THE TAPIR'S TALE

PROGRAMMING, PHILOSOPHY, RECURSION, INFINITY AND THE MIND.

ANDERS JANMYR

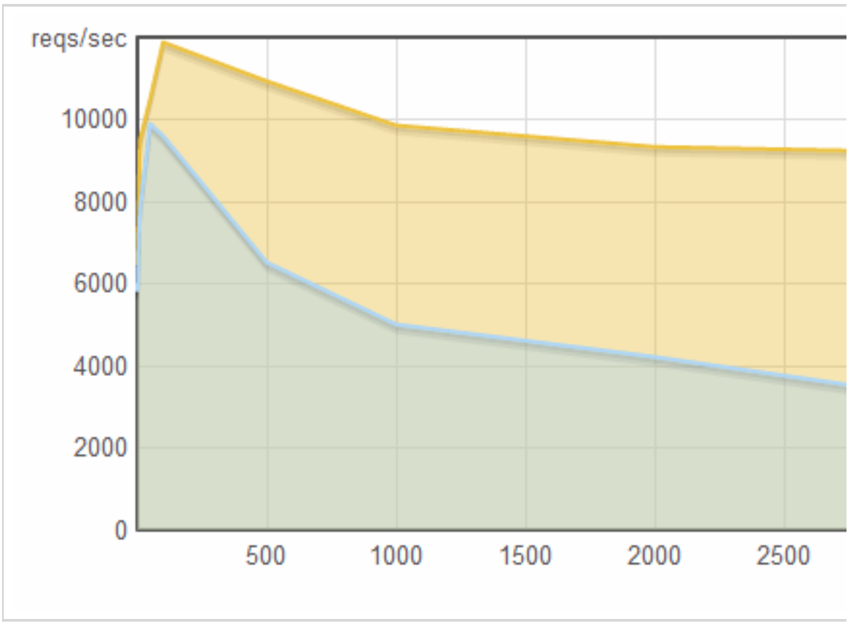
@andersjanmyr

VIEW MY COMPLETE PROFILE

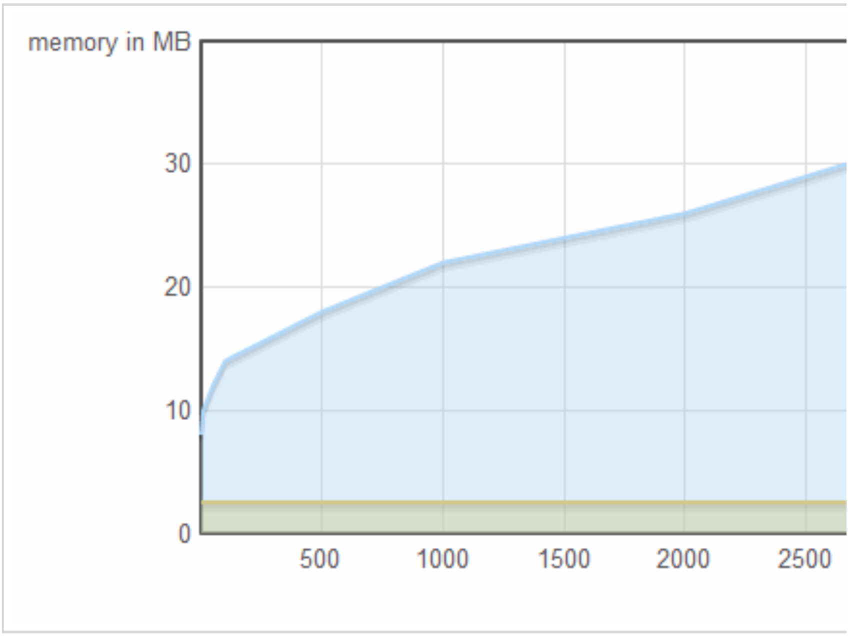
Sunday, May 15, 2011

A Not Very Short Introduction To Node.js

Node.js is a set of asynchronous libraries, built on top of the Google V8 Javascript Engine. Node is used for server side development in Javascript. Do you feel the rush of the 90's coming through your head. It is not the revival of LiveWire, Node is a different beast. Node is a single threaded process, focused on doing networking right. Right, in this case, means without blocking I/O. All the libraries built for Node use non-blocking I/O. This is a really cool feature, which allows the single thread in Node to serve thousands of request per second. It even lets you run multiple servers in the same thread. Check out the performance characteristics of Nginx and Apache that utilize the same technique.



