七天学会NodeJS

NodeJS基础

什么是NodeJS

JS是脚本语言，脚本语言都需要一个解析器才能运行。对于写在HTML页面里的JS，浏览器充当了解析器的角色。而对于需要独立运行的JS，NodeJS就是一个解析器。

每一种解析器都是一个运行环境，不但允许JS定义各种数据结构，进行各种计算，还允许JS使用运行环境提供的内置对象和方法做一些事情。例如运行在浏览器中的JS的用途是操作DOM，浏览器就提供了`document`之类的内置对象。而运行在NodeJS中的JS的用途是操作磁盘文件或搭建HTTP服务器，NodeJS就相应提供了`fs`、`http`等内置对象。

有啥用处

尽管存在一听说可以直接运行JS文件就觉得很酷的同学，但大多数同学在接触新东西时首先关心的是有啥用处，以及能带来啥价值。

NodeJS的作者说，他创造NodeJS的目的是为了实现高性能Web服务器，他首先看重的是事件机制和异步IO模型的优越性，而不是JS。但是他需要选择一种编程语言实现他的想法，这种编程语言不能自带IO功能，并且需要能良好支持事件机制。JS没有自带IO功能，天生就用于处理浏览器中的DOM事件，并且拥有一大群程序员，因此就成为了天然的选择。

如他所愿，NodeJS在服务端活跃起来，出现了大批基于NodeJS的Web服务。而另一方面，NodeJS让前端众如获神器，终于可以让自己的能力覆盖范围跳出浏览器窗口，更大批的前端工具如雨后春笋。

因此，对于前端而言，虽然不是人人都要拿NodeJS写一个服务器程序，但简单可至使用命令交互模式调试JS代码片段，复杂可至编写工具提升工作效率。

NodeJS生态圈正欣欣向荣。

如何安装

安装程序

NodeJS提供了一些安装程序，都可以在nodejs.org这里下载并安装。

Windows系统下，选择和系统版本匹配的`.msi`后缀的安装文件。Mac OS X系统下，选择`.pkg`后缀的安装文件。

编译安装

Linux系统下没有现成的安装程序可用，虽然一些发行版可以使用`apt-get`之类的方式安装，但不一定能安装到最新版。因此Linux系统下一般使用以下方式编译方式安装NodeJS。

1. 确保系统下g++版本在4.6以上，python版本在2.6以上。
2. 从nodejs.org下载`tar.gz`后缀的NodeJS最新版源代码包并解压到某个位置。
3. 进入解压到的目录，使用以下命令编译和安装。

```
$ ./configure
$ make
$ sudo make install
```

如何运行

打开终端，键入`node`进入命令交互模式，可以输入一条代码语句后立即执行并显示结果，例如：

```
$ node
> console.log('Hello World!');
Hello World!
```

如果要运行一大段代码的话，可以先写一个JS文件再运行。例如有以下`hello.js`。

```
function hello() {
    console.log('Hello World!');
}
hello();
```

写好后在终端下键入`node hello.js`运行，结果如下：

```
$ node hello.js
Hello World!
```

权限问题

在Linux系统下，使用NodeJS监听80或443端口提供HTTP(S)服务时需要root权限，有两种方式可以做到。

一种方式是使用`sudo`命令运行NodeJS。例如通过以下命令运行的`server.js`中有权限使用80和443端口。一般推荐这种方式，可以保证仅为有需要的JS脚本提供root权限。

```
$ sudo node server.js
```

另一种方式是使用`chmod +s`命令让NodeJS总是以root权限运行，具体做法如下。因为这种方式让任何JS脚本都有了root权限，不太安全，因此在需要很考虑安全的系统下不推荐使用。

```
$ sudo chown root /usr/local/bin/node
$ sudo chmod +s /usr/local/bin/node
```

模块

编写稍大一点的程序时一般都会将代码模块化。在NodeJS中，一般将代码合理拆分到不同的JS文件中，每一个文件就是一个模块，而文件路径就是模块名。

在编写每个模块时，都有`require`、`exports`、`module`三个预先定义好的变量可供使用。

require

`require`函数用于在当前模块中加载和使用别的模块，传入一个模块名，返回一个模块导出对象。模块名可使用相对路径（以`./`开头），或者是绝对路径（以`/`或`C:`之类的盘符开头）。另外，模块名中的`.js`扩展名可以省略。以下是一个例子。

```
var foo1 = require('./foo');
var foo2 = require('./foo.js');
var foo3 = require('/home/user/foo');
var foo4 = require('/home/user/foo.js');

// foo1至foo4中保存的是同一个模块的导出对象。
```

另外，可以使用以下方式加载和使用一个JSON文件。

```
var data = require('./data.json');
```

exports

`exports`对象是当前模块的导出对象，用于导出模块公有方法和属性。别的模块通过`require`函数使用当前模块时得到的就是当前模块的`exports`对象。以下例子中导出了一个公有方法。

```
exports.hello = function () {
    console.log('Hello World!');
};
```

module

通过`module`对象可以访问到当前模块的一些相关信息，但最多的用途是替换当前模块的导出对象。例如模块导出对象默认是一个普通对象，如果想改成一个函数的话，可以使用以下方式。

```
module.exports = function () {
    console.log('Hello World!');
};
```

以上代码中，模块默认导出对象被替换为一个函数。

模块初始化

一个模块中的JS代码仅在模块第一次被使用时执行一次，并在执行过程中初始化模块的导出对象。之后，缓存起来的导出对象被重复利用。

主模块

通过命令行参数传递给NodeJS以启动程序的模块被称为主模块。主模块负责调度组成整个程序的其它模块完成工作。例如通过以下命令启动程序时，`main.js`就是主模块。

```
$ node main.js
```

完整示例

例如有以下目录。

```
- /home/user/hello/
    - util/
        counter.js
    main.js
```

其中`counter.js`内容如下：

```
var i = 0;

function count() {
    return ++i;
}

exports.count = count;
```

该模块内部定义了一个私有变量`i`，并在`exports`对象导出了一个公有方法`count`。

主模块`main.js`内容如下：

```
var counter1 = require('./util/counter');
var    counter2 = require('./util/counter');

console.log(counter1.count());
```

```
console.log(counter2.count());
console.log(counter2.count());
```

运行该程序的结果如下：

```
$ node main.js
1
2
3
```

可以看到，`counter.js`并没有因为被require了两次而初始化两次。

二进制模块

虽然一般我们使用JS编写模块，但NodeJS也支持使用C/C++编写二进制模块。编译好的二进制模块除了文件扩展名是`.node`外，和JS模块的使用方式相同。虽然二进制模块能使用操作系统提供的所有功能，拥有无限的潜能，但对于前端同学而言编写过于困难，并且难以跨平台使用，因此不在本教程的覆盖范围内。

小结

本章介绍了有关NodeJS的基本概念和使用方法，总结起来有以下知识点：

- NodeJS是一个JS脚本解析器，任何操作系统下安装NodeJS本质上做的事情都是把NodeJS执行程序复制到一个目录，然后保证这个目录在系统PATH环境变量下，以便终端下可以使用`node`命令。
- 终端下直接输入`node`命令可进入命令交互模式，很适合用来测试一些JS代码片段，比如正则表达式。
- NodeJS使用CMD模块系统，主模块作为程序入口点，所有模块在执行过程中只初始化一次。
- 除非JS模块不能满足需求，否则不要轻易使用二进制模块，否则你的用户会叫苦连天。

代码的组织和部署

有经验的C程序员在编写一个新程序时首先从make文件写起。同样的，使用NodeJS编写程序前，为了有个良好的开端，首先需要准备好代码的目录结构和部署方式，就如同修房子要先搭脚手架。本章将介绍与之相关的各种知识。

模块路径解析规则

我们已经知道，`require`函数支持斜杠（`/`）或盘符（`C:`）开头的绝对路径，也支持`./`开头的相对路径。但这两种路径在模块之间建立了强耦合关系，一旦某个模块文件的存放位置需要变更，使用该模块的其它模块的代码也需要跟着调整，变得牵一发动全身。因此，`require`函数支持第三种形式的路径，写法类似于`foo/bar`，并依次按照以下规则解析路径，直到找到模块位置。

1. 内置模块

   如果传递给`require`函数的是NodeJS内置模块名称，不做路径解析，直接返回内部模块的导出对象，例如`require('fs')`。

2. node_modules目录

   NodeJS定义了一个特殊的`node_modules`目录用于存放模块。例如某个模块的绝对路径是`/home/user/hello.js`，在该模块中使用`require('foo/bar')`方式加载模块时，则NodeJS依次尝试使用以下路径。

   ```
   /home/user/node_modules/foo/bar
   /home/node_modules/foo/bar
   /node_modules/foo/bar
   ```

3. NODE_PATH环境变量

   与PATH环境变量类似，NodeJS允许通过NODE_PATH环境变量来指定额外的模块搜索路径。NODE_PATH环境变量中包含一到多个目录路径，路径之间在Linux下使用`:`分隔，在Windows下使用`;`分隔。例如定义了以下NODE_PATH环境变量：

   ```
   NODE_PATH=/home/user/lib:/home/lib
   ```

   当使用`require('foo/bar')`的方式加载模块时，则NodeJS依次尝试以下路径。

   ```
   /home/user/lib/foo/bar
   /home/lib/foo/bar
   ```

包（package）

我们已经知道了JS模块的基本单位是单个JS文件，但复杂些的模块往往由多个子模块组成。为了便于管理和使用，我们可以把由多个子模块组成的大模块称做`包`，并把所有子模块放在同一个目录里。

在组成一个包的所有子模块中，需要有一个入口模块，入口模块的导出对象被作为包的导出对象。例如有以下目录结构。

```
- /home/user/lib/
    - cat/
        head.js
        body.js
        main.js
```

其中`cat`目录定义了一个包，其中包含了3个子模块。`main.js`作为入口模块，其内容如下：

```
var head = require('./head');
var body = require('./body');

exports.create = function (name) {
    return {
        name: name,
        head: head.create(),
        body: body.create()
    };
};
```

在其它模块里使用包的时候，需要加载包的入口模块。接着上例，使用`require('/home/user/lib/cat/main')`能达到目的，但是入口模块名称出现在路径里看上去不是个好主意。因此我们需要做点额外的工作，让包使用起来更像是单个模块。

index.js

当模块的文件名是`index.js`，加载模块时可以使用模块所在目录的路径代替模块文件路径，因此接着上例，以下两条语句等价。

```
var cat = require('/home/user/lib/cat');
var cat = require('/home/user/lib/cat/index');
```

这样处理后，就只需要把包目录路径传递给`require`函数，感觉上整个目录被当作单个模块使用，更有整体感。

package.json

如果想自定义入口模块的文件名和存放位置，就需要在包目录下包含一个`package.json`文件，并在其中指定入口模块的路径。上例中的`cat`模块可以重构如下。

```
- /home/user/lib/
    - cat/
        + doc/
        - lib/
```

```
            head.js
            body.js
            main.js
        + tests/
        package.json
```

其中`package.json`内容如下。

```
{
    "name": "cat",
    "main": "./lib/main.js"
}
```

