

NODE.JS

NODE创建一个服务器



```
//载入http模块 将变量赋值给http
const http = require('http');
// .createServer()创建一个服务器
http
  .createServer(function (request, response) {
    // 发送 http 头部
    // 状态码为: 200 ok
    // 文件类型: text/plain
    // 解决中文乱码
    response.setHeader(
      "Content-Type", "text/html; charset=utf-8"
    )
    // 发送响应数据
    response.end('Hello Node.js');
  })
  // 将代码绑定 8888 端口
  .listen(8888);
console.log('Server running at http://127.0.0.1:8888/');
```



```
// 引入express框架
const express = require('express');
// 创建对象
const app = express();
// 收到请求的路径
app.get('/', (req, res) => {
  // 显示的内容
  res.send('你好呀! '');
```

```
    console.log('请求路径: ', req.url);
  });
app.get('/name', (req, res) => {
  console.log('请求路径: ', req.url);

  res.send('我是陈士博!');
});
app.get('/exit', (req, res) => {
  console.log('请求路径: ', req.url);
});
// 监听端口号
app.listen(8000, () => {
  console.log('8000端口已启动');
});
```

NPM的基本操作

④ 安装

安装:

```
npm install [name] -g(全局安装)
```

安装指定版本:

```
npm install [name]@[version]
```

推荐使用淘宝镜像:

安装淘宝镜像:

```
npm install -g cnpm --registry=https://registry.npmmirror.com
```

使用淘宝镜像安装:

```
cnpm install [name]
```

H3 操作

模块列表：

```
npm list -g(全局)
```

搜索模块：

```
npm search [name]
```

卸载模块：

```
npm uninstal [name]
```

更新模块：

```
npm update [name]
```

创建模块：

```
npm init
```

H3 package.json

这是储存 `npm` 所有已安装的软件包的名称和版本的地方。

- `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。

- `browserslist` 用于告知要支持哪些浏览器（及其版本）。

`dependencies --save` 安装时依赖

`devDependencies --save-dev` 开发时依赖

设置作为开发依赖安装的 `npm` 软件包的列表。

它们不同于 `dependencies`，因为它们只需安装在开发机器上，而无需在生产环境中运行代码。

当使用 `npm` 或 `yarn` 安装软件包时：



BASH

```
npm install --save-dev <PACKAGENAME>
yarn add --dev <PACKAGENAME>
```

该软件包会被自动地插入此列表中。

示例：



JSON

```
"devDependencies": {
  "autoprefixer": "^7.1.2",
  "babel-core": "^6.22.1"
}
```

- 如果写入的是 `~0.13.0`，则只更新补丁版本：即 `0.13.1` 可以，但 `0.14.0` 不可以。
- 如果写入的是 `^0.13.0`，则要更新补丁版本和次版本：即 `0.13.1`、`0.14.0`、依此类推。
- 如果写入的是 `0.13.0`，则始终使用确切的版本。

若要发觉软件包的新版本，则运行 `npm outdated`。

以下是一个仓库中一些过时的软件包的列表，该仓库已很长时间没有更新：

```
~/www/flaviocopes.com/themes/ghostwriter
~ /www/flaviocopes.com/themes/ghostwriter master* 1d 31m 26s
> npm outdated
Package           Current  Wanted  Latest  Location
autoprefixer      8.6.5   8.6.5   9.1.0  ghostwriter
css-loader        0.28.4  0.28.11 1.0.0  ghostwriter
cssnano           3.10.0  3.10.0  4.0.5  ghostwriter
extract-text-webpack-plugin  2.1.2   2.1.2   3.0.2  ghostwriter
node-sass          4.5.3   4.9.2   4.9.2  ghostwriter
normalize.css      7.0.0   7.0.0   8.0.0  ghostwriter
optimize-css-assets-webpack-plugin  2.0.0   2.0.0   5.0.0  ghostwriter
postcss-cli        5.0.1   5.0.1   6.0.0  ghostwriter
postcss-discard-comments  2.0.4   2.0.4   4.0.0  ghostwriter
sass-loader        6.0.6   6.0.7   7.1.0  ghostwriter
style-loader       0.18.2  0.18.2  0.21.0  ghostwriter
webpack            3.0.0   3.12.0  4.16.4  ghostwriter

~/www/flaviocopes.com/themes/ghostwriter master*
>
```

这些更新中有些是主版本。运行 `npm update` 不会更新那些版本。主版本永远不会被这种方式更新，因为它们（根据定义）会引入重大的更改，`npm` 希望为你减少麻烦。

若要将所有软件包更新到新的主版本，则全局地安装 `npm-check-updates` 软件包：

```
BASH
npm install -g npm-check-updates
```

然后运行：

```
BASH
ncu -u
```

这会升级 `package.json` 文件的 `dependencies` 和 `devDependencies` 中的所有版本，以便 `npm` 可以安装新的主版本。

现在可以运行更新了：



BASH

```
npm update
```

如果只是下载了项目还没有 `node_modules` 依赖包，并且想先安装新的版本，则运行：



BASH

```
npm install
```

H3 版本控制

因为 `npm` 设置了一些规则，可用于在 `package.json` 文件中选择要将软件包更新到的版本（当运行 `npm update` 时）。

- `^`: 只会执行不更改最左边非零数字的更新。如果写入的是 `^0.13.0`，则当运行 `npm update` 时，可以更新到 `0.13.1`、`0.13.2` 等，但不能更新到 `0.14.0` 或更高版本。如果写入的是 `^1.13.0`，则当运行 `npm update` 时，可以更新到 `1.13.1`、`1.14.0` 等，但不能更新到 `2.0.0` 或更高版本。
- `~`: 如果写入的是 `~0.13.0`，则当运行 `npm update` 时，会更新到补丁版本：即 `0.13.1` 可以，但 `0.14.0` 不可以。
- `>`: 接受高于指定版本的任何版本。
- `≥`: 接受等于或高于指定版本的任何版本。
- `≤`: 接受等于或低于指定版本的任何版本。
- `<`: 接受低于指定版本的任何版本。
- `=`: 接受确切的版本。
- `-`: 接受一定范围的版本。例如：`2.1.0 - 2.6.2`。
- `||`: 组合集合。例如 `< 2.1 || > 2.6`。

模块

H3 文件操作

- `r+` 打开文件用于读写。
- `w+` 打开文件用于读写，将流定位到文件的开头。如果文件不存在则创建文件。
- `a` 打开文件用于写入，将流定位到文件的末尾。如果文件不存在则创建文件。
- `a+` 打开文件用于读写，将流定位到文件的末尾。如果文件不存在则创建文件。



```
var fs = require('fs');
// 同步读取
var data = fs.readFileSync('./name.txt');

console.log(data.toString());
console.log('程序执行结束！');
// 异步读取
fs.readFile('./name.txt', function (err, data) {
  if (err) {
    console.log(err);
  }
  console.log(data.toString());
});
```

文件描述符：



```
const fs = require('fs');

fs.open('./name.txt', 'r', (err, fd) => {
  //fd 是文件描述符。
  console.log(fd);
});
```

输出：



文件属性：



```
const fs = require('fs')
fs.stat('./name.txt', (err, stats) => {
  if (err) {
    console.error(err)
    return
  }
  //可以访问 `stats` 中的文件属性
})
```

输出：



```
Stats {
  dev: 1007774682,
  mode: 33206,
  nlink: 1,
  uid: 0,
  gid: 0,
  rdev: 0,
  blksize: 4096,
  ino: 1688849860292098,
  size: 22,
  blocks: 0,
  atimeMs: 1644588713029.6116,
  mtimeMs: 1644588713029.6116,
  ctimeMs: 1644588938357.6018,
  birthtimeMs: 1644242437116.9336,
  atime: 2022-02-11T14:11:53.030Z,
  mtime: 2022-02-11T14:11:53.030Z,
  ctime: 2022-02-11T14:15:38.358Z,
  birthtime: 2022-02-07T14:00:37.117Z
}
```

文件的信息包含在属性变量中。可以通过属性提取哪些信息？

很多，包括：

- 使用 `stats.isFile()` 和 `stats.isDirectory()` 判断文件是否目录或文件。
- 使用 `stats.isSymbolicLink()` 判断文件是否符号链接。
- 使用 `stats.size` 获取文件的大小（以字节为单位）。

文件路径：

可以使用以下方式将此模块引入到文件中：



JS

```
const path = require('path')
```

现在可以开始使用其方法。

从路径中获取信息：

给定一个路径，可以使用以下方法从其中提取信息：

- `dirname`：获取文件的父文件夹。
- `basename`：获取文件名部分。
- `extname`：获取文件的扩展名。

例如：



```
JSconst notes = '/users/joe/notes.txt'  
path.dirname(notes) // /users/joe  
path.basename(notes) //  
notes.txt  
path.extname(notes) // .txt
```

可以通过为 `basename` 指定第二个参数来获取不带扩展名的文件名：



JS

```
path.basename(notes, path.extname(notes)) //notes
```

使用路径:

可以使用 `path.join()` 连接路径的两个或多个片段:



JS

```
const name = 'joe'  
path.join('/', 'users', name, 'notes.txt') // '/users/joe/notes.txt'
```

可以使用 `path.resolve()` 获得相对路径的绝对路径计算:



JS

```
path.resolve('joe.txt') // '/Users/joe/joe.txt' 如果从主文件夹运行。
```

在此示例中, Node.js 只是简单地将 `/joe.txt` 附加到当前工作目录。如果指定第二个文件夹参数, 则 `resolve` 会使用第一个作为第二个的基础:



JS

```
path.resolve('tmp', 'joe.txt') // '/Users/joe/tmp/joe.txt' 如果从主文件夹运行。
```

如果第一个参数以斜杠开头, 则表示它是绝对路径:



JS

```
path.resolve('/etc', 'joe.txt') // '/etc/joe.txt'
```

`path.normalize()` 是另一个有用的函数, 当包含诸如 `..`、`..` 或双斜杠之类的相对说明符时, 其会尝试计算实际的路径:



JS

```
path.normalize('/users/joe/.. //test.txt') // '/users/test.txt'
```

解析和规范化都不会检查路径是否存在。其只是根据获得的信息来计算路径。

H4 读取文件

读取文件最简单的方法为`fs.readFile()`方法，向其传入文件路径、编码、以及会带上文件数据（以及错误）进行调用的回调函数：



```
const fs = require('fs')
fs.readFile('/Users/joe/test.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err)
    return
  }
  console.log(data)
})
```

另外，也可以使用同步的版本 `fs.readFileSync()`：



```
const fs = require('fs')
try {
  const data = fs.readFileSync('./name.txt', 'utf8')
  console.log(data)
} catch (err) {
  console.error(err)
}
```

H4 写入文件



```
const fs = require('fs');

const content = '一些内容';
//{ flag: 'a' }用于指定打开形式 也可以省略
fs.writeFile('./test.txt', content, { flag: 'a' }, (err) => {
  if (err) {
    console.error(err);
    return;
  }
  //文件写入成功。
});
```

另外，也可以使用同步的版本 `fs.writeFileSync()`：



```
const fs = require('fs')
const content = '一些内容'
try {
  const data =
    fs.writeFileSync('./test.txt', content) //文件写入成功。
} catch (err) {
  console.error(err)
}
```

④ 追加到文件

将内容追加到文件末尾的便捷方法是 `fs.appendFile()`（及其对应的 `fs.appendFileSync()`）：



```
JScst content = '一些内容'
fs.appendFile('file.log', content, err => {
  if (err) {
    console.error(err)
    return
  } //完成!
})
```

④ 文件夹操作

检查文件夹是否存在：

`fs.access()` 检查文件夹是否存在以及node是否有权限访问

创建新的文件夹：

使用 `fs.mkdir()` 或 `fs.mkdirSync()` 创建新的文件夹。

```
● ○ ●

const fs = require('fs')
const folderName = '/Users/joe/test'
try {
  if (!fs.existsSync(folderName)) {
    fs.mkdirSync(folderName)
  }
} catch (err) {
  console.error(err)
}
```

读取目录内容：

使用 `fs.readdir()` 或 `fs.readdirSync()` 可以读取目录的内容。

```
● ○ ●

const fs = require('fs')
const path = require('path')
const folderPath = '/Users/joe'
fs.readdirSync(folderPath)
```

上面这段代码将会返回所有子文件的相对路径。

```
● ○ ●

fs.readdirSync(folderPath).map(fileName => {
  return path.join(folderPath, fileName)
})
```

上边这段代码可以获取完整的路径。



```
const isFile = fileName => {
    return fs.lstatSync(fileName).isFile()
}
fs.readdirSync(folderPath).map(fileName => {
    return path.join(folderPath, fileName)
})
.filter(isFile)
```

重命名文件夹：

使用 `fs.rename()` 或 `fs.renameSync()` 可以重命名文件夹。第一个参数是当前的路径，第二个参数是新的路径：



```
const fs = require('fs')
fs.rename('/Users/joe'
        , '/Users/roger'
        , err => {
    if (err) {
        console.error(err)
        return
    } //完成
})
```

`fs.renameSync()` 是同步的版本：



```
const fs = require('fs')
try {
    fs.renameSync('/Users/joe', '/Users/roger')
} catch (err) {
    console.error(err)
}
```

删除文件夹：

使用 `fs.rmdir()` 或 `fs.rmdirSync()` 可以删除文件夹。

删除包含内容的文件夹可能会更复杂。

在这种情况下，最好安装 `fs-extra` 模块，该模块非常受欢迎且维护良好。它是 `fs` 模块的直接替代品，在其之上提供了更多的功能。

在此示例中，需要的是 `remove()` 方法。

使用以下命令安装：



```
npm install fs-extra
```

并像这样使用它：



```
const fs = require('fs-extra')
const folder = '/Users/joe'
fs.remove(folder, err => { console.error(err)})
```

