu không xác định.
- 'end': sự kiện được kích hoạt với ý nghĩa không còn gói tin nào nữa



- 'close': sự kiện báo đóng kết nối

Ngoài ra lớp này còn một số trường giúp cung cấp thêm các thông tin về request:

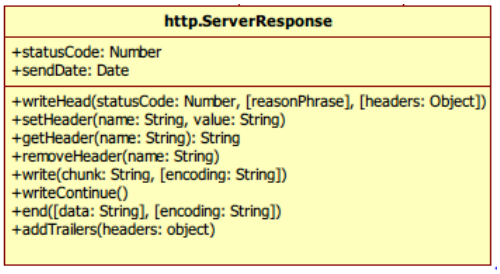
- method: phương thức yêu cầu : GET/POST/..
- url: xâu biểu diễn các yêu cầu (ví dụ: '?id='1'& type=2')
- httpVersion: phiên bản version
- trailer: HTTP trailers
- headers: đối trượng chứa tất cả các thông tin trường header của request



Lớp http.serverResponse

Dây là lớp được tạo ra do server khi có sự kiện 'request'. Cấu trúc lớp nà được thể hiện như hình bên. Trong đó, một số phương thức quan trọng:

- writeHeader: thiết lập thông tin header trả về cho client
- write : viết thông tin trả dữ liệu trả về cho client.
- End : tín hiệu báo đã truyền xong dữ liệu cho server



http Client

Một trong các đặc điểm quan trọng của Nodejs là phía server có thể thực hiện các yêu cầu HTTP tới các địa chỉ khác như một máy khách. Điều này đặc biệt hữu ích trong trường hợp muốn duy trì kết nối giữa các server.

Để tạo một yêu cầu HTTP ta thực hiện thông qua lệnh :

```
http.request(options: String, callback: function);
```

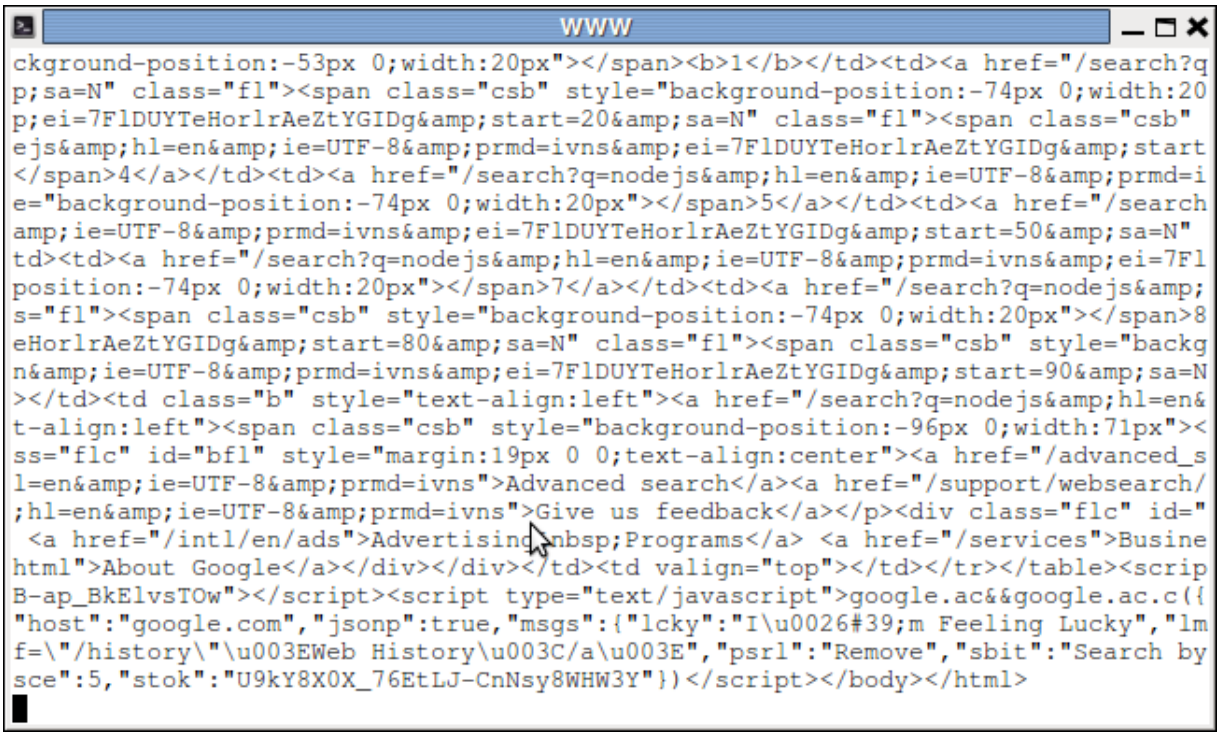
Options là một xâu hoặc một đối tượng. Trong trường hợp là đối tượng nó sẽ được tự động chuyển thông qua hàm url.parse(). Một số tùy chọn cho option:

