Node.js 基础

入门资料 https://nodejs.dev/learn

简介

- Node.js 是一个开源和**跨平台**的 JavaScript 运行时环境
- 在浏览器之外运行 V8 JavaScript 引擎
- 单线程运行
- 异步事件驱动
- 它将事件循环作为一个运行时结构，对用户是隐藏的
- HTTP 是 Node.js 中的一等公民，设计时考虑到了流式和低延迟

安装

linux/mac 推荐使用 nvm 安装 node 并管理版本

windows 推荐使用 nvs 安装 node 并管理版本

Hello World

web 服务器

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

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
    const {method, url, headers,} = req;
    // console.log({method, url, headers});

    const pathname = URL.parse(url).pathname;
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
```

```
    res.end(`Hello ${pathname}`);
});

server.listen(port, hostname, () => {
    console.log(`Server running at http://${hostname}:${port}/`);
});
```

npm 包管理

- Node.js 标准的软件包管理器
- 下载和管理 Node.js 包依赖
- 世界上最大的单一语言包管理器

基本命令

- 生成记录文件 `package.json`

```
npm init
```

- 安装依赖包

```
// 将安装到当前项目根目录下的 node_modules 文件夹里，并将版本信息记录到 package.
json 文件
npm install package

// 安装到全局环境 -g
npm install package -g

// 记录为dev环境包,包信息将记录到devDependencies
npm install package --dev
```

- 卸载依赖包

```
npm uninstall package
```

package.json

项目的清单数据，存储项目基本信息

常见属性

- `version` 表明了当前的版本。
- `name` 设置了应用程序 / 软件包的名称。
- `description` 是应用程序 / 软件包的简短描述。
- `main` 设置了应用程序的入口点。
- `private` 如果设置为 `true`，则可以防止应用程序 / 软件包被意外地发布到 `npm`。
- `scripts` 定义了一组可以运行的 node 脚本。
- `dependencies` 设置了作为依赖安装的 `npm` 软件包的列表。
- `devDependencies` 设置了作为开发依赖安装的 `npm` 软件包的列表。
- `engines` 设置了此软件包 / 应用程序在哪个版本的 Node.js 上运行。

版本号约定规则

semer 语义化版本

版本格式：主版本号 . 次版本号 . 修订号，版本号递增规则如下：

1. 主版本号：当你做了不兼容的 API 修改，
2. 次版本号：当你做了向下兼容的功能性新增，
3. 修订号：当你做了向下兼容的问题修正。

先行版本号及版本编译信息可以加到“主版本号 . 次版本号 . 修订号”的后面，作为延伸，比如希腊字母版本 base、alpha、beta 、RC 、 release 等

安装包版本设定

```
{
    "dependencies": {
        "foo": "1.0.0 - 2.9999.9999",
        "bar": ">=1.0.2 <2.1.2",          必须大于等于1.0.2版本且小于2.1.2版本
        "baz": ">1.0.2 <=2.3.4",          必须大于1.0.2版本且小于等于2.3.4版本
```

```
        "boo": "2.3.1",                    必须匹配这个版本
        "boo": "~2.3.1",                   约等于2.3.1，只更新最小版本，相当于2.3.x，即>=2.3.1 <2.4.0
        "thr": "2.3.x",
        "boo": "^2.3.1",                   与2.3.1版本兼容，相当于2.x.x，即>=2.3.1 < 3.0.0,不改变大版本号。
        "qux": "<1.0.0 || >=2.3.1 <2.4.5 || >=2.5.2 <3.0.0",
        "asd": "http://asdf.com/asdf.tar.gz",    在版本上指定一个压缩包的url，当执行npm install 时这个压缩包会被下载并安装到本地。
        "til": "~1.2",
        "elf": "~1.2.3",
        "two": "2.x",
        "lat": "latest",                安装最新版本
        "dyl": "file:../dyl",           使用本地路径
        "adf": "git://github.com/user/project.git#commit-ish"     使用git URL加commit-ish
    }
```

koa 的 package.json

```
{
  "name": "koa",
  "version": "2.13.1",
  "description": "Koa web app framework",
  "main": "lib/application.js",
  "exports": {
    ".": {
      "require": "./lib/application.js",
      "import": "./dist/koa.mjs"
    },
    "./*": "./*.js",
    "./*.js": "./*.js",
    "./package": "./package.json",
    "./package.json": "./package.json"
  },
  "scripts": {
    "test": "jest --forceExit",
    "lint": "eslint --ignore-path .gitignore .",
    "authors": "git log --format='%aN <%aE>' | sort -u > AUTHORS",
    "build": "gen-esm-wrapper . ./dist/koa.mjs",
    "prepare": "npm run build"
  },
  "repository": "koajs/koa",
  "keywords": [
    "web",
    "app",
    "http",
```

