在使用 Node.js 编写一个完整的项目时，程序中往往需要用到一些可配置的变量，从而使得程序能在不同的环境中运行。本文将介绍几种常见的方法。

1）通过环境变量指定配置

环境变量（environment variables） 一般是指在操作系统中用来指定操作系统运行环境的一些参数，如：临时文件夹位置和系统文件夹位置等。比如HOME表示当前用户的根目录，TMPDIR表示系统临时目录等，我们可以通过设置一些特定的环境变量，程序在启动时可以读取这些环境变量并做相应的初始化动作。

在 Node.js 中可以通过process.env来访问当前的环境变量信息，比如：

```
{ PATH: '/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin',
    TMPDIR: '/var/folders/rs/g4wqpvvj7bj08t35dxvfm0rr0000gn/T/',
    LOGNAME: 'glen',
    XPC_FLAGS: '0x0',
    HOME: '/Users/glen',
    TERM: 'xterm-256color',
    COLORFGBG: '7;0',
    USER: 'glen',
    ITERM_PROFILE: 'Glen',
    TERM_PROGRAM: 'iTerm.app',
    XPC_SERVICE_NAME: '0',
    SHELL: '/bin/zsh',
    ITERM_SESSION_ID: 'w0t4p0',
    PWD: '/Users/glen/work',
    __CF_USER_TEXT_ENCODING: '0x1F5:0x0:0x0',
    LC_CTYPE: 'UTF-8',
    SHLVL: '1',
    OLDPWD: '/Users/glen/work',
    ZSH: '/Users/glen/.oh-my-zsh',
    PAGER: 'less',
    LESS: '-R',
    LSCOLORS: 'Gxfxcxdxbxegedabagacad',
    AUTOJUMP_SOURCED: '1',
    AUTOJUMP_ERROR_PATH: '/Users/glen/Library/autojump/errors.log',
    RUST_SRC_PATH: '/Users/glen/work/source/rust/src',
    _: '/usr/local/bin/node'
}
```

设置环境变量

环境变量的名字一般为大写，多个单词之间可通过下划线来连接。

Windows 系统下可通过set命令来设置环境变量，比如：

```
$ set HELLO_MSG="Hello, world!"
```

Linux 系统下可通过export命令来设置，比如：

```
$ export HELLO_MSG="Hello, world!"
```

在 Node.js 中读取环境变量

创建文件1.js，代码如下：

```
console.log(process.env.HELLO_MSG);
```

然后在命令行中执行：

```
$ export HELLO_MSG="Hello, world" && node 1.js
```

控制台将输出Hello, world，即我们启动程序时给环境变量HELLO_MSG设置的值。

2）通过配置文件指定配置

一些规模较小的项目往往会通过单一的配置文件来存储其配置，比如 CNode 中文社区的开源项目 nodeclub 在启动时会载入文件config.js，该文件的大概结构如下：

```
var config = {
    // debug 为 true 时，用于本地调试
    debug: true,


    name: 'Nodeclub', // 社区名字
    description: 'CNode：Node.js专业中文社区', // 社区的描述
    keywords: 'nodejs, node, express, connect, socket.io',
```

```
    // 其他配置项...
};
module.exports = config;
```

在程序启动的时候，可以使用require()来载入此文件，得到一个对象，然后通过此对象的属性来读取相应的配置信息：

```
// 载入配置文件
var config = require('./config');

// 以下为使用到配置的部分代码：
if (!config.debug && config.oneapm_key) {
    require('oneapm');
}

app.use(session({
    secret: config.session_secret,
    store: new RedisStore({
        port: config.redis_port,
        host: config.redis_host,
    }),
    resave: true,
    saveUninitialized: true,
}))

app.listen(config.port, function () {
    logger.log('NodeClub listening on port', config.port);
    logger.log('God bless love....');
    logger.log('You can debug your app with http://' + config.hostname + ':' + config.port);
    logger.log('');
});
```

