

Node 和 Java 的对比

Java 特点

- 请求到来的时候可以多开一个线程去处理，也就是说多个线程去处理多个请求，适合**cpu 密集型**（运算）。每个线程会等待 I/O 的处理（阻塞式），等处理完毕再返回结果
- 同一资源上锁问题
- 多线程靠的是切换上下文，会浪费一些性能

Node 特点

- 事件驱动
- 非阻塞式 I/O（文件读写）

js主线程执行是单线程的，但是在读取文件的时候node会多开一个线程(不会阻塞js线程)去处理，当这个线程处理完毕时会通过**事件驱动**通知js线程。node是多线程的，但js执行过程是单线程的。

Node中js执行是单线程的，不需要开启多个线程，节约资源，但不适合做大量运算，适合执行一些**异步 I/O 操作**但Node可以多开一个子进程

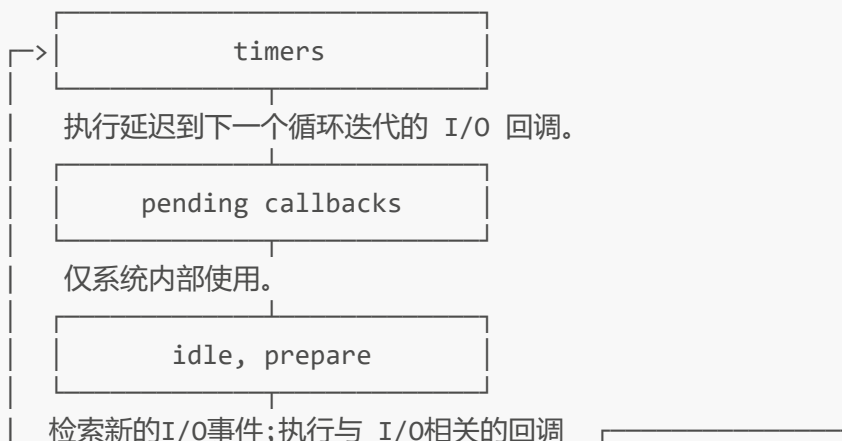
Node 中的 this 和 global

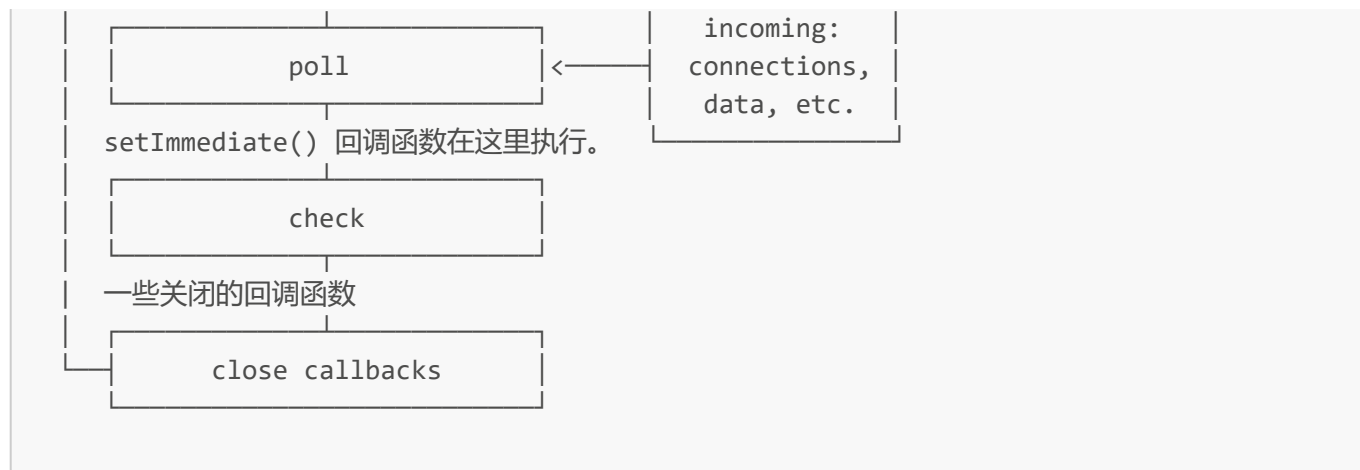
使用node命令来执行一个文件，他会把这个文件当成一个模块，默认把this修改成{}（this = module.exports = exports），但是自执行函数（箭头函数除外）里的this是global

Node 中的 event loop

最新的（11 以后）node版本中的事件环和浏览器差不多，执行一个**宏任务**然后清空**微任务**和nextTick，但对**宏任务**进行了分层，当event loop执行到某个阶段时会将当前阶段对应的队列依次执行。当该队列已用尽或达到回调限制，事件循环将移动到下一阶段。

1. 主栈的代码执行完毕以后会清空**微任务**（nextTick优先于promise.then），
2. 进入到事件环当中轮询（每执行一个宏任务会清空**微任务**和nextTick）





```

const fs = require("fs");
fs.readFile("./note.md", "utf8", () => {
  //poll的回调函数，执行完毕后会先清空setImmediate
  setTimeout(() => {
    console.log(5);
  });
  setImmediate(() => {
    console.log(3);
  });
  setImmediate(() => {
    console.log(4);
  });
});

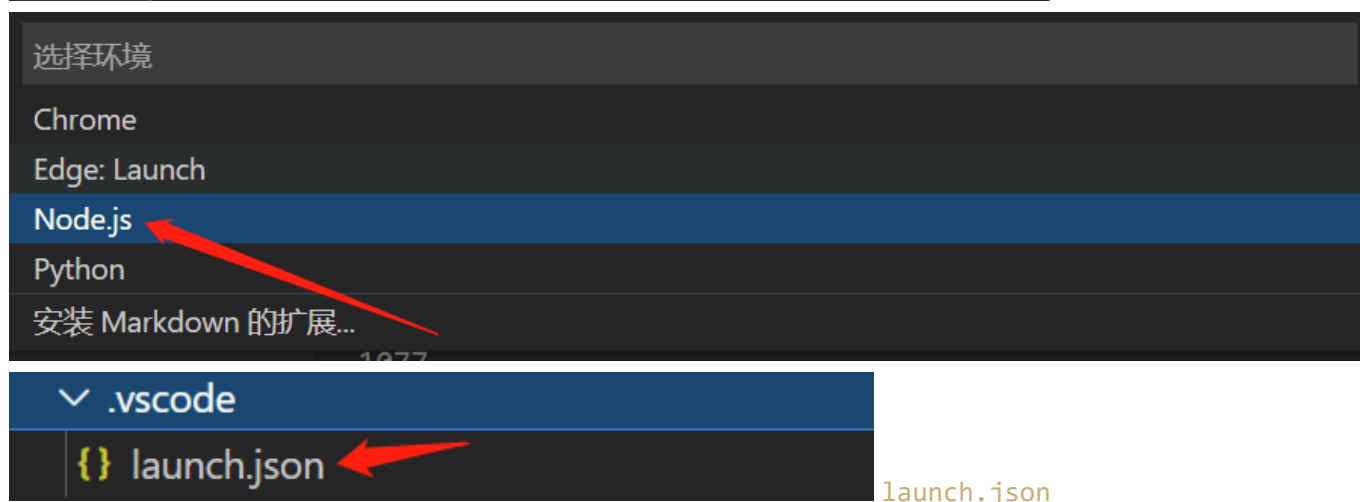
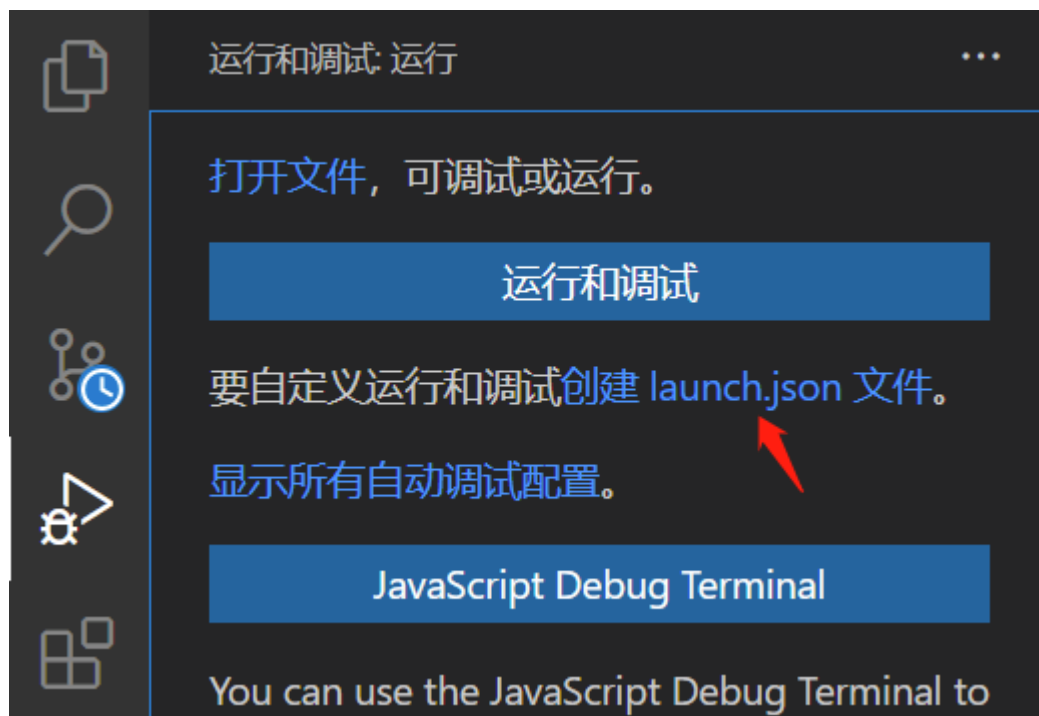
setTimeout(() => {
  console.log(1);
}, 0);
setImmediate(() => {
  console.log(2);
});
//1 2 3 4 5
  
```

模块化规范

- **amd**和**cmd**规范：用**define**字段来定义文件的依赖关系
- **commonjs**规范：基于文件读写，如果依赖了某个文件会进行文件读取，是**动态**（执行后才知道结果，不可以**tree-shaking**）的（不必须写在顶层作用域中）
 - 一个文件一个模块
 - **require**导入
 - **module.exports**导出
- **esModule**规范：每次引用一个模块就会发请求，**静态**（可以**tree-shaking**）的，只能在顶层作用域中引用一个模块（**es7**新增的**import("xxx")**也能实现动态导入）
 - 一个文件一个模块
 - **export**导出
 - **import** 导入

- `umd`规范：兼容`amd`、`cmd`和`commonjs`，但不支持`esModule`

Node 中如何调试内置模块代码



```
{
  // 使用 IntelliSense 了解相关属性。
  // 悬停以查看现有属性的描述。
  // 欲了解更多信息, 请访问: https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "type": "pwa-node",
      "request": "launch",
      "name": "Launch Program",
      "skipFiles": [
        "<node_internals>/**" //把这个删除可以看node内置模块代码
      ],
      "program": "${file}"
    }
  ]
}
```

```
}  
]  
}
```

process 进程模块

process

常用属性

1. `platform`: 运行平台
2. `cwd`: `current working directory` 当前工作目录, 在哪里运行就是哪里, 可以改变

