# Lời nói đầu

Internet không còn là một tập hợp các trang web tĩnh được sử dụng một cách thụ động. Người dùng trình duyệt đã mong đợi một trải nghiệm tương tác phong phú hơn nhiều. Trong hơn một thập kỷ qua, các ứng dụng mạng đã trở nên giống với các ứng dụng máy tính để bàn. Ngoài ra, việc nhận biết các đặc điểm xã hội của thông tin đã truyền cảm hứng cho sự phát triển của các loại giao diện và hình ảnh hóa mới mô hình hóa trạng thái mạng động, nơi người dùng đang xem sự thay đổi theo thời gian thực chứ không phải làm mờ dần các bức ảnh chụp nhanh bị mắc kẹt trong quá khứ.

Mặc dù kỳ vọng của chúng tôi đối với phần mềm đã thay đổi, các công cụ có sẵn cho chúng tôi với tư cách là nhà phát triển phần mềm không thay đổi nhiều. Máy tính nhanh hơn và kiến trúc chip đa lõi là phổ biến. Lưu trữ dữ liệu rẻ hơn, cũng như băng thông. Tuy nhiên, chúng tôi tiếp tục phát triển với các công cụ được thiết kế trước các trang web hàng tỷ người dùng và quản lý nút nhấn của các cụm máy ảo dựa trên đám mây.

Vì lý do này, việc phát triển các ứng dụng mạng vẫn là một quá trình quá tốn kém và chậm chạp. Các nhà phát triển sử dụng các ngôn ngữ, phong cách lập trình khác nhau, phức tạp hóa việc bảo trì mã, gỡ lỗi và hơn thế nữa. Quá thường xuyên, các vấn đề về mở rộng quy mô đến quá sớm, lấn át khả năng của những người thường là một nhóm nhỏ và thiếu kinh nghiệm. Các tính năng phần mềm hiện đại phổ biến, chẳng hạn như dữ liệu thời gian thực, trò chơi nhiều người chơi và không gian chỉnh sửa cộng tác, hệ thống yêu cầu có khả năng mang hàng nghìn kết nối đồng thời mà không bị bẻ cong. Tuy nhiên, chúng tôi vẫn bị giới hạn trong các khuôn khổ được thiết kế để hỗ trợ chúng tôi trong việc xây dựng các ứng dụng CRUD ràng buộc một cơ sở dữ liệu quan hệ duy nhất trên một máy chủ với một người dùng đang chạy một trang web nhiều trang trong trình duyệt trên máy tính để bàn.

Node giúp các nhà phát triển xây dựng các ứng dụng mạng linh hoạt hơn trên quy mô lớn. Được xây dựng trên C ++ và đi kèm với công cụ V8 của Google, Node rất nhanh và nó hiểu JavaScript. Node đã tập hợp ngôn ngữ lập trình phổ biến nhất trên thế giới và trình biên dịch JavaScript nhanh nhất hiện nay, đồng thời cho phép nhóm đó dễ dàng truy cập vào hệ điều hành thông qua các liên kết C ++. Node đại diện cho một sự thay đổi trong cách phần mềm mạng được thiết kế và xây dựng.

Cuốn sách này bao gồm những gì?

*Chương 1*, Tìm hiểu Môi trường Nút, đưa ra mô tả ngắn gọn về các vấn đề cụ thể mà Node cố gắng giải quyết, tập trung vào cách vòng lặp sự kiện đơn luồng của nó được thiết kế, triển khai và sử dụng. Chúng ta cũng sẽ tìm hiểu về cách động cơ V8 của Google có thể được định cấu hình và quản lý, cũng như các phương pháp hay nhất khi xây dựng các chương trình Node.

*Chương 2*, Tìm hiểu về lập trình theo hướng sự kiện không đồng bộ, đi sâu vào đặc điểm cơ bản trong thiết kế của Node: lập trình hướng sự kiện, không đồng bộ. Đến cuối chương này, bạn sẽ hiểu cách sử dụng các sự kiện, lệnh gọi lại và bộ định thời trong Node, cũng như cách thức hoạt động của vòng lặp sự kiện để kích hoạt I / O tốc độ cao trên các hệ thống tệp, mạng và quy trình.

*Chương 3*, Truyền dữ liệu qua các nút và máy khách, mô tả cách các luồng dữ liệu I / O được đan qua hầu hết các phần mềm mạng, được phát ra bởi các máy chủ tệp hoặc phát sóng để đáp ứng yêu cầu HTTP GET. Ở đây chúng ta tìm hiểu cách Node tạo điều kiện cho việc thiết kế, triển khai và cấu thành phần mềm mạng, sử dụng các ví dụ về máy chủ HTTP, các luồng tệp có thể đọc và ghi cũng như các mô-đun và mẫu Node tập trung I / O khác.

*Chương 4*, Sử dụng Node để Truy cập Hệ thống Tập tin, đưa ra những điều bạn cần biết khi truy cập vào nút hệ thống tệp, cùng với các kỹ thuật xử lý tải lên tệp và các hoạt động tệp được nối mạng khác.

*Chương 5*, Quản lý Nhiều Kết nối Máy khách Đồng thời, chỉ cho bạn cáchNode trợ giúp trong việc giải quyết các vấn đề đi kèm với môi trường khối lượng lớn, đồng thời cao mà các ứng dụng web hiện đại, cộng tác yêu cầu. Thông qua các ví dụ, tìm hiểu cách theo dõi hiệu quả trạng thái người dùng, định tuyến các yêu cầu HTTP, xử lý các phiên và xác thực các yêu cầu bằng cách sử dụng cơ sở dữ liệu Redis và khung ứng dụng web Express.

*Chương 6*, Tạo ứng dụng thời gian thực, khám phá AJAX, Server-Sent-Events và giao thức WebSocket, thảo luận về ưu và nhược điểm của chúng và cách triển khai từng giao thức bằng Node. Chúng tôi kết thúc chương bằng cách xây dựng một ứng dụng chỉnh sửa tài liệu cộng tác.

*Chương 7*, Sử dụng nhiều quy trình, dạy cách phân phối các cụm quy trình Node trên các bộ xử lý đa lõi và các kỹ thuật khác để mở rộng ứng dụng Node. Một cuộc điều tra về sự khác biệt giữa lập trình trong môi trường đơn và đa luồng dẫn đến một cuộc thảo luận về cách sinh sản, phân nhánh và giao tiếp với các quy trình con trong Node và chúng tôi xây dựng một công cụ phân tích ghi lại và hiển thị các hành động chuột của nhiều khách hàng được kết nối thông qua một cụm ổ cắm web.

*Chương 8*, Mở rộng ứng dụng của bạn, phác thảo một số kỹ thuật để phát hiện thời điểm mở rộng quy mô, quyết định cách mở rộng quy mô và mở rộng các ứng dụng Node trên nhiều máy chủ và dịch vụ đám mây, với các ví dụ bao gồm: cách sử dụng RabbitMQ làm hàng đợi tin nhắn, sử dụng NGINX để proxy Node máy chủ và sử dụng Amazon Web Services trong ứng dụng của bạn.

*Chương 9*, Kiểm tra ứng dụng của bạn, giải thích cách triển khai các bài kiểm tra đơn vị, chức năng và tích hợp với Node. Chúng ta sẽ khám phá một số thư viện thử nghiệm, bao gồm xác nhận Node gốc, hộp cát và mô-đun gỡ lỗi. Ví dụ sử dụng Grunt, Mocha, PhantomJS và các công cụ xây dựng và thử nghiệm khác đi kèm với cuộc thảo luận.

*Phụ lục A*, Tổ chức công việc của bạn, đưa ra các mẹo sử dụng hệ thống quản lý gói npm. Tìm hiểu cách tạo gói, xuất bản gói và quản lý gói.

*Phụ lục B*, Giới thiệu Khung đường dẫn, trình bày cách sử dụng khung ứng dụng đầy đủ mạnh mẽ này để xây dựng ứng dụng web tiếp theo của bạn chỉ bằng JavaScript, nhờ Node và khả năng xử lý hàng nghìn ứng dụng khách được kết nối đồng thời.

*Phụ lục C*, Tạo tiện ích bổ sung C ++ của riêng bạn, cung cấp giới thiệu ngắn gọn về cách xây dựng các tiện ích bổ sung C ++ của riêng bạn và cách sử dụng chúng từ Within Node.

Những gì bạn cần cho cuốn sách này:

Bạn sẽ cần phải làm quen với JavaScript và đã cài đặt một bản sao của Node trên máy phát triển hoặc máy chủ của mình, Phiên bản 0.10.21 trở lên. Bạn nên biết cách cài đặt các chương trình trên máy này, vì bạn sẽ cần cài đặt Redis, cùng với các thư viện khác, như PhantomJS. Việc cài đặt Git và học cách sao chép kho GitHub sẽ cải thiện đáng kể trải nghiệm của bạn.

Bạn nên cài đặt RabbitMQ để có thể làm theo các ví dụ bằng cách sử dụng hàng đợi tin nhắn. Các phần về cách sử dụng NGINX cho máy chủ Node proxy tất nhiên sẽ yêu cầu bạn có thể cài đặt và sử dụng máy chủ web đó. Để xây dựng các tiện ích bổ sung C ++, bạn sẽ cần cài đặt trình biên dịch thích hợp trên hệ thống của mình.

Các ví dụ trong cuốn sách này được xây dựng và thử nghiệm trong môi trường dựa trên UNIX (bao gồm cả Mac OS X), nhưng bạn cũng có thể chạy tất cả các ví dụ Node trên hệ điều hành dựa trên Windows. Bạn có thể có được trình cài đặt cho hệ thống của mình, và nhị phân, từ http://www.nodejs.org.

Cuốn sách này dành cho ai

Cuốn sách này dành cho các nhà phát triển muốn xây dựng các ứng dụng mạng dung lượng cao, chẳng hạn như mạng xã hội, môi trường chỉnh sửa tài liệu cộng tác, giao diện web theo hướng dữ liệu thời gian thực, trò chơi nối mạng và các phần mềm I / O-nặng khác. Nếu bạn là nhà phát triển JavaScript phía máy khách, đọc cuốn sách này sẽ dạy bạn cách trở thành

một lập trình viên phía máy chủ sử dụng ngôn ngữ bạn đã biết. Nếu bạn là một hacker C ++, Node là một dự án mã nguồn mở được xây dựng bằng ngôn ngữ đó, mang đến cho bạn cơ hội tuyệt vời để tạo ra tác động thực sự trong một cộng đồng lớn và đang phát triển, thậm chí còn nổi tiếng bằng cách giúp phát triển công nghệ mới thú vị này.

Cuốn sách này cũng dành cho các nhà quản lý kỹ thuật và những người khác đang tìm kiếm lời giải thích về các khả năng và triết lý thiết kế của Node. Cuốn sách chứa đầy các ví dụ về cách Node giải quyết các vấn đề mà các công ty phần mềm hiện đại đang gặp phải về tính đồng thời cao, các ứng dụng thời gian thực đẩy khối lượng dữ liệu khổng lồ thông qua các mạng ngày càng phát triển. Node đã được doanh nghiệp chấp nhận và bạn nên xem xét nó cho dự án tiếp theo của mình.

**Quy ước:**

Trong cuốn sách này, bạn sẽ tìm thấy một số kiểu văn bản phân biệt giữa các loại thông tin khác nhau. Dưới đây là một số ví dụ về các phong cách này và giải thích về ý nghĩa của chúng.

Các từ mã trong văn bản, tên bảng cơ sở dữ liệu, tên thư mục, tên tệp, phần mở rộng tệp, tên đường dẫn, URL giả, đầu vào của người dùng và điều khiển Twitter được hiển thị như sau: "Để nhập mô-đun vào chương trình Node của bạn, hãy sử dụng yêu cầu chỉ thị. " Một khối mã được đặt như sau:

var EventEmitter = require('events').EventEmitter; var Counter = function(init) { this.increment = function() { init++; this.emit('incremented', init); }

Khi chúng tôi muốn thu hút sự chú ý của bạn đến một phần cụ thể của khối mã, các dòng hoặc mục có liên quan được in đậm:

var size = process.argv[2]; var totl = process.argv[3] || 100; var buff = []; for(var i=0; i < totl; i++) {

Preface

buff.push(new Buffer(size));

process.stdout.write(process.memoryUsage().heapTotal + "\n"); }

Window.onload = function() {

// When all requested document resources are loaded,

// do something with the resulting environment

}

element.onclick = function() {

// Do something when the user clicks on this element

}

Bất kỳ đầu vào hoặc đầu ra dòng lệnh nào được viết như sau:

> node process.js 1000000 100> out.file

# CHƯƠNG I

# Môi trường NODE

Mục tiêu của Node là cung cấp một cách dễ dàng để xây dựng các chương trình mạng có thể mở rộng.

- Ryan Dahl, người tạo ra Node.js

Các WWW (World Wide Web) làm cho các đối tượng siêu phương tiện trên Internet có thể kết nối với nhau, giao tiếp thông qua một bộ giao thức Internet tiêu chuẩn, thường HTTP (Giao thức truyền siêu văn bản). Sự phát triển về mức độ phức tạp, số lượng và loại ứng dụng web cung cấp các bộ sưu tập được quản lý của các đối tượng này thông qua trình duyệt đã làm tăng sự quan tâm đến các công nghệ hỗ trợ việc xây dựng và quản lý các ứng dụng được nối mạng phức tạp. Node là một trong những công nghệ như vậy. Bằng cách thành thạo Node, bạn đang học cách xây dựng thế hệ phần mềm tiếp theo.

Việc nắm giữ thông tin mà bất kỳ người nào có được là vô ích. Độ phức tạp theo quy mô; sự nhầm lẫn kéo theo sự phức tạp. Khi độ phân giải mờ đi, lỗi xảy ra.

Tương tự, biểu đồ hoạt động mô tả tất cả I / O (Đầu vào / Đầu ra)

Các tương tác mà một ứng dụng có thể hình thành giữa khách hàng và nhà cung cấp phải được lập kế hoạch và quản lý cẩn thận, tránh để năng lực của cả hệ thống và người tạo ra nó bị quá tải. Điều này liên quan đến việc kiểm soát hai chiều thông tin: khối lượng và hình dạng.

Khi một ứng dụng mạng mở rộng quy mô, khối lượng thông tin mà nó phải nhận biết, tổ chức và duy trì tăng lên. Khối lượng này, xét về luồng I / O, mức sử dụng bộ nhớ và CPU (bộ phận xử lý trung tâm) tải, mở rộng khi nhiều khách hàng kết nối hơn và ngay cả khi họ rời đi (về dữ liệu người dùng cụ thể liên tục).

Việc mở rộng khối lượng thông tin này cũng tạo gánh nặng cho nhà phát triển ứng dụng hoặc nhóm các nhà phát triển. Các vấn đề về tỷ lệ bắt đầu xuất hiện, thường cho thấy sự thất bại trong việc dự đoán chính xác hành vi của các hệ thống lớn so với hành vi của các hệ thống nhỏ. Một lớp dữ liệu được thiết kế để lưu trữ vài nghìn bản ghi có thể chứa được vài triệu không? Các thuật toán được sử dụng để tìm kiếm một số bản ghi có đủ hiệu quả để tìm kiếm nhiều bản ghi khác không? Máy chủ này có thể xử lý 10.000 kết nối máy khách đồng thời không? Các cạnh của sự đổi mới là sắc nét và cắt giảm nhanh chóng, giúp giảm thời gian cân nhắc chính xác khi chi phí sai sót đang được tăng lên. Hình dạng của các đối tượng bao gồm toàn bộ ứng dụng trở nên vô định hình và khó hiểu, đặc biệt khi các sửa đổi đặc biệt được thực hiện, theo phản ứng, để phản ứng với lực căng động trong hệ thống. Những gì được mô tả trong đặc tả là một hệ thống con nhỏ có thể đã được vá vào rất nhiều hệ thống khác đến nỗi ranh giới thực tế của nó bị hiểu nhầm. Không thể theo dõi chính xác đường viền của các bộ phận tổng hợp của tổng thể.

Cuối cùng một ứng dụng trở nên không thể đoán trước. Thật nguy hiểm khi người ta không thể dự đoán tất cả các trạng thái trong tương lai của một ứng dụng, hoặc các tác dụng phụ của sự thay đổi. Bất kỳ số lượng máy chủ, ngôn ngữ lập trình, kiến trúc phần cứng, phong cách quản lý nào, v.v., đều đã cố gắng khắc phục vấn đề khó chữa về rủi ro sau tăng trưởng, về sự thất bại đe dọa thành công. Thông thường các hệ thống có độ phức tạp lớn hơn được bán như một phương pháp chữa bệnh.

Thay vào đó, Node đã chọn sự rõ ràng và đơn giản. Có một luồng, liên kết với một vòng lặp sự kiện. Các tác vụ bị trì hoãn được đóng gói, nhập và thoát khỏi bối cảnh thực thi thông qua các lệnh gọi lại. Các hoạt động I / O tạo ra các luồng dữ liệu theo sự kiện, các luồng này được chuyển qua

một ngăn xếp duy nhất. Đồng thời được quản lý bởi hệ thống, trừu tượng hóa các nhóm luồng và đơn giản hóa việc quản lý bộ nhớ. Các phần phụ thuộc và thư viện được giới thiệu thông qua một hệ thống quản lý gói, được đóng gói gọn gàng và dễ dàng phân phối, cài đặt và gọi.

Các nhà phát triển có kinh nghiệm đều đã phải vật lộn với các vấn đề mà Node muốn giải quyết:

- Làm thế nào để phục vụ hàng ngàn khách hàng đồng thời một cách hiệu quả

- Mở rộng quy mô các ứng dụng được nối mạng ngoài một máy chủ duy nhất

- Ngăn chặn các hoạt động I / O trở thành tắc nghẽn

- Loại bỏ các điểm lỗi đơn lẻ, do đó đảm bảo độ tin cậy

- Đạt được song song một cách an toàn và có thể dự đoán được

Mỗi năm trôi qua, chúng ta thấy các ứng dụng và phần mềm hợp tác chịu trách nhiệm quản lý các mức độ đồng thời mà chỉ vài năm trước đây được coi là hiếm. Quản lý đồng thời, cả về xử lý kết nối và thiết kế ứng dụng, là chìa khóa để xây dựng kiến trúc web có thể mở rộng.

Trong cuốn sách này, chúng ta sẽ nghiên cứu các kỹ thuật mà các nhà phát triển Node chuyên nghiệp sử dụng để giải quyết những vấn đề này. Trong chương này, chúng ta sẽ khám phá cách một ứng dụng Node được thiết kế, hình dạng và kết cấu dấu chân của nó trên máy chủ cũng như bộ công cụ và tính năng cơ bản mạnh mẽ mà Node cung cấp cho các nhà phát triển. Xuyên suốt chúng ta sẽ xem xét các ví dụ phức tạp hơn chứng minh cách kiến trúc đơn giản, toàn diện và nhất quán của Node giải quyết tốt nhiều vấn đề khó khăn.

## Mở rộng JavaScript

Khi thiết kế Node, JavaScript không phải là lựa chọn ngôn ngữ gốc của Ryan Dahl. Tuy nhiên, sau khi khám phá nó, anh ấy đã tìm thấy một ngôn ngữ hiện đại rất tốt mà không có ý kiến về luồng, hệ thống tệp, xử lý các đối tượng nhị phân, quy trình, mạng và các khả năng khác mà người ta mong đợi tồn tại trong ngôn ngữ lập trình của hệ thống. JavaScript, bị giới hạn nghiêm ngặt cho trình duyệt, không được sử dụng và không được triển khai, các tính năng này.

Dahl được hướng dẫn bởi một số nguyên tắc cứng nhắc:

- Một chương trình / quy trình Node chạy trên một luồng duy nhất, sắp xếp thực hiện thông qua một vòng lặp sự kiện

- Các ứng dụng web chuyên sâu vào I / O, do đó, cần tập trung vào việc làm cho I / O nhanh chóng Luồng chương trình luôn được điều hướng thông qua các lệnh gọi lại không đồng bộ

- Các hoạt động CPU đắt tiền nên được chia thành các quy trình song song riêng biệt, tạo ra các sự kiện khi có kết quả

- Các chương trình phức tạp nên được lắp ráp từ các chương trình đơn giản hơn

Nguyên tắc chung là, các hoạt động không bao giờ được chặn. Mong muốn của Node về tốc độ (đồng thời cao) và hiệu quả (sử dụng tài nguyên tối thiểu) đòi hỏi phải giảm lãng phí. Quá trình chờ đợi là một quá trình lãng phí, đặc biệt là khi chờ đợi I / O.

Thiết kế hướng sự kiện, không đồng bộ của JavaScript phù hợp với mô hình này. Các ứng dụng bày tỏ sự quan tâm đến một số sự kiện trong tương lai và được thông báo khi sự kiện đó xảy ra. Mẫu JavaScript phổ biến này sẽ quen thuộc với bạn:

Window.onload = function() {

// When all requested document resources are loaded,

// do something with the resulting environment

}

element.onclick = function() {

// Do something when the user clicks on this element

}

Thời gian để một hành động I / O hoàn thành là không xác định, vì vậy, mẫu là yêu cầu thông báo khi một sự kiện I / O được phát ra, bất cứ khi nào có thể, cho phép các hoạt động khác được hoàn thành trong thời gian chờ đợi.

Node thêm một lượng lớn chức năng mới cho JavaScript. Về cơ bản, các phần bổ sung cung cấp các thư viện I / O đã xảy ra sự kiện cung cấp quyền truy cập hệ thống của nhà phát triển không có sẵn đối với JavaScript dựa trên trình duyệt, chẳng hạn như ghi vào hệ thống tệp hoặc mở một quy trình hệ thống khác. Ngoài ra, môi trường được thiết kế theo mô-đun, cho phép các chương trình phức tạp được lắp ráp từ các thành phần nhỏ hơn và đơn giản hơn.

Hãy xem cách Node đã nhập mô hình sự kiện của JavaScript, mở rộng nó và sử dụng nó trong việc tạo giao diện cho các lệnh hệ thống mạnh mẽ.

## Sự kiện

Nhiều phần mở rộng JavaScript trong Node phát ra các sự kiện. Những sự kiện này là những trường hợp của sự kiện.EventEmitter. Bất kỳ đối tượng nào cũng có thể mở rộng EventEmitter, cung cấp nhà phát triển với một bộ công cụ thanh lịch để xây dựng các giao diện không đồng bộ chặt chẽ cho các phương thức đối tượng.

Làm việc thông qua ví dụ này để trình bày cách thiết lập EventEmitter đối tượng là nguyên mẫu của một hàm tạo hàm. Vì mỗi phiên bản được xây dựng hiện có EventEmitter đối tượng tiếp xúc với chuỗi nguyên mẫu của nó, điều này cung cấp một tham chiếu tự nhiên đến sự kiện API (Giao diện lập trình ứng dụng).

Do đó, các phương thức instance có thể phát ra các sự kiện và chúng có thể được lắng nghe. Ở đây, chúng tôi phát ra số lượng mới nhất bất cứ khi nào counter.increment phương thức được gọi và liên kết một lệnh gọi lại với sự kiện tăng dần, điều này chỉ đơn giản là in giá trị bộ đếm hiện tại vào dòng lệnh:

var EventEmitter = require('events').EventEmitter;

var Counter = function(init) { this.increment = function() {

init++;

this.emit('incremented', init);

}}

Counter.prototype = new EventEmitter(); var counter = new Counter(10); var callback = function(count) { console.log(count);

}

counter.addListener('incremented', callback);

counter.increment(); // 11 counter.increment(); // 12

counter.increment (); // 11 counter.increment (); // 12

Để xóa những người nghe sự kiện bị ràng buộc với quầy tính tiền, sử dụng quầy tính tiền.

removeListener ('tăng dần', gọi lại). Để nhất quán với dựa trên trình duyệt JavaScript, counter.on và counter.addListener có thể hoán đổi cho nhau.

Việc bổ sung EventEmitter như một đối tượng có thể mở rộng làm tăng đáng kể khả năng của JavaScript trên máy chủ. Đặc biệt, nó cho phép các luồng dữ liệu I / O được xử lý theo hướng sự kiện, phù hợp với nguyên tắc lập trình không đồng bộ, không chặn của Node:

var Readable = require('stream').Readable;

var readable = new Readable; var count = 0;

readable.\_read = function() { if(++count > 10) { return readable.push(null);

}

setTimeout(function() { readable.push(count + "\n");

}, 500);

}; readable.pipe(process.stdout);

Trong chương trình này, chúng tôi đang tạo ra một Có thể đọc được luồng và chuyển bất kỳ dữ liệu nào được đẩy vào luồng này tới process.stdout. Cứ sau 500 mili giây, chúng tôi tăng một bộ đếm và đẩy số đó (thêm một dòng mới) vào luồng, dẫn đến một chuỗi số tăng dần được ghi vào thiết bị đầu cuối. Khi chuỗi của chúng tôi đạt đến giới hạn (10), chúng tôi đẩy vô giá trị vào luồng, khiến nó kết thúc. Đừng lo lắng nếu bạn không hiểu đầy đủ về cách Có thể đọc được được triển khai tại đây — các luồng sẽ được giải thích đầy đủ trong các chương sau. Chỉ cần lưu ý cách hành động đẩy dữ liệu lên luồng gây ra một sự kiện tương ứng để kích hoạt, cách nhà phát triển có thể chỉ định một lệnh gọi lại tùy chỉnh để xử lý sự kiện này và cách dữ liệu mới được thêm vào có thể được chuyển hướng đến các luồng khác. Node được thiết kế sao cho các hoạt động I / O được triển khai nhất quán dưới dạng các luồng dữ liệu sự kiện, không đồng bộ.

Cũng cần lưu ý tầm quan trọng của kiểu I / O này. Bởi vì vòng lặp sự kiện của Node chỉ cần tài nguyên cam kết để xử lý các lệnh gọi lại, nhiều lệnh khác có thể được xử lý trong thời gian chờ giữa mỗi khoảng thời gian.

Như một bài tập, hãy triển khai lại đoạn mã trước đó để dữ liệu đã phát ra được chuyển đến một tệp. Bạn sẽ cần sử dụng fs.createWriteStream:

var fs = require('fs');

var writeStream = fs.createWriteStream("./counter", {

flags : 'w', mode: 0666

});

## Môđun

Trong cuốn sách của anh ấy Nghệ thuật lập trình Unix, Eric Raymond đề xuất Quy tắc mô đun:

Các nhà phát triển nên xây dựng một chương trình từ các phần đơn giản được kết nối bởi các giao diện được xác định rõ ràng, do đó các vấn đề là cục bộ và các phần của chương trình có thể được thay thế trong các phiên bản sau để hỗ trợ các tính năng mới. Quy tắc này nhằm mục đích tiết kiệm thời gian gỡ lỗi mã phức tạp, dài và khó đọc.

Ý tưởng về việc xây dựng các hệ thống phức tạp từ các mảnh nhỏ, được liên kết với nhau một cách lỏng lẻo được thấy trong lý thuyết quản lý, lý thuyết về chính phủ, sản xuất vật chất và nhiều bối cảnh khác. Về phát triển phần mềm, nó khuyên các nhà phát triển chỉ nên đóng góp thành phần đơn giản nhất và hữu ích nhất cần thiết trong một hệ thống lớn hơn. Các hệ thống lớn rất khó lý luận, đặc biệt nếu ranh giới của các thành phần của nó mờ nhạt.

Một trong những khó khăn chính khi xây dựng các chương trình JavaScript có thể mở rộng là thiếu một giao diện tiêu chuẩn để tập hợp một chương trình mạch lạc từ nhiều chương trình nhỏ hơn. Ví dụ: một ứng dụng web điển hình có thể tải các phần phụ thuộc bằng cách sử dụng chuỗi < script> trong thẻ < đầu> phần của tài liệu HTML:

<head>

<script src = "fileA.js"> </script>

<script src = "fileB.js"> </script>

</head>

Có nhiều vấn đề với loại giải pháp này:

- Tất cả các phần phụ thuộc tiềm năng phải được khai báo trước khi cần thiết - tính năng bao gồm động yêu cầu các bản hack phức tạp.

- Các tập lệnh được giới thiệu không được đóng gói một cách cưỡng bức — không có gì ngăn chặn mã trong cả hai tệp từ việc ghi vào cùng một đối tượng chung. Không gian tên có thể dễ dàng va chạm, khiến việc tiêm tùy tiện trở nên nguy hiểm.

- Tập tin không thể giải quyết fileB dưới dạng một tập hợp — một ngữ cảnh có thể giải quyết được chẳng hạn như fileB.method không có sẵn.

- < script> bản thân phương pháp không có hệ thống, loại trừ việc thiết kế các dịch vụ mô-đun hữu ích, chẳng hạn như nhận thức về sự phụ thuộc hoặc kiểm soát phiên bản.

- Không thể dễ dàng xóa hoặc ghi đè các tập lệnh.

- Vì những nguy hiểm và khó khăn này, chia sẻ không phải là nỗ lực, làm giảm cơ hội cộng tác trong một hệ sinh thái mở.

Việc chèn các đoạn mã không thể đoán trước được vào ứng dụng thường xuyên làm nản lòng những nỗ lực định hình chức năng có thể đoán trước được. Điều cần thiết là một cách chuẩn để tải và chia sẻ các mô-đun lập trình kín đáo.

Theo đó, Node đã đưa ra khái niệm về gói, tuân theo đặc điểm kỹ thuật CommonJS. Gói là một tập hợp các tệp chương trình được đóng gói với một tệp kê khai mô tả bộ sưu tập. Sự phụ thuộc, quyền tác giả, mục đích, cấu trúc và các siêu dữ liệu quan trọng khác được hiển thị theo cách chuẩn. Điều này khuyến khích việc xây dựng các hệ thống lớn từ nhiều hệ thống nhỏ, phụ thuộc lẫn nhau. Có lẽ còn quan trọng hơn, nó khuyến khích chia sẻ:

Những gì tôi đang mô tả ở đây không phải là vấn đề kỹ thuật. Đó là vấn đề của mọi người cùng nhau và đưa ra quyết định bước tiếp và bắt đầu xây dựng một cái gì đó lớn hơn và thú vị hơn cùng nhau.

- Kevin Dangoor, người tạo ra CommonJS

Theo nhiều cách, sự thành công của Node là do sự phát triển về số lượng và chất lượng các gói có sẵn cho cộng đồng nhà phát triển, được phân phối qua hệ thống quản lý gói của Node, npm. Các lựa chọn thiết kế của hệ thống này, cả về mặt xã hội và kỹ thuật, đã giúp ích rất nhiều để giúp JavaScript trở thành một lựa chọn chuyên nghiệp khả thi cho việc lập trình hệ thống.

Thông tin mở rộng hơn về việc tạo và quản lý các gói Node có thể được tìm thấy trong Phụ lục A, Tổ chức công việc của bạn. Điểm mấu chốt là: xây dựng các chương trình từ các gói nếu có thể và chia sẻ các gói đó khi có thể. Hình dạng của các ứng dụng của bạn sẽ rõ ràng hơn và dễ bảo trì hơn. Điều quan trọng là, nỗ lực của hàng nghìn nhà phát triển khác có thể được liên kết với các ứng dụng thông qua npm, trực tiếp bằng cách đưa vào và gián tiếp khi các gói chia sẻ được các thành viên của cộng đồng Node thử nghiệm, cải tiến, cấu trúc lại và thay đổi cấu trúc.

Trái với suy nghĩ thông thường, npm không phải là chữ viết tắt của Trình quản lý gói nút ( hoặc thậm chí là một từ viết tắt):

https://npmjs.org/doc/faq.html#If-npm-is-an-acronymtại sao-nó-không bao giờ viết hoa

## Môi trường Mạng

I / O trong trình duyệt bị xáo trộn không thương tiếc, vì những lý do rất chính đáng — nếu JavaScript trên bất kỳ trang web cụ thể nào có thể truy cập vào hệ thống tệp của bạn hoặc mở kết nối mạng với bất kỳ máy chủ nào, WWW sẽ là một nơi kém thú vị hơn.

Đối với Node, I / O có tầm quan trọng cơ bản và trọng tâm của nó ngay từ đầu là đơn giản hóa việc tạo ra các hệ thống có thể mở rộng với yêu cầu I / O cao. Có khả năng trải nghiệm đầu tiên của bạn với Node là viết một máy chủ HTTP.

Node hỗ trợ một số giao thức mạng tiêu chuẩn ngoài HTTP, chẳng hạn như TLS / SSL (Bảo mật lớp truyền tải / Lớp cổng bảo mật), và UDP (Giao thức dữ liệu người dùng). Với những công cụ này, chúng tôi có thể dễ dàng xây dựng các chương trình mạng có khả năng mở rộng, vượt xa những AJAX (JavaScript và Xml không đồng bộ) các kỹ thuật quen thuộc với nhà phát triển JavaScript.

Hãy tạo một chương trình đơn giản cho phép người dùng gửi dữ liệu giữa hai máy chủ UDP:

var dgram = require('dgram'); var client = dgram.createSocket("udp4"); var server = dgram.createSocket("udp4"); var message = process.argv[2] || "message"; message = new Buffer(message);

server

.on("message", function (msg) {

process.stdout.write("Got message: " + msg + "\n"); process.exit();

})

.bind(41234); client.send(message, 0, message.length, 41234, "localhost");

Giả sử tên tệp chương trình là udp.js một tin nhắn có thể được gửi qua UDP bằng cách chạy chương trình này từ thiết bị đầu cuối như sau:

nút udp.js "tin nhắn của tôi"

Điều này sẽ dẫn đến kết quả sau:

Có tin nhắn: tin nhắn của tôi

Đầu tiên chúng tôi thiết lập các máy chủ UDP của mình, một máy chủ hoạt động như một đài truyền hình, một máy chủ khác hoạt động như một người nghe. process.argv chứa thông tin lệnh hữu ích, bao gồm các đối số dòng lệnh bắt đầu tại chỉ mục (2), trong trường hợp này sẽ chứa " tin nhắn của tôi ”. UDP yêu cầu tin nhắn phải Đệm các đối tượng, vì vậy chúng tôi đảm bảo rằng một số thông báo tồn tại và chuyển đổi nó.

Máy chủ UDP là một ví dụ của EventEmitter, phát ra một sự kiện thông báo khi các thông báo được nhận trên cổng mà nó bị ràng buộc. Máy chủ này chỉ lặp lại tin nhắn đã nhận. Tất cả những gì còn lại phải làm là gửi tin nhắn, hành động nào được thực hiện bởi máy khách, chuyển thông điệp của chúng ta đến cổng # 41234.

Việc di chuyển các luồng dữ liệu xung quanh lớp I / O của ứng dụng của bạn được đơn giản hóa trong Node. Không khó để chia sẻ các luồng dữ liệu qua các máy chủ giao thức khác nhau, vì các luồng dữ liệu được chuẩn hóa thông qua các giao diện của Node. Chi tiết giao thức được xử lý cho bạn.

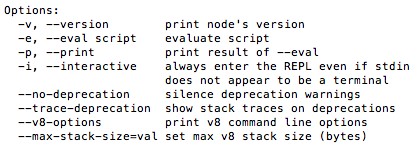
Hãy tiếp tục khám phá I / O, quá trình đối tượng và các sự kiện. Đầu tiên, chúng ta hãy đi sâu vào cỗ máy cung cấp năng lượng cho lõi của Node.

## V8

V8 là công cụ JavaScript của Google, được viết bằng C ++. Nó biên dịch và thực thi mã JavaScript bên trong VM (Máy ảo). Khi một trang web được tải vào Google Chrome thể hiện một số loại hiệu ứng động, như tự động cập nhật danh sách hoặc nguồn cấp tin tức, bạn sẽ thấy JavaScript, được biên dịch bởi V8, đang hoạt động.

Trong khi bản thân Node sẽ quản lý hiệu quả các hoạt động I / O, quá trình đối tượng đề cập đến thời gian chạy V8. Do đó, điều quan trọng là phải hiểu cách cấu hình môi trường V8, đặc biệt là khi ứng dụng của bạn phát triển về kích thước.

Bằng cách gõ nút -h vào một bảng điều khiển, một cái gì đó như sau sẽ được hiển thị:



Chúng tôi có thể xem danh sách các tùy chọn V8 có thể truy cập như thế nào thông qua –-V8-tùy chọn cờ.

Danh sách các tùy chọn cấu hình cho V8 là một danh sách dài, vì vậy chúng tôi sẽ không trình bày từng tùy chọn ở đây. Khi chúng tôi xem xét cuốn sách, các tùy chọn liên quan sẽ được thảo luận sâu hơn. Tuy nhiên, sẽ rất hữu ích nếu tóm tắt một số tùy chọn được cung cấp để quản lý giới hạn hệ thống và bộ nhớ, cũng như những tùy chọn được sử dụng để định cấu hình bộ lệnh của JavaScript, giới thiệu một số tính năng mới trong ES6 (EcmaScript6), thường được gọi là Hòa hợp.

Có thể xem phiên bản V8 được cài đặt Node của bạn bằng cách nhập:

nút –e "console.log (process.versions.v8)"

## Bộ nhớ và các giới hạn

Một tùy chọn cấu hình V8 rất mạnh đủ quan trọng để đưa nó vào bộ sưu tập của riêng Node: - max-stack-size. Hãy xem xét một số quyền hạn mới mà một nhà phát triển Node đã được trao để có thể định cấu hình thời gian chạy JavaScript cụ thể.

Cố gắng phá vỡ một hệ thống là một cách tuyệt vời để khám phá các giới hạn và hình dạng của nó. Hãy viết một chương trình sẽ làm hỏng V8: var count = 0;

var count = 0;

(function curse() { console.log(++count); curse(); })()

Hàm tự thực thi, khép kín này sẽ gọi chính nó một cách đệ quy mãi mãi, hoặc cho đến khi nó bị buộc phải dừng lại. Mỗi lần lặp lại nguyền rủa thêm một khung khác vào ngăn xếp cuộc gọi. Sự phát triển không kiểm soát này cuối cùng sẽ khiến thời gian chạy JavaScript bị sụp đổ, trích dẫn a RangeError: Đã vượt quá kích thước ngăn xếp cuộc gọi tối đa.

Mục đích của - max-stack-size bây giờ nên rõ ràng. Tùy chọn V8 trực tiếp tương đương là --kích thước ngăn xếp, được truyền một giá trị, trong KB (Kilobyte), để nâng cao giới hạn này. Thử nghiệm với chương trình trên, lưu ý số lần lặp có thể ở các cài đặt khác nhau.

Mặc dù có khả năng đạt đến giới hạn này thể hiện thuật toán được thiết kế không chính xác, nhưng việc có thể mở rộng chiều rộng của không gian hoạt động có sẵn cho quy trình Node sẽ bổ sung vào bộ sưu tập các giải pháp có sẵn cho các nhà phát triển.

Trên máy 32 bit và 64 bit, mặc định phân bổ bộ nhớ của V8 lần lượt là 700 MB và 1400 MB. Trong các phiên bản V8 mới hơn, giới hạn bộ nhớ trên hệ thống 64 bit không còn được thiết lập bởi V8, về mặt lý thuyết cho thấy không có giới hạn. Tuy nhiên, OS (Hệ điều hành) trên đó Node đang chạy luôn có thể giới hạn dung lượng bộ nhớ mà V8 có thể sử dụng, do đó không thể nói chung giới hạn thực sự của bất kỳ quá trình nhất định nào.

V8 cung cấp - max\_old\_space\_size , cho phép kiểm soát lượng bộ nhớ có sẵn cho một quy trình, chấp nhận một giá trị tính bằng MB. Nếu bạn cần tăng phân bổ bộ nhớ, chỉ cần chuyển tùy chọn này giá trị mong muốn khi tạo một quy trình Node.

Nó thường là một chiến lược tuyệt vời để giảm phân bổ bộ nhớ khả dụng cho một phiên bản Node nhất định, đặc biệt là khi chạy nhiều phiên bản. Cũng như giới hạn ngăn xếp, hãy cân nhắc xem liệu nhu cầu bộ nhớ lớn có được ủy thác tốt hơn cho lớp lưu trữ chuyên dụng, chẳng hạn như cơ sở dữ liệu trong bộ nhớ hoặc tương tự hay không.

Bạn có thể tìm thấy một cuộc thảo luận bổ sung với nhóm V8 về quan điểm của họ về cách phân bổ bộ nhớ tại đây: http://code.google.com/p/v8/issues/detail?id=847

Một trong những ưu điểm chính của các ngôn ngữ cấp cao hiện đại như JavaScript là quản lý tự động bộ nhớ thông qua GC (Thu gom rác). Các chiến lược GC rất nhiều và phức tạp, nhưng tất cả đều tuân theo một ý tưởng cốt lõi đơn giản: thường xuyên, bộ nhớ được cấp phát miễn phí không còn được sử dụng nữa.

Hạn chế của GC tự động là nó có tác dụng hãm tốc độ quá trình. Mặc dù những ưu điểm rõ ràng của GC tự động vượt trội hơn những nhược điểm của nó trong đa số trường hợp, nhưng nhà phát triển Node vẫn còn cơ hội để kiểm soát một số hành vi của nó. Điều này chủ yếu được thực hiện thông qua các cờ –-Nouse\_idle\_notification và - lộ\_gc.

Vượt qua –-Nouse\_idle\_notification cờ sẽ ra lệnh cho V8 bỏ qua các cuộc gọi thông báo nhàn rỗi từ Node, đó là các yêu cầu tới V8 yêu cầu nó chạy GC ngay lập tức, vì quá trình Node hiện đang không hoạt động. Vì Node tích cực với các lệnh gọi này (hiệu quả tạo ra các phương tiện chặn sạch), nên việc thừa GC có thể làm chậm ứng dụng của bạn. Lưu ý rằng việc sử dụng cờ này không tắt GC; GC chỉ đơn giản là chạy ít thường xuyên hơn. Trong những trường hợp thích hợp, kỹ thuật này có thể tăng hiệu suất.

- expo\_gc đầu quy trình GC theo cách thủ công. Kết hợp vớigiới thiệu một phương pháp toàn cục mới cho quy trình Node, gc (), cho phép mã JavaScript bắt

- nouse\_idle\_notification nhà phát triển hiện có thể kiểm soát ở một mức độ nào đó tần suất GC chạy. Tại bất kỳ thời

điểm nào trong mã JavaScript của tôi, tôi có thể chỉ cần gọi gc () và khởi động bộ thu.

Có thể điều chỉnh mức sử dụng bộ nhớ và GC chắc chắn hữu ích. Hãy nhớ rằng khối lượng ứng dụng có thể nhanh chóng tăng lên theo những cách không thể đoán trước. Nếu dung lượng bộ nhớ của ứng dụng của bạn ổn định ở gần giới hạn phân bổ của V8, bạn nên bắt đầu nghĩ đến việc mở rộng quy mô theo chiều ngang. Sử dụng bộ nhớ một cách khôn ngoan và tách các phiên bản Node mới nếu thích hợp.

## Sự tương thích

JavaScript chưa bao giờ ngừng phát triển và hiện nó đang trải qua một thời kỳ phục hưng, được giúp đỡ một phần không nhỏ bởi sự phổ biến của Node. Phiên bản tiếp theo của ngôn ngữ này, có tên là Harmony, giới thiệu một số tính năng và khái niệm mới quan trọng.

Thông tin thêm về ES6 Harmony có thể được tìm thấy tại: http://wiki.ecmascript.org/doku.php?id=harmony:harmony.

Các tùy chọn Harmony có sẵn là:

|  |  |
| --- | --- |
| Cờ | Sự miêu tả |
| - hài hòa\_typeof  - hài hòa  - mô-đun hài hòa  - peace\_proxies  - sự hài hòa  - hòa hợp | Bật ngữ nghĩa cho typeof Bật phạm vi khối  Bật mô-đun (ngụ ý phạm vi khối) Bật proxy  Bật bộ sưu tập (bộ, bản đồ và bản đồ yếu) Bật tất cả các tính năng (ngoại trừ typeof) |

Việc thảo luận sâu hơn về những tính năng mới này nằm ngoài phạm vi của cuốn sách này. Tuy nhiên, cần phải nói rằng khả năng sử dụng các tính năng JavaScript mới nhất hiện nay, thông qua Node, mang lại cho nhà phát triển một lợi thế lớn — không có cuộc chiến trình duyệt nào trên máy chủ để kìm hãm sự đổi mới.

Ví dụ, ES6's bản đồ yếu cho phép sử dụng các chuỗi không phải là khóa trong HashMap:

"use strict" let owners = new WeakMap();

let task = {

title : "Big Project"

};

owners.set(task, 'John'); function owner(task) { if(owners.has(task)) {

return console.log(owners.get(task));

}

console.log("No owner for this task.");

}

owner(task); // "John" owner({}); // "No owner for this task"

Như một bài tập, người đọc có thể ánh xạ các luồng đầu vào (cố định) với các luồng đầu ra (thay đổi) theo cách tương tự.

## Đối tượng quá trình

Bây giờ, chúng ta đã rõ về cách cấu trúc của Node, về V8, vòng lặp sự kiện, v.v. Bây giờ chúng ta sẽ thảo luận chi tiết về cách các hướng dẫn bạn viết (một chương trình JavaScript) được V8 biên dịch thành danh sách các hướng dẫn có ngữ cảnh thực thi có thể truy cập thông qua Node gốc quá trình vật lý.

Luồng duy nhất tạo thành cột sống của vòng lặp sự kiện của Node là vòng lặp sự kiện của V8. Khi các hoạt động I / O được bắt đầu trong vòng lặp này, chúng được ủy quyền cho libuv, quản lý yêu cầu bằng cách sử dụng môi trường riêng (đa luồng, không đồng bộ). libuv thông báo hoàn thành các hoạt động I / O, cho phép bất kỳ lệnh gọi lại nào đang chờ sự kiện này được đưa lại vào chuỗi V8 chính để thực thi:

V8

ủy quyền I / 0 cho libuv

Nhóm chủ đề

đa lu

~~ồ~~

ng không đồng bộ

Gọi hàm

gọi lại

Hệ thống tập tin

Mạng lưới

Điểm giao quá trình đối tượng cung cấp thông tin và kiểm soát quá trình đang chạy hiện tại. Nó là một ví dụ của EventEmitter, có thể truy cập từ bất kỳ phạm vi nào và hiển thị các con trỏ cấp thấp rất hữu ích. Hãy xem xét chương trình sau:

very useful low-level pointers. Consider the following program:

var size = process.argv[2]; var totl = process.argv[3] || 100; var buff = []; for(var i=0; i < totl; i++) { buff.push(new Buffer(size));

process.stdout.write(process.memoryUsage().heapTotal + "\n"); }

}

Tải xuống mã mẫu:

Bạn có thể tải xuống các tệp mã mẫu cho tất cả các sách Packt mà bạn đã mua từ tài khoản của mình tại http://www.packtpub.com. Nếu bạn đã mua cuốn sách này ở nơi khác, bạn có thể truy cập http: //www.packtpub. com / support và đăng ký để nhận các tập tin qua e-mail trực tiếp cho bạn.

Giả sử tệp chương trình được đặt tên process.js, nó sẽ được thực thi như vậy:

> node process.js 1000000 100

Ngữ cảnh thực thi này trước tiên tìm nạp hai đối số dòng lệnh qua quá trình. argv, xây dựng một cấu trúc lặp để tăng mức sử dụng bộ nhớ tùy thuộc vào các đối số này và phát ra dữ liệu sử dụng bộ nhớ khi mỗi lần phân bổ mới được thực hiện. Chương trình gửi đầu ra tới stdout, nhưng cách khác có thể truyền đầu ra cho các quy trình khác hoặc thậm chí một tệp:

> node process.js 1000000 100> out.file

Trên thực tế, sự quen thuộc console.log được triển khai trong lõi Node như một trình bao bọc xung quanh process.stdout.write:

console.log = function (d) { process.stdout.write (d + '\ n');

}

Quá trình Node bắt đầu bằng cách xây dựng một ngăn xếp thực thi duy nhất, với ngữ cảnh toàn cục tạo thành cơ sở của ngăn xếp. Các hàm trên ngăn xếp này thực thi trong ngữ cảnh, cục bộ, của riêng chúng (đôi khi được gọi là phạm vi), vẫn được bao bọc trong ngữ cảnh toàn cục (mà bạn sẽ nghe gọi là đóng). Bởi vì Node là sự kiện, bất kỳ ngữ cảnh thực thi nhất định nào cũng có thể cam kết luồng đang chạy để xử lý bối cảnh thực thi cuối cùng. Đây là mục đích của các hàm gọi lại.

Hãy xem xét sơ đồ sau của một giao diện đơn giản để truy cập hệ thống tệp:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Bối cảnh toàn cầu   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Var fs = request ('fs'); hệ thống  tệp hàm () { Bối cảnh thực thi   |  |  |  |  | | --- | --- | --- | --- | | hệ thống tệp hàm () { Bối cảnh thực thi   |  |  |  | | --- | --- | --- | | fs.readdir (dir, function (err, list) {   |  |  | | --- | --- | | . . . | Bối cảnh thực thi (gọi lại l / O) |   }); |   } |   } | |

Nếu chúng ta khởi tạo Hệ thống tập tin và gọi readDir một cấu trúc ngữ cảnh thực thi lồng nhau sẽ được tạo: (global (fileSystem (readDir (hàm ẩn danh))))

). Ngăn xếp thực thi đồng thời được đưa vào luồng quy trình đơn của Node. Ngăn xếp này vẫn còn trong bộ nhớ cho đến khi libuv báo cáo rằng fs.readdir đã hoàn tất, lúc này lệnh gọi lại ẩn danh đã đăng ký sẽ kích hoạt, giải quyết ngữ cảnh thực thi đang chờ xử lý duy nhất. Vì không có sự kiện nào khác đang chờ xử lý và việc duy trì các đóng không còn cần thiết nữa, toàn bộ cấu trúc có thể được chia nhỏ một cách an toàn (ngược lại, bắt đầu bằng ẩn danh) và quy trình có thể thoát, giải phóng bất kỳ bộ nhớ được cấp phát nào. Phương pháp xây dựng và chia nhỏ một ngăn xếp này là những gì cuối cùng vòng lặp sự kiện của Node đang thực hiện.

Chúng ta sẽ khám phá bộ đầy đủ các lệnh và thuộc tính có trong quá trình khi chúng tôi tiếp tục phát triển các ví dụ và thư viện trong cuốn sách này.

## Vòng lặp Đọc-Đánh giá-In và thực hiện một chương trình Node

Điểm giao REPL (Đọc-Đánh giá-In-Vòng lặp) đại diện cho Node shell. Để nhập lời nhắc trình bao, hãy nhập Node qua thiết bị đầu cuối của bạn mà không cần chuyển tên tệp:

> nút

Bây giờ bạn có quyền truy cập vào một quy trình Node đang chạy và có thể chuyển các lệnh JavaScript cho quy trình này. Ví dụ, sau khi nhập 2 + 2 vỏ sẽ gửi 4 đến stdout. Node's REPL là một nơi tuyệt vời để thử, gỡ lỗi, kiểm tra hoặc chơi với mã JavaScript.

Vì REPL là một đối tượng gốc, các chương trình cũng có thể sử dụng các phiên bản làm ngữ cảnh để chạy JavaScript một cách tương tác. Ví dụ: ở đây chúng tôi tạo chức năng tùy chỉnh của riêng mình sayH Hello, thêm nó vào ngữ cảnh require('repl').start("> ").context.sayHello = function() { return "Hello" };

Vào sayHello () tại dấu nhắc sẽ dẫn đến xin chào được gửi đến stdout.

Hãy tiếp thu mọi thứ chúng ta đã học trong chương này và tạo REPL tương tác cho phép chúng ta thực thi JavaScript trên máy chủ từ xa.

Tạo hai tệp, repl\_client.js và repl\_server.js, bằng cách sử dụng mã sau và chạy mỗi cửa sổ đầu cuối của riêng nó, /\* repl\_client.js \*/ var net = require('net'); var sock = net.connect(8080); process.stdin.pipe(sock); sock.pipe(process.stdout);

/\* repl\_server.js \*/ var repl = require('repl') var net = require('net') net.createServer(function(socket) { repl

.start({ prompt : '> ', input : socket, output : socket, terminal : true

})

.on('exit', function () { socket.end()

})

}).listen(8080)

repl\_client chỉ cần tạo một kết nối ổ cắm mới với cổng 8080 xuyên qua mạng lưới. kết nối, và chuyển mọi dữ liệu đến từ stdin ( thiết bị đầu cuối của bạn) thông qua ổ cắm đó. Tương tự, bất kỳ dữ liệu nào đến từ socket được chuyển đến stdout ( thiết bị đầu cuối của bạn).

Rõ ràng là chúng tôi đã tạo ra một cách để nhận đầu vào và gửi nó qua một ổ cắm đến cổng 8080, lắng nghe bất kỳ dữ liệu nào mà ổ cắm có thể gửi lại cho chúng tôi.

repl\_server đóng vòng lặp. Đầu tiên chúng tôi tạo một TCP (Giao thức điều khiển truyền) máy chủ với net.createServer, liên kết với cổng 8080 qua. nghe. Cuộc gọi lại được chuyển đến net.createServer sẽ nhận được một tham chiếu đến ổ cắm liên kết. Trong phần bao bọc của lệnh gọi lại đó, chúng tôi tạo một phiên bản REPL mới, tạo cho nó một dấu nhắc (> ở đây, nhưng có thể là bất kỳ chuỗi nào), cho biết rằng cả hai đều phải lắng nghe đầu vào từ và phát sóng đầu ra đến, tham chiếu ổ cắm đã chuyển, chỉ ra rằng dữ liệu ổ cắm phải được coi là thiết bị đầu cuối dữ liệu (có mã hóa đặc biệt).

Bây giờ chúng ta có thể gõ một cái gì đó như console.log ("xin chào") vào thiết bị đầu cuối của khách hàng và xem xin chào hiển thị.

Để xác nhận rằng việc thực thi các lệnh JavaScript của chúng tôi đang diễn ra trong phiên bản máy chủ, hãy nhập console.log (process.argv) vào máy khách và máy chủ sẽ hiển thị một đối tượng chứa đường dẫn quy trình hiện tại, sẽ là repl\_server.js.

Từ minh chứng này, cần phải rõ rằng chúng tôi đã tạo ra một cách để điều khiển từ xa các quy trình Node. Từ đây là một bước ngắn để đến với các công cụ phân tích đa nút, quản lý bộ nhớ từ xa, quản trị máy chủ tự động, v.v.

## Tóm lược

Trong chương này, chúng tôi đã phác thảo các vấn đề chính mà các nhà thiết kế của Node tìm cách giải quyết và cách giải pháp của họ đã giúp việc tạo ra các hệ thống mạng đồng thời cao, có thể mở rộng dễ dàng hơn cho một cộng đồng mở các nhà phát triển. Chúng ta đã thấy cách JavaScript đã được cấp những sức mạnh mới rất hữu ích, cách mô hình sự kiện của nó đã được mở rộng và cách V8 có thể được cấu hình để tùy chỉnh thêm thời gian chạy JavaScript.

Thông qua các ví dụ, chúng ta đã biết cách I / O được xử lý bởi Node, cách lập trình REPL, cũng như cách quản lý đầu vào và đầu ra cho đối tượng quy trình. Mục tiêu của việc chứng minh cách Node cho phép các ứng dụng được xây dựng một cách thông minh từ các phần được định hình tốt theo một cách có nguyên tắc đã bắt đầu. Trong chương tiếp theo, chúng ta sẽ nghiên cứu sâu hơn về lập trình không đồng bộ, tìm hiểu cách quản lý các chuỗi sự kiện phức tạp hơn và phát triển các chương trình mạnh mẽ hơn bằng cách sử dụng mô hình của Node.

# CHƯƠNG 2

# Hiểu về lập trình không đồng bộ

Lập trình theo hướng sự kiện

Cách tốt nhất để dự đoán tương lai là phát minh ra nó.

- Alan Kay

Loại bỏ các quy trình chặn thông qua việc sử dụng I / O theo hướng sự kiện, không đồng bộ là nguyên tắc tổ chức chính của Node. Chúng tôi đã tìm hiểu cách thiết kế này giúp các nhà phát triển định hình thông tin và bổ sung dung lượng: các quy trình nhẹ, độc lập và không chia sẻ gì giao tiếp thông qua các lệnh gọi lại được đồng bộ hóa trong một vòng lặp sự kiện có thể dự đoán được.

Cùng với sự phát triển về mức độ phổ biến của Node là sự tăng trưởng về số lượng các hệ thống và ứng dụng sự kiện được thiết kế tốt. Để một công nghệ mới thành công, nó phải loại bỏ các vấn đề hiện có và / hoặc cung cấp cho người tiêu dùng một giải pháp tốt hơn với chi phí thấp hơn về thời gian, công sức hoặc giá cả. Trong thời gian tồn tại ngắn ngủi và màu mỡ, cộng đồng Node đã hợp tác chứng minh rằng mô hình phát triển mới này là một giải pháp thay thế khả thi cho các công nghệ hiện có. Số lượng và chất lượng của các giải pháp dựa trên Node cung cấp năng lượng cho các ứng dụng cấp doanh nghiệp cung cấp thêm bằng chứng rằng những ý tưởng mới này không chỉ mới lạ mà còn được ưa thích hơn.

Trong chương này chúng ta sẽ đi sâu hơn về cách Node thực hiện lập trình hướng sự kiện. Chúng tôi sẽ bắt đầu bằng cách giải nén các ý tưởng và lý thuyết mà các ngôn ngữ và môi trường hướng sự kiện bắt nguồn và vật lộn với nó, trong nỗ lực xóa bỏ những quan niệm sai lầm và khuyến khích sự thành thạo. Tiếp theo phần giới thiệu này, sẽ trình bày chi tiết hơn về cách bộ định thời, lệnh gọi lại, sự kiện I / O, điều khiển luồng và vòng lặp sự kiện được triển khai và sử dụng. Lý thuyết sẽ được thực hành khi chúng tôi xây dựng một tệp đơn giản nhưng mẫu mực và các ứng dụng hướng dữ liệu, nêu bật điểm mạnh của Node và cách nó thành công trong tham vọng đơn giản hóa các thiết kế ứng dụng mạng.

## 2.1. Phát sự kiện

Việc hiểu chính xác về tổng chi phí cuối cùng của việc yêu cầu thực hiện một dịch vụ luôn là điều tốt.

I / O đắt tiền. Trong biểu đồ sau (lấy từ Ryan Dahl ' bản trình bày ban đầu trên Node) chúng ta có thể thấy nhiều chu kỳ xung nhịp mà các tác vụ hệ thống điển hình tiêu thụ. Chi phí tương đối của các hoạt động I / O là rất đáng chú ý.

|  |  |
| --- | --- |
| Bộ nhớ đệm L1  Bộ nhớ đệm L2  RAM  Đĩa  Mạng lưới | 3 chu kỳ  14 chu kỳ  250 chu kỳ  41.000.000 chu kỳ  240.000.000 chu kỳ |

Lý do là đủ rõ ràng: đĩa là một thiết bị vật lý, một đĩa kim loại quay tròn truyền dữ liệu ở tốc độ không thể sánh với tốc độ của bộ nhớ cache trên chip hoặc gần chip di chuyển dữ liệu giữa CPU và RAM (Bộ nhớ truy cập ngẫu nhiên ). Tương tự, một mạng bị ràng buộc bởi tốc độ mà dữ liệu có thể di chuyển qua các "dây" kết nối của nó, được điều chế bởi các bộ điều khiển của nó. Ngay cả khi thông qua cáp quang, bản thân ánh sáng cũng cần 0,1344 giây để di chuyển khắp thế giới. Trong một mạng được sử dụng bởi hàng tỷ người thường xuyên tương tác ở những khoảng cách xa, độ trễ này sẽ hình thành.

Trong thị trường truyền thống được mô tả bởi một ứng dụng chạy trên hệ thống chặn, việc mua một hoạt động tệp yêu cầu chi tiêu đáng kể tài nguyên, như chúng ta có thể thấy trong bảng trước. Điều này chủ yếu là do sự khan hiếm: một số lượng quy trình cố định hay còn gọi là "đơn vị lao động" có sẵn, mỗi quy trình chỉ có thể xử lý một nhiệm vụ duy nhất và khi sự sẵn có của lao động giảm, chi phí của nó (cho khách hàng) tăng lên.

Sự đột phá trong tư duy được phản ánh bởi thiết kế của Node rất dễ hiểu khi người ta nhận ra rằng hầu hết các luồng công nhân dành thời gian chờ đợi — để có thêm hướng dẫn, một nhiệm vụ phụ cần hoàn thành, v.v. Ví dụ: một quy trình được chỉ định để phục vụ lệnh định dạng ổ cứng của tôi sẽ dành tất cả các tài nguyên được phân bổ của nó để quản lý một quy trình làm việc như sau:

Giao tiếp với trình điều khiển thiết bị rằng yêu cầu định dạng đã được thực hiện

Không hoạt động, chờ đợi một khoảng thời gian "không thể biết trước"

Nhận tín hiệu định dạng đã hoàn thành

Thông báo cho khách hàng

Dọn dẹp; tắt



Trong hình trước, chúng ta thấy rằng một nhân viên đắt tiền đang tính phí cho khách hàng một khoản phí cố định trên một đơn vị thời gian bất kể có bất kỳ công việc hữu ích nào đang được thực hiện hay không (khách hàng đang trả tiền như nhau cho hoạt động và nhàn rỗi). Hay nói một cách khác, điều đó không nhất thiết là đúng, và thường đơn giản là không đúng, rằng các nhiệm vụ phụ bao gồm tổng số nhiệm vụ, mỗi nhiệm vụ đòi hỏi nỗ lực hoặc chuyên môn giống hệt nhau, và do đó, thật lãng phí nếu phải trả một cái giá đắt cho lao động rẻ mạt như vậy. .

Thông cảm, chúng ta cũng phải công nhận rằng người lao động này không thể làm tốt hơn ngay cả khi đã sẵn sàng và có thể giải quyết nhiều công việc hơn — ngay cả người lao động có thiện chí tốt nhất cũng không thể làm gì đối với các nút thắt I / O. Công nhân ở đây là I / O ràng buộc.

A chặn do đó, quá trình được hiểu rõ hơn là một nhàn rỗi quy trình, và nhàn rỗi

quy trình là tắc nghẽn trong tác vụ cụ thể và cho luồng ứng dụng tổng thể. Điều gì sẽ xảy ra nếu nhiều khách hàng có thể chia sẻ cùng một nhân viên, chẳng hạn như thời điểm một nhân viên thông báo tình trạng sẵn sàng do tắc nghẽn I / O, một công việc khác từ một khách hàng khác có thể được bắt đầu?

Node đã hàng hóa I / O thông qua việc giới thiệu một môi trường mà tài nguyên hệ thống (lý tưởng là) không bao giờ nhàn rỗi. Lập trình hướng sự kiện do Node thực hiện phản ánh mục tiêu đơn giản là giảm chi phí hệ thống tổng thể

bằng cách khuyến khích chia sẻ lao động đắt đỏ, chủ yếu bằng cách giảm số lượng nút thắt vào / ra xuống 0. Chúng ta không còn có một khối lao động phổ thông được định giá một cách bất lực; chúng tôi có thể giảm tất cả nỗ lực thành các đơn vị rời rạc với các hình dạng được phân định chính xác và do đó thừa nhận việc định giá chính xác hơn nhiều. Các khoản chi giống nhau về vốn có thể tài trợ cho một số lượng lớn hơn nhiều giao dịch đã hoàn thành, làm tăng hiệu quả của thị trường và tiềm năng của thị trường về sản phẩm mới và chủng loại sản phẩm mới. Nhiều giao dịch đồng thời hơn có thể được xử lý trên cùng một cơ sở hạ tầng với cùng một mức chi phí.

Nếu trạng thái bắt đầu, dừng và không hoạt động của một quá trình được hiểu là sự kiện có thể được đăng ký và hoạt động khi chúng ta có thể bắt đầu thảo luận về cách các hệ thống cực kỳ phức tạp có thể được xây dựng trong mô hình mới này, và về cơ bản, khá đơn giản để nắm bắt.

Môi trường trong đó nhiều công việc của khách hàng được lên lịch hợp tác trông như thế nào? Và thông điệp này được truyền giữa các sự kiện được xử lý như thế nào?

## 2.2. Hợp tác

Luồng công nhân được mô tả trong phần trước là một ví dụ về máy chủ chặn. Mỗi công nhân được giao một nhiệm vụ hoặc quy trình, với mỗi quy trình chỉ có thể chấp nhận một yêu cầu công việc. Họ sẽ chặn các yêu cầu khác, ngay cả khi chạy không tải:

I / O

Quá trình

BỊ CHẶN

Quá trình

Quá trình

Quá trình

I / O

BỊ CHẶN

Quá trình

Điều tốt hơn là một môi trường làm việc hợp tác, nơi người lao động có thể được giao những nhiệm vụ mới để thực hiện, thay vì làm việc không tải. Để đạt được như vậy

mục tiêu cần thiết là một tổng đài ảo, nơi các yêu cầu về dịch vụ có thể được gửi đến các nhân viên có sẵn và nơi nhân viên có thể thông báo cho tổng đài về tính khả dụng của họ.

Một cách để đạt được mục tiêu này là duy trì ý tưởng có một lượng lao động sẵn có, nhưng nâng cao hiệu quả bằng cách giao nhiệm vụ cho các công nhân khác nhau khi họ vào làm:

Công nhân

Khách hàng

Điều phối

Công nhân

Công nhân

Một nhược điểm của phương pháp này là số lượng lập kế hoạch và giám sát công nhân cần được thực hiện. Người điều phối phải điền vào một dòng yêu cầu liên tục, đồng thời quản lý các thông báo đến từ những người làm việc về tính khả dụng của họ, chia nhỏ các yêu cầu thành các nhiệm vụ có thể quản lý và phân loại chúng một cách hiệu quả sao cho số lượng công nhân không hoạt động nhiều nhất.

Có lẽ quan trọng nhất, điều gì sẽ xảy ra khi tất cả công nhân đã được đặt hết? Người điều phối có bắt đầu gửi yêu cầu từ khách hàng không? Điều phối cũng cần nhiều tài nguyên và ngay cả tài nguyên của người điều phối cũng có giới hạn. Nếu các yêu cầu tiếp tục được gửi đến và không có nhân viên nào sẵn sàng phục vụ họ, người điều phối sẽ làm gì? Quản lý hàng đợi? Hiện tại chúng ta gặp phải tình trạng nhân viên điều độ không còn làm đúng công việc (phái cử), và phải chịu trách nhiệm về việc ghi sổ và lưu giữ các danh sách, làm giảm hiệu quả hoạt động hơn nữa.

## 2.3. Xếp hàng

Để tránh lấn át bất kỳ ai, chúng tôi có thể thêm một vùng đệm giữa khách hàng và người điều phối.

Nhân viên mới này chịu trách nhiệm quản lý quan hệ khách hàng. Thay vì nói chuyện trực tiếp với người điều phối, khách hàng nói chuyện với người quản lý dịch vụ, chuyển các yêu cầu của người quản lý và tại một thời điểm nào đó trong tương lai sẽ nhận được cuộc gọi rằng nhiệm vụ của họ đã được hoàn thành. Các yêu cầu công việc được thêm vào hàng đợi công việc được ưu tiên (một chồng các đơn đặt hàng với đơn hàng quan trọng nhất ở trên cùng) và người quản lý này đợi một khách hàng khác bước qua cửa. Hình sau mô tả các tình huống:

Khách hàng

Khách hàng

Khách hàng

Hàng đợi được ưu tiên trong số

nhiệm vụ khách hàng là

quản lý

Điều phối viên lấy từ

xếp hàng và chuyển lại các gói

đã hoàn thành

Công nhân

Công nhân

Công nhân

Khi nhân viên không hoạt động, người điều phối có thể lấy mục đầu tiên trên ngăn xếp, chuyển bất kỳ gói nào mà nhân viên đã hoàn thành và thường duy trì một môi trường làm việc lành mạnh, nơi không có gì bị rơi hoặc mất. Nếu đến một thời điểm mà tất cả nhân viên đều nhàn rỗi và hàng đợi nhiệm vụ trống, văn phòng có thể ngủ một lúc cho đến khi khách hàng tiếp theo đến.

Mô hình cuối cùng này truyền cảm hứng cho thiết kế của Node. Sửa đổi chính là chiếm nhóm công nhân chỉ với các tác vụ I / O và ủy thác công việc còn lại cho luồng đơn của V8. Nếu một chương trình JavaScript được hiểu là máy khách, thì Node là trình quản lý dịch vụ chạy qua các hướng dẫn được cung cấp và ưu tiên chúng. Khi gặp phải một tác vụ có khả năng bị chặn (I / O, bộ hẹn giờ và luồng), nó sẽ được chuyển cho người điều phối (nhóm luồng libuv). Nếu không, lệnh được xếp hàng đợi để vòng lặp sự kiện bật lên và thực thi.

## 2.4. Lắng nghe sự kiện

Trong chương trước, chúng ta đã được giới thiệu về EventEmitter giao diện. Đây là giao diện sự kiện chính mà chúng ta sẽ gặp khi chuyển chương này sang chương khác, vì nó cung cấp lớp nguyên mẫu cho nhiều đối tượng Node hiển thị các giao diện sự kiện, chẳng hạn như tệp và luồng mạng. Đa dạng đóng, thoát, dữ liệu, và các sự kiện khác được hiển thị bởi các API mô-đun khác nhau báo hiệu sự hiện diện của EventEmitter và chúng tôi sẽ tìm hiểu về các mô-đun và các trường hợp sử dụng này khi chúng tôi tiến bộ.

Thay vào đó, mục đích chính của phần này là thảo luận về một số nguồn sự kiện — Ký hiệu, giao tiếp quy trình con, sự kiện thay đổi hệ thống tệp và thực thi trì hoãn.

## 2.5. Tín hiệu

Theo nhiều cách, lập trình theo sự kiện giống như lập trình ngắt phần cứng. Ngắt làm đúng như tên gọi của chúng. Họ sử dụng khả năng của mình để làm gián đoạn bất cứ điều gì một bộ điều khiển hoặc CPU hoặc bất kỳ thiết bị nào khác đang làm, yêu cầu rằng nhu cầu cụ thể của họ phải được bảo dưỡng ngay lập tức.

Trên thực tế, Node quá trình đối tượng phơi bày tiêu chuẩn Giao diện hệ điều hành di động (POSIX) tên tín hiệu, sao cho quá trình nút có thể đăng ký các sự kiện hệ thống này.

Tín hiệu là một dạng truyền thông liên quá trình hạn chế được sử dụng trong Unix, Unix-like và các hệ điều hành tương thích POSIX khác. Nó là một thông báo không đồng bộ được gửi đến một quá trình hoặc một chuỗi cụ thể trong cùng một quá trình để thông báo cho nó về một sự kiện đã xảy ra.

- http://en.wikipedia.org/wiki/POSIX\_signal

Đây là một cách rất đơn giản và tự nhiên để hiển thị quá trình Node với các sự kiện tín hiệu của hệ điều hành (OS). Người ta có thể định cấu hình người nghe để bắt các tín hiệu hướng dẫn quy trình Node khởi động lại hoặc cập nhật một số tệp cấu hình hoặc đơn giản là dọn dẹp và tắt.

Ví dụ, SIGINT tín hiệu được gửi đến một quá trình khi thiết bị đầu cuối điều khiển của nó phát hiện Ctrl-C ( hoặc tương đương) tổ hợp phím. Tín hiệu này cho một quá trình biết rằng một ngắt đã được yêu cầu. Nếu một quy trình Node đã ràng buộc một lệnh gọi lại với sự kiện này, thì hàm đó có thể ghi lại yêu cầu trước khi kết thúc, thực hiện một số công việc dọn dẹp khác hoặc thậm chí bỏ qua yêu cầu:

setInterval(function() {}, 1e6); process.on('SIGINT', function() { console.log('SIGINT signal received'); process.exit(1); })

Ở đây chúng tôi đã thiết lập một khoảng thời gian trong tương lai xa để quá trình không kết thúc ngay lập tức và một trình nghe SIGINT. Khi người dùng gửi một Ctrl-C ngắt đến thiết bị đầu cuối kiểm soát quá trình này, thông báo Đã nhận được tín hiệu SIGINT sẽ được ghi trên thiết bị đầu cuối và quá trình sẽ kết thúc.

Bây giờ hãy xem xét một tình huống trong đó quá trình Node đang thực hiện một số công việc đang diễn ra, chẳng hạn như phân tích cú pháp nhật ký. Có thể hữu ích nếu có thể gửi tín hiệu cho quá trình đó, chẳng hạn như cập nhật các tệp cấu hình của bạn hoặc là khởi động lại quá trình quét. Bạn có thể muốn gửi các tín hiệu như vậy từ dòng lệnh. Bạn có thể muốn có một quy trình khác làm như vậy — một phương pháp được gọi là Giao tiếp giữa các quá trình (IPC).

Tạo một tệp có tên ipc.js chứa mã sau:

setInterval(function() {}, 1e6); process.on('SIGUSR1', function() { console.log('Got a signal!'); });

SIGUSR1 ( và SIGUSR2) là các tín hiệu do người dùng xác định (chúng được kích hoạt bởi không có hành động cụ thể nào). Điều này làm cho chúng trở thành tín hiệu lý tưởng cho chức năng tùy chỉnh.

Để gửi một lệnh đến một quy trình, bạn phải xác định ID quy trình (PID). Với

có PID trong tay, các quy trình có thể được giải quyết và do đó được giao tiếp với. Nếu PID được chỉ định cho ipc.js sau khi được chạy qua Node là 123, sau đó chúng tôi có thể gửi quá trình đó SIGUSR1 ra hiệu bằng dòng lệnh sau:

kill –s SIGUSR1 123

Một cách đơn giản để tìm PID cho một quy trình Node nhất định trong UNIX là tìm kiếm danh sách quy trình hệ thống để biết tên của chương trình cho biết quy trình đang chạy. Nếu ipc.js hiện đang thực thi, PID của nó được tìm thấy bằng cách nhập dòng lệnh sau vào bảng điều khiển / thiết bị đầu cuối:

ps aux | grep ipc.js Thử nó.

## 2.6. Dĩa

Một phần cơ bản trong thiết kế của Node là tạo hoặc phân nhánh các quy trình khi thực hiện song song hoặc mở rộng quy mô hệ thống — chẳng hạn như tạo luồng luồng. Chúng tôi sẽ sử dụng các quy trình con theo nhiều cách khác nhau trong suốt cuốn sách này và học cách tạo và sử dụng chúng. Ở đây, trọng tâm sẽ là tìm hiểu cách xử lý các sự kiện giao tiếp giữa các quy trình con.

Để tạo một quy trình con, người ta chỉ cần gọi cái nĩa phương pháp của quy trình\_ con , chuyển cho nó tên của một tệp chương trình để thực thi trong quy trình mới:

var cp = request ('child\_process'); var child = cp.fork (\_\_ dirname + '/lovechild.js');

Bằng cách này, bất kỳ số lượng quy trình con nào cũng có thể được tiếp tục chạy. Ngoài ra, trên các máy đa lõi, các quy trình đã phân nhánh sẽ được phân phối (bởi HĐH) đến các lõi khác nhau. Trải rộng các quy trình nút trên các lõi (thậm chí cả các máy khác) và quản lý IPC là (một) cách để mở rộng quy mô ứng dụng Node theo cách ổn định, dễ hiểu và có thể dự đoán được.

Mở rộng phần trước, bây giờ chúng ta có thể có quá trình phân nhánh (cha mẹ) gửi và lắng nghe, các thông điệp từ quá trình phân nhánh (con):

child.on('message', function(msg) { console.log('Child said: ', msg);

}); child.send("I love you");

Tương tự, tiến trình con (chương trình của nó được định nghĩa trong lovechild.js) có thể gửi và nghe tin nhắn:

// lovechild.js

process.on('message', function(msg) { console.log('Parent said: ', msg); process.send("I love you too");

});

Đang chạy parent.js nên fork một quy trình con và gửi tin nhắn cho con đó. Đứa trẻ nên trả lời bằng hiện vật:

Parent said: I love you

Child said: I love you too

Một ý tưởng rất mạnh mẽ khác là chuyển một máy chủ mạng một đối tượng cho một đối tượng con. Kỹ thuật này cho phép nhiều quy trình, bao gồm cả quy trình chính, chia sẻ trách nhiệm phục vụ các yêu cầu kết nối, phân tán tải trên các lõi.

Ví dụ: chương trình sau sẽ khởi động một máy chủ mạng, rẽ nhánh một quy trình con và chuyển con này làm tham chiếu máy chủ:

var child = require('child\_process').fork('./child.js'); var server = require('net').createServer(); server.on('connection', function(socket) { socket.end('Parent handled connection');

});

server.listen(8080, function() {

child.send("The parent message", server); });

Ngoài việc chuyển một thông báo tới một tiến trình con làm đối số đầu tiên gửi, mã trước đó cũng gửi xử lý máy chủ cho chính nó như một đối số thứ hai. Máy chủ con của chúng tôi hiện có thể trợ giúp công việc kinh doanh dịch vụ của gia đình:

// child.js

process.on('message', function(msg, server) { console.log(msg);

server.on('connection', function(socket) { socket.end('Child handled connection');

});

});

Quy trình con này sẽ in ra thông báo đã gửi tới bảng điều khiển của bạn và bắt đầu lắng nghe các kết nối, chia sẻ tay cầm máy chủ đã gửi. Liên tục kết nối với máy chủ này tại localhost: 8080 sẽ cho kết quả hoặc Kết nối được xử lý con hoặc là Kết nối do cha mẹ xử lý đang được hiển thị; hai quy trình riêng biệt đang cân bằng tải máy chủ. Cần phải rõ rằng kỹ thuật này, khi được kết hợp với giao thức nhắn tin liên quá trình đơn giản đã được thảo luận trước đây, chứng minh cách sáng tạo của Ryan Dahl thành công trong việc cung cấp một cách dễ dàng để xây dựng các chương trình mạng có thể mở rộng.

Chúng ta sẽ thảo luận về tính năng mới của Node cụm mô-đun, mở rộng (và đơn giản hóa) kỹ thuật đã thảo luận trước đó trong các chương sau. Nếu bạn quan tâm đến cách các tay cầm máy chủ được chia sẻ, hãy truy cập cụm tài liệu tại liên kết sau:

http://nodejs.org/api/cluster.html

Đối với những người thực sự tò mò, hãy kiểm tra cụm mã chính nó tại:

https://github.com/joyent/node/blob/ c668185adde3a474585a11f172b8387e270ec23b / lib / cluster. js # L523-558

## 2.7. Tệp sự kiện

Hầu hết các ứng dụng sử dụng một số hệ thống tệp, đặc biệt là những hệ thống có chức năng như dịch vụ web. Ngoài ra, một ứng dụng chuyên nghiệp có thể sẽ ghi lại thông tin về việc sử dụng, các chế độ xem dữ liệu kết xuất trước trong bộ nhớ cache hoặc thực hiện các thay đổi thường xuyên khác đối với các tệp và cấu trúc thư mục.

Node cho phép các nhà phát triển đăng ký nhận thông báo về các sự kiện tệp thông qua fs.watch phương pháp. Các đồng hồ đeo tay phương thức sẽ phát các sự kiện đã thay đổi trên cả hai tệp và các thư mục.

đồng hồ chấp nhận ba đối số, theo thứ tự:

Đường dẫn tệp hoặc thư mục đang được xem. Nếu tệp không tồn tại, ENOENT (không có thực thể) lỗi sẽ được ném ra, vì vậy sử dụng fs.exists tại một số điểm hữu ích trước đó được khuyến khích.

Một đối tượng tùy chọn tùy chọn:

° dai dẳng (Boolean): Nút giữ cho các quy trình tồn tại miễn là ở đó là "việc cần làm". Theo mặc định, một trình theo dõi tệp đang hoạt động sẽ hoạt động như một cờ liên tục cho Node. Đặt tùy chọn này thành cờ sai không phải giữ cho quá trình chung tồn tại nếu người xem là hoạt động duy nhất giữ cho nó chạy.

Hàm nghe, nhận hai đối số:

° Tên của sự kiện thay đổi (một trong số đổi tên hoặc là thay đổi).

° Tên tệp đã được thay đổi (quan trọng khi xem thư mục).

Một số hệ điều hành sẽ không phải trả về đối số này.

Ví dụ này sẽ tự thiết lập một trình theo dõi, thay đổi tên tệp của chính nó và thoát:

var fs = require('fs');

fs.watch(\_\_filename, { persistent: false }, function(event, filename)

{

console.log(event); console.log(filename); })

setImmediate(function() {

fs.rename(\_\_filename, \_\_filename + '.new', function() {}); });

Hai dòng, đổi tên và tên của tệp gốc, lẽ ra phải được in ra bảng điều khiển.

Các kênh của Người theo dõi có thể bị đóng bất kỳ lúc nào bằng đoạn mã sau:

var w = fs.watch('file', function(){})

w.close();

Cần lưu ý rằng fs.watch phụ thuộc rất nhiều vào cách hệ điều hành chủ quản lý các sự kiện tệp và theo tài liệu Node:

"Các fs.watch API không nhất quán 100% trên các nền tảng và không khả dụng trong một số trường hợp. "

Tác giả đã có những trải nghiệm rất tốt với mô-đun trên nhiều hệ thống khác nhau, chỉ lưu ý rằng đối số tên tệp là vô giá trị trong các lệnh gọi lại trên việc triển khai OS X. Tuy nhiên, hãy đảm bảo chạy thử nghiệm trên kiến trúc cụ thể của bạn— hãy tin tưởng nhưng hãy xác minh.

## 2.8. Thực hiện hoãn lại

Đôi khi người ta cần trì hoãn việc thực thi một hàm. JavaScript truyền thống sử dụng bộ hẹn giờ cho mục đích này, setTimeout và setInterval

chức năng. Node giới thiệu một quan điểm khác về các trình xác định, chủ yếu là phương tiện kiểm soát thứ tự mà một lệnh gọi lại thực thi liên quan đến các sự kiện I / O, cũng như các sự kiện hẹn giờ một cách chính xác.

Chúng ta sẽ tìm hiểu thêm về thứ tự này trong cuộc thảo luận vòng lặp sự kiện diễn ra sau đây. Hiện tại, chúng tôi sẽ kiểm tra hai loại nguồn sự kiện được hoãn lại cung cấp cho nhà phát triển khả năng lên lịch thực thi gọi lại xảy ra trước hoặc sau khi xử lý các sự kiện I / O được xếp hàng đợi.

## 2.9. process.nextTick

Một phương thức của Node gốc quá trình mô-đun, process.nextTick tương tự như quen thuộc setTimeout trong đó nó trì hoãn việc thực thi hàm gọi lại của nó cho đến một thời điểm nào đó trong tương lai. Tuy nhiên, so sánh không chính xác; một danh sách tất cả được yêu cầu nextTick gọi lại được đặt ở đầu hàng đợi sự kiện và được xử lý, toàn bộ và theo thứ tự, trước I / O hoặc các sự kiện hẹn giờ và sau thực thi tập lệnh hiện tại (mã JavaScript thực thi đồng bộ trên chuỗi V8).

Việc sử dụng chính của nextTick trong một chức năng là hoãn việc phát các sự kiện kết quả cho người nghe trên ngăn xếp thực thi hiện tại cho đến khi người gọi có cơ hội đăng ký trình nghe sự kiện — để cho chương trình hiện đang thực thi có cơ hội liên kết các lệnh gọi lại với EventEmitter.emit sự kiện. Nó có thể được coi là

một mẫu được sử dụng ở bất kỳ nơi nào hành vi không đồng bộ nên được mô phỏng. Ví dụ: hãy tưởng tượng một hệ thống tra cứu có thể lấy từ bộ nhớ cache hoặc lấy dữ liệu mới từ kho dữ liệu. Bộ nhớ đệm nhanh và không cần lệnh gọi lại, trong khi lệnh gọi I / O dữ liệu sẽ cần chúng. Nhu cầu gọi lại trong trường hợp thứ hai lập luận cho việc mô phỏng hành vi gọi lại với nextTick Trong trường hợp đầu tiên. Điều này cho phép một API nhất quán, cải thiện độ rõ ràng của việc triển khai mà không tạo gánh nặng cho nhà phát triển trách nhiệm xác định xem có sử dụng lệnh gọi lại hay không.

Đoạn mã sau dường như thiết lập một giao dịch đơn giản; khi một ví dụ của EventEmitter phát ra một khởi đầu Nhật ký sự kiện " Đã bắt đầu" vào bảng điều khiển:

var events = require('events');

function getEmitter() {

var emitter = new events.EventEmitter();

emitter.emit('start'); return emitter; } var myEmitter = getEmitter();

myEmitter.on("start", function() { console.log("Started"); });

Tuy nhiên, kết quả mong đợi sẽ không xảy ra. Bộ phát sự kiện được khởi tạo bên trong

getEmitter phát ra " khởi đầu" trước khi được trả lại, thực hiện sai nhiệm vụ tiếp theo của một người nghe, đến trễ một bước, bỏ lỡ thông báo sự kiện.

Để giải quyết tình trạng chủng tộc này, chúng ta có thể sử dụng process.nextTick:

var events = require('events'); function getEmitter() {

var emitter = new events.EventEmitter(); process.nextTick(function() { emitter.emit('start');

});

return emitter;

}

var myEmitter = getEmitter(); myEmitter.on('start', function() { console.log('Started'); })

Đây là phần đính kèm của băt đâu xử lý được phép xảy ra trước để phát ra sự kiện bắt đầu bởi bộ phát được khởi tạo trong getEmitter.

Vì có thể gọi đệ quy nextTick, có thể dẫn đến một vòng lặp đệ quy vô hạn nextTick cuộc gọi (bỏ đói vòng lặp sự kiện, ngăn chặn I / O), tồn tại một cơ chế an toàn dự phòng trong Node giới hạn số lần đệ quy nextTick

các cuộc gọi được đánh giá trước khi mang lại I / O: process.maxTickDepth. Đặt giá trị này (mặc định là 1000) nếu một cấu trúc như vậy trở nên cần thiết — mặc dù những gì bạn có thể muốn sử dụng trong trường hợp như vậy là thiết lập ngay lập tức.

## 2.10. setIm Instant

setIm Instant về mặt kỹ thuật là một thành viên của lớp bộ định thời ( setInterval, setTimeout). Tuy nhiên, không có cảm giác về thời gian gắn liền với nó — không có số mili giây để đợi đối số sẽ được gửi. Phương pháp này thực sự phù hợp với chị em hơn process.nextTick, khác nhau theo một cách rất quan trọng; trong khi các cuộc gọi lại được xếp hàng đợi bởi nextTick sẽ thực hiện trước I / O và các sự kiện hẹn giờ, các cuộc gọi lại được xếp hàng đợi bởi setIm Instant sẽ được gọi sau Sự kiện I / O.

Việc đặt tên của hai phương thức này là khó hiểu: nextTick xảy ra trước thiết lập ngay lập tức.

Phương thức này phản ánh hành vi tiêu chuẩn của bộ định thời trong đó lệnh gọi của nó sẽ trả về một đối tượng có thể được chuyển đến hủy bỏ ngay lập tức, hủy bỏ setIm Instant theo cùng một cách celTimeout hủy bỏ bộ hẹn giờ với setTimeout.

## 2.11. Bộ hẹn giờ

Bộ hẹn giờ được sử dụng để lên lịch các sự kiện trong tương lai. Chúng được sử dụng khi một người tìm cách trì hoãn việc thực thi một số khối mã cho đến khi một số mili giây cụ thể trôi qua, để lên lịch thực thi định kỳ một chức năng cụ thể hoặc sắp xếp một số chức năng ngay lập tức cho phần sau.

JavaScript cung cấp hai bộ định thời không đồng bộ: setInterval () và setTimeout ().

Người ta giả định rằng người đọc đã biết đầy đủ về cách đặt (và hủy) các bộ hẹn giờ này, vì vậy sẽ có rất ít thời gian dành cho việc thảo luận về cú pháp. Thay vào đó, chúng tôi sẽ tập trung nhiều hơn vào các chi tiết "gotchas" và "ít nổi tiếng hơn" về thời gian chờ và khoảng thời gian.

Điểm mấu chốt sẽ là điều này: khi sử dụng bộ hẹn giờ, người ta không nên đưa ra giả định về số lượng thực tế thời gian sẽ hết hạn trước khi lệnh gọi lại được đăng ký cho bộ hẹn giờ này kích hoạt hoặc về thứ tự các lệnh gọi lại. Bộ hẹn giờ nút là không phải ngắt. Bộ hẹn giờ chỉ đơn giản là hứa thực hiện càng gần càng tốt với thời gian đã chỉ định (mặc dù chưa bao giờ trước đó), như với mọi nguồn sự kiện khác, lập lịch vòng lặp sự kiện.

Ít nhất một điều bạn có thể chưa biết về bộ hẹn giờ ...