```
    "application",
    "framework",
    "middleware",
    "rack"
  ],
  "license": "MIT",
  "dependencies": {
    "accepts": "^1.3.5",
    "cache-content-type": "^1.0.0",
    "content-disposition": "~0.5.2",
    "content-type": "^1.0.4",
    "cookies": "~0.8.0",
    "debug": "^4.3.2",
    "delegates": "^1.0.0",
    "destroy": "^1.0.4",
    "encodeurl": "^1.0.2",
    "escape-html": "^1.0.3",
    "fresh": "~0.5.2",
    "http-assert": "^1.3.0",
    "http-errors": "^1.6.3",
    "koa-compose": "^4.1.0",
    "on-finished": "^2.3.0",
    "only": "~0.0.2",
    "parseurl": "^1.3.2",
    "statuses": "^1.5.0",
    "type-is": "^1.6.16",
    "vary": "^1.1.2"
  },
  "devDependencies": {
    "eslint": "^7.32.0",
    "eslint-config-standard": "^16.0.3",
    "eslint-plugin-import": "^2.18.2",
    "eslint-plugin-node": "^11.1.0",
    "eslint-plugin-promise": "^5.1.0",
    "eslint-plugin-standard": "^5.0.0",
    "gen-esm-wrapper": "^1.0.6",
    "jest": "^27.0.6",
    "supertest": "^3.1.0"
  },
  "engines": {
    "node": ">= 12"
  },
  "files": [
    "dist",
    "lib"
  ],
  "jest": {
    "testEnvironment": "node"
```

```
    }
}
```

Node 脚本启动与退出

启动

```
node xxx.js
```

退出

- 控制台中运行程序时，可以使用 `ctrl-C` 将其关闭
- 编程 `process.exit(exitCode)`

```
process.exit(0)
```

环境变量与脚本参数

环境变量：NODE_ENV

```
// linux环境
export NODE_ENV = "development"

//node环境获取 NODE_ENV为node约定环境变量配置名
process.env.NODE_ENV
```

脚本参数：process.argv

- 以数组存储
- 包含启动 Node.js 进程时传入的命令行参数
- 如有一定规则，需自己解析
- 第一个元素将是 `process.execPath`
- 第二个元素将是正在执行的 JavaScript 文件的路径

执行：

```
node process-args.js one two=three four
```

```
// process-args.js
const { argv } = require('process');

// 打印 process.argv
argv.forEach((val, index) => {
  console.log(`${index}: ${val}`);
});
```

输出：

```
0: /usr/local/bin/node
1: /Users/mjr/work/node/process-args.js
2: one
3: two=three
4: four
```

文件系统

`fs` 模块提供了许多非常实用的函数来访问文件系统并与文件系统进行交互

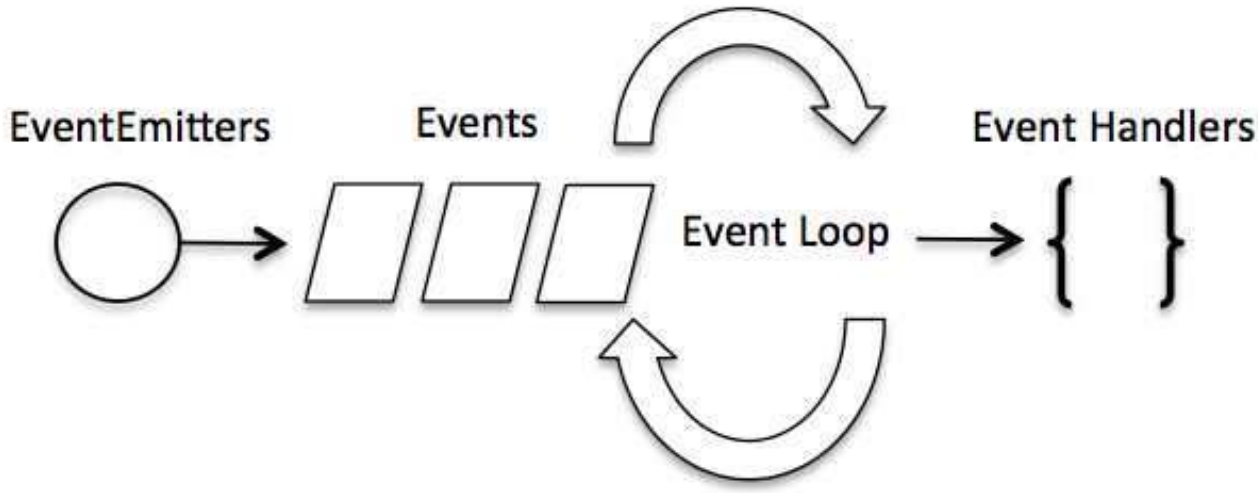
```
const { readFile } = require('fs');