如webpack会自动查找运行webpack的目录下查找`webpack.config.js`

3. `env`: 执行代码时传入环境
4. `argv`: 执行代码时传入的参数

```
//在终端命令行中执行set NODE_ENV=development, 再执行node index.js  
if (process.env.NODE_ENV === "development") {  
  console.log("dev");  
} else {  
  console.log("prod");  
}
```

`commander` 命令行管家, 一个nodejs接收用户命令处理模块

```
const program = require("commander");  
program.option("-p,--port <n>", "set user port");  
program.option("-f,--file <n>", "set user directory");  
program  
  .command("create")  
  .description("创建项目")  
  .action(() => {  
    console.log("创建项目");  
  });  
program.parse(process.argv);  
  
const options = program.opts();  
if (options.port) {  
  // 开启个本地服务  
}  
console.log(options);
```

5. `nextTick`: 当前执行栈的底部

node中自己实现的，不属于node中的EventLoop，优先级比promise更高

开启子进程

进程和线程的概念：

- 进程 是系统进行资源分配和调度的基本单位
- 线程 是被包含在进程中的，是进程中实际运作的单位

node运行时主进程中只有一个主线程，如果主线程挂了那整个应用就终止了。单线程无法充分利用cpu内核，现在用node还是采用开启子进程的方式

一个问题：

```
//如果访问/sum服务器会卡主，马上再开一个浏览器窗口访问/浏览器也会卡住，因为服务器还没有处理完上一个结果
const server = http.createServer((req, res) => {
  if (req.url === "/sum") {
    // node中的线程交互 可以利用事件环 回调
    let sum = 0;
    for (let i = 0; i < 100 * 10000; i++) {
      sum += i;
    }
    res.end("total", sum);
  } else {
    res.end("other");
  }
});

server.listen(5000);
```

为了提高性能可以在node中开启子进程，专门做一些运算的处理，最后再将结果传递给主进程

```
/*
$ node argv.js --port 3000 --info abc
{ port: '3000', info: 'abc' }
*/
let argv = process.argv.slice(2).reduce((memo, current, index, arr) => {
  if (current.startsWith("--")) {
    memo[current.slice(2)] = arr[index + 1];
  }
  return memo;
}, {});
console.log(argv);
```

使用 spawn 开启子进程

./worker/child.js

```

let sum = 0;
for (let i = 0; i < 100 * 100 * 10000; i++) {
  sum += i;
}
// console.log process.stdout.write 是调用的同一个方法
// process.stderr.write

//子给父数据
process.stdout.write(process.argv.slice(2).toString());

//子给父数据
process.stdout.write("sum:" + sum);

process.stdin.on("data", function (data) {
  //父给子数据
  console.log(data.toString(), "SON");
});

```

./parent.js

```

// node中提供了 child_process 作为创建子进程的模块s

const { spawn } = require("child_process"); // 产卵
const path = require("path");
// 使用流的好处就是可以输出一点拿到一点，坏处就是写起来麻烦

// process.stdin // 用户的标准输入 0
// process.stdout // 用户的标准输出 1
// process.stderr // 用户的错误输出 2 fs.open(function(fd){})

// 相当于命令行中 node ./worker/child.js a b c
const cp = spawn("node", ["child.js", "a", "b", "c"], {
  // fs.createReadStream
  cwd: path.resolve(__dirname, "worker"),
  // stdio:'ignore' // 忽略子进程的输出
  // stdio: 'inherit' // 将子进程中的process 改为了父进程的process 方便输出，我们可以
  // 通过数组的方式指定stdio 也可以通过 inherit来默认指定
  stdio: ["pipe", "pipe", 2], // 共享错误输出 但是标准输入和标准输出之间建立一个管道
});

//子给父数据
cp.stdout.on("data", function (data) {
  console.log(data.toString());
});

//父给子数据
cp.stdin.write("父给子数据");

cp.on("error", function (err) {
  console.log(err);
});

```

```
});
cp.on("exit", function () {
  console.log("子进程退出了");
});
cp.on("close", function () {
  console.log("子进程关闭");
});
```

使用 fork 开启子进程（基于 spawn）

fork默认会用node执行

./worker/child.js

```
let sum = 0;
for (let i = 0; i < 100 * 100 * 10000; i++) {
  sum += i;
}

process.send("hello", function () {
  console.log("子进程给父进程发送消息成功");
});

process.on("message", function (data) {
  console.log("收到了父亲的消息", data.toString());

  process.exit(); //子进程退出, 结束ipc
});
```

./parent.js

```
const { fork } = require("child_process");
const path = require("path");

const cp = fork("sum1.js", {
  // = spawn fork比spawn用起来更方便 提供了ipc的方式
  cwd: path.resolve(__dirname, "worker"),
  // stdio:[0,1,2,'ipc'] // 默认 process.stdin stdout stderr 通过ipc来进行通信 ipc
  // 不会断开 可以使用cp.kill() 或process.kill(cp.pid)来断开并结束子进程
});

cp.on("message", function (data) {
  console.log("儿子给我的数据", data);
  // process.nextTick(()=>{
  //   process.kill(cp.pid); // pid 就是进程的唯一标识 kill 来杀死进程
  // })

  //父给子发数据
  cp.send("welcome", function () {});
```

```
});

cp.on("exit", function () {
  console.log("exit");
});
```

用fork，就是为了用ipc(inter process communication)，但是有的时候，仅仅为了拿到子进程的输出，不作别的事，可以使用execFile

使用 execFile/exec 获取执行后的输出（基于 spawn）

execFile和exec的唯一区别就是默认会开启shell，也就是说使用exec会在命令行中执行，可以拿到像path这样的环境变量

```
const { execFile, spawn, fork, exec } = require("child_process");
const path = require("path");

// 在node里启动子进程 定时爬数据， 批量执行打包操作， 执行一些sh脚本 会用到

// execFile 执行的时候会传递一个最大的输出限制 maxBuffer
// shell 这个命令不在命令行中执行 shell: 默认false ,传递的参数的方式是数组
// const cp = execFile('node',['--version'],{
//   cwd:path.resolve(__dirname,'worker')
// },function(err,stdout,stderr){ // 回调不支持大的输出，大的输出得用流
//   console.log(err)
//   console.log(stdout)
// })

//与execFile的唯一区别：默认开启shell
const cp = exec(
  "path",
  {
    // 注意这种方式不要直接将用户的输入作为结果，可能会导致危险操作
    cwd: path.resolve(__dirname, "worker"),
  },
  function (err, stdout, stderr) {
    // 回调不支持大的输出，大的输出得用流
    console.log(err);
    console.log(stdout);
  }
);

// cp.stdout.on('data',function(chunk){
//   console.log(chunk.toString())
// })

// process 中就这5个方法 spawn fork execFile exec execFileSync
// 父进程如果挂了 子进程一定会终止
```

使用 cluster 创建集群

集群与分布式的概念区别：

- 集群：10个项目，10个人来做，每个做不同的项目，集群都是多个人干一件事
- 分布式：10个项目，10个人，一个个来做

node中集群如何实现的？实现的时候，内部创建子进程用的是fork，会用一个叫NODE_UNIQUE_ID环境变量来标识子进程，子进程在使用http.createServer创建服务的时候并没有真正的创建一个服务，而是将创建的信息传递给主进程，主进程接收到第一个子进程服务的时候会根据创建信息创建服务，在接收到后面的子进程的创建信息时不会创建服务，而是将他们存起来，等请求到来的时候主进程会通过轮询方式分发给子进程去处理

```
const cluster = require("cluster"); // 多个人做同一件事 多个进程
const http = require("http");
const path = require("path");
const cpus = require("os").cpus();

if (cluster.isMaster) {
  // 默认执行肯定是主进程
  for (let i = 0; i < cpus.length; i++) {
    cluster.fork(); // child_process.fork
  }
} else {
  // 调用fork的时候 会默认让此文件再次执行 但是这时候isMaster为false
  const server = http.createServer((req, res) => {
    res.end("child" + process.pid);
  });
  server.listen(4000); // 不是同一个服务被监听多次，而是只有一个服务，负责分发
}
```

Node 核心模块

- fs.readFileSync
- fs.existsSync
- path.resolve：解析绝对路径，解析默认采用 process.cwd() 如果有路径/ 会回到根目录
- path.join：仅仅是拼接，不会产生绝对路径，遇到/ 也会拼在一起

有/就用join，没有/就用resolve

- path.extname
- path.basename
- path.relative：根据路径获取相对路径
- path.dirname：取当前文件的父路径

vm 沙箱模块

- runInThisContext：在全局上下文中执行

一个node.js文件中的作用域是使用函数function
(exports,module,require,__dirname,filename){ }包裹过后的, 也即:

```
const vm = require("vm");
global.a = 1;
var b = 2;

vm.runInThisContext("console.log(typeof a);"); //number
vm.runInThisContext("console.log(typeof this.a);"); //number this相当于global
vm.runInThisContext("console.log(typeof b);"); //undefined
vm.runInThisContext("console.log(typeof require);"); //undefined
```

- `runInNewContext`是在一个全新的作用域下执行, 也拿不到`global`的属性

```
const vm = require("vm");
global.a = 1;
var b = 2;

let obj = {c:3}
vm.runInNewContext("a = typeof a",obj);
vm.runInNewContext("b = typeof b",obj);
vm.runInNewContext("global = typeof global",obj);
vm.runInNewContext("c = typeof c",obj);

console.log(obj);//{ c: 'number', a: 'undefined', b: 'undefined', global:
'undefined' }
```

