

第1章 Node.js入门

Node.js开发实战教程

111001110

```
00001111011101010111010010111000  
01110100101111110100001111010010  
10100111010010101010101001010100  
101010101001000000011001010010
```



内容导航

Contents



1.1 Node.js 简介

- ◆ 什么是 Node.js
- ◆ Node.js 的特点
- ◆ Node.js 的应用场合

1.2 部署 Node.js 开发环境

1.3 开始开发 Node.js 应用程序

Section

00

【学习目标】

- (1) 了解什么是Node.js;
- (2) 熟悉Node.js的特点和应用场合。

Section

01

【什么是Node.js】

- Node.js简称Node，是一个可以使JavaScript运行在服务器端的开发平台。
- JavaScript本是一种Web前端语言，Node.js让JavaScript成为服务器端脚本语言。
- Node.js选择JavaScript作为实现语言的原因：
 - JavaScript满足CommonJS标准，符合事件驱动，用户较多且门槛较低；
 - Chrome的V8引擎具有出色的性能。
- Node.js将V8引擎封装起来，作为服务器运行平台，以执行JavaScript编写的后端脚本程序。

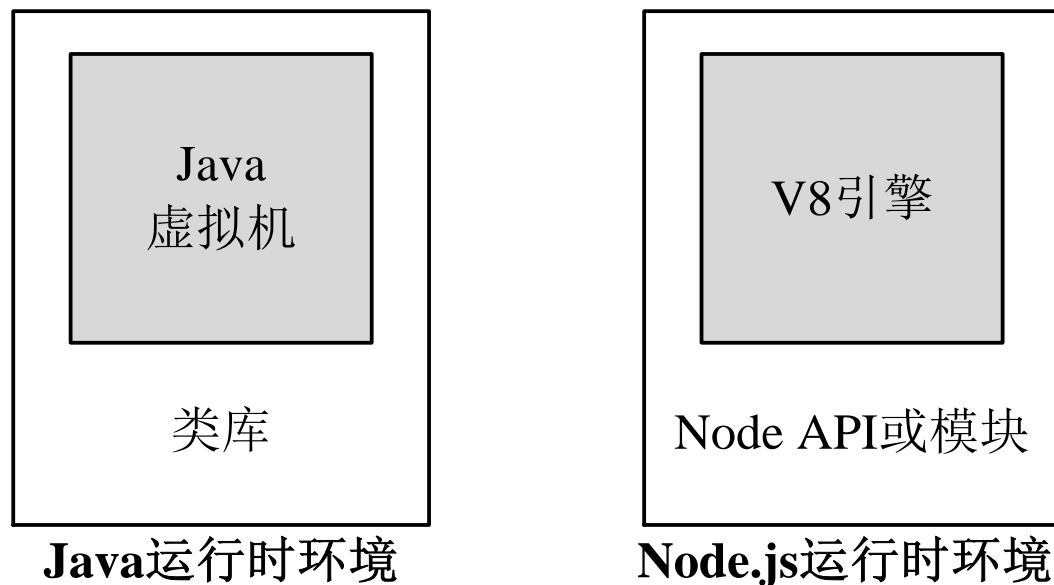
Section

01

【什么是Node.js】

- Node.js运行时环境包含执行JavaScript程序所需的一切条件。该引擎会将JavaScript代码转换为更快的机器码。
- Node.js进一步提升JavaScript的能力，使JavaScript可以访问文件、读取数据库、访问进程，从而胜任后端任务。
- 使用Node.js 的最大优点是开发人员可以在客户端和服务端编写JavaScript，打通了前后端。
- Node.js 发展迅速，目前已成为JavaScript服务器端运行平台的事实标准。

Node.js与Java运行时环境对比



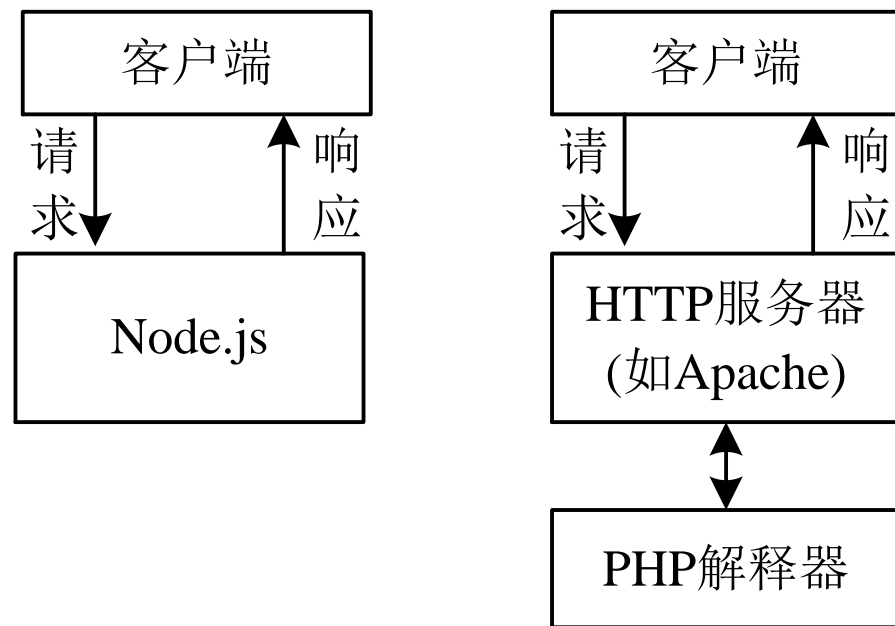
Section

01

【什么是Node.js】

- Node.js是跨平台的，能运行在Windows、macOS和Linux平台上。
- Node.js除了自己的标准类库之外，还可使用大量的第三方模块系统来实现代码的分享和重用。
- 与其他后端脚本语言不同的是，Node.js内置了处理网络请求和响应的函数库，也就是自备了HTTP服务器，所以不需要额外部署HTTP服务器。

Node.js与PHP对HTTP请求的处理



Section

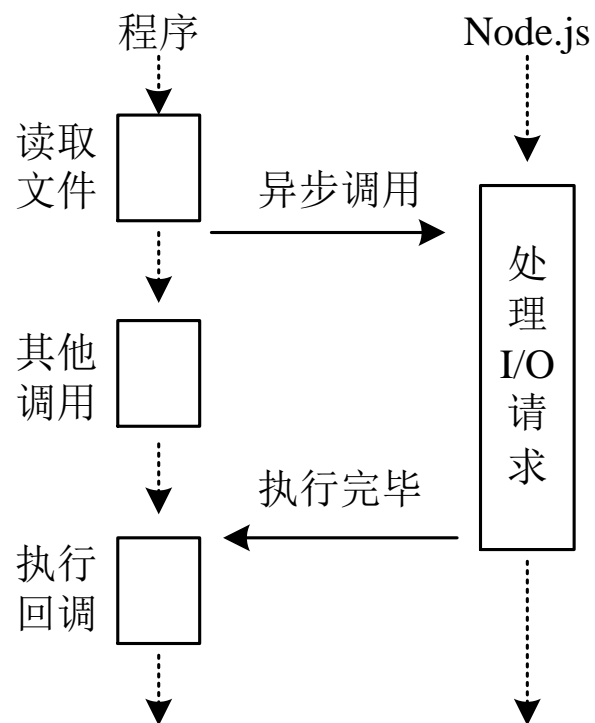
02

【Node.js的特点】

▶▶▶ 非阻塞I/O

- 非阻塞I/O又称异步式I/O，是Node.js的重要特点。
- 阻塞I/O是指线程在执行过程中遇到I/O操作时，操作系统会撤销该线程的CPU控制权，使其暂停执行，处于等待状态，同时将资源转让给其他线程。
- 非阻塞I/O是指当线程遇到I/O操作时，不会以阻塞方式等待I/O操作完成或数据返回，而只是将I/O请求转发给操作系统，继续执行下一条指令。