- host: tên miền hoặc IP muốn kết nối đến. Mặc định, tên miền là 'localhost'
- port: cổng kết nối
- method: phương thức HTTP (GET, POST,..). Mặc định phương thức kết nối là 'GET'
- path: đường dẫn yêu cầu
- headers: các thông tin headers
- Agent: user-agent của máy. Đây là một thể hiện của lớp http.Agent

Ví dụ sau minh họa cho việc thực hiện một yêu cầu tìm kiếm tới Google:

```
var http = require('http');
var query = ' Nodejs Information';
var opts = {
    host : 'google.com',
    port : 80,
    path: '/search?q=' + query,
    method: 'GET'
};

var req = http.request(opts, function(res){
    console.log(res); //display response information
    res.setEncoding('utf8');
    res.on('data', function(data){
        console.log(data);
    }
});

req.end(); //send request
```

Đây là kết quả khi thực hiện xong chương trình: Ngoài ra bạn cũng có thể sử dụng http.get() trong trường hợp phương thức HTTP chỉ là GET, cách sử dụng cũng tương tự như trên.



3.2.3 Net

Module net là lớp bao của module http. Module này cho phép thiết lập và tạo các TCP server. Nó chứa đựng các phướng thức cho việc tạo cả một server và client. Chúng ta có thể sử dụng module này với **require('net');**
Một số phương thức của module Net: net.createServer([options], [connectionListener]) Tạo ra một TCP server mới. Thamm số connectionListener được tự động thiết lập việc bắt sự kiện "connection". options là một đối tượng với mặc định

true

```
var net = require('net');
var server = net.createServer(function(c) { //'connection' listener
  console.log('server connected');
  c.on('end', function() {
    console.log('server disconnected');
  });
  c.write('hello\r\n');
  c.pipe(c);
});

server.listen(8124, function() { //'listening' listener
  console.log('server bound');
});
```

net.connect(options, [connectionListener])

net.createConnection(options, [connectionListener])

Xây dựng một đối tượng socket mới và mở socket nhất định. Khi socket được thành lập, sự kiện 'connect' sẽ được sinh ra.

Với TCP sockets, đối số options được đối tượng đặc tả:

- **port**: Port của client để kết nối (Required).
- **host**: Host của client để kết nối tới, mặc định là 'localhost'.
- **localAddress**: Giao diện địa phương để ràng buộc các kết nối.

Đây là một ví dụ:

```
var net = require('net');
var client = net.connect({port: 8124},
    function() { //'connect' listener
  console.log('client connected');
  client.write('world!\r\n');
});

client.on('data', function(data) {
  console.log(data.toString());
  client.end();
});

client.on('end', function() {
  console.log('client disconnected');
});
```

net.connect(options, [host], [connectionListener])

net.createConnection(options, [host], [connectionListener])

Tạo ra một TCP kết nối từ `port` trong `host`. Nếu `host` bị bỏ qua, `'localhost'` sẽ được chọn giả định. Tham số connectListener sẽ được thêm vào như việc bắt sự kiện cho sự kiện `"connect"`

net.connect(path, [connectListener])

net.createConnection(path, [connectListener])

Tạo ra unix socket kết nối theo `path`. Tham số connectListener sẽ được thêm vào như việc bắt sự kiện cho sự kiện `"connect"`.