Tất cả chúng ta đều quen thuộc với các đối số tiêu chuẩn để setTimeout: một chức năng gọi lại và khoảng thời gian chờ. Bạn có biết rằng nhiều đối số bổ sung được chuyển cho hàm gọi lại không? setTimeout (gọi lại, thời gian, [passArg1, passArg2…])

## 2.12. setTimeout

Thời gian chờ được sử dụng để trì hoãn việc thực thi một hàm cho đến một số phần nghìn giây trong tương lai:

Hãy xem xét đoạn mã sau:

setTimeout (a, 1000); setTimeout (b, 1001);

Người ta mong đợi chức năng đó b sẽ thực thi sau chức năng a. Tuy nhiên, điều này không thể được đảm bảo— a có thể làm theo b, Hoặc cách khác xung quanh.

Bây giờ, hãy xem xét sự khác biệt nhỏ hiện diện trong đoạn mã sau:

setTimeout (a, 1000); setTimeout (b, 1000);

Trình tự thực hiện của a và b có thể dự đoán được trong trường hợp này. Về cơ bản, Node duy trì một bản đồ đối tượng nhóm các cuộc gọi lại có độ dài thời gian chờ giống hệt nhau. Isaac Schlueter, lãnh đạo hiện tại của dự án Node, nói theo cách này:

Node sử dụng một đối tượng bộ hẹn giờ mức thấp cho mỗi giá trị thời gian chờ. Nếu bạn đính kèm nhiều lệnh gọi lại cho một giá trị thời gian chờ, chúng sẽ xảy ra theo thứ tự, vì chúng đang ngồi trong một hàng đợi. Tuy nhiên, nếu chúng ở các giá trị thời gian chờ khác nhau, thì chúng sẽ sử dụng bộ hẹn giờ trong các chuỗi khác nhau và do đó phải tuân theo sự thay đổi của bộ lập lịch [CPU].

- https://groups.google.com/forum/#!msg/nodejs-dev/kiowz4iht4Q/T0RuSwAeJV0J

Thứ tự các lệnh gọi lại hẹn giờ được đăng ký trong một phạm vi thực thi giống hệt nhau không xác định được thứ tự thực hiện cuối cùng trong mọi trường hợp.

Ngoài ra, tồn tại thời gian chờ tối thiểu là một phần nghìn giây cho thời gian chờ. Việc chuyển giá trị 0, -1 hoặc không phải số sẽ được chuyển thành giá trị nhỏ nhất này.

## 2.13. setInterval

Người ta có thể nghĩ đến nhiều trường hợp có thể thực thi định kỳ một hàm sẽ hữu ích. Thăm dò một nguồn dữ liệu vài giây một lần và đẩy các bản cập nhật là một kiểu phổ biến. Chạy bước tiếp theo trong một hoạt ảnh cứ sau vài mili giây là một trường hợp sử dụng khác, cũng như thu thập rác. Đối với những trường hợp này setInterval là một công cụ tốt:

var khoảngId = setInterval (function () {...}, 100);

Cứ sau 100 mili giây, hàm gọi lại đã gửi sẽ thực thi, một quá trình có thể bị hủy bỏ với clearInterval (khoảng thời gian).

Thật không may, như với setTimeout, hành vi này không phải lúc nào cũng đáng tin cậy. Điều quan trọng, nếu hệ thống chậm trễ (chẳng hạn như một số chặn viết sai trong khi loop) chiếm vòng lặp sự kiện trong một khoảng thời gian, các khoảng thời gian được đặt trước và hoàn thành trong khoảng thời gian tạm thời đó sẽ có kết quả của chúng được xếp hàng đợi trên ngăn xếp. Khi vòng lặp sự kiện được bỏ chặn và mở, tất cả các lệnh gọi lại khoảng thời gian sẽ được kích hoạt theo trình tự, về cơ bản là ngay lập tức, làm mất bất kỳ loại thời gian trễ nào mà chúng dự định.

May mắn thay, không giống như JavaScript dựa trên trình duyệt, các khoảng thời gian khá đáng tin cậy hơn trong Node, thường có thể duy trì tính chu kỳ dự kiến trong các tình huống sử dụng bình thường.

## 2.14. unf và ref

Một chương trình Node không sống sót mà không có lý do để làm như vậy. Một tiến trình sẽ tiếp tục chạy miễn là vẫn còn các lệnh gọi lại đang chờ xử lý. Khi những thứ đó được xóa, quá trình Node không còn gì để làm và nó sẽ thoát.

Ví dụ: đoạn mã ngớ ngẩn sau đây sẽ giữ cho tiến trình Node chạy mãi mãi:

Var intervalId = setInterval(function() {}, 1000);

Mặc dù hàm gọi lại đã đặt không có gì hữu ích hoặc thú vị, nó vẫn tiếp tục được gọi — và đây là hành vi đúng, vì một khoảng thời gian sẽ tiếp tục chạy cho đến khi clearInterval được sử dụng để ngăn chặn nó.

Có những trường hợp sử dụng bộ hẹn giờ để làm điều gì đó thú vị với I / O bên ngoài, hoặc một số cấu trúc dữ liệu hoặc giao diện mạng mà khi các nguồn sự kiện bên ngoài đó ngừng xảy ra hoặc biến mất, bộ đếm thời gian tự dừng là cần thiết. Thông thường người ta sẽ bẫy không liên quan trạng thái của bộ hẹn giờ ở một nơi khác trong chương trình và hủy bộ hẹn giờ từ đó. Điều này có thể trở nên khó khăn hoặc thậm chí là vụng về, vì giờ đây cần phải có một mớ mối quan tâm không cần thiết, một mức độ phức tạp hơn.

Các làm hư cho phép nhà phát triển xác nhận các hướng dẫn sau: khi bộ đếm thời gian này là nguồn sự kiện duy nhất còn lại để vòng lặp sự kiện xử lý, hãy tiếp tục và kết thúc quá trình.

Hãy kiểm tra chức năng này với ví dụ ngớ ngẩn trước đó của chúng tôi, điều này sẽ dẫn đến quá trình kết thúc thay vì chạy mãi mãi:

var intervalId = setInterval(function() {}, 1000); intervalId.unref();

Lưu ý rằng làm hư là một phương thức của giá trị không trong suốt được trả về khi khởi động bộ đếm thời gian (là một đối tượng).

Bây giờ hãy thêm một nguồn sự kiện bên ngoài, một bộ đếm thời gian. Sau khi nguồn bên ngoài đó được làm sạch (trong khoảng 100 mili giây), quá trình sẽ kết thúc. Chúng tôi gửi thông tin đến bảng điều khiển để ghi lại những gì đang xảy ra:

setTimeout(function() { console.log("now stop");

}, 100);

var intervalId = setInterval(function() { console.log("running")

}, 1); Bạn có thể đưa bộ hẹn giờ trở về hoạt động bình thường của nó với ref, sẽ hoàn tác một làm hư phương pháp:

var khoảngId = setInterval (function () {}, 1000); khoảngId.unref (); khoảngId.ref ();

Quá trình được liệt kê sẽ tiếp tục vô thời hạn, như trong ví dụ ngớ ngẩn ban đầu của chúng tôi.

## 2.15. Hiểu vòng lặp sự kiện

Node xử lý các hướng dẫn JavaScript bằng một luồng duy nhất. Trong chương trình JavaScript của bạn, không có hai hoạt động nào sẽ thực thi chính xác cùng một lúc, như có thể xảy ra trong môi trường đa luồng. Hiểu được thực tế này là điều cần thiết để hiểu cách một chương trình hoặc quy trình Node được thiết kế và chạy.

Điều này không có nghĩa là chỉ có một luồng đang được sử dụng trên máy lưu trữ tiến trình Node này. Đơn giản chỉ cần viết một cuộc gọi lại không tạo ra sự song song một cách kỳ diệu! Gợi lại Chương 1, Tìm hiểu Môi trường Nút, và cuộc thảo luận của chúng tôi về quá trình object — Tính đơn giản "single thread" của Node trên thực tế là một sự trừu tượng được tạo ra vì lợi ích của các nhà phát triển. Tuy nhiên, điều quan trọng cần nhớ là có nhiều luồng chạy trong nền quản lý I / O (và những thứ khác), và những luồng này chèn các hướng dẫn không thể đoán trước, ban đầu được đóng gói dưới dạng lệnh gọi lại, vào một chuỗi JavaScript duy nhất để xử lý.

Node thực hiện từng hướng dẫn một cho đến khi không có hướng dẫn nào khác để thực thi, không có thêm đầu vào hoặc đầu ra để phát trực tuyến và không có lệnh gọi lại nào đang chờ xử lý.

Ngay cả các sự kiện bị hoãn lại (chẳng hạn như thời gian chờ) yêu cầu một sự kiện gián đoạn cuối cùng trong vòng lặp sự kiện để thực hiện lời hứa của chúng.

Ví dụ như sau trong khi vòng lặp sẽ không bao giờ kết thúc:

var stop = false; setTimeout(function() { stop = true; }, 1000); while(stop === false) {};

Mặc dù người ta có thể mong đợi, trong khoảng một giây, việc gán Boolean thật đến biến dừng lại, vấp ngã trong khi có điều kiện và làm gián đoạn vòng lặp của nó, điều này sẽ không bao giờ xảy ra. Tại sao? Điều này trong khi vòng lặp bỏ đói vòng lặp sự kiện bằng cách chạy vô hạn, tham lam kiểm tra và kiểm tra lại một giá trị không bao giờ có cơ hội để thay đổi, vì vòng lặp sự kiện không bao giờ có cơ hội lập lịch gọi lại bộ hẹn giờ của chúng tôi để thực thi.

Như vậy, lập trình Node có nghĩa là lập trình vòng lặp sự kiện. Trước đây chúng ta đã thảo luận về nguồn sự kiện được xếp hàng đợi và được sắp xếp và sắp xếp theo thứ tự khác trên vòng lặp sự kiện này — sự kiện I / O, sự kiện hẹn giờ, v.v.

Khi viết mã không xác định, điều bắt buộc là không có giả định nào về các lệnh gọi lại cuối cùng được thực hiện. Sự trừu tượng là Node che giấu sự phức tạp của nhóm luồng mà luồng JavaScript chính đơn giản trôi nổi, dẫn đến một số kết quả đáng ngạc nhiên.

Bây giờ chúng ta sẽ hoàn thiện sự hiểu biết chung này với nhiều thông tin hơn về cách chính xác, thứ tự thực hiện lệnh gọi lại cho mỗi kiểu này được xác định trong vòng lặp sự kiện của Node.

## 2.16. Bốn nguồn chân lý

Chúng ta đã tìm hiểu về bốn nhóm chính của nguồn sự kiện hoãn lại, có vị trí và mức độ ưu tiên trên ngăn xếp mà chúng ta sẽ chứng minh bây giờ:

Các khối thực thi: Các khối mã JavaScript bao gồm chương trình Node, là biểu thức, vòng lặp, hàm, v.v. Điêu nay bao gôm

EventEmitter các sự kiện được phát ra trong ngữ cảnh thực thi hiện tại.

Bộ hẹn giờ: Các cuộc gọi lại được hoãn lại vào một thời điểm nào đó trong tương lai được chỉ định bằng mili giây, chẳng hạn như setTimeout và setInterval.

I / O: Các lệnh gọi lại đã chuẩn bị trở lại chuỗi chính sau khi được ủy quyền cho nhóm luồng được quản lý của Node, chẳng hạn như các cuộc gọi hệ thống tệp và trình nghe mạng.

Các khối thực thi hoãn lại: Chủ yếu là các chức năng được sắp xếp trên ngăn xếp theo các quy tắc của setIm Instant và tiếp theo

Chúng tôi đã tìm hiểu cách thức thực hiện hoãn lại setIm Instant sắp xếp các cuộc gọi lại của nó sau I / O gọi lại trong hàng đợi sự kiện và nextTick sắp xếp các cuộc gọi lại của nó trước I / O và gọi lại hẹn giờ.

Một thách thức cho người đọc. Sau khi chạy đoạn mã sau, thứ tự mong đợi của các tin nhắn đã ghi là gì?

var fs = request ('fs');

var fs = require('fs');

var EventEmitter = require('events').EventEmitter; var pos = 0;

var messenger = new EventEmitter();

// Listener for EventEmitter messenger.on("message", function(msg) { console.log(++pos + " MESSAGE: " + msg);

});

// (A) FIRST

console.log(++pos + " FIRST");

// (B) NEXT process.nextTick(function() { console.log(++pos + " NEXT")

})

// (C) QUICK TIMER setTimeout(function() { console.log(++pos + " QUICK TIMER")

}, 0)

// (D) LONG TIMER setTimeout(function() { console.log(++pos + " LONG TIMER")

}, 10)

// (E) IMMEDIATE setImmediate(function() { console.log(++pos + " IMMEDIATE") })

// (F) MESSAGE HELLO! messenger.emit("message", "Hello!");

// (G) FIRST STAT fs.stat(\_\_filename, function() { console.log(++pos + " FIRST STAT");

});

// (H) LAST STAT fs.stat(\_\_filename, function() { console.log(++pos + " LAST STAT");

});

// (I) LAST console.log(++pos + " LAST");

The output of is program is:

FIRST (A).

MESSAGE: Hello! (F).

LAST (I).

NEXT (B).

QUICK TIMER (C).

FIRST STAT (G).

LAST STAT (H).

IMMEDIATE (E).

LONG TIMER (D).

Hãy phá vỡ đoạn mã trước đó:

A, F và tôi thực thi trong luồng chương trình chính và như vậy, chúng sẽ có mức độ ưu tiên đầu tiên trong luồng chính (điều này là hiển nhiên; JavaScript của bạn thực thi các hướng dẫn của nó theo thứ tự được viết, bao gồm cả việc thực thi đồng bộ phát ra gọi lại).

Với ngăn xếp cuộc gọi chính đã cạn kiệt, vòng lặp sự kiện bây giờ gần như đang đọc để xử lý các hoạt động I / O. Đây là thời điểm mà các yêu cầu nextTick được vinh dự xếp vào đầu hàng đợi sự kiện. Đây là khi B được hiển thị.

Phần còn lại của thứ tự phải rõ ràng. Bộ hẹn giờ và hoạt động I / O sẽ được xử lý tiếp theo, (C, G, H) sau đó là kết quả của setIm Instant callback (E), luôn đến sau khi thực hiện bất kỳ phản hồi I / O và bộ hẹn giờ nào.

Cuối cùng, thời gian chờ dài (D) đến, là một sự kiện tương đối xa.

Lưu ý rằng việc sắp xếp lại các biểu thức trong chương trình này sẽ không phải thay đổi thứ tự đầu ra (ngoài khả năng sắp xếp lại các kết quả STAT, điều này chỉ ngụ ý rằng chúng đã được trả về từ nhóm luồng theo thứ tự khác, vẫn là một nhóm theo đúng thứ tự liên quan đến hàng đợi sự kiện).

## 2.17. Gọi lại và lỗi

Các thành viên của cộng đồng Node phát triển các gói và dự án mới mỗi ngày. Do tính chất sự kiện của Node, các lệnh gọi lại tràn ngập các cơ sở mã này. Chúng tôi đã xem xét một số cách quan trọng trong đó các sự kiện có thể được xếp hàng đợi, gửi đi và xử lý thông qua việc sử dụng các lệnh gọi lại. Hãy dành một chút thời gian để phác thảo

các phương pháp hay nhất, đặc biệt là về quy ước thiết kế lệnh gọi lại và xử lý lỗi, đồng thời thảo luận một số mẫu hữu ích khi thiết kế chuỗi sự kiện và lệnh gọi lại phức tạp.

## 2.18. Quy ước

May mắn thay, những người tạo Node đã đồng ý với các quy ước lành mạnh về cách cấu trúc lệnh gọi lại từ rất sớm. Điều quan trọng là phải tuân theo truyền thống này. Sự sai lệch dẫn đến những bất ngờ, đôi khi là những bất ngờ rất tồi tệ và nói chung làm như vậy sẽ tự động khiến một API trở nên khó xử, một đặc điểm mà các nhà phát triển khác sẽ nhanh chóng chán nản.

Một là trả về một kết quả hàm bằng cách thực hiện một lệnh gọi lại, xử lý các đối số nhận được bởi một lệnh gọi lại hoặc thiết kế chữ ký cho một lệnh gọi lại trong API của bạn. Bất kể tình huống nào đang được xem xét, người ta nên tuân theo quy ước liên quan đến trường hợp đó:

Đối số đầu tiên được trả về cho một hàm gọi lại là bất kỳ thông báo lỗi nào, tốt nhất là ở dạng một đối tượng lỗi. Nếu không có lỗi nào được báo cáo, vị trí này phải chứa vô giá trị giá trị.

Khi chuyển một lệnh gọi lại cho một hàm, nó phải được gán vào vị trí cuối cùng của chữ ký hàm. Các API nên được thiết kế nhất quán theo cách này.

Bất kỳ số lượng đối số nào có thể tồn tại giữa lỗi và vùng gọi lại.

Để tạo một đối tượng lỗi:

Lỗi mới ("Đối số phải là một Chuỗi!")

## 2.19. Biết lỗi của bạn

Điều tuyệt vời là cộng đồng Node đã tự động thông qua một quy ước bắt buộc các nhà phát triển phải siêng năng và báo cáo lỗi. Tuy nhiên, người ta sẽ làm gì với lỗi một khi chúng được nhận?

Nói chung là một ý tưởng rất hay khi tập trung xử lý lỗi trong một chương trình. Thường thì một hệ thống xử lý lỗi tùy chỉnh sẽ được thiết kế, có thể gửi tin nhắn cho khách hàng, thêm vào nhật ký, v.v. Đôi khi tốt nhất là phi lỗi, tạm dừng quá trình.

Node cung cấp các công cụ nâng cao hơn để xử lý lỗi. Đặc biệt, Node's

miền hệ thống giúp giải quyết một vấn đề mà các hệ thống có sự kiện gặp phải: làm cách nào có thể tạo dấu vết ngăn xếp nếu toàn bộ tuyến của một cuộc gọi đã bị xóa khi nó chuyển từ lệnh gọi lại này sang lệnh gọi lại?

Mục đích của miền rất đơn giản: hàng rào và gắn nhãn bối cảnh thực thi sao cho tất cả các sự kiện xảy ra bên trong nó đều được xác định như vậy, cho phép nhiều dấu vết ngăn xếp thông tin hơn. Bằng cách tạo một số miền khác nhau cho mỗi phân đoạn quan trọng của chương trình của bạn, một chuỗi lỗi có thể được hiểu đúng.

Ngoài ra, điều này cung cấp một cách để bắt lỗi và xử lý chúng, thay vì cho phép toàn bộ quy trình Node của bạn sụp đổ.

Trong ví dụ sau, chúng tôi sẽ tạo hai miền: appDomain và fsDomain.

var domain = require("domain"); var fs = require("fs");

var fsDomain = domain.create(); fsDomain.on("error", function(err) { console.error("FS error", err); });

var appDomain = domain.create(); appDomain.on('error', function(err) { console.log("APP error", err); });

Bây giờ chúng tôi gói chương trình chính trong appDomain, và hệ thống tệp gọi trong fsDomain.

Sau đó, chúng tôi tạo ra một lỗi trong fsDomain bằng cách cố gắng mở một tệp không tồn tại:

appDomain.run(function() { process.nextTick(function() { fsDomain.run(function() {

fs.open('no\_file\_here', 'r', function(err, fd) {

if(err) { throw err;

}

appDomain.dispose();

});

});

});

});

Khi mã trước đó thực thi, một cái gì đó tương tự như thế này sẽ được gửi lại tới thiết bị đầu cuối:

FS error { [Error: ENOENT, open 'non-existent file'] errno: 34, code: 'ENOENT', path: 'non-existent file', domain:

{ domain: null,

\_events: { error: [Function] },

\_maxListeners: 10, members: [] }, domainThrown: true }

Bây giờ, hãy tạo một lỗi trong appDomain bằng cách thêm mã này, mã này sẽ tạo ra lỗi tham chiếu (như không b được định nghĩa):

appDomain.run (function () { a = b;

process.nextTick (function () {

. . .

Một lỗi tương tự như trong mã quý phải được tạo và báo cáo bởi appDomain.

Chú ý lệnh appDomain.dispose. Vì việc duy trì các ngữ cảnh lỗi này sẽ tiêu tốn một số bộ nhớ, nên tốt nhất là loại bỏ chúng khi không còn cần thiết nữa — ví dụ: sau khi mã chứa chúng đã được thực thi thành công. Chúng ta sẽ tìm hiểu các cách sử dụng nâng cao hơn của công cụ này khi chúng ta tiến vào các lãnh thổ phức tạp hơn.

Khi một ứng dụng phát triển về mức độ phức tạp, nó sẽ ngày càng hữu ích hơn để có thể bẫy lỗi và xử lý chúng đúng cách, có lẽ chỉ khởi động lại một phần của ứng dụng khi nó bị lỗi chứ không phải toàn bộ hệ thống.

## 2.20. Xây kim tự tháp

Đơn giản hóa các luồng điều khiển đã là mối quan tâm của cộng đồng Node kể từ khi bắt đầu dự án. Thật vậy, lời chỉ trích tiềm năng này là một trong những dự đoán đầu tiên của Ryan Dahl, người đã thảo luận rất lâu về nó trong buổi nói chuyện khi anh ấy giới thiệu Node với cộng đồng các nhà phát triển JavaScript.

Bởi vì việc thực thi mã trì hoãn thường yêu cầu lồng ghép các lệnh gọi lại bên trong các lệnh gọi lại, một chương trình Node đôi khi có thể bắt đầu giống như một kim tự tháp nghiêng, còn được gọi là "Kim tự tháp của Doom".

Theo đó, có một số gói Node có sẵn để giải quyết vấn đề, sử dụng các chiến lược đa dạng như tương lai, sợi, thậm chí cả mô-đun C ++ hiển thị trực tiếp các luồng hệ thống. Người đọc được khuyến khích thử nghiệm với những điều này:

|  |  |
| --- | --- |
| Không đồng bộ  Thuần hóa Sợi  Lời hứa | https://github.com/caolan/async https://github.com/maxtaco/tamejs https://github.com/laverdet/nodesợi https://github.com/kriskowal/q |

Một điểm chung thú vị hơn có sẵn ở đây để chúng tôi xem xét về các lựa chọn API trong Node. Dahl có thể đã phản ứng với lời chỉ trích này, ví dụ, biến một trong những thư viện được liệt kê thành phần cốt lõi của Node, hoặc thực sự thay đổi toàn bộ cách viết JavaScript. Thay vào đó, nó được để cho cộng đồng xác định các phương pháp hay nhất và viết các gói phù hợp. Đây là Nút đường.

Mikeal Rogers, khi thảo luận về lý do tại sao Promises bị loại bỏ khỏi lõi Node, đưa ra lập luận mạnh mẽ trong liên kết sau về lý do tại sao việc phát triển tính năng cho cộng đồng lại dẫn đến một sản phẩm cốt lõi mạnh hơn: http://www.futurealoof.com/posts/broken-promises.html

## 2.21. Cân nhắc

Bất kỳ nhà phát triển nào cũng thường xuyên đưa ra quyết định có tác động sâu rộng. Rất khó để dự đoán tất cả các hậu quả có thể xảy ra do một đoạn mã mới hoặc

một lý thuyết thiết kế mới. Vì lý do này, có thể hữu ích khi giữ cho hình dạng mã của bạn đơn giản và buộc bản thân phải tuân thủ nhất quán các thông lệ chung của các nhà phát triển Node khác. Đây là một số nguyên tắc bạn có thể thấy hữu ích, như sau:

Nói chung, hãy cố gắng nhắm vào mã nông. Kiểu tái cấu trúc này không phổ biến trong các môi trường không có sự kiện — hãy nhắc nhở bản thân về nó bằng cách thường xuyên đánh giá lại các điểm vào và ra và các chức năng được chia sẻ.

Nếu có thể, hãy cung cấp một ngữ cảnh chung cho mục nhập lại gọi lại. Đóng cửa là những công cụ rất mạnh trong JavaScript và phần mở rộng là Node. Miễn là độ dài khung ngữ cảnh của các lệnh gọi lại kèm theo không quá mức.

Đặt tên cho bạn các chức năng. Ngoài việc hữu ích trong các cấu trúc đệ quy sâu, mã gỡ lỗi dễ dàng hơn nhiều khi dấu vết ngăn xếp chứa các tên hàm riêng biệt, trái ngược với vô danh.

Suy nghĩ kỹ về các ưu tiên. Thứ tự, trong đó một kết quả nhất định đến hoặc một lệnh gọi lại được thực hiện, có thực sự quan trọng không? Điều quan trọng, nó có liên quan đến hoạt động I / O không? Nếu vậy, hãy cân nhắc nextTick và thiết lập ngay lập tức.

Cân nhắc sử dụng máy trạng thái hữu hạn để quản lý các sự kiện của bạn. Các máy trạng thái (đáng ngạc nhiên là) không được biểu diễn trong các cơ sở mã JavaScript. Khi một lệnh gọi lại vào lại luồng chương trình, nó có thể đã thay đổi trạng thái của ứng dụng của bạn và việc phát hành lệnh gọi không đồng bộ chính là một chỉ báo có khả năng trạng thái sắp thay đổi.

## 2.22. Lắng nghe các thay đổi tệp

Hãy áp dụng những gì chúng ta đã học được. Mục đích là tạo một máy chủ mà khách hàng có thể kết nối và nhận cập nhật từ Twitter. Trước tiên, chúng tôi sẽ tạo một quy trình để truy vấn Twitter cho bất kỳ tin nhắn nào có dấu thăng # nodejs và viết bất kỳ thông báo nào tìm thấy vào một tweets.txt tệp ở dạng khối 140 byte. Sau đó, chúng tôi sẽ tạo một máy chủ mạng để phát các thông báo này tới một máy khách. Các chương trình phát sóng đó sẽ được kích hoạt bằng cách viết các sự kiện trên tweets.txt tập tin. Bất cứ khi nào ghi xảy ra, các khối 140 byte được đọc không đồng bộ từ con trỏ đọc khách hàng cuối cùng đã biết. Điều này sẽ xảy ra cho đến khi chúng tôi đến cuối tệp, phát sóng khi chúng tôi tiếp tục. Cuối cùng, chúng tôi sẽ tạo một client.html trang yêu cầu, nhận và hiển thị các thông báo này. Mặc dù ví dụ này chắc chắn được tạo ra, nhưng nó thể hiện:

Lắng nghe hệ thống tệp để biết các thay đổi và phản hồi các sự kiện đó

Sử dụng các sự kiện dòng dữ liệu để đọc và ghi tệp

Phản hồi các sự kiện mạng

Sử dụng thời gian chờ cho trạng thái bỏ phiếu

Sử dụng chính máy chủ Node làm đài phát sự kiện mạng

Để xử lý việc phát sóng máy chủ, chúng tôi sẽ sử dụng Sự kiện do máy chủ gửi (SSE) giao thức, một giao thức mới đang được chuẩn hóa như một phần của HTML5.

Trước tiên, chúng ta sẽ tạo một máy chủ Node lắng nghe các thay đổi trên một tệp và phát bất kỳ nội dung mới nào đến máy khách. Mở trình chỉnh sửa của bạn và tạo một tệp server.js:

var fs = request ("fs"); var http = request ('http');

var theUser = null; var userPos

= 0; var tweetFile = "tweets.txt";

Chúng tôi sẽ chấp nhận một kết nối người dùng duy nhất, con trỏ có người dùng. Các userPos sẽ lưu trữ vị trí cuối cùng mà khách hàng này đọc từ trong tweetFile:

http.createServer(function(request, response) {

response.writeHead(200, {

'Content-Type': 'text/event-stream',

'Cache-Control': 'no-cache',

'Access-Control-Allow-Origin': '\*'

});

theUser = response;

response.write(':' + Array(2049).join(' ') + '\n'); response.write('retry: 2000\n');

response.socket.on('close', function() {

theUser = null; });

}).listen(8080);

Tạo một máy chủ HTTP lắng nghe trên cổng 8080, sẽ lắng nghe và xử lý một kết nối duy nhất, lưu trữ phản ứng đối số, đại diện cho đường ống kết nối máy chủ với máy khách. Các phản ứng đối số triển khai giao diện luồng có thể ghi, cho phép chúng tôi viết thông báo cho máy khách:

var sendNext = function (fd) { var buffer = new Buffer (140);

fs.read (fd, buffer, 0, 140, userPos \* 140, function (err, num) { if (! err && num> 0 && theUser) {

+ + userPos;

theUser.write ('data:' + buffer.toString ('utf-8', 0, num) + '\ n \ n');

return process.nextTick (function () { sendNext (fd);

});

}

}); }

Chúng tôi tạo ra một chức năng để gửi tin nhắn của khách hàng. Chúng tôi sẽ kéo bộ đệm 140 byte ra khỏi luồng có thể đọc được liên kết với tweets.txt tệp, tăng bộ đếm vị trí tệp của chúng tôi lên từng lần đọc. Chúng tôi ghi bộ đệm này vào luồng có thể ghi ràng buộc máy chủ của chúng tôi với máy khách. Khi hoàn tất, chúng tôi xếp hàng đợi một cuộc gọi lặp lại của cùng một hàm bằng cách sử dụng nextTick, lặp lại cho đến khi chúng tôi gặp lỗi, không nhận được dữ liệu hoặc máy khách ngắt kết nối:

hàm start () { fs.open (tweetFile, 'r', function (err, fd) { nếu (sai) { return setTimeout (start, 1000);

}

fs.watch (tweetFile, function (event, filename) { if (event === "change") { sendNext (fd);

}

});

});

}

khởi đầu();

Cuối cùng, chúng tôi bắt đầu quá trình bằng cách mở tweets.txt nộp hồ sơ và theo dõi mọi thay đổi, gọi điện sendNext bất cứ khi nào tweet mới được viết. Khi chúng tôi khởi động máy chủ, có thể chưa tồn tại tệp để đọc, vì vậy chúng tôi thăm dò ý kiến bằng cách sử dụng setTimeout cho đến khi một cái tồn tại.

Bây giờ chúng tôi có một máy chủ đang tìm kiếm các thay đổi tệp để phát sóng, chúng tôi cần tạo dữ liệu. Đầu tiên chúng tôi cài đặt TWiT Gói Twitter cho Node, thông qua npm.

Sau đó, chúng tôi tạo một quy trình có công việc duy nhất là ghi dữ liệu mới vào tệp:

var fs = require("fs"); var Twit = require('twit');

var twit = new Twit({ consumer\_key: 'your key', consumer\_secret: 'your secret', access\_token: 'your token', access\_token\_secret: 'your secret token' })

Để sử dụng ví dụ này, bạn sẽ cần tài khoản nhà phát triển Twitter. Ngoài ra, cũng có tùy chọn thay đổi mã liên quan, như sau, để chỉ cần ghi các chuỗi 140 byte ngẫu nhiên vào tweets.txt.

var tweetFile = "tweets.txt";

var writeStream = fs.createWriteStream (tweetFile, { cờ : "a"

});

Điều này thiết lập một con trỏ luồng tới cùng một tệp mà máy chủ của chúng tôi sẽ xem. Chúng tôi sẽ ghi vào tệp này:

var tweetFile = "tweets.txt";

var writeStream = fs.createWriteStream(tweetFile, { flags : "a" });

Vì tin nhắn Twitter không bao giờ dài hơn 140 byte nên chúng ta có thể đơn giản hóa thao tác đọc / ghi bằng cách luôn viết các đoạn 140 byte, ngay cả khi một số không gian đó trống. Khi chúng tôi nhận được bản cập nhật, chúng tôi sẽ tạo một bộ đệm số lượng tin nhắn x

140 rộng byte và ghi các phần 140 byte đó vào bộ đệm này:

var check = function() {

twit.get('search/tweets', { q: '#nodejs since:2013-01-01'

}, function(err, reply) {

var buffer = cleanBuffer(reply.statuses.length \* 140); reply.statuses.forEach(function(obj, idx) { buffer.write(obj.text, idx\*140, 140);

});

writeStream.write(buffer);

})

setTimeout(check, 10000);

}; check();Bây giờ chúng tôi tạo một hàm sẽ được hỏi cứ sau mười giây để kiểm tra các tin nhắn có chứa thẻ bắt đầu bằng # nodejs. Twitter trả về một mảng các đối tượng tin nhắn. Một thuộc tính đối tượng mà chúng tôi quan tâm là # bản văn của tin nhắn. Tính số byte cần thiết để biểu diễn các trang mới này ( 140 x số lượng tin nhắn),

tìm nạp một bộ đệm sạch và lấp đầy nó với các khối 140 byte cho đến khi tất cả các thư được viết. Cuối cùng, dữ liệu này được ghi vào tweets.txt , gây ra một sự kiện thay đổi xảy ra mà máy chủ của chúng tôi được thông báo.

Phần cuối cùng là trang khách hàng. Đây là một trang khá đơn giản và cách thức hoạt động của nó phải quen thuộc với người đọc. Điều duy nhất cần lưu ý là sử dụng SSE lắng nghe cổng 8080 trên localhost. Cần phải rõ ràng làm thế nào, khi nhận được một tweet mới từ máy chủ, một phần tử danh sách được thêm vào vùng chứa danh sách không có thứ tự # danh sách:

<!DOCTYPE html>

<html>

<head>

<title></title>

</head> <script>

window.onload = function() {

var list = document.getElementById("list");

var evtSource = new

EventSource("http://localhost:8080/events");

evtSource.onmessage = function(e) { var newElement = document.createElement("li"); newElement.innerHTML = e.data; list.appendChild(newElement);

}

}

</script>

<body>

<ul id="list"></ul>

</body>

</html>

Để đọc thêm về SSE, hãy tham khảo Chương 6, Tạo ứng dụng thời gian thực, hoặc bạn có thể truy cập liên kết sau:

https://developer.mozilla.org/en-US/docs/Server-sent\_ sự kiện / using\_server-sent\_events

## Tóm lược

Lập trình với các sự kiện không phải lúc nào cũng dễ dàng. Việc điều khiển và chuyển đổi ngữ cảnh, xác định mô hình thường gây nhầm lẫn cho những người mới với các hệ thống đã xảy ra sự kiện. Sự mất kiểm soát dường như liều lĩnh này và sự phức tạp dẫn đến khiến nhiều nhà phát triển tránh xa những ý tưởng này. Sinh viên trong các khóa học lập trình nhập môn thường phát triển

một tư duy trong đó luồng chương trình có thể được ra lệnh, trong đó luồng thực thi chương trình không tiến hành tuần tự từ A đến B có thể làm sai lệch sự hiểu biết.

Bằng cách kiểm tra sự phát triển của các vấn đề kiến trúc, Node hiện đang cố gắng để giải quyết các ứng dụng mạng — về quy mô, về tổ chức mã, về khối lượng dữ liệu và độ phức tạp chung, về nhận thức trạng thái cũng như về ranh giới quy trình và dữ liệu được xác định rõ ràng — chúng tôi đã học cách quản lý các sự kiện này hàng đợi có thể được thực hiện một cách thông minh. Chúng tôi đã thấy các nguồn sự kiện khác nhau được xếp chồng lên nhau có thể dự đoán được như thế nào để xử lý một vòng lặp sự kiện và cách các sự kiện trong tương lai xa có thể nhập và nhập lại ngữ cảnh bằng cách sử dụng cách đóng và sắp xếp gọi lại thông minh.

Bây giờ chúng ta đã có hiểu biết cơ bản về miền thiết kế và đặc điểm của Node, cụ thể là cách lập trình theo sự kiện được thực hiện bằng cách sử dụng nó. Bây giờ chúng ta hãy chuyển sang các ứng dụng lớn hơn, nâng cao hơn của kiến thức này.

# CHƯƠNG 3

# Luồng dữ liệu giữa NODE và máy khách

Giờ đây, chúng ta đã có một bức tranh rõ ràng hơn về cách thức mà đạo đức thiết kế tập trung vào I / O của Node được phản ánh qua các API mô-đun khác nhau của nó, mang lại một môi trường nhất quán và có thể dự đoán được để phát triển. Trong chương này, chúng ta sẽ khám phá cách dữ liệu, với nhiều hình dạng và kích thước, được lấy từ các tệp hoặc các nguồn khác, có thể được đọc, viết và thao tác dễ dàng như bằng cách sử dụng Node. Cuối cùng, chúng ta sẽ học cách sử dụng Node để phát triển các máy chủ nối mạng với giao diện I / O nhanh chóng hỗ trợ các ứng dụng đồng thời cao chia sẻ dữ liệu thời gian thực trên hàng nghìn máy khách cùng một lúc.

Những ai làm việc với phần mềm dựa trên Internet chắc hẳn đã nghe nói về "Dữ liệu lớn". Tầm quan trọng của hiệu quả I / O không bị mất đi đối với những người chứng kiến sự tăng trưởng bùng nổ về khối lượng dữ liệu.

Ngụ ý của việc mở rộng dữ liệu này là sự gia tăng kích thước của các đối tượng phương tiện riêng lẻ được truyền qua mạng. Các video được xem dài hơn. Các hình ảnh đang được xem lớn hơn. Các tìm kiếm rộng hơn và tập kết quả lớn hơn.

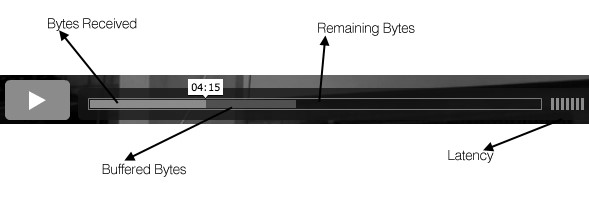
Ngoài ra, các ứng dụng mạng mở rộng thường trải rộng trên nhiều máy chủ, yêu cầu việc xử lý các luồng dữ liệu được phân phối trên các quy trình. Ngoài ra, việc kết hợp các luồng khác nhau thành một luồng đầu ra tùy chỉnh duy nhất đã trở thành chữ ký của các dịch vụ đám mây và nhiều API khác.

Bất kỳ ai đã truy cập trang web chuyên về hiển thị các đối tượng phương tiện lớn (chẳng hạn như video) cũng có thể gặp phải độ trễ. Trong một thế giới lý tưởng, người ta sẽ đưa ra yêu cầu đối với một máy chủ lolcats.mpg và sẽ nhận được tệp này ngay lập tức.

Tuy nhiên, việc di chuyển một lượng lớn dữ liệu (byte) cần nhiều thời gian. Khi kích thước của các tệp có thể truy cập qua Internet bắt đầu tăng kích thước, từ một số kilobyte

lên nhiều gigabyte, thời gian tải xuống tăng song song, từ vài phút đến vài giờ. Ban đầu, để xem video, trình xem phương tiện của một người (thường là trình duyệt) yêu cầu một bản sao hoàn chỉnh trước khi bắt đầu phát lại. Đối với người dùng, điều này có nghĩa là mất nhiều thời gian chờ đợi để nhìn vào thanh tiến trình hoặc các biểu tượng quay. Phải mất rất ít tầm nhìn để thấy rằng giải pháp này là không thể thực hiện được.

Vì nhu cầu rất cao đối với các đối tượng truyền thông lớn, những người quan tâm đến việc phân phối các đối tượng này đã khuyến khích sự phát triển của các cơ chế phân phối tốt hơn. Khi Internet trở thành mạng phân phối thông tin phi thực tế của thế giới, cho cả các đối tượng truyền thông cố định và truyền dữ liệu trực tiếp, các giải pháp tốt hơn đã xuất hiện. Đặc biệt, phát trực tuyến phương tiện truyền thông đã trở thành tiêu chuẩn. Người đọc chắc chắn sẽ quen thuộc với phương tiện truyền phát trực tuyến, và đặc biệt là ý tưởng về đệm một dòng suối. Các phần nhỏ (ví dụ: luồng video được "phát" khi chúng được nhận) đến quá sớm sẽ bị lưu vào bộ đệm (giúp giải quyết "trục trặc" và độ trễ), cho đến khi toàn bộ tệp đã đến:



Ở đây, một tệp phát trực tuyến chỉ đơn giản là một luồng dữ liệu được phân chia thành các lát, trong đó mỗi lát có thể được xem một cách độc lập bất kể tính khả dụng của những phần khác. Người ta có thể ghi vào một luồng dữ liệu, hoặc nghe trên một luồng dữ liệu, tự do phân bổ động các byte, bỏ qua các byte, để định tuyến lại các byte. Các luồng dữ liệu có thể được chia nhỏ, nhiều quy trình có thể chia sẻ xử lý phân vùng, các luồng có thể được chuyển đổi và chèn lại, và các luồng dữ liệu có thể được phát ra một cách chính xác và được quản lý một cách sáng tạo.

Nhắc lại cuộc thảo luận của chúng tôi về phần mềm hiện đại và Quy tắc Môđun, chúng ta có thể thấy cách các luồng tạo điều kiện thuận lợi cho việc tạo ra các quy trình không chia sẻ độc lập thực hiện tốt một nhiệm vụ và kết hợp với nhau có thể tạo ra một kiến trúc có thể dự đoán được mà độ phức tạp của nó không cản trở việc đánh giá chính xác hình dạng của nó. Nếu các giao diện với dữ liệu là không đối nghịch, thì bản đồ dữ liệu có thể được mô hình hóa chính xác độc lập với các cân nhắc về khối lượng dữ liệu hoặc định tuyến.

Quản lý I / O trong Node liên quan đến quản lý các sự kiện dữ liệu liên kết với các luồng dữ liệu. Một nút Suối đối tượng là một ví dụ của EventEmitter. Giao diện trừu tượng này được thực hiện trong các mô-đun và đối tượng Node khác nhau. Hãy bắt đầu bằng cách hiểu Node's Suối mô-đun, sau đó chuyển sang thảo luận về cách xử lý I / O mạng trong Node thông qua các Suối triển khai; đặc biệt là mô-đun HTTP.

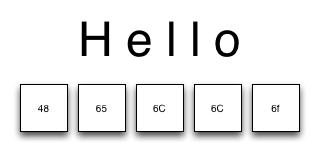
## 3.1. Khám phá luồng

Dựa theo Bjarne Stoustrup trong cuốn sách của anh ấy Ngôn ngữ lập trình C ++, Ấn bản thứ ba:

Thiết kế và triển khai một cơ sở đầu vào / đầu ra chung cho một ngôn ngữ lập trình nổi tiếng là khó ... Cơ sở I / O phải dễ sử dụng, thuận tiện và an toàn; hiệu quả và linh hoạt; và trên hết, hoàn thành.

Không có gì ngạc nhiên khi một nhóm thiết kế, tập trung vào việc cung cấp I / O hiệu quả và dễ dàng, đã cung cấp một cơ sở như vậy thông qua Node. Thông qua một giao diện đối xứng và đơn giản, xử lý bộ đệm dữ liệu và phát trực tuyến các sự kiện để người triển khai không phải làm như vậy, Node's Suối mô-đun là cách ưa thích để quản lý các luồng dữ liệu không đồng bộ cho cả mô-đun nội bộ và hy vọng là đối với các mô-đun mà nhà phát triển sẽ tạo.

Một luồng trong Node chỉ đơn giản là một chuỗi các byte. Tại bất kỳ thời điểm nào, một luồng chứa bộ đệm byte và bộ đệm này có độ dài bằng 0 hoặc lớn hơn:



Bởi vì mỗi ký tự trong một luồng được xác định rõ ràng và vì mọi loại dữ liệu kỹ thuật số đều có thể được thể hiện bằng byte, bất kỳ phần nào của luồng đều có thể được chuyển hướng hoặc "đường ống",

đến bất kỳ luồng nào khác, các phần khác nhau của luồng có thể được gửi đến các trình xử lý khác nhau, v.v. Bằng cách này, các giao diện đầu vào và đầu ra của dòng đều linh hoạt và có thể dự đoán được và có thể dễ dàng ghép nối.

Các dòng kỹ thuật số được mô tả tốt bằng cách sử dụng sự tương tự của chất lỏng, nơi các byte riêng lẻ (giọt nước) được đẩy qua một đường ống. Trong Node, các luồng là các đối tượng đại diện cho các luồng dữ liệu có thể được ghi vào và đọc từ một cách không đồng bộ. Triết lý của Node là một luồng không chặn, I / O được xử lý qua các luồng, và do đó, thiết kế của Suối API tự nhiên sao chép triết lý chung này. Trên thực tế, không có cách nào khác để tương tác với các luồng ngoại trừ theo cách không đồng bộ, theo sự kiện — bạn bị ngăn chặn I / O theo thiết kế.

Năm lớp cơ sở riêng biệt được hiển thị thông qua phần tóm tắt Suối giao diện: Đọc được,

Có thể ghi, in hai mặt, chuyển đổi, và PassThrough. Mỗi lớp cơ sở kế thừa từ

EventEmitter, mà chúng ta biết đến như một giao diện mà các bộ xử lý và bộ phát sự kiện có thể bị ràng buộc.

Như chúng ta sẽ tìm hiểu và ở đây sẽ nhấn mạnh, Suối giao diện là một trừu tượng giao diện. Một giao diện trừu tượng có chức năng như một loại bản thiết kế hoặc định nghĩa, mô tả các tính năng phải được tích hợp trong mỗi phiên bản được xây dựng của Suối

vật. Ví dụ, một Có thể đọc được triển khai luồng là bắt buộc để triển khai một phương thức đọc công khai ủy quyền cho \_ nội bộ của giao diện đọc phương pháp.

Nói chung, tất cả triển khai luồng phải tuân theo các nguyên tắc sau:

Miễn là dữ liệu tồn tại để gửi, viết vào một luồng cho đến khi hoạt động đó trở lại sai, tại thời điểm đó, việc triển khai sẽ chờ đợi cống sự kiện, cho biết rằng dữ liệu luồng được lưu trong bộ đệm đã làm trống

Tiếp tục gọi đọc cho đến khi nhận được giá trị null, tại thời điểm đó, hãy đợi một đọc được sự kiện trước khi tiếp tục đọc

Một số Node I / Omodules được triển khai dưới dạng luồng. Ổ cắm mạng, trình đọc tệp và trình ghi, stdin và stdout, zlib, và như thế. Tương tự, khi triển khai một nguồn dữ liệu có thể đọc được hoặc trình đọc dữ liệu, người ta nên triển khai giao diện đó dưới dạng Suối giao diện.

Điều quan trọng cần lưu ý là kể từ Nút 0.10.0, Suối giao diện thay đổi theo một số cách cơ bản. Nhóm Node đã cố gắng hết sức để triển khai các giao diện tương thích ngược, sao cho (hầu hết) các chương trình cũ hơn sẽ tiếp tục hoạt động mà không cần sửa đổi. Trong chương này, chúng ta sẽ không dành thời gian thảo luận về các tính năng cụ thể của API cũ hơn này, tập trung vào thiết kế hiện tại (và tương lai). Người đọc được khuyến khích tham khảo tài liệu trực tuyến của Node để biết thông tin về việc di chuyển các chương trình cũ hơn.

## 3.2. Triển khai các luồng có thể đọc được

Các luồng sản xuất dữ liệu mà một quy trình khác có thể quan tâm thường được triển khai bằng cách sử dụng Có thể đọc được suối. A Có thể đọc được luồng tiết kiệm cho người triển khai tất cả công việc quản lý hàng đợi đọc, xử lý việc phát ra các sự kiện dữ liệu, v.v.

Để tạo ra một Có thể đọc được suối:

var stream = request ('stream'); var var stream = require('stream'); var readable = new stream.Readable({

encoding : "utf8", highWaterMark : 16000, objectMode: true });

Như đã nói ở trên, Có thể đọc được được hiển thị như một lớp cơ sở, có thể được khởi tạo thông qua ba tùy chọn:

mã hóa: Giải mã bộ đệm thành mã hóa được chỉ định, mặc định là UTF-8.

highWaterMark: Số byte cần giữ trong bộ đệm nội bộ trước khi ngừng đọc từ nguồn dữ liệu. Mặc định là 16 KB.

objectMode: Yêu cầu luồng hoạt động như một luồng đối tượng thay vì một luồng byte, chẳng hạn như luồng các đối tượng JSON thay vì byte trong một tệp. Mặc định sai.

Trong ví dụ sau, chúng tôi tạo một mô hình Cho ăn đối tượng có các thể hiện sẽ kế thừa Có thể đọc được giao diện luồng. Việc triển khai của chúng tôi chỉ cần triển khai phần tóm tắt

\_đọc phương pháp của Đọc được, cái nào sẽ đẩy dữ liệu cho người tiêu dùng cho đến khi không còn gì để đẩy nữa, tại thời điểm đó, nó kích hoạt Có thể đọc được luồng để phát ra một sự kiện "kết thúc" bằng cách đẩy một vô giá trị giá trị:

var Feed = function(channel) { var readable = new stream.Readable({

encoding : "utf8"

}); var news = [ "Big Win!",

"Stocks Down!",

"Actor Sad!"

];

readable.\_read = function() { if(news.length) {

return readable.push(news.shift() + "\n");

}

readable.push(null);

};

return readable;

}

Bây giờ chúng tôi đã triển khai, người tiêu dùng có thể muốn tạo luồng và lắng nghe các sự kiện phát trực tiếp. Hai sự kiện chính là đọc được và kết thúc.

Sự kiện có thể đọc được phát ra miễn là dữ liệu đang được đẩy vào luồng. Nó cảnh báo người tiêu dùng kiểm tra dữ liệu mới thông qua đọc phương pháp của Có thể đọc được.

Lưu ý lại cách Có thể đọc được triển khai phải cung cấp \_ riêng tư đọc phương pháp, dịch vụ công cộng đọc phương pháp tiếp xúc với API người tiêu dùng.

Sự kiện kết thúc sẽ được phát ra bất cứ khi nào vô giá trị giá trị được chuyển cho đẩy phương pháp của chúng tôi Có thể đọc được thực hiện.

Ở đây, chúng tôi thấy một người tiêu dùng sử dụng các phương pháp này để hiển thị dữ liệu luồng mới, cung cấp thông báo khi luồng ngừng gửi dữ liệu:

var feed = new Feed(); feed.on("readable", function() { var data = feed.read();

data && process.stdout.write(data);

});

feed.on("end", function() { console.log("No more news"); });

Tương tự, chúng ta có thể triển khai một luồng đối tượng thông qua việc sử dụng objectMode Lựa chọn:

var readable = new stream.Readable({ objectMode : true

}); var prices = [

{ price : 1 },

{ price : 2 }

];

... readable.push(prices.shift());

// > { prices : 1 }

// > { prices : 2 }

Ở đây chúng ta thấy rằng mỗi sự kiện đọc đang nhận một đối tượng, chứ không phải một bộ đệm hoặc chuỗi.

cuối cùng đọc phương pháp của một Có thể đọc được luồng có thể được truyền một đối số cho biết số byte được đọc từ bộ đệm bên trong của luồng. Ví dụ: nếu muốn một tệp được đọc từng byte một, người ta có thể triển khai người dùng sử dụng một quy trình tương tự như:

readable.push("Sequence of bytes");

... feed.on("readable", function() { var character;

while(character = feed.read(1)) { console.log(character);

};

});

// > S

// > e

// > q // > ...

Ở đây cần phải rõ ràng rằng Có thể đọc được bộ đệm của luồng được lấp đầy với một số byte cùng một lúc, nhưng được đọc từ một cách riêng lẻ.

## 3.3. Đẩy và kéo

Chúng tôi đã thấy cách một Có thể đọc được thực hiện sẽ sử dụng đẩy để điền vào bộ đệm luồng để đọc. Khi thiết kế các triển khai này, điều quan trọng là phải xem xét cách quản lý khối lượng ở hai đầu luồng. Việc đẩy nhiều dữ liệu vào một luồng hơn mức có thể đọc có thể dẫn đến các biến chứng xung quanh việc vượt quá không gian khả dụng (bộ nhớ). Ở phía người tiêu dùng, điều quan trọng là phải duy trì nhận thức về các sự kiện chấm dứt và cách đối phó với các lần tạm dừng trong luồng dữ liệu.

Người ta có thể so sánh hành vi của các luồng dữ liệu chạy qua mạng với hành vi của nước chảy qua vòi.

Giống như với nước qua vòi, nếu một khối lượng dữ liệu lớn hơn đang được đẩy vào luồng đọc hơn là có thể được thoát ra khỏi luồng một cách hiệu quả ở đầu cuối của người tiêu dùng thông qua đọc, rất nhiều áp lực ngược hình thành, khiến dữ liệu tồn đọng bắt đầu tích tụ trong bộ đệm của đối tượng luồng. Bởi vì chúng tôi đang giải quyết các giới hạn toán học nghiêm ngặt, đọc đơn giản là không thể bị buộc phải giải phóng áp lực này bằng cách đọc nhanh hơn — có thể có giới hạn về dung lượng bộ nhớ khả dụng,

hoặc hạn chế khác. Do đó, việc sử dụng bộ nhớ có thể tăng cao một cách nguy hiểm, bộ đệm có thể bị tràn, v.v.

Do đó, việc triển khai luồng cần nhận thức và phản hồi lại phản hồi từ đẩy hoạt động. Nếu hoạt động trở lại sai điều này chỉ ra rằng việc triển khai sẽ ngừng đọc từ nguồn của nó (và ngừng đẩy) cho đến \_ đọc yêu cầu được thực hiện.

Kết hợp với những điều trên, nếu không có thêm dữ liệu để đẩy nhưng nhiều hơn được mong đợi trong tương lai việc thực hiện nên đẩy một chuỗi rỗng (""), không thêm dữ liệu vào hàng đợi nhưng đảm bảo một tương lai đọc được biến cố.

Trong khi cách xử lý phổ biến nhất của bộ đệm luồng là đẩy đối với nó (xếp hàng dữ liệu trong một dòng), có những trường hợp người ta có thể muốn đặt dữ liệu ở phía trước bộ đệm (nhảy dòng). Node cung cấp một không tạm bợ hoạt động cho những trường hợp này, hành vi nào giống với push, ngoài sự khác biệt nói trên về vị trí đệm.

## 3.4. Luồng ghi

A Ghi luồng chịu trách nhiệm chấp nhận một số giá trị (luồng byte, một chuỗi) và ghi dữ liệu đó vào một đích. Truyền dữ liệu vào vùng chứa tệp là một trường hợp sử dụng phổ biến.

To create a Writable stream:

var stream = require('stream'); var readable = new stream.Writable({ highWaterMark : 16000, decodeStrings: true });

Các Ghi hàm tạo luồng có thể được khởi tạo với hai tùy chọn:

highWaterMark: Số byte tối đa mà bộ đệm của luồng sẽ chấp nhận trước khi trả về false khi ghi. Mặc định là 16 KB

decodeStrings: Có chuyển đổi chuỗi thành bộ đệm trước khi viết hay không. Mặc định là đúng.

Như với Có thể đọc được suối, tùy chỉnh Ghi triển khai luồng phải thực hiện một \_ viết trình xử lý, sẽ được chuyển các đối số được gửi đến viết phương pháp của các thể hiện.

Người ta nên nghĩ về một Ghi luồng dưới dạng mục tiêu dữ liệu, chẳng hạn như cho tệp bạn đang tải lên. Về mặt khái niệm, điều này không khác với việc thực hiện push trong một Có thể đọc được luồng, nơi người ta đẩy dữ liệu cho đến khi nguồn dữ liệu cạn kiệt, chuyển vô giá trị để chấm dứt việc đọc. Ví dụ: ở đây chúng tôi viết 100 byte thành stdout:

var stream = require('stream'); var writable = new stream.Writable({ decodeStrings: false

});

writable.\_write = function(chunk, encoding, callback) { console.log(chunk); callback();

}

var w = writable.write(new Buffer(100)); writable.end(); console.log(w); // Will be `true`

Có hai điều chính cần lưu ý ở đây.

Đầu tiên, \_ của chúng tôi viết thực hiện cháy gọi lại hàm ngay sau khi viết, một lệnh gọi lại luôn hiện diện, bất kể trường hợp viết

phương thức được chuyển trực tiếp một cuộc gọi lại. Lời gọi này rất quan trọng để chỉ ra trạng thái của nỗ lực ghi, cho dù là thất bại (lỗi) hay thành công.

Thứ hai, cuộc gọi đến viết trả lại thật. Điều này chỉ ra rằng bộ đệm bên trong của

Ghi triển khai đã được làm trống sau khi thực hiện ghi được yêu cầu. Điều gì sẽ xảy ra nếu chúng tôi gửi một lượng rất lớn dữ liệu, đủ để vượt quá kích thước mặc định của bộ đệm bên trong? Sửa đổi ví dụ trên, phần sau sẽ trả về sai:

var w = writable.write(new Buffer(16384)); console.log(w); // Will be 'false'

Lý do mà viết này trả về false là nó đã đạt đến highWaterMark

tùy chọn — giá trị mặc định là 16 KB ( 16 \* 1024). Nếu chúng tôi thay đổi giá trị này thành 16383, viết một lần nữa sẽ trả về true (hoặc người ta có thể chỉ cần tăng giá trị của nó lên).

Làm gì khi viết trả về sai? Người ta chắc chắn không nên tiếp tục gửi dữ liệu! Quay trở lại phép ẩn dụ của chúng ta về nước trong vòi: khi dòng chảy đầy, người ta nên đợi nó rút hết trước khi gửi thêm dữ liệu. Điểm giao Suối

việc thực hiện sẽ phát ra một cống sự kiện bất cứ khi nào an toàn để viết lại. Khi nào viết trả lại sai lắng nghe cống sự kiện trước khi gửi thêm dữ liệu.

Tổng hợp những gì chúng ta đã học được, chúng ta hãy tạo Ghi suối với một highWaterMark giá trị 10 byte. Chúng tôi sẽ gửi một bộ đệm chứa hơn 10 byte (bao gồm A ký tự) vào luồng này, kích hoạt cống sự kiện, tại thời điểm đó chúng tôi viết một Z tính cách. Từ ví dụ này, rõ ràng là Node's

Suối thực hiện là quản lý tràn bộ nhớ trọng tải ban đầu của chúng tôi, cảnh báo phương pháp ghi ban đầu của lần tràn này, thực hiện việc cạn kiệt có kiểm soát của bộ đệm nội bộ và thông báo cho chúng tôi khi nào là an toàn viết lần nữa:

var stream = require('stream'); var writable = new stream.Writable({ highWaterMark: 10

});

writable.\_write = function(chunk, encoding, callback) { process.stdout.write(chunk); callback();

}

writable.on("drain", function() { writable.write("Z\n");

});

var buf = new Buffer(20, "utf8"); buf.fill("A"); console.log(writable.write(buf.toString())); // false

Kết quả phải là một chuỗi 20 A ký tự, theo sau là sai, sau đó là ký tự Z.

Dữ liệu linh hoạt trong một Có thể đọc được luồng có thể được chuyển hướng dễ dàng đến một Ghi suối. Ví dụ: mã sau sẽ lấy bất kỳ dữ liệu nào được gửi bởi một thiết bị đầu cuối ( stdin là một Có thể đọc được stream) và chuyển nó đến đích Ghi suối, stdout:

process.stdin.pipe (process.stdout);

Bất cứ khi nào Ghi luồng được chuyển đến một Có thể đọc được dòng suối ống phương pháp, một ống sự kiện sẽ khai hỏa. Tương tự, khi a Ghi luồng bị xóa như một điểm đến cho một Có thể đọc được suối, mở ống cháy sự kiện.

Để loại bỏ một ống, sử dụng như sau:

unpipe (luồng đích)

## 3.5. Luồng song công

Luồng song công có thể đọc và ghi được. Ví dụ: một máy chủ TCP được tạo trong Node để lộ một ổ cắm có thể được đọc và ghi vào:

Node exposes a socket that can be both read from and written to:

var stream = require("stream"); var net = require("net"); net .createServer(function(socket) {

socket.write("Go ahead and type something!"); socket.on("readable", function() { process.stdout.write(this.read())

});

})

.listen(8080);

Khi được thực thi, mã này sẽ tạo ra một máy chủ TCP có thể được kết nối qua Telnet:

telnet 127.0.0.1 8080

Khi kết nối, thiết bị đầu cuối kết nối sẽ in ra Hãy tiếp tục và gõ một cái gì đó! - viết vào ổ cắm. Bất kỳ văn bản nào được nhập vào thiết bị đầu cuối kết nối sẽ được lặp lại tới stdout của thiết bị đầu cuối chạy máy chủ TCP ( đọc

hiểu từ ổ cắm). Việc triển khai giao thức truyền thông hai chiều (song công) này thể hiện rõ ràng cách các quy trình độc lập có thể hình thành các nút của một ứng dụng phức tạp và đáp ứng, cho dù giao tiếp qua mạng hay trong phạm vi của một quy trình đơn lẻ.

Các tùy chọn được gửi khi xây dựng Song công ví dụ hợp nhất những người được gửi đến Có thể đọc được

và Ghi luồng, không có tham số bổ sung. Thật vậy, loại luồng này chỉ đơn giản là đảm nhận cả hai vai trò và các quy tắc để tương tác với nó tuân theo các quy tắc cho chế độ tương tác đang được sử dụng.

Vì một luồng Song công đảm nhận cả vai trò đọc và ghi, nên bất kỳ triển khai nào cũng được yêu cầu thực hiện cả hai \_ viết và \_ đọc phương pháp, một lần nữa tuân theo các chi tiết triển khai tiêu chuẩn được cung cấp cho loại luồng liên quan.

## 3.6. Chuyển đổi luồng

Đôi khi, dữ liệu luồng cần được xử lý, thường là trong trường hợp người ta đang viết một số loại giao thức nhị phân hoặc chuyển đổi dữ liệu "nhanh chóng" khác. A Biến đổi luồng được thiết kế cho mục đích này, hoạt động như một Song công dòng chảy nằm giữa một Có thể đọc được dòng và một Ghi suối.

A Biến đổi luồng được khởi tạo bằng cách sử dụng các tùy chọn tương tự được sử dụng để khởi tạo một

Song công suối. Ở đâu Biến đổi khác với bình thường Song công dòng yêu cầu của nó rằng việc triển khai tùy chỉnh chỉ cung cấp một \_ biến đổi , ngoại trừ \_ viết và \_ đọc yêu cầu về phương pháp.

Các \_ biến đổi phương thức sẽ nhận ba đối số, đầu tiên là bộ đệm đã gửi, một đối số mã hóa tùy chọn và cuối cùng là một lệnh gọi lại \_ biến đổi dự kiến sẽ gọi khi quá trình chuyển đổi hoàn tất:

\_transform = function(buffer, encoding, cb) { var transformation = "..."; this.push(transformation) cb(); }

Hãy tưởng tượng một chương trình muốn chuyển đổi ASCII (Mã tiêu chuẩn Hoa Kỳ để trao đổi thông tin) mã thành các ký tự ASCII, nhận đầu vào từ stdin. Chúng tôi chỉ cần chuyển đầu vào của chúng tôi đến một Biến đổi luồng, sau đó chuyển đầu ra của nó đến stdout:

var stream = require('stream'); var converter = new stream.Transform(); converter.\_transform = function(num, encoding, cb) { this.push(String.fromCharCode(new Number(num)) + "\n") cb();

} process.stdin.pipe(converter).pipe(process.stdout);

Tương tác với chương trình này có thể tạo ra kết quả giống như sau:

A

B

Ā

ā

Ví dụ về luồng biến đổi sẽ được trình bày trong ví dụ kết thúc chương này.

## 3.7. Sử dụng luồng PassThrough

Loại luồng này là một triển khai nhỏ của luồng Chuyển đổi, chỉ đơn giản là chuyển các byte đầu vào đã nhận qua luồng đầu ra. Điều này rất hữu ích nếu một người không yêu cầu bất kỳ sự chuyển đổi nào đối với dữ liệu đầu vào và chỉ đơn giản là muốn dễ dàng ghi Có thể đọc được truyền đến một Ghi suối.

PassThrough các luồng có lợi ích tương tự như các chức năng ẩn danh của JavaScript, giúp dễ dàng khẳng định chức năng tối thiểu mà không cần quá nhiều phiền phức. Ví dụ: không cần thiết phải triển khai một lớp cơ sở trừu tượng, như cách làm với \_ đọc phương pháp của một Có thể đọc được suối. Hãy xem xét việc sử dụng một PassThrough phát trực tiếp như một gián điệp sự kiện:

var fs = require('fs');

var stream = new require('stream').PassThrough(); spy.on('end', function() {

console.log("All data has been sent");

});

fs.createReadStream("./passthrough.js").pipe(spy).pipe(process.std out);

## 3.8. Tạo máy chủ HTTP

HTTP là một giao thức truyền dữ liệu không trạng thái được xây dựng dựa trên mô hình yêu cầu / phản hồi: máy khách thực hiện yêu cầu đến máy chủ, sau đó sẽ trả lại phản hồi. Tạo điều kiện thuận lợi cho loại giao tiếp mạng nhanh chóng này là loại I / O Node được thiết kế để vượt trội hơn, không ngạc nhiên khi nó được xác định chủ yếu là một bộ công cụ để tạo máy chủ — mặc dù nó chắc chắn có thể được sử dụng để làm nhiều việc hơn thế nữa. Trong suốt cuốn sách này, chúng tôi sẽ tạo ra nhiều cách triển khai của máy chủ HTTP, cũng như các máy chủ giao thức khác và sẽ thảo luận về các phương pháp hay nhất chuyên sâu hơn, được ngữ cảnh hóa trong các trường hợp kinh doanh cụ thể. Người ta mong đợi rằng bạn đã có một số kinh nghiệm làm điều tương tự. Vì cả hai lý do này, chúng tôi sẽ nhanh chóng chuyển qua phần tổng quan chung sang một số cách sử dụng chuyên biệt hơn.

Đơn giản nhất, một máy chủ HTTP phản hồi các nỗ lực kết nối và quản lý dữ liệu khi nó đến và khi nó được gửi đi. Máy chủ Node thường được tạo bằng cách sử dụng createServer phương thức của mô-đun http var http = require('http');

var server = http.createServer(function(request, response) { console.log("Got Request Headers:"); console.log(request.headers); response.writeHead(200, {

'Content-Type': 'text/plain'

});

response.write("PONG"); response.end(); }); server.listen(8080);

Đối tượng được trả lại bởi http.createServer là một ví dụ của http.Server, mở rộng EventEmitter, phát các sự kiện mạng khi chúng xảy ra, chẳng hạn như kết nối hoặc yêu cầu của máy khách. Đoạn mã trên là một cách phổ biến để viết máy chủ Node. Tuy nhiên, điều đáng chỉ ra là trực tiếp khởi tạo http.Server lớp đôi khi là một cách hữu ích để phân biệt các tương tác máy chủ / máy khách riêng biệt. Chúng tôi sẽ sử dụng định dạng đó cho các ví dụ sau.

Ở var http = require('http'); var server = new http.Server(); server.on("connection", function(socket) { console.log("Client arrived: " + new Date()); socket.on("end", function() {

console.log("Client left: " + new Date());

});

}) server.listen(8080);

Khi xây dựng các hệ thống đa người dùng, đặc biệt là các hệ thống đa người dùng đã được xác thực, điểm này trong giao dịch máy chủ-máy khách là một nơi tuyệt vời để xác thực máy khách và mã theo dõi, bao gồm cài đặt hoặc đọc cookie và các biến phiên khác hoặc truyền phát sự kiện khách hàng đến người khác khách hàng làm việc cùng nhau trong một ứng dụng thời gian thực đồng thời.

Bằng cách thêm trình xử lý cho các yêu cầu, chúng tôi đi đến mẫu yêu cầu / phản hồi phổ biến hơn, được xử lý như một Có thể đọc được suối. Khi khách hàng ĐĂNG một số dữ liệu, chúng tôi có thể bắt dữ liệu đó như sau:

server.on("request", function(request, response) { request.setEncoding("utf8"); request.on("readable", function() { console.log(request.read())

});

});

Thử gửi một số dữ liệu đến máy chủ này bằng Xoăn:

curl http://localhost:8080 -d "Here is some data"

Bằng cách sử dụng các sự kiện kết nối, chúng ta có thể tách mã xử lý kết nối của mình một cách độc đáo, nhóm nó thành các miền chức năng được xác định rõ ràng được mô tả chính xác là thực thi để đáp ứng các sự kiện cụ thể.

Ví dụ, chúng tôi có thể đặt bộ hẹn giờ trên các kết nối máy chủ. Ở đây chúng tôi chấm dứt các kết nối máy khách không gửi được dữ liệu mới trong khoảng thời gian khoảng hai giây:

server.setTimeout(2000, function(socket) { socket.write("Too Slow!", "utf8"); socket.end(); });

Nếu một người chỉ muốn đặt số mili giây không hoạt động trước khi một ổ cắm được cho là đã hết thời gian, chỉ cần sử dụng server.timeout = (Số nguyên) num\_milliseconds.

Để tắt thời gian chờ của ổ cắm, hãy chuyển một giá trị 0 (không).

Bây giờ chúng ta hãy xem cách mô-đun HTTP của Node có thể được sử dụng để tham gia vào các tương tác mạng thú vị hơn.

## 3.9. Thực hiện các yêu cầu HTTP

Ứng dụng mạng thường cần thiết để thực hiện các cuộc gọi HTTP bên ngoài. Máy chủ HTTP cũng thường được gọi để thực hiện các dịch vụHTTP cho các máy khách đưa ra yêu cầu. Node cung cấp một giao diện dễ dàng để thực hiện các cuộc gọi HTTP bên ngoài.

Ví dụ: mã sau sẽ lấy trang đầu của google.com:

var http = require('http'); http.request({ host: 'www.google.com', method: 'GET', path: "/"

}, function(response) { response.setEncoding("utf8"); response.on("readable", function() { console.log(response.read())

});

}).end();

Như chúng ta có thể thấy, chúng tôi đang làm việc với luồng có thể đọc được, luồng này có thể được ghi vào một tệp.

Một mô-đun Node phổ biến để quản lý các yêu cầu HTTP là Mikeal Rogers ' yêu cầu: <https://github.com/mikeal/request> Vì http.get("http://www.google.com/", function(response) { console.log("Status: " + response.statusCode);

}).on('error', function(err) { console.log("Error: " + err.message); });

Bây giờ chúng ta hãy xem xét một số triển khai nâng cao hơn của máy chủ HTTP, nơi chúng tôi thực hiện các dịch vụ mạng chung cho các máy khách.

## 3.10. Proxying và đào hầm

Đôi khi, rất hữu ích khi cung cấp một phương tiện để một máy chủ hoạt động như một proxy, hoặc nhà môi giới, cho các máy chủ khác. Điều này sẽ cho phép một máy chủ phân phối tải cho các máy chủ khác, chẳng hạn. Một cách sử dụng khác là cung cấp quyền truy cập vào một máy chủ bảo mật cho những người dùng không thể kết nối trực tiếp với máy chủ đó. Việc một máy chủ trả lời cho nhiều hơn một URL cũng thường xảy ra — bằng cách sử dụng proxy, một máy chủ có thể chuyển tiếp yêu cầu đến đúng người nhận.

Bởi vì Node có giao diện luồng nhất quán trên khắp các giao diện mạng của nó, chúng ta có thể xây dựng một proxy HTTP đơn giản chỉ trong một vài dòng mã. Ví dụ: chương trình sau sẽ thiết lập một máy chủ HTTP trên cổng 8080, máy chủ này sẽ phản hồi bất kỳ yêu cầu nào bằng cách tìm nạp trang đầu của Google và chuyển nó trở lại máy khách:

var http = require('http'); var server = new http.Server();

server.on("request", function(request, socket) { http.request({ host: 'www.google.com', method: 'GET', path: "/", port: 80

}, function(response) { response.pipe(socket);

}).end();

}); server.listen(8080);

Sau khi máy chủ này nhận được ổ cắm máy khách, có thể tự do đẩy nội dung từ bất kỳ luồng nào có thể đọc được trở lại máy khách và đây là kết quả của việc GET of www.google.com được phát trực tiếp. Ví dụ, người ta có thể dễ dàng thấy cách một máy chủ nội dung bên ngoài quản lý lớp bộ nhớ đệm cho ứng dụng của bạn có thể trở thành một điểm cuối proxy.

Sử dụng các ý tưởng tương tự, chúng ta có thể tạo một dịch vụ đường hầm, sử dụng hỗ trợ CONNECT gốc của Node. Đường hầm liên quan đến việc sử dụng máy chủ proxy làm trung gian để giao tiếp với máy chủ từ xa thay mặt cho khách hàng. Sau khi máy chủ proxy của chúng tôi kết nối với máy chủ từ xa, nó có thể chuyển các thông báo qua lại giữa máy chủ đó và máy khách. Điều này có lợi khi kết nối trực tiếp giữa máy khách và máy chủ từ xa là không thể hoặc không được mong muốn.

Đầu tiên, chúng tôi sẽ thiết lập một máy chủ proxy phản hồi các yêu cầu HTTP CONNECT, sau đó thực hiện một yêu cầu KẾT NỐI tới máy chủ đó. Proxy nhận được của khách hàng của chúng tôi Yêu cầu đối tượng, chính ổ cắm của khách hàng và cái đầu ( gói đầu tiên) của luồng đào đường hầm. Sau đó, chúng tôi mở ổ cắm mạng từ xa được yêu cầu. Tất cả những gì còn lại cần làm là tạo đường hầm, chúng tôi thực hiện bằng cách chuyển dữ liệu từ xa đến máy khách và dữ liệu máy khách tới kết nối từ xa:

var http = require('http'); var net = require('net'); var url = require('url'); var proxy = new http.Server();

proxy.on('connect', function(request, clientSocket, head) { var reqData = url.parse('http://' + request.url); var remoteSocket = net.connect(reqData.port, reqData.hostname, function() {

clientSocket.write('HTTP/1.1 200 \r\n\r\n');

remoteSocket.write(head); remoteSocket.pipe(clientSocket); clientSocket.pipe(remoteSocket);

});

}).listen(8080);

var request = http.request({

port: 8080, hostname: 'localhost', method: 'CONNECT', path: 'www.google.com:80'

}); request.end();

request.on('connect', function(res, socket, head) {

socket.setEncoding("utf8");

socket.write('GET / HTTP/1.1\r\nHost: www.google.com:80\r\ nConnection: close\r\n\r\n'); socket.on('readable', function() { console.log(socket.read());

});

socket.on('end', function() { proxy.close();

});

});

## 3.11. HTTPS, TLS (SSL) và bảo mật máy chủ của bạn

Tính bảo mật của các ứng dụng web đã trở thành một chủ đề thảo luận quan trọng trong những năm gần đây. Các ứng dụng truyền thống thường được hưởng lợi từ các mô hình bảo mật đã được kiểm tra tốt và trưởng thành được thiết kế vào các máy chủ chính và các ngăn xếp ứng dụng làm nền tảng cho các triển khai lớn. Vì lý do này hay lý do khác, các ứng dụng web được phép tham gia vào thế giới thử nghiệm của logic kinh doanh phía máy khách và các dịch vụ web mở được che chắn bởi một bức màn lộng lẫy.

Vì Node thường xuyên được triển khai như một máy chủ web, nên cộng đồng bắt buộc phải chấp nhận trách nhiệm bảo mật các máy chủ này. HTTPS là một giao thức truyền dẫn an toàn — về cơ bản HTTP được mã hóa được hình thành bằng cách xếp giao thức HTTP lên trên giao thức SSL / TLS.

## 3.12. Tạo chứng chỉ tự ký để phát triển

Để hỗ trợ kết nối SSL, máy chủ sẽ cần một chứng chỉ được ký hợp lệ. Trong khi phát triển, việc tạo một chứng chỉ tự ký sẽ dễ dàng hơn nhiều, điều này sẽ cho phép một người sử dụng mô-đun HTTPS của Node.

Đây là các bước cần thiết để tạo chứng chỉ cho sự phát triển. Lưu ý rằng quá trình này không tạo ra một chứng chỉ thực và không phải an toàn — nó chỉ đơn giản là cho phép chúng tôi phát triển trong môi trường HTTPS. Từ một thiết bị đầu cuối:

openssl genrsa -out server-key.pem 2048 openssl req -new -key server-key.pem -out server-csr.pem

openssl x509 -req -in server-csr.pem -signkey server-key.pem -out server-cert.pem

Các khóa này hiện có thể được sử dụng để phát triển máy chủ HTTPS. Nội dung của các tệp này chỉ cần được chuyển cùng với các tùy chọn đến máy chủ Node:

var https = request ('https'); var fs = request ('fs');

https.createServer ({ key: fs.readFileSync ('server-key.pem'), cert: fs.readFileSync ('server-cert.pem')

}, function (req, res) {

. . .

}). nghe (443)

Miễn phí bảo đảm thấp Chứng chỉ SSL có sẵn từ các chứng chỉ không phải http://www.startssl.com/ đối với những trường hợp tự ký là lý tưởng trong quá trình phát triển.

## 3.13. Cài đặt chứng chỉ SSL thực

Để chuyển một ứng dụng an toàn ra khỏi môi trường phát triển và sang môi trường tiếp xúc với Internet, bạn cần phải mua một chứng chỉ thực. Giá của các chứng chỉ này đã giảm qua từng năm và sẽ dễ dàng tìm được các nhà cung cấp chứng chỉ có giá hợp lý với mức độ bảo mật đủ cao. Một số nhà cung cấp thậm chí còn cung cấp chứng chỉ sử dụng cá nhân miễn phí.

Việc thiết lập chứng chỉ chuyên nghiệp chỉ cần thay đổi các tùy chọn HTTPS mà chúng tôi đã giới thiệu ở trên. Các nhà cung cấp khác nhau sẽ có các quy trình và tên tệp khác nhau. Thông thường, bạn sẽ cần tải xuống hoặc nhận từ nhà cung cấp dịch vụ riêng tư. Chìa khóa tệp, chứng chỉ miền đã ký của bạn. crt tệp và một gói mô tả

var options = {

key : fs.readFileSync("mysite.key"), cert : fs.readFileSync("mysite.com.crt"), ca : [ fs.readFileSync("gd\_bundle.crt") ] };

Điều quan trọng cần lưu ý là ca tham số phải được gửi dưới dạng mảng, ngay cả khi gói chứng chỉ đã được nối thành một tệp.

## 3.14. Đối tượng yêu cầu

Các thông báo phản hồi và yêu cầu HTTP tương tự nhau, bao gồm:

Một dòng trạng thái cho một yêu cầu sẽ giống như GET / index.html HTTP / 1.1 và phản hồi sẽ giống như HTTP / 1.1 200 OK

Không hoặc nhiều tiêu đề, mà trong một yêu cầu có thể bao gồm Bộ ký tự chấp nhận: UTF-8 hoặc Từ:

user@server.com , và trong các câu trả lời có thể giống

Content-Type: text / html và Content-Length: 1024

Nội dung thư, cho phản hồi có thể là một trang HTML và cho một yêu cầu ĐĂNG có thể là một số dữ liệu biểu mẫu

Chúng ta đã thấy cách các giao diện máy chủ HTTP trong Node dự kiến sẽ hiển thị một trình xử lý yêu cầu và cách trình xử lý này sẽ được chuyển qua một số dạng của đối tượng yêu cầu và phản hồi, mỗi giao diện thực hiện một luồng có thể đọc hoặc có thể ghi.

Chúng tôi sẽ trình bày sâu hơn về việc xử lý dữ liệu POST và dữ liệu Header ở phần sau của chương này. Trước khi thực hiện, hãy xem qua cách phân tích một số thông tin đơn giản hơn có trong một yêu cầu.

## 3.15. Mô-đun URL

Bất cứ khi nào một yêu cầu được gửi đến máy chủ HTTP, đối tượng yêu cầu sẽ chứa thuộc tính url, xác định tài nguyên được nhắm mục tiêu. Điều này có thể truy cập thông qua request.url.

Mô-đun URL của Node được sử dụng để phân tách một chuỗi URL điển hình thành các phần cấu thành của nó. Hãy xem xét hình sau:



Chúng tôi thấy cách url.parse phương thức phân tách các chuỗi, và ý nghĩa của mỗi phân đoạn phải rõ ràng. Nó cũng có thể rõ ràng rằng truy vấn sẽ hữu ích hơn nếu chính nó được phân tích cú pháp thành các cặp Khóa / Giá trị. Điều này được thực hiện bằng cách vượt qua thật như là đối số thứ hai của phân tích cú pháp phương thức này sẽ thay đổi giá trị trường truy vấn được cung cấp ở trên thành một bản đồ khóa / giá trị hữu ích hơn: truy vấn: {filter: 'sports', maxresults: '20'}

Điều này đặc biệt hữu ích khi phân tích cú pháp các yêu cầu GET.

Có một đối số cuối cùng cho url.parse liên quan đến sự khác biệt giữa hai URL này: <http://www.example.org//www.example.org> URL thứ hai ở đây là một ví dụ về tính năng thiết kế (tương đối không xác định) của giao thức HTTP: URL tương đối của giao thức (về mặt kỹ thuật, a tham chiếu đường dẫn mạng), trái ngược với phổ biến hơn tuyệt đối URL.

Để tìm hiểu thêm về cách các tham chiếu đường dẫn mạng được sử dụng để giải quyết giao thức tài nguyên trơn tru, hãy truy cập liên kết sau: http://tools.ietf.org/html/rfc3986#section-4.2

Vấn đề đang được thảo luận là: url.parse sẽ coi một chuỗi bắt đầu bằng dấu gạch chéo là chỉ đường dẫn, không phải máy chủ. Ví dụ, url.parse ("// www.example.org") sẽ đặt các giá trị sau trong tổ chức và con đường lĩnh vực: máy chủ: null, đường dẫn: '//www.example.org'

Điều chúng tôi thực sự muốn là ngược lại:

máy chủ: 'www.example.org', đường dẫn: null

Để giải quyết vấn đề này, hãy vượt qua thật như là đối số thứ ba để url.parse, cho biết phương thức mà dấu gạch chéo biểu thị một máy chủ, không phải một đường dẫn:

url.parse ("// www.example.org", null, true)

Cũng có trường hợp nhà phát triển muốn tạo URL, chẳng hạn như khi đưa ra yêu cầu qua http.request. Các phân đoạn của URL đã nói có thể nằm rải rác trên nhiều cấu trúc và biến dữ liệu khác nhau và sẽ cần được tập hợp lại. Người ta hoàn thành điều này bằng cách truyền một đối tượng giống như đối tượng được trả về từ url.parse phương pháp url. định dạng.

Đoạn mã sau sẽ tạo chuỗi URL http://www.example.org:

url.format({ protocol: 'http:', host: 'www.example.org' })

Tương tự, người ta cũng có thể sử dụng url.resolve phương pháp để tạo chuỗi URL trong trường hợp phổ biến là yêu cầu nối URL cơ sở và đường dẫn:

url.resolve ("http://example.org/a/b", "c / d") //

'http://example.org/a/c/d'

url.resolve ("http://example.org/a/b", "/ c / d") //

'http://example.org/c/d'

url.resolve ("http://example.org", "http://google.com") // 'http://google.com/'

Mô-đun Chuỗi truy vấn

Như chúng ta đã thấy với URL mô-đun, các chuỗi truy vấn thường cần được phân tích cú pháp thành một bản đồ của các cặp khóa / giá trị. Các Chuỗi truy vấn mô-đun sẽ phân tách một chuỗi truy vấn hiện có thành các phần của nó hoặc tập hợp một chuỗi truy vấn từ bản đồ các cặp khóa / giá trị.

Ví dụ, querystring.parse ("foo = bar & bingo = bango") sẽ trở lại:

{ foo: 'bar',

bingo: 'bango'}

Nếu chuỗi truy vấn của chúng tôi không được định dạng bằng dấu phân tách "&" và "=" thông thường, thì Chuỗi truy vấn mô-đun cung cấp phân tích cú pháp có thể tùy chỉnh. Đối số thứ hai cho Chuỗi truy vấn có thể là chuỗi phân tách tùy chỉnh và đối số thứ ba là chuỗi gán tùy chỉnh. Ví dụ: phần sau sẽ trả về cùng một ánh xạ như đã cho trước đó trên một chuỗi truy vấn có định dạng tùy chỉnh:

var qs = request ("chuỗi truy vấn"); console.log (qs.parse ("foo: bar ^ bingo: bango", "^", ":"))

// {foo: 'bar', bingo: 'bango'}

Người ta có thể soạn một chuỗi truy vấn bằng cách sử dụng Querystring.stringify phương pháp:

console.log (qs.stringify ({foo: 'bar', bingo: 'bango'})) // foo = bar & bingo = bango

Như với phân tích cú pháp, xâu chuỗi cũng chấp nhận các đối số phân tách và gán tùy chỉnh:

console.log (qs.stringify ({foo: 'bar', bingo: 'bango'}, "^", ":"))

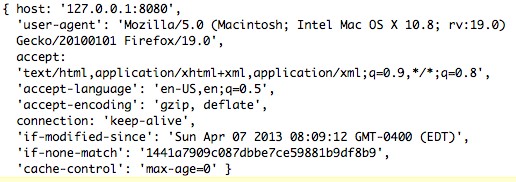
// foo: bar ^ bingo: bango

Các chuỗi truy vấn thường được kết hợp với các yêu cầu GET, được thấy sau? tính cách. Như chúng ta đã thấy ở trên, trong những trường hợp này, tự động phân tích cú pháp các chuỗi này bằng cách sử dụng mô-đun url là giải pháp đơn giản nhất. Tuy nhiên, các chuỗi được định dạng theo cách như vậy cũng hiển thị khi chúng tôi xử lý dữ liệu POST và trong những trường hợp này, Chuỗi truy vấn mô-đun được sử dụng thực sự. Chúng ta sẽ thảo luận về cách sử dụng này ngay sau đây. Nhưng trước tiên, một số điều về tiêu đề HTTP.

## 3.16. Làm việc với tiêu đề

Mỗi yêu cầu HTTP được gửi đến máy chủ Node có thể sẽ chứa thông tin tiêu đề hữu ích và các máy khách thường mong đợi nhận được thông tin gói tương tự từ máy chủ. Node cung cấp các giao diện đơn giản để đọc và ghi các tiêu đề. Chúng ta sẽ lướt qua những giao diện đơn giản đó, làm rõ một số chi tiết. Cuối cùng, chúng ta sẽ thảo luận về cách sử dụng tiêu đề nâng cao hơn có thể được triển khai trong Node, nghiên cứu một số trách nhiệm mạng phổ biến mà máy chủ Node có thể sẽ cần phải đáp ứng.

Một tiêu đề yêu cầu thông thường sẽ trông giống như sau:



Tiêu đề là các cặp Khóa / Giá trị đơn giản. Các khóa yêu cầu luôn được viết thường. Bạn có thể sử dụng bất kỳ định dạng trường hợp nào khi đặt các phím phản hồi.

Đọc tiêu đề rất đơn giản. Đọc thông tin tiêu đề bằng cách kiểm tra request.header đối tượng, là ánh xạ 1: 1 của các cặp Khóa / Giá trị của tiêu đề. Để tìm nạp tiêu đề "accept" từ ví dụ trước, chỉ cần đọc yêu cầu.

headers.accept.

Số lượng tiêu đề đến có thể được giới hạn bằng cách đặt maxHeadersCount thuộc tính của máy chủ HTTP của bạn.

Nếu các tiêu đề được ưu tiên đọc theo chương trình, Node cung cấp response.getHeader , chấp nhận khóa tiêu đề làm đối số đầu tiên của nó.

Mặc dù tiêu đề yêu cầu là các cặp Khóa / Giá trị đơn giản, nhưng khi viết tiêu đề, chúng ta cần một giao diện biểu cảm hơn. Vì phản hồi thường phải gửi mã trạng thái, Node cung cấp một cách đơn giản để chuẩn bị dòng trạng thái phản hồi và nhóm tiêu đề trong một lệnh:

response.writeHead(200, { 'Content-Length': 4096,

'Content-Type': 'text/plain'

});

Để đặt tiêu đề riêng lẻ, người ta có thể sử dụng response.setHeader, truyền hai đối số: khóa tiêu đề, theo sau là giá trị tiêu đề.

Để đặt nhiều tiêu đề có cùng tên, người ta có thể chuyển một mảng đến phản ứng.

setHeader:

response.setHeader("Set-Cookie", ["session:12345", "language=en"]);

Đôi khi, có thể cần phải xóa tiêu đề phản hồi sau khi tiêu đề đó đã được "xếp hàng đợi". Điều này được thực hiện bằng cách sử dụng response.removeHeader, chuyển tên tiêu đề cần xóa làm đối số.

Tiêu đề phải được viết trước khi viết phản hồi. Đó là lỗi khi viết tiêu đề sau khi phản hồi đã được gửi.

## 3.17. Sử dụng cookie

Giao thức HTTP là không trạng thái. Bất kỳ yêu cầu nhất định nào không có thông tin về các yêu cầu trước đó. Đối với một máy chủ, điều này có nghĩa là không thể xác định xem hai yêu cầu có nguồn gốc từ cùng một trình duyệt hay không. Cookie được phát minh để giải quyết vấn đề này. Cookie chủ yếu được sử dụng để chia sẻ trạng thái giữa máy khách (thường là trình duyệt) và máy chủ, tồn tại dưới dạng tệp văn bản nhỏ được lưu trữ trong trình duyệt.

Cookie không an toàn. Thông tin cookie truyền giữa máy chủ và máy khách ở dạng văn bản thuần túy. Có bất kỳ số điểm giả mạo nào ở giữa. Ví dụ: các trình duyệt cho phép dễ dàng truy cập chúng. Đây là một ý tưởng hay, vì không ai muốn thông tin trên trình duyệt hoặc máy cục bộ của họ bị ẩn khỏi họ, ngoài tầm kiểm soát của họ.