Node 中的 commonjs 规范实现原理

1. 获得绝对路径文件名
2. 查看缓存, 如果有就直接返回
3. 创建一个`module`对象, 里面有一个`exports`空对象, 将这个`module`缓存下载
4. 根据不同的文件后缀名执行不同的策略
 - `.json`: 用`JSON.parse`解析文件内容, 赋值给`module.exports`
 - `.js`: 将文件内容前后包裹成一个函数字符串, 这个函数接收`module`、`exports`、`__dirname`、`__filename`、`require`五个参数, 然后用`vm.runInThisContext`执行后返回一个实体函数`fn`, 然后用`fn.call`将`this`改为`module.exports`, 放入参数去执行, 执行完毕后返回`module.exports`

```
function Module(id) {
  this.id = id;
  this.exports = {};
}
Module._cache = {};
Module._extensions = {
```

```

".js"(module) {
  let script = fs.readFileSync(module.id, "utf8");
  let templateFn = `(function(exports,module,require,__dirname,__filename)
${script})`;
  let fn = vm.runInThisContext(templateFn);
  let exports = module.exports;
  let thisValue = exports; // this = module.exports = exports;
  let filename = module.id;
  let dirname = path.dirname(filename);

  //this = module.exports = exports
  // 函数的call 的作用 1.改变this指向 2.让函数执行
  fn.call(thisValue, exports, module, req, dirname, filename); // 调用了a模块
  module.exports = 100;
},
".json"(module) {
  let script = fs.readFileSync(module.id, "utf8");
  module.exports = JSON.parse(script);
},
};
Module._resolveFilename = function (id) {
  let filePath = path.resolve(__dirname, id);
  let isExists = fs.existsSync(filePath);
  if (isExists) return filePath;
  // 尝试添加后缀
  let keys = Object.keys(Module._extensions);

  for (let i = 0; i < keys.length; i++) {
    let newPath = filePath + keys[i];
    if (fs.existsSync(newPath)) return newPath;
  }
  throw new Error("module not found");
};
Module.prototype.load = function () {
  let ext = path.extname(this.id); // 获取文件后缀名
  Module._extensions[ext](this);
};

function require(filename) {
  filename = Module._resolveFilename(filename); // 1.创建一个绝对引用地址, 方便后续读取
  let cacheModule = Module._cache[filename];
  if (cacheModule) return cacheModule.exports;

  const module = new Module(filename); // 2.根据路径创建一个模块
  Module._cache[filename] = module; // 缓存模块 根据的是文件名来缓存
  module.load(); // 就是让用户给module.exports 赋值
  return module.exports;
}

```

Node 中的模块查找过程

核心模块 -> 第三方模块 -> 文件模块

核心模块查找

如`require("fs")`

第三方模块查找

如`require("co")`

会根据`module.paths`的路径查找，也即先查找这个文件当前目录下的`node_modules`，如果没有会一直查找上一个文件夹下的`node_modules`，直到根目录下的`node_modules`

文件模块查找

如`require("./a/b.js")`

1. 查找同名文件：如果没有会尝试添加后缀查找`.js`、`.json`文件
2. 查找同名文件夹：先看`package.json`中的`main`中的文件，然后会看有没有`index`的文件

npm 相关知识

npm: node package manager

注意`npm install packageName -d`不会将包安装成开发依赖，`-d`和`-D`是不一样的，`npm`不识别`-d`

```
npm root -g

#安装生产依赖和开发依赖
npm install

#只安装生产依赖
npm install --production

#打包，默认不包含node_modules
npm pack
```

npm: node registry manager

```
nrm use taobao
nrm use npm
nrm use cnpm
```

nvm: node version manager

```
nvm install stable ## 安装最新稳定版 node
nvm install <version> ## 安装指定版本
nvm uninstall <version> ## 删除已安装的指定版本
nvm use <version> ## 切换使用指定的版本node
nvm ls ## 列出所有安装的版本
nvm ls-remote ## 列出所有远程服务器的版本
nvm current ## 显示当前的版本
nvm alias <name> <version> ## 给不同的版本号添加别名
nvm unalias <name> ## 删除已定义的别名
nvm reinstall-packages <version> ## 在当前版本 node 环境下, 重新 全局安装指定版本号的 npm 包
nvm alias default [node版本号] ##设置默认版本
```

全局安装

```
npm install packageName -g
```

全局安装只能在命令行中使用, 不能在文件中使用

本地安装

```
npm install packageName
```

如果执行`npm install`的时候当前目录下没有`package.json`那么会一直向上级目录查找有`package.json`的文件夹, 如果没有找到这样的文件夹, 它会在当前目录下创建`package.json`, 再安装

依赖关系

- 开发依赖, 如`webpack gulp`

```
npm install webpack --save-dev
```

- 生产依赖, 如`jquery`

```
npm install jquery --save
```

- `peerDependencies`同等依赖: 如`bootstrap`依赖`jquery`就会提示去安装`jquery`, 仅仅起到提示作用
- 可选依赖: 仅仅起到提示作用
- 打包依赖: 执行`npm pack`时会加入到`node_modules`的包

命令行工具开发

新建一个`global-module`文件夹, 里面创建`gm1.js`、`gm2.js`、`package.json`, 文件内容如下

```
//package.json
{
  "name": "global-module",
  "bin": {
    "gm1": "./gm1.js",
```

```

    "gm2": "./gm2.js"
  }
}

//gm1.js
#!/usr/bin/env node
console.log("global-module1111");

//gm2.js
#!/usr/bin/env node
console.log("global-module2222");

```

然后使用 `npm link` 命令链接到电脑的 `npm` 目录下

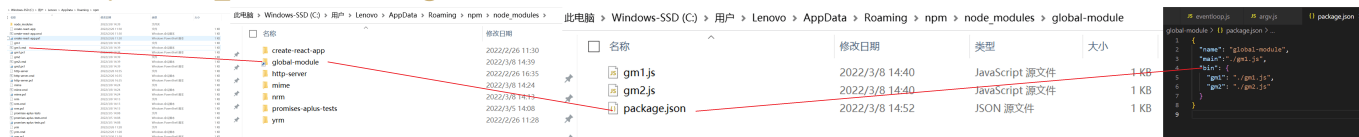
```

Lenovo@DESKTOP-RTLA60R MINGW64 ~/Desktop/workspace/前端学习/Node/global-module
(master)
$ npm link

```

npm link 原理

在电脑的环境变量中有一个 `npm` 的目录，里面存放有很多 `.cmd` 文件，在命令行中使用这些命令，就会找到这些文件执行，执行 `npm link` 相当于在 `npm` 文件夹下创建一个可执行文件，然后把它链接到 `global-module`（其实是先在 `npm/node_modules` 下创建 `global-module` 文件夹的快捷方式，然后让可执行文件指向该快捷方式）



npm run 和 npx

`npm run script` 会默认在执行命令之前，将 `node_modules/.bin` 添加到环境变量下，然后再在 `package.json` 中查找 `script`，执行命令，命令执行完毕后会删掉这个环境变量

`npx` 和 `npm` 类似，但是如果模块不存在会先安装，使用后删除

```
npx mime 1.js
```

```

{
  "name": "4.node-module",
  "version": "3.1.0",
  "description": "",
  "main": "1.module.js",
  "scripts": {
    "mime": "mime a.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {

```

```
    "mime": "^2.5.2"
  },
  "dependencies": {
    "bootstrap": "^4.6.0",
    "jquery": "^2.2.4"
  },
  "peerDependencies": {},
  "optionalDependencies": {},
  "bundledDependencies": ["bootstrap"]
}
```

版本控制

版本格式: `major.minor.patch`

不兼容旧版的修改时要更新 `major` 版本号

- `^2.2.4`: 指定 `major`
- `~2.2.4`: 指定 `major.minor`
- `>=2.1`: 版本号大于或等于 2.1.0
- `<=2.2`: 版本号小于或等于 2.2

使用 `npm version` 升级版本

```
#v2.0.0=>v3.0.0
npm version major

#v1.10.0=>v1.11.0
npm version minor

#v2.0.0=>v2.0.1
npm version patch
```

预发版:

- `alpha`: 内部测试版
- `beta`: 测试版
- `rc`: 最终测试版本

发布 npm 包

每次发布包的时候都需要升级版本号

使用 `.npmignore` 可以忽略要发布的文件

```
#需要先切换到npm源
npm use npm

npm addUser

npm publish

npm unpublish
```

模板引擎原理

模板符号以外的字符拼接成字符串，模板符号以内的字符拼接成js执行的样子，然后用with将字符串包裹起来，最外层再包裹成一个函数形式，再调用new Function返回一个函数，最后传入参数去执行这个函数

```
async function renderFile(filename, options) {
  let content = await read(filename, "utf8");
  content = content.replace(/<%(.*?)%>/g, function () {
    return "${" + arguments[1] + "}"; // 获取对应的内容做这件事
  });
  let head = 'let str = "";\\nwith(obj){\\n str+=`';
  let body = (content = content.replace(/<%(.*?)%>/g, function () {
    return "\\n" + arguments[1] + "\\nstr+=`";
  }));
  let tail = "`} return str";
  let fn = new Function("obj", head + body + tail);
  return fn(options);
}
```

events 模块（Node 中的发布订阅）

events 模块的简单实现

```
function EventEmitter() {
  this._events = {};
}
EventEmitter.prototype.on = function (eventName, callback) {
  if (!this._events) {
    this._events = {};
  }
  if (this._events[eventName]) {
    this._events[eventName].push(callback);
  } else {
    this._events[eventName] = [callback];
  }
};
```



```
EventEmitter.prototype.emit = function (eventName, ...args) {
  this._events[eventName].forEach((fn) => {
    fn(...args);
  });
};
EventEmitter.prototype.off = function (eventName, callback) {
  if (this._events && this._events[eventName]) {
    // 如果存储的方法 和 传入的不一样就留下，一样的就不要了
    this._events[eventName] = this._events[eventName].filter(
      (fn) => fn !== callback && fn.l !== callback
    );
  }
};
EventEmitter.prototype.once = function (eventName, callback) {
  const one = () => {
    // 绑定执行完毕后移除
    callback(); // 切片编程 就是增加逻辑
    this.off(eventName, one);
  };
  one.l = callback; // 自定义属性
  this.on(eventName, one);
};
module.exports = EventEmitter;
```

让 Node 增加代码提示

`npm install @types/node` 可以支持node提示 (仅仅是安装了ts的提示而已，为了方便)

Buffer 模块

基本单位：8 位 -> 1 个字节 1024 个字节 -> 1k 1024k -> 1m

1 个字节可表示 0~255 ($2^{*8}-1$)

后台获取的数据都是buffer，包括后面的文件操作也都是buffer形式

buffer大小在创建时固定，不能更改

编码规范/编码的发展史

编码的发展史：ASCII (美国人) -> GB18030/GBK -> unicode -> UTF8

单字节 (英文字母，符号.. 都是一个字节)

中国为了能标识自己 (gb2312/GB18030/GBK) 对于文字来说是由2个字节组成的

unicode 希望统一所有编码 -> 可变字节长度交 没有统一成功

utf组织解决了这个问题 (utf8编码 一个汉字有3个字节组成)

utf8是实现unicode的标准之一，全部统一成utf8, node 不支持gbk 只支持utf8，因此 Buffer.from("汉")返回3个字节

创建 buffer

一般情况下，会用alloc来声明一个buffer，或者把字符串转换成buffer使用

```
let buffer1 = Buffer.alloc(5);
console.log(buffer1[0]); // 像数组（但是和数组有区别），数组可以扩展，buffer不能扩展，
// 可以用索引取值，取出来的内容是10进制

// 此方法用的非常少，我们不会直接填16进制
let buffer2 = Buffer.from([0x25, 0x26, 300]); // 超过255 会取余
console.log(buffer2[0]); //37

let buffer3 = Buffer.from("前端"); //6个字节
console.log(buffer3); //<Buffer e5 89 8d e7 ab af>
```

base64

base64字符串可以放到任何路径的链接里（减少请求的发送）文件大小会变大，base64转化完毕后会比之前的文件大1/3

base64编码原理：拿一个汉字来举例，将汉字转化为utf8格式的2进制，一共3个字节24位，base64要求1个字节不超过十进制的64，也就是不超过2进制的111111，原来是3*8共24位，转换为4*6共24位（但是总长度会在前面补0，也就成了32位），然后再将字节转换为十进制，将这个十进制对照一个索引表取值，最后得到长度为4的字符