如此一来，就同样可以使用`require('/home/user/lib/cat')`的方式加载模块。NodeJS会根据包目录下的`package.json`找到入口模块所在位置。

命令行程序

使用NodeJS编写的东西，要么是一个包，要么是一个命令行程序，而前者最终也会用于开发后者。因此我们在部署代码时需要一些技巧，让用户觉得自己是在使用一个命令行程序。

例如我们用NodeJS写了个程序，可以把命令行参数原样打印出来。该程序很简单，在主模块内实现了所有功能。并且写好后，我们把该程序部署在`/home/user/bin/node-echo.js`这个位置。为了在任何目录下都能运行该程序，我们需要使用以下终端命令。

```
$ node /home/user/bin/node-echo.js Hello World
Hello World
```

这种使用方式看起来不怎么像是一个命令行程序，下边的才是我们期望的方式。

```
$ node-echo Hello World
```

Linux

在Linux系统下，我们可以把JS文件当作shell脚本来运行，从而达到上述目的，具体步骤如下：

1. 在shell脚本中，可以通过`#!`注释来指定当前脚本使用的解析器。所以我们首先在`node-echo.js`文件顶部增加以下一行注释，表明当前脚本使用NodeJS解析。

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

   NodeJS会忽略掉位于JS模块首行的`#!`注释，不必担心这行注释是非法语句。

2. 然后，我们使用以下命令赋予`node-echo.js`文件执行权限。

```
$ chmod +x /home/user/bin/node-echo.js
```

3. 最后，我们在PATH环境变量中指定的某个目录下，例如在`/usr/local/bin`下边创建一个软链文件，文件名与我们希望使用的终端命令同名，命令如下：

```
$ sudo ln -s /home/user/bin/node-echo.js /usr/local/bin/node-echo
```

这样处理后，我们就可以在任何目录下使用`node-echo`命令了。

Windows

在Windows系统下的做法完全不同，我们得靠`.cmd`文件来解决问题。假设`node-echo.js`存放在`C:\Users\user\bin`目录，并且该目录已经添加到PATH环境变量里了。接下来需要在该目录下新建一个名为`node-echo.cmd`的文件，文件内容如下：

```
@node "C:\User\user\bin\node-echo.js" %*
```

这样处理后，我们就可以在任何目录下使用`node-echo`命令了。

工程目录

了解了以上知识后，现在我们可以来完整地规划一个工程目录了。以编写一个命令行程序为例，一般我们会同时提供命令行模式和API模式两种使用方式，并且我们会借助三方包来编写代码。除了代码外，一个完整的程序也应该有自己的文档和测试用例。因此，一个标准的工程目录都看起来像下边这样。

```
- /home/user/workspace/node-echo/   # 工程目录
    - bin/                          # 存放命令行相关代码
        node-echo
    + doc/                          # 存放文档
    - lib/                          # 存放API相关代码
        echo.js
    - node_modules/                 # 存放三方包
        + argv/
    + tests/                        # 存放测试用例
    package.json                    # 元数据文件
    README.md                       # 说明文件
```

其中部分文件内容如下：

```
/* bin/node-echo */
var argv = require('argv'),
    echo = require('../lib/echo');
```

```
console.log(echo(argv.join(' ')));


/* lib/echo.js */
module.exports = function (message) {
    return message;
};


/* package.json */
{
    "name": "node-echo",
    "main": "./lib/echo.js"
}
```

以上例子中分类存放了不同类型的文件，并通过`node_moudles`目录直接使用三方包名加载模块。此外，定义了`package.json`之后，`node-echo`目录也可被当作一个包来使用。

NPM

NPM是随同NodeJS一起安装的包管理工具，能解决NodeJS代码部署上的很多问题，常见的使用场景有以下几种：

- 允许用户从NPM服务器下载别人编写的三方包到本地使用。
- 允许用户从NPM服务器下载并安装别人编写的命令行程序到本地使用。
- 允许用户将自己编写的包或命令行程序上传到NPM服务器供别人使用。

可以看到，NPM建立了一个NodeJS生态圈，NodeJS开发者和用户可以在里边互通有无。以下分别介绍这三种场景下怎样使用NPM。

下载三方包

需要使用三方包时，首先得知道有哪些包可用。虽然npmjs.org提供了个搜索框可以根据包名来搜索，但如果连想使用的三方包的名字都不确定的话，就请百度一下吧。知道了包名后，比如上边例子中的`argv`，就可以在工程目录下打开终端，使用以下命令来下载三方包。

```
$ npm install argv
...
argv@0.0.2 node_modules\argv
```

下载好之后，`argv`包就放在了工程目录下的`node_modules`目录中，因此在代码中只需要通过`require('argv')`的方式就好，无需指定三方包路径。

以上命令默认下载最新版三方包，如果想要下载指定版本的话，可以在包名后边加上`@<version>`，例如通过以下命令可下载0.0.1版的`argv`。

```
$ npm install argv@0.0.1
...
argv@0.0.1 node_modules\argv
```

如果使用到的三方包比较多，在终端下一个包一条命令地安装未免太人肉了。因此NPM对`package.json`的字段做了扩展，允许在其中申明三方包依赖。因此，上边例子中的`package.json`可以改写如下：

```
{
    "name": "node-echo",
    "main": "./lib/echo.js",
    "dependencies": {
        "argv": "0.0.2"
    }
}
```

这样处理后，在工程目录下就可以使用`npm install`命令批量安装三方包了。更重要的是，当以后`node-echo`也上传到了NPM服务器，别人下载这个包时，NPM会根据包中申明的三方包依赖自动下载进一步依赖的三方包。例如，使用`npm install node-echo`命令时，NPM会自动创建以下目录结构。

```
- project/
    - node_modules/
        - node-echo/
            - node_modules/
                + argv/
            ...
    ...
```

如此一来，用户只需关心自己直接使用的三方包，不需要自己去解决所有包的依赖关系。

安装命令行程序

从NPM服务上下载安装一个命令行程序的方法与三方包类似。例如上例中的`node-echo`提供了命令行使用方式，只要`node-echo`自己配置好了相关的`package.json`字段，对于用户而言，只需要使用以下命令安装程序。

```
$ npm install node-echo -g
```

参数中的`-g`表示全局安装，因此`node-echo`会默认安装到以下位置，并且NPM会自动创建好Linux系统下需要的软链文件或Windows系统下需要的`.cmd`文件。

```
- /usr/local/                # Linux系统下
    - lib/node_modules/
        + node-echo/
        ...
    - bin/
        node-echo
        ...
    ...

- %APPDATA%\npm\             # Windows系统下
    - node_modules\
        + node-echo\
        ...
    node-echo.cmd
    ...
```

发布代码

第一次使用NPM发布代码前需要注册一个账号。终端下运行`npm adduser`，之后按照提示做即可。账号搞定后，接着我们需要编辑`package.json`文件，加入NPM必需的字段。接着上边`node-echo`的例子，`package.json`里必要的字段如下。

```
{
    "name": "node-echo",           # 包名，在NPM服务器上须要保持唯一
    "version": "1.0.0",            # 当前版本号
    "dependencies": {              # 三方包依赖，需要指定包名和版本号
        "argv": "0.0.2"
      },
    "main": "./lib/echo.js",       # 入口模块位置
    "bin" : {
        "node-echo": "./bin/node-echo"      # 命令行程序名和主模块位置
    }
}
```

之后，我们就可以在`package.json`所在目录下运行`npm publish`发布代码了。

版本号

使用NPM下载和发布代码时都会接触到版本号。NPM使用语义版本号来管理代码，这里简单介绍一下。

语义版本号分为`X.Y.Z`三位，分别代表主版本号、次版本号和补丁版本号。当代码变更时，版本号按以下原则更新。

```
+ 如果只是修复bug，需要更新Z位。

+ 如果是新增了功能，但是向下兼容，需要更新Y位。

+ 如果有大变动，向下不兼容，需要更新X位。
```

版本号有了这个保证后，在申明三方包依赖时，除了可依赖于一个固定版本号外，还可依赖于某个范围的版本号。例如`"argv": "0.0.x"`表示依赖于`0.0.x`系列的最新版`argv`。NPM支持的所有版本号范围指定方式可以查看官方文档。

灵机一点

除了本章介绍的部分外，NPM还提供了很多功能，`package.json`里也有很多其它有用的字段。除了可以在npmjs.org/doc/查看官方文档外，这里再介绍一些NPM常用命令。

- NPM提供了很多命令，例如`install`和`publish`，使用`npm help`可查看所有命令。
- 使用`npm help <command>`可查看某条命令的详细帮助，例如`npm help install`。
- 在`package.json`所在目录下使用`npm install . -g`可先在本地安装当前命令行程序，可用于发布前的本地测试。
- 使用`npm update <package>`可以把当前目录下`node_modules`子目录里边的对应模块更新至最新版本。
- 使用`npm update <package> -g`可以把全局安装的对应命令行程序更新至最新版。
- 使用`npm cache clear`可以清空NPM本地缓存，用于对付使用相同版本号发布新版本代码的人。
- 使用`npm unpublish <package>@<version>`可以撤销发布自己发布过的某个版本代码。

小结

本章介绍了使用NodeJS编写代码前需要做的准备工作，总结起来有以下几点：

- 编写代码前先规划好目录结构，才能做到有条不紊。

- 稍大些的程序可以将代码拆分为多个模块管理，更大些的程序可以使用包来组织模块。
- 合理使用`node_modules`和`NODE_PATH`来解耦包的使用方式和物理路径。
- 使用NPM加入NodeJS生态圈互通有无。
- 想到了心仪的包名时请提前在NPM上抢注。

文件操作

让前端觉得如获神器的不是NodeJS能做网络编程，而是NodeJS能够操作文件。小至文件查找，大至代码编译，几乎没有一个前端工具不操作文件。换个角度讲，几乎也只需要一些数据处理逻辑，再加上一些文件操作，就能够编写出大多数前端工具。本章将介绍与之相关的NodeJS内置模块。

开门红

NodeJS提供了基本的文件操作API，但是像文件拷贝这种高级功能就没有提供，因此我们先拿文件拷贝程序练手。与`copy`命令类似，我们的程序需要能接受源文件路径与目标文件路径两个参数。

小文件拷贝

我们使用NodeJS内置的`fs`模块简单实现这个程序如下。

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

function copy(src, dst) {
    fs.writeFileSync(dst, fs.readFileSync(src));
}

function main(argv) {
    copy(argv[0], argv[1]);
}

main(process.argv.slice(2));
```

以上程序使用`fs.readFileSync`从源路径读取文件内容，并使用`fs.writeFileSync`将文件内容写入目标路径。

> **豆知识：** `process`是一个全局变量，可通过`process.argv`获得命令行参数。由于`argv[0]`固定等于NodeJS执行程序的绝对路径，`argv[1]`固定等于主模块的绝对路径，因此第一个命令行参数从`argv[2]`这个位置开始。

大文件拷贝

上边的程序拷贝一些小文件没啥问题，但这种一次性把所有文件内容都读取到内存中后再一次性写入磁盘的方式不适合拷贝大文件，内存会爆仓。对于大文件，我们只能读一点写一点，直到完成拷贝。因此上边的程序需要改造如下。

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

function copy(src, dst) {
    fs.createReadStream(src).pipe(fs.createWriteStream(dst));
}

function main(argv) {
    copy(argv[0], argv[1]);
}

main(process.argv.slice(2));
```

