

Node.js 학습을 하는 사람이라면 한번쯤은 꼭 봐둬야 할

Ryan Dahl: Introduction to Node.js

동영상 주소: <http://www.youtube.com/watch?v=M-sc73Y-zQA>

주석과 해설: 채수원, blog.doortts.com

Node.js 학습모임인 Octobersky.js 활동의 일환으로 만들어 보았습니다. Ryan Dahl이 JSconf에서 2010년에 발표한 동영상입니다. 라이언 달의 초기 발표라 그런지 엄청 떨면서 이야기를 하는데다가 말도 어버버~ 하는 경우가 많습디만, Node.js를 본격적으로 학습해 볼 생각이 있으시다면 한 번 꼭 보시길 권장하는 동영상입니다. 동영상 보실 때 도움이 될 수 있도록 라이언의 발표 문장에 주석과 해석을 담았습니다. **따라서 동영상과 함께 보시길 강력히 권장합니다.** 동영상 함께 보기 행사는 1회 진행 후 작년 말부터 미뤄오다가 3월 3일에 함께 보기 이벤트를 가졌었고, 후기 겸 정리해 보았습니다. 번역과 주석이 섞여있고 개인적인 의견도 다수 들어있습니다. 너그럽게 봐주세요. :)

<광고!> OctoberSkyJS?

저희 모임은 국내에 아직 이렇다 할 자료가 없던 시절 (2011년 10월)에 멤버들이 합심하여 초보자들도 쉽게 node.js를 공부할 수 있도록 다양한 자료를 만들고 번역해 모으는 일을 하였습니다. 정리한 작업 결과물은 아래 사이트에서 살펴보실 수 있습니다.

Node.js 학습 가이드(One page navigation of studying Node.js)

<https://github.com/octoberskyjs/home>

동참하고 싶으신 분은 모임페이지: <http://www.facebook.com/octoberskyjs> 를 이용해 주세요. :)

node.js

ry@tinyclouds.org

May 5, 2010



node.js is a set of bindings to the V8 javascript VM.

Allows one to script programs that do I/O in javascript.

Focused on performance.



안녕하세요? 전 라이언입니다. node.js는 구글 V8 자바스크립트 가상머신(VM)의 바인딩의 세트입니다. 자바스크립트를 이용해 I/O 스크립트 프로그래밍 할 수 있게 해줍니다. 전 속도를 빠르게 만드는데 집중했습니다.

```
100 concurrent clients
1 megabyte response
```

```
node      822 req/sec
nginx     708
thin      85
mongrel   4
```

(bigger is better)

먼저 이걸 약간 우스운 벤치마킹인데요. 웹 서버에서 유저당 1M의 응답을 보낼 때 100명의 동시사용자가 있다고 가정한 상황입니다. node가 가장 성능이 뛰어납니다. NGINX는 익숙하실 겁니다. NGINX서버는 전체가 다 C로 구현되었으며 이벤트루프에 완전 최적화되어 있습니다. node는 그런 nginx보다도 빠릅니다.

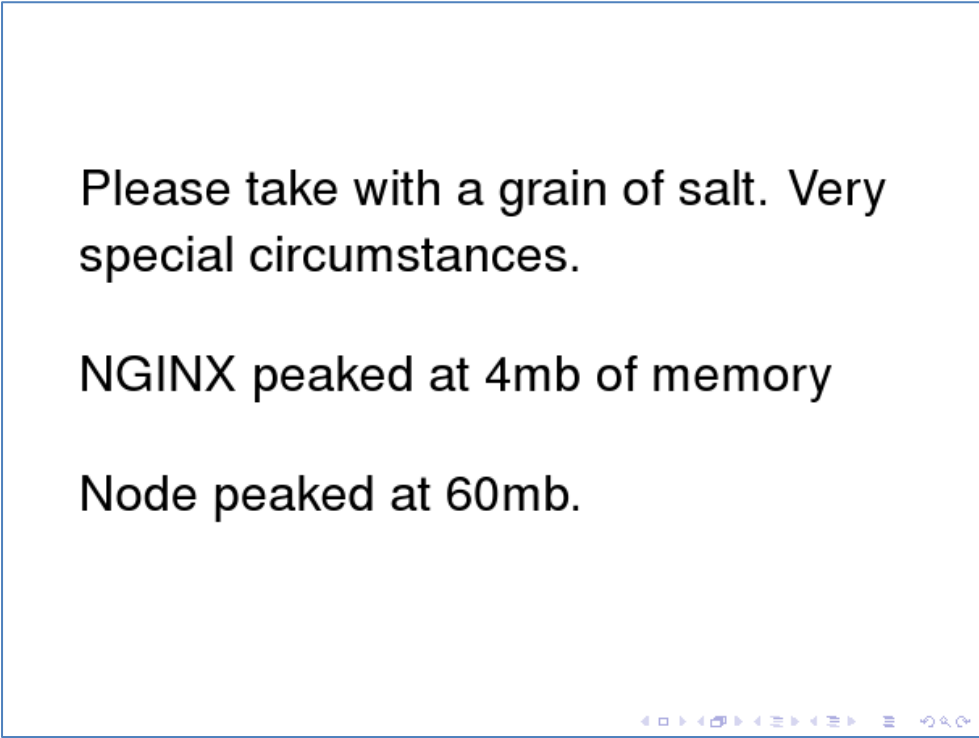
The code:

```
1 http = require('http')
2 Buffer = require('buffer').Buffer;
3
4 n = 1024*1024;
5 b = new Buffer(n);
6 for (var i = 0; i<n; i++) b[i] = 100;
7
8 http.createServer(function (req, res) {
9   res.writeHead(200);
10  res.end(b);
11 }).listen(8000);
```

벤치마크에 사용한 코드입니다. 첫 번째, 두 번째 줄은 필요한 모듈을 포함하는 건데요, CommonJS 모듈 시스템을 사용했습니다. 지금은 http 모듈과 buffer 모듈이 있는데요. buffer는 raw 메모리를 할당합니다. 자바스크립트로는 특히나 잘 안 되는 부분이죠. 8번째 줄에서 11번째 줄까지는 웹서버를 만드는 부분입니다. 이 부분은 콜백으로 되어 있는데요, 콜백은 request가 웹서버로 들어올때마다 매번 호출이 일어나는 부분입니다.

콜백은 200번 Ok http 헤더응답으로 buffer에 있는 내용을 담아보내는 일을 합니다.

마지막줄은 localhost의 8000번 포트로 리스닝을 하도록 만든 부분입니다.



Please take with a grain of salt. Very special circumstances.

NGINX peaked at 4mb of memory

Node peaked at 60mb.

nginx는 4M 메모리를 쓰고 node 60M 메모리를 사용했습니다. 테스트에 사용한 코드는 특화된 프로그램 인데요 항상 메모리에 내용을 덩어리로 채워넣습니다. nginx는 1M짜라 파일을 사용했는데요, (4M 메모리 만 사용하고 있는데요) 파일시스템 polling 등의 뭔가가 더 있었을겁니다. 그럼에도 불구하고 스택파일을 메모리를 이용해 제공했을때 node.js는 최적화 된 상태의 웹서버와 동급의 성능을 보여줍니다.

이와 관련된 사상을 좀 더 이야기 해보겠습니다.

Node.js의 Buffer

<http://nodejs.org/docs/latest/api/buffer.html>

공식문서에 따르면 Buffer는 다음과 같습니다..

“순수 자바스크립트는 유니코드에는 친화적이지만 바이너리 데이터에는 그렇지 못합니다. TCP 스트림이나 파일 시스템을 다루어야 할 때는 옥텟(octet) 스트림을 처리해야 할 필요가 있습니다. 노드는 옥텟 스트림을 소비하고 생성하고 조작하기 위한 몇 가지 전략을 가지고 있습니다. (기본적으로) Raw 데이터는 Buffer 클래스의 인스턴스에 저장됩니다. 하나의 Buffer는 정수 배열과 비슷합니다만 V8 힙(heap)밖의 raw 메모리에 대응됩니다. 하나의 Buffer는 사이즈를 재조정할 수 없습니다.”

