NPM: node package manager. Useful for handles package dependencies.

Package.json: manage dependencies and its version

Js has no classes.

Where in Java our myCar, asked to honk, says “go look at this class over there, which is my blueprint, to find the code you need”, JavaScript says “go look at that other object over there, which is my prototype, it has the code you are looking for”. Building objects via an object-object relationship is called Prototype-based programming, versus Class-based programming used in more traditional languages like Java.

In java : Car myCar = new Car();

In js:

var makeCar = function() {

var newCar = {}

newCar.honk = function() {

console.log('honk honk');

};

return newCar;

};

myCar1 = makeCar();

myCar2 = makeCar();

myCar3 = makeCar();

(1) 1000 mycar() will create 1000 objects allocating memory for 1000 functions.

1. they don’t share anything
2. Property:Whenever we call a function on an object, the JavaScript interpreter tries to find that function within the queried object. But if it doesn’t find the function within the object itself, it asks the object for the pointer to its prototype, then goes to the prototype, and asks for the function there. If it is found, it is then executed.

Var car = function() {};

Car.prototype.honk = function() {

Console.log(“honk honk”);

};

Var myCar1 = new Car();

Var myCar2 = new Car();

myCar1.honk(); // executes Car.prototype.honk() and outputs “honk honk”

myCar2.honk(); // executes Car.prototype.honk() and outputs “honk honk”

1. we can change Car.prototype in run time

var Car = function() {};

Car.prototype.honk = function() {

console.log('honk honk');

};

var myCar1 = new Car();

var myCar2 = new Car();

myCar1.honk();

myCar2.honk();

// executes Car.prototype.honk() and outputs "honk honk"

// executes Car.prototype.honk() and outputs "honk honk"

Car.prototype.honk = function() {

console.log('meep meep');

};

myCar1.honk();

myCar2.honk();

1. we can also treat only one of our cars differently at runtime:

var Car = function() {};

Car.prototype.honk = function() {

console.log('honk honk');

};

var myCar1 = new Car();

var myCar2 = new Car();

myCar1.honk();

myCar2.honk();

// executes Car.prototype.honk() and outputs "honk honk"

// executes Car.prototype.honk() and outputs "honk honk"

myCar2.honk = function() {

console.log('meep meep');

}; // add additional behavior for myCar2

myCar1.honk();

myCar2.honk();

// executes Car.prototype.honk() and outputs "honk honk"

// executes myCar2.honk() and outputs "meep meep"

1. different in java like language and js

That’s one of the major differences to class-based programming: while objects are relatively “rigid” e.g. in Java, where the structure of an object cannot be changed at runtime, in JavaScript, the prototype-based approach links objects of a certain class more loosely together, which allows to change the structure of objects at any time.

Comparsion:

var C = function() {

this.f = function(foo) {

console.log(foo);

};

};

var a = [];

for (var i = 0; i < 1000000; i++) {

a.push(new C());

}

Creates 1000000 objects that all have the function directly attahed to them.

var C = function() {};

C.prototype.f = function(foo) {

console.log(foo);

};

var a = [];

for (var i = 0; i < 1000000; i++) {

a.push(new C());

}

1. Inheritance(inheritance runs through a chain of prototypes)

var Vehicle = function() {};

Vehicle.prototype.drive = function() {

console.log('vrooom...');

};

var Car = function() {};

Car.prototype = new Vehicle();

Car.prototype.honk = function() {

console.log('honk honk');

};

var myCar = new Car();

myCar.honk();

myCar.drive();

// outputs "honk honk"

// outputs "vrooom..."

(8) There is only one single piece of JavaScript code that can be executed within

the event loop at any given time.

Read file system is expensive compare with others. So all IO operations in node is asynchronously.

1. Two principles for writing responsive Nodejs applications:

Handle IO-intensive operations through asynchronous operations

Keep your own code (that is, everything that happens synchronously within event loop iterations) as lean as possible

1. optimizing code performance and control flow management using the async library

Async.parallel to collect results after both of request finished.