以上程序使用`fs.createReadStream`创建了一个源文件的只读数据流，并使用`fs.createWriteStream`创建了一个目标文件的只写数据流，并且用`pipe`方法把两个数据流连接了起来。连接起来后发生的事情，说得抽象点的话，水顺着水管从一个桶流到了另一个桶。

API走马观花

我们先大致看看NodeJS提供了哪些和文件操作有关的API。这里并不逐一介绍每个API的使用方法，官方文档已经做得很好了。

Buffer（数据块）

> **官方文档：** http://nodejs.org/api/buffer.html

JS语言自身只有字符串数据类型，没有二进制数据类型，因此NodeJS提供了一个与`String`对等的全局构造函数`Buffer`来提供对二进制数据的操作。除了可以读取文件得到`Buffer`的实例外，还能够直接构造，例如：

```
var bin = new Buffer([ 0x68, 0x65, 0x6c, 0x6c, 0x6f ]);
```

`Buffer`与字符串类似，除了可以用`.length`属性得到字节长度外，还可以用`[index]`方式读取指定位置的字节，例如：

```
bin[0]; // => 0x68;
```

Buffer与字符串能够互相转化，例如可以使用指定编码将二进制数据转化为字符串：

```
var str = bin.toString('utf-8'); // => "hello"
```

或者反过来，将字符串转换为指定编码下的二进制数据：

```
var bin = new Buffer('hello', 'utf-8'); // => <Buffer 68 65 6c 6c 6f>
```

Buffer与字符串有一个重要区别。字符串是只读的，并且对字符串的任何修改得到的都是一个新字符串，原字符串保持不变。至于Buffer，更像是可以做指针操作的C语言数组。例如，可以用[index]方式直接修改某个位置的字节。

```
bin[0] = 0x48;
```

而.slice方法也不是返回一个新的Buffer，而更像是返回了指向原Buffer中间的某个位置的指针，如下所示。

```
[ 0x68, 0x65, 0x6c, 0x6c, 0x6f ]
    ^           ^
    |           |
   bin     bin.slice(2)
```

因此对.slice方法返回的Buffer的修改会作用于原Buffer，例如：

```
var bin = new Buffer([ 0x68, 0x65, 0x6c, 0x6c, 0x6f ]);
var sub = bin.slice(2);

sub[0] = 0x65;
console.log(bin); // => <Buffer 68 65 65 6c 6f>
```

也因此，如果想要拷贝一份Buffer，得首先创建一个新的Buffer，并通过.copy方法把原Buffer中的数据复制过去。这个类似于申请一块新的内存，并把已有内存中的数据复制过去。以下是一个例子。

```
var bin = new Buffer([ 0x68, 0x65, 0x6c, 0x6c, 0x6f ]);
var dup = new Buffer(bin.length);

bin.copy(dup);
dup[0] = 0x48;
console.log(bin); // => <Buffer 68 65 6c 6c 6f>
console.log(dup); // => <Buffer 48 65 65 6c 6f>
```

总之，Buffer将JS的数据处理能力从字符串扩展到了任意二进制数据。

Stream（数据流）

> **官方文档：** http://nodejs.org/api/stream.html

当内存中无法一次装下需要处理的数据时，或者一边读取一边处理更加高效时，我们就需要用到数据流。NodeJS中通过各种`Stream`来提供对数据流的操作。

以上边的大文件拷贝程序为例，我们可以为数据来源创建一个只读数据流，示例如下：

```
var rs = fs.createReadStream(pathname);

rs.on('data', function (chunk) {
    doSomething(chunk);
});

rs.on('end', function () {
    cleanUp();
});
```

> **豆知识：** `Stream`基于事件机制工作，所有`Stream`的实例都继承于NodeJS提供的EventEmitter。

上边的代码中`data`事件会源源不断地被触发，不管`doSomething`函数是否处理得过来。代码可以继续做如下改造，以解决这个问题。

```
var rs = fs.createReadStream(src);

rs.on('data', function (chunk) {
    rs.pause();
    doSomething(chunk, function () {
        rs.resume();
    });
});

rs.on('end', function () {
    cleanUp();
});
```

以上代码给`doSomething`函数加上了回调，因此我们可以在处理数据前暂停数据读取，并在处理数据后继续读取数据。

此外，我们也可以为数据目标创建一个只写数据流，示例如下：

```
var rs = fs.createReadStream(src);
var ws = fs.createWriteStream(dst);

rs.on('data', function (chunk) {
    ws.write(chunk);
});

rs.on('end', function () {
    ws.end();
});
```

我们把`doSomething`换成了往只写数据流里写入数据后，以上代码看起来就像是一个文件拷贝程序了。但是以上代码存在上边提到的问题，如果写入速度跟不上读取速度的话，只写数据流内部的缓存会爆仓。我们可以根据`.write`方法的返回值来判断传入的数据是写入目标了，还是临时放在了缓存了，并根据`drain`事件来判断什么时候只写数据流已经将缓存中的数据写入目标，可以传入下一个待写数据了。因此代码可以改造如下：

```
var rs = fs.createReadStream(src);
var ws = fs.createWriteStream(dst);

rs.on('data', function (chunk) {
    if (ws.write(chunk) === false) {
        rs.pause();
    }
});

rs.on('end', function () {
    ws.end();
});

ws.on('drain', function () {
    rs.resume();
});
```

以上代码实现了数据从只读数据流到只写数据流的搬运，并包括了防爆仓控制。因为这种使用场景很多，例如上边的大文件拷贝程序，NodeJS直接提供了`.pipe`方法来做这件事情，其内部实现方式与上边的代码类似。

File System（文件系统）

> **官方文档：** http://nodejs.org/api/fs.html

NodeJS通过`fs`内置模块提供对文件的操作。`fs`模块提供的API基本上可以分为以下三类：

- 文件属性读写。

  其中常用的有`fs.stat`、`fs.chmod`、`fs.chown`等等。

- 文件内容读写。

  其中常用的有`fs.readFile`、`fs.readdir`、`fs.writeFile`、`fs.mkdir`等等。

- 底层文件操作。

  其中常用的有`fs.open`、`fs.read`、`fs.write`、`fs.close`等等。

NodeJS最精华的异步IO模型在`fs`模块里有着充分的体现，例如上边提到的这些API都通过回调函数传递结果。以`fs.readFile`为例：

```
fs.readFile(pathname, function (err, data) {
    if (err) {
        // Deal with error.
    } else {
        // Deal with data.
    }
});
```

如上边代码所示，基本上所有`fs`模块API的回调参数都有两个。第一个参数在有错误发生时等于异常对象，第二个参数始终用于返回API方法执行结果。

此外，`fs`模块的所有异步API都有对应的同步版本，用于无法使用异步操作时，或者同步操作更方便时的情况。同步API除了方法名的末尾多了一个`Sync`之外，异常对象与执行结果的传递方式也有相应变化。同样以`fs.readFileSync`为例：

```
try {
    var data = fs.readFileSync(pathname);
    // Deal with data.
} catch (err) {
    // Deal with error.
}
```

`fs`模块提供的API很多，这里不一一介绍，需要时请自行查阅官方文档。

Path（路径）

> **官方文档：** http://nodejs.org/api/path.html

操作文件时难免不与文件路径打交道。NodeJS提供了`path`内置模块来简化路径相关操作，并提升代码可读性。以下分别介绍几个常用的API。

- path.normalize

  将传入的路径转换为标准路径，具体讲的话，除了解析路径中的`.`与`..`外，还能去掉多余的斜杠。如果有程序需要使用路径作为某些数据的索引，但又允许用户随意输入路径时，就需要使用该方法保证路径的唯一性。以下是一个例子：

  ```
  var cache = {};

  function store(key, value) {
      cache[path.normalize(key)] = value;
  }

  store('foo/bar', 1);
  store('foo//baz//../bar', 2);
  console.log(cache);  // => { "foo/bar": 2 }
  ```

  > **坑出没注意：** 标准化之后的路径里的斜杠在Windows系统下是`\`，而在Linux系统下是`/`。如果想保证任何系统下都使用`/`作为路径分隔符的话，需要用`.replace(/\\/g, '/')`再替换一下标准路径。

- path.join

  将传入的多个路径拼接为标准路径。该方法可避免手工拼接路径字符串的繁琐，并且能在不同系统下正确使用相应的路径分隔符。以下是一个例子：

  ```
  path.join('foo/', 'baz/', '../bar'); // => "foo/bar"
  ```

- path.extname

  当我们需要根据不同文件扩展名做不同操作时，该方法就显得很好用。以下是一个例子：

  ```
  path.extname('foo/bar.js'); // => ".js"
  ```

`path`模块提供的其余方法也不多，稍微看一下官方文档就能全部掌握。

遍历目录

遍历目录是操作文件时的一个常见需求。比如写一个程序，需要找到并处理指定目录下的所有JS文件时，就需要遍历整个目录。

递归算法

遍历目录时一般使用递归算法，否则就难以编写出简洁的代码。递归算法与数学归纳法类似，通过不断缩小问题的规模来解决问题。以下示例说明了这种方法。

```
function factorial(n) {
    if (n === 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

上边的函数用于计算N的阶乘（N!）。可以看到，当N大于1时，问题简化为计算N乘以N-1的阶乘。当N等于1时，问题达到最小规模，不需要再简化，因此直接返回1。

> **陷阱：** 使用递归算法编写的代码虽然简洁，但由于每递归一次就产生一次函数调用，在需要优先考虑性能时，需要把递归算法转换为循环算法，以减少函数调用次数。

遍历算法

目录是一个树状结构，在遍历时一般使用深度优先+先序遍历算法。深度优先，意味着到达一个节点后，首先接着遍历子节点而不是邻居节点。先序遍历，意味着首次到达了某节点就算遍历完成，而不是最后一次返回某节点才算数。因此使用这种遍历方式时，下边这棵树的遍历顺序是`A > B > D > E > C > F`。

```
        A
       / \
      B   C
     / \   \
    D   E   F
```

同步遍历

了解了必要的算法后，我们可以简单地实现以下目录遍历函数。

```
function travel(dir, callback) {
    fs.readdirSync(dir).forEach(function (file) {
        var pathname = path.join(dir, file);

        if (fs.statSync(pathname).isDirectory()) {
            travel(pathname, callback);
        } else {
            callback(pathname);
        }
    });
}
```

可以看到，该函数以某个目录作为遍历的起点。遇到一个子目录时，就先接着遍历子目录。遇到一个文件时，就把文件的绝对路径传给回调函数。回调函数拿到文件路径后，就可以做各种判断和处理。因此假设有以下目录：