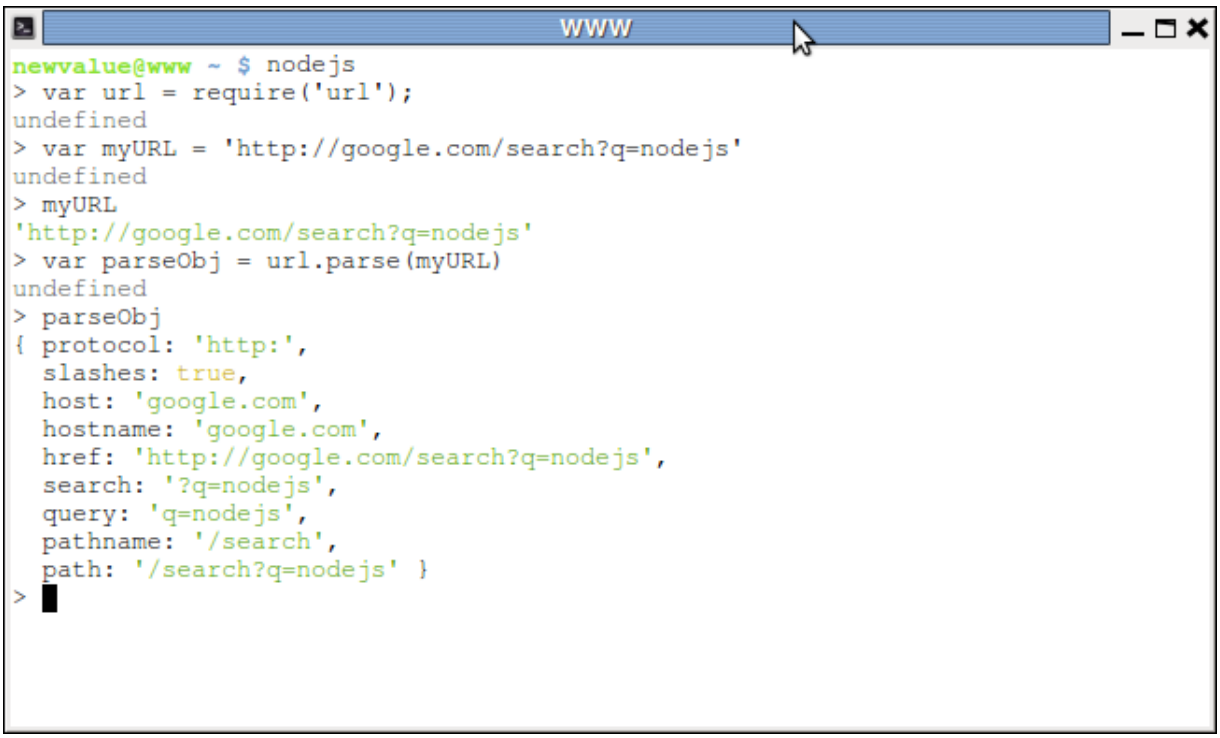
3.2.4 URL & QueryString

Đây là 2 module tiện ích hỗ trợ cho việc xử lý URL và các truy vấn.
Module URL giúp cho việc xử lý URL. Với module này ta có thể tách xâu URL thành đối tượng biểu diễn thông tin: host, hostname, querystring...
URL cung cấp 3 phương thức để xử lý xâu:

- parse: Chuyển từ xâu truy vấn thành một đối tượng chứa thông tin về url. Các thông tin đó gồm:
  - protocol : giao thức kết nối

  - hostname : tên host, sử dụng khi không có thông tin trường host
  - host: thay thế cho hostname và port
  - pathname: đường dẫn
  - search: thành phần biểu diễn yêu cầu nội dung tìm kiếm là query
  - query: nội dung tìm kiếm
  - hash: phần thông tin bắt đầu bởi dấu #
- format: chuyển từ đối tượng thành url
- resolve(baseURL: String, href: String) : xây dựng xâu biểu diễn url từ 2 thành phần based URL và href. Ví dụ ta có xâu cơ bản "http://google.com/test" và href là "/search?q=nodejs", xâu trả về sẽ là "http://google.com/search?q=nodejs"