将buffer编码为base64: buffer.toString("base64")

```
function chineseCharToBase64(chineseChar) {
  const r = Buffer.from(chineseChar);
  let binary = "";
  for (let i = 0; i < 3; i++) {
    binary += r[i].toString(2);
  }

  console.log("binary", binary); //binary 111001011000100110001101

  const base64Binary = [];
  for (let i = 0; i < 4; i++) {
    //3 x 8 => 6 * 4
    base64Binary[i] = binary.slice(i * 6, i * 6 + 6);
  }

  console.log("base64Binary", base64Binary); //[ '111001', '011000', '100110', '001101' ]

  const base64TableIndexArray = base64Binary.map((b) => parseInt(b, 2));
```

```
// 0-63 取值范围是 64
let str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
str += str.toLocaleLowerCase();
str += "0123456789+/";

let base64 = "";
for (let i = 0; i < 4; i++) {
  base64 += str[base64TableIndexArray[i]];
}
return base64;
}
```

将 Buffer 转为指定编码字符串

- `buffer.toString("utf8")`
- `buffer.toString("base64")`

Buffer.isBuffer

`buffer.length = buffer.byteLength`

这两个都是返回的字节长度

buffer.slice 截取

`slice`

`buffer`内部存的是引用地址，当使用`slice`截取`buffer`的时候，返回的新`buffer`里存的是上一个`buffer`的引用地址，因此更改新的`buffer`也会改掉原`buffer`。

```
let buffer4 = Buffer.from([1, 2, 3, 4, 5]); // 内部存的是引用地址
let sliceBuffer = buffer4.slice(0, 1);
sliceBuffer[0] = 100;
console.log(buffer4); //<Buffer 64 02 03 04 05>
```

buffer.copy 拷贝

`buffer.copy(targetBuffer, targetStart, sourceStart = 0, sourceEnd = this.length)`方法是相当于是深拷贝，修改`targetBuffer`不会造成对源`buffer`的修改

```
const buf1 = Buffer.from("前");
const buf2 = Buffer.from("端");
const bigBuf = Buffer.alloc(6);
buf1.copy(bigBuf, 0, 0, 3);
buf2.copy(bigBuf, 3, 0, 3);
console.log(bigBuf.toString()); //前端
```

```
console.log(buf1, buf2, bigBuf); //<Buffer e5 89 8d> <Buffer e7 ab af> <Buffer e5 89 8d e7 ab af>
bigBuf[0] = 100;
console.log(buf1); //<Buffer e5 89 8d>
```

Buffer.concat

`concat`相当于深拷贝，修改拼接后的`buffer`不会造成对源`buffer`的修改

```
const buf1 = Buffer.from("前");
const buf2 = Buffer.from("端");
const bigBuf = Buffer.concat([buf1, buf2], 100); //会生成100个字节，然后将他们拼接，
拼接完后剩余部分都是0
bigBuf[0] = 100;
console.log(buf1); //<Buffer e5 89 8d>
```

自定义方法 Buffer.prototype.split

```
Buffer.prototype.split = function (sep) {
  let len = Buffer.from(sep).length; // 强制将字符串转化成buffer
  let offset = 0;
  let current;

  let arr = [];
  while (-1 !== (current = this.indexOf(sep, offset))) {
    arr.push(this.slice(offset, current));
    offset = current + len;
  }
  arr.push(this.slice(offset));

  return arr;
};
```

fs 文件模块

`fs.readFile`：读取的时候默认不写编码是`buffer`类型，如果文件不存在则报错

`fs.writeFile`：写入的时候默认会将内容以`utf8`格式写入（默认会调用`toString`方法），如果文件不存在会创建

这种方法只能是读完后写，适用于小文件，大文件用这样的方法会导致淹没可用内存（例如内存 8 个 g，文件 3 个 g，淹没了 3 个 g）

可以用`fs.open`、`fs.read`、`fs.write`、`fs.close`实现边读边写

复制操作

```

const fs = require("fs");

function copy(source, target, cb) {
  // 使用3个字节来实现一个拷贝功能
  const BUFFER_SIZE = 3;
  const buffer = Buffer.alloc(BUFFER_SIZE);
  let r_offset = 0;
  let w_offset = 0;
  // 读取一部分数据 就 写入一部分数据
  // w 写入操作 r 读取操作 a 追加操作 r+ 以读取为准可以写入操作 w+ 以写入为准可以执行
  // 读取操作
  // 权限 3组 rwx组合 421 = 777(八进制) (当前用户的权限 用户所在的组的权限 其他人
  // 权限)
  // 0o666 = 438
  // exe 文件 bat 文件能执行的文件
  // 读取的文件必须要存在, 否则会报异常, 读取出来的结果都是buffer类型
  // 写入文件的时候文件不存在会创建, 如果文件有内容会被清空
  fs.open(source, "r", function (err, rfd) {
    fs.open(target, "w", function (err, wfd) {
      // 异步回调的方式实现功能 需要用递归
      // 同步代码 可以采用while循环
      function next() {
        fs.read(
          rfd,
          buffer,
          0,
          BUFFER_SIZE,
          r_offset,
          function (err, bytesRead) {
            // bytesRead真正读取到的个数
            if (err) return cb(err);
            if (bytesRead) {
              fs.write(
                wfd,
                buffer,
                0,
                bytesRead,
                w_offset,
                function (err, written) {
                  r_offset += bytesRead;
                  w_offset += written;
                  next();
                }
              );
            } else {
              fs.close(rfd, () => {});
              fs.close(wfd, () => {});
              cb();
            }
          }
        );
      }
    });
  });
}

```

```
        next();
    });
});
}
```

目录操作

创造目录

回调方式

```
function mkdir(pathStr, cb) {
    let pathList = pathStr.split("/"); // [a,b,c,d]
    let index = 1;
    function make(err) {
        // co模型
        if (err) return cb(err);
        if (index === pathList.length + 1) return cb();
        let currentPath = pathList.slice(0, index++).join("/"); // [a] [a,b]
        fs.stat(currentPath, function (err) {
            if (err) {
                fs.mkdir(currentPath, make);
            } else {
                make();
            }
        });
    }
    make();
}
```

promise方式

```
const fs = require("fs").promises; // node.11后可以直接.promises
const { existsSync, exists } = require("fs");
async function mkdir(pathStr) {
    let pathList = pathStr.split("/");
    for (let i = 1; i <= pathList.length; i++) {
        let currentPath = pathList.slice(0, i).join("/");
        if (!existsSync(currentPath)) {
            await fs.mkdir(currentPath);
        }
    }
}
```

删除目录

目录相关API

目录相关API:

- `fs.rmdir` `fs.rmdirSync`
- `fs.readdir` 查看目录中的子列表, 数组
- `fs.stat` 文件状态, 文件的信息 修改时间、创建时间、目录状态 (`isFile` `isDirectory`)
- `fs.unlink` 删除文件 (`fs.rename`)

串行

回调方式

```
const fs = require("fs");
const path = require("path");
function rmdir(dir, cb) {
  // 写递归 不要思考多层, 先把父子写明白
  fs.stat(dir, function (err, statObj) {
    if (statObj.isDirectory()) {
      fs.readdir(dir, function (err, dirs) {
        // dirs=>[a.js,b]
        dirs = dirs.map((item) => path.join(dir, item)); // [a/a.js,a/b]
        // 把目录里面的拿出来 1个删除完毕后删除第二个
        let index = 0;
        function step() {
          // 将儿子都删除完毕后删除自己即可
          if (index === dirs.length) return fs.rmdir(dir, cb);
          // 删除第一个成功后继续调用step继续删除, 直到全部删除完毕后 删除自己
          rmdir(dirs[index++], step);
        }
        step();
      });
    } else {
      // 如果是文件直接删除即可
      fs.unlink(dir, cb);
    }
  });
}
```

串行广度遍历

```
const fs = require("fs").promises;
const path = require("path");
async function rmdir(dir) {
  let stack = [dir]; // 先把根放入
  let index = 0; // 指针
  let currentNode; // 不停的移动指针
  while ((currentNode = stack[index++])) {
    let statObj = await fs.stat(currentNode); // 看下指针指到的文件
    if (statObj.isDirectory()) {
      // 如果是目录, 将子节点存放到栈中
    }
  }
}
```

```

    let dirs = await fs.readdir(currentNode); // 读取目录是包含文件的
    dirs = dirs.map((item) => path.join(currentNode, item));
    stack = [...stack, ...dirs];
  }
}
// let len = stack.length
// while () {
// 倒叙删除即可
// }
}

```

并发

基于回调的并发

```

function rmdir(dir, cb) {
  fs.stat(dir, function (err, statObj) {
    if (statObj.isDirectory()) {
      fs.readdir(dir, function (err, dirs) {
        dirs = dirs.map((item) => path.join(dir, item));
        if (!dirs.length) {
          return fs.rmdir(dir, cb);
        }
        let i = 0;
        function done() {
          if (++i == dirs.length) {
            return fs.rmdir(dir, cb);
          }
        }
        for (let i = 0; i < dirs.length; i++) {
          rmdir(dirs[i], done);
        }
      });
    } else {
      // 如果是文件直接删除即可
      fs.unlink(dir, cb);
    }
  });
}

```

基于 Promise.all 的并发

```

const fs = require("fs").promises;
const path = require("path");
async function rmdir(dir) {
  let statObj = await fs.stat(dir);
  if (statObj.isDirectory()) {
    let dirs = await fs.readdir(dir);

```



```
    await Promise.all(dirs.map((item) => rmdir(path.join(dir, item))));
    await fs.rmdir(dir);
  } else {
    return fs.unlink(dir);
  }
}
```

流

fs 可读/写流 的原理

`fs.createReadStream`内部原理是继承`events`和`stream.readable`类，当调用`rs.read`方法时，实际是调用父类的`read`方法，里面会调用子类的`_read`方法，使用发布订阅模式对`fs.open`和`fs.read`进行封装，将其解耦合

`fs.createWriteStream`内部原理是继承`events`和`stream.writable`类，当调用`rs.write`方法时，实际是调用父类的`write`方法，里面会调用子类的`_write`方法，使用发布订阅模式对`fs.open`和`fs.write`进行封装，将其解耦合

fs.createReadStream 可读流

可读流用法

为什么要有流呢？因为为了节约内存，如果用`readFile`或`writeFile`之类的API每次读取或者写入，如果文件过大都会很消耗内存，用流的话可以设置缓存区读一点写一点

```
let rs = fs.createReadStream("./a.txt", {
  // 创建可读流一般情况下不用自己传递参数
  flags: "r",
  encoding: null, // 编码就是buffer
  autoClose: true, // 相当于需要调用close方法
  // start:1,
  //end:4,// end 是包后的
  highWaterMark: 3, // 每次读取的数据个数 默认是64 * 1024 字节
});

rs.on("open", function (fd) {
  console.log("open", fd);
});

rs.on("data", function (chunk) {
  console.log(chunk.toString());
  rs.pause(); // 不再触发data事件
});

rs.on("end", function () {
  // 当文件读取完毕后会触发end事件
  console.log("end");
});

rs.on("close", function () {
  console.log("close");
});
```

```
});  
rs.on("error", function (err) {  
  console.log(err, "err");  
});  
setInterval(() => {  
  rs.resume(); // 再次触发data事件  
, 1000);
```