也可以与 promise 一起使用：



```
fs.remove(folder)
  .then(() => {
    //完成
  })
  .catch(err => {
    console.error(err)
  })
```

或使用 `async/await`：



```
async function removeFolder(folder) {  
  try {  
    await fs.remove(folder)      //完成  
  } catch (err) {  
    console.error(err)  
  }  
}  
  
const folder = '/Users/joe'  
removeFolder(folder)
```

④ 文件全部方法

- `fs.access()`: 检查文件是否存在，以及 Node.js 是否有权限访问。
- `fs.appendFile()`: 追加数据到文件。如果文件不存在，则创建文件。
- `fs.chmod()`: 更改文件（通过传入的文件名指定）的权限。相关方法：
`fs.lchmod()`、`fs.fchmod()`。
- `fs.chown()`: 更改文件（通过传入的文件名指定）的所有者和群组。相关方法：
`fs.fchown()`、`fs.lchown()`。
- `fs.close()`: 关闭文件描述符。
- `fs.copyFile()`: 拷贝文件。
- `fs.createReadStream()`: 创建可读的文件流。
- `fs.createWriteStream()`: 创建可写的文件流。
- `fs.link()`: 新建指向文件的硬链接。
- `fs.mkdir()`: 新建文件夹。
- `fs.mkdtemp()`: 创建临时目录。
- `fs.open()`: 设置文件模式。
- `fs.readdir()`: 读取目录的内容。
- `fs.readFile()`: 读取文件的内容。相关方法：`fs.read()`。
- `fs.readlink()`: 读取符号链接的值。
- `fs.realpath()`: 将相对的文件路径指针（`.`、`..`）解析为完整的路径。
- `fs.rename()`: 重命名文件或文件夹。
- `fs.rmdir()`: 删除文件夹。
- `fs.stat()`: 返回文件（通过传入的文件名指定）的状态。相关方法：`fs.fstat()`、`fs.lstat()`。

- `fs.symlink()`: 新建文件的符号链接。
- `fs.truncate()`: 将传递的文件名标识的文件截断为指定的长度。相关方法：
`fs.ftruncate()`。
- `fs.unlink()`: 删除文件或符号链接。
- `fs.unwatchFile()`: 停止监视文件上的更改。
- `fs.utimes()`: 更改文件（通过传入的文件名指定）的时间戳。相关方法：
`fs.futimes()`。
- `fs.watchFile()`: 开始监视文件上的更改。相关方法：
`fs.watch()`。
- `fs.writeFile()`: 将数据写入文件。相关方法：
`fs.write()`。

关于 `fs` 模块的特殊之处是，所有的方法默认情况下都是异步的，但是通过在前面加上 `Sync` 也可以同步地工作。

例如：

- `fs.rename()`
- `fs.renameSync()`
- `fs.write()`
- `fs.writeFileSync()`

H3 事件监听



```
//event.js 文件
var EventEmitter = require('events').EventEmitter;
var event = new EventEmitter();
// 注册一个事件为some_event的监听器
event.on('some_event', function () {
  console.log('some_event 事件触发');
});
setInterval(function () {
  // 像event发送事件some_event 会调用事件some_event的监听器
  event.emit('some_event');
}, 1000);
```

传入参数:



```
var EventEmitter = require('events').EventEmitter;
var event = new EventEmitter();
// 注册第一个事件的监听器
event.on('fun1', function (age1, age2) {
  console.log('fun1', age1, age2);
});
// 注册第二个事件的监听器
event.on('fun2', function (age1, age2) {
  console.log('fun2', age1, age2);
});
var i = 0;
setInterval(function () {
  // 提交时可以传递参数，写在事件之后
  if (i < 5) {
    event.emit('fun1', 'age1 触发啦', 'age2触发啦');
    i++;
  } else {
    event.emit('fun2', 'age1', 'age2');
    i++;
  }
}, 1000);
```

实例:



```
var events = require('events');
var eventEmitter = new events.EventEmitter();

// 监听器 #1
var listener1 = function listener1() {
  console.log('监听器 listener1 执行。');
}

// 监听器 #2
var listener2 = function listener2() {
  console.log('监听器 listener2 执行。');
}

// 绑定 connection 事件，处理函数为 listener1
eventEmitter.addListener('connection', listener1);
```

```
// 绑定 connection 事件，处理函数为 listener2
eventEmitter.on('connection', listener2);

var eventListeners = eventEmitter.listenerCount('connection');
console.log(eventListeners + " 个监听器监听连接事件。");

// 处理 connection 事件
eventEmitter.emit('connection');

// 移除监听绑定的 listener1 函数
eventEmitter.removeListener('connection', listener1);
console.log("listener1 不再受监听。");

// 触发连接事件
eventEmitter.emit('connection');

eventListeners = eventEmitter.listenerCount('connection');
console.log(eventListeners + " 个监听器监听连接事件。");

console.log("程序执行完毕。");
```

以上代码，执行结果如下所示：



```
node main.js
2 个监听器监听连接事件。
监听器 listener1 执行。
监听器 listener2 执行。
listener1 不再受监听。
监听器 listener2 执行。
1 个监听器监听连接事件。
程序执行完毕。
```

④ buffer缓冲区

④ 创建缓冲区



```
// 创建一个长度为 10、且用 0 填充的 Buffer。
const buf1 = Buffer.alloc(10);
```

```
// 创建一个长度为 10、且用 0x1 填充的 Buffer。  
const buf2 = Buffer.alloc(10, 1);  
  
// 创建一个长度为 10、且未初始化的 Buffer。  
// 这个方法比调用 Buffer.alloc() 更快，  
// 但返回的 Buffer 实例可能包含旧数据，  
// 因此需要使用 fill() 或 write() 重写。  
const buf3 = Buffer.allocUnsafe(10);  
  
// 创建一个包含 [0x1, 0x2, 0x3] 的 Buffer。  
const buf4 = Buffer.from([1, 2, 3]);  
  
// 创建一个包含 UTF-8 字节 [0x74, 0xc3, 0xa9, 0x73, 0x74] 的 Buffer。  
const buf5 = Buffer.from('tést');  
  
// 创建一个包含 Latin-1 字节 [0x74, 0xe9, 0x73, 0x74] 的 Buffer。  
const buf6 = Buffer.from('tést', 'latin1');
```

④ 写入缓冲区



语法:

```
buf.write(string[, offset[, length]][, encoding])
```



实例:

```
buf = Buffer.alloc(256);  
len = buf.write("www.runoob.com");  
  
console.log("写入字节数 : "+ len);
```

④ 读取缓冲区



语法:

```
// encoding 使用的编码 默认为'utf-8'  
// start 开始读取的位置 默认为0  
// end 结束的位置 默认为结尾  
buf.toString([encoding[, start[, end]]])
```



实例:

```
buf = Buffer.alloc(26);  
for (var i = 0 ; i < 26 ; i++) {  
    buf[i] = i + 97;  
}  
  
console.log( buf.toString('ascii'));           // 输出:  
abcdefghijklmnopqrstuvwxyz  
console.log( buf.toString('ascii',0,5));        // 使用 'ascii' 编码，并输出:  
出: abcde  
console.log( buf.toString('utf8',0,5));        // 使用 'utf8' 编码，并输出:  
出: abcde  
console.log( buf.toString(undefined,0,5)); // 使用默认的 'utf8' 编码，并输出:  
输出: abcde
```



输出:

```
$ node main.js  
abcdefghijklmnopqrstuvwxyz  
abcde  
abcde  
abcde
```

将buffer转换为JSON对象:



`buf.toJSON()`

H4 缓冲区合并

- **list** - 用于合并的 Buffer 对象数组列表。
- **totalLength** - 指定合并后 Buffer 对象的总长度。



语法:

```
Buffer.concat(list[, totalLength])
```



实例:

```
var buffer1 = Buffer.from(['菜鸟教程']);
var buffer2 = Buffer.from(['www.runoob.com']);
var buffer3 = Buffer.concat([buffer1,buffer2]);
console.log("buffer3 内容: " + buffer3.toString());
```



输出:

```
buffer3 内容: 菜鸟教程www.runoob.com
```

H4 缓冲区比较



语法:

```
buf.compare(otherBuffer);
```



实例：

```
var buffer1 = Buffer.from('ABC');
var buffer2 = Buffer.from('ABCD');
var result = buffer1.compare(buffer2);

if(result < 0) {
    console.log(buffer1 + " 在 " + buffer2 + "之前");
} else if(result == 0){
    console.log(buffer1 + " 与 " + buffer2 + "相同");
} else {
    console.log(buffer1 + " 在 " + buffer2 + "之后");
}
```



输出：

ABC在ABCD之前

H4 拷贝缓冲区：

- **targetBuffer** - 要拷贝的 Buffer 对象。
- **targetStart** - 数字, 可选, 默认: 0
- **sourceStart** - 数字, 可选, 默认: 0
- **sourceEnd** - 数字, 可选, 默认: buffer.length



语法：

```
buf.copy(targetBuffer[, targetStart[, sourceStart[, sourceEnd]]])
```



实例：

```
var buf1 = Buffer.from('abcdefghijkl');
var buf2 = Buffer.from('RUNOOB');

//将 buf2 插入到 buf1 指定位置上
buf2.copy(buf1, 2);

console.log(buf1.toString());
```



输出：

```
abRUNOOBijkl
```

④ 缓冲区长度：



语法：

```
buf.length
```

[buffer手册](#)

③ 控制台方法

传入多个变量：会全部打印出来，中间用空格间隔



```
const x = 'x'
const y = 'y'
console.log(x, y)
```



```
console.log('我的%s已经%d岁', '猫', 2)
```

- `%s` 会格式化变量为字符串
- `%d` 会格式化变量为数字
- `%i` 会格式化变量为其整数部分
- `%o` 会格式化变量为对象

H4 清空控制台

`console.clear()` 会清除控制台（其行为可能取决于所使用的控制台）。

H4 元素计数

`console.count()` 是一个便利的方法，出现的次数将会打印在后边。



```
const x = 1
const y = 2
const z = 3
console.count(
  'x 的值为 ' + x + ' 且已经检查了几次? '
)
console.count(
  'x 的值为 ' + x + ' 且已经检查了几次? '
)
console.count(
  'y 的值为 ' + y + ' 且已经检查了几次? '
)
```

数苹果和橙子：



```
JS
const oranges = ['橙子', '橙子']
const apples = ['苹果']
oranges.forEach(fruit => {
  console.count(fruit)
})
apples.forEach(fruit => {
  console.count(fruit)
})
```

H4 打印堆栈踪迹

可以使用 `console.trace()` 实现：



```
JS
const function2 = () => console.trace()
const function1 = () => function2()
function1()
```

这会打印堆栈踪迹。如果在 Node.js REPL 中尝试此操作，则会打印以下内容：



```
BASH
Trace
  at function2 (repl:1:33)
  at function1 (repl:1:25)
  at repl:1:1
  at ContextifyScript.Script.runInThisContext (vm.js:44:33)
  at REPLServer.defaultEval (repl.js:239:29)
  at bound (domain.js:301:14)
  at REPLServer.runBound [as eval] (domain.js:314:12)
  at REPLServer.onLine (repl.js:440:10)
  at emitOne (events.js:120:20)
  at REPLServer.emit (events.js:210:7)
```

H4 计算用时

可以使用 `time()` 和 `timeEnd()` 轻松地计算函数运行所需的时间：



JS

```
const doSomething = () => console.log('测试')
const measureDoingSomething = () => {
  console.time('doSomething()')
  //做点事，并测量所需的时间。
  doSomething()
  console.timeEnd('doSomething()')
}
measureDoingSomething()
```

④ 输出错误：



```
console.error()
```

将输出内容变色：

可以使用 `npm install chalk` 进行安装，然后就可以使用它：



JS

```
const chalk = require('chalk')
console.log(chalk.yellow('你好'))
```

④ 创建进度条：

[Progress](#) 是一个很棒的软件包，可在控制台中创建进度条。 使用 `npm install progress` 进行安装。

以下代码段会创建一个 10 步的进度条，每 100 毫秒完成一步。 当进度条结束时，则清除定时器：



```
JSSconst ProgressBar = require('progress')
const bar = new ProgressBar(':bar', { total: 10 })
const timer = setInterval(() => {
  bar.tick()
  if (bar.complete) {
    clearInterval(timer)
  }
}, 100)
```

H3 导出及导入

可以通过两种方式进行操作。

第一种方式是将对象赋值给 `module.exports` (这是模块系统提供的对象)，这会使文件只导出该对象：

这种方法可以直接导入，并且直接使用方法。



```
JSSconst car = { brand: 'Ford', model: 'Fiesta' }
module.exports = car
//在另一个文件中
const car = require('./car')
```

第二种方式是将要导出的对象添加为 `exports` 的属性。这种方式可以导出多个对象、函数或数据：

这种方法不可以直接使用方法，导入后需要使用 `item.car` 来使用。



```
JSSconst car = { brand: 'Ford', model: 'Fiesta' }
exports.car = car
```

或直接



```
JS
exports.car = {
  brand: 'Ford',
  model: 'Fiesta'
}
```

在另一个文件中，则通过引用导入的属性来使用它：



```
JS
const items = require('./items')
items.car
```

或



```
JS
const car = require('./items').car
```

`module.exports` 和 `export` 之间有什么区别？

前者公开了它指向的对象。后者公开了它指向的对象的属性。

H3 事件循环

js执行时分为两个队列：

- 调用堆栈：处理 js 中 调用 的事件
- 消息队列：处理一些 触发事件 (定时器、用户点击、敲击键盘)
- 作业队列：处理Promise以及基于 promise 构建的 `async/await` 异步函数，可以尽快的执行异步函数的结果