Ta có ví dụ cho việc chuyển đổi xâu truy vấn:



Lớp QueryString cung cấp các công cụ giúp xử lý xâu truy vấn. Lớp này cung cấp một số phương thức cơ bản sau:

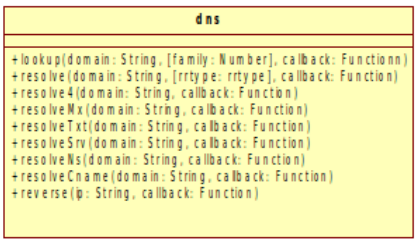
- **stringify(obj: Object, [sep: String], [eq])** : Chuyển một đối tượng biểu diễn truy vấn thành một xâu. Mặc định Node sử dụng dấu ngăn cách giữa các truy vấn là dấu '&' và biểu diễn gán giá trị bằng dấu '='. Ta có thể tùy chỉnh các dấu này nếu cần
- **parse(str,[sep: String],[eq: String],[options: Object])**: Chuyển xâu thành đối tượng biểu diễn truy vấn
- **phương thức escape() và unescape()**: 2 phương thức này hoạt động như phương thức stringify và parse nhưng có thể được ghi đè nếu cần thiết.

Ta có ví dụ sau:

```
querystring.parse('foo=bar&baz=qux&baz=quux&corge')

// returns

{ foo: 'bar', baz: ['qux', 'quux'], corge: '' }

querystring.stringify({ foo: 'bar', baz: ['qux', 'quux'], corge: '' })

// returns

'foo=bar&baz=qux&baz=quux&corge='
```

3.2.5 DNS

Chức năng module này là giúp phân giải tên miền từ tên miền sang địa chỉ IP và ngược lại. Module này hỗ trợ cả phân giải dựa trên IPV4 và IPV6.



3.3 Một số module công cụ

3.3.1 Console

Đây là module toàn cục, tức ta không cần phải nạp module để chạy được các hàm mà nó cung cấp. Module này cung cấp một đối tượng 'console' giúp ta có thể in ra các thông tin. Ví dụ:

```
console.log('hello world');
```

Trong trường hợp muốn ghi thông báo lỗi, ta sử dụng lệnh warn:

Để in dấu vết, ta dùng hàm :

```
console.trace();
```

3.3.2 Util

Module này chức năng tương tự như console nhưng mạnh mẽ hơn. Một số hàm thông dụng:

- util.format(format,[]): định dạng văn bản hiển thị ra tương tự như hàm printf trong C
- util.debug(string: str): block tiến trình hiện tại và in xâu str ra stderr
- util.inspect(obj: object, [option: object]): trả về biểu diễn xâu của đối tượng. Đây là một công cụ hữu hiệu trong trường hợp muốn debug thông tin.Một số tùy chọn khi gọi hàm này:
  - showHidden: thiết lập hiển thị với các đối tượng không có tính chất liệt kê
  - depth: giới hạn độ xâu kiểm tra
  - colors: hiển thị màu sắc cho kết quả
- util.isArray: dùng để kiểm tra đối tượng truyền vào có phải là một mảng hay không

3.3.3 Event Emitter

Trong Node có nhiều đối tượng có thể sinh ra nhiều sự kiện. Chẳng hạn như, một TCP server có thể sinh ra một sự kiện "connect" trong mỗi lần mấy client kết nối tới.

emitter.addListener(event, listener)

emitter.on(event, listener)

Thêm một listeners vào cuối mảng listener cho sự kiện được đặc tả.

```
server.on('connection', function (stream) {
  console.log('someone connected!');
});
```

emitter.once(event, listener)

Nếu muốn bắt sự kiện về liên kết tới server, bạn sử dụng:

```
server.once('connection', function (stream) {
  console.log('Ah, we have our first user!');
});
```

emitter.removeListeners(event, listener)