Tuy nhiên, cookie cũng được sử dụng khá rộng rãi để duy trì thông tin trạng thái hoặc con trỏ đến thông tin trạng thái, đặc biệt trong trường hợp phiên người dùng hoặc các tình huống xác thực khác.

Giả định rằng bạn đã quen với cách hoạt động của cookie nói chung. Ở đây chúng ta sẽ thảo luận về cách cookie được tìm nạp, phân tích cú pháp và đặt bởi máy chủ Node HTTP. Chúng tôi sẽ sử dụng ví dụ về một máy chủ phản hồi lại giá trị của một cookie đã gửi. Nếu không có cookie nào tồn tại, máy chủ sẽ tạo cookie đó và hướng dẫn khách hàng yêu cầu lại.

Đầu tiên, chúng tôi tạo một máy chủ để kiểm tra các tiêu đề yêu cầu cho cookie:

var http = request ('http'); var url = request

('url');

var server = http.createServer (function (request, response) { var cookies = request.headers.cookie;

Lưu ý rằng cookie được lưu trữ dưới dạng bánh quy thuộc tính của request.headers. Nếu không có cookie nào tồn tại cho miền này,

if(!cookies) {

var cookieName = "session"; var cookieValue = "123456"; var expiryDate = new Date();

expiryDate.setDate(expiryDate.getDate() + 1);

var cookieText = cookieName + '=' + cookieValue + ';expires='

+ expiryDate.toUTCString() + ';';

response.setHeader('Set-Cookie', cookieText); response.writeHead(302, {

'Location': '/'

});

return response.end(); }

Nếu chúng tôi đặt cookie này lần đầu tiên, khách hàng sẽ được hướng dẫn thực hiện một yêu cầu khác đến cùng máy chủ này. Vì hiện tại đã có cookie được đặt cho miền này, yêu cầu tiếp theo sẽ chứa cookie của chúng tôi, chúng tôi sẽ xử lý tiếp theo:

cookies.split(';').forEach(function(cookie) { var m = cookie.match(/(.\*?)=(.\*)$/); cookies[m[1].trim()] = (m[2] || '').trim();

});

response.end("Cookie set: " + cookies.toString());

}).listen(8080);

## 3.18. Hiểu các loại nội dung

Khách hàng thường sẽ chuyển một tiêu đề yêu cầu cho biết phản hồi mong đợi MIME

( Tiện ích mở rộng thư Internet đa mục đích) kiểu. Khách hàng cũng sẽ chỉ ra kiểu MIME của một nội dung yêu cầu. Tương tự như vậy, máy chủ sẽ cung cấp thông tin tiêu đề về kiểu MIME của nội dung phản hồi. Loại MIME cho HTML là text / html, ví dụ.

Như chúng ta đã thấy, một phản hồi HTTP có trách nhiệm thiết lập các tiêu đề mô tả thực thể mà nó chứa.

Tương tự, một yêu cầu GET thông thường sẽ chỉ ra loại tài nguyên, loại MIME, nó mong đợi như một phản hồi. Tiêu đề yêu cầu như vậy có thể trông như thế này:

Chấp nhận: text / html

Máy chủ có trách nhiệm nhận các hướng dẫn như vậy để chuẩn bị một thực thể nội dung phù hợp với kiểu MIME đã gửi và nếu nó có thể làm như vậy, nó sẽ trả về một tiêu đề phản hồi tương tự:

Nội dung-Loại: text / html; charset = utf-8

Bởi vì các yêu cầu cũng xác định tài nguyên cụ thể mong muốn (chẳng hạn như / tệp / chỉ mục. html), máy chủ phải đảm bảo rằng tài nguyên được yêu cầu mà nó đang truyền trực tiếp trở lại máy khách trên thực tế là đúng kiểu MIME. Mặc dù có vẻ rõ ràng rằng một tài nguyên được tiện ích mở rộng xác định html trên thực tế là kiểu MIME text / html, điều này hoàn toàn không chắc chắn — một hệ thống tệp không làm gì để ngăn một tệp hình ảnh được cung cấp phần mở rộng "html". Phần mở rộng phân tích cú pháp là một phương pháp không hoàn hảo để xác định loại tệp. Chung ta cân lam nhiêu hơn.

UNIX tập tin chương trình có thể xác định kiểu MIME của một tệp hệ thống. Ví dụ: người ta có thể xác định kiểu MIME của tệp không có phần mở rộng (ví dụ: nguồn) bằng cách chạy lệnh này:

file --brief --mime resource

Chúng tôi chuyển các đối số hướng dẫn tập tin để xuất ra kiểu MIME của tài nguyên và đầu ra phải ngắn gọn (chỉ kiểu MIME và không có thông tin nào khác).

Lệnh này có thể trả về một cái gì đó như văn bản / thuần túy; charset = us-ascii.

Ở đây chúng tôi có một công cụ để giải quyết vấn đề của chúng tôi.

Để biết thêm thông tin về tập tin tham khảo tiện ích đi đến liên kết sau:

http://unixhelp.ed.ac.uk/CGI/man-cgi?file

Nhắc lại rằng Node có thể sinh ra các quy trình con, chúng ta có một giải pháp cho vấn đề xác định chính xác kiểu MIME của tệp hệ thống.

Chúng ta có thể sử dụng lệnh Node người thực thi phương pháp của Node quy trình\_ con để xác định kiểu MIME của một tệp như vậy:

var exec = require('child\_process').exec;

exec("file --brief --mime resource", function(err, mime) { console.log(mime); });

Kỹ thuật này cũng hữu ích khi xác thực một tệp được truyền trực tuyến từ một vị trí bên ngoài. Tuân theo tiên đề "không bao giờ tin tưởng vào khách hàng", bạn nên kiểm tra xem liệu Loại nội dung tiêu đề của tệp được đăng lên máy chủ Node khớp với kiểu MIME thực tế của tệp đã nhận vì nó tồn tại trên hệ thống tệp cục bộ.

## 3.19. Xử lý yêu cầu biểu tượng yêu thích

Khi truy cập một URL qua trình duyệt, người ta thường sẽ nhận thấy một biểu tượng nhỏ trong tab trình duyệt hoặc trong thanh địa chỉ của trình duyệt. Biểu tượng này là một hình ảnh, có tên favicon.ico và nó được tìm nạp theo từng yêu cầu. Như vậy, một yêu cầu HTTP GET thường kết hợp hai yêu cầu — một cho biểu tượng yêu thích và một cho tài nguyên được yêu cầu.

Các nhà phát triển nút thường ngạc nhiên trước yêu cầu nhân đôi này. Bất kỳ triển khai nào của máy chủ HTTP đều phải xử lý các yêu cầu biểu tượng yêu thích. Để làm như vậy, máy chủ phải kiểm tra loại yêu cầu và xử lý nó cho phù hợp. Ví dụ sau minh họa một phương pháp làm như vậy:

var http = require('http');

http.createServer(function(request, response) { if(request.url === '/favicon.ico') { response.writeHead(200, {

'Content-Type': 'image/x-icon'

});

return response.end();

}

response.writeHead(200, {

'Content-Type': 'text/plain'

});

response.write('Some requested resource');

response.end();

}).listen(8080);

Mã này sẽ chỉ gửi một luồng hình ảnh trống cho biểu tượng yêu thích. Nếu có biểu tượng yêu thích để gửi, người ta sẽ chỉ cần đẩy dữ liệu đó qua luồng phản hồi, như chúng ta đã thảo luận trước đây.

## 3.20. Xử lý dữ liệu POST

Một trong những phương pháp REST phổ biến nhất được sử dụng trong các ứng dụng mạng là POST. Theo đặc điểm kỹ thuật REST, POST là không phải Idempotent, trái ngược với hầu hết các phương pháp nổi tiếng khác (GET, PUT, DELETE, v.v.). Điều này được đề cập để chỉ ra rằng việc xử lý dữ liệu POST thường sẽ có hậu quả ảnh hưởng đến trạng thái của ứng dụng, và do đó cần được xử lý cẩn thận.

Bây giờ chúng ta sẽ thảo luận về việc xử lý loại dữ liệu POST phổ biến nhất, được gửi qua biểu mẫu. Loại POST phức tạp hơn — tải lên nhiều phần — sẽ được thảo luận trong Chương 4, Sử dụng Node để Truy cập Hệ thống Tập tin.

Hãy tạo một máy chủ sẽ trả về một biểu mẫu cho máy khách và gửi lại bất kỳ dữ liệu nào mà máy khách gửi cùng với biểu mẫu đó. Trước tiên, chúng tôi sẽ cần kiểm tra URL yêu cầu, xác định xem đây là yêu cầu biểu mẫu hay gửi biểu mẫu, trả về HTML cho biểu mẫu trong trường hợp đầu tiên và phân tích cú pháp dữ liệu đã gửi trong trường hợp thứ hai:

var http = require('http'); var qs = require('querystring'); http.createServer(function(request, response) {

var body = ""; if(request.url === "/") { response.writeHead(200, {

"Content-Type": "text/html"

});

return response.end(

'<form action="/submit" method="post">\ <input type="text" name="sometext">\

<input type="submit" value="Upload">\

</form>'

); }

Lưu ý rằng biểu mẫu mà chúng tôi phản hồi có một trường duy nhất có tên một số tiếp theo. Biểu mẫu này phải ĐĂNG dữ liệu trong biểu mẫu sometext = enter\_text:

if(request.url === "/submit") { request.on('readable', function() { body += request.read();

});

request.on('end', function() { var fields = qs.parse(body); response.end("Thanks!"); console.log(fields)

}); }

}).listen(8080);

Sau khi luồng ĐĂNG của chúng tôi kết thúc và khách hàng được thông báo rằng ĐĂNG đã được nhận, chúng tôi phân tích cú pháp dữ liệu đã đăng bằng cách sử dụng Querystring.parse, cung cấp cho chúng tôi bản đồ Khóa / Giá trị có thể truy cập qua các trường ["somedata"].

## 3.21. Tạo và phát trực tuyến hình ảnh với Node

Sau khi xem qua các chiến lược chính để bắt đầu và chuyển hướng các luồng dữ liệu, hãy thực hành lý thuyết bằng cách tạo một dịch vụ để truyền trực tuyến (được đặt tên phù hợp) PNG

( Biểu đồ minh họa mạng lưới không dây) hình ảnh cho khách hàng. Tuy nhiên, đây sẽ không phải là một máy chủ tệp đơn giản. Mục tiêu là tạo luồng dữ liệu PNG bằng cách tạo luồng đầu ra của một

ImageMagick đổi hoạt động thực thi trong một quy trình riêng biệt vào luồng phản hồi của kết nối HTTP, nơi bộ chuyển đổi đang dịch một luồng khác của SVG (Đồ họa Vectơ có thể mở rộng) dữ liệu được tạo trong một ảo hóa DOM

( Mô hình Đối tượng Tài liệu) tồn tại trong thời gian chạy Node. Bắt đầu nào.

Mã đầy đủ cho ví dụ này có thể được tìm thấy trong gói mã của bạn.

Mục tiêu của chúng tôi là sử dụng Node để tạo biểu đồ hình tròn động trên máy chủ dựa trên yêu cầu của khách hàng. Ứng dụng khách sẽ chỉ định một số giá trị dữ liệu và PNG đại diện cho dữ liệu đó trong một chiếc bánh sẽ được tạo. Chúng tôi sẽ sử dụng D3.js thư viện, cung cấp một API Javascript để tạo trực quan hóa dữ liệu và jsdom Gói NPM, cho phép chúng tôi tạo một DOM ảo trong một quy trình Node.

Ngoài ra, PNG mà chúng tôi tạo sẽ được ghi vào một tệp. Nếu các yêu cầu trong tương lai chuyển các đối số truy vấn tương tự đến dịch vụ của chúng tôi, thì chúng tôi sẽ có thể nhanh chóng chuyển kết xuất hiện có ngay lập tức mà không cần phải tạo lại nó.

## 3.22. Truyền dữ liệu qua các nút và khách hàng

Biểu đồ hình tròn biểu thị một phạm vi nếu tỷ lệ phần trăm có tổng của nó lấp đầy tổng diện tích của một hình tròn, được hiển thị dưới dạng các lát cắt. Dịch vụ của chúng tôi sẽ vẽ một biểu đồ như vậy dựa trên các giá trị mà khách hàng gửi. Trong hệ thống của chúng tôi, khách hàng được yêu cầu gửi các giá trị cộng đến 1, chẳng hạn như .5,

. 3, .2. Do đó, máy chủ của chúng tôi khi nhận được yêu cầu sẽ cần tìm nạp các tham số truy vấn cũng như tạo một khóa duy nhất ánh xạ tới các yêu cầu trong tương lai có cùng tham số truy vấn:

|  |  |
| --- | --- |
| var values = url.parse(request.url, true).query['values']. | var values = url.parse(request.url, true).query['values']. |

Tại đây, chúng tôi thấy mô-đun URL đang hoạt động, lấy ra các giá trị dữ liệu của chúng tôi. Ngoài ra, chúng tôi tạo khóa trên các giá trị này bằng cách sắp xếp các giá trị trước tiên, sau đó nối chúng thành một chuỗi mà chúng tôi sẽ sử dụng làm tên tệp cho biểu đồ hình tròn được lưu trong bộ nhớ cache của chúng tôi. Chúng tôi sắp xếp các giá trị vì lý do này: cùng một biểu đồ đạt được bằng cách gửi. 5 .3 .2 và. 3 .5 .2. Bằng cách sắp xếp và kết hợp chúng trở thành tên tệp. 2 .3 .5.

Trong một ứng dụng sản xuất, cần phải thực hiện nhiều công việc hơn để đảm bảo rằng truy vấn được hình thành tốt, chính xác về mặt toán học, v.v. Trong ví dụ của chúng tôi, chúng tôi giả định các giá trị thích hợp đang được gửi.

## 3.23. Tạo, lưu vào bộ nhớ đệm và gửi biểu diễn PNG

Giả sử chúng ta không có bản sao được lưu trong bộ nhớ cache, chúng ta sẽ cần tạo một bản sao. Hệ thống của chúng tôi hoạt động bằng cách tạo một DOM ảo, sử dụng D3 để tạo biểu đồ tròn SVG bên trong DOM đó, chuyển SVG đã tạo đến ImageMagick đổi chương trình chuyển đổi dữ liệu SVG thành biểu diễn PNG. Ngoài ra, chúng tôi sẽ cần lưu trữ PNG đã tạo trên hệ thống tệp.

Chúng tôi sử dụng jsdom để tạo DOM và sử dụng D3 để tạo biểu đồ hình tròn. Chuyến thăm https://github.com/tmpvar/jsdom để tìm hiểu về cách jsdom hoạt động và http://d3js.org/ để tìm hiểu về cách sử dụng D3 để tạo SVG. Giả sử rằng chúng tôi đã tạo SVG này và lưu trữ nó trong một biến svg, sẽ chứa một chuỗi tương tự như sau:

<svg width="200" height="200">

<g transform="translate(100,100)">

<defs>

<radialgradient id="grad-0" gradientUnits="userSpaceOnUse" cx="0" cy="0" r="100"> <stop offset="0" stop-color="#7db9e8"></stop> ...

. . .

Bây giờ chúng ta phải chuyển đổi SVG đó thành PNG. Để làm điều này, chúng tôi tạo ra một tiến trình con chạy ImageMagick đổi chương trình và truyền trực tuyến dữ liệu SVG của chúng tôi tới stdin của quá trình đó:

var svgToPng = spawn ("convert", ["svg:", "png: -"]); svgToPng.stdin.write (svg); svgToPng.stdin.end ();

Các stdout sau đó svgToPng quy trình (a ReadableStream) sẽ đẩy một luồng byte đại diện cho PNG của biểu đồ hình tròn. Luồng này sẽ được chuyển đến hai

Dòng ghi: đối tượng phản hồi, để khách hàng của chúng tôi nhận được dữ liệu PNG và tệp PNG mới, tên tệp nào sẽ được lưu trữ trong biến cacheKey. Một luồng phản hồi đã tồn tại; bây giờ chúng ta phải tạo luồng tệp:

var filewriter = fs.createWriteStream (cacheKey); filewriter.on ("open", function (err) {

. . .

Khi chúng tôi đã mở thành công một luồng tệp, chúng tôi có tất cả các luồng của chúng tôi tại chỗ: a ReadableStream đại diện bởi stdout của quá trình chuyển đổi; a

WordsStream liên kết với một tệp trên hệ thống tệp của chúng tôi; và WordsStream

được đại diện bởi đối tượng phản hồi của máy chủ của chúng tôi. Bây giờ chúng ta phải đạt được luồng này: luồng chuyển đổi PNG> luồng tệp> luồng phản hồi:

svgToPng.stdout.pipe(file).pipe(response);

Tuy nhiên, điều này sẽ không hoạt động như sau: a WordsStream ( tệp) không thể được chuyển sang một tệp khác

WordsStream ( phản hồi), như vậy một luồng không đẩy dữ liệu. Một cách để giải quyết đây là để sử dụng một TransformStream:

var streamer = new stream.Transform(); streamer.\_transform = function(data, enc, cb) { filewriter.write(data); this.push(data); cb();

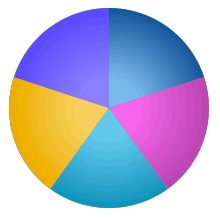
}; svgToPng.stdout.pipe(streamer).pipe(response);

Nhắc lại cuộc thảo luận của chúng tôi về luồng, những gì đang xảy ra ở đây sẽ rõ ràng. A TransformStream chức năng như một DuplexStream, vừa có thể đọc được vừa có thể ghi. Chúng tôi ghi đè \_ trừu tượng của nó biến đổi và thực hiện ghi tệp của chúng tôi ở đó. Khi chúng tôi ghi thành công vào một tệp, sau đó chúng tôi đẩy cùng một dữ liệu về phía trước vào luồng phản hồi. Bây giờ chúng ta có thể đạt được chuỗi luồng mong muốn:

svgToPng.stdout.pipe(streamer).pipe(response);

## 3.24. Truyền dữ liệu qua các nút và khách hàng

Và khách hàng nhận được một biểu đồ hình tròn:



Cuối cùng, chúng tôi cần xử lý các trường hợp biểu đồ hình tròn được yêu cầu đã được hiển thị và có thể được truyền trực tiếp từ hệ thống tệp:

fs.exists(cacheKey, function(exists) { response.writeHead(200, {

'Content-Type': 'image/png'

});

if(exists) {

fs.createReadStream(cacheKey) .on('readable', function() { var chunk;

while(chunk = this.read()) { response.write(chunk);

}

})

.on('end', function() { response.end();

});

return; } ...

Khi chúng tôi đã xác định rằng một tệp được lưu trong bộ nhớ cache tồn tại, chúng tôi chạy trong khi vòng lặp kéo dữ liệu ra khỏi luồng tệp, ghi dữ liệu này vào một đối tượng phản hồi, cho đến khi luồng tệp kết thúc.

## Tóm lược

Như chúng ta đã tìm hiểu, các nhà thiết kế của Node đã thành công trong việc tạo ra một giải pháp đơn giản, dễ dự đoán và thuận tiện cho vấn đề rất khó khăn trong việc kích hoạt I / O hiệu quả giữa các nguồn và mục tiêu khác nhau. Trừu tượng của nó Suối giao diện tạo điều kiện thuận lợi cho việc khởi tạo các giao diện có thể đọc và ghi nhất quán, đồng thời việc mở rộng giao diện này thành các yêu cầu và phản hồi HTTP, hệ thống tệp, quy trình con và các kênh dữ liệu khác làm cho việc lập trình dòng với Node trở thành một trải nghiệm thú vị.

Bây giờ chúng ta đã học cách thiết lập máy chủ HTTP để xử lý các luồng dữ liệu đến từ nhiều máy khách được kết nối đồng thời và cách cung cấp cho các máy khách đó các luồng có bộ đệm, chúng ta có thể bắt đầu tham gia sâu hơn vào nhiệm vụ xây dựng đồng thời cấp doanh nghiệp hệ thống thời gian thực với Node.

# CHƯƠNG 4

# Sử dụng Node để truy cập Hệ thống tập tin

Chúng ta có các đối tượng liên tục - chúng được gọi là tệp.

- Ken Thompson

Một tập tin chỉ đơn giản là một phần dữ liệu được lưu giữ, thường là trên một số phương tiện cứng như ổ cứng. Các tệp thường bao gồm một chuỗi các byte có mã hóa ánh xạ vào một số mẫu khác, như một chuỗi số hoặc xung điện. Có thể có số lượng mã hóa gần như vô hạn, với một số mã hóa phổ biến là tệp văn bản, tệp hình ảnh và tệp nhạc. Các tệp có độ dài cố định và để được đọc, mã hóa ký tự của chúng phải được giải mã bởi một số loại trình đọc, như máy nghe nhạc MP3 hoặc trình xử lý văn bản.

Khi một tệp đang được truyền tải, di chuyển qua một dây cáp sau khi nó bị hút khỏi một số thiết bị lưu trữ, nó không khác gì bất kỳ luồng dữ liệu nào khác chạy qua dây. Trạng thái rắn trước đây của nó chỉ là một bản thiết kế ổn định có thể được sao chép, cắt nhỏ và làm lại một cách dễ dàng và vô hạn.

Chúng tôi đã thấy cách các luồng có sự kiện phản ánh các nguyên tắc thiết kế cốt lõi thông báo cho thiết kế của Node, nơi các luồng byte sẽ được đọc và ghi vào, và được chuyển vào các luồng khác, tạo ra các sự kiện luồng có liên quan, chẳng hạn như kết thúc. Tệp được hiểu đơn giản là các vùng chứa dữ liệu, chứa đầy các byte có thể được trích xuất hoặc chèn một phần hoặc toàn bộ.

Ngoài sự giống nhau tự nhiên của chúng với các luồng, các tệp còn hiển thị các đặc điểm của các đối tượng. Tệp có các thuộc tính mô tả giao diện có sẵn để truy cập nội dung tệp — cấu trúc dữ liệu với các thuộc tính và phương thức truy cập liên quan.

Một hệ thống tập tin phản ánh một số khái niệm về cách tổ chức tệp — cách chúng được xác định, nơi chúng được lưu trữ, cách chúng được truy cập, v.v. Hệ thống tệp chung cho người dùng UNIX là UFS (Hệ thống tệp Unix), trong khi người dùng Windows có thể quen thuộc với NTFS (Hệ thống tệp công nghệ mới).

Điều thú vị là các nhà thiết kế của hệ điều hành Plan 9 (một nhóm bao gồm Ken Thompson) quyết định có tất cả các giao diện điều khiển được biểu diễn dưới dạng hệ thống tệp, sao cho tất cả các giao diện hệ thống (trên các thiết bị, trên các ứng dụng) được mô hình hóa dưới dạng các hoạt động tệp. Xử lý tệp như công dân hạng nhất là một triết lý mà Hệ điều hành UNIX cũng sử dụng — sử dụng tệp làm tham chiếu đến các đường ống và ổ cắm được đặt tên, trong số những thứ khác, mang lại cho các nhà phát triển sức mạnh to lớn khi định hình luồng dữ liệu.

Các đối tượng tệp cũng mạnh mẽ và hệ thống mà chúng cư trú bên trong thể hiện các giao diện I / O cơ bản phải dễ sử dụng, nhất quán và rất nhanh. Không ngạc nhiên, Node's tập tin mô-đun chỉ hiển thị một giao diện như vậy.

Chúng tôi sẽ xem xét việc xử lý tệp trong Node từ hai khía cạnh sau: cách nội dung dữ liệu tệp được truyền vào và ra (đọc và ghi) và cách các thuộc tính của đối tượng tệp được sửa đổi, chẳng hạn như thay đổi quyền đối với tệp.

Ngoài ra, chúng tôi sẽ đề cập đến các trách nhiệm của máy chủ Node, về việc chấp nhận tải lên tệp và phục vụ các yêu cầu tệp. Bằng cách làm việc thông qua các ví dụ

của trình vòng lặp thư mục và máy chủ tệp, phạm vi đầy đủ và hành vi của API hệ thống tệp của Node sẽ trở nên rõ ràng.

Đối với những người có kiến thức về C ++ và ham hiểu biết lành mạnh, việc triển khai hệ thống tệp trong Node có thể được duyệt tại đây: https://github.com/ joyent / node / blob / master / src / node\_file.cc.

## 4.1. Thư mục và lặp qua các tệp và thư mục

Cải thiện hiệu suất của Node liên quan đến hệ thống tệp là một trong những thách thức chính mà nhóm Node cốt lõi thực hiện. Ví dụ: một so sánh tiêu cực (thiếu sót) với tốc độ phân phối tệp của môi trường Vert.x đã dẫn đến một số nhận xét thú vị từ các nhà phát triển và người dùng của cả hai hệ thống, mặc dù không phải tất cả đều mang tính xây dựng, nhưng là một bài đọc tốt nếu ai đó quan tâm đến cách hệ thống tệp được truy cập bởi

Node và các hệ thống khác: http://vertxproject.wordpress.com/2012/05/09/ vert-x-vs-node-js-simple-http-benchmark /.

Thông thường, một hệ thống tệp nhóm các tệp thành các bộ sưu tập, thường được gọi là thư mục. Một người điều hướng qua các thư mục để tìm các tệp riêng lẻ. Khi tệp đích được tìm thấy, đối tượng tệp phải được bao bọc bởi một giao diện hiển thị nội dung tệp để đọc và ghi.

Vì quá trình phát triển Node thường liên quan đến việc tạo ra các máy chủ vừa chấp nhận vừa phát ra dữ liệu tệp, nên cần rõ tốc độ truyền ở lớp I / O hoạt động và quan trọng này là như thế nào. Như đã đề cập trước đó, tệp cũng có thể được hiểu là các đối tượng và các đối tượng có một số thuộc tính nhất định.

## 4.2. Các loại tệp

Có sáu loại tệp thường gặp trên hệ thống UNIX:

Tệp thông thường: Chúng chứa một mảng byte một chiều và không thể chứa các tệp khác.

Thư mục: Đây cũng là các tệp, được triển khai theo một cách đặc biệt để chúng có thể mô tả bộ sưu tập các tệp khác.

Ổ cắm: Được sử dụng cho IPC, cho phép các tiến trình trao đổi dữ liệu.

Đường ống được đặt tên: Một lệnh chẳng hạn như ps aux | nút grep tạo ra một đường ống, đường ống này sẽ bị phá hủy sau khi kết thúc hoạt động. Các đường ống được đặt tên là liên tục và có thể xác định được địa chỉ, đồng thời có thể được nhiều quy trình sử dụng cho IPC vô thời hạn.

Tệp thiết bị: Đây là các đại diện của các thiết bị I / O, các quy trình chấp nhận các luồng dữ liệu. / dev / null thường là một ví dụ về tệp thiết bị ký tự (chấp nhận các luồng I / O nối tiếp) và / dev / sda là một ví dụ về tệp thiết bị khối (cho phép truy cập ngẫu nhiên I / O cho các khối dữ liệu) đại diện cho một ổ đĩa dữ liệu.

Liên kết: Đây là những con trỏ đến các tệp khác gồm hai loại, liên kết cứng và liên kết tượng trưng. Liên kết cứng trỏ trực tiếp đến tệp khác và không thể phân biệt được với tệp đích. Các liên kết tượng trưng là các con trỏ gián tiếp và có thể phân biệt được với các tệp bình thường.

Hầu hết các tương tác với hệ thống tệp Node chỉ gặp phải hai loại đầu tiên, với loại thứ ba chỉ gián tiếp thông qua API Node. Giải thích sâu hơn về các loại còn lại nằm ngoài phạm vi của cuộc thảo luận này. Tuy nhiên, Node cung cấp một bộ hoạt động tệp đầy đủ thông qua tập tin và người đọc phải có ít nhất một số quen thuộc với toàn bộ phạm vi và sức mạnh của các loại tệp.

Nghiên cứu các đường ống được đặt tên sẽ thưởng cho người đọc quan tâm đến việc hiểu cách Node được thiết kế để hoạt động với các luồng và đường ống. Hãy thử điều này từ một thiết bị đầu cuối: mkfifo tênpipe

Tiếp theo ls –l một danh sách tương tự như thế này sẽ được hiển thị:

prw-r - r-- 1 nhân viên hệ thống 0 tháng 5 01 07:52 tênpipe

Lưu ý p cờ ở chế độ tệp. Bạn đã tạo một đường ống được đặt tên. Bây giờ nhập cái này vào cùng một thiết bị đầu cuối, đẩy một số byte vào đường ống có tên: echo "xin chào"> namepipe

Có vẻ như quá trình sẽ bị treo. Nó không - các đường ống, giống như ống nước, phải mở ở cả hai đầu để hoàn thành công việc xả nước. Chúng tôi đã bơm một số byte vào… bây giờ thì sao?

Mở một thiết bị đầu cuối khác, điều hướng đến cùng một thư mục và nhập:

mèo tênpipe

xin chào sẽ xuất hiện dưới dạng nội dung của ống tên đang đỏ bừng. Cũng lưu ý rằng thiết bị đầu cuối đầu tiên không còn bị treo.

## 4.3. Đường dẫn tệp

Hầu hết các hệ thống tệp được cung cấp bởi Node sẽ yêu cầu thao tác với các đường dẫn tệp và vì mục đích này, chúng tôi sử dụng con đường mô-đun. Chúng ta có thể soạn, phân tách và liên kết các đường dẫn với mô-đun này. Thay vì thực hiện thủ công các quy trình chia nhỏ chuỗi đường dẫn của riêng bạn và tạo lại và nối, hãy cố gắng bình thường hóa mã của bạn bằng cách ủy quyền thao tác đường dẫn cho mô-đun này.

Sử dụng path.normalize bất cứ khi nào làm việc với chuỗi đường dẫn tệp có nguồn không đáng tin cậy hoặc không đáng tin

cậy, để đảm bảo định dạng có thể dự đoán:

var path = require('path');

path.normalize("../one////two/./three.html"); // -> ../one/two/three.html

Sử dụng path.join bất cứ khi nào tạo một con đường duy nhất ngoài các đoạn đường dẫn:

path.join("../","one","two","three.html");

// -> ../one/two/three.html

Sử dụng path.dirname để cắt tên thư mục ra khỏi đường dẫn: path.dirname ("../ one / two / three.html");

// ../một hai

Sử dụng path.basename để thao tác phân đoạn đường dẫn cuối cùng:

path.basename ("../ one / two / three.html"); // -> ba.html

// Xóa phần mở rộng tệp khỏi tên cơ sở path.basename ("../ one / two / three.html", ".html");

// -> ba

Sử dụng path.extname để cắt từ dấu chấm cuối cùng (.) đến cuối chuỗi đường dẫn:

var pstring = "../one/two/three.html"; path.extname (pstring);

// -> .html

//

// Cái nào giống với:

// pstring.slice (pstring.lastIndexOf ("."));

Sử dụng path.relative để tìm đường dẫn tương đối từ đường dẫn tuyệt đối này đến đường dẫn tuyệt đối khác:

path.relative(

'/one/two/three/four',

'/one/two/thumb/war'

);

// -> ../../thumb/war

path.resolve('/one/two', '/three/four');

// -> /three/four

path.resolve('/one/two/three', '../', 'four', '../../five') // -> /one/five

Nghĩ về các đối số được chuyển đến path.resolve như là một chuỗi của CD cuộc gọi:

cd /one/two/three cd ../ cd four cd ../../five pwd

// -> /one/five

Nếu danh sách các đối số được chuyển đến path.resolve không thể cung cấp một đường dẫn tuyệt đối, tên thư mục hiện tại cũng được sử dụng. Ví dụ, nếu chúng ta đang ở

/users/home/john/:

path.resolve('one','two/three','four'); // -> /users/home/john/one/two/three/four

Các đối số này giải quyết thành một đường dẫn tương đối một hai ba bốn do đó được đặt trước tên thư mục hiện tại.

## 4.4. Thuộc tính tệp

Đối tượng tệp thể hiện một số thuộc tính của nó, bao gồm một tập hợp siêu dữ liệu hữu ích về dữ liệu tệp. Nếu một người đang sử dụng Node để chạy một máy chủ HTTP thì nó sẽ là cần thiết

để xác định độ dài tệp của bất kỳ tệp nào được yêu cầu thông qua GET chẳng hạn. Việc xác định thời gian tệp được sửa đổi lần cuối cho thấy việc sử dụng trên nhiều loại ứng dụng.

Để đọc các thuộc tính của một tệp, hãy sử dụng fs.stat:

fs.stat("file.txt", function(err, stats) { console.log(stats); });

Trong ví dụ trước, số liệu thống kê sẽ là một fs.Stats đối tượng mô tả tệp thông qua bản đồ các thuộc tính:

dev: 2051, // id of device containing this file mode: 33188, // bitmask, status of the file nlink: 1, // number of hard links uid: 0, // user id of file owner gid: 0, // group id of file owner rdev: 0, // device id (if device file) blksize: 4096, // I/O block size ino: 27396003, // a unique file inode number size: 2000736, // size in bytes blocks: 3920, // number of blocks allocated

atime: Fri May 3 2013 15:39:57 GMT-0500 (CDT), // last access mtime: Fri May 3 2013 17:22:46 GMT-0500 (CDT), // last modified ctime: Fri May 3 2013 17:22:46 GMT-0500 (CDT) // last status change

An fs.Stats đối tượng tiết lộ một số phương pháp hữu ích để truy cập dữ liệu thuộc tính tệp:

Sử dụng stats.isFile để kiểm tra các tệp tiêu chuẩn

Sử dụng stats.isDirectory để kiểm tra các thư mục

Sử dụng stats.isBlockDevice để kiểm tra các tệp thiết bị loại khối

Sử dụng stats.isCharacterDevice để kiểm tra các tệp thiết bị loại ký tự

Sử dụng stats.isSymbolicLink sau một fs.lstat để tìm các liên kết tượng trưng

Sử dụng stats.isFIFO để xác định các đường ống được đặt tên

Sử dụng stats.isSocket để kiểm tra ổ cắm

Còn hai nữa chỉ số các phương pháp có sẵn:

fs.fstat (fd, gọi lại): Giống fs.stat, ngoại trừ một bộ mô tả tệp fd được chuyển chứ không phải là một đường dẫn tệp

fs.lstat (đường dẫn, gọi lại): An fs.stat trên một liên kết tượng trưng sẽ trả về một fs.Stats đối tượng cho tệp đích, trong khi fs.lstat sẽ trả lại một fs.Stats đối tượng cho chính tệp liên kết

Hai phương pháp sau đây đơn giản hóa thao tác trên dấu thời gian tệp:

fs.utimes (đường dẫn, atime, mtime, callback): Thay đổi quyền truy cập và sửa đổi dấu thời gian trên một tệp tại con đường. Thời gian truy cập và sửa đổi tệp được lưu trữ dưới dạng các bản sao của JavaScript Ngày vật. Date.getTime chẳng hạn, sẽ trả về số mili giây đã trôi qua kể từ nửa đêm (UTC) vào tháng 1

1 năm 1970.

fs.futimes (fd, atime, mtime, callback): Thay đổi quyền truy cập và sửa đổi dấu thời gian trên bộ mô tả tệp fd. Tương tự với fs.utimes.

Thông tin thêm về thao tác ngày và giờ với JavaScript có thể được tìm thấy ở đây: https://developer.mozilla.org/en-US/ docs / JavaScript / Reference / Global\_Objects / Date.

## 4.5. Mở và đóng tệp

Một trong những quy tắc không chính thức điều chỉnh dự án Node là không trừu tượng hóa một cách không cần thiết khỏi các chi tiết triển khai hệ điều hành hiện có. Như chúng ta sẽ thấy, các tham chiếu đến bộ mô tả tệp xuất hiện trong toàn bộ API tệp của Node. Đối với POSIX (Giao diện Hệ điều hành Di động), một bộ mô tả tệp chỉ đơn giản là một số nguyên (không âm) tham chiếu duy nhất đến một tệp cụ thể. Vì Node đã lập mô hình hệ thống tệp của nó trên POSIX, nên không ngạc nhiên khi một bộ mô tả tệp được biểu diễn trong Node dưới dạng số nguyên.

Nhắc lại cuộc thảo luận của chúng ta về cách các thiết bị và các phần tử khác của HĐH được biểu diễn dưới dạng tệp, có thể lý do rằng các luồng I / O tiêu chuẩn ( stdin, stdout, stderr) cũng sẽ có bộ mô tả tệp. Trên thực tế, đó là trường hợp:

console.log(process.stdin.fd); // 0 console.log(process.stdout.fd); // 1 console.log(process.stderr.fd); // 2 fs.fstat(1, function(err, stat) { console.log(stat); // an fs.Stats object });

Trình mô tả tệp rất dễ lấy và cách thuận tiện để chuyển các tham chiếu tệp. Hãy xem cách các bộ mô tả tệp được tạo và sử dụng bằng cách kiểm tra cách thực hiện các thao tác mở và đóng tệp ở mức thấp bằng Node. Khi chương tiếp tục, chúng tôi sẽ điều tra các giao diện tinh chỉnh hơn cho các luồng tệp.

fs.open (đường dẫn, cờ, [chế độ], gọi lại)

Đang cố gắng mở một tệp tại con đường. gọi lại sẽ nhận được bất kỳ ngoại lệ nào với thao tác làm đối số

fs.open("path.js", "r", function(err, fileDescriptor) { console.log(fileDescriptor); // An integer, like `7` or `23` });

cờ nhận một chuỗi cho biết các loại hoạt động mà người gọi mong đợi thực hiện trên bộ mô tả tệp trả về. Ý nghĩa của chúng phải rõ ràng:

r: Mở tệp để đọc, ném một ngoại lệ nếu tệp không tồn tại.

r +: Mở tệp để đọc và ghi, đưa ra một ngoại lệ nếu tệp không tồn tại.

w: Mở tệp để ghi, tạo tệp nếu tệp không tồn tại và cắt tệp xuống 0 byte nếu tệp tồn tại.

wx: Giống w, nhưng mở tệp ở chế độ độc quyền, có nghĩa là nếu tệp đã tồn tại, nó sẽ không được mở, và hoạt động mở sẽ không thành công. Điều này hữu ích nếu nhiều quá trình có thể đồng thời cố gắng tạo cùng một tệp.

w +: Mở tệp để đọc và ghi, tạo tệp nếu nó không tồn tại và cắt bớt tệp thành 0 byte nếu nó tồn tại.

wx +: Giống wx ( và w), mở thêm tệp để đọc.

a: Mở tệp để bổ sung, tạo tệp nếu tệp không tồn tại.

cây rìu: Giống a, nhưng mở tệp ở chế độ độc quyền, có nghĩa là nếu tệp đã tồn tại, nó sẽ không được mở, và hoạt động mở sẽ không thành công. Điều này hữu ích nếu nhiều quá trình có thể đồng thời cố gắng tạo cùng một tệp.

a +: Mở tệp để đọc và bổ sung, tạo tệp nếu tệp không tồn tại.

rìu +: Giống cây rìu ( và a), mở thêm tệp để đọc.

Khi một thao tác có thể tạo một tệp mới, hãy sử dụng tùy chọn chế độ để đặt quyền cho tệp này bằng chữ số bát

fs.open("index.html","w", 755, function(err, fd) {

fs.read(fd, …); });

fs.close (fd, gọi lại)

Các fs.close (fd, gọi lại) phương thức đóng một bộ mô tả tệp. Lệnh gọi lại nhận một đối số, bất kỳ ngoại lệ nào được đưa ra trong lệnh gọi. Một thói quen tốt là đóng tất cả các trình mô tả tệp đã được mở.

## 4.6. Hoạt động tệp

Node thực hiện các chức năng POSIX tiêu chuẩn để làm việc với các tệp mà người dùng UNIX sẽ quen thuộc. Chúng tôi sẽ không trình bày sâu về từng thành viên của bộ sưu tập mở rộng này, thay vào đó tập trung vào một số ví dụ thường được sử dụng. Đặc biệt, chúng ta sẽ đi sâu thảo luận về các phương pháp mở bộ mô tả tệp và thao tác với dữ liệu tệp, đọc và thao tác các thuộc tính tệp cũng như di chuyển qua các thư mục hệ thống tệp. Tuy nhiên, người đọc được khuyến khích thử nghiệm với toàn bộ bộ, mà danh sách sau đây mô tả ngắn gọn. Lưu ý rằng tất cả các phương pháp này là hoạt động tệp không đồng bộ, không chặn.

fs.rename (oldName, newName, callback)

Các fs.rename (oldName, newName, callback) phương thức đổi tên tệp tại tên Cu đến tên mới. Lệnh gọi lại nhận một đối số, bất kỳ ngoại lệ nào được đưa ra trong lệnh gọi. fs.truncate (đường dẫn, len, gọi lại)

Các fs.truncate (đường dẫn, len, gọi lại) phương pháp thay đổi độ dài của nộp tại con đường bởi len byte. Nếu len đại diện cho độ dài ngắn hơn độ dài hiện tại của tệp, tệp được cắt ngắn theo độ dài đó. Nếu len lớn hơn, độ dài tệp được đệm bằng cách thêm byte rỗng (\ x00) cho đến khi len đạt được. Lệnh gọi lại nhận một đối số, bất kỳ ngoại lệ nào được đưa ra trong lệnh gọi.

fs.ftruncate (fd, len, callback)

Các fs.ftruncate (fd, len, callback) phương pháp giống như fs.truncate, ngoại trừ việc thay vì chỉ định một tệp, một bộ mô tả tệp được chuyển dưới dạng fd.

fs.chown (đường dẫn, uid, gid, gọi lại)

Các fs.chown (đường dẫn, uid, gid, gọi lại) phương thức thay đổi quyền sở hữu của tệp tại con đường. Sử dụng điều này để đặt liệu người dùng uid hoặc nhóm gid có quyền truy cập vào một tệp. Lệnh gọi lại nhận một đối số, bất kỳ ngoại lệ nào được đưa ra trong lệnh gọi.

fs.fchown (fd, uid, gid, callback)

Các fs.fchown (fd, uid, gid, callback) phương pháp giống như fs.chown, ngoại trừ việc thay vì chỉ định đường dẫn tệp, bộ mô tả tệp được chuyển dưới dạng fd.

fs.lchown (đường dẫn, uid, gid, gọi lại)

Các fs.lchown (đường dẫn, uid, gid, gọi lại) phương pháp giống như fs.chown, ngoại trừ việc trong trường hợp liên kết tượng trưng, quyền sở hữu của tệp liên kết bị thay đổi, nhưng không phải là liên kết được tham chiếu.

fs.chmod (đường dẫn, chế độ, gọi lại)

Các fs.chmod (đường dẫn, chế độ, gọi lại) phương pháp thay đổi chế độ ( quyền)

trên một tập tin tại con đường. Bạn đang đặt các bit đọc (4), ghi (2) và thực thi (1) cho tệp này, có thể được gửi bằng các chữ số bát phân:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | [đọc | [w] nghi thức | Hành hình | Toàn bộ |
| Chủ nhân  Nhóm Khác | 4 4  4 | 2 0  0 | 1 1  1 | 7 5 5 chmod (755) |

Bạn cũng có thể sử dụng các biểu diễn tượng trưng, chẳng hạn như g + rw cho nhóm đọc và viết, tương tự như các đối số mà chúng tôi đã thấy mở tập tin sớm hơn. Để biết thêm thông tin về thiết lập chế độ tệp, tham khảo ý kiến: http://en.wikipedia.org/wiki/Chmod.

Lệnh gọi lại nhận một đối số, bất kỳ ngoại lệ nào được đưa ra trong lệnh gọi.

fs.fchmod (fd, chế độ, gọi lại)

Các fs.fchmod (fd, chế độ, gọi lại) phương pháp giống như fs.chmod, ngoại trừ điều đó thay thế xác định đường dẫn tệp, bộ mô tả tệp được chuyển dưới dạng fd.

fs.lchmod (đường dẫn, chế độ, gọi lại)

Các fs.lchmod (đường dẫn, chế độ, gọi lại) phương pháp giống như fs.chmod, ngoại trừ điều đó trong

trường hợp các quyền liên kết tượng trưng trên chính tệp liên kết bị thay đổi, nhưng không thay đổi các quyền của liên kết được tham chiếu.

fs.link (srcPath, dstPath, callback)

Các fs.link (srcPath, dstPath, callback) tạo ra một liên kết cứng giữa srcPath và dstPath. Đây là cách tạo nhiều đường dẫn khác nhau đến chính xác cùng một tệp. Ví dụ: thư mục sau chứa một tệp target.txt và hai liên kết cứng— a.txt và b.txt —Mỗi điểm đến tệp này:



Lưu ý rằng target.txt trống rỗng. Nếu nội dung của tệp đích bị thay đổi, độ dài của các tệp liên kết cũng sẽ bị thay đổi. Thay đổi nội dung của tệp đích:

echo "xin chào" >> target.txt

Điều này dẫn đến cấu trúc thư mục mới này, thể hiện rõ ràng các tham chiếu cứng:



Lệnh gọi lại nhận một đối số, bất kỳ ngoại lệ nào được đưa ra trong lệnh gọi.

fs.symlink (srcPath, dstPath, [type], callback)

Các fs.symlink (srcPath, dstPath, [type], callback) phương pháp tạo ra

một liên kết tượng trưng giữa srcPath và dstPath. Không giống như các liên kết cứng được tạo bằng

fs.link, các liên kết tượng trưng chỉ đơn giản là con trỏ đến các tệp khác và bản thân chúng không phản hồi với các thay đổi trong tệp đích. Liên kết mặc định kiểu là tệp. Các tùy chọn khác là thư mục và đường giao nhau, tùy chọn cuối cùng là loại dành riêng cho Windows bị bỏ qua trên các hệ thống khác. Lệnh gọi lại nhận một đối số, bất kỳ ngoại lệ nào được đưa ra trong lệnh gọi.

So sánh và đối chiếu các thay đổi thư mục được mô tả trong fs.link thảo luận về những điều sau:



Không giống như liên kết cứng, liên kết tượng trưng không thay đổi độ dài khi tệp đích của chúng (trong trường hợp này target.txt) thay đổi độ dài. Ở đây chúng ta thấy cách thay đổi độ dài của mục tiêu từ 0 byte thành 6 byte không ảnh hưởng đến độ dài của bất kỳ liên kết tượng trưng bị ràng buộc nào:



fs.readlink (đường dẫn, gọi lại)

Liên kết tượng trưng đã cho tại con đường, trả về tên tệp của tệp được nhắm mục tiêu:

fs.readlink ('a.txt', function (err, targetFName) { console.log (targetFName); // target.txt

});

fs.realpath (đường dẫn, [bộ đệm], gọi lại)

Các fs.realpath (đường dẫn, [bộ đệm], gọi lại) phương pháp cố gắng tìm đường dẫn thực nộp tại con đường. Đây là một cách hữu ích để tìm đường dẫn tuyệt đối đến tệp, giải quyết các liên kết tượng trưng và thậm chí dọn dẹp các dấu gạch chéo không liên quan và các đường dẫn không đúng định dạng khác. Ví dụ:

fs.realpath ('file.txt', function (err, ResolutionPath) { console.log (ResolutionPath); // `/ real / path / to / file.txt` });

Hoặc, thậm chí:

fs.realpath('.////./file.txt', function(err, resolvedPath) {

// still `/real/path/to/file.txt`

});

Nếu một số đoạn đường dẫn cần giải quyết đã được biết, người ta có thể vượt qua bộ nhớ đệm trong số các đường dẫn được ánh xạ:

var cache = {'/ etc': '/ private / etc'};

fs.realpath ('/ etc / passwd', cache, function (err, ResolutionPath) { console.log (ResolutionPath); // `/ private / etc / passwd`

});

fs.unlink (đường dẫn, gọi lại)

Các fs.unlink (đường dẫn, gọi lại) phương pháp loại bỏ tệp tại con đường - tương đương với việc xóa một tệp. Lệnh gọi lại nhận một đối số, bất kỳ ngoại lệ nào được đưa ra trong cuộc gọi.

fs.rmdir (đường dẫn, gọi lại)

Các fs.rmdir (đường dẫn, gọi lại) phương pháp xóa thư mục tại con đường. Tương đương với việc xóa một thư mục.

Lưu ý rằng nếu thư mục không trống, điều này sẽ tạo ra một ngoại lệ. Lệnh gọi lại nhận một đối số, bất kỳ ngoại lệ nào được đưa ra trong cuộc gọi.

fs.mkdir (đường dẫn, [chế độ], gọi lại)

Các fs.mkdir (đường dẫn, [chế độ], gọi lại) phương pháp tạo một thư mục tại con đường. Để đặt chế độ của thư mục mới, hãy sử dụng bản đồ bit quyền được mô tả trong fs.chmod.

Lưu ý rằng nếu thư mục này đã tồn tại, một ngoại lệ sẽ được ném ra. Lệnh gọi lại nhận một đối số, bất kỳ ngoại lệ nào được đưa ra trong cuộc gọi.

fs.exists (đường dẫn, gọi lại)

Các fs.exists (đường dẫn, gọi lại) phương pháp kiểm tra xem tệp có tồn tại ở con đường. Lệnh gọi lại sẽ nhận được giá trị Boolean true hoặc false.

fs.fsync (fd, gọi lại)

Giữa thời điểm yêu cầu một số dữ liệu được ghi vào tệp được thực hiện và dữ liệu đó hoàn toàn tồn tại trên thiết bị lưu trữ, dữ liệu ứng viên tồn tại trong bộ đệm hệ thống lõi. Độ trễ này thường không liên quan nhưng trong một số trường hợp cực đoan, chẳng hạn như sự cố hệ thống, cần phải nhấn mạnh rằng tệp phản ánh trạng thái đã biết trên thiết bị lưu trữ ổn định.

fs.fsync sao chép tất cả dữ liệu trong lõi của tệp được tham chiếu bởi bộ mô tả tệp fd vào đĩa

(hoặc thiết bị lưu trữ khác). Lệnh gọi lại nhận một đối số, bất kỳ ngoại lệ nào được đưa ra trong cuộc gọi.

## 4.7. Tính đồng bộ

Thuận tiện, Node's tập tin mô-đun cung cấp các đối tác đồng bộ cho mỗi phương thức không đồng bộ mà chúng tôi đã đề cập, được biểu thị bằng hậu tố Đồng bộ hóa. Ví dụ: phiên bản đồng bộ của fs.mkdir Là fs.mkdirSync.

Một cuộc gọi đồng bộ cũng có thể trực tiếp trả về kết quả của nó, loại bỏ nhu cầu gọi lại. Trong khi trình diễn việc tạo máy chủ HTTPS trong Chương 3, Truyền dữ liệu qua các nút và máy khách, chúng tôi đã thấy cả một trường hợp sử dụng tốt cho mã đồng bộ và một ví dụ về chỉ định trực tiếp kết quả mà không cần gọi lại:

key: fs.readFileSync ('server-key.pem'), cert: fs.readFileSync ('server-cert.pem')

Chào! Node không thực thi nghiêm ngặt lập trình không đồng bộ? Không phải lúc nào mã chặn cũng sai? Tất cả các nhà phát triển được khuyến khích tuân thủ các thiết kế không chặn và bạn được khuyến khích tránh mã hóa đồng bộ — nếu gặp vấn đề ở nơi mà hoạt động đồng bộ dường như là giải pháp duy nhất, thì có thể vấn đề đã bị hiểu nhầm. Tuy nhiên, các trường hợp cạnh yêu cầu một đối tượng tệp hiện có đầy đủ trong bộ nhớ trước khi thực hiện các hướng dẫn tiếp theo (thao tác chặn) vẫn tồn tại. Node cung cấp cho nhà phát triển sức mạnh để phá vỡ truyền thống không đồng bộ nếu nó là giải pháp khả thi duy nhất (mà có lẽ không phải vậy!).

Một hoạt động đồng bộ mà các nhà phát triển thường xuyên sử dụng (có lẽ mà không nhận ra) là yêu cầu chỉ thị:

yêu cầu ('fs')

Cho đến khi phần phụ thuộc được nhắm mục tiêu bởi request được khởi tạo hoàn toàn, các hướng dẫn JavaScript tiếp theo sẽ không thực thi (tải tệp chặn vòng lặp sự kiện). Ryan Dahl đã đấu tranh với quyết định này để đưa các hoạt động đồng bộ (đặc biệt là các hoạt động tệp) vào Node, như anh ấy đã đề cập tại Google Tech Talk vào ngày 9 tháng 7 năm 2013 ( http: // www. youtube.com/watch?v=F6k8lTrAE2g):

Tôi nghĩ đây là một thỏa hiệp OK. Nó khiến tôi đau đớn trong nhiều tháng, khi đánh mất sự thuần khiết của việc có một hệ thống mô-đun không đồng bộ. Nhưng, tôi nghĩ nó ổn.

Và sau đó:

Nó đơn giản hóa mã rất nhiều để có thể chỉ cần gắn vào "yêu cầu, yêu cầu, yêu cầu" và không phải thực hiện gọi lại khi tải ... Tôi nghĩ đó là một thỏa hiệp tương đối ổn. […] Thực sự có hai phần trong chương trình của bạn: có giai đoạn tải và khởi động… và bạn không thực sự quan tâm xem nó chạy nhanh như thế nào… bạn sẽ tải các mô-đun và các thứ… giai đoạn thiết lập daemon của bạn, nói chung, là đồng bộ. Đó là khi bạn vào vòng lặp sự kiện của mình để phục vụ các yêu cầu, bạn cần phải hết sức cẩn thận về điều này. […] Tôi sẽ cung cấp cho mọi người I / O tệp đồng bộ. Nếu họ làm điều đó trong máy chủ… thì sẽ không quá khủng khiếp, phải không? Điều quan trọng là không bao giờ để họ làm I / O mạng đồng bộ.

Mã đồng bộ có lợi thế là dễ dự đoán, vì không có gì khác xảy ra cho đến khi hoàn thành hướng dẫn này. Khi khởi động một máy chủ, điều này hiếm khi xảy ra, Dahl gợi ý rằng một chút chắc chắn và đơn giản sẽ đi được một chặng đường dài. Ví dụ, việc tải các tệp cấu hình có thể có ý nghĩa khi khởi tạo máy chủ.

Đôi khi mong muốn sử dụng các lệnh đồng bộ trong phát triển Node chỉ đơn giản là một tiếng kêu cứu; một nhà phát triển bị choáng ngợp bởi các cấu trúc gọi lại lồng nhau sâu sắc. Nếu đã từng đối mặt với nỗi đau này, hãy thử một số thư viện callback-taming được đề cập trong

## 4.8. Di chuyển qua các thư mục

Hãy áp dụng những gì chúng ta đã học và tạo một trình lặp thư mục. Mục tiêu của dự án này là tạo một hàm chấp nhận một đường dẫn thư mục và trả về một đối tượng JSON phản ánh hệ thống phân cấp thư mục của tệp, các nút của nó bao gồm các đối tượng tệp. Chúng tôi cũng sẽ biến trình đi bộ thư mục của mình trở thành trình phân tích cú pháp dựa trên sự kiện mạnh mẽ hơn, phù hợp với triết lý Node.

Để di chuyển qua các thư mục lồng nhau, trước tiên người ta phải có khả năng đọc một thư mục. Thư viện hệ thống tệp của Node cung cấp fs.readdir lệnh cho mục đích này:

fs.readdir('.', function(err, files) {

console.log(files); // list of all files in current directory });

Nhớ rằng mọi thứ đều là một tệp, chúng ta sẽ cần phải làm nhiều việc hơn là chỉ đơn giản lấy một danh sách thư mục; chúng ta phải xác định loại của từng thành viên trong danh sách tệp của chúng ta. Bằng cách thêm fs.stat chúng tôi đã hoàn thành phần lớn logic:

(function(dir) {

fs.readdir(dir, function(err, list) { list.forEach(function(file) {

fs.stat(dir + "/" + file, function(err, stat) { if(stat.isDirectory()) {

return console.log("Found directory : " + file);

}

console.log("Found file : " + file);

});

});

});

})(".");

Hàm tự thực thi này nhận đối số đường dẫn thư mục ("."), Gấp danh sách thư mục đó thành một mảng tên tệp, tìm nạp một fs.Stats đối tượng cho từng thứ này và đưa ra quyết định dựa trên loại tệp được chỉ định (thư mục hoặc không phải thư mục) về việc cần làm tiếp theo. Tại thời điểm này, chúng tôi cũng có sẵn tên của tệp hiện tại và các thuộc tính của nó. Rõ ràng, chúng tôi đã lập bản đồ cho một thư mục.

Bây giờ chúng ta phải lập bản đồ các thư mục trong các thư mục, lưu trữ kết quả trong một đối tượng JSON phản ánh cây hệ thống tệp lồng nhau, mỗi lá trên cây là một đối tượng tệp. Chuyển một cách đệ quy các đường dẫn hàm đọc thư mục của chúng tôi đến các thư mục con và nối kết quả trả về dưới dạng các nhánh của đối tượng cuối cùng là bước tiếp theo:

var walk = function(dir, done) { var results = {};

fs.readdir(dir, function(err, list) { var pending = list.length; if(err || !pending) { return done(err, results);

}

list.forEach(function(file) { var dfile = dir + "/" + file; fs.stat(dfile, function(err, stat) { if(stat.isDirectory()) {

return walk(dfile, function(err, res) { results[file] = res;

!--pending && done(null, results);

});

}

results[file] = stat;

!--pending && done(null, results);

});

});

});

};

walk(".", function(err, res) {

console.log(require('util').inspect(res, {depth: null})); });

Chúng tôi tạo ra một đi bộ phương thức nhận đường dẫn thư mục và lệnh gọi lại nhận biểu đồ thư mục hoặc lỗi khi đi bộ đã hoàn thành, theo phong cách của Node. Không cần nhiều mã hóa để tạo một trình đi bộ trên cây tệp rất nhanh, không chặn, hoàn chỉnh với số liệu thống kê về tệp.

Bây giờ chúng ta hãy xuất bản các sự kiện bất cứ khi nào gặp phải một thư mục hoặc tệp, mang lại sự linh hoạt cho việc triển khai trong tương lai để xây dựng biểu diễn hệ thống tệp của riêng nó. Để làm điều này, chúng tôi sẽ sử dụng EventEmitter vật:

var walk = function(dir, done, emitter) {

...

emitter = emitter || new (require('events').EventEmitter);

... if(stat.isDirectory()) {

emitter.emit('directory', dfile, stat); return walk(dfile, function(err, res) { results[file] = res;

!--pending && done(null, results);

}, emitter);

}

emitter.emit('file', dfile, stat); results[file] = stat;

... return emitter;

}

walk("/usr/local", function(err, res) {

...

}).on("directory", function(path, stat) {

console.log("Directory: " + path + " - " + stat.size);

}).on("file", function(path, stat) {

console.log("File: " + path + " - " + stat.size);

});

// File: index.html – 1024

// File: readme.txt – 2048

// Directory: images - 106

// File images/logo.png - 4096 // ...

Bây giờ chúng ta biết cách khám phá và giải quyết các tệp, chúng ta có thể bắt đầu đọc và ghi vào chúng.

## 4.9. Đọc từ một tệp

Trong cuộc thảo luận của chúng tôi về trình mô tả tệp, chúng tôi đã đề cập đến một phương pháp mở tệp, tìm nạp trình mô tả tệp và cuối cùng là đẩy hoặc kéo dữ liệu thông qua tham chiếu đó. Đọc tệp là một hoạt động phổ biến. Đôi khi việc quản lý chính xác bộ đệm đọc có thể cần thiết và Node cho phép điều khiển từng byte. Trong các trường hợp khác, người ta chỉ muốn một luồng không rườm rà và dễ sử dụng.

## 4.10. Đọc từng byte

Các fs.read là cách cấp thấp nhất mà Node cung cấp để đọc tệp.

fs.read (fd, bộ đệm, độ lệch, độ dài, vị trí, gọi lại)

Các tệp bao gồm các byte được sắp xếp theo thứ tự và các byte này có thể được giải quyết bởi

Chức vụ, liên quan đến đầu của tệp (vị trí không [0]). Sau khi chúng tôi có một bộ mô tả tệp fd, chúng ta có thể bắt đầu đọc chiều dài số byte và chèn chúng vào Đệm vật đệm, chèn bắt đầu tại một bộ đệm nhất định bù lại. Ví dụ: để sao chép 8366 byte bắt đầu từ Chức vụ 309 tệp có thể đọc được fd thành

a đệm bắt đầu từ một bù lại trong số 100, chúng tôi sẽ sử dụng: fs.read (fd, bộ đệm, 100, 8366, 309, gọi lại).

Đoạn mã sau trình bày cách mở và đọc một tệp trong các phần 512 byte:

fs.open('path.js', 'r', function(err, fd) { fs.fstat(fd, function(err, stats) { var totalBytes = stats.size; var buffer = new Buffer(totalBytes); var bytesRead = 0;

// Each call to read should ensure that chunk size is

// within proper size ranges (not too small; not too large). var read = function(chunkSize) {

fs.read(fd, buffer, bytesRead, chunkSize, bytesRead, function(err, numBytes, bufRef) {

if((bytesRead += numBytes) < totalBytes) { return read(Math.min(512, totalBytes - bytesRead));

} fs.close(fd);

console.log("File read complete. Total bytes read: " + totalBytes);

// Note that the callback receives a reference to the

// accumulating buffer console.log(bufRef.toString());

});

}

read(Math.min(512, totalBytes));

}); });

Bộ đệm kết quả có thể được đặt ở nơi khác (bao gồm cả đối tượng phản hồi máy chủ). Nó cũng có thể được điều khiển bằng các phương pháp của Node Đệm đối tượng, chẳng hạn như chuyển đổi thành chuỗi UTF8 với buffer.toString ("utf8").

## 4.11. Tìm nạp toàn bộ tệp cùng một lúc

Thông thường, người ta chỉ cần tìm nạp toàn bộ tệp mà không cần bất kỳ nghi lễ hoặc kiểm soát tốt nào. Node cung cấp một phương thức tắt cho chính xác điều này.

fs.readFile (đường dẫn, [tùy chọn], gọi lại)

Tìm nạp dữ liệu có trong tệp con đường có thể được thực hiện trong một bước:

fs.readFile('/etc/passwd', function(err, fileData) { if(err) { throw err;

}

console.log(fileData);

// <Buffer 48 65 6C 6C 6F … >

});

Chúng tôi xem cách gọi lại nhận được một bộ đệm. Có thể mong muốn hơn khi nhận dữ liệu tệp trong một mã hóa chung, chẳng hạn như UTF8. Chúng tôi có thể chỉ định mã hóa của dữ liệu trả về, cũng như chế độ đọc, bằng cách sử dụng tùy chọn đối tượng, có hai thuộc tính có thể có:

mã hóa: Một chuỗi, chẳng hạn như utf8. Mặc định là null (không có mã hóa).

cờ: Chế độ tệp dưới dạng chuỗi. Mặc định là r.

Sửa đổi ví dụ trước:

fs.readFile('/etc/passwd', function(err, { encoding : "utf8" }, fileData) {

... console.log(fileData);

// "Hello ..."

});

## 4.12. Tạo một luồng có thể đọc được

Trong khi fs.readFile là một cách tuyệt vời, đơn giản để hoàn thành một nhiệm vụ chung,

nó có nhược điểm đáng kể là yêu cầu toàn bộ tệp phải được đọc vào bộ nhớ trước khi bất kỳ phần nào của tệp được gửi đến lệnh gọi lại. Đối với các tệp lớn, đặc biệt là các tệp lớn được truy cập thường xuyên (chẳng hạn như video), đây là vấn đề vì chúng tôi không thể dự đoán chính xác khối lượng dữ liệu mà chúng tôi sẽ lưu vào bộ đệm tại bất kỳ thời điểm nào. Điều này có nghĩa là dung lượng bộ nhớ có sẵn cho Node có thể bị thiếu hụt một cách không thể đoán trước, làm hỏng toàn bộ ứng dụng hoặc tệ hơn, khiến khách hàng của bạn tức giận khi khả năng phản hồi giảm và lỗi bắt đầu xuất hiện.

Trong chương trước, chúng ta đã tìm hiểu về các luồng dữ liệu và Suối vật. Trong khi các tệp được xử lý dễ dàng và tự nhiên bằng cách sử dụng các luồng có thể đọc được, Node cung cấp giao diện phát trực tuyến tệp chuyên dụng cung cấp phương tiện truyền tệp nhỏ gọn mà không cần xây dựng thêm, với tính linh hoạt hơn so với fs.readFile.

fs.createReadStream (đường dẫn, [tùy chọn])

Các fs.createReadStream (đường dẫn, [tùy chọn]) phương thức trả về một luồng có thể đọc được đối tượng cho tệp tại con đường. Sau đó, bạn có thể thực hiện các hoạt động luồng trên đối tượng được trả về, chẳng hạn như ống().

Lựa chọn tiếp theo đã khả thi:

cờ: Đối số chế độ tệp dưới dạng một chuỗi. Mặc định là r.

mã hóa: Một trong utf8, ascii, hoặc là cơ sở64. Mặc định không có mã hóa.

fd: Một người có thể đặt con đường thành null, thay vào đó chuyển lệnh gọi một bộ mô tả tệp.

chế độ: Biểu diễn hệ bát phân của chế độ tệp, mặc định là 0666.

đệmSize: Kích thước phân đoạn, tính bằng byte, của luồng đọc nội bộ. Mặc định là 64

\* 1024 byte. Bạn có thể đặt giá trị này thành bất kỳ số nào, nhưng việc cấp phát bộ nhớ được kiểm soát chặt chẽ bởi

hệ điều hành chủ, điều này có thể bỏ qua yêu cầu. Xem: https: // nhóm.

google.com/forum/?fromgroups#!topic/nodejs/p5FuU1oxbeY.

Tự động đóng: Có tự động đóng trình mô tả tệp hay không (a la fs.close).

Giá trị mặc định là true. Bạn có thể muốn đặt giá trị này thành false và đóng theo cách thủ công nếu bạn đang chia sẻ bộ mô tả tệp trên nhiều luồng, vì việc đóng bộ mô tả sẽ làm gián đoạn bất kỳ trình đọc nào khác.

khởi đầu: Bắt đầu đọc từ vị trí này. Mặc định là 0.

kết thúc: Ngừng đọc ở vị trí này. Mặc định là độ dài byte tệp.

Đọc từng dòng một tệp

Mặc dù đọc từng dòng tệp từng byte là đủ cho bất kỳ công việc phân tích cú pháp tệp nào, nhưng các tệp văn bản nói riêng thường được đọc từng dòng một cách hữu ích hơn, chẳng hạn như khi đọc tệp nhật ký. Chính xác hơn, bất kỳ luồng nào cũng có thể được hiểu theo các khối dữ liệu được phân tách bằng các ký tự dòng mới, thường là "\ r \ n" trên hệ thống UNIX. Node cung cấp một mô-đun gốc có các phương pháp đơn giản hóa việc truy cập vào các phần được phân tách bằng dòng mới trong các luồng dữ liệu.

Mô-đun Readline

Các Đường đọc mô-đun có một mục tiêu đơn giản nhưng mạnh mẽ, đó là làm cho việc đọc một luồng dữ liệu theo từng dòng dễ dàng hơn. Phần lớn giao diện của nó được thiết kế để làm cho việc nhắc dòng lệnh trở nên dễ dàng hơn, do đó các giao diện lấy thông tin đầu vào của người dùng sẽ dễ thiết kế hơn.

Hãy nhớ rằng Node được thiết kế cho I / O, rằng các hoạt động I / O thường liên quan đến việc di chuyển dữ liệu giữa các luồng có thể đọc và có thể ghi, và stdout và stdin là các giao diện luồng giống hệt với các luồng tệp được trả về bởi fs.createReadStream

và fs.createWriteStream, chúng ta sẽ xem xét cách mô-đun này có thể được sử dụng tương tự để nhắc các luồng tệp cho một dòng văn bản.

Các Đường đọc mô-đun sẽ được sử dụng trong suốt cuốn sách (và ở phần sau của chương này), cho các mục đích ngoài cuộc thảo luận hiện tại về đọc tệp, chẳng hạn như khi thiết kế giao diện dòng lệnh. Đối với những người thiếu kiên nhẫn, thêm thông tin về mô-đun này có thể được tìm thấy tại: http://nodejs.org/api/readline.html.

Để bắt đầu làm việc với Đường đọc một mô-đun phải tạo một giao diện xác định luồng đầu vào và luồng đầu ra. Các tùy chọn giao diện mặc định ưu tiên sử dụng làm giao diện đầu cuối. Các tùy chọn chúng tôi quan tâm là:

đầu vào: Cần thiết. Luồng có thể đọc được đang được lắng nghe.

đầu ra: Cần thiết. Luồng có thể ghi được ghi vào.

thiết bị đầu cuối: Đặt điều này thành true nếu cả hai luồng đầu vào và đầu ra phải được coi như một thiết bị đầu cuối Unix hoặc TTY (TeleTYpewriter). Đối với các tệp, bạn sẽ đặt giá trị này thành false.

Việc đọc các dòng của tệp được thực hiện dễ dàng thông qua hệ thống này. Ví dụ: giả sử một người có một tệp từ điển liệt kê các từ thông dụng bằng tiếng Anh, người ta có thể muốn đọc danh sách thành một mảng để xử lý:

var fs = require('fs'); var readline = require('readline');

var rl = readline.createInterface({ input: fs.createReadStream("dictionary.txt"),

terminal: false

}); var arr = []; rl.on("line", function(ln) { arr.push(ln.trim())

});

// aardvark

// abacus

// abaisance // ...

Lưu ý cách chúng tôi vô hiệu hóa hành vi TTY, tự xử lý các dòng mà không chuyển hướng đến một dòng đầu ra.

Như mong đợi với Node I / Omodule, chúng tôi đang làm việc với các sự kiện phát trực tiếp. Các sự kiện mà thính giả có thể quan tâm là:

hàng: Nhận dòng được đọc gần đây nhất, dưới dạng một chuỗi

tạm ngừng: Được gọi bất cứ khi nào luồng bị tạm dừng

sơ yếu lý lịch: Được gọi bất cứ khi nào một luồng được tiếp tục

đóng: Được gọi bất cứ khi nào luồng bị đóng

Ngoại trừ hàng, những tên sự kiện này phản ánh Đường đọc phương pháp, tạm dừng một luồng Readline.pause, sơ yếu lý lịch với Readline.resume, và đóng với Readline.close.

## 4.13. Ghi vào một tệp

Giống như khi đọc tệp, Node cung cấp một bộ sưu tập phong phú các công cụ để ghi vào tệp. Chúng ta sẽ xem cách Node giúp dễ dàng nhắm mục tiêu nội dung của tệp theo từng byte, vì nó chuyển các luồng dữ liệu liên tục vào một tệp có thể ghi.

## 4.14. Viết từng byte

Các fs.write là cách cấp thấp nhất mà Node cung cấp để ghi tệp. Phương pháp này cho phép chúng tôi kiểm soát chính xác nơi các byte sẽ được ghi vào một tệp.

fs.write (fd, bộ đệm, độ lệch, độ dài, vị trí, gọi lại)

Để viết tập hợp các byte giữa các vị trí 309 và 8675 ( chiều dài 8366) trong tổng số đệm đến tệp được tham chiếu bởi bộ mô tả tệp fd, chèn bắt đầu từ Chức vụ 100:

var buffer = new Buffer(8675); fs.open("index.html", "w", function(err, fd) {

fs.write(fd, buffer, 309, 8366, 100, function(err, writtenBytes, buffer) {

console.log("Wrote " + writtenBytes + " bytes to file");

// Wrote 8366 bytes to file

});

Lưu ý rằng đối với các tệp được mở trong append ( a) một số hệ điều hành có thể bỏ qua Chức vụ , luôn thêm dữ liệu vào cuối tệp. Ngoài ra, nó không an toàn khi gọi fs.write nhiều lần trên cùng một tệp mà không cần chờ gọi lại. Sử dụng fs.createWriteStream trong những trường hợp đó.

Với sự kiểm soát chính xác như vậy, chúng tôi có thể cấu trúc các tệp một cách thông minh. Trong ví dụ sau (có phần giả thiết), chúng tôi tạo một cơ sở dữ liệu dựa trên tệp chứa thông tin được lập chỉ mục về điểm số bóng chày trong sáu tháng cho một đội. Chúng tôi muốn có thể nhanh chóng tra cứu xem đội này thắng hay thua (hoặc không thi đấu) vào một ngày nhất định.

Vì một tháng có thể có nhiều nhất 31 ngày, chúng tôi có thể (ngẫu nhiên) tạo lưới dữ liệu 6 x 31 trong tệp này, đặt một trong ba giá trị vào mỗi ô lưới: L (thua), W (thắng), N (không trò chơi). Để giải trí, chúng tôi cũng tạo ra một CLI (Giao diện dòng lệnh) vào cơ sở dữ liệu của chúng tôi với một ngôn ngữ truy vấn cơ bản. Ví dụ này sẽ làm rõ cách fs.read, fs.write và Đệm các đối tượng được sử dụng để thao tác chính xác các byte trong tệp:

var fs = require('fs'); var readline = require('readline'); var cells = 186; // 6 x 31 var buffer = new Buffer(cells);

while(cells--) {

// 0, 1 or greater

var rand = Math.floor(Math.random() \* 3);

// 78 = "N", 87 = "W", 76 = "L"

buffer[cells] = rand === 0 ? 78 : rand === 1 ? 87 : 76;

}

fs.open("scores.txt", "r+", function(err, fd) { fs.write(fd, buffer, 0, buffer.length, 0, function(err, writtenBytes, buffer) { var rl = readline.createInterface({

input: process.stdin, output: process.stdout

});

var quest = function() {

rl.question("month/day:", function(index) {

if(!index) { return rl.close();

}

var md = index.split('/');

var pos = parseInt(md[0] -1) \* 31 + parseInt(md[1] -1); fs.read(fd, new Buffer(1), 0, 1, pos, function(err, br, buff)

{

var v = buff.toString();

console.log(v === "W" ? "Win!" : v === "L" ? "Loss..." : "No game"); quest();

});

});

}; quest();

}); });

Sau khi chạy, chúng tôi có thể chỉ cần nhập cặp tháng / ngày và nhanh chóng truy cập vào ô dữ liệu đó. Thêm vào kiểm tra giới hạn cho các giá trị đầu vào sẽ là một cải tiến đơn giản. Đẩy luồng tệp qua giao diện người dùng trực quan hóa có thể là một bài tập hay.

## 4.15. Viết các khối dữ liệu lớn

Đối với các hoạt động ghi đơn giản fs.write có thể quá mức cần thiết. Đôi khi tất cả những gì cần thiết là cách tạo một tệp mới với một số nội dung. Cũng phổ biến là nhu cầu nối dữ liệu vào cuối tệp, như người ta có thể làm trong hệ thống ghi nhật ký. Các fs.writeFile và fs.appendFile các phương pháp có thể giúp chúng tôi giải quyết các tình huống đó.

fs.writeFile (đường dẫn, dữ liệu, [tùy chọn], gọi lại)

Các fs.writeFile (đường dẫn, dữ liệu, [tùy chọn], gọi lại) phương pháp viết nội dung của dữ liệu vào tệp tại con đường. Các dữ liệu đối số có thể là một Đệm hoặc một chuỗi. Lựa chọn tiếp theo đã khả thi:

mã hóa: Mặc định là utf8. Nếu dữ liệu là bộ đệm, tùy chọn này bị bỏ qua.

chế độ: Biểu diễn hệ bát phân của chế độ tệp, mặc định là 0666.

cờ: Viết cờ, mặc định là w.

Cách sử dụng rất đơn giản:

fs.writeFile('test.txt', 'A string or Buffer of data', function(err) { if(err) {

return console.log(err);

}

// File has been written });

fs.appendFile (đường dẫn, dữ liệu, [tùy chọn], gọi lại)

Tương tự với fs.writeFile, ngoại trừ việc dữ liệu được thêm vào cuối tệp tại con đường. Ngoài ra, cờ tùy chọn mặc định thành a.

## 4.16. Tạo luồng có thể ghi

Nếu dữ liệu được ghi vào tệp đến từng phần (chẳng hạn như xảy ra khi tải tệp lên), hãy truyền trực tuyến dữ liệu đó qua WordsStream giao diện đối tượng cung cấp tính linh hoạt và hiệu quả hơn.

fs.createWriteStream (đường dẫn, [tùy chọn])

Các fs.createWriteStream (đường dẫn, [tùy chọn]) phương thức trả về một luồng có thể ghi

đối tượng cho tệp tại con đường.

Lựa chọn tiếp theo đã khả thi:

cờ: Đối số chế độ tệp dưới dạng một chuỗi. Mặc định là w.

mã hóa: Một trong utf8, ascii, hoặc là cơ sở64. Mặc định không có mã hóa.

chế độ: Biểu diễn hệ bát phân của chế độ tệp, mặc định là 0666.

khởi đầu: Phần bù, cho biết vị trí trong tệp nơi bắt đầu ghi.

Ví dụ: chương trình nhỏ này hoạt động như một trình xử lý văn bản đơn giản nhất thế giới, ghi tất cả đầu vào đầu cuối vào một tệp, cho đến khi thiết bị đầu cuối bị đóng:

var writer = fs.createWriteStream ("novel.txt", 'w'); process.stdin.pipe (người viết);

## 4.17. Cảnh báo

Các tác dụng phụ của việc mở bộ mô tả tệp và đọc từ nó là rất ít, do đó trong quá trình phát triển bình thường, người ta rất ít suy nghĩ về những gì đang thực sự xảy ra trong hệ thống. Thông thường, việc đọc tệp không thay đổi hình dạng, khối lượng hoặc tính khả dụng của dữ liệu.

Khi ghi vào tệp, một số mối quan tâm phải được giải quyết, chẳng hạn như:

Có đủ không gian lưu trữ có thể ghi không?

Quá trình khác có đồng thời truy cập tệp này không, hoặc thậm chí xóa nó?

Phải làm gì nếu thao tác ghi không thành công hoặc bị chấm dứt bất thường giữa dòng?

Chúng tôi đãđã thấy cờ chế độ ghi độc quyền ( wx) điều đó có thể giúp ích trong trường hợp có nhiều ghi các quy trình đồng thời cố gắng tạo một tệp. Các giải pháp đầy đủ cho tất cả các mối quan tâm mà người ta có thể phải đối mặt khi ghi vào tệp nói chung là khó tìm ra hoặc trình bày ngắn gọn. Node khuyến khích lập trình không đồng bộ. Tuy nhiên, đối với hệ thống tệp nói riêng, đôi khi cần phải lập trình đồng bộ, xác định. Bạn được khuyến khích lưu ý những vấn đề này và các vấn đề khác, đồng thời không chặn I / O bất cứ khi nào có thể.

Cung cấp tệp tĩnh

Bất kỳ ai sử dụng Node để tạo máy chủ web sẽ cần phản hồi các yêu cầu HTTP một cách thông minh. Yêu cầu AHTTP đến máy chủ web cho một tài nguyên mong đợi một số loại phản hồi. Máy chủ tệp tĩnh tệp cơ bản có thể trông giống như sau:

http.createServer(function(request, response) { if(request.method !== "GET") {

return response.end("Simple File Server only does GET");

} fs

.createReadStream(\_\_dirname + request.url)

.pipe(response);

}).listen(8000);

Dịch vụ máy chủ này GET các yêu cầu trên cổng 8000, hy vọng sẽ tìm thấy một tệp cục bộ tại một đường dẫn tương đối tương đương với phân đoạn đường dẫn URL. Chúng tôi thấy Node giúp chúng tôi truyền trực tuyến dữ liệu tệp cục bộ dễ dàng như thế nào, chỉ đơn giản là đường dẫn ReadableStream thành một WordsStream đại diện cho một kết nối ổ cắm máy khách. Đây là một số lượng lớn các chức năng được thực hiện một cách an toàn trong một số ít dòng.

Cuối cùng, nhiều thứ khác sẽ được thêm vào, chẳng hạn như xử lý các quy trình cho các phương thức HTTP tiêu chuẩn, xử lý lỗi và yêu cầu không đúng định dạng, đặt tiêu đề phù hợp, quản lý yêu cầu biểu tượng yêu thích, v.v.

Hãy xây dựng một máy chủ tệp hợp lý hữu ích với Node, một máy chủ sẽ phản hồi các yêu cầu HTTP bằng cách phát trực tiếp tài nguyên và sẽ tôn trọng các yêu cầu bộ nhớ đệm. Trong quá trình này, chúng tôi sẽ đề cập đến cách quản lý chuyển hướng nội dung. Ở phần sau của chương này, chúng ta cũng sẽ xem xét việc triển khai tải lên tệp. Lưu ý rằng một máy chủ web tuân thủ đầy đủ tất cả các tính năng của HTTP là một con thú phức tạp, vì vậy những gì chúng ta đang tạo nên được coi là một khởi đầu tốt chứ không phải kết thúc.

## 4.18. Chuyển hướng yêu cầu

Đôi khi khách hàng sẽ cố gắng ĐƯỢC URL không chính xác hoặc không đầy đủ theo một cách nào đó, tài nguyên có thể đã bị di chuyển hoặc có nhiều cách tốt hơn để đưa ra yêu cầu tương tự. Lần khác, a BÀI ĐĂNG có thể tạo một tài nguyên mới tại một vị trí mới mà khách hàng không thể biết, yêu cầu một số thông tin tiêu đề phản hồi trỏ đến URI mới được tạo. Hãy xem xét hai trường hợp chuyển hướng phổ biến mà ai đó triển khai máy chủ tệp tĩnh với Node có thể gặp phải.

Hai tiêu đề phản hồi là cơ bản để chuyển hướng:

Vị trí: Điều này cho biết sự chuyển hướng đến một vị trí có thể tìm thấy nội dung đã nói

Nội dung-Vị trí: Điều này có nghĩa là để chỉ ra URL nơi người yêu cầu sẽ tìm thấy vị trí ban đầu của thực thể nằm trong phần nội dung phản hồi

Ngoài ra, có hai trường hợp sử dụng cụ thể cho các tiêu đề này:

Cung cấp thông tin về vị trí của tài nguyên mới được tạo để đáp ứng BÀI ĐĂNG.

Để thông báo cho khách hàng về vị trí thay thế cho tài nguyên được yêu cầu để đáp ứng ĐƯỢC.

Có thể có nhiều cặp Vị trí và Nội dung-Vị trí

tiêu đề có mã trạng thái HTTP, 3xx ( chuyển hướng) được thiết lập cụ thể. Trên thực tế, các tiêu đề này thậm chí có thể xuất hiện cùng nhau trong cùng một phản hồi. Người dùng được khuyến khích đọc các phần có liên quan của thông số kỹ thuật HTTP / 1.1, vì chỉ một nhóm nhỏ các trường hợp phổ biến được thảo luận ở đây.

## 4.19. Vị trí

Đáp lại một BÀI ĐĂNG với một 201 mã trạng thái cho biết một tài nguyên mới đã được tạo URI của nó được gán cho Vị trí và ứng dụng khách có thể tiếp tục và sử dụng URI đó trong tương lai. Lưu ý rằng khách hàng quyết định xem có tìm nạp tài nguyên này hay không và khi nào. Như vậy, nói một cách chính xác, đây không phải là một chuyển hướng.

Ví dụ: hệ thống có thể tạo tài khoản mới bằng cách đăng thông tin người dùng mới lên máy chủ, mong đợi nhận được vị trí của trang người dùng mới:

POST /path/addUser HTTP/1.1 Content-Type: application/x-www-form-urlencoded name=John&group=friends …

Status: 201

Location: http://website.com/users/john.html

Tương tự, trong trường hợp yêu cầu tạo tài nguyên đã được chấp nhận nhưng chưa được thực hiện, máy chủ sẽ chỉ ra trạng thái 202. Đây sẽ là trường hợp trong ví dụ trước nếu việc tạo bản ghi người dùng mới đã được ủy quyền cho hàng đợi công nhân, điều này có thể vào một thời điểm nào đó trong tương lai tạo bản ghi tại một Vị trí.

Chúng ta sẽ thấy một triển khai thực tế chứng minh cách sử dụng này ở phần sau của chương, khi chúng ta thảo luận về tải lên tệp.

## 4.20. Nội dung-Vị trí

Khi một ĐƯỢC được tạo cho một tài nguyên có nhiều đại diện và chúng có thể được tìm thấy tại các vị trí tài nguyên riêng biệt, nội dung-vị trí tiêu đề cho thực thể cụ thể phải được trả lại.

Ví dụ: thương lượng định dạng nội dung là một ứng cử viên tốt cho Nội dung-Vị trí sự điều khiển. Một người có thể quan tâm đến việc truy xuất tất cả các bài đăng trên blog trong một tháng nhất định, có thể có sẵn tại một URL như http://example.com/september/. NHẬN yêu cầu với Chấp nhận tiêu đề của ứng dụng / json sẽ nhận được phản hồi ở định dạng JSON. Một yêu cầu cho XML sẽ nhận được biểu diễn đó.

Nếu một cơ chế bộ nhớ đệm đang được sử dụng, các tài nguyên đó có thể có thay thế vĩnh viễn địa điểm, chẳng hạn như http://example.com/cache/september.json hoặc là http: // example.com/cache/september.xml. Một người sẽ gửi thông tin vị trí bổ sung này qua Nội dung-Vị trí, trong một đối tượng phản hồi giống như:

Status: 200

Content-Type: application/json

Content-Location: http://blogs.com/cache/allArticles.json

… JSON entity body

Trong trường hợp URL được yêu cầu đã được di chuyển, vĩnh viễn hoặc tạm thời,

3xx nhóm mã trạng thái có thể được sử dụng với Nội dung-Vị trí để chỉ trạng thái này. Ví dụ: để chuyển hướng một yêu cầu đến một URL đã được di chuyển vĩnh viễn, người ta phải gửi mã 301:

function requestHandler(request,response) { var newPath = "/thedroids.html"; response.writeHead(301, {

'Content-Location': newPath

});

response.end(); }

## 4.21. Triển khai bộ nhớ đệm tài nguyên

Một nguyên tắc chung, không bao giờ sử dụng các nguồn lực cung cấp thông tin không liên quan cho khách hàng. Đối với máy chủ HTTP, việc gửi lại các tệp mà máy khách đã sở hữu là chi phí I / O không cần thiết, chính xác là cách triển khai máy chủ Node sai, làm tăng độ trễ cũng như ảnh hưởng tài chính khi phải trả cho băng thông bị chiếm dụng.

Các trình duyệt duy trì bộ nhớ cache của các tệp mà họ đã tìm nạp và ETag ( Thẻ thực thể) xác định các tệp này. ETag là một tiêu đề phản hồi do máy chủ gửi để xác định duy nhất các thực thể mà chúng đang trả về, chẳng hạn như tệp. Khi một tệp thay đổi trên một máy chủ, máy chủ đó sẽ gửi một ETag khác cho tệp đó, cho phép các máy khách theo dõi các thay đổi của tệp.

Khi khách hàng đưa ra yêu cầu đối với máy chủ đối với tài nguyên có trong bộ đệm ẩn của máy khách đó, yêu cầu đó sẽ chứa tiêu đề If-None-Match được đặt thành giá trị của ETag được liên kết với tài nguyên đã lưu trong bộ nhớ cache. Tiêu đề If-None-Match có thể chứa một hoặc nhiều ETags:

If-None-Match: "686897696a7c876b7e"

Nếu-Không-Khớp: "686897696a7c876b7e", "923892329b4c796e2e"

Máy chủ hiểu tiêu đề này là chỉ trả về toàn bộ nội dung thực thể của tài nguyên được yêu cầu nếu không thẻ ETags nào đã gửi khớp với thẻ thực thể tài nguyên hiện tại. Nếu một trong các thẻ ETags đã gửi khớp với thẻ thực thể hiện tại, máy chủ sẽ phản hồi bằng trạng thái 304 (không được sửa đổi), điều này sẽ dẫn đến việc trình duyệt tìm nạp tài nguyên từ bộ nhớ cache bên trong của nó.

Giả sử rằng chúng ta có một fs.Stats đối tượng có sẵn, việc quản lý các điều khiển bộ nhớ cache trên một tài nguyên có thể được thực hiện dễ dàng với Node:

var etag = crypto.createHash('md5').update(stat.size + stat.mtime).

digest('hex');

if(request.headers['if-none-match'] === etag) { response.statusCode = 304; return response.end();

} else {

// stream the requested resource

}

Chúng tôi tạo ra một etag cho tệp hiện tại bằng cách tạo MD5 có kích thước tệp hiện tại và thời gian sửa đổi lần cuối của nó và khớp với tiêu đề If-None-Match đã gửi. Nếu cả hai không khớp nhau, biểu diễn tài nguyên đã thay đổi và phiên bản mới phải được gửi lại cho khách hàng yêu cầu. Lưu ý rằng thuật toán cụ thể mà người ta nên sử dụng để tạo etag không được chỉ định chính thức. Kỹ thuật ví dụ sẽ hoạt động tốt cho hầu hết các mục đích.

Chào! Thế còn Sửa đổi lần cuối và Nếu-Không sửa đổi-Kể từ?

