

目录

说明	1.1
介绍	1.2
环境准备及工具使用	1.3
2.1 环境准备及使用	1.3.1
2.2 IDE工具使用	1.3.2
2.3 初始JS文件加载工具	1.3.3
语法规格	1.4
模块接口	1.5
4.1 Timer	1.5.1
4.2 GPIO	1.5.2
4.3 UART	1.5.3
4.4 PWM	1.5.4
4.5 SPI	1.5.5
4.6 I2C	1.5.6
4.7 HiLink	1.5.7
4.8 Buffer	1.5.8
4.9 RTC	1.5.9
4.10 System	1.5.10
4.11 OTA	1.5.11
4.12 其它	1.5.12
调试	1.6
常见问题	1.7
限制与约束	1.8

MapleJS是华为面向物联网（IoT）设备侧应用开发而提供的轻量化JavaScript引擎及其开发环境。

注：若遇到此文档显示不完整的问题，请升级您的阅读器版本或更换阅读器。

- [1 简介](#)
 - [1.1 背景说明](#)
 - [1.2 特点](#)
 - [1.3 各模块介绍](#)

1 简介

本章为MapleJS基本介绍。

1.1 背景说明

MapleJS是华为推出的面向物联网（IoT）设备侧应用开发的轻量化JavaScript引擎，及其配套的开发工具集。MapleJS可以运行在LiteOS物联网实时操作系统之上并支持HiLink物联网协议，使得开发者能够在资源受限的嵌入式设备上使用JavaScript进行开发；并通过提供统一的设备能力抽象接口，向开发者屏蔽硬件差异，使其更加聚焦业务实现，从而提升IoT设备应用开发效率。

1.2 特点

- 轻量化：Flash占用小于100KB，空载时RAM占用小于32KB；
- 支持语言标准：支持ECMAScript 5.1标准；
- 垂直整合：与LiteOS整合优化，达到最优能耗/性能比。

1.3 各模块介绍

为了方便广大开发者的开发活动并进一步形成良好的生态，MapleJS提供了一整套完善的开发环境。主要划分为以下四个部分。

- **MapleJS引擎：**对JS代码进行高效的解释执行；
- **开发工具套：**提供了一套完整的从编码、编译，到部署、调试的端到端的集成开发环境，并在开发周期中持续性提供辅助优化的能力；
- **面向设备型开发框架：**支持事件驱动的编程模型，并提供统一的硬件抽象接口、系统抽象接口、网络协议接口等，让开发者能够方便快速调用，编程自由度得以进一步释放；
- **行业共享仓库：**提供面向行业应用的共享库，便于第三方开发者快速开发行业应用。

- [2 环境准备及IDE工具使用](#)

2 环境准备及IDE工具使用

本章主要介绍MapleJS运行所需的环境准备和相关工具使用。

2.1 环境准备及使用

2.1.1 版本获取及完整性校验

当前只支持通过华为公司版本发布流程获取。该流程会对所有文件进行OpenPGP签名，生成签名文件“文件名.扩展名.asc”可以通过手工方式进行验证，具体请参考：<http://support.huawei.com/carrier/digitalSignatureAction>

2.1.2 目录介绍

版本包主要包含JS引擎头文件、JS引擎静态库、编译调试工具三部分，解压缩之后各个文件夹内容如下所示：

- include: 定义JS引擎接口的头文件；
- lib: JS引擎的静态库，根据使用场景选择相应的版本，其中
 - Debug: 支持debug功能，支持parser；
 - Release_Parser: 不支持debug功能，支持parser；
 - full: 相比pl-cnv，增加对PWM/I2C/SPI/GPIO/RTC/FS模块的支持
 - pl-cnv: 支持TIMER/UART/BUFFER/OTA/HILINK/SYSTEM基本模块
 - Release: 不支持debug功能，不支持parser；
 - full: 相比pl-cnv，增加对PWM/I2C/SPI/GPIO/RTC/FS模块的支持
 - pl-cnv: 支持TIMER/UART/BUFFER/OTA/HILINK/SYSTEM基本模块
- tools:
 - antjs-snapshot.exe: 编译器的可执行文件，将JS源码编译成字节码；
 - maple-client-serial.py: debugger客户端工具；
 - vscode-antjs-x64.vsix: 64位Visual Studio Code插件；
 - vscode-antjs-x32.vsix: 32位Visual Studio Code插件。
 - init-js-load-tool.exe: 初始JS文件加载工具，可将JS文件加载至bin文件。

2.1.3 接口介绍

JS引擎当前基于事件驱动的运行模式，主要提供3个接口(其中引擎上层事件驱动框架的名字为AntJS，因此接口以“antjs_”打头)：

- void antjs_event_queue_init (int capacity); 用于初始化事件驱动框架的事件队列，在运行antjs_entry前，首先要运行该接口，其中
 - capacity: 表示事件队列的大小。
- void antjs_entry (void); JS引擎入口函数，用于启动引擎的事件框架，在使用引擎加载运行JS代码前，首先要运行该接口。
- int antjs_notify (antjs_event_t typ, int in_isr, unsigned int data_len, void * data); 用于向JS引擎发送需要处理的事件，具体做的事情由使用的事件决定，其中
 - typ: 表示处理的事件类型，具体参考2.1.4节；
 - in_isr: 表示处理的事件是否由中断引起；
 - data_len: 表示与事件相关的数据的长度；
 - data: 表示与事件相关的数据。

2.1.4 事件介绍

- EVENT_EVAL_FILE: JS引擎加载以文件形式存在的JS文本文件或字节码文件；
 - JS引擎会根据引擎的状态自动初始化引擎
- EVENT_EVAL_BUFFER: JS引擎加载以字符串格式的JS代码或字节码；
 - JS引擎会根据引擎的状态自动初始化引擎
- EVENT_CALLBACK_INVOKE: 主要用于引擎内部事件处理。

下面展示如何向JS引擎发送EVENT_EVAL_FILE和EVENT_EVAL_BUFFER事件：

```

/* Start loading and executing the JS file, and use JS file path as the data */
antjs_notify (EVENT_EVAL_FILE, 0, strlen ("/spiffs/index.js") + 1, "/spiffs/index.js");

/* Start loading and executing the js string buffer, and use string buffer as the data */
char antjsBuffer[50] = "print(\"Hello World from EVENT_EVAL_BUFFER!\")";
antjs_notify (EVENT_EVAL_BUFFER, 0, strlen (antjsBuffer) + 1, antjsBuffer);

```

EVENT_CALLBACK_INVOKE事件用于引擎内部事件处理, 这里就不展示如何使用了.

2.1.5 如何与LiteOS集成

JS引擎与Lite OS集成时首先需要在工程中添加以下文件: `libantjs-liteos-xxxx.a`, `libantjs-all.a`, `js-main.c`, `antjs.h`, 然后请参考以下代码样例实现在LiteOS上启动JS引擎。

```

void
maplejs_main_entry (void *param)
{
    TSK_INIT_PARAM_S initparam;
    UINT32 ret = LOS_OK;
    hilink_debug ("HUAWEI AntJS Engine Start\r\n");

    /* Initialize LiteOS's file system.*/
    los_vfs_init ();

    /* Initialize the event queue. Define the capacity of queue according to the actual situation.*/
    antjs_event_queue_init(10);

    /* Create LiteOS's task to start the JS engine.
     * Please refer to LiteOS's documents to set the task parameters.
     */
    initparam.pfnTaskEntry = (TSK_ENTRY_FUNC) antjs_entry;
    initparam.usTaskPrio = LOS_TASK_PRIORITY_LOWEST - 1;
    initparam.pcName = "antjs_entry";
    initparam.uwStackSize = ANTJS_ENGINE_TASK_STACK_SIZE;
    ret = LOS_TaskCreate (&antjs_task_id, &initparam);
    if (ret != LOS_OK)
    {
        hilink_debug("create task fail!\r\n");
    }

    /* The uart_deploy_task is used in the development and debugging phase
     * to deploy files from the PC to the device through a serial port.
     * It does not need to be started in the commercial phase.
     */
    initparam.pfnTaskEntry = (TSK_ENTRY_FUNC) uart_multiplexing_task;
    initparam.usTaskPrio = LOS_TASK_PRIORITY_LOWEST-1;
    initparam.pcName = "uart_deploy_task";
    initparam.uwStackSize = ANTJS_TASK_STACK_SIZE;
    ret = LOS_TaskCreate (&uart_deploy_task_id, &initparam);
    if (ret != LOS_OK)
    {
        hilink_debug("create task fail!\r\n");
    }
    antjs_notify (EVENT_EVAL_FILE, 0, strlen (FILE_PATH) + 1, FILE_PATH);
} /* maplejs_main_entry */

void
js_main(void)
{
    TSK_INIT_PARAM_S initparam;
    UINT32 ret = LOS_OK;
    initparam.pfnTaskEntry = (TSK_ENTRY_FUNC) maplejs_main_entry;
    initparam.usTaskPrio = MAPLEJS_TASK_PRIORITY;
    initparam.pcName = "maplejs_main_entry";
    initparam.uwStackSize = MAPLEJS_TASK_STACK_SIZE;
    ret = LOS_TaskCreate (&maplejs_main_task_id, &initparam);
    if (ret != LOS_OK)
    {
        hilink_debug("create maplejs task fail!\r\n");
    }
} /* js_main */