Code exampel:

Client.js:

"use strict";

var request = require("request");

var async = require("async");

var name, status;

var getUserName = function(callback) {

request.get(

"http://localhost:8080/getUserName?id=1234",

function(err, res, body) {

var result = JSON.parse(body);

callback(err, result.value);

}

);

};

var getUserStatus = function(callback) {

request.get(

"http://localhost:8080/getUserStatus?id=1234",

function(err, res, body) {

var result = JSON.parse(body);

callback(err, result.value);

}

);

};

async.parallel([getUserName, getUserStatus], function(err, results) {

console.log(results);

});

Server.js

"use strict";

var http = require("http");

var url = require("url");

var queryString = require("querystring");

http.createServer(function(request, response) {

var pathname = url.parse(request.url).pathname;

var query = url.parse(request.url).query;

var id = queryString.parse(query)["id"];

var result = {

"pathname": pathname,

"id": id,

"value": Math.floor(Math.random() \* 100)

};

setTimeout(function() {

response.writeHead(200, {"Content-Type": "application/json"});

response.end(JSON.stringify(result));

}, 2000 + Math.floor(Math.random() \* 1000));

}).listen(8080, function(){

console.log("Echo Server listening on poart 8080");

});

Async.series

"use strict";

var request = require("request");

var async = require("async");

var url = "http://localhost:8080/";

async.series([

function(callback) {

request.get(url + "getUserName?id=1234", function(err, res, body) {

console.log("Name:", JSON.parse(body).value);

callback(null);

});

},

function(callback) {

request.get(url + "getUserCountry?id=1234", function(err, res, body) {

console.log("Country:", JSON.parse(body).value);

callback(null);

});

},

function(callback) {

request.get(url + "getUserAge?id=1234", function(err, res, body) {

console.log("Age:", JSON.parse(body).value);

callback(null);

});

},

function(callback) {

request.get(url + "getUserSex?id=1234", function(err, res, body) {

console.log("Sex:", JSON.parse(body).value);

callback(null);

});

},

])

(10)Node-mysql:

"use strict";

var mysql = require("mysql");

var connection = mysql.createConnection({

host: "localhost",

user: "root",

password: "root"

});

connection.query(

"SELECT \* FROM node.test;",

function(err, results, fields) {

if(err){

console.log(err);

}else {

console.log(results)

}

connection.end();

}

);

"use strict";

var mysql = require("mysql");

var connection = mysql.createConnection(

{

host: "localhost",

user: "root",

password: "root"

}

);

connection.query("create database node", function (err) {

if(err) {

console.log("could not create node database!");

}

});

connection.query("use node", function (err) {

if(err) {

console.log("could not switch to database!!!");

}

});

connection.query("create table test (id INT(11) AUTO\_INCREMENT, content VARCHAR(255), PRIMARY KEY(id))",

function(err) {

if(err) {

console.log("create table test failer!!!");

}

});

connection.query("INSERT INTO test (content) VALUES ('hello')");

connection.query("INSERT INTO test (content) VALUES ('world')");

connection.end();

Two ways of reading mysql:

(1) connection.query(

'SELECT id, content FROM test',

function (err, results, fields) {

for (var i = 0; i < results.length; i++) {

console.log('Content of id ' + results[i].id +

' is ' + results[i].content);

}

}

);

Disadvantage:

The nodejs application will not block while the database retrieves 1000000 rows, However, once we receive the results,and the callback function will take a long time deal with the rows.Besides, during this time, our application will block. Maybe our appliction can not do anything during about 10 seconds.

The solutions:

instead of using the all-or-nothing callback pattern, we can utilize node-mysql’s streaming API.