Buffer를 쓴건 Buffer 클래스가 메모리를 v8 엔진의 heap 메모리 밖의 native 메모리힙을 사용하기 때문에, 아파치 벤치마크시에 node.js 엔진이 사용하는 메모리영역, 즉 v8 엔진영역의 메모리가 테스트에 관여되지 않도록 하기 위함인 것으로 보입니다.

I/O needs to be done differently

I/O는 다른 식으로 다루어져야 합니다.. 우리는 I/O를 완전히 잘못 쓰고 있습니다. 어떤 단계에 이르면 엉망이 되죠. 이걸 우린 좀 다르게, 그리고 이전 보다 좀 더 쉽게 할 수 있습니다. Node.js 프로그램은 I/O 처리를 얼마나 쉽고 이상적인 형태로 할 수 있는지 보여주는 어떤 시도입니다.

많은 웹 애플리케이션들이 아래와 같은 코드를 가집니다.

```
result = query('select * from T');  
// use result
```

많은 애플리케이션, 특히 웹 애플리케이션은 일반적으로 이런 데이터베이스에 액세스하는 코드를 작성합니다. 데이터베이스는 자신의 컴퓨터일수도 있고, Los Angeles에 있을 수도 있죠. 어쨌든 I/O가 일어납니다. function으로 추상화해서 I/O를 실행하고 호출하면 L.A.로부터 함수의 결과를 받습니다.

여기서의 의문점은, 이 쿼리함수가 작업하는 동안 소프트웨어는 뭐하고 있냐라는 거죠.

In many cases, just waiting for the response.

많은 경우, 클라이언트는 가만히 앉아서 서버의 응답을 기다리며 아무것도 안합니다.

Modern Computer Latency

L1: 3 cycles

L2: 14 cycles

RAM: 250 cycles

DISK: 41,000,000 cycles

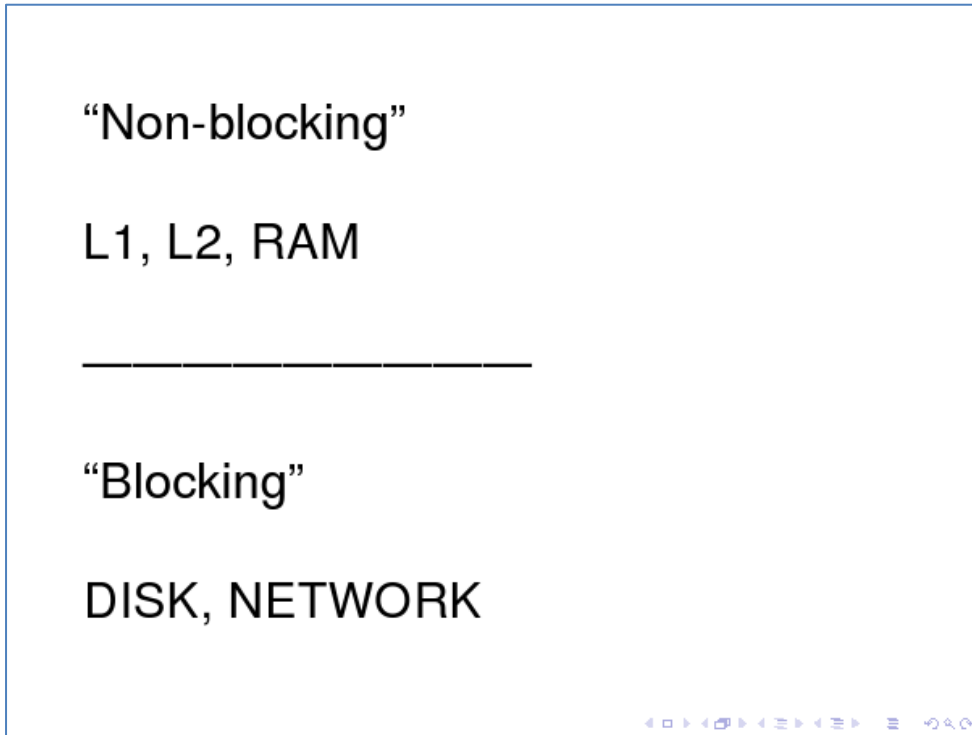
NETWORK: 240,000,000 cycles

Modern Computer Latency

(CPU에 들어 있는 캐시메모리인) L1, L2는 굉장히 빠릅니다. 몇 몇 사이클이면 됩니다. 디스크나 네트워크

쪽으로 가보면 수 백만 사이클이 필요합니다. 따라서 위쪽 3개를 다루는 방법과 아래쪽의 DISK와 네트워크를 동일한 방법으로 다루어서는 안됩니다. 이것들은 근본적으로 다릅니다.

이건 인터넷에서 들은 이야기입니다만, CPU에 접근하는 건 책상으로 가서 서랍을 열어서 서류 몇 장 꺼내는 거고 RAM은 복도를 지나가서 다른 방의 누군가랑 얘기 좀 하다 뭔가 들고 오는 거고, 디스크에 접근하는 건 비행기를 타고 도쿄에 가서 서류를 받아오는 것과 같습니다.



일반적으로 함수중 L1, L2, RAM에 접근하는 함수는 non-blocking, 디스크 네트워크는 blocking 함수가 됩니다.

```
result = query('select * from T');  
// use result
```

BLOCKS

위와 같은 코드를 볼 때 '이건 블락!'이라는걸 인지하는 것이 굉장히 중요합니다.

Better software can multitask.

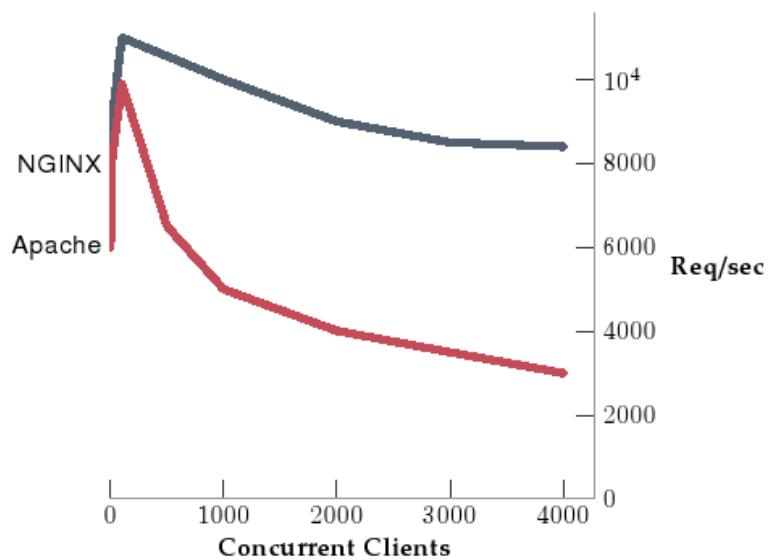
Other threads of execution can run
while waiting.

모든 소프트웨어가 호출될 때 블럭킹을 만들진 않습니다. 블럭킹 하는 함수도 멀티태스킹으로 개선될 수 있죠. 멀티스레드 환경에서는 하나가 블럭당해도 다른 스레드로 전환해서 다른 일들을 계속 진행할 수 있습니다.

Is that the best that can be done?