Gỡ bỏ một listener từ mảng listener mà đặc tả cho sự kiện.

```
var callback = function(stream) {
  console.log('someone connected!');
};
server.on('connection', callback);
// ...
server.removeListener('connection', callback);
```

emitter.removeAllListener([event])

Nếu bạn cần, bạn có thể gỡ bỏ việc bắt sự kiện từ Event Emitter bởi một lời gọi đơn giản

```
server.removeAllListeners('connection');
```

Tạo một Event Emitter

```
var EventEmitter = require('events').EventEmitter,
    util          = require('util');

// Here is the MyClass Contructor
var MyClass = function(option1, option2) {
    this.option1 = option1;
    this.option2 = option2;
}

util.inherits(MyClass, EventEmitter);
```

Trong trường hợp của này, MyClass có thể sinh ra các sự kiện:

```
MyClass.prototype.someMethod = function() {
    this.emit('custom event', 'some arguments');
}
```

```
var myInstance = new myClass(1, 2);
myInstance.on('custom event', function() {
    console.log('got a custom event!');
});
```

3.3.4 Buffer

Buffer là đối tượng cơ bản quan trọng nhất trong NodeJS. Khi truyền dữ liêu qua mạng, dữ liệu được xử lý dưới dạng nhị phân. Tuy nhiên, trong ngôn ngữ Javascript mặc định không có kiểu dữ liệu này. NodeJS đã thêm kiểu dữ liệu này. Khi dữ liệu gửi đến, mặc định là dữ liệu nhị phân. Để có thể chuyển sang dữ liệu văn bản, cần thiết lập kiểu mã hóa của văn bản. Ví dụ nếu văn bản mã hóa utf8, ta cần phải xác định kiểu mã hóa:

```
        buf.setEncoding('utf8');
```

Để khởi tạo một Buffer từ xâu, ta làm như sau:

```
        buf = new Buffer(str: string, [encoding]);
```

Ví dụ ta muốn tạo Buffer biểu diễn xâu "hello world" :

```
        var buf = new Buffer("hello world");
```

Trên kiểu Buffer ta có một số phương thức sau:

- **write(string: string, [offset: Number], [length: Number], [encoding: string])**: ghi dữ liệu vào xâu.

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

- **toString([encoding],[start], [end])**: Trả về xâu biểu diễn Buffer tương ứng.
- **toJSON()**: trả về đối tượng JSON.

```
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>
```

- **byteLength(string: string, [encoding])**: trả về độ dài xâu

```
// example byteLength method
str = '\u00bd + \u00bc = \u00be';

console.log(str + ": " + str.length + "characters, " +
    Buffer.byteLength(str, 'utf8') + " bytes");
```

- **concat(list: array, [totalLength: Number])**: Nối các buffer vào buffer cũ.
- **slice([start:Number],[end: Number])**: Trả về một Buffer mới. Buffer này tham chiếu đến cùng vùng nhớ với Buffer cũ nhưng bị giới han bởi chỉ số start và end.

```
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));
```