Đây là những tiêu đề tốt và cũng hữu ích trong trường hợp lưu tệp vào bộ nhớ đệm. Thật vậy, người ta nên đặt Sửa đổi lần cuối tiêu đề nếu có thể khi phản hồi các yêu cầu thực thể. Các kỹ thuật mà chúng tôi đang mô tả ở đây bằng cách sử dụng ETag sẽ hoạt động tương tự với các thẻ này và trên thực tế, việc sử dụng cả Etags và các thẻ khác này được khuyến khích. Để biết thêm thông tin, tham khảo ý kiến: http://www.w3.org/Protocols/rfc2616/rfc2616sec13.html # sec13.3.4.

## 4.22. Xử lý tải lên tệp

Có vẻ như bất kỳ ai đọc câu này đều đã có ít nhất một lần trải nghiệm việc tải tệp từ máy khách lên máy chủ. Một số thậm chí có thể đã triển khai dịch vụ tải tệp lên, một máy chủ sẽ nhận và làm điều gì đó hữu ích với luồng dữ liệu nhiều phần. Trong các môi trường phát triển phổ biến, nhiệm vụ này đã được thực hiện rất dễ dàng. Ví dụ: trong môi trường PHP, dữ liệu đã tải lên được xử lý tự động và cung cấp trên toàn cầu, được phân tích cú pháp gọn gàng và đóng gói thành một mảng tệp hoặc giá trị trường biểu mẫu mà nhà phát triển không cần viết một dòng mã nào.

Thật không may, Node để lại việc triển khai xử lý tải lên tệp cho nhà phát triển, một công việc khó khăn mà nhiều nhà phát triển có thể không thể hoàn thành thành công hoặc an toàn.

May mắn thay, Felix Geisendorfer đã tạo ra Ghê gớm mô-đun, một trong những đóng góp ban đầu quan trọng nhất cho dự án Node. Mô-đun cấp doanh nghiệp được triển khai rộng rãi với phạm vi thử nghiệm rộng rãi, nó không chỉ giúp xử lý tệp tải lên trong tích tắc mà còn có thể được sử dụng như một công cụ hoàn chỉnh để xử lý việc gửi biểu mẫu. Chúng tôi sẽ sử dụng thư viện này để thêm khả năng tải tệp lên máy chủ tệp của chúng tôi.

Để biết thêm thông tin về cách tải lên tệp HTTP được thiết kế và các vấn đề triển khai phức tạp mà nhà phát triển phải khắc phục, hãy tham khảo đặc tả dữ liệu đa phần / biểu mẫu

( http://www.w3.org/TR/html401/interact/forms. html # h-17.13.4.2) và Geisendorfer phân tích về cách Ghê gớm được hình thành và phát triển ( http: //debuggable.com/posts/parsing-file-uploads-at500-mb-s-with-node-js: 4c03862e-351c-4faa-bb674365cbdd56cb).

Đầu tiên, cài đặt ghê gớm qua npm:

npm cài đặt đáng gờm

Bây giờ bạn có thể yêu cầu nó: var ghê gớm = request ('ghê gớm');

Chúng tôi sẽ giả định rằng các tệp tải lên sẽ được đăng lên máy chủ của chúng tôi theo đường dẫn / uploads /, và tải lên đến thông qua một biểu mẫu HTML trông giống như sau:

<form action="/uploads" enctype="multipart/form-data" method="post">

Title: <input type="text" name="title"><br />

<input type="file" name="upload" multiple="multiple"><br />

<input type="submit" value="Upload">

</form>

Biểu mẫu này sẽ cho phép khách hàng viết một số loại tiêu đề cho tải lên và chọn một (hoặc nhiều) tệp để tải lên. Tại thời điểm này, trách nhiệm duy nhất của chúng tôi trên máy chủ của chúng tôi là phát hiện đúng cách khi BÀI ĐĂNG yêu cầu đã được thực hiện và chuyển đối tượng yêu cầu có liên quan đến Form.

Chúng tôi sẽ không đề cập đến mọi phần của thiết kế API đáng gờm toàn diện, mà tập trung vào chìa khóa BÀI

ĐĂNG các sự kiện mà thư viện trưng bày. Như mở rộng đáng kể EventEmitter, chúng tôi sử dụng on (eventName, callback) định dạng để bắt dữ liệu tệp, dữ liệu trường và các sự kiện kết thúc, gửi phản hồi đến máy khách mô tả những gì máy chủ đã xử lý thành công:

http.createServer(function(request, response) { var rm = request.method.toLowerCase(); if(request.url === '/uploads' && rm === 'post') { var form = new formidable.IncomingForm(); form.uploadDir = process.cwd();

var resp = ""; form

.on("file", function(field, File) { resp += "File : " + File.name + "<br />";

})

.on("field", function(field, value) { resp += field + " : " + value + "<br />";

})

.on("end", function() {

response.writeHead(200, {'content-type': 'text/html'});

response.end(resp);

})

.parse(request); return;

}

}).listen(8000);

Chúng tôi thấy ở đây như thế nào ghê gớm cá thể nhận được một http.Incoming đối tượng thông qua nó phân tích cú pháp và cách đặt đường dẫn ghi cho các tệp đến bằng cách sử dụng uploadDir thuộc tính của cá thể đó. Ví dụ đặt thư mục này thành thư mục cục bộ. Việc triển khai thực có thể sẽ nhắm mục tiêu đến một thư mục tải lên chuyên dụng hoặc thậm chí hướng tệp đã nhận đến một dịch vụ lưu trữ, nhận lại vị trí lưu trữ cuối cùng (có thể nhận nó qua HTTP và Vị trí tiêu đề…).

Cũng cần lưu ý cách gọi lại sự kiện tệp nhận được Tập tin đối tượng dưới dạng đối số thứ hai, chứa thông tin tệp quan trọng bao gồm:

kích thước: Kích thước của tệp đã tải lên, tính bằng byte

con đường: Vị trí hiện tại của tệp đã tải lên trên hệ thống tệp cục bộ, chẳng hạn như như / tmp / bdf746a445577332e38be7cde3a98fb3

Tên: Tên ban đầu của tệp khi nó tồn tại trên hệ thống tệp khách, chẳng hạn như như lolcats.jpg

kiểu: Loại kịch bản của tệp, chẳng hạn như hình ảnh / png

Trong một vài dòng mã, chúng tôi đã triển khai một lượng đáng kể BÀI ĐĂNG quản lý dữ liệu. Formced cũng cung cấp các công cụ để xử lý các chỉ báo tiến trình, xử lý các lỗi mạng, v.v. mà người đọc có thể tìm hiểu bằng cách truy cập https://github.com/felixge/node-formosystem.

## 4.23. Để tất cả chúng cùng nhau

Nhắc lại cuộc thảo luận của chúng ta về xử lý biểu tượng yêu thích từ chương trước và thêm những gì chúng ta đã học về bộ nhớ đệm tệp và tải tệp lên, giờ đây chúng ta có thể xây dựng một xử lý máy chủ tệp đơn giản ĐƯỢC và BÀI ĐĂNG yêu cầu:

http.createServer(function(request, response) { var rm = request.method.toLowerCase(); if(rm === "post") {

var form = new formidable.IncomingForm(); form.uploadDir = process.cwd(); form

.on("file", function(field, file) {

// process files

})

.on("field", function(field, value) {

// process POSTED field data

})

.on("end", function() { response.end("Received");

})

.parse(request); return;

}

// We can only handle GET requests at this point if(rm !== "get") {

return response.end("Unsupported Method");

}

var filename = \_\_dirname + request.url; fs.stat(filename, function(err, stat) { if(err) {

response.statusCode = err.errno === 34 ? 404 : 500; return response.end()

}

var etag = crypto.createHash('md5').update(stat.size + stat.

mtime).digest('hex');

response.setHeader('Last-Modified', stat.mtime); if(request.headers['if-none-match'] === etag) { response.statusCode = 304; return response.end(); }

response.setHeader('Content-Length', stat.size); response.setHeader('ETag', etag); response.statusCode = 200;

fs.createReadStream(filename).pipe(response);

});

}).listen(8000);

Lưu ý mã trạng thái 404 (không tìm thấy) và 500 (lỗi máy chủ nội bộ).

Thời lượng nội dung được đo bằng byte, không phải ký tự. Thông thường, dữ liệu của bạn sẽ ở dạng ký tự byte đơn (hello dài năm byte), nhưng không phải lúc nào cũng vậy. Nếu bạn đang xác định độ dài của bộ đệm luồng, hãy sử dụng Buffer.byteLength.

## Tóm lược

Trong chương này, chúng ta đã thấy cách API của Node là một bản đồ toàn diện cho các ràng buộc hệ thống tệp gốc, thể hiện đầy đủ các chức năng cho nhà phát triển trong khi yêu cầu rất ít mã hoặc độ phức tạp. Ngoài ra, chúng tôi đã thấy cách các tệp được gói dễ dàng vào Suối đối tượng và cách thức nhất quán này với phần còn lại của thiết kế Node đơn giản hóa các tương tác giữa các loại I / O khác nhau, chẳng hạn như giữa dữ liệu mạng và tệp.

Chúng tôi cũng đã học được vài điều về cách xây dựng máy chủ với Node có thể đáp ứng kỳ vọng của khách hàng thông thường, dễ dàng triển khai tải lên tệp và bộ nhớ đệm tài nguyên. Sau khi đề cập đến các tính năng chính của Node, đã đến lúc sử dụng các kỹ thuật này để xây dựng các ứng dụng lớn hơn có thể xử lý hàng nghìn máy khách.

# CHƯƠNG 5

# Quản lý đồng thời nhiều kết nối

Nếu mọi người cùng đỡ lấy bầu trời, thì một người không mệt mỏi.

Châm ngôn Tshi

Duy trì mức thông lượng cao trong khi quản lý hàng nghìn giao dịch khách hàng đồng thời trong môi trường không thể đoán trước và "bùng nổ" của phần mềm nối mạng là điều mà các nhà phát triển mong đợi đối với việc triển khai Node của họ. Với lịch sử của các giải pháp thất bại và không phổ biến, bài toán đồng thời thậm chí còn được gán từ chữ số riêng của nó: "Bài toán C10K". Phần mềm mạng có thể tự tin phục vụ 10.000 máy khách đồng thời nên được thiết kế như thế nào?

Câu hỏi làm thế nào để xây dựng tốt nhất các hệ thống đồng thời cao đã gây ra nhiều lý thuyết trong vài thập kỷ qua, với cuộc tranh luận chủ yếu giữa hai lựa chọn thay thế, chủ đề và sự kiện:

Phân luồng cho phép các lập trình viên viết mã đường thẳng và dựa vào hệ điều hành để tính toán chồng chéo và I / O bằng cách chuyển đổi giữa các luồng một cách rõ ràng. Giải pháp thay thế, các sự kiện, cho phép các lập trình viên quản lý đồng thời một cách rõ ràng bằng cách cấu trúc mã như một trình xử lý đơn luồng phản ứng với các sự kiện (chẳng hạn như hoàn thành I / O không chặn, thông báo dành riêng cho ứng dụng hoặc sự kiện hẹn giờ).

"Khung thiết kế cho các hệ thống đồng thời cao" (Welsh, Gribble,Brewer & Culler, 2000), tr. 2. http://www.eecs.harvard.edu/~mdw/

paper / event.pdf

Hai điểm quan trọng được đưa ra trong phần trích dẫn trước:

Các nhà phát triển thích viết có cấu trúc mã (đường thẳng; đơn luồng) ẩn sự phức tạp của nhiều hoạt động đồng thời nếu có thể

## 5.1. Hiệu quả I / O là yếu tố cân nhắc chính của các ứng dụng đồng thời cao

Cho đến gần đây, ngôn ngữ lập trình và các khuôn khổ liên quan không (nhất thiết) được tối ưu hóa cho phần mềm thực thi trên các nút trong mạng phân tán hoặc thậm chí trên các bộ xử lý. Các thuật toán được mong đợi là xác định; dữ liệu được ghi vào cơ sở dữ liệu sẽ ngay lập tức có sẵn để đọc. Trong thời đại cơ sở dữ liệu nhất quán và luồng điều khiển không đồng bộ này, các nhà phát triển không còn có thể mong đợi biết trạng thái chính xác của một ứng dụng tại bất kỳ thời điểm nào.

một thách thức đôi khi làm đau đầu các kiến trúc sư của các hệ thống đồng thời cao.

Như chúng ta đã học trong Chương 2, Tìm hiểu về lập trình theo hướng sự kiện không đồng bộ,

Thiết kế của Node cố gắng kết hợp các ưu điểm của cả luồng và sự kiện, phục vụ tất cả các máy khách trên một luồng duy nhất (một vòng lặp sự kiện bao quanh thời gian chạy JavaScript) trong khi ủy thác công việc chặn (I / O) cho một nhóm luồng được tối ưu hóa để thông báo cho luồng chính của thay đổi trạng thái thông qua hệ thống thông báo sự kiện.

Hãy suy nghĩ rõ ràng về cách triển khai máy chủ HTTP sau, đang chạy trên một CPU duy nhất, đang phản hồi từng yêu cầu của khách hàng bằng cách gói một hàm gọi lại trong ngữ cảnh của yêu cầu và đẩy ngữ cảnh thực thi đó vào một ngăn xếp liên tục được làm trống và xây dựng lại trong một luồng duy nhất liên kết với một vòng lặp sự kiện:

require('http').createServer(function(req, res) { res.writeHead(200, {'Content-Type': 'text/plain'}); res.end('Hello client from' + req.connection.remoteAddress); console.log(req); }).listen(8000);

Sơ đồ:

Quy trình nút

ĐĂNG NHẬP SỰ KIỆN V8

chuỗi thực thi

gotFile ()

response.end ()

gotData ()

Yêu cầu

Yêu cầu

response.end ()

Khách hàng

LIBUV

TCP 1

TCP 2

TCP 3

TCP 4

. . .

LIBUV

Khách hàng

Tập tin

Khách hàng

DB

Khách hàng

Mặt khác, một máy chủ như Apache quay một chuỗi cho mỗi yêu cầu của khách hàng:

Quy trình Apache 2

CÁ NHÂN

(64

MB mỗi

)

DB

Kết nối

Thời gian chạy

Yêu cầu

Người xử lý

DB

Kết nối

Thời gian chạy

Yêu cầu

Người xử lý

DB

Kết nối

Thời gian chạy

Yêu cầu

Người xử lý

DB

Kết nối

Thời gian chạy

Yêu cầu

Người xử lý

Khách hàng

Khách hàng

Khách hàng

Khách hàng

Hai cách tiếp cận này rất khác nhau. Tuyên bố ngầm trong thiết kế của Node là thế này: dễ lý giải hơn về phần mềm có tính đồng thời cao khi luồng chương trình được tổ chức dọc theo một luồng duy nhất và việc giảm độ trễ I / O làm tăng số lượng máy khách đồng thời có thể được hỗ trợ ngay cả trong một- mô hình thực thi phân luồng. Yêu cầu thứ hai sẽ được kiểm tra sau, nhưng bây giờ hãy xem việc xây dựng các quy trình cơ bản có quy mô tự nhiên dễ dàng như thế nào.

Chúng tôi sẽ trình bày cách theo dõi và quản lý các mối quan hệ giữa các quy trình đồng thời bằng cách sử dụng Node, đặc biệt là các quy trình phục vụ nhiều khách hàng đồng thời. Mục tiêu của chúng tôi là thiết lập hiểu biết cơ bản về cách tiểu bang nên được mô hình hóa trong máy chủ Node hoặc các quy trình khác. Làm thế nào mà một mạng xã hội trực tuyến lớn phục vụ bạn thông tin tùy chỉnh phù hợp với tình bạn hoặc sở thích của bạn? Làm thế nào để giỏ hàng của bạn được duy trì qua nhiều phiên mua sắm mà không bị biến mất, thậm chí còn chứa các đề xuất dựa trên lịch sử mua hàng của bạn? Làm thế nào để một khách hàng duy nhất có thể tương tác với các khách hàng khác?

## 5.2. Hiểu đồng thời

Tất cả chúng ta đều đồng ý rằng có những sự kiện bất ngờ trên thế giới và nhiều sự kiện trong số đó xảy ra chính xác cùng một lúc. Cũng rõ ràng rằng trạng thái của bất kỳ hệ thống nhất định nào có thể bao gồm bất kỳ số lượng trạng thái con nào, trong đó hệ quả đầy đủ của những thay đổi trạng thái dù là nhỏ cũng khó dự đoán — sức mạnh của cánh bướm đủ để đánh đầu một hệ thống lớn hơn nhiều vào một trạng thái thay thế. Và chúng ta cũng biết rằng khối lượng và hình dạng của một hệ thống, theo thời gian, thay đổi theo những cách khó dự đoán.

Trong luận án PHD của mình "Cơ sở ngữ nghĩa diễn viên", được viết vào năm 1981, William Clinger đề xuất rằng công việc của anh ấy là:

. . . được thúc đẩy bởi triển vọng của các máy tính toán song song cao bao gồm hàng chục, hàng trăm hoặc thậm chí hàng nghìn bộ vi xử lý độc lập, mỗi bộ đều có bộ nhớ cục bộ và bộ xử lý truyền thông riêng, giao tiếp qua mạng truyền thông hiệu suất cao.

Hóa ra, Clinger đang làm gì đó. Đồng thời là một thuộc tính của các hệ thống bao gồm nhiều hoạt động thực thi đồng thời và phần mềm mạng mà chúng ta đang xây dựng giống với phần mềm mà ông đã hình dung, chỉ lớn hơn nhiều, trong đó "hàng trăm hoặc thậm chí hàng nghìn" là giới hạn thấp hơn, không phải cao hơn.

Node giúp truy cập đồng thời, đồng thời mở rộng quy mô trên nhiều lõi, nhiều quy trình và nhiều máy. Điều quan trọng cần lưu ý là Node coi trọng tính đơn giản và nhất quán của các chương trình cũng như nhanh nhất giải pháp, áp dụng và thực thi I / O không chặn trong nỗ lực mang lại tính đồng thời cao thông qua các giao diện được thiết kế tốt và có thể dự đoán được. Đây là ý của Dahl khi anh ấy nói, "Mục tiêu của Node là cung cấp dễ dàng cách xây dựng có thể mở rộng các chương trình mạng ”.

## 5.3. Thật hạnh phúc, nó cũng chỉ ra rằng Node rất nhanh.

Đồng thời không phải là song song

Một vấn đề có thể được giải quyết bằng cách chia một số vấn đề thành các vấn đề nhỏ hơn, rải các vấn đề nhỏ hơn đó cho một nhóm người hoặc công nhân có sẵn để làm việc song song và cung cấp đồng thời các kết quả song song.

Nhiều quy trình giải quyết đồng thời một phần của một vấn đề toán học đơn lẻ là một ví dụ về tính song song.

Rob Pike, hacker thuật sĩ chung và người đồng phát minh ra ngôn ngữ lập trình Go của Google định nghĩa đồng thời theo cách này:

Tương tranh là một cách để cấu trúc một thứ để bạn có thể, có thể, sử dụng song song để thực hiện công việc tốt hơn. Nhưng song song không phải là mục tiêu của sự đồng thời; mục tiêu của concurrency là một cấu trúc tốt.

Các khuôn khổ phát triển ứng dụng đồng thời cao thành công cung cấp vốn từ vựng đơn giản và biểu đạt để mô tả các hệ thống như vậy.

Thiết kế của Node gợi ý rằng việc đạt được mục tiêu chính - cung cấp một cách dễ dàng để xây dựng các chương trình mạng có thể mở rộng - bao gồm việc đơn giản hóa cách thức cấu trúc và cấu trúc thứ tự thực thi của các quy trình cùng tồn tại. Node giúp một nhà phát triển gặp khó khăn với một chương trình có nhiều thứ xảy ra cùng một lúc (chẳng hạn như phục vụ nhiều khách hàng đồng thời) để tổ chức mã của họ tốt hơn.

Điều này không có nghĩa là Node được thiết kế để thừa nhận hiệu quả nhằm duy trì các giao diện đơn giản - khác xa với nó. Thay vào đó, ý tưởng là chuyển trách nhiệm thực hiện xử lý song song hiệu quả ra khỏi nhà phát triển và sang thiết kế cốt lõi của hệ thống, để nhà phát triển tự do cấu trúc đồng thời thông qua một hệ thống gọi lại đơn giản và có thể đoán trước, an toàn trước các bế tắc và các bẫy khác.

Sự đơn giản phù hợp của Node xuất hiện vào thời điểm tốt, khi mạng xã hội và cộng đồng phát triển cùng với dữ liệu của thế giới. Các hệ thống đang được mở rộng đến những kích thước mà ít ai có thể dự đoán được. Đây là thời điểm thích hợp cho những suy nghĩ mới, chẳng hạn như cách mô tả và thiết kế các hệ thống này cũng như cách chúng đưa ra yêu cầu và phản hồi lẫn nhau.

## 5.4. Yêu cầu định tuyến

HTTP là một giao thức truyền dữ liệu được xây dựng dựa trên mô hình yêu cầu / phản hồi. Sử dụng giao thức này, nhiều người trong chúng ta thông báo tình trạng hiện tại của mình cho bạn bè, mua quà cho gia đình hoặc thảo luận về một dự án qua e-mail với đồng nghiệp. Một số lượng đáng kinh ngạc người đã phụ thuộc vào giao thức Internet nền tảng này.

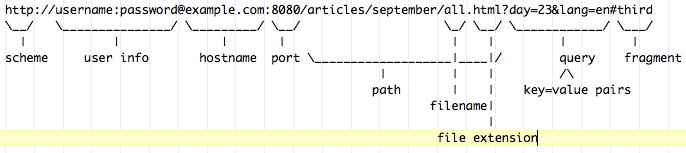
Thông thường, một ứng dụng khách trình duyệt sẽ đưa ra một yêu cầu HTTP GET tới một máy chủ. Máy chủ này sau đó trả về tài nguyên được yêu cầu, thường được biểu diễn dưới dạng tài liệu HTML. HTTP là không trạng thái, có nghĩa đơn giản là mỗi yêu cầu hoặc phản hồi không lưu giữ thông tin nào về các yêu cầu hoặc phản hồi trước đó — với mỗi chuyển động quay lại và chuyển tiếp qua các trang web, toàn bộ trạng thái trình duyệt sẽ bị phá hủy và xây dựng lại từ đầu.

Máy chủ định tuyến các yêu cầu thay đổi trạng thái từ máy khách, cuối cùng khiến các biểu diễn trạng thái mới được trả lại, mà máy khách (thường là trình duyệt) vẽ lại hoặc báo cáo. Khi WWW lần đầu tiên hình thành mô hình này có ý nghĩa. Phần lớn, mạng mới này được hiểu là một hệ thống tệp phân tán, bất kỳ ai có trình duyệt web đều có thể truy cập được, nơi có thể yêu cầu một tài nguyên cụ thể (chẳng hạn như một bài báo) từ một máy tính cung cấp tệp (một máy chủ) nằm ở đâu đó trên mạng (tại một giao thức Internet hoặc là IP địa chỉ) thông qua một yêu cầu HTTP (chẳng hạn như GET) bằng cách nhập một URL (ví dụ: www.example.org/articles/april/showers.html).

Người dùng yêu cầu một trang và trang đó xuất hiện, có thể chứa các liên kết (siêu) đến các trang liên quan.

Tuy nhiên, vì giao thức không trạng thái không duy trì thông tin ngữ cảnh nên người điều hành máy chủ gần như không thể phát triển mối quan hệ thú vị hơn với khách truy cập qua một loạt yêu cầu hoặc để khách truy cập tổng hợp động nhiều phản hồi vào một chế độ xem.

Ngoài ra, khả năng diễn đạt của các yêu cầu bị hạn chế bởi cả bản thân giao thức và do thiếu nội dung máy chủ đủ phong phú để hỗ trợ một cách hữu ích từ vựng mô tả nhiều hơn. Phần lớn, các yêu cầu thẳng thừng như chỉ tay vào đối tượng ham muốn— "Hãy lấy cái đó cho tôi". Xem xét các phần của một URL điển hình:



Chúng ta có thể thấy cách làm việc của ứng dụng khách liên quan đến việc mô tả một vị trí tài nguyên đơn giản, với các tham số truy vấn và mục tiêu tài nguyên là một suy nghĩ khó xử sau đó, gần như không sử dụng được sau khi sử dụng nhiều hơn một vài bộ mô tả tài nguyên. Mặc dù điều này có thể thực hiện được trong thời đại của các tài liệu đơn giản trong các hệ thống phân cấp dễ hiểu và bất biến, nhưng nhu cầu và sự phức tạp của phần mềm nối mạng hiện đại đã khiến khái niệm ban đầu không thể hoạt động và cần được cải tiến.

Sự vụng về của việc chuyển các cặp khóa / giá trị ngày càng phức tạp để duy trì trạng thái của người dùng đã bắt đầu làm nản lòng tham vọng của phương tiện mới này. Rất nhanh chóng, các nhà phát triển trở nên rõ ràng rằng sự phụ thuộc ngày càng tăng vào Internet khi lớp truyền thông tiện ích tạo nền tảng cho thông tin, phần mềm và thương mại của thế giới đòi hỏi một cách tiếp cận tinh tế hơn.

Theo thời gian, những hạn chế này đã được khắc phục thông qua sự kết hợp của các cải tiến đối với giao thức HTTP, việc đưa JavaScript vào trình duyệt, các công nghệ như cookie của trình duyệt và sự đổi mới từ các nhà phát triển xây dựng các sản phẩm và dịch vụ khai thác những tiến bộ này.

Tuy nhiên, bản thân giao thức HTTP vẫn tiếp tục bị chi phối bởi các mô-típ giống nhau của các tài nguyên giống tệp riêng lẻ tồn tại ở một đường dẫn riêng biệt và cố định và được xác định bằng một tên thường không mô tả.

Những gì thực sự tồn tại trên nhiều máy chủ hiện nay là một phần mềm phức tạp chỉ định các giao diện mạng cho các mô hình dữ liệu. Giao tiếp với các loại ứng dụng nối mạng này liên quan đến việc lấy và thiết lập trạng thái của mô hình dữ liệu đó, cả nói chung và khi nó áp dụng nói riêng cho khách hàng đưa ra yêu cầu.

Các khách hàng triển khai giải pháp thời gian thực đều thiết lập và nhận các biểu diễn trạng thái tài nguyên trên máy chủ. Máy chủ ứng dụng phải báo cáo trạng thái của máy khách liên quan đến nhiều quy trình (cơ sở dữ liệu, tệp, công cụ quy tắc, công cụ tính toán, v.v.) theo từng yêu cầu và thường đơn phương về các thay đổi trạng thái ứng dụng (ví dụ: người dùng mất quyền truy cập). Khách hàng thường không phải là trình duyệt, mà là các máy chủ khác. Họ nên giao tiếp như thế nào?

## 5.6. Tìm hiểu các tuyến đường

Định tuyến ánh xạ URL tới các hành động. Thay vì xây dựng giao diện ứng dụng theo đường dẫn URL đến các tệp cụ thể có chứa một số logic, thiết kế với các tuyến liên quan đến việc gán một chức năng cụ thể cho sự kết hợp riêng biệt giữa đường dẫn URL và phương thức yêu cầu. Ví dụ: một dịch vụ web chấp nhận yêu cầu danh sách các thành phố có thể được gọi theo cách sau:

TẢI /services/cities.php?country=usa&state=ohio

Khi máy chủ của bạn nhận được yêu cầu này, nó sẽ chuyển thông tin URL đến một quy trình PHP sẽ thực thi logic ứng dụng trong city.php, chẳng hạn như đọc truy vấn, phân tích cú pháp quốc gia và tiểu bang, gọi cơ sở dữ liệu, xây dựng phản hồi và gửi lại.

Node có lợi ích lớn là có thể hoạt động như một máy chủ và môi trường ứng dụng. Máy chủ có thể gửi các yêu cầu trực tiếp. Do đó, sẽ có ý nghĩa hơn khi sử dụng URL như những tuyên bố đơn giản về ý định:

GET / listCities / usa / ohio

Trong máy chủ Node, chúng tôi có thể sử dụng mã nào đó giống như mã sau để xử lý các yêu cầu này cho các thành phố:

var app = http.createServer(function(request, response) {

var url = request.url; var method = request.method; if(method === "GET") {

if(url === "/listCities/usa/ohio") { database.call("usa","ohio",function(err, data) { response.writeHead(200, {'Content-Type':

'application/json' });

// Return list of cities in Ohio, USA response.end(JSON.stringify(data));

});

}

if(url === "/listCities/usa/arizona") { ... } if(url === "/listCities/canada/ontario") { ... } }

Một điều tốt và một điều không tốt sẽ xảy ra:

Xử lý URL được tổ chức rõ ràng ở một nơi

## 5.7. Mã không thể lặp lại

Viết ra mọi lộ trình có thể sẽ không hoạt động. Chúng tôi sẽ giữ nguyên tổ chức, nhưng cần tạo các biến trong các tuyến, ưu tiên xác định một biểu thức tuyến chung như sau:

/ listCities /: country /: state

Phương pháp listCities có thể chấp nhận Quốc gia và tiểu bang Biến đổi đối số, được xác định bằng tiền tố dấu hai chấm (:). Trong máy chủ của chúng tôi, chúng tôi cần chuyển đổi biểu thức tượng trưng này thành một biểu thức chính quy. Trong trường hợp này RegExp / ^ \ /

listCities \ / ([^ \ / \.] +) \ / ([^ \ / \.] +) \ /? $ / có thể được sử dụng để trích xuất theo thứ tự các giá trị từ URL mẫu của chúng tôi vào một bản đồ giá trị tương tự như:

{ country: "usa", state: "ohio" }

Bằng cách coi các yêu cầu là biểu thức, thiết kế máy chủ của chúng tôi đã trở nên gọn gàng hơn một chút, định tuyến độc đáo bất kỳ tổ hợp quốc gia / tiểu bang nào đến một hàm xử lý chung:

if(request.method === "GET") {

var match = request.url.match(/^\/listCities\/([^\/\.]+)\/

([^\/\.]+)\/?$/); if(match) {

database.call(match[1],match[2],function(err, data) {…}

}

Hình thức định tuyến yêu cầu này đã "giành được tranh luận" trong cộng đồng Node, là hành vi mặc định của các khung và công cụ khác nhau. Trên thực tế, cách nghĩ này về các yêu cầu định tuyến đã được chấp nhận trong nhiều môi trường phát triển khác, chẳng hạn như Ruby on Rails. Do đó, hầu hết các khung ứng dụng web cho Node đã được phát triển xung quanh việc định tuyến.

Khung ứng dụng web phổ biến nhất cho Node là TJ Holowaychuk ' S bày tỏ

và chúng tôi sẽ sử dụng khung này thường xuyên trong cuốn sách này khi thiết kế máy chủ định tuyến. Bạn có thể cài đặt nó qua npm bằng cách chạy npm install –g express.

## 5.8. Sử dụng Express để định tuyến các yêu cầu

Express đơn giản hóa sự phức tạp của việc xác định các quy trình đối sánh tuyến đường. Ví dụ của chúng tôi có thể được viết theo cách sau bằng cách sử dụng Express:

var express = require('express'); var app = express();

app.get('/listCities/:country/:state', function(request, response){ var country = request.params.country; var state = request.params.state;

response.end("You asked for country: " + country + " and state: " + state);

}); app.listen(8080);

GET /listCities/usa/ohio

// You asked for country: usa and state: ohio

GET /didnt/define/this

// Cannot GET /didnt/define/this

GET /listCities // note missing arguments

// Cannot GET /listCities

Instantiating Express cung cấp một máy chủ web được định dạng đầy đủ được bao bọc trong một API phát triển ứng dụng dễ sử dụng. Dịch vụ thành phố của chúng tôi được xác định rõ ràng và các biến của nó được nêu, hy vọng sẽ được gọi qua GET (một người cũng có thể sử dụng app.post (...) hoặc là ứng dụng. đặt(...) hoặc bất kỳ phương thức HTTP tiêu chuẩn nào khác).

Express cũng giới thiệu ý tưởng về chuỗi các quy trình xử lý yêu cầu, trong Express được hiểu là phần mềm trung gian. Trong ví dụ của chúng tôi, chúng tôi đang gọi một hàm duy nhất để xử lý yêu cầu thành phố. Điều gì sẽ xảy ra nếu trước khi gọi cơ sở dữ liệu của chúng tôi, chúng tôi muốn kiểm tra xem người dùng đã được xác thực chưa? Chúng tôi có thể thêm một xác thực () phương pháp trước đối với phương thức dịch vụ chính của chúng tôi:

var authenticate = function(request, response, next) { if(validUser) { next(); }

else { response.end("INVALID USER!"); }

}

app.get('/listCities/:country/:state', authenticate, function(request, response) { ... });

Nói cách khác, phần mềm trung gian có thể được xâu chuỗi, đơn giản hóa việc tạo ra các chuỗi thực thi phức tạp, tuân theo một cách độc đáo Quy tắc Môđun. Nhiều loại phần mềm trung gian để xử lý biểu tượng yêu thích, ghi nhật ký, tải lên, yêu cầu tệp tĩnh, v.v. đã được phát triển. Để tìm hiểu thêm, hãy truy cập liên kết sau:

http://www.express.com

Sau khi thiết lập cách thích hợp để máy chủ Node được định cấu hình cho các yêu cầu định tuyến, bây giờ chúng ta có thể bắt đầu thảo luận về cách xác định máy khách thực hiện yêu cầu, gán cho máy khách đó một ID phiên duy nhất và quản lý phiên đó theo thời gian.

## 5.9. Sử dụng Redis để theo dõi trạng thái khách hàng

Đối với một số ứng dụng và ví dụ trong chương này, chúng tôi sẽ sử dụng Redis, một trong ký ức Khóa / Giá trị (KV) cơ sở dữ liệu được phát triển bởi Salvatore Sanfilippo và hiện được hỗ trợ bởi Pivotal. Thông tin thêm về Redis có thể được tìm thấy tại http://redis.io.

Một đối thủ cạnh tranh nổi tiếng với Redis là Đã ghi nhớ ( http://memcached.org).

Nói chung, bất kỳ máy chủ nào phải duy trì trạng thái phiên của nhiều máy khách sẽ cần lớp dữ liệu tốc độ cao với hiệu suất đọc / ghi gần như tức thời, vì xác thực yêu cầu và chuyển đổi trạng thái người dùng có thể xảy ra nhiều lần theo mỗi yêu cầu. Cơ sở dữ liệu quan hệ được hỗ trợ bởi tệp truyền thống có xu hướng thực hiện nhiệm vụ này chậm hơn so với cơ sở dữ liệu KV trong bộ nhớ. Chúng tôi sẽ sử dụng Redis để theo dõi trạng thái khách hàng.

Redis là một máy chủ đơn luồng với cài đặt đơn giản:

wget http://download.redis.io/redis-stable.tar.gz tar xvzf redis-stable.tar.gz cd redis-ổn định làm

Hiện có một máy chủ và một tiện ích dòng lệnh có sẵn trong / src thư mục của thư mục cài đặt. Bạn nên làm cho chúng dễ dàng truy cập trên dòng lệnh:

sudo cp redis-server / usr / local / bin / sudo cp redis-cli / usr / local / bin /

Để khởi động Redis, hãy chạy dòng lệnh sau:

redis-server

Để tương tác với Redis:

redis-cli

Các bước trước là đủ khi khám phá sự phát triển của Redis. Trong quá trình sản xuất, người ta sẽ cần phải định cấu hình đúng cách cho bản dựng, cấp phát bộ nhớ, đặt mật khẩu, v.v. Khi đã sẵn sàng, bạn có thể tìm kiếm thêm thông tin từ liên kết sau:

http://redis.io/topics/quickstart

Đáng chú ý, Amazon's ElastiCache dịch vụ cho phép Redis "trên đám mây" làm bộ nhớ đệm trong bộ nhớ, với khả năng tự động mở rộng và dự phòng:

http://aws.amazon.com/elasticache/

Redis hỗ trợ giao diện tiêu chuẩn cho các hành động mong đợi, chẳng hạn như nhận hoặc thiết lập các cặp Khóa / Giá trị. Đến được giá trị được lưu trữ tại một khóa, trước tiên hãy khởi động Redis CLI:

redis> set somerandomkey "who am I?" redis> get somerandomkey "who am I?"

Redis sẽ trả về (nil) khi khóa không tồn tại. Hãy bộ một chìa khóa:

redis> đặt somerandomkey "tôi là ai?" redis> lấy somerandomkey

"tôi là ai?"

Để sử dụng Redis trong môi trường Node, chúng ta sẽ cần một số loại ràng buộc. Chúng tôi sẽ sử dụng Matt Ranney ' S node\_redis mô-đun. Cài đặt nó với npm bằng dòng lệnh sau:

npm cài đặt Rentis redis

Để đặt giá trị trong Redis và lấy lại giá trị đó, bây giờ chúng ta có thể thực hiện việc này trong Node: var redis = request ("redis");

khách hàng var = redis.createClient ();

client.set ("userId", "jack", function (err) { client.get ("userId", function (err, data) { console.log (dữ liệu); // "jack"

});

});

## 5.10. Lưu trữ dữ liệu người dùng

Quản lý nhiều người dùng có nghĩa là ít nhất theo dõi thông tin người dùng của họ, một số được lưu trữ lâu dài (ví dụ: địa chỉ, lịch sử mua hàng và danh sách liên hệ) và một số dữ liệu phiên được lưu trữ trong thời gian ngắn (thời gian kể từ khi đăng nhập, điểm số trận đấu gần đây nhất và câu trả lời gần đây nhất ).

Thông thường, chúng tôi sẽ tạo một giao diện an toàn hoặc tương tự, cho phép quản trị viên tạo tài khoản người dùng. Người đọc sẽ rõ cách tạo một giao diện như vậy vào cuối chương này. Đối với các ví dụ tiếp theo, chúng tôi sẽ chỉ cần tạo một người dùng, hoạt động như một tình nguyện viên. Hãy tạo Jack:

redis> mật khẩu jack hset "beantalk" redis> tên đầy đủ của jack hset "Jack Spratt"

Thao tác này sẽ tạo khóa trong Redis— Jack —Chứa một hàm băm giống như:

{

"password": "beantalk",

"tên đầy đủ": "Jack Spratt"

}

Nếu chúng tôi muốn tạo một băm và thêm một số cặp KV cùng một lúc, chúng tôi có thể đạt được điều trước với hmset chỉ huy:

redis> mật khẩu jack hmset "beantalk" tên đầy đủ là "Jack Spratt"

Hiện nay Jack tồn tại: redis> jack hgetall 1) "mật khẩu"

"cây đậu"

"tên đầy đủ"

"Jack Spratt"

Chúng ta có thể sử dụng lệnh sau để tìm nạp giá trị được lưu trữ cho một trường cụ thể trong Jack ' tài khoản của:

redis> hget jack password // "beantalk"

## 5.11. Xử lý phiên

Làm thế nào để một máy chủ biết được liệu yêu cầu khách hàng hiện tại có phải là một phần của chuỗi các yêu cầu trước đó hay không? Các ứng dụng web tương tác với khách hàng thông qua các chuỗi giao dịch dài— giỏ hàng chứa các mặt hàng cần mua sẽ vẫn ở đó ngay cả khi người mua hàng chuyển hướng đi để thực hiện một số hoạt động mua sắm so sánh. Chúng tôi sẽ gọi đây là phiên họp, có thể chứa bất kỳ số lượng cặp KV nào, chẳng hạn như tên người dùng, danh sách sản phẩm hoặc lịch sử đăng nhập của người dùng.

Các phiên được bắt đầu, kết thúc và theo dõi như thế nào? Có nhiều cách để tấn công vấn đề này, tùy thuộc vào nhiều yếu tố tồn tại theo những cách khác nhau trên các kiến trúc khác nhau. Đặc biệt, nếu nhiều máy chủ đang được sử dụng để xử lý các máy khách, thì dữ liệu phiên được chia sẻ giữa chúng như thế nào?

Chúng tôi sẽ sử dụng cookie để lưu trữ ID phiên cho khách hàng, đồng thời xây dựng một máy chủ bỏ phiếu dài đơn giản. Hãy nhớ rằng khi các ứng dụng ngày càng phức tạp, hệ thống đơn giản này sẽ cần được mở rộng. Ngoài ra, việc thăm dò ý kiến lâu dài như một công nghệ đang tạo nền tảng cho các kỹ thuật socket mạnh mẽ hơn mà chúng ta sẽ khám phá trong các cuộc thảo luận xung quanh việc xây dựng hệ thống thời gian thực. Tuy nhiên, các vấn đề chính gặp phải khi giữ nhiều máy khách được kết nối đồng thời trên một máy chủ và theo dõi các phiên của chúng, cần được chứng minh.

## 5.12. Cookie và trạng thái khách hàng

Netscape cung cấp đặc điểm kỹ thuật sơ bộ cho cookie, vào năm 1997:

Cookie là một cơ chế chung mà các kết nối phía máy chủ ... có thể sử dụng để lưu trữ và truy xuất thông tin trên phía máy khách của kết nối. Việc bổ sung trạng thái phía máy khách đơn giản, liên tục, mở rộng đáng kể khả năng của các ứng dụng máy khách / máy chủ dựa trên Web ...

Máy chủ, khi trả về một đối tượng HTTP cho máy khách, cũng có thể gửi một phần thông tin trạng thái mà máy khách sẽ lưu trữ ... Bất kỳ yêu cầu HTTP nào trong tương lai được thực hiện bởi máy khách ... sẽ bao gồm một bản truyền giá trị hiện tại của trạng thái đối tượng từ máy khách trở lại máy chủ. Đối tượng trạng thái được gọi là cookie, không có lý do thuyết phục.

- http://lib.ru/WEBMASTER/cookie\_spec.txt\_with-big-pictures.html

Ở đây, chúng tôi có một trong những nỗ lực đầu tiên để "sửa chữa" tính chất không trạng thái của

HTTP, cụ thể là việc duy trì trạng thái phiên. Đó là một nỗ lực tốt, nó vẫn là một phần cơ bản của Web.

Chúng tôi đã biết cách đọc và đặt Bánh quy tiêu đề với Node. Express làm cho quá trình dễ dàng hơn một chút:

var express = demand ('express'); var app = express ();

app.use (express.cookieParser ());

app.get ('/ mycookie', function (request, response) { response.end (request.cookies.node\_cookie);

});

app.get ('/', function (request, response) { response.cookie ('node\_cookie', Math.floor (Math.random () \* 10e10)); response.end ("Tập hợp cookie");

});

app.listen (8000);

Lưu ý sử dụng , cho phép chúng tôi bật phần mềm trung gian xử lý cookie cho Express. Ở đây, chúng ta thấy rằng bất cứ khi nào khách hàng truy cập vào máy chủ của chúng tôi, khách hàng này sẽ được gán một số ngẫu nhiên làm cookie. Bằng cách điều hướng đến / mycookie khách hàng này có thể thấy cookie.

## 5.13. Một cuộc thăm dò đơn giản

Chúng ta cần tạo ra một môi trường đồng thời, một môi trường có nhiều máy khách được kết nối đồng thời. Chúng tôi sẽ sử dụng máy chủ thăm dò dài để thực hiện việc này, truyền phát tới tất cả các máy khách được kết nối qua stdin. Ngoài ra, mỗi khách hàng sẽ được chỉ định một ID phiên duy nhất, được sử dụng để xác định http.serverResponse đối tượng mà chúng tôi sẽ đẩy dữ liệu đến.

Bỏ phiếu dài là một kỹ thuật theo đó máy chủ giữ kết nối máy khách cho đến khi có sẵn dữ liệu để gửi. Khi dữ liệu cuối cùng được gửi đến máy khách, máy khách sẽ kết nối lại với máy chủ và quá trình tiếp tục. Nó được thiết kế như một cải tiến về thăm dò ý kiến ngắn, đó là kỹ thuật không hiệu quả trong việc kiểm tra một cách mù quáng với máy chủ để tìm thông tin mới sau mỗi vài giây hoặc lâu hơn, với hy vọng có dữ liệu mới. Bỏ phiếu dài chỉ yêu cầu kết nối lại sau khi cung cấp dữ liệu hữu hình cho khách hàng.

Chúng tôi sẽ sử dụng hai tuyến đường. Tuyến đầu tiên được mô tả bằng cách sử dụng dấu gạch chéo (/), một yêu cầu miền gốc. Các lệnh gọi đến đường dẫn này sẽ trả về một số HTML tạo thành giao diện người dùng khách. Con đường thứ hai là / thăm dò ý kiến, mà máy khách sẽ sử dụng để kết nối lại với máy chủ sau khi nhận được một số dữ liệu.

Giao diện người dùng khách hàng cực kỳ đơn giản: mục đích duy nhất của nó là tạo Yêu cầu HTTP XML (XHR) tới một máy chủ (sẽ giữ yêu cầu đó cho đến khi một số dữ liệu đến), lặp lại bước này ngay sau khi nhận được một số dữ liệu. Giao diện người dùng của chúng tôi sẽ hiển thị danh sách các thư nhận được trong một danh sách không có thứ tự. Đối với bit XHR, chúng ta sẽ sử dụng thư viện jQuery. Bất kỳ thư viện tương tự nào cũng có thể được sử dụng và việc xây dựng một triển khai JavaScript thuần túy không khó.

HTML:

<ul id = "results"> </ul> JavaScript:

function longPoll () {

$ .get ('http://www.pathjs.com/poll', function (data) {

$ ('<li>' + data + '</li>') .appendTo ('# results'); longPoll ();

}); } longPoll ();

Máy chủ cũng đơn giản, chịu trách nhiệm chính trong việc thiết lập các ID phiên và giữ các kết nối máy khách đồng thời cho đến khi có sẵn dữ liệu, dữ liệu này được phát tới tất cả các máy khách được kết nối. Các kết nối này được lập chỉ mục thông qua ID phiên, được duy trì bằng cookie:

var connections = {}; app.use(express.cookieParser()); app.get('/poll', function(request, response){ var id = request.cookies.node\_poll\_id; if(!id) { return;

}

connections[id] = response;

});

app.get('/', function(request, response) { fs.readFile('./poll\_client.html', function(err, data) { response.cookie('node\_poll\_id', Math.floor(Math.random() \*

10e10));

response.writeHead(200, {'Content-Type': 'text/html'}); response.end(data);

});

}); app.listen(2112);

Sau khi khách hàng đã tìm nạp giao diện người dùng, nó đã sẵn sàng để nhận dữ liệu. Trong ví dụ này, chúng tôi sẽ cho phép dữ liệu đến qua stdin được phát tới tất cả các máy khách được kết nối:

var broadcast = function(msg) {

var conn;

for(conn in connections) { connections[conn].end(msg);

}

}

process.stdin.on('readable', function() {

var msg = this.read(); msg && broadcast(msg.toString()); });

Chạy máy chủ này trên dòng lệnh và kết nối với máy chủ qua trình duyệt. Một trang gần như trống sẽ được hiển thị. Quay lại dòng lệnh và nhập một số văn bản — thông báo này sẽ ngay lập tức xuất hiện trong trình duyệt của bạn. Hãy thử nó với một số cửa sổ trình duyệt đang mở, tất cả sẽ nhận được những gì bạn đã nhập.

Ví dụ nhỏ này cho thấy một lợi thế khác của mô hình đơn luồng có sự kiện của Node, đặc biệt là khi nói đến việc xây dựng máy chủ, việc đóng cửa. Chúng tôi thấy cách một chức năng đơn lẻ, nhận các ràng buộc yêu cầu / phản hồi cho một ứng dụng khách duy nhất, thông qua việc đóng, có thể truy cập dữ liệu chung một cách an toàn. Đây phát sóng phương pháp đang truy cập các thành viên của kết nối đối tượng được tạo trước đó trong app.get —Một phạm vi phương thức đang truy cập dữ liệu được tạo trong phạm vi bên ngoài. Mỗi cuộc gọi đến trình xử lý yêu cầu HTTP được bảo vệ khỏi sự can thiệp từ một luồng hoặc quy trình khác, nhưng vẫn có thể được tiếp xúc với dữ liệu được chia sẻ hữu ích thông qua quá trình đóng.

Một cách sử dụng thú vị của cái mới WeakMap đặc điểm của JavaScript là sử dụng đối tượng yêu cầu làm khóa trong bản đồ (hãy nhớ sử dụng cờ hòa hợp khi bắt đầu quy trình Node của bạn).

## 5.14. Tập trung các trạng thái

Bằng cách giữ các phiên người dùng tập trung trong cơ sở dữ liệu Redis, chúng tôi có thể có nhiều máy chủ truy cập vào cùng một dữ liệu phiên. Đây là ưu điểm chính của việc tập trung dữ liệu phiên. Ngoài ra, có thể thực hiện phân tích các phiên hiện tại (và thậm chí cả các phiên trước đây).

Chúng tôi đã thấy cách chúng tôi có thể sử dụng cookie để theo dõi các kết nối của người dùng. Vấn đề chính của giải pháp này là nó chỉ hoạt động trên một máy chủ duy nhất. Điều gì sẽ xảy ra nếu hệ thống của chúng tôi đang chạy hai máy chủ khách riêng biệt? Bản đồ kết nối chúng tôi đã sử dụng trước đó được xác định bởi các kết nối đến địa phương người phục vụ. Điều gì sẽ xảy ra nếu dữ liệu được ghi từ Xa xôi người phục vụ? Làm cách nào để chúng tôi đảm bảo rằng tất cả các máy chủ đều được thông báo về tất cả các sự kiện, để mỗi máy chủ có thể đẩy thông tin mới đến tất cả nó là khách hàng?

Redis giới thiệu ý tưởng về pub / sub, hoặc xuất bản / đăng ký, có thể được sử dụng để gửi thông báo qua nhiều quy trình Node. Cách sử dụng rất đơn giản và trực quan:

var redis = require("redis"); var receiver = redis.createClient(); var publisher = redis.createClient(); receiver.subscribe("firehose");

receiver.on("message", function(channel, message) {

console.log("Received message: " + message + " on channel: " + channel);

});

setTimeout(function() {

publisher.publish("firehose", "Hello!"); }, 1000);

Sau khi tạo nhà xuất bản và người nhận khách hàng, chúng tôi đợi khoảng một giây và xuất bản tin nhắn lên một kênh, trong trường hợp này có người đăng ký thông báo về việc nhận được tin nhắn.

Vấn đề với việc mở rộng cơ sở hạ tầng bỏ phiếu đơn giản của chúng tôi được giải quyết bằng cách một vài thay đổi nhỏ. Hãy xem xét lại mã bắt và phát tin nhắn:

Var broadcast = function(msg) {

var conn;

for(conn in connections) { connections[conn].end(msg);

}

}

process.stdin.on('readable', function() {

var msg = this.read(); msg && broadcast(msg.toString()); });

Hãy chuyển đổi hệ thống này thành một hệ thống xuất bản / đăng ký, cho phép bất kỳ số lượng máy chủ nào trong số này tồn tại trong khi vẫn duy trì phạm vi phát sóng đầy đủ khi có tin nhắn mới:

var redis = require("redis"); var receiver = redis.createClient(); var publisher = redis.createClient();

receiver.subscribe("stdin\_message"); receiver.on("message", function(channel, message) {

var conn;

for(conn in connections) { connections[conn].end(message);

}

console.log("Received message: " + message + " on channel: " + channel);

});

process.stdin.on('readable', function() { var msg = this.read();

msg && publisher.publish("stdin\_message", msg.toString()); });

Thay vì ưu tiên máy chủ cục bộ, tất cả việc phát tin nhắn được thực hiện "mạng nhận biết", cho phép tất cả các phiên bản của máy chủ này có cơ hội phát sóng tới các máy khách mà nó đang quản lý, bất kể máy chủ nào đang thực sự nhận dữ liệu tin nhắn.

Khi chúng ta tiến bộ qua các ứng dụng lớn hơn trong các chương sau, các kỹ thuật nhắn tin mạng nâng cao hơn sẽ được giải nén. Một điểm mấu chốt quan trọng là việc nhân rộng các máy chủ Node là các chiến lược mở rộng nút điển hình. Lập kế hoạch cho một cửa hàng phiên được phân phối và xây dựng lớp giao tiếp nội bộ để thực hiện việc phân phối đó.

## 5.15. Xác thực kết nối

Cùng với việc thiết lập các đối tượng phiên máy khách, máy chủ Node thường yêu cầu thông tin xác thực. Lý thuyết và thực hành về bảo mật web rất rộng rãi. Chúng tôi muốn đơn giản hóa sự hiểu biết của mình thành hai trường hợp xác thực chính:

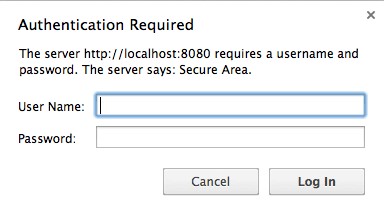
Khi giao thức dây là HTTPS

Khi đó là HTTP

Đầu tiên là tự nhiên an toàn, và thứ hai là không. Đầu tiên chúng ta sẽ học cách để triển khai Xác thực cơ bản trong Node, và trong lần thứ hai, hệ thống phản hồi thử thách sẽ được mô tả.

Xác thực cơ bản

Như đã đề cập, Xác thực cơ bản gửi văn bản thuần túy qua dây có chứa tổ hợp tên người dùng / mật khẩu, sử dụng tiêu đề HTTP tiêu chuẩn. Nó là một giao thức đơn giản và nổi tiếng. Bất kỳ máy chủ nào gửi tiêu đề chính xác sẽ khiến bất kỳ trình duyệt nào hiển thị hộp thoại đăng nhập, như hộp thoại sau:



Tuy nhiên, phương pháp này vẫn không an toàn, gửi dữ liệu không được mã hóa một cách đơn giản văn bản qua dây. Vì mục đích đơn giản, chúng tôi sẽ chứng minh phương pháp xác thực này trên máy chủ HTTP, nhưng cần phải nhấn mạnh rằng trong việc sử dụng máy chủ trong thế giới thực phải được giao tiếp thông qua một giao thức an toàn, chẳng hạn như HTTPS.

Hãy triển khai giao thức xác thực này với Node. Sử dụng cơ sở dữ liệu người dùng được phát triển trước đó trong Redis, chúng tôi xác thực thông tin đăng nhập đã gửi bằng cách kiểm tra các đối tượng người dùng để tìm mật khẩu khớp, xử lý lỗi và thành công.

var http = require('http'); var redis = require("redis"); var client = redis.createClient();

http.createServer(function(req, res) { var auth = req.headers['authorization']; if(!auth)

Bằng cách gửi một 401 trạng thái và ' ủy quyền ' trên một kết nối máy khách mới, một hộp thoại giống như ảnh chụp màn hình trước đó sẽ xuất hiện:

res.writeHead(401, {'WWW-Authenticate': 'Basic realm="Secure Area"'});

return res.end('<html><body>Please enter some credentials.</body></html>');

} var tmp = auth.split(' ');

Phân chia trên một khoảng trắng, chuỗi xác thực ban đầu trông giống như Cơ bản Y2hhcmxlczoxMjM0NQ == và chúng ta cần phần thứ hai. Lưu ý ' base64 ' chuyển đổi trong đoạn mã sau:

var buf = new Buffer(tmp[1], 'base64'); var plain\_auth = buf.toString(); var creds = plain\_auth.split(':'); var username = creds[0]; var password = creds[1]; // Find this user record client.get(username, function(err, data) { if(err || !data) {

res.writeHead(401, {'WWW-Authenticate': 'Basic realm="Secure Area"'}); return res.end('<html><body>You are not authorized.</body></html>');

}

res.statusCode = 200;

res.end('<html><body>Welcome!</body></html>');

});

}).listen(8080);

Bằng cách này, một hệ thống đăng nhập đơn giản có thể được thiết kế. Bởi vì các trình duyệt sẽ tự nhiên nhắc người dùng yêu cầu quyền truy cập vào miền được bảo vệ, ngay cả hộp thoại đăng nhập cũng được quan tâm.

Để tìm hiểu về cách triển khai Xác thực cơ bản bằng Express, hãy truy cập liên kết sau:

http://expressjs.com/api.html#basicAuth

## 5.16. Handshaking

Một phương pháp xác thực khác cần xem xét trong các tình huống không thể thiết lập kết nối HTTPS là hệ thống phản hồi thử thách:

|  |  |  |
| --- | --- | --- |
| |  | | --- | | Khách hàng |  |  | | --- | | Người phục vụ |   Chuyển id khách hàng, yêu cầu quyền truy cập  Thử thách khóa công khai  Trả lời với hàm băm được yêu cầu |

Trong trường hợp này, một máy khách yêu cầu máy chủ cấp quyền truy cập cho một người dùng, tên người dùng, ID cụ thể hoặc tương tự. Thông thường, dữ liệu này sẽ được gửi qua biểu mẫu đăng nhập. Hãy giả lập một kịch bản thử thách / phản hồi, sử dụng ví dụ về người dùng mà chúng tôi đã tạo trước đó— Jack.

Thiết kế và mục đích chung của hệ thống thử thách / phản hồi nhằm mục đích tránh gửi bất kỳ dữ liệu mật khẩu nào dưới dạng văn bản thuần túy qua mạng. Vì vậy, chúng tôi sẽ cần quyết định chiến lược mã hóa mà cả máy khách và máy chủ đều chia sẻ. Đối với ví dụ của chúng tôi, hãy sử dụng SHA256 thuật toán. Điểm giao tiền điện tử thư viện chứa tất cả các công cụ cần thiết để tạo loại băm này. Khách hàng có thể không, vì vậy chúng tôi phải cung cấp một. Chúng tôi sẽ sử dụng cái được phát triển bởi Chris Veness, có thể tải xuống từ liên kết sau:

http://csrc.nist.gov/groups/ST/toolkit/examples.html

Để bắt đầu đăng nhập này, khách hàng sẽ cần gửi một yêu cầu xác thực cho người dùng Jack:

GET /authenticate/jack

Đáp lại, khách hàng sẽ nhận được một khóa công khai do máy chủ tạo — một thách thức. Khách hàng bây giờ phải tạo một chuỗi Jack ' mật khẩu s có tiền tố bằng khóa này. Tạo hàm băm SHA256 từ nó và chuyển hàm băm kết quả cho / đăng nhập/. Máy chủ cũng sẽ tạo cùng một hàm băm SHA256 — nếu cả hai khớp nhau, máy khách sẽ được xác thực:

<script src="sha256.js"></script>

<script>

$.get("/authenticate/jack", function(publicKey) { if(publicKey === "no data") {

return alert("Cannot log in.");

}

// Expect to receive a challenge: the client should be able

// to derive a SHA456 hash

// String in this format: publicKey + password. Return that // string.

//

var response = Sha256.hash(publicKey + "beanstalk");

$.get("/login/" + response, function(verdict) { if(verdict === "failed") {

return alert("No Dice! Not logged in.");

}

alert("You're in!");

});

});

</script>

Bản thân máy chủ rất đơn giản, bao gồm hai tuyến xác thực đã đề cập. Chúng ta có thể thấy trong đoạn mã sau khi nhận được tên người dùng, ( giắc cắm) trước tiên máy chủ sẽ kiểm tra hàm băm của người dùng hiện có trong Redis, phá vỡ quá trình bắt tay nếu không tìm thấy dữ liệu nào như vậy. Nếu bản ghi tồn tại, chúng tôi tạo khóa công khai mới, ngẫu nhiên, soạn hàm băm SHA256 có liên quan và trả lại giá trị thử thách này cho khách hàng. Ngoài ra, chúng tôi đặt băm này làm khóa trong Redis, với giá trị của nó là tên người dùng đã gửi:

var crypto = require('crypto'); var fs = require('fs'); var express = require('express'); var app = express(); var redis = require("redis"); var client = redis.createClient();

app.get('/authenticate/:username', function(request, response){ var publicKey = Math.random();

var username = request.params.username; // This is always

"jack"

// ... get jack's data from redis client.hgetall(username, function(err, data) { if(err || !data) {

return response.end("no data");

}

// Creating the challenge hash var challenge =

crypto.createHash('sha256').update(publicKey + data.password).digest('hex');

// Store challenge for later match

client.set(challenge, username); response.end(challenge);

});

});

app.get('/login/:response', function(request, response){ var hash = request.params.response; client.exists(hash, function(err, exists) { if(err || !exists) {

return response.end("failed");

}

});

client.del(hash, function() { response.end("OK");

});

});

Bên trong / đăng nhập/ trình xử lý định tuyến, chúng tôi có thể thấy cách kiểm tra được thực hiện nếu phản hồi tồn tại dưới dạng khóa trong Redis và nếu được tìm thấy, chúng tôi ngay lập tức xóa khóa. Điều này là cần thiết vì một số lý do, trong đó không ít lý do ngăn cản những người khác gửi cùng một phản hồi và giành quyền truy cập. Nói chung, chúng tôi cũng không muốn những thứ này trở nên vô dụng, chúng tôi muốn các chìa khóa chất đống. Đây là một vấn đề: điều gì sẽ xảy ra nếu một khách hàng không bao giờ đáp lại thử

thách? Vì việc dọn dẹp khóa chỉ xảy ra khi a / đăng nhập/ đã cố gắng, khóa này sẽ không bao giờ bị xóa.

Không giống như hầu hết các kho dữ liệu KV, Redis giới thiệu ý tưởng về khóa hết hạn, nơi một bộ hoạt động có thể chỉ định một Thời gian để sống (TTL) cho một chìa khóa. Ví dụ, ở đây chúng tôi sử dụng setex lệnh đặt khóa tên người dùng giá trị 183 và chỉ định rằng khóa này sẽ hết hạn sau một giây:

client.setex ("doomed", 10, "story", function (err) {...});

Tính năng này cung cấp một giải pháp tuyệt vời cho vấn đề của chúng tôi. Bằng cách thay thế khách hàng. đặt (thử thách, tên người dùng); dòng với dòng sau:

client.setex(challenge, 5, username)

chúng tôi đảm bảo rằng, không có vấn đề gì, khóa này sẽ biến mất trong 5 giây. Làm những việc theo cách này cũng hoạt động như một biện pháp an ninh ánh sáng, để lại một cửa sổ rất ngắn để phản hồi vẫn hợp lệ và nghi ngờ về phản hồi chậm.

## Tóm lược

Node cung cấp một bộ công cụ giúp thiết kế và bảo trì các ứng dụng mạng quy mô lớn đang đối mặt với vấn đề C10K. Trong chương này, chúng tôi đã thực hiện các bước đầu tiên để tạo các ứng dụng mạng với nhiều máy khách đồng thời, theo dõi thông tin phiên và thông tin đăng nhập của họ. Sự khám phá về đồng thời này đã chứng minh một số kỹ thuật định tuyến, theo dõi và phản hồi cho khách hàng. Chúng tôi đã đề cập đến một số kỹ thuật đơn giản để sử dụng khi mở rộng quy mô, chẳng hạn như triển khai nhắn tin nội bộ thông qua hệ thống xuất bản / đăng ký được xây dựng bằng cơ sở dữ liệu Redis.

Giờ đây, chúng tôi đã sẵn sàng đi sâu hơn vào việc thiết kế phần mềm thời gian thực — bước tiếp theo hợp lý sau khi đạt được tính đồng thời cao và độ trễ thấp bằng cách sử dụng Node. Chúng tôi sẽ mở rộng các ý tưởng được nêu ra trong quá trình thảo luận về việc bỏ phiếu dài và đặt chúng trong bối cảnh các vấn đề và giải pháp mạnh mẽ hơn.

đọc thêm

Đồng thời và song song là những khái niệm phong phú đã được nghiên cứu và tranh luận nghiêm túc. Khi một kiến trúc ứng dụng được thiết kế để ưu tiên các chủ đề, sự kiện hoặc một số kết hợp, có khả năng là các kiến trúc sư đã kiên định về cả hai khái niệm. Bạn được khuyến khích nhúng một ngón chân vào lý thuyết và đọc các bài viết sau. Sự hiểu biết rõ ràng về chính xác nội dung cuộc tranh luận sẽ cung cấp một khuôn khổ khách quan có thể được sử dụng để đưa ra quyết định chọn (hoặc không chọn) Node.

Một số con số:

https://cs.uwaterloo.ca/~brecht/papers/getpaper. php? file = euroys-2007.pdf

Chủ đề là một ý tưởng tồi:

http://www.cs.sfu.ca/~vaughan/teaching/431/papers/ousterhoutthreads-usenix96.pdf

Sự kiện là một ý tưởng tồi:

http://static.usenix.org/events/hotos03/tech/full\_papers/ vonbehren / vonbehren.pdf

Làm thế nào về cùng nhau?

http://repository.upenn.edu/cgi/viewcontent. cgi? article = 1391 & context = cis\_papers

Đó là một sự phân đôi sai lầm:

http://swtch.com/~rsc/talks/threads07/

# CHƯƠNG 6

# TẠO ỨNG DỤNG THỜI GIAN THỰC

Không có gì trường tồn ngoài sự thay đổi.

- Heraclitus

Phần mềm thời gian thực là gì? Danh sách bạn bè được cập nhật ngay khi một người tham gia hoặc thoát. Cập nhật giao thông tự động truyền vào điện thoại thông minh của người lái xe cho con đường tốt nhất về nhà. Trang thể thao của một tờ báo điện tử ngay lập tức cập nhật bảng điểm và bảng xếp hạng khi điểm được tính trong một trận đấu thực tế. Người dùng loại phần mềm này mong đợi các phản ứng thay đổi được truyền đạt nhanh chóng và kỳ vọng này đòi hỏi sự tập trung đặc biệt vào việc giảm độ trễ mạng từ nhà thiết kế phần mềm. Cập nhật dữ liệu I / O phải xảy ra dọc theo khung thời gian dưới giây.

Hãy lùi lại và xem xét các đặc điểm chung của môi trường và cộng đồng Node khiến nó trở thành một công cụ tuyệt vời để tạo các loại ứng dụng mạng đáp ứng này.

Có thể lập luận, một số xác nhận về thiết kế của Node được tìm thấy trong cộng đồng khổng lồ gồm các nhà phát triển mở đóng góp các hệ thống Node cấp doanh nghiệp, một số nhóm được hỗ trợ tài chính và được các công ty Internet lớn như Microsoft, LinkedIn, eBay và những người khác. Các hệ thống doanh nghiệp đa lõi, nhiều máy chủ đang được tạo bằng phần mềm miễn phí chủ yếu được viết bằng JavaScript.

Tại sao rất nhiều công ty chuyển sang Node khi thiết kế hoặc cập nhật sản phẩm của họ? Danh sách sau liệt kê các lý do tại sao:

Node cung cấp hệ thống quản lý gói npm tuyệt vời, tích hợp dễ dàng với hệ thống kiểm soát phiên bản Git. Đường cong học tập nông cạn giúp ngay cả các nhà phát triển thiếu kinh nghiệm lưu trữ, sửa đổi và phân phối các mô-đun, chương trình và ý tưởng mới một cách an toàn. Các nhà phát triển có thể phát triển các mô-đun riêng tư trên các kho lưu trữ Git riêng và phân phối các kho này một cách an toàn trong một mạng riêng sử dụng npm. Do đó, cộng đồng

của người dùng và nhà phát triển Node đã nhanh chóng mở rộng, một số thành viên đạt được danh tiếng lớn. Nếu bạn xây dựng nó, họ sẽ đến.

Node đã dỡ bỏ rào cản truy cập hệ thống đối với một nhóm lớn các lập trình viên lành nghề, đột ngột giải phóng tài năng bị dồn nén vào một khối lượng trống, mang đến cơ hội cho hệ sinh thái mà một dự án mới phổ biến cần nhiều cải tiến về cơ sở hạ tầng mang lại. Vấn đề là ở đây: Node đã hợp nhất cơ hội đồng thời với các sự kiện JavaScript gốc; API được thiết kế tuyệt vời của nó cho phép người dùng mô hình lập trình nổi tiếng tận dụng lợi thế của I / O đồng thời cao. Nếu bạn thưởng cho họ, họ sẽ đến.

Node đã dỡ bỏ rào cản truy cập mạng đối với một nhóm lớn các nhà phát triển JavaScript có công việc và tham vọng đã bắt đầu phát triển nhanh hơn hộp cát nhỏ có sẵn cho các nhà phát triển khách hàng. Không nên quên rằng khoảng thời gian kéo dài từ khi JavaScript ra đời vào năm 1995 đến nay là gần 20 năm. Gần một thế hệ các nhà phát triển đã phải vật lộn để cố gắng triển khai các ý tưởng mới cho các ứng dụng mạng trong một môi trường phát triển theo hướng sự kiện được biết đến, thậm chí được xác định bởi những hạn chế của nó. Qua một đêm, Node đã loại bỏ những hạn chế đó. Nếu bạn dọn đường, chúng sẽ đến.

Node cung cấp một cách dễ dàng để xây dựng các chương trình mạng có thể mở rộng, nơi mà I / O mạng không còn là điểm nghẽn. Sự thay đổi thực sự không phải là từ một hệ thống phổ biến khác sang Node — nó tránh xa ý tưởng rằng các tài nguyên phức tạp và đắt tiền là cần thiết để xây dựng và duy trì các ứng dụng hiệu quả đòi hỏi sự đồng thời có thể bùng nổ. Nếu một kiến trúc mạng linh hoạt và có thể mở rộng có thể đạt được với chi phí rẻ, các tài nguyên được giải phóng có thể được hướng đến giải quyết các thách thức phần mềm cấp bách khác, chẳng hạn như lọc dữ liệu song song, mở rộng các trò chơi nhiều người chơi, xây dựng nền tảng giao dịch thời gian thực hoặc cộng tác biên tập tài liệu, thậm chí triển khai mã trực tiếp thay đổi trong hệ thống nóng. Sự tự tin tạo nên sự tiến bộ. Nếu bạn làm cho nó dễ dàng, họ sẽ đến.

Node xuất hiện vào thời điểm khi những trang web động xây dựng đó bắt đầu chạy trước những hạn chế của các máy chủ không được trang bị để thực hiện trơn tru nhiều yêu cầu nhỏ, đồng thời. Kiến trúc sư phần mềm bây giờ phải giải quyết một số vấn đề thú vị: các quy tắc của "thời gian thực" - người dùng sẽ hài lòng với "sớm" hay "bây giờ" là phản hồi đúng duy nhất? Và, cách tốt nhất để thiết kế hệ thống có trách nhiệm đáp ứng những mong muốn của người dùng là gì?

Trong chương này, chúng ta sẽ tìm hiểu ba kỹ thuật tiêu chuẩn có sẵn cho các nhà phát triển sử dụng khi xây dựng các ứng dụng mạng thời gian thực: AJAX, WebSockets

Sự kiện do máy chủ gửi (SSE). Mục tiêu của chúng tôi cho chương này là tìm hiểu những lợi ích và hạn chế của từng kỹ thuật này và triển khai từng kỹ thuật với Node. Hãy nhớ rằng chúng tôi đang hướng tới việc đạt được một kiến trúc nhất quán phản ánh thiết kế luồng sự kiện của Node, chúng tôi cũng sẽ xem xét mức độ tốt của mỗi kỹ thuật cho vay chính nó để biểu diễn như một luồng có thể đọc, có thể ghi hoặc song công.

Chúng tôi sẽ kết thúc chương này với việc xây dựng một trình soạn thảo mã cộng tác, điều này sẽ chứng minh các cơ hội mà Node cung cấp cho những người tìm cách xây dựng phần mềm nhóm thời gian thực. Khi bạn làm việc theo cách của mình thông qua các ví dụ và xây dựng các ứng dụng của riêng mình, đây là một số câu hỏi đáng để tự hỏi:

Khối lượng tin nhắn tôi dự kiến sẽ giao dịch mỗi giây là bao nhiêu? Dự kiến có bao nhiêu máy khách được kết nối đồng thời vào thời gian cao điểm và thấp điểm?

Kích thước và hình dạng trung bình của các thông điệp được truyền đi là bao nhiêu? Lớn như thế nào? Phức tạp như thế nào?

Tôi có thể chấp nhận sự cố liên lạc không thường xuyên hoặc tin nhắn bị rớt nếu sự giảm giá này khiến tôi có độ trễ trung bình thấp hơn không?

Tôi có thực sự cần giao tiếp hai chiều hay một bên chịu trách nhiệm về gần như tất cả khối lượng tin nhắn? Tôi có cần một giao diện giao tiếp phức tạp không?

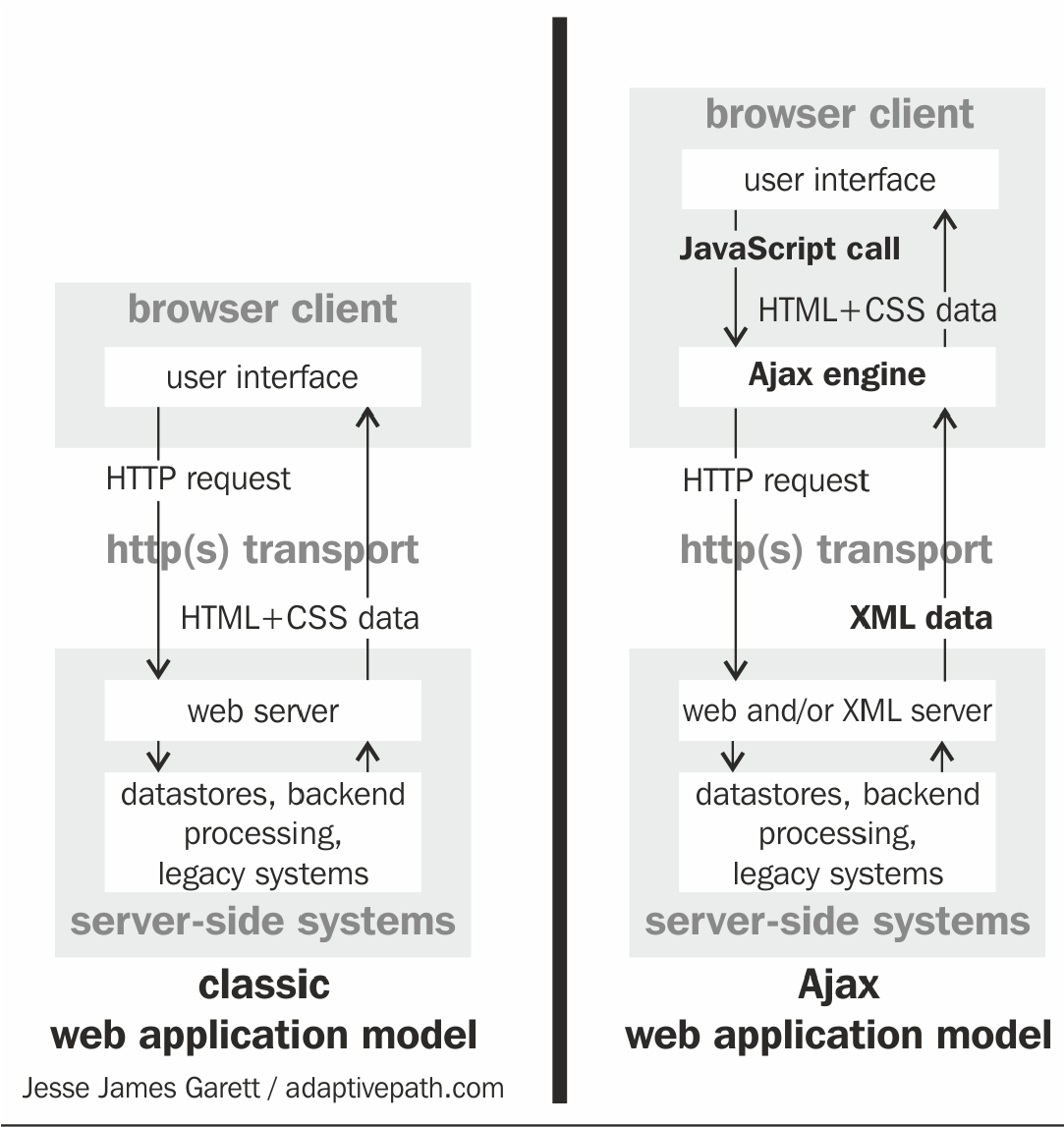
Ứng dụng của tôi sẽ chạy trong những loại mạng nào? Sẽ có là máy chủ proxy giữa máy khách và máy chủ Node của tôi? Những giao thức nào được hỗ trợ?

Tôi cần một giải pháp phức tạp hay sẽ đơn giản và dễ hiểu, thậm chí hơi chậm, các giải pháp mang lại những lợi ích khác về lâu dài?

## 6.1. Giới thiệu AJAX

Năm 2005, Jesse James Garrett xuất bản một bài báo trong đó anh cố gắng cô đọng những thay đổi mà anh ấy đã thấy theo cách mà các trang web được thiết kế thành một khuôn mẫu. Sau khi nghiên cứu xu hướng này, Garrett đề xuất rằng các trang cập nhật động đại diện cho một làn sóng phần mềm mới, giống như phần mềm máy tính để bàn, và ông đặt ra từ viết tắt "AJAX" để mô tả khái niệm công nghệ thúc đẩy sự chuyển động nhanh chóng như vậy đối với "ứng dụng web".

Đây là sơ đồ anh ấy sử dụng để chứng minh mô hình chung:



Bài báo gốc có thể được tìm thấy tại http://www.adaptivepath.com/ ý tưởng / ajax-phương pháp tiếp cận mới-ứng dụng web.

Sơ đồ "công cụ Ajax" mà Garrett nói đến, trên thực tế đã tồn tại trong hầu hết các trình duyệt phổ biến vào năm 2000, và thậm chí còn sớm hơn ở một số trình duyệt. Triển khai JavaScript của XMLHttpRequest (XHR) đối tượng trong các trình duyệt này đã cung cấp cho các trang web khả năng yêu cầu "khối" HTML hoặc dữ liệu khác từ máy chủ. Cập nhật một phần có thể được áp dụng động cho một trang web, tạo cơ hội cho các loại giao diện người dùng mới. Ví dụ: ảnh mới nhất từ một sự kiện có thể xuất hiện một cách kỳ diệu với người dùng mà không cần người dùng đó chủ động yêu cầu làm mới trang hoặc nhấp vào nút "Ảnh tiếp theo".

Quan trọng hơn, Garrett cũng hiểu cách thế giới đồng bộ, không trạng thái của Internet "cũ" đang trở thành một thế giới không đồng bộ, trạng thái như thế nào. Cuộc trò chuyện giữa máy khách và máy chủ không còn bị trật bánh do mất trí nhớ đột ngột và có thể tiếp tục hữu ích trong thời gian dài hơn, chia sẻ thông tin ngày càng hữu ích. Garret coi đây là một sự chuyển đổi sang một thế hệ phần mềm mạng mới.

## 6.2. Trả lời cuộc gọi

Nếu các thay đổi có thể được đưa vào ứng dụng web mà không yêu cầu xây dựng lại toàn bộ trạng thái và hiển thị trạng thái, thì việc cập nhật thông tin khách hàng sẽ trở nên rẻ hơn. Máy khách và máy chủ có thể nói chuyện thường xuyên hơn, thường xuyên trao đổi thông tin. Máy chủ có thể nhận ra, ghi nhớ và phản hồi ngay lập tức với mong muốn của khách hàng, được hỗ trợ bởi các giao diện phản ứng thu thập các hành động của người dùng và phản ánh tác động của những hành động đó trong giao diện người dùng trong thời gian thực.

Với AJAX, việc xây dựng môi trường đa người dùng hỗ trợ cập nhật thời gian thực cho chế độ xem của từng khách hàng về trạng thái ứng dụng tổng thể liên quan đến việc thăm dò máy chủ thường xuyên bằng cách khách hàng kiểm tra các bản cập nhật quan trọng:

Thăm dò ý kiến

KHÁCH HÀNG

MÁY CHỦ NODE

THỜI GIAN

Trò chuyện mạng là liên tục và thường dư thừa.

Hạn chế đáng kể của phương pháp thăm dò ý kiến tiểu bang này là nhiều yêu cầu trong số này sẽ không có kết quả. Khách hàng đã trở thành một kỷ lục bị phá vỡ, liên tục yêu cầu cập nhật trạng thái bất kể những cập nhật đó có sẵn hay sắp có. Khi một ứng dụng dành thời gian hoặc năng lượng để thực hiện các tác vụ không cần thiết, sẽ tồn tại một số lợi ích rõ ràng cho người dùng hoặc nhà cung cấp (hoặc cả hai) bù đắp chi phí này. Ngoài ra, mỗi cuộc gọi vô ích làm tăng thêm chi phí xây dựng sau đó phá bỏ các kết nối HTTP.

Một hệ thống như vậy chỉ có thể chụp ảnh nhanh trạng thái theo khoảng thời gian định kỳ và vì khoảng thời gian thăm dò đó có thể tăng lên vài giây trong nỗ lực giảm bớt cuộc trò chuyện dư thừa trên mạng, nhận thức của chúng ta về những thay đổi trạng thái có thể bắt đầu xuất hiện, chỉ chậm hơn một chút so với Tin tức.

Chúng ta đã thấy một giải pháp tốt hơn trong chương trước — thăm dò ý kiến dài, kỹ thuật cho phép máy chủ giữ kết nối máy khách cho đến khi có dữ liệu mới:

Thăm dò ý kiến dài

KHÁCH HÀNG

MÁY CHỦ NODE

THỜI GIAN

Một kết nối được giữ bởi một máy chủ cho đến khi dữ liệu sẵn sàng để phát. Khi nhận

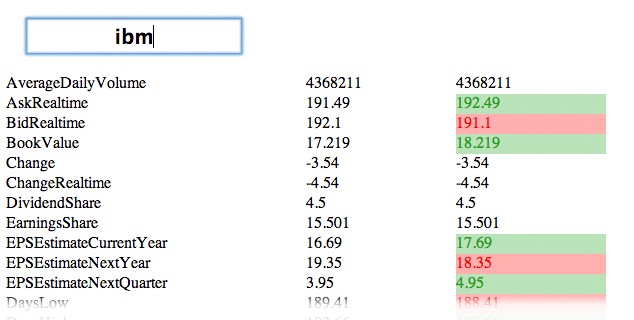
được dữ liệu mới, khách hàng sẽ thiết lập lại kết nối.

Kỹ thuật AJAX cải tiến này không hoàn toàn thoát khỏi chi phí xây dựng và phá vỡ các kết nối mạng, nhưng vẫn giảm đáng kể số lượng các hoạt động tốn kém như vậy. Nói chung, AJAX không thể cung cấp giao diện sự kiện mượt mà, giống như dòng, đòi hỏi rất nhiều dịch vụ tham gia để duy trì trạng thái vì các kết nối thường xuyên bị hỏng và sau đó được thiết lập lại.

Tuy nhiên, AJAX vẫn là một lựa chọn thực sự cho một số ứng dụng, đặc biệt là các ứng dụng đơn giản, nơi mà khoảng thời gian bỏ phiếu lý tưởng được khá nhiều người biết đến, mỗi cuộc thăm dò là một cơ hội tốt để thu thập các kết quả hữu ích. Hãy sử dụng Node để xây dựng một máy chủ có thể giao tiếp với dịch vụ báo cáo chứng khoán và xây dựng một ứng dụng khách thăm dò yêu cầu định kỳ máy chủ này kiểm tra các thay đổi và báo cáo chúng.

## 6.3. Tạo mã cổ phiếu

Cuối cùng, chúng tôi sẽ tạo một ứng dụng cho phép khách hàng chọn một cổ phiếu và theo dõi những thay đổi trong các điểm dữ liệu liên quan đến cổ phiếu đó, chẳng hạn như giá của nó và để làm nổi bật những thay đổi tích cực hoặc tiêu cực:



Để tạo ra khách hàng, công việc của chúng tôi là tối thiểu. Chúng tôi chỉ cần thăm dò máy chủ của mình vài giây một lần hoặc lâu hơn, cập nhật giao diện của chúng tôi để phản ánh bất kỳ thay đổi dữ liệu nào. Hãy sử dụng jQuery làm nhà cung cấp thư viện AJAX của var fetch = function() {

$.getJSON("/service", function(data) {

// Do something with data updateDisplay(data);

// Call again in 5 seconds setTimeout(fetch, 5000);

});

} fetch();

Máy chủ Node sẽ nhận được yêu cầu cập nhật này, thực hiện một số I / O (kiểm tra cơ sở dữ liệu, gọi một dịch vụ bên ngoài) và trả lời bằng dữ liệu, mà khách hàng có thể sử dụng.

Trong ví dụ của chúng tôi, Node sẽ được sử dụng để kết nối với Yahoo! Ngôn ngữ truy vấn (YQL) dịch vụ mà Yahoo! mô tả theo cách này:

Yahoo! Ngôn ngữ truy vấn là một ngôn ngữ giống SQL diễn đạt cho phép bạn truy vấn, lọc và kết hợp dữ liệu trên các dịch vụ Web. Với YQL, các ứng dụng chạy nhanh hơn với ít dòng mã hơn và dấu chân mạng nhỏ hơn.

Ví dụ, nếu tôi muốn đọc báo giá cổ phiếu mới nhất từ Yahoo! Tài chính, truy vấn của tôi có thể giống như sau:

http://query.yahooapis.com/v1/public/yql?q=select \* từ yahoo. Financial.quotes trong đó ký hiệu bằng ("MSFT")

Chúng tôi sẽ xây dựng một máy chủ Node lắng nghe các khách hàng yêu cầu cập nhật dữ liệu cho một ký hiệu chứng khoán nhất định, chẳng hạn như "IBM". Máy chủ Node sau đó sẽ tạo một truy vấn YQL cho biểu tượng cổ phiếu đó và thực hiện truy vấn đó qua http.get, đóng gói dữ liệu nhận được một cách độc đáo cho ứng dụng khách đang gọi và gửi lại.

Gói này cũng sẽ được chỉ định một callIn thuộc tính, cho biết số của mili giây khách hàng nên đợi trước khi gọi lại. Đây là một kỹ thuật hữu ích cần nhớ, vì máy chủ dữ liệu chứng khoán của chúng tôi sẽ có ý tưởng tốt hơn về điều kiện lưu lượng truy cập và tần suất cập nhật so với máy khách. Thay vì khách hàng kiểm tra một cách mù quáng theo một lịch trình cố định, máy chủ của chúng tôi có thể hiệu chỉnh lại tần suất này sau mỗi cuộc gọi, thậm chí yêu cầu khách hàng ngừng gọi!

Vì thiết kế này, đặc biệt là thiết kế trực quan, có thể được thực hiện thông qua bất kỳ cách nào, chúng tôi sẽ chỉ đơn giản xem xét chức năng cốt lõi cần thiết cho khách hàng của chúng tôi, có trong phần sau lấy phương pháp var fetch = function() { var symbol = $("#symbol").val();

$.getJSON("/?symbol=" + symbol, function(data) { if(!data.callIn) { return;

}

setTimeout(fetch, data.callIn); if(data.error) {

return console.error(data.error);

}

var quote = data.quote;

var keys = fetchNumericFields(quote);

…

updateDisplay(symbol, quote, keys);

});

};

Người dùng trên trang này nhập các ký hiệu chứng khoán vào hộp nhập liệu có ID # Biểu tượng. Dữ liệu này sau đó được tìm nạp từ dịch vụ dữ liệu của chúng tôi. Trong đoạn mã trước, chúng ta thấy lệnh gọi dịch vụ được thực hiện qua $. getJSON phương thức jQuery, dữ liệu JSON đang được nhận và setTimeout tài sản đang được thiết lập bằng cách sử dụng callIn khoảng thời gian được gửi lại bởi Node.

Máy chủ của chúng tôi chịu trách nhiệm môi giới cuộc gọi khách hàng ở trên với Yahoo! Dịch vụ dữ liệu tài chính. Giả sử rằng chúng tôi đã định cấu hình máy chủ đúng cách để nhận thành công các ký hiệu chứng khoán từ máy khách, chúng tôi cần mở kết nối HTTP đến dịch vụ YQL, đọc bất kỳ phản hồi nào và trả lại dữ liệu này:

http.get(query, function(res) { var data = ""; res

.on('readable', function() { var d;

while(d = this.read()) { data += d;

}

})

.on('end', function() { var out = {}; try {

data = JSON.parse(data); out.quote = data.query.results.quote; out.callIn = 5000;

} catch(e) { out = {

error: "Received empty data set", callIn: 10000

};

}

response.writeHead(200, {

"Content-type" : "application/json"

});

response.end(JSON.stringify(out));

});

}).on('error', function(e) { response.writeHead(200, {

"Content-type" : "application/json"

});

response.end(JSON.stringify({ error: "System Error", callIn: null

}));

});

Ở đây, chúng ta thấy một ví dụ điển hình về lý do tại sao nên để máy chủ, người quan sát trạng thái chính, điều chỉnh tần suất mà khách hàng thăm dò ý kiến. Nếu nhận được đối tượng dữ liệu thành công, chúng tôi đặt khoảng thời gian thăm dò ( callIn) đến khoảng năm giây. Nếu xảy ra lỗi, chúng tôi tăng độ trễ đó lên 10 giây. Có thể dễ dàng thấy chúng tôi có thể làm nhiều hơn, có lẽ, điều chỉnh các kết nối hơn nữa nếu lỗi lặp lại xảy ra. Do đó, thường sẽ có giới hạn về tốc độ ứng dụng có thể thực hiện yêu cầu đối với dịch vụ bên ngoài (chẳng hạn như giới hạn số lượng cuộc gọi có thể được thực hiện trong một giờ), đây cũng là một kỹ thuật hữu ích để đảm bảo rằng ứng dụng không đổi bỏ phiếu không vượt quá các giới hạn tỷ lệ này.