```

在工程的main()中添加js_main()的调用。

```
void main(void)
{
    uint32_t uwRet;
    ...
    js_main();
    ...
}
```

请注意在引擎的主任务中禁止主动释放时间片，这样会影响引擎的正常启动。例如一下程序通过 `LOS_TaskDelay(2000)` 主动释放了2000个时间片，导致引擎的主任务被挂起，引擎无法正常启动。

```
maplejs_main_entry (void *param)
{
    TSK_INIT_PARAM_S initparam;
    UINT32 ret = LOS_OK;
    hilink_debug ("HUAWEI AntJS Engine Start\r\n");
    LOS_TaskDelay(2000);
    los_vfs_init ();
    antjs_event_queue_init(10);
    ...
    antjs_notify (EVENT_EVAL_FILE, 0, strlen (FILE_PATH) + 1, FILE_PATH);
} /* maplejs_main_entry */
```

2.1.6 如何生成/运行字节码文件

1. 使用 `tools/antjs-snapshot.exe` 编译js文件生成字节码文件(后缀为snapshot)，在windows CMD 上使用下面的命令：

```
tools/antjs-snapshot.exe generate file.js -o file.snapshot
```

2. 其他不变，即在 `create_js_task` 中，调用`antjs_notify`

```
/* Start loading and executing the JS file, and use JS file path as the data */
antjs_notify (EVENT_EVAL_FILE, 0, strlen (FILE_PATH) + 1, FILE_PATH);
```

这里假设需要运行的字节码文件已经保存在 `FILE_PATH` 执行的路径里。注意只有lib/Release下的版本支持执行snapshot文件。

2.1.7 依赖

- LiteOS，支持文件系统；
- 爱联模组；
- Flash空间>100KB，内存>32KB。

2.2 IDE工具使用

MapleJS for Visual Studio Code (vsc-antjs) 是以 Visual Studio code 为基础的MapleJS开发工具，目前已支持以下特性：

- 向已连接设备部署代码文件并运行
- 一键重启MapleJS设备
- 源文件压缩
- 中止已连接设备JS引擎的运行

2.2.1 需求

VS code版本不低于 July 2018 v1.26.1，并根据VS code的架构选择安装 x32 或 x64 版本的插件

建议同时安装 [ESLint](#) 和 [JavaScript \(ES6\) code snippets](#)，有助于正确编码

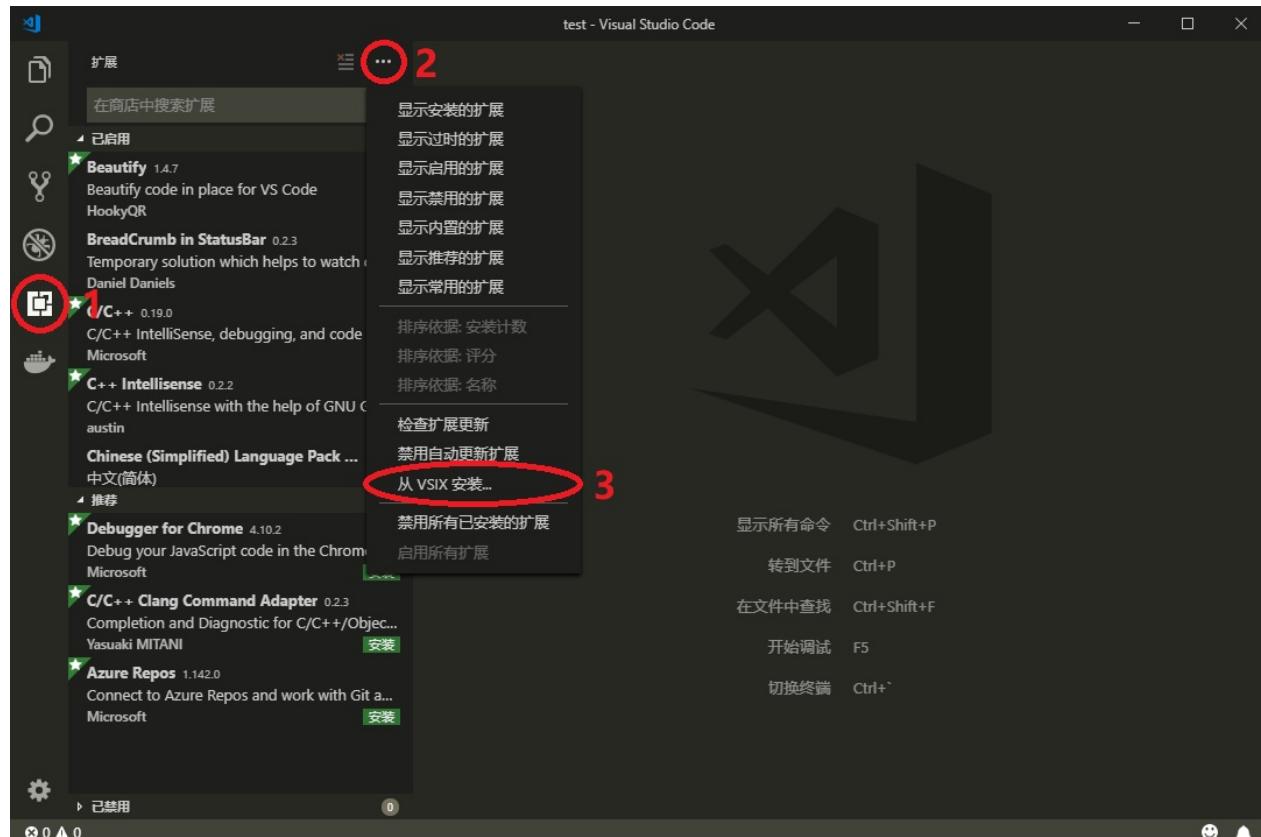
vsc-antjs 需要通过串口读写数据，所以在连接设备前，需要确保已安装USB转串口驱动：

- [Silabs drivers for Espressif esp8266/esp32](#)
- [CH43x drivers for Espressif esp8266/esp32](#)
- [FTDI drivers for CC3200, CC3220](#)

2.2.2 安装

方式一：通过VS code客户端

操作步骤如下图



之后在弹出的窗口内找到自己的插件安装包即可，安装完毕后VS code右下角会提示重新加载客户端，重启即可使用此插件

方式二：通过命令行

首先下载对应版本的插件安装包 `*.vsix`，然后使用命令行切换到插件所在目录，然后执行下面的代码

```
code --install-extension vscode-antjs-0.4.0-x32.vsix
```