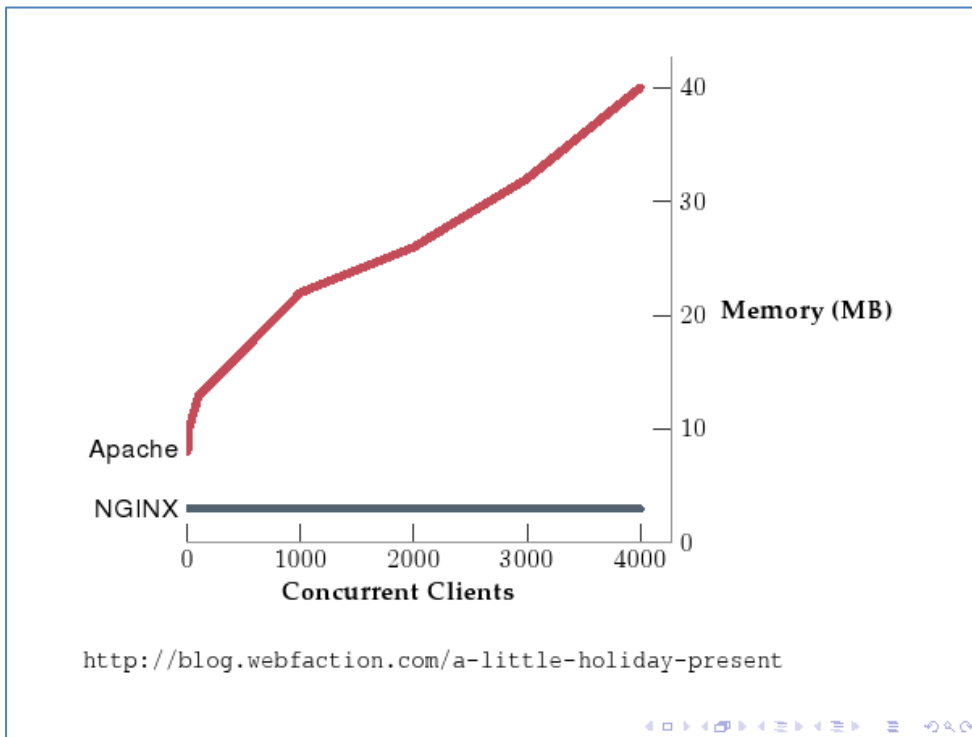
A look at Apache and NGINX.

그런데 다른 스레드를 사용하는 것, 이게 우리가 할 수 있는 최선인가요? 아파치와 NGINX를 한번 보죠.



<http://blog.webfaction.com/a-little-holiday-present>

인터넷에 구한 벤치마크 자료입니다. 가로축은 동접 클라이언트의 숫자이고요, 세로축은 초당 리퀘스트 처리량인데요 높을 수록 좋습니다. NGINX는 아파치의 거의 3배를 처리합니다. 이번엔 메모리 사용량을 살펴보겠습니다.



NGINX는 메모리 측면에서 매우 안정적입니다. 아파치는 클라이언트가 늘수록 메모리 사용량도 계속 늘어납니다. 만약 수 많은 사용자들이 여러분의 웹 서버에 머물고 있다면, 이를테면 채팅 서비스 같은 걸로 말이죠, 이런 경우 사실 뭔가 특별한 일을 하진 않을 겁니다. 즉, 때때로 idle 상태가 될 테지만 (메모리 사용량을 놓고 봤을 때) 아파치는 그런 경우 처리에 한계가 있습니다. 계속 각각에 대해 스레드를 만들기 때문입니다.

Apache vs NGINX

The difference?

Apache uses one thread per connection.

NGINX doesn't use threads. It uses an **event loop**.

아파치와 NGINX의 진정한 차이점은 아파치는 각각의 커넥션에 스레드를 사용하고 NGINX는 이벤트 루프(event loop)를 사용한다는 점입니다.

- ▶ Context switching is not free
- ▶ Execution stacks take up memory

For massive concurrency, **cannot** use an OS thread for each connection.

- **컨택스트 스위칭은 공짜가 아니다.**

- 실행스택은 메모리를 차지한다.

스레드간의 컨텍스트 스위칭은 비용이 들고 느리게 만듭니다. 물론 컨텍스트 스위칭이 빠르긴 합니다. 스케줄러가 매우 빠르게 동작지만 어쨌든 비용은 듭니다. 스레드는 (보통) 메모리를 메가바이트로 사용합니다. 거대 규모의 동시성 처리를 해야할 때 각각의 연결을 위해 OS스레드를 사용할 수 없다는 건 누구나 아는 확실한 사실입니다..

Green threads or coroutines can
improve the situation dramatically

BUT there is still machinery involved
to create the **illusion** of holding
execution on I/O.

그린스레드나 코루틴은 이런 상황을 개선할 수 있습니다.

사실 그린쓰레드는 가상머신으로 구동되는 쓰레드라 OS 쓰레드의 한계를 넘을 수는 있지만, 성능측면에서 +-가 있습니다. 라이언도 여기서 설명하는 게 그린쓰레드에서 IO가 블럭될때 발생하는 문제점, 모든 쓰레드가 stop될수 있다는 것에 대해 이야기하네요. 참고로 그린쓰레드가 잘 적용되었다고 보는 언어가 얼랭(erlang)의 그린 프로세스라고 들었습니다. 예전에 고부하 상황 처리 안정성에서 아파치를 넘어섰다는 이야기도 그것의 일환으로 보입니다. :D

그린 스레드를 사용한다고 해도 여전히 머신 의존적인(machinery) 문제가 남아있습니다. 이를테면 DB를 생성하는 함수를 호출하면 여러분의 프로그램은 멈춥니다. 전체 프로그램이 멈추는 건 아니지만 해당 진행되던 프로그램은 멈춥니다. 멈추고는 아무것도 안 하게 될 수 있습니다. 안 좋은 점은 단지 멈추기만 하는 것이 아닙니다. 여러분의 프로그램이 다른 일들을 하려면 IO 실행을 잠시 멈춘 것에 대한 일류전(illusion, 멀티스레드에서 이야기하는 진행상태)을 만들어야 합니다. 비용이 들죠.

Code like this

```
result = query('select..');  
// use result
```

either **blocks the entire process** or
implies **multiple execution stacks**.

이 코드는 전체 프로세스를 블록하거나 복수개의 실행 스택을 가지고 있다는 걸 의미합니다. 그럼 이번엔 다음과 같은 코드가 있다고 가정해 보겠습니다.

But a line of code like this

```
query('select...', function (result) {
    // use result
});
```

allows the program to return to the event loop immediately.

No machinery required.

인터프리터가 즉 실행하다가 이 부분에서 쿼리를 쿼리를 실행하고 콜백을 지정한 다음 이하 계속 진행합니다. LA에서 응답을 오는걸 기다릴 필요가 없습니다. 결과가 뭐가 되었든 커널은 결과를 버퍼로 보냅니다. 여기에는 머신 의존적인 부분이 거의 없습니다. 단지 콜백을 위한 포인터만 잡고 있으면 됩니다. 커널

에게 결과가 나오면 알려달라고 말해놓으면 되는 거죠.

```
query('select..', function (result) {  
  // use result  
});
```

This is how I/O should be done.

이것이 바로 I/O가 어떤 식으로 동작해야 하는 건지를 보여주는 예입니다.

그런데 사람들이 이벤트 루프를 사용하지 않는 이유는 뭡까요?

So why isn't everyone using event loops, callbacks, and non-blocking I/O?

For reasons both **cultural** and **infrastructural**.

여러분이 이렇게 하지 않는 데는 매우 그럴듯한 이유가 있습니다. 사실 여러 가지 이유가 있죠.

Cultural Bias

We're taught I/O with this:

```
1 puts("Enter your name: ");  
2 var name = gets();  
3 puts("Name: " + name);
```

We're taught to demand input and do nothing until we have it.

문화적인 편견

이 부분에 대한 주해는 "왜 Node인가? <http://blog.doortts.com/219>"라는 좋은 글이 번역되어 있으니 그걸로 대체합니다. :)

Cultural Bias

Code like

```
1 puts("Enter your name: ");
2 gets(function (name) {
3     puts("Name: " + name);
4 });
```

is rejected as too complicated.

이렇게 작성해야 하면, '복잡해서 안돼요. 스파게티코드네요.'라는 식으로 거부한다네요.

Missing Infrastructure

So why isn't everyone using event loops?

Single threaded event loops require I/O to be non-blocking

Most libraries are not.

진짜 더 큰 문제는 기반구조가 없다는 겁니다.

만약 싱글스레드에 이벤트 루핑을 사용한다고 가정해 보겠습니다. 이런 경우 다음과 식으로 작업하게 되

mysql 클라이언트를 재 작성해야 하는 것처럼 말이죠

But users are confused how to combine with other available libraries.

◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡ 🔍

Ryan Dahl: Introduction to Node.js 동영상 해설

Too Much Infrastructure

Users still require expert knowledge of event loops, non-blocking I/O.

그리고 이벤트머신 서버를 사용해도 여전히 전문적인 지식이 필요해요

이게 사소한 것일 순 있지만 루비에 기본으로 탑재된 mysql 쿼리 라이브러리도 블록!입니다.