The graph for memory usage is even better.



Read more about it at the Web Faction Blog

OK, so what's the catch? The catch is that all code that does I/O, or anything slow at all, has to be called in an asynchronous style.

```
// Synchronous
var result = db.query("select * from T");
// Use result
```

Top Posts - PostRank

- 10.0 ASP.NET MVC vs. Rails3
- 10.0 A Not Very Short Introduction To Node.js
- 9.5 Mastery
- 8.9 Ruby, an Exceptional Language
- 8.6 Why Ruby?

Powered by Postrank

Blog Archive

- ▼ 2011 (9)
  - ► June (1)
  - ▼ May (3)
    - Ruby, an Exceptional Language
    - A Not Very Short Introduction To Node.js
    - Mastery
  - ► April (1)
  - ► March (2)
  - ► February (1)

```
// Asynchronous
db.query("select * from T", function (resu
    // Use result
});
```

So, all libraries that deal with IO has to be re-implemented with this style of programming. The good news is that even though Node has only been around for a couple of years, there are more than 1800 libraries available. The libraries are of varying quality but the popularity of Node shows good promise to deliver high-quality libraries for anything that you can imagine.

HISTORY

Node is definitely not the first of its kind. The non-blocking `select()` loop, that is at the heart of Node, dates back to 1983.

Twisted appeared in Python (2002) and EventMachine in Ruby (2003).

This year a couple of newcomers appeared.

Goliath, which builds on EventMachine, and uses fibers to allow us to program in an synchronous style even though it is asynchronous under the hood.

And, the Async Framework in .Net, which enhances the compiler with the keywords `async` and `await` to allow for very elegant asynchronous programming.

GET STARTED

This example uses OSX as an example platform, if you use something else you will have to google for instructions.

- ▶ January (1)
- ▶ 2010 (18)
- ▶ 2009 (37)
- ▶ 2008 (20)
- ▶ 2007 (7)
- ▶ 2006 (14)

Software Books

Agile Software Development, Principles, Patterns, and Practices

AspectJ in Action: Practical Aspect-Oriented Programming

Clean Code

Concepts, Techniques, and Models of Computer Programming

Domain-Driven Design: Tackling Complexity in the Heart of Software

Effective Java Programming Language Guide

Extreme

```
# Install Node using Homebrew
$ brew install node
==> Downloading http://nodejs.org/dist/nod
##########################################
==> ./configure --prefix=/usr/local/Cellar
==> make install
==> Caveats
Please add /usr/local/lib/node to your NOD
==> Summary
/usr/local/Cellar/node/0.4.7: 72 files, 7.
```

When installed you have access to the `node` command-line command. When invoked without arguments, it start a REPL.

```
$ node
> function hello(name) {
... return 'hello ' + name;
... }
> hello('tapir')
'hello tapir'
>
```

When invoked with a script it runs the script.

```
// hello.js
setTimeout(function() {
    console.log('Tapir');
}, 2000);
console.log('Hello');
```

```
$ node hello.js
Hello
...
Tapir
```

Programming Explained: Embrace Change
High Performance Javascript
Javascript Patterns
Javascript, the Good Parts
jQuery Cookbook
jQuery in Action
Metaprogramming Ruby
Object Design: Roles, Responsibilities, and Collaborations
Object Models: Strategies, Patterns, and Applications
Object-Oriented Software Construction
On LISP: Advanced Techniques for Common LISP
Programming in Haskell
Purely Functional Data Structures
Real World Haskell
Refactoring: Improving the Design of Existing Code
Ruby Best Practices