Node.js的非阻塞I/O



Section

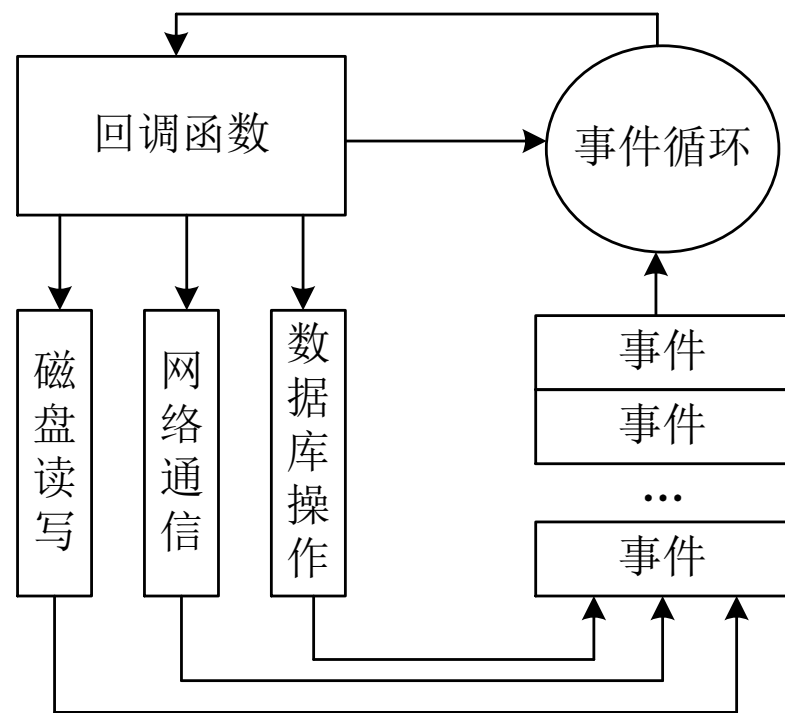
02

【Node.js的特点】

▶▶▶ 事件驱动

- 非阻塞I/O是一种异步方式的I/O，与事件驱动密不可分。
- 事件驱动以事件为中心，Node.js将每一个任务都当成事件来处理。Node.js在执行过程中会维护一个事件队列，需执行的每个任务都会加入事件队列并提供一个包含处理结果的回调函数。
- 在事件驱动模型中，会生成一个事件循环线程来监听事件，不断地检查是否有未处理的事件。
- Node.js的异步机制是基于事件的，所有磁盘I/O、网络通信、数据库查询事件都以非阻塞的方式请求，返回的结果由事件循环线程来处理。

Node.js事件循环机制



Section

02

【Node.js的特点】

▶▶▶ 单线程

- Node.js的应用程序是单进程、单线程的，但是通过事件和回调支持并发，性能变得非常高。
- 在阻塞模式下，一个线程只能处理一项任务，要想提高吞吐量必须使用多线程。
- 在非阻塞模式下，线程不会被I/O操作阻塞，该线程所使用的CPU核心利用率永远是100%，I/O操作以事件的方式通知操作系统。
- Node.js在主线程中维护一个事件队列，当接收到请求后，就将该请求作为一个事件放入该队列中，然后继续接收其他请求。
- Node.js内部通过线程池来完成非阻塞I/O操作，Node.js的单线程是指对JavaScript层面的任务处理是单线程的，而Node.js本身是一个多线程平台。



Node.js采用非阻塞I/O与事件驱动相结合的编程模式，与传统同步I/O线性编程思维有很大的不同，Node.js程序的控制很大程度要依靠事件和回调函数，这不符合开发人员的常规线性思路，需要将一个完整的逻辑拆分为若干单元（事件），从而增加了开发和调试的难度。

Section

03

【Node.js的应用场合】

▶▶▶ 适合用Node.js的场合

- REST API: REST API是一种前后端分离的应用程序架构。
- 单页Web应用: 加载单个HTML页面, 并在用户与应用程序交互时动态更新该页面的Web应用程序。
- 统一Web应用的UI层: Node.js是面向服务的架构, 其能够更好地实现前后端的依赖分离, 可以将所有的关键业务逻辑都封装成REST API, UI层只需要考虑如何用这些API构建具体的应用。
- 准实时系统: 如聊天系统、微博系统、博客系统的准实时社交系统, 特点是轻量级、高流量, 没有复杂的计算逻辑。
- 游戏服务器: 程序员不必使用C语言就能开发游戏的服务器程序。
- 微服务架构: Node.js也可用于实现基于微服务架构的应用。

Section

03

【Node.js的应用场合】

▶▶▶ 不适合用Node.js的场合

- 数据加密和解密。
- 数据压缩和解压。
- 模板渲染。

▶▶▶ 弥补Node.js不足的解决方案

存在的问题	解决方案
CPU密集型任务偏向于CPU计算操作，需要Node.js直接处理，在事件队列中，如果前面的CPU计算任务没有完成，那么后面的任务就会被阻塞，出现响应慢的情况，使得后续I/O操作无法发起	将大型运算任务分解为多个小任务，适时释放CPU计算空间资源，以免阻塞I/O调用的发起
单线程无法利用多核CPU。多CPU或多核CPU的服务器当Node.js被CPU密集型任务占用，导致其他任务被阻塞时，其他CPU核心处于闲置状态，从而造成资源浪费；Node.js程序一旦在某个环节崩溃，整个系统都会崩溃，这会影响其可靠性	<ul style="list-style-type: none">（1）部署Nginx反向代理和负载均衡，开启多个进程，绑定多个端口（2）使用cluster模块构建应用集群，启动多个Node.js实例，开启多个进程以监听同一个端口

内容导航

Contents



1.1 Node.js 简介

1.2 部署Node.js开发环境

- ◆ 在 Windows 系统上安装 Node.js
- ◆ 在 Linux 系统上安装 Node.js
- ◆ 管理 Node.js 版本
- ◆ 交互式运行环境——REPL
- ◆ 安装开发工具 Visual Studio Code