AJAX là kỹ thuật ban đầu để tạo các ứng dụng thời gian thực. Nó vẫn hữu ích trong một số trường hợp, nhưng đã được thay thế bằng các phương tiện vận chuyển hiệu quả hơn. Khi chúng ta rời khỏi phần này, hãy ghi nhớ một số ưu điểm và nhược điểm của việc thăm dò ý kiến:

**Ưu điểm**

**Nhược điểm**

|  |  |
| --- | --- |
| Lý thuyết và thực hành của REST có sẵn, cho phép giao tiếp chuẩn hơn  Không cần bất kỳ máy chủ giao thức đặc biệt nào, dễ dàng thực hiện thăm dò bằng máy chủ HTTP tiêu chuẩn  HTTP nổi tiếng và được triển khai nhất quán | Việc tạo và ngắt kết nối dẫn đến chi phí cho độ trễ mạng, đặc biệt nếu được thực hiện thường xuyên  Khách hàng phải yêu cầu dữ liệu; máy chủ không thể đơn phương cập nhật máy khách khi dữ liệu mới đến  Thậm chí thăm dò ý kiến dài nhân đôi lưu lượng  mạng cần thiết để duy trì kết nối liên tục |

Dữ liệu bị đẩy và kéo một cách mù quáng, thay vì được phát và nghe trên các kênh một cách suôn sẻ

Bây giờ chúng ta hãy chuyển sang thảo luận về một số giao thức mới hơn, một phần được thiết kế để giải quyết một số vấn đề mà chúng tôi đã tìm thấy với AJAX: WebSockets và SSE.

## 6.4. Giao tiếp hai chiều với Socket.IO

Chúng ta đã quen thuộc với ổ cắm là gì. Đặc biệt, chúng ta biết cách thiết lập và quản lý các kết nối TCP socket bằng Node, cũng như cách truyền dữ liệu qua chúng theo hai chiều hoặc một chiều.

W3C đã đề xuất một API socket cho phép các trình duyệt giao tiếp với máy chủ socket qua một kết nối liên tục. Socket.IO là một thư viện tạo điều kiện thuận lợi cho việc thiết lập các kết nối socket liên tục cho những người phát triển với Node, cung cấp cả máy chủ socket dựa trên Node và một lớp mô phỏng cho các trình duyệt không hỗ trợ WebSocket API nguyên bản.

Trước tiên, hãy cùng tìm hiểu sơ lược về cách triển khai API WebSocket gốc và cách xây dựng một máy chủ ổ cắm hỗ trợ giao thức này bằng Node. Sau đó, chúng tôi sẽ xây dựng một ứng dụng vẽ cộng tác bằng cách sử dụng socket.io với Node.

Bạn có thể tìm thấy thông số kỹ thuật đầy đủ cho API WebSocket tại http://www.w3.org/TR/websockets/.

Tài liệu và hướng dẫn cài đặt Socket.IO can được tìm thấy tại http://socket.io/.

## 6.5. Sử dụng API WebSocket

Giao tiếp socket hiệu quả, chỉ xảy ra khi một trong các bên có điều gì đó hữu ích để nói:

ổ cắm

KHÁCH HÀNG

MÁY CHỦ NODE

THỜI GIAN

Giao tiếp là hai chiều và chỉ xảy ra khi cần thiết

Mô hình nhẹ này là một lựa chọn tuyệt vời cho các ứng dụng yêu cầu truyền thông điệp tần số giữa máy khách và máy chủ, chẳng hạn như trong trò chơi mạng nhiều người chơi hoặc phòng trò chuyện.

Theo W3C, API WebSocket nhằm "cho phép các ứng dụng web duy trì liên lạc hai chiều với các quy trình phía máy chủ". Giả sử rằng chúng tôi đã thiết lập một máy chủ socket chạy tại localhost: 8080, chúng tôi có thể kết nối với máy chủ này từ một trình duyệt chứa dòng JavaScript sau:

var conn = new WebSocket ("ws: // localhost: 8080", ['json', 'xml']);

WebSocket mong đợi hai đối số: một URL có tiền tố là lược đồ URI ws: //, và danh sách giao thức con tùy chọn, có thể là một mảng hoặc một chuỗi giao thức đơn lẻ mà máy chủ có thể triển khai.

Để thiết lập kết nối ổ cắm an toàn, hãy sử dụng wss: // tiếp đầu ngữ. Đối với máy chủ HTTPS, bạn cần sử dụng chứng chỉ SSL khi khởi động máy chủ.

Khi một yêu cầu socket được thực hiện, các sự kiện kết nối mở, đóng, lỗi, và thông điệp có thể được xử lý bởi một trình duyệt:

<head>

<title></title>

<script>

var conn = new WebSocket("ws://localhost:8080",'json'); conn.onopen = function() {

conn.send('Hello from the client!');

};

conn.onerror = function(error) { console.log('Error! ' + error);

};

conn.onclose = function() {

console.log("Server has closed the connection!");

};

conn.onmessage = function(msg) { console.log('Received: ' + msg.data);

};

</script>

</head>

Để triển khai máy chủ WebSocket trong Node, chúng tôi sẽ sử dụng ws mô-đun do Einar Otto Stangvik tạo, có thể tải xuống từ https://github.com/ einaros / ws. Sau khi cài đặt ws bằng npm ( npm cài đặt ws), thiết lập một máy chủ ổ cắm Node rất đơn giản:

var SocketServer = require('ws').Server; var wss = new SocketServer({port: 8080}); wss.on('connection', function(ws) { ws.on('message', function(message) { console.log('received: %s', message);

});

ws.send("You've connected!"); });

Ở đây, chúng ta thấy cách máy chủ đơn giản lắng nghe kết nối và thông điệp sự kiện từ khách hàng, phản hồi khi cần thiết. Nếu có nhu cầu chấm dứt kết nối (có thể, nếu máy khách mất quyền), máy chủ có thể chỉ cần phát ra đóng sự kiện mà khách hàng có thể lắng nghe:

ws.close ();

Sơ đồ chung cho một ứng dụng sử dụng API WebSocket để tạo giao tiếp hai chiều do đó trông giống như sau:

MÁY CHỦ NODE

máy chủ ổ cắm

ws: //

TRÌNH DUYỆT

ws: //

TRÌNH DUYỆT

ws: //

TRÌNH DUYỆT

Việc triển khai trình duyệt WebSocket gốc ở đây được sử dụng để giao tiếp với máy chủ socket tùy chỉnh của chúng tôi, máy chủ này sẽ đưa ra các yêu cầu từ máy khách cũng như phát dữ liệu hoặc thông tin mới cho khách hàng khi cần thiết.

## 6.6. Socket.IO

Như đã đề cập trước đó, Socket.IO nhằm mục đích cung cấp một lớp mô phỏng sẽ sử dụng triển khai WebSocket gốc trong các trình duyệt hỗ trợ nó, hoàn nguyên về các phương thức khác (chẳng hạn như long-polling) để mô phỏng API gốc trong các trình duyệt không hỗ trợ. Đây là một thực tế quan trọng cần ghi nhớ: những lợi ích thực sự của WebSockets sẽ không tồn tại cho tất cả các trình duyệt máy khách.

Tuy nhiên, Socket.IO thực hiện rất tốt công việc che giấu sự khác biệt của trình duyệt và vẫn là một lựa chọn tốt khi luồng điều khiển được tạo sẵn bởi các socket là một mô hình giao tiếp mong muốn cho ứng dụng của bạn.

Trong triển khai WebSocket đã đưa ra trước đó, rõ ràng là máy chủ ổ cắm độc lập với bất kỳ tệp khách hàng cụ thể nào. Chúng tôi đã viết một số JavaScript để thiết lập kết nối WebSocket trên máy khách, chạy độc lập máy chủ ổ cắm bằng Node. Không giống như triển khai gốc này, Socket.IO yêu cầu một thư viện khách tùy chỉnh được cài đặt trên một máy chủ ngoài socket.io mô-đun máy chủ:

MÁY CHỦ NODE

socket.io

socket.io

TRÌNH DUYỆT

socket.io

TRÌNH DUYỆT

socket.io

TRÌNH DUYỆT

Socket.IO có thể được cài đặt bằng trình quản lý gói npm:

npm install socket.io

Việc thiết lập ghép nối ổ cắm máy khách / máy chủ rất đơn giản. Trên máy chủ:

var io = require('socket.io').listen(8080); io.sockets.on('connection', function (socket) { socket.emit('broadcast', { message: 'Hi!' }); socket.on('clientmessage', function (data) { console.log("Client said" + data);

});

});

On the client:

<script src="/socket.io/socket.io.js"></script>

<script>

var socket = io.connect('http://localhost'); socket.on('broadcast', function(data) { console.log("Server sent: " + data);

socket.emit('clientmessage', { message: 'ohai!' });

});

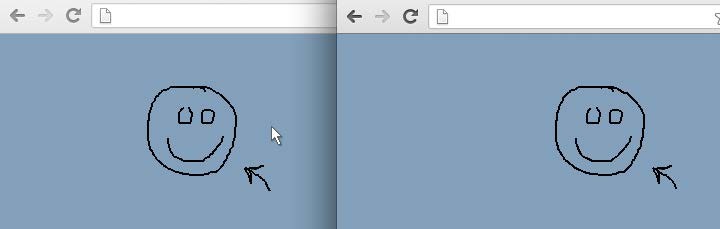
</script>

Chúng tôi có thể thấy cách cả máy khách và máy chủ đang sử dụng cùng một tệp, socket.io.js.

Máy chủ sử dụng Socket.IO xử lý việc cung cấp socket.io.js gửi cho khách hàng tự động khi được yêu cầu Nó cũng sẽ nhảy ra Socket.IOAPI giống với Node tiêu chuẩn đến mức nào EventEmitter giao diện.

## 6.7. Vẽ cộng tác

Hãy tạo một ứng dụng vẽ cộng tác bằng Socket.IO và Node. Chúng tôi muốn tạo một canvas trống sẽ hiển thị đồng thời tất cả "tác phẩm bút" đang được thực hiện bởi các máy khách được kết nối:



Từ cuối máy chủ, có rất ít việc phải làm. Khi máy khách cập nhật tọa độ bằng cách di chuyển chuột của họ, máy chủ chỉ cần truyền phát thay đổi này cho tất cả các máy khách được kết nối:

io.sockets.on('connection', function(socket) { var id = socket.id;

socket.on('mousemove', function(data) { data.id = id;

socket.broadcast.emit('moving', data);

});

socket.on('disconnect', function() {

socket.broadcast.emit('clientdisconnect', id);

});

});

Socket.IO tự động tạo một ID duy nhất cho mỗi kết nối socket. Chúng tôi sẽ chuyển ID này bất cứ khi nào xảy ra các sự kiện rút thăm mới, cho phép các máy khách tiếp nhận theo dõi cách nhiều người dùng được kết nối. Tương tự, khi một máy khách ngắt kết nối, tất cả các máy khách khác được hướng dẫn xóa các tham chiếu của họ đến máy khách này. Sau đó, chúng ta sẽ xem cách ID này được sử dụng trong giao diện người dùng ứng dụng để duy trì một con trỏ đại diện cho tất cả các máy khách được kết nối.

Đây là một ví dụ tuyệt vời về cách đơn giản để tạo ra nhiều người dùng, Di chuyển vào các ứng dụng mạng một dòng bằng Node và các gói được tạo bởi cộng đồng Node. Hãy chia nhỏ những gì máy chủ này đang làm.

Bởi vì chúng tôi sẽ cần phân phối tệp HTML mà máy khách sẽ sử dụng để vẽ, một nửa thiết lập máy chủ liên quan đến việc tạo máy chủ tệp tĩnh. Để thuận tiện, chúng tôi sẽ sử dụng gói node-static, có thể được tải xuống từ https://github.com/ cloudhead / node-static. Việc triển khai của chúng tôi sẽ phục vụ index.html gửi cho bất kỳ khách hàng nào kết nối.

Việc triển khai Socket.IO của chúng tôi hy vọng sẽ nhận được di chuột sự kiện từ máy khách và nhiệm vụ duy nhất của nó là gửi cho tất cả các máy khách được kết nối các tọa độ mới này, nó thực hiện bằng cách phát ra di chuyển sự kiện thông qua nó phát sóng phương pháp. Khi một khách hàng thay đổi trạng thái canvas bằng cách vẽ một đường thẳng, tất cả khách hàng sẽ nhận được thông tin cần thiết để cập nhật chế độ xem của họ về trạng thái canvas trong thời gian thực.

Với lớp giao tiếp được xây dựng, bây giờ chúng ta phải tạo các chế độ xem máy khách. Như đã đề cập, mỗi khách hàng sẽ tải một index.html tệp chứa phần tử canvas cần thiết và JavaScript cần thiết để lắng nghe di chuyển sự kiện, cũng như socket.io

Emitter phát sóng các sự kiện rút khách hàng đến máy chủ của chúng tôi:

<style type="text/css">

/\* CSS styling for the pointers and canvas \*/

</style>

<script src="/socket.io/socket.io.js"></script>

<script src="/script.js"></script>

</head>

<body>

<div id="pointers"></div>

<canvas id="canvas" width="2000" height="1000"></canvas>

</body>

A con trỏ phần tử được tạo để giữ các biểu diễn hiển thị cho con trỏ

của tất cả các máy khách được kết nối, sẽ cập nhật khi các máy khách được kết nối di chuyển con trỏ và / hoặc vẽ thứ gì đó.

Trong script.js , trước tiên chúng tôi thiết lập trình xử lý sự kiện trên Tranh sơn dầu phần tử, theo dõi sự kết hợp của Di chuột xuống và di chuột sự kiện chỉ ra một hành động hòa. Lưu ý cách chúng tôi tạo bộ đệm thời gian 50 mili giây, trì hoãn việc phát sóng mỗi sự kiện vẽ, giảm một chút độ phân giải của bản vẽ nhưng tránh quá nhiều sự kiện mạng:

var socket = io.connect("/"); var prev = {};

var canvas = document.getElementById('canvas'); var context = canvas.getContext('2d');

var pointerContainer = document.getElementById("pointers"); var pointer = document.createElement("div"); pointer.setAttribute("class", "pointer"); var drawing = false; var clients = {}; var pointers = {};

var drawLine = function(fromx, fromy, tox, toy) { context.moveTo(fromx, fromy); context.lineTo(tox, toy); context.stroke();

}; canvas.onmouseup = canvas.onmousemove = canvas.onmousedown = function(e) { switch(e.type) { case "mouseup": drawing = false; break; case "mousemove":

if(now() - lastEmit > 50) { socket.emit('mousemove', {

'x' : e.pageX,

'y' : e.pageY,

'drawing': drawing

}); lastEmit = now();

} if(drawing) {

drawLine(prev.x, prev.y, e.pageX, e.pageY);

prev.x = e.pageX; prev.y = e.pageY;

} break; case "mousedown": drawing = true; prev.x = e.pageX; prev.y = e.pageY; break; default: break;

}

}

Bất cứ khi nào một hành động hòa xảy ra (sự kết hợp của một Di chuột xuống và một di chuột sự kiện), chúng tôi vẽ đường được yêu cầu trên máy của khách hàng và sau đó phát các tọa độ mới này tới máy chủ Socket.IO của chúng tôi thông qua socket.emit ('mousemove',…), nhớ vượt qua Tôi giá trị của khách hàng vẽ. Đến lượt máy chủ sẽ phát chúng qua socket.broadcast.emit ('di chuyển', dữ liệu), cho phép khách hàng người nghe để vẽ các đường tương đương trên Tranh sơn dầu thành phần:

socket.on('moving', function(data) { if(!clients.hasOwnProperty(data.id)) { pointers[data.id] = pointerContainer.appendChild(pointer.

cloneNode());

}

pointers[data.id].style.left = data.x + "px"; pointers[data.id].style.top = data.y + "px";

if(data.drawing && clients[data.id]){

drawLine(clients[data.id].x, clients[data.id].y, data.x, data.y);

}

clients[data.id] = data; clients[data.id].updated = now(); });

Trong bộ lắng nghe này, một ứng dụng khách sẽ thiết lập một con trỏ ứng dụng khách mới nếu ID khách hàng đã gửi chưa được nhìn thấy trước đó và tạo hoạt ảnh cho cả bản vẽ của một đường và con trỏ khách hàng, tạo ra hiệu ứng của nhiều con trỏ vẽ các đường riêng biệt trong một chế độ xem khách hàng .

Nhớ lại clientdisconnect sự kiện chúng tôi theo dõi trên máy chủ của mình, chúng tôi cũng cho phép khách hàng lắng nghe những ngắt kết nối này, xóa các tham chiếu đến các khách hàng bị mất khỏi cả chế độ xem (con trỏ trực quan) và khách hàng vật:

socket.on("clientdisconnect", function(id) { delete clients[id]; if(pointers[id]) {

pointers[id].parentNode.removeChild(pointers[id]);

}

});

Socket.IO là một công cụ tuyệt vời để xem xét khi xây dựng môi trường tương tác, đa người dùng, nơi cần truyền dữ liệu hai chiều nhanh chóng liên tục.

Bây giờ, hãy xem những ưu và nhược điểm của Socket.IO:

|  |  |
| --- | --- |
| Ưu điểm | Nhược điểm |
| Giao tiếp hai chiều nhanh chóng  cần thiết cho trò chơi thời gian thực,  các công cụ chỉnh sửa cộng tác và các ứng dụng khác  Chi phí thấp hơn so với yêu cầu giao thức HTTP tiêu chuẩn, giảm giá gửi một gói qua mạng | Số lượng kết nối ổ cắm liên tục được phép có thể bị giới hạn ở phía máy chủ hoặc bất kỳ nơi nào ở giữa  Nhiều trình duyệt tự nhiên không hỗ trợ các socket web, buộc các thư viện mô phỏng, chẳng hạn như Socket.IO, phải dùng đến các kỹ thuật bỏ phiếu dài hoặc các kỹ thuật kém hiệu quả khác |
| Bản chất phát trực tuyến của các socket phù hợp về mặt khái niệm với kiến trúc Node — máy khách và máy chủ chỉ đơn giản là chuyển dữ liệu qua lại thông qua các giao diện nhất quán | Yêu cầu máy chủ giao thức tùy chỉnh và thường là thư viện máy khách tùy chỉnh  Nhiều proxy và proxy ngược được biết đến là ổ cắm gây nhiễu  triển khai, dẫn đến mất khách hàng |

Một thư viện socket tuyệt vời khác để phát triển Node là SockJS ( https://github.com/sockjs).

Lắng nghe sự kiện do máy chủ gửi SSE không phức tạp và cụ thể. Chúng sẽ được sử dụng khi phần lớn quá trình truyền dữ liệu tiến hành một chiều từ máy chủ đến máy khách. Một khái niệm truyền thống và tương tự là công nghệ "đẩy". SSE chuyển các tin nhắn văn bản với định dạng đơn giản. Nhiều loại ứng dụng nhận các cập nhật trạng thái ngắn gọn hoặc thay đổi trạng thái dữ liệu một cách thụ động. SSE là một sự phù hợp tuyệt vời cho các loại ứng dụng này.

Giống như WebSocket, SSE cũng loại bỏ các đoạn chat dư thừa của AJAX. Không giống như WebSocket, kết nối SSE chỉ quan tâm đến việc truyền dữ liệu từ máy chủ đến máy khách được kết nối:

Sự kiện do máy chủ gửi

KHÁCH HÀNG

MÁY CHỦ NODE

THỜI GIAN

Mong đợi sự bắt tay ban đầu, giao tiếp là đơn hướng

và chỉ xảy ra khi cần thiết

Máy khách kết nối với máy chủ hỗ trợ SSE bằng cách chuyển EventSource hàm tạo một đường dẫn:

var eventSource = new EventSource ('/ login');

Ví dụ này của EventSource bây giờ sẽ phát ra các sự kiện dữ liệu có thể đăng ký bất cứ khi nào nhận được dữ liệu mới từ máy chủ.

## 6.8. Sử dụng API EventSource

Cách thức mà EventSource các phiên bản phát ra các sự kiện dữ liệu có thể đăng ký bất cứ khi nào dữ liệu mới được nhận từ máy chủ giống như cách Có thể đọc được các luồng phát ra các sự kiện dữ liệu trong Node, như chúng ta có thể thấy trong ứng dụng khách mẫu này:

<script>

var eventSource = new EventSource('/login');

eventSource.addEventListener('message', function(broadcast) {

console.log("got message: " + broadcast);

});

eventSource.addEventListener('open', function() {

console.log("connection opened");

});

eventSource.addEventListener('error', function() { console.log("connection error/closed");

});

</script>

An EventSource instance tạo ra ba sự kiện mặc định:

mở: Khi kết nối được mở thành công, sự kiện này sẽ kích hoạt thông điệp: Trình xử lý được chỉ định cho sự kiện này sẽ nhận một đối tượng có dữ liệu thuộc tính chứa tin nhắn quảng bá lỗi: Điều này kích hoạt bất cứ khi nào xảy ra lỗi máy chủ hoặc máy chủ ngắt kết nối hoặc cắt đứt kết nối của nó với máy khách này

Tạo thành một phần của giao thức HTTP tiêu chuẩn, một máy chủ đáp ứng các yêu cầu SSE yêu cầu cấu hình tối thiểu. Máy chủ sau sẽ chấp nhận EventSource ràng buộc và phát ngày hiện tại cho khách hàng bị ràng buộc mỗi giây:

var http = require("http"); var url = require("url");

http.createServer(function(request, response) { var parsedURL = url.parse(request.url, true); var pathname = parsedURL.pathname; var args = pathname.split("/"); var method = args[1]; if(method === "login") { response.writeHead(200, {

"Content-Type": "text/event-stream",

"Cache-Control": "no-cache",

"Connection": "keep-alive"

});

response.write(":" + Array(2049).join(" ") + "\n"); response.write("retry: 2000\n");

response.on("close", function() { console.log("client disconnected");

});

setInterval(function() {

response.write("data: " + new Date() + "\n\n");

}, 1000); return;

}

}).listen(8080);

Máy chủ này lắng nghe các yêu cầu và chọn những yêu cầu được thực hiện trên đường dẫn / đăng nhập, mà nó diễn giải như một yêu cầu cho một EventSource sự ràng buộc. Thiết lập một EventSource kết nối chỉ đơn giản là vấn đề phản hồi yêu cầu với Loại nội dung tiêu đề của text / event-stream. Ngoài ra, chúng tôi chỉ ra rằng khách hàng của

Kiểm soát bộ nhớ đệm hành vi nên được đặt thành không có bộ nhớ cache, vì chúng tôi mong đợi nhiều tài liệu gốc trên kênh này.

Từ điểm kết nối phản ứng đối tượng của ứng dụng khách này sẽ vẫn là một đường ống mở mà các thông báo có thể được gửi thông qua viết. Hãy xem hai dòng tiếp theo:

response.write (":" + Array (2049) .join ("") + "\ n"); response.write ("thử lại: 2000 \ n");

Lần viết đầu tiên này là điều chỉnh cho tính năng triển khai XHR trong một số trình duyệt, điều này cuối cùng yêu cầu tất cả các luồng SSE phải được bắt đầu bằng một phần đệm 2 KB. Điều này viết hành động chỉ cần xảy ra một lần và không liên quan đến các thông báo tiếp theo.

Để đọc thêm về lý do tại sao phần đệm 2 KB này lại cần thiết, hãy truy cập

http://blogs.msdn.com/b/ieinternals/ archive / 2010/04/06 / comet-streaming-in-internetexplorer-with-xmlhttprequest-and-xdomainrequest. aspx? pageIndex = 1.

Một trong những ưu điểm của SSE là các máy khách sẽ tự động cố gắng kết nối lại với máy chủ, nếu kết nối đó bị ngắt. Số mili giây trước khi thử lại sẽ khác nhau giữa các ứng dụng khách và có thể được kiểm soát bằng cách sử dụng thử lại mà chúng tôi sử dụng ở đây để đặt khoảng thời gian thử lại hai phần nghìn giây.

Cuối cùng, chúng tôi lắng nghe ý kiến của khách hàng đóng sự kiện kích hoạt khi máy khách ngắt kết nối và bắt đầu phát thời gian trong khoảng thời gian một giây:

setInterval (function () { response.write ("data:" + new Date () + "\ n \ n"); }, 1000);

Một trang web có thể liên kết với máy chủ thời gian này và hiển thị thời gian máy chủ hiện tại:

<html>

<head>

<script>

var ev = new EventSource('/login');

ev.addEventListener("message", function(broadcast) { document.getElementById("clock").innerHTML = broadcast.data;

});

</script>

</head>

<body>

<div id="clock"></div>

</body>

</html>

Vì kết nối là một chiều nên có thể thiết lập bất kỳ số lượng dịch vụ nào với tư cách là nhà xuất bản rất dễ dàng, với khách hàng ràng buộc riêng lẻ với các dịch vụ này thông qua EventSource các trường hợp. Ví dụ, giám sát máy chủ có thể đạt được dễ dàng bằng cách sửa đổi máy chủ ở trên sao cho nó định kỳ gửi giá trị của process.memoryUsage (). Như một bài tập, hãy sử dụng SSE để triển khai lại dịch vụ cổ phiếu mà chúng ta đã đề cập trước đó trong phần về AJAX.

## 6.9. Giao thức luồng EventSource

Sau khi máy chủ đã thiết lập kết nối máy khách, giờ đây nó có thể gửi tin nhắn mới qua kết nối liên tục này bất kỳ lúc nào. Những thông báo này bao gồm một hoặc nhiều dòng văn bản, được giới hạn bởi một hoặc một số trong bốn trường sau:

biến cố: Đây là một loại sự kiện. Tin nhắn được gửi không có trường này sẽ kích hoạt chung của khách hàng EventSource xử lý sự kiện cho bất kỳ tin nhắn nào. Nếu được đặt thành một chuỗi, chẳng hạn như "điểm chuẩn", thông điệp người xử lý sẽ không phải được gọi, với việc xử lý được ủy quyền cho một trình xử lý được ràng buộc bằng cách sử dụng EventSource.addEvent

## 6.10. Trình nghe ('điểm cuối'…).

dữ liệu: Đây là tin nhắn đang được gửi. Điều này luôn luôn là của Chuỗi loại, mặc dù nó có thể vận chuyển hữu ích các đối tượng đi qua JSON.stringify ().

Tôi: Nếu được đặt, giá trị này sẽ xuất hiện dưới dạng lastEventID thuộc tính của đối tượng tin nhắn đã gửi. Điều này có thể hữu ích cho việc đặt hàng, sắp xếp và các hoạt động khác trên máy khách.

thử lại: Khoảng thời gian kết nối lại, tính bằng mili giây.

Gửi tin nhắn bao gồm việc soạn các chuỗi có chứa các tên trường liên quan và kết thúc bằng các dòng mới. Đây là tất cả các thông báo hợp lệ:

response.write("id:" + (++message\_counter) + "\n"); response.write("data: I'm a message\n\n"); response.write("retry: 10000\n\n"); response.write("id:" + (++message\_counter) + "\n"); response.write("event: stock\n");

response.write("data: " + JSON.stringify({price: 100, change: -2}) +

"\n\n");

response.write("event: stock\n"); response.write("data: " + stock.price + "\n"); response.write("data: " + stock.change + "\n"); response.write("data: " + stock.symbol + "\n\n"); response.write("data: Hello World\n\n");

Chúng ta có thể thấy rằng nhiều dữ liệu các trường cũng có thể được thiết lập. Một điều quan trọng cần lưu ý là dòng mới kép ("\ n \ n ") được gửi sau sau cùng trường dữ liệu. Các trường trước đó chỉ nên sử dụng một dòng mới.

Mặc định EventSource sự kiện khách hàng ( mở, tin nhắn, và đóng) đủ để mô hình hóa hầu hết các giao diện ứng dụng. Tất cả các chương trình phát sóng từ máy chủ được bắt trong một thông điệp trình xử lý, chịu trách nhiệm định tuyến thông báo hoặc cập nhật ứng dụng khách, giống như cách mà ủy quyền sự kiện sẽ hoạt động khi làm việc với các sự kiện trong DOM bằng JavaScript.

Hệ thống này có thể không lý tưởng trong trường hợp cần nhiều mã nhận dạng thông báo duy nhất, lấn át một chức năng xử lý duy nhất. Chúng ta có thể sử dụng biến cố trường thông báo SSE để tạo tên sự kiện tùy chỉnh có thể được khách hàng ràng buộc riêng lẻ, tách biệt các mối quan tâm một cách gọn gàng.

Ví dụ: nếu hai sự kiện đặc biệt actionA và actionB đang được phát sóng, máy chủ của chúng tôi sẽ cấu trúc chúng như thế này:

event: actionA\n data: Message A here\n\n

event: actionB\n data: Message B here\n\n

Khách hàng của chúng tôi sẽ liên kết với họ theo cách bình thường, như được hiển thị trong đoạn mã sau:

ev.addEventListener("actionA", function(broadcast) { console.log(broadcast.data); });

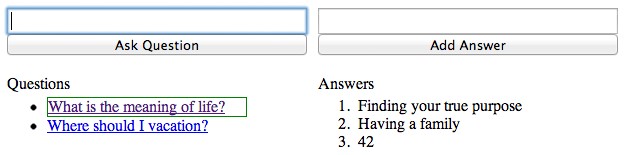
ev.addEventListener("actionB", function(broadcast) { console.log(broadcast.data); });

Trong trường hợp một chức năng xử lý thư đơn lẻ trở nên quá dài hoặc quá phức tạp, hãy xem xét các thư và trình xử lý được đặt tên duy nhất.

Đặt câu hỏi và nhận câu trả lời

Điều gì sẽ xảy ra nếu chúng tôi muốn tạo một giao diện cho sở thích? Hãy xây dựng một ứng dụng cho phép bất kỳ số lượng người nào hỏi và / hoặc trả lời câu hỏi. Người dùng của chúng tôi sẽ tham gia máy chủ cộng đồng, xem danh sách các câu hỏi mở và câu trả lời cho những câu hỏi đó, đồng thời nhận thông tin cập nhật theo thời gian thực bất cứ khi nào câu hỏi hoặc câu trả lời mới được thêm vào. Có hai hoạt động chính để mô hình hóa:

Mỗi khách hàng phải được thông báo bất cứ khi nào khách hàng khác đặt câu hỏi hoặc đăng câu trả lời



• Khách hàng có thể đặt câu hỏi hoặc cung cấp câu trả lời

Mức độ thay đổi lớn nhất sẽ xảy ra ở đâu trong một nhóm lớn những người đóng góp đồng thời cho cộng đồng này?

Bất kỳ khách hàng cá nhân nào cũng có thể đặt một vài câu hỏi hoặc đưa ra một vài câu trả lời. Khách hàng cũng sẽ chọn câu hỏi và hiển thị câu trả lời cho họ. Chúng tôi sẽ chỉ cần đáp ứng một số ít các yêu cầu từ máy khách đến máy chủ, chẳng hạn như khi gửi một câu hỏi hoặc câu trả lời mới đến máy chủ. Hầu hết công việc sẽ là đáp ứng các yêu cầu của khách hàng với dữ liệu (danh sách các câu trả lời cho một câu hỏi) và chuyển trạng thái ứng dụng sang tất cả khách hàng được kết nối (câu hỏi mới được thêm vào; câu trả lời mới được đưa ra). Mối quan hệ một-nhiều tồn tại đối với các máy khách trong các ứng dụng cộng tác như vậy ngụ ý rằng một chương trình phát sóng máy khách duy nhất có thể tạo ra một số lượng phát sóng máy chủ bằng với số lượng máy khách được kết nối — 1 đến 10 K hoặc hơn. SSE rất phù hợp ở đây, vì vậy hãy bắt đầu.

Ba hoạt động chính cho ứng dụng này như sau:

Đặt câu hỏi

Trả lời câu hỏi

Chọn câu hỏi

Một trong hai hành động này sẽ thay đổi trạng thái ứng dụng. Vì trạng thái này phải được phản ánh trên tất cả các máy khách, chúng tôi sẽ lưu trữ trạng thái của ứng dụng trên máy chủ của chúng tôi — tất cả các câu hỏi, câu trả lời và mối quan hệ của khách hàng với các đối tượng dữ liệu này. Chúng tôi cũng sẽ cần xác định duy nhất từng khách hàng. Thông thường, người ta sẽ sử dụng cơ sở dữ liệu để lưu giữ một số thông tin này, nhưng cho mục đích của chúng tôi, chúng tôi sẽ chỉ lưu trữ dữ liệu này trong máy chủ Node của chúng tôi:

var USER\_ID = 1e10; var clients = {}; var clientQMap = {}; var questions = {}; var answers = {}; var removeClient = function(id) { if(id) { delete clients[id]; delete clientQMap[id];

}

}

các đối tượng lưu trữ, chúng tôi cũng sẽ cần phải lưu trữ chính các đối tượng khách — các máy khách được gán một ID duy nhất có thể được sử dụng để tra cứu thông tin (chẳng hạn như ổ cắm của máy khách) khi các chương trình phát sóng được thực hiện.

Chúng tôi chỉ muốn phát dữ liệu câu trả lời cho khách hàng đã bày tỏ sự quan tâm đến câu hỏi cụ thể — vì giao diện người dùng của khách hàng chỉ hiển thị câu trả lời cho một câu hỏi duy nhất, chúng tôi tất nhiên sẽ không truyền phát câu trả lời cho khách hàng một cách bừa bãi. Vì lý do này, chúng tôi giữ một clientQMap đối tượng, ánh xạ một câu hỏi tới tất cả các khách hàng đang nghe câu hỏi đó, bằng ID.

Các removeClient phương thức đơn giản: khi một máy khách ngắt kết nối, phương thức sẽ xóa dữ liệu của nó khỏi nhóm. Chúng ta sẽ gặp lại điều này sau.

Với thiết lập này, tiếp theo, chúng tôi cần xây dựng máy chủ của mình để phản hồi / đăng nhập

đường dẫn, được sử dụng bởi EventSource để lấy kết nối. Dịch vụ này chịu trách nhiệm định cấu hình luồng sự kiện thích hợp cho máy khách, lưu trữ Phản ứng đối tượng để sử dụng sau này và gán cho người dùng một số nhận dạng duy nhất, sẽ được sử dụng cho các yêu cầu của ứng dụng khách trong tương lai để xác định ứng dụng khách và tìm nạp ổ cắm giao tiếp của ứng dụng khách đó:

http.createServer(function(request, response) { var parsedURL = url.parse(request.url, true); var pathname = parsedURL.pathname; var args = pathname.split("/");

// Lose initial null value args.shift(); var method = args.shift(); var parameter = decodeURIComponent(args[0]); var sseUserId = request.headers['\_sse\_user\_id\_']; if(method === "login") { response.writeHead(200, {

"Content-Type": "text/event-stream",

"Cache-Control": "no-cache"

});

response.write(":" + Array(2049).join(" ") + "\n"); // 2kB response.write("retry: 2000\n"); removeClient(sseUserId);

// A very simple id system. You'll need something more secure. sseUserId = (USER\_ID++).toString(36); clients[sseUserId] = response; broadcast(sseUserId, { type : "login", userId : sseUserId

});

broadcast(sseUserId, { type : "questions", questions : questions

});

response.on("close", function() { removeClient(sseUserId); });

// In order to keep the connection alive we send a "heartbeat" every 10 seconds.

https://bugzilla.mozilla.org/show\_bug.cgi?id=444328 setInterval(function() {

broadcast(sseUserId, new Date().getTime(), "ping");

}, 10000); return; }

Sau khi thiết lập các tham số yêu cầu, máy chủ của chúng tôi sẽ kiểm tra yêu cầu cho một \_ sse\_user\_id\_ tiêu đề, là chuỗi duy nhất được gán cho người dùng trong / đăng nhập ban đầu EventSource trói buộc:

sseUserId = (USER\_ID++).toString(36); clients[sseUserId] = response;

ID này sau đó được gửi đến khách hàng thông qua một chương trình phát sóng ngay lập tức, một cơ hội mà chúng tôi sử dụng để gửi hàng loạt câu hỏi hiện tại:

broadcast(sseUserId, sseUserId, "login");

Máy khách bây giờ chịu trách nhiệm chuyển ID này bất cứ khi nào nó thực hiện cuộc gọi. Bằng cách lắng nghe / đăng evSource.addEventListener('login', function(broadcast) {

USER\_ID = JSON.parse(broadcast.data);

});

var xhr = new XMLHttpRequest(); xhr.open("POST", "/...");

xhr.setRequestHeader('\_sse\_user\_id\_', USER\_ID); ...

. . .

Nhớ lại rằng chúng tôi vừa tạo một luồng sự kiện một chiều từ máy chủ của chúng tôi đến máy khách của chúng tôi. Kênh này được sử dụng để giao tiếp với khách hàng — không phải response.end ()

hoặc tương tự. Các phát sóng phương pháp, được tham chiếu trong / đăng nhập, hoàn thành nhiệm vụ truyền phát sự kiện luồng như được hiển thị trong đoạn mã sau:

var broadcast = function (toId, msg, eventName) { if (toId === "\*") { for (var p in client) { broadcast (p, msg);

} trở về; }

var clientSocket = client [toId]; if (! clientSocket) {

trở về;

}

eventName && clientSocket.write ("event:" + eventName + "\ n"); clientSocket.write ("id:" + (++ UNIQUE\_ID) + "\ n"); clientSocket.write ("data:" + JSON.stringify (msg) + "\ n \ n");

}

Quét mã này từ dưới lên. Lưu ý mục đích chính của phát sóng là lấy ID khách hàng, tra cứu luồng sự kiện của khách hàng đó và ghi vào đó, chấp nhận tên sự kiện tùy chỉnh nếu cần. Tuy nhiên, vì chúng tôi sẽ thường xuyên phát sóng tới tất cả khách hàng được kết nối, chúng tôi cho phép một cờ \* đặc biệt để biểu thị quảng bá hàng loạt.

Mọi thứ hiện đã được thiết lập, chỉ yêu cầu định nghĩa dịch vụ cho ba hoạt động chính của ứng dụng này: thêm câu hỏi và câu trả lời mới, đồng thời ghi nhớ câu hỏi mà mỗi khách hàng đang theo dõi.

Khi các câu hỏi được đặt ra, chúng tôi đảm bảo rằng câu hỏi là duy nhất, hãy thêm nó vào câu hỏi bộ sưu tập và cho mọi người biết danh sách câu hỏi mới:

if(method === "askquestion") {

// Already asked?

if(questions[parameter]) { return response.end();

}

questions[parameter] = sseUserId; broadcast("\*", { type : "questions", questions : questions

});

return response.end(); }

Việc xử lý các câu trả lời gần giống nhau, ngoại trừ việc ở đây chúng tôi muốn chỉ phát các câu trả lời mới cho if(method === "addanswer") {

... answers[curUserQuestion] = answers[curUserQuestion] || []; answers[curUserQuestion].push(parameter); for(var id in clientQMap) { if(clientQMap[id] === curUserQuestion) { broadcast(id, { type : "answers", question : curUserQuestion, answers : answers[curUserQuestion]

});

}

}

return response.end(); }

Cuối cùng, chúng tôi lưu trữ các thay đổi theo sở thích của khách hàng bằng cách cập nhật clientQMap:

if(method === "selectquestion") { if(parameter && questions[parameter]) { clientQMap[sseUserId] = parameter; broadcast(sseUserId, {

type : "answers", question : parameter,

answers : answers[parameter] ? answers[parameter] : []

});

}

return response.end(); }

Mặc dù chúng tôi sẽ không đi quá sâu vào HTML phía máy khách và JavaScript cần thiết để hiển thị giao diện này, nhưng chúng tôi sẽ xem xét cách một số sự kiện cốt lõi sẽ được xử lý.

Giả sử giao diện người dùng được hiển thị bằng HTML, liệt kê các câu trả lời ở một bên và các câu hỏi ở bên kia, chứa các biểu mẫu để thêm câu hỏi và câu trả lời mới, cũng như để chọn câu hỏi để theo dõi, mã khách hàng của chúng tôi rất nhẹ và dễ làm theo. Sau khi thương lượng ban đầu / đăng nhập bắt tay với máy chủ của chúng tôi, khách hàng này chỉ cần gửi dữ liệu mới qua HTTP khi được gửi. Việc xử lý phản hồi của máy chủ được gói gọn thành ba sự kiện, giúp dễ dàng theo dõi việc xử lý luồng sự kiện:

var USER\_ID = null;

var evSource = new EventSource('/login'); var answerContainer = document.getElementById('answers'); var questionContainer = document.getElementById('questions'); var showAnswer = function(answers) { answerContainer.innerHTML = ""; var x = 0;

for(; x < answers.length; x++) { var li = document.createElement('li'); li.appendChild(document.createTextNode(answers[x])); answerContainer.appendChild(li);

}

}

var showQuestion = function(questions) { questionContainer.innerHTML = ""; for(var q in questions) {

//... show questions, similar to #showAnswer

}

}

evSource.addEventListener('message', function(broadcast) { var data = JSON.parse(broadcast.data); switch(data.type) { case "questions":

showQuestion(data.questions); break; case "answers":

showAnswer(data.answers);

break; case "notification": alert(data.message); break; default: throw "Received unknown message type"; break;

} });

evSource.addEventListener('login', function(broadcast) {

USER\_ID = JSON.parse(broadcast.data);

});

Giao diện này chỉ cần đợi dữ liệu câu hỏi và câu trả lời mới và hiển thị nó trong danh sách. Ba lần gọi lại là đủ để giữ cho ứng dụng khách này luôn cập nhật, bất kể nhiều ứng dụng khách khác nhau cập nhật trạng thái của ứng dụng như thế nào.

|  |  |
| --- | --- |
| Ưu điểm | Nhược điểm |
| Nhẹ: Bằng cách sử dụng giao thức HTTP gốc, một máy chủ SSE có thể được tạo với một vài tiêu đề đơn giản  Có thể gửi dữ liệu đơn phương đến khách hàng mà không yêu cầu | Hỗ trợ trình duyệt không nhất quán yêu cầu một thư viện tùy chỉnh để giao tiếp từ máy khách đến máy chủ, nơi không được hỗ trợ  trình duyệt thường sẽ thăm dò ý kiến lâu dài  Một cách duy nhất: Không mang lại lợi thế cho các trường hợp cần giao tiếp hai chiều |
| khớp các cuộc gọi của khách hàng  Tự động kết nối lại các kết nối bị rớt, làm cho SSE trở thành liên kết mạng đáng tin cậy  Đơn giản, dễ tùy chỉnh và định dạng nhắn tin dễ hiểu | Máy chủ phải gửi "nhịp tim" sau mỗi 10 giây hoặc lâu hơn để giữ kết nối hoạt động |

EventSource không được hỗ trợ bởi tất cả các trình duyệt (đặc biệt là IE). Bạn có thể tìm thấy thư viện mô phỏng tuyệt vời cho SSE tại https://github.com/Yaffle/EventSource.

## 6.11. Xây dựng ứng dụng chỉnh sửa tài liệu cộng tác

Bây giờ chúng ta đã xem xét các kỹ thuật khác nhau để xem xét khi xây dựng một ứng dụng cộng tác, hãy cùng nhau tạo một trình soạn thảo mã cộng tác bằng cách sử dụng một trong những đóng góp thú vị nhất cho Node sẽ ra mắt vào năm ngoái: Chuyển đổi hoạt động (OT).

Đối với cuộc thảo luận của chúng ta ở đây, OT sẽ được hiểu là một công nghệ cho phép nhiều người chỉnh sửa đồng thời cùng một tài liệu — chỉnh sửa tài liệu cộng tác. Google đã mô tả dự án Wave (hiện không còn tồn tại) của họ theo cách sau:

Chỉnh sửa tài liệu cộng tác có nghĩa là nhiều người chỉnh sửa có thể chỉnh sửa tài liệu được chia sẻ cùng một lúc. Nó hoạt động trực tiếp và đồng thời khi người dùng có thể thấy những thay đổi mà người khác đang thực hiện, thao tác gõ phím bằng cách gõ phím. Google Wave cung cấp khả năng chỉnh sửa đồng thời trực tiếp các tài liệu văn bản đa dạng thức.

Nguồn: http://www.waveprotocol.org/whitepapers/operational-transform

Một trong những kỹ sư tham gia vào dự án Wave là Joseph Gentle, và ông Gentle đủ tốt để viết một mô-đun mang công nghệ OT đến với cộng đồng Node, có tên ShareJS ( www.sharejs.org). Chúng tôi sẽ sử dụng mô-đun này để tạo một ứng dụng cho phép mọi người tạo một tài liệu mới có thể chỉnh sửa cộng tác.

Ví dụ này theo sau (và vay mượn từ) nhiều ví dụ có trong kho lưu trữ ShareJS GitHub. Để tìm hiểu sâu hơn về các khả năng của ShareJS, hãy truy cập https://github.com/share/ShareJS.

Để bắt đầu, chúng ta sẽ cần một trình soạn thảo mã để liên kết lớp OT của chúng ta với. Đối với dự án này, chúng tôi sẽ sử dụng trình chỉnh sửa Ace xuất sắc, có thể được sao chép từ https://github.com/ ajaxorg / át chủ bài.

Việc thiết lập một trình soạn thảo Ace không yêu cầu nhiều hơn là sao chép kho lưu trữ và viết HTML sau:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Editor</title>

<style type="text/css" media="screen"> body {

overflow: hidden;

}

#editor { margin: 0; position: absolute; top: 0; bottom: 0; left: 0; right: 0;

}

</style>

</head>

<body>

<pre id="editor">function hello(items) { alert("Hello World!");

}</pre>

<script src="/ace/ace.js" charset="utf-8"></script>

<script src="/ace/mode-javascript.js"></script>

<script src="/ace/theme-eclipse.js"></script>

<script>

var editor = ace.edit("editor"); editor.getSession().setMode(new (require("ace/mode/javascript").

Mode));

editor.setTheme("ace/theme/eclipse");

</script>

</body>

</html>

Hãy chuyển đổi trình chỉnh sửa phẳng này thành một trình cộng tác. Trước tiên, chúng tôi thêm phần sau vào < script> bộ sưu tập:

<script src = "/ channel / bcsocket.js"> </script>

<script src = "/ share / share.js"> </script>

<script src = "/ share / ace.js"> </script>

ShareJS sử dụng bcsocket thư viện, là một triển khai của máy chủ kênh Trình duyệt Google cho Node. Thư viện này hoạt động như một thư viện socket mà thông qua đó các máy khách và máy chủ ShareJS có thể giao tiếp. Sau đó, chúng tôi tải chính thư viện ShareJS và cuối cùng là tệp liên kết cho phép các tài liệu Ace thông báo cho ShareJS về các phép biến đổi và được chuyển đổi động.

Khi chúng tôi có các thư viện này ở trạng thái sẵn sàng, chúng tôi có thể liên kết với máy chủ ShareJS:

sharejs.open("documentname", 'text', function(error, doc) { if(error) { throw(error);

} if(doc.created) {

doc.insert(0, "var helloWorld = function() { \n alert('Hello

World!'); \n};");

}

doc.attach\_ace(editor); editor.setReadOnly(false); });

Ở đây, chúng tôi đang thực hiện một yêu cầu tới máy chủ ShareJS yêu cầu tạo tài liệu có tên nhất định. Nếu việc tạo tài liệu Cựu ước đó thành công, hãy thêm một số văn bản vào tài liệu đó và liên kết trình soạn thảo Ace của chúng tôi với luồng tài liệu Cựu ước đó.

Tất cả những gì còn lại cần làm là tạo máy chủ Node hiển thị chức năng của OT cho máy khách. Chúng ta sẽ cần một số cách để duy trì các tài liệu theo thời gian, vì các đối tượng dữ liệu lớn này có thể sẽ lấn át quá trình Node nếu tất cả được lưu trữ trong bộ nhớ. Trong quá trình thực hiện, chúng tôi sẽ sử dụng Redis. Ngoài ra, chúng tôi sẽ xây dựng máy chủ này bằng cách sử dụng khung Express, có thể dễ dàng cấu hình để chạy các ứng dụng bằng ShareJS.

Như chúng ta đã thấy trong các ví dụ trước đó, toàn bộ máy chủ này có thể được xây dựng bằng cách sử dụng mã nhỏ đáng // ShareJS uses CoffeeScript, requiring this compiler. require('coffee-script'); var express = require('express'); var sharejs = require('share'); var server = express();

server.use(express.static(\_\_dirname + '/')); var options = { db: {type: 'none'}, auth: function(client, action) { action.accept();

}

};

// Let's try and enable redis persistence if redis is installed...

try { require('redis'); options.db = {type: 'redis'};

} catch(e) {}

console.log("Server started"); // Bind sharejs interfaces to our server sharejs.server.attach(server, options); server.listen(8080); process.title = 'editor demo'

Chức năng của máy chủ này rất đơn giản. Khi tài liệu được yêu cầu (thông qua một cuộc gọi của khách hàng đến sharejs.open), máy chủ này sẽ mở ra một kết nối liên tục đến ứng dụng khách yêu cầu và sẽ chia sẻ bất kỳ thay đổi nào mà các ứng dụng khách khác liên kết với cùng một tài liệu có thể thực hiện với nó. Như đã đề cập trước đó, Redis được sử dụng để duy trì dữ liệu.

Một điều cần lưu ý là auth được truyền vào đối tượng tùy chọn ShareJS.

Chúng tôi ở đây chấp nhận tất cả các kết nối đến. Thay vào đó, nếu chúng tôi quan tâm đến việc có thể từ chối các kết nối dựa trên một số lược đồ ủy quyền, chúng tôi sẽ chuyển đổi hàm này thành một cái gì đó giống như đoạn mã sau:

auth: function(client, action) { if(client.sessionId in authenticatedSessions) { return action.accept();

}

action.reject(); }

Các khách hàng tham số được gửi đến hàm này chứa bản phân tích đầy đủ về tiêu đề yêu cầu và dữ liệu phiên, có thể được sử dụng để xác thực ứng dụng khách theo những cách chúng tôi đã sử dụng trong suốt cuốn sách này. Các hoạt động tham số sẽ chứa một đối tượng mô tả loại hành động Cựu ước được yêu cầu, chẳng hạn như cập nhật hoặc là đọc, cũng như tên của tài liệu được yêu cầu. Những giá trị khác này có thể được sử dụng để xác thực thậm chí còn sáng suốt hơn — một số ứng dụng khách có thể được phép đọc nhưng không được thực hiện thay đổi, chẳng hạn.

Máy chủ này hiện có thể được sử dụng để chia sẻ trạng thái tài liệu trên tất cả các máy khách yêu cầu tài liệu được đặt tên giống nhau, tạo điều kiện cho việc chỉnh sửa cộng tác.

Một thư viện Cựu ước khác cần xem xét là https://github.com/ Hoạt động-Chuyển đổi / ot.js /.

## Tóm lược

Trong chương này, chúng ta đã xem xét ba trong số các chiến lược chính được sử dụng khi xây dựng các ứng dụng thời gian thực: AJAX, WebSocket và SSE. Chúng tôi đã chỉ ra rằng các ứng dụng cộng tác không tầm thường có thể được phát triển với rất ít mã bằng cách sử dụng Node. Chúng tôi cũng đã thấy cách một số chiến lược này cho phép mô hình hóa giao tiếp máy khách / máy chủ như một giao diện luồng dữ liệu sự kiện. Chúng tôi đã xem xét ưu và nhược điểm của các kỹ thuật khác nhau này và chúng tôi đã xem qua một số ví dụ rõ ràng về những nơi tốt nhất để sử dụng từng kỹ thuật.

Ngoài ra, chúng tôi đã chỉ ra cách có thể xây dựng và quản lý các mã nhận dạng máy khách và ngày trạng thái trong máy chủ Node, sao cho các thay đổi trạng thái có thể được gói gọn trong một vị trí trung tâm và được truyền tải đến nhiều máy khách được kết nối một cách an toàn và dễ đoán. Để chứng minh chất lượng của các mô-đun đang được phát triển với cộng đồng Node, chúng tôi đã tạo ra một hệ thống chỉnh sửa mã cộng tác thông qua việc sử dụng chuyển đổi hoạt động.

Trong chương tiếp theo, chúng ta sẽ xem xét cách phối hợp các nỗ lực của nhiều tiến trình Node chạy đồng thời. Thông qua các ví dụ, chúng ta sẽ học cách đạt được quá trình xử lý song song với Node, từ việc tạo ra nhiều tiến trình con chạy các chương trình Unix đến việc tạo các cụm máy chủ socket Node cân bằng tải.

# CHƯƠNG 7

# Sử dụng nhiều tiến trình

"Đó là một điều rất đáng buồn khi ngày nay có quá ít thông tin vô ích."

- Oscar Wilde

Tầm quan trọng của hiệu quả I / O không bị mất đi đối với những người chứng kiến khối lượng dữ liệu ngày càng tăng nhanh chóng được tạo ra trong một số lượng ngày càng tăng của các ứng dụng. Nội dung do người dùng tạo (blog, video, tweet, bài đăng) đang trở thành loại nội dung internet hàng đầu và xu hướng này đã di chuyển song song với sự gia tăng của phần mềm xã hội, nơi ánh xạ các điểm giao nhau giữa các nội dung tạo ra sự gia tăng theo cấp số nhân ở một cấp độ khác của dữ liệu.

Một số kho dữ liệu, chẳng hạn như Google, Facebook và hàng trăm công ty khác, hiển thị dữ liệu của họ cho công chúng thông qua một API, thường là miễn phí. Mỗi mạng này đều thu thập khối lượng nội dung, ý kiến, mối quan hệ, v.v. đáng kinh ngạc từ người dùng của họ, dữ liệu được tăng cường thêm nhờ nghiên cứu thị trường và các loại phân tích lưu lượng và sử dụng khác nhau. Hầu hết các API này là hai chiều, thu thập và cất giữ dữ liệu được tải lên bởi các thành viên của họ cũng như phục vụ dữ liệu đó.

Nút đã đến trong giai đoạn mở rộng dữ liệu này. Trong chương này, chúng ta sẽ tìm hiểu cách Node giải quyết nhu cầu sắp xếp, hợp nhất, tìm kiếm và xử lý một lượng lớn dữ liệu. Tinh chỉnh phần mềm của bạn để nó có thể xử lý lượng lớn dữ liệu một cách an toàn và không tốn kém, là điều quan trọng khi xây dựng các ứng dụng mạng nhanh và có thể mở rộng.

Chúng tôi sẽ giải quyết các vấn đề về quy mô cụ thể trong chương tiếp theo. Trong chương này, chúng ta sẽ nghiên cứu một số phương pháp hay nhất khi thiết kế hệ thống trong đó nhiều quá trình Node hoạt động cùng nhau trên khối lượng lớn dữ liệu.

Là một phần của cuộc thảo luận đó, chúng tôi sẽ nghiên cứu các chiến lược cho song song khi xây dựng các ứng dụng nặng về dữ liệu, tập trung vào cách tận dụng lợi thế của nhiều môi trường CPU, sử dụng nhiều nhân công và tận dụng chính hệ điều hành để đạt được hiệu quả của song song. Quá trình tập hợp các ứng dụng từ các đơn vị xử lý có sẵn và hiệu quả này sẽ được chứng minh bằng ví dụ.

Như đã lưu ý trong Chương 5, Quản lý Nhiều Kết nối Máy khách Đồng thời, đồng thời không giống như song song. Mục tiêu của đồng thời là cấu trúc tốt cho các chương trình, trong đó việc mô hình hóa những phức tạp vốn có trong việc kết hợp nhiều quy trình đồng thời được đơn giản hóa. Mục tiêu của song song là tăng hiệu suất của ứng dụng bằng cách chia sẻ các phần của một nhiệm vụ hoặc tính toán cho nhiều công nhân. Sẽ rất hữu ích khi nhớ lại tầm nhìn của Clinger về "… hàng chục, hàng trăm hoặc thậm chí hàng nghìn bộ vi xử lý độc lập, mỗi bộ đều có bộ nhớ cục bộ và bộ xử lý truyền thông riêng, giao tiếp qua mạng truyền thông hiệu suất cao."

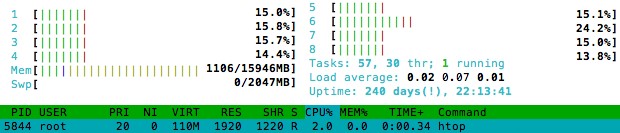
Chúng ta đã thảo luận về cách Node giúp chúng ta lập luận về luồng điều khiển không xác định. Chúng ta hãy nhớ lại cách các nhà thiết kế của Node tuân theo Quy tắc Mô-đun, khuyến khích chúng ta viết các phần đơn giản được kết nối bằng các giao diện sạch sẽ. Quy tắc này dẫn đến ưu tiên cho các quy trình được nối mạng đơn giản giao tiếp với nhau bằng một giao thức chung. Một quy tắc liên quan là Quy tắc về sự đơn giản, được nêu như sau:

Các nhà phát triển nên thiết kế đơn giản bằng cách tìm cách chia hệ thống chương trình thành các phần nhỏ, hợp tác dễ hiểu. Quy tắc này nhằm mục đích không khuyến khích sự yêu thích của các nhà phát triển đối với việc viết "những phức tạp đẹp đẽ và phức tạp" trong các chương trình dễ gặp lỗi trong thực tế.

- http://en.wikipedia.org/wiki/Unix\_phiosystemhy

Chúng ta nên ghi nhớ quy tắc này khi chúng ta tiếp tục chương này. Để chế ngự khối lượng dữ liệu ngày càng mở rộng, chúng ta có thể xây dựng các khối nguyên khối khổng lồ, phức tạp và mạnh mẽ với hy vọng rằng chúng vẫn đủ lớn và mạnh mẽ. Ngoài ra, chúng ta có thể xây dựng các đơn vị xử lý nhỏ và hữu ích có thể được kết hợp thành một nhóm xử lý duy nhất ở bất kỳ quy mô nào, không giống như cách mà siêu máy tính có thể được tạo ra từ hàng nghìn hoặc hàng triệu bộ xử lý hàng hóa giá rẻ.

Một trình xem quy trình sẽ hữu ích khi làm việc thông qua chương này. Một điều tốt cho hệ thống Unix là htop, có thể tải xuống từ http: //htop.sourceforge. mạng lưới/. Công cụ này cung cấp, trong số những thứ khác, một cái nhìn về việc sử dụng CPU và bộ nhớ; ở đây, chúng tôi thấy cách tải được trải qua tất cả tám lõi:



Mô hình đơn luồng của Node

Được chụp toàn bộ, môi trường Node thể hiện một cách hữu ích cả hai hiệu quả của song song đa luồng và một cú pháp biểu cảm có thể sử dụng được cho các ứng dụng có tính đồng thời cao. Việc sử dụng Node không hạn chế nhà phát triển, quyền truy cập của nhà phát triển vào tài nguyên hệ thống hoặc các loại ứng dụng mà nhà phát triển có thể muốn xây dựng.

Tuy nhiên, một số lượng đáng ngạc nhiên những lời chỉ trích dai dẳng về Node dựa trên sự hiểu lầm này. Như chúng ta sẽ thấy, niềm tin rằng Node không đa luồng và do đó, chậm hoặc không sẵn sàng cho thời gian quan trọng, chỉ đơn giản là bỏ lỡ vấn đề. JavaScript là một luồng; ngăn xếp Node không. JavaScript đại diện cho ngôn ngữ được sử dụng để điều phối việc thực thi một số quy trình C ++ đa luồng, thậm chí cả các tiện ích bổ sung C ++ riêng do bạn, nhà phát triển tạo ra. Node cung cấp JavaScript, chạy qua V8, chủ yếu như một công cụ để lập mô hình đồng thời. Ngoài ra, người ta có thể viết toàn bộ ứng dụng chỉ bằng JavaScript đơn giản là một lợi ích khác của nền tảng. Bạn không bao giờ bị mắc kẹt với JavaScript — bạn có thể viết phần lớn ứng dụng của mình bằng C ++ nếu đó là lựa chọn của bạn.

Trong chương này, chúng tôi sẽ cố gắng tháo gỡ những hiểu lầm này, mở đường cho sự phát triển lạc quan với Node. Đặc biệt, chúng tôi sẽ nghiên cứu các kỹ thuật để lan tỏa nỗ lực trên các lõi, quy trình và luồng. Hiện tại, phần này sẽ cố gắng làm rõ khả năng của một luồng đơn như thế nào (gợi ý: đó thường là tất cả những gì bạn cần).

## 7.1. Lợi ích của lập trình đơn luồng

Bạn sẽ khó tìm thấy một số lượng đáng kể các kỹ sư phần mềm chuyên nghiệp làm việc trên phần mềm cấp doanh nghiệp sẵn sàng phủ nhận rằng việc phát triển phần mềm đa luồng là điều khó khăn. Tuy nhiên, tại sao làm tốt có khó không?

Không phải lập trình đa luồng là khó - cái khó nằm ở sự phức tạp của đồng bộ hóa luồng. Rất khó để xây dựng tính đồng thời cao bằng cách sử dụng mô hình luồng, đặc biệt là các mô hình trong đó trạng thái được chia sẻ. Việc lường trước mọi cách mà một hành động được thực hiện trong một luồng có thể ảnh hưởng đến tất cả những luồng khác là điều gần như không thể xảy ra khi một ứng dụng phát triển vượt ra ngoài những hình dạng cơ bản nhất. Các vướng mắc và va chạm nhân lên nhanh chóng, đôi khi làm hỏng bộ nhớ dùng chung, đôi khi tạo ra các lỗi gần như không thể theo dõi được.

Các nhà thiết kế của Node đã chọn nhận ra các lợi thế về tốc độ và song song hóa

của các chủ đề mà không yêu cầu các nhà phát triển phải làm như vậy. Đặc biệt, các nhà thiết kế của Node muốn cứu các nhà phát triển khỏi việc quản lý những khó khăn đi kèm với hệ thống phân luồng:

Bộ nhớ dùng chung và hành vi khóa dẫn đến các hệ thống rất khó lý luận khi chúng ngày càng phức tạp.

Giao tiếp giữa các tác vụ yêu cầu thực hiện nhiều loại nguyên thủy đồng bộ hóa, chẳng hạn như mutexes và semaphores, biến điều kiện, v.v. Một môi trường vốn đã đầy thách thức đòi hỏi các công cụ phức tạp cao, mở rộng mức độ chuyên môn cần thiết để hoàn thành các hệ thống thậm chí tương đối đơn giản.

Điều kiện cuộc đua và bế tắc là những cạm bẫy phổ biến trong các loại hệ thống này. Các hoạt động đọc và ghi đồng thời trong không gian chương trình dùng chung dẫn đến các vấn đề về trình tự, trong đó hai luồng có thể ở trong một "cuộc đua" không thể đoán trước để giành quyền ảnh hưởng đến trạng thái, sự kiện hoặc đặc tính hệ thống chính khác.

Bởi vì việc duy trì các ranh giới đáng tin cậy giữa các luồng và trạng thái của chúng là rất khó, việc đảm bảo rằng một thư viện (đối với Node sẽ là một "mô-đun") an toàn cho luồng sẽ tiêu tốn rất nhiều thời gian của nhà phát triển. Tôi có thể biết rằng thư viện này sẽ không phá hủy một số phần ứng dụng của tôi không? Việc đảm bảo an toàn cho luồng đòi hỏi sự siêng năng của nhà phát triển thư viện và những đảm bảo này có thể có điều kiện: ví dụ: một thư viện có thể an toàn cho luồng khi đọc — nhưng không phải khi viết.

Đối số chính cho luồng đơn là luồng điều khiển rất khó trong môi trường đồng thời và đặc biệt là khi truy cập bộ nhớ hoặc thứ tự thực thi mã là không thể đoán trước:

Thay vì liên quan đến việc khóa tùy ý và các va chạm khác, các nhà phát triển có thể tập trung vào việc xây dựng các chuỗi thực thi mà thứ tự có thể dự đoán được.

Vì quá trình song song được thực hiện thông qua việc sử dụng nhiều quy trình, mỗi quy trình có một không gian bộ nhớ riêng biệt và riêng biệt, giao tiếp giữa các quy trình vẫn không phức tạp — thông qua Quy tắc đơn giản, chúng tôi không chỉ đạt được các thành phần đơn giản và không có lỗi mà còn có khả năng tương tác dễ dàng hơn.

Bởi vì trạng thái không được chia sẻ (tùy ý) giữa các quá trình Node riêng lẻ, một quá trình đơn lẻ được tự động bảo vệ khỏi các lần truy cập bất ngờ từ các quá trình khác dựa trên việc phân bổ lại bộ nhớ hoặc độc quyền tài nguyên. Giao tiếp thông qua các kênh rõ ràng bằng cách sử dụng các giao thức cơ bản, tất cả đều gây khó khăn cho việc viết các chương trình tạo ra những thay đổi không thể đoán trước qua các quy trình.

An toàn luồng là một trong những mối quan tâm ít hơn để các nhà phát triển lãng phí thời gian lo lắng. Bởi vì đồng thời đơn luồng loại bỏ các xung đột có trong đồng thời đa luồng, nên việc phát triển có thể tiến hành nhanh hơn, chắc chắn hơn.

Khách hàng

Khách hàng

X

X

X

Chủ đề, mỗi chủ đề

trạng thái chia sẻ, yêu cầu chặn truy cập

và các loại trừ khác

Các quy trình độc lập không có trạng thái chia sẻ. Dữ liệu đã

lọc được chuyển qua tin nhắn–

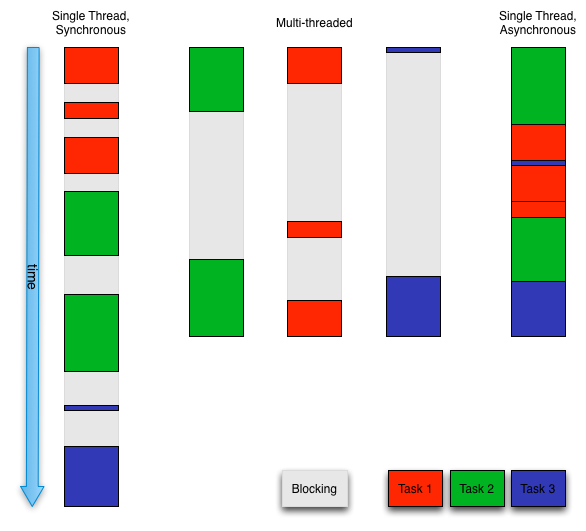
không có va chạm

SINGLE-THREADED,

BẤT NGỜ

ĐA NĂNG

Một luồng duy nhất được quản lý hiệu quả bởi một vòng lặp sự kiện mang lại sự ổn định, khả năng bảo trì, khả năng đọc và khả năng phục hồi cho các chương trình Node. Tin quan trọng là Node tiếp tục cung cấp tốc độ và sức mạnh của đa luồng cho các nhà phát triển của nó — sự sáng chói thiết kế của Node làm cho sức mạnh đó trở nên minh bạch, phản ánh một phần mục đích đã nêu của Node là mang lại nhiều sức mạnh nhất cho hầu hết mọi người với ít khó khăn nhất.



Trong sơ đồ trước, sự khác biệt giữa hai mô hình đơn luồng và mô hình đa luồng được hiển thị.

Không có lối thoát nào khỏi các hoạt động chặn — ví dụ: đọc từ một tệp sẽ luôn luôn mất một thời gian. Mô hình đồng bộ một luồng buộc mỗi tác vụ phải đợi những tác vụ khác hoàn thành trước khi bắt đầu, tốn nhiều thời gian hơn. Một số tác vụ có thể được bắt đầu song song bằng cách sử dụng các luồng, thậm chí vào các thời điểm khác nhau, trong đó tổng thời gian thực thi không lâu hơn thời gian thực hiện của luồng chạy dài nhất. Khi sử dụng luồng, nhà phát triển sẽ chịu trách nhiệm đồng bộ hóa hoạt động của từng luồng riêng lẻ, sử dụng khóa hoặc các công cụ lập lịch khác. Điều này có thể trở nên rất phức tạp khi số lượng chủ đề tăng lên, và trong sự phức tạp này có những lỗi rất tinh vi và khó tìm.

Thay vì để nhà phát triển phải vật lộn với sự phức tạp này, Node tự quản lý các luồng I / O. Bạn không cần quản lý vi mô phân luồng I / O; người ta chỉ cần thiết kế một ứng dụng để thiết lập các điểm khả dụng của dữ liệu (gọi lại) và các hướng dẫn sẽ được thực thi khi dữ liệu đã nói sẵn có. Các luồng cung cấp hiệu quả tương tự, nhưng việc quản lý của chúng được tiếp xúc với nhà phát triển thông qua một giao diện dễ hiểu.

## 7.2. Đa luồng đã là nguyên bản và minh bạch

Nhóm luồng I / O của Node thực thi trong phạm vi hệ điều hành và công việc của nó được phân phối trên các lõi (giống như bất kỳ công việc nào khác được lập lịch bởi hệ điều hành sẽ được phân phối tương tự). Khi bạn đang chạy Node, bạn đã tận dụng được khả năng thực thi đa luồng của nó.

Trong cuộc thảo luận sắp tới về các quy trình con và Cụm mô-đun, chúng ta sẽ thấy kiểu song song này — của nhiều quy trình song song — đang hoạt động. Chúng ta sẽ xem cách Node không bị từ chối toàn bộ sức mạnh của một hệ điều hành.

Như chúng ta đã thấy trước đó, khi thảo luận về kiến trúc cốt lõi của Node, chuỗi V8 trong đó một chuỗi thực thi các chương trình JavaScript được liên kết với libuv, có chức năng như bộ điều phối sự kiện I / O cấp độ hệ thống. Trong khả năng này, libuv xử lý bộ hẹn giờ, cuộc gọi hệ thống tệp, cuộc gọi mạng và các hoạt động I / O khác được yêu cầu bởi quy trình JavaScript liên quan hoặc các lệnh mô-đun, chẳng hạn như fs.readFile, http.createServer, và như thế. Do đó, vòng lặp sự kiện V8 chính được hiểu tốt nhất là một giao diện lập trình luồng điều khiển, được hỗ trợ và cung cấp bởi đại biểu hệ thống, đa luồng, hiệu quả cao. libuv.

Burt Belder, một trong những người đóng góp cốt lõi của Node, cũng là một trong những người đóng góp cốt lõi cho libuv. Trên thực tế, sự phát triển của Node đã kích thích sự gia tăng đồng thời libuv phát triển, một vòng phản hồi chỉ cải thiện tốc độ và độ ổn định của cả hai dự án. Nó đã hợp nhất và thay thế libeo và libev các thư viện đã hình thành lõi ban đầu của ngăn xếp Node.

Hãy xem xét một quy tắc khác của Raymond, Quy tắc phân tách: " Chính sách tách biệt khỏi chủ nghĩa cơ chế; giao diện tách biệt với các công cụ ". Công cụ hỗ trợ phong cách lập trình hướng sự kiện, không đồng bộ của Node là libuv; giao diện cho công cụ đó là thời gian chạy JavaScript của V8. Tiếp tục với Raymond:

Ví dụ, một cách để thực hiện việc tách biệt đó là viết ứng dụng của bạn dưới dạng thư viện các quy trình dịch vụ C được điều khiển bởi một ngôn ngữ kịch bản nhúng, với luồng điều khiển ứng dụng được viết bằng ngôn ngữ kịch bản chứ không phải C.

Khả năng sắp xếp các quy trình hệ điều hành song song siêu hiệu quả trong sự trừu tượng của một luồng dự đoán duy nhất tồn tại theo thiết kế, không phải là một nhượng bộ. Nó kết luận một phân tích thực tế về cách có thể cải thiện quá trình phát triển ứng dụng, và nó chắc chắn không phải là giới hạn đối với những gì có thể.

Giải nén chi tiết libuv có thể được tìm thấy tại http: // nikhilm. github.io/uvbook/. Burt Belder cũng nói chuyện chuyên sâu về làm sao libuv hoạt động dưới mui xe tại http://www.youtube. com / xem? v = nGn60vDSxQ4.

## 7.3. Tạo quy trình con

Phát triển phần mềm không còn là lĩnh vực của các chương trình nguyên khối. Các ứng dụng chạy trên mạng không thể loại trừ khả năng tương tác. Các ứng dụng hiện đại được phân phối và tách rời. Giờ đây, chúng tôi xây dựng các ứng dụng kết nối người dùng với các tài nguyên được phân phối trên Internet. Nhiều người dùng đang truy cập các tài nguyên được chia sẻ đồng thời. Một hệ thống phức tạp sẽ dễ hiểu hơn nếu toàn bộ được hiểu là một tập hợp các giao diện đến các chương trình giải quyết một hoặc một số vấn đề liên quan được xác định rõ ràng. Trong một hệ thống như vậy, người ta mong đợi (và mong muốn) rằng các quá trình sẽ không hoạt động.

Một lời chỉ trích ban đầu về Node là nó không có nhận thức đa nhân. Nghĩa là, nếu một máy chủ Node đang chạy trên một máy có nhiều lõi, nó sẽ không thể tận dụng được mã lực bổ sung này. Bên trong lời chỉ trích có vẻ hợp lý này đã che giấu một thành kiến phi lý do dựa trên một người quản lý: một chương trình không thể phân bổ rõ ràng bộ nhớ và các "luồng" thực thi để thực hiện song song không thể xử lý các vấn đề cấp doanh nghiệp.

Sự chỉ trích này là một sự chỉ trích dai dẳng. Nó cũng không đúng.

Trong khi một quy trình Node chạy trên một lõi đơn, bất kỳ số lượng quy trình Node nào cũng có thể được "quay vòng" thông qua việc sử dụng quy trình\_ con mô-đun. Cách sử dụng cơ bản của mô-đun này rất đơn giản: chúng tôi tìm nạp một ChildProcess và lắng nghe các sự kiện dữ liệu. Ví dụ này sẽ gọi lệnh Unix ls, liệt kê thư mục hiện tại:

var spawn = demand ('child\_process'). spawn; var ls

= spawn ('ls', ['-lh', '.']);

ls.stdout.on ('có thể đọc được', function () { var d = this.read ();

d && console.log (d.toString ());

});

ls.on ('close', function (code) { console.log ('tiến trình con đã thoát bằng mã' + mã); });

Ở đây, chúng tôi sinh ra ls xử lý (thư mục danh sách) và đọc từ kết quả có thể đọc được Suối, nhận được một cái gì đó như:

rw-r - r-- 1 gốc rễ 43 Ngày 9 tháng 7 19:44 index.html

rw-rw-r-- 1 root root 278 Jul 15 16:36 child\_example.js- rw-r - r-- 1 root root 1.2K Jul 14 19:08 server.js

tiến trình con đã thoát với mã 0

Bất kỳ số lượng quy trình con nào cũng có thể được tạo ra theo cách này. Điều quan trọng là lưu ý ở đây rằng khi một tiến trình con được tạo ra, hoặc được tạo ra theo cách khác, hệ điều hành sẽ tự gán trách nhiệm cho tiến trình đó cho một CPU nhất định. Node không chịu trách nhiệm về cách một hệ điều hành phân bổ tài nguyên. Kết quả là trên một máy có tám lõi, có khả năng sinh ra tám quy trình sẽ dẫn đến việc mỗi quy trình được phân bổ cho các bộ xử lý độc lập. Nói cách khác, các tiến trình con được Hệ điều hành tự động lan truyền trên các CPU, tạo điều kiện lừa dối rằng Node không thể tận dụng hết môi trường đa lõi.

Mỗi tiến trình Node mới (con) được cấp phát 10 MB bộ nhớ và đại diện cho một phiên bản V8 mới sẽ mất ít nhất 30 mili giây để khởi động. Mặc dù không chắc rằng bạn sẽ tạo ra hàng nghìn quy trình này, nhưng việc hiểu cách truy vấn và đặt giới hạn hệ điều hành trên các quy trình do người dùng tạo là có lợi. htop hoặc là hàng đầu sẽ báo cáo số lượng quy trình hiện đang chạy hoặc bạn có thể sử dụng ps aux | wc –l từ dòng lệnh. Lệnh Unix ulimit (http://ss64.com/ bash / ulimit.html) cung cấp thông tin quan trọng về giới hạn người dùng trên một hệ điều hành. Đi qua ulimit, các –U đối số sẽ hiển thị số lượng quy trình người dùng tối đa có thể được tạo ra. Việc thay đổi giới hạn được thực hiện bằng cách chuyển nó làm đối số: ulimit –u 8192.

Các quy trình\_ con mô-đun đại diện cho một lớp hiển thị bốn phương thức chính: đẻ trứng, ngã ba, hành, và tập tin thực thi. Các phương thức này trả về một ChildProcess đối tượng kéo dài EventEmitter, hiển thị giao diện cho các sự kiện con và một số chức năng hữu ích để quản lý các quy trình con. Chúng ta sẽ xem xét các phương pháp chính của nó và tiếp tục thảo luận về các ChildProcess giao diện.

## 7.4. Quá trình sinh sản

Lệnh mạnh mẽ này cho phép một chương trình Node khởi động và tương tác với các quy trình được tạo ra thông qua các lệnh hệ thống. Trong ví dụ trước, chúng tôi đã sử dụng đẻ trứng để gọi một quy trình hệ điều hành gốc, ls, chuyển lệnh đó lh và. tranh luận. Bằng cách này, bất kỳ quá trình nào cũng có thể được bắt đầu giống như người ta có thể bắt đầu nó thông qua một dòng lệnh. Phương thức này có ba đối số:

chỉ huy: Một lệnh được thực thi bởi hệ điều hành shell

tranh luận ( tùy chọn): Đây là các đối số dòng lệnh, được gửi dưới dạng một mảng

các tùy chọn: Một bản đồ cài đặt tùy chọn cho đẻ trứng

Các tùy chọn cho đẻ trứng cho phép hành vi của nó được tùy chỉnh cẩn thận:

cwd ( Chuỗi): Theo mặc định, lệnh sẽ hiểu thư mục làm việc hiện tại của nó giống với thư mục đang gọi tiến trình Node đẻ trứng. Thay đổi cài đặt đó bằng cách sử dụng chỉ thị này.

env ( Đối tượng): Điều này được sử dụng để chuyển các biến môi trường cho một tiến trình con. Ví dụ:

hãy xem xét việc sinh sản một đứa trẻ với một đối tượng môi trường như sau:

{

tên: "Sandro",

vai trò: "quản trị viên"

}

Môi trường tiến trình con sẽ có quyền truy cập vào các giá trị này tách ra ( Boolean): Khi cha mẹ sinh ra con, cả hai quá trình hình thành một nhóm và phụ huynh thường là trưởng nhóm đó. Để đặt một đứa trẻ làm trưởng nhóm, hãy sử dụng tách rời. Điều này sẽ cho phép đứa trẻ tiếp tục chạy ngay cả sau khi cha mẹ đã ra ngoài. Điều này là do cha mẹ sẽ đợi con thoát ra theo mặc định. Bạn có thể gọi child.unref () để cho vòng lặp sự kiện của cha mẹ biết rằng nó không được tính tham chiếu con và thoát ra nếu không có tác phẩm nào khác tồn tại.

uid ( Số): Đặt uid ( nhận dạng người dùng) chỉ thị cho quy trình con, xét về quyền hệ thống tiêu chuẩn, chẳng hạn như UID có đặc quyền thực thi trên quy trình con.

gid ( Số): Đặt gid ( nhận dạng nhóm) chỉ thị cho quy trình con, xét về quyền tiêu chuẩn của hệ thống, chẳng hạn như GID có các đặc quyền thực thi trên quy trình con.

stdio ( Chuỗi hoặc Mảng): Các quy trình con có bộ mô tả tệp, ba bộ đầu tiên là bộ mô tả I / O tiêu chuẩn process.stdin, process.stdout và process.stderr, theo thứ tự (fds = 0,1,2). Chỉ thị này cho phép các bộ mô tả đó được xác định lại, kế thừa, v.v.

Xem xét đầu ra của chương trình quy trình con sau:

process.stdout.write (new Buffer ("Xin chào!"));

Ở đây, một phụ huynh sẽ lắng nghe con.stdout. Nếu thay vào đó, chúng ta muốn một đứa trẻ kế thừa cha mẹ của nó stdio, sao cho khi đứa trẻ viết thư cho process.stdout

những gì được phát ra được chuyển qua đường ống của cha mẹ process.stdout, chúng tôi sẽ chuyển các bộ mô tả tệp mẹ có liên quan cho con, ghi đè chính nó: đẻ trứng ("nút", ['./reader.js', './afile.txt'], { stdio: [process.stdin, process.stdout, process.stderr]});

Trong trường hợp này, đầu ra của trẻ sẽ chuyển thẳng đến kênh đầu ra chuẩn của cha mẹ. Cũng thấy cái nĩa, dưới đây, để biết thêm thông tin về loại mô hình này.

Mỗi bộ mô tả tệp trong số ba (hoặc nhiều hơn) có thể nhận một trong sáu giá trị:

° ống: Điều này tạo ra một đường ống giữa con và mẹ. Vì ba bộ mô tả tệp con đầu tiên đã được hiển thị cho phụ huynh

( child.stdin, child.stdout, và child.stderr) đây chỉ là cần thiết trong các triển khai con phức tạp hơn.

° ipc: Điều này tạo ra một kênh IPC để chuyển các thông báo giữa một đứa trẻ và cha mẹ. Một quy trình con có thể có tối đa một bộ mô tả tệp IPC. Khi kết nối này được thiết lập, cha mẹ có thể giao tiếp với trẻ qua con. gửi. Nếu đứa trẻ gửi tin nhắn JSON thông qua bộ mô tả tệp này, những phát thải đó có thể bị bắt bằng cách sử dụng child.on ("tin nhắn"). Nếu chạy chương trình Node khi còn nhỏ, nó có thể là lựa chọn tốt hơn để sử dụng ChildProcess.fork, có tích hợp kênh nhắn tin này.

° Làm lơ: Các bộ mô tả tệp 0-2 sẽ có / dev / null gắn liền với chúng. Đối với những người khác, bộ mô tả tệp được tham chiếu sẽ không được đặt trên con.

° Một đối tượng luồng: Điều này cho phép cha mẹ chia sẻ một luồng đứa trẻ. Đối với mục đích trình diễn, một đứa trẻ sẽ viết cùng một nội dung cho bất kỳ WordsStream, chúng tôi có thể làm một cái gì đó như thế này:

varriter = fs.createWriteStream ("./ a.out"); writer.on ('open', function () {

var cp = spawn ("node", ['./reader.js'], {stdio: [null, writer, null]

});

});

Con bây giờ sẽ tìm nạp nội dung của nó và chuyển nó đến bất kỳ luồng đầu ra nào mà nó đã được gửi:

fs.createReadStream ('cache.data'). pipe (process.stdout);

° Một số nguyên: Id bộ mô tả tệp.

° vô giá trị và chưa xác định: Đây là các giá trị mặc định. Đối với bộ mô tả tệp 0-2 ( stdin, stdout, và stderr) một đường ống được tạo ra. Những người khác mặc định là Làm lơ.

Ngoài việc vượt qua stdio cài đặt dưới dạng một mảng, một số nhóm chung nhất định có thể được triển khai bằng cách chuyển một giá trị chuỗi phím tắt:

'bỏ qua' = ['bỏ qua', 'bỏ qua', 'bỏ qua']

'pipe' = ['pipe', 'pipe', 'pipe']

'inherit' = [process.stdin, process.stdout, process.stderr] hoặc là [ 0,1,2]

Chúng tôi đã chỉ ra một số ví dụ về việc sử dụng đẻ trứng để chạy các chương trình Node như các tiến trình con. Mặc dù đây là cách sử dụng hoàn toàn hợp lệ (và là một cách tốt để thử các tùy chọn API), đẻ trứng chủ yếu để chạy các lệnh hệ thống. Xem cuộc thảo luận của cái nĩa, bên dưới, để biết thêm thông tin về việc chạy các quy trình Node khi còn nhỏ.

Cần lưu ý rằng khả năng sinh ra bất kỳ quy trình hệ thống nào có nghĩa là người ta có thể sử dụng Node để chạy các môi trường ứng dụng khác được cài đặt trên hệ điều hành. Nếu một người đã cài đặt ngôn ngữ PHP phổ biến, thì có thể làm như sau:

var spawn = demand ('child\_process'). spawn; var php = spawn ("php", ['-r', 'print "Xin chào từ PHP!";']);

php.stdout.on ('có thể đọc được', function () { var d;

while (d = this.read ()) { console.log (d.toString ());

}

});

// Xin chào từ PHP!

## 7.5. Chạy một chương trình lớn hơn, thú vị hơn sẽ dễ dàng như vậy.

Ngoài sự dễ dàng mà người ta có thể chạy Java hoặc Ruby hoặc các chương trình khác thông qua Node bằng cách sử dụng kỹ thuật này một cách không đồng bộ, ở đây chúng tôi cũng có một câu trả lời tốt cho những lời chỉ trích dai dẳng về Node: JavaScript không nhanh như các ngôn ngữ khác để xử lý các con số, hoặc thực hiện các tác vụ nặng khác với CPU. Điều này đúng, theo nghĩa là Node chủ yếu được tối ưu hóa cho hiệu quả I / O và giúp quản lý các ứng dụng đồng thời cao và JavaScript là một ngôn ngữ thông dịch không tập trung nhiều vào tính toán nặng.

Tuy nhiên, sử dụng đẻ trứng, người ta có thể rất dễ dàng chuyển các tính toán lớn và các quy trình chạy dài trên các công cụ phân tích hoặc công cụ tính toán để tách các quy trình trong môi trường khác. Vòng lặp sự kiện đơn giản của Node sẽ chắc chắn thông báo cho ứng dụng chính khi các hoạt động đó được thực hiện, tích hợp liền mạch dữ liệu kết quả. Trong thời gian chờ đợi, ứng dụng chính vẫn miễn phí để tiếp tục phục vụ khách hàng.

## 7.6. Quy trình rèn

Giống đẻ trứng, cái nĩa bắt đầu một quy trình con, nhưng được thiết kế để chạy các chương trình Node với lợi ích bổ sung là tích hợp sẵn kênh giao tiếp. Thay vì chuyển lệnh hệ thống tới cái nĩa là đối số đầu tiên của nó, một đối số chuyển đường dẫn đến một chương trình Node. Như với đẻ trứng, các tùy chọn dòng lệnh có thể được gửi dưới dạng đối số thứ hai, có thể truy cập qua process.argv trong quá trình con được chia nhỏ.

Một đối tượng tùy chọn tùy chọn có thể được chuyển làm đối số thứ ba của nó, với các tham số sau:

cwd ( Chuỗi): Theo mặc định, lệnh sẽ hiểu thư mục làm việc hiện tại của nó giống với thư mục đang gọi tiến trình Node cái nĩa. Thay đổi cài đặt đó bằng cách sử dụng chỉ thị này.

env ( Đối tượng): Điều này được sử dụng để chuyển các biến môi trường cho một tiến trình con.

Xem đẻ trứng.

mã hóa ( Chuỗi): Điều này đặt mã hóa của kênh giao tiếp.

executePath ( String): Đây là tệp thực thi được sử dụng để tạo tiến trình con.

im lặng ( Boolean): Theo mặc định, một con được chia nhỏ sẽ có stdio liên kết với cha mẹ ( child.stdout giống hệt với parent.stdout, ví dụ). Đặt tùy chọn này thành thật vô hiệu hóa hành vi này.

Một sự khác biệt quan trọng giữa cái nĩa và đẻ trứng đó có phải là quá trình con của người cũ không không tự động thoát khi nó được hoàn thành. Một đứa trẻ như vậy phải thoát ra một cách rõ ràng khi nó được hoàn thành, dễ dàng hoàn thành thông qua process.exit ().

Trong ví dụ sau, chúng ta tạo một phần tử con phát ra một số gia tăng mỗi phần mười giây, số này cha mẹ của nó sau đó chuyển vào bảng điều khiển hệ thống. Đầu tiên, chương trình con:

var cnt = 0;

setInterval (function () { process.stdout.write ("->" + cnt ++); }, 100);

Một lần nữa, điều này sẽ chỉ đơn giản là viết một số tăng dần đều. Ghi nhớ điều đó với

cái nĩa một đứa trẻ sẽ thừa kế stdio của cha mẹ của nó, chúng ta chỉ cần tạo con để nhận đầu ra trong một thiết bị đầu cuối đang chạy tiến trình mẹ:

var fork = demand ('child\_process'). fork; ngã ba ('./ emitter.js');

// -> 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 ...

Các im lặng tùy chọn có thể được chứng minh ở đây. Thao tác sau sẽ tắt bất kỳ đầu ra nào cho thiết bị đầu cuối:

fork ('./ emitter.js', [], {silent: true});

Dễ dàng tạo nhiều quy trình song song. Hãy nhân số con được tạo ra:

ngã ba ('./ emitter.js'); ngã ba ('./ emitter.js'); ngã ba ('./ emitter.js');

// 0 -> 0 -> 0 -> 1 -> 1 -> 1 -> 2 -> 2 -> 2 -> 3 -> 3 -> 3 -> 4 ...