NETWORKING

As I mentioned above, Node is focused on networking. That means it should be easy to write networking code. Here is a simple echo server.

```
// Echo Server
var net = require('net');

var server = net.createServer(function(soc
    socket.on('data', function(data) {
        console.log(data.toString());
        socket.write(data);
    });
});
server.listen(4000);
```

And here is a simple HTTP server.

```
// HTTP Server
var http = require('http');
var web = http.createServer(function(reque
  response.writeHead(200, {
    'Content-Type': 'text/plain'
  });
  response.end('Tapirs are beautiful!\n');
});
web.listen(4001);
```

Quite similar. A cool thing is that the servers can be started from the same file and node will, happily, serve both HTTP and echo requests from the same thread without any problems. Let's try them out!

```
# curl the http service
$ curl localhost:4001
Tapirs are beautiful!
```

Smalltalk Best Practice Patterns

Smalltalk,Objects and Design

Structure and Interpretation of Computer Programs

Test-Driven Development: A Practical Guide

The Little Schemer

The Mythical Man Month

The Pragmatic Programmer: From Journeyman to Master

The Rails 3 Way

The Reasoned Schemer

The Ruby Way

The Seasoned Schemer

The Well Grounded Rubyist

Working Effectively with Legacy Code

Wetware Books

A Whole New Mind

An Introduction to

```
# use netcat to send the string to the ech
$ echo 'Hello beautiful tapir' | nc localh
Hello beautiful tapir
```

MODULES

Node comes with a selection of built in modules. Ryan Dahl says that they try to keep the core small, but even so the built-in modules cover a lot of useful functionality.

- net - contains tcp/ip related networking functionality.
- http - contains functionality for dealing with the HTTP protocol.
- util - holds common utility functions, such as log, inherits, pump, ...
- fs - contains filesystem related functionality, remember that everything should be asynchronous.
- events - contains the EventEmitter that is used for dealing with events in a consistent way. It is used internally but it can be used externally too.

AN EXAMPLE

Here is an example of a simple module.

```
// module tapir.js

// require another module
var util = require('util');

function eat(food) {
  util.log('eating '+ food);
}

// export a function
exports.eat = eat;
```

Zen Buddhism
Brain Rules
Flow: The Psychology of Optimal Experience
Gödel, Escher, Bach: An Eternal Golden Braid
Heidegger and a Hippo Walk Through Those Pearly Gates
Infinity and the Mind
Lila: An Inquiry Into Morals
Man's Search For Meaning
Plato and a Platypus Walk into a Bar: Understanding Philosophy Through Jokes
Strangers to Ourselves
The User Illusion
Zen and the Art of Motorcycle Maintenance

Other Books

A Short History of

As you can see it looks like a normal Javascript file and it even looks like it has global variables. It doesn't. When a module is loaded it is wrapped in code, similar to this.

```
var module = { exports: {}};
(function(module, exports){
  // module code from file
  ...
})(module, module.exports);
```

As you can see the code is wrapped in a function and an empty object with an `export` property is sent into it. This is used by the file to export only the functions that it want to publish.

The `require` function works in symphony with the module and it returns the exported functions to the caller.

NODE PACKAGE MANAGER, NPM

To allow simple handling of third-party packages, Node uses `npm`. It can be installed like this:

```
$ curl http://npmjs.org/install.sh | sh
...
```

And used like this:

```
$ npm install -g express
mime@1.2.1 /usr/local/lib/node_modules/exp
connect@1.4.0 /usr/local/lib/node_modules/
qs@0.1.0 /usr/local/lib/node_modules/expre
/usr/local/bin/express -> /usr/local/lib/n
```

Nearly Everything
Are Your Lights On?
Presentation Zen
Your Competent Child

Links

My links on del.icio.us
The Tapir Gallery

```
express@2.3.2 /usr/local/lib/node_modules/
```