```
- /home/user/
    - foo/
        x.js
    - bar/
        y.js
    z.css
```

使用以下代码遍历该目录时，得到的输入如下。

```
travel('/home/user', function (pathname) {
    console.log(pathname);
});


-----------------------
/home/user/foo/x.js
/home/user/bar/y.js
/home/user/z.css
```

异步遍历

如果读取目录或读取文件状态时使用的是异步API，目录遍历函数实现起来会有些复杂，但原理完全相同。`travel`函数的异步版本如下。

```
function travel(dir, callback, finish) {
    fs.readdir(dir, function (err, files) {
```

```
        (function next(i) {
            if (i < files.length) {
                var pathname = path.join(dir, files[i]);

                fs.stat(pathname, function (err, stats) {
                    if (stats.isDirectory()) {
                        travel(pathname, callback, function () {
                            next(i + 1);
                        });
                    } else {
                        callback(pathname, function () {
                            next(i + 1);
                        });
                    }
                });
            } else {
                finish && finish();
            }
        }(0));
    });
}
```

这里不详细介绍异步遍历函数的编写技巧，在后续章节中会详细介绍这个。总之我们可以看到异步编程还是蛮复杂的。

文本编码

使用NodeJS编写前端工具时，操作得最多的是文本文件，因此也就涉及到了文件编码的处理问题。我们常用的文本编码有`UTF8`和`GBK`两种，并且`UTF8`文件还可能带有BOM。在读取不同编码的文本文件时，需要将文件内容转换为JS使用的`UTF8`编码字符串后才能正常处理。

BOM的移除

BOM用于标记一个文本文件使用Unicode编码，其本身是一个Unicode字符（"\uFEFF"），位于文本文件头部。在不同的Unicode编码下，BOM字符对应的二进制字节如下：

```
    Bytes      Encoding
----------------------------

    FE FF       UTF16BE
    FF FE       UTF16LE
    EF BB BF    UTF8
```

因此，我们可以根据文本文件头几个字节等于啥来判断文件是否包含BOM，以及使用哪种Unicode编码。但是，BOM字符虽然起到了标记文件编码的作用，其本身却不属于文件内容的一部分，如果读取文本文件时不去掉BOM，在某些使用场景下就会有问题。例如我们把几个JS文件合并成一个文件后，如果文件中间含有BOM字符，就会导致浏览器JS语法错误。因此，使用NodeJS读取文本文件时，一般需要去掉BOM。例如，以下代码实现了识别和去除UTF8 BOM的功能。

```
function readText(pathname) {
    var bin = fs.readFileSync(pathname);

    if (bin[0] === 0xEF && bin[1] === 0xBB && bin[2] === 0xBF) {
        bin = bin.slice(3);
    }

    return bin.toString('utf-8');
}
```

GBK转UTF8

NodeJS支持在读取文本文件时，或者在`Buffer`转换为字符串时指定文本编码，但遗憾的是，GBK编码不在NodeJS自身支持范围内。因此，一般我们借助`iconv-lite`这个三方包来转换编码。使用NPM下载该包后，我们可以按下边方式编写一个读取GBK文本文件的函数。

```
var iconv = require('iconv-lite');

function readGBKText(pathname) {
    var bin = fs.readFileSync(pathname);

    return iconv.decode(bin, 'gbk');
}
```

单字节编码

有时候，我们无法预知需要读取的文件采用哪种编码，因此也就无法指定正确的编码。比如我们要处理的某些CSS文件中，有的用GBK编码，有的用UTF8编码。虽然可以一定程度可以根据文件的字节内容猜测出文本编码，但这里要介绍的是有些局限，但是要简单得多的一种技术。

首先我们知道，如果一个文本文件只包含英文字符，比如`Hello World`，那无论用GBK编码或是UTF8编码读取这个文件都是没问题的。这是因为在这些编码下，ASCII0~128范围内字符都使用相同的单字节编码。

反过来讲，即使一个文本文件中有中文等字符，如果我们需要处理的字符仅在ASCII0~128范围内，比如除了注释和字符串以外的JS代码，我们就可以统一使用单字节编码来读取文件，不用关心文件

的实际编码是GBK还是UTF8。以下示例说明了这种方法。

1. GBK编码源文件内容：

   ```
   var foo = '中文';
   ```

2. 对应字节：

   ```
   76 61 72 20 66 6F 6F 20 3D 20 27 D6 D0 CE C4 27 3B
   ```

3. 使用单字节编码读取后得到的内容：

   ```
   var foo = '{乱码}{乱码}{乱码}{乱码}';
   ```

4. 替换内容：

   ```
   var bar = '{乱码}{乱码}{乱码}{乱码}';
   ```

5. 使用单字节编码保存后对应字节：

   ```
   76 61 72 20 62 61 72 20 3D 20 27 D6 D0 CE C4 27 3B
   ```

6. 使用GBK编码读取后得到内容：

   ```
   var bar = '中文';
   ```

这里的诀窍在于，不管大于0xEF的单个字节在单字节编码下被解析成什么乱码字符，使用同样的单字节编码保存这些乱码字符时，背后对应的字节保持不变。

NodeJS中自带了一种`binary`编码可以用来实现这个方法，因此在下例中，我们使用这种编码来演示上例对应的代码该怎么写。

```
function replace(pathname) {
    var str = fs.readFileSync(pathname, 'binary');
    str = str.replace('foo', 'bar');
    fs.writeFileSync(pathname, str, 'binary');
}
```

小结

本章介绍了使用NodeJS操作文件时需要的API以及一些技巧，总结起来有以下几点：

- 学好文件操作，编写各种程序都不怕。
- 如果不是很在意性能，`fs`模块的同步API能让生活更加美好。
- 需要对文件读写做到字节级别的精细控制时，请使用`fs`模块的文件底层操作API。
- 不要使用拼接字符串的方式来处理路径，使用`path`模块。
- 掌握好目录遍历和文件编码处理技巧，很实用。

网络操作

不了解网络编程的程序员不是好前端，而NodeJS恰好提供了一扇了解网络编程的窗口。通过NodeJS，除了可以编写一些服务端程序来协助前端开发和测试外，还能够学习一些HTTP协议与Socket协议的相关知识，这些知识在优化前端性能和排查前端故障时说不定能派上用场。本章将介绍与之相关的NodeJS内置模块。

开门红

NodeJS本来的用途是编写高性能Web服务器。我们首先在这里重复一下官方文档里的例子，使用NodeJS内置的`http`模块简单实现一个HTTP服务器。

```
var http = require('http');


http.createServer(function (request, response) {
    response.writeHead(200, { 'Content-Type': 'text-plain' });
    response.end('Hello World\n');
}).listen(8124);
```

以上程序创建了一个HTTP服务器并监听`8124`端口，打开浏览器访问该端口`http://127.0.0.1:8124/`就能够看到效果。

> **豆知识：** 在Linux系统下，监听1024以下端口需要root权限。因此，如果想监听80或443端口的话，需要使用`sudo`命令启动程序。

API走马观花

我们先大致看看NodeJS提供了哪些和网络操作有关的API。这里并不逐一介绍每个API的使用方法，官方文档已经做得很好了。

HTTP

> **官方文档：** http://nodejs.org/api/http.html

'http'模块提供两种使用方式：

- 作为服务端使用时，创建一个HTTP服务器，监听HTTP客户端请求并返回响应。
- 作为客户端使用时，发起一个HTTP客户端请求，获取服务端响应。

首先我们来看看服务端模式下如何工作。如开门红中的例子所示，首先需要使用.createServer方法创建一个服务器，然后调用.listen方法监听端口。之后，每当来了一个客户端请求，创建服务器时传入的回调函数就被调用一次。可以看出，这是一种事件机制。

HTTP请求本质上是一个数据流，由请求头（headers）和请求体（body）组成。例如以下是一个完整的HTTP请求数据内容。

```
POST / HTTP/1.1
User-Agent: curl/7.26.0
Host: localhost
Accept: */*
Content-Length: 11
Content-Type: application/x-www-form-urlencoded

Hello World
```

可以看到，空行之上是请求头，之下是请求体。HTTP请求在发送给服务器时，可以认为是按照从头到尾的顺序一个字节一个字节地以数据流方式发送的。而http模块创建的HTTP服务器在接收到完整的请求头后，就会调用回调函数。在回调函数中，除了可以使用request对象访问请求头数据外，还能把request对象当作一个只读数据流来访问请求体数据。以下是一个例子。

```
http.createServer(function (request, response) {
    var body = [];

    console.log(request.method);
    console.log(request.headers);

    request.on('data', function (chunk) {
        body.push(chunk);
    });

    request.on('end', function () {
        body = Buffer.concat(body);
        console.log(body.toString());
    });
}).listen(80);

----------------------------------
POST
{ 'user-agent': 'curl/7.26.0',
```

```
  host: 'localhost',
  accept: '*/*',
  'content-length': '11',
  'content-type': 'application/x-www-form-urlencoded' }
Hello World
```

HTTP响应本质上也是一个数据流，同样由响应头（headers）和响应体（body）组成。例如以下是一个完整的HTTP请求数据内容。

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 11
Date: Tue, 05 Nov 2013 05:31:38 GMT
Connection: keep-alive

Hello World
```

在回调函数中，除了可以使用`response`对象来写入响应头数据外，还能把`response`对象当作一个只写数据流来写入响应体数据。例如在以下例子中，服务端原样将客户端请求的请求体数据返回给客户端。

```
http.createServer(function (request, response) {
    response.writeHead(200, { 'Content-Type': 'text/plain' });

    request.on('data', function (chunk) {
        response.write(chunk);
    });

    request.on('end', function () {
        response.end();
    });
}).listen(80);
```

接下来我们看看客户端模式下如何工作。为了发起一个客户端HTTP请求，我们需要指定目标服务器的位置并发送请求头和请求体，以下示例演示了具体做法。

```
var options = {
        hostname: 'www.example.com',
        port: 80,
        path: '/upload',
        method: 'POST',
```

```
        headers: {
            'Content-Type': 'application/x-www-form-urlencoded'
        }
    };

var request = http.request(options, function (response) {});

request.write('Hello World');
request.end();
```

可以看到，.request方法创建了一个客户端，并指定请求目标和请求头数据。之后，就可以把request对象当作一个只写数据流来写入请求体数据和结束请求。另外，由于HTTP请求中GET请求是最常见的一种，并且不需要请求体，因此http模块也提供了以下便捷API。

```
http.get('http://www.example.com/', function (response) {});
```

当客户端发送请求并接收到完整的服务端响应头时，就会调用回调函数。在回调函数中，除了可以使用response对象访问响应头数据外，还能把response对象当作一个只读数据流来访问响应体数据。以下是一个例子。

```
http.get('http://www.example.com/', function (response) {
    var body = [];

    console.log(response.statusCode);
    console.log(response.headers);

    response.on('data', function (chunk) {
        body.push(chunk);
    });

    response.on('end', function () {
        body = Buffer.concat(body);
        console.log(body.toString());
    });
});

-----------------------------------
200
{ 'content-type': 'text/html',
  server: 'Apache',
```

```
  'content-length': '801',
  date: 'Tue, 05 Nov 2013 06:08:41 GMT',
  connection: 'keep-alive' }
<!DOCTYPE html>
...
```