可读流实现

```
const EventEmitter = require("events");  
const fs = require("fs");  
class ReadStream extends EventEmitter {  
  constructor(path, options = {}) {  
    super();  
    // 放在实例上  
    this.path = path;  
    this.flags = options.flags || "r";  
    this.encoding = options.encoding || null;  
    this.autoClose = options.autoClose || true;  
    this.start = options.start || 0;  
    this.end = options.end;  
    this.highWaterMark = options.highWaterMark || 64 * 1024;  
    this.flowing = false; // pause resume  
    this.open(); // 文件打开操作 注意这个方法是异步的  
    // 注意用户监听了data事件 才需要读取  
    this.on("newListener", function (type) {  
      if (type === "data") {  
        this.flowing = true;  
        this.read();  
      }  
    });  
    this.offset = this.start; // 默认start = offset  
  }  
  pipe(ws) {  
    this.on("data", (data) => {  
      let flag = ws.write(data);  
      if (!flag) {  
        this.pause();  
      }  
    });  
    ws.on("drain", () => {  
      this.resume();  
    });  
  }  
  resume() {  
    if (!this.flowing) {  
      this.flowing = true;  
      this.read();  
    }  
  }  
}
```

```
pause() {
  this.flowing = false;
}
read() {
  // once events模块中的绑定一次
  // 希望在open之后才能拿到fd
  if (typeof this.fd !== "number") {
    return this.once("open", () => this.read());
  }
  let howMuchToRead = this.end
    ? Math.min(this.end - this.offset + 1, this.highWaterMark)
    : this.highWaterMark;
  const buffer = Buffer.alloc(howMuchToRead);
  // 读取文件中内容, 每次读取this.highWaterMark个
  fs.read(
    this.fd,
    buffer,
    0,
    howMuchToRead,
    this.offset,
    (err, bytesRead) => {
      if (bytesRead) {
        this.offset += bytesRead;
        this.emit("data", buffer.slice(0, bytesRead));
        if (this.flowing) {
          // 用于看是否递归读取
          this.read();
        }
      } else {
        this.emit("end");
        this.destroy();
      }
    }
  );
}
destroy(err) {
  if (err) {
    this.emit("error", err);
  }
  if (this.autoClose) {
    fs.close(this.fd, () => this.emit("close"));
  }
}
open() {
  fs.open(this.path, this.flags, (err, fd) => {
    if (err) {
      return this.destroy(err);
    }
    this.fd = fd;
    this.emit("open", fd);
  });
}
```

```
module.exports = ReadStream;
```

fs.createWriteStream 可写流

可写流的使用

```
const fs = require("fs");
const path = require("path");
const WriteStream = require("./WriteStream");
const ws = fs.createWriteStream(path.resolve(__dirname, "../a.txt"), {
  highWaterMark: 6,
});

// 10个数 希望使用3个字节内存来处理
let i = 0; // 写入0-9个
function write() {
  let flag = true;
  while (i < 10 && flag) {
    flag = ws.write(i++ + "");
  }
}
ws.on("drain", function () {
  // 只有当我们写入的数据达到了预期，并且数据被清空后才会触发drain事件
  console.log("写完了");
  write();
});

write(); // 执行写入操作
```

可写流的实现

```
class WriteStream extends EventEmitter {
  constructor(path, options) {
    super();
    this.path = path;
    this.flags = options.flags || "w";
    this.encoding = options.encoding || "utf8";
    this.mode = options.mode || 0o666;
    this.autoClose = options.autoClose || true;
    this.start = options.start || 0;
    this.highWaterMark = options.highWaterMark || 16 * 1024;

    this.len = 0; // 用于维持有多少数据没有被写入到文件中的
    this.needDrain = false;
    this.cache = new Queue();
    this.writing = false; // 用于标识是否是第一次写入
    this.offset = this.start; // 偏移量
    this.open();
  }
}
```

```

}
open() {
  fs.open(this.path, this.flags, this.mode, (err, fd) => {
    this.fd = fd;
    this.emit("open", fd);
  });
}
clearBuffer() {
  // 先写入成功后 调用clearBuffer -》 写入缓存的第一个, 第一个完成后, 再继续第二个
  let data = this.cache.poll();
  if (data) {
    this._write(data.chunk, data.encoding, data.cb);
  } else {
    this.writing = false;
    if (this.needDrain) {
      this.emit("drain");
    }
  }
}
// 切片编程
write(chunk, encoding = this.encoding, cb = () => {}) {
  // Writable 类中的
  // 1.将数据全部转化成buffer
  chunk = Buffer.isBuffer(chunk) ? chunk : Buffer.from(chunk);
  this.len += chunk.length;
  let returnValue = this.len < this.highWaterMark;
  // 当数据写入后 需要在手动的将 this.len--;
  this.needDrain = !returnValue;
  let userCb = cb;
  cb = () => {
    userCb();
    this.clearBuffer(); // 清空缓存逻辑
  };
  // 此时我需要 判断你是第一次给我的, 还是不是第一次
  if (!this.writing) {
    // 当前没有正在写入说明是第一次的
    // 需要真正执行写入的操作
    this.writing = true;
    this._write(chunk, encoding, cb);
  } else {
    this.cache.offer({
      chunk,
      encoding,
      cb,
    });
  }
  return returnValue;
}
_write(chunk, encoding, cb) {
  if (typeof this.fd !== "number") {
    return this.once("open", () => this._write(chunk, encoding, cb));
  }
  fs.write(this.fd, chunk, 0, chunk.length, this.offset, (err, written) => {
    this.offset += written; // 维护偏移量
  });
}

```

```
        this.len -= written; // 把缓存的个数减少
        cb(); // 写入成功了
    });
}
}
module.exports = WriteStream;
```

pipe 用法和原理

`rs.pipe(ws)`, 写入的时候根据`ws.write(data)`返回的值做判断是否暂停, 根据`ws.on("drain", fn)`来设置继续写入

```
pipe(ws) {
    this.on("data", (data) => {
        let flag = ws.write(data);
        if (!flag) {
            this.pause();
        }
    });
    ws.on("drain", () => {
        this.resume();
    });
}
```

自定义流

自定义可写流

```
const { Writable, Duplex, Transform, Readable } = require("stream");
class MyWrite extends Writable {
    // writeable write -> _write
    _write(chunk, encoding, cb) {
        console.log(chunk);
        //cb(); // clearBuffer;
    }
}
```

自定义可读流

```
class MyRead extends Readable {
    // writeable write -> _write
    _read() {
        // 内部会不停的调用_read
        console.log(arguments);
        if (this.i === 5) {
            return this.push(null);
        }
    }
}
```

```
    }  
    this.push(this.i++ + ""); // 调用this.emit("data");  
  }  
}
```

自定义双工流

```
class MyDuplex extends Duplex {  
  // 可以满足 读写的功能  
  _read() {}  
  _write() {}  
}
```

自定义转化流

```
class MyTransform extends Transform {  
  _transform(chunk, encoding, cb) {  
    // 参数和可写流一样  
    chunk = chunk.toString().toUpperCase();  
    this.push(chunk); // this.emit('data')  
    cb();  
  }  
}  
  
let transform = new MyTransform();  
  
process.stdin.pipe(transform).pipe(process.stdout);
```

http

nodemon是一个可以监听**node**服务器文件内容变化后重启的包, `nodemon server.js`

基本使用

```
// http是node内置模块 可以直接来使用  
  
const http = require("http");  
const url = require("url");  
// request (获取请求的信息) -> response (给浏览器写数据使用response)  
// 流: http 内部是基于tcp的(net模块, socket双向通信) http1.1 他是一个半双工的  
// 内部基于socket 将其分割出了 request, response 底层实现还是要基于socket  
  
// 底层基于发布订阅模式  
// 底层用socket来通信, http会增加一些header信息, 请求来了之后需要在socket中读取数据, 并  
解析成请求头
```

```

// 学http就是学header， 还有解析请求， 响应数据

// url 由多部分组成
// http://username:password@www.zhufeng.com:80/a?a=1#aaa
const server = http.createServer((req, res) => {
  // 先获取请求行 请求方法 请求路径 版本号
  console.log("请求行-----start-----");
  console.log(req.method); // 请求方法是大写的
  console.log(req.url); // 请求路径是从 路径开始 到hash的前面，默认没写路径就是/，/代表的是服务端根路径
  const { pathname, query } = url.parse(req.url, true);
  console.log(pathname, query); // query就是get请求的参数
  console.log("请求行-----end-----");
  console.log("请求头-----start-----");
  console.log(req.headers); // 获取浏览器的请求头，node中所有的请求头都是小写的
  console.log("请求头-----end-----");

  // post请求和put请求有请求体 req是可读流
  // 大文件上传需要分片，或者用客户端上传
  let chunk = [];
  console.log("读取请求体-----start-----");
  req.on("data", function (data) {
    // 可读流读取的数据都是buffer类型
    chunk.push(data); // 因为服务端接受到的数据可能是分段传输的，我们需要自己将传输的数据拼接起来
  });
  req.on("end", function () {
    // 将浏览器发送的数据全部读取完毕
    console.log(Buffer.concat(chunk).toString());
    console.log("读取请求体-----end-----");
  });

  // 响应状态码，可以字节设定一般情况不设定
  // res.statusCode = 500; // 更改浏览器响应的状态
  // res.statusMessage = 'my define';

  // 响应头 res就是一个可写流
  res.setHeader("MyHeader", 1);

  // 响应体 (如果是路径 那就把响应内容返回给页面，如果是ajax 则放到ajax中的向应力)
  res.write("hello"); // socket.write
  res.write("world");
  res.end("ok"); // 写完了 end => write + close
});
// server.on('request',function (req,res) {
//   console.log('client come on')
// })
server.listen(4000, function () {
  // 监听成功后的回调
  console.log("server start 4000");
});
// 每次更新代码需要重新启动服务，才能运行最新代码
// nodemon 开发时可以使用nodemon 监控文件变化 重新启动
// npm install nodemon -g

```


缓存

强制缓存

服务器对客户端说，在一定时间范围内别找我

使用`Cache-Control` 或`Expires`头，两个都设置浏览器会用`cache-control`

`Expires`受限于本地时间，如果修改了本地时间，可能会造成缓存失效

不会向服务器发请求，默认状态码为`200`

`Cache-Control`常见值

- `no-cache` 每次都像服务器发送请求，**会**存到浏览器的缓存里
- `no-store` 每次都像服务器发送请求，但是**不会**缓存到浏览器里

```
res.setHeader("Cache-Control", "max-age=10"); // 10 秒 设置缓存的时长 相对时间
res.setHeader("Expires", new Date(Date.now() + 10 * 1000).toGMTString()); // 绝对时间
```

