第 2 章　尝试构建第一个 Electron 程序

学习完第 1 章的内容，你应该对 Electron 的背景以及它的基本原理有了一个初步的了解，现在是时候开始进入实战环节了。在本章的内容中，我们会通过从 0 到 1 搭建一个简单的 Electron 应用，让大家对 Electron 的开发环境、基本项目结构以及主进程与渲染进程的使用有一个基本的了解。我们期望你在学习完本章后，已经具备了开发一个 Electron 应用所需要的最小知识集合，从而有能力自己动手开发一个简单的 Electron 应用。由于目前用户量最大的操作系统还是微软的 Windows 操作系统，所以后面所有的示例都是在 Windows 环境上开发和运行的。使用 Mac 或 Linux 作为开发环境的小伙伴不用担心，得益于 Electron 的跨平台性，除了个别与操作系统特性强关联的接口外，本章节中的 Demo 程序源码都是可以完整在 Mac 或 Linux 上开发和运行的。

接下来我们就开始尝试构建第一个 Electron 程序吧！在这之前，我们需要搭建好开发 Electron 应用所需要的环境。

2.1　Node.js 环境搭建

由于我们需要使用 npm 包管理器来安装 Electron，而 npm 依赖于 Node.js，所以我们需要先在计算机中安装 Node.js 环境。

2.1.1 下载 Node.js

打开 Node.js 的官网，我们可以看到官方提供了两个版本，一个是 14.15.3 LTS，另一个是 15.5.0 Current，如图 2-1 所示。这里我们只需要下载 14.15.3 LTS 版本即可。

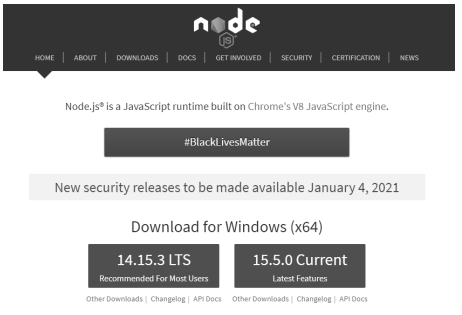

图 2-1　Node.js 官网主页

2.1.2　安装 Node.js

当安装包下载完成后，双击安装程序进入安装界面，如图 2-2 所示。

单击“Next”按钮，进入权限许可确认的界面，如图 2-3 所示。

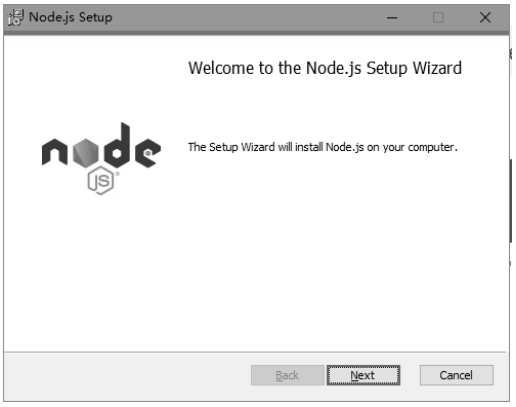

图 2-2　Node.js 安装界面第一步

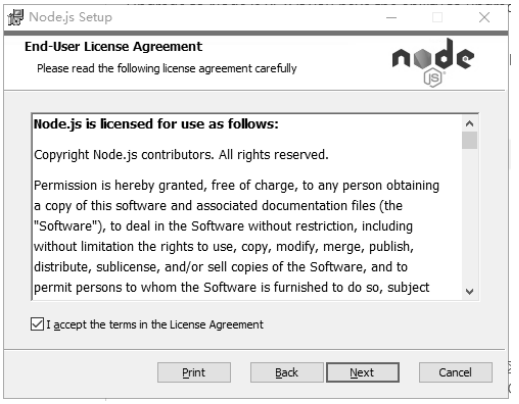

图 2-3　权限许可界面

此处阅读完 Node.js 的权限说明后，选中“I aaept the terms in the License Agrement”复选框，然后继续单击“Next”按钮进入安装目录选择界面，如图 2-4 所示。

默认情况下，Node.js 会安装到计算机系统盘的 Program Files 文件夹下，并在该文件夹中创建一个文件名为 nodejs 的目录，Node.js 的所有文件都会存放在这个目录中。一般情况下，你无须更改这个默认路径。如果你对安装目录有特殊的需求，可以通过单击“Change”按钮更改安装路径。笔者当前所使用的计算机的系统盘为 C 盘，所以我们能从图中看到默认安装目录是：C:\Program Files\nodejs\。

