

Node.js基础

NodeJS是什么

node.js是一个异步的事件驱动的JavaScript运行时

<https://nodejs.org/en/>

类比学习运行时这个概念

- JRE java 运行时环境
- C Runtime
- .NET Common Language Runtime

运行时runtime就是**程序运行的时候**。运行时库就是**程序运行的时候**所需要依赖的库。

运行的时候指的是指令加载到内存并由CPU执行的时候。C代码编译成可执行文件的时候，指令没有被CPU执行，这个时候算是编译时，就是**编译的时候**。

node.js特性：

- [非阻塞I/O](#)
- [事件驱动](#)

下一代Node deno

<https://studygolang.com/articles/13101>

与前端的不同

- JS核心语法不变
- 前端 BOM DOM
- 后端 fs http buffer event os
- 运行node程序

```
// 01-runnode.js
console.log('hello,node.js!');
console.log('run me use: node 01-runnode!');
```

运行： `node 01-runnode.js`

每次修改js文件需重新执行才能生效，安装nodemon可以监视文件改动，自动重启：

```
npm i -g nodemon
```

- 调试node程序: Debug - Start Debugging

模块(module)

- 使用模块(module)
 - node内建模块

```
// 02-useModule.js
// 内建模块直接引入
const os = require('os')
const mem = os.freemem() / os.totalmem() * 100
console.log(`内存占用率${mem}%`)
```

- 第三方模块

```
// 同级CPU占用率, 先安装
npm i cpu-stat -s
```

```
// 导入并使用
const cpuStat = require("cpu-stat");
cpuStat.usagePercent((err, percent) => {
  console.log(`CPU占用: ${percent.toFixed(2)}%`);
});
```

// 介绍回调方法 const util = require('util') const getCpu = util.promisify(cpuStat.usagePercent)
getCpu().then(percent => { console.log(CPU占用: \${percent.toFixed(2)}%) })

```
//加入定时器定时显示统计
const showState = async () => {
  const mem = (os.freemem() / os.totalmem()) * 100
  const percent = await getCpu()
  console.log(`CPU占用:${percent.toFixed(2)}% 内存: ${mem} %`)
}
setInterval(showStatic,1000)
```

- 自定义模块: 代码分割、复用手段

```
// 编写一个独立配置文件conf.js
module.exports = {showState}
// 或者
module.exports.showState = showState

// 导入并使用, 02-useModule.js
// 使用相对路径, 可省略后缀, 若是文件夹则导入其内部index.js
const conf = require('./conf')
console.log(conf);
```

导出内容可以是导出对象的属性, currency.js

```
exports.rmbToDollar = function(){...}

// 导入并使用, 02-useModule.js
const {rmbToDollar, dollarToRmb} = require('./currency')
console.log(rmbToDollar(10));
console.log(dollarToRmb(10));
```

如果要给模块传递配置, 通常返回一个工厂函数

```
// currency.js
let rate; // 保存汇率

function rmbToDollar(rmb) {
  return rmb / rate;
}
function dollarToRmb(dollar) {
  return dollar * rate;
}

module.exports = function(r) {
  rate = r;
  return {
    rmbToDollar,
    dollarToRmb
  };
};

// useModule.js
const {rmbToDollar, dollarToRmb} = require('./currency')(6)
```

ES6导入语法: Node处于试验阶段 需要js 变为mjs node9.0以上 node --experimental-modules ./server.mjs

import http from 'http'

核心API

- fs - 文件系统

```
// 03-fs.js
const fs = require('fs');

// 同步调用
const data = fs.readFileSync('./conf.js'); //代码会阻塞在这里
console.log(data);
```

```

// 下面是同步调用，实际中建议使用异步版本
fs.readFile('./conf.js', (err, data) => {
  if (err) throw err;
  console.log(data);
});
console.log('其他操作');

// fs常常搭配path api使用
const path = require('path')
fs.readFile(path.resolve(path.resolve(__dirname, './app.js')), (err, data) => {
  if (err) throw err;
  console.log(data);
});

// promisify
const {promisify} = require('util')
const readFile = promisify(fs.readFile)
readFile('./conf.js').then(data=>console.log(data))

// fs Promises API node v10
const fsp = require("fs").promises;
fsp
  .readFile("./confs.js")
  .then(data => console.log(data))
  .catch(err => console.log(err));

// async/await
(async () => {
  const fs = require('fs')
  const { promisify } = require('util')
  const readFile = promisify(fs.readFile)
  const data = await readFile('./index.html')
  console.log('data', data)
})();

// 引用方式
Buffer.from(data).toString('utf-8')