安装之后，打开VS Code即可在扩展列表内看到 vsc-antjs

2.2.3 配置

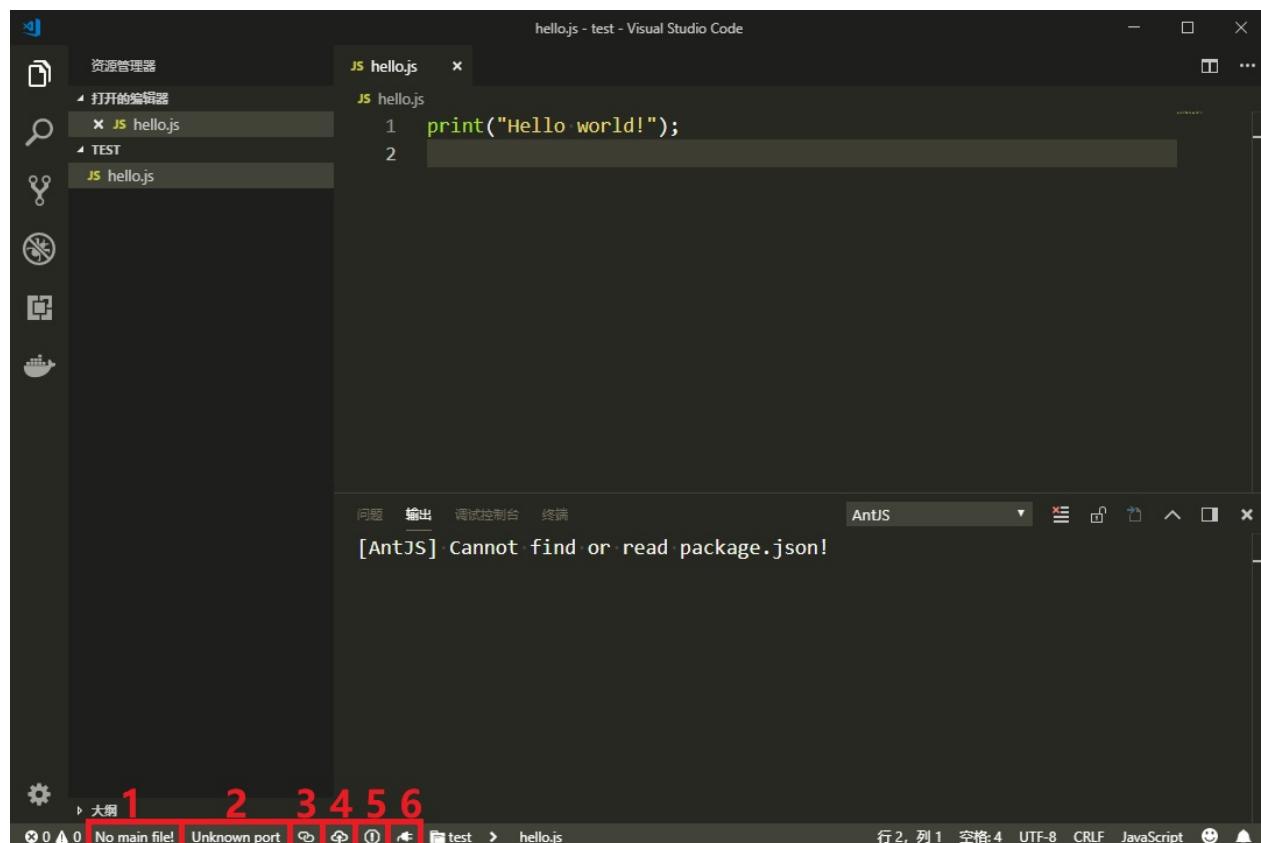
vsc-antjs支持通过配置文件进行配置，读取的配置文件为工作目录下的 `package.json`，使用VS code打开工作目录（必须使用打开文件夹才可以使用完整的插件功能），vsc-antjs会读取 `package.json` 中的五项配置：

```
{
  "main": "src/main.js",
  "port": "COM5",
  "isCompress": true,
  "isPacket": true,
  "packOpt": {
    "treeshake": true
  }
}
```

- main: 该参数指定用于部署到设备中的目标JS文件
- port: 该参数指定用于连接设备的串口号
- isCompress: 该参数指定部署前是否对目标文件进行压缩，默认为 `true`
- isPacket: 该参数指定部署文件时是否使用分包部署的方式， 默认为 `true`
- packOpt: 该参数用于配置 `rollup` 机制，目前支持以下配置项：
 - treeshake，用于配置是否启用 `treeshaking` 机制， 默认为 `true`； `treeshaking` 是一种死代码删除优化，如果不希望去除冗余代码，可以设置为 `false`，关闭该优化；

2.2.4 使用

插件支持通过按钮和命令面板两种方式使用，效果一致。主要的按钮有6个，如下图所示：



按钮1： main file，点击此按钮会弹出所有可供选择的JS文件，选中的文件可通过deploy按钮部署到设备上；

按钮2: COM port, 点击此按钮会弹出可以使用的COM口，插件会通过选中的COM口连接设备；

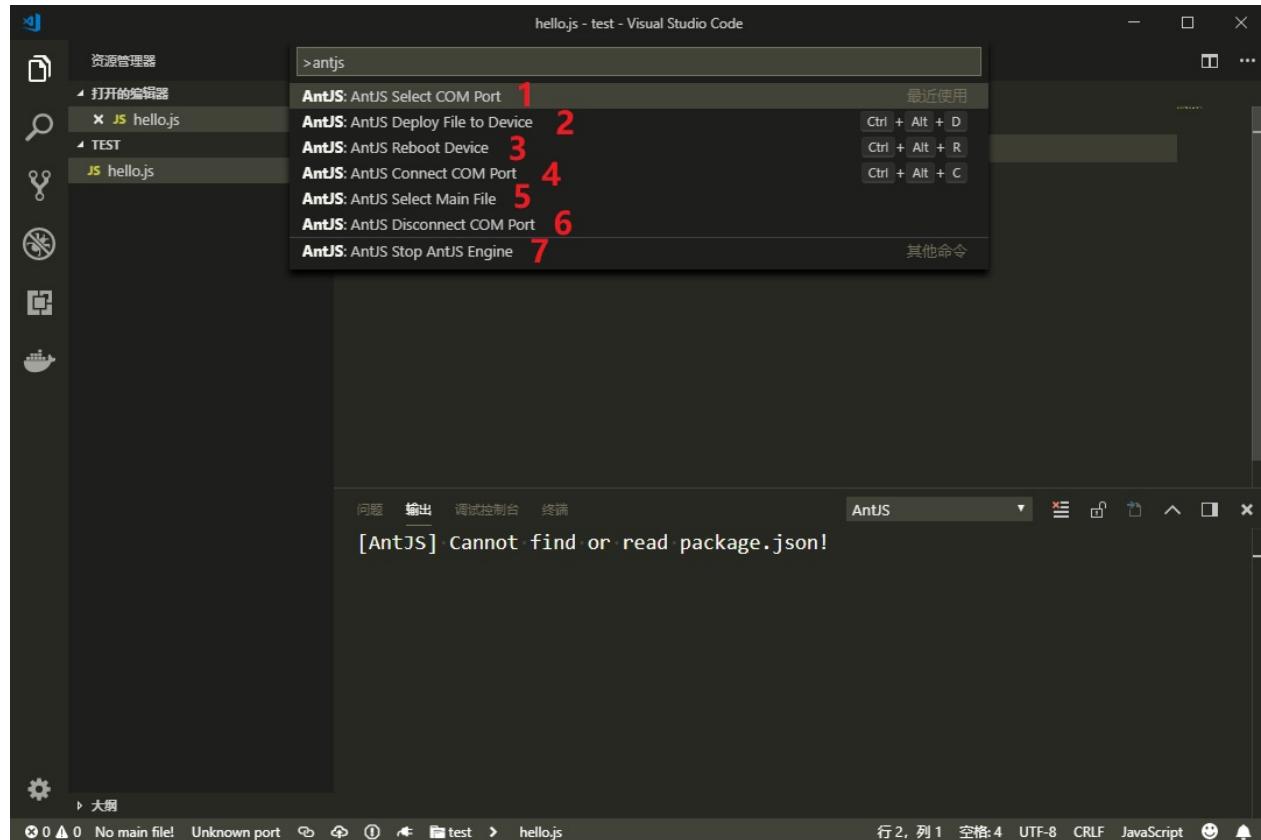
按钮3: connect & disconnect, 点击此按钮会通过按钮2选中的COM口与设备建立连接或断开连接；