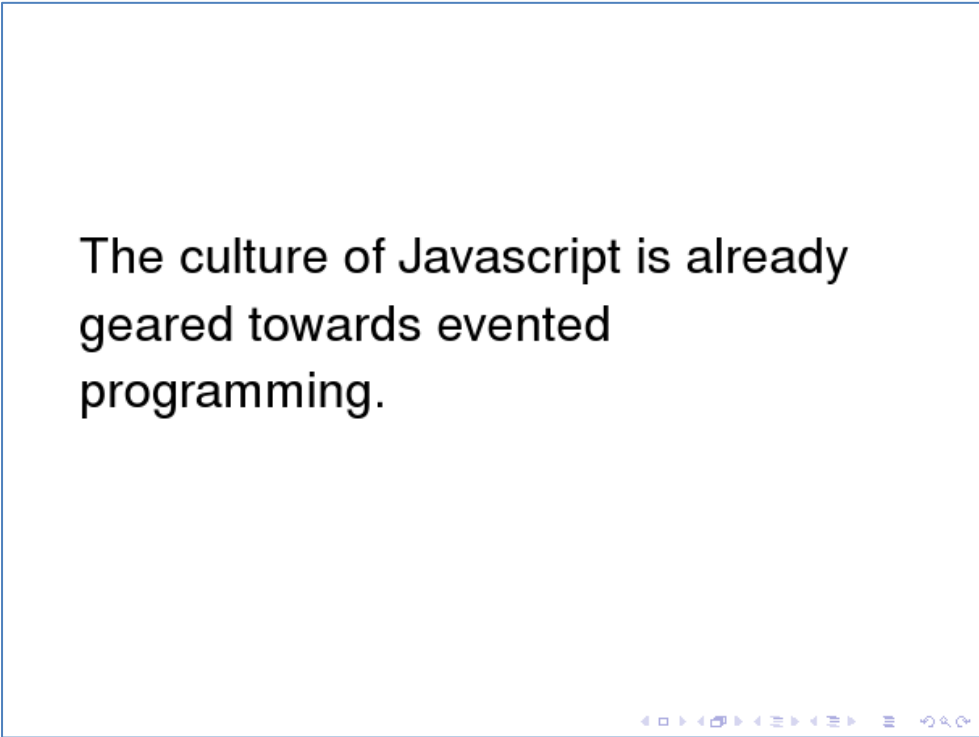
Javascript designed specifically to be used with an event loop:

- ▶ Anonymous functions, closures.
- ▶ Only one callback at a time.
- ▶ I/O through DOM event callbacks.

브라우저 자바스크립트는 진짜 잘 되어 있어요. 놀라울 정도로 잘 되어 있습니다. 특히 이벤트 루프를 다

루기 좋게 설계되어 있습니다. 익명함수, 클로저 같은 언어 구조가 바로 그런 요소입니다. 브라우저 자바스크립트를 놓고 봤을 때, 프로그래밍 모델 측면에서 정말 아름답다고 여겨지는 부분은, 쉽게 콜백을 만들 수 있다는 겁니다. 누군가가 버튼을 누르면 즉시 버튼 콜백이 일어납니다. 그리고 한 번에 한가지 콜백만 일어나죠.

누구도, 어떤 브라우저 사용자도 이벤트 루프나 비블리킹 I/O에 대해 들어본 적이 없습니다. 이런 컨셉이 웹사이트 프로그래밍을 하는 데에 필요가 없습니다. 왜냐하면 매우 제한된 환경내에서 특정 종류의 I/O만 사용하도록 제한되어있기 때문입니다.



The culture of Javascript is already geared towards evented programming.

자바스크립트의 문화는 이미 이벤트 프로그래밍에 맞게 잘 설계되어 있습니다.

This is the **node.js** project:

To provide a **purely evented,**
non-blocking infrastructure to
script **highly concurrent** programs.



(노드 이야기가 이제야 다시 나오네요) 높은 수준의 동시성 프로그램을 스크립트를 작성하는 것 만으로 처리 가능한 순수 이벤트 방식의 언 블러킹 인프라스트럭처를 제공하는 것. 그것이 바로 node.js 프로젝트 입니다.

Design Goals



Design Goals

No function should directly perform I/O.

To receive info from disk, network, or another process **there must be a callback.**

설계 목표

- 어떤 함수도 I/O를 직접 수행해선 안 된다.
- 디스크나 네트워크, 혹은 다른 프로세스로부터 정보를 받으려면 반드시 콜백이 있어야 한다.

이를 테면

b = a()

와 같이 작성하지 않고 node.js에서는

```
a( function(b) {  
    ...  
});
```

로 만들도록 유도하는데, 이렇게 하면 가시적으로 어디서 I/O 호출이 일어나는지 알 수 있게 됩니다. 한 마디로 콜백을 강제화 시켜놓았습니다..

Design Goals

Low-level.

Stream everything; never force the buffering of data.

Do not remove functionality present at the POSIX layer. For example, support half-closed TCP connections.



설계 목표

- 모든 것이 스트림. 버퍼링하도록 강제하지 않는다.
- POSIX(유닉스 운영체계에 기반을 두고 있는 일련의 표준 운영체제 인터페이스) 레이어에서 제공되는 기능을 제거하지 않는다. 이를테면 half-closed TCP 연결지원 같은 기능을 제거하지 않았습니다.

Design Goals

Have built-in support for the most important protocols:

DNS, HTTP, TLS

(Especially because these are so difficult to do yourself)

설계목표

- 가장 중요한 프로토콜들을 위한 지원을 내장한다.

솔직히 인터넷은 ip가 다가 아니죠.

DNS, HTTP, TLS, 이 세개는 매우 중요합니다. 다 알아야 하죠. 인프라스트럭처에 가까운 프로토콜입니다.

인터넷의 근간이되는 프로토콜입니다. 이들에 대한 훌륭한 라이브러리를 만드는 것입니다. 사실 HTTP는 극도로 어렵고요. DNS 또한 그렇습니다.

node.js를 이용하면 잘 쓸 수 있습니다.

Design Goals

Support many HTTP features.

- ▶ Chunked encoding
- ▶ Pipelined messages
- ▶ Hanging requests for comet applications.

설계 목표

- 수 많은 HTTP 피쳐, 즉 HTTP의 스펙으로 정한 기능들을 지원한다.

사실 저는 살짝 HTTP 너드(nerd)입니다. 그래서 HTTP의 코어 피쳐들을 지원할 수 있게 만들었습니다. Pipelined message 같은 경우엔 실제 웹서버들이 지원하는 기능들입니다. 채팅 같은 걸 만든다면 리퀘스트를 잡아 놓고 대기할 수 있어야 하는데 아시다시피 일반적인 웹 서버 만으로는 영망이 되기 쉽습니다.

Design Goals

The API should be both familiar to **client-side JS programmers** and **old school UNIX hackers**.

Be platform independent.



설계 목표

- API는 클라이언트 자바스크립트 개발자들과 올드 스쿨 유닉스 해커들 모두에게 친숙해야 한다.
- 플랫폼 독립적이다

자바스크립트 개발자와 서버개발자가 타임아웃에 대해서도 동일하게 느낄 수 있게 `setTimeout`이 `node.js` 에도 동일한 이름으로 있는거군요. :D

Design Goals

Simply licensed (Almost 100% MIT/BSD)

Few dependencies

Static linking



설계목표

- 단순한 라이선스
- 의존성이 적음
- 정적으로 링크함

Opensni에 걸린 라이선스만 제외하고 나머지는 마음대로 쓸 수 있는 MIT 혹은 BSD 라이선스 입니다.
openssl 은 자체 라이선스이지만 아파치 라이선스와 유사합니다.