1.3 开始开发Node.js应用程序

Section

00

【学习目标】

- (1) 掌握在Windows和Linux平台上安装Node.js的方法;
- (2) 安装开发工具Visual Studio Code, 熟悉其基本用法。

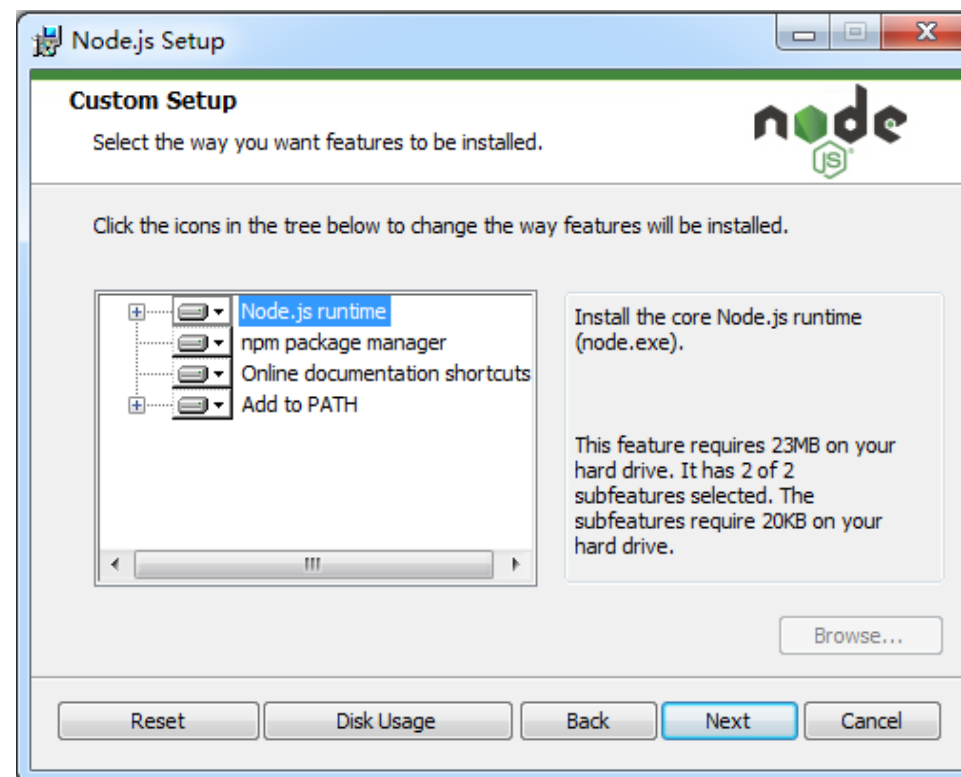
Section

01

【在Windows系统上安装Node.js】

- 两种文件格式的安装包
 - Windows安装包 (.msi)
 - Windows二进制文件 (.exe) 安装包
- 检查Node.js版本
`node --version`

自定义安装



Section

02

【在Linux系统上安装Node.js】

▶▶▶ Linux操作系统上安装Node.js的方式

- 源代码
- 二进制发行版
- 软件源安装
- n模块
- nvm

Section

02

【在Linux系统上安装Node.js】

▶▶▶ 使用二进制发行版安装Node.js

- 从Node.js官网下载二进制发行版的安装包Linux Binaries (x64)。
- 根据官网说明执行安装步骤。

(1) 创建Node.js安装目录，这里准备安装到/usr/local/lib/nodejs目录下。

```
sudo mkdir -p /usr/local/lib/nodejs
```

(2) 将下载的二进制发行版安装包解压到该目录。

```
sudo tar -xJvf node-v10.16.0-linux-x64.tar.xz -C /usr/local/lib/nodejs
```

(3) 编辑环境变量配置文件 ~/.profile，将以下内容添加到该文件末尾并保存该文件。

```
VERSION=v10.16.0
```

```
DISTRO=linux-x64
```

```
export PATH=/usr/local/lib/nodejs/node-v10.16.0-linux-x64/bin:$PATH
```

Section

02

【在Linux系统上安装Node.js】

▶▶▶ 使用二进制发行版安装Node.js

(4) 运行该配置文件，以使新的环境变量配置生效。

```
[zxp@host-test ~]$ . ~/.profile
```

(5) 测试Node.js安装是否成功。

先查看node版本：

```
[zxp@host-test ~]$ node -v  
v10.16.0
```

再查看npm的版本：

```
[zxp@host-test ~]$ npm -v  
6.9.0
```

(6) 要使其他用户也能运行和使用Node.js，需要创建以下软链接：

```
sudo ln -s /usr/local/lib/nodejs/node-v10.16.0-linux-x64/bin/node /usr/bin/node  
sudo ln -s /usr/local/lib/nodejs/node-v10.16.0-linux-x64/bin/npm /usr/bin/npm  
sudo ln -s /usr/local/lib/nodejs/node-v10.16.0-linux-x64/bin/npm /usr/bin/npm  
sudo ln -s /usr/local/lib/nodejs/node-v10.16.0-linux-x64/bin/npm /usr/bin/npm
```

Section

02

【在Linux系统上安装Node.js】

▶▶▶ 软件源安装Node.js

```
zxp@host-b: ~ $ sudo apt-get install nodejs
zxp@host-b: ~ $ sudo apt install nodejs:i386
zxp@host-b: ~ $ node -v
v8.10.0
zxp@host-b: ~ $ sudo apt install npm
zxp@host-b: ~ $ npm -v
3.5.2
```


Section

02

【在Linux系统上安装Node.js】

▶▶▶ 使用n模块升级和管理Node.js版本

(1) 需要升级将npm升级到目前最新版本:

```
sudo npm install npm@latest -g
```

(2) 以全局方式（使用选项-g）安装n模块:

```
zxp@host-b: ~$ sudo npm install -g n
```

- n模块常用命令

n: 不带任何选项或参数会列出已安装的所有Node.js版本。

n latest: 安装最新的Node.js版本。

n stable: 安装最新的Node.js稳定版本。

n lts: 安装最新的长期支持的Node.js版本。

n rm [版本号]: 删除指定的Node.js版本。

n -h: 给出帮助信息。

Section


03


【管理Node.js版本】

有时候可能要同时开发多个项目，而每个项目所使用的Node.js版本不同，或者要用更新的版本进行试验和学习，在一台计算机上处理这种情况比较麻烦，这时就可以使用多版本Node.js管理工具。


- nvm是专门的Node版本管理器。
- nvm在Windows平台上的名称为nvm-windows。
- 以在windows系统上安装和使用nvm为例讲解。
- <https://github.com/coreybutler/nvm-windows/releases>


 [nvm-noinstall.zip](#)

 [nvm-noinstall.zip.checksum.txt](#)

 [nvm-setup.zip](#)

 [nvm-setup.zip.checksum.txt](#)

 [Source code \(zip\)](#)

 [Source code \(tar.gz\)](#)