按钮4: deploy, 点击此按钮会将按钮1选中的文件压缩，然后通过按钮2选中的COM口部署到设备里面去；

按钮5: stop, 点击此按钮可中止已连接设备JS引擎的运行；

按钮6: reboot, 点击此按钮可重启设备。

命令面板可以通过组合键 `ctrl + shift + p` 打开，支持的命令如下图所示：

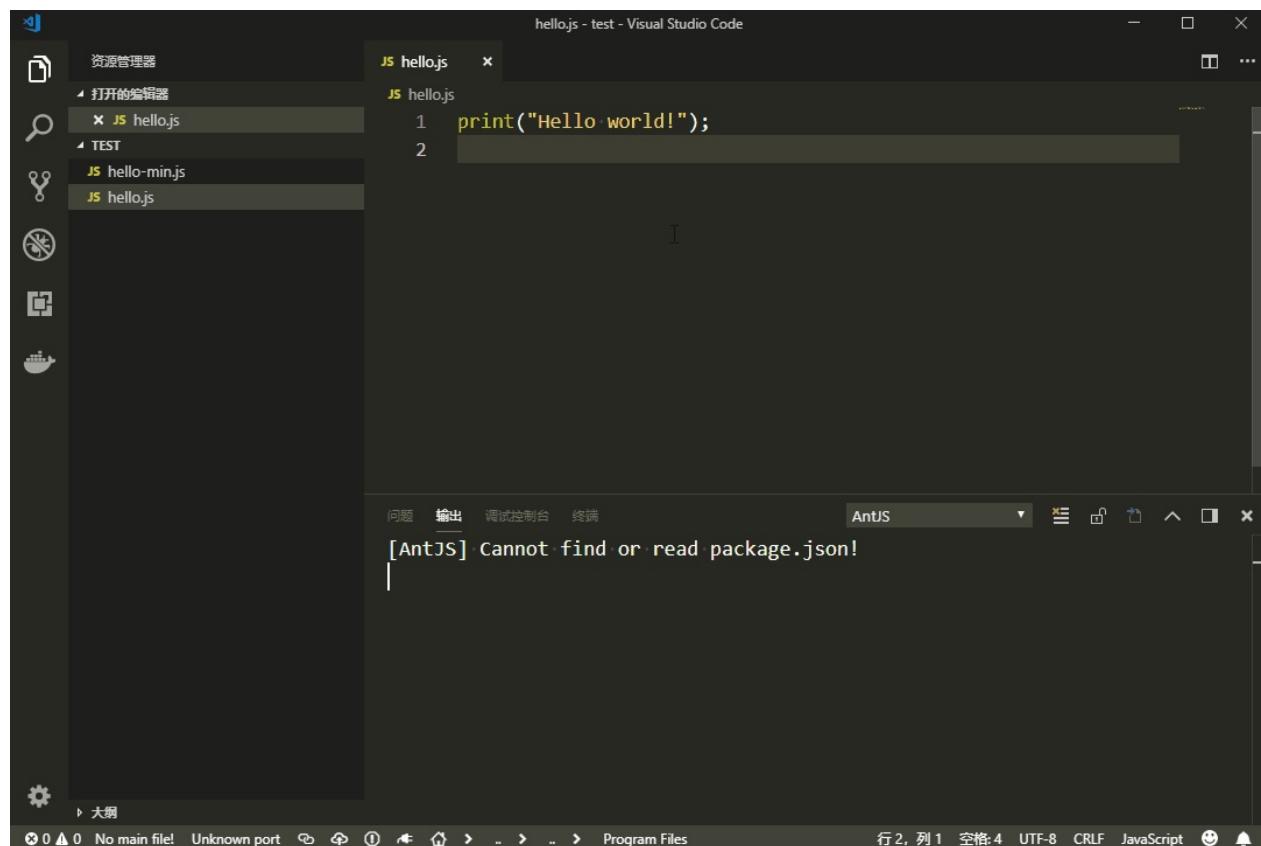


插件共注册了7个命令，如上图所示，所有的命令和点击按钮的效果都是一样的。按钮3在这里对应了两个命令，分别为 `connect` 和 `disconnect`。

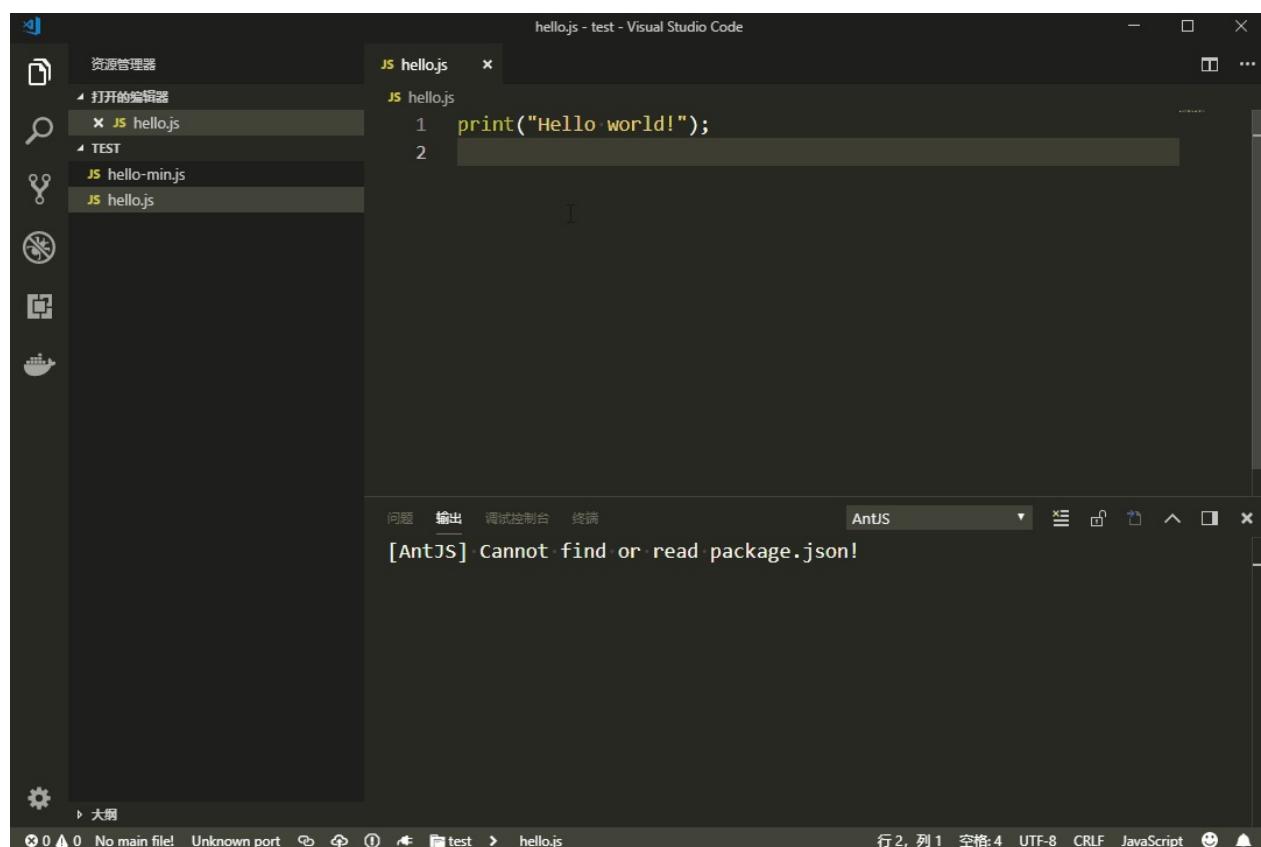
下面是对插件支持的每个操作的动图演示，分别通过按钮和命令行两种方式实现

2.2.4.1 选择文件

使用按钮：点击选择文件按钮（无选中下显示为 `No main file!`，已有选中下显示已选中的文件名），在弹出的列表中选择需要部署的文件，选择完成后，选择文件按钮会显示刚被选中的文件名