As you can see, installing a module also installs its dependencies. This works because a module can be package with meta-data, like so:

```
// express/package.json
{
  "name": "express",
  "description": "Sinatra inspired web dev
  "version": "2.3.2",
  "author": "TJ Holowaychuk <tj@vision-med
  "contributors": [
    { "name": "TJ Holowaychuk", "email": "
    { "name": "Guillermo Rauch", "email":
  ],
  "dependencies": {
    "connect": ">= 1.4.0 < 2.0.0",
    "mime": ">= 0.0.1",
    "qs": ">= 0.0.6"
  },
  "keywords": ["framework", "sinatra", "we
  "repository": "git://github.com/visionme
  "main": "index",
  "bin": { "express": "./bin/express" },
  "engines": { "node": ">= 0.4.1 < 0.5.0"
}
```

The `package.json` contains information about who made the module, its dependencies, along with some additional information to enable better searching facilities.

Npm installs the modules from a common respository, which contains more than 1800 modules.

NOTEWORTHY MODULES

Express is probably the most used of all third-party modules. It is a Sinatra clone and it is very good, just like Sinatra.

```
// Create a server
var app = express.createServer();
app.listen(4000);

// Mount the root (/) and redirect to inde
app.get('/', function(req, res) {
  res.redirect('/index.html');
});

// Handle a post to /quiz
app.post('/quiz', function(req, res) {
  res.send(quiz.create().id.toString());
});
```

Express uses *Connect* to handle middleware. Middleware is like Rack but for Node (No wonder that Node is nice to work with when it borrows its ideas from Ruby :)

```
connect(
      // Add a logger
      connect.logger()
      // Serve static file from the curren
    , connect.static(__dirname)
      // Compile Sass and Coffescript file
    , connect.compiler({enable: ['sass', '
      // Profile all requests
    , connect.profiler()
  ).listen(3000);
```

Another popular library is *Socket.IO*. It handles the usual socket variants, such as WebSocket, Comet, Flash Sockets, etc...

```
var http = require('http');
var io = require('socket.io');

server = http.createServer(function(req, r
server.listen(80);

// socket.io attaches to an existing serve
var socket = io.listen(server);
socket.on('connection', function(client){
  // new client is here!
  client.on('message', function(){ ... })
  client.on('disconnect', function(){ ...
});
```

MySql has a library for Node.

```
client.query(
  'SELECT * FROM ' + TEST_TABLE,
  // Note the callback style
  function(err, results, fields) {
    if (err) { throw err; }

    console.log(results);
    console.log(fields);
    client.end();
  }
);
```

And *Mongoose* can be used for accessing MongoDB.

```
// Declare the schema
var Schema = mongoose.Schema
  , ObjectId = Schema.ObjectId;

var BlogPost = new Schema({
    author     : ObjectId
  , title      : String
  , body       : String
  , date       : Date
});

// Use it
```

```
var BlogPost = mongoose.model('BlogPost');

// Save
var post = new BlogPost();
post.author = 'Stravinsky';
instance.save(function (err) {
  //
});

// Find
BlogPost.find({}, function (err, docs) {
  // docs.forEach
});
```

TEMPLATING ENGINES

Everytime a new platform makes its presence, it brings along a couple of new templating languages and Node is no different. Along with the popular ones from the Ruby world, like Haml and Erb (EJS in Node), comes some new ones like Jade and some browser templating languages like Mustache and jQuery templates. I'll show examples of Jade and Mu (Mustache for Node).

I like *Jade*, because it is a Javascript dialect of Haml and it seems appropriate to use if I'm using Javascript on the server side.

```
!!! 5
html(lang="en")
  head
    title= pageTitle
    script(type='text/javascript')
      if (foo) {
        bar()
      }
  body
    h1 Jade - node template engine
    #container
      - if (youAreUsingJade)
```

```
        p You are amazing
      - else
        p Get on it!
```

I'm not really sure if I like Mustache or not, but I can surely see the value of having a templating language which works both on the server side and in the browser.

```
<h1>{{header}}</h1>
{{#bug}}
{{/bug}}

{{#items}}
  {{#first}}
    <li><strong>{{name}}</strong></li>
  {{/first}}
  {{#link}}
    <li><a href="{{url}}">{{name}}</a></li
  {{/link}}
{{/items}}

{{#empty}}
  <p>The list is empty.</p>
{{/empty}}
```

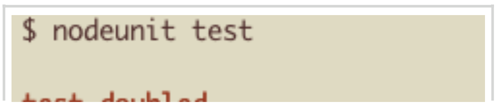
TESTING

Node comes with assertions built in, and all testing frameworks build on the Assert module, so it is good to know.