当js遇到这种定时器时，将会把定时器传入 消息队列，当 调用堆栈 执行结束时将会执行消息队列，由于这种机制，我们不必等待这类代码，可以接着执行其他代码，提高执行效率，像Promise类异步函数为作业队列，将会在调用堆栈执行之后、消息队列执行之前执行。

一个简单的事件循环的阐释

举个例子：

```
● ○ ●

const bar = () => console.log('bar')
const baz = () => console.log('baz')
const foo = () => {
  console.log('foo')
  bar()
  baz()
}
foo()
```

此代码会如预期地打印：

```
● ○ ●

foo
bar
baz
```

当运行此代码时，会首先调用 `foo()`。在 `foo()` 内部，会首先调用 `bar()`，然后调用 `baz()`。

H4 消息队列

`setTimeout(() => {}, 0)` 的用例是调用一个函数，但是是在代码中的每个其他函数已被执行之后。

举个例子：



```
const bar = () => console.log('bar')
const baz = () => console.log('baz')
const foo = () => {
  console.log('foo')
  setTimeout(bar, 0)
  baz()
}
foo()
```

该代码会打印：



```
foo
baz
bar
```

当运行此代码时，会首先调用 `foo()`。在 `foo()` 内部，会首先调用 `setTimeout`，将 `bar` 作为参数传入，并传入 0 作为定时器指示它尽快运行。然后调用 `baz()`。

④ ES6作业队列



```
const bar = () => console.log('bar')
const baz = () => console.log('baz')
const foo = () => {
  console.log('foo')
  setTimeout(bar, 0)
  new Promise((resolve, reject) => resolve('应该在 baz 之后、bar 之前')).then(resolve => console.log(resolve))
  baz()
}
foo()
```

这会打印：



```
foo  
baz  
应该在 baz 之后、bar 之前  
bar
```

一个重要的概念：process.nextTick()



```
process.nextTick() => {  
    //做些事情  
}
```

当在其中传入函数时，则告诉 js 引擎，在调用堆栈结束之后立刻执行其中的函数。

执行顺序：调用堆栈 > nextTick() > 作业队列 > 消息队列

setImmediate()



```
setImmediate() => {  
    //运行一些东西  
}
```

setImmediate将会在下一次迭代中运行

执行顺序：调用堆栈 > nextTick() > 作业队列 > 消息队列 > setImmediate()

H3 Promise

Promise是一种处理异步代码，为了防止陷入回调地狱（回调函数一层套一层）



```
let done = true  
const isItDoneYet = new Promise((resolve, reject) => {  
    if (done) {
```

```
        const workDone = '这是创建的东西'
        resolve(workDone)
    } else {
        const why = '仍然在处理其他事情'
        reject(why)
    }
})
isItDoneYet.then((req) =>{
    console.log(req)
}).catch(res =>{
    console.log(res)
})
```

当成功的时候，将参数传入resolve()，进入被解决状态，失败的时候，将参数传入reject()，进入被拒绝状态。

调用then()的req参数为resolve中的参数，调用catch()的参数为reject中的参数。

```
const fs = require('fs')
const getFile = (fileName) => { return new Promise((resolve, reject) =>
{
    fs.readFile(fileName, (err, data) => {
        if (err) {
            reject(err) // 调用 `reject` 会导致 promise 失败，无论是否传入错误作为参数,
                        // 且不再进行下去。
        }
        resolve(data)
    })
})
}

getFile('/etc/passwd')
    .then(data => console.log(data))
    .catch(err => console.error(err))
```

例如，这是使用 promise 获取并解析 JSON 资源的方法：



```
JSError getFirstUserData = () => { return fetch('/users.json') // 获取用户列表
  .then(response => response.json()) // 解析 JSON
  .then(users => users[0]) // 选择第一个用户
  .then(user => fetch(`/users/${user.name}`)) // 获取用户数据
  .then(userResponse => userResponse.json()) // 解析 JSON}
getFirstUserData()
```

④ async/await

异步函数会返回 promise，例如以下示例：



```
const doSomethingAsync = () => {
  return new Promise(resolve => {
    setTimeout(() => resolve('做些事情'), 3000)
  })
}
```

要调用上边函数时，则再前面加上await，之后调用的代码就会停止直到Promise被解决或者被拒绝，但同时客户端函数必须被定义async。



```
const doSomething = async () => {
  console.log(await doSomethingAsync())
```

实例：

```
const doSomethingAsync = () => {
  return new Promise((resolve) => {
    setTimeout(() => resolve('做些事情'), 3000);
  });
};

const doSomething = async () => {
  console.log(await doSomethingAsync());
};

console.log('之前');
doSomething();
console.log('之后');
```

输出：

```
之前
之后
做些事情 // 3 秒之后
```

这是使用 `async/await` 获取并解析 JSON 资源的方法：

```
const getFirstUserData = async () => {
  const response = await fetch('/users.json') // 获取用户列表
  const users = await response.json() // 解析 JSON
  const user = users[0] // 选择第一个用户
  const userResponse = await fetch(`users/${user.name}`) // 获取用户数
  const userData = await userResponse.json() // 解析 JSON
  return userData
}

getFirstUserData()
```

多个异步实例串联：

```

const promiseToDoSomething = () => {
  return new Promise((resolve) => {
    setTimeout(() => resolve('做些事情'), 10000);
  });
};

const watchOverSomeoneDoingSomething = async () => {
  const something = await promiseToDoSomething();
  return something + ' 查看';
};

const watchOverSomeoneWatchingSomeoneDoingSomething = async () => {
  const something = await watchOverSomeoneDoingSomething();
  return something + ' 再次查看';
};

watchOverSomeoneWatchingSomeoneDoingSomething().then((res) => {
  console.log(res);
});

```

H3 事件触发器

如果你在浏览器中使用 JavaScript，则你会知道通过事件处理了许多用户的交互：鼠标的单击、键盘按钮的按下、对鼠标移动的反应等等。

在后端，Node.js 也提供了使用 [events 模块](#) 构建类似系统的选项。

具体上，此模块提供了 [EventEmitter](#) 类，用于处理事件。

初始化：



```

const EventEmitter = require('events')
const eventEmitter = new EventEmitter()

```

该对象具有两个方法：

- `on` 用于注册回调函数（事件再被触发时执行）
- `emit` 用于提交、触发事件

实例：

```
● ○ ●  
const EventEmitter = require('events');  
const eventEmitter = new EventEmitter();  
//注册事件  
eventEmitter.on('start', () => {  
    console.log('事件被触发了');  
});  
eventEmitter.emit('start');
```

也可以传递参数：

```
● ○ ●  
eventEmitter.on('start', number => {  
    console.log(`开始 ${number}`)  
})  
eventEmitter.emit('start', 23)
```

多个参数：

```
● ○ ●  
eventEmitter.on('start', (start, end) => {  
    console.log(`从 ${start} 到 ${end}`)  
})  
eventEmitter.emit('start', 1, 100)
```

EventEmitter 对象还公开了其他几个与事件进行交互的方法，例如：

- `once()`：添加单次监听器。
- `removeListener() / off()`：从事件中移除事件监听器。
- `removeAllListeners()`：移除事件的所有监听器。

④ 搭建http服务器

这是一个简单的 HTTP web 服务器的示例：

```
const http = require('http');
const hostname = 'localhost';
const port = 3000;
// 创建一个http服务器
// request 提供了请求的详细信息。 通过它可以访问请求头和请求的数据。
// response 用于构造要返回给客户端的数据。
const server = http.createServer((req, res) => {
    // 设置 statusCode 属性为 200，以表明响应成功。
    res.statusCode = 200;
    // 还设置了 Content-Type 响应头：
    res.setHeader('Content-Type', 'text/plain');
    // 最后结束并关闭响应，将内容作为参数添加到 end()：
    res.end('你好世界\n');
});
server.listen(port, () => {
    console.log(`服务器运行在 http://${hostname}:${port}/`);
});
```

简要分析一下。这里引入了 `http` 模块。

使用该模块来创建 HTTP 服务器。

服务器被设置为在指定的 `3000` 端口上进行监听。当服务器就绪时，则 `listen` 回调函数会被调用。

传入的回调函数会在每次接收到请求时被执行。每当接收到新的请求时，`request` 事件会被调用，并提供两个对象：一个请求（`http.IncomingMessage` 对象）和一个响应（`http.ServerResponse` 对象）。

`request` 提供了请求的详细信息。通过它可以访问请求头和请求的数据。

`response` 用于构造要返回给客户端的数据。

在此示例中：



```
res.statusCode = 200
```

设置 statusCode 属性为 200，以表明响应成功。

还设置了 Content-Type 响应头：



```
res.setHeader('Content-Type', 'text/plain')
```

最后结束并关闭响应，将内容作为参数添加到 end()：



```
res.end('你好世界\n')
```

④ 发送http请求

执行GET请求：



```
const https = require('https');
const options = {
  hostname: 'nodejs.cn',
  port: 443,
  path: '/todos',
  method: 'GET',
};
const req = https.request(options, (res) => {
  console.log(`状态码: ${res.statusCode}`);
  res.on('data', (d) => {
    process.stdout.write(d);
  });
});
req.on('error', (error) => {
  console.error(error);
});
req.end();
```

执行POST请求：

```
● ● ●

const https = require('https');

const data = JSON.stringify({
  todo: '做点事情',
});

const options = {
  hostname: 'nodejs.cn',
  port: 443,
  path: '/todos',
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Content-Length': data.length,
  },
};

const req = https.request(options, (res) => {
  console.log(`状态码: ${res.statusCode}`);

  res.on('data', (d) => {
    process.stdout.write(d);
  });
});

req.on('error', (error) => {
  console.error(error);
});

req.write(data);
req.end();
```

④ axios库发送http请求



```
const axios = require('axios');

axios
  .post('http://nodejs.cn/todos', {
    todo: '做点事情',
  })
  .then((res) => {
    console.log(`状态码: ${res.status}`);
    console.log(res);
  })
  .catch((error) => {
    console.error(error);
  });

```

axios为第三方库，使用 `npm install axios` 下载后使用。

④ 路径模块

`path` 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。

无需安装。作为 Node.js 核心的组成部分，可以通过简单地引用来使用它：



```
JS
const path = require('path')
```

该模块提供了 `path.sep`（作为路径段分隔符，在 Windows 上是 `\`，在 Linux/macOS 上是 `/`）和 `path.delimiter`（作为路径定界符，在 Windows 上是 `;`，在 Linux/macOS 上是 `:`）。

还有这些 `path` 方法：

```
path.basename()
```

返回路径的最后一部分。第二个参数可以过滤掉文件的扩展名：



JS

```
require('path').basename('/test/something') //something
require('path').basename('/test/something.txt') //something.txt
require('path').basename('/test/something.txt', '.txt') //something
```

`path.dirname()`

返回路径的目录部分：



JS

```
require('path').dirname('/test/something') // /test
require('path').dirname('/test/something/file.txt') // /test/something
```

`path.basename()`

返回路径的扩展名部分。



JS

```
require('path').basename('/test/something') // ''
require('path').basename('/test/something/file.txt') // '.txt'
```

`path.isAbsolute()`

如果是绝对路径，则返回 true。



JS

```
require('path').isAbsolute('/test/something') // true
require('path').isAbsolute('./test/something') // false
```

`path.join()`

连接路径的两个或多个部分：



JS

```
const name = 'joe'  
require('path').join('/', 'users', name, 'notes.txt')  
// '/users/joe/notes.txt'
```

`path.normalize()`

当包含类似 `..`、`..` 或双斜杠等相对的说明符时，则尝试计算实际的路径：



JS

```
require('path').normalize('/users/joe/.. //test.txt') // '/users/test.txt'
```

`path.parse()`

解析对象的路径为组成其的片段：

- `root`：根路径。
- `dir`：从根路径开始的文件夹路径。
- `base`：文件名 + 扩展名
- `name`：文件名
- `ext`：文件扩展名

例如：



JS

```
require('path').parse('/users/test.txt')
```

结果是：



```
JS
{
  root: '/',
  dir: '/users',
  base: 'test.txt',
  ext: '.txt',
  name: 'test'
}
```

path.relative()

接受 2 个路径作为参数。 基于当前工作目录，返回从第一个路径到第二个路径的相对路径。

例如：



```
JS
require('path').relative('/Users/joe', '/Users/joe/test.txt')
// 'test.txt'
require('path').relative('/Users/joe', '/Users/joe/something/test.txt')
// 'something/test.txt'
```

path.resolve()

可以使用 `path.resolve()` 获得相对路径的绝对路径计算：



```
JS
path.resolve('joe.txt') // '/Users/joe/joe.txt' 如果从主文件夹运行
```

通过指定第二个参数，`resolve` 会使用第一个参数作为第二个参数的基准：



```
JS
path.resolve('tmp', 'joe.txt') // '/Users/joe/tmp/joe.txt' 如果从主文件夹运行
```

如果第一个参数以斜杠开头，则表示它是绝对路径：



JS

```
path.resolve('/etc', 'joe.txt') // '/etc/joe.txt'
```

H3 操作系统模块

引入：



```
const os = require('os')
```

os.arch():

返回标识底层架构的字符串，例如 `arm`、`x64`、`arm64`。



```
const os = require('os');
console.log(os.arch());
```

输出：



```
x64
```

os.cpus():