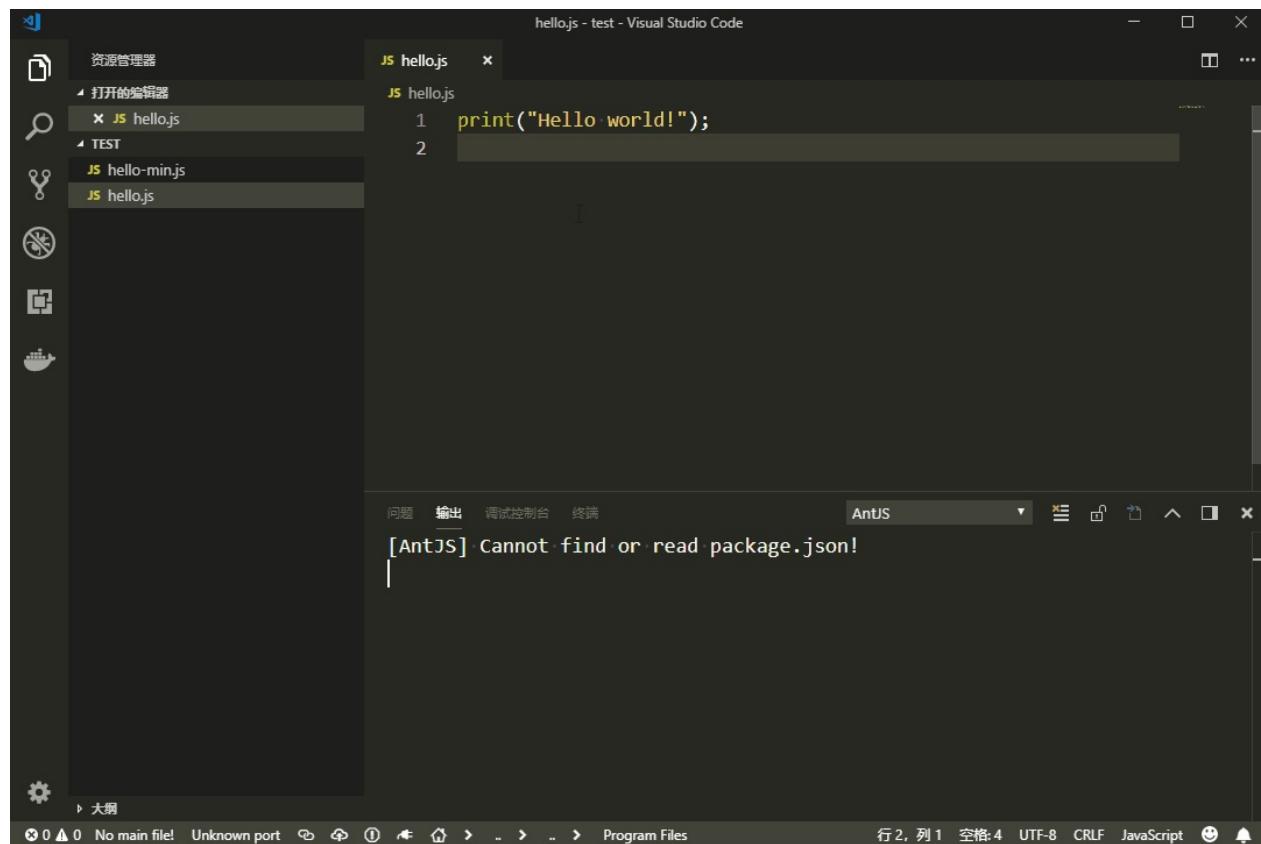


使用命令：使用 **Ctrl+Shift+p**，在弹出的命令列表中选择 **AntJS: AntJS Select Main File**，在弹出的列表中选择需要部署的文件，选择完成后，选择文件按钮会显示刚被选中的文件名

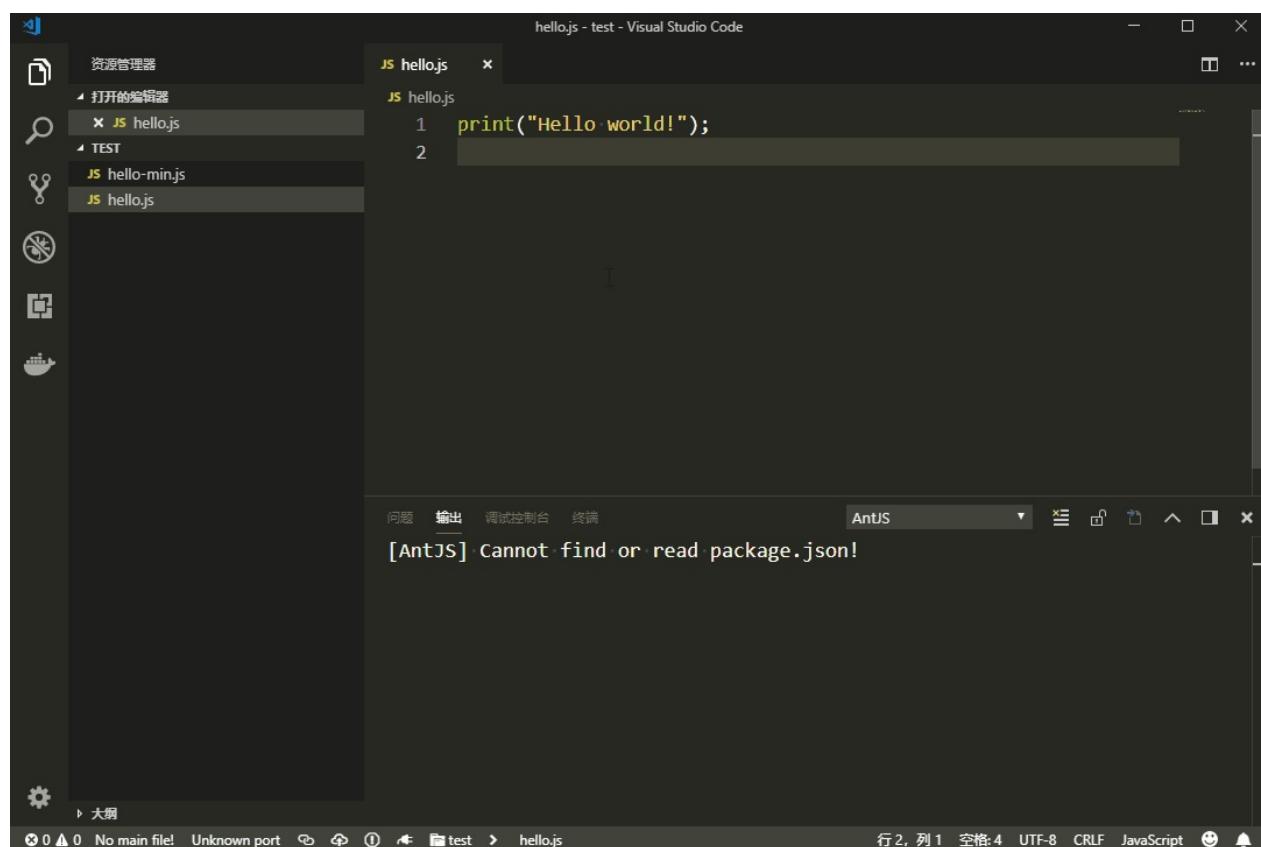


2.2.4.2 选择串口

使用按钮：点击选择串口按钮（无选中下显示为 `unknown port`，已有选中下显示已选中的串口号），在弹出的列表中选择设备连接的串口，选择完成后，选择文件按钮会显示刚被选中的串口号