HTTPS

> **官方文档：** http://nodejs.org/api/https.html

`https`模块与`http`模块极为类似，区别在于`https`模块需要额外处理SSL证书。

在服务端模式下，创建一个HTTPS服务器的示例如下。

```
var options = {
        key: fs.readFileSync('./ssl/default.key'),
        cert: fs.readFileSync('./ssl/default.cer')
    };

var server = https.createServer(options, function (request, response) {
        // ...
    });
```

可以看到，与创建HTTP服务器相比，多了一个`options`对象，通过`key`和`cert`字段指定了HTTPS服务器使用的私钥和公钥。

另外，NodeJS支持SNI技术，可以根据HTTPS客户端请求使用的域名动态使用不同的证书，因此同一个HTTPS服务器可以使用多个域名提供服务。接着上例，可以使用以下方法为HTTPS服务器添加多组证书。

```
server.addContext('foo.com', {
    key: fs.readFileSync('./ssl/foo.com.key'),
    cert: fs.readFileSync('./ssl/foo.com.cer')
});

server.addContext('bar.com', {
    key: fs.readFileSync('./ssl/bar.com.key'),
    cert: fs.readFileSync('./ssl/bar.com.cer')
});
```

在客户端模式下，发起一个HTTPS客户端请求与`http`模块几乎相同，示例如下。

```
var options = {
        hostname: 'www.example.com',
        port: 443,
        path: '/',
        method: 'GET'
    };

var request = https.request(options, function (response) {});

request.end();
```

但如果目标服务器使用的SSL证书是自制的，不是从颁发机构购买的，默认情况下`https`模块会拒绝连接，提示说有证书安全问题。在`options`里加入`rejectUnauthorized: false`字段可以禁用对证书有效性的检查，从而允许`https`模块请求开发环境下使用自制证书的HTTPS服务器。

URL

> **官方文档：** http://nodejs.org/api/url.html

处理HTTP请求时`url`模块使用率超高，因为该模块允许解析URL、生成URL，以及拼接URL。首先我们来看看一个完整的URL的各组成部分。

```
                           href
 -----------------------------------------------------------------
                             host              path
                      --------------- ----------------------------
 http: // user:pass @ host.com : 8080 /p/a/t/h ?query=string #hash
 -----    ---------   --------   ---- -------- ------------- -----
protocol     auth     hostname   port pathname     search     hash
                                                ------------
                                                  query
```

我们可以使用`.parse`方法来将一个URL字符串转换为URL对象，示例如下。

```
url.parse('http://user:pass@host.com:8080/p/a/t/h?query=string#hash');
/* =>
{ protocol: 'http:',
  auth: 'user:pass',
  host: 'host.com:8080',
  port: '8080',
```

```
  hostname: 'host.com',
  hash: '#hash',
  search: '?query=string',
  query: 'query=string',
  pathname: '/p/a/t/h',
  path: '/p/a/t/h?query=string',
  href: 'http://user:pass@host.com:8080/p/a/t/h?query=string#hash' }
*/
```

传给`.parse`方法的不一定要是一个完整的URL，例如在HTTP服务器回调函数中，`request.url`不包含协议头和域名，但同样可以用`.parse`方法解析。

```
http.createServer(function (request, response) {
    var tmp = request.url; // => "/foo/bar?a=b"
    url.parse(tmp);
    /* =>
    { protocol: null,
      slashes: null,
      auth: null,
      host: null,
      port: null,
      hostname: null,
      hash: null,
      search: '?a=b',
      query: 'a=b',
      pathname: '/foo/bar',
      path: '/foo/bar?a=b',
      href: '/foo/bar?a=b' }
    */
}).listen(80);
```

`.parse`方法还支持第二个和第三个布尔类型可选参数。第二个参数等于`true`时，该方法返回的URL对象中，`query`字段不再是一个字符串，而是一个经过`querystring`模块转换后的参数对象。第三个参数等于`true`时，该方法可以正确解析不带协议头的URL，例如`//www.example.com/foo/bar`。

反过来，`format`方法允许将一个URL对象转换为URL字符串，示例如下。

```
url.format({
    protocol: 'http:',
    host: 'www.example.com',
```

```
    pathname: '/p/a/t/h',
    search: 'query=string'
});
/* =>
'http://www.example.com/p/a/t/h?query=string'
*/
```

另外，`.resolve`方法可以用于拼接URL，示例如下。

```
url.resolve('http://www.example.com/foo/bar', '../baz');
/* =>
http://www.example.com/baz
*/
```

Query String

> **官方文档：** http://nodejs.org/api/querystring.html

`querystring`模块用于实现URL参数字符串与参数对象的互相转换，示例如下。

```
querystring.parse('foo=bar&baz=qux&baz=quux&corge');
/* =>
{ foo: 'bar', baz: ['qux', 'quux'], corge: '' }
*/

querystring.stringify({ foo: 'bar', baz: ['qux', 'quux'], corge: '' });
/* =>
'foo=bar&baz=qux&baz=quux&corge='
*/
```

Zlib

> **官方文档：** http://nodejs.org/api/zlib.html

`zlib`模块提供了数据压缩和解压的功能。当我们处理HTTP请求和响应时，可能需要用到这个模块。

首先我们看一个使用`zlib`模块压缩HTTP响应体数据的例子。这个例子中，判断了客户端是否支持gzip，并在支持的情况下使用`zlib`模块返回gzip之后的响应体数据。

```
http.createServer(function (request, response) {
    var i = 1024,
```

```
        data = '';

    while (i--) {
        data += '.';
    }

    if ((request.headers['accept-encoding'] || '').indexOf('gzip') !== -1) {
        zlib.gzip(data, function (err, data) {
            response.writeHead(200, {
                'Content-Type': 'text/plain',
                'Content-Encoding': 'gzip'
            });
            response.end(data);
        });
    } else {
        response.writeHead(200, {
            'Content-Type': 'text/plain'
        });
        response.end(data);
    }
}).listen(80);
```

接着我们看一个使用zlib模块解压HTTP响应体数据的例子。这个例子中，判断了服务端响应是否使用gzip压缩，并在压缩的情况下使用zlib模块解压响应体数据。

```
var options = {
        hostname: 'www.example.com',
        port: 80,
        path: '/',
        method: 'GET',
        headers: {
            'Accept-Encoding': 'gzip, deflate'
        }
    };

http.request(options, function (response) {
    var body = [];

    response.on('data', function (chunk) {
```

```
        body.push(chunk);
    });

    response.on('end', function () {
        body = Buffer.concat(body);

        if (response.headers['content-encoding'] === 'gzip') {
            zlib.gunzip(body, function (err, data) {
                console.log(data.toString());
            });
        } else {
            console.log(data.toString());
        }
    });
}).end();
```

Net

> **官方文档：** http://nodejs.org/api/net.html

net模块可用于创建Socket服务器或Socket客户端。由于Socket在前端领域的使用范围还不是很广，这里先不涉及到WebSocket的介绍，仅仅简单演示一下如何从Socket层面来实现HTTP请求和响应。

首先我们来看一个使用Socket搭建一个很不严谨的HTTP服务器的例子。这个HTTP服务器不管收到啥请求，都固定返回相同的响应。

```
net.createServer(function (conn) {
    conn.on('data', function (data) {
        conn.write([
            'HTTP/1.1 200 OK',
            'Content-Type: text/plain',
            'Content-Length: 11',
            '',
            'Hello World'
        ].join('\n'));
    });
}).listen(80);
```

接着我们来看一个使用Socket发起HTTP客户端请求的例子。这个例子中，Socket客户端在建立连接后发送了一个HTTP GET请求，并通过`data`事件监听函数来获取服务器响应。

```
var options = {
        port: 80,
        host: 'www.example.com'
    };

var client = net.connect(options, function () {
        client.write([
            'GET / HTTP/1.1',
            'User-Agent: curl/7.26.0',
            'Host: www.baidu.com',
            'Accept: */*',
            '',
            ''
        ].join('\n'));
    });

client.on('data', function (data) {
    console.log(data.toString());
    client.end();
});
```

灵机一点

使用NodeJS操作网络，特别是操作HTTP请求和响应时会遇到一些惊喜，这里对一些常见问题做解答。

- 问： 为什么通过`headers`对象访问到的HTTP请求头或响应头字段不是驼峰的？

  答： 从规范上讲，HTTP请求头和响应头字段都应该是驼峰的。但现实是残酷的，不是每个HTTP服务端或客户端程序都严格遵循规范，所以NodeJS在处理从别的客户端或服务端收到的头字段时，都统一地转换为了小写字母格式，以便开发者能使用统一的方式来访问头字段，例如`headers['content-length']`。

- 问： 为什么`http`模块创建的HTTP服务器返回的响应是`chunked`传输方式的？

  答： 因为默认情况下，使用`.writeHead`方法写入响应头后，允许使用`.write`方法写入任意长度的响应体数据，并使用`.end`方法结束一个响应。由于响应体数据长度不确定，因此NodeJS自动在响应头里添加了`Transfer-Encoding: chunked`字段，并采用`chunked`传输方式。但是当响应体

数据长度确定时，可使用.writeHead方法在响应头里加上Content-Length字段，这样做之后NodeJS就不会自动添加Transfer-Encoding字段和使用chunked传输方式。

- 问： 为什么使用http模块发起HTTP客户端请求时，有时候会发生socket hang up错误？

  答： 发起客户端HTTP请求前需要先创建一个客户端。http模块提供了一个全局客户端http.globalAgent，可以让我们使用.request或.get方法时不用手动创建客户端。但是全局客户端默认只允许5个并发Socket连接，当某一个时刻HTTP客户端请求创建过多，超过这个数字时，就会发生socket hang up错误。解决方法也很简单，通过http.globalAgent.maxSockets属性把这个数字改大些即可。另外，https模块遇到这个问题时也一样通过https.globalAgent.maxSockets属性来处理。

小结

本章介绍了使用NodeJS操作网络时需要的API以及一些坑回避技巧，总结起来有以下几点：

- http和https模块支持服务端模式和客户端模式两种使用方式。
- request和response对象除了用于读写头数据外，都可以当作数据流来操作。
- url.parse方法加上request.url属性是处理HTTP请求时的固定搭配。
- 使用zlib模块可以减少使用HTTP协议时的数据传输量。
- 通过net模块的Socket服务器与客户端可对HTTP协议做底层操作。
- 小心踩坑。

进程管理

NodeJS可以感知和控制自身进程的运行环境和状态，也可以创建子进程并与其协同工作，这使得NodeJS可以把多个程序组合在一起共同完成某项工作，并在其中充当胶水和调度器的作用。本章除了介绍与之相关的NodeJS内置模块外，还会重点介绍典型的使用场景。