协商缓存

使用`Last-Modified`

缺点

- 改变了文件的最后修改时间，但文件内容没有变，就不会走缓存
- 只能以秒计时，如果修改得很快，还是会走缓存，不能返回正确的资源

```
const ctime = statObj.ctime.toGMTString();
if (req.headers["if-modified-since"] === ctime) {
  res.statusCode = 304; // 去浏览器缓存中找吧
  res.end(); // 表示此时服务器没有响应结果
} else {
  res.setHeader("Last-Modified", ctime);
  res.setHeader("Content-Type", mime.getType(filePath) + ";charset=utf-8");
  fs.createReadStream(filePath).pipe(res);
}
```

使用`ETag`

缺点：计算`ETag`可能会比较消耗性能

```
let content = fs.readFileSync(filePath);
let etag = crypto.createHash("md5").update(content).digest("base64");
if (req.headers["if-none-match"] === etag) {
  res.statusCode = 304;
  res.end();
} else {
  res.setHeader("Etag", etag);
  res.setHeader("Content-Type", mime.getType(filePath) + ";charset=utf-8");
  fs.createReadStream(filePath).pipe(res);
}
```

缓存策略

对于经常变动的资源，可以设置`Cache-Control = no-cache`，然后配合`ETag`或者`Last-Modified`

对于代码文件，可以给一个哈希文件名，然后设置`Cache-Control`一个较长的时间，这样只有代码文件名更改了以后才会向服务器重新发起请求

都设置的情况：

但是好像没必要，因为还是会隔一段时间会请求服务器，应该让不同的文件采用不同的策略？

```
function cacheFile(req, res, requestFile, statObj) {
  // 先设置强制缓存，在执行强制缓存的时候 默认是不会执行对比的缓存的
  res.setHeader("Cache-Control", "max-age=10");
  res.setHeader("Expires", new Date(Date.now() + 10 * 1000).toGMTString());

  const lastModified = statObj.ctime.toGMTString();
  const etag = crypto
    .createHash("md5")
    .update(readFileSync(requestFile))
    .digest("base64");

  res.setHeader("Last-Modified", lastModified);
  res.setHeader("Etag", etag);

  let ifModifiedSince = req.headers["if-modified-since"];
  let ifNoneMatch = req.headers["if-none-match"];
  // 如果文件 修改时间都不一样了，那就直接走新的就好了
  if (lastModified !== ifModifiedSince) {
    // 有可能时间一样但是 内容不一样
    return false;
  }
  if (ifNoneMatch !== etag) {
    // 一般情况 指纹的生成不会全量根据文件生成
    return false;
  }
  return true;
}
```

gzip

gzip 不适合重复率低的内容，gzip 核心就是相同替换的方案

不适合视频等重复率不高的，html 文件这种比较适合

node 默认不支持 br 转换

```
function sendFile(req, res, requestFile, statObj) {
  let createGzip;
  if ((createGzip = gzipFile(req, res, requestFile, statObj))) {
    // 看一下支不支持压缩 如果支持 就返回一个压缩转换流
    return createReadStream(requestFile).pipe(createGzip).pipe(res);
  }

  createReadStream(requestFile).pipe(res);
}

function gzipFile(req, res, requestFile, statObj) {
  // 浏览器会给我发一个 accept-encoding 的字段 看一下浏览器支持什么压缩
  let encodings = req.headers["accept-encoding"];
  if (encodings) {
    // 浏览器支持压缩
    if (encodings.includes("gzip")) {
      res.setHeader("Content-Encoding", "gzip"); // 浏览器要知道服务器的压缩类型
      return zlib.createGzip();
    } else if (encodings.includes("deflate")) {
      res.setHeader("Content-Encoding", "deflate");
      return zlib.createDeflate();
    }
  }
  return false; // 浏览器不支持压缩
}
```

实现一个 http-server

src/util.js

```
const os = require("os");

function getIp() {
  let interfaces = os.networkInterfaces();
  interfaces = Object.values(interfaces).reduce((memo, current) => {
    return memo.concat(current);
  }, []);
  let ip = interfaces.find((item) => {
    return item.family === "IPv4" && item.cidr.startsWith("192");
  });
  return ip;
}
```

```
this.getIp = getIp;
```

src/index.js

```
const http = require("http");
const url = require("url");
const path = require("path");
const util = require("./util");
const fs = require("fs").promises;
const chalk = require("chalk"); // 第三方模块 颜色模块;
const mime = require("mime");
const crypto = require("crypto");
const { createReadStream, readFileSync } = require("fs");
const ejs = require("ejs");
const zlib = require("zlib");
const template = readFileSync(path.resolve(__dirname, "template.html"), "utf8");
class Server {
  constructor(serverOptions) {
    this.port = serverOptions.port;
    this.directory = serverOptions.directory;
    this.cache = serverOptions.cache;
    this.gzip = serverOptions.gzip;
    this.handleRequest = this.handleRequest.bind(this);
    this.template = template;
  }
  async handleRequest(req, res) {
    // 1. 获取请求路径 以当前目录为基准查找文件, 如果文件存在不是文件夹则直接返回
    let { pathname } = url.parse(req.url); // 获取解析的路径
    pathname = decodeURIComponent(pathname);
    let requestFile = path.join(this.directory, pathname);
    try {
      let statObj = await fs.stat(requestFile);
      if (statObj.isDirectory()) {
        const dirs = await fs.readdir(requestFile);
        // 根据数据和模板 渲染内容
        let fileContent = await ejs.render(this.template, {
          dirs: dirs.map((dir) => ({
            name: dir,
            url: path.join(pathname, dir),
          })),
        });
        res.setHeader("Content-Type", "text/html; charset=utf-8");
        res.end(fileContent);
      } else {
        this.sendFile(req, res, requestFile, statObj);
      }
    } catch (e) {
      console.log(e);
      this.sendError(req, res, e);
    }
  }
}
```

```
}
cacheFile(req, res, requestFile, statObj) {
  // 先设置强制缓存，在执行强制缓存的时候 默认是不会执行对比的缓存的
  res.setHeader("Cache-Control", "max-age=10");
  res.setHeader("Expires", new Date(Date.now() + 10 * 1000).toGMTString());

  const lastModified = statObj.ctime.toGMTString();
  const etag = crypto
    .createHash("md5")
    .update(readFileSync(requestFile))
    .digest("base64");

  res.setHeader("Last-Modified", lastModified);
  res.setHeader("Etag", etag);

  let ifModifiedSince = req.headers["if-modified-since"];
  let ifNoneMatch = req.headers["if-none-match"];
  // 如果文件 修改时间都不一样了，那就直接走新的就好了
  if (lastModified !== ifModifiedSince) {
    // 有可能时间一样但是 内容不一样
    return false;
  }
  if (ifNoneMatch !== etag) {
    // 一般情况 指纹的生成不会全量根据文件生成
    return false;
  }
  return true;
}
gzipFile(req, res, requestFile, statObj) {
  // 浏览器会给我发一个 accept-encoding的字段 我要看一下浏览器支持什么压缩
  let encodings = req.headers["accept-encoding"];
  if (encodings) {
    // 浏览器支持压缩
    if (encodings.includes("gzip")) {
      res.setHeader("Content-Encoding", "gzip"); // 浏览器要知道服务器的压缩类型
      return zlib.createGzip();
    } else if (encodings.includes("deflate")) {
      res.setHeader("Content-Encoding", "deflate");
      return zlib.createDeflate();
    }
  }
  return false; // 浏览器不支持压缩
}
sendFile(req, res, requestFile, statObj) {
  // 判断有没有缓存 如果有缓存，就使用对比缓存处理

  if (this.cacheFile(req, res, requestFile, statObj)) {
    res.statusCode = 304;
    return res.end();
  }
  res.setHeader("Content-Type", mime.getType(requestFile) + ";charset=utf-8");
  let createGzip;
  if ((createGzip = this.gzipFile(req, res, requestFile, statObj))) {
    // 看一下支不支持压缩 如果支持 就返回一个压缩流
  }
}
```

```

    return createReadStream(requestFile).pipe(createGzip).pipe(res);
  }

  // 我们返回文件 需要给浏览器提供内容类型 和 内容的编码格式

  // 需要将文件读取出来并且返回

  // 如果你不结束，浏览器相当于没有接受完毕，
  createReadStream(requestFile).pipe(res); // 流. ws.write() ws.write() ws.end()
}
sendError(req, res, e) {
  res.statusCode = 404;
  res.end(`Not Found`);
}
start() {
  // 启动服务监听错误信息，如果端口占用 累加1
  const server = http.createServer(this.handleRequest);
  server.listen(this.port, () => {
    // 订阅方法 监听成功后会触发
    console.log(chalk.yellow("Starting up http-server, serving ./"));
    console.log(chalk.yellow("Available on:"));
    console.log(
      `http://${util.getIp().address} :${chalk.green(this.port)}`
    );
    console.log(`http://127.0.0.1:${chalk.green(this.port)}`);
  });
  server.on("error", (err) => {
    if (err.errno === "EADDRINUSE") {
      server.listen(++this.port);
    }
  });
}
}

module.exports = Server;

// 范围请求
// 文件格式处理 图片 字符串 json...
// 跨域的header
// 多语言实现，防盗链
// 反向代理
// 重定向
// cookie session (header)

// header ~~~~

```

src/template.html

```

<!DOCTYPE html>
<html lang="en">
<head>

```

```
<meta charset="UTF-8" />
<meta http-equiv="X-UA-Compatible" content="IE=edge" />
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
<title>Document</title>
</head>
<body>
  <%dirs.forEach(dir=>{%>
    <li><a href="<%=dir.url%>"><%=dir.name%></a></li>
  <%})%>
</body>
</html>
```

bin/config.js

```
const options = {
  port: {
    option: "-p, --port <n>", // 根据commander 的 option('')
    default: 8080,
    usage: "fs --port 3000",
    description: "set fs port",
  },
  gzip: {
    option: "-g, --gzip <n>",
    default: 1,
    usage: "fs --gzip 0", // 禁用压缩
    description: "set fs gzip",
  },
  cache: {
    option: "-c, --cache <n>",
    default: 1,
    usage: "fs --cache 0", // 禁用缓存
    description: "set fs gzip",
  },
  directory: {
    option: "-d, --directory <d>",
    default: process.cwd(),
    usage: "fs --directory d:", // 禁用缓存
    description: "set fs directory",
  },
};

module.exports = options;
```

bin/www.js

```
#!/usr/bin/env node

// 这里需要有一个帮助文档 命令行的帮助文档
const program = require("commander");
```

```

const options = require("./config");
program.name("fs");
program.usage("[options]");

// 解析 当前运行进程传递的参数

const examples = new Set();
const defaultMapping = {};
Object.entries(options).forEach(([key, value]) => {
  examples.add(value.usage);
  defaultMapping[key] = value.default;
  program.option(value.option, value.description);
});

program.on("--help", function () {
  console.log("\nExamples:");
  examples.forEach((item) => {
    console.log(`  ${item}`);
  });
});

program.parse(process.argv);
let userArgs = program.opts();
// 合并最终的参数 需要启动一个服务
let serverOptions = Object.assign(defaultMapping, userArgs);

// 启动一个服务
const Server = require("../src/index");
let server = new Server(serverOptions);
server.start();