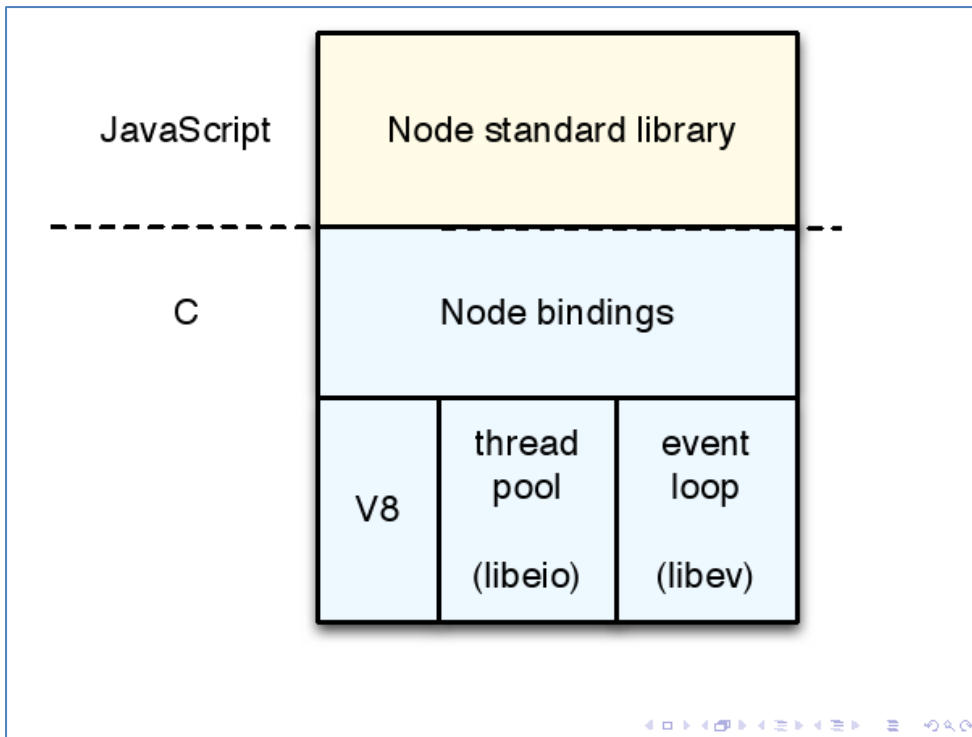
Design Goals

Make it enjoyable.



Architecture





Node.js의 아키텍처

v8 = 자바스크립트 실행

스레드 풀은 I/O(=file) 관련 작업을 수행합니다. libeio에 대해서는 다음 링크를 참고하세요.

libeio?

이벤트 기반의 모든 게 비동기로 동작하는 C언어용 I/O 라이브러리. 기본적으로 POSIX API에 기반을 두고 있으며 파일 처리 관련 작업을 한다. read, write, open, close, stat, unlink, fdasync, mknod, readdir 등의 작업을 비동기로 처리한다.

참고: <http://software.schmorp.de/pkg/libeio.html>

이벤트루프(libev)는 기본적으로는 select 래퍼입니다.

libev?

다양한 기능을 가진 고성능 이벤트 루프 라이브러리. libevent를 따라 느슨하게 모델링 되었지만 이전에 존재했던 버그나 제한사항이 없다. 프로세스 감시, 절대시각에 기반한 주기적 타이머, 그리고 epoll/kqueue/event ports/inotify/eventfd/signalfd등을 지원한다. (POSIX라이브러리입니다. :)

참고: <http://software.schmorp.de/pkg/libev.html>, <http://www.ibm.com/developerworks/kr/aix/library/au->

libev/

select?

동기적 I/O 입출력 다중화에 사용하는 POSIX함수

참고: <http://www.ibm.com/developerworks/kr/aix/library/au-libev/>

Node bindings는 C와 자바스크립트 바인딩 하는 부분이고요.

그리고 그 위는 자바스크립트로 이루어진 노드 기본 라이브러리 영역입니다.

The JavaScript layer can access the
main thread.

The C layer can use multiple threads.

자바스크립트 레이어가 메인 스레드이고 한 개의 실행스택을 가집니다. 하지만 C 레이어에서는 여러개의 스레드를 사용할 수 있습니다. 이를테면 gzip 라이브러리를 쓸 때는 5개의 스레드로 포크해서 동시에 호출할 수 있습니다. 하지만 이건 C를 사용했을 때에 한정해서 입니다.

Deficiency? No. Feature.

Threads should be used by experts only.



부족한가? 아니오. 피쳐(=스펙으로 정한 기능)입니다.

싱글 스레드라 부족한가? 라는 질문이죠. 부족하지 않다는 건 싱글 스레드라는 것 자체가 노리고 정한 스펙이란 뜻입니다. 뭐, 정말 그런건지는 모르겠지만 말입니다.

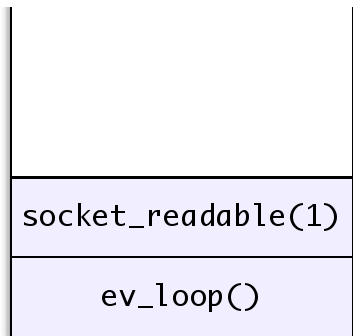
멀티 스레드는 전문가들만 사용해야 합니다. 뭐 사람들이 이런 제 말을 믿지 않겠지만 말입니다. **‘전 전문가인데요’**라고 말하는 사람이라면 C로 작성하세요. 그럼 됩니다. 그게 적절한 경계점입니다.

There is exactly one execution stack in Node.

노드에는 오직 한 개의 실행 스택만 있습니다.

ev_loop() 이건 select 인척 하는 이벤트 루프 콜입니다.

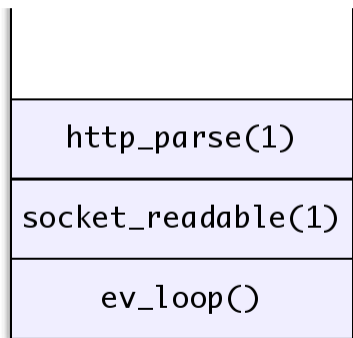
Node execution stack



누군가가 웹서버에 접속하면 select가 반환됩니다. 콜백을 얻게 되죠. 소켓이 이야기 합니다.

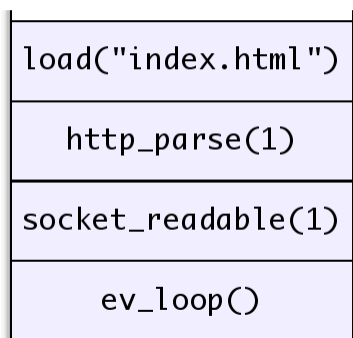
“1번 소켓이 읽기가능상태임!”

Node execution stack



그럼 데이터를 읽어 들어서 구문분석(parsing) 작업을 합니다.

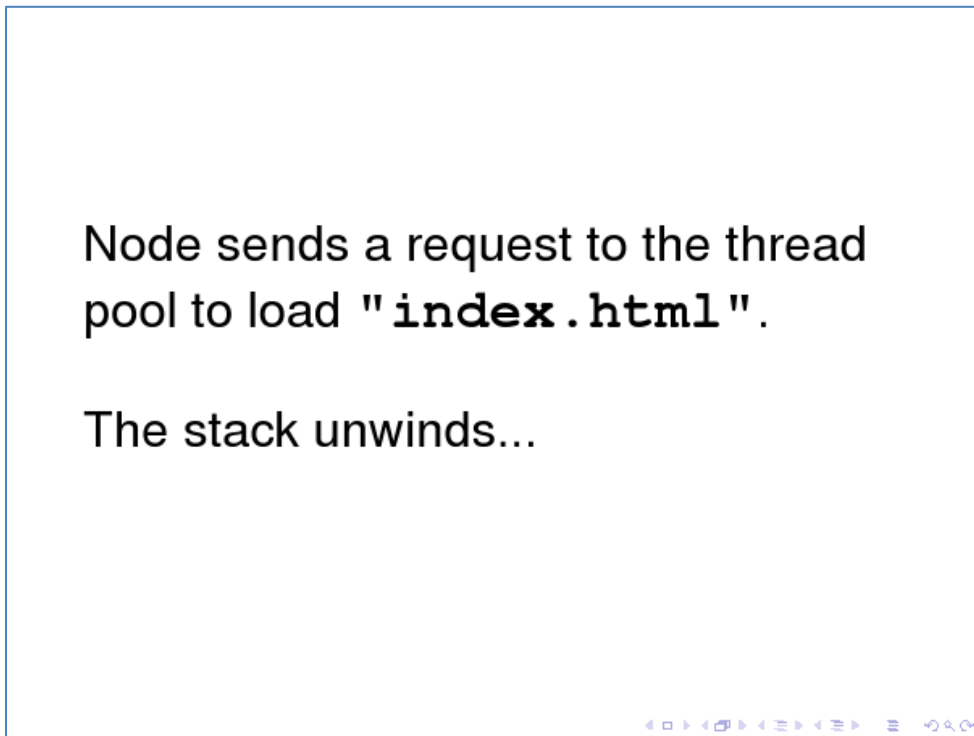
Node execution stack



살펴 봤더니 http 요청은 index.html 에 대한 요청이었습니다.