- nvm-noinstall.zip: 这个是绿色免安装版本，但是使用之前需要配置
- nvm-setup.zip: 这是一个安装包，下载之后点击安装，无需配置就可以使用，方便。
- Source code(zip): zip压缩的源码
- Source code(tar.gz): tar.gz的源码，一般用于*nix系统

Section

03

【管理Node.js版本】

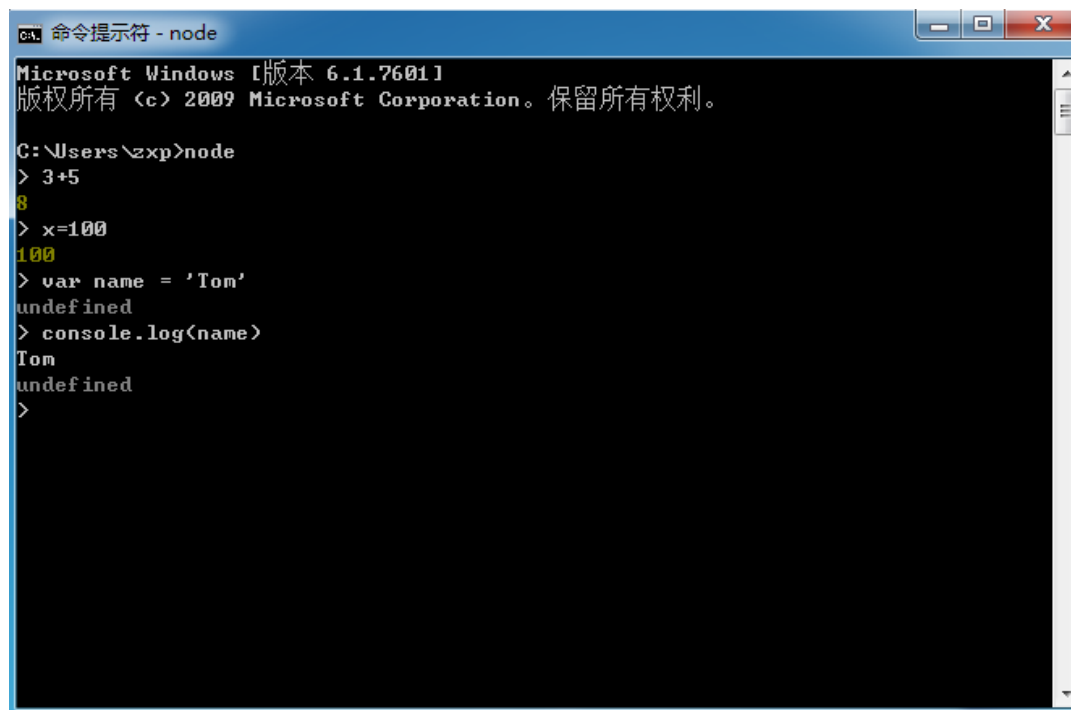
- `nvm arch [32|64]` : 显示node是运行在32位还是64位模式。指定32或64来覆盖默认体系结构。
- `nvm install <version> [arch]`: 该可以是node.js版本或最新稳定版本latest。（可选[arch]）指定安装32位或64位版本（默认为系统arch）。设置[arch]为all以安装32和64位版本。在命令后面添加`--insecure`，可以绕过远端下载服务器的SSL验证。
- `nvm list [available]`: 列出已经安装的node.js版本。可选的available，显示可下载版本的部分列表。这个命令可以简写为`nvm ls [available]`。
- `nvm on`: 启用node.js版本管理。
- `nvm off`: 禁用node.js版本管理(不卸载任何东西)
- `nvm proxy [url]`: 设置用于下载的代理。留[url]空白，以查看当前的代理。设置[url]为none删除代理。
- `nvm node_mirror [url]`: 设置node镜像，默认为<https://nodejs.org/dist/>。我建议设置为淘宝的镜像<https://npm.taobao.org/mirrors/node/>
- `nvm npm_mirror [url]`: 设置npm镜像，默认为<https://github.com/npm/npm/archive/>。我建议设置为淘宝的镜像<https://npm.taobao.org/mirrors/npm/>
- `nvm uninstall <version>`: 卸载指定版本的nodejs。
- `nvm use [version] [arch]`: 切换到使用指定的nodejs版本。可以指定32/64位[arch]。`nvm use <arch>`将继续使用所选版本，但根据提供的值切换到32/64位模式的<arch>
- `nvm root [path]`: 设置 nvm 存储node.js不同版本的目录 ,如果未设置，将使用当前目录。
- `nvm version`: 显示当前运行的nvm版本，可以简写为`nvm v`

Section

04

【交互式运行环境——REPL】

- 功能——JavaScript脚本的简单调试
- 主要执行的操作
 - 读取用户输入，解析输入的JavaScript数据结构并将其存储在内存中。
 - 执行输入的数据结构，打印（输出）结果。
 - 循环操作以上步骤直到用户按下两次 <Ctrl> + <C> 组合键。
- 进入命令行界面，执行node命令即可启动Node终端，出现 “>” 提示符表示进入REPL命令行交互界面。



```
命令提示符 - node
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

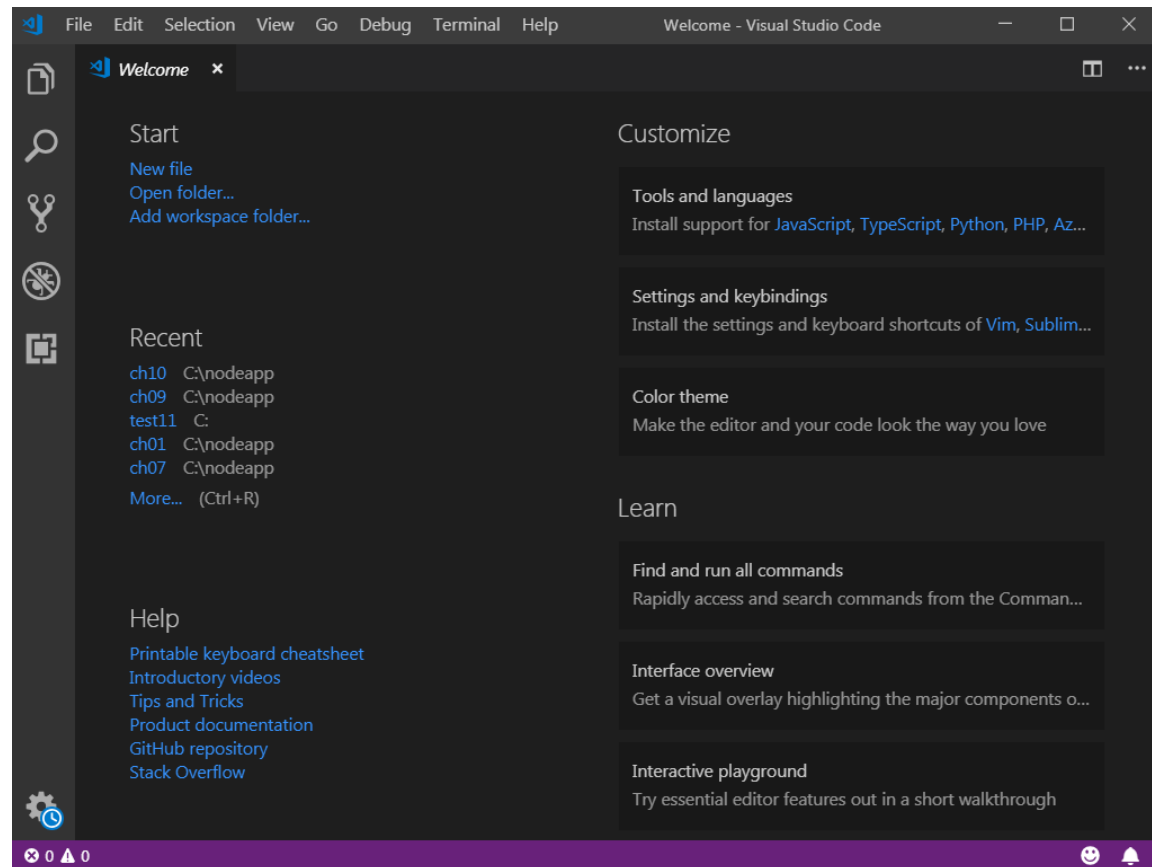
C:\Users\zxp>node
> 3+5
8
> x=100
100
> var name = 'Tom'
undefined
> console.log(name)
Tom
undefined
>
```