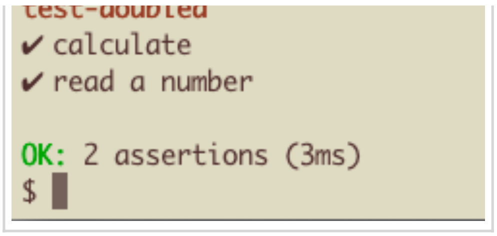
```
assert.ok(value, [message]);
assert.equal(actual, expected, [message])
assert.notEqual(actual, expected, [message
assert.deepEqual(actual, expected, [messag
assert.strictEqual(actual, expected, [mess
assert.throws(block, [error], [message])
assert.doesNotThrow(block, [error], [messa
assert.ifError(value)
assert.fail(actual, expected, message, ope
// Example
// assert.throws(function, regexp)
assert.throws(
  function() { throw new Error("Wrong valu
  /value/
);
```

Apart from that there are at least 30 different testing frameworks to use. I have chosen to use NodeUnit since I find that it handles asynchronous testing well, and it has a nice UTF-8 output that looks good in the terminal,

```
// ./test/test-doubled.js
var doubled = require('../lib/doubled');

// Exported functions are run by the test
exports['calculate'] = function (test) {
    test.equal(doubled.calculate(2), 4);
    test.done();
};

// An asynchronous test
exports['read a number'] = function (test)
    test.expect(1); // Make sure the asser

    var ev = new events.EventEmitter();
    process.openStdin = function () { retu
    process.exit = test.done;

    console.log = function (str) {
        test.equal(str, 'Doubled: 24');
    };

    doubled.read();
    ev.emit('data', '12');
};
```



```
$ nodeunit test
```



```
test-doubled
✔ calculate
✔ read a number

OK: 2 assertions (3ms)
$ █
```

DEPLOYMENT

There are already a lot of platforms providing Node as a service (PaaS , Platform as a Service). Most of them are using Heroku style deployment by pushing to a Git remote. I'll show three alternatives that all provide free Node hosting.

JOYENT (NO.DE)

Joyent, the employers of Ryan Dahl, give you `ssh` access so that you can install the modules you need. Deployment is done by pushing to a Git remote.

```
$ ssh node@my-machine.no.de
$ nmp install express
$ git remote add node node@andersjanmyr.no
$ git push node master
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 321 bytes, do
Total 3 (delta 2), reused 0 (delta 0)
remote: Starting node v0.4.7...
remote: Successful
To node@andersjanmyr.no.de:repo
  8f59169..c1177b0  master -> master
```

NODESTER

Nodester, gives you a command line tool, `nodester`, that you use to install modules. Deployment by pushing to a Git remote.

```
$ nodester npm install express
$ git push nodester master
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 341 bytes, do
Total 3 (delta 2), reused 0 (delta 0)
remote: Syncing repo with chroot
remote: From /node/hosted_apps/andersjanmy
remote:    38f4e6e..8f59169  master     ->
remote: Updating 38f4e6e..8f59169
remote: Fast-forward
remote:  Gemfile.lock |   10 ++++------
remote:  1 files changed, 4 insertions(+),
remote: Checking ./.git/hooks/post-receive
remote: Attempting to restart your app: 13
remote: App restarted..
remote:
remote:
remote:      \m/ Nodester out \m/
remote:
remote:
To ec2-user@nodester.com:/node/hosted_apps
   38f4e6e..8f59169  master -> master
```

CLOUD FOUNDRY

Cloud Foundry is one of the most interesting platforms in the cloud. It was genius by VM Ware to open source the platform, allowing anyone to set up their own cloud if they wish. If you don't want to setup your own Cloud Foundry Cloud, you can use the service hosted at cloundfoundry.com.