Tại thời điểm này, cần rõ ràng rằng bằng cách sử dụng cái nĩa chúng tôi đang tạo ra nhiều bối cảnh thực thi song song, trải rộng trên tất cả các lõi máy.

Điều này đủ đơn giản, nhưng kênh giao tiếp tích hợp cái nĩa cung cấp giúp việc giao tiếp với những đứa trẻ được chia sẻ trở nên dễ dàng hơn và sạch sẽ hơn. Hãy xem xét những điều sau:

## 7.7. Cha mẹ

var fork = demand ('child\_process'). fork; var cp = fork ('./ child.js');

cp.on ('message', function (msgobj) { console.log ('Parent got message:', msgobj.text); }); cp.send ({ text: "I love you"});

Đứa trẻ

process.on ('message', function (msgobj) { console.log ('Con nhận được tin nhắn:', msgobj.text); process.send ({ text: msgobj.text + 'too'});

});

Bằng cách thực thi tập lệnh mẹ, chúng ta sẽ thấy những điều sau trong bảng điều khiển của mình:

Con nhận được tin nhắn: Con yêu mẹ. Cha mẹ đã nhận được tin nhắn: Con cũng yêu mẹ

## 7.8. Quá trình đệm đầu ra

Trong trường hợp đầu ra hoàn chỉnh trong bộ đệm của một quy trình con là đủ, không cần quản lý dữ liệu thông qua các sự kiện, quy trình\_ con cung cấp người thực thi phương pháp. Phương thức này có ba đối số:

chỉ huy: Một chuỗi dòng lệnh. không giống đẻ trứng và cái nĩa, truyền đối số cho một lệnh thông qua một mảng, đối số đầu tiên này chấp nhận một chuỗi lệnh đầy đủ, chẳng hạn như ps aux | nút grep.

các tùy chọn: Đây là một đối số tùy chọn.

° cwd ( String): Điều này đặt thư mục làm việc cho tiến trình lệnh.

° env ( Đối tượng): Đây là bản đồ của các cặp khóa-giá trị sẽ được hiển thị với tiến trình con.

° mã hóa ( Chuỗi): Đây là mã hóa luồng dữ liệu của trẻ. Giá trị mặc định là ' utf8 '.

° hết giờ ( Number): Điều này chỉ định mili giây để đợi quá trình hoàn tất, tại thời điểm đó, quá trình con sẽ được gửi các killSignal.maxBuffer giá trị.

° killSignal.maxBuffer ( Số): Đây là số byte tối đa được phép trên stdout hoặc là stderr. Khi vượt quá con số này, quá trình sẽ bị giết. Mặc định này là 200 KB.

° killSignal ( Chuỗi): Tiến trình con nhận được tín hiệu này sau một thời gian chờ. Mặc định này là SIGTERM.

• gọi lại: Điều này nhận được ba đối số: an lỗi đối tượng, nếu có; stdout

(a Đệm đối tượng chứa kết quả); stderr ( a Đệm đối tượng chứa dữ liệu lỗi, nếu có). Nếu quá trình bị giết, Error.signal sẽ chứa tín hiệu tiêu diệt.

Khi nào bạn muốn hành vi đệm của người thực thi nhưng đang nhắm mục tiêu một tệp Node, hãy sử dụng

tập tin thực thi. Quan trọng, thi hành không tạo ra một vỏ con mới, điều này làm cho nó ít tốn kém hơn khi chạy.

## 7.9. Giao tiếp với con bạn

Tất cả các trường hợp của ChildProcess mở rộng đối tượng EventEmitter, hiển thị các sự kiện hữu ích để quản lý các kết nối dữ liệu con. Ngoài ra, ChildProcess đối tượng cho thấy một số phương pháp hữu ích để tương tác trực tiếp với trẻ em. Bây giờ chúng ta hãy xem xét những điều đó, bắt đầu với các thuộc tính và phương thức:

child.connected: Khi một đứa trẻ bị ngắt kết nối với cha mẹ của nó qua child.disconnect (), cờ này sẽ được đặt thành sai.

child.stdin: Đây là một WordsStream tương ứng với tiêu chuẩn của trẻ trong.

child.stdout: Đây là một ReadableStream tương ứng với tiêu chuẩn ra của trẻ.

child.stderr: Đây là một ReadableStream tương ứng với sai số tiêu chuẩn của trẻ.

child.pid: Đây là một số nguyên đại diện cho ID quy trình (PID) được gán cho quy trình con.

child.kill: Điều này cố gắng kết thúc một tiến trình con, gửi cho nó một tín hiệu tùy chọn. Nếu không có tín hiệu nào được chỉ định, mặc định là SIGTERM ( để biết thêm về các tín hiệu, xem http://unixhelp.ed.ac.uk/CGI/man-cgi?signal+7). Trong khi phương pháp tên âm thanh đầu cuối, nó không được đảm bảo sẽ giết một quá trình — nó chỉ gửi một tín hiệu đến một quá trình. Nguy hiểm, nếu giết chết được thử trên một quá trình đã thoát, có thể một quá trình khác mới được gán PID của quá trình đã chết sẽ nhận được tín hiệu, với hậu quả không thể xác định được. Phương pháp này sẽ kích hoạt một đóng sự kiện mà tín hiệu được sử dụng để đóng quá trình.

child.disconnect (): Lệnh này cắt kết nối IPC giữa con và cha của nó. Sau đó đứa trẻ sẽ chết một cách duyên dáng, vì nó không có kênh IPC để giữ nó sống. Bạn cũng có thể gọi process.disconnect () từ bên trong chính đứa trẻ. Khi một đứa trẻ đã ngắt kết nối, kết nối cờ trên tham chiếu con đó sẽ được đặt thành sai.

## 7.10. Gửi tin nhắn cho tiến trình con

Như chúng ta đã thấy trong cuộc thảo luận về cái nĩa, và khi sử dụng ipc tùy chọn trên đẻ trứng, quy trình con có thể được gửi tin nhắn qua con. gửi, với thông báo được truyền làm đối số đầu tiên. Máy chủ TCP hoặc ổ cắm, có thể được chuyển cùng với thông báo như một lập luận thứ hai. Bằng cách này, một máy chủ TCP có thể lan truyền các yêu cầu trên nhiều tiến trình con. Ví dụ: máy chủ sau phân phối xử lý socket trên một số quy trình con bằng tổng số CPU hiện có. Mỗi con được chia nhỏ được cấp một ID duy nhất, ID này sẽ báo cáo khi bắt đầu. Bất cứ khi nào máy chủ TCP nhận được một ổ cắm, ổ cắm đó sẽ được chuyển như một bộ xử lý cho một tiến trình con ngẫu nhiên. Quy trình con đó sau đó sẽ gửi một phản hồi duy nhất, chứng tỏ rằng việc xử lý socket đang được phân phối.

Cha mẹ

var fork = demand ('child\_process'). fork; var net = request ('net'); var con = [];

request ('os'). cpus (). forEach (function (f, idx) { children.push (fork ("./ child.js", [idx]));

});

net.createServer (function (socket) { var rand = Math.floor (Math.random () \* children.length); con [rand] .send (null, socket); }). nghe (8080);

Đứa trẻ

var id = process.argv [2];

process.on ('message', function (n, socket) { socket.write ('child' + id + 'là máy chủ của bạn hôm nay. \ r \ n'); socket.end ();

});

Khởi động máy chủ mẹ trong cửa sổ đầu cuối. Trong một cửa sổ khác, chạy telnet

127.0.0.1 8080. Bạn sẽ thấy một cái gì đó tương tự như đầu ra sau, với một ID con ngẫu nhiên được hiển thị trên mỗi kết nối (giả sử tồn tại nhiều lõi):

Đang thử 127.0.0.1 ...

…

con 3 là máy chủ của bạn ngày hôm nay. Kết nối bị đóng bởi máy chủ nước ngoài.

Phân tích cú pháp một tệp bằng nhiều quy trình

Một trong những nhiệm vụ mà nhiều nhà phát triển sẽ đảm nhận là xây dựng bộ xử lý logfile. Một tệp nhật ký có thể rất lớn và dài nhiều MB. Bất kỳ chương trình đơn lẻ nào hoạt động trên một tệp rất lớn đều có thể dễ dàng gặp sự cố bộ nhớ hoặc đơn giản là chạy quá chậm. Nó có ý nghĩa khi xử lý một tệp lớn theo từng phần. Chúng tôi sẽ xây dựng một bộ xử lý nhật ký đơn giản có thể chia nhỏ một tệp lớn thành nhiều phần và chỉ định một trong số một số nhân viên con, chạy chúng song song.

Toàn bộ mã cho ví dụ này có thể được tìm thấy trong logproc thư mục của gói mã. Chúng tôi sẽ tập trung vào các thói quen chính:

Xác định số dòng trong logfile

Chia chúng thành nhiều phần bằng nhau

Tạo một phần tử con cho mỗi đoạn và chuyển nó hướng dẫn phân tích cú pháp

Lắp ráp và hiển thị kết quả

Để có số lượng từ trong tệp của chúng tôi, chúng tôi sử dụng wc ra lệnh với child.exec như được hiển thị trong đoạn mã sau:

child.exec ("wc -l" + tên tệp, hàm (e, fL) { fileLength = parseInt (fL.replace (tên tệp, "")); var fileRanges = [];

var oStart = 1; var oEnd = fileChunkLength;

while (oStart <fileLength) { fileRanges.push ({ offsetStart: oStart,

offsetEnd : oEnd

})

oStart = oEnd + 1;

oEnd = Math.min (oStart + fileChunkLength, fileLength);

}

Giả sử chúng tôi sử dụng fileChunkLength trong tổng số 500.000 dòng. Điều này có nghĩa là bốn quy trình con sẽ được tạo và mỗi quy trình sẽ được yêu cầu xử lý khoảng 500.000 dòng trong tệp của chúng tôi, chẳng hạn như từ 1 đến 500.000:

var w = child.fork ('bin / worker'); w.send ({

|  |  |
| --- | --- |
| tập tin offsetStart offsetEnd | : tên tệp,  : range.offsetStart,  : range.offsetEnd |

});

w.on ('message', function (chunkData) {

// chuyển dữ liệu kết quả vào một bộ giảm tốc. });

Bản thân mỗi công nhân này sẽ sử dụng một quy trình con để lấy phần được phân bổ của họ, sử dụng quyến rũ, người bản địa Trình chỉnh sửa luồng cho Unix:

process.on ('message', function (m) { var filename = m.file; var sed = "sed -n '" + m.offsetStart + "," + m.offsetEnd + "p'" + tên tệp;

var reader = request ('child\_process'). execute (sed, { maxBuffer: 1024 \* 1000000

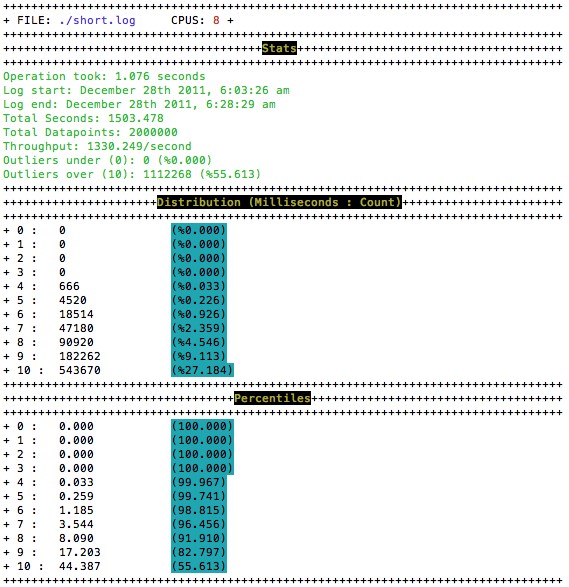
}, function (err, data, stderr) {

// Tách tập tin thành từng dòng và xử lý. //

data = data.split ("\ n");

. . .

Ở đây chúng tôi đang thực hiện lệnh sed –n '500001,1000001p' logfile.txt, kéo phạm vi dòng đã cho và trả về để xử lý. Sau khi chúng tôi xử lý xong các cột dữ liệu (thêm chúng, v.v.), con này sẽ trả lại dữ liệu của nó cho cái chính (như đã mô tả trước đó) và kết quả dữ liệu sẽ được ghi vào một tệp, nếu không sẽ bị thao túng hoặc gửi đến stdout, như được hiển thị trong kết quả sau:



Tệp đầy đủ cho ví dụ này dài hơn nhiều, nhưng tất cả mã bổ sung đó chỉ đơn thuần là định dạng và các chi tiết khác — quản lý quy trình con Node mà chúng tôi đã mô tả đủ để tạo ra một hệ thống song song để xử lý số sẽ xử lý hàng triệu dòng mã trong giây. Bằng cách sử dụng nhiều quy trình hơn, trải rộng trên nhiều lõi hơn, tốc độ phân tích cú pháp nhật ký có thể giảm hơn nữa.

Khi bạn có logproc thư mục, chạy lệnh npm cài đặt, và thực hiện phân tích cú pháp nhật ký ví dụ bằng lệnh bin nút / master.js –f ./short.log –rmin 0 –rmax 20. Bạn nên thấy một ánh xạ phân phối đơn giản của dữ liệu có trong ngắn gọn.log. Hãy tiếp tục và chơi với những con số.

## 7.11. Sử dụng mô-đun cụm

Như chúng ta đã thấy khi xử lý các tệp nhật ký lớn, mô hình của bộ điều khiển cha chính cho nhiều quy trình con phù hợp để mở rộng theo chiều dọc trong Node. Để đáp ứng điều này, API Node đã được tăng cường bởi cụm mô-đun, chính thức hóa mẫu này và giúp đạt được thành tích dễ dàng hơn. Tiếp tục với mục đích cốt lõi của Node là giúp xây dựng phần mềm mạng có khả năng mở rộng dễ dàng hơn, mục tiêu cụ thể của cluster là tạo điều kiện thuận lợi cho việc chia sẻ các cổng mạng giữa nhiều phần mềm con.

Ví dụ: đoạn mã sau tạo một nhóm các quy trình công nhân chia sẻ cùng một kết nối HTTP:

var cluster = request ('cluster'); var http = request ('http'); var numCPUs = request ('os'). cpus (). length;

if (cluster.isMaster) { for (var i = 0; i <numCPUs; i ++) { cluster.fork ();

}

}

if (cluster.isWorker) { http.createServer (function (req, res) { res.writeHead (200);

res.end ("Xin chào từ" + cluster.worker.id); }). nghe (8080);

}

Chúng tôi sẽ tìm hiểu chi tiết ngay sau đây. Bây giờ, hãy lưu ý rằng cluster.fork không có đối số. Cái gì cái nĩa không có lệnh hoặc đối số tệp làm gì? Trong một cụm, hành động mặc định là phân nhánh chương trình hiện tại. Chúng tôi thấy trong cụm. isMaster, hành động là để phân nhánh con (một cho mỗi CPU có sẵn). Khi chương trình này được thực thi lại trong bối cảnh rẽ nhánh, cluster.isWorker sẽ là thật và một máy chủ HTTP mới chạy trên một cổng chia sẻ được bắt đầu. Nhiều quy trình đang chia sẻ tải cho một máy chủ.

Khởi động và kết nối với máy chủ này bằng trình duyệt. Bạn sẽ thấy một cái gì đó giống như Xin chào từ 8, số nguyên tương ứng với duy nhất cluster.worker.id giá trị của nhân viên được giao trách nhiệm xử lý yêu cầu của bạn. Việc cân bằng giữa tất cả các worker được xử lý tự động, vì vậy việc làm mới trình duyệt của bạn một vài lần sẽ dẫn đến các ID worker khác nhau được hiển thị.

Sau đó, chúng ta sẽ xem xét một ví dụ về việc chia sẻ một máy chủ socket trên một cụm. Hiện tại, chúng ta sẽ bố trí API cụm, chia thành hai phần: các phương thức, thuộc tính và sự kiện có sẵn cho cụm chủ và những phần có sẵn cho phần con. Vì nhân viên trong ngữ cảnh này được xác định bằng cách sử dụng fork, tài liệu cho phương pháp đó quy trình\_ con cũng có thể áp dụng ở đây.

cluster.isMaster: Đây là giá trị Boolean cho biết quy trình có phải là quy trình chính hay không.

cluster.isWorker: Đây là giá trị Boolean cho biết liệu quá trình có được chia nhỏ từ một cái hay không.

cluster.worker: Điều này sẽ mang một tham chiếu đến đối tượng worker hiện tại, chỉ có sẵn cho một tiến trình con.

cluster.workers: Đây là một hàm băm chứa các tham chiếu đến tất cả các đối tượng worker đang hoạt động, được khóa bởi ID worker. Sử dụng điều này để lặp qua tất cả các đối tượng worker. Điều này chỉ tồn tại trong quy trình chính.

cluster.setupMaster ([cài đặt]): Đây là một cách thuận tiện để đi qua một bản đồ của các đối số mặc định sẽ được sử dụng khi một con được chia nhỏ. Nếu tất cả trẻ em sẽ phân tách cùng một tệp (như thường lệ), bạn sẽ tiết kiệm thời gian bằng cách đặt nó ở đây. Các giá trị mặc định có sẵn như sau:

° thi hành ( String): Đây là đường dẫn tệp đến tệp quy trình, mặc định đến \_\_ tên tệp.

° args ( Array): Điều này chứa các Chuỗi được gửi dưới dạng đối số cho tiến trình con. Mặc định là tìm nạp các đối số với process.argv. lát (2).

° im lặng ( Boolean): Điều này chỉ định có hay không gửi đầu ra cho cái stdio, mặc định là sai.

cluster.fork ([env]): Điều này tạo ra một quy trình mới của người làm việc. Chỉ quy trình chính mới có thể gọi phương thức này. Để hiển thị bản đồ các cặp khóa-giá trị cho môi trường quy trình của trẻ, hãy gửi một đối tượng tới env.

cluster.disconnect ([gọi lại]): Điều này được sử dụng để chấm dứt tất cả công nhân trong một cụm. Một khi tất cả các nhân viên đã chết một cách duyên dáng, quá trình cụm sẽ tự kết thúc nếu nó không có thêm sự kiện nào để chờ đợi. Để được thông báo khi tất cả trẻ em đã hết hạn, hãy vượt qua gọi lại.

## 7.12. Sự kiện cụm

Đối tượng cụm phát ra một số sự kiện được liệt kê như sau:

cái nĩa: Điều này được kích hoạt khi master cố gắng fork một đứa trẻ mới. Điều này không giống như Trực tuyến. Điều này nhận được một đối tượng worker.

Trực tuyến: Điều này được kích hoạt khi cái chính nhận được thông báo rằng một phần tử con bị ràng buộc hoàn toàn.

Điều này khác với cái nĩa sự kiện và nhận một đối tượng worker.

nghe: Khi nhân viên thực hiện một hành động yêu cầu nghe() gọi (chẳng hạn như khởi động một máy chủ HTTP), sự kiện này sẽ được kích hoạt trong cái chính. Sự kiện phát ra hai đối số: một đối tượng worker và đối tượng địa chỉ chứa địa chỉ, cảng, và addressType giá trị của kết nối.

ngắt kết nối: Điều này được gọi bất cứ khi nào con ngắt kết nối, có thể xảy ra thông qua các sự kiện thoát quy trình hoặc sau khi gọi con.kill (). Điều này sẽ kích hoạt trước khi lối ra sự kiện — chúng không giống nhau. Điều này nhận được một đối tượng worker.

lối ra: Bất cứ khi nào một đứa trẻ chết, sự kiện này sẽ được phát ra. Sự kiện nhận được ba đối số:

một đối tượng worker, số mã thoát và chuỗi tín hiệu, chẳng hạn như ĐĂNG KÝ, khiến quy trình bị giết.

thiết lập: Điều này được gọi là sau cluster.setupMaster đã thực hiện.

Thuộc tính đối tượng công nhân

Công nhân có các thuộc tính và phương pháp sau:

worker.id: Đây là ID duy nhất được chỉ định cho một nhân viên, cũng đại diện cho khóa của nhân viên trong cluster.workers mục lục.

worker.process: Điều này chỉ định một ChildProcess đối tượng tham chiếu đến một công nhân.

worker.suicide: Những công nhân gần đây đã có giết chết hoặc là ngắt kết nối kêu gọi họ sẽ có tự sát thuộc tính được đặt thành thật.

worker.send (tin nhắn, [sendHandle]): Tham khảo child\_process.fork (), mà đã được đề cập trước đó.

worker.kill ([signal]): Điều này giết chết một công nhân. Thạc sĩ có thể kiểm tra công nhân này tự sát tài sản để xác định cái chết là cố ý hay vô tình. Mặc định tín hiệu giá trị được gửi là SIGTERM.

worker.disconnect (): Điều này sẽ hướng dẫn nhân viên ngắt kết nối. Quan trọng là, các kết nối hiện có với nhân viên không bị chấm dứt ngay lập tức (như với

giết chết), nhưng được phép thoát bình thường trước khi công nhân ngắt kết nối hoàn toàn. Điều này là do các kết nối hiện có có thể tồn tại trong một thời gian rất dài. Một cách tốt là thường xuyên kiểm tra xem nhân viên đã thực sự ngắt kết nối hay chưa, có thể sử dụng thời gian chờ.

## 7.13. Sự kiện công nhân

Người lao động cũng phát ra các sự kiện, chẳng hạn như những sự kiện được đề cập trong danh sách sau:

thông điệp: Xem child\_process.fork

Trực tuyến: Điều này giống hệt với cluster.online, ngoại trừ việc kiểm tra chỉ chống lại nhân viên được chỉ định

nghe: Điều này giống hệt với cluster.listening, ngoại trừ việc kiểm tra chỉ chống lại nhân viên được chỉ định

ngắt kết nối: Điều này giống hệt với cluster.disconnect, ngoại trừ việc kiểm tra chỉ chống lại nhân viên được chỉ định

lối ra: Xem lối ra sự kiện cho quy trình\_ con

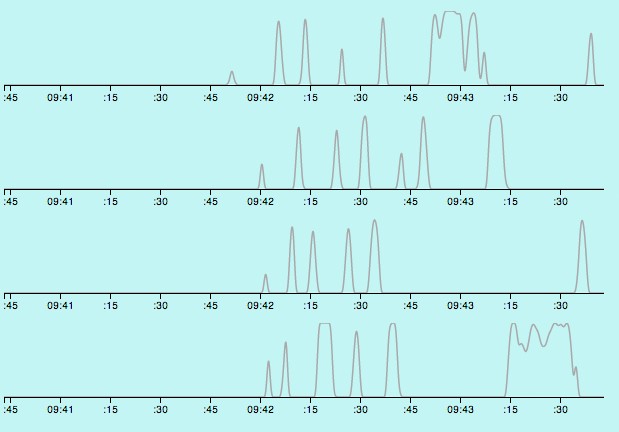
thiết lập: Điều này được gọi là sau cluster.setupMaster đã thực hiện

Bây giờ, bằng cách sử dụng những gì chúng ta biết về mô-đun cụm, hãy triển khai một công cụ thời gian thực để phân tích các luồng dữ liệu do nhiều người dùng đồng thời tương tác với một ứng dụng phát ra.

## 7.14. Cập nhật hoạt động thời gian thực của nhiều kết quả của nhân viên

Sử dụng những gì đã học, chúng tôi sẽ xây dựng một hệ thống đa xử lý để theo dõi hành vi của tất cả khách truy cập vào một trang web mẫu. Điều này sẽ bao gồm hai phân đoạn chính: thư viện máy khách do WebSocket cung cấp, sẽ phát sóng mỗi khi người dùng di chuyển chuột và giao diện quản trị hiển thị tương tác của người dùng cũng như khi người dùng kết nối và ngắt kết nối khỏi hệ thống. Mục tiêu của chúng tôi là chỉ ra cách

một hệ thống phức tạp hơn có thể được thiết kế (chẳng hạn như một hệ thống theo dõi và vẽ biểu đồ cho mỗi lần nhấp, vuốt hoặc tương tác khác mà người dùng có thể thực hiện). Giao diện quản trị cuối cùng sẽ hiển thị đồ thị hoạt động cho một số người dùng và giống như sau:



Bởi vì hệ thống này sẽ theo dõi vị trí X và Y của mỗi chuyển động chuột do tất cả người dùng thực hiện, chúng tôi sẽ truyền tải luồng dữ liệu liên tục này trên tất cả các lõi máy có sẵn bằng cách sử dụng cụm, với mỗi công nhân trong cụm chia sẻ gánh nặng mang một lượng lớn dữ liệu socket được đưa vào một cổng chia sẻ duy nhất.

Một nơi tốt để bắt đầu là thiết kế trang máy khách giả, trang này chỉ chịu trách nhiệm bắt tất cả các sự kiện di chuyển của chuột và phát chúng, thông qua WebSocket, tới máy chủ socket được phân cụm của chúng tôi. Chúng tôi đang sử dụng bản gốc WebSocket thực hiện; bạn có thể muốn sử dụng thư viện để xử lý các trình duyệt cũ hơn (chẳng hạn như Socket.IO):

<head>

<script> var connection = new WebSocket ('ws: //127.0.0.1: 8081', ['json']); connection.onopen = function () {

var userId = 'user' + Math.floor (Math.random () \* 10e10); document.onmousemove = function (e) { connect.send (JSON.stringify ({ id: userId, x: ví dụ,

y: ey

}));

}

}

</script> </head>

Ở đây, chúng ta chỉ cần bật tính năng cơ bản di chuột theo dõi, sẽ phát sóng vị trí của con chuột của người dùng trên mỗi chuyển động đến ổ cắm của chúng tôi. Ngoài ra, chúng tôi gửi cùng một ID người dùng duy nhất, vì danh tính khách hàng theo dõi sẽ rất quan trọng đối với chúng tôi sau này. Lưu ý rằng trong môi trường sản xuất, bạn sẽ muốn triển khai trình tạo ID duy nhất thông minh hơn, có thể là thông qua mô-đun xác thực phía máy chủ.

Để thông tin này đến được với các máy khách khác, một máy chủ ổ cắm tập trung phải được thiết lập. Như đã đề cập, chúng tôi sẽ muốn máy chủ socket này được phân cụm. Các quy trình con được phân cụm, mỗi bản sao của chương trình sau, sẽ xử lý dữ liệu chuột do máy khách gửi:

var SServer = request ('ws'). Server; var socketServer = new SServer ({port: 8081

});

socketServer.on ('kết nối', hàm (ổ cắm) { var lastMessage = null; var kill =

function () { if (lastMessage) {

process.send ({

kill: lastMessage.id

});

}

}

socket.on ('message', function (message) { lastMessage = JSON.parse (tin nhắn);

process.send (lastMessage);

});

socket.on ('đóng', giết); socket.on ('error', kill); });

Trong phần trình diễn này, chúng tôi đang sử dụng thư viện máy chủ socket rất nhanh và được thiết kế tốt của Einar Otto Stangvik, ws, đó là được lưu trữ trên GitHub tại https://github.com/einaros/ws.

Rất may, mã của chúng tôi vẫn rất đơn giản. Chúng tôi có một máy chủ socket lắng nghe thông báo (hãy nhớ rằng máy khách đang gửi một đối tượng bằng chuột X và Y, cũng như một ID người dùng). Cuối cùng, khi dữ liệu được nhận ( thông điệp sự kiện), chúng tôi phân tích cú pháp JSON đã nhận thành một đối tượng và chuyển nó trở lại chủ cụm của chúng tôi thông qua process.send. Lưu ý cũng như cách chúng tôi lưu trữ tin nhắn cuối cùng ( tin nhắn cuối cùng), được thực hiện vì lý do kế toán, vì khi kết nối chấm dứt, chúng tôi sẽ cần chuyển cùng ID người dùng cuối cùng được nhìn thấy trên kết nối này cho quản trị viên.

Các phần để bắt các chương trình phát dữ liệu khách hàng hiện đã được thiết lập. Sau khi nhận được dữ liệu này, nó sẽ được chuyển đến giao diện quản trị như hình trước đó như thế nào?

Chúng tôi đã thiết kế hệ thống này có tính đến khả năng mở rộng và chúng tôi muốn tách bộ sưu tập dữ liệu khỏi hệ thống truyền phát dữ liệu. Cụm máy chủ socket của chúng tôi có thể chấp nhận luồng dữ liệu liên tục từ nhiều nghìn máy khách và phải được tối ưu hóa để thực hiện điều đó. Nói cách khác, cụm nên ủy thác trách nhiệm phát dữ liệu hoạt động của chuột cho một hệ thống khác, thậm chí cho các máy chủ khác.

Trong chương tiếp theo, chúng ta sẽ xem xét các công cụ nhắn tin và mở rộng quy mô nâng cao hơn, chẳng hạn như hàng đợi tin nhắn và phát sóng UDP. Đối với mục đích của chúng tôi ở đây, chúng tôi sẽ chỉ cần tạo một máy chủ HTTP chịu trách nhiệm quản lý các kết nối từ quản trị viên và phát các bản cập nhật hoạt động chuột cho họ. Chúng tôi sẽ sử dụng SSE cho việc này, vì luồng dữ liệu chỉ cần là một chiều, từ máy chủ đến máy khách.

Máy chủ HTTP sẽ triển khai một hệ thống xác thực rất cơ bản cho thông tin đăng nhập của quản trị viên, giữ các kết nối thành công theo cách cho phép cụm ổ cắm của chúng tôi phát các bản cập nhật hoạt động của chuột cho tất cả. Nó cũng sẽ hoạt động như một máy chủ tệp tĩnh cơ bản, gửi cả HTML máy khách và quản trị viên khi được yêu cầu, mặc dù chúng tôi sẽ chỉ tập trung vào cách nó xử lý hai tuyến: admin / tên quản trị; và / nhận / tên quản trị. Khi đã hiểu máy chủ, chúng ta sẽ đi vào cách cụm ổ cắm của chúng ta kết nối với nó.

Tuyến đường đầu tiên / admin / adminname chịu trách nhiệm chính trong việc xác thực thông tin đăng nhập của quản trị viên, đồng thời đảm bảo rằng đây không phải là đăng nhập trùng lặp. Khi danh tính đó được thiết lập, chúng tôi có thể gửi lại một trang HTML cho giao diện quản trị. Mã khách hàng cụ thể được sử dụng để vẽ các biểu đồ trong hình trước đó sẽ không được thảo luận ở đây. Những gì chúng tôi cần là kết nối SSE với máy chủ của chúng tôi để các công cụ vẽ đồ thị của giao diện nhận được các bản cập nhật theo thời gian thực về hoạt động của chuột. Một số JavaScript trên trang của quản trị viên được trả lại thiết lập một kết nối như vậy:

var ev = new EventSource ('/ accept / adminname'); ev.addEventListener ("mở", function () { console.log ("Đã mở kết nối");

});

ev.addEventListener ("nội dung", hàm (dữ liệu) {

// Làm điều gì đó với dữ liệu chuột, như vẽ biểu đồ.

}

Trên máy chủ của mình, chúng tôi triển khai / nhận / tên quản trị tuyến đường:

if (phương thức === "nhận") {

// Quản trị viên không xác định; từ chối nếu (!

quản trị viên [adminId]) { trả về response.end ();

}

response.writeHead (200, {

"Content-Type": "text / event-stream",

"Cache-Control": "no-cache",

"Kết nối": "duy trì hoạt động"

});

response.write (":" + Array (2049) .join ("") + "\ n"); response.write ("thử lại: 2000 \ n");

response.on ("close", function () { quản trị viên [adminId] = {};

});

setInterval (function () { response.write ("dữ liệu: PING \ n \ n");

}, 15000);

quản trị viên [adminId] .socket = response; trở về; }

Mục đích chính của lộ trình này là thiết lập kết nối SSE và lưu trữ kết nối của quản trị viên, để sau này chúng tôi có thể phát tới nó.

Bây giờ chúng tôi sẽ thêm các phần sẽ chuyển dữ liệu hoạt động của chuột cùng với giao diện trực quan hóa. Mở rộng hệ thống con này trên các lõi bằng cách sử dụng cụm mô-đun là bước tiếp theo của chúng tôi. Cluster master bây giờ chỉ cần đợi dữ liệu chuột từ các con của nó phục vụ socket, như đã mô tả trước đây.

Chúng tôi sẽ sử dụng các ý tưởng tương tự được trình bày trong phần thảo luận trước đó về cụm, chỉ cần tạo mã máy chủ socket trước trên tất cả các CPU có sẵn:

if (cluster.isMaster) { var i;

for (i = 0; i <numCPUs; i ++) { cluster.fork ();

}

cụm

. on ('exit', function (worker, code, signal) { console.log ('worker' + worker.process.pid + 'die');

})

// ... thêm các trình nghe khác nếu cần

// Thiết lập trình nghe socket worker

Mục tiêuect.keys (cluster.workers) .forEach (function (id) { c luster.workers [id] .on ('message', function (msg) { var a;

cho (một trong các quản trị viên) { if (quản trị viên [a] .socket) { admin [a] .socket.write ("data:" + JSON.stringify (msg) +

"\ n \ n");

}

}

});

});

}

Dữ liệu hoạt động của chuột truyền vào một nhân viên cụm thông qua một ổ cắm và được phát qua process.send cho cụm chủ được mô tả trước đó. Trên mỗi thông báo của worker, chúng tôi chạy qua tất cả các quản trị viên được kết nối và gửi dữ liệu chuột đến giao diện trực quan của họ, sử dụng SSE. Các quản trị viên hiện có thể theo dõi sự xuất hiện của khách hàng, cũng như mức độ hoạt động cá nhân của họ.

## Tóm lược

Đây là chương đầu tiên chúng ta thực sự bắt đầu kiểm tra Node's khả năng mở rộng mục tiêu. Sau khi xem xét các lập luận khác nhau cho và chống lại các cách suy nghĩ khác nhau về tính đồng thời và song song, chúng tôi đã hiểu được cách Node đã duy trì thành công các lợi thế của phân luồng và xử lý song song trong khi gói tất cả sự phức tạp đó trong một mô hình đồng thời vừa dễ dàng lý do và mạnh mẽ.

Sau khi đi sâu hơn vào cách các quy trình hoạt động và cụ thể là cách các quy trình con có thể giao tiếp với nhau, thậm chí sinh ra các con khác, chúng tôi đã xem xét hai trường hợp sử dụng. Một ví dụ về cách kết hợp các quy trình lệnh Unix gốc một cách liền mạch với các quy trình customNode đã đưa chúng ta đến một kỹ thuật hiệu quả và đơn giản để xử lý các tệp lớn. Các cụm mô-đun sau đó đã được áp dụng cho vấn đề

về cách chia sẻ trách nhiệm xử lý một ổ cắm bận giữa nhiều nhân viên, khả năng chia sẻ các ổ cắm giữa các quy trình thể hiện một khía cạnh mạnh mẽ trong thiết kế của Node.

Sau khi thấy các ứng dụng Node có thể được mở rộng theo chiều dọc như thế nào, bây giờ chúng ta có thể xem xét tỷ lệ theo chiều ngang trên nhiều hệ thống và máy chủ. Trong chương tiếp theo, chúng ta sẽ tìm hiểu cách kết nối Node với các dịch vụ của bên thứ ba, chẳng hạn như Amazon và Facebook, giao tiếp giữa các mạng với hàng đợi tin nhắn, thiết lập nhiều máy chủ Node đằng sau proxy, v.v.

# CHƯƠNG 8

# Mở rộng ứng dụng

Tiến hóa là một quá trình phân nhánh và mở rộng liên tục.

Stephen Jay Gould

Khả năng mở rộng và hiệu suất không giống nhau:

Thuật ngữ "hiệu suất" và "khả năng mở rộng" thường được sử dụng thay thế cho nhau,

nhưng cả hai đều khác biệt: hiệu suất đo tốc độ mà một yêu cầu duy nhất có thể được thực hiện, trong khi khả năng mở rộng đo khả năng của một yêu cầu để duy trì hiệu suất của nó khi tải ngày càng tăng. Ví dụ: hiệu suất của một yêu cầu có thể được báo cáo là tạo ra phản hồi hợp lệ trong vòng ba giây, nhưng khả năng mở rộng của yêu cầu đo khả năng của yêu cầu để duy trì thời gian phản hồi ba giây đó khi tải người dùng tăng lên.

Steven Haines, "Pro Java EE 5"

Trong chương trước, chúng ta đã xem xét cách các cụm nút có thể được sử dụng để tăng hiệu suất của một ứng dụng. Thông qua việc sử dụng các cụm quy trình và công nhân, chúng tôi đã học được cách cung cấp kết quả một cách hiệu quả khi đối mặt với nhiều yêu cầu đồng thời. Chúng tôi đã học cách mở rộng Node theo chiều dọc, giữ nguyên một dấu chân (một máy chủ duy nhất) và tăng thông lượng bằng cách tích lũy sức mạnh của các CPU có sẵn.

Trong chương này chúng ta sẽ tập trung vào ngang khả năng mở rộng: ý tưởng là một ứng dụng bao gồm các đơn vị tự cung cấp và độc lập (máy chủ) có thể được mở rộng bằng cách thêm nhiều đơn vị hơn mà không làm thay đổi mã của ứng dụng.

Chúng tôi muốn tạo một kiến trúc trong đó có thể thêm hoặc bớt bất kỳ số lượng máy chủ hỗ trợ Node nào được đóng gói và tối ưu hóa để đáp ứng các nhu cầu thay đổi, tự động mở rộng quy mô mà không bao giờ yêu cầu viết lại hệ thống. Chúng tôi muốn chia sẻ công việc trên các hệ thống khác nhau, đẩy các yêu cầu đến Hệ điều hành, tới máy chủ khác, tới dịch vụ của bên thứ ba, đồng thời điều phối các hoạt động I / O đó một cách thông minh bằng cách sử dụng phương pháp đồng thời có sự kiện của Node.

Thông qua song song kiến trúc, hệ thống của chúng tôi có thể quản lý khối lượng dữ liệu tăng hiệu quả hơn. Các hệ thống chuyên biệt có thể được tách biệt khi cần thiết, thậm chí được mở rộng quy mô độc lập hoặc theo nhóm khác.

Node đặc biệt phù hợp để xử lý hai khía cạnh chính của kiến trúc được chia tỷ lệ theo chiều ngang.

Đầu tiên, Node thực thi I / O không chặn, sao cho việc chiếm giữ bất kỳ một đơn vị nào sẽ không gây ra một chuỗi khóa làm hỏng toàn bộ ứng dụng. Bởi vì không có thao tác I / O đơn lẻ nào sẽ chặn toàn bộ hệ thống, nên việc tích hợp các dịch vụ của bên thứ ba có thể được thực hiện một cách tự tin, khuyến khích một kiến trúc tách rời.

Thứ hai, Node rất coi trọng việc hỗ trợ càng nhiều giao thức truyền thông mạng nhanh càng tốt. Cho dù thông qua cơ sở dữ liệu dùng chung, hệ thống tệp được chia sẻ hay hàng đợi tin nhắn, mạng hiệu quả của Node và Suối các lớp cho phép nhiều máy chủ đồng bộ hóa nỗ lực của họ trong việc cân bằng tải. Chẳng hạn, việc có thể quản lý hiệu quả các kết nối socket được chia sẻ sẽ giúp ích khi mở rộng một cụm máy chủ giống như một cụm quy trình.

Trong chương này, chúng ta sẽ xem xét cách cân bằng lưu lượng giữa nhiều máy chủ chạy Node, cách các máy chủ riêng biệt này có thể giao tiếp và cách các cụm này có thể liên kết và hưởng lợi từ các dịch vụ đám mây chuyên biệt.

## 8.1. Khi nào mở rộng quy mô?

Lý thuyết xoay quanh việc mở rộng quy mô ứng dụng là một chủ đề phức tạp và thú vị tiếp tục được hoàn thiện và mở rộng. Một cuộc thảo luận toàn diện về chủ đề sẽ yêu cầu một số cuốn sách, được sắp xếp cho các môi trường và nhu cầu khác nhau. Đối với mục đích của chúng tôi, chúng tôi sẽ chỉ đơn giản là học cách nhận biết khi nào cần mở rộng quy mô (hoặc thậm chí thu nhỏ).

Có một kiến trúc linh hoạt có thể thêm và bớt tài nguyên khi cần thiết là điều cần thiết cho một chiến lược mở rộng quy mô linh hoạt. Giải pháp mở rộng quy mô dọc không phải lúc nào cũng đủ (chỉ cần thêm bộ nhớ hoặc CPU sẽ không mang lại những cải tiến cần thiết). Khi nào nên cân nhắc việc chia tỷ lệ ngang?

Một cách đơn giản nhưng hữu ích để kiểm tra mức sử dụng CPU và bộ nhớ do các tiến trình Node chạy trên máy chủ chỉ huy là sử dụng Unix ps lệnh, kết hợp với grep: ps –A u | nút grep

Điều này sẽ liệt kê tất cả các quy trình Node của bạn và phân bổ tài nguyên của chúng.

## 8.2. Độ trễ mạng

Khi thời gian phản hồi của mạng vượt quá một số ngưỡng, chẳng hạn như mỗi yêu cầu mất vài giây, có khả năng hệ thống đã vượt qua trạng thái ổn định.

Mặc dù cách dễ nhất để phát hiện ra vấn đề này là đợi khách hàng phàn nàn về các trang web chậm, nhưng tốt hơn hết là bạn nên tạo các bài kiểm tra căng thẳng có kiểm soát dựa trên môi trường ứng dụng hoặc máy chủ tương đương.

AB (Ghế dài Apache) là một cách đơn giản và dễ hiểu để thực hiện các bài kiểm tra căng thẳng đối với máy chủ. Công cụ này có thể được định cấu hình theo nhiều cách, nhưng loại kiểm tra bạn sẽ thực hiện để đo thời gian phản hồi mạng cho máy chủ của mình nói chung là đơn giản.

Ví dụ: hãy kiểm tra thời gian phản hồi cho máy chủ Node đơn giản này:

http.createServer (chức năng (yêu cầu, phản hồi) { response.writeHeader (200, {"Content-Type": "text / trơn"}); response.write ("Xin chào Thế giới");

response.end ();

}). nghe (2112)

Đây là cách người ta có thể kiểm tra việc chạy 10000 yêu cầu đối với máy chủ đó, với đồng thời là 100 (số lượng yêu cầu đồng thời): ab -n 10000 -c 100 http://yourserver.com/

Nếu mọi việc suôn sẻ, bạn sẽ nhận được một báo cáo tương tự như sau:

Mức đồng thời: 100

Thời gian thực hiện các bài kiểm tra: 9,658 giây

Hoàn thành các yêu cầu: 10000

Yêu cầu không thành công: 0

Viết lỗi: 0

Tổng số đã chuyển: 1120000 byte

HTML đã chuyển: 110000 byte

Yêu cầu mỗi giây: 1035,42 [# / giây] (trung bình)

|  |  |
| --- | --- |
| Thời gian cho mỗi yêu cầu:  Thời gian cho mỗi yêu cầu:  Tốc độ truyền tải: | 96,579 [mili giây] (trung bình)  0,966 [mili giây] (nghĩa là trên tất cả các yêu cầu đồng thời)  113,25 [Kbyte / giây] đã nhận được |

thời gian kết nối (mili giây)

|  |  |  |  |
| --- | --- | --- | --- |
| Kết nối:  Chế biến:  Đang chờ đợi:  Toàn bộ: | trung bình [+/- sd] tối thiểu  0 0 0,4  54 96 11,7  96 11,7  96 11,6 | 0  90  89  90 | tối đa  6  136  136  136 |

Phần trăm yêu cầu được phân phát trong một thời gian nhất định (mili giây)

50% 90

66% 98

…

99% 133

100% 136 (yêu cầu dài nhất)

Có rất nhiều thông tin hữu ích có trong báo cáo này. Cụ thể, một người nên tìm kiếm các yêu cầu không thành công và tỷ lệ phần trăm các yêu cầu đang chạy lâu.

Hệ thống kiểm tra phức tạp hơn nhiều tồn tại, nhưng ab là một ảnh chụp nhanh hiệu suất nhanh chóng và hiệu quả. Tập thói quen tạo môi trường thử nghiệm phản chiếu hệ thống sản xuất của bạn và thử nghiệm chúng.

Chạy ab trên cùng một máy chủ đang chạy quy trình Node mà bạn đang kiểm tra tất nhiên sẽ ảnh hưởng đến tốc độ kiểm tra. Bản thân trình chạy thử nghiệm sử dụng nhiều tài nguyên máy chủ, vì vậy kết quả của bạn sẽ bị sai lệch. Tài liệu đầy đủ cho ab có thể được tìm thấy trong liên kết sau: http://httpd.apache.org/docs/2.2/programs/ab.html

## 8.3. CPU nóng

Khi mức sử dụng CPU bắt đầu đạt mức tối đa, hãy bắt đầu nghĩ đến việc tăng số lượng đơn vị xử lý yêu cầu của khách hàng. Hãy nhớ rằng trong khi việc thêm một CPU mới vào máy một CPU sẽ mang lại những cải tiến tức thì và to lớn, nhưng việc thêm một CPU khác vào máy 32 lõi không nhất thiết sẽ mang lại một cải tiến tương đương. Tính toán chậm không phải lúc nào cũng là do tính toán chậm.

Như đã đề cập trước đây, htop là một cách tuyệt vời để có cái nhìn tổng quan nhanh về hiệu suất máy chủ của bạn. Bởi vì nó hiển thị trực quan tải được đặt trên mỗi lõi trong thời gian thực, đây là một cách tuyệt vời để có được ý tưởng về những gì đang xảy ra. Ngoài ra, trung bình tải của máy chủ của bạn được tóm tắt độc đáo với ba giá trị. Đây là một máy chủ vui vẻ:

Tải trung bình: 0,00 0,01 0,00

Những giá trị này nghĩa là gì? Mức trung bình tải "tốt" hay "xấu" là gì?

Cả ba số đều đo mức sử dụng CPU, hiển thị các phép đo được thực hiện ở các khoảng thời gian một, năm và mười lăm phút. Nhìn chung, có thể kỳ vọng rằng phụ tải ngắn hạn sẽ cao hơn phụ tải dài hạn. Nếu tính trung bình theo thời gian máy chủ của bạn không quá căng thẳng, thì có khả năng là khách hàng đang có trải nghiệm tốt.

Trên máy đơn lõi, tải trung bình nên duy trì trong khoảng 0,00 đến 1,00. Mọi yêu cầu sẽ có một số. Thời gian — câu hỏi là liệu yêu cầu có được thực hiện nhiều thời gian hơn cần thiết - và liệu có sự chậm trễ do quá tải hay không.

Nếu CPU có thể được coi là một cái ống, thì phép đo bằng 0,00 có nghĩa là không có ma sát quá mức hoặc độ trễ khi đẩy một giọt nước qua. Phép đo 1,00 cho thấy rằng đường ống của chúng tôi đang ở công suất của nó — nước chảy trôi chảy, nhưng bất kỳ nỗ lực bổ sung nào để đẩy nước qua sẽ phải đối mặt với sự chậm trễ hoặc áp suất ngược. Điều này chuyển thành độ trễ trên mạng, với các yêu cầu mới tham gia vào hàng đợi ngày càng tăng.

Máy đa lõi chỉ đơn giản là nhân ranh giới đo. Một máy có bốn lõi đạt công suất khi tải trung bình đạt 4,00.

Cách bạn chọn để phản ứng với mức trung bình tải phụ thuộc vào các chi tiết cụ thể của ứng dụng. Không có gì lạ khi các máy chủ chạy các mô hình toán học thấy mức trung bình CPU của chúng đạt đến công suất tối đa — trong những trường hợp như vậy bạn muốn tất cả tài nguyên sẵn có dành riêng cho việc thực hiện các phép tính. Mặt khác, một máy chủ tệp đang chạy hết công suất có thể đáng để điều tra.

Nói chung, mức trung bình tải trên 0,60 nên được điều tra. Mọi việc không khẩn cấp, nhưng có thể có một vấn đề xung quanh. Một máy chủ thường xuyên đạt 1,00 sau khi tất cả các tối ưu hóa đã biết đã được thực hiện là một ứng cử viên rõ ràng cho việc mở rộng quy mô, tất nhiên là bất kỳ máy chủ nào vượt quá mức trung bình đó.

Node cũng cung cấp thông tin quy trình gốc thông qua hệ điều hành mô-đun:

var os = request ('os'); //

Tải trung bình, như một Array console.log

(os.loadavg ());

// Tổng bộ nhớ còn trống

console.log (os.totalmem ()); console.log (os.freemem ());

// Thông tin về CPU, dưới dạng Array console.log (os.cpus ());

## 8.4. Sử dụng ổ cắm

Khi số lượng kết nối ổ cắm liên tục bắt đầu vượt quá khả năng của bất kỳ máy chủ Node đơn lẻ nào, tuy nhiên đã được tối ưu hóa, bạn sẽ cần phải nghĩ đến việc dàn trải các máy chủ xử lý ổ cắm của người dùng. Sử dụng socket.io điều đó là có thể để kiểm tra số lượng máy khách được kết nối bất kỳ lúc nào bằng cách sử dụng lệnh sau:

io.sockets.clients ()

Nói chung, tốt nhất là theo dõi số lượng kết nối ổ cắm web trong ứng dụng, thông qua một số loại hệ thống theo dõi / ghi nhật ký.

Nhiều bộ mô tả tệp

Khi số lượng bộ mô tả tệp được mở trong một hệ điều hành di chuột gần đến giới hạn của nó, có khả năng là quá nhiều quy trình Node đang hoạt động, tệp đang mở hoặc các bộ mô tả tệp khác (chẳng hạn như ổ cắm hoặc đường ống có tên) đang hoạt động. Nếu những con số cao này không phải do lỗi hoặc thiết kế xấu, thì đã đến lúc thêm một máy chủ mới.

Kiểm tra số lượng bộ mô tả tệp đang mở, thuộc bất kỳ loại nào, có thể được thực hiện bằng cách sử dụng lsof:

# lsof | wc –l // 1345

## 8.5. Dữ liệu creep

Khi lượng dữ liệu đang được quản lý bởi một máy chủ cơ sở dữ liệu duy nhất bắt đầu vượt quá hàng triệu hàng hoặc rất nhiều gigabyte bộ nhớ, đã đến lúc phải nghĩ đến việc mở rộng quy mô. Ở đây, bạn có thể chọn chỉ cần dành một máy chủ duy nhất cho cơ sở dữ liệu của mình, bắt đầu chia nhỏ cơ sở dữ liệu hoặc thậm chí chuyển sang giải pháp lưu trữ đám mây được quản lý sớm hơn là muộn hơn. Việc khôi phục sau lỗi lớp dữ liệu hiếm khi là cách khắc phục nhanh chóng và nói chung rất nguy hiểm nếu chỉ có một điểm lỗi đối với một thứ quan trọng như tất cả dữ liệu của bạn.

Nếu bạn đang sử dụng Redis, thông tin lệnh sẽ cung cấp hầu hết dữ liệu bạn cần để đưa ra các quyết định này. Ví dụ:

redis> thông tin

# Khách hàng khách hàng kết nối: 1

Block\_clients: 0 # Ký ức used\_memory: 17683488 used\_memory\_human: 16,86 triệu used\_memory\_rss: 165900288 used\_memory\_peak: 226730192 used\_memory\_peak\_human: 216,23 triệu used\_memory\_lua: 31744 mem\_fragmentation\_ratio: 9.38

# CPU used\_cpu\_sys: 13998,77 used\_cpu\_user: 21498.45 used\_cpu\_sys\_children: 1,60

used\_cpu\_user\_children: 7.19

. . .

Thông tin thêm về INFO có thể được tìm thấy tại liên kết này:

http://redis.io/commands/info

Đối với MongoDB, bạn có thể sử dụng db.stats () chỉ huy:

> db.stats (1024)

{ "bộ sưu tập": 3,

"đối tượng": 5,

"avgObjSize": 39,2,

"dataSize": 0,

"StorageSize": 12,

"numExtents": 3,

"chỉ mục": 1,

"indexSize": 7,

"fileSize": 196608,

"nsSizeMB": 16,

. . .

"ok": 1}

Vượt qua đối số 1024 cờ số liệu thống kê để hiển thị tất cả các giá trị theo kilobyte. Tìm thêm thông tin tại đây: http://docs.mongodb.org/manual/reference/command/dbStats/

## 8.6. Công cụ giám sát máy chủ

Có một số công cụ có sẵn để giám sát máy chủ, nhưng một số công cụ được thiết kế riêng cho Node. Một ứng cử viên mạnh mẽ là StrongOps ( http://strongloop.com/ strongloop-suite / strongops /) từ StrongLoop, một công ty được thành lập bởi những người đóng góp quan trọng cho cốt lõi của Node. Dịch vụ đám mây này được tích hợp dễ dàng với ứng dụng Node, cung cấp một bảng điều khiển hữu ích hiển thị mức sử dụng CPU, thời gian phản hồi trung bình, v.v.

Các công cụ giám sát tốt khác cần xem xét được liệt kê sau đây:

Hướng đạo sinh: https://scoutapp.com/

Nagios: http://www.nagios.org/

Munin: http://munin-monitoring.org/

Màn hình: http://mmonit.com/monit/

Chạy nhiều máy chủ Node

Có thể dễ dàng mua một số máy chủ và sau đó chạy một số quy trình Node trên chúng. Nhưng làm thế nào các máy chủ riêng biệt đó có thể được điều phối để chúng tạo thành một phần của

một ứng dụng duy nhất? Một khía cạnh của vấn đề này liên quan đến việc nhóm nhiều máy chủ giống nhau xung quanh một điểm vào duy nhất. Làm cách nào để chia sẻ kết nối máy khách trên một nhóm máy chủ?

## 8.7. Chuyển tiếp và đảo ngược proxy

A Ủy quyền là ai đó hoặc cái gì đó hành động thay mặt cho người khác.

A proxy chuyển tiếp thường làm việc thay mặt khách hàng trong mạng riêng, môi giới các yêu cầu đến mạng bên ngoài, chẳng hạn như truy xuất dữ liệu từ Internet. Trước đó trong cuốn sách này, chúng ta đã xem xét cách người ta có thể thiết lập một máy chủ proxy bằng Node, nơi máy chủ Node hoạt động như một trung gian, chuyển tiếp các yêu cầu từ máy khách

tới các máy chủ mạng khác, thường là qua Internet. Các nhà cung cấp web ban đầu như AOL hoạt động theo cách sau:

khách hàng

khách hàng

ở đằng trước

Ủy quyền

Internet

khách hàng

Mạng riêng tư

Quản trị viên mạng sử dụng proxy chuyển tiếp khi họ phải hạn chế quyền truy cập vào thế giới bên ngoài, tức là Internet. Nếu người dùng mạng đang tải xuống phần mềm độc hại từ somebadwebsite.com thông qua tệp đính kèm e-mail, quản trị viên có thể chặn quyền truy cập vào vị trí đó. Các giới hạn truy cập vào các trang mạng xã hội có thể được áp dụng đối với mạng văn phòng. Một số quốc gia thậm chí còn hạn chế quyền truy cập vào các trang web công cộng theo cách này.

A proxy ngược, không có gì đáng ngạc nhiên, hoạt động theo cách ngược lại, chấp nhận các yêu cầu từ một mạng công cộng và phục vụ các yêu cầu đó trong một mạng riêng mà khách hàng có ít khả năng hiển thị. Quyền truy cập trực tiếp vào máy chủ của máy khách trước tiên được ủy quyền cho một proxy ngược:

người phục vụ

đảo ngược

Ủy quyền

Internet

người phục vụ

người phục vụ

Mạng riêng tư

Đây là loại proxy mà chúng tôi có thể sử dụng để cân bằng các yêu cầu từ máy khách trên nhiều máy chủ Node. Khách hàng X không giao tiếp trực tiếp với bất kỳ máy chủ nhất định nào. Một nhà môi giới là điểm tiếp xúc đầu tiên, có thể chỉ đạo X đến máy chủ ít tải hơn hoặc máy chủ nằm gần X, hoặc theo một cách nào đó là máy chủ tốt nhất cho X để truy cập vào lúc này.

Chúng ta sẽ xem xét cách triển khai các proxy ngược khi mở rộng Node, thảo luận về các triển khai sử dụng Nginx và những người sử dụng mô-đun Node gốc.

## 8.8. Nginx làm proxy

Nginx (phát âm là "Engine X") là một lựa chọn phổ biến khi ẩn các máy chủ Node sau proxy. Nginx là một máy chủ web hiệu suất cao rất phổ biến thường được sử dụng làm máy chủ proxy. Có một số điều ngẫu nhiên trong thực tế là Nginx là một lựa chọn phổ biến với các nhà phát triển Node, dựa trên thiết kế của nó:

Nginx có thể phục vụ nhiều yêu cầu hơn mỗi giây với [ít] tài nguyên hơn vì kiến trúc của nó. Nó bao gồm một quy trình chính, ủy quyền làm việc cho một hoặc nhiều quy trình công nhân. Mỗi worker xử lý nhiều yêu cầu theo hướng sự kiện hoặc không đồng bộ bằng cách sử dụng chức năng đặc biệt từ hạt nhân Linux (epoll / select / thăm dò ý kiến). Điều này cho phép Nginx xử lý một số lượng lớn các yêu cầu đồng thời một cách nhanh chóng với chi phí rất thấp.

- http://www.linuxjournal.com/magazine/nginx-high-performanceweb-server-and-reverse-proxy

Nginx cũng giúp cân bằng tải đơn giản rất dễ dàng. Trong các ví dụ của chúng tôi, chúng tôi sẽ thấy cách proxy thông qua Nginx đi kèm với cân bằng tải "ra khỏi hộp".

Giả định rằng bạn đã cài đặt Nginx (nếu chưa, vui lòng truy cập http: //wiki.nginx. org / Main để bắt đầu). Bây giờ chúng ta phải tăng cường nginx.conf tập tin, cụ thể là http phần. Mục tiêu của chúng tôi là tạo một máy chủ Nginx duy nhất có Duy Nhất trách nhiệm là phân phối đồng đều các yêu cầu của máy khách trên một số máy chủ Node, sao cho dung lượng của toàn bộ hệ thống có thể được mở rộng mà không thay đổi bất kỳ điều gì về cách máy khách có thể hoặc nên kết nối với ứng dụng của chúng tôi:

Internet

yêu cầu

Node one.example.com

Thượng nguồn

one.example.com

two.example.com

one.example.com

Node two.example.com

Node three.example.com

NGINX

Trong http phần của nginx.config được xác định một số Thượng nguồn máy chủ sẽ là ứng cử viên để chuyển hướng:

http {

. . .

ngược dòng app\_myservers { máy chủ first.example.com; máy chủ second.example.com;

máy chủ third.example.com;

} . . . }

Bây giờ chúng tôi đã thiết lập nhóm ứng viên, chúng tôi cần định cấu hình Nginx để nó sẽ chuyển tiếp các yêu cầu đến một trong các thành viên của nó:

người phục vụ {

nghe 0,0.0.0:80;

server\_name example.com my\_website;

. . .

vị trí / {

proxy\_http\_version 1.1; proxy\_set\_header Nâng cấp $ http\_upgrade; proxy\_set\_header Kết nối "nâng cấp"; proxy\_set\_header X-Real-IP $ remote\_addr;

proxy\_set\_header X-Được chuyển tiếp-Cho $ proxy\_add\_x\_nticed\_for; proxy\_set\_header Máy chủ lưu trữ $ http\_host;

proxy\_set\_header X-NginX-Proxy true;

proxy\_pass http: // app\_myservers /;

proxy\_redirect tắt;

}

}

Điểm mấu chốt là dòng này:

proxy\_pass http: // app\_myservers /;

Lưu ý cách đặt tên app\_myservers khớp với tên của định nghĩa ngược dòng của chúng tôi. Điều này sẽ làm cho những gì đang xảy ra rõ ràng: một máy chủ Nginx lắng nghe trên cổng 80 sẽ chuyển yêu cầu đến một định nghĩa máy chủ như được chứa trong app\_myservers. Nếu định nghĩa ngược dòng chỉ có một máy chủ trong đó, máy chủ đó sẽ nhận được tất cả lưu lượng truy cập. Nếu một số máy chủ được xác định, Nginx sẽ cố gắng phân phối lưu lượng truy cập đồng đều giữa chúng.

Cũng có thể cân bằng tải trên một số máy chủ cục bộ bằng cách sử dụng cùng một kỹ thuật.

Một người chỉ cần chạy các máy chủ Node khác nhau trên các cổng khác nhau, chẳng hạn như máy chủ 127.0.0.1:8001; người phục vụ 127.0.0.1:8002; ...

Bởi vì chúng tôi có thể sẽ muốn kiểm soát chính xác hơn cách lưu lượng truy cập được phân phối trên các máy chủ ngược dòng của chúng tôi, nên có các chỉ thị khác có thể được áp dụng cho các định nghĩa máy chủ ngược dòng.

Để kiểm soát tỷ trọng tương đối của phân phối lưu lượng, chúng tôi sử dụng cân nặng chỉ thị:

ngược dòng app\_myservers { máy chủ first.example.com weight = 1; máy chủ second.example.com weight = 2;

máy chủ thứ ba.example.com weight = 4;

}

Định nghĩa này cho biết Nginx phân phối lượng tải cho máy chủ thứ hai nhiều gấp đôi so với máy chủ đầu tiên và gấp bốn lần cho máy chủ thứ ba. Ví dụ, máy chủ có nhiều bộ nhớ hơn hoặc CPU có thể được ưu tiên. Một cách khác để sử dụng hệ thống này là tạo một A / B kịch bản thử nghiệm, trong đó một máy chủ chứa thiết kế mới được đề xuất nhận được một số phần nhỏ trong tổng lưu lượng truy cập, sao cho các chỉ số trên máy chủ thử nghiệm (doanh số, lượt tải xuống, thời lượng tương tác, v.v.) có thể được so sánh với mức trung bình rộng hơn.

Hai lệnh hữu ích khác có sẵn, hoạt động cùng nhau để quản lý các lỗi kết nối:

max\_fails: Đây là số lần giao tiếp với máy chủ không thành công trước khi đánh dấu máy chủ đó là không hoạt động. Khoảng thời gian mà những hư hỏng này phải xảy ra được xác định bởi fail\_timeout.

fail\_timeout: Khoảng thời gian trong đó max\_fails phải xảy ra, cho biết rằng một máy chủ không hoạt động. Con số này cũng cho biết khoảng thời gian sau khi máy chủ được đánh dấu là không hoạt động mà Nginx sẽ cố gắng truy cập lại máy chủ được gắn cờ.

Ví dụ:

ngược dòng app\_myservers { server first.myservers.com weight = 1 max\_fails = 2 fail\_ timeout = 20 giây; server second.myservers.com weight = 2 max\_fails = 10 fail\_ timeout = 5m;

máy chủ third.myservers.com weight = 4;

}

Tài liệu đầy đủ cho Nginx có thể được tìm thấy tại liên kết sau: http://wiki.nginx.org/Configuration

## 8.9. Sử dụng HTTP Proxy

Node được thiết kế để tạo điều kiện thuận lợi cho việc tạo ra phần mềm mạng, vì vậy không có gì ngạc nhiên khi một số mô-đun ủy quyền đã được phát triển. Nhóm nghiên cứu tại NodeJitsu đã phát hành proxy mà họ sử dụng trong sản xuất, Http Proxy. Hãy xem cách chúng ta sẽ sử dụng nó để định tuyến các yêu cầu đến các máy chủ Node khác nhau.

Không giống như với Nginx, toàn bộ ngăn xếp định tuyến của chúng tôi sẽ tồn tại trong Node. Máy chủ One Node sẽ chạy proxy của chúng tôi, lắng nghe trên cổng 80. Chúng tôi sẽ trình bày ba trường hợp sau:

Chạy nhiều máy chủ Node trên các cổng riêng biệt trên cùng một máy.

Sử dụng một hộp như một bộ định tuyến thuần túy, ủy quyền cho các URL bên ngoài.

## 8.10. Tạo bộ cân bằng tải vòng cơ bản.

Ví dụ ban đầu, hãy xem cách sử dụng mô-đun này để chuyển hướng các yêu cầu:

var httpProxy = request ('http-proxy'); var proxy = httpProxy.createServer ({target: {

máy chủ: 'www.example.com', cổng: 80

}

}). nghe (80);

Bằng cách khởi động máy chủ này trên cổng 80 của máy cục bộ, chúng tôi có thể chuyển hướng người dùng đến một URL khác.

Để chạy một số máy chủ Node riêng biệt, mỗi máy chủ phản hồi với một URL khác nhau, trên một máy, người ta chỉ cần xác định một bộ định tuyến:

var httpProxy = httpProxy.createServer ({

bộ định tuyến: {

|  |  |
| --- | --- |
| 'www.mywebsite.com'  'www.myothersite.com' | : '127.0.0.1:8001',  : '127.0.0.1:8002', |

}

});

httpProxy.listen (80);

Đối với mỗi trang web riêng biệt của mình, giờ đây bạn có thể trỏ máy chủ định danh DNS của mình (thông qua bản ghi ANAME hoặc CNAME) đến cùng một điểm cuối (bất cứ nơi nào chương trình Node này đang chạy) và chúng sẽ phân giải tới các máy chủ Node khác nhau. Điều này rất hữu ích khi bạn muốn chạy một số trang web nhưng không muốn tạo một máy chủ vật lý mới cho mỗi trang. Một chiến lược khác là xử lý các đường dẫn khác nhau trong cùng một trang web trên các máy chủ Node khác nhau:

var httpProxy = httpProxy.createServer ({

bộ định tuyến: {

'www.mywebsite.com/friends': '127.0.0.1:8001',

'www.mywebsite.com/foes' : '127.0.0.1:8002',

}

});

httpProxy.listen (80);

Điều này cho phép chức năng chuyên biệt trong ứng dụng của bạn được xử lý bởi các máy chủ được cấu hình duy nhất.

Việc thiết lập bộ cân bằng tải cũng rất đơn giản. Như với Nginx's Thượng nguồn chỉ thị, chúng tôi chỉ cần liệt kê các máy chủ cần được cân bằng và chuyển qua chúng:

var httpProxy = request ('http-proxy'); var address = [

{host: 'one.example.com', port: 80}, {host: 'two.example.com', port: 80}

];

httpProxy.createServer (function (req, res, proxy) { var target = address.shift (); proxy.proxyRequest (req, res, target);

address.push (đích); }). nghe (80);

Không giống như với Nginx, chúng tôi chịu trách nhiệm thực hiện việc cân bằng thực tế. Trong ví dụ này, chúng tôi đối xử bình đẳng với các máy chủ, tuần hoàn theo thứ tự. Sau khi máy chủ đã chọn được ủy quyền, nó sẽ được trả về phần phía sau của danh sách.

Rõ ràng là ví dụ này có thể dễ dàng mở rộng để phù hợp với các lệnh khác, như của Nginx cân nặng.

Một lựa chọn tốt khác để ủy quyền là bouncy mô-đun được tạo bởi James Halliday. Bạn sẽ tìm thấy kho lưu trữ trong liên kết sau:

https://github.com/substack/bouncy

## 8.11. Hàng đợi tin nhắn - RabbitMQ

Một trong những cách tốt nhất để đảm bảo rằng các máy chủ phân tán duy trì một kênh giao tiếp đáng tin cậy là gộp sự phức tạp của các cuộc gọi thủ tục từ xa thành một đơn vị riêng biệt — một hàng đợi nhắn tin. Khi một quy trình muốn gửi thông báo đến một quy trình khác, thông báo có thể đơn giản được đặt vào hàng đợi này — giống như danh sách việc cần làm cho ứng dụng của bạn — với dịch vụ hàng đợi thực hiện công việc đảm bảo thông báo được gửi cũng như phân phối bất kỳ trả lời lại cho người gửi ban đầu.

Có một số hàng đợi tin nhắn cấp doanh nghiệp có sẵn, nhiều hàng đợi trong số họ triển khai AMQP ( Giao thức xếp hàng thư nâng cao). Chúng tôi sẽ tập trung vào một triển khai rất ổn định và nổi tiếng: RabbitMQ.

Để cài đặt RabbitMQ trong môi trường của bạn, hãy làm theo hướng dẫn được tìm thấy trong trang web: http://www.rabbitmq.com/download.html.

Bạn cũng sẽ cần cài đặt Erlang ( hướng dẫn được đưa ra theo Khách hàng chính thức ở liên kết trước).

Sau khi cài đặt, bạn sẽ khởi động máy chủ RabbitMQ bằng lệnh này:

bắt đầu dịch vụ thỏmq-server

Để tương tác với RabbitMQ bằng Node, chúng tôi sẽ sử dụng mô-đun node-amqp được tạo bởi Theo Schlossnagle:

npm cài đặt amqp

Để sử dụng hàng đợi tin nhắn, trước tiên người ta phải tạo một người tiêu dùng — một ràng buộc với RabbitMQ sẽ lắng nghe các tin nhắn được xuất bản vào hàng đợi. Người tiêu dùng cơ bản nhất sẽ lắng nghe tất cả các thông điệp:

var amqp = request ('amqp');

var Consumer = amqp.createConnection ({host: 'localhost', cổng: 5672});

trao đổi var;

Consumer.on ('sẵn sàng', function () {

Exchange = Consumer.exchange ('node-topic-exchange', {type:

"đề tài"});

Consumer.queue ('node-topic-queue', function (q) {

q.bind (trao đổi, '#');

q.subscribe (chức năng (tin nhắn) {

// Tin nhắn là bộ đệm

console.log (message.data.toString ('utf8'));

});

});

});

Bây giờ chúng tôi đang lắng nghe các thông báo từ máy chủ RabbitMQ được liên kết với cổng 5672.

Khi người tiêu dùng này thiết lập kết nối, nó sẽ thiết lập tên của hàng đợi mà nó sẽ lắng nghe và trói buộc đến một đổi. Trong ví dụ này, chúng tôi tạo một chủ đề

đổi ( mặc định), đặt cho nó một tên duy nhất. Chúng tôi cũng cho biết rằng chúng tôi muốn lắng nghe tất cả tin nhắn qua #. Tất cả những gì còn lại cần làm là đăng ký vào hàng đợi, nhận một đối tượng tin nhắn. Chúng tôi sẽ tìm hiểu thêm về đối tượng tin nhắn khi chúng tôi tiến bộ. Còn bây giờ, hãy lưu ý điều quan trọng dữ liệu thuộc tính, chứa các tin nhắn đã gửi.

Bây giờ chúng ta đã thiết lập được người tiêu dùng, hãy công bố một thông báo tới sàn giao dịch. Nếu mọi việc suôn sẻ, chúng tôi sẽ thấy thông báo đã gửi xuất hiện trong bảng điều khiển của chúng tôi:

Consumer.on ('sẵn sàng', function () {

// ...

exchange.publish ("some-topic", "Hello!");

});

// Xin chào!

Chúng tôi đã học đủ để triển khai các công cụ mở rộng quy mô hữu ích. Nếu chúng ta có một số quy trình Node phân tán, ngay cả trên các máy chủ vật lý khác nhau, mỗi có thể gửi tin nhắn cho nhau một cách đáng tin cậy qua RabbitMQ. Mỗi quy trình chỉ cần thực hiện một người đăng ký hàng đợi trao đổi để nhận tin nhắn và trao đổi nhà xuất bản để gửi tin nhắn.

## 8.12. Các loại trao đổi

RabbitMQ cung cấp ba loại trao đổi: trực tiếp, fanout, và đề tài. Sự khác biệt xuất hiện trong cách mỗi loại quy trình trao đổi phím định tuyến —Đối số đầu tiên được gửi đến trao đổi. xuất bản.

Trao đổi trực tiếp khớp trực tiếp với các khóa định tuyến. Một ràng buộc hàng đợi như sau phù hợp với chỉ có tin nhắn được gửi đến ' phòng-1 ':

queue.bind (trao đổi, 'phòng-1');

Bởi vì không cần phân tích cú pháp, trao đổi trực tiếp có thể xử lý nhiều thông điệp hơn so với trao đổi chủ đề trong một khoảng thời gian nhất định.

Trao đổi fanout là bừa bãi; nó định tuyến các thông điệp đến tất cả các hàng đợi liên kết với nó, bỏ qua các khóa định tuyến. Loại trao đổi này được sử dụng cho các chương trình phát sóng rộng rãi.

Trao đổi chủ đề khớp với các khóa định tuyến dựa trên các ký tự đại diện # và \*. Không giống như các loại khác, khóa định tuyến để trao đổi chủ đề phải bao gồm các từ được phân tách bằng dấu chấm— " động vật.dogs.poodle ", ví dụ. Một # khớp với không hoặc nhiều từ — nó sẽ khớp với mọi tin nhắn (như chúng ta đã thấy trong ví dụ trước), giống như một trao đổi fanout. Ký tự đại diện khác là \* và ký tự này khớp với chính xác một từ.

Trao đổi trực tiếp và trao đổi fanout có thể được thực hiện bằng cách sử dụng mã gần giống như ví dụ trao đổi chủ đề đã cho, chỉ yêu cầu loại trao đổi được thay đổi và các hoạt động ràng buộc phải biết cách chúng sẽ được liên kết với các khóa định tuyến (người đăng ký fanout nhận được tất cả các tin nhắn, bất kể của khóa; đối với trực tiếp khóa định tuyến phải khớp trực tiếp).

Ví dụ cuối cùng này sẽ hướng dẫn về cách hoạt động của các cuộc trao đổi chủ đề. Chúng tôi sẽ tạo ba hàng đợi với các quy tắc đối sánh khác nhau, lọc các thông báo mà mỗi hàng đợi nhận được từ trao đổi:

Consumer.on ('sẵn sàng', function () {

// Khi tất cả 3 hàng đợi đã sẵn sàng, hãy xuất bản. var cnt = 3;

var queueReady = function () { nếu (- cnt> 0) { trở về;

}

Exchange.publish ("animal.dogs.poodles", "Poodle!");

exchange.publish ("animal.dogs.dachshund", "Dachshund!"); Exchange.publish ("animal.cats.shorthaired", "Shorthaired

Con mèo!"); exchange.publish ("animal.dogs.shorthaired", "Shorthaired Chó!");

Exchange.publish ("animal.misc", "Misc!");

}

var exchange = Consumer.exchange ('chuyên đề', {type: "topic"}); Consumer.queue ('queue-1', function

(q) {

q.bind (trao đổi, 'động vật. \*. shorthaired');

q.subscribe (chức năng (tin nhắn) { console.log ('động vật. \*. shorthaired ->' + message.data.toString ('utf8'));

});

queueReady ();

});

Consumer.queue ('queue-2', function (q) {

q.bind (trao đổi, '#');

q.subscribe (chức năng (tin nhắn) { console.log ('# ->' + message.data.toString ('utf8'));

});

queueReady ();

});

Consumer.queue ('queue-3', function (q) {

q.bind (trao đổi, '\* .cats. \*');

q.subscribe (chức năng (tin nhắn) { console.log ('\*. mèo. \* ->' + message.data.toString ('utf8'));

});

queueReady ();

});

});

// # -> Poodle!

// động vật. \*. shorthaired -> Mèo lông ngắn!

// \*. mèo. \* -> Mèo lông ngắn!

// # -> Dachshund!

// # -> Mèo lông ngắn!

// động vật. \*. shorthaired -> Chó có lông ngắn!

// # -> Chó lông ngắn!

// # -> Khác!

Các node-amqp mô-đun chứa các phương pháp khác để kiểm soát các kết nối, hàng đợi và trao đổi; đặc biệt, nó chứa các phương pháp để loại bỏ hàng đợi khỏi trao đổi và người đăng ký khỏi hàng đợi. Nói chung, thay đổi trang điểm của

hàng đợi đang chạy có thể dẫn đến lỗi không mong muốn, vì vậy hãy sử dụng những lỗi này một cách thận trọng.

Để tìm hiểu thêm về AMQP (và các tùy chọn có sẵn khi thiết lập với node-amqp) truy cập liên kết sau: http://www.rabbitmq.com/tutorials/amqp-concept.html

## 8.13. Sử dụng mô-đun UDP của Node

UDP (Giao thức sơ đồ người dùng) là một giao thức nhắn tin Internet lõi nhẹ, cho phép các máy chủ truyền tải một cách ngắn gọn biểu đồ dữ liệu. UDP được thiết kế với tối thiểu chi phí giao thức, cơ chế phân phối, đặt hàng và ngăn chặn trùng lặp nhằm đảm bảo hiệu suất cao. UDP là một lựa chọn tốt khi không yêu cầu độ tin cậy hoàn hảo và truyền tốc độ cao, chẳng hạn như những gì được tìm thấy trong các trò chơi video nối mạng và các ứng dụng hội nghị truyền hình.

Điều này không có nghĩa là UDP thường không đáng tin cậy. Trong hầu hết các ứng dụng, nó gửi tin nhắn với xác suất cao. Nó chỉ đơn giản là không phù hợp khi hoàn hảo độ tin cậy là cần thiết, chẳng hạn như trong một ứng dụng ngân hàng. Nó là một ứng cử viên tuyệt vời cho các ứng dụng theo dõi và ghi nhật ký, và cho các dịch vụ nhắn tin không quan trọng.

Tạo máy chủ UDP với Node rất đơn giản:

var dgram = request ("dgram"); var socket = dgram.createSocket ("udp4"); socket.on ("tin nhắn", hàm

(tin nhắn, thông tin) { console.log ("socket got:" + msg + "from" + info.address + ":" + info.port);

});

socket.bind (41234); socket.on ("đang nghe", function () { console.log ("Nghe datagram.");

});

Các trói buộc lệnh nhận ba đối số, như sau:

Hải cảng: Các Số nguyên số cổng.

Địa chỉ: Đây là một địa chỉ tùy chọn. Nếu điều này không được chỉ định, hệ điều hành sẽ cố gắng lắng nghe trên tất cả các địa chỉ (thường là những gì bạn muốn). Bạn cũng có thể thử sử dụng 0.0.0.0 một cách rõ ràng.

gọi lại: Đây là một lệnh gọi lại tùy chọn, không nhận được đối số.

Ổ cắm này bây giờ sẽ phát ra một thông điệp sự kiện bất cứ khi nào nó nhận được một gói dữ liệu thông qua cổng 41234. Sự kiện gọi lại nhận bản thân thông báo dưới dạng tham số đầu tiên và bản đồ thông tin gói là tham số thứ hai:

Địa chỉ: IP gốc.

gia đình: Một trong IPv4 hoặc IPv6.

Hải cảng: Cổng xuất phát.

kích thước: Kích thước của tin nhắn tính bằng byte.

Bản đồ này tương tự như bản đồ được trả về khi gọi socket.address ().

Ngoài các sự kiện nghe và tin nhắn, một ổ cắm UDP cũng phát ra đóng và lỗi sự kiện sau đó nhận được một lỗi đối tượng bất cứ khi nào xảy ra lỗi. Để đóng ổ cắm UDP (và kích hoạt sự kiện đóng), hãy sử dụng server.close ().

Gửi tin nhắn thậm chí còn dễ dàng hơn:

var client = dgram.createSocket ("udp4"); var message = new Buffer ("UDP nói Xin chào!");

client.send (message, 0, message.length, 41234, "localhost", function (err, byte) {

client.close (); });

Các gửi phương pháp có dạng client.send (bộ đệm, độ lệch, độ dài, cổng, máy chủ lưu trữ, cuộc gọi lại):

đệm: Một bộ đệm chứa gói dữ liệu được gửi đi

bù lại: Số nguyên cho biết vị trí trong bộ đệm nơi bắt đầu sơ đồ dữ liệu

chiều dài: Số byte trong một sơ đồ dữ liệu. Kết hợp với bù lại giá trị này xác định bộ đệm datagramwithin đầy đủ

Hải cảng: Số nguyên xác định cổng đích

Địa chỉ: Một chuỗi chỉ ra IP đích cho datagram

gọi lại: Một chức năng gọi lại tùy chọn, được gọi sau khi quá trình gửi diễn ra

Kích thước của một sơ đồ dữ liệu không được vượt quá 65507 byte, bằng 2 ^ 16-1 ( 65535) byte trừ đi 8 byte được tiêu đề UDP sử dụng trừ đi 20 byte được tiêu đề IP sử dụng.

Bây giờ chúng tôi có một ứng cử viên khác cho nhắn tin liên quá trình. Sẽ khá dễ dàng để thiết lập một máy chủ giám sát cho ứng dụng nút của chúng tôi, lắng nghe trên ổ cắm UDP để cập nhật chương trình và số liệu thống kê được gửi từ các quy trình khác. Tốc độ giao thức đủ nhanh cho các hệ thống thời gian thực và bất kỳ sự cố mất gói hoặc trục trặc UDP nào khác sẽ không đáng kể được coi là phần trăm tổng khối lượng theo thời gian.

Lấy ý tưởng phát sóng xa hơn, chúng tôi cũng có thể sử dụng dgram để tạo một máy chủ đa hướng. A đa hướng chỉ đơn giản là phát một-nhiều máy chủ. Chúng tôi có thể phát sóng tới một loạt các IP đã được bảo lưu vĩnh viễn dưới dạng địa chỉ phát đa hướng:

Tiện ích mở rộng Máy chủ lưu trữ cho Đa hướng IP [RFC1112] chỉ định các tiện ích mở rộng cần thiết của việc triển khai Máy chủ lưu trữ Giao thức Internet (IP) để hỗ trợ đa hướng. Địa chỉ phát đa hướng nằm trong dải từ 224.0.0.0 đến 239.255.255.255.

- http://www.iana.org/assignments/multicast-addresses/ multicast-address.xhtml

Ngoài ra, phạm vi giữa 224.0.0.0 và 224.0.0.255 được dành riêng cho các giao thức định tuyến đặc biệt.

Ngoài ra, một số cổng nhất định được cấp phát để UDP (và TCP) sử dụng, bạn có thể tìm thấy danh sách trong số đó trong liên kết sau:

https://en.wikipedia.org/wiki/List\_of\_TCP\_and\_UDP\_port\_numbers

Kết quả của tất cả thông tin hấp dẫn này là biết rằng có một khối IP và cổng dành riêng cho UDP và / hoặc đa hướng và bây giờ chúng ta sẽ sử dụng một số trong số chúng để triển khai đa hướng qua UDP với Node.

## 8.14. Đa hướng UDP với Node

Sự khác biệt duy nhất giữa việc thiết lập máy chủ UDP đa hướng và máy chủ "tiêu chuẩn" là ràng buộc với một cổng UDP đặc biệt để gửi và cho biết rằng chúng tôi muốn lắng nghe tất cả bộ điều hợp mạng có sẵn. Quá trình khởi chạy máy chủ đa hướng của chúng tôi trông giống như đoạn mã sau:

var socket = dgram.createSocket ('udp4'); var multicastAddress

= '230.1.2.3';

var multicastPort = 5554;

socket.bind (multicastPort); socket.on ("đang nghe", function () { this.setMulticastTTL (64);

this.addMembership (multicastAddress);

});

Khi chúng tôi đã quyết định về một cổng phát đa hướng và một địa chỉ và đã ràng buộc, chúng tôi bắt nghe sự kiện và cấu hình máy chủ của chúng tôi. Lệnh quan trọng nhất là socket.addMembership, yêu cầu hạt nhân tham gia nhóm phát đa hướng tại đa địa chỉ. Các ổ cắm UDP khác hiện có thể đăng ký vào nhóm phát đa hướng tại địa chỉ này.

Datagram nhảy qua các mạng giống như bất kỳ gói mạng nào. Các setMulticastTTL phương thức được sử dụng để đặt số bước nhảy (Thời gian tồn tại) tối đa mà một gói dữ liệu được phép thực hiện trước khi nó bị hủy bỏ và không được phân phối. Phạm vi chấp nhận được là 0-255, với mặc định là một (1) trên hầu hết các hệ thống. Thông thường, đây không phải là một thiết lập mà người ta cần lo lắng, nhưng nó có sẵn khi các giới hạn chính xác có ý nghĩa, chẳng hạn như khi gói tin rẻ và bước nhảy đắt.

Nếu bạn cũng muốn cho phép nghe trên giao diện cục bộ, sử dụng socket.setBroadcast (true) và ổ cắm. setMulticastLoopback (true). Điều này thường không cần thiết.

Cuối cùng, chúng tôi sẽ sử dụng máy chủ này để truyền phát tin nhắn tới tất cả người nghe UDP trên đa địa chỉ. Bây giờ, hãy tạo hai ứng dụng khách sẽ lắng nghe đa tuyến:

dgram.createSocket ('udp4')

. on ('message', function (message, remote) { console.log ('Client1 đã nhận tin nhắn' + tin nhắn + 'từ' + remote.address + ':' + remote.port);

})

. ràng buộc (multicastPort, multicastAddress); dgram.createSocket ('udp4')

. on ('message', function (message, remote) {

console.log ('Client2 đã nhận tin nhắn' + tin nhắn + 'từ' + remote.address + ':' + remote.port);

})

. ràng buộc (multicastPort, multicastAddress);

Bây giờ chúng tôi có hai máy khách đang nghe cùng một cổng phát đa hướng. Tất cả những gì còn lại cần làm là đa hướng. Trong ví dụ này, chúng tôi sẽ sử dụng setTimeout để gửi một giá trị bộ đếm mỗi giây:

var cnt = 1; người gửi var;

(sender = function () { var msg = new Buffer ("Đây là thông báo #" + cnt); socket.send (

tin nhắn,

0, msg.length, multicastPort,

multicastAddress

);

+ + cnt;

setTimeout (người gửi, 1000);

}) ();

Các giá trị bộ đếm sẽ tạo ra một cái gì đó giống như sau:

Client2 đã nhận được tin nhắn Đây là tin nhắn # 1 từ 67.40.141.16:5554 Client1 đã nhận được tin nhắn Đây là tin nhắn số 1 từ 67.40.141.16:5554 Client2 đã nhận được tin nhắn Đây là tin nhắn số 2 từ 67.40.141.16:5554 Client1 đã nhận được tin nhắn Đây là tin nhắn số 2 từ 67.40.141.16:5554 Client2 đã nhận được tin nhắn Đây là tin nhắn số 3 từ 67.40.141.16:5554

. . .

Chúng tôi có hai khách hàng đang nghe các chương trình phát sóng từ một nhóm cụ thể. Hãy thêm một khách hàng khác, lắng nghe trên một nhóm khác — giả sử tại địa chỉ multicast 230.3.2.1:

dgram.createSocket ('udp4')

. on ('message', function (message, remote) { console.log ('Client3 đã nhận tin nhắn' + tin nhắn + 'từ' + remote.address + ':' + remote.port);

})

. ràng buộc (multicastPort, '230.3.2.1');

Bởi vì máy chủ của chúng tôi hiện đang truyền tin nhắn đến một địa chỉ khác, chúng tôi sẽ cần thay đổi cấu hình máy chủ của mình và thêm địa chỉ mới này với một địa chỉ khác addMembership gọi:

socket.on ("đang nghe", function () { this.addMembership (multicastAddress);

this.addMembership ('230.3.2.1');

});

Bây giờ chúng tôi có thể gửi đến cả hai địa chỉ:

(sender = function () { socket.send (

. . .

multicastAddress

);

socket.send (

. . .

'230.3.2.1'

);

// ... }) ();

Không có gì ngăn khách hàng truyền phát cho những người khác trong nhóm hoặc thậm chí các thành viên của nhóm khác:

dgram.createSocket ('udp4')

. on ('message', function (message, remote) { var msg = new Buffer ("Đang gọi nhóm ban đầu!"); // 230.1.2.3 là địa chỉ multicast

socket.send ( msg, 0, msg.length, multicastPort, '230.1.2.3');

})

. ràng buộc (multicastPort, '230.3.2.1');

Bất kỳ quá trình Node nào có địa chỉ trên giao diện mạng của chúng tôi giờ đây đều có thể lắng nghe trên một địa chỉ đa hướng UDP cho các tin nhắn, cung cấp một hệ thống liên lạc giữa các quá trình nhanh chóng và trang nhã.

## 8.15. Sử dụng Dịch vụ Web của Amazon trong ứng dụng của bạn

Khi một vài nghìn người dùng trở thành vài triệu người dùng, khi cơ sở dữ liệu mở rộng đến hàng terabyte dữ liệu, chi phí và độ phức tạp của việc duy trì một ứng dụng bắt đầu áp đảo các nhóm không đủ kinh nghiệm, kinh phí và / hoặc thời gian. Khi đối mặt với tốc độ phát triển nhanh chóng, đôi khi rất hữu ích khi giao trách nhiệm về một hoặc nhiều khía cạnh trong ứng dụng của bạn cho các nhà cung cấp dịch vụ dựa trên đám mây. AWS (Dịch vụ web của Amazon) chỉ là một bộ các dịch vụ điện toán đám mây, được cung cấp bởi amazon.com.