readFile('./a.json', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

Events 事件触发器

- Node.js 的大部分核心 API 都是围绕惯用的异步事件驱动架构构建
- 所有触发事件的对象都是 `EventEmitter` 类的实例

- `EventEmitter` 对象触发事件时，所有绑定到该特定事件的函数都会被同步地调用
- 被调用的监听器返回的任何值都将被忽略和丢弃

事件循环



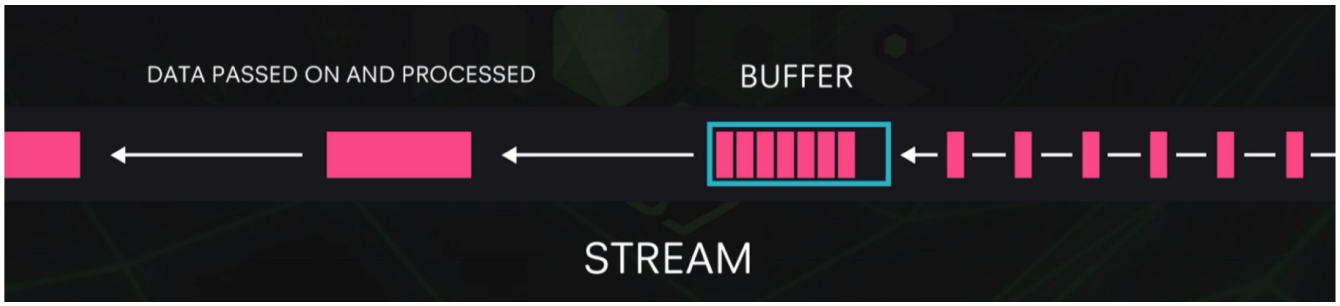
hello events

```
const EventEmitter = require('events');
const readline = require('readline')
// 继承EventEmitter, 方便扩展
class MyEmitter extends EventEmitter {}

const readlineInterface = readline.createInterface({
    input: process.stdin,
    output: process.stdout
})

// 创建 eventEmitter 对象
const myEmitter = new MyEmitter();

const readyHandler = function () {
    readlineInterface.question(`what's your name ? `, name => {
        // 触发 data_received 事件
        myEmitter.emit('data_received', name)
        readlineInterface.close()
    })
}

myEmitter.on('ready', readyHandler);
```

```
23
24 myEmitter.on('data_received', (name) => {
25     console.log(`Hello ${name}!`)
26 });
27
```

Stream 流

- 以高效的方式处理读 / 写文件、网络通信、或任何类型的端到端的信息交换
- 和 Unix 操作系统的流概念一样（管道符号 `|` ）
- 许多 Node.js 核心模块提供了原生的流处理功能