With Cloud Foundry, you install the modules locally and then they are automatically deployed as part of the `vmc push`. Push in this case does not mean `git push`, but instead, copy all the files from my local machine to the server.

```
$ npm install express  # Install locally
mime@1.2.1 ./node_modules/express/node_mod
connect@1.4.0 ./node_modules/express/node_
qs@0.1.0 ./node_modules/express/node_modul
express@2.3.0 ./node_modules/express

$ vmc push
Would you like to deploy from the current
Application Name: snake
Application Deployed URL: 'snake.cloudfoun
Detected a Node.js Application, is this co
Memory Reservation [Default:64M] (64M, 128
Creating Application: OK
Would you like to bind any services to 'sn
Uploading Application:
  Checking for available resources: OK
  Packing application: OK
  Uploading (1K): OK
Push Status: OK
Staging Application: OK
Starting Application: ........OK
```

TOOLS

There are of course a bunch of tools that come with a new platform, Jake, is a Javascript version of Rake, but I am happy with Rake and I don't see the need to switch. But, there are some tools that I cannot live without when using Node.

RELOADERS

If you use the vanilla `node` command then you have to restart it every time you make a change to a file. That is awfully annoying and there are already a number of solutions to the problem.

```
# Nodemon watches the files in your direct
$ npm install nodemon
nodemon@0.3.2 ../node_modules/nodemon
```

```
$ nodemon server.js
30 Apr 08:21:23 - [nodemon] running server
...
# Saving the file
30 Apr 08:22:01 - [nodemon] restarting due

# Alternative
$ npm install supervisor
$ supervisor server.js
DEBUG: Watching directory '/evented-progra
```

DEBUGGERS

Another tool that it is hard to live without is a debugger. Node comes with one built in. It has a `gdb` flavor to it and it is kind of rough.

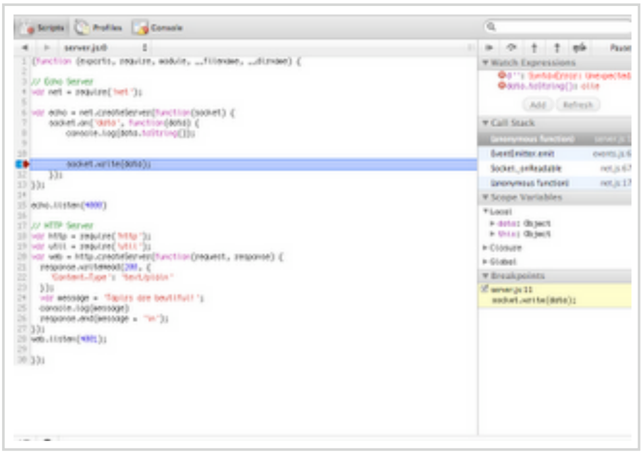
```
$ node debug server.js
debug> run
debugger listening on port 5858
connecting...ok
break in #<Socket> ./server.js:9
    debugger;

debug> p data.toString();
tapir
```

```
// Javascript
var echo = net.createServer(function(socke
  socket.on('data', function(data) {
      debugger; // <= break into debugger
      socket.write(data);
  });
});
```

If you want a GUI debugger, it is possible to use the one that comes with Chrome by installing the `node-inspector`. It is started similarly to the built in debugger, but the `--debug` is an option instead of a subcommand.

```
$ node-inspector &
visit http://0.0.0.0:8080/debug?port=5858

$ node --debug server.js debugger listenin
```