Bạn sẽ cần tài khoản AWS để sử dụng các ví dụ này. Tất cả các dịch vụ mà chúng tôi sẽ khám phá đều miễn phí hoặc gần như miễn phí cho các mục đích sử dụng phát triển khối lượng thấp. Để tạo tài khoản trên AWS, hãy truy cập liên kết sau: http://aws.amazon.com/

Sau khi tạo tài khoản, bạn sẽ có thể quản lý tất cả các dịch vụ của mình thông qua bảng điều khiển AWS tại: https://console.aws.amazon.com/console/home

Trong phần này, chúng ta sẽ tìm hiểu cách sử dụng ba dịch vụ AWS phổ biến:

Để lưu trữ tài liệu và tệp, chúng tôi sẽ kết nối với Amazon S3 (Dịch vụ lưu trữ đơn giản)

Cơ sở dữ liệu Khóa / Giá trị của Amazon, DynamoDB

Để quản lý một lượng lớn e-mail, chúng tôi sẽ tận dụng SES (Dịch vụ Email Đơn giản) Để truy cập các dịch vụ này, chúng tôi sẽ sử dụng AWS SDK cho Node, có thể tìm thấy tại liên kết sau:

https://github.com/aws/aws-sdk-js Để cài đặt mô-đun, hãy chạy lệnh sau:

npm cài đặt aws-sdk

Tài liệu đầy đủ cho aws-sdk mô-đun có thể được tìm thấy tại liên kết sau: http://docs.aws.amazon.com/AWSJavaScriptSDK/latest/ frame.html

Xác thực Các nhà phát triển đã đăng ký AWS được chỉ định hai số nhận dạng:

Một công chúng ID khóa truy cập ( một chuỗi gồm 20 ký tự chữ và số).

A Khóa truy cập bí mật ( một chuỗi 40 ký tự). Điều rất quan trọng là giữ cho Khóa bí mật của bạn ở chế độ riêng tư.

Amazon cũng cung cấp cho các nhà phát triển khả năng xác định khu vực để giao tiếp, chẳng hạn như " us-East-1 ”. Điều này cho phép các nhà phát triển nhắm mục tiêu các máy chủ gần nhất (điểm cuối khu vực) cho các yêu cầu của họ.

Điểm cuối khu vực và cả hai khóa xác thực đều cần thiết để thực hiện yêu cầu.

Để biết bảng phân tích các điểm cuối khu vực, hãy truy cập liên kết sau: http://docs.aws.amazon.com/general/latest/gr/rande. html

Vì chúng ta sẽ sử dụng cùng một thông tin đăng nhập trong mỗi ví dụ sau, hãy tạo một tệp config.json duy nhất được sử dụng lại:

{

|  |  |
| --- | --- |
| "accessKeyId"  "secretAccessKey"  "khu vực"  "apiVersions"  "s3"  "ses" | : "your-key",  : "Bí mật của bạn",  : "us-East-1",  : {  : "2006-03-01",  : "2010-12-01", |

"dynamicodb" : "2012-08-10"

}

}

Chúng tôi cũng định cấu hình các phiên bản API cụ thể mà chúng tôi sẽ sử dụng cho các dịch vụ. Nếu API dịch vụ của Amazon thay đổi, điều này sẽ đảm bảo mã của chúng tôi sẽ tiếp tục hoạt động.

Một phiên AWS hiện có thể được khởi tạo chỉ với hai dòng mã. Giả sử rằng hai dòng này tồn tại trước bất kỳ mã ví dụ nào sau đây:

var AWS = request ('aws-sdk');

AWS.config.loadFromPath ('./ config.json');

## 8.16. Lỗi

Khi thử nghiệm với các dịch vụ này, có khả năng các mã lỗi sẽ xuất hiện đôi khi. Do sự phức tạp của chúng và bản chất của điện toán đám mây, các dịch vụ này đôi khi có thể phát ra các lỗi đáng ngạc nhiên hoặc không mong muốn. Ví dụ, vì S3 chỉ có thể hứa hẹn nhất quán cuối cùng trong một số khu vực và tình huống, việc cố gắng đọc một khóa vừa được ghi vào có thể không phải lúc nào cũng thành công. Chúng tôi sẽ khám phá danh sách đầy đủ các mã lỗi cho từng dịch vụ và bạn có thể tìm thấy chúng tại các vị trí sau:

• S3

http://docs.aws.amazon.com/AmazonS3/latest/API/ErrorResponses. html

• DynamoDB

http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/ Welcome.html

• SES

http://docs.aws.amazon.com/ses/latest/DeveloperGuide/api-errorcodes.html

và

http://docs.aws.amazon.com/ses/latest/DeveloperGuide/smtpresponse-codes.html

Vì ban đầu sẽ khó dự đoán nơi có thể phát sinh lỗi, điều quan trọng là phải sử dụng miền mô-đun hoặc mã kiểm tra lỗi khác khi bạn tiếp tục.

Ngoài ra, một khía cạnh tinh tế nhưng cơ bản của mô hình bảo mật và nhất quán của Amazon là sự đồng bộ hóa nghiêm ngặt của thời gian và thời gian máy chủ web của nó như được hiểu bởi một máy chủ đưa ra yêu cầu. Chênh lệch tối đa là 15 phút. Mặc dù điều này có vẻ như là một thời gian dài, nhưng trên thực tế, thời gian trôi là rất phổ biến. Khi phát triển, hãy coi chừng 403: Các lỗi bị cấm tương tự như một trong các lỗi sau:

SignatureDoesNotMatch: Lỗi này có nghĩa là chữ ký đã hết hạn

RequestTimeTooSkewed: Chênh lệch giữa thời gian yêu cầu và thời gian hiện tại quá lớn

Nếu những lỗi như vậy gặp phải, thời gian nội bộ của máy chủ đưa ra yêu cầu có thể đã trôi qua. Nếu vậy, thời gian của máy chủ đó sẽ cần được đồng bộ hóa. Trên Unix, người ta có thể sử dụng NTP (Giao thức thời gian mạng) để đạt được tính đồng bộ. Một giải pháp là sử dụng các lệnh sau:

rdate 129.6.15.28 ntpdate 129.6.15.28

Để biết thêm thông tin về NTP và đồng bộ hóa thời gian, hãy truy cập liên kết sau:

http://www.pool.ntp.org/en/

Hãy bắt đầu sử dụng các dịch vụ AWS, bắt đầu với dịch vụ tệp phân tán, S3.

## 8.17. Sử dụng S3 để lưu trữ tệp

S3 có thể được sử dụng để lưu trữ bất kỳ tệp nào mà người ta mong đợi có thể lưu trữ trên hệ thống tệp. Thông thường nhất, nó được sử dụng để lưu trữ các tệp phương tiện như hình ảnh và video. S3 cũng là một hệ thống lưu trữ tài liệu tuyệt vời, đặc biệt rất phù hợp để lưu trữ các đối tượng JSON nhỏ hoặc các đối tượng dữ liệu tương tự.

Ngoài ra, các đối tượng S3 có thể truy cập thông qua HTTP, điều này làm cho việc truy xuất rất tự nhiên và các phương thức REST như PUT / DELETE / UPDATE được hỗ trợ. S3 hoạt động rất giống như người ta mong đợi một máy chủ tệp điển hình hoạt động, được trải rộng trên các máy chủ trên toàn cầu và cung cấp dung lượng lưu trữ vô hạn cho tất cả các mục đích thực tế.

S3 sử dụng khái niệm về một Gầu múc như một loại hệ quả đối với "ổ cứng". Mỗi tài khoản S3 có thể chứa 100 nhóm (đây là giới hạn cứng), không có giới hạn về số lượng tệp chứa trong mỗi nhóm.

## 8.18. Làm việc với xô

Tạo một thùng rất dễ dàng:

var S3 = new AWS.S3 ();

S3.createBucket ({

Nhóm: 'nodejs-book'

}, function (err, data) { if (err) {throw new Error ("Không thể tạo nhóm."); } console.log (dữ liệu);

});

Chúng tôi sẽ nhận được một bản đồ dữ liệu có chứa nhóm Vị trí, và một Yêu cầu ID:

{ Vị trí: '/ nodejs-book',

RequestId: 'A0F2B68C72BE8A29'}

Có khả năng là nhiều hoạt động khác nhau sẽ được thực hiện đối với một cái xô. Như một sự tiện lợi aws-sdk cho phép tự động xác định tên nhóm trong danh sách tham số cho tất cả các hoạt động tiếp theo:

var S3 = new AWS.S3 ({ params: {Bucket: 'nodejs-book'}

});

S3.createBucket (function (err, data) {// ...});

Sử dụng listBuckets để tìm nạp một loạt các nhóm hiện có:

S3.listBuckets (function (err, data) { console.log (data.Buckets);

});

// [{Tên: 'nodejs-book',

// Ngày tạo: Thứ Hai, ngày 15 tháng 7 năm 2013 22:17:08 GMT-0500 (CDT)},

// . . .

// ]

Tên nhóm là toàn cầu cho tất cả người dùng S3. Không người dùng nào của S3 có thể sử dụng tên nhóm mà người dùng khác đã xác nhận. Nếu tôi có một cái thùng tên foo không người dùng S3 nào khác có thể sử dụng tên nhóm đó. Đây là một món gotcha mà nhiều người bỏ lỡ.

## 8.19. Làm việc với các đối tượng

Hãy thêm một tài liệu vào nodejs-book xô trên S3:

var S3 = new AWS.S3 ({

params: {Bucket: 'nodejs-book' }

});

var body = JSON.stringify ({foo: "bar"}); var s3Obj = {

Khóa: 'demo / putObject / first.json',

Body: cơ thể,

ServerSideEncryption: "AES256", ContentType: "application / json",

ContentLength: body.length,

ACL: "riêng tư"

}

S3.putObject (s3Obj, function (err, data) { if (err) {ném Lỗi mới ("Lỗi PUT"); } console.log (dữ liệu);

});

Nếu PUT thành công, lệnh gọi lại của nó sẽ nhận được một đối tượng tương tự như sau:

{ETag: '"9bb58f26192e4ba00f01e2e7b136bbd8", ServerSideEncryption: 'AES256',

RequestId: 'C7AAD7FF0A14F979'}

Bạn được khuyến khích tham khảo tài liệu SDK và thử nghiệm với tất cả các thông số putObject chấp nhận. Ở đây chúng tôi tập trung vào hai trường bắt buộc duy nhất và một vài trường hữu ích và phổ biến:

Khóa: Tên để xác định duy nhất tệp của bạn trong nhóm này.

Nội dung: Bộ đệm, Chuỗi hoặc Luồng bao gồm phần thân tệp.

ServerSideEncryption: Có mã hóa tệp trong S3 hay không. Tùy chọn duy nhất hiện tại là AES256 (đó là một lựa chọn tốt!).

ContentType: Loại MIME chuẩn.

ContentLength: Một chuỗi chỉ ra IP đích cho datagram.

ACL: Quyền truy cập soạn trước, chẳng hạn như riêng tư hoặc là public-read-write.

Tham khảo tài liệu S3.

Đó là một ý tưởng tốt để có đối tượng Chìa khóa giống như một đường dẫn hệ thống tệp, giúp sắp xếp và truy xuất sau này. Trên thực tế, bảng điều khiển S3 của Amazon phản ánh mô hình này trong giao diện người dùng của nó:



Hãy phát trực tuyến hình ảnh lên đến S3:

var fs = request ('fs');

fs.stat ("./ exploremage.jpg", function (err, stat) { var s3Obj = {

Khóa: 'demo / putObject / retmage.jpg',

Nội dung: fs.createReadStream ("./ exploremage.jpg"),

ContentLength: stat.size,

ContentType: "image / jpeg",

ACL: "đọc công khai"

}

S3.putObject (s3Obj, function (err, data) {// ...}); })

Như chúng tôi đã đưa ra hình ảnh này đọc công khai quyền, nó sẽ có thể truy cập được tại URL sau:

https://s3.amazonaws.com/nodejs-book/demos/putObject/testimage.jpg

Tìm nạp một đối tượng từ S3 và truyền trực tuyến nó vào hệ thống tệp cục bộ thậm chí còn dễ dàng hơn:

var outFile = fs.createWriteStream ('./ fetchedfile.jpg'); S3.getObject ({

Chìa khóa : 'demo / putObject / retmage.jpg'

}). createReadStream (). pipe (outFile);

Ngoài ra, chúng ta có thể bắt các sự kiện dữ liệu trên quá trình truyền phân đoạn HTTP:

S3.getObject ({

Chìa khóa : 'demo / putObject / testfile.jpg'

})

. on ('httpData', function (chunk) { outFile.write (chunk); })

. on ('httpDone', function () {outFile.end ();})

. gửi ();

Để xóa một đối tượng:

S3.deleteObject ({

Nhóm: 'nodejs-book',

Chìa khóa : 'demo / putObject / Optimism.jpg'

}, function (err, data) {// . . . });

Để xóa nhiều đối tượng, hãy chuyển một Mảng (đến tối đa 1000 đối tượng):

S3.deleteObjects ({

Gầu múc : 'nodejs-book',

Xóa bỏ : {

Các đối tượng : [{

Chìa khóa : 'demo / putObject / first.json'

}, {

Chìa khóa : 'demo / putObject / retmage2.jpg'

}]

}

}, function (err, data) {// . . . });

## 8.20. Sử dụng AWS với máy chủ Node

Tổng hợp những gì chúng ta biết về máy chủ Node, truyền dữ liệu tệp qua đường dẫn và HTTP, nên rõ ràng cách gắn S3 làm hệ thống tệp chỉ trong một vài dòng mã:

http.createServer (chức năng (yêu cầu, phản hồi) { var requestFile = request.url.substring (1); S3.headObject ({

Key: requestFile

}, function (err, data) {

// 404, v.v.

nếu (sai) { response.writeHead (err.statusCode);

trả về response.end (err.name);

}

response.writeHead (200, {

"Sửa đổi lần cuối" : data.LastModified,

"Thời lượng nội dung" : data.ContentLength,

"Loại-Nội dung" : data.ContentType,

"ETag" : data.ETag

});

S3.getObject ({

Chìa khóa : requestFile

}). createReadStream (). pipe (response);

});

}). nghe (8080);

Máy chủ HTTP Node tiêu chuẩn nhận được một URL yêu cầu. Trước tiên, chúng tôi thử vận hành HEAD bằng cách sử dụng aws-sdk phương pháp headObject, hoàn thành hai điều:

Chúng tôi sẽ xác định xem tệp có sẵn không

Chúng tôi sẽ có thông tin tiêu đề cần thiết để xây dựng phản hồi

Sau khi xử lý bất kỳ lỗi mã trạng thái nào không phải 200, chúng tôi chỉ cần đặt tiêu đề phản hồi của mình và truyền tệp trở lại người yêu cầu, như đã trình bày trước đó.

Một hệ thống như vậy cũng có thể hoạt động như một không an toàn, theo cả hai hướng; nếu S3, hoặc tệp, không khả dụng, chúng tôi có thể liên kết với một hệ thống tệp khác, phát trực tuyến từ đó. Ngược lại, nếu hệ thống tệp cục bộ ưa thích của chúng tôi bị lỗi, chúng tôi có thể chuyển sang hệ thống tệp S3 dự phòng của mình.

Xem amazon / s3-redirect.js tệp trong gói mã có sẵn tại trang web Packt để biết ví dụ về việc sử dụng chuyển hướng 302 để gắn kết hệ thống tệp AWS một cách tương tự.

S3 là một hệ thống lưu trữ dữ liệu mạnh mẽ với nhiều tính năng tiên tiến hơn những tính năng mà chúng tôi đã đề cập, chẳng hạn như lập phiên bản đối tượng, quản lý thanh toán tải xuống và thiết lập các đối tượng dưới dạng tệp torrent. Với sự hỗ trợ của nó cho các luồng, aws-sdk mô-đun giúp các nhà phát triển Node dễ dàng làm việc với S3 như thể nó là một hệ thống tệp cục bộ.

## 8.21. Lấy và thiết lập dữ liệu với DynamoDB

DynamoDB (DDB) là một cơ sở dữ liệu NoSQL cung cấp thông lượng rất cao và khả năng dự đoán có thể dễ dàng mở rộng. DDB được thiết kế cho thâm dụng dữ liệu

ứng dụng, thực hiện bản đồ lớn / thu nhỏ và các truy vấn phân tích khác với độ trễ thấp và đáng tin cậy. Nói như vậy, nó cũng là một giải pháp cơ sở dữ liệu tuyệt vời cho các ứng dụng web nói chung.

Sách trắng thông báo DynamoDB có ảnh hưởng lớn, gây ra sự quan tâm thực sự đến cơ sở dữ liệu NoSQL và truyền cảm hứng cho nhiều người, bao gồm cả Apache Cassandra. Bài báo đề cập đến các khái niệm nâng cao, nhưng thưởng cho nghiên cứu cẩn thận:

http://www.allthingsdistributed.com/files/amazonDyo-sosp2007.pdf

Cơ sở dữ liệu Dynamo là một tập hợp các bảng, là một tập hợp các mục, là một tập hợp các thuộc tính. Mỗi mục trong một bảng (hoặc hàng, nếu bạn muốn) phải có một khóa chính, hoạt động như một chỉ mục cho bảng. Mỗi mục có thể có bất kỳ số lượng thuộc tính nào (lên đến giới hạn 65 KB) ngoài khóa chính.

Đây là một vật phẩm, có năm thuộc tính, một thuộc tính đóng vai trò là khóa chính ( Tôi):

{ Id = 123

Ngày = "1375314738466"

UserId = "DD9DDD8892"

Giỏ hàng = ["song1", "song2"] Hành động =

"mua"}

Hãy tạo một bảng có cả khóa chính và khóa phụ:

var AWS = request ('aws-sdk');

AWS.config.loadFromPath ('../ config.json'); var db = new AWS.DynamoDB (); db.createTable ({

TableName: 'mua hàng',

AttributeDefinitions: [{

AttributeName: "Id", AttributeType: "N"

}, {

Tên thuộc tính: "Ngày", Loại thuộc tính: "N"

}],

KeySchema: [{

Tên thuộc tính: 'Id', KeyType: 'HASH'

}, {

AttributeName: 'Date', KeyType: 'RANGE'

}],

Thông lượng được cấp phép: { ReadCapacityUnits: 2,

WriteCapacityUnits: 2

}

}, function (err, data) {console.log (using.inspect (data)); });

Lệnh gọi lại sẽ nhận được một đối tượng tương tự như sau:

{ TableDescription: {

AttributeDefinitions: [// giống với những gì đã được gửi], CreationDateTime: 1375315748.029,

ItemCount: 0,

KeySchema: [// giống với những gì đã được gửi],

Thông lượng được cấp phép: { NumberOfDecreasesToday: 0,

ReadCapacityUnits: 2,

WriteCapacityUnits: 2

},

TableName: 'mua hàng',

TableSizeBytes: 0,

TableStatus: 'CREATING'

}

}

Việc tạo / xóa bảng không diễn ra ngay lập tức — về cơ bản bạn đang xếp hàng để tạo bảng (lưu ý TableStatus). Tại một thời điểm nào đó trong tương lai (gần), bảng sẽ tồn tại. Vì không thể thay đổi định nghĩa bảng DDB mà không xóa bảng và xây dựng lại nó, trên thực tế, độ trễ này không phải là điều gì đó sẽ ảnh hưởng đến ứng dụng của bạn — hãy xây dựng một lần, sau đó làm việc với các mục.

Các bảng DDB phải được cung cấp một lược đồ chỉ ra các thuộc tính mặt hàng sẽ hoạt động như các khóa, được xác định bởi KeySchema. Mỗi thuộc tính trong KeySchema có thể là một PHẠM VI

hoặc một HASH. Phải có một chỉ số như vậy; có thể có nhiều nhất hai. Mỗi mục được thêm vào phải chứa bất kỳ khóa xác định nào, với nhiều thuộc tính bổ sung như mong muốn.

Mỗi mặt hàng trong KeySchema phải được so khớp với số lượng các mục trong AttributeDefinitions. Trong AttributeDefinitions mỗi thuộc tính có thể là một số (" N ") hoặc một chuỗi (" S"). Khi thêm hoặc sửa đổi các thuộc tính, luôn cần xác định các thuộc tính theo kiểu của nó cũng như theo tên.

Để thêm một mục:

db.putItem ({

TableName: "mua hàng",

Mục : {

Id: {"N": "123"},

Ngày: {"N": "1375314738466"}, UserId: {"S":

"DD9DDD8892"},

Giỏ hàng: {"SS": ["song1", "song2"]}, Hành động: {"S": "mua"}

}

}, chức năng() { // ... });

Ngoài các khóa chính và khóa phụ (tùy chọn), chúng tôi muốn thêm các thuộc tính khác vào mặt hàng của mình. Mỗi loại phải được cung cấp một trong các loại sau:

S: A Chuỗi

N: Một số

B: Một chuỗi được mã hóa Base64

SS: Một mảng chuỗi (Tập hợp chuỗi)

NS: Một mảng số (Bộ số)

BS: Một mảng các chuỗi được mã hóa Base64 (bộ Base64)

Tất cả các mục sẽ cần có cùng số cột này — một lần nữa, các lược đồ động là không phải một tính năng của DDB.

Giả sử rằng chúng ta đã tạo một bảng giống như bảng sau:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tôi | Ngày | Hoạt động | xe đẩy | Tên người dùng |
| 123  124  125 | 1375314738466  1375314738467  1375314738468 | mua mua mua | {"song1", "song2"} {"song2", "song4"}  {"song12", "song6"} | DD9DDD8892  DD9EDD8892  DD9EDD8890 |

Bây giờ chúng ta hãy thực hiện một số thao tác tìm kiếm.

## 8.22. Tìm kiếm cơ sở dữ liệu

Có hai loại hoạt động tìm kiếm có sẵn: truy vấn và quét. Việc quét trên bảng với một khóa chính, không có ngoại lệ, sẽ tìm kiếm mọi mục trong bảng, trả về những mục phù hợp với tiêu chí tìm kiếm của bạn. Điều này có thể rất chậm trên bất kỳ thứ gì ngoại trừ cơ sở dữ liệu nhỏ. Truy vấn là một tra cứu khóa trực tiếp. Chúng tôi sẽ xem xét các truy vấn trước. Lưu ý rằng trong ví dụ này, chúng ta sẽ giả định rằng bảng này chỉ có một khóa chính.

Để tìm nạp Hoạt động và xe đẩy thuộc tính cho mặt hàng 124, chúng tôi sử dụng mã sau:

db.getItem ({

TableName: "mua hàng",

Chìa khóa : {

Id: {"N" : "124"}

},

AttributesToGet: ["Action", "Cart"]}, function (err, res) {

console.log (use.inspect (res, {depth: 10}));

});

Cái nào sẽ trở lại:

{ Mục: {

Hành động: {S: 'buy'},

Giỏ hàng: {SS: ['song2', 'song4']}

}

}

Để chọn tất cả các thuộc tính, chỉ cần bỏ qua AttributesToGet Định nghĩa.

Quét đắt hơn, nhưng cho phép nhiều tìm kiếm liên quan hơn. Tính hữu ích của các phím phụ đặc biệt rõ rệt khi thực hiện quét, cho phép chúng tôi tránh phải quét toàn bộ bảng. Trong ví dụ quét đầu tiên của chúng tôi, chúng tôi sẽ làm việc như thể chỉ có một khóa chính. Sau đó, chúng tôi sẽ hướng dẫn cách lọc bản quét bằng phím phụ.

Để lấy tất cả các bản ghi có xe đẩy thuộc tính chứa song2, sử dụng mã sau:

db.scan ({

TableName: "mua hàng",

ScanFilter: {

"Xe đẩy": {

"AttributeValueList": [{

"S": "song2"

}],

"ComparisonOperator": "CONTAINS"

}, }

}, function (err, res) { console.log (using.inspect (res, { độ sâu: 10

})); });

Điều này sẽ trả về tất cả các giá trị thuộc tính cho các mặt hàng có Tôi 123 và 124. Bây giờ hãy sử dụng khóa phụ của chúng tôi để lọc thêm điều này:

db.scan ({

TableName: "mua hàng",

ScanFilter: {

"Ngày": {

"AttributeValueList": [{

"N": "1375314738467"

}],

"ComparisonOperator": "EQ"

},

"Xe đẩy": {

"AttributeValueList": [{

"S": "song2"

}],

"ComparisonOperator": "CONTAINS"

},

}

}, function (err, res) { console.log (use.inspect (res, {depth: 10}));

});

Bộ lọc mới này giới hạn kết quả ở mục 124.

## 8.23. Gửi thư qua SES

Amazon mô tả các vấn đề mà SES được thiết kế để giải quyết theo cách này:

Việc xây dựng các giải pháp email quy mô lớn để gửi các thông điệp tiếp thị và giao dịch thường là một thách thức phức tạp và tốn kém cho các doanh nghiệp. Để tối ưu hóa tỷ lệ email được gửi thành công, các doanh nghiệp phải giải quyết những vấn đề phức tạp như như quản lý máy chủ email, cấu hình mạng và đáp ứng các tiêu chuẩn khắt khe của Nhà cung cấp dịch vụ Internet (ISP) cho nội dung email.

Ngoài các vấn đề về quy mô mạng điển hình vốn có trong việc phát triển bất kỳ hệ thống nào, việc cung cấp dịch vụ e-mail đặc biệt khó khăn do sự phổ biến của thư rác. Rất khó để gửi một số lượng lớn các e-mail không mong muốn mà không bị đưa vào danh sách đen, ngay cả khi người nhận có thể nhận được chúng. Hệ thống kiểm soát thư rác được tự động hóa; dịch vụ của bạn phải được liệt kê trong "danh sách trắng", được các nhà cung cấp e-mail và trình theo dõi thư rác khác nhau sử dụng để tránh tỷ lệ phần trăm thấp e-mail của bạn kết thúc ở đâu đó không phải hộp thư đến của khách hàng. Dịch vụ Amail phải có danh tiếng tốt với đúng người hoặc nó gần như trở nên vô dụng.

Dịch vụ SES của Amazon có danh tiếng cần thiết, cung cấp cho các nhà phát triển ứng dụng dịch vụ e-mail dựa trên đám mây đáng tin cậy và có thể xử lý khối lượng e-mail gần như vô hạn. Trong phần này, chúng ta sẽ tìm hiểu cách sử dụng SES bởi một ứng dụng Node như một dịch vụ gửi thư đáng tin cậy.

Đảm bảo rằng bạn có quyền truy cập SES bằng cách truy cập bảng điều khiển dành cho nhà phát triển của bạn. Khi bạn lần đầu tiên đăng ký với SES, bạn sẽ được Hộp cát truy cập. Khi ở chế độ này, bạn chỉ được sử dụng trình mô phỏng hộp thư của Amazon hoặc gửi e-mail đến địa chỉ mà bạn đã xác minh (chẳng hạn như địa chỉ của riêng mình). Bạn có thể yêu cầu quyền truy cập sản xuất, nhưng vì mục đích của chúng tôi, bạn sẽ chỉ cần xác minh một địa chỉ e-mail để kiểm tra.

Bởi vì chi phí sử dụng một dịch vụ như SES sẽ tăng khi khối lượng thư của bạn tăng lên, bạn có thể muốn kiểm tra định kỳ hạn ngạch của mình:

var ses = new AWS.SES (); ses.getSendQuota (function (err, data) { console.log (err, dữ liệu);

});

Gửi tin nhắn:

ses.sendEmail ({

Nguồn: " spaquali@gmail.com ",

Nơi Đến : {

Địa chỉ đăng ký: [" spaquali@gmail.com "]

},

Thông điệp : {

Chủ đề: {Dữ liệu: "NodeJS và AWS SES"}, Nội dung: {

Văn bản: {Dữ liệu: "Nó đã hoạt động!" }

}

}

}, function (err, resp) { console.log (tương ứng);

});

Lệnh gọi lại sẽ nhận được một cái gì đó giống như đầu ra sau:

RequestId: '623144c0-fa5b-11e2-8e49-f73ce5ee2612'

MessageId: '0000014037f1a167-587a626e-ca1f-4440-a4b0-81756301bc28-000000'

Nhiều người nhận, nội dung cơ thể HTML và tất cả các tính năng khác mà người ta mong đợi từ một dịch vụ thư đều có sẵn.

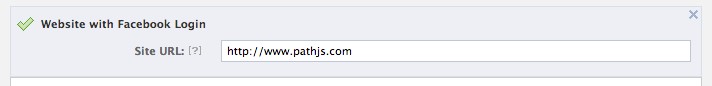
## 8.24. Xác thực với Facebook Connect

Việc xây dựng một hệ thống xác thực có thể mở rộng rất khó. Mặc dù có vẻ dễ dàng — tạo một bảng hoặc danh sách người dùng và thông tin đăng nhập của họ, thay đổi danh sách đó bằng một biểu mẫu web đơn giản và bạn đã hoàn tất!

Nếu mục tiêu của bạn là tạo ra một hệ thống có thể mở rộng có thể phát triển khi cơ sở người dùng của bạn phát triển thì bạn sẽ cần phải làm nhiều hơn nữa. Sẽ cần thiết phải tạo hệ thống e-mail và cơ sở dữ liệu có thể mở rộng, quản lý phiên và nhiều hệ thống con khác cần thiết để đáp ứng kỳ vọng của người dùng về một đường dẫn có độ trễ thấp từ đăng ký đến đăng nhập đến nội dung mong muốn.

Hãy cùng nhau tổng hợp những gì chúng ta đã học được với AWS với Connect, dịch vụ xác thực của Facebook. Bạn có thể xem mã đầy đủ cho ví dụ này trong / Facebook thư mục của gói mã của bạn có sẵn tại trang web Packt.

Để bắt đầu, hãy truy cập Develop.facebook.com, thiết lập tài khoản nhà phát triển, sau đó tạo ứng dụng. Hãy nhớ đặt đúng đường dẫn ứng dụng của bạn như được hiển thị trong ảnh chụp màn hình sau:



Sau khi đăng ký, hãy tìm và sao chép hai khóa ứng dụng của bạn. Bạn sẽ cần những thứ này sau này.

Để tạo một máy chủ Node cho phép người dùng đăng nhập thông qua Connect, chúng tôi sẽ sử dụng hộ chiếu mô-đun npm, giúp dễ dàng kết nối với nhiều dịch vụ đám mây, bao gồm cả Facebook. Ngoài ra, chúng tôi sẽ sử dụng S3 để lưu trữ một đối tượng người dùng đơn giản, cho phép chúng tôi ánh xạ các ID đăng nhập Connect vào hồ sơ ứng dụng nội bộ của chúng tôi. Chúng tôi sẽ xây dựng tất cả điều này bằng cách sử dụng Express, hoạt động tốt với hộ chiếu.

Khởi tạo S3 theo các ví dụ trước của chúng tôi và tạo một thùng để lưu trữ hồ sơ người dùng. hộ chiếu hoạt động với phép ẩn dụ của một "chiến lược". Khi chúng tôi đã thiết lập thông tin đăng nhập của mình bằng khóa đăng nhập Facebook, chúng tôi cung cấp một lệnh gọi lại để nắm bắt kết quả của nỗ lực đăng nhập của người dùng — Facebook sẽ chuyển hướng đến URL này sau khi người dùng đã cố gắng xác thực.

Nếu đăng nhập thành công, hãy ánh xạ bản ghi của người dùng này tới cửa hàng S3 của chúng tôi, đặt khóa đối tượng S3 thành ID Facebook của người dùng. Giờ đây, chúng tôi có một chiến lược để chấp nhận đăng nhập bằng Connect:

var FacebookStrategy = request ('passport-facebook'). Strategy; //

Đối với bản demo, chúng tôi sử dụng một chiến lược phiên đơn giản. // Bạn có thể sẽ xử lý các phiên bằng cách sử dụng DB. passport.serializeUser (function (user, done) { done (null, user);

});

passport.deserializeUser (function (obj, done) { xong (null, obj);

});

passport.use (FacebookStrategy mới ({

clientID : 'ID của bạn', clientSecret : 'Bí mật của bạn',

gọi lại URL : "http://www.pathjs.com/auth/facebook/callback"

},

function (accessToken, refreshToken, profile, done) { profile.local = { lastLogin: new Date (). getTime (), sessionId: parseInt (Math.random () \* 100000)

}

var s3Obj = {

Khóa: profile.id,

Nội dung: JSON.stringify (hồ sơ),

ServerSideEncryption : "AES256",

ContentType: "application / json",

ACL: "riêng tư"

}

S3.putObject (s3Obj, function (err, data) { if (err) {hoàn thành xong (err); } hoàn thành xong

(err, hồ sơ);

}); }));

Khi người dùng truy cập trang web của chúng tôi, giờ đây chúng tôi có thể giới thiệu họ bằng thông tin đăng nhập Facebook.

Tất cả những gì còn lại cần làm là tạo bày tỏ máy chủ với các đường dẫn cần thiết để xác thực và đăng xuất, bao gồm cả đường dẫn để xử lý lệnh gọi lại của Facebook:

var app = express (); app.configure (function () { app.set ('lượt xem', \_\_dirname + '/ lượt xem'); app.set ('view engine', 'jade'); // thiết lập khác

});

app.get ('/', function (req, res) { res.render ('chỉ mục', { người dùng: req.user

});

});

app.get ('/ auth / facebook', passport.authenticate ('facebook')); app.get (

'/ auth / facebook / callback', passport.authenticate ('facebook', { failRedirect: '/'

}),

function (req, res) { res.redirect ('/');

}

);

app.get ('/ logout', function (req, res) { req.logout (); res.redirect ('/');

});

app.listen (8080);

Một bước cuối cùng là sử dụng Ngọc bích công cụ tạo khuôn mẫu để tạo các trang đăng nhập / đăng xuất (xem phần bày tỏ khởi tạo). Chìa khóa quan trọng là chuyển trách nhiệm hiển thị ứng dụng của chúng tôi mục lục , nằm ở gốc máy chủ, tới một mẫu Ngọc:

app.get ('/', function (req, res) { res.render ('chỉ mục', { người dùng: req.user

});

});

Mẫu này có thể được tìm thấy trong index.jade tập tin nằm ở facebook / lượt xem / trong gói mã của bạn.

Đó là nó! Ứng dụng Node của bạn, vốn đã nhanh, nay đã giao trách nhiệm xác thực và lưu trữ hồ sơ người dùng cho các chuyên gia trong việc mở rộng (và bảo mật) các hệ thống như vậy. Ứng dụng của bạn hiện có thể xử lý hàng nghìn kết nối đồng thời và mở rộng gần như vô thời hạn bằng cách tận dụng hai trong số các cơ sở hạ tầng ứng dụng lớn nhất và ổn định nhất trên thế giới — một cơ sở hạ tầng miễn phí và cơ sở hạ tầng còn lại với chi phí rất thấp.

## Tóm lược

Các ứng dụng dữ liệu lớn đã đặt ra trách nhiệm đáng kể cho các nhà phát triển ứng dụng mạng để chuẩn bị cho quy mô. Node đã cung cấp sự trợ giúp trong việc tạo ra một môi trường phát triển ứng dụng thân thiện với mạng có thể dễ dàng kết nối với các thiết bị khác trên mạng, chẳng hạn như các dịch vụ đám mây và đặc biệt là các máy chủ Node khác.

Trong chương này, chúng ta đã học một số chiến lược tốt để mở rộng quy mô máy chủ Node, từ phân tích việc sử dụng CPU đến giao tiếp giữa các quy trình. Với kiến thức mới của chúng tôi về hàng đợi tin nhắn và UDP, chúng tôi có thể xây dựng mạng lưới máy chủ Node mở rộng theo chiều ngang, cho phép chúng tôi xử lý ngày càng nhiều lưu lượng truy cập bằng cách sao chép các nút hiện có. Sau khi nghiên cứu cân bằng tải và ủy quyền với Node, chúng tôi có thể tự tin thêm dung lượng cho các ứng dụng của mình. Khi kết hợp với các dịch vụ đám mây do AWS và Facebook cung cấp, chúng tôi có thể mở rộng quy mô kho dữ liệu và quản lý tài khoản người dùng một cách dễ dàng với chi phí thấp.

Khi các ứng dụng của chúng tôi phát triển, chúng tôi sẽ cần phải duy trì nhận thức liên tục về cách từng bộ phận, cũng như toàn bộ, đang hoạt động như thế nào. Trước khi một thành phần mới được thêm vào, sự tích hợp đó sẽ cần được kiểm tra.

Trong chương tiếp theo, chúng ta sẽ tìm hiểu các chiến lược để kiểm tra hệ thống của bạn, cả ở quy mô lớn và quy mô nhỏ.

# CHƯƠNG 9

# Kiểm thử

Khi địa hình không thống nhất với bản đồ, hãy tin tưởng vào địa hình.

- Sách hướng dẫn quân đội Thụy Sĩ

Bởi vì một cộng đồng hoàn toàn cam kết chia sẻ mã đang xây dựng Node và khả năng tương tác giữa các mô-đun là rất quan trọng, nên không có gì ngạc nhiên khi các công cụ và khuôn khổ kiểm tra mã đã gia nhập hệ sinh thái của Node ngay sau khi thành lập. Thật vậy, nhóm Node lõi phân tích cú pháp thông thường đã thêm khẳng định mô-đun sớm, đề xuất một sự công nhận rằng kiểm tra là một phần cơ bản của quá trình phát triển.

Phần mềm kiểm thử là một hoạt động phức tạp và hầu như vẫn chưa được xác định rõ ràng. Nó cũng cần thiết cho tất cả các giai đoạn của một dự án phát triển không tầm thường. Tập hợp các kỳ vọng đi kèm với tính từ "doanh nghiệp" khi được kết hợp với "phần mềm" là một nhóm lớn, bao gồm ít nhất là bảo mật, tính ổn định, tính nhất quán, khả năng dự đoán và quy mô. Phần mềm được kỳ vọng sẽ dự đoán và tích hợp nhanh chóng những thay đổi nhiều mặt và không thể đoán trước về khối lượng và hình dạng dữ liệu, kỳ vọng của người dùng, mục tiêu kinh doanh, nhân sự, thậm chí cả quy định của chính phủ và môi trường pháp lý.

Kiểm tra không chỉ là một quá trình phát hiện lỗi và sửa lỗi. Ví dụ: phát triển theo hướng kiểm tra, đòi hỏi phải có kiểm tra trước sự tồn tại của bất kỳ mã nào! Nói chung, kiểm thử là quá trình so sánh giữa hành vi hiện có và hành vi mong muốn trong phần mềm, nơi thông tin mới liên tục được đưa trở lại quy trình. Theo nghĩa này, thử nghiệm liên quan đến việc mô hình hóa các kỳ vọng và xác minh rằng các chức năng riêng lẻ, các đơn vị cấu thành và các đường dẫn thực hiện có thỏa mãn mong đợi của nhà thiết kế, lập trình viên, chủ sở hữu sản phẩm, khách hàng và toàn bộ tổ chức hay không.

Để đạt được mục tiêu này, các bộ phận của hệ thống phải được kiểm tra tốt, sao cho việc đưa vào hoặc loại trừ chúng sẽ có tác động nhất quán và đã biết đối với cả bản thân hệ thống và trên các kỳ vọng đối với hệ thống, chẳng hạn như doanh số bán hàng, tốc độ triển khai hoặc khả năng xử lý dữ liệu an toàn. Do đó, kiểm tra cũng là quản lý rủi ro. Bằng cách này, các điểm bất thường có thể được định lượng và đủ điều kiện, trong khi các vết lồi trên địa hình giờ đây có thể thông báo hữu ích cho hiểu biết hiện tại của chúng ta về bản đồ sao cho số lần sai lầm (hoặc khiếm khuyết) giảm và sự tự tin của chúng ta tăng lên. Kiểm tra giúp chúng tôi đo lường khi chúng tôi hoàn thành.

Có rất nhiều trường có tư tưởng thi khi nào, thi cái gì, thi ai, thi như thế nào. Trong chương này, chúng ta sẽ tập trung vào một số mẫu đã biết và hữu ích để kiểm tra các ứng dụng Node, điều tra các công cụ Node gốc để kiểm tra tính toàn vẹn của mã, kiểm tra chung với khung Mocha và kiểm tra trình duyệt không sử dụng; lần cuối cùng cho phép thử nghiệm JavaScript dựa trên trình duyệt từ trong môi trường Node.

Khi bạn chuyển qua chương này, có thể hữu ích khi nhớ rằng việc tích hợp triết lý kiểm thử vào một dự án là rất khó để thực hiện tốt. Thông thường, việc triển khai trở nên khó khăn hơn khi kích thước của một cơ sở mã chưa được kiểm tra tăng lên và / hoặc khi số lượng người đóng góp tăng lên. Một số người có thể nói rằng một chiến lược kiểm tra toàn diện nên được triển khai càng sớm càng tốt — điều gì đó cần xem xét khi bạn bắt tay vào dự án Node tiếp theo của mình.

## 9.1. Tại sao kiểm tra lại quan trọng

Một chiến lược kiểm tra tốt xây dựng sự tự tin thông qua việc tích lũy bằng chứng và ngày càng rõ ràng. Trong một công ty, điều này có thể có nghĩa là một số tiêu chí để thực hiện chiến lược kinh doanh đã được đáp ứng, cho phép phát hành một sản phẩm mới. Các nhà phát triển trong nhóm dự án có được niềm vui của một thẩm phán tự động xác nhận hoặc phủ nhận liệu các thay đổi được cam kết với cơ sở mã là đúng đắn. Với một khuôn khổ thử nghiệm tốt, việc tái cấu trúc lại mất đi sự nguy hiểm của nó: lời cảnh báo "nếu bạn phá vỡ nó, bạn sẽ sở hữu nó" từng gây áp lực tiêu cực cho các nhà phát triển với những ý tưởng mới không còn đáng ngại nữa. Với một hệ thống kiểm soát phiên bản tốt và quy trình thử nghiệm / phát hành, bất kỳ thay đổi đột phá nào đều có thể được khôi phục mà không có tác động tiêu cực, giải phóng sự tò mò và thử nghiệm.

Ba loại kiểm tra phổ biến là: kiểm thử đơn vị, kiểm tra chức năng và kiểm tra tích hợp. Mặc dù mục tiêu của chúng tôi trong chương này không phải là đưa ra một lý thuyết chung về cách để kiểm tra các ứng dụng, sẽ rất hữu ích khi tóm tắt ngắn gọn các bài kiểm tra đơn vị, chức năng và tích hợp là gì, thành viên nào của nhóm quan tâm nhất đến từng ứng dụng và cách chúng ta có thể chia nhỏ cơ sở mã thành các đơn vị có thể kiểm tra.

## 9.2. Bài kiểm tra đơn vị

Kiểm thử đơn vị liên quan đến các đơn vị hành vi của hệ thống. Mỗi đơn vị đang được kiểm tra nên đóng gói một tập hợp các đường dẫn mã rất nhỏ, không vướng mắc. Khi kiểm tra đơn vị không thành công, điều này lý tưởng nên chỉ ra rằng một phần riêng biệt của chức năng tổng thể bị hỏng. Nếu một chương trình có một tập hợp các bài kiểm tra đơn vị được mô tả tốt, thì mục đích và hành vi mong đợi của toàn bộ chương trình sẽ dễ dàng để hiểu, toàn bộ chương trình phải dễ hiểu. Thử nghiệm đơn vị áp dụng một quan điểm hạn chế cho các phần nhỏ của hệ thống, không quan tâm đến cách các phần đó có thể được gói lại thành các khối chức năng lớn hơn.

Một thử nghiệm đơn vị mẫu có thể được mô tả theo cách này: khi giá trị 123 được chuyển cho một phương thức validate\_phone\_number

(), kiểm tra sẽ trả về false. Không có sự nhầm lẫn về những gì đơn vị này làm, và một lập trình viên có thể sử dụng nó một cách tự tin.

Các bài kiểm tra đơn vị thường được viết và đọc bởi các lập trình viên. Các phương pháp lớp là ứng cử viên tốt cho các bài kiểm tra đơn vị, cũng như các điểm cuối dịch vụ khác có chữ ký đầu vào ổn định và được hiểu rõ, với kết quả đầu ra dự kiến có thể được xác thực chính xác. Nói chung, người ta cho rằng các bài kiểm tra đơn vị chạy nhanh chóng. Nếu một bài kiểm tra đơn vị mất nhiều thời gian để thực thi thì có khả năng đoạn mã được kiểm tra phức tạp hơn nhiều so với mức cần thiết.

Kiểm thử đơn vị không quan tâm đến cách một chức năng hoặc phương pháp được kiểm tra sẽ nhận được đầu vào của nó hoặc cách nó sẽ được sử dụng nói chung. Một bài kiểm tra cho một thêm vào phương thức không nên quan tâm đến việc phương thức sẽ được sử dụng trong máy tính hay một nơi nào khác, nó chỉ cần kiểm tra xem hai đầu vào số nguyên (3,4) có khiến đơn vị phát ra kết quả chính xác (7) hay không. Kiểm thử đơn vị không quan tâm đến vị trí của nó trong cây phụ thuộc. Vì lý do này, các bài kiểm tra đơn vị thường sẽ "mô phỏng" hoặc "sơ khai" các nguồn dữ liệu, chẳng hạn như chuyển hai số nguyên ngẫu nhiên đến một thêm vào phương pháp. Miễn là các đầu vào là điển hình, chúng không cần phải thực tế. Ngoài ra, các bài kiểm tra đơn vị tốt là đáng tin cậy: không bị cản trở bởi các yếu tố phụ thuộc bên ngoài, chúng sẽ vẫn có giá trị bất kể hệ thống xung quanh chúng thay đổi như thế nào.

Kiểm tra đơn vị chỉ xác nhận rằng một thực thể hoạt động riêng lẻ. Kiểm tra xem các đơn vị có thể hoạt động tốt khi kết hợp với nhau hay không là mục đích của kiểm tra chức năng.

## 9.3. Kiểm tra chức năng

Khi các bài kiểm tra đơn vị liên quan đến các hành vi cụ thể, các bài kiểm tra chức năng được thiết kế để xác nhận các phần chức năng. Sự không rõ ràng của từ gốc "chức năng", đặc biệt đối với các lập trình viên, có thể dẫn đến nhầm lẫn, trong đó "kiểm thử đơn vị" được gọi là "kiểm tra chức năng" và ngược lại. Một bài kiểm tra chức năng kết hợp nhiều đơn vị thành một phần chức năng, chẳng hạn như, "khi người dùng nhập tên người dùng và mật khẩu và nhấp vào gửi, người dùng đó sẽ được đăng nhập vào hệ thống". Chúng ta có thể dễ dàng nhận thấy rằng nhóm chức năng này sẽ bao gồm nhiều bài kiểm tra đơn vị, một để xác thực tên người dùng, một để xử lý một lần nhấp vào nút, v.v.

Kiểm tra chức năng thường là mối quan tâm của những người chịu trách nhiệm về một số miền cụ thể trong một ứng dụng. Trong khi các lập trình viên và nhà phát triển sẽ vẫn là những người thực hiện các thử nghiệm này, các nhà quản lý sản phẩm hoặc các bên liên quan tương tự thường sẽ nghĩ ra chúng. Phần lớn các thử nghiệm này kiểm tra xem các thông số kỹ thuật của sản phẩm lớn hơn có được đáp ứng hay không, thay vì kiểm tra tính đúng kỹ thuật.

Bài kiểm tra đơn vị mẫu cho validate\_phone\_number đưa ra trước đó có thể tạo thành một phần của kiểm tra chức năng với mô tả này: khi người dùng nhập sai số điện thoại, hãy mở hộp thoại trợ giúp mô tả định dạng phù hợp ở quốc gia của người dùng đó. Việc một ứng dụng làm phiền để giúp người dùng mắc lỗi với số điện thoại là một nỗ lực trừu tượng rất khác so với việc đơn giản xác nhận một thực thể kỹ thuật như

số điện thoại. Các bài kiểm tra chức năng có thể được coi là mô hình trừu tượng về mức độ hoạt động của một số tập hợp các đơn vị với nhau để đáp ứng nhu cầu sản phẩm.

Bởi vì các thử nghiệm chức năng được thực hiện dựa trên sự kết hợp của nhiều đơn vị, dự kiến rằng, không giống như kiểm tra đơn vị riêng biệt, việc thực hiện chúng sẽ bao gồm việc trộn lẫn các mối quan tâm từ bất kỳ số lượng đối tượng hoặc hệ thống bên ngoài nào. Trong ví dụ đăng nhập trước, chúng ta thấy cách kiểm tra chức năng tương đối đơn giản có thể cắt ngang cơ sở dữ liệu, giao diện người dùng, bảo mật và các lớp ứng dụng khác. Bởi vì nó phức tạp hơn về mặt thành phần, một bài kiểm tra chức năng dự kiến sẽ mất nhiều thời gian hơn để chạy hơn một bài kiểm tra đơn vị. Các thử nghiệm chức năng dự kiến sẽ thay đổi ít thường xuyên hơn các thử nghiệm đơn vị, chẳng hạn như các thay đổi về chức năng thường đại diện cho các bản phát hành chính, trái ngược với các thay đổi nhỏ mà các sửa đổi thử nghiệm đơn vị thường chỉ ra.

Lưu ý rằng, giống như các bài kiểm tra đơn vị, các bài kiểm tra chức năng tự tách biệt khỏi mối quan tâm về cách tổng thể nhóm chức năng được kiểm tra, liên quan đến phần còn lại của một ứng dụng. Vì lý do này, dữ liệu giả có thể được sử dụng làm bối cảnh để chạy các thử nghiệm chức năng, vì bản thân nhóm chức năng không quan tâm đến ảnh hưởng của nó đối với trạng thái ứng dụng chung, là lĩnh vực kiểm tra tích hợp.

## 9.4. Kiểm tra tích hợp

Kiểm tra tích hợp đảm bảo rằng toàn bộ hệ thống được kết nối chính xác với nhau, sao cho người dùng sẽ cảm thấy ứng dụng đang hoạt động chính xác. Theo cách này, các bài kiểm tra tích hợp thường xác nhận chức năng mong đợi của toàn bộ ứng dụng hoặc một trong một nhóm nhỏ các chức năng quan trọng của sản phẩm.

Sự khác biệt quan trọng nhất giữa tích hợp và các loại thử nghiệm khác đang được thảo luận là các thử nghiệm tích hợp phải được thực hiện trong môi trường thực tế, trên cơ sở dữ liệu thực với dữ liệu miền thực, trên máy chủ và các hệ thống khác phản ánh môi trường sản xuất mục tiêu. Bằng cách này, các bài kiểm tra tích hợp có thể dễ dàng phá vỡ các bài kiểm tra đơn vị và chức năng đã vượt qua trước đây.

Ví dụ: một bài kiểm tra đơn vị cho validate\_phone\_number có thể đã cho một đầu vào như 555-123-4567, nhưng trong quá trình kiểm tra tích hợp, nó sẽ không vượt qua một số dữ liệu hệ thống thực (và hợp lệ) như 555.123.4567. Tương tự, một bài kiểm tra chức năng có thể kiểm tra thành công khả năng mở hộp thoại trợ giúp của một hệ thống lý tưởng, nhưng khi được tích hợp với trình duyệt mới hoặc thời gian chạy khác, nó sẽ thấy rằng chức năng mong đợi không đạt được. Một ứng dụng chạy tốt với một hệ thống tệp cục bộ có thể bị lỗi khi chạy với hệ thống tệp phân tán.

Do sự phức tạp tăng thêm này, các kiến trúc sư hệ thống — các thành viên trong nhóm có thể áp dụng quan điểm cấp cao hơn về tính đúng đắn của hệ thống — thường thiết kế các thử nghiệm tích hợp. Các bài kiểm tra này tìm ra lỗi trong hệ thống dây điện mà các bài kiểm tra cô lập không thể nhận ra. Không có gì ngạc nhiên khi các bài kiểm tra tích hợp thường có thể mất nhiều thời gian để chạy, thường được thiết kế để không chỉ chạy các tình huống đơn giản mà còn để bắt chước các môi trường thực tế, tải cao dự kiến.

Một công cụ mới mạnh mẽ để thực hiện kiểm tra sâu về hiệu suất của ứng dụng Node là DTrace. Đám mây lưu trữ của Joyent cung cấp quyền truy cập vào dụng cụ ( http://www.joyent.com/blog/bruning-questionsdebugging-node-apps-with-dtrace), và hỗ trợ ngày càng tăng. A

Tổng quan tốt đẹp về DTrace có thể được tìm thấy tại đây:

http://mcavage.github.com/presentations/dtrace\_

conf\_2012-04-03 / # chương trình làm việc

## 9.5. Công cụ kiểm tra và gỡ lỗi Node Native

Sở thích đối với mã được thử nghiệm đã hình thành một phần đặc tính của cộng đồng Node kể từ khi thành lập, phản ánh trong thực tế là hầu hết các mô-đun Node phổ biến, ngay cả những mô-đun đơn giản, đều được phân phối với các bộ thử nghiệm. Trong khi quá trình phát triển phía trình duyệt với JavaScript đã phải chịu đựng trong nhiều năm mà không có các công cụ kiểm tra có thể sử dụng được, thì bản phân phối Node tương đối non trẻ có rất nhiều. Có lẽ vì vậy, nhiều khung thử nghiệm của bên thứ ba hoàn thiện và dễ sử dụng đã được phát triển cho Node. Điều này khiến nhà phát triển không có lý do gì để viết mã chưa được kiểm tra! Chúng ta hãy xem xét một số công cụ được cung cấp để gỡ lỗi và kiểm tra các chương trình Node.

## 9.6. Ghi vào bảng điều khiển

Đầu ra bảng điều khiển là công cụ kiểm tra và gỡ lỗi cơ bản nhất, cung cấp một cách nhanh chóng để xem điều gì đang xảy ra tại một số điểm trong tập lệnh. Có thể truy cập toàn cầu console.log thường được sử dụng khi gỡ lỗi.

Node đã làm phong phú thêm cơ chế đầu ra tiêu chuẩn này bằng các phương pháp hữu ích hơn, nhu la console.error (Chuỗi, Chuỗi…), mà in các đối số cho stderr hơn là stdout, và console.dir (Đối tượng), cái nào chạy use.inspect ( xem sau) trên đối tượng được cung cấp và ghi kết quả vào stdout.

Mẫu sau thường thấy khi một nhà phát triển muốn theo dõi thời gian một đoạn mã thực thi:

var start = new Date (). getTime (); for (x = 0; x <1000; x

++) {...}

console.log (new Date (). getTime () - start); // Một thời gian, tính bằng mili giây

Các console.time và console.timeEnd các phương pháp chuẩn hóa mẫu này:

console.time ('Thêm 1000000 bản ghi');

var rec = [];

for (var i = 0; i <1000000; i ++) { rec.push (1);

}

console.timeEnd ('Thêm 1000000 bản ghi');

//> Thêm 1000000 bản ghi: 59ms

Chúng ta sẽ thấy các phương pháp bảng điều khiển đặc biệt khác khi thảo luận về khẳng định mô-đun và thực hiện dấu vết ngăn xếp ở phần sau của chương này.

Có thể truy cập nhiều công cụ đầu ra hơn thông qua sử dụng mô-đun:

use.debug (Chuỗi): Chặn quá trình đang chạy và xuất chuỗi đến stderr:

> request ('use'). debug ("Error!") GỢI Ý: Lỗi!

use.error (Chuỗi, [Chuỗi…]): Giống như gỡ lỗi, nhưng có thể gửi nhiều chuỗi đến stderr:

> request ('use'). error ("Error1", "Error2", "Error3")

Error1

Error2

Error3

use.puts (Chuỗi, [Chuỗi…]): Xuất nhiều chuỗi thành stdout, thêm một dòng mới sau mỗi đối số:

> request ('use'). put ("A", "B", "C")

A

B

C

use.print (Chuỗi, [Chuỗi…]): Xuất nhiều chuỗi thành stdout, mà không cần thêm dòng mới sau mỗi đối số:

> request ('dùng'). print (1,2,3)

123

use.log (Chuỗi): Xuất một thông báo có dấu thời gian tới stdout:

> request ('use'). log ('Tin nhắn có dấu thời gian') 20 tháng 8 18:32:08 - Tin nhắn có

dấu thời gian

## 9.7. Định dạng đầu ra bảng điều khiển

Các phương pháp trước đều rất hữu ích khi ghi các chuỗi đơn giản. Thông thường, dữ liệu ghi nhật ký hữu ích có thể cần được định dạng, bằng cách gộp nhiều giá trị vào một chuỗi đơn hoặc bằng cách hiển thị gọn gàng một đối tượng dữ liệu phức tạp. Các use.format và use.inspect có thể sử dụng các phương pháp để xử lý các trường hợp này.

Phương thức use.format (format, [arg, arg…])

Phương pháp này cho phép một chuỗi định dạng được tạo ra từ các trình giữ chỗ, mỗi chuỗi sẽ nắm bắt và hiển thị các giá trị bổ sung được chuyển vào. Ví dụ:

> use.format ('% s:% s', 'foo', 'bar')

'foo: bar'

Ở đây, chúng ta thấy rằng hai trình giữ chỗ (tiền tố là%) được thay thế theo thứ tự bằng các đối số đã truyền. Phần giữ chỗ mong đợi một trong ba loại giá trị sau:

%S: Một chuỗi

% d: Một số, số nguyên hoặc số thực

% j: Một đối tượng JSON

Nếu một số lượng lớn hơn các đối số được gửi đi, các đối số bổ sung được chuyển đổi thành chuỗi thông qua use.inspect (), và được nối với phần cuối của đầu ra, được phân tách bằng dấu cách:

> use.format ('% s:% s', 'foo', 'bar', 'baz'); 'foo: bar baz'

Nếu không có chuỗi định dạng nào được gửi, các đối số chỉ được chuyển đổi thành chuỗi và được nối với nhau, phân tách bằng dấu cách.

## 9.8. Phương thức use.inspect (object, [options])

Sử dụng phương pháp này khi muốn biểu diễn chuỗi của một đối tượng. Thông qua việc thiết lập các tùy chọn khác nhau, giao diện của đầu ra có thể được kiểm soát:

showHidden: Giá trị mặc định là false. Nếu đúng, các thuộc tính không liệt kê của đối tượng sẽ được hiển thị.

chiều sâu: Một định nghĩa đối tượng, chẳng hạn như đối tượng JSON, có thể được lồng sâu vào nhau. Theo mặc định use.inspect chỉ đi qua hai cấp độ vào đối tượng. Sử dụng tùy chọn này để tăng (hoặc giảm) độ sâu đó.

màu sắc: Cho phép tô màu đầu ra (xem đoạn mã sau).

customInspect: Nếu đối tượng đang được xử lý có quan sát phương thức được định nghĩa, đầu ra của phương thức đó sẽ được sử dụng thay cho phương pháp chuỗi mặc định của Node (xem đoạn mã sau). Giá trị mặc định là true.

Đặt trình kiểm tra tùy chỉnh:

var ut = request ('use'); var obj = function () {

this.foo = "thanh";

}

obj.prototype.inspect = function () {

trả về "CUSTOM INSPECTOR";

}

console.log (used.inspect (new obj))

// KIỂM TRA TÙY CHỈNH

console.log (use.inspect (new obj, {customInspect: false})) // {foo: 'bar'}

Điều này có thể rất hữu ích khi ghi nhật ký các đối tượng phức tạp hoặc các đối tượng có giá trị quá lớn khiến đầu ra của bảng điều khiển không thể đọc được.

Các màu sắc: đúng tùy chọn của use.inspect cho phép tô màu đầu ra đối tượng, có thể hữu ích cho các đối tượng phức tạp. Màu sắc được gán cho các kiểu dữ liệu. Các nhiệm vụ mặc định được đặt trong use.inspect.styles:

{đặc biệt: 'lục lam', số: 'màu vàng', boolean: 'màu vàng',

không xác định: 'xám', null: 'bold', string: 'green', ngày: 'đỏ tươi', regexp: 'red'}

Các phép gán màu mặc định này có thể được hoán đổi bằng một trong các mã màu ANSI được hỗ trợ được lưu trữ trong use.inspect.colors đối tượng: đậm, nghiêng, gạch dưới, đảo ngược, trắng, xám, đen, xanh lam, lục lam, xanh lục, đỏ tươi, đỏ và vàng. Ví dụ: để có số lượng giá trị của các đối tượng được hiển thị trong màu xanh lá thay vì mặc định của màu vàng sử dụng mã sau:

use.inspect.styles.number = "xanh"; console.log (using.inspect ([1,2,4,5,6], {Colors: true})); // [1,2,3,4,5,6] Các số có màu xanh lục

## 9.9. Trình gỡ lỗi Node

Hầu hết các nhà phát triển đã sử dụng IDE để phát triển. Một tính năng chính của tất cả các môi trường phát triển tốt là quyền truy cập vào trình gỡ lỗi, cho phép thiết lập các điểm ngắt trong chương trình ở những nơi cần kiểm tra trạng thái hoặc các khía cạnh khác của thời gian chạy.

V8 được phân phối với trình gỡ lỗi mạnh mẽ (thường được thấy là cấp nguồn cho bảng điều khiển công cụ dành cho nhà phát triển của trình duyệt Google Chrome) và Node này có thể truy cập được. Nó được gọi bằng cách sử dụng chỉ thị gỡ lỗi:

> nút gỡ lỗi somescript.js

Bây giờ có thể thực hiện gỡ lỗi kiểm tra và kiểm tra từng bước đơn giản trong một chương trình nút. Hãy xem xét chương trình sau:

myVar = 123; setTimeout (function () { trình gỡ lỗi;

console.log ("thế giới");

}, 1000); console.log ("xin chào");

Lưu ý trình gỡ lỗi chỉ thị. Thực thi chương trình này mà không sử dụng trình gỡ lỗi chỉ thị sẽ dẫn đến xin chào được hiển thị, theo sau là thế giới, một giây sau. Khi sử dụng chỉ thị, người ta sẽ thấy điều này:

> nút gỡ lỗi somescript.js

<trình gỡ lỗi đang nghe trên cổng 5858 đang kết nối

... ok

ngắt trong debug-sample.js: 1

myVar = 123;

setTimeout (hàm () {3 trình gỡ lỗi;gỡ lỗi>

Khi một điểm ngắt bị tấn công, chúng ta sẽ được cung cấp CLI cho chính trình gỡ lỗi, từ đó chúng ta có thể thực hiện một số gỡ lỗi tiêu chuẩn và các lệnh khác:

tiếp hoặc là c: Tiếp tục thực hiện từ điểm ngắt cuối cùng, cho đến điểm ngắt tiếp theo

bươc hoặc là S: Bước vào, tức là tiếp tục chạy cho đến khi một dòng nguồn mới (hoặc điểm ngắt) được nhấn, sau đó trả lại quyền điều khiển cho trình gỡ lỗi

kế tiếp hoặc là n: Giống như bươc, nhưng các lệnh gọi hàm được thực hiện trên dòng nguồn mới được thực hiện mà không dừng lại

ngoài hoặc là o: Bước ra, tức là thực thi phần còn lại của hàm hiện tại và quay lại hàm mẹ

backtrace hoặc là bt: Theo dõi các bước đến khung thực thi hiện tại, tương tự như:

…

# 3 Module.\_compile module.js: 456: 26

# 4 Module.\_extensions..js module.js: 474: 10

# 5 Module.load module.js: 356: 32 … Vân vân.

setBreakpoint () hoặc là sb (): Đặt điểm ngắt trên dòng hiện tại

setBreakpoint (Số nguyên) hoặc là sb (Số nguyên): Đặt điểm ngắt trên dòng được chỉ định

clearBreakpoint () hoặc là cb (): Xóa điểm ngắt trên dòng hiện tại

clearBreakpoint (Số nguyên) hoặc là cb (Số nguyên): Xóa điểm ngắt trên dòng được chỉ định

chạy: Nếu tập lệnh của trình gỡ lỗi đã kết thúc, điều này sẽ bắt đầu lại

khởi động lại: Kết thúc và bắt đầu lại tập lệnh

tạm ngừng hoặc là p: Tạm dừng mã đang chạy

giết chết: Hủy tập lệnh đang chạy

bỏ: Thoát khỏi trình gỡ lỗi

phiên bản: Hiển thị phiên bản V8

tập lệnh: Liệt kê tất cả các tập lệnh đã tải

Để lặp lại lệnh trình gỡ lỗi cuối cùng, chỉ cần nhấn Đi vào trên bàn phím của bạn.

Quay lại tập lệnh chúng tôi đang gỡ lỗi, nhập tiếp vào trình gỡ lỗi sẽ dẫn đến kết quả sau:

gỡ lỗi> tiếp tục

<xin chào // ... một khoảng dừng 1000 mili giây sẽ xảy ra, sau đó ... ngắt trong debug-sample.js: 3

myVar = 123;

setTimeout (hàm () {3 trình gỡ lỗi;

4 console.log ("thế giới");

5}, 1000); gỡ lỗi>

Nhận thấy như thế nào xin chào không được in khi chúng tôi khởi động trình gỡ lỗi, mặc dù người ta sẽ mong đợi lệnh console.log ('xin chào') để thực thi trước khi đạt được điểm ngắt trong setTimeout gọi lại. Trình gỡ lỗi không thực thi tại thời điểm chạy: nó đang đánh giá tại thời điểm biên dịch cũng như tại thời điểm chạy, cho bạn khả năng hiển thị sâu về cách mã bytecode cho chương trình của bạn đang được lắp ráp và cuối cùng được thực thi.

Thường sẽ hữu ích tại một điểm ngắt để thực hiện một số kiểm tra, chẳng hạn như giá trị của các biến. Có một lệnh bổ sung có sẵn cho trình gỡ lỗi: repl,

cho phép điều này. Hiện tại, trình gỡ lỗi của chúng tôi bị dừng sau khi đã phân tích cú pháp thành công tập lệnh và thực thi chức năng đầu tiên được đẩy vào vòng lặp sự kiện, console.log ('xin chào'). Điều gì sẽ xảy ra nếu chúng tôi muốn kiểm tra giá trị của myVar?

Sử dụng repl: gỡ lỗi> repl

Nhấn Ctrl + C để rời khỏi repl gỡ lỗi

> myVar

123

Chơi xung quanh REPL tại đây, thử nghiệm cách nó có thể được sử dụng.

Tại thời điểm này, chương trình của chúng tôi có một lệnh duy nhất còn lại để thực thi, in

thế giới. Một ngay lập tức tiếp lệnh sẽ thực thi lệnh cuối cùng này, vòng lặp sự kiện sẽ không có gì phải làm nữa và tập lệnh của chúng tôi sẽ kết thúc:

gỡ lỗi> tiếp tục <thế giới chương trình chấm dứt gỡ lỗi>

Như một thử nghiệm, hãy chạy lại tập lệnh, sử dụng kế tiếp thay vì tiếp, ngay trước khi thực hiện bối cảnh cuối cùng này. Tiếp tục đánh Đi vào và cố gắng làm theo mã đang được thực thi. Bạn sẽ thấy điều đó sau thế giới được in timers.js script sẽ được đưa vào ngữ cảnh thực thi này, khi Node dọn dẹp sau khi kích hoạt thời gian chờ. Chạy tập lệnh lệnh trong trình gỡ lỗi tại thời điểm này. Bạn sẽ thấy một cái gì đó như:

gỡ lỗi> tiếp theo ngắt thời gian. js: 125

123

124 gỡ lỗi (msecs + 'danh sách trống'); khẳng định 125 (L.isEmpty (danh sách));

list.close ();

xóa danh sách [msecs];

gỡ lỗi> tập lệnh \* 37: timer.js

46: debug-sample.js gỡ lỗi>

Sẽ rất hữu ích khi thử nghiệm với nhiều phương pháp khác nhau, tìm hiểu cả những gì đang xảy ra khi Node thực thi các tập lệnh ở cấp độ sâu, cũng như giúp đáp ứng nhu cầu gỡ lỗi của bạn.

Có thể hữu ích khi đọc tài liệu sau, mô tả cách giao diện trình gỡ lỗi Google Chrome được sử dụng tại https: //Develop.google.com/chrome-developer-tools/docs/ javascript-debugging # breakpoints.

Miroslav Bajtos ' mô-đun kiểm tra nút là thứ cần xem xét để gỡ lỗi, cho phép nhà phát triển gỡ lỗi từ xa một ứng dụng Node từ trình duyệt Chrome:

https://github.com/node-ins Inspector/node-ins rà

## 9.10. Mô-đun khẳng định

Điểm giao khẳng định mô-đun được sử dụng để kiểm tra đơn vị đơn giản. Trong nhiều trường hợp, nó đủ để làm giàn giáo cơ bản cho các thử nghiệm hoặc được sử dụng làm thư viện xác nhận cho các khung thử nghiệm (như Mocha, như chúng ta sẽ thấy ở phần sau). Cách sử dụng rất đơn giản: chúng tôi muốn khẳng định sự thật của điều gì đó và thông báo lỗi nếu khẳng định của chúng tôi không đúng. Ví dụ:

> Request ('khẳng định'). bằng (1,2, "Không bằng!") AssertionError: Không bằng!

tại repl: 1: 20

. . .

Nếu khẳng định là đúng (cả hai giá trị đều bằng nhau) sẽ không có gì được trả về:

> Request ('khẳng định'). bằng (1,1, "Không bằng!") chưa xác định

Tuân theo Quy tắc im lặng của UNIX (khi một chương trình không có gì đáng ngạc nhiên, thú vị hoặc hữu ích để nói, nó sẽ không nói gì), các xác nhận chỉ trả về một giá trị khi xác nhận không thành công. Giá trị trả về có thể được tùy chỉnh bằng cách sử dụng đối số thông báo tùy chọn, như đã thấy trong phần trước.

Các khẳng định API mô-đun bao gồm một tập hợp các thao tác so sánh với các chữ ký cuộc gọi giống hệt nhau: giá trị thực, giá trị mong đợi và thông báo tùy chọn để hiển thị khi so sánh không thành công. Các phương thức thay thế hoạt động như phím tắt hoặc trình xử lý cho các trường hợp đặc biệt cũng được cung cấp.

Cần phải phân biệt giữa so sánh danh tính (===) và so sánh bình đẳng (==) mà trước đây thường được gọi là so sánh bình đẳng nghiêm ngặt (như trường hợp trong khẳng định API). Bởi vì JavaScript sử dụng kiểu nhập động, khi hai giá trị thuộc các kiểu khác nhau được so sánh bằng cách sử dụng toán tử bình đẳng ==, một nỗ lực được thực hiện để ép buộc (hoặc ép kiểu) một giá trị vào giá trị kia, một loại phép toán mẫu số chung. Ví dụ:

1 == "1" // true false == "0" // true false == null // false

Như bạn có thể mong đợi, những kiểu so sánh này có thể dẫn đến kết quả đáng ngạc nhiên. Lưu ý các kết quả dễ đoán hơn khi so sánh danh tính được sử dụng:

1 === "1" // false false === "0" // false false === null // false

Điều cần nhớ là toán tử === không thực hiện cưỡng chế kiểu trước khi so sánh, trong khi toán tử bình đẳng so sánh sau khi ép kiểu.

Ngoài ra, bởi vì các đối tượng có thể chứa các giá trị giống nhau nhưng không được dẫn xuất từ cùng một hàm tạo, danh tính của hai đối tượng có cùng giá trị là khác biệt đối với các đối tượng, việc nhận dạng yêu cầu cả hai toán hạng tham chiếu đến cùng một đối tượng:

var a = function () {}; var b = new a;

var c = new a; var d = b;

console.log (a == function () {}) // false console.log (b == c) // false

console.log (b == d) // true console.log (b.constructor === c.constructor); // thật

Cuối cùng, khái niệm bình đẳng sâu sắc được sử dụng để so sánh đối tượng mà danh tính không cần chính xác. Hai đối tượng hoàn toàn bằng nhau nếu cả hai đều sở hữu cùng một số thuộc tính được sở hữu, cùng một nguyên mẫu, cùng một bộ khóa (mặc dù không nhất thiết phải cùng một thứ tự) khóa và các giá trị tương đương (không giống nhau) cho mỗi thuộc tính của chúng:

var a = [1,2,3]; var b =

[1,2,3];

khẳng định.deepEqual (a, b); // vượt qua

khẳng định.strictEqual (a, b); // ném AssertionError: [1,2,3] === [1,2,3]

Sẽ rất hữu ích khi kiểm tra các giả định của bạn về cách các giá trị được hiểu so với nhau bằng cách thiết kế một số thử nghiệm khẳng định. Các kết quả có thể làm bạn ngạc nhiên. Mô-đun khẳng định của Node cung cấp các phương pháp so sánh sau:.

khẳng định.equal (thực tế, dự kiến, [nội dung]): Kiểm tra sự bình đẳng bị ép buộc

với dấu ==.

khẳng định.notEqual (thực tế, dự kiến, [nội dung]): Kiểm tra sự bình đẳng bị ép buộc với! =.

khẳng định.deepEqual (thực tế, dự kiến, [nội dung]): Kiểm tra sự bình đẳng sâu sắc.

khẳng định.notDeepEqual (thực tế, dự kiến, [nội dung]): Kiểm tra cho

bất bình đẳng sâu sắc.

khẳng định.strictEqual (thực tế, dự kiến, [nội dung]): Kiểm tra danh tính sự tương đương. Giống hệt với việc sử dụng toán tử ===.

khẳng định.notStrictEqual (thực tế, dự kiến, [nội dung]): Kiểm tra cho nhận dạng không phù hợp. Giống hệt với việc sử dụng toán tử! ==.

khẳng định (giá trị, [nội dung]): Ném lỗi nếu giá trị đã gửi không trung thực.

khẳng định.ok (giá trị, [nội dung]): Giống hệt với khẳng định (giá trị).

khẳng định.ifError (giá trị): Ném lỗi nếu giá trị là trung thực.

khẳng định.throws (khối, [lỗi], [nội dung]): Kiểm tra xem mã được cung cấp khối ném một lỗi. Giá trị lỗi tùy chọn có thể là hàm tạo lỗi, biểu thức chính quy hoặc hàm xác thực trả về giá trị Boolean.

khẳng định.doesNotThrow (khối, [lỗi], [nội dung]): Kiểm tra nếu cung cấp khối mã không xuất hiện lỗi.

khẳng định.fail (thực tế, dự kiến, tin nhắn, nhà điều hành): Ném một ngoại lệ. Điều này hữu ích nhất khi ngoại lệ bị mắc kẹt bởi khối try / catch:

thử { khẳng định.fail (1,2, "Xấu!", "KHÔNG PHẢI EQ")

} bắt (e) { console.log (e)

}

Đoạn mã trước tạo ra kết quả sau:

{message: 'Bad!', thực tế: 1, mong đợi: 2, toán tử: 'NOT EQ', name: 'AssertionError: Bad!' }

Phương thức tắt để ghi lại kết quả xác nhận có sẵn trong API bảng điều khiển:

> console.assert (1 == 2, "Không!") AssertionError: Không!

Để có giải thích chi tiết hơn về cách so sánh được thực hiện trong JavaScript, tham khảo ý kiến: https://developer.mozilla.org/en-US/docs/Web/ JavaScript / Tham chiếu / Toán tử / Comparison\_Operators.

Điểm giao khẳng định mô-đun bị ảnh hưởng mạnh mẽ bởi CommonJS đặc điểm kỹ thuật kiểm tra: http://wiki.commonjs.org/wiki/Unit\_ Testing / 1.0.

## 9.11. Hộp cát

Đối với một số ứng dụng, rất hữu ích khi có thể chạy một tập lệnh trong ngữ cảnh được bảo vệ, tách biệt khỏi phạm vi ứng dụng chung. Đối với những mục đích này, Node cung cấp vm mô-đun, môi trường hộp cát bao gồm phiên bản V8 mới và bối cảnh thực thi giới hạn để chạy các khối tập lệnh. Ví dụ: một người có thể muốn thực thi mã được gửi từ trình chỉnh sửa dựa trên web trong một máy ảo như vậy:

var vm = request ('vm'); var sandbox =

{

đếm: 2

}

var someRandomCode = 'var foo = ++ count;'; vm.runInNewContext

(someRandomCode, hộp cát);

console.log (Request ('dùng'). Kiểm tra (hộp cát));

// {count: 3, foo: 3}

Ở đây chúng ta thấy cách một hộp cát được cung cấp trở thành phạm vi thực thi cục bộ cho tập lệnh được cung cấp. Tập lệnh đang chạy chỉ có thể hoạt động trong đối tượng hộp cát được cung cấp và bị từ chối quyền truy cập vào ngay cả các hình cầu Node tiêu chuẩn, chẳng hạn như quá trình đang chạy:

. . .

var someRandomCode = 'var foo = ++ count; process.exit (0); '; vm.runInNewContext

(someRandomCode, hộp cát);

// vm.runInNewContext (someRandomCode, sandbox);

// ^

// ReferenceError: process không được xác định

Nhóm Node cốt lõi đã đánh dấu vm là không ổn định, có nghĩa là API và hành vi của nó có thể thay đổi bất kỳ lúc nào. Hai điểm khác cũng cần được nêu rõ:

Mô-đun này không đảm bảo một "nhà tù" hoàn toàn an toàn trong đó mã hoàn toàn không đáng tin cậy có thể được thực thi. Nếu đây là nhu cầu của bạn, hãy cân nhắc chạy một quy trình riêng biệt với các quyền cấp hệ thống thích hợp.

Bởi vì vm quay một phiên bản V8 mới, mỗi lệnh gọi tiêu tốn khoảng mili giây thời gian khởi động và khoảng hai megabyte bộ nhớ. Nó nên được sử dụng một cách tiết kiệm.

Với mục đích kiểm tra mã, vm mô-đun có thể khá hiệu quả, đặc biệt là ở khả năng buộc mã chạy trong một ngữ cảnh hạn chế. Ví dụ: khi thực hiện kiểm thử đơn vị, người ta có thể tạo một môi trường đặc biệt với dữ liệu giả lập mô phỏng môi trường mà trong đó tập lệnh được kiểm tra sẽ chạy. Điều này có thể tốt hơn việc tạo ngữ cảnh cuộc gọi nhân tạo với dữ liệu giả. Ngoài ra, hộp cát này sẽ cho phép bối cảnh thực thi cho mã mới được kiểm soát tốt hơn, cung cấp khả năng bảo vệ tốt khỏi rò rỉ bộ nhớ và các va chạm không lường trước khác có thể bong bóng trong khi thử nghiệm.

## 9.12. Phân biệt giữa phạm vi cục bộ và bối cảnh thực thi

Trước khi trình bày các ví dụ khác, chúng ta cần phân biệt giữa phạm vi cục bộ của một quy trình và bối cảnh thực thi của nó. Sự phân biệt sẽ giúp hiểu được sự khác biệt giữa hai vm phương pháp, vm.runInThisContext và vm.runInNewContext.

Bối cảnh thực thi của một quy trình Node đại diện cho ngữ cảnh thời gian chạy bên trong V8, bao gồm các phương thức Node gốc và các đối tượng toàn cục khác ( process, console, setTimeout, và như thế). Các thành viên của đối tượng này có thể dễ dàng theo dõi bằng cách sử dụng Node REPL:

nút> toàn cầu

Phạm vi cục bộ của chương trình Node chứa các định nghĩa thời gian chạy, chẳng hạn như các biến được xác định bằng cách sử dụng var.

Một tập lệnh được thực thi thông qua vm.runInNewContext không có khả năng hiển thị trong cả hai phạm vi— ngữ cảnh của nó bị giới hạn trong đối tượng hộp cát mà nó đã được chuyển đến, như đã thấy trước đó.

Một tập lệnh được thực thi thông qua vm.runInThisContext có khả năng hiển thị phạm vi thực thi toàn cục, nhưng không hiển thị phạm vi cục bộ. Chúng tôi có thể chứng minh điều này như sau:

var localVar = 123;

var tc = vm.runInThisContext ('localVar = 321;'); console.log (localVar, tc);

var ev = eval ('localVar = 321;'); console.log (localVar, ev); // 123 321

// 321 321

Do đó, các tập lệnh được chạy trong các ngữ cảnh thông qua vm. Nó thường hữu ích khi biên dịch trước các ngữ cảnh và tập lệnh, đặc biệt là khi chúng được sử dụng lặp đi lặp lại.

Sử dụng vm.createContext ([hộp cát]) để biên dịch ngữ cảnh thực thi và chuyển vào bản đồ khóa / giá trị, chẳng hạn như { a: 1, b: 2}. Bây giờ chúng ta sẽ xem xét cách sử dụng các đối tượng đã biên dịch này cùng nhau.

## 9.13. Sử dụng các ngữ cảnh đã biên dịch

Sau khi nhận được một chuỗi mã JavaScript, trình biên dịch V8 sẽ cố gắng hết sức để tối ưu hóa mã thành phiên bản đã biên dịch chạy hiệu quả hơn. Bước biên dịch này phải thực hiện mỗi lần vm phương thức ngữ cảnh nhận mã dưới dạng một chuỗi. Nếu mã không thay đổi và được sử dụng lại ít nhất một lần, hãy sử dụng vm.createScript

(mã, [tên tệp]) để biên dịch nó một lần và mãi mãi.

Đối tượng Script này cũng kế thừa runInThisContext và runInNewContext phương pháp từ vm. Ở đây, chúng tôi chạy một tập lệnh đã biên dịch trong cả hai ngữ cảnh, chứng minh cách x và y các biến đang được tăng dần tồn tại trong các phạm vi được cách ly hoàn toàn:

x = 0; y = 0;

var script = vm.createScript ('++ x, ++ y;');

var thi đua = vm.createContext ({x: 0, y: 0}); for (var i = 0; i <1000; i ++) {

script.runInThisContext ();

script.runInNewContext (giả lập);

}

console.log (x, y); console.log (thi đua.x, thi đua.y);

// 1000 1000 1000 // 1000 1000

1000

Cả hai tập lệnh đều được sửa đổi giống nhau x và y các biến, thì kết quả đầu ra thay vào đó sẽ là 2000 2000. Ngoài ra, nếu runInNewContext script không được thông qua một lớp mô phỏng (hộp cát) nó sẽ ném ReferenceError, không có quyền truy cập vào toàn cầu x và y các giá trị. Hãy thử nó ra.

Tùy chọn tên tập tin đối số được sử dụng để gỡ lỗi, xuất hiện trong dấu vết ngăn xếp. Nó có thể hữu ích để cung cấp một khi gỡ lỗi.

## 9.14. Lỗi và ngoại lệ

Các thuật ngữ "lỗi" và "ngoại lệ" thường được sử dụng thay thế cho nhau. Trong môi trường Node, hai khái niệm này không giống nhau. Lỗi và ngoại lệ là khác nhau. Ngoài ra, định nghĩa về lỗi và ngoại lệ trong Node không nhất thiết phải phù hợp với các định nghĩa tương tự trong các ngôn ngữ và môi trường phát triển khác.

Thông thường, điều kiện lỗi trong chương trình Node là điều kiện không nghiêm trọng cần được phát hiện và xử lý, được thấy rõ ràng nhất trong Lỗi ở đối số đầu tiên quy ước được hiển thị bởi mẫu gọi lại Node điển hình. Một ngoại lệ là một lỗi nghiêm trọng (lỗi hệ thống) mà một môi trường lành mạnh không nên bỏ qua hoặc cố gắng xử lý.

Người ta gặp phải ba bối cảnh lỗi phổ biến trong Node và sẽ phản hồi có thể đoán trước được:

Một bối cảnh đồng bộ: Điều này thường sẽ xảy ra trong bối cảnh một chức năng, nơi phát hiện một chữ ký cuộc gọi không hợp lệ hoặc một lỗi không nghiêm trọng khác. Hàm chỉ nên trả về một đối tượng lỗi; Lỗi mới (…) hoặc một số chỉ báo nhất quán khác cho thấy lệnh gọi hàm không thành công.

Ngữ cảnh không đồng bộ: Khi được mong đợi phản hồi bằng cách kích hoạt một hàm gọi lại, ngữ cảnh thực thi phải chuyển lỗi đối tượng, với thông báo thích hợp, làm đối số đầu tiên cho lệnh gọi lại đó.

Bối cảnh sự kiện: Trích dẫn tài liệu Node: "Khi một cá thể EventEmitter gặp lỗi, hành động điển hình là phát ra lỗi biến cố. Các sự kiện lỗi được coi như một trường hợp đặc biệt trong nút. Nếu không có trình lắng nghe nào cho nó, thì hành động mặc định là in một dấu vết ngăn xếp và thoát khỏi chương trình. "Sử dụng các sự kiện mà các sự kiện được mong đợi.

Rõ ràng, đây là ba tình huống mà lỗi được phát hiện một cách có kiểm soát, trước khi nó gây mất ổn định cho toàn bộ ứng dụng. Không quá sa đà vào mã hóa phòng thủ, nên cố gắng kiểm tra đầu vào và các nguồn khác để tìm lỗi và loại bỏ chúng đúng cách.

Một lợi ích bổ sung của việc luôn trả lại một lỗi đối tượng là quyền truy cập vào thuộc tính ngăn xếp của đối tượng đó. Ngăn xếp lỗi hiển thị nguồn gốc của một lỗi, mỗi liên kết trong chuỗi chức năng, gọi hàm dẫn đến lỗi. Một điển hình Error.stack dấu vết sẽ trông như thế này:

> console.log (Lỗi mới ("Thông báo Lỗi của Tôi"). stack); Lỗi: Thông báo lỗi của

tôi

tại Object. <anonymous> (/js/errorstack.js:1:75) tại Module.\_compile (module.js: 449: 26) tại Object.Module.\_extensions..js (module.js: 467: 10)

. . .

Tương tự, ngăn xếp luôn có sẵn thông qua console.trace phương pháp:

> console.trace ("Đầu ngăn xếp")

Dấu vết: Đầu ngăn xếp tại Object. <anonymous> (/js/stackhead.js:1:71) tại Module.\_compile (module.js: 449: 26) tại Object.Module.\_extensions..js (module.js: 467: 10)

. . .

Cần phải rõ thông tin này hỗ trợ gỡ lỗi như thế nào, giúp đảm bảo rằng luồng logic của ứng dụng của chúng ta là hợp lý.

Một dấu vết ngăn xếp bình thường bị cắt bớt sau một tá cấp độ. Nếu dấu vết ngăn xếp dài hơn hữu ích cho bạn, hãy thử Matt Insler's longjohn tại https://github.com/mattinsler/longjohn.

Đồng thời, chạy và kiểm tra js / stacktrace.js gửi vào gói của bạn một số ý tưởng về cách thông tin ngăn xếp có thể được sử dụng khi báo cáo lỗi hoặc thậm chí kết quả kiểm tra.

Xử lý ngoại lệ là khác nhau. Các trường hợp ngoại lệ là các lỗi không mong muốn hoặc nghiêm trọng đã làm mất ổn định ứng dụng. Chúng nên được xử lý cẩn thận; một hệ thống ở trạng thái ngoại lệ là không ổn định, với các trạng thái tương lai không xác định và nên được tắt và khởi động lại một cách duyên dáng. Đây là điều thông minh để làm.

Thông thường, các ngoại lệ được mắc trong các khối try / catch:

thử {something.that = wontWork; } catch (throwError) {

// làm gì đó với cú ném

}

Việc trang bị một cơ sở mã với các khối thử / bắt và cố gắng lường trước tất cả các lỗi có thể trở nên khó quản lý và khó sử dụng. Ngoài ra, điều gì sẽ xảy ra nếu một ngoại lệ mà bạn không lường trước được — một ngoại lệ không cần thận trọng — xảy ra? Làm thế nào để bạn tiếp tục nơi bạn đã dừng lại?

Node chưa có một cách tích hợp tốt để xử lý các trường hợp ngoại lệ quan trọng chưa được giải quyết. Đây là một điểm yếu của nền tảng. Một ngoại lệ chưa được giải quyết sẽ tiếp tục nổi lên thông qua ngăn xếp thực thi cho đến khi nó chạm vào vòng lặp sự kiện, nơi, giống như cờ lê trong bánh răng của máy, nó sẽ phá hủy toàn bộ quá trình.

Tốt nhất chúng tôi có là đính kèm uncaughtException xử lý chính quy trình:

process.on ('uncaughtException', function (err) { console.log ('Bắt ngoại lệ:' + err); });

setTimeout (function () { console.log ("Đã bắt được ngoại lệ và điều này có thể chạy."); }, 1000);

némAnUncaughtException ();

//> Bắt ngoại lệ: ReferenceError: throwAnUncaughtException không được xác định

//> Đã bắt được ngoại lệ và điều này có thể chạy.

Mặc dù không có gì theo sau mã ngoại lệ của chúng tôi sẽ thực thi, thời gian chờ vẫn sẽ kích hoạt, vì quá trình quản lý để bắt ngoại lệ, tự lưu. Tuy nhiên, đây là một cách xử lý ngoại lệ rất vụng về.

Đã thêm gần đây miền mô-đun đã cố gắng sửa lỗ hổng này trong thiết kế của Node. Chúng ta sẽ thảo luận về mô-đun miền tiếp theo như một công cụ tốt hơn để xử lý các ngoại lệ.

## 9.15. Mô-đun miền

Việc xử lý lỗi trong mã không đồng bộ cũng khó theo dõi:

hàm f () { tạo ra lỗi mới ("error somwhere!")

}

setTimeout (f, 1000 \* Math.random ()); setTimeout (f, 1000 \* Math.random ());

Chức năng nào gây ra lỗi? Rất khó để nói. Việc chèn các công cụ quản lý ngoại lệ một cách thông minh cũng rất khó. Rất khó để biết phải làm gì tiếp theo. Điểm giao

miền mô-đun cố gắng trợ giúp vấn đề này và các vấn đề bản địa hóa ngoại lệ khác. Bằng cách này, mã có thể được kiểm tra và xử lý lỗi với độ chính xác cao hơn.