单击“Next”按钮进入选择安装 Node.js 相关组件的界面，如图 2-5 所示。

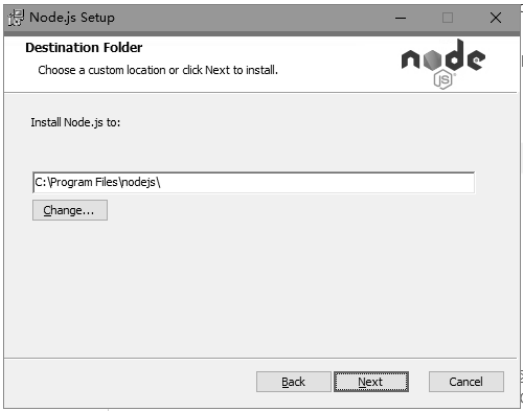

图 2-4　选择安装路径界面

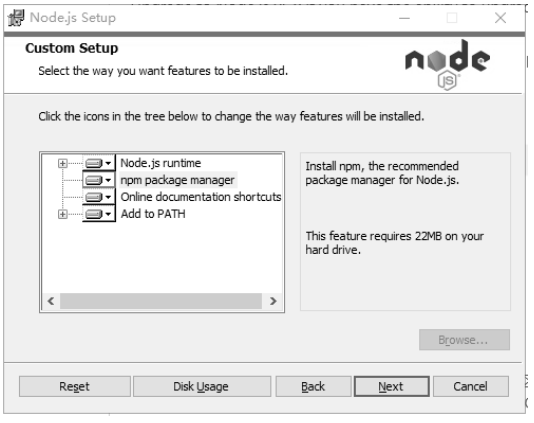

图 2-5　选择安装组件界面

界面中提供了多个选项，分别如下。

- Node.js runtime 是 Node.js 运行时所需要的核心组件。
- npm package manager 是我们准备用来安装 Electron 及其他三方模块的包管理组件。
- Add to Path 可以在 Node.js 安装成功后，自动将 Node.js 和 npm 运行路径添加到 Windows 系统的 Path 环境变量中，这样我们就可以在安装完成后直接打开命令行工具执行 Node.js 和 npm 命令了。

当然，这些组件可以选择不在当前安装，而在真正用到它们的时候才自动安装，如图 2-6 所示。

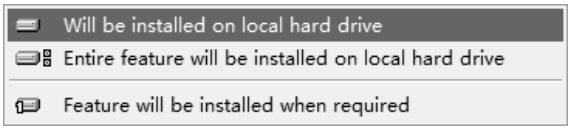

图 2-6　选择组件安装策略

为了保证日后开发时的连贯性，这里建议都在当前默认安装完。单击“Next”按钮进入安装原生模块工具集界面，如图 2-7 所示。

许多的 npm 模块需要在安装的时候进行编译，如果你的操作系统缺少这些对应的编译环境或工具，那么在执行 npm install 命令的过程中就会报错，导致无法成功安装。此处选中这个复选框，会自动安装需要的编译环境和工具。取消选中，则需要自己在用到时视情况安装。如果你对这些编译环境相关的知识不是很了解，那么此处我们建议选中。

接下来单击“Finish”按钮完成安装，如图 2-8 所示。

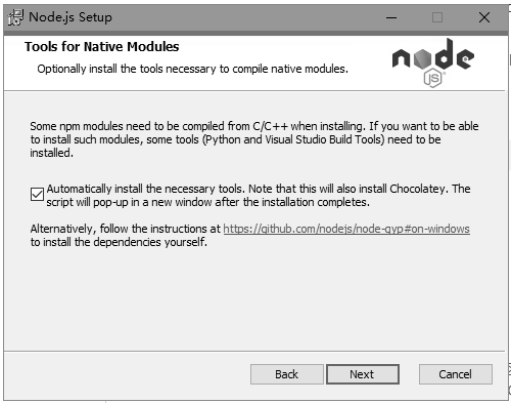

图 2-7　原生模块安装界面

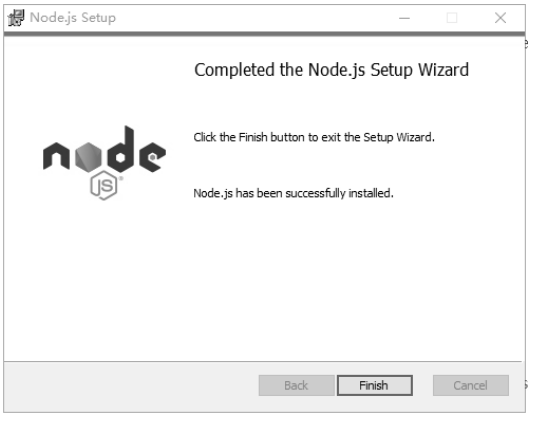

图 2-8　Node.js 安装成功界面

2.1.3　配置环境变量

在上一小节的讲解的安装过程中，如果默认选择了 Add to Path，那么在 Node.js 安装完成时，环境变量就已经配置好了。此时我们打开 Windows 的 CMD 命令行工具，测试一下 Node.js 和 npm 包管理器是否配置成功。

在命令行中输入 node -v 命令，如正确输出 Node.js 的版本号，说明 Node.js 已经安装并配置成功，如图 2-9 所示。

在命令行中输入 npm -v 命令，如正确输出 npm 的版本号，说明 npm 已经安装并配置成功，如图 2-10 所示。

图 2-9　CMD 命令行中执行 node -v 的结果

图 2-10　CMD 中执行 npm -v 的结果

如果版本号没有显示出来，而是提示找不到 node 或 npm 命令，则需要在系统的系统属性设置里面，检查一下 node 的环境变量是否添加。可以通过右键单击“计算机→系统属性→环境变量”找到 Path 变量，确认 Path 变量中是否有 Node.js 的安装路径，如图 2-11 所示。