After that you can just fire up Chrome on the URL, http://0.0.0.0:8080/debug?port=5858 and you can debug the node process just as if it was running in the browser.



```
(function (exports, require, module, __filename, __dirname) {

// Echo Server
var net = require('net');

var echo = net.createServer(function(socket) {
    socket.on('data', function(data) {
        console.log(data.toString());


        socket.write(data);
    });
});

echo.listen(4000)

// HTTP Server
var http = require('http');
var util = require('util');
var web = http.createServer(function(request, response) {
  response.writeHead(200, {
    'Content-Type': 'text/plain'
  });
  var message = 'Tapirs are beutiful!';
  console.log(message)
  response.end(message + '\n');
});
web.listen(4001);

});
```

IDIOMS

Idioms, patterns, techniques, call it what you like. Javascript code is littered with callbacks, and event more so with Node. Here are some tips on how to write good asynchronous code with Node.

RETURN ON CALLBACKS

It is easy to forget to escape from the function after a callback has been called. An easy way to remedy this problem is to call return before every call to a callback. Even though the value is never used by the caller, it is an easy pattern to recognize and it prevents bugs.

```
function doSomething(response, callback) {
  doAsyncCall('tapir', function(err, resul
    if (err) {
      // return on the callback
      return callback(err);
    }
    // return on the callback
    return callback(null, result);
  });
}
```

EXCEPTIONS IN CALLBACKS

Exceptions that occur in callbacks cannot be handled the way we are used to, since the context is different. The solution to this is to pass along the exception as a parameter to the callback. In Node the convetion is to pass the error as the first parameter into the callback.

```
function insertIntoTable(row, function(err
  if (err) return callback(err);
  ...

  // Everything is OK
  return callback(null, 'row inserted');
}
```

PARALLEL EXECUTION

If you have multiple tasks that need to be finished before you take some new action, this can be handled with a simple counter. Here is an example of a simple function that starts up a bunch of functions in parallel and waits for all of them to finish before calling the callback.

```
// Do all in parallel
function doAll(collection, callback) {
  var left = collection.length;
  collection.forEach(function(fun) {
    fun(function() {
      if (--left == 0) callback();
    });
  });
};

// Use it
var result = [];
doAll([
  function(callback) {
    setTimeout(function() {result.push(1);
  function(callback) {
    setTimeout(function() {result.push(2);
  function(callback) {
    setTimeout(function() {result.push(3);
  ], function() { return result; }

// returns [3, 1, 2]
```

SEQUENTIAL EXECUTION

Sometimes the ordering is important. Here is a simple function that makes sure that the calls are executed in sequence. It uses recursion to to make sure that the calls are handled in the correct order. It also uses the Node function `process.nextTick()` to prevent the stack from getting to large for large collections. Similar results can be obtained with `setTimeout()` in browser Javascript. It can be seen as a simple trick to achieve *tail recursion*.

```
function doInSequence(collection, callback
    var queue = collection.slice(0); // Du

    function iterate() {
      if (queue.length === 0) return callb
      // Take the first element
      var fun = queue.splice(0, 1)[0];
      fun(function(err) {
        if (err) throw err;
        // Call it without building up the
        process.nextTick(iterate);
      });
    }
    iterate();
}


var result = [];
doInSequence([
  function(callback) {
    setTimeout(function() {result.push(1);
  function(callback) {
    setTimeout(function() {result.push(2);
  function(callback) {
    setTimeout(function() {result.push(3);
  ], function() { return result; });

// Returns [1, 2, 3]
```

LIBRARY SUPPORT FOR ASYNCHRONOUS PROGRAMMING

If you don't want to write these functions yourself, there are a few libraries that can help you out. I'll show two version that I like.

FIBERS

Fibers are also called co-routines. Fibers provide two functions, *suspend* and *resume*, which allows us to write code in a synchronous looking style. In the Node version of fibers, node-fibers, suspend and resume are called `yield()` and `run()` instead.

```
require('fibers');
var print = require('util').print;

function sleep(ms) {
    var fiber = Fiber.current;
    setTimeout(function() { fiber.run(); }
    yield();
}

Fiber(function() {
    print('wait... ' + new Date + '\n');
    sleep(1000);
    print('ok... ' + new Date + '\n');
}).run();
print('back in main\n');
```

Fibers are a very nice way of writing asynchronous code but, in Node, they have one drawback. They are not supported without patching the V8 virtual machine. The patching is done when you install `node-fibers` and you have to run the command `node-fibers` instead of `node` to use it.