Đơn giản nhất, miền thiết lập một ngữ cảnh trong đó một hàm hoặc một đoạn mã khác có thể được chạy sao cho bất kỳ lỗi nào xảy ra trong liên kết miền ngầm định đó sẽ được chuyển đến một trình xử lý lỗi miền cụ thể. Ví dụ:

var dom = domain.create (); dom.on ('error', function

(err) { console.error ('error', err.stack);

});

dom.run (function () { ném Lỗi mới ("lỗi miền của tôi"); });

// error Error: my domain error // at

/js/basicdomain.js:10:8

// ...

Ở đây, một hàm thông báo lỗi được chạy trong ngữ cảnh của một miền có thể

để bắt các ngoại lệ đó một cách thông minh, ràng buộc ngầm tất cả các trình phát sự kiện, bộ định thời và các yêu cầu khác được tạo trong ngữ cảnh đó.

Đôi khi một phương thức có thể được tạo ở nơi khác (không nằm trong ngữ cảnh ngầm định của một domain.run) nhưng vẫn được kết hợp tốt nhất với miền bên ngoài. Các thêm vào phương thức tồn tại chỉ cho ràng buộc rõ ràng như vậy:

var dom = domain.create (); dom.on ("error", function

(err) { console.log (err);

});

var somefunc = function () { ném Lỗi mới ('Lỗi ràng buộc rõ ràng');

}

dom.add (somefunc); dom.run (function () { somefunc ();

});

// [Lỗi: Lỗi liên kết rõ ràng]

Ở đây chúng ta thấy cách một hàm không bị ràng buộc ngầm trong chạy ngữ cảnh vẫn có thể được thêm vào ngữ cảnh đó một cách rõ ràng. Để xóa ngữ cảnh thực thi khỏi miền, hãy sử dụng

domain.remove. Một mảng của tất cả các bộ định thời, hàm và các bộ phát khác được thêm một cách rõ ràng hoặc ngầm định vào miền có thể truy cập được qua domain.members.

Cách mà JavaScript của trói buộc phương thức liên kết một hàm với một ngữ cảnh, tương tự như domain.bind phương thức cho phép một hàm độc lập được liên kết với một miền:

var dom = domain.create (); dom.on ("lỗi",

...);

fs.readFile ('somefile', dom.bind (function (err, data) { if (err) {throw new Error ('bad file call'); }}));

// {[Error: bad call] // domain\_thrown:

true,

// ...

Ở đây chúng ta thấy cách bất kỳ chức năng nào có thể được bao bọc bởi một miền lỗi cụ thể "nội tuyến", một tính năng đặc biệt hữu ích để quản lý các ngoại lệ trong các lệnh gọi lại. Lưu ý rằng các đối tượng Lỗi được xử lý thông qua một miền có các thuộc tính đặc biệt được thêm vào chúng:

lỗi. miền: Miền đã xử lý lỗi.

error.domainEmitter: Nếu EventEmitter kích hoạt một sự kiện lỗi trong một miền, chức năng này sẽ được gắn cờ.

lỗi. tên miền Lệnh gọi lại đã chuyển lỗi làm đối số đầu tiên của nó.

error.domainThrown: Một Boolean cho biết lỗi có được ném ra hay không. Các lỗi được chuyển cho các lệnh gọi lại hoặc các lỗi được EventEmitter thông báo không đánh dấu cờ này.

Một phương pháp khác, domain.intercept, chức năng tương tự như miền. trói buộc, nhưng đơn giản hóa việc xử lý lỗi trong các lệnh gọi lại, như vậy nhà phát triển sẽ không cần phải kiểm tra lặp lại (hoặc thậm chí đặt) đối số đầu tiên của mọi lệnh gọi lại cb (err, dữ liệu) vì lỗi. Một ví dụ được tìm thấy trong js / domainintercept.js tệp trong gói mã của bạn.

Chúng tôi có thể muốn kiểm soát khi miền đang hoạt động, thoát khỏi miền một cách có trách nhiệm khi mã của chúng tôi đang chuyển sang một ngăn xếp thực thi khác (có thể với các miền riêng của nó), có thể nhập lại miền gốc sau đó. Đối với điều này, chúng tôi sử dụng miền. đi vào và domain.exit các phương pháp. Giả sử chúng tôi đã thiết lập hai miền dom1 và dom2, lần đầu tiên phát ra lỗi miền 1 va thu hai lỗi miền 2, chúng ta có thể di chuyển giữa các ngữ cảnh miền như sau:

dom1.add (aFuncThatThrows); dom1.run (function () { dom1.exit (); dom2.enter ();

aFuncThatThrows ();

});

// lỗi miền 2

Bất kỳ số đi vào và lối ra sự kiện có thể được sử dụng. Lưu ý rằng không có thay đổi nào được thực hiện đối với chính các đối tượng miền, vì thoát không đóng miền hoặc bất kỳ thứ nào tương tự. Nếu một miền cần bị hủy, bạn nên sử dụng domain.dispose phương pháp này cũng sẽ cố gắng dọn dẹp mọi I / O miền trong chuyến bay, hủy luồng, xóa bộ hẹn giờ, bỏ qua lệnh gọi lại, v.v.

## 9.16. Thử nghiệm trang web không cần đầu với ZombieJS và Mocha

Mặc dù chúng tôi đang làm việc trên máy chủ với Node, chắc chắn rằng phần lớn những gì chúng tôi đang lập trình sẽ ảnh hưởng đến những gì chúng tôi đang kết nối với dịch vụ của chúng tôi thông qua trình duyệt. Vì dự kiến rằng chúng tôi sẽ sử dụng Node để kiểm tra mã phía máy chủ của mình, tại sao không tích hợp cả kiểm tra phía trình duyệt?

Chúng ta sẽ tìm hiểu sâu hơn về cách thống nhất toàn bộ chuỗi thực hiện / xây dựng / thử nghiệm / triển khai trong phần tiếp theo. Bây giờ, chúng ta hãy xem xét khung kiểm tra Mocha dễ sử dụng và cách một số kiểm tra trình duyệt đơn giản có thể được thực hiện ngay từ máy chủ.

Các thư viện chúng tôi sẽ sử dụng trong phần này và phần tiếp theo có sẵn thông qua các nguồn sau:

Mocha: http://visionmedia.github.io/mocha/

Tiếng càu nhàu: http://www.gruntjs.com/

ZombieJS: http://zombie.labnotes.org/

PhantomJS: http://www.phantomjs.org/

## 9.17. Mocha

Mocha mong muốn một thư mục chứa các đơn vị có thể kiểm tra sẽ chứa một / kiểm tra thư mục, trong đó tồn tại nhiều thông số kỹ thuật hoặc đặc điểm kỹ thuật kiểm tra và các tệp, tạo nên bộ kiểm tra mà Mocha sẽ áp dụng cho ứng dụng của bạn. Bạn sẽ mô tả những gì bạn đang thử nghiệm và Mocha, một cách thông minh, cung cấp diễn tả phương pháp để làm điều đó:

var khẳng định = request ("khẳng định") description

('Mảng', function () { description ('# indexOf ()', function () { it ('nên trả về -1 khi không có giá trị', function () {

khẳng định.equal (-1, [1,2,3] .indexOf (5)); khẳng định.equal (-1, [1,2,3] .indexOf (0));

})

})

})

Ở đây chúng tôi đang mô tả một bài kiểm tra kiểm tra các hoạt động của mảng. Các diễn tả và nó chỉ thị chỉ đơn giản là công cụ tổ chức, cung cấp tiêu đề cho bộ thử nghiệm đang được mô tả (# Chỉ số()), và cung cấp mô tả về từng bài kiểm tra trong bộ ( nên trả về -1…). Nếu bạn chạy điều này thông qua Mocha, một số báo cáo thử nghiệm được định dạng gọn gàng với các mô tả này sẽ được ghi vào bảng điều khiển và dưới mỗi trạng thái đạt / không đạt của thử nghiệm.

Chúng ta đã thấy trong ví dụ trước về cách Node khẳng định mô-đun đang được sử dụng. Mocha không phải là một thư viện xác nhận, nó là một trình chạy thử nghiệm giúp việc sử dụng thư viện xác nhận dễ dàng hơn. Bạn không cần sử dụng thư viện của Node; các thư viện xác nhận phổ biến khác là Chai ( http://chaijs.com/api/assert/) và, bởi nhà sản xuất Mocha, Should ( https://github.com/visionmedia/should.js/).

Mô tả đầy đủ về Mocha nằm ngoài phạm vi của cuốn sách này. Mô tả trước đó phải đủ để hiểu cách thức hoạt động của các thông số kỹ thuật kiểm tra trong các ví dụ sau. Bạn được khuyến khích truy cập trang web Mocha và tự xem qua các ví dụ.

## 9.18. Kiểm tra web không đầu

Một cách để kiểm tra xem giao diện người dùng có hoạt động hay không là trả tiền cho một số người để tương tác với một trang web thông qua trình duyệt và báo cáo bất kỳ lỗi nào họ tìm thấy. Điều này tốn kém, không hiệu quả và cung cấp phạm vi bảo hiểm không đầy đủ, vì không có gì đảm bảo rằng tất cả các lỗi sẽ được tìm thấy. Tự động hóa quá trình kiểm tra giao diện người dùng bằng các công cụ mô phỏng trình duyệt là chủ đề của phần này.

Một trình duyệt, bị loại bỏ các nút và các điều khiển khác, về cơ bản là một chương trình xác thực và chạy JavaScript, HTML và CSS. Việc HTML đã xác thực được hiển thị trực quan trên màn hình của bạn chỉ đơn giản là hệ quả của việc con người chỉ có thể nhìn bằng mắt. Máy chủ có thể giải thích logic của mã đã biên dịch và xem kết quả tương tác của bạn với mã đó mà không cần thành phần trực quan. Có lẽ vì đôi mắt thường được tìm thấy trong đầu của một người, trình duyệt chạy trên máy chủ thường được gọi là trình duyệt không có đầu. Chúng ta sẽ tìm hiểu cách thực hiện kiểm tra trình duyệt không đầu, trước tiên với ZombieJS (trình duyệt không đầu được xây dựng tùy chỉnh) và sau đó với PhantomJS (phiên bản không đầu của công cụ WebKit).

Hầu hết các trang web sẽ sử dụng biểu mẫu theo một cách nào đó và nhiệm vụ phổ biến của các nhà phát triển là xác thực đầu vào của biểu mẫu, chẳng hạn như người dùng được cảnh báo về mật khẩu không khớp hoặc e-mail không hợp lệ trước khi gửi biểu mẫu. Hãy xem xét biểu mẫu sau:

<input type = "text" id = "email" size = "20" />

<input type = "password" id = "password" size = "20" />

<input type = "submit" value = "Đăng nhập" onclick = "return checkForm (this.parentNode) "/>

Đây rõ ràng là một hình thức đăng nhập, yêu cầu địa chỉ e-mail và mật khẩu. Chúng tôi cũng thấy rằng nút gửi của biểu mẫu này sẽ ủy quyền xác thực cho một số JavaScript, cuối cùng ngăn việc gửi nếu giá trị biểu mẫu không hợp lệ. Làm cách nào để chúng tôi kiểm tra xem biểu mẫu này có đang xác thực chính xác dữ liệu biểu mẫu thông qua Node đang chạy trên máy chủ hay không?

Xin vui lòng tham khảo mocha-zombie thư mục trong gói mã của bạn khi làm việc trong ví dụ này. Để bắt đầu, hãy chạy cài đặt npm trong thư mục này. Tham khảo ý kiến readme.md để được hướng dẫn thêm.

Giả sử rằng tệp HTML trước đó có sẵn tại localhost: 8000, chúng tôi có thể kiểm tra nó chỉ bằng cách sử dụng ZombieJS và Node's khẳng định mô-đun, triển khai khung thử nghiệm của riêng chúng tôi:

var zombie = request ("zombie"); var khẳng định = yêu cầu ("khẳng định"); var count = 0;

var testFramework = function (meth, v1, v2) {

+ + đếm; thử { khẳng định [meth] (v1, v2);

console.log ("Kiểm tra" + count + "OK!"); } bắt (e) { console.log ("Kiểm tra" + count + "Lỗi!"); console.log (e);

}

}

browser = new zombie ();

browser.visit ("http: // localhost: 8000 / login.html", function () { testFramework ("ok", browser.success);

testFramework ("ok", browser.query ('# nonexistingId'));

});

// Kiểm tra 1 OK!

// Kiểm tra 2 Lỗi!

// {name: 'AssertionError',

// thực tế: không xác định,

// mong đợi: đúng,

// toán tử: '==',

// tin nhắn: '"undefined" == true'}

Ở đây chúng ta thấy thuộc tính đối tượng ZombieJS sự thành công đang được kiểm tra (có tìm thấy trang không?) và truy vấn DOM cho phần tử có # không tồn tạiId đang được chạy trên DOM không đầu mới của chúng tôi, thử nghiệm đầu tiên thành công và thử nghiệm thứ hai không thành công. Khung thử nghiệm giả của chúng tôi chỉ cần bắt lỗi xác nhận và báo cáo kết quả.

Tham khảo trang web ZombieJS để biết đầy đủ các phương pháp có sẵn để kiểm tra DOM theo những cách phức tạp hơn nhiều.

Mocha là một khung thử nghiệm tốt hơn nhiều so với của chúng tôi, vì vậy hãy tích hợp Mocha và ZombieJS để kiểm tra xem biểu mẫu đăng nhập của chúng tôi có cả trường e-mail và mật khẩu hay không:

var server = someServerRunningOn8000; description ('Kiểm tra Đăng nhập Đơn giản', function () { this.browser = new zombie ({site: 'http: // localhost: 8000'}); this.browser.visit ('/ login.html');

it ('phải có đầu vào với id là #email', function () { khẳng định.ok (this.browser.success);

khẳng định.ok (this.browser.query ('input # email'));

});

it ('nên có đầu vào với id là #password', function () { khẳng định.ok (this.browser.success);

khẳng định.ok (this.browser.query ('input # password'));

});

});

Sau khi gọi mocha trên dòng lệnh, bạn sẽ thấy một cái gì đó như thế này được hiển thị:



Định cấu hình Mocha với /

test / mocha.opts

tập tin,

một ví dụ nằm trong gói mã của bạn. Để biết thêm thông tin, hãy truy cập

http://visionmedia.github.io/mocha/#mocha.opts.

Ví dụ đi kèm chứa một tệp thử nghiệm phức tạp hơn, nhưng chúng tôi chỉ đang sơ lược ở đây. Có nhiều cách để suy nghĩ và thiết kế các bài kiểm tra, vì vậy đây chỉ là điểm khởi đầu khi bạn học cách mô tả ứng dụng của mình bằng cách sử dụng các xác nhận logic và tìm hiểu sâu hơn về Mocha và các thư viện xác nhận khác nhau.

## 9.19. Sử dụng Grunt, Mocha và PhantomJS để kiểm tra và triển khai các dự án

Với kiến thức mới của chúng tôi về các trình duyệt không đầu và khung thử nghiệm, hãy phát triển một hệ thống xây dựng / thử nghiệm / triển khai thực tế hơn một chút bằng cách sử dụng Node. Chúng tôi sẽ tiếp tục sử dụng Mocha và hoán đổi ZombieJS cho PhantomJS vững chắc, không giống như ZombieJS không phải là trình giả lập, nó là một trình duyệt WebKit thực sự, làm cho nó trở thành một môi trường thử nghiệm mạnh mẽ và chính xác hơn.

Chúng tôi sẽ giới thiệu Grunt, JavaScript Test Runner, được thiết kế để tự động hóa tác vụ, chẳng hạn như tự động chạy các bài kiểm tra Mocha và triển khai bản dựng hiện tại, nếu tất cả các bài kiểm tra đó đều vượt qua.

Mục tiêu là tạo ra một hệ thống cho phép các nhà phát triển viết các trang HTML đơn giản và có Grunt để tự động kiểm tra và triển khai chúng. Nó là để biến mã nguồn này:

<head>

<! - build: css: deploy css / style.min.css ->

<link rel = "stylesheet" href = "css / normalize.css">

<link rel = "stylesheet" href = "css / main.css">

<! - / build ->

</head>

<body>

<! - build: js: deploy js / app.min.js ->

<script src = "js / file1.js"> </script>

<script src = "js / file2.js"> </script>

<script src = "js / file3.js"> </script>

<! - / build ->

<! - build: remove: deploy ->

<script src = "../../ test / framework.js"> </script>

<script src = "../../ test / spec / file1.js"> </script>

<script src = "../../ test / spec / file2.js"> </script>

<script src = "../../ test / spec / file3.js"> </script>

<! - / build ->

</body>

Vào mã đã triển khai này:

<head>

<link rel = "stylesheet" href = "css / style.min.css">

</head>

<body>

<script src = "js / app.min.js"> </script>

</body>

Đây là nơi mà quá trình kiểm tra và rút gọn mã được thực hiện tự động, tuân theo các chỉ thị xây dựng xung quanh các đoạn liên kết và tập lệnh.

Vào cuối quá trình này, một / xây dựng thư mục sẽ được tạo, chứa cả sân khấu/ và một triển khai / thư mục, mỗi thư mục chứa các bản dựng cụ thể của các tệp trong bản gốc / nguồn danh mục. Đây là một quy trình công việc đầy đủ có thể giúp bạn kiểm tra và triển khai các dự án của mình.

## 9.20. Làm việc với Grunt

Quy trình làm việc hiện tại mà Grunt cố gắng đơn giản hóa là xoay quanh Unix làm tiện ích. Một điển hình làm tệp để tự động chạy Mocha sẽ được đặt tên Makefile và trông giống như sau:

TESTS = test / \*. Js

kiểm tra:

mocha

. PHONY: thử nghiệm

Nó hơi khó hiểu, nhưng các ý tưởng rất đơn giản:

Thiết lập một người chạy thử nghiệm sẽ chạy lệnh mocha.

Yêu cầu người chạy đó tìm kiếm các tệp đặc tả JavaScript trong thư mục kiểm tra/.

Grunt cố gắng hoàn thành các mục tiêu tương tự trong môi trường Node, nơi việc gọi các hàm mô-đun có ý nghĩa hơn việc chạy các lệnh hệ thống và sử dụng cú pháp JSON tiêu chuẩn để khai báo các phụ thuộc hệ thống có ý nghĩa hơn.

Các nhiệm vụ Grunt được viết cho một Gruntfile.js, và một để chạy thử nghiệm Mocha có thể trông như thế này:

module.exports = function (grunt) { grunt.initConfig ({ mocha: {

tất cả: ['./test/\*.js']

}

});

grunt.loadNpmTasks ('grunt-mocha');

grunt.registerTask ('default', ['mocha']); }

Thêm nhiều tác vụ theo cùng một mô hình. Để thêm jshint tác vụ sẽ kiểm tra mã nguồn JavaScript của bạn để tìm các lỗi có thể xảy ra, chúng tôi thêm một số thứ như sau:

. . .

jshint: { tùy chọn: { jshintrc: ".jshintrc"

},

các tệp: ["Gruntfile.js", "task / \*. js"]

}

. . .

this.loadNpmTasks ("grunt-Contrib-jshint");

. . .

## 9.21. Kiểm tra ứng dụng của bạn

Hãy tiếp tục và khám phá mocha-phantom-grunt thư mục trong gói mã của bạn. Bạn có thể bắt đầu bằng cách cài đặt nó và chạy tiếng càu nhàu:

> cài đặt npm

. . .

> càu nhàu

Đọc qua đầu ra và xem liệu bạn có thể theo dõi những gì đang diễn ra không. Sau đó, đi sâu vào mã và làm theo cách Grunt thiết lập các trình chạy tác vụ mà cuối cùng sẽ xác thực, kiểm tra và triển khai dự án của bạn một cách dễ dàng.

## Tóm lược

Cộng đồng Node đã chấp nhận thử nghiệm ngay từ đầu và nhiều khung thử nghiệm và công cụ gốc được cung cấp cho các nhà phát triển. Trong chương này, chúng ta đã xem xét lý do tại sao kiểm thử lại quan trọng đối với sự phát triển phần mềm hiện đại, cũng như điều gì đó về kiểm thử chức năng, đơn vị và tích hợp, chúng là gì và cách chúng có thể được sử dụng. Với vm mô-đun chúng tôi đã học cách tạo ngữ cảnh đặc biệt để thử nghiệm các chương trình JavaScript, đồng thời chọn một số kỹ thuật để tạo mã không đáng tin cậy trong hộp cát.

Ngoài ra, chúng tôi đã học cách làm việc với bộ công cụ xử lý lỗi và kiểm tra Node mở rộng, từ ghi nhật ký bảng điều khiển biểu cảm hơn đến Di chuyển lên một dòng theo dõi và gỡ lỗi. Thông qua khẳng định và miền các mô-đun mà chúng tôi đã tiếp xúc với các cơ chế báo cáo và bắt lỗi, cho chúng tôi một cách mạnh mẽ hơn để nhắm mục tiêu cụ thể các lỗi và ngoại lệ, thay vì chỉ bắt tất cả trong một trình xử lý chung.

Cuối cùng, chúng tôi đã học cách thiết lập một hệ thống xây dựng và kiểm tra thích hợp bằng cách sử dụng Grunt và Mocha, trong quá trình thử nghiệm với hai thư viện thử nghiệm trình duyệt không đầu khác nhau, tìm hiểu hai cách mà thử nghiệm đó có thể được thực hiện trong mỗi cách và cách các trình duyệt ảo này có thể được tích hợp hoàn toàn với các môi trường thử nghiệm khác.

# PHỤ LỤC A

# Tổ chức công việc của bạn

"Cách để xây dựng một hệ thống phức tạp hoạt động là xây dựng nó từ những hệ thống rất đơn giản hoạt động."

- Kevin Kelly

Hệ thống quản lý mô-đun đơn giản của Node khuyến khích phát triển các cơ sở mã có thể bảo trì. Nhà phát triển Node may mắn có một hệ sinh thái phong phú gồm các gói được xác định rõ ràng với các giao diện nhất quán dễ kết hợp, thường được phân phối qua npm. Khi phát triển các giải pháp cả đơn giản và phức tạp, nhà phát triển Node thường sẽ tìm thấy nhiều phần của hệ thống đó đã được làm sẵn và có thể nhanh chóng biên soạn các mô-đun nguồn mở đó thành các hệ thống lớn hơn, nhất quán và có thể dự đoán được.

Vì giá trị của bất kỳ mô-đun nào có thể được nhân lên bằng cách thêm các mô-đun con hữu ích, hệ sinh thái mô-đun Node đã phát triển nhanh chóng.

Chúng tôi sẽ trình bày chi tiết về cách Node hiểu mô-đun và đường dẫn mô-đun, cách mô-đun được định nghĩa, cách sử dụng mô-đun trong kho lưu trữ gói npm và cách tạo và chia sẻ npm mới. Bằng cách tuân theo một số quy tắc đơn giản, bạn sẽ thấy dễ dàng định hình cấu trúc ứng dụng của mình và giúp những người khác làm việc với những gì bạn đã tạo.

"Mô-đun" và "gói" sẽ được sử dụng thay thế cho nhau để mô tả tệp hoặc bộ sưu tập tệp được biên dịch và trả lại bởi yêu cầu ().

## 1. Đang tải và sử dụng mô-đun

Các nhà thiết kế Node tin rằng hầu hết các mô-đun nên được phát triển trong vùng người dùng — bởi các nhà phát triển, cho các nhà phát triển. Một nỗ lực như vậy được thực hiện để hạn chế sự phát triển của thư viện tiêu chuẩn. Thư viện tiêu chuẩn của Node chứa danh sách ngắn các mô-đun sau:

|  |  |  |
| --- | --- | --- |
| Mạng và I / O | Chuỗi và đệm | Tiện ích |
| TTY  UDP / Datagram  HTTP HTTPS  Mạng lưới  DNS  TLS / SSL  Đường đọc  Hệ thống tập tin | Con đường  Đệm  Url  StringDecoder Chuỗi truy vấn | Tiện ích  VM  Đường đọc Miền  Bảng điều khiển  Khẳng định |
| Mã hóa và nén | Môi trường | Sự kiện và Luồng |
| ZLIB  Tiền điện tử  PunyCode | Quá trình  Hệ điều hành  Mô-đun | Quy trình con  Cụm  Sự kiện Suối |

Các mô-đun được tải qua toàn cầu yêu cầu câu lệnh chấp nhận tên mô-đun hoặc đường dẫn như một đối số duy nhất. Bạn được khuyến khích tăng cường hệ sinh thái mô-đun bằng cách tạo mô-đun mới hoặc kết hợp mô-đun mới.

Bản thân hệ thống mô-đun được triển khai trong yêu cầu (mô-đun) mô-đun.

## Hiểu đối tượng mô-đun

Mô-đun Node chỉ đơn giản là một tệp JavaScript được mong đợi để gán một giá trị hữu ích cho các module.exports bất động sản:

module.exports = new function () { this.it = function (it) { console.log (nó);

}

}

Bây giờ chúng tôi có một mô-đun có thể được yêu cầu bởi một tệp khác. Nếu mô-đun này đã được xác định trong tệp ./ say.js, sau đây là một cách để sử dụng nó:

var say = request ("./ say"); say.it ("Xin chào");

// Xin chào

Lưu ý rằng nó không cần thiết để sử dụng. js hậu tố. Chúng ta sẽ thảo luận về cách Node giải quyết các đường dẫn ngay sau đây.

Các mô-đun cốt lõi của Node cũng được xác định bằng cách sử dụng tiêu chuẩn mô-đun. hàng xuất khẩu mẫu, có thể thấy bằng cách duyệt qua mã nguồn xác định bảng điều khiển: https://github.com/joyent/node/blob/ master / lib / console.js.

Sau khi tải, các mô-đun được lưu vào bộ nhớ đệm dựa trên tên tệp đã phân giải của chúng, được giải quyết liên quan đến mô-đun gọi. Các cuộc gọi tiếp theo tới yêu cầu (./ myModule) sẽ trả về các đối tượng giống hệt nhau (được lưu trong bộ nhớ đệm). Tuy nhiên, lưu ý rằng việc truy cập vào cùng một mô-đun thông qua một đường dẫn tương đối khác (chẳng hạn như ../../ myModule) sẽ trả về một đối tượng khác — hãy nghĩ đến việc bộ nhớ đệm được khóa bởi các đường dẫn mô-đun tương đối.

Ảnh chụp nhanh của bộ nhớ cache hiện tại có thể được tìm nạp qua request ('module') .\_ cache).

Bản thân đối tượng mô-đun chứa một số thuộc tính hữu ích có thể đọc được, chẳng hạn như:

module.filename: Tên của tệp xác định mô-đun này.

module.loaded: Cho dù mô-đun đang trong quá trình tải. Boolean true nếu được tải.

module.parent: Mô-đun yêu cầu mô-đun này, nếu có.

module.children: Các mô-đun yêu cầu của mô-đun này, nếu có.

Bạn có thể xác định xem một mô-đun đang được thực thi trực tiếp thông qua node module.js hoặc thông qua request ('./ module.js') bởi kiểm tra nếu mô-đun request.main ===, cái nào sẽ trở lại

thật trong trường hợp trước đây.

## Giải quyết các đường dẫn mô-đun

Các yêu cầu câu lệnh thường được nhìn thấy khi duyệt (hoặc xây dựng) một chương trình Node. Bạn sẽ nhận thấy rằng đối số được truyền cho request có thể có nhiều dạng, chẳng hạn như tên của mô-đun lõi hoặc đường dẫn tệp.

Mã giả sau, được lấy từ tài liệu Node, là mô tả có thứ tự về các bước được thực hiện khi giải quyết

đường dẫn mô-đun:

require(X) from module at path Y

If X is a core module,

return the core module

STOP

If X begins with './' or '/' or '../'

LOAD\_AS\_FILE(Y + X)

LOAD\_AS\_DIRECTORY(Y + X)

LOAD\_NODE\_MODULES(X, dirname(Y))

THROW "not found"LOAD\_AS\_FILE(X)

If X is a file, load X as JavaScript text. STOP

If X.js is a file, load X.js as JavaScript text. STOP

If X.node is a file, load X.node as binary addon. STOPLOAD\_AS\_DIRECTORY(X)

If X/package.json is a file,

Parse X/package.json, and look for "main" field.

let M = X + (json main field)

LOAD\_AS\_FILE(M)

If X/index.js is a file, load X/index.js as JavaScript text. STOP

If X/index.node is a file, load X/index.node as binary addon. STOPLOAD\_NODE\_MODULES(X, START)

let DIRS=NODE\_MODULES\_PATHS(START)

for each DIR in DIRS:

LOAD\_AS\_FILE(DIR/X)

LOAD\_AS\_DIRECTORY(DIR/X)

NODE\_MODULES\_PATHS(START)

let PARTS = path split(START)

let ROOT = index of first instance of "node\_modules" in PARTS, or 0

let I = count of PARTS - 1

let DIRS = []

while I > ROOT,

if PARTS[I] = "node\_modules" CONTINUE

c. DIR = path join(PARTS[0 .. I] + "node\_modules")

DIRS = DIRS + DIR

let I = I - 16. return DIRS

/user/home/sandro/node\_modules/library.js /user/home/node\_modules/library.js /user/node\_modules/library.js /node\_modules/library.js

Tổ chức các tệp và / hoặc mô-đun của bạn thành các thư mục luôn là một ý tưởng hay. Một cách hữu ích, Node cho phép các mô-đun được tham chiếu thông qua thư mục chứa chúng, theo hai cách. Với một thư mục, Node trước tiên sẽ cố gắng tìm một package.json tập tin trong thư mục đó, cách khác là tìm kiếm index.js tập tin. Chúng tôi sẽ thảo luận về việc sử dụng gói hàng. json các tập tin trong phần tiếp theo. Ở đây chúng ta cần chỉ ra rằng nếu yêu cầu được thông qua thư mục ./ myModule nó sẽ tìm kiếm là ./ myModule / index.js.

Nếu bạn đã đặt NODE\_PATH biến môi trường thì Node sẽ sử dụng thông tin đường dẫn đó để thực hiện các tìm kiếm tiếp theo nếu không tìm thấy mô-đun được yêu cầu qua các kênh bình thường. Vì lý do lịch sử $ TRANG CHỦ / .node\_

mô-đun, $ HOME / .node\_libraries, và $ PREFIX / lib / nút cũng sẽ được tìm kiếm. $ TRANG CHỦ đại diện cho thư mục chính của người dùng và $ PREFIX thường sẽ là vị trí Node được cài đặt.

## Sử dụng npm

Như đã đề cập khi thảo luận về cách Node thực hiện tra cứu đường dẫn, các mô-đun có thể được chứa trong một thư mục. Nếu bạn đang phát triển một chương trình dưới dạng mô-đun cho những người khác

để sử dụng, bạn nên gói mô-đun đó trong thư mục riêng của nó và xuất bản nó. Hệ thống quản lý gói npm được thiết kế để giúp bạn làm điều đó.

Như chúng ta đã thấy trong suốt các ví dụ trong cuốn sách này, package.json tệp mô tả mô-đun, ghi lại hữu ích tên mô-đun, số phiên bản, phụ thuộc, v.v. Nó phải tồn tại nếu bạn muốn xuất bản gói của mình qua npm. Trong phần này, chúng tôi sẽ điều tra các thuộc tính chính của tệp này, đồng thời đưa ra một số gợi ý và mẹo về cách định cấu hình và phân phối các mô-đun của bạn.

Thử npm giúp json để tìm nạp tài liệu chi tiết cho tất cả package.json cánh đồng, hoặc thăm https://npmjs.org/doc/json.html.

A package.json tệp phải tuân theo đặc tả JSON. Ví dụ, các thuộc tính và giá trị phải được đặt trong dấu ngoặc kép.

## Khởi tạo một tệp gói

Bạn có thể tạo tệp gói bằng tay hoặc sử dụng tiện ích npm init công cụ dòng lệnh, sẽ nhắc bạn về một loạt giá trị và tự động tạo package.json tập tin. Hãy chạy qua một số giá trị sau:

Tên: ( Bắt buộc) Chuỗi này là chuỗi sẽ được chuyển đến yêu cầu () theo thứ tự để tải mô-đun của bạn. Làm cho nó ngắn gọn và mang tính mô tả, chỉ sử dụng các ký tự chữ và số — tên này sẽ được sử dụng trong URL, đối số dòng lệnh và tên thư mục. Cố gắng tránh sử dụng "js" hoặc "nút" trong tên.

phiên bản: ( Bắt buộc) npm sử dụng lập phiên bản ngữ nghĩa, trong đó tất cả chúng đều hợp lệ:

° > = 1,0,2 <2,1,2

° 2.1.x

° ~ 1,2

Để biết thêm thông tin về số phiên bản, hãy truy cập https://npmjs.org/ doc / misc / semver.html.

sự miêu tả: Khi mọi người tìm kiếm npmjs.org đối với các gói, đây là những gì họ sẽ đọc. Làm cho nó ngắn gọn và mang tính mô tả.

điểm vào: Đây là tệp nên đặt module.exports —Nó xác định vị trí của định nghĩa đối tượng mô-đun.

từ khóa: Danh sách từ khóa được phân tách bằng dấu phẩy sẽ giúp những người khác tìm thấy mô-đun của bạn trong sổ đăng ký.

giấy phép: Node là một cộng đồng mở thích các giấy phép dễ dãi. "MIT" và "BSD" là những thứ tốt ở đây.

Bạn cũng có thể muốn đặt riêng tư lĩnh vực để thật trong khi bạn đang phát triển mô-đun của mình. Điều này đảm bảo rằng npm sẽ từ chối xuất bản nó, tránh việc vô tình phát hành mã không đầy đủ hoặc nhạy cảm về thời gian.

Sử dụng tập lệnh

npm cuối cùng là một công cụ xây dựng và tập lệnh trường trong tệp gói của bạn cho phép bạn thiết lập các lệnh xây dựng khác nhau được thực thi tại một số điểm sau một số npm

các lệnh. Ví dụ: bạn có thể muốn giảm thiểu JavaScript hoặc thực thi một số quy trình khác để xây dựng các phụ thuộc mà mô-đun của bạn sẽ cần bất cứ khi nào cài đặt npm được thực thi. Các chỉ thị có sẵn là:

xuất bản trước, xuất bản, xuất bản sau: Điều hành bởi npm xuất bản chỉ huy.

cài đặt trước, cài đặt, postinstall: Chạy bởi cài đặt npm chỉ huy.

cài đặt trước, gỡ cài đặt, sau gỡ cài đặt: Chạy bởi gỡ cài đặt npm chỉ huy.

cập nhật trước, cập nhật, cập nhật sau: Chạy bởi cập nhật npm chỉ huy.

đẹp nhất, kiểm tra, hậu kiểm: Chạy bởi kiểm tra npm chỉ huy.

prestop, stop, poststop: Chạy bởi npm dừng lại chỉ huy.

prestart, start, poststart: Chạy bởi npm bắt đầu chỉ huy.

khởi động lại trước, khởi động lại, khởi động lại sau: Chạy bởi npm khởi động lại chỉ huy. Lưu ý rằng npm khởi động lại sẽ chạy dừng lại và khởi đầu tập lệnh nếu không khởi động lại kịch bản được cung cấp.

Cần rõ ràng rằng các lệnh có tiền tố sẽ chạy trước và các lệnh có bài đăng tiền tố sẽ chạy sau lệnh chính của chúng (chẳng hạn như công bố) được thực thi.

Các tệp gói thể hiện việc sử dụng các chỉ thị tập lệnh có thể được tìm thấy trong mocha-zombie thư mục trong gói mã cho Chương 9, Kiểm tra ứng dụng của bạn.

## Khai báo phụ thuộc

Có khả năng bản thân một mô-đun đã cho sẽ phụ thuộc vào các mô-đun khác. Những phụ thuộc này được khai báo trong package.json sử dụng bốn thuộc tính liên quan, đó là:

phụ thuộc: Các phụ thuộc cốt lõi của mô-đun của bạn nên nằm ở đây.

devDependencies: Một số mô-đun chỉ cần thiết trong quá trình phát triển (và không cần sản xuất), chẳng hạn như những mô-đun được sử dụng để thử nghiệm. Khai báo các mô-đun đó tại đây.

npm install sẽ luôn cài đặt cả hai sự phụ thuộc và

devDependencies. Để cài đặt gói npm mà không cần tải devDependencies, sử dụng lệnh npm cài đặt - sản xuất.

BundledDependencies: Node đang thay đổi nhanh chóng, các gói npm cũng vậy. Bạn có thể muốn khóa một gói phụ thuộc nhất định vào một tệp gói duy nhất và có những tệp đó được xuất bản cùng với gói của bạn, sao cho chúng sẽ không thay đổi thông thường cập nhật npm quá trình.

tùy chọn Chứa các mô-đun là tùy chọn. Nếu không thể tìm thấy hoặc cài đặt các mô-đun này, quá trình xây dựng sẽ không dừng lại (giống như các lỗi tải phụ thuộc khác). Sau đó, bạn có thể kiểm tra sự tồn tại của mô-đun này trong mã ứng dụng của mình.

Các phần phụ thuộc thường được xác định bằng tên gói npm theo sau là thông tin lập phiên bản:

"dependencies" : {

"express" : "3.3.5"

}

Tuy nhiên, chúng cũng có thể trỏ đến một tarball:

"foo": "http://foo.com/foo.tar.gz"

Bạn có thể trỏ đến kho lưu trữ GitHub:

"herder": "git: //github.com/sandro-pasquali/herder.git#master"

Hoặc thậm chí là phím tắt:

"herder": "sandro-pasquali / herder"

Các đường dẫn GitHub này cũng có sẵn để npm cài đặt. Ví dụ, npm cài đặt sandro-pasquali / herder.

Ngoài ra, trong trường hợp chỉ những người có xác thực thích hợp mới có thể cài đặt mô-đun, định dạng sau có thể được sử dụng để tạo nguồn các kho lưu trữ an toàn:

"dependencies": {

"a-private-repo":

"git+ssh://git@github.com:user/repo.git#master"

}

Bằng cách tổ chức hợp lý các phần phụ thuộc của bạn theo loại và tìm nguồn cung cấp các phần phụ thuộc đó một cách thông minh, các yêu cầu xây dựng sẽ dễ dàng đáp ứng bằng cách sử dụng hệ thống gói của Node.

## Gói xuất bản

Để xuất bản lên npm, bạn sẽ cần tạo một người dùng. npm adduser sẽ kích hoạt một loạt lời nhắc yêu cầu tên, e-mail và mật khẩu của bạn. Sau đó, bạn có thể sử dụng lệnh này trên nhiều máy để cấp quyền cho cùng một tài khoản người dùng.

Để đặt lại mật khẩu npm của bạn, hãy truy cập https://npmjs.org/forgot.

Khi bạn đã xác thực với npm, bạn sẽ có thể xuất bản các gói của mình bằng lệnh npm xuất bản. Đường dẫn dễ nhất là chạy lệnh này từ thư mục gói của bạn. Bạn cũng có thể nhắm mục tiêu một thư mục khác để xuất bản (hãy nhớ rằng package.json tệp phải tồn tại trong thư mục đó).

Bạn cũng có thể xuất bản một kho lưu trữ TAR được giải nén có chứa một thư mục gói được định cấu hình đúng cách.

Lưu ý rằng nếu phiên bản lĩnh vực hiện tại package.json tệp thấp hơn hoặc bằng đối với gói hiện có, đã xuất bản npm sẽ khiếu nại và từ chối xuất bản. Bạn có thể ghi đè điều này bằng cách sử dụng - lực lượng tranh luận với công bố, nhưng bạn có thể muốn cập nhật phiên bản và xuất bản lại.

Để loại bỏ một gói sử dụng npm hủy xuất bản <name> [@ <version>]. Lưu ý rằng khi một gói được xuất bản các nhà phát triển khác có thể phụ thuộc vào nó. Vì lý do này, bạn thực sự không khuyến khích loại bỏ các gói mà người khác đang sử dụng. Nếu điều bạn muốn làm là không khuyến khích sử dụng một phiên bản, hãy sử dụng npm không dùng nữa <tên> [@ <phiên bản>] <tin nhắn>.

Để hỗ trợ thêm cho sự cộng tác, npm cho phép nhiều chủ sở hữu được đặt cho một gói:

npm chủ sở hữu ls <tên gói>: Liệt kê những người dùng có quyền truy cập vào một mô-đun.

npm chủ sở hữu thêm <người dùng> <tên gói>: Chủ sở hữu được thêm vào sẽ có đầy đủ quyền truy cập, bao gồm khả năng sửa đổi gói và thêm các chủ sở hữu khác.

npm chủ sở hữu rm <người dùng> <tên gói>: Xóa chủ sở hữu và ngay lập tức thu hồi tất cả các đặc quyền.

Tất cả chủ sở hữu đều có đặc quyền như nhau — không có các biện pháp kiểm soát truy cập đặc biệt, chẳng hạn như có thể cấp quyền ghi nhưng không thể xóa quyền truy cập.

## Cài đặt gói và mã nhị phân trên toàn cầu

Một số mô-đun Node hữu ích như các chương trình dòng lệnh. Thay vì yêu cầu một cái gì đó chẳng hạn như> node module.js để chạy một chương trình, chúng ta có thể chỉ cần nhập> mô-đun trên bảng điều khiển và để chương trình thực thi. Nói cách khác, chúng tôi có thể muốn coi một mô-đun như một tệp thực thi được cài đặt trên hệ thống $ CON ĐƯỜNG và do đó có thể truy cập từ mọi nơi. Có hai cách để đạt được điều này bằng cách sử dụng npm.

Cách đầu tiên và đơn giản nhất là cài đặt một gói bằng cách sử dụng - g (toàn cầu) tranh luận:

npm cài đặt -g mô-đun

Nếu một gói được dự định là một ứng dụng dòng lệnh nên được cài đặt trên toàn cầu, thì bạn nên đặt thích hơn tài sản của bạn package.json nộp cho thật. Mô-đun sẽ vẫn cài đặt cục bộ, nhưng người dùng sẽ được cảnh báo về ý định toàn cầu của nó.

Một cách khác để đảm bảo quyền truy cập toàn cầu là bằng cách thiết lập một gói thùng rác bất động sản:

"name": "aModule",

"thùng rác" : {

"aModule": "./path/to/program"

}

Khi mô-đun này được cài đặt, aModule sẽ được hiểu là một lệnh CLI toàn cục. Bất kỳ số lượng chương trình nào như vậy có thể được ánh xạ tới thùng rác. Là một phím tắt, một chương trình đơn lẻ có thể được ánh xạ như vậy:

"name": "aModule",

"bin": "./path/to/program"

Trong trường hợp này là tên của chính gói ( aModule) sẽ được hiểu là lệnh hoạt động.

Chia sẻ kho

Các mô-đun nút thường được lưu trữ trong hệ thống kiểm soát phiên bản, cho phép một số nhà phát triển quản lý mã gói. Vì lý do này, kho lĩnh vực

package.json có thể được sử dụng để chỉ các nhà phát triển đến một kho lưu trữ như vậy, nếu mong muốn sự cộng tác. Ví dụ:

"repository" : {

"type" : "git",

"url" : "http://github.com/sandro-pasquali/herder.git"

}

"repository" : {

"type" : "svn",

"url" : "http://v8.googlecode.com/svn/trunk/"

}

Tương tự, bạn có thể muốn hướng người dùng đến nơi cần gửi báo cáo lỗi bằng cách sử dụng trường lỗi:

"bugs": {

"url": "https://github.com/sandro-pasquali/herder/issues"

}

# PHỤ LỤC B

# Giới thiệu con đường Khuôn khổ

“Những việc làm lớn được tạo nên từ những việc làm nhỏ”.

- Lão Tử

JavaScript là ngôn ngữ duy nhất được cài đặt nguyên bản trong các trình duyệt. Tất cả các ứng dụng dựa trên web phụ thuộc vào JavaScript để cung cấp giao diện người dùng của chúng. Nếu bạn đang sử dụng một khuôn khổ được xây dựng trên một ngôn ngữ khác (chẳng hạn như Ruby on Rails), bạn phải hợp nhất hai mô hình phát triển hoàn toàn khác nhau, một cho máy khách và một cho máy chủ. Với sự xuất hiện của Node và JavaScript trên máy chủ, quy trình làm việc đáng tiếc và khó xử này đã được thực hiện trơn tru và thanh lịch.

Path Framework là một nền tảng phát triển ứng dụng toàn ngăn xếp mạnh mẽ sẽ giúp bạn tạo các ứng dụng cấp doanh nghiệp trong thời gian ngắn hơn. Không giống như Express, được giới hạn ở phía máy chủ của một ứng dụng và các khuôn khổ phía máy khách như EmberJS không cung cấp trợ giúp về việc xây dựng máy chủ, Path thống nhất quy trình phát triển ứng dụng của bạn. Với việc tập trung vào toàn bộ ngăn xếp, Path cho phép nhà phát triển JavaScript xây dựng các ứng dụng hoàn chỉnh bằng một ngôn ngữ duy nhất có thể mở rộng, dễ bảo trì, kiểm tra và sửa đổi. Tất nhiên, Node chiếm một vị trí trung tâm trong thiết kế Path.

Mục tiêu của Path là giúp các nhà phát triển dễ dàng xây dựng các ứng dụng web hoạt động giống như các ứng dụng gốc. Điều này có nghĩa là các ứng dụng một trang đáp ứng với các tính năng mạnh mẽ.

Dữ liệu được đồng bộ hóa tự động giữa máy khách và máy chủ khi bạn xây dựng

một ứng dụng với Path. Không còn phải xây dựng các móc đồng bộ hóa máy khách / máy chủ phức tạp và dễ xảy ra lỗi. Các giao diện máy khách được cấu tạo từ các mô-đun với bất kỳ số lượng và độ phức tạp nào, dễ dàng sử dụng cú pháp thanh lịch, có tính khai báo và lấy dữ liệu làm trung tâm. Dữ liệu thời gian thực truyền qua các nhóm khách hàng, theo cả hai hướng, trong tích tắc. Việc lập trình các trình xử lý tuyến đường trên máy chủ cũng dễ dàng như việc liên kết một cú nhấp chuột vào một nút.

Một số tính năng chính của Path như sau:

Chỉ cần JavaScript: Chuyển đổi ngữ cảnh rất khó. Cuối cùng, một nhóm thống nhất nói một ngôn ngữ chung, trên toàn bộ ngăn xếp.

Làm mới giao diện người dùng tự động: Các mô-đun giao diện người dùng của bạn tự động cập nhật khi dữ liệu mà chúng bị ràng buộc với những thay đổi. Liên kết dữ liệu hai chiều mà không có nút thắt.

Không còn CRUD: Vượt ra khỏi mô hình CRUD cũ bẩn thỉu thành một mô hình giao tiếp dựa trên ổ cắm hiệu quả được thiết kế để phát triển ứng dụng thời gian thực hiện đại.

Cập nhật mã nóng: Thay đổi mã của bạn trên máy chủ và các máy khách sẽ được cập nhật tự động.

Khả năng tương tác: Bất cứ thứ gì có thể giao tiếp qua WebSockets đều có thể giao tiếp với máy chủ Path.

Tải mô-đun đơn giản bụi bẩn: Cú pháp HTML khai báo đơn giản giúp bạn soạn giao diện người dùng theo cách dễ tạo và người khác dễ đọc.

Bộ nhớ đệm mô-đun tự động: Nó có nghĩa là tải phụ thuộc thông minh.

Dễ dàng như HTML: Các mô-đun được thể hiện tự nhiên bằng cách lồng các phần tử HTML. Các mô-đun đã tải có thể tự tạo thêm các cây phụ thuộc mô-đun. Thuộc tính phần tử lưu trữ các tùy chọn và cài đặt thành phần khác. Sự kiện của người dùng bị ràng buộc với các hành động của máy chủ mà không sử dụng bất kỳ JavaScript nào.

Có thể nhúng: Sử dụng nhiều hay ít Path tùy thích. Con đường ở đây để giúp đỡ, không cản trở. Các khung công tác khác thậm chí có thể chia sẻ máy chủ Đường dẫn của bạn.

Năng suất vượt trội: Đường dẫn đi kèm với các ứng dụng dựng sẵn mà bạn có thể bắt đầu ngay lập tức.

Để bắt đầu, hãy sao chép kho lưu trữ Đường dẫn từ đó và https://github.com/sandro-pasquali/pathjs.git, làm theo hướng dẫn trên trang đó.

## Quản lý trạng thái

Như chúng ta đã thảo luận trước đó:

Nếu các thay đổi có thể được đưa vào ứng dụng web mà không yêu cầu xây dựng lại hoàn toàn trạng thái và hiển thị trạng thái thì việc cập nhật thông tin khách hàng trở nên rẻ hơn. Máy khách và máy chủ có thể nói chuyện thường xuyên hơn, thường xuyên trao đổi thông tin. Máy chủ có thể nhận ra, ghi nhớ và phản hồi ngay lập tức với mong muốn của khách hàng, được hỗ trợ bởi các giao diện phản ứng thu thập các hành động của người dùng và phản ánh tác động của những hành động đó trong giao diện người dùng trong thời gian thực.

Một câu hỏi rất quan trọng mà mọi nhà phát triển ứng dụng phải hỏi là: nhà nước nên cư trú ở đâu?

Tất nhiên, dữ liệu vĩnh viễn cuối cùng phải nằm trong một lớp liên tục như

cơ sở dữ liệu phía máy chủ. Ngoài ra, dữ liệu trạng thái nhất thời (chẳng hạn như mục điều hướng nào được đánh dấu trong giao diện người dùng) có thể vui vẻ tồn tại trong ứng dụng khách. Tuy nhiên, nếu một ứng dụng hiển thị các bản ghi ngân hàng thực trong lưới dữ liệu trong trình duyệt, thì các thay đổi được thực hiện đối với bản ghi chuẩn (cơ sở dữ liệu) sẽ đồng bộ hóa với các thay đổi trong mô hình khách hàng như thế nào?

Ví dụ: hãy tưởng tượng một ứng dụng ngân hàng trong đó giao diện người dùng dựa trên trình duyệt cho người dùng biết rằng cô ấy có số dư là 100 USD. Giao diện người dùng này được "vẽ" dựa trên yêu cầu trước đó đối với máy chủ, hỏi đại loại như: Số dư ngân hàng của Mary là bao nhiêu ? Rõ ràng con số đó có thể thay đổi trong giây tiếp theo, hoặc thậm chí là mili giây, do tính chất tự động của ngân hàng. Làm thế nào Mary có thể chắc chắn rằng khi thanh toán tờ 100 USD, cô ấy vẫn còn 100 USD trong ngân hàng?

Một giải pháp là định kỳ hết hạn hiệu lực của trạng thái máy khách và thăm dò ý kiến của máy chủ về các thay đổi trạng thái, đồng bộ hóa lại trạng thái máy khách và máy chủ. Ngoài ra, các yêu cầu của khách hàng được xác thực lại trên máy chủ — Mary sẽ được cảnh báo nếu cô ấy cố gắng chi tiêu 100 USD mà cô ấy không có. Trạng thái khách hàng là lạc quan, và giao dịch là được

xác nhận lại.

Tuy nhiên, có một số vấn đề với mô hình này:

Không phải tất cả các trạng thái đều có thể tồn tại trên máy khách: Dữ liệu trạng thái bảo mật phải vẫn còn trên máy chủ, có nghĩa là máy khách chỉ có thể xử lý các phần trạng thái, làm cho nỗ lực tốt nhất để duy trì trạng thái máy khách được đồng bộ hóa theo định nghĩa là không hoàn hảo.

Xác thực lại không phải là một giải pháp được chấp nhận trong nhiều trường hợp: Xem xét một game bắn súng thời gian thực. Nếu một người chơi thay đổi trạng thái của người chơi khác (giả sử bằng cách giết người chơi đó), thì không thể chấp nhận được tại một thời điểm nào đó trong tương lai kích hoạt lại người chơi đã bị giết chỉ đơn giản vì khách hàng của game bắn súng có thông tin vị trí không chính xác khiến lượt giết ban đầu không hợp lệ.

Lạc quan dẫn đến bảo mật sai lầm: Vô nghĩa là một từ có thể được sử dụng để mô tả dữ liệu trạng thái có thể có hoặc có thể không hợp lệ. Ví dụ: một chương trình giao dịch cổ phiếu hiển thị giá sai cho FOO Corporation có thể khiến nhà giao dịch bán khống BAR Corporation. Nếu trạng thái của BAR (theo khách hàng) khớp với trạng thái trên máy chủ, giao dịch sẽ diễn ra và nhà giao dịch có thể mất tiền. Bất kỳ bit thông tin nào tự nó không thể xác nhận tất cả các bit thông tin khác. Tính không xác định là một tính năng của các hệ thống đồng thời và những hệ thống này thách thức suy diễn kỹ thuật xác nhận trạng thái — liệu một phép so sánh có hợp lệ hay không không phải lúc nào cũng được xác định.

Nhiều kỹ thuật đã được sử dụng để đồng bộ hóa trạng thái máy khách và máy chủ — cookie, trường biểu mẫu ẩn, URL béo và những thứ khác. Đường dẫn có một cái nhìn khác, dựa trên

những lợi thế mà Node mang lại, đặc biệt là khi được ghép nối với giao thức WebSocket. Nếu chúng ta có thể tạo các máy chủ đáp ứng cao được liên kết với các máy khách thông qua các đường ống hiệu quả, trong đó thời gian phản hồi dưới giây dự kiến, thì tốt hơn là chỉ cần giữ thông tin trạng thái quan trọng trên máy chủ và chấm dứt mọi nỗ lực tạo phản xạ phía máy khách trên một tấm gương dễ bị nhòe.

## Cầu nối phân chia máy khách / máy chủ

Node được thiết kế cho môi trường đồng thời cao, nơi nhiều máy khách kết nối đồng thời có thể được đáp ứng nhu cầu của họ một cách nhanh chóng và dễ đoán. Khả năng này cũng ngụ ý rằng mỗi máy khách có thể mong đợi một máy chủ Node phản hồi nhanh chóng khi bản thân nó gửi nhiều yêu cầu đồng thời.

Một trong những niềm tin thiết kế chính của Path là việc giữ trạng thái được đồng bộ hóa giữa các máy khách và máy chủ rất phức tạp và khó làm tốt. Vì lý do này, các ứng dụng được xây dựng với Path trao đổi dữ liệu giữa các máy khách và máy chủ độc quyền thông qua giao thức WebSockets.

Sau đây là chế độ xem cấp cao về cách Path xử lý các yêu cầu của khách hàng. Chúng tôi thấy cách cụm mô-đun được sử dụng để chia sẻ trách nhiệm xử lý các kết nối socket giữa một số máy chủ socket:



Bất kỳ thiết bị nào có thể triển khai

Giao thức WebSockets.

ws: //

ws: //

ws: //

Một trình duyệt. ví dụ, mở một kết nối ổ cắm

với một máy chủ như vậy:

var connection = new WebSocket ('ws: //127.0.0,I: 808l');

ws

A

Cụm

của hệ thống cống rãnh, một cái cho mỗi lõi. Mỗi hoạt động

trong quy trình riêng của nó. chia sẻ một cổng duy nhất. Phân phối

tải trên các quá trình là

được thực hiện bởi hệ điều hành của nó.

Máy chủ socket thực thi trong một phạm vi chung,

bậc thầy

Node process.Via IPC tích hợp sẵn và việc sử dụng kết thúc, chia

sẻ tài liệu tham khảo và các giao tiếp kế toán khác giữa

bậc thầy

và

đứa

trẻ

được hoàn thành.

Dữ liệu kết nối máy khách được chia sẻ giữa các quy trình con

trong cụm. Các kết nối được khởi tạo khi trạng thái người dùng

obiects, là các máy trạng thái.

Mỗi yêu cầu là một Giao dịch. Mỗi giao dịch mất

Trạng thái người dùng và cấu hình và khởi động nó. ràng buộc các

trình xử lý sự kiện dữ liệu, thường phát

dữ liệu trở lại khách hàng.

ws

ws

V8

V8

V8

Cụm

Giao dịch

Trạng thái người dùng

Các tác nhân sau đó được gán cho Trạng thái Người dùng. Các Tác

nhân này có nhiệm vụ thỏa mãn Giao dịch.

Các tác nhân phát ra dữ liệu / sự kiện lỗi.

trên ("dữ liệu")

on ("xong")

on ("error")

Mỗi Actor thực thi trong Context, đi qua

kết quả của nó cho Diễn viên tiếp theo, nó bất kỳ. Các diễn viên có

thể được sử dụng bởi nhiều Quốc gia Người dùng

Diễn viên

Khi một yêu cầu đã được gán cho một máy chủ, yêu cầu đó phải được thực hiện. Chúng tôi sẽ gọi đó là một giao dịch. Mỗi giao dịch đang được thực hiện cho một người dùng duy nhất và như vậy, việc thực hiện được ủy quyền cho một đối tượng người dùng. Đối tượng người dùng này là một máy trạng thái, chứa bất kỳ thông tin trạng thái cụ thể nào cho người dùng này. Tùy thuộc vào các yêu cầu thực hiện giao dịch, một hoặc nhiều tác nhân sẽ được chỉ định, đưa ra kết quả của họ (hoặc cảnh báo lỗi). Người dùng được xác thực sẽ có một máy duy nhất để xử lý yêu cầu của họ, trong khi những người dùng khác sẽ dùng chung một máy "khách".

Bằng cách này, mỗi yêu cầu có thể được đóng khung trong phạm vi người dùng, được giải quyết chính xác trong kinh điển trạng thái và trả lời một cách tự tin.

## Gửi và nhận

Ngoài việc giữ trạng thái trên máy chủ, Path cũng nhằm mục đích giảm trách nhiệm của người gọi trong việc giữ ngữ cảnh cuộc gọi cho đến khi nhận được phản hồi. Loại bảo trì khung cuộc gọi dự đoán thực thi gọi lại này là điều kiện thiết yếu của gần như tất cả các mẫu phát triển AJAX và thường được biểu diễn như sau:

// Trong một số ngữ cảnh thực thi, chẳng hạn như đầu vào tự động hoàn thành someXhrProxy.get ("/ a / path /", function (data) {

// Một lệnh gọi lại được liên kết với ngữ cảnh thực thi thông qua các bao đóng});

Một số thư viện phía máy khách cố gắng đơn giản hóa mô hình này bằng những nội dung trừu tượng như Promises, nhưng họ bỏ sót điểm: một cuộc gọi nên "khai hỏa và quên đi", công việc của nó chỉ là truyền một yêu cầu. Tác động của hành động đó, đối với cả máy chủ và máy khách, không phải là mối quan tâm của người gọi. Nhà phát triển chỉ cần xác nhận thay đổi trạng thái mong muốn hoặc yêu cầu một số thông tin. Path tạo điều kiện thuận lợi cho việc tách biệt các mối quan tâm này, loại bỏ chức năng cuộc gọi của bất kỳ trách nhiệm nào trong việc duy trì các ngữ cảnh cuộc gọi tại chức năng cấp và tự quản lý luồng thực thi:

Tất cả các phần tử có thể ràng buộc hành động của người dùng bằng cách sử dụng thuộc tính dữ liệu. Tất cả các sự

kiện tiêu chuẩn đều được bao gồm - di chuột qua, nhấp chuột, thay đổi, v.v.

Sự kiện ràng buộc là

**khai báo,**

thông qua

**hành động dữ liệu**

thuộc tính.

<

phần tử ... /

>

data-action = "click / user / getEmail ...

Thời gian chạy đường dẫn được ràng buộc, thông qua ủy quyền sự kiện, với tất cả các sự kiện

xảy ra trong một tài liệu. Khi các sự kiện ràng buộc xảy ra,

Đường dẫn sẽ chuẩn bị một đối tượng sự kiện và định tuyến nó.

Path.route

Ràng buộc đường dẫn:

path.open ("/ User / getEmail)

. click (function () {... // handler}

Bởi vì các sự kiện bị mắc kẹt thông qua ủy quyền (thay vì

ràng buộc rõ ràng trên cơ sở từng phần tử) các phần tử có thể được

đưa vào / loại bỏ mà không có tác dụng phụ.

Thời gian chạy đường dẫn

Path hoạt động theo mô hình "cháy và quên".

**gửi**

tới máy chủ thông qua một tuyến đường, đăng một số dữ liệu nếu cần. Máy chủ sẽ xử lý yêu cầu và

trả lời bằng

**resultData**

vật. Đường dẫn lắng nghe trên một ổ cắm, căn chỉnh

**callld**

với dữ liệu cuộc gọi ban đầu và định tuyến Dữ liệu kết quả đến các

đường dẫn đang nghe trên

**nhận được**

phương pháp.

Khách hàng

path.send ('/ user / getEmail');

Người phục vụ

callld, [postData]

WS: //

callld, resultData

path.open ("/ user / getEmail")

. nhận (function (resultData) {...})

Bất kỳ trình xử lý hoặc hàm nào khác có thể gọi path.route ("/ path / here", "method")

Đường dẫn sử dụng mô hình khai báo để ràng buộc các hành động của người dùng. Ví dụ: sau đây là tất cả những gì cần thiết để ràng buộc nhấp chuột sự kiện đến một phần tử:

a href="#" data-action="click/create/user/jack">Create user</a>

Chúng tôi tạo hành động để thực thi khi phần tử đó được nhấp bằng cách mở đường dẫn đã đặt và lắng nghe nhấp chuột biến cố:

path

.open("/create/user/:username")

.click(function(username) {

console.log("New user: " + username); })

Nhấp vào phần tử sẽ ghi lại Người dùng mới: jack vào bảng điều khiển.

Để gửi một yêu cầu đến máy chủ, chúng tôi làm theo một mẫu tương tự. Ví dụ: nếu chúng tôi muốn định tuyến / tạo / người dùng / jack đường dẫn đến máy chủ, tạo người dùng mới, chúng tôi sẽ sử dụng gửi chỉ huy:

path

.open("/create/user/:username")

.click(function(username) {

path.send("/create/user/" + username); })

Xem các ví dụ trong gói mã của bạn để biết thêm các giao dịch liên quan.

Đạt được kiến trúc mô-đun

Mô hình khai báo của Path cũng giúp bạn dễ dàng tải các mô-đun và phần phụ thuộc qua HTML. Một số mục tiêu chính của hệ thống tải mô-đun của nó như sau:

Bó: Khi biết rằng một nhóm mô-đun là cần thiết, hãy gói chúng thành một gói và gửi chúng theo một yêu cầu.

Bộ nhớ đệm: Sau khi một mô-đun đã được truy xuất từ máy chủ, nó sẽ không cần phải được yêu cầu lại. Đường dẫn duy trì một bộ nhớ cache của dữ liệu mô-đun trên máy khách.

Giao tiếp: Đường dẫn định tuyến các sự kiện thông qua các đường dẫn như / bấm vào / lấy / email. Các mô-đun cũng làm như vậy, truyền dữ liệu của chúng cùng module / moduleName / eventName các đường dẫn, cho phép người nghe hành động trên các đường truyền đó.

Đóng gói: Không được phép truy cập trực tiếp vào nội bộ của mô-đun. Tất cả thông tin liên lạc đều thông qua đường dẫn, như đã mô tả trước đây.

Mẫu và liên kết dữ liệu: Mỗi mô-đun dễ dàng liên kết với một nguồn dữ liệu, được phục vụ thông qua một máy chủ ổ cắm thời gian thực. Khi dữ liệu mới đến, mô-đun sẽ tự động cập nhật chế độ xem của nó.

Path được thiết kế để giúp đơn giản hóa việc thiết kế các kiến trúc mô-đun. Thông qua cú pháp khai báo, Path giúp bạn dễ dàng xem ứng dụng được sắp xếp như thế nào, ngay cả đối với một người không có kỹ thuật. Ngoài ra, cơ chế liên kết tự nhiên cho các mô-đun đơn giản hóa hơn nữa việc hiểu "những gì đang diễn ra", cũng như hỗ trợ tạo các bản cập nhật giao diện người dùng theo thời gian thực, vì các liên kết dữ liệu tự động nhận các bản cập nhật socket, chuyển dữ liệu mới vào các mẫu chế độ xem mới khi nó đến :

<

div class = "notifier"> <ldiv

>

<

div class = "module 'name =" grid'

>

<

div class = "pager"

data-params = "perPage: 10">

/div

>

<

<

>

/div

Các mô-đun được khai báo qua HTML. Bất kỳ phần tử nào cho trước

lớp học

thuộc

tính của

mô-đun

và một

Tên (

ví dụ. "lưới")

Cây phụ thuộc được phản ánh trực tiếp bởi cây DOM. Ở

đây, "máy nhắn tin" chắc chắn sẽ có

"lưới" được tải khi nó được khởi tạo.

Đường dẫn sẽ thực hiện chuyển qua DOM để tìm

kiếm các định nghĩa mô-đun, tiến triển một cấp xuống

cây. Đây, nghĩa là

"Trình thông báo" và "lưới" sẽ được nhận dạng

và tìm nạp.

xuống một mức độ ...

Ràng buộc đường dẫn:

path.open ("/ user / getEmail")

. click (function () {... // handler}

Thời gian chạy đường dẫn

notifer, mô-đun

máy nhắn tin

WS

js, html, gói css

WS

Cả hai mô-đun sẽ được tìm nạp, đóng

gói và trả về dưới dạng

phản ứng duy nhất.

Quá trình vượt qua tiếp tục cho đến khi tất cả

các mô-đun đã được tìm nạp, DOM được cập

nhật với mỗi gói mới.

Khi các gói mô-đun đến máy khách, chúng sẽ được lưu vào bộ

nhớ đệm. Nếu "lưới" mô-đun được yêu cầu hai lần, yêu cầu thứ

hai sẽ chỉ chạy lại Javaacrlpt được tải trước, chèn lại HTML, v.v.

js, html, gói css

<

trình thông báo

>

<

grid

>

<

đã thêm

>

<

phần tử

>

<

máy nhắn tin

>

<

khác

>

<

phần tử

>

<

/pager

>

>

<

/grid

DOM

NB:

Điều này cũng có nghĩa là HTML được trả về trong một gói mô-đun có thể

chinh no

khai báo các

mô-đun, các dưỡng này bị bắt trong lần vượt qua tiếp theo.

Ví dụ: nếu tôi có một danh sách các mục để hiển thị, tôi có thể sử dụng một mô-đun đặc biệt có tên danh sách. Mô-đun danh sách có thể muốn các mẹo công cụ xuất hiện khi các mục trong danh sách của nó được di chuột qua, vì vậy nó có thể phụ thuộc vào chú giải công cụ sự phụ thuộc. Mối quan hệ này sẽ được thể hiện như sau:

<div class="module" name="list">

<div class="module" name="tooltip"></div>

<ul>

<li>...</li>

<li>...</li>

</ul>

</div>

Các danh sách mô-đun bây giờ có thể sử dụng chú giải công cụ đặc trưng. Một cách khác để thể hiện mối quan hệ này như sau:

<div class = "module" name = "tooltip"> </div>

<div class = "module" name = "list">

<ul>

<li> ... </li>

<li> ... </li>

</ul>

</div>

Ưu điểm của phương pháp thứ hai là, trong khi tiếp tục đảm bảo rằng chú giải công cụ tồn tại khi danh sách được tải, cả hai mô-đun có thể được tải song song, vì chúng tồn tại ở cùng một cấp trong một cây DOM.

Xu hướng tích hợp này hướng tới việc tạo điều kiện cho việc đóng gói và tải song song giúp các ứng dụng được phát triển với Path vẫn hoạt động hiệu quả, bất kể số lượng mô-đun hoặc độ phức tạp của các phụ thuộc. Tính linh hoạt do Path cung cấp ở đây làm cho việc tải và gán phụ thuộc trở nên minh bạch và hiệu quả.

Tự động ràng buộc cũng dễ dàng đạt được. Hãy xem xét HTML sau:

<textarea data-binding = "form.something.selected"> </textareosystem <select data-binding =

"form.something.selected">

<option value = "one"> 1 </option>

<option value = "two"> 2 </option> </select>

Lưu ý cách cả hai lựa chọn phần tử và textarea phần tử có các thuộc tính liên kết dữ liệu giống hệt nhau. Đường dẫn sẽ tự động "kết nối" mối quan hệ này sao cho bất cứ khi nào lựa chọn mới được thực hiện nội dung của textarea sẽ phản ánh dữ liệu mới được chọn. Với rất ít công việc, các mối quan hệ dữ liệu rất phức tạp và các vòng phản hồi tương tác có thể được tạo bằng Path.

Điều tra các ví dụ được đưa ra trong gói mã của bạn và thử nghiệm với các chuỗi và tổ hợp mô-đun khác nhau, các ràng buộc dữ liệu khác nhau. Với Path, chúng tôi thực tế có thể bắt đầu thiết kế các ứng dụng trong đó một ứng dụng khách được hiểu đơn giản là một trình phát sự kiện và trình nghe sự kiện khác, không khác gì một Suối hoặc là process.stdin, điều này cũng đúng đối với nhận thức của khách hàng về máy chủ.

Với thiết kế toàn diện toàn ngăn xếp, Path là khung thời gian thực được thiết kế cho nhà phát triển Node.

# PHỤ LỤC C

# Tạo của riêng bạn Tiện ích bổ sung C ++

"Nếu hai (người) trong cùng một công việc đồng ý với nhau mọi lúc thì một người là vô ích. Nếu họ không đồng ý mọi lúc, thì cả hai đều vô dụng."

- Darryl F. Zanuck

Một mô tả rất phổ biến về Node là: Node.js cho phép JavaScript chạy trên

máy chủ. Trong khi đúng, điều này cũng gây hiểu lầm. Những người tạo ra Node đã tổ chức và liên kết các thư viện C ++ mạnh mẽ theo cách mà hiệu quả của chúng có thể được khai thác mà các nhà phát triển không cần phải hiểu sự phức tạp của chúng. Node loại bỏ sự phức tạp của đa người dùng, I / O đa luồng đồng thời bằng cách gói mô hình đồng thời đó vào một môi trường đơn luồng dễ hiểu và đã được hàng triệu nhà phát triển web hiểu rõ.

Khi bạn đang làm việc với Node, cuối cùng bạn đang làm việc với C ++, một ngôn ngữ có tính phù hợp với phát triển phần mềm cấp doanh nghiệp không ai có thể nghi ngờ.

Nền tảng C ++ của Node đưa ra lời nói dối khi tuyên bố rằng Node chưa sẵn sàng cho doanh nghiệp. Những tuyên bố này làm nhầm lẫn vai trò của JavaScript trong Node stack thực sự là gì. Các ràng buộc với Redis và các trình điều khiển cơ sở dữ liệu khác được sử dụng thường xuyên trong các chương trình Node là các liên kết C – nhanh, gần "metal". Các ràng buộc quy trình của Node ( đẻ trứng, thực thi, v.v.) tạo điều kiện tích hợp trơn tru các thư viện hệ thống mạnh mẽ (chẳng hạn như ImageMagick) với các trình duyệt không có đầu và luồng dữ liệu HTTP. Với Node, chúng tôi có thể truy cập vào bộ chương trình Unix gốc cực kỳ mạnh mẽ thông qua JavaScript dễ dàng và thoải mái. Và tất nhiên, chúng ta có thể viết các tiện ích bổ sung C ++ của riêng mình.

Diễn giải mô tả của Giáo sư Keith Devlin trong Giải tích: Một trong những cái nhiều nhất Công nghệ thành công ( https://www.youtube.com/watch?v=8ZLC0egL6pc), những là một số tính năng của công nghệ tiêu dùng thành công:

Nó sẽ loại bỏ khó khăn hoặc vất vả trong quá trình hoàn thành nhiệm vụ.

## Nó phải dễ học và sử dụng.

Sẽ dễ học và dễ sử dụng hơn phương pháp phổ biến, nếu có.

Sau khi đã học, nó có thể được sử dụng mà không cần chuyên gia hướng dẫn liên tục. Người dùng vẫn có thể nhớ và / hoặc rút ra hầu hết hoặc tất cả các quy tắc chi phối các tương tác với công nghệ theo thời gian.

Có thể sử dụng nó mà không cần biết nó hoạt động như thế nào.

Hy vọng rằng, khi bạn nghĩ về lớp vấn đề mà Node muốn giải quyết, và hình thức của giải pháp mà nó cung cấp, năm tính năng trên có thể dễ dàng nhìn thấy trong công nghệ mà Node đại diện. Node rất thú vị khi học và sử dụng, với giao diện nhất quán và dễ đoán. Điều quan trọng là, Node "chui" chạy các công cụ cực kỳ mạnh mẽ mà nhà phát triển chỉ cần hiểu về API của họ.

Thật tuyệt vời, Node, V8, libuv và các thư viện khác tạo nên ngăn xếp Node có nguồn mở, một thực tế quan trọng giúp phân biệt Node với nhiều đối thủ cạnh tranh hơn nữa. Một người không chỉ có thể đóng góp trực tiếp vào các thư viện lõi mà còn có thể cắt và dán các khối mã và các quy trình khác để sử dụng trong công việc của mình. Bạn sẽ thấy sự phát triển của mình thành một nhà phát triển Node tốt hơn là cơ hội để đồng thời trở thành một lập trình viên C ++ giỏi hơn.

Đây không phải là một bài học sơ lược về C ++, để bạn tự theo đuổi nghiên cứu này. Đừng để bị đe dọa! Nhóm ngôn ngữ C được thiết kế bằng cách sử dụng các biểu mẫu và thành ngữ không giống như những gì bạn đã quen với JavaScript. Ví dụ, nếu..thì cấu trúc, cho các vòng lặp, hàm và giá trị trả về từ các hàm, ngay cả các dòng kết thúc bằng dấu chấm phẩy, là trọng tâm của JavaScript và thường xuyên được sử dụng trong lập trình C ++. Là một nhà phát triển JavaScript, bạn sẽ có thể hiểu thiết kế và mục tiêu của các ví dụ sau với một chút nỗ lực và có thể tham gia vào lập trình C ++ để giải quyết ý nghĩa của những phần không rõ ràng. Mở rộng các ví dụ này lặp đi lặp lại là một cách tuyệt vời để bước vào thế giới lập trình C ++ một cách nhẹ nhàng.

## Hello World

Hãy xây dựng tiện ích bổ sung đầu tiên của chúng tôi. Để phù hợp với truyền thống, tiện ích bổ sung này sẽ dẫn đến một mô-đun Node sẽ in ra Chào thế giới!. Mặc dù đây là một ví dụ rất đơn giản, nhưng nó mô tả cấu trúc của tất cả các tiện ích bổ sung C ++ mà bạn sẽ xây dựng. Điều này cho phép bạn từng bước thử nghiệm các lệnh và cấu trúc mới, nâng cao kiến thức của bạn theo các bước dễ hiểu.

Các tệp C ++ thường được cung cấp một. cc sự mở rộng. Để bắt đầu, chúng tôi tạo xin chào.cc tập tin:

#include <node.h>

#include <v8.h> using namespace v8;

Handle<Value> Method(const Arguments& args) {

HandleScope scope;

return scope.Close(String::New("Hello World!")); }

void init(Handle<Object> target) { target->Set(String::NewSymbol("hello"),

FunctionTemplate::New(Method)->GetFunction());

}

NODE\_MODULE(hello, init)

Ví dụ này bao gồm các thư viện Node và V8, xây dựng mô-đun của chúng tôi bằng cách sử dụng cùng một thư viện Các mô-đun Node được xây dựng trên NODE\_MODULE chỉ đơn giản là một macro giúp đơn giản hóa việc xuất một hàm khởi tạo, mà mọi tiện ích bổ sung sẽ cung cấp. Tên của chính mô-đun phải được chuyển như một đối số đầu tiên. Đối số thứ hai phải trỏ đến hàm khởi tạo.

Bạn đang nhúng mã C ++ vào thời gian chạy V8 để JavaScript có thể được liên kết trong phạm vi liên quan. V8 phải xác định phạm vi tất cả các phân bổ mới được thực hiện trong mã của bạn và vì vậy bạn sẽ cần phải bọc mã bạn viết, mở rộng V8. Cuối cùng, bạn sẽ thấy một số trường hợp của Xử lý <Giá trị> cú pháp, gói mã C ++ trong các ví dụ sau. So sánh các trình bao bọc này với những gì sẽ được xác định trong hàm khởi tạo được đẩy ra NODE\_MODULE nên làm rõ cách Node đang được liên kết với các phương thức C ++ thông qua cầu nối V8.

Để tìm hiểu thêm về cách V8 nhúng mã C ++, hãy truy cập https://developers.google.com/v8/embed.

Không giống như JavaScript, C ++ phải được biên dịch trước khi thực thi. Bước biên dịch này chuyển đổi mã có thể đọc được của con người thành mã bytecode được tối ưu hóa. Để biên dịch mã C ++ sao cho nó có thể dễ dàng được sử dụng như một phần bổ trợ Node, chúng ta sẽ cần sử dụng node-gyp, công cụ cần thiết để xây dựng các tiện ích bổ sung của chúng tôi. Để cài đặt nó, chỉ cần chạy npm cài đặt -g node-gyp.

Lợi ích của việc sử dụng node-gyp ( so với các trình biên dịch gốc như gcc) là nó loại bỏ nỗi đau của việc giải quyết sự khác biệt giữa các nền tảng. Bạn chỉ có thể viết mã và biết rằng quá trình xây dựng sẽ hoạt động. Để kết thúc, hãy biên dịch xin chào.cc tập tin.

Bước đầu tiên là tạo binding.gyp tệp này sẽ cung cấp cho trình biên dịch thông tin xây dựng cần thiết. Một tệp cơ bản sẽ đủ trong trường hợp của chúng tôi:

{

"targets": [

{

"target\_name": "hello",

"sources": [ "hello.cc" ]

}

]

}

]

}

Lưu ý Tên mục tiêu của " xin chào", tương ứng với Mục tiêu-

> Đặt (Chuỗi :: NewSymbol ("xin chào") dòng trong của chúng tôi. cc tệp và liên kết của nguồn xin chào.cc tập tin.

Trong trường hợp nhiều hơn một tệp nguồn cần được biên dịch, chỉ cần thêm nó vào nguồn mảng.

Trong cùng một thư mục, hãy nhập cấu hình node-gyp vào thiết bị đầu cuối của bạn. Bạn sẽ thấy thông tin về quá trình xây dựng được in vào thiết bị đầu cuối của bạn. Sau khi hoàn thành một mới xây dựng/ thư mục sẽ tồn tại, chứa các tệp khác nhau. Quan trọng là, sẽ tồn tại một Makefile. Hãy tiếp tục và thay đổi thư mục hiện tại thành xây dựng/ và chạy làm.

Bạn sẽ thấy một cái gì đó như thế này trong thiết bị đầu cuối của mình:

CXX (target) Release / obj.target / hello / hello.o SOLINK\_MODULE (target) Release / hello.node SOLINK\_MODULE (target) Release / hello.node: Đã xong

Quá trình xây dựng hiện đã hoàn tất. Như được mô tả trong đầu ra trước, Giải phóng/ thư mục đã được tạo có chứa một biên dịch xin chào.node tập tin. Đây là tệp chúng tôi sẽ liên kết đến trong mã Node của chúng tôi.

Là một phím tắt bạn có thể sử dụng xây dựng cấu hình node-gyp đến cấu hình và xây dựng trong một bước.

Khi chúng tôi đã xây dựng và biên dịch tiện ích bổ sung của mình, chúng tôi có thể bắt đầu sử dụng nó ngay lập tức trong các chương trình Node của riêng chúng tôi. Quay lại thư mục chứa xin chào.cc. Trong thư mục này, tạo chương trình Node sau:

var addon = demand ('./ build / Release / hello'); console.log (addon.hello ());

Chạy chương trình này và bạn sẽ thấy Chào thế giới! hiển thị trong thiết bị đầu cuối của bạn. Bây giờ bạn là một lập trình viên C ++!

## Tạo máy tính

Tất nhiên người ta sẽ không bao giờ bận tâm đến việc viết một tiện ích bổ sung để chỉ lặp lại các chuỗi. Nhiều khả năng bạn sẽ muốn hiển thị API hoặc giao diện cho các chương trình Node của mình. Hãy tạo một máy tính đơn giản, với hai phương pháp: thêm vào và trừ đi.

Trong ví dụ này, chúng tôi sẽ trình bày cách chuyển các đối số từ JavaScript đến các phương thức trong một tiện ích bổ sung và gửi lại bất kỳ kết quả nào.

Mã hoàn chỉnh cho ví dụ này sẽ được tìm thấy trong gói mã của bạn. Phần nội dung của chương trình có thể được nhìn thấy trong đoạn mã này, nơi chúng tôi xác định một giao diện cho hai phương thức của mình, mỗi phương thức mong đợi nhận được hai số làm đối số:

#include <node.h>

#include <v8.h> using namespace v8;

Handle<Value> Add(const Arguments& args) {

HandleScope scope;

if(args.Length() < 2) {

ThrowException(Exception::TypeError(String::New("Wrong number

of arguments")));

return scope.Close(Undefined()); }

if(!args[0]->IsNumber() || !args[1]->IsNumber()) {

ThrowException(Exception::TypeError(String::New("Wrong

arguments")));

return scope.Close(Undefined());

}

Local<Number> num = Number::New(args[0]->NumberValue() +

args[1]->NumberValue()); return scope.Close(num); }

Handle<Value> Subtract(const Arguments& args) { // Similar method to do subtraction...

}

void Init(Handle<Object> exports) { exports->Set(String::NewSymbol("add"),

FunctionTemplate::New(Add)->GetFunction()); exports->Set(String::NewSymbol("subtract"),

FunctionTemplate::New(Subtract)->GetFunction());

}

## NODE\_MODULE(calculator, Init)

Chúng ta có thể nhanh chóng thấy rằng hai phương pháp đã được xác định phạm vi: Thêm vào và Trừ đi

( Trừ đi được định nghĩa gần giống với Thêm vào, chỉ với một sự thay đổi của toán tử). Trong Thêm vào phương pháp chúng tôi thấy một Tranh luận đối tượng (gợi nhớ đến JavaScript của

tranh luận đối tượng) được kiểm tra độ dài (chúng tôi mong đợi hai đối số) và loại (chúng tôi muốn số). Hãy xem kỹ cách đóng lại của phương thức này:

Local<Number> num = Number::New(args[0]->NumberValue() + args[1]-

>NumberValue()); return scope.Close(num);

Trong khi có vẻ như có rất nhiều điều đang diễn ra, nó thực sự khá đơn giản: V8 được hướng dẫn để phân bổ không gian cho Con số biến có tên num, được gán giá trị của việc cộng hai số của chúng ta với nhau. Khi hoạt động này đã hoàn thành, chúng tôi đóng phạm vi thực thi và trả về số Chúng tôi không phải lo lắng về quản lý bộ nhớ cho tham chiếu này, vì điều đó được xử lý tự động bởi V8.

Cuối cùng, chúng ta thấy trong phần sau không chỉ cách chương trình cụ thể này xác định giao diện của nó mà còn ở mức độ sâu, các mô-đun Node và hàng xuất khẩu thực tế đối tượng được liên kết:

void Init(Handle<Object> exports) { exports->Set(String::NewSymbol("add"),

FunctionTemplate::New(Add)->GetFunction()); exports->Set(String::NewSymbol("subtract"),

FunctionTemplate::New(Subtract)->GetFunction());

}

Như trong " xin chào" ví dụ, ở đây chúng ta thấy các ký hiệu mới (đây chỉ là các loại chuỗi) thêm vào và trừ đi, đại diện cho các tên phương thức cho mô-đun Node mới của chúng tôi. Chức năng của chúng được thực hiện dễ dàng theo dõi

FunctionTemplate :: New (Thêm) -> GetFunction ()).

Sử dụng máy tính của chúng tôi từ một chương trình Node giờ đây khá dễ dàng:

var Calculator = request ('./ build / Release / Calculator');

console.log (Calculator.add (2,3)); console.log (Calculator.subtract (3,2));

Khi điều này được thực thi, bạn sẽ thấy thông tin sau được hiển thị trong thiết bị đầu cuối của mình:

5

1

## Triển khai lệnh gọi lại

Để phù hợp với mô hình điển hình của một chương trình Node, các tiện ích bổ sung cũng triển khai khái niệm gọi lại. Như người ta có thể mong đợi trong một chương trình Node, một phần bổ trợ C ++ thực hiện một hoạt động tốn kém và tốn thời gian sẽ hiểu được khái niệm về các hàm thực thi không đồng bộ.

Đoạn mã sau sẽ hiển thị một phương thức sẽ trả lại thời gian hệ thống hiện tại cho bất kỳ lệnh gọi lại nào mà nó được gửi:

#include <node.h>

#include <ctime> using namespace v8;

Handle<Value> GetTime(const Arguments& args) {

HandleScope scope;

Local<Function> cb = Local<Function>::Cast(args[0]);

const unsigned argc = 1; time\_t stamp = time(0);

Local<Value> argv[argc] = {

Local<Value>::New(Number::New(stamp))

};

cb->Call(Context::GetCurrent()->Global(), argc, argv);

return scope.Close(Undefined());

}

void Init(Handle<Object> exports, Handle<Object> module) {

module->Set(String::NewSymbol("exports"),

FunctionTemplate::New(GetTime)->GetFunction());

}

## NODE\_MODULE(callback, Init)

Ở đây chúng tôi bao gồm ctime thư viện tiêu chuẩn, sử dụng phương pháp thời gian của nó khi thiết lập Dành thời gian xử lý. Điều kỳ diệu xảy ra bên trong ô điều khiển này, nơi đối số hàm của bạn được liên kết và gọi đúng cách.

Khi bạn đã tạo tệp ràng buộc và biên dịch mọi thứ, hãy tiếp tục và chạy mô-đun gọi lại của bạn:

var timeNow = request ('./ build / Release / callback');

timeNow (function (tem) { console.log (tem);

});

// 1481315296

Bắt đầu với những công cụ này, bạn có thể bắt đầu phát triển một số tiện ích bổ sung đơn giản không kém, cuối cùng chuyển sang lãnh thổ sâu hơn.

## Bớt tư tưởng

Có thể dễ dàng liên kết các mô-đun C ++ với chương trình Node của bạn là một mô hình mới mạnh mẽ. Do đó, có thể rất hấp dẫn khi bắt đầu viết các tiện ích bổ sung C ++ cho mọi phân đoạn có thể nhận dạng của chương trình của bạn. Mặc dù đây có thể là một cách học hiệu quả, nhưng nó không nhất thiết phải là ý tưởng tốt nhất về lâu dài. Mặc dù chắc chắn đúng rằng nói chung, C ++ được biên dịch sẽ chạy nhanh hơn mã JavaScript, hãy nhớ rằng V8 cuối cùng đang sử dụng một kiểu biên dịch khác trên mã JavaScript mà nó đang chạy. JavaScript chạy trong V8 chạy rất hiệu quả. Ngoài ra, chúng tôi không muốn đánh mất tính đơn giản của tổ chức và thời gian chạy đơn luồng có thể dự đoán được của JavaScript khi thiết kế các tương tác phức tạp trong môi trường đồng thời cao. Nhớ lại: Node ra đời một phần là một nỗ lực giúp nhà phát triển không phải làm việc với các luồng và sự phức tạp liên quan khi thực hiện I / O. Do đó, hãy thử và ghi nhớ hai quy tắc sau:

Một mô-đun C ++ có thực sự sẽ chạy nhanh hơn không ?: Câu trả lời không phải lúc nào cũng có. Bước bổ sung là nhảy vào một ngữ cảnh thực thi khác và sau đó quay lại V8 sẽ là lãng phí nếu tiện ích bổ sung của bạn chỉ trả về một chuỗi tĩnh. Bài nói chuyện của Felix Geisendorfer mô tả công việc của mình với xây dựng các ràng buộc MySQL nhanh cung cấp một số thông tin chi tiết về các quyết định này: http://www.youtube.com/watch?v=Kdwwvps4J9A.

Việc tách cơ sở mã của bạn ảnh hưởng đến khả năng bảo trì như thế nào ?: Mặc dù sẽ khó cho bất kỳ nhà phát triển nào đề xuất sử dụng mã kém hiệu quả hơn, nhưng đôi khi mức tăng hiệu suất không đáng kể không vượt qua được mức tăng

phức tạp có thể dẫn đến lỗi khó tìm hơn hoặc khó khăn hơn khi chia sẻ hoặc quản lý codebase của bạn.

Node đã hợp nhất một API JavaScript tuyệt đẹp với một ngăn xếp ứng dụng cực kỳ mạnh mẽ và dễ dàng mở rộng. Với khả năng tích hợp C ++ vào các ứng dụng của bạn, không có lý do gì để loại trừ Node khỏi danh sách các công nghệ cần xem xét cho dự án tiếp theo của bạn.

## Liên kết và tài nguyên

Tài liệu về Node cho các tiện ích bổ sung rất tuyệt vời: http://nodejs.org/api/ addons.html.

Bạn có thể tìm thấy một kho chứa nhiều ví dụ về tiện ích bổ sung tại đây:

https://github.com/rvagg/node-addon-examples.

Một tài nguyên tuyệt vời cho những người học C ++: http://www.learncpp.com/.

Khi bạn cảm thấy tự tin hơn, mã nguồn cho các mô-đun cốt lõi của Node là một nơi tuyệt vời để cả khám phá và học hỏi: https://github.com/ joyent / node / tree / master / src. .

------------------------------