如果 Path 变量中无 Node.js 的安装路径，只需要手动添加即可。在这之后，重新开启 CMD 命令行工具，就能正常使用 node 和 npm 命令了。

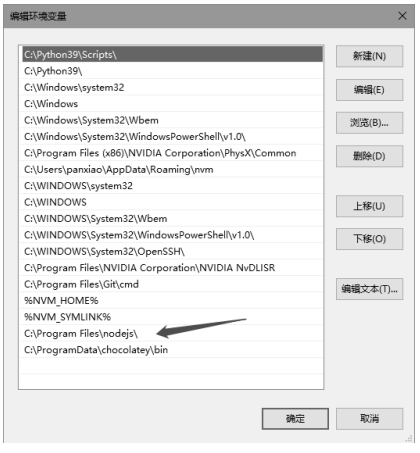

图 2-11　配置环境变量

2.2　Electron 环境搭建

在 Node.js 的环境准备就绪后，我们接着开始准备 Electron 的环境。正如上一小节内容中所提到的，我们要通过 npm 来安装 Electron。为了后续更方便地使用 Electron 的相关命令，我们现在准备使用下面的命令把 Electron 安装在全局。

```
npm install electron -g
```

执行该命令后，npm 包管理器会从 Electron 的官方源地址下载 Electron 并安装。但是由于网络的原因，国内使用官方源下载 Electron 经常会遇到下载超时或无故中断的情况，所以这里我们还需要配置一个淘宝的镜像源来解决这个问题。要更改 Electron 的下载源，需要安装前在 CMD 中执行下面这条命令。

```
npm config set ELECTRON_MIRROR https://npm.taobao.org/mirrors/electron/
```

配置完 Electron 的下载源之后，重新执行安装命令进行安装。安装完成后，我们同样通过执行 electron -v 命令来判断 Electron 是否安装成功，如图 2-12 所示。

图 2-12　CMD 中安装 Electron 的界面

默认情况下，Electron 会安装与你计算机处理器架构匹配的版本（例如在 X64 架构的计算机上会下载 X64 版本的 Electron）。如果你需要在 X64 架构的计算机上安装其他处理器架构的 Electron 版本，则需要在安装命令中单独加上--arch 参数。

```
npm install electron --arch=ia32 -g
```

在 Windows 操作系统中，可以通过“控制面板→系统和安全→系统”路径打开系统信息窗口，在这个窗口中看到计算机处理器的架构信息，如图 2-13 所示。

图 2-13　计算机信息查看窗口

2.3　实现一个系统信息展示应用

在前两节中，我们已经准备好了 Node.js 和 Electron 环境，那么接下来可以开始动手创建一个简单而又相对完整的 Electron 项目了。在这一小节中，我们准备通过从 0 开始实现一个用于展示系统信息的桌面应用，来让大家对 Electron 应用开发有一个相对全面的了解。在开始之前，我们还需要完成两项工作：

- ❑ 在磁盘中创建一个名称为 SystemInfoApp 的文件夹。
- ❑ 打开一个代码编辑器，并在编辑器中打开 SystemInfoApp 文件夹（推荐使用 Visual Studio Code，后续简称 VSCode）。

2.3.1　初始化项目

在 VSCode 的终端中，输入 npm init 命令来创建一个基于 npm 包管理的工作空间。执行该命令后，终端会提示需要输入多个步骤的信息。由于我们现在属于 DEMO 项目，这些并不是重点，所以这里可以通过一直按 Enter 键跳过。跳过所有步骤之后，在项目中生成了一个名为 package.json 的文件。package.json 是存在于项目根目录的 JSON 文件，它用于记录和描述当前项目的基本信息，如项目名、版本号等。同时它也负责管理当前

项目所依赖的第三方包。这里生成的 package.json 文件内容如下。

```
{
    "name": "systeminfoapp",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "",
    "license": "ISC"
}
```

name 属性的值是在初始化项目时根据当前所在文件夹的名称而定的，它用来表示当前项目的名称。你可以将它改为其他任意的名字，但是要注意以下规则。

- 必须是一个单词且全部字母必须是小写。
- 不能用下画线（_）或点（.）开头
- 可以包含连字符以及下画线（_）

version 属性表示当前项目的版本，往往在发布该项目的时候用到。

scripts 属性是一个 json 对象，用于自定义项目的脚本命令。我们在命令行中进入项目根目录，通过 npm run ×××命令可以执行在 scripts 中自定义的脚本命令。当前项目中自定义了一个 test 命令用于打印信息，我们通过执行 npm run test 命令可以在命令行中看到相应的输出，如图 2-14 所示。

```
C:\Users\panxiao\Desktop\Demos\SystemInfoApp>npm run test

> systeminfoapp@1.0.0 test C:\Users\panxiao\Desktop\Demos\SystemInfoApp
> echo "Error: no test specified" && exit 1
```