요청받은 내용이 메모리에는 아직 없네요. 어쩔 수 없이 하드 디스크를 뒤져서 메모리로 읽어 들여야 합

니다. `load("index.html")`부분이 해당 부분입니다.

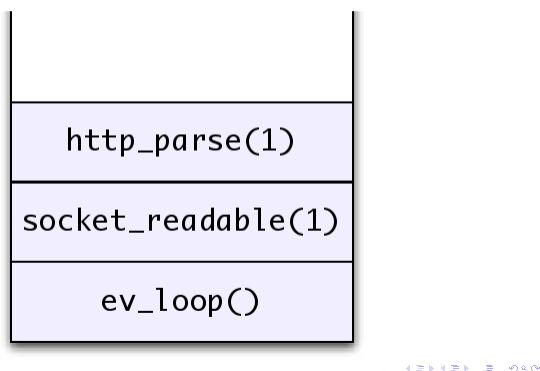


노드는 스레드로 이와 같은 내용을 넘기면서

"오케이~ 스레드! `index.html`을 로드해 줄래? 그리고 다 되면 알려줘"

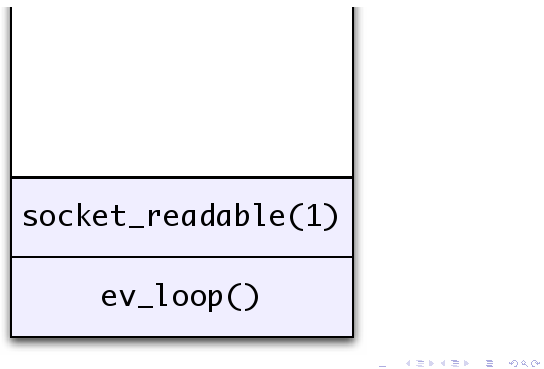
그리고는 스택을 되감습니다. 이벤트 루프 상태로 롤백 하는 거죠.

Node execution stack



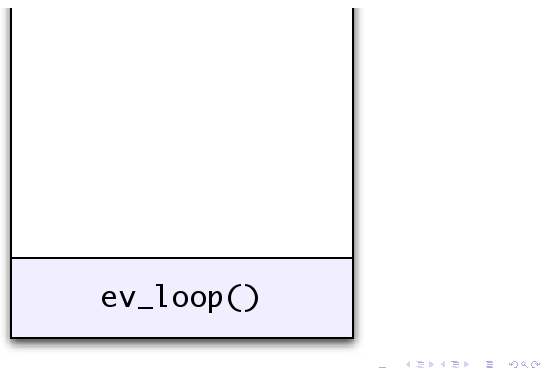
스택(stack)이니까 하나씩 꺼내 가면서 롤백!

Node execution stack



롤백!

Node execution stack



ev_loop 상태까지 롤백!

The request is sent to the disk.

Millions of clock cycles will pass
before the process hears back from
it.



이제 디스크로 요청이 보내지고 대기(idle) 상태가 됩니다. 수백만 클럭 사이클 후에 스레드로부터 응답을 받게됩니다. 그러니까 아주아주 나중에 말이죠

In the meantime someone else
connects to the server.

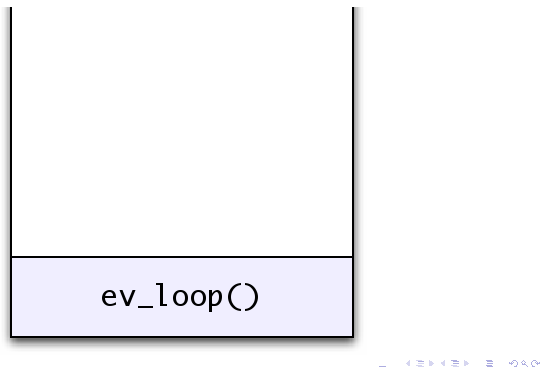
This time requesting an in-memory
resource.



반면에 다른 누군가가 서버에 접속합니다.

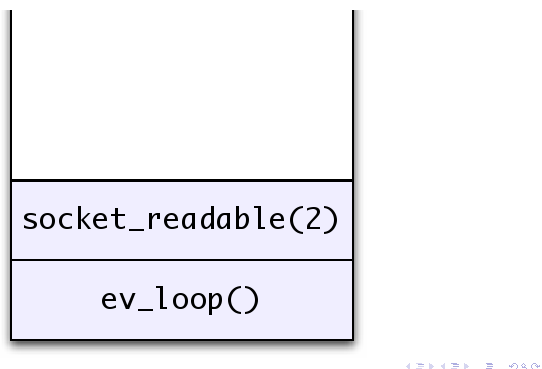
이번에는 메모리에 이미 있는 무언가 간단한 걸 요청한다고 가정해 보겠습니다.

Node execution stack



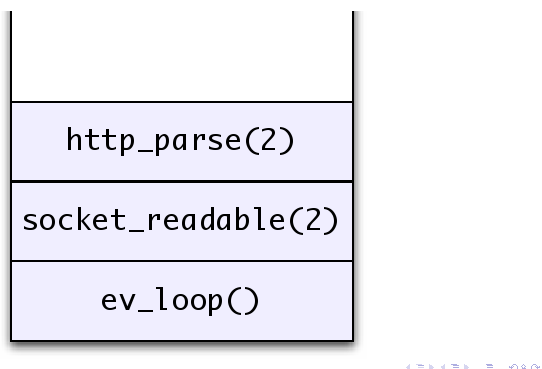
이벤트 루프에 들어가고

Node execution stack



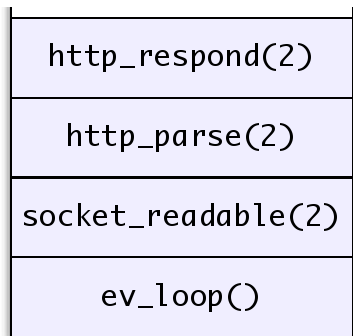
호출하고. 오! 소켓2번이 사용가능하네요

Node execution stack



리퀘스트를 파싱해서

Node execution stack

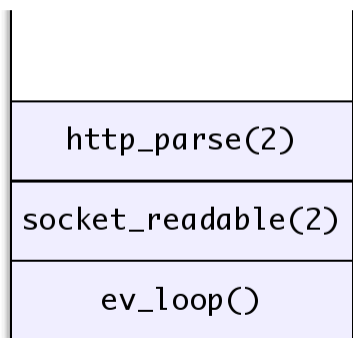


바로 응답을 합니다.

메모리에 있기 때문에 바로 보낼 수 있습니다.

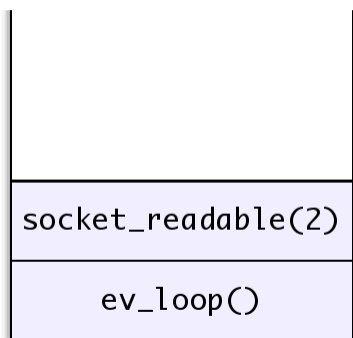
두 번째 웨이브죠. 스택을 다시 되감습니다.

Node execution stack



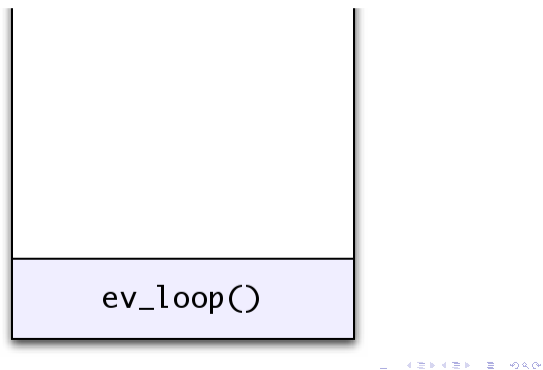
롤백!