THE async LIBRARY

If you don't want to use the patched version of V8, I can recommend the async library. Async provides around 20 functions that include the usual 'functional' suspects (map, reduce, filter, forEach...) as well as some common patterns for asynchronous flow control (parallel, series, waterfall...). All these functions assume you follow the Node convention of providing a single callback as the last argument of your async function.

```
async.map(['file1','file2','file3'], fs.st
    // results is now an array of stats fo
});

async.filter(['file1','file2','file3'], pa
    // results now equals an array of the
});

async.parallel([
    function(){ ... },
    function(){ ... }
], callback);
```

```
async.series([
    function(){ ... },
    function(){ ... }
], callback);
```

CONCLUSION

Node is definitely an interesting platform. The possibility to have Javascript running through the whole stack, from the browser all the way down into the database (if you use something like CouchDB or MongoDB) really appeals to me. The easy way to deploy code to multiple, different cloud providers is also a good argument for Node.

POSTED BY ANDERS JANMYR AT 09:49
LABELS: JAVASCRIPT, NODE, NODEJS, TUTORIAL

15 COMMENTS:

Anonymous said...

Should the second 'req' in:

```
app.get('/', function(req, req) {
res.redirect('/index.html');
});
```

be 'res'?:

```
app.get('/', function(req, res) {
res.redirect('/index.html');
});
```

Great article though!

16 MAY, 2011 06:57

<a href="http://twitter.com/#!/andersjanmyr">@andersjanmyr</a> said...

Yes, you're right, fixed it, thanks!

16 MAY, 2011 08:33

Soludra Ar'thela said...

This post has been removed by the author.
16 MAY, 2011 10:30

Jonathan Castello said...

In the "EXCEPTIONS IN CALLBACKS" section, you have "callback(err, data) { ... }". I think you meant "function", not "callback".

Great article, I learned a few things about Node I didn't already :)

16 MAY, 2011 10:32

Anders Janmyr said...

@Jonathan, thanks! I fixed it.

16 MAY, 2011 10:45

Adrian Quark said...

In the example on sequential execution, I think you meant process.nextTick(iterate) instead of process.nextTick(iterate()).

17 MAY, 2011 03:19

Anders Janmyr said...

@Adrian, you're right. Thanks!

17 MAY, 2011 06:44

jamuraa said...

Thanks for this post, it was very useful for me. I don't think I 'got' node.js before this, because all the tutorials that I had seen before now stopped shortly after introducing npm and didn't go as far as the whole stack.

17 MAY, 2011 13:46

Anders Janmyr said...

@jamuraa, I'm glad it helped you!

17 MAY, 2011 14:19

obowah said...

the mongoose example isn't async - should put the find() call in the save() function callback. i think

20 MAY, 2011 21:19

Anders Janmyr said...

@obowah In this case it is meant as separate examples and not as a sequence of instructions. If it was you are correct!

21 MAY, 2011 12:12

VA said...

Best thing so far for me... Truly thanks

29 MAY, 2011 13:39

Anders Janmyr said...

@VA, I'm glad you liked it!

29 MAY, 2011 15:42

Andrew said...

Hey Anders,

This was a really good read. I particularly like the examples at the end of how to write JS that is to execute in series or parallel.

Node's biggest barrier to entry for non-functional programmers is the shift to working with callbacks and context.

More clear, practical examples like this will surely demystify it for a lot of people.

Keep up the good work!

27 JULY, 2011 16:55

Anders Janmyr said...

@Andrew, thanks for the feedback, I'm glad you liked it.

28 JULY, 2011 08:46

Post a Comment

LINKS TO THIS POST

Create a Link

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)