Section

05

【安装开发工具Visual Studio Code】

- Visual Studio Code用于集中进行编码、运行和调试。
- 以在Windows 7 64位系统上安装Visual Studio Code为例。
- 从微软官网上下载Visual Studio Code 64位Windows版本的安装包进行安装。



内容导航

Contents



1.1 Node.js 简介

1.2 部署Node.js开发环境

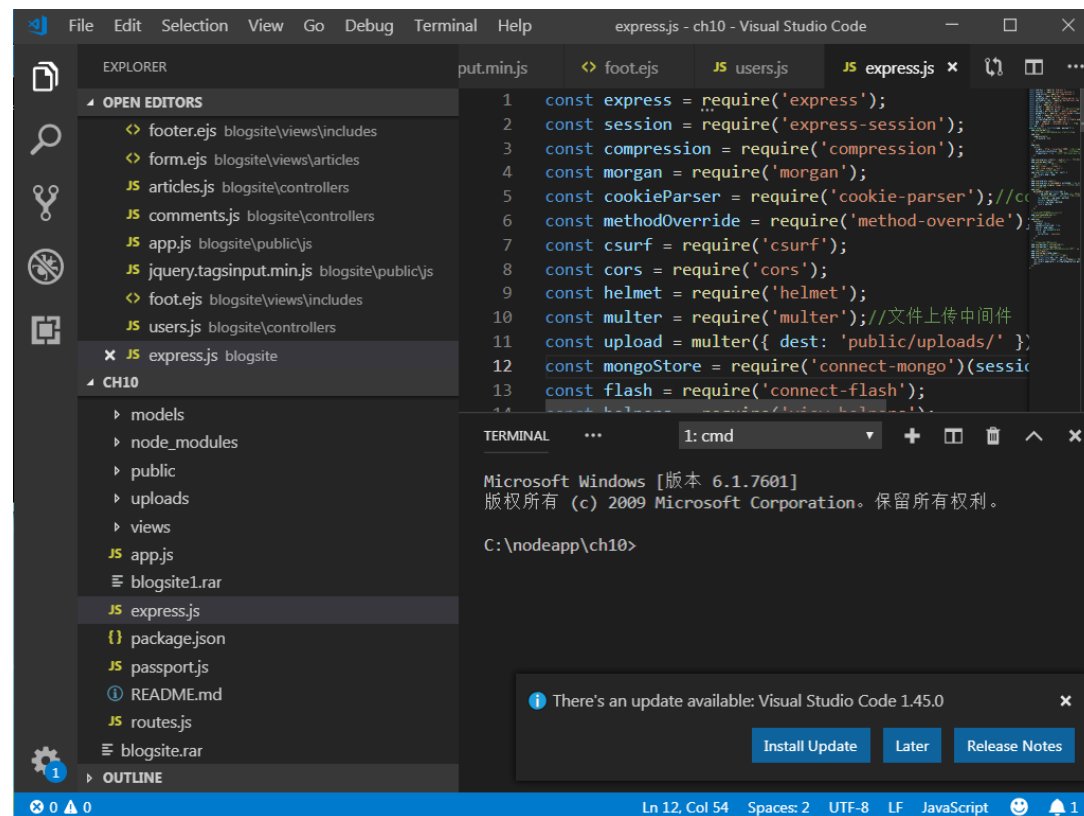
1.3 开始开发Node.js应用程序

- ◆ 构建第一个 Node.js应用程序
- ◆ 运行 Node.js 程序
- ◆ 调试 Node.js 程序

Section
00

【学习目标】

- (1) 熟悉开发工具Visual Studio Code的基本使用;
- (2) 掌握Node.js应用程序的编写、运行和调试的基本方法。



Section

01

【实战演练——构建第一个Node.js应用程序】

▶▶▶ 编写程序

//导入http模块

```
const http = require('http');
const httpServer = http.createServer(function (req, res) {
  //设置响应头信息
  res.writeHead(200, {'Content-Type': 'text/plain'});
  // 发送响应数据 "Hello World!"
  res.end('Hello World!\n');
});
httpServer.listen(8080,function(){
  //向终端输出如下信息
  console.log('服务器正在8080端口上监听! ');
});
```