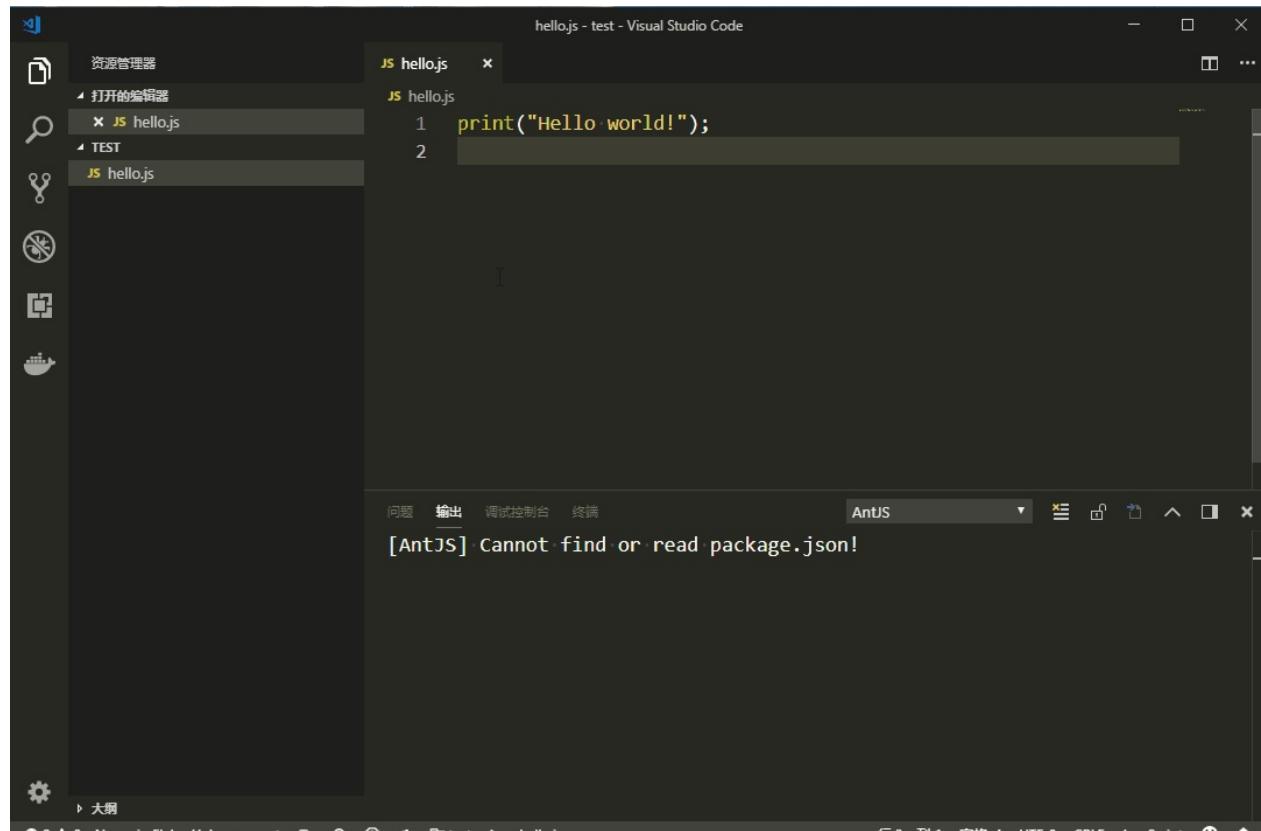


使用命令：使用 `Ctrl+Shift+p`，在弹出的命令列表中选择 `AntJS: AntJS Select COM Port`，在弹出的列表中选择设备连接的串口，选择完成后，选择文件按钮会显示刚被选中的串口号