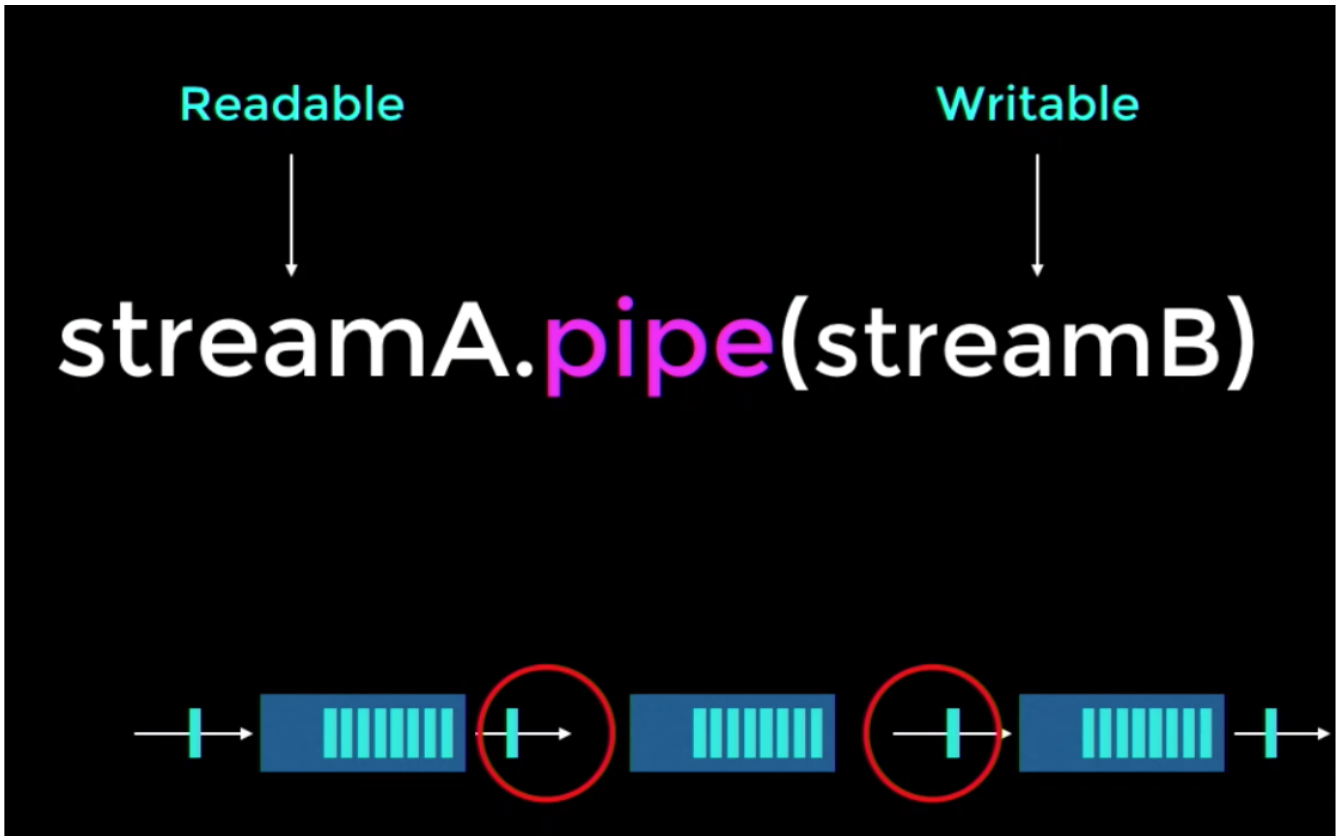


Node.js 中有四种基本的流类型：

- `Writable`：可以写入数据的流（例如，`fs.createWriteStream()`）。
- `Readable`：可以从中读取数据的流（例如，`fs.createReadStream()`）。
- `Duplex`：`Readable` 和 `Writable` 的流（例如，`net.Socket`）。
- `Transform`：可以在写入和读取数据时修改或转换数据的 `Duplex` 流（例如，`zlib.createDeflate()`）

pipe

pipe 的概念就相当于一个“水管”，将 readable 连接至 writable



大文件传输

```
const http = require('http');
const URL = require('url');
const querystring = require('querystring');
const fs = require('fs');

const hostname = '127.0.0.1';
const port = 3000;
console.log('pid', process.pid)

const server = http.createServer((req, res) => {
// `req` 是 http.IncomingMessage，它是可读流。
    // `res` 是 http.ServerResponse，它是可写流。

    const {method, url, headers,} = req;
    // console.log({method, url, headers})

    const {pathname, query} = URL.parse(req.url);// 解析url结构
    const queryObj = querystring.parse(query); // 解析出url里的query参数

    if (method === 'GET' && pathname === '/large_file') {
```

```
        if (queryObj.type === 'readFile') {
            fs.readFile(`${__dirname}/large_file.data`, (err, data) => {
                console.log('read end')
                res.end(data)
            })
            return;
        }

        const stream = fs.createReadStream(`${__dirname}/large_file.data`)
        stream.pipe(res)
        return;
    }

    res.end(`Hello ${pathname}`)
})
server.listen(port, hostname, () => {
    console.log(`Server running at http://${hostname}:${port}/`);
});

// top -pid n  查看不同文件传输方式的内存变化
// 访问 localhost:3000/large_file 使用流传输
// 访问 localhost:3000/large_file?type=readFile 读取文件后传输
```