图 2-14　npm run test 命令执行结果

一般情况下，为了让启动命令规范化，我们期望团队中每个项目使用统一命令来启动程序，例如 npm run start。因此，在 scripts 属性中我们需要增加如下代码来满足这个要求。

```
"scripts": {
"start": "electron .",
"test": "echo \"Error: no test specified\" && exit 1"
},
```

main 属性描述当前项目的程序入口，它的值为应用入口文件相对于项目根目录的路径。在使用命令自动初始化的项目中，main 的默认值为根目录下 index.js 文件的相对路径。使用 electron 命令启动当前项目时，会从 main 属性中读取入口文件并启动。我们通过如下命令尝试启动应用：

```
electron .
```

或

```
npm run start
```

当我们执行上面的命令后，看到屏幕中弹出了错误提示框，如图 2-15 所示。不用担心，由于目前项目根目录下还没有 index.js，所以这是正常现象。在后面的内容中，我们会重点补全 index.js 的代码并讲解它。

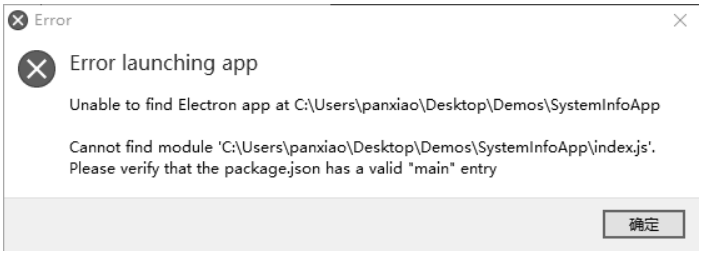

图 2-15　启动应用时的错误提示

2.3.2　程序目录结构

到目前为止，项目的文件夹中还只有一个 package.json 文件。这显然是不够的，为了完成这个项目的需求，我们还需要在根目录为项目创建如下几个必要的文件。

- index.js 文件：整个程序的入口，也是 Electron 应用主进程（主进程的概念会在下一个章节中详细讲解）代码所在。该文件内容主要负责控制 Electron 程序的生命周期、窗口的创建等逻辑。
- index.html 文件：窗口中页面的 html 元素代码。
- window.js 文件：窗口中页面的 Java Script 脚本代码。
- index.css 文件：窗口界面的 CSS 样式代码。

补齐必要文件后，项目目录结构如图 2-16 所示。

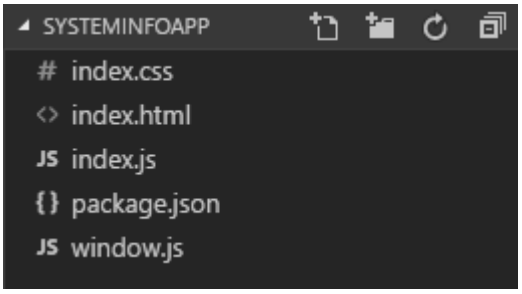

图 2-16　项目目录结构

2.3.3　应用主进程

在第 1 章的内容中，我们在介绍 Electron 时提到了主进程的概念。主进程在 Electron 中是非常重要的，应用程序需要通过运行在主进程中的代码来实现程序启动、窗口管理以及系统调用。对于当前这个项目，index.js 就是运行在主进程中的代码。它负责控制 Electron 程序的生命周期以及窗口的创建。在这个文件内容的开头，我们先把需要使用到的模块引入，代码如下所示。

```
// Chapter2/index.js
const electron = require('electron');
const app = electron.app;
const url = require('url');
const path = require('path');
```

electron 模块包含了 Electron 框架提供给开发人员在开发过程中所需要用到的 API。接下来我们需要通过 electron 模块来获取到其中的 app 模块。

app 模块负责控制应用程序的生命周期，提供了各个生命周期的回调来让开发者在这些关键的时间点上实现业务逻辑。系统信息展示应用将会用到其中的“ready”“window-all-closed”等生命周期事件，在接下来的内容中会进行展示。

url 和 path 将用来生成窗口需要加载的 html 文件的本地路径。

接下来我们在 index.js 文件中加入创建窗口相关的代码，代码如下所示。

```
// Chapter2/index.js
let window = null;

function createWindow() {

   window = new electron.BrowserWindow({
```

```
    width: 600,
    height: 400
  })

  const url = url.format({
    protocol: 'file',
    pathname: path.join(__dirname, 'index.html')
  })

  window.loadURL(url)

  window.on('close', function(){
    window = null;
  })
}
```

这段代码首先声明了一个默认值为 null 的变量 window。window 会在下面创建窗口的代码中被赋值为窗口的引用，用于后续对窗口进行操作。接着声明了一个名为 createWindow 的函数，在该函数中，使用 electron.BrowserWindow 创建一个宽 600px，高 400px 的窗口，将 new 操作返回的对象引用赋值给 window 变量。窗口创建完毕后，通过 window.loadURL 方法加载一个本地的 html 文件，作为窗口内展示的内容。

loadURL 方法需要传入一个本地 html 文件的路径。在当前项目中，该路径为根目录下的 index.html 文件路径。这里我们并没有直接传入手动拼接的 index.html 文件路径，而是用 url 模块的 format 方法来生成文件路径。url 模块提供的 format 方法，能通过配置化的方式来生成一个完整的 url 路径，比手动拼接更加方便，也更加标准化。我们在调用 format 方法时主要传入如下 3 个参数来生成 url 路径。