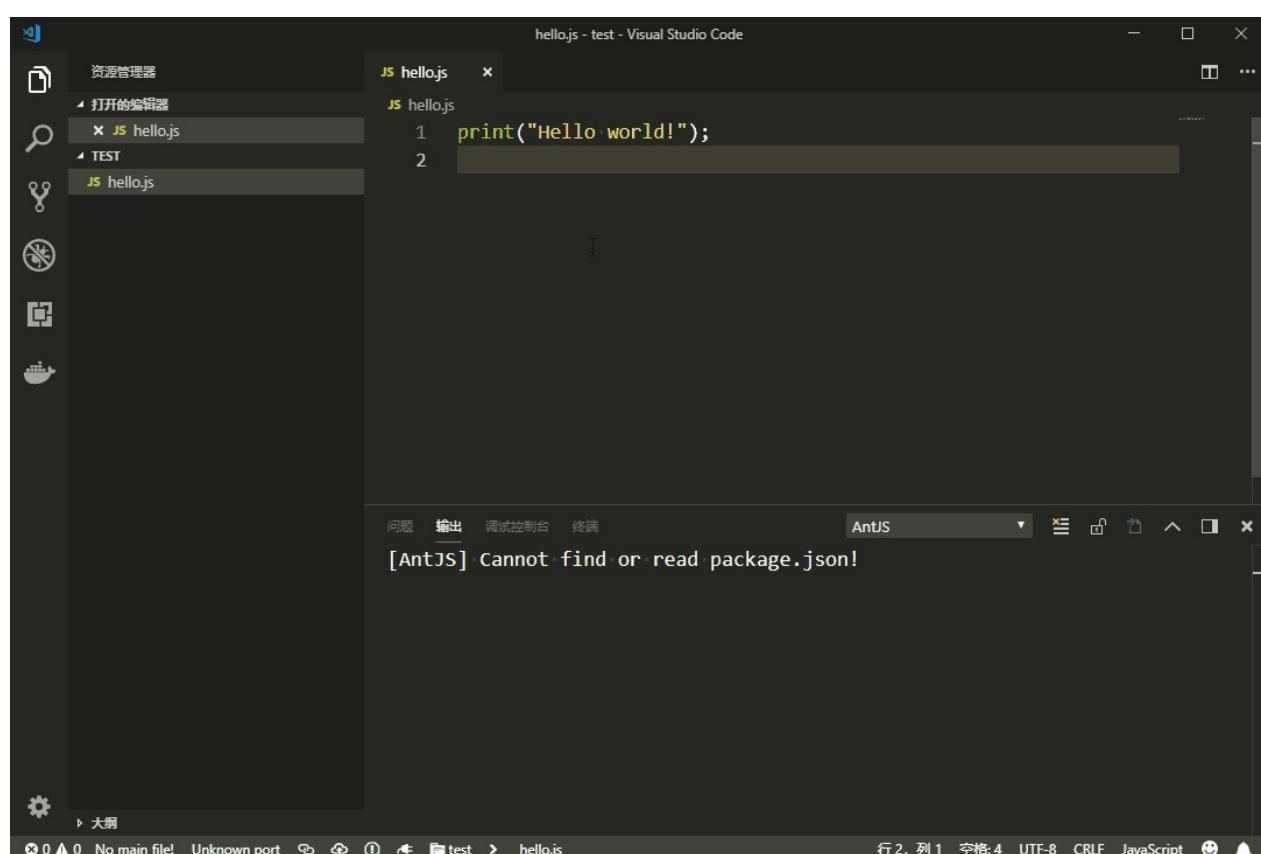


2.2.4.3 串口连接/断开连接

使用按钮：点击  按钮，若设备已连接，则断开连接，否则建立连接

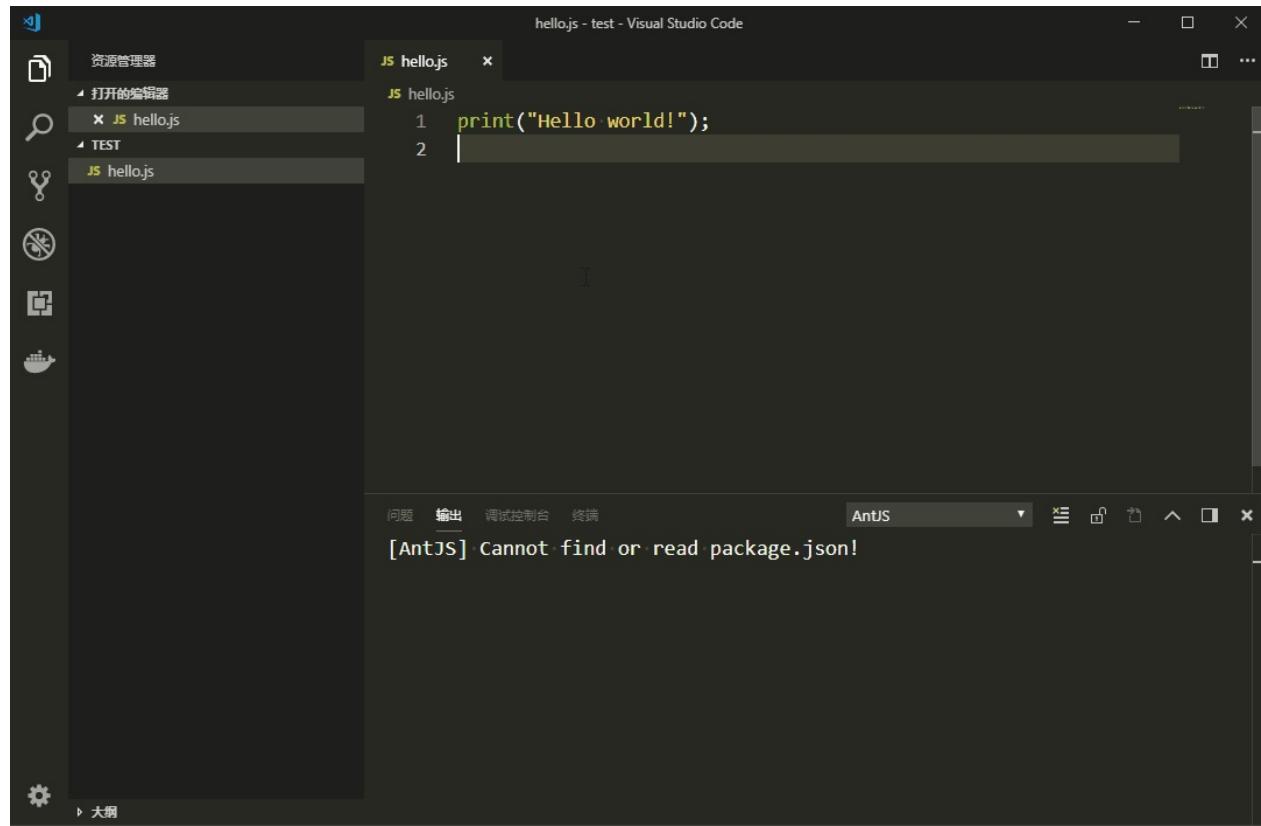


使用命令：使用 **Ctrl+Shift+p**，在弹出的命令列表中选择 **AntJS: AntJS Connect COM Port**，和设备建立连接，选择 **AntJS: AntJS Disconnect COM Port**，和设备断开连接

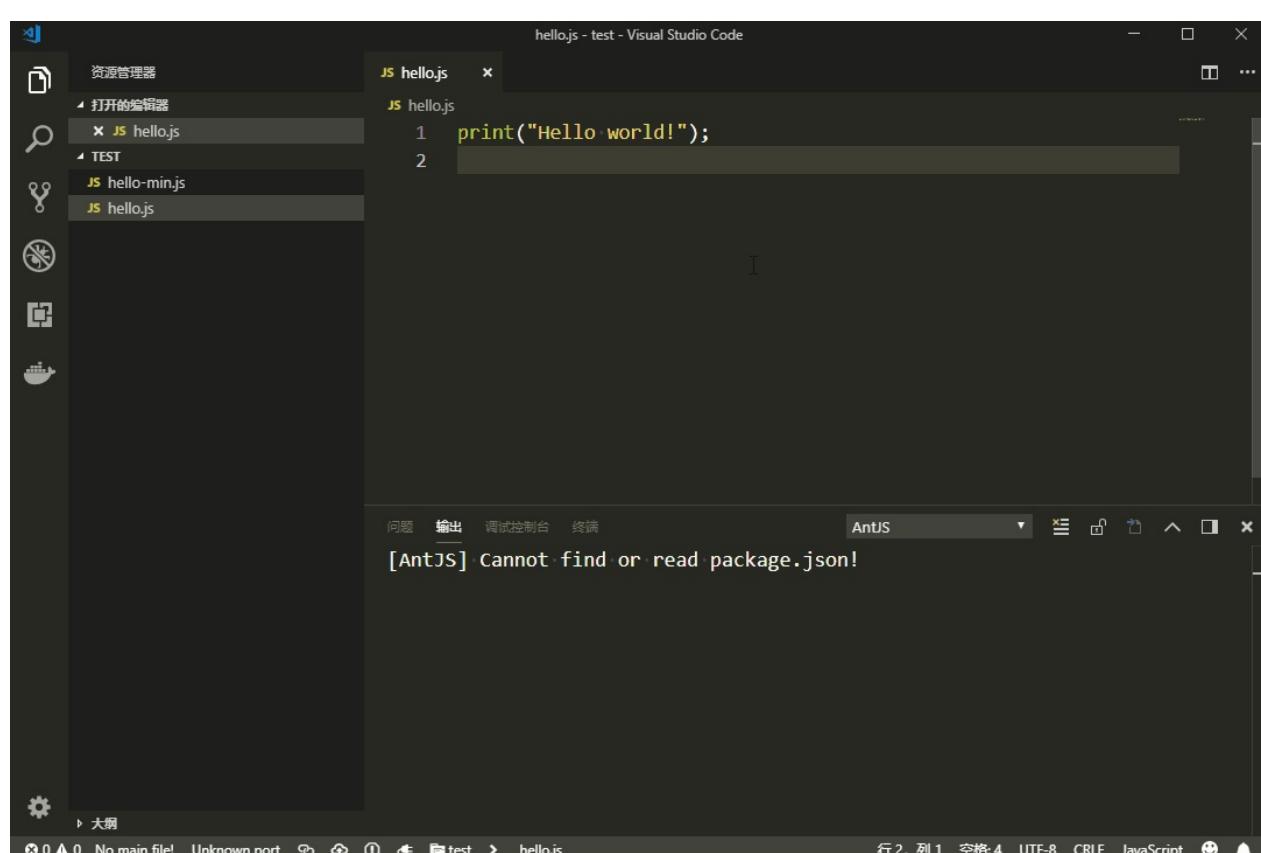


2.2.4.4 文件部署

使用按钮：点击 按钮，若已设置好文件和串口，且已连接，则直接部署文件，否则会进行相关提示



使用命令：使用 **Ctrl+Shift+p**，在弹出的命令列表中选择 **AntJS: AntJS Deploy File to Device**，若已设置好文件和串口，且已连接，则直接部署文件，否则会进行相关提示