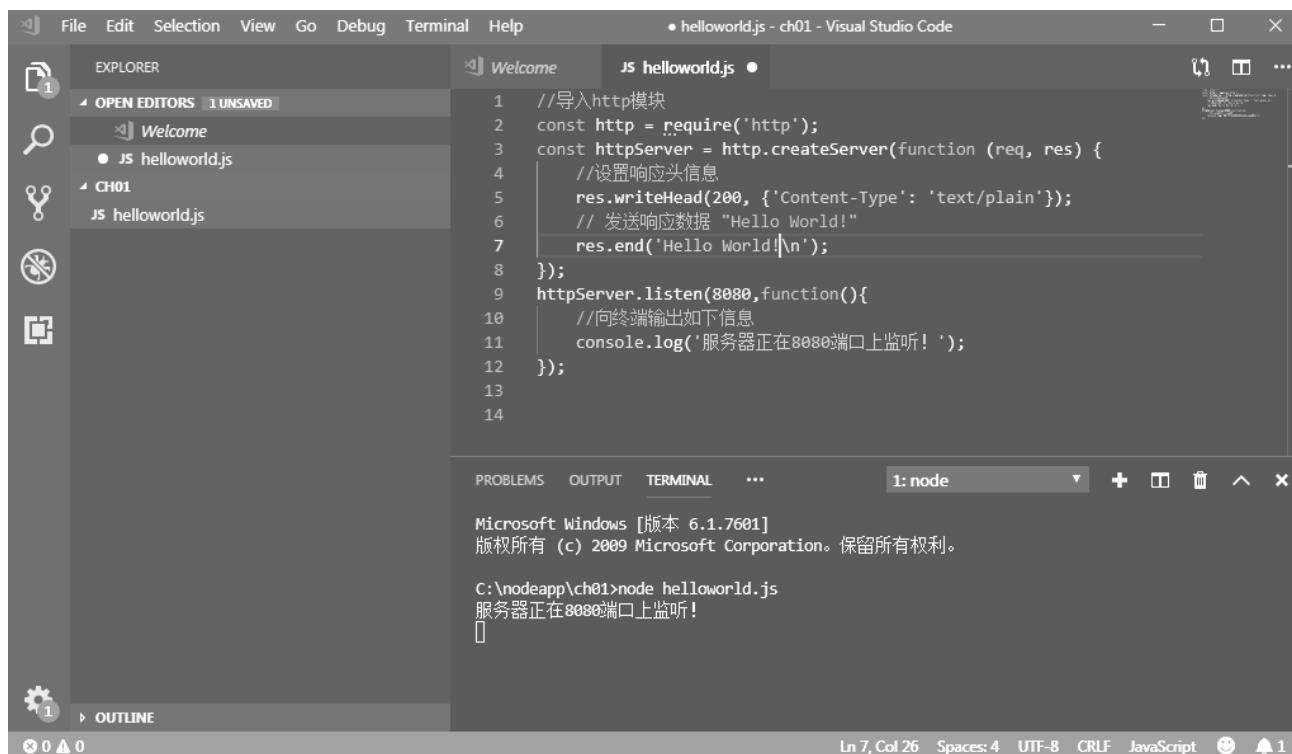

Section

01

【实战演练——构建第一个Node.js应用程序】

▶▶▶ 测试程序

➤ 在终端窗口中运行程序进行测试



The screenshot shows the Visual Studio Code interface with a file explorer on the left, a code editor in the center, and a terminal window at the bottom. The code editor displays a JavaScript file named `helloworld.js` with the following content:

```
1 //导入http模块
2 const http = require('http');
3 const httpServer = http.createServer(function (req, res) {
4   //设置响应头信息
5   res.writeHead(200, {'Content-Type': 'text/plain'});
6   // 发送响应数据 "Hello World!"
7   res.end('Hello World!\n');
8 });
9 httpServer.listen(8080,function(){
10   //向终端输出如下信息
11   console.log('服务器正在8080端口上监听! ');
12 });
13
14
```

The terminal window shows the command `node helloworld.js` being executed, and the output is `服务器正在8080端口上监听!`.

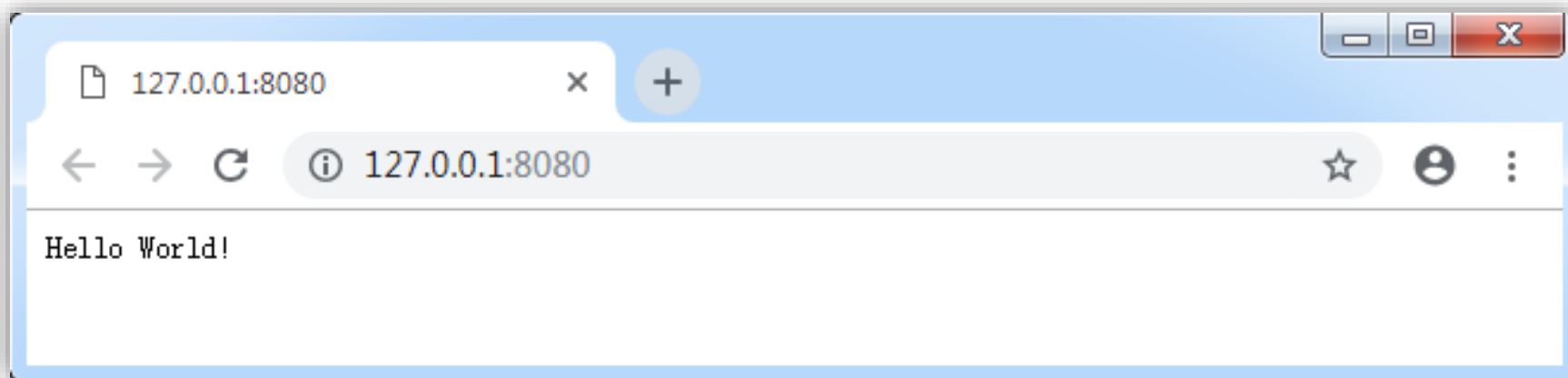
Section

02

【实战演练——构建第一个Node.js应用程序】

▶▶▶ 测试程序

➤ 通过浏览器访问Web应用程序进行测试



Section 01

【实战演练——构建第一个Node.js应用程序】

程序结构分析

部分	源代码	说明
1	<pre>const http = require('http');</pre>	导入模块。导入Node.js自带的http模块，并将实例化的HTTP组件赋值给变量http。模块是Node.js程序组织可重用代码的方式，可使用require()方法来载入模块
2	<pre>const httpServer = http.createServer(function (req, res) { res.writeHead(200, {'Content-Type': 'text/plain'}); res.end('Hello World!\n'); });</pre>	创建 HTTP 服务器。调用 http 模块提供的 http.createServer()方法创建服务器，使用一个回调函数作为参数，该回调函数又接受两个参数，分别是代表客户端的请求对象和向客户端发送的响应对象，所有请求和响应都由此回调函数处理
3	<pre>httpServer.listen(8080,function(){ console.log('服务器正在8080端口上监听！ '); });</pre>	启动HTTP服务器，并设置监听器的端口号。http.createServer()方法返回一个HTTP服务器对象，它使用listen()方法启动HTTP服务器以监听连接、指定端口号。该方法包含一个回调函数参数，用于设置启动HTTP服务器之后的操作

Section

02

【运行Node.js程序】

▶▶▶ 使用node命令运行Node.js程序

- node命令语法

```
node [Sections] [ -e script | script.js | - ] [arguments]
```

- 运行当前目录下的index.js脚本文件，可以使用点号代替： node .
- 按下<Ctrl> + <C>组合键终止正在运行的Node.js程序。
- 选项-e (--eval) 表示直接执行某语句：

```
node -e "console.log('Hello World!');"
```

▶▶▶ 使用npm命令运行Node.js程序

- 在当前目录下运行npm init --yes生成配置文件package.json，然后进行修改后执行npm start。

```
{  
  "scripts": {  
    "start": "node demo.js",  
    "test": "node test.js"  
  }  
}
```