```

读取数据类型为Buffer

- Buffer - 用于在 TCP 流、文件系统操作、以及其他上下文中与八位字节流进行交互。八位字节组成的数组，可以有效的在JS中存储二进制数据

```

// 04-buffer.js
// 创建一个长度为10字节以0填充的Buffer
const buf1 = Buffer.alloc(10);
console.log(buf1);

// 创建一个Buffer包含ascii.

```

```
// ascii 查询 http://ascii.911cha.com/
const buf2 = Buffer.from('a')
console.log(buf2,buf2.toString())

// 创建Buffer包含UTF-8字节
// UTF-8: 一种变长的编码方案, 使用 1~6 个字节来存储;
// UTF-32: 一种固定长度的编码方案, 不管字符编号大小, 始终使用 4 个字节来存储;
// UTF-16: 介于 UTF-8 和 UTF-32 之间, 使用 2 个或者 4 个字节来存储, 长度既固定又可变。
const buf3 = Buffer.from('Buffer创建方法');
console.log(buf3);

// 写入Buffer数据
buf1.write('hello');
console.log(buf1);

// 读取Buffer数据
console.log(buf3.toString());

// 合并Buffer
const buf4 = Buffer.concat([buf1, buf3]);
console.log(buf4.toString());

// 可以尝试修改fs案例输出文件原始内容
```

Buffer类似数组, 所以很多数组方法它都有 GBK 转码 iconv-lite

- http: 用于创建web服务的模块

创建一个http服务器, 05-http.js

```
const http = require('http');
const server = http.createServer((request, response) => {
  console.log('there is a request');
  response.end('a response from server');
});
server.listen(3000);
```

```
// 打印原型链
function getPrototypeChain(obj) {
  var protoChain = [];
  while (obj = Object.getPrototypeOf(obj)) { //返回给定对象的原型。如果没有继承属性, 则返回 null 。
    protoChain.push(obj);
  }
  protoChain.push(null);
  return protoChain;
}
```

显示一个首页

```
const {url, method} = request;
if (url === '/' && method === 'GET') {
  fs.readFile('index.html', (err, data) => {
    if (err) {
      response.writeHead(500, { 'Content-Type': 'text/plain;charset=utf-8' });
      response.end('500, 服务器错误');
      return ;
    }
    response.statusCode = 200;
    response.setHeader('Content-Type', 'text/html');
    response.end(data);
  });
} else {
  response.statusCode = 404;
  response.setHeader('Content-Type', 'text/plain;charset=utf-8');
  response.end('404, 页面没有找到');
}
```

编写一个接口

```
else if (url === '/users' && method === 'GET') {
  response.writeHead(200, { 'Content-Type': 'application/json' });
  response.end(JSON.stringify([{name:'tom',age:20}, ...]));
}
```

- stream - 是用于与node中流数据交互的接口

```
//创建输入输出流,06-stream.js
const rs = fs.createReadStream('./conf.js')
const ws = fs.createWriteStream('./conf2.js')
rs.pipe(ws);

//二进制友好, 图片操作,06-stream.js
const rs2 = fs.createReadStream('./01.jpg')
const ws2 = fs.createWriteStream('./02.jpg')
rs2.pipe(ws2);

//响应图片请求, 05-http.js
const {url, method, headers} = request;

else if (method === 'GET' && headers.accept.indexOf('image/*') !== -1) {
  fs.createReadStream('.'+url).pipe(response);
}
```

Accept代表发送端（客户端）希望接受的数据类型。比如：Accept: text/xml; 代表客户端希望接受的数据类型是xml类型。

Content-Type代表发送端（客户端|服务器）发送的实体数据的数据类型。比如：Content-Type: text/html; 代表发送端发送的数据格式是html。

二者合起来，Accept:text/xml; Content-Type:text/html，即代表希望接受的数据类型是xml格式，本次请求发送的数据的数据格式是html。