(2) var query = connection.query('SELECT id, content FROM test');

query.on('row', function(row) {

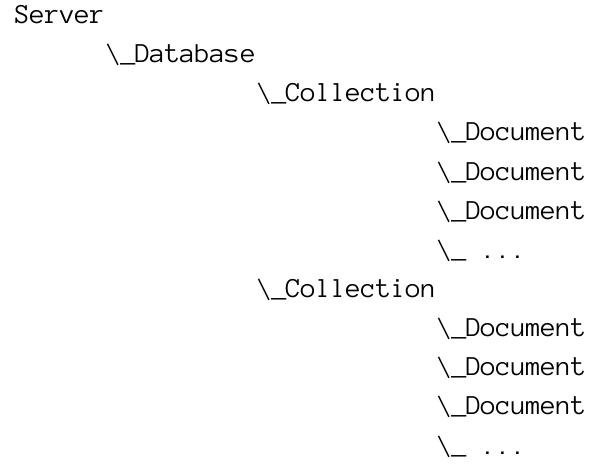
console.log('Content of id ' + row.id + ' is ' + row.content);

});

While this doesn’t make our application faster in the narrower sense (it will still take the assumed 10 seconds until all rows have been handled), it does make it way more efficient. Our application now only blocks during the handling of one row, immediately returning control to the event loop of our application, allowing to execute other code in between.

1. sql queries secure against attacks
2. Nodejs and mongoDB

MongoDB is a document-oriented NoSQL database that stores objects in the following hierarchy.



First , install mongodb(

<https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/#overview)>

1. an example for create connection and insert some documents in mongodb

**"use strict";**

**var MongoClient = require("mongodb").MongoClient;**

**MongoClient.connect(**

**"mongodb://127.0.0.1:27017/accouting",**

**function(err, connection) {**

**if(!err) {**

**console.log("connected success!!!");**

**}**

**//console.log(err);**

**var collection = connection.collection("customers");**

**collection.insert({"name": "jane doe"}, function(err, count) {**

**if(!err) {**

**console.log("insert success!!!");**

**}**

**collection.find().toArray(function(err, documents) {**

**console.dir(documents);**

**connection.close();**

**});**

**});**

**}**

**)**

2) then we write some codes to update the documents in mongodb

"use strict";

**var MongoClient = require("mongodb").MongoClient;**

**MongoClient.connect(**

**"mongodb://127.0.0.1:27017/accouting",**

**function(err, connection) {**

**if(!err) {**

**console.log("connected success!!!");**

**}**

**var collection = connection.collection("customers");**

**collection.update({},{"$set": {"age": 24 }},{"multi": true}, function(err, count) {**

**if(!err) {**

**console.log("update success!!!", count, "documents");**

**}**

**collection.find().toArray(function(err, documents) {**

**console.dir(documents);**

**connection.close();**

**});**

**});**

**}**

**)**

1. **create index and test index**

**How to:**

**"use strict";**

**var MongoClient = require("mongodb").MongoClient;**

**MongoClient.connect(**

**"mongodb://127.0.0.1:27017/accouting",**

**function(err, connection) {**

**if(!err) {**

**console.log("connected success!!!");**

**}**

**var collection = connection.collection("customers");**

**collection.ensureIndex("v", function (err, indexName) {**

**connection.close();**

**});**

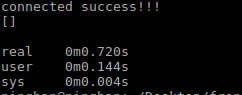
**}**

**);**

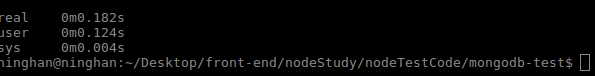
**Speed up:**

**200000 nums**

Without index:



With index:



Improve effiencies:

Bad example:

collection.find().toArray(function (err, documents) {

console.dir(documents);

});

Good example:

collection.find().each(function (err, document) {

console.dir(document);

});

Making use with stream api:

var doFind = function (callback) {

var stream = collection.find(

{},

{'sort': '\_id'}

).stream();

stream.on('data', function(document) {

console.dir(document);

});

stream.on('close', function() {

callback();

});

};