- protocol：字符串类型。protocol 属性定义了生成 url 的协议。比较常见的协议有“http”“file”“ftp”等。由于我们此处要加载的是本地文件，所以需要使用“file”协议。
- pathname：字符串类型。pathname 属性定义了 url 中的路径部分。例如在 https://www.electronjs.org/docs/api 这个 url 中，pathname 为/docs/api。由于在这个项目中使用的 url 协议为 file，所以 pathname 需要设置为 index.html 文件的绝对路径。为了能让生成的路径字符串有更好的兼容性，这里通过 path.join(__dirname, 'index.html')生成绝对路径。在 Node.js 中，__dirname 总是指向被执行 Java Script 文件的绝对路径，当前被执行的 js 文件为根目录中的 index.js，所以__dirname 在这里指向的是项目根路径。format 方法在执行完后生成的 url 为 file://C:\Users\panxiao\Desktop\Demos\SystemInfoApp\index.html。

在代码的最后，我们通过 window 对象注册了一个在窗口关闭时触发 close 事件的回调。当 close 事件触发时，意味着窗口被销毁，我们将保存该窗口引用的 window 变量重新赋值为 null 值。

还记得前面提到的能控制Electron应用程序生命周期app模块吗？接下来我们将要使用到它。app 对象提供了一系列生命周期相关的事件，例如“ready”“active”“will-quit”以及“quit”等。通过注册这些事件的回调函数，我们可以在这些事件触发时，执行对应的业务逻辑。在这个示例中，我们使用到了 app 模块提供的两个重要事件，window-all-closed 和 ready，代码如下所示。

```
// Chapter2/index.js
…
app.on('window-all-closed', function () {
  app.quit();
})

app.on('ready', function () {
  if (window === null) {
    createWindow();
  }
})
…
```

我们来看看这两个事件的定义。

- window-all-closed：该事件在所有已创建的窗口全部关闭时触发，此时意味着用户已经关闭了应用的所有窗口。在大部分的场景中，当所有窗口都已被关闭时，应用的默认行为是退出。但不排除一些例外情况，例如，一些支持托盘运行的应用，即使窗口都关闭了，还是希望应用能继续保持在后台运行，并在用户需要使用的时候单击托盘重新打开窗口。值得注意的是，如果在主进程代码中显示调用 app.quit()方法之后，Electron 会自动关闭所有窗口，但并不会触发 window-all-closed 事件。
- ready：该事件的触发表示 Electron 已经初始化完成。在 Electron 中，很多的 API 是需要在 ready 事件触发之后才能被正常调用的，例如，我们即将使用到的 BrowserWindow 对象，该对象用于创建一个窗口。如果在 ready 事件的回调之外调用 new BrowserWindow()，那么应用程序将会无法启动，并弹出如图 2-17 所示的错误提示。因此，在使用 Electron 提供的 API 时，需要注意它可以被正常调用的生命周期范围是什么。

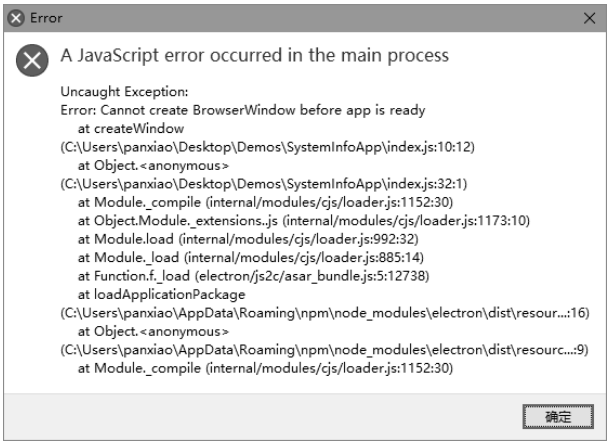

图 2-17　在 ready 事件回调之外创建窗口的错误提示

我们不期望在这个示例中有两个相同的窗口同时存在，因此当 Electron 初始化完成并触发 ready 事件之后，我们需要在该事件的回调函数中先对 window 变量进行判断，确认窗口是否已经被创建。如果 window 变量为 null，说明该窗口尚未被创建，接着就可以开始通过上面提到过的 createWindow 方法创建窗口并显示了。

在这个示例中，应用不需要在没有窗口的情况下保持后台运行。如果窗口都被用户手动关闭了，那么我们认为用户的意图是想要完全退出这个应用。因此，当 window-all-closed 事件触发时，我们调用 app.quit 方法让整个应用退出。