开门红

我们已经知道了NodeJS自带的fs模块比较基础，把一个目录里的所有文件和子目录都拷贝到另一个目录里需要写不少代码。另外我们也知道，终端下的cp命令比较好用，一条cp -r source/* target命令就能搞定目录拷贝。那我们首先看看如何使用NodeJS调用终端命令来简化目录拷贝，示例代码如下：

```
var child_process = require('child_process');
var util = require('util');

function copy(source, target, callback) {
    child_process.exec(
        util.format('cp -r %s/* %s', source, target), callback);
}

copy('a', 'b', function (err) {
    // ...
});
```

从以上代码中可以看到，子进程是异步运行的，通过回调函数返回执行结果。

API走马观花

我们先大致看看NodeJS提供了哪些和进程管理有关的API。这里并不逐一介绍每个API的使用方法，官方文档已经做得很好了。

Process

> **官方文档：** http://nodejs.org/api/process.html

任何一个进程都有启动进程时使用的命令行参数，有标准输入标准输出，有运行权限，有运行环境和运行状态。在NodeJS中，可以通过`process`对象感知和控制NodeJS自身进程的方方面面。另外需要注意的是，`process`不是内置模块，而是一个全局对象，因此在任何地方都可以直接使用。

Child Process

> **官方文档：** http://nodejs.org/api/child_process.html

使用`child_process`模块可以创建和控制子进程。该模块提供的API中最核心的是`.spawn`，其余API都是针对特定使用场景对它的进一步封装，算是一种语法糖。

Cluster

> **官方文档：** http://nodejs.org/api/cluster.html

`cluster`模块是对`child_process`模块的进一步封装，专用于解决单进程NodeJS Web服务器无法充分利用多核CPU的问题。使用该模块可以简化多进程服务器程序的开发，让每个核上运行一个工作

进程，并统一通过主进程监听端口和分发请求。

应用场景

和进程管理相关的API单独介绍起来比较枯燥，因此这里从一些典型的应用场景出发，分别介绍一些重要API的使用方法。

如何获取命令行参数

在NodeJS中可以通过`process.argv`获取命令行参数。但是比较意外的是，`node`执行程序路径和主模块文件路径固定占据了`argv[0]`和`argv[1]`两个位置，而第一个命令行参数从`argv[2]`开始。为了让`argv`使用起来更加自然，可以按照以下方式处理。

```
function main(argv) {
    // ...
}


main(process.argv.slice(2));
```

如何退出程序

通常一个程序做完所有事情后就正常退出了，这时程序的退出状态码为`0`。或者一个程序运行时发生了异常后就挂了，这时程序的退出状态码不等于`0`。如果我们在代码中捕获了某个异常，但是觉得程序不应该继续运行下去，需要立即退出，并且需要把退出状态码设置为指定数字，比如`1`，就可以按照以下方式：

```
try {
    // ...
} catch (err) {
    // ...
    process.exit(1);
}
```

如何控制输入输出

NodeJS程序的标准输入流（stdin）、一个标准输出流（stdout）、一个标准错误流（stderr）分别对应`process.stdin`、`process.stdout`和`process.stderr`，第一个是只读数据流，后边两个是只写数据流，对它们的操作按照对数据流的操作方式即可。例如，`console.log`可以按照以下方式实现。

```
function log() {
    process.stdout.write(
```

```
        util.format.apply(util, arguments) + '\n');
}
```

如何降权

在Linux系统下，我们知道需要使用root权限才能监听1024以下端口。但是一旦完成端口监听后，继续让程序运行在root权限下存在安全隐患，因此最好能把权限降下来。以下是这样一个例子。

```
http.createServer(callback).listen(80, function () {
    var env = process.env,
        uid = parseInt(env['SUDO_UID'] || process.getuid(), 10),
        gid = parseInt(env['SUDO_GID'] || process.getgid(), 10);

    process.setgid(gid);
    process.setuid(uid);
});
```

上例中有几点需要注意：

1. 如果是通过`sudo`获取root权限的，运行程序的用户的UID和GID保存在环境变量`SUDO_UID`和`SUDO_GID`里边。如果是通过`chmod +s`方式获取root权限的，运行程序的用户的UID和GID可直接通过`process.getuid`和`process.getgid`方法获取。
2. `process.setuid`和`process.setgid`方法只接受`number`类型的参数。
3. 降权时必须先降GID再降UID，否则顺序反过来的话就没权限更改程序的GID了。

如何创建子进程

以下是一个创建NodeJS子进程的例子。

```
var child = child_process.spawn('node', [ 'xxx.js' ]);

child.stdout.on('data', function (data) {
    console.log('stdout: ' + data);
});

child.stderr.on('data', function (data) {
    console.log('stderr: ' + data);
});

child.on('close', function (code) {
```

```
    console.log('child process exited with code ' + code);
});
```

上例中使用了`.spawn(exec, args, options)`方法，该方法支持三个参数。第一个参数是执行文件路径，可以是执行文件的相对或绝对路径，也可以是根据PATH环境变量能找到的执行文件名。第二个参数中，数组中的每个成员都按顺序对应一个命令行参数。第三个参数可选，用于配置子进程的执行环境与行为。

另外，上例中虽然通过子进程对象的`.stdout`和`.stderr`访问子进程的输出，但通过`options.stdio`字段的不同配置，可以将子进程的输入输出重定向到任何数据流上，或者让子进程共享父进程的标准输入输出流，或者直接忽略子进程的输入输出。

进程间如何通讯

在Linux系统下，进程之间可以通过信号互相通信。以下是一个例子。

```
/* parent.js */
var child = child_process.spawn('node', ['child.js']);

child.kill('SIGTERM');

/* child.js */
process.on('SIGTERM', function () {
    cleanUp();
    process.exit(0);
});
```

在上例中，父进程通过`.kill`方法向子进程发送`SIGTERM`信号，子进程监听`process`对象的`SIGTERM`事件响应信号。不要被`.kill`方法的名称迷惑了，该方法本质上是用来给进程发送信号的，进程收到信号后具体要做啥，完全取决于信号的种类和进程自身的代码。

另外，如果父子进程都是NodeJS进程，就可以通过IPC（进程间通讯）双向传递数据。以下是一个例子。

```
/* parent.js */
var child = child_process.spawn('node', ['child.js'], {
        stdio: [0, 1, 2, 'ipc']
    });

child.on('message', function (msg) {
    console.log(msg);
});
```

```
child.send({ hello: 'hello' });

/* child.js */
process.on('message', function (msg) {
    msg.hello = msg.hello.toUpperCase();
    process.send(msg);
});
```

可以看到，父进程在创建子进程时，在`options.stdio`字段中通过`ipc`开启了一条IPC通道，之后就可以监听子进程对象的`message`事件接收来自子进程的消息，并通过`.send`方法给子进程发送消息。在子进程这边，可以在`process`对象上监听`message`事件接收来自父进程的消息，并通过`.send`方法向父进程发送消息。数据在传递过程中，会先在发送端使用`JSON.stringify`方法序列化，再在接收端使用`JSON.parse`方法反序列化。

如何守护子进程

守护进程一般用于监控工作进程的运行状态，在工作进程不正常退出时重启工作进程，保障工作进程不间断运行。以下是一种实现方式。

```
/* daemon.js */
function spawn(mainModule) {
    var worker = child_process.spawn('node', [ mainModule ]);

    worker.on('exit', function (code) {
        if (code !== 0) {
            spawn(mainModule);
        }
    });
}

spawn('worker.js');
```

可以看到，工作进程非正常退出时，守护进程立即重启工作进程。

小结

本章介绍了使用NodeJS管理进程时需要的API以及主要的应用场景，总结起来有以下几点：

- 使用`process`对象管理自身。
- 使用`child_process`模块创建和管理子进程。

异步编程

NodeJS最大的卖点——事件机制和异步IO，对开发者并不是透明的。开发者需要按异步方式编写代码才用得上这个卖点，而这一点也遭到了一些NodeJS反对者的抨击。但不管怎样，异步编程确实是NodeJS最大的特点，没有掌握异步编程就不能说是真正学会了NodeJS。本章将介绍与异步编程相关的各种知识。

回调

在代码中，异步编程的直接体现就是回调。异步编程依托于回调来实现，但不能说使用了回调后程序就异步化了。我们首先可以看看以下代码。

```
function heavyCompute(n, callback) {
    var count = 0,
        i, j;

    for (i = n; i > 0; --i) {
        for (j = n; j > 0; --j) {
            count += 1;
        }
    }

    callback(count);
}

heavyCompute(10000, function (count) {
    console.log(count);
});

console.log('hello');

-- Console ----------------------------
100000000
hello
```

可以看到，以上代码中的回调函数仍然先于后续代码执行。JS本身是单线程运行的，不可能在一段代码还未结束运行时去运行别的代码，因此也就不存在异步执行的概念。

但是，如果某个函数做的事情是创建一个别的线程或进程，并与JS主线程并行地做一些事情，并在事情做完后通知JS主线程，那情况又不一样了。我们接着看看以下代码。

```
setTimeout(function () {
    console.log('world');
}, 1000);

console.log('hello');

-- Console ------------------------------
hello
world
```

这次可以看到，回调函数后于后续代码执行了。如同上边所说，JS本身是单线程的，无法异步执行，因此我们可以认为`setTimeout`这类JS规范之外的由运行环境提供的特殊函数做的事情是创建一个平行线程后立即返回，让JS主进程可以接着执行后续代码，并在收到平行进程的通知后再执行回调函数。除了`setTimeout`、`setInterval`这些常见的，这类函数还包括NodeJS提供的诸如`fs.readFile`之类的异步API。

另外，我们仍然回到JS是单线程运行的这个事实上，这决定了JS在执行完一段代码之前无法执行包括回调函数在内的别的代码。也就是说，即使平行线程完成工作了，通知JS主线程执行回调函数了，回调函数也要等到JS主线程空闲时才能开始执行。以下就是这么一个例子。

```
function heavyCompute(n) {
    var count = 0,
        i, j;

    for (i = n; i > 0; --i) {
        for (j = n; j > 0; --j) {
            count += 1;
        }
    }
}

var t = new Date();

setTimeout(function () {
    console.log(new Date() - t);
}, 1000);

heavyCompute(50000);
```

```
-- Console ----------------------------
8520
```

可以看到，本来应该在1秒后被调用的回调函数因为JS主线程忙于运行其它代码，实际执行时间被大幅延迟。

代码设计模式

异步编程有很多特有的代码设计模式，为了实现同样的功能，使用同步方式和异步方式编写的代码会有很大差异。以下分别介绍一些常见的模式。

函数返回值

使用一个函数的输出作为另一个函数的输入是很常见的需求，在同步方式下一般按以下方式编写代码：

```
var output = fn1(fn2('input'));
// Do something.
```

而在异步方式下，由于函数执行结果不是通过返回值，而是通过回调函数传递，因此一般按以下方式编写代码：

```
fn2('input', function (output2) {
    fn1(output2, function (output1) {
        // Do something.
    });
});
```

可以看到，这种方式就是一个回调函数套一个回调函多，套得太多了很容易写出>形状的代码。

遍历数组

在遍历数组时，使用某个函数依次对数据成员做一些处理也是常见的需求。如果函数是同步执行的，一般就会写出以下代码：

```
var len = arr.length,
    i = 0;

for (; i < len; ++i) {
    arr[i] = sync(arr[i]);
}

// All array items have processed.
```