仿写一个简版Express

- 体验express, 07-express.js

```
// npm i express
const express = require("express");
const app = express();
app.get("/", (req, res) => {
  res.end("Hello World");
});
app.get("/users", (req, res) => {
  res.end(JSON.stringify([{ name: "tom", age: 20 }]));
});
app.listen(3000, () => {
  console.log("Example app listen at 3000");
});
```

- 实现kexpress, 08-kexpress.js

```
const http = require('http')
const url = require('url')

const router = []
class Application {
  get(path, handler) {
    if (typeof path === 'string') {
      router.push({
        path,
        method: 'get',
        handler
      })
    }
  }
  listen() {
    const server = http.createServer((req, res) => {
      const { pathname } = url.parse(req.url, true)
      for (const route of router) {
        const { path, method, handler } = route
        if (pathname === path && req.method.toLowerCase() === method) {
          return handler(req, res)
        }
      }
    })
  }
}
```

```

    })
    server.listen(...arguments)
  }
}
module.exports = function createApplication() {
  return new Application()
}

```

事件(event)

```

const { EventEmitter } = require('events')
const event = new EventEmitter()
event.on('customer_event', () => {
  console.log('custom...' + new Date())
})
setInterval(() => {
  event.emit('customer_event')
}, 1000)

```

```

const { EventEmitter } = require('events')
// index.js
const chalk = require('chalk')
process.on('uncaughtException', function (err) {
  console.log(err);
});
app.get('/a', (req, res) => {
  abc() // 故意执行一个没有定义的方法
  res.end('aaa')
})

```

事件 EventLoop

<https://www.imoooc.com/article/40020> 一次搞懂Event loop

setTimeout/setImmediate/process.nextTick的区别

https://blog.csdn.net/hkh_1012/article/details/53453138

EventLoop是什么

一个循环 每次循环叫tick 每次循环的代码叫task

- V8引擎单线程无法同时干两件事
- 文件读取、网络IO缓慢且具有不确定性
- 要通过异步回调方式处理又称为异步IO
- 先同步再异步 异步放入队列等同步完成后在执行 每次循环叫一个tick (process.nextTick())



```
while (eventLoop.waitForTask()) {  
  eventLoop.processNextTask()  
}
```

异步任务的区分

唯一，整个事件循环当中，仅存在一个；执行为同步，同一个事件循环中的microtask会按队列顺序，串行执行完毕；

- microtasks(微任务):
 - process.nextTick
 - promise
- tasks(宏任务):
 - setTimeout
 - setInterval
 - setImmediate

```
// 等待一下事件队列  
(new Promise(resolve => {  
  console.log('A resolve')  
  resolve()  
})).  
.then(() => console.log('B promise then...'))  
  
setImmediate(() => {  
  console.log('C set Immediate ...')  
})  
// setTimeout, 放入Event Table中, 1秒后将回调函数放入宏任务的Event Queue中  
setTimeout(() => {  
  console.log('D setTimeout ...')  
}, 0)
```