使用配置文件与使用环境变量来指定配置相比，配置文件的可读性更强，可以表示一些更复杂的结构，而使用环境变量一般只限于key=value的形式。但在配置项数量较少时，使用环境变量会更简单，比如项目中只需要配置一个监听端口，可以简单使用export PORT=3000 && node app.js命令来启动程序，而不需要单独创建一个配置文件。大多数时候往往会结合这两种方式来进行，下文讲详细讲解。

其他配置文件格式

一般为了方便，在 Node.js 项目中会习惯使用.js文件格式，它的好处是可以使用通过程序来动态生成一些配置项，比如 nodeclub 的其中一个配置项：

```
var config = {
    // 文件上传配置
    // 注：如果填写 qn_access，则会上传到 7牛，以下配置无效
    upload: {
        path: path.join(__dirname, 'public/upload/'),
        url: '/public/upload/'
    },
}
```

其中使用到了path.join()和__dirname来生成upload.path。

JSON格式

另外，我们也可以使用 JSON 格式的配置文件，比如文件config.json：

```
{
    "debug": true,
    "name": "Nodeclub",
    "description": "CNode：Node.js专业中文社区",
    "keywords": "nodejs, node, express, connect, socket.io"
}
```

在程序中可以通过以下方式来载入JSON文件配置：

```
// 通过require()函数
var config = require('./config.json');

// 读取文件并使用JSON.parse()解析
var fs = require('fs');
var config = JSON.parse(fs.readFileSync('./config.json').toString());
```

大多数时候，我们往往需要添加一些备注信息来说明某个配置项的使用方法及用途，在标准JSON文件中是不允许添加备注的，我们可以使用strip-json-comments模块来去掉配置文件中的备注，再将其当作标准的JSON来解析。

比如以下是带备注信息的JSON配置文件：

```
{
    // debug 为 true 时，用于本地调试
    "debug": true,
    // 社区名字
    "name": "Nodeclub",
    // 社区的描述
    "description": "CNode：Node.js专业中文社区",
    "keywords": "nodejs, node, express, connect, socket.io"
}
```

我们可以编写一个loadJSONFile()函数来载入带有备注的JSON文件：

```
var fs = require('fs');
var stripJsonComments = require('strip-json-comments');

function loadJSONFile (file) {
    var json = fs.readFileSync(file).toString();
    return JSON.parse(stripJsonComments(json));
}

var config = loadJSONFile('./config.json');
console.log(config);
```

YAML格式

YAML 是面向所有编程语言的对人类友好的数据序列化标准。其最大的优点是可读性较好，比如以下 YAML 格式的配置：

```
name: John Smith
age: 37
spouse:
    name: Jane Smith
    age: 25
children:
    - name: Jimmy Smith
        age: 15
    - name: Jenny Smith
        age: 12
```

其对应的JSON结构如下：

```
{
    "age": 37,
    "spouse": {
        "age": 25,
        "name": "Jane Smith"
    },
    "name": "John Smith",
    "children": [
        {
            "age": 15,
            "name": "Jimmy Smith"
        },
        {
            "age": 12,
            "name": "Jenny Smith"
        }
    ]
}
```

在 Node.js 中可以通过yamljs模块来解析 YAML 格式，比如可以编写一个loadYAMLFile()函数来载入 YAML 格式的配置文件：

```
var fs = require('fs');
var YAML = require('yamljs');

function loadYAMLFile (file) {
    return YAML.parse(fs.readFileSync(file).toString());
}

var config = loadYAMLFile('./config.yaml');
console.log(config);
```

根据运行环境选择不同的配置

大多数情况下，程序在本地开发环境和生产环境中的配置信息是不一样的，比如开发时连接到的数据库里面的数据是模拟出来的，而生产环境要连接到实际的数据库上，因此我们需要

让程序能根据不同的运行环境来载入不同的配置文件。

使用单一配置文件名

以 nodeclub 项目为例，其载入的配置文件名为./config.js，项目中有一个默认配置文件./config.default.js。要运行程序，首先需要复制一份默认配置文件，并保存为./config.js，再根据当前运行环境来修改./config.js。