如果函数是异步执行的，以上代码就无法保证循环结束后所有数组成员都处理完毕了。如果数组成员必须一个接一个串行处理，则一般按照以下方式编写异步代码：

```
(function next(i, len, callback) {
    if (i < len) {
        async(arr[i], function (value) {
            arr[i] = value;
            next(i + 1, len, callback);
        });
    } else {
        callback();
    }
}(0, arr.length, function () {
    // All array items have processed.
}));
```

可以看到，以上代码在异步函数执行一次并返回执行结果后才传入下一个数组成员并开始下一轮执行，直到所有数组成员处理完毕后，通过回调的方式触发后续代码的执行。

如果数组成员可以并行处理，但后续代码仍然需要所有数组成员处理完毕后才能执行的话，则异步代码会调整成以下形式：

```
(function (i, len, count, callback) {
    for (; i < len; ++i) {
        (function (i) {
            async(arr[i], function (value) {
                arr[i] = value;
                if (++count === len) {
                    callback();
                }
            });
        }(i));
    }
}(0, arr.length, 0, function () {
    // All array items have processed.
}));
```

可以看到，与异步串行遍历的版本相比，以上代码并行处理所有数组成员，并通过计数器变量来判断什么时候所有数组成员都处理完毕了。

异常处理

JS自身提供的异常捕获和处理机制——try..catch..，只能用于同步执行的代码。以下是一个例子。

```
function sync(fn) {
    return fn();
}

try {
    sync(null);
    // Do something.
} catch (err) {
    console.log('Error: %s', err.message);
}

-- Console ---------------------------
Error: object is not a function
```

可以看到，异常会沿着代码执行路径一直冒泡，直到遇到第一个try语句时被捕获住。但由于异步函数会打断代码执行路径，异步函数执行过程中以及执行之后产生的异常冒泡到执行路径被打断的位置时，如果一直没有遇到try语句，就作为一个全局异常抛出。以下是一个例子。

```
function async(fn, callback) {
    // Code execution path breaks here.
    setTimeout(function ()  {
        callback(fn());
    }, 0);
}

try {
    async(null, function (data) {
        // Do something.
    });
} catch (err) {
    console.log('Error: %s', err.message);
}

-- Console ---------------------------
/home/user/test.js:4
        callback(fn());
```

```
                  ^

TypeError: object is not a function
    at null._onTimeout (/home/user/test.js:4:13)
    at Timer.listOnTimeout [as ontimeout] (timers.js:110:15)
```

因为代码执行路径被打断了，我们就需要在异常冒泡到断点之前用`try`语句把异常捕获住，并通过回调函数传递被捕获的异常。于是我们可以像下边这样改造上边的例子。

```
function async(fn, callback) {
    // Code execution path breaks here.
    setTimeout(function ()  {
        try {
            callback(null, fn());
        } catch (err) {
            callback(err);
        }
    }, 0);
}

async(null, function (err, data) {
    if (err) {
        console.log('Error: %s', err.message);
    } else {
        // Do something.
    }
});

-- Console ---------------------------
Error: object is not a function
```

可以看到，异常再次被捕获住了。在NodeJS中，几乎所有异步API都按照以上方式设计，回调函数中第一个参数都是`err`。因此我们在编写自己的异步函数时，也可以按照这种方式来处理异常，与NodeJS的设计风格保持一致。

有了异常处理方式后，我们接着可以想一想一般我们是怎么写代码的。基本上，我们的代码都是做一些事情，然后调用一个函数，然后再做一些事情，然后再调用一个函数，如此循环。如果我们写的是同步代码，只需要在代码入口点写一个`try`语句就能捕获所有冒泡上来的异常，示例如下。

```
function main() {
    // Do something.
```

```
    syncA();
    // Do something.
    syncB();
    // Do something.
    syncC();
}

try {
    main();
} catch (err) {
    // Deal with exception.
}
```

但是，如果我们写的是异步代码，就只有呵呵了。由于每次异步函数调用都会打断代码执行路径，只能通过回调函数来传递异常，于是我们就需要在每个回调函数里判断是否有异常发生，于是只用三次异步函数调用，就会产生下边这种代码。

```
function main(callback) {
    // Do something.
    asyncA(function (err, data) {
        if (err) {
            callback(err);
        } else {
            // Do something
            asyncB(function (err, data) {
                if (err) {
                    callback(err);
                } else {
                    // Do something
                    asyncC(function (err, data) {
                        if (err) {
                            callback(err);
                        } else {
                            // Do something
                            callback(null);
                        }
                    });
                }
            });
```

```
        }
    });
}

main(function (err) {
    if (err) {
        // Deal with exception.
    }
});
```

可以看到，回调函数已经让代码变得复杂了，而异步方式下对异常的处理更加剧了代码的复杂度。如果NodeJS的最大卖点最后变成这个样子，那就没人愿意用NodeJS了，因此接下来会介绍NodeJS提供的一些解决方案。

域（Domain）

> **官方文档：** http://nodejs.org/api/domain.html

NodeJS提供了`domain`模块，可以简化异步代码的异常处理。在介绍该模块之前，我们需要首先理解“域”的概念。简单的讲，一个域就是一个JS运行环境，在一个运行环境中，如果一个异常没有被捕获，将作为一个全局异常被抛出。NodeJS通过`process`对象提供了捕获全局异常的方法，示例代码如下

```
process.on('uncaughtException', function (err) {
    console.log('Error: %s', err.message);
});

setTimeout(function (fn) {
    fn();
});

-- Console ------------------------------
Error: undefined is not a function
```

虽然全局异常有个地方可以捕获了，但是对于大多数异常，我们希望尽早捕获，并根据结果决定代码的执行路径。我们用以下HTTP服务器代码作为例子：

```
function async(request, callback) {
    // Do something.
    asyncA(request, function (err, data) {
```

```
        if (err) {
            callback(err);
        } else {
            // Do something
            asyncB(request, function (err, data) {
                if (err) {
                    callback(err);
                } else {
                    // Do something
                    asyncC(request, function (err, data) {
                        if (err) {
                            callback(err);
                        } else {
                            // Do something
                            callback(null, data);
                        }
                    });
                }
            });
        }
    });
}

http.createServer(function (request, response) {
    async(request, function (err, data) {
        if (err) {
            response.writeHead(500);
            response.end();
        } else {
            response.writeHead(200);
            response.end(data);
        }
    });
});
```

以上代码将请求对象交给异步函数处理后，再根据处理结果返回响应。这里采用了使用回调函数传递异常的方案，因此`async`函数内部如果再多几个异步函数调用的话，代码就变成上边这副鬼样子了。为了让代码好看点，我们可以在每处理一个请求时，使用`domain`模块创建一个子域（JS子运行

环境）。在子域内运行的代码可以随意抛出异常，而这些异常可以通过子域对象的error事件统一捕获。于是以上代码可以做如下改造：

```
function async(request, callback) {
    // Do something.
    asyncA(request, function (data) {
        // Do something
        asyncB(request, function (data) {
            // Do something
            asyncC(request, function (data) {
                // Do something
                callback(data);
            });
        });
    });
}

http.createServer(function (request, response) {
    var d = domain.create();

    d.on('error', function () {
        response.writeHead(500);
        response.end();
    });

    d.run(function () {
        async(request, function (data) {
            response.writeHead(200);
            response.end(data);
        });
    });
});
```

可以看到，我们使用.create方法创建了一个子域对象，并通过.run方法进入需要在子域中运行的代码的入口点。而位于子域中的异步函数回调函数由于不再需要捕获异常，代码一下子瘦身很多。

陷阱

无论是通过process对象的uncaughtException事件捕获到全局异常，还是通过子域对象的error事件捕获到了子域异常，在NodeJS官方文档里都强烈建议处理完异常后立即重启程序，而不是让程序

继续运行。按照官方文档的说法，发生异常后的程序处于一个不确定的运行状态，如果不立即退出的话，程序可能会发生严重内存泄漏，也可能表现得很奇怪。

但这里需要澄清一些事实。JS本身的`throw..try..catch`异常处理机制并不会导致内存泄漏，也不会让程序的执行结果出乎意料，但NodeJS并不是存粹的JS。NodeJS里大量的API内部是用C/C++实现的，因此NodeJS程序的运行过程中，代码执行路径穿梭于JS引擎内部和外部，而JS的异常抛出机制可能会打断正常的代码执行流程，导致C/C++部分的代码表现异常，进而导致内存泄漏等问题。

因此，使用`uncaughtException`或`domain`捕获异常，代码执行路径里涉及到了C/C++部分的代码时，如果不能确定是否会导致内存泄漏等问题，最好在处理完异常后重启程序比较妥当。而使用`try`语句捕获异常时一般捕获到的都是JS本身的异常，不用担心上诉问题。

小结

本章介绍了JS异步编程相关的知识，总结起来有以下几点：

- 不掌握异步编程就不算学会NodeJS。
- 异步编程依托于回调来实现，而使用回调不一定就是异步编程。
- 异步编程下的函数间数据传递、数组遍历和异常处理与同步编程有很大差别。
- 使用`domain`模块简化异步代码的异常处理，并小心陷阱。

大示例

学习讲究的是学以致用和融会贯通。至此我们已经分别介绍了NodeJS的很多知识点，本章作为最后一章，将完整地介绍一个使用NodeJS开发Web服务器的示例。

需求

我们要开发的是一个简单的静态文件合并服务器，该服务器需要支持类似以下格式的JS或CSS文件合并请求。

```
http://assets.example.com/foo/??bar.js,baz.js
```

在以上URL中，`??`是一个分隔符，之前是需要合并的多个文件的URL的公共部分，之后是使用`,`分隔的差异部分。因此服务器处理这个URL时，返回的是以下两个文件按顺序合并后的内容。

```
/foo/bar.js
/foo/baz.js
```

另外，服务器也需要能支持类似以下格式的普通的JS或CSS文件请求。

```
http://assets.example.com/foo/bar.js
```

以上就是整个需求。

第一次迭代

快速迭代是一种不错的开发方式，因此我们在第一次迭代时先实现服务器的基本功能。

设计

简单分析了需求之后，我们大致会得到以下的设计方案。

```
           +---------+   +-----------+   +----------+
request -->|  parse  |-->|  combine  |-->|  output  |--> response
           +---------+   +-----------+   +----------+
```

也就是说，服务器会首先分析URL，得到请求的文件的路径和类型（MIME）。然后，服务器会读取请求的文件，并按顺序合并文件内容。最后，服务器返回响应，完成对一次请求的处理。