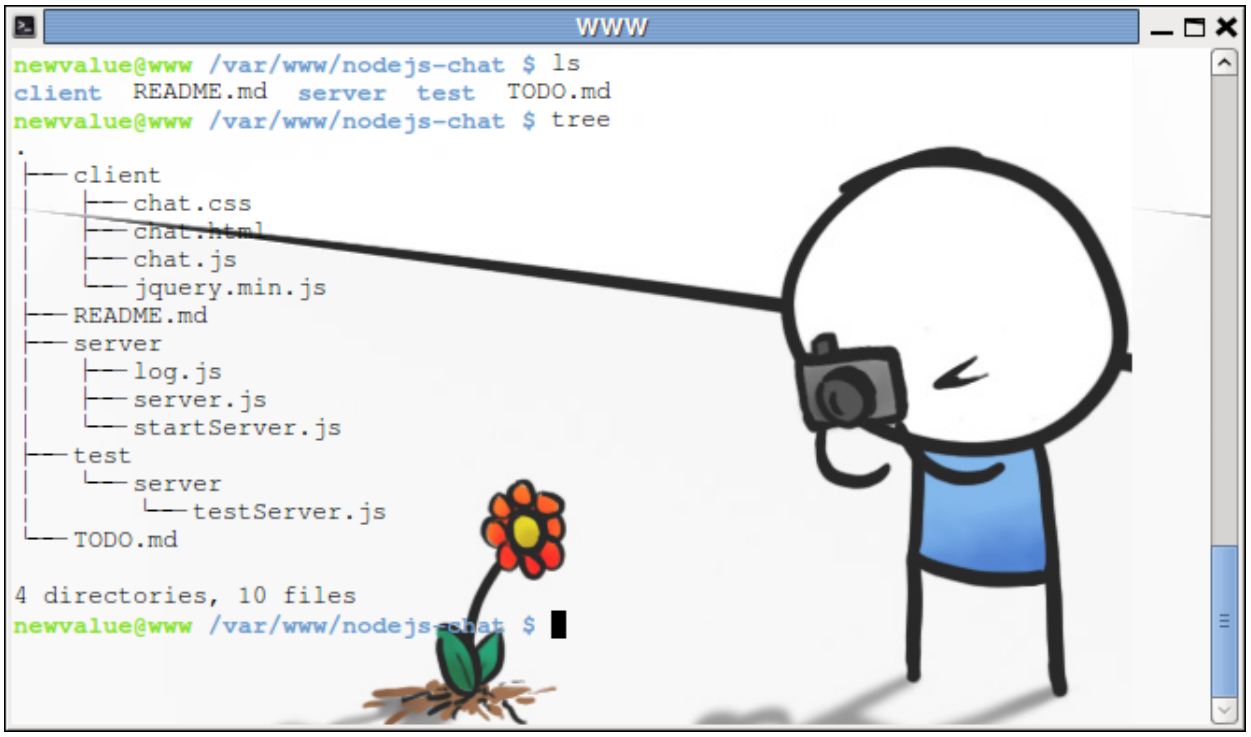
- **fill(value: Number,[offset: Number],[end: Number])**: điền đầy Buffer với dữ liệu value.

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

Chương 4

Ví dụ minh họa với NodeJS

Trong phần này, ta hãy cùng tìm hiểu một ứng dụng chat đơn giản. Mã nguồn của chương trình có thể tìm thấy tại địa chỉ:
http://github.com/
Cấu trúc chương trình được thể hiện qua hình dưới đây:



Tên file/Thư mục	Miêu tả
Client	Thư mục hiển thị ngươi dùng
client/server.js	Quản lý giao tiếp giữa client và server
Server/	Chứa mã nguồn server
Server/server.js	Module Server
server/log.js	Module log: quản lý thông báo, ghi chép
server/startServer.js	Tạo một thể hiện của server và chạy
test/server/test.js	Test

4.1 Module Server

Đây là module chính của chương trình.Nhiệm vụ của nó là khởi tạo một server lắng nghe và phản hồi các kết nối.
Trong ví dụ này, để thực hiện kết nối giữa server và client, ta sẽ sử dụng WebSocket – một công nghệ mới cho phép tạo các luồng socket trên nền HTTP. Do vậy tta phải khởi tạo một TCP Server:

```
this.server = tcp.createServer(function(socket){

});
        server.listen(7000);
```

Để có thể quản lý các kết nối từ client, ta tạo một mảng lưu trữ lại các kết nối.

```
this.connections = []
...
var self = this;
this.server = tcp.createServer(function(socket){
    var connection = new module.Connection(self, socket);
    self.addConnection(connection);
});

this.addConnection = function(connection){
    self.connctions.push(connection);
    connection.id = self.nextIdAssign++;
}
```

Khi có một yêu cầu từ phía client gửi đến, ta sẽ gửi broadcast các tin nhắn tới các user khác:

```
//gui tin nhan tu client
this.say = function(){
    self.broadcast(function(conn){
        return ['said', {
            'is self': (conn.id == connection.id),
            'message': message,
            'nick': connection.nick
        }
    });
};
//broad cast
this.broadcast = function(fn){
    self.connection.each(function(connection){
        var data = fn(connection);
        connection.send(data);
    }
}
```