post 数据接受

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

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
    // `req` 是 http.IncomingMessage，它是可读流。
    // `res` 是 http.ServerResponse，它是可写流。
    const {method, url, headers,} = req;
    // console.log({method, url, headers})

    const pathname = URL.parse(req.url).pathname;
    if (method === 'POST' && pathname === '/post_data') {
        let body = '';
        // 以 utf8 字符串形式获取数据。如果未设置编码，则将接收缓冲区对象。
        req.setEncoding('utf8');
        // 一旦添加了监听器，则可读流就会触发 'data' 事件。
        req.on('data', (chunk) => {
```

```
            body += chunk;
        });

        // 'end' 事件表示已经接收到整个正文。
        req.on('end', () => {
            try {
                const data = JSON.parse(body);
                // 给用户回写数据
                res.write(typeof data);
                res.end();
            } catch (err) {
                // 无法被JSON解析！
                res.statusCode = 400;
                return res.end(`error: ${err.message}`);
            }
        });
    } else {
        res.statusCode = 200;
        res.setHeader('Content-Type', 'text/plain');
        res.end(`Hello ${pathname}`);
    }
});

server.listen(port, hostname, () => {
    console.log(`Server running at http://${hostname}:${port}/`);
});

// $ curl localhost:3000/post_data -d "{}"
// object
// $ curl localhost:3000/post_data -d "\"foo\""
// string
// $ curl localhost:3000/post_data -d "not json"
// error: Unexpected token o in JSON at position 1
```

web 服务框架 KOA

- 非常精简，仅包含中间件内核和 http 模块抽象，不含任何中间件
- 基于 Promises 的控制流程，避免回调地狱
- 抽象 node 的 request/response
- 路由、解析、安全等均使用第三方库

hello world

```
const Koa = require('koa');
const app = new Koa();

app.use(async ctx => {
  ctx.body = 'Hello World';
});