返回关于系统上可用的 CPU 的信息。



```
const os = require('os');
console.log(os.cpus());
```

os.endianness():

根据是使用 大端序或小端序 编译 Node.js，返回 BE 或 LE。

os.freemem():

返回代表系统中可用内存的字节数。

os.homedir():

返回到当前用户的主目录的路径。

os.hostname():

返回主机名。

输出：



```
LAPTOP-NHQPG308
```

os.loadavg():

返回操作系统对平均负载的计算。仅在Linux和macOS上返回有意义的值

```
os.networkInterfaces():
```

返回系统上可用的网络接口的详细信息。

例如：

```
{  
    WLAN: [  
        {  
            address: 'fd14:7740:ee9a:1b00:ec76:c588:3858:78e5',  
            netmask: 'ffff:ffff:ffff:ffff::',  
            family: 'IPv6',  
            mac: '38:68:93:92:74:b7',  
            internal: false,  
            cidr: 'fd14:7740:ee9a:1b00:ec76:c588:3858:78e5/64',  
            scopeid: 0  
        },  
        {  
            address: 'fd14:7740:ee9a:1b00:d054:6ba:6126:5748',  
            netmask: 'ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff',  
            family: 'IPv6',  
            mac: '38:68:93:92:74:b7',  
            internal: false,  
            cidr: 'fd14:7740:ee9a:1b00:d054:6ba:6126:5748/128',  
            scopeid: 0  
        },  
        {  
            address: 'fe80::ec76:c588:3858:78e5',  
            netmask: 'ffff:ffff:ffff:ffff::',  
            family: 'IPv6',  
            mac: '38:68:93:92:74:b7',  
            internal: false,  
            cidr: 'fe80::ec76:c588:3858:78e5/64',  
            scopeid: 16  
        },  
        {  
            address: '192.168.8.101',  
            netmask: '255.255.255.0',  
            family: 'IPv4',  
            mac: '38:68:93:92:74:b7',  
            internal: false,  
            cidr: '192.168.8.101/24'  
        }  
    ],  
    'Loopback Pseudo-Interface 1': [  
        {
```

```
        address: '::1',
        netmask: 'ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff',
        family: 'IPv6',
        mac: '00:00:00:00:00:00',
        internal: true,
        cidr: '::1/128',
        scopeid: 0
    },
{
    address: '127.0.0.1',
    netmask: '255.0.0.0',
    family: 'IPv4',
    mac: '00:00:00:00:00:00',
    internal: true,
    cidr: '127.0.0.1/8'
}
]
```

os.platform():

返回为node编译的平台：

- darwin
- freebsd
- linux
- openbsd
- win32
- ...等

os.release()

返回标识操作系统版本号的字符串。

os.tmpdir()

返回指定的临时文件夹的路径。

```
os.totalmem()
```

返回表示系统中可用的总内存的字节数。

```
os.type()`
```

标识操作系统：

- Linux
 - macOS 上为 Darwin
 - Windows 上为 Windows_NT
-

```
os.uptime()
```

返回自上次重新启动以来计算机持续运行的秒数。

```
os.userInfo()
```

返回包含当前 username、uid、gid、shell 和 homedir 的对象。

④ 事件模块



```
const EventEmitter = require('events')
const door = new EventEmitter()
```

```
emitter.on/emit.addListener():
```

注册事件。

```
emitter.emit():
```

触发事件。按照事件被注册的顺序同步地调用每个事件监听器。



```
door.emit("slam") // 触发 "slam" 事件。
```

```
emitter.eventNames()
```

返回字符串（表示在当前 `EventEmitter` 对象上注册的事件）数组：



```
door.eventNames()
```

```
emitter.getMaxListeners()
```

获取可以添加到 `EventEmitter` 对象的监听器的最大数量（默认为 10，但可以使用 `setMaxListeners()` 进行增加或减少）。

```
emitter.listenerCount()
```

获取作为参数传入的事件监听器的个数。

```
emitter.listeners()
```

获取作为参数传入的事件监听器的数组。

```
emitter.off() / emitter.removeListener()
```

移除监听器。

emitter.once()

添加当事件在注册之后首次被触发时调用的回调函数。该回调只会被调用一次，不会再被调用。



```
const EventEmitter = require('events')  
const ee = new EventEmitter()  
ee.once('my-event', () => {  
    //只调用一次回调函数。  
})
```

emitter.prependListener()

当使用 `on` 或 `addListener` 添加监听器时，监听器会被添加到监听器队列中的最后一个，并且最后一个被调用。使用 `prependListener` 则可以在其他监听器之前添加并调用。

emitter.prependOnceListener()

当使用 `once` 添加监听器时，监听器会被添加到监听器队列中的最后一个，并且最后一个被调用。使用 `prependOnceListener` 则可以在其他监听器之前添加并调用。

emitter.removeAllListeners()

移除 `EventEmitter` 对象的所有监听特定事件的监听器：



```
JS  
door.removeAllListeners('open')
```

```
emitter.removeListener()
```

移除特定的监听器。可以通过将回调函数保存到变量中（当添加时），以便以后可以引用它：



JS

```
const doSomething = () => {}
door.on('open', doSomething)
door.removeListener('open', doSomething)
```

```
emitter.setMaxListeners()
```

设置可以添加到 `EventEmitter` 对象的监听器的最大数量（默认为 10，但可以增加或减少）。



JS

```
door.setMaxListeners(50)
```

H3 http模块

引入：



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

H4 属性

`http.METHODS`

此属性列出了所有支持HTTP方法：



```
> require('http').METHODS
[ 'ACL',
  'BIND',
  'CHECKOUT',
  'CONNECT',
  'COPY',
  'DELETE',
  'GET',
  'HEAD',
  'LINK',
  'LOCK',
  'M-SEARCH',
  'MERGE',
  'MKACTIVITY',
  'MKCALENDAR',
  'MKCOL',
  'MOVE',
  'NOTIFY',
  'OPTIONS',
  'PATCH',
  'POST',
  'PROPFIND',
  'PROPPATCH',
  'PURGE',
  'PUT',
  'REBIND',
  'REPORT',
  'SEARCH',
  'SUBSCRIBE',
  'TRACE',
  'UNBIND',
  'UNLINK',
  'UNLOCK',
  'UNSUBSCRIBE' ]
```

http.STATUS_CODES

此属性列出了所有的 HTTP 状态代码及其描述:



```
> require('http').STATUS_CODES
{ '100': 'Continue',
  '101': 'Switching Protocols',
  '102': 'Processing',
  '200': 'OK',
```

```
'201': 'Created',
'202': 'Accepted',
'203': 'Non-Authoritative Information',
'204': 'No Content',
'205': 'Reset Content',
'206': 'Partial Content',
'207': 'Multi-Status',
'208': 'Already Reported',
'226': 'IM Used',
'300': 'Multiple Choices',
'301': 'Moved Permanently',
'302': 'Found',
'303': 'See Other',
'304': 'Not Modified',
'305': 'Use Proxy',
'307': 'Temporary Redirect',
'308': 'Permanent Redirect',
'400': 'Bad Request',
'401': 'Unauthorized',
'402': 'Payment Required',
'403': 'Forbidden',
'404': 'Not Found',
'405': 'Method Not Allowed',
'406': 'Not Acceptable',
'407': 'Proxy Authentication Required',
'408': 'Request Timeout',
'409': 'Conflict',
'410': 'Gone',
'411': 'Length Required',
'412': 'Precondition Failed',
'413': 'Payload Too Large',
'414': 'URI Too Long',
'415': 'Unsupported Media Type',
'416': 'Range Not Satisfiable',
'417': 'Expectation Failed',
'418': "I'm a teapot",
'421': 'Misdirected Request',
'422': 'Unprocessable Entity',
'423': 'Locked',
'424': 'Failed Dependency',
'425': 'Unordered Collection',
'426': 'Upgrade Required',
'428': 'Precondition Required',
'429': 'Too Many Requests',
'431': 'Request Header Fields Too Large',
'451': 'Unavailable For Legal Reasons',
'500': 'Internal Server Error',
'501': 'Not Implemented',
'502': 'Bad Gateway',
```

```
'503': 'Service Unavailable',
'504': 'Gateway Timeout',
'505': 'HTTP Version Not Supported',
'506': 'Variant Also Negotiates',
'507': 'Insufficient Storage',
'508': 'Loop Detected',
'509': 'Bandwidth Limit Exceeded',
'510': 'Not Extended',
'511': 'Network Authentication Required' }
```

http.globalAgent

指向 Agent 对象的全局实例，该实例是 `http.Agent` 类的实例。

用于管理 HTTP 客户端连接的持久性和复用，它是 Node.js HTTP 网络的关键组件。

方法

http.createServer()

返回 `http.Server` 类的新实例，创建。

用法：



```
const server = http.createServer((req, res) => {
  // 使用此回调处理每个单独的请求。
})
```

http.request()

发送HTTP请求到服务器，并创建`http.ClientRequest`类的实例。

http.get()

类似于 `http.request()`，但会自动地设置 HTTP 方法为 GET，并自动地调用 `req.end()`。

H4 类

HTTP 模块提供了 5 个类：

- `http.Agent`
- `http.ClientRequest`
- `http.Server`
- `http.ServerResponse`
- `http.IncomingMessage`

`http.Agent`

Node.js 会创建 `http.Agent` 类的全局实例，以管理 HTTP 客户端连接的持久性和复用，这是 Node.js HTTP 网络的关键组成部分。

该对象会确保对服务器的每个请求进行排队并且单个 socket 被复用。

它还维护一个 socket 池。出于性能原因，这是关键。

`http.ClientRequest`

当 `http.request()` 或 `http.get()` 被调用时，会创建 `http.ClientRequest` 对象。

当响应被接收时，则会使用响应（`http.IncomingMessage` 实例作为参数）来调用 `response` 事件。

返回的响应数据可以通过以下两种方式读取：

- 可以调用 `response.read()` 方法。
 - 在 `response` 事件处理函数中，可以为 `data` 事件设置事件监听器，以便可以监听流入的数据。
-

`http.Server`

当使用 `http.createServer()` 创建新的服务器时，通常会实例化并返回此类。

拥有服务器对象后，就可以访问其方法：

- `close()` 停止服务器不再接受新的连接。
 - `listen()` 启动 HTTP 服务器并监听连接。
-

`http.ServerResponse`

由 `http.Server` 创建，并作为第二个参数传给它触发的 `request` 事件。

通常在代码中用作 `res`：



JS

```
const server = http.createServer((req, res) => {
  //res 是一个 http.ServerResponse 对象。
})
```

在事件处理函数中总是会调用的方法是 `end()`，它会关闭响应，当消息完成时则服务器可以将其发送给客户端。必须在每个响应上调用它。

以下这些方法用于与 HTTP 消息头进行交互：

- `getHeaderNames()` 获取已设置的 HTTP 消息头名称的列表。
- `getHeaders()` 获取已设置的 HTTP 消息头的副本。
- `setHeader('headername', value)` 设置 HTTP 消息头的值。
- `getHeader('headername')` 获取已设置的 HTTP 消息头。
- `removeHeader('headername')` 删除已设置的 HTTP 消息头。
- `hasHeader('headername')` 如果响应已设置该消息头，则返回 true。
- `headersSent()` 如果消息头已被发送给客户端，则返回 true。

在处理消息头之后，可以通过调用 `response.writeHead()`（该方法接受 `statusCode` 作为第一个参数，可选的状态消息和消息头对象）将它们发送给客户端。

若要在响应正文中发送数据给客户端，则使用 `write()`。它会发送缓冲的数据到 HTTP 响应流。

如果消息头还未被发送，则使用 `response.writeHead()` 会先发送消息头，其中包含在请求中已被设置的状态码和消息，可以通过设置 `statusCode` 和 `statusMessage` 属性的值进行编辑：



JS

```
response.statusCode = 500  
response.statusMessage = '内部服务器错误'
```

`http.IncomingMessage`

`http.IncomingMessage` 对象可通过以下方式创建：

- `http.Server`，当监听 `request` 事件时。
- `http.ClientRequest`，当监听 `response` 事件时。

它可以用来自访问响应：

- 使用 `statusCode` 和 `statusMessage` 方法来访问状态。
- 使用 `headers` 方法或 `rawHeaders` 来访问消息头。
- 使用 `method` 方法来访问 HTTP 方法。
- 使用 `httpVersion` 方法来访问 HTTP 版本。
- 使用 `url` 方法来访问 URL。
- 使用 `socket` 方法来访问底层的 socket。

因为 `http.IncomingMessage` 实现了可读流接口，因此数据可以使用流访问。

H3 流

H4 什么是流

流是为 Node.js 应用程序提供动力的基本概念之一。

它们是一种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。

流不是 Node.js 特有的概念。它们是几十年前在 Unix 操作系统中引入的，程序可以通过管道运算符 (`|`) 对流进行相互交互。

例如，在传统的方式中，当告诉程序读取文件时，这会将文件从头到尾读入内存，然后进行处理。

使用流，则可以逐个片段地读取并处理（而无需全部保存在内存中）。

Node.js 的 `stream` 模块 提供了构建所有流 API 的基础。所有的流都是 `EventEmitter` 的实例。

H4 为什么要使用流

相对于使用其他的数据处理方法，流基本上提供了两个主要优点：

- **内存效率**：无需加载大量的数据到内存中即可进行处理。
- **时间效率**：当获得数据之后即可立即开始处理数据，这样所需的时间更少，而不必等到整个数据有效负载可用才开始。