```

package.json

```

{
  "name": "file-server",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "bin": {
    "fs": "./bin/www.js",
    "file-server": "./bin/www.js"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "commander": "^7.2.0",
    "ejs": "^3.1.6",

```



```
"mime": "^2.5.2"
}
}
```

设置允许跨域响应头 cors

```
if (req.headers.origin) {
  // 表示谁来访问服务器都可以 (cookie跨域不能使用 *)
  res.setHeader("Access-Control-Allow-Origin", req.headers.origin);

  // 服务器告诉浏览器 我能识别你自定的header
  res.setHeader("Access-Control-Allow-Headers", "Content-Type,token");

  // 每隔10s试探一次
  res.setHeader("Access-Control-Max-Age", "10");
  // 我服务可以接受哪些方法的请求
  res.setHeader("Access-Control-Allow-Methods", "GET,PUT,DELETE,POST,OPTIONS");

  // 表示运行携带cookie了
  res.setHeader("Access-Control-Allow-Credentials", true);

  if (req.method === "OPTIONS") {
    return res.end(); // 表示是一个试探型请求 不要处理就ok
  }
}
```

content-type 的处理 /文件上传的处理

```
let buf = Buffer.concat(buffer); // 前端传递的数据
// http1.0中的特点 为了能识别不同类型 需要通过请求头来处理
if (req.headers["content-type"] === "application/json") {
  let obj = JSON.parse(buf.toString()); // 回显json
  res.setHeader("Content-Type", "application/json");
  res.end(JSON.stringify(obj));
} else if (req.headers["content-type"] === "text/plain") {
  res.setHeader("Content-Type", "text/plain");
  res.end(buf.toString());
} else if (
  req.headers["content-type"] === "application/x-www-form-urlencoded"
) {
  // {username:"123"} username=123&password=456 a=1; b=2
  let r = querystring.parse(buf.toString(), "&", "="); // 可以将查询字符串 转化成对象
  res.end(JSON.stringify(r));
} else if (req.headers["content-type"].includes("multipart/form-data")) {
  let boundary = "--" + req.headers["content-type"].split("=")[1];
  let lines = buf.split(boundary).slice(1, -1);
  const r = {};
  lines.forEach((line) => {
```

```
// 把header 和 内容进行拆分
let [head, body] = line.split(`\r\n\r\n`);
head = head.toString();
let name = head.match(/name="(.*?)"/)[1];
if (head.includes("filename")) {
  // 文件上传 将文件内容上传到服务器的上传文件夹中
  let buffer = line.slice(head.length + 4, -2);
  let filename = uuid.v4();
  // 你最终创建完名字之后 还会同步到数据库里 , 下次查找数据库, 再找到对应的文件名
  fs.writeFileSync(path.join(uploadPath, filename), buffer);
  r[name] = {
    filename,
    size: buffer.length,
    // 文件大小 fs.stat
  };
  // 文件名一般都是随机的
} else {
  // key:value
  // 直接把信息放到一个对象中即可
  r[name] = body.toString().slice(0, -2);
}
});
res.end(JSON.stringify(r));
}
```

下载头

```
// 下载头
res.setHeader("Content-Type", "application/octet-stream");
// 设置不识别的header 也会变成下载文件, 设置对了才行
res.setHeader("Content-Disposition", "attachment;filename=FileName.txt");
```

crypto 加密模块

使用md5对数据进行摘要

```
// crypto是我们node中提供好的用于加密的包 各种摘要算法和加密算法

// md5 算法 hash算法 摘要算法 (md5 无法反解)

// 1.md5
// 不可逆
// 相同的内容摘要出的结果相同
// 摘要的内容不同 结果完全不同 (雪崩效应)
// 摘要不同的内容 长度是相同的

// 撞库不叫解密, 为了安全 你可以把一个md5值多次加密 md5(md5(md5(xxx)))
```

```
const crypto = require("crypto");
//                                摘要的内容          摘要的格式
let r1 = crypto.createHash("md5").update("abcd").digest("base64");
//                                分开摘要   如果内部使用了流，可以读一点摘要一点
let r2 = crypto
  .createHash("md5")
  .update("a")
  .update("b")
  .update("cd")
  .digest("base64");
console.log(r1, r2); //一样
```

加盐算法

应用场景: jwt

```
// 2.加盐算法 (盐值, 秘钥)
// 可以把秘钥生成一个1k大小的 随机的字符, 在用作秘钥  jwt的原理
let r3 = crypto.createHmac("sha256", "zf1").update("abcd").digest("base64");
console.log(r3);

// 做一个简述 md5的特点
```

KOA

express 和 koa 的对比

- express 基于 es5, koa 基于 es6
- express 比较全内置了很多功能, koa 内部核心是非常小巧的 (我们可以通过扩展的插件来进行扩展)
- express 和 koa 都是可以自己去使用来实现 mvc 功能的, 没有约束
- express 他处理异步的方式都是回调函数 koa 处理异步的方式都是 async + await
- 学会一个另一个就容易很多。用起来很方便

koa 的基本使用

```
const Koa = require("../koa/lib/application");

// 使用koa就创建一个应用实例
const app = new Koa();

app.use((ctx) => {
  // ctx中扩展了 请求和响应的方法
  // 先理解成 res.end();
  ctx.body = "hello";

  // ctx中 有5个比较重要的属性
```

```
// app 当前应用实例 可以在app上扩展公共方法
// req.res 原生node中的req和res
// request.response 是koa里面自己封装的

// koa中对request和response进行一层抽象 叫request和response。在开发的时候 我们尽量避免操作原生的req和res
//console.log(ctx.req.url);
//console.log(ctx.request.req.url);

//console.log(ctx.request.query); // ctx.request.__proto__.__proto__
console.log(ctx.query); // 希望不通过 vm = new Vue{data:{}} vm.xxx => vm.data.xxx
});

app.listen(3000, function () {
  console.log(`server start 3000`);
}); // 监听一个端口号, 同我们的node中http的listen方法
```

koa 中的隔离问题

打开koa源码可以发现

```
const http = require("http");
const context = require("./context");
const request = require("./request");
const response = require("./response");
const Stream = require("stream");
const EventEmitter = require("events");

class Application extends EventEmitter {
  constructor() {
    super(); // EventEmitter.call(this);
    this.context = Object.create(context); // 实现每次创建一个应用都有自己的全新上下文
    this.request = Object.create(request);
    this.response = Object.create(response);
  }
  createContext(req, res) {
    let ctx = Object.create(this.context); // 这个目的是为了每次请求的时候 都拥有自己的上下文, 而且自己的上下文是可以获取公共上下文声明的变量、属性
    let request = Object.create(this.request);
    let response = Object.create(this.response);

    ctx.request = request; // 上下文中包含着request
    ctx.req = ctx.request.req = req; // 默认上下文中包含着 原生的req

    ctx.response = response;
    ctx.res = ctx.response.res = res; // 这个的目的和request的含义是一致的, 就是可以在我们的response对象中 通过this.res 拿到原生res

    return ctx;
  }
}
```

```
//...  
}
```

也就是说`ctx.__proto__.__proto__ = context`，这样做是为了起到隔离上下文的作用

koa 中的属性代理

在koa中`ctx.body = ctx.response.body = value`，具体它是通过定义`getter`和`setter`实现的

```
function defineGetter(target, key) {  
  // proxy , defineProperty  
  context.__defineGetter__(key, function () {  
    //这里的this是ctx  
    //ctx.__proto__.__proto__ = context  
    return this[target][key]; // ctx.request.query / ctx.request.path;  
  });  
}  
function defineSetter(target, key) {  
  // proxy , defineProperty  
  context.__defineSetter__(key, function (value) {  
    this[target][key] = value; // ctx.body = ctx.response.body = value  
  });  
}  
defineGetter("response", "body");  
defineSetter("response", "body");
```

koa 中的响应体处理

```
let _body = ctx.body;  
if (typeof _body === "string" || Buffer.isBuffer(_body)) {  
  return res.end(_body);  
} else if (typeof _body === "number") {  
  return res.end(_body + "");  
} else if (_body instanceof Stream) {  
  // 可以设置成下载头  
  //res.setHeader('Content-Type', 'application/octet-stream');  
  // 设置不识别的header 也会变成下载文件，设置对了才行  
  // res.setHeader('Content-Disposition', 'attachment;filename=FileName.txt');  
  return _body.pipe(res);  
} else if (typeof _body == "object") {  
  return res.end(JSON.stringify(_body));  
}
```

koa 中的中间件和 compose 的实现

在 `next` 前面必须加 `await` 或者 `return`，因为后面的中间件不知道是不是异步的，加`await`是为了保证中间件完整地执行，而且所有的异步方法都必须变成`promise`，因为`await`只能等待`promise`完成

koa中的compose函数

```
compose(ctx) {  
  // 组合是要将 数组里的函数 一个个执行  
  let index = -1;  
  const dispatch = (i) => {  
  
    if (i <= index) return Promise.reject(new Error('next() called  
multiple times'))  
    index = i;  
  
    // 如果没有中间件 直接成功即可  
    if (this.middlewares.length === i) return Promise.resolve();  
  
    return Promise.resolve(this.middlewares[i](ctx, () => dispatch(i +  
1)))  
  }  
  return dispatch(0);  
}
```

自定义中间件

所有中间件都是函数执行后返回一个中间件函数，目的是方便扩展

koa-bodyparser

```
function bodyParser() {  
  return async (ctx, next) => {  
    // 自己定义一个请求体  
    ctx.request.body = await new Promise((resolve, reject) => {  
      let arr = [];  
      ctx.req.on("data", function (chunk) {  
        arr.push(chunk);  
      });  
      ctx.req.on("end", function () {  
        // get 请求没有请求体 会直接触发end事件  
        resolve(Buffer.concat(arr));  
      });  
    });  
    await next();  
  };  
}
```

koa-static

```
function static(dirname) {  
  return async (ctx, next) => {
```

```
let filePath = path.join(dirname, ctx.path);
// 如果文件路径 不是文件的话 就不能处理了, 需要调用下一个中间件, 如果自己能处理。就不需要向下执行了
try {
  const statObj = await fs.stat(filePath);
  if (statObj.isDirectory()) {
    filePath = path.join(filePath, "index.html");
  }
  ctx.set("Content-Type", mime.getType(filePath) + ";charset=utf-8");
  ctx.body = await fs.readFile(filePath);
} catch {
  await next(); // 自己处理不了 向下执行
}
};
}
```