现在主进程（index.js）所有的代码已经编写完成，代码如下所示。

```
// Chapter2/index.js
const electron = require('electron');
const app = electron.app;
const url = require('url');
const path = require('path');

let window = null;

function createWindow() {

  window = new electron.BrowserWindow({
    width: 600,
    height: 400,
    webPreferences: {
      nodeIntegration: true
    }
```

```
  })

  const urls = url.format({
    protocol: 'file',
    pathname: path.join(__dirname, 'index.html')
  })


  window.loadURL(urls);

  window.on('close', function(){
    window = null;
  })
}

app.on('window-all-closed', function () {
  app.quit();
})

app.on('ready', function () {
  if (window === null) {
    createWindow();
  }
})
```

2.3.4　窗口页面

在上一小节中我们主要实现了主进程的相关逻辑，接下来我们开始实现渲染进程的相关逻辑。在主进程中我们通过 new electron.BrowserWindow()代码来创建了一个窗口，该窗口内页面所在的进程就是渲染进程，每个渲染进程本质上是一个基于 Chromium 的浏览器。你会在接下来的学习过程中发现，在 Electron 窗口中实现界面跟在普通浏览器中没有什么太大的区别。本示例的窗口中将使用 index.html、window.js 以及 index.css3 个文件，它们的代码都运行在这个“浏览器”环境之内。

在开始开发之前，我们先来看看这个项目的窗口页面子，如图 2-18 所示。

如图 2-18 所示，我们需要在页面中展示 5 个系统硬件相关的信息，分别为 CPU 型号、CPU 架构、平台类型、当前剩余内存及总内存。在系统信息没有获取并显示之前，统一用 Loading 字符串做占位符。

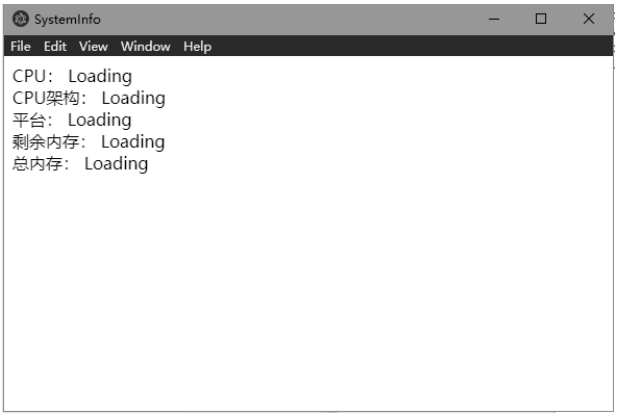

图 2-18　系统信息展示应用的界面

出于安全性的考虑，这些系统相关的信息原本在浏览器的 Web 页面中是无法获取的，但是在 Electron 的窗口页面中却可以。在 Electron 中开发页面与在浏览器中开发页面的主要区别在于，Electron 页面中的 Java Script 脚本可以使用 Node.js 的模块，例如，你可以在 Java Script 脚本中通过 require 引入 OS 模块来获取系统相关的数据（这个模块也会在本项目中会使用到），或者可以直接访问 process、__dirname 以及 global 等 Node.js 中才有的全局模块或变量。正如第 1 章所述，Electron 底层通过进程间通信实现了这一切。因此，在 Electron 中开发页面会比浏览器中开发页面具有更高的自由度，这同时也使得开发者能在 Electron 的窗口页面中，开发出原本在浏览器中很难实现甚至是无法实现的功能。

在主进程的代码中，我们在使用 BrowserWindow 对象创建窗口的时候，给构造函数传入了一个特殊的参数 nodeIntegration，代码如下所示。

```
window = new electron.BrowserWindow({
  width: 600,
  height: 400,
  webPreferences: {
    nodeIntegration: true
  }
});
```

当 nodeIntegration 设置为 true 时，渲染进程中运行的 Java Script 脚本将允许引用 Node.js 中的模块来实现功能（这里需要注意的是，在 Electron 5.0 版本之前，nodeIntegration 的默认值为 true。而在 Electron 5.0 版本之后，默认值变更为 false。如果开发者在代码中没有指定它为 true 时，渲染进程将无法使用 Node.js 模块）。

窗口准备就绪，接下来我们开始编写页面的 html 代码。首先，我们在 index.html 中

添加一个基础的模板框架，代码如下所示。

```
// Chapter2/index.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>SystemInfo</title>
    <link rel="stylesheet" href="./index.css">
</head>
<body>
…
    <script type="text/Java Script" src="./window.js"></script>
</body>
</html>
```

在这段基础的 html 代码中，我们在 head 标签中通过 link 标签来引入根目录下的样式文件 index.css，同时在 body 标签的结尾处通过 script 标签引入根目录下的脚本文件 window.js。引入脚本和样式的方式与 Web 浏览器中是一样的，只不过这里所引用的资源都存在本地，而不是在服务器中。紧接着，我们需要根据页面的布局和信息内容，在 body 中编写对应的 html 元素，代码如下所示。

```
// Chapter2/index.html
    <div id='cpu'>
      CPU：
      <span>Loading</span>
    </div>
    <div id='cpu-arch'>
      CPU 架构：
      <span>Loading</span>
    </div>
    <div id='platform'>
      平台：
      <span>Loading</span>
    </div>
    <div id='freemem'>
      剩余内存：
      <span>Loading</span>
    </div>
    <div id='totalmem'>
      总内存：
```

```
        <span>Loading</span>
    </div>
```