H4 示例

一个典型的例子是从磁盘读取文件。

使用 Node.js 的 `fs` 模块，可以读取文件，并在与 HTTP 服务器建立新连接时通过 HTTP 提供文件：

```
● ● ●  
  
const http = require('http')
const fs = require('fs')
const server = http.createServer(function(req, res) {
  fs.readFile(__dirname + '/data.txt', (err, data) => {
    res.end(data)
  })
})
server.listen(3000)
```

`readFile()` 读取文件的全部内容，并在完成时调用回调函数。

回调中的 `res.end(data)` 会返回文件的内容给 HTTP 客户端。

如果文件很大，则该操作会花费较多的时间。以下是使用流编写的相同内容：

```
http = require('http');
const fs = require('fs');
const server = http.createServer((req, res) => {
  const stream = fs.createReadStream(__dirname + '/test.txt');
  stream.pipe(res);
});
server.listen(3000);
```

当要发送的数据块已获得时就立即开始将其流式传输到 HTTP 客户端，而不是等待直到文件被完全读取。

H4 pipe()

上面的示例使用了 `stream.pipe(res)` 这行代码：在文件流上调用 `pipe()` 方法。

该代码的作用是什么？它获取来源流，并将其通过管道传输到目标流。

在来源流上调用它，在该示例中，文件流通过管道传输到 HTTP 响应。

`pipe()` 方法的返回值是目标流，这是非常方便的事情，它使得可以链接多个 `pipe()` 调用，如下所示：

```
src.pipe(dest1).pipe(dest2)
```

此构造相对于：

```
src.pipe(dest1)
dest1.pipe(dest2)
```

H4 流驱动的 Node.js API

由于它们的优点，许多 Node.js 核心模块提供了原生的流处理功能，最值得注意的有：

- `process.stdin` 返回连接到 `stdin` 的流。
- `process.stdout` 返回连接到 `stdout` 的流。
- `process.stderr` 返回连接到 `stderr` 的流。
- `fs.createReadStream()` 创建文件的可读流。
- `fs.createWriteStream()` 创建到文件的可写流。
- `net.connect()` 启动基于流的连接。
- `http.request()` 返回 `http.ClientRequest` 类的实例，该实例是可写流。
- `zlib.createGzip()` 使用 `gzip`（压缩算法）将数据压缩到流中。
- `zlib.createGunzip()` 解压缩 `gzip` 流。
- `zlib.createDeflate()` 使用 `deflate`（压缩算法）将数据压缩到流中。
- `zlib.createInflate()` 解压缩 `deflate` 流。

H4 不同类型的流

流分为四类：

- `Readable`：可以通过管道读取、但不能通过管道写入的流（可以接收数据，但不能向其发送数据）。当推送数据到可读流中时，会对其进行缓冲，直到使用者开始读取数据为止。
- `Writable`：可以通过管道写入、但不能通过管道读取的流（可以发送数据，但不能从中接收数据）。
- `Duplex`：可以通过管道写入和读取的流，基本上相对于是可读流和可写流的组合。
- `Transform`：类似于双工流、但其输出是其输入的转换的转换流。

H4 创建可读流

从 `stream 模块` 获取可读流，对其进行初始化并实现 `readable._read()` 方法。

首先创建流对象：



```
const Stream = require('stream')
const readableStream = new Stream.Readable()
```

然后实现 `_read` :



```
readableStream._read = () => {}
```

也可以使用 `read` 选项实现 `_read` :



```
const readableStream = new Stream.Readable({
  read() {}
})
```

现在，流已初始化，可以向其发送数据了：



```
readableStream.push('hi!')
readableStream.push('ho!')
```

④ 如何创建可写流

若要创建可写流，需要继承基本的 `Writable` 对象，并实现其 `_write()` 方法。

首先创建流对象：



```
const Stream = require('stream')
const writableStream = new Stream.Writable()
```

然后实现 `_write` :



```
writableStream._write = (chunk, encoding, next) => {
  console.log(chunk.toString())
  next()
}
```

现在，可以通过以下方式传输可读流：



JS

```
process.stdin.pipe(writableStream)
```

H4 从可读流中获取数据

如何从可读流中读取数据？ 使用可写流：



```
const Stream = require('stream');
const readableStream = new Stream.Readable({
  read() {},
});
const writableStream = new Stream.Writable();
writableStream._write = (chunk, encoding, next) => {
  console.log(chunk.toString());
  next();
};
readableStream.pipe(writableStream);
readableStream.push('hi!');
readableStream.push('ho!');
```

也可以使用 `readable` 事件直接地消费可读流：



```
readableStream.on('readable', () => {
  console.log(readableStream.read())
})
```

如何发送数据到可写流

使用流的 `write()` 方法:



```
writableStream.write('hey!\n')
```

④ 使用信号通知已结束写入的可写流

使用 `end()` 方法:



```
const Stream = require('stream');
const readableStream = new Stream.Readable({
  read() {},
});
const writableStream = new Stream.Writable();

writableStream._write = (chunk, encoding, next) => {
  console.log(chunk.toString());
  next();
};
readableStream.pipe(writableStream);
readableStream.push('hi!');
readableStream.push('ho!');
writableStream.end();
```

EXPRESS 框架

第一个 Express 框架实例

接下来我们使用 Express 框架来输出 "Hello World"。

以下实例中我们引入了 `express` 模块，并在客户端发起请求后，响应 "Hello World" 字符串。

创建 `express_demo.js` 文件，代码如下所示:



```
//express_demo.js 文件
var express = require('express');
var app = express();

app.get('/', function (req, res) {
    res.send('Hello World');
})

var server = app.listen(8081, function () {
    var host = server.address().address
    var port = server.address().port

    console.log("应用实例，访问地址为 http://%s:%s", host, port)
})
```

请求和响应

Express 应用使用回调函数的参数：**request** 和 **response** 对象来处理请求和响应的数据。



```
app.get('/', function (req, res) {
    // --
})
```

Request 对象 - **request** 对象表示 HTTP 请求，包含了请求查询字符串，参数，内容，HTTP 头部等属性。常见属性有：

01. **req.app**: 当callback为外部文件时，用**req.app**访问**express**的实例
02. **req.baseUrl**: 获取路由当前安装的URL路径
03. **req.body / req.cookies**: 获得「请求主体」/ Cookies
04. **req.fresh / req.stale**: 判断请求是否还「新鲜」
05. **req.hostname / req.ip**: 获取主机名和IP地址
06. **req.originalUrl**: 获取原始请求URL
07. **req.params**: 获得路由的parameters

08. req.path: 获取请求路径
09. req.protocol: 获取协议类型
10. req.query: 获取URL的查询参数串
11. req.route: 获取当前匹配的路由
12. req.subdomains: 获取子域名
13. req.accepts(): 检查可接受的请求的文档类型
14. req.acceptsCharsets / req.acceptsEncodings / req.acceptsLanguages: 返回指定字符集的第一个可接受字符编码
15. req.get(): 获取指定的HTTP请求头
16. req.is(): 判断请求头Content-Type的MIME类型

Response 对象 - response 对象表示 HTTP 响应，即在接收到请求时向客户端发送的 HTTP 响应数据。常见属性有：

01. res.app: 同req.app一样
02. res.append(): 追加指定HTTP头
03. res.set()在res.append()后将重置之前设置的头
04. res.cookie(name, value [, option]): 设置Cookie
05. option: domain / expires / httpOnly / maxAge / path / secure / signed
06. res.clearCookie(): 清除Cookie
07. res.download(): 传送指定路径的文件
08. res.get(): 返回指定的HTTP头
09. res.json(): 传送JSON响应
10. res.jsonp(): 传送JSONP响应
11. res.location(): 只设置响应的Location HTTP头，不设置状态码或者close response
12. res.redirect(): 设置响应的Location HTTP头，并且设置状态码302
13. res.render(view,[locals],callback): 渲染一个view，同时向callback传递渲染后的字符串，如果在渲染过程中有错误发生next(err)将会被自动调用。callback将会被传入一个可能发生的错误以及渲染后的页面，这样就不会自动输出了。
14. res.send(): 传送HTTP响应
15. res.sendFile(path [, options] [, fn]): 传送指定路径的文件 -会自动根据文件extension设定Content-Type
16. res.set(): 设置HTTP头，传入object可以一次设置多个头

17. res.status(): 设置HTTP状态码

18. res.type(): 设置Content-Type的MIME类型

路由

```
● ○ ●

var express = require('express');
var app = express();

// 主页输出 "Hello World"
app.get('/', function (req, res) {
    console.log("主页 GET 请求");
    res.send('Hello GET');
})

// POST 请求
app.post('/', function (req, res) {
    console.log("主页 POST 请求");
    res.send('Hello POST');
})

// /del_user 页面响应
app.get('/del_user', function (req, res) {
    console.log("/del_user 响应 DELETE 请求");
    res.send('删除页面');
})

// /list_user 页面 GET 请求
app.get('/list_user', function (req, res) {
    console.log("/list_user GET 请求");
    res.send('用户列表页面');
})

// 对页面 abcd, abxcd, ab123cd, 等响应 GET 请求
app.get('/ab*cd', function(req, res) {
    console.log("/ab*cd GET 请求");
    res.send('正则匹配');
})

var server = app.listen(8081, function () {

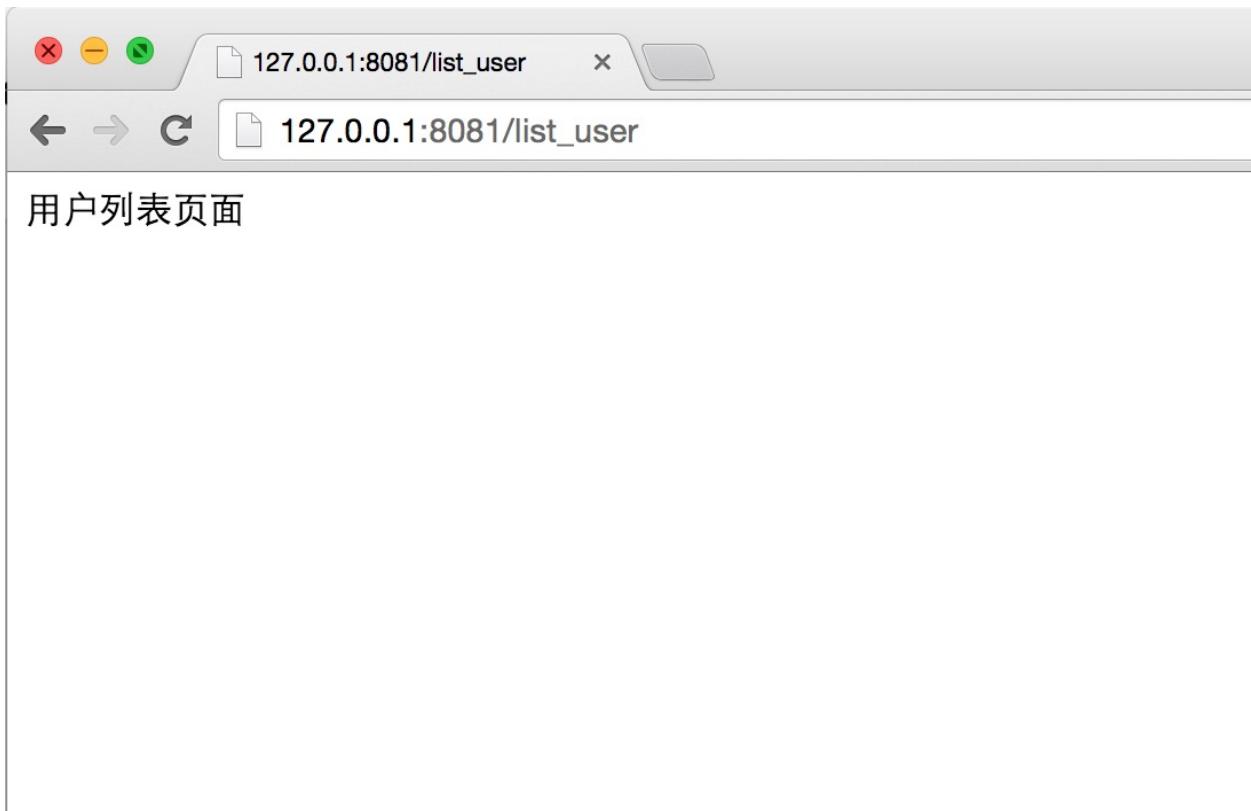
    var host = server.address().address
    var port = server.address().port

    console.log("应用实例, 访问地址为 http://%s:%s", host, port)
})
```

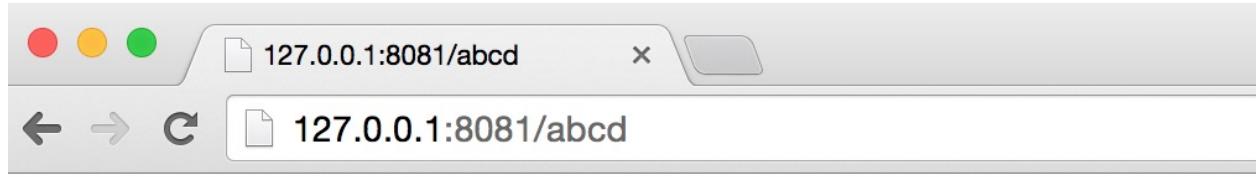
})