路由

```
const Koa = require("koa");
const Router = require("koa-router");
const app = new Koa();
const router = new Router();
app.use(router.routes());
router.get("/user/add", async (ctx, next) => {
  //这里的next是匹配的下一个路由
  ctx.body = "user/add";
});
```

前端存储方式 cookie session localStorage sessionStorage indexDb token

cookie

每次请求的时候自动携带, 大小限制4k, 不建议存放敏感信息, 如果被劫持到 (中间人攻击)

- name
- value
- domain: 设置那个域下可以使用, 可以跨父域和子域, 默认是当前域
- expires/max-age: cookie存活时间
- path: 当什么路径时可以访问cookie, 默认是/都能被访问到
- httpOnly: 表示浏览器无法通过代码来获取和更改, 但是可以通过浏览器控制台获取或者更改 (控制台->存储)

- `secure`: 只允许https下携带

koa中的设置和获取cookie:

```
ctx.cookies.get("name");
ctx.cookies.set("age", "18", { httpOnly: true });
```

koa 中的 cookie 加盐

koa会对`name=value`这个内容进行签名（sha1算法），生成一个新的`${name}.sig=签名后的内容`的cookie，下次在服务器中获取cookie时会和另一个生成的cookie进行验证，如果验证不对，会抛弃原cookie

但是用户可以将其他浏览器中的`${cookie}=...`和`${cookie}.sig=...`放在自己浏览器中，所以还是不是很安全

用法:

```
app.keys = ["secret"];
//...
ctx.cookies.get("visit", { signed: true });
ctx.cookies.set("visit", visitCount, { signed: true });
```

加盐算法:

```
// base64Url 需要特殊处理 + = /
const sign = (value) => {
  return crypto
    .createHmac("sha1", app.keys.join(""))
    .update(value)
    .digest("base64")
    .replace(/\+/g, "-")
    .replace(/\=/g, "")
    .replace(/\//g, "_");
};
```

session

在服务器开辟一个空间来存储用户对应的信息（是根据对应cookie获得信息的，因为放在服务器里，可以存储敏感信息），`session`基于cookie的比cookie安全

可以将session存到数据库中实现持久化

基于cookie（加盐）获取session的例子:


```
const session = {}; // 用来存储用户和信息的映射关系,浏览器拿不到
const cardName = "connect.sid"; // 卡的名字
const uuid = require("uuid");
router.get("/wash", async (ctx, next) => {
  // 洗澡的例子
  // 第一次来洗澡 需要办一张卡 冲上钱,把卡号告诉你
  // 下次你自动带上卡 就ok
  let id = ctx.cookies.get(cardName, { signed: true });
  if (id && session[id]) {
    session[id].mny -= 20;
    ctx.body = `mny ` + session[id].mny;
  } else {
    let cardId = uuid.v4();
    session[cardId] = { mny: 500 };
    ctx.cookies.set(cardName, cardId, { httpOnly: true, signed: true });
    ctx.body = `mny 500`;
  }
});
```

localStorage

特点就是关掉浏览器后数据依然存在, 有大小限制 5m, 不能跨域

sessionStorage

如果页面不关闭就不会销毁 (如访问时存储滚动条地址), 大小限制 5m, 不能跨域

token (JWT)

jwt (jsonwebtoken) 不需要服务器存储, 没有跨域限制, 是目前最流行的跨域身份验证解决方案

session的问题: 比如如果不使用数据库的话就不能在多个网站中共享登录状态, 如果存在数据库中丢失的话会有认证问题, jwt就解决了这样的服务器存储登录数据的问题

JWT 包含了使用. 分隔的三部分:

- Header 头部

```
{ "alg": "HS256", "typ": "JWT" }
// algorithm => HMAC SHA256
// type => JWT
```

- Payload 负载、载荷: JWT 规定了 7 个官方字段

```
iss (issuer): 签发人
exp (expiration time): 过期时间
sub (subject): 主题
aud (audience): 受众
```

nbf (Not Before): 生效时间
iat (Issued At): 签发时间
jti (JWT ID): 编号

- **Signature** 签名: 对前两部分的签名, 防止数据篡改

```
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret);
```

使用方式:

- 通过**requestHeader**传输: **Authorization: Bearer <token>**
- 通过**url**传输: **http://www.xxx.com/pwa?token=xxxxx**
- 如果是**post**请求也可以放在请求体中

在**koa**中使用:

```
let Koa = require("koa");
let Router = require("koa-router");
let bodyparser = require("koa-bodyparser");
let jwt = require("jwt-simple");
let router = new Router();
let app = new Koa();
app.use(bodyparser());
let secret = "zfpX";
// 验证是否登陆
router.post("/login", async (ctx) => {
  let { username, password } = ctx.request.body;
  if (username === "admin" && password === "admin") {
    let token = jwt.encode(username, secret);
    ctx.body = {
      code: 200,
      username,
      token,
    };
  }
});
// 验证是否有权限
router.get("/validate", async (ctx) => {
  let Authorization = ctx.get("authorization");
  let [, token] = Authorization.split(" ");
  if (token) {
    try {
      let r = jwt.decode(token, secret);
      ctx.body = {
        code: 200,
        username: r,
        token,
      };
    }
  }
});
```

```
    } catch (e) {
      ctx.body = {
        code: 401,
        data: "没有登陆",
      };
    }
  } else {
    ctx.body = {
      code: 401,
      data: "没有登陆",
    };
  }
});
app.use(router.routes());
app.listen(4000);
```

mongo

安装

可视化操作工具 robo3T

mongo 特点

- 非关系型
- 值就是传统对象类型
- 性能高，不需要解析SQL语句，数据之间没有耦合，方便扩展
- 简单，前端操作对象的感觉

比如只是把数据存起来，不考虑查询和复杂关联的话，可以考虑MongoDB

mongo有可能会丢失数据

mongo 基本语句

- `show dbs` 显示所有的数据库（默认免密的）
- `use admin` (如果不存在也可以使用)
- `show collections` (mysql table) 显示所有的集合
- `db.system.users.find`(查询条件)
- `db.system.users.deleteOne`(查询条件)
- `db.createUser()` 创建用户的
- `db.student.insert({name:"",age:18})`
- `use user`
- `db.auth('账号','密码')`

基本操作：可以在admin中创建根用户，再创建其他数据库。在针对某个数据库中，创建用户，赋予当前权限，下次链接这个数据库时可以登录这个账号

mongoose

文档

初始化

```
const mongoose = require("mongoose");
// 1.链接mongodb, 返回链接成功后的对象
let conn = mongoose.createConnection("mongodb://jw:jw@localhost:27017/user", {
  useNewUrlParser: true,
  useUnifiedTopology: true,
}); // 创建连接

conn.on("connected", function () {
  console.log("链接成功了");
});
// 主要的目的是操作数据 增删改查
// Schema 骨架 根据这个骨架来创建内容
// Schema 是用来规范文档的
let StudentSchema = new mongoose.Schema(
  {
    // vue 属性校验
    username: {
      type: String,
      required: true,
    },
    password: String,
    age: Number,
    birthday: {
      type: Date,
      default: Date.now,
    },
  },
  { collection: "Student" }
); // 设置固定的名字
// 通过骨架 来创建模型 => 集合 db.student.insert()
let Student = conn.model("Student", StudentSchema); // student 就是模型

// 模型可以操作数据, 多的字段不会插入, 少的就是空
// mongo 库 (数据库) -> 集合 (表) -> 文档 (内容)
```

增加

```
let arr = [];
for (let i = 0; i < 10; i++) {
  arr.push({ username: "zf" + i, password: "zf" + i, age: i });
}
// 批量的插入 要采用数组的方式可以实现多次的插入 合并成一次操作
let r = await Student.create(arr);
```

查询

```
// 0、1表示出现还是不出现，不能混合使用，_id除外
let r = await Student.findOne({ username: "zf" }, { username: 1, password: 1 });
//找的是_id
let r2 = await Student.findById("5ef5f7095d185f27b88ec62b");
```

修改

```
// a.修改年龄大于6的一个 年龄增加一
// where基本不用，性能差 lt:less than gt:great than lte gte inc 递增
await student.updateOne({ age: { $gt: 6 } }, { $inc: { age: 1 } });

//b. 模糊查询后更新
await student.updateMany(
  { username: /zf/ },
  { $set: { password: "123456" } }
);

//c. 数组操作
// $set设置数组 $push 数组新增 $addToSet 新增（已经有的话不会增加） $pop 删除
...正1往后删除，-1往左删除
await student.updateMany({ username: /zf/ }, { $set: { hobby: ["吃饭"] } });
await student.updateMany({ username: /zf/ }, { $push: { hobby: ["吃饭"] } });
await student.updateMany({ username: /zf/ }, { $addToSet: { hobby: ["吃饭", "游泳"] } });

// d. 或查询
let r = await Student.updateOne({$or:[{username:/a/},{age:31}]},{ $pop:
{hobby:-1}});
// console.log(r)
```

删除

```
Student.deleteMany({});
Student.deleteOne();
```

分页查询

```
// 每页2条 当前是第2页 分页
let limit = 2;
let currentPage = 2;
// find 返回的是一个游标 指针 并不是结果 先查询 再排序 再跳过再限制
let r = await Student.find({})
  .limit(limit)
```

```
.skip((currentPage - 1) * limit)
.sort({ age: -1 }); //年龄倒序
console.log(r);
conn.close();
```

关联操作

MongoDB本身是没有ref，只是基于了mongoose这个工具才有这个功能

```
const mongoose = require("mongoose");

const conn = mongoose.createConnection("mongodb://jw:jw@localhost:27017/user", {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});
let StudentSchema = new mongoose.Schema(
  {
    username: {
      type: String,
      required: true,
    },
    password: String,
    age: Number,
    birthday: {
      type: Date,
      default: Date.now,
    },
    hobby: [],
  },
  { collection: "Student" }
);
let Student = conn.model("Student", StudentSchema);

let HomeworkSchema = new mongoose.Schema(
  {
    title: String,
    content: String,
    student: {
      ref: "Student", //与Student集合关联，相当于MySQL的外键
      type: mongoose.SchemaTypes.ObjectId, // 用户的id号
    },
  },
  { collection: "Homework" }
);

let Homework = conn.model("Homework", HomeworkSchema);

(async () => {
  let user = await Student.create({ username: "zs", password: "ls" });

  let work = await Homework.create({
```

```
    title: "标题",
    content: "内容",
    student: user._id,
  });

let r = await HomeWork.findById("5ef5ffafd12605015ca0d641").populate(
  "student",
  { username: 1 } //只要username
);

//查询后要修改 操作数据有两种方式 模型来操作, 通过文档来自己操作自己
r.title = "标题2";
await r.save(); // 修改操作
// 什么时候用文档 什么时候集合
conn.close();

// mongoose.createIndex()
// mongoose.Aggregate 聚合
})();
```