Section

02

【运行Node.js程序】

▶▶▶ 使用nodemon监视文件改动并自动重启Node.js程序

- 安装该模块

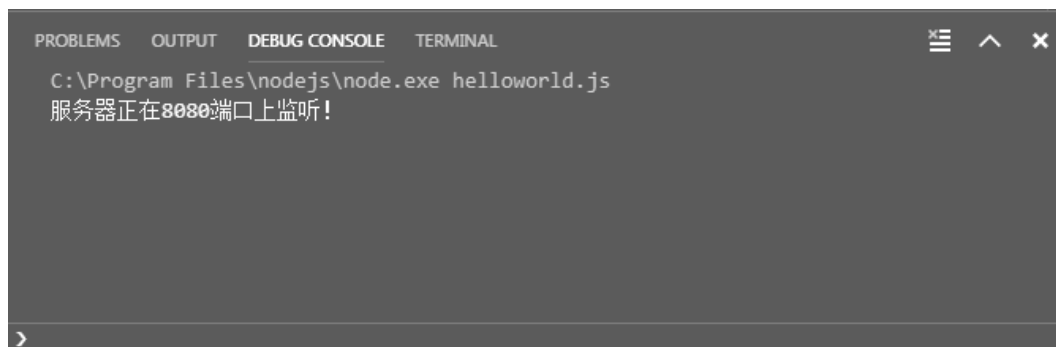
```
npm i nodemon -g
```

- 监控并运行Node.js程序

```
nodemon 文件路径
```

▶▶▶ 在Visual Studio Code中运行Node.js程序

- 新开一个终端窗口，在其中执行node命令
- 以非调试方式启动当前的Node.js脚本文件



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
C:\Program Files\nodejs\node.exe helloworld.js
服务器正在8080端口上监听!
```

Section

03

【调试Node.js程序】

▶▶▶ 使用日志工具进行调试

- 使用console.log()方法检查变量或字符串的值，记录脚本调用的函数，或记录来自第三方服务的响应。
- 使用console.warn()或console.error()方法记录警告或错误信息。

▶▶▶ 使用Node.js内置调试器

- Node.js内置一个进程外的调试实用程序，可通过V8检查器和内置调试客户端访问。
- 执行node命令时加上inspect参数，指定要调试的脚本的路径。
- Node.js调试脚本示例