接下来你可以尝试访问 <http://127.0.0.1:8081> 不同的地址，查看效果。

在浏览器中访问 http://127.0.0.1:8081/list_user，结果如下图所示：

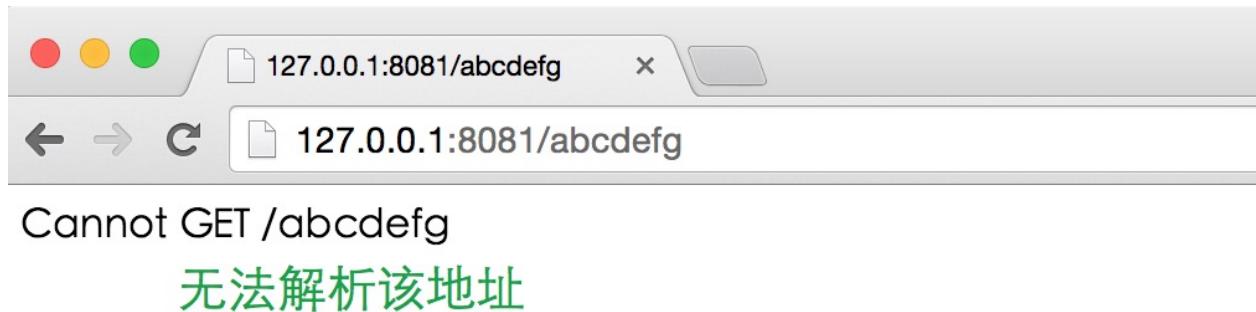


在浏览器中访问 <http://127.0.0.1:8081/abcd>，结果如下图所示：



正则匹配

在浏览器中访问 <http://127.0.0.1:8081/abcdefg>，结果如下图所示：



静态文件

Express 提供了内置的中间件 `express.static` 来设置静态文件如：图片， CSS, JavaScript 等。

你可以使用 `express.static` 中间件来设置静态文件路径。例如，如果你将图片， CSS, JavaScript 文件放在 public 目录下，你可以这么写：



```
app.use('/public', express.static('public'));
```

我们可以到 public/images 目录下放些图片,如下所示:



```
node_modules
server.js
public/
public/images
public/images/logo.png
```

实例：



```
var express = require('express');
var app = express();

app.use('/public', express.static('public'));

app.get('/', function (req, res) {
  res.send('Hello World');
})

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port

  console.log("应用实例, 访问地址为 http://%s:%s", host, port)
})
```

执行以上代码：

在浏览器中访问 <http://127.0.0.1:8081/public/images/logo.png> (本实例采用了菜鸟教程的 logo) , 结果如下图所示：



GET方法

以下实例演示了在表单中通过 GET 方法提交两个参数，我们可以使用 server.js 文件内的 `process_get` 路由器来处理输入：

`index.html` 文件代码：

```
<html>
  <body>
    <form action="http://127.0.0.1:8081/process_get" method="GET">
      First Name: <input type="text" name="first_name"> <br>

      Last Name: <input type="text" name="last_name">
      <input type="submit" value="Submit">
    </form>
  </body>
</html>
```

`server.js` 文件代码：

```
var express = require('express');
var app = express();

app.use('/public', express.static('public'));

app.get('/index.html', function (req, res) {
  res.sendFile(__dirname + "/" + "index.html");
})

app.get('/process_get', function (req, res) {
```

```
// 输出 JSON 格式
var response = {
  "first_name":req.query.first_name,
  "last_name":req.query.last_name
};
console.log(response);
res.end(JSON.stringify(response));
}

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

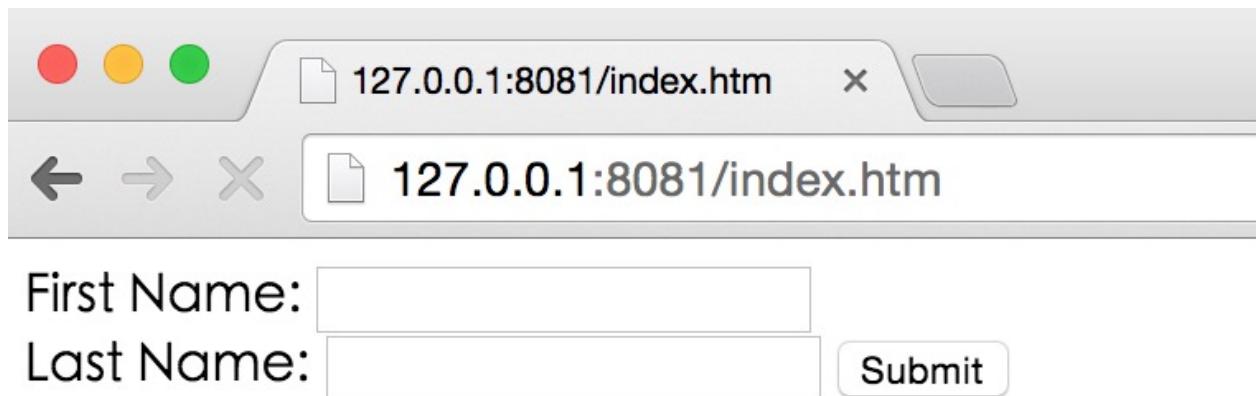
  console.log("应用实例，访问地址为 http://%s:%s", host, port)
})
```

执行以上代码：

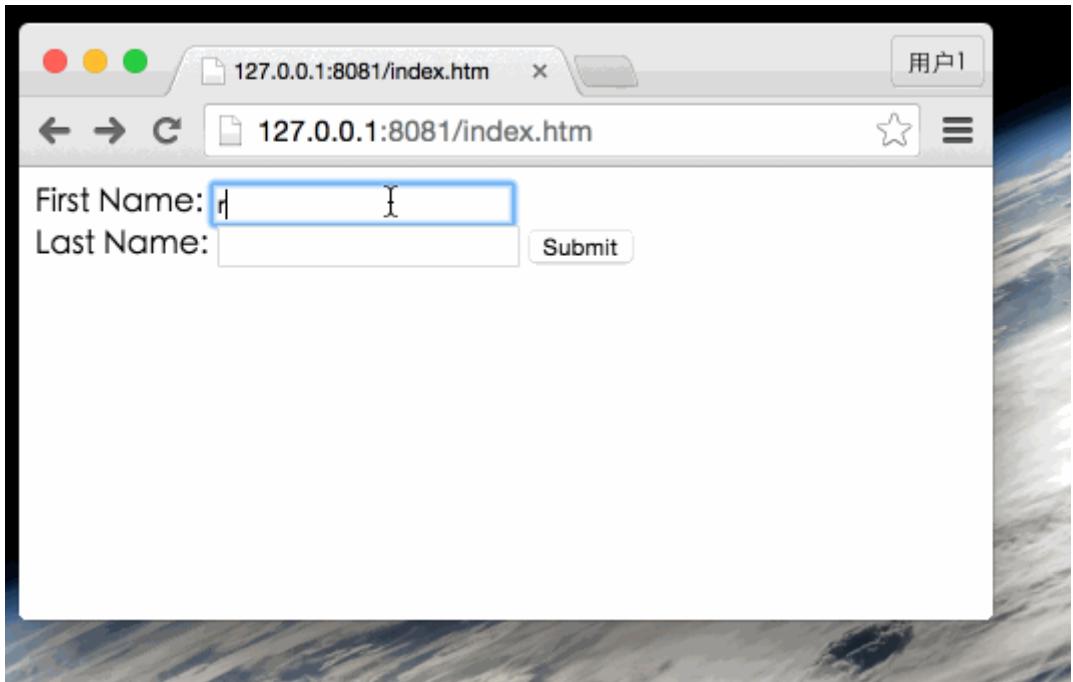
```
● ○ ●

node server.js
应用实例，访问地址为 http://0.0.0.0:8081
```

浏览器访问 <http://127.0.0.1:8081/index.html>，如图所示：



现在你可以向表单输入数据，并提交，如下演示：



POST

以下实例演示了在表单中通过 POST 方法提交两个参数，我们可以使用 server.js 文件内的 `process_post` 路由器来处理输入：

`index.html` 文件代码：

```
<html>
<body>
<form action="http://127.0.0.1:8081/process_post" method="POST">
First Name: <input type="text" name="first_name"> <br>

Last Name: <input type="text" name="last_name">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

`server.js` 文件代码：

```
var express = require('express');
var app = express();
var bodyParser = require('body-parser');
```

```
// 创建 application/x-www-form-urlencoded 编码解析
var urlencodedParser = bodyParser.urlencoded({ extended: false })

app.use('/public', express.static('public'));

app.get('/index.html', function (req, res) {
  res.sendFile(__dirname + "/" + "index.html");
})

app.post('/process_post', urlencodedParser, function (req, res) {

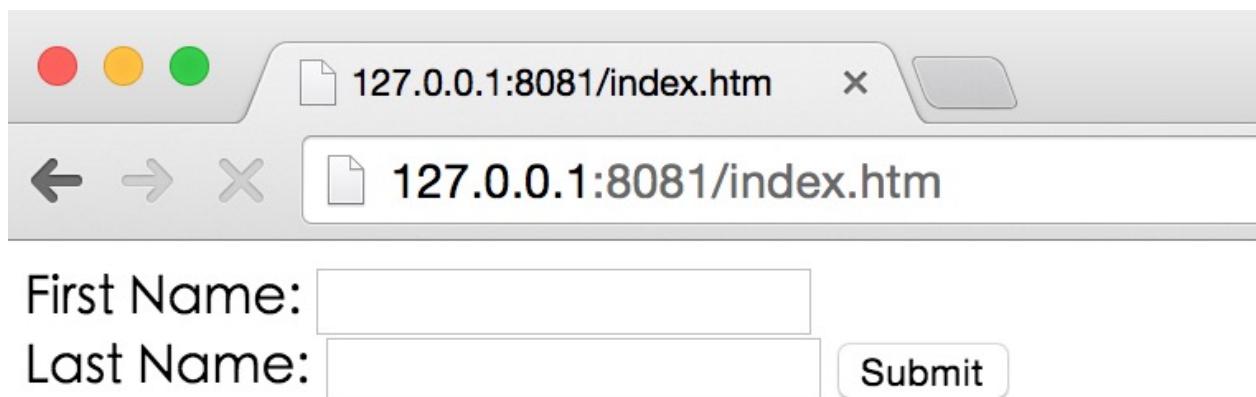
  // 输出 JSON 格式
  var response = {
    "first_name": req.body.first_name,
    "last_name": req.body.last_name
  };
  console.log(response);
  res.end(JSON.stringify(response));
})

var server = app.listen(8081, function () {

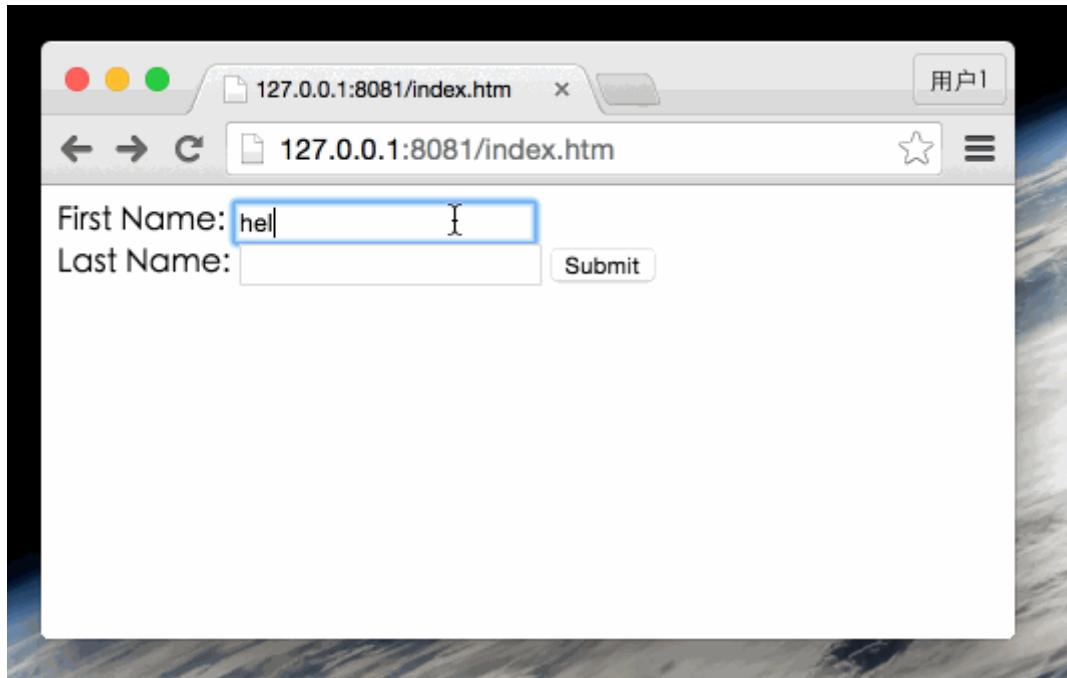
  var host = server.address().address
  var port = server.address().port

  console.log("应用实例，访问地址为 http://%s:%s", host, port)
})
```

浏览器访问 <http://127.0.0.1:8081/index.html>，如图所示：



现在你可以向表单输入数据，并提交，如下演示：



文件上传

以下我们创建一个用于上传文件的表单，使用 POST 方法，表单 enctype 属性设置为 multipart/form-data。

index.html 文件代码：

```
index.html 文件代码：

<html>
<head>
<title>文件上传表单</title>
</head>
<body>
<h3>文件上传：</h3>
选择一个文件上传：<br />
<form action="/file_upload" method="post" enctype="multipart/form-data">
<input type="file" name="image" size="50" />
<br />
<input type="submit" value="上传文件" />
</form>
</body>
</html>
```

server.js 文件代码：