在上面的代码中，我们给每一个 div 元素都根据内容特征设定了一个唯一的 id 标识。例如，存放 CPU 信息的外层 div 元素，我们给它定义了一个名为“cpu”的 id。这么做的目的是便于在页面的脚本中通过 id 定位到对应的元素并填充内容。span 元素是最终显示系统信息的地方，在未被填充新内容前先使用 Loading 占位，等到脚本获取到对应的内容后就会将内容插入 span 元素中替换 Loading 占位符。将以上两部分代码整合后，通过 npm run start 命令运行程序，可以看到图 2-19 的效果。

页面基本的骨架已经搭建完毕，接下来我们需要在 window.js 文件中编写代码来获取系统信息填入对应的元素中。除了能在 window.js 中使用 Node.js 模块以外，编写 window.js 中的代码与编写普通 Web 前端页面中的代码没有本质上的区别。在 window.js 文件开头，我们通过如下代码引入 Node.js 的 OS 模块，它提供了一系列操作系统相关的方法和属性，通过这些方法和属性可以拿到我们所需要系统信息。

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

接着我们定义一系列方法来获取对应的系统信息，代码如下所示。

```
// Chapter2/window.js
...
function getCpu() {
  const cpus = os.cpus();
  if (cpus.length > 0) {
    return cpus[0].model;
  } else {
    return '';
  }
}

function getFreemem() {
  return `${convert(os.freemem())}G`;
}

function getTotalmem(){
  return `${convert(os.totalmem())}G`;
}

function convert(bytes) {
  return (bytes/1024/1024/1024).toFixed(2);
}
```

```
...
```

getCpu 方法调用了 OS 模块的 cpus 方法来获取当前计算机的 CPU 信息。cpus 方法返回的是一个数组，该数组的长度等于当前计算机所使用的 CPU 的核心数。以笔者的计算机为例，使用的是四核 I5 的处理器，在调用 cpus 方法后，返回了一个长度为 4 的数组，如图 2-19 所示。

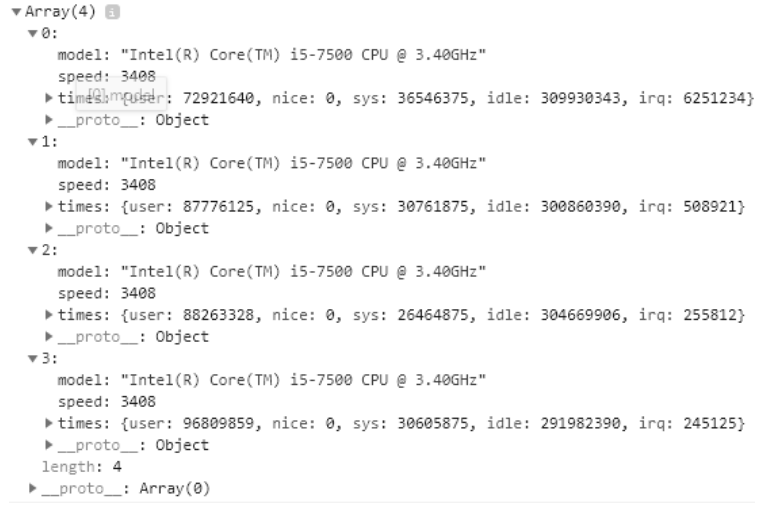

图 2-19　CPU 信息

数组中每个元素都是一个对象，每个对象代表其中一个 CPU 核心的信息。除了 CPU 型号和主频率外，还能在 times 属性中看到核心的使用情况。由于在本项目中只需要展示 CPU 的型号信息，所以这里只需要取数组 0 号元素的 model 属性值来进行展示即可。为了让程序更加健壮，代码中在获取数组 0 号元素之前要先对数组的长度进行判断。当长度大于 0 时，返回 0 号元素的 model 属性值。当长度小于 0 时，直接返回字符串。

getFreemem 和 getTotalmem 方法，分别通过调用 os.freemem 与 os.totalmem 方法返回当前内存的剩余空间和总空间。需要注意的是，这两个方法返回的的内存数值的单位是 bytes，为了更直观地展示内存信息，需要通过 convert 方法将单位 bytes 转换成 GB，并通过 toFixed 方法保留两位小数。

在文件的最后，我们通过经典的 DOM 操作将系统信息显示在页面中，代码如下所示。由于架构信息与平台信息可直接通过 os.platform 与 os.arch 直接获取到，并且不需要进行特殊的处理，所以这里并没有将其单独封装成方法。

```
// Chapter2/window.js
```

```
...
document.querySelector('#cpu-arch span').innerHTML = os.arch();
document.querySelector('#cpu span').innerHTML = getCpu();
document.querySelector('#platform span').innerHTML = os.platform();
document.querySelector('#freemem span').innerHTML = getFreemem();
document.querySelector('#totalmem span').innerHTML = getTotalmem();
```