由于./config.js文件已经被添加到.gitignore文件中，因此我们./config.js文件的修改不会被纳入到项目的版本管理中，所以不同机器中的./config.js不会产生冲突，可以使用各自的配置来启动程序。

通过环境变量指定配置文件名

我们可以通过环境变量来指定配置文件，比如：

```
$ export CONFIG_FILE="./config/production.js"
```

然后可以通过以下方式来载入配置文件：

```
var path = require('path');
var config = require(path.resolve(process.env.CONFIG_FILE));
```

另外，也可以通过环境变量来指定当前运行环境的名称，然后在指定目录下载入相应的配置，比如：

```
$ export NODE_ENV="production"
```

然后可以通过以下方式来载入配置文件：

```
var path = require('path');
var configFile = path.resolve('./config', process.env.NODE_ENV + '.js');
var config = require(configFile);
```

使用 config 模块来读取配置

config 模块是 NPM 上下载量最高的 Node.js 配置文件管理模块，其实现原理与上文中介绍的方法大同小异，在实际开发中我们可以考虑使用这个现成的模块。下面将介绍此模块的简单使用方法。

config模块通过环境变量NODE_CONFIG_DIR来指定配置文件所在的目录，默认为./config（即当前运行目录下的config目录），通过环境变量NODE_ENV来指定当前的运行环境版本。

配置文件使用 JSON 格式，模块加载后，会首先载入默认的配置文件${NODE_CONFIG_DIR}/default.json，再载入文件${NODE_CONFIG_DIR}/${NODE_ENV}.json，如果配置项有冲突则覆盖默认的配置。

比如我们新建默认配置文件config/default.json：

```
{
```

```
    // Customer module configs
    "Customer": {
        "dbConfig": {
            "host": "localhost",
            "port": 5984,
            "dbName": "customers"
        },
        "credit": {
            "initialLimit": 100,
            // Set low for development
            "initialDays": 1
        }
    }
}
```

再新建production环境配置文件config/production.json：

```
{
    "Customer": {
        "dbConfig": {
            "host": "prod-db-server"
        },
        "credit": {
            "initialDays": 30
        }
    }
}
```

再新建测试文件1.js：

```
var config = require('config');
console.log(config);
```

执行程序，可看到其输出的结果为默认的配置：

```
{ Customer:
      { dbConfig: { host: 'localhost', port: 5984, dbName: 'customers' },
          credit: { initialLimit: 100, initialDays: 1 } } }
```

假如要使用production的配置，则使用以下命令启动：

```
$ export NODE_ENV=production && node 1.js
```

则其输出将是如下结果：

```
{ Customer:
      { dbConfig: { host: 'prod-db-server', port: 5984, dbName: 'customers' },
          credit: { initialLimit: 100, initialDays: 30 } } }
```

在production.json文件中，重新定义了Customer.dbConfig.host和Customer.credit.initialDays这两个配置项，所以在production环境中仅这两项被覆盖为

新的值，而其他配置项则使用default.json中指定的值。

载入config模块后，其返回的对象实际上就是当前的配置信息，同时提供了两个方法get()和has()来操作配置项。比如：

```
var config = require('config');
console.log(config);
console.log(config.get('Customer'));
console.log(config.get('Customer.dbConfig'));
console.log(config.has('Customer.dbConfig.host'));
console.log(config.has('Customer.dbConfig.host2'));
```

执行程序后输出结果如下：

```
{ Customer:
      { dbConfig: { host: 'localhost', port: 5984, dbName: 'customers' },
          credit: { initialLimit: 100, initialDays: 1 } } }
{ dbConfig: { host: 'localhost', port: 5984, dbName: 'customers' },
    credit: { initialLimit: 100, initialDays: 1 } }
{ host: 'localhost', port: 5984, dbName: 'customers' }
true
false
```

其中get()用来获取指定配置，可以使用诸如Customer.dbConfig这样的格式，如果配置项不存在则会抛出异常。has()用来检测指定配置项是否存在，如果存在则返回true。