```
var express = require('express');
var app = express();
var fs = require("fs");

var bodyParser = require('body-parser');
var multer = require('multer');

app.use('/public', express.static('public'));
app.use(bodyParser.urlencoded({ extended: false }));
app.use(multer({ dest: '/tmp/' }).array('image'));

app.get('/index.html', function (req, res) {
    res.sendFile(__dirname + "/" + "index.html");
})

app.post('/file_upload', function (req, res) {

    console.log(req.files[0]); // 上传的文件信息

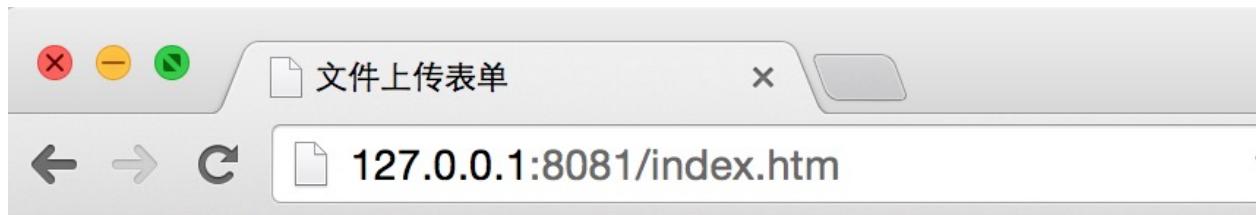
    var des_file = __dirname + "/" + req.files[0].originalname;
    fs.readFile(req.files[0].path, function (err, data) {
        fs.writeFile(des_file, data, function (err) {
            if( err ){
                console.log( err );
            }else{
                response = {
                    message:'File uploaded successfully',
                    filename:req.files[0].originalname
                };
            }
            console.log( response );
            res.end( JSON.stringify( response ) );
        });
    });
})

var server = app.listen(8081, function () {

    var host = server.address().address
    var port = server.address().port

    console.log("应用实例，访问地址为 http://%s:%s", host, port)
})
```

浏览器访问 <http://127.0.0.1:8081/index.html>, 如图所示:

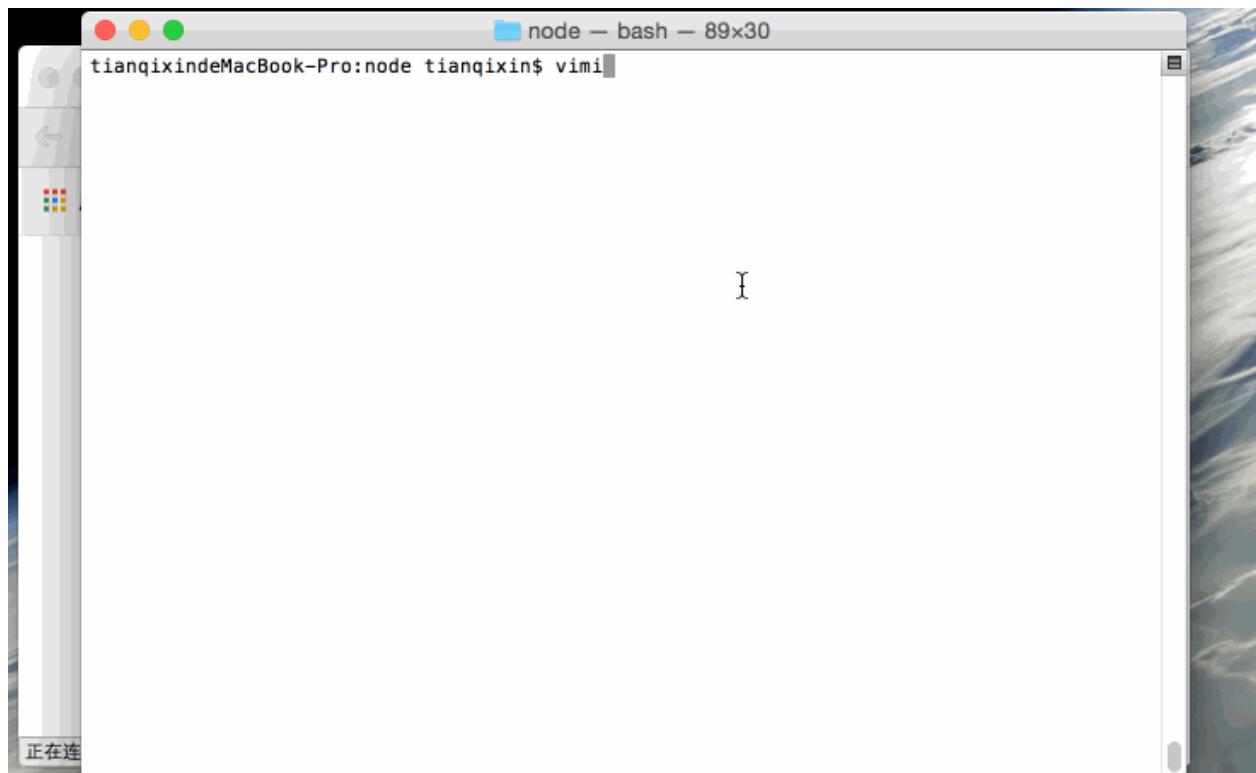


文件上传:

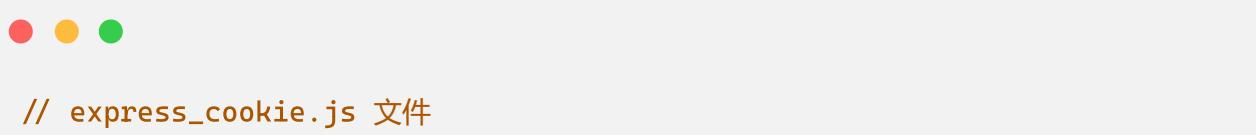
选择一个文件上传:

未选择任何文件

现在你可以向表单输入数据，并提交，如下演示:



Cookie 管理:



```
var express      = require('express')
var cookieParser = require('cookie-parser')
var util = require('util');

var app = express()
app.use(cookieParser())

app.get('/', function(req, res) {
  console.log("Cookies: " + util.inspect(req.cookies));
})

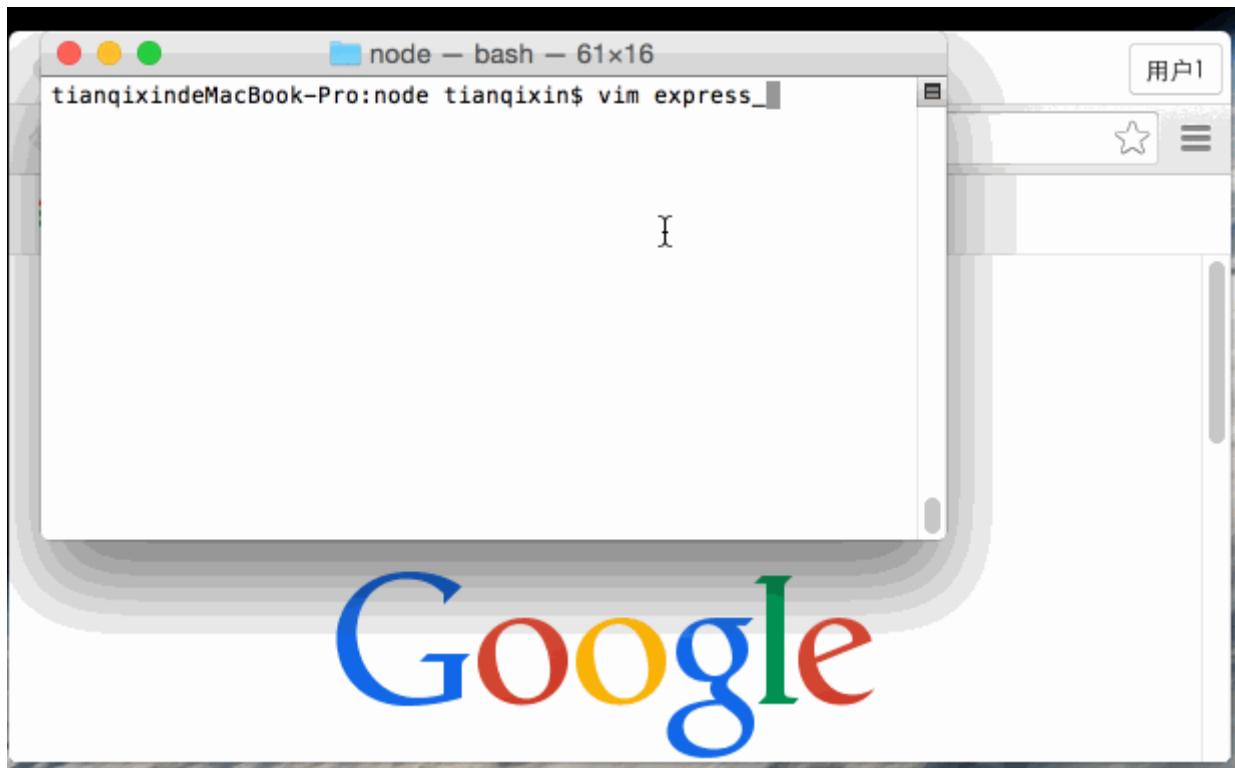
app.listen(8081)// express_cookie.js 文件
var express      = require('express')
var cookieParser = require('cookie-parser')
var util = require('util');

var app = express()
app.use(cookieParser())

app.get('/', function(req, res) {
  console.log("Cookies: " + util.inspect(req.cookies));
})

app.listen(8081)
```

现在你可以访问 <http://127.0.0.1:8081> 并查看终端信息的输出，如下演示：



EXPRESS路由

H3 创建一个基本的路由：



```
const express = require("express")
const app = express()
app.get("/",(req, res) => {
    console.log("Hello world")
})
app.post("/",(req, res) => {
    console.log("POST require")
})
// 启动服务器 运行在80端口
app.listen(80,() => {
    console.log("Server running at https://127.0.0.1")
})
```

H3 模块化路由：

创建模块化路由：

文件名： router.js



```
// 导入 express
const express = require("express")
// 创建路由对象
const router = express.Router()
// 挂接具体的路由
router.get("/user/list",(req, res) => {
    res.send('Get user list.')
})
// POST请求
router.post("/user/list", (req, res) => {
    res.send('Add new user.')
})
// 向外导出路由对象
module.export = router
```

使用路由模块：

app.use() 用来注册全局中间件；

如果想要给路由添加统一的访问前缀：

在 app.use('/api', router) 中添加第一个位置添加参数；这样想要访问的时候必须都加上前缀，否则会 404

文件名： index.js



```
// 导入 router 模块
const router = require("./router")
// 注册 路由 模块

app.use(router)
//启动服务
app.listen(80, () => {
    console.log("server running at https://127.0.0.1")
})
```

中间件

④ 定义一个最简单的中间件



```
const express = require("express")
const app = express()
// 定义一个中间件 mw指向的是一个中间件函数
const mw = function(req, res, next){
    console.log("这是一个最简单的中间件")
    // 把流转关系，转交给下一个中间件或路由
    next()
}
// 使用 app.use() 将中间件注册为全局生效
app.use(mw)
```

```
app.get('/', (req, res) => {
  res.send("Home page")
})

app.get('/user', (req, res) => {
  res.send("User page")
})
app.listen(80, () => {
  console.log("Server running at http://127.0.0.1")
})
```

也可以直接把中间件写在 `app.use()` 中,这样写是一个全局中间件

```
const express = require("express")
const app = express()

// 使用 app.use() 将中间件注册为全局生效
app.use(function(req, res, next){
  console.log("这是一个最简单的中间件")
  // 把流转关系, 转交给下一个中间件或路由
  next()
})

app.get('/', (req, res) => {
  res.send("Home page")
})

app.get('/user', (req, res) => {
  res.send("User page")
})
app.listen(80, () => {
  console.log("Server running at http://127.0.0.1")
})
```

局部生效的中间件:

在 `app.get()` 中传入第二个参数来定义局部中间件, 仅在此路由中生效

```
const express = require("express")
const app = express()
```

```

const mw = function(req, res, next){
    console.log("这个是一个最简单中间件")
}

//使用局部中间件
app.get('/', mw, (req, res) => {
    console("Home page")
})

app.get('/user'(req, res) => {
    console.log("User page")
})

app.listen(80, () => {
    console.log("Server running at http://127.0.0.1")
})

```

多个局部生效的中间件：

app.get('/', mw, mw1, (req, res) => {}) 等价于

app.get('/', [mw, mw1], (req, res) => {})



```

const express = require("express")
const app = express()

const mw = function(req, res, next){
    console.log("这个是一个最简单中间件")
}

const mw1 = function(req, res, next){
    console.log("这个是一个最简单中间件")
}

//使用局部中间件
app.get('/', mw, mw1, (req, res) => {
    console("Home page")
})

//上下等价
app.get('/', [mw, mw1], (req, res) => {
    console("Home page")
})

app.get('/user'(req, res) => {
    console.log("User page")
})

app.listen(80, () => {
    console.log("Server running at http://127.0.0.1")
})

```

H3 中间件的作用

多个中间件会共享 req,res 中的参数



```
const express = require("express")
const app = express()

// 使用 app.use() 将中间件注册为全局生效
app.use(function(req, res, next){
    const time = Date.now()
    // 为req对象，挂载自定义属性，从而把时间共享给每一个路由
    req.startTime = time
    next()
})

app.get('/', (req, res) => {
    res.send("Home page" + req.startTime)
})

app.get('/user', (req, res) => {
    res.send("User page" + req.startTime)
})
app.listen(80, () => {
    console.log("Server running at http://127.0.0.1")
})
```

H3 Express内置中间件

01. `express.static()` 快速托管静态资源的内置中间件，eg：HTML文件、图片、CSS样式。
02. `express.json()` 解析 JSON 格式的请求体数据 (4.16.0+)
03. `express.urlencoded()` 解析 URL-encoded 格式的请求体数据 (4.16.0+)

H3 写接口

文件: `index.js`