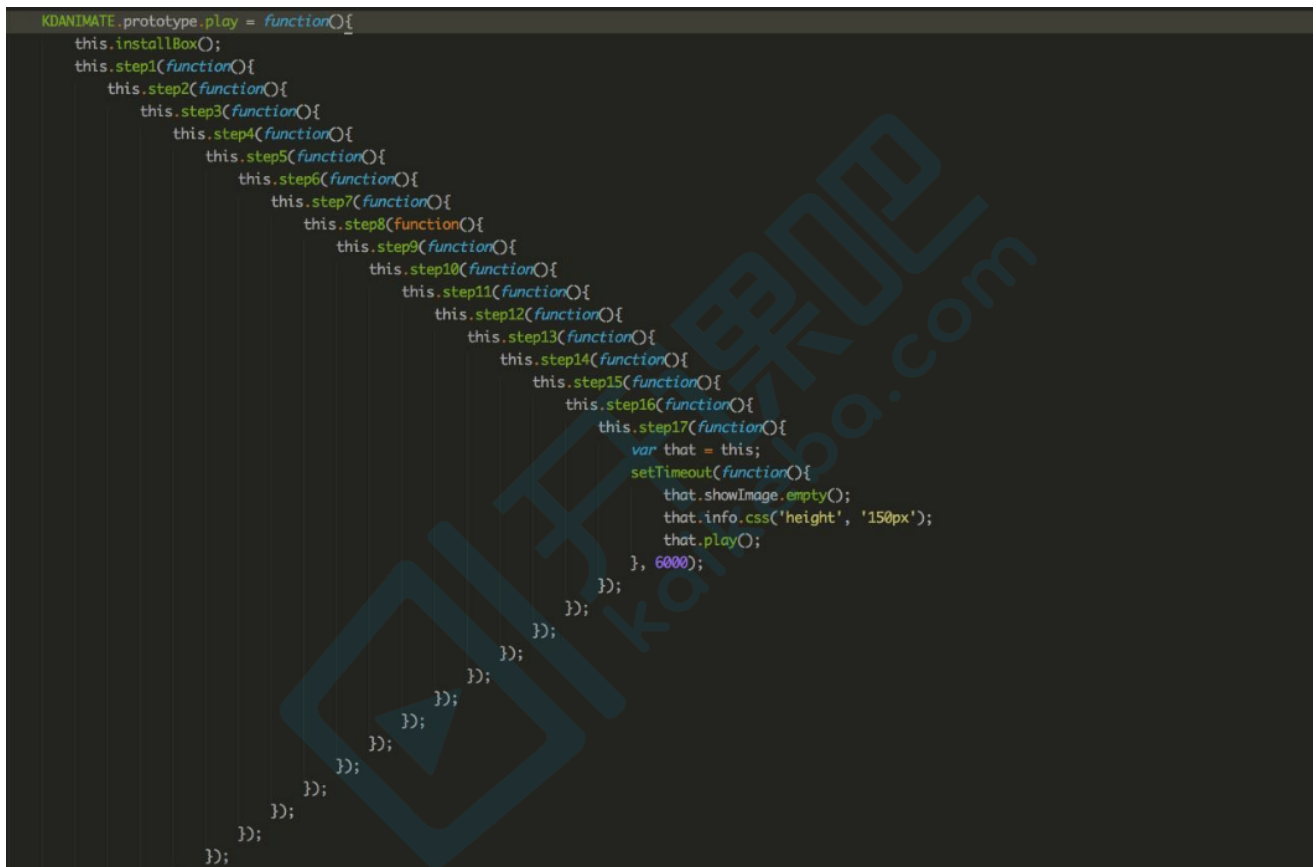
```
process.nextTick(() => {
  console.log('E nextTick ...')
})
```

Node流程控制

如何让异步任务串行化

- promise then
- ES6 Generater
- ES7 Async/Await

让一步任务串行话



```
KDANIMATE.prototype.play = function(){
  this.installBox();
  this.step1(function(){
    this.step2(function(){
      this.step3(function(){
        this.step4(function(){
          this.step5(function(){
            this.step6(function(){
              this.step7(function(){
                this.step8(function(){
                  this.step9(function(){
                    this.step10(function(){
                      this.step11(function(){
                        this.step12(function(){
                          this.step13(function(){
                            this.step14(function(){
                              this.step15(function(){
                                this.step16(function(){
                                  this.step17(function(){
                                    var that = this;
                                    setTimeout(function(){
                                      that.showImage.empty();
                                      that.info.css('height', '150px');
                                      that.play();
                                    }, 6000);
                                  });
                                });
                              });
                            });
                          });
                        });
                      });
                    });
                  });
                });
              });
            });
          });
        });
      });
    });
  });
});
```

- callback

```
const log = name => {
  console.log(`Log.....${name} ${new Date().toLocaleDateString()}`)
}

const delay = 1000
setTimeout(() => {
  logTime('Callback')
  setTimeout(() => {
    logTime('Callback')
  }, delay)
}, delay)
```

- 使用Promise

```
const promise = name => new Promise(resolve => {
  setTimeout(() => {
    resolve()
    log(name)
  }, delay)
})
promise('Promise')
  .then(() => promise('P2'))
  .then(() => {
    promise('p3')
  })
```

- Generator 和 yield 和 iterator ES6

```
const generator = function* (name) {
  yield promise(name);
  yield promise(name);
}
const gen = generator('Generator')
gen.next().value.then(() => {
  gen.next()
})
```

- 自己实现一个co库

```
let co = function(gen, name) {
  var it = gen(name);
  var ret = it.next();
  ret.value.then(function(res) {
    it.next(res);
  });
}
co(generator, 'co');
```

- Async和Await组合 ES7

```
(async() => {
  await promise('Async/await')
  await promise('Async/await')
})()
```