```
global.x = 5;
setTimeout(() => { //此处用到的回调函数的形式是箭头函数，() =>相当于function()
  debugger;
  console.log('world');
}, 1000);
console.log('hello');
```

Section

03

【调试Node.js程序】

▶▶▶ 在Visual Studio Code中调试Node.js程序

➤ 调试步骤

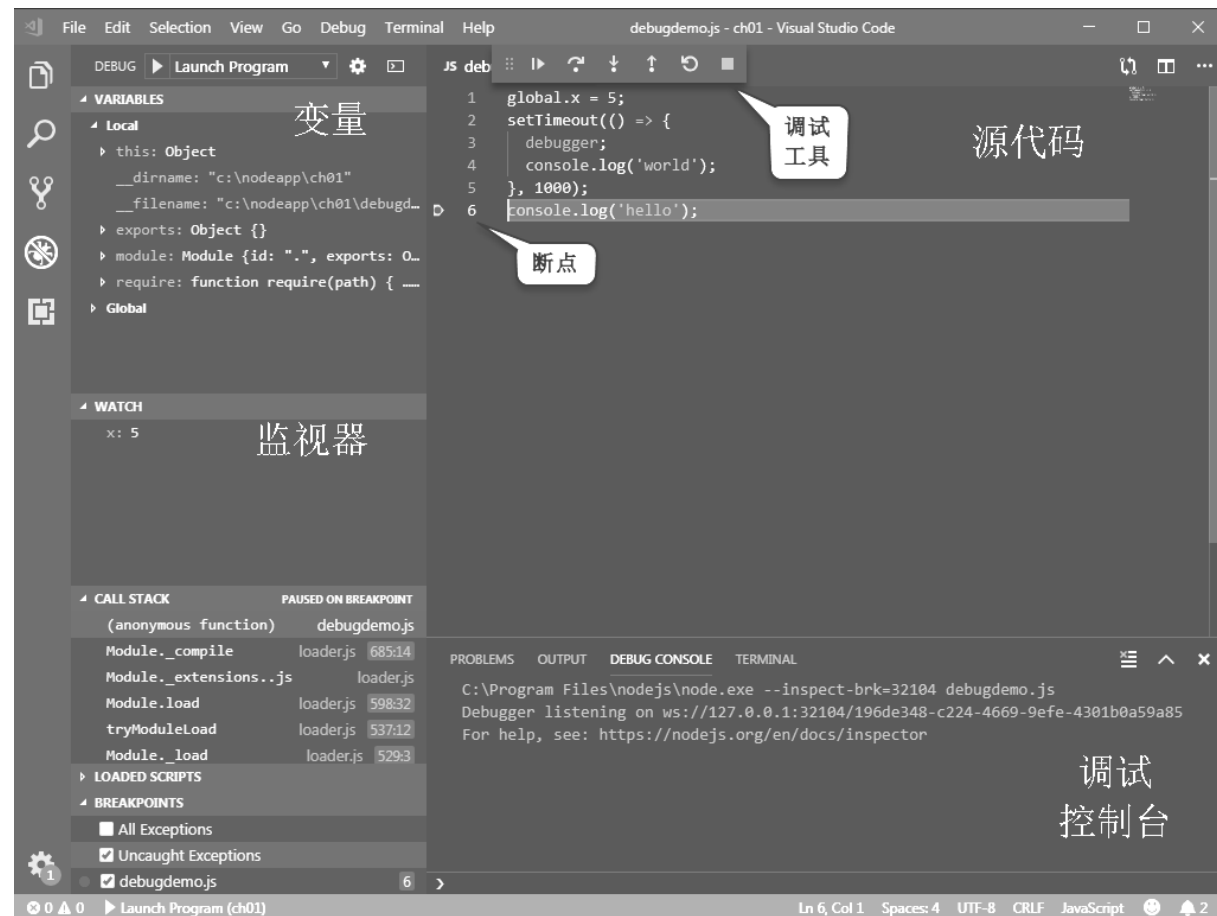
(1) 设置调试配置文件。关键是设置program属性，使其指向要运行的脚本文件。

(2) 设置断点。

(3) 根据需要设置监视器。

(4) 启动该脚本的调试。

Node.js脚本的调试

调试
控制台

Section

03

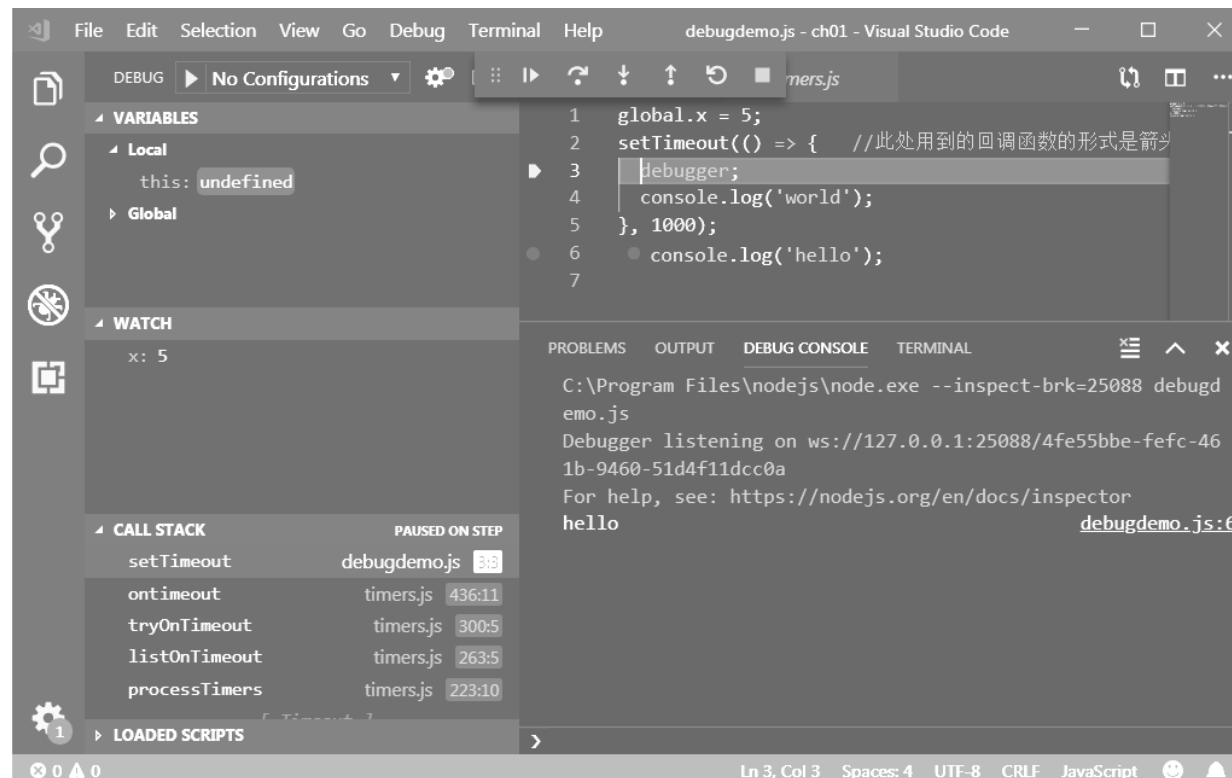
【调试Node.js程序】

▶▶▶ 在Visual Studio Code中调试Node.js程序

➤ 调试器3种单步执行方式

- Step Into (<F11> 键)：单步执行，遇到子函数就进入该函数并且继续单步执行。
- Step Out (<Shift>+<F11> 组合键)：当单步执行到子函数内时，使用它执行完子函数余下部分，并返回上一层函数。
- Step Over (<F10> 键)：在单步执行时，在函数内遇到子函数时不会进入子函数内单步执行，而是将子函数整个执行完毕并返回下一条语句。

单步执行

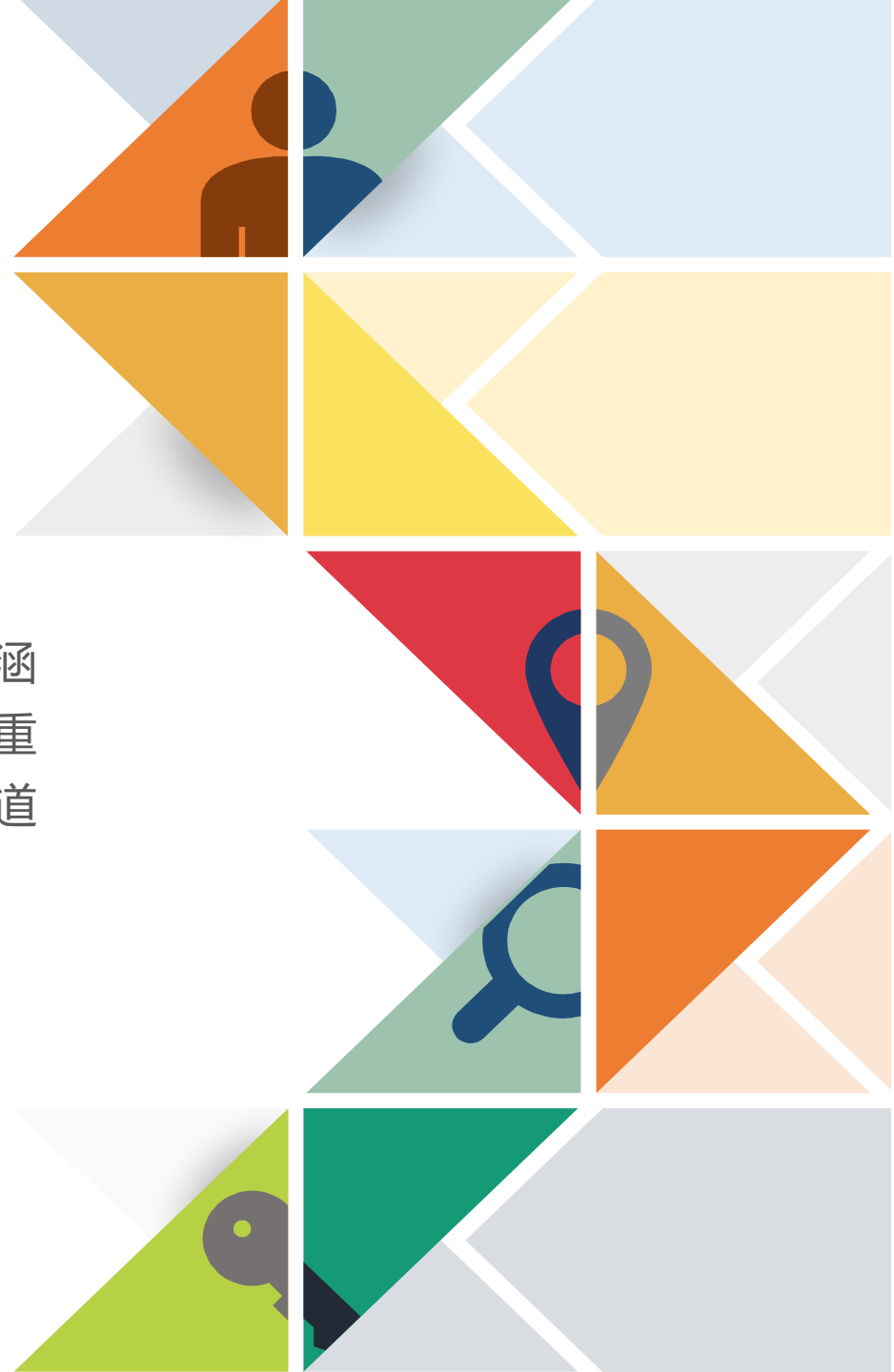




本章小结

Summary

本章的内容主要是Node.js的基础知识和基本操作技能，涵盖了Node.js的简介、Node.js开发环境搭建与调试。读者应重点了解Node.js的非阻塞I/O、事件驱动和单线程的特点，知道Node.js适合哪些应用场合。





Thank you