app.listen(3000);
```

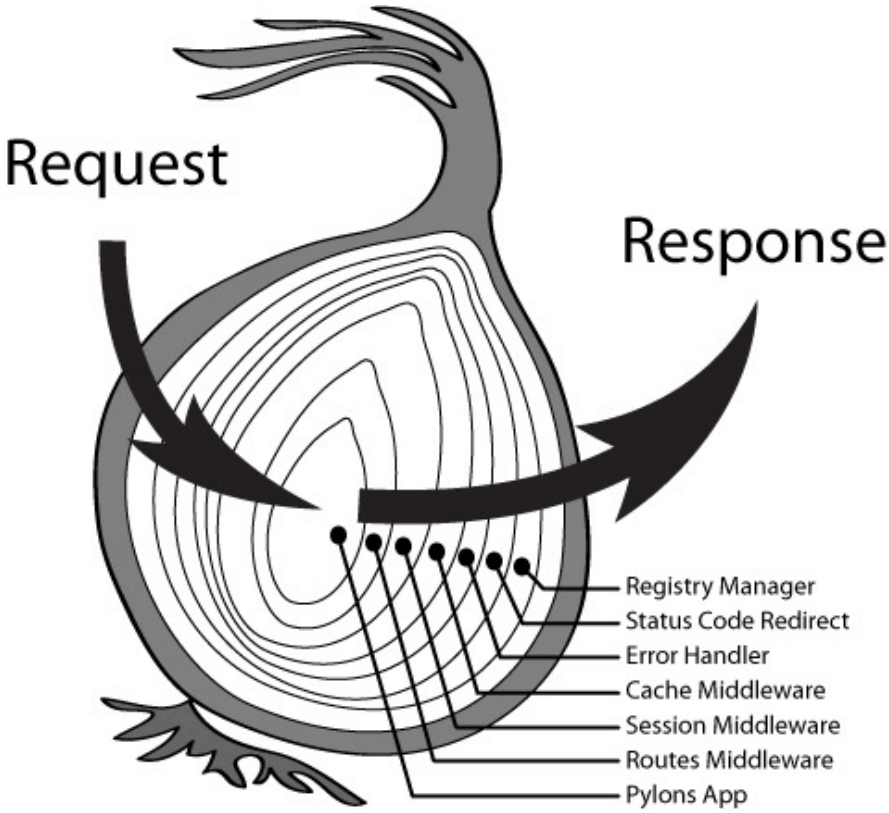
ctx 上下文

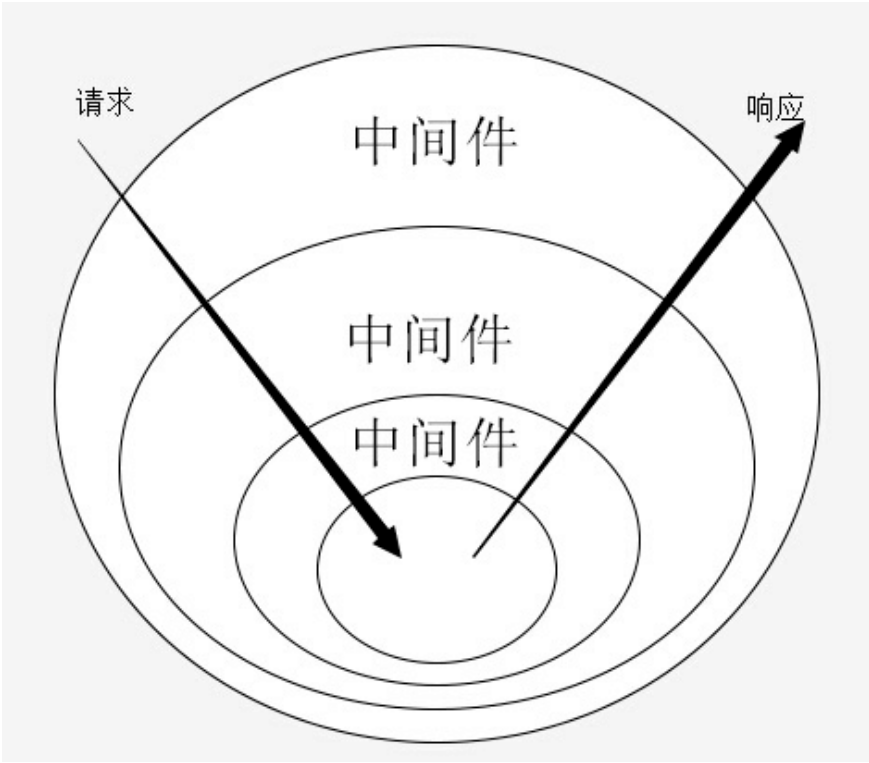
- 将 node 的 `request` 和 `response` 对象封装到了自己的 `ctx.request` 和 `ctx.response` 对象
- 为编写 Web 应用程序和 API 提供了许多有用的方法
- 每个请求都将创建一个 `Context`

参考 https://koajs.com/#context

洋葱模型与中间件

- 通过 app.use 注册中间件处理函数
- 利用 Promise 使中间件串联起来，利用 async 实现同步式的调用
- 通过 next 调用下游中间件，执行完后控制流返回上游
- next 调用返回的是 Promise

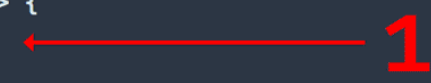





```
const Koa = require('koa');
const app = new Koa();

// x-response-time
app.use(async (ctx, next) => {
```

```
    const start = Date.now();
    await next();
    const ms = Date.now() - start;
    ctx.set('X-Response-Time', `${ms}ms`);
});

// logger
app.use(async (ctx, next) => {
    const start = Date.now();
    await next();
    const ms = Date.now() - start;
    console.log(`${ctx.method} ${ctx.url} - ${ms}`);
});

// response
app.use(async ctx => {
    ctx.body = 'Hello World';
});

app.listen(3000);
```

```
const Koa = require('koa');
const app = new Koa();

// x-response-time

app.use(async (ctx, next) => {
  const start = Date.now();
  await next();
  const ms = Date.now() - start;
  ctx.set('X-Response-Time', `${ms}ms`);
});

// logger

app.use(async (ctx, next) => {
  const start = Date.now();
  await next();
  const ms = Date.now() - start;
  console.log(`${ctx.method} ${ctx.url} - ${ms}`);
});

// response

app.use(async ctx => {
  ctx.body = 'Hello World';
});

app.listen(3000);
```



作业

1. 用 node 实现文件存储 api，客户端通过 post 请求发送文件数据
2. 用 node 实现网络文件下载脚本，可通过参数指定下载地址和存储地址，下载网络文件到本地，并显示下载进度和传输速度