另外，服务器在读取文件时需要有个根目录，并且服务器监听的HTTP端口最好也不要写死在代码里，因此服务器需要是可配置的。

实现

根据以上设计，我们写出了第一版代码如下。

```
var fs = require('fs'),
    path = require('path'),
    http = require('http');

var MIME = {
    '.css': 'text/css',
    '.js': 'application/javascript'
};

function combineFiles(pathnames, callback) {
    var output = [];

    (function next(i, len) {
        if (i < len) {
            fs.readFile(pathnames[i], function (err, data) {
                if (err) {
                    callback(err);
```

```
                } else {
                    output.push(data);
                    next(i + 1, len);
                }
            });
        } else {
            callback(null, Buffer.concat(output));
        }
    }(0, pathnames.length));
}

function main(argv) {
    var config = JSON.parse(fs.readFileSync(argv[0], 'utf-8')),
        root = config.root || '.',
        port = config.port || 80;

    http.createServer(function (request, response) {
        var urlInfo = parseURL(root, request.url);

        combineFiles(urlInfo.pathnames, function (err, data) {
            if (err) {
                response.writeHead(404);
                response.end(err.message);
            } else {
                response.writeHead(200, {
                    'Content-Type': urlInfo.mime
                });
                response.end(data);
            }
        });
    }).listen(port);
}

function parseURL(root, url) {
    var base, pathnames, parts;

    if (url.indexOf('??') === -1) {
        url = url.replace('/', '/??');
```

```
    }

    parts = url.split('??');
    base = parts[0];
    pathnames = parts[1].split(',').map(function (value) {
        return path.join(root, base, value);
    });

    return {
        mime: MIME[path.extname(pathnames[0])] || 'text/plain',
        pathnames: pathnames
    };
}

main(process.argv.slice(2));
```

以上代码完整实现了服务器所需的功能，并且有以下几点值得注意：

1. 使用命令行参数传递JSON配置文件路径，入口函数负责读取配置并创建服务器。
2. 入口函数完整描述了程序的运行逻辑，其中解析URL和合并文件的具体实现封装在其它两个函数里。
3. 解析URL时先将普通URL转换为了文件合并URL，使得两种URL的处理方式可以一致。
4. 合并文件时使用异步API读取文件，避免服务器因等待磁盘IO而发生阻塞。

我们可以把以上代码保存为`server.js`，之后就可以通过`node server.js config.json`命令启动程序，于是我们的第一版静态文件合并服务器就顺利完工了。

另外，以上代码存在一个不那么明显的逻辑缺陷。例如，使用以下URL请求服务器时会有惊喜。

```
    http://assets.example.com/foo/bar.js,foo/baz.js
```

经过分析之后我们会发现问题出在`/`被自动替换`/??`这个行为上，而这个问题我们可以到第二次迭代时再解决。

第二次迭代

在第一次迭代之后，我们已经有了一个可工作的版本，满足了功能需求。接下来我们需要从性能的角度出发，看看代码还有哪些改进余地。

设计

把`map`方法换成`for`循环或许会更快一些，但第一版代码最大的性能问题存在于从读取文件到输出响应的过程当中。我们以处理`/??a.js,b.js,c.js`这个请求为例，看看整个处理过程中耗时在哪儿。

```
 发送请求          等待服务端响应            接收响应
---------+---------------------+------------>
          --                                  解析请求
            ------                            读取a.js
                  ------                      读取b.js
                        ------                读取c.js
                              --              合并数据
                                --            输出响应
```

可以看到，第一版代码依次把请求的文件读取到内存中之后，再合并数据和输出响应。这会导致以下两个问题：

1. 当请求的文件比较多比较大时，串行读取文件会比较耗时，从而拉长了服务端响应等待时间。
2. 由于每次响应输出的数据都需要先完整地缓存在内存里，当服务器请求并发数较大时，会有较大的内存开销。

对于第一个问题，很容易想到把读取文件的方式从串行改为并行。但是别这样做，因为对于机械磁盘而言，因为只有一个磁头，尝试并行读取文件只会造成磁头频繁抖动，反而降低IO效率。而对于固态硬盘，虽然的确存在多个并行IO通道，但是对于服务器并行处理的多个请求而言，硬盘已经在做并行IO了，对单个请求采用并行IO无异于拆东墙补西墙。因此，正确的做法不是改用并行IO，而是一边读取文件一边输出响应，把响应输出时机提前至读取第一个文件的时刻。这样调整后，整个请求处理过程变成下边这样。

```
发送请求 等待服务端响应 接收响应
---------+----+------------------------------>
          --                                  解析请求
            --                                检查文件是否存在
              --                              输出响应头
                ------                        读取和输出a.js
                      ------                  读取和输出b.js
                            ------            读取和输出c.js
```

按上述方式解决第一个问题后，因为服务器不需要完整地缓存每个请求的输出数据了，第二个问题也迎刃而解。

实现

根据以上设计，第二版代码按以下方式调整了部分函数。

```
function main(argv) {
    var config = JSON.parse(fs.readFileSync(argv[0], 'utf-8')),
        root = config.root || '.',
        port = config.port || 80;

    http.createServer(function (request, response) {
        var urlInfo = parseURL(root, request.url);

        validateFiles(urlInfo.pathnames, function (err, pathnames) {
            if (err) {
                response.writeHead(404);
                response.end(err.message);
            } else {
                response.writeHead(200, {
                    'Content-Type': urlInfo.mime
                });
                outputFiles(pathnames, response);
            }
        });
    }).listen(port);
}

function outputFiles(pathnames, writer) {
    (function next(i, len) {
        if (i < len) {
            var reader = fs.createReadStream(pathnames[i]);

            reader.pipe(writer, { end: false });
            reader.on('end', function() {
                next(i + 1, len);
            });
        } else {
            writer.end();
        }
    }(0, pathnames.length));
}

function validateFiles(pathnames, callback) {
```

```
    (function next(i, len) {
        if (i < len) {
            fs.stat(pathnames[i], function (err, stats) {
                if (err) {
                    callback(err);
                } else if (!stats.isFile()) {
                    callback(new Error());
                } else {
                    next(i + 1, len);
                }
            });
        } else {
            callback(null, pathnames);
        }
    }(0, pathnames.length));
}
```

可以看到，第二版代码在检查了请求的所有文件是否有效之后，立即就输出了响应头，并接着一边按顺序读取文件一边输出响应内容。并且，在读取文件时，第二版代码直接使用了只读数据流来简化代码。

第三次迭代

第二次迭代之后，服务器本身的功能和性能已经得到了初步满足。接下来我们需要从稳定性的角度重新审视一下代码，看看还需要做些什么。

设计

从工程角度上讲，没有绝对可靠的系统。即使第二次迭代的代码经过反复检查后能确保没有bug，也很难说是否会因为NodeJS本身，或者是操作系统本身，甚至是硬件本身导致我们的服务器程序在某一天挂掉。因此一般生产环境下的服务器程序都配有一个守护进程，在服务挂掉的时候立即重启服务。一般守护进程的代码会远比服务进程的代码简单，从概率上可以保证守护进程更难挂掉。如果再做得严谨一些，甚至守护进程自身可以在自己挂掉时重启自己，从而实现双保险。

因此在本次迭代时，我们先利用NodeJS的进程管理机制，将守护进程作为父进程，将服务器程序作为子进程，并让父进程监控子进程的运行状态，在其异常退出时重启子进程。

实现

根据以上设计，我们编写了守护进程需要的代码。

```
var cp = require('child_process');

var worker;

function spawn(server, config) {
    worker = cp.spawn('node', [ server, config ]);
    worker.on('exit', function (code) {
        if (code !== 0) {
            spawn(server, config);
        }
    });
}

function main(argv) {
    spawn('server.js', argv[0]);
    process.on('SIGTERM', function () {
        worker.kill();
        process.exit(0);
    });
}

main(process.argv.slice(2));
```

此外，服务器代码本身的入口函数也要做以下调整。

```
function main(argv) {
    var config = JSON.parse(fs.readFileSync(argv[0], 'utf-8')),
        root = config.root || '.',
        port = config.port || 80,
        server;

    server = http.createServer(function (request, response) {
        ...
    }).listen(port);

    process.on('SIGTERM', function () {
        server.close(function () {
            process.exit(0);
        });
```

```
    });
}
```

我们可以把守护进程的代码保存为`daemon.js`，之后我们可以通过`node daemon.js config.json`启动服务，而守护进程会进一步启动和监控服务器进程。此外，为了能够正常终止服务，我们让守护进程在接收到`SIGTERM`信号时终止服务器进程。而在服务器进程这一端，同样在收到`SIGTERM`信号时先停掉HTTP服务再正常退出。至此，我们的服务器程序就靠谱很多了。

第四次迭代

在我们解决了服务器本身的功能、性能和可靠性的问题后，接着我们需要考虑一下代码部署的问题，以及服务器控制的问题。

设计

一般而言，程序在服务器上有一个固定的部署目录，每次程序有更新后，都重新发布到部署目录里。而一旦完成部署后，一般也可以通过固定的服务控制脚本启动和停止服务。因此我们的服务器程序部署目录可以做如下设计。

```
- deploy/
    - bin/
        startws.sh
        killws.sh
    + conf/
        config.json
    + lib/
        daemon.js
        server.js
```

在以上目录结构中，我们分类存放了服务控制脚本、配置文件和服务器代码。

实现

按以上目录结构分别存放对应的文件之后，接下来我们看看控制脚本怎么写。首先是`start.sh`。

```
#!/bin/sh
if [ ! -f "pid" ]
then
    node ../lib/daemon.js ../conf/config.json &
    echo $! > pid
fi
```

然后是`killws.sh`。

```
#!/bin/sh
if [ -f "pid" ]
then
    kill $(tr -d '\r\n' < pid)
    rm pid
fi
```

于是这样我们就有了一个简单的代码部署目录和服务控制脚本，我们的服务器程序就可以上线工作了。

后续迭代

我们的服务器程序正式上线工作后，我们接下来或许会发现还有很多可以改进的点。比如服务器程序在合并JS文件时可以自动在JS文件之间插入一个;来避免一些语法问题，比如服务器程序需要提供日志来统计访问量，比如服务器程序需要能充分利用多核CPU，等等。而此时的你，在学习了这么久NodeJS之后，应该已经知道该怎么做了。

小结

本章将之前零散介绍的知识点串了起来，完整地演示了一个使用NodeJS开发程序的例子，至此我们的课程就全部结束了。以下是对新诞生的NodeJSer的一些建议。

- 要熟悉官方API文档。并不是说要熟悉到能记住每个API的名称和用法，而是要熟悉NodeJS提供了哪些功能，一旦需要时知道查询API文档的哪块地方。
- 要先设计再实现。在开发一个程序前首先要有一个全局的设计，不一定要很周全，但要足够能写出一些代码。
- 要实现后再设计。在写了一些代码，有了一些具体的东西后，一定会发现一些之前忽略掉的细节。这时再反过来改进之前的设计，为第二轮迭代做准备。
- 要充分利用三方包。NodeJS有一个庞大的生态圈，在写代码之前先看看有没有现成的三方包能节省不少时间。
- 不要迷信三方包。任何事情做过头了就不好了，三方包也是一样。三方包是一个黑盒，每多使用一个三方包，就为程序增加了一份潜在风险。并且三方包很难恰好只提供程序需要的功能，每多使用一个三方包，就让程序更加臃肿一些。因此在决定使用某个三方包之前，最好三思而后行。