现在通过 npm run start 命令启动应用，就可以看到我们想要的效果了，如图 2-20 所示。

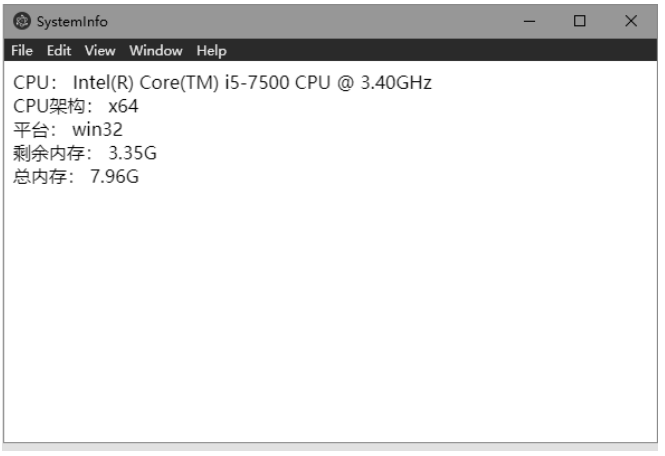

图 2-20　展示系统信息的界面

到目前为止，index.css 文件的内容还空着。为了让页面更美观一些，我们来添加一些简单的样式，使得系统信息内容可以对齐并且有独立的颜色。在编写样式代码之前，我们需要稍微改造一下 index.html 文件，给 body 中的元素增加如下所示的类名和标签。

```
<div id='cpu' class='info'>
        <label>CPU：</label>
        <span>Loading</span>
</div>
```

接着使用新增加的类名来编写对应的 CSS 样式，代码如下所示。

```
// Chapter2/index.css
.info{
   padding: 5px;
}

.info label{
   display: inline-block;
```

```
    width: 100px;
}

.info span{
    color: blue;
}
```

现在通过 npm run start 命令启动应用，就可以看到带样式的页面了，如图 2-21 所示。

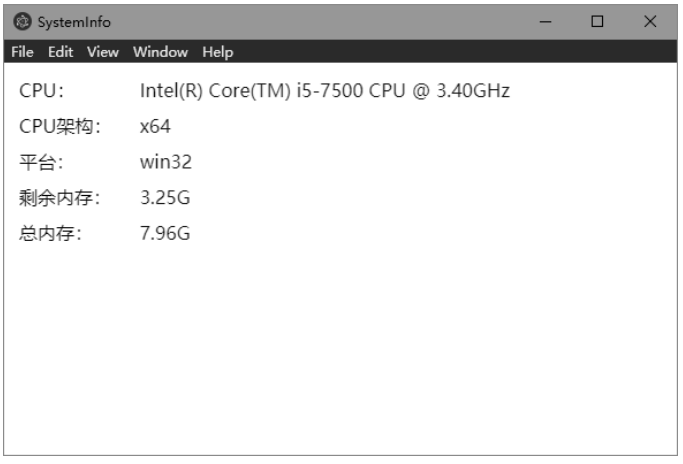

图 2-21　添加样式后的界面

2.4　总　　结

- 在安装 Electron 之前需要先安装 Node.js 环境，并在安装完成后通过 node -v 命令确认是否安装成功。
- Node.js 的安装包会在安装完成后，自动设置 Windows 的环境变量。如果使用 node 或 npm 命令时提示无法找到该命令，需要去环境变量设置中检查是否设置成功。
- 通过设置国内的镜像源，可以加速 Electron 的下载。
- Electron 会默认下载与当前计算机处理器架构匹配的版本。可以在安装命令中指定 arch 来下载适配其他处理器架构的 Electron 版本。
- 在系统信息展示项目中，主进程中的主要负责管理程序和窗口的生命周期。渲染进程负责显示窗口页面内容，执行页面脚本。

- Electron 5.0 版本之后，通过将创建窗口时传入的 webPreferences 对象内的 nodeIntegration 属性设置为 true，可以让在渲染进程中执行的脚本有权限使用 Node.js 模块。
- Node.js 提供的 OS 模块能让开发人员获取到系统相关的信息。

本章节的内容通过讲解一个最简单的 Electron 应用的开发过程，来帮助你入门 Electron。学习完这些内容，你已经可以自己搭建并开发一个简单的 Electron 应用了。为了内容上更关注 Electron 的相关知识，在系统信息展示应用功能实现中，我们没有使用在 Web 前端开发中常用的框架，而是使用最基础的 Html、Java Script 和 CSS 完成的。Electron 之外的内容，我们会尽量保持简单，不期望在这些方面耗费掉你额外的学习精力。这个约定不仅适用于在本章节，后续的章节也是如此。如果你已经完全掌握本章节内容，我们建议你可以在系统信息展示应用源代码的基础上，按照 Electron 官方文档的说明，尝试使用一下 Electron 其模块的 API，为学习后面章节的内容打下基础。

本章节中所涉及的完整代码，可以访问 https://github.com/ForeverPx/ElectronInAction/tree/main/Chapter2。在学习本章的过程中，建议你下载源代码，亲手构建并运行，以达到最佳学习效果。