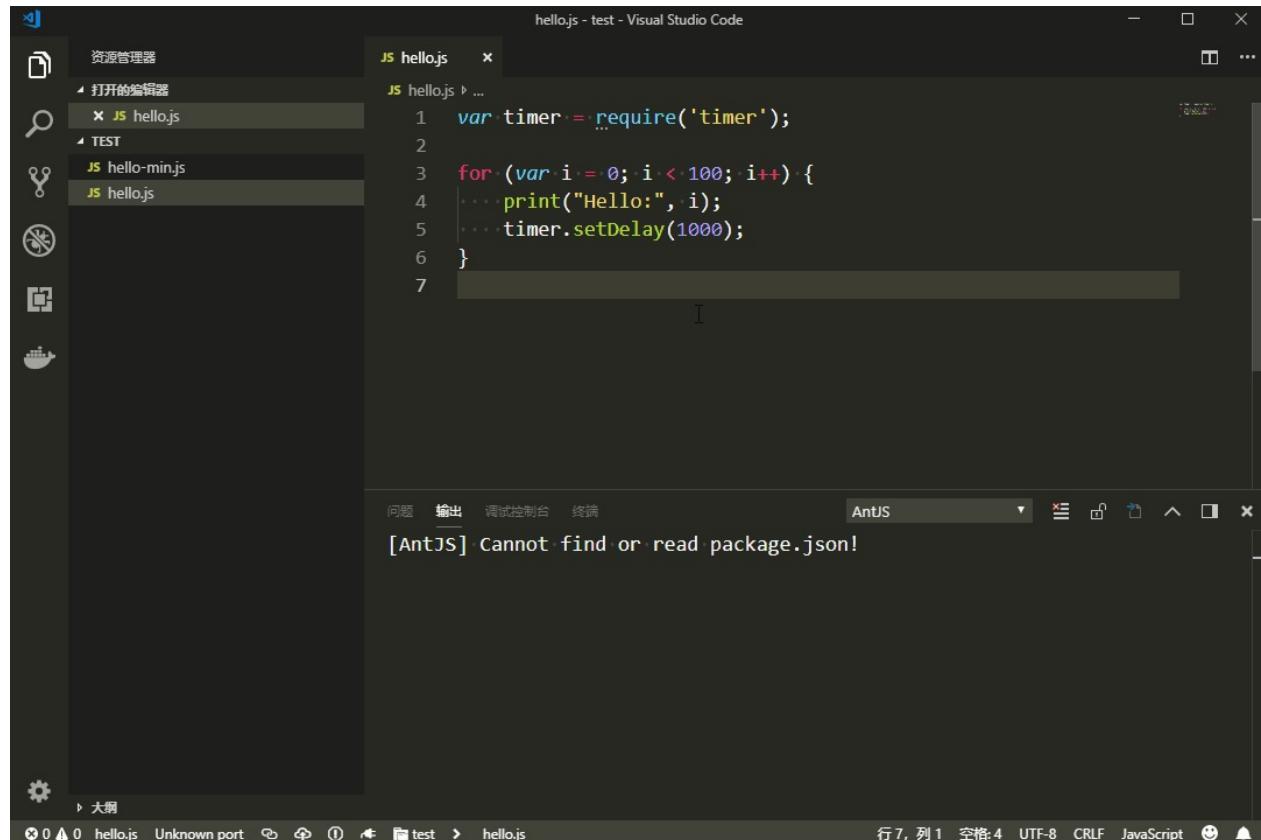


注意事项：

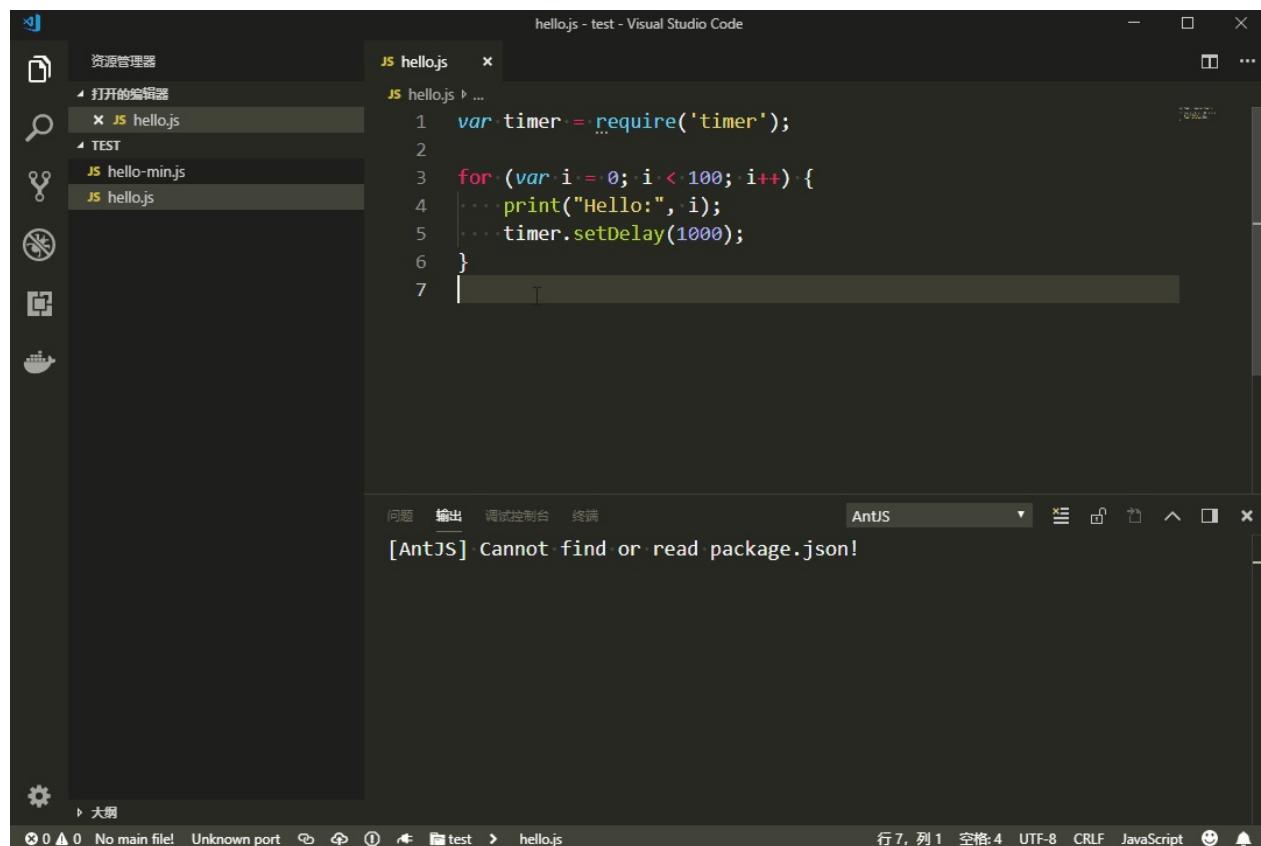
- 不支持部署空文件
- 部署文件前会将JS文件压缩，该功能不支持在严格模式中被禁止的语法

2.2.4.5 中止引擎

使用按钮：点击①按钮，若设备已连接，则会向设备发送stop script命令使得设备中止JS引擎的运行（由于设备和PC通过串口连接，在直观上可能存在延迟），否则会进行相关提示



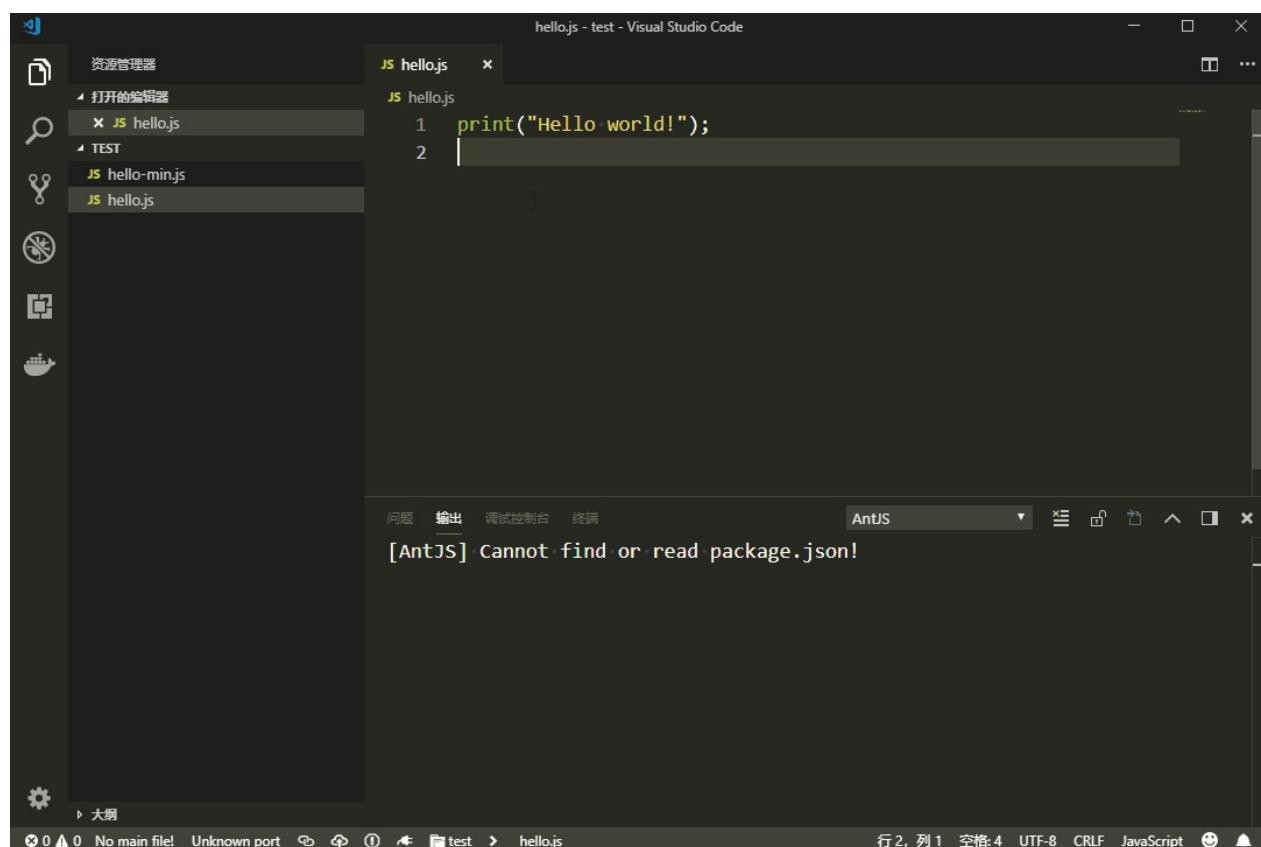
使用命令：使用 **Ctrl+Shift+p**，在弹出的命令列表中选择 **AntJS: AntJS Stop AntJS Engine**，若设备已连接，则会向设备发送stop script命令使得设备中止JS引擎的运行（由于设备和PC通过串口连接，在直观上可能存在延迟），否则会进行相关提示