Node execution stack



웃싸! 롤백!

Node execution stack



웃싸! 롤백!

The process sits idle.

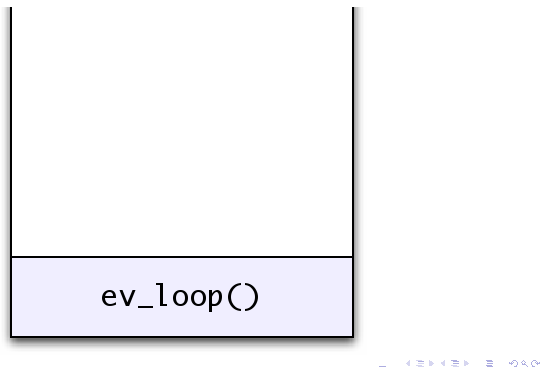
The first request is still hanging.

Eventually the disk responds:

프로세스는 다시 idle 상태로 대기합니다.

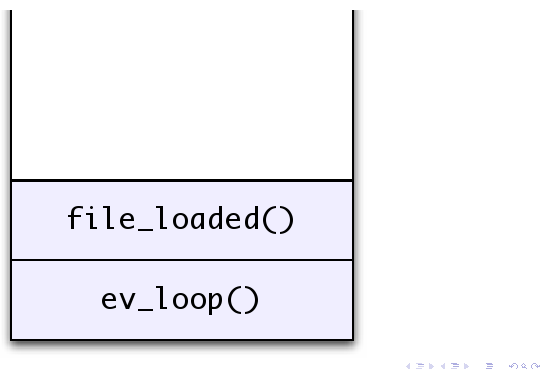
웹사이트로 첫 번째 요청을 했던 사람은 여전히 기다리고 있죠. 그런데 이건 매우 짧은 시간의 일입니다. 길지 않죠. (시간이 흘러) 드디어 디스크로부터 응답을 받습니다.

Node execution stack



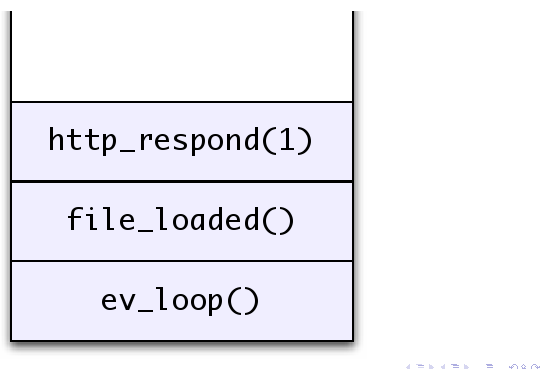
이벤트 루프 상태

Node execution stack



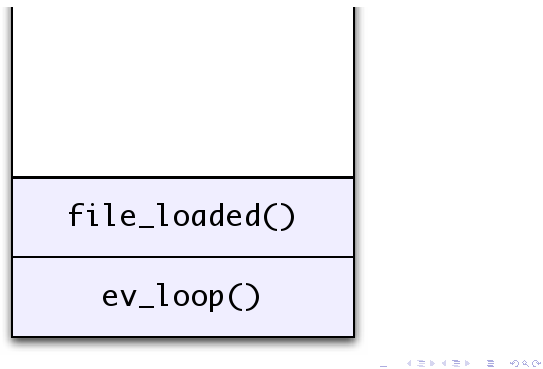
콜백이 일어납니다. '어! 이제 `index.html`이 메모리에 다 로드 되었어요! 여깁어요'

Node execution stack



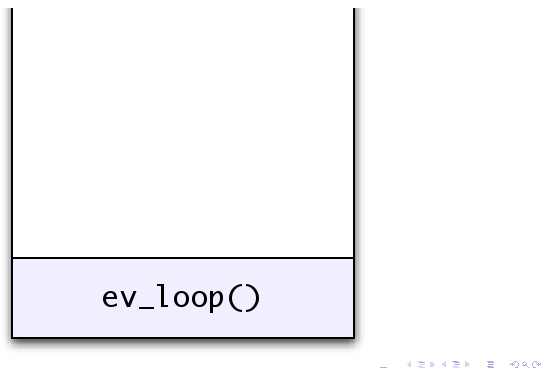
포인터가 위치하고, 첫 번째 요청에 응답합니다.

Node execution stack



롤백!

Node execution stack



끝! 다시 이벤트 루프상태로.

단 하나의 스택으로 다 처리 했습니다. 매우 간결합니다.

(이어지는 설명요약) 코루틴으로 하려면 매우 복잡해집니다.

(using node 0.1.93)

`http://nodejs.org/`

No dependencies other than Python for the build system. V8 is included in the distribution.

V8엔진은 배포판에 포함되어 있습니다.

```
1 var sys = require('sys');
2
3 setTimeout(function () {
4     sys.puts('world');
5 }, 2000);
6 sys.puts('hello');
```

A program which prints “hello”, waits 2 seconds, outputs “world”, and then exits.

require로 sys 모듈을 읽어들입니다. 3번째 줄에서 5번째 줄 사이의 내용은 2000밀리초, 즉 2초 후에 world를 출력하는 타이머입니다. 6번째 줄은 hello를 출력합니다. 즉 hello 출력하고 2초 뒤에 world를 출력합니다.

```
1 var sys = require('sys');
2
3 setTimeout(function () {
4     sys.puts('world');
5 }, 2000);
6 sys.puts('hello');
```

Node exits automatically when there is nothing else to do.

더 이상 할 일이 없으면 자동으로 프로그램을 종료합니다.

```
% node hello_world.js  
hello
```

2 seconds later...

```
% node hello_world.js  
hello  
world  
%
```

A program which:

- ▶ starts a TCP server on port 8000
- ▶ send the peer a message
- ▶ close the connection

다음 프로그램은 8000번 포트를 사용하는 TCP서버를 시작합니다.

피어측에 메시지를 보내고 커넥션을 닫습니다.

```
1 net = require('net');  
2  
3 var s = net.createServer();  
4 s.addListener('connection',  
5     function (c) {  
6         c.end('hello!\n');  
7     });  
8  
9 s.listen(8000);
```

간단하죠.

File I/O is non-blocking too.
(Something typically hard to do.)

File I/O도 non-blocking입니다. 때때로 그렇게 만들기 어렵습니다만 node.js는 스레드풀을 내장해서 사용합니다.

As an example, a program that outputs the last time `/etc/passwd` was modified:

```
1 var stat = require('fs').stat,  
2     puts = require('sys').puts;  
3  
4 stat('/etc/passwd', function (e, s) {  
5     if (e) throw e;  
6     puts('modified: ' + s.mtime);  
7 });
```

이번 예제는 `etc/passwd`의 최종 수정시간을 찾는 프로그램입니다. 1번째 줄에서 `fs`, 즉 파일시스템 모듈을 읽어들이고 `sys` 모듈을 읽어 들여, 각각 `stat` 함수와 `puts` 함수를 지정합니다. 4번째 줄에서 `stat` 함수를 이용해서 `/etc/passwd` 파일의 상태를 확인하는데요, 해당 결과를 콜백 함수로 넘깁니다. 콜백 함수에 해당하는 `function(e, s)`에서 `e`는 에러를 `s`는 `stat` 함수의 실행결과입니다. 보통은 인자 `e`에 `null` 값이 들어갑니다.