Khi client gửi yêu cầu hủy kết nối, ta phải loại bỏ kết nối của client ra khỏi danh sách kết nối:

```
this.removeConnection = function(connection){
    self.connection.remove(connection);
    log.debug('Connection: ' + self.connections.length);
}
```

Bây giờ ta cùng xét đên lớp kết nối. Mục đích lớp này để quản lý kết nối. Một thể hiện của lớp này cần lưu trữ thông tin server quản lý nó, socket, id và nick đang kết nối tới. Id này do lớp Server đánh số. Trước hết nó được dùng để gửi lại tin nhắn do server nhận được ở trên.

Ngoài ra trong ví dụ này, ta sẽ cho phép người dùng đăng ký tên để sử dụng. Để tiện cho việc này, ta viết thêm phương thức quản lý yêu cầu. Ta sẽ quản lý 2 loại yêu cầu:

- Đăng ký nick
- Gửi nhận tin nhắn

```
this.Connection.prototype.handlers = {
    assignNick: function(connection, data) {
        log.debug(connection.logFormat('Assigning nick: ' + data));
        connection.nick = data;

        connection.send(['lastMessage', connection.server.lastNMessages(5)]);
        connection.server.broadcast(function(conn) {
            return ['status', { 'nick': connection.nick, 'status': 'on' }];
        })
    },

    say: function(connection, data) {
        connection.server.say(connection, data);
    }
};
```

4.2 Module Client

Để thực hiện kết nối với server ta sẽ sử dụng WebSocket để tạo một socket yêu cầu tới server:

. frame

```
NodeJSChat.ws = new WebSocket('ws://localhost:7000')
NodeJSChat.ws.onmessage = NodeJSChat.onmessage;
NodeJSChat.ws.onclose = NodeJSChat.onclose;
NodeJSChat.ws.onopen = NodeJSChat.onopen;
```

Từ đó ta sẽ xây dựng các hàm callback tương ứng với từng sự kiện:

```
,

    onclose: function() {
        NodeJsChat.debug('socket closed');
        if (!NodeJsChat.windowClosing) {
            alert('Whoops! Looks like you lost internet
            connection or the server went down');
    }
},
```

```
onmessage: function(evt) {
    var jsonData = JSON.parse(evt.data);

    var action  = jsonData[0];
    var data    = jsonData[1];

    if (NodeJsChat.handlers[action]) {
      NodeJsChat.handlers[action](data);
    }
    else {
      // This should be in a separate function.
      $('#messages').append('<tr><td cospan="2">Whoops, bad message
                    from the server</td></tr>');
    }
},

onopen: function() {
    NodeJsChat.debug('Connected...');
    NodeJsChat.send([ 'assignNick',  NodeJsChat.nick ]);
},
```

Khi người dùng nhấn gửi tin, ta sẽ sử dụng socket này để gửi tới server:

```
say: function(message) {
    NodeJsChat.send([ 'say',  message ]);
    return false;
},

send: function(data) {
    var dataStr = JSON.stringify(data);
    NodeJsChat.ws.send(dataStr);
},
```

Cuối cùng, ta xây dựng một phương thức điều khiển chung cho mọi yêu cầu từ client.Các yêu cầu gồm có:

- Cập nhật các tin nhắn gần nhất.
- Ghi nhận trạng thái hiện tại.

```
said: function(data) {
    var message = '<div class="message">';

    var prefix  = '';
    var tdClass = '';
    if (data.isSelf) {
      prefix  = 'You:';
      tdClass = 'me';
    }
    else {
      prefix  = data.nick + ':';
      tdClass = 'nick';
    }
    message     += '<div class="' + tdClass + '">';
    message     += prefix
    message     += '</div>';

    message     += '<div class="message">';
    message     += data.message;
    message     += '</div>';

    message     += '</div>';

    $('#messages').append(message);
  },

  status: function(data) {
    var nick = data.nick;
    var stat = data['status'];

    var message = '<div class="status">' + nick
            + ' just logged ' + stat + '</div>';
    $('#messages').append(message);
  }
};
```