```
const express = require('express');

const app = express();

const router = require('./apiRouter');

// 注册router
app.use('/api', router);

app.listen(8080, () => {
  console.log('Express server running at http://127.0.0.1:8080');
});
```

文件: `apiRouter.js`

```
const express = require('express');
const router = express.Router();

router.get('/get', (req, res) => {
  // 使用 res.query 获取客户端通过查询字符串，发送到服务器的数据
  const query = req.query;
  // 调用 res.send() 方法，向客户端响应处理结果
  res.send({
    status: 0, // 0 : 成功 1: 失败
    msg: 'GET 请求成功', // 状态描述
    data: query, // 需要相应给客户端的数据
  });
});

// 导出
module.exports = router;
```

可以使用 `req.query` 来获取用户通过查询的字符串

可以使用 `req.body` 来获取用户传过来的字符串，注意需要先用中间件进行解析

④ 使用CORS解决跨域问题

如果遇到跨域问题：

```
const cors = require("cors")
const express = require("express")
const app = express()
app.use(cors())
```

这样就可以解决跨域问题了

SQL语句

④ SELECT 查询

语法：关键字h大小写都可以

```
-- 这是注释

-- 从 FROM 指定的表中，查询出所有数据。* 表示所有列
SELECT * FROM 表名称

-- 从 FROM 指定的表中，查询出指定列名称的数据
SELECT 列名称 FROM 表名称

select username, password from users
```

④ INSERT 插入

语法：



```
insert into 表名(元素, 元素) values('待添加的元素', '待添加的元素')
```



```
-- 向users这个表中的 username, password 属性添加元素  
insert into users(username, password) values('tony stark', '098123')
```

H3 Update 更新

注意：要对应字符类型，字符类型一定要加 ''



```
UPDATE 表名称 SET 列名称=新值 WHERE 列名称=某值
```

```
UPDATE users SET username='chenbobo' WHERE username='admin'
```

```
UPDATE users SET password='admin123', status=1 WHERE id=1
```

H3 DELETE 删除



```
DELETE FROM 表名 WHERE 条件
```

H3 WHERE 语句

用于匹配语句：



```
-- 在 person 表中匹配 age<20 的目标  
SELECT * FROM person WHERE age<20
```

注：

不等于的表示方法: <>

也可以使用 !=

④ AND / OR 联结词

AND 相当于 js 中的 &&

OR 相当于 js 中的 ||



```
select * from person where age < 20 and id>1  
select * from person where age < 20 or name=chen
```

⑤ ORDER BY 排序

升序排序:

对于 user 标准中的数据, 按照 status 进行升序排序



```
-- 一下这两条语句等价  
select * from users order by status  
select * from users order by status ASC
```

降序排序:

使用 desc 就是降序排序



```
select * from users order by id desc
```

多重排序:

按照状态进行降序排序, 再按照用户名首字母进行升序排序



```
select * from users order by status desc,username asc
```

H3 COUNT(*) 总条数

返回表中的总数:



```
select count(*) from users where status=0
```

| | count(*) |
|---|----------|
| ▶ | 2 |

使用 AS 关键字其别名



```
select count(*) as total from users where status=0
```

| | total |
|---|-------|
| ▶ | 2 |

项目中操作**MySQL**

H3 连接数据库



```
const mysql = require('mysql');
// 建立与 MySQL 的连接
const db = mysql.createPool({
    // 数据库的 ip 地址
    host: 'localhost',
    // 登录数据库的账号
    user: 'root',
    // 登录数据库的密码
    password: 'chen20020423',
    //操作哪个数据库
    database: 'my_db_01',
});
```

H3 查询数据库



```
const mysql = require('mysql');
// 建立与 MySQL 的连接
const db = mysql.createPool({
    // 数据库的 ip 地址
    host: 'localhost',
    // 登录数据库的账号
    user: 'root',
    // 登录数据库的密码
    password: 'chen20020423',
    //操作哪个数据库
    database: 'my_db_01',
});
// 在 query 中第一个参数可以设置为 sql 语句
```

输出：



```
[  
  RowDataPacket {  
    id: 1,  
    username: 'admin',  
    password: '123456',  
    status: 0  
  }  
]
```

③ 占位符



```
// ? 表示占位符  
const sqlStr = 'insert into users(username,password) values(?,?)';  
db.query(sqlStr, [user.username, user.password], (err, results) => {  
  if (err) return console.log('插入失败, 用户名已存在');  
  if (results.affectedRows === 1) console.log('插入数据成功');  
});
```

查询数据库返回的是一个数组；

插入、删除数据库返回的是一个对象，可以通过查询 `results.affectedRows === 1` 来判断是否插入成功。

④ 插入数据库快捷方式



```
// ? 表示占位符  
const user = {username:'liuneng',password:'123456'}  
const sqlStr = 'insert into users set ?';  
db.query(sqlStr,user, (err, results) => {  
  if (err) return console.log('插入失败, 用户名已存在');  
  if (results.affectedRows === 1) console.log('插入数据成功');  
});
```

H3 更新数据库快捷方式



```
const user = {id:7,username:'liuneng',password:'123456'}
const sqlStr = 'update users set ? where id=?';
db.query(sqlStr,user, (err, results) => {
    if (err) return console.log('插入失败, 用户名已存在');
    if (results.affectedRows === 1) console.log('插入数据成功');
});
```

H3 删 除语句



```
const user = {id:7,username:'liuneng',password:'123456'}
const sqlStr = 'delete from users where id=?';
db.query(sqlStr, 3, (err, results) => {
    if (err) return console.log('插入失败, 用户名已存在');
    if (results.affectedRows === 1) console.log('插入数据成功');
});
```

注意：很多时候当用户选择删除时我们并不会进行直接删除，我们会将用户的状态从 0 改为 1，这样就可以防止用户后期反悔，避免不必要的损失，而是执行 `update` 更新：



```
const user = {id:7,username:'liuneng',password:'123456'}
const sqlStr = 'update users set status=? where id=?';
db.query(sqlStr, [1,6], (err, results) => {
    if (err) return console.log('g');
    if (results.affectedRows === 1) console.log('插入数据成功');
});
```

身份认证

H3 Session

不支持跨域请求

01. 导入中间件



```
const session = requier('express-session');
```

02. 配置中间件



```
app.use(  
    session({  
        secret: 'chenbobo',  
        resave: false,  
        saveUninitialized: true,  
    })  
);
```

03. 储存session



```
app.post('/api/login', (req, res) => {  
    // 判断用户提交的登录信息  
    if (req.body.username !== 'admin' || req.body.password !== '123456')  
    {  
        return res.send({ status: 1, message: '登陆失败!' });  
    }  
    // 将登录成功的用户信息储存在 session 中  
    req.session.user = req.body;  
    res.session.islogin = true;  
  
    res.send({ status: 0, message: '登录成功' });  
});
```

04. 读取session



```
// 获取用户姓名的接口
app.get('/api/username', (req, res) => {
  if (!req.session.islogin) {
    return res.send({ status: 1, message: '登陆失败!' });
  }
  res.send({ status: 0, message: '登陆成功!' });
});
```

05. 销毁 session



```
app.get('/api.outlogin', (req, res) => {
  req.session.destroy();
  res.send({
    status: 0,
    message: '清空成功!',
  });
});
```

实例：



```
const express = require('express');
const app = express();
const session = require('express-session');
// 配置 session 中间件
app.use(
  session({
    secret: 'chenbobo',
    resave: false,
    saveUninitialized: true,
  })
);

// 托管静态页面
app.use(express.static('./pages'));

// 解析 POST 提交过来的表单数据
app.use(express.urlencoded({ extended: false }));
```

```
// 登录的 API 接口
app.post('/api/login', (req, res) => {
    // 判断用户提交的登录信息
    if (req.body.username !== 'admin' || req.body.password !== '123456') {
        return res.send({ status: 1, message: '登陆失败!' });
    }
    // 将登录成功的用户信息储存在 session 中
    req.session.user = req.body;
    res.session.islogin = true;

    res.send({ status: 0, message: '登录成功!' });
});

// 获取用户名的接口
app.get('/api/username', (req, res) => {
    if (!req.session.islogin) {
        return res.send({ status: 1, message: '登陆失败!' });
    }
    res.send({ status: 0, message: '登陆成功!' });
});

// 清空 session 数据
app.get('/api.outlogin', (req, res) => {
    req.session.destroy();
    res.send({
        status: 0,
        message: '清空成功!',
    });
});

app.listen(8080, () => {
    console.log('Server running at http://127.0.0.1:8080');
});
```

③ JWT

支持跨域请求

01. 安装并导入 JWT 相关的两个包，分别是

jsonwebtoken：加密token 和 express-jwt：解密token



```
const jwt = require('jsonwebtoken')
const expressJWT = require('express-jwt')
```

02. 允许跨域资源共享



```
// 允许跨域资源共享
const cors = require('cors')
```

03. 定义 secret 密钥，建议将密钥命名为 secretKey



```
const secretKey = 'chenbobo No1 ^_^'
```

04. 注册将 JWT 字符串解析还原成 JSON 对象的中间件

注意：只要配置成功了 express-jwt 这个中间件，就可以把解析出来的用户信息，挂载到 req.user 属性上



```
// 除了路径为此正则表达式的路径不需要解密
app.use(expressJWT({ secret: secretKey }).unless({ path: [/^\/api\//] }))
```

05. 在登录成功之后，调用 jwt.sign() 方法生成 JWT 字符串。并通过 token 属性发送给客户端

```
● ● ●  
// 参数1: 用户的信息对象  
// 参数2: 加密的秘钥  
// 参数3: 配置对象, 可以配置当前 token 的有效期  
// 记住: 千万不要把密码加密到 token 字符中  
const tokenStr = jwt.sign({ username: userinfo.username }, secretKey, {  
  expiresIn: '30s'  
})  
res.send({  
  status: 200,  
  message: '登录成功!',  
  token: tokenStr, // 要发送给客户端的 token 字符串  
})
```

06. 向用户端发送用户信息

```
● ● ●  
app.get('/admin/getinfo', function (req, res) {  
  // TODO_05: 使用 req.user 获取用户信息, 并使用 data 属性将用户信息发送给客户  
  // 端  
  console.log(req.user)  
  res.send({  
    status: 200,  
    message: '获取用户信息成功!',  
    data: req.user, // 要发送给客户端的用户信息  
  })  
})
```

07. 使用全局错误处理中间件, 捕获解析 JWT 失败后产生的错误



```
app.use((err, req, res, next) => {
    // 这次错误是由 token 解析失败导致的
    if (err.name === 'UnauthorizedError') {
        return res.send({
            status: 401,
            message: '无效的token',
        })
    }
    res.send({
        status: 500,
        message: '未知的错误',
    })
})
```

实例：



```
// 导入 express 模块
const express = require('express')
// 创建 express 的服务器实例
const app = express()

// TODO_01: 安装并导入 JWT 相关的两个包，分别是 jsonwebtoken 和 express-jwt
const jwt = require('jsonwebtoken')
const expressJWT = require('express-jwt')

// 允许跨域资源共享
const cors = require('cors')
app.use(cors())

// 解析 post 表单数据的中间件
const bodyParser = require('body-parser')
app.use(bodyParser.urlencoded({ extended: false }))

// TODO_02: 定义 secret 密钥，建议将密钥命名为 secretKey
const secretKey = 'itheima No1 ^_^'

// TODO_04: 注册将 JWT 字符串解析还原成 JSON 对象的中间件
// 注意：只要配置成功了 express-jwt 这个中间件，就可以把解析出来的用户信息，挂载到
// req.user 属性上
app.use(expressJWT({ secret: secretKey }).unless({ path: [/^\/api\//] }))
```

```
// 登录接口
app.post('/api/login', function (req, res) {
    // 将 req.body 请求体中的数据，转存为 userinfo 常量
    const userinfo = req.body
    // 登录失败
    if (userinfo.username !== 'admin' || userinfo.password !== '000000') {
        return res.send({
            status: 400,
            message: '登录失败!',
        })
    }
    // 登录成功
    // TODO_03: 在登录成功之后，调用 jwt.sign() 方法生成 JWT 字符串。并通过 token 属性发送给客户端
    // 参数1：用户的信息对象
    // 参数2：加密的秘钥
    // 参数3：配置对象，可以配置当前 token 的有效期
    // 记住：千万不要把密码加密到 token 字符中
    const tokenStr = jwt.sign({ username: userinfo.username },
secretKey, { expiresIn: '30s' })
    res.send({
        status: 200,
        message: '登录成功!',
        token: tokenStr, // 要发送给客户端的 token 字符串
    })
})

// 这是一个有权限的 API 接口
app.get('/admin/getinfo', function (req, res) {
    // TODO_05: 使用 req.user 获取用户信息，并使用 data 属性将用户信息发送给客户端
    console.log(req.user)
    res.send({
        status: 200,
        message: '获取用户信息成功!',
        data: req.user, // 要发送给客户端的用户信息
    })
})

// TODO_06: 使用全局错误处理中间件，捕获解析 JWT 失败后产生的错误
app.use((err, req, res, next) => {
    // 这次错误是由 token 解析失败导致的
    if (err.name === 'UnauthorizedError') {
        return res.send({
            status: 401,
            message: '无效的token',
        })
    }
})
```

```
        }

        res.send({
            status: 500,
            message: '未知的错误',
        })
    }

// 调用 app.listen 方法，指定端口号并启动web服务器
app.listen(8888, function () {
    console.log('Express server running at http://127.0.0.1:8888')
})
```