Road Map

Now:

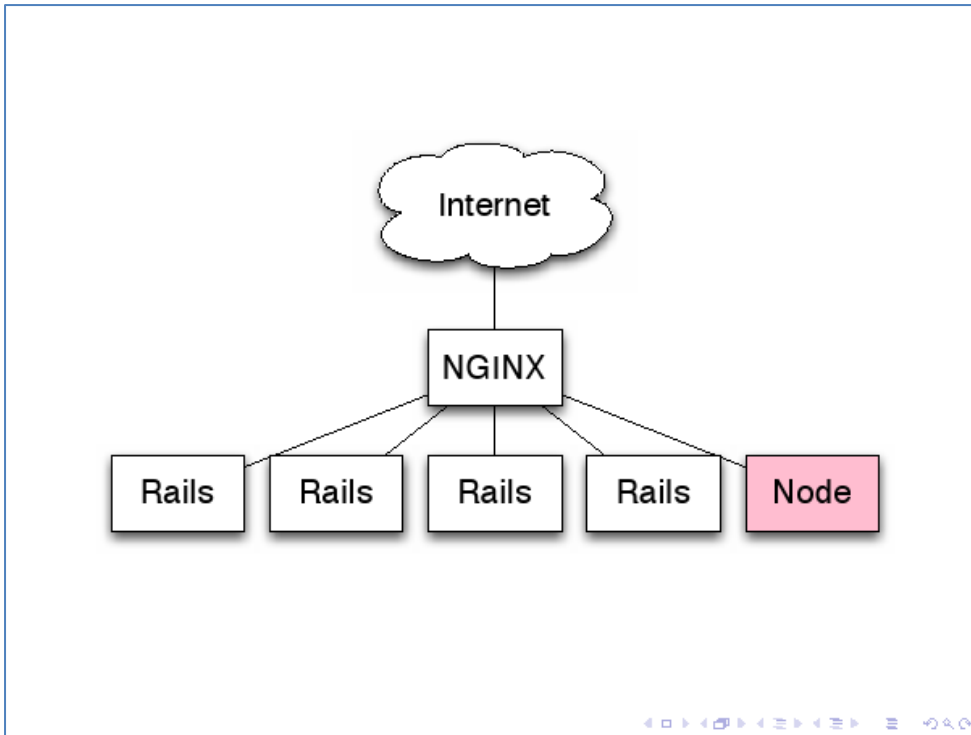
Solution for real-time problems along
side traditional stack.

(long poll, WebSocket)

현재는 전통적인 스택(아파치, NGINX등)과 함께 사용하는 리얼타임 문제에 대한 솔루션입니다. Long poll
이나 웹소켓같은. (참고: 리얼타임 웹을 위한 기초 기술들 <http://blog.doortts.com/229>)

현재는 아파치 같은 서버들이 잘 지원하지 못하는 채팅 프로그램 같은 것에 사용됩니다.

프로세스를 따로 분리해서 node.js가 처리하도록 만듭니다.



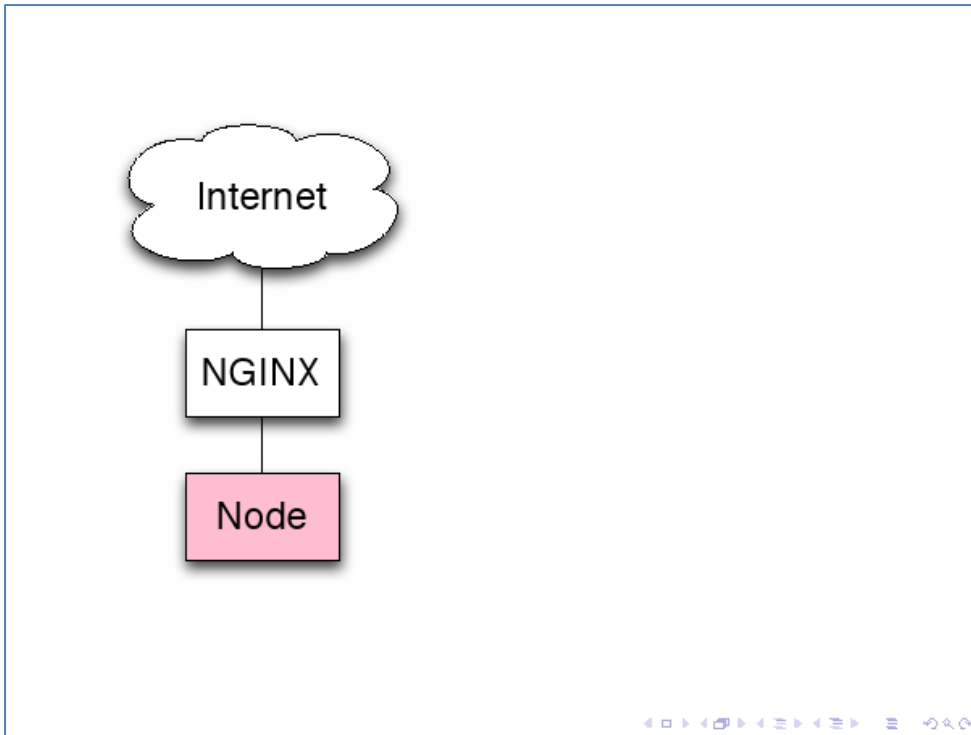
NGINX를 노드 밸런서로 사용하는 방식이고요.

As frameworks built on Node mature,
use it for entire websites.

For security set it behind a stable
web server.

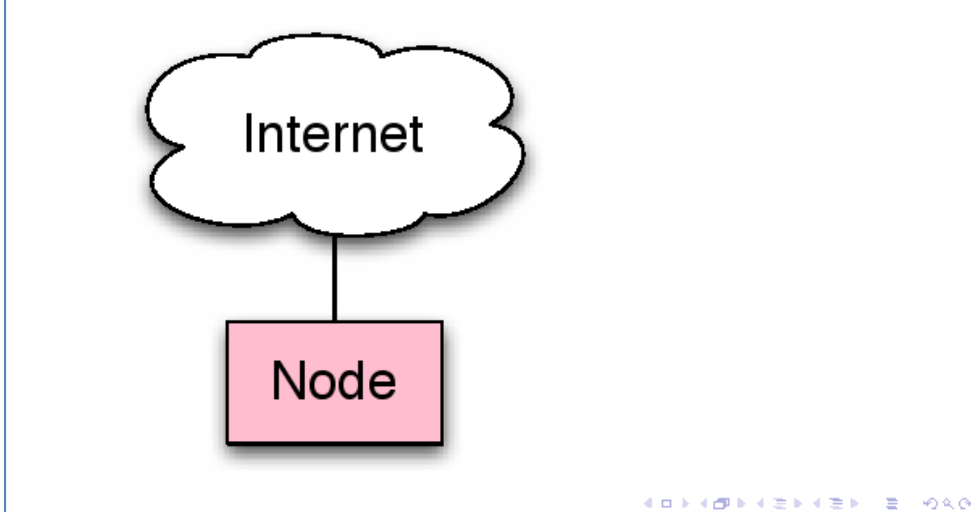
Load balancing unnecessary.

매번 이렇게 작성할 수는 없으니까 프레임워크가 필요할 겁니다. 향후 노드가 성숙해지면 노드 기반의 프레임워크가 전체 웹사이트에 사용될 겁니다. 보안적인 문제로 안정적인 웹 서버 뒤에 놓을 수도 있습니다. 로드 밸런싱이 불필요 합니다.



이런 식의 그림이 되겠죠. (실제로 express 같은 프레임워크를 쓰면 위 말처럼 구성은 가능합니다만....)

When Node is stable, remove the front-end web server entirely.



노드는 리퀘스트 핸들링에 굉장히 강합니다.

언젠가 노드가 성숙해지고 사용하는데 익숙해지면, 스크립트 하나 작성해서 80port로 바로 서비스 할 수 있을 겁니다. 지금은 어색하게 느껴질테지만 말입니다.

Questions...?

`http://nodejs.org/`

`ry@tinyclouds.org`



질문은 위 주소로 보내면 안되어요! 이젠 `nodejs.org`를 이용해야죠.

자! 동영상과 함께 보셨나요? 아니면 이 문서만 보셨나요? 이 문서만 보시면 안된다니깐요!! 꼭 동영상보시고요, 즐거운 `node.js` 라이프가 되세요!

혹시 기타 의견 있으시거나 질문 있으시면 아래 주소로 메일 주시고, 기간한정 `node.js` 학습공유 모임인 `OctoberSkyJs`도 많이 찾아주세요! :D

`doortts@Gmail.com`

감사합니다.

작성: 채수원

버전: v.1.0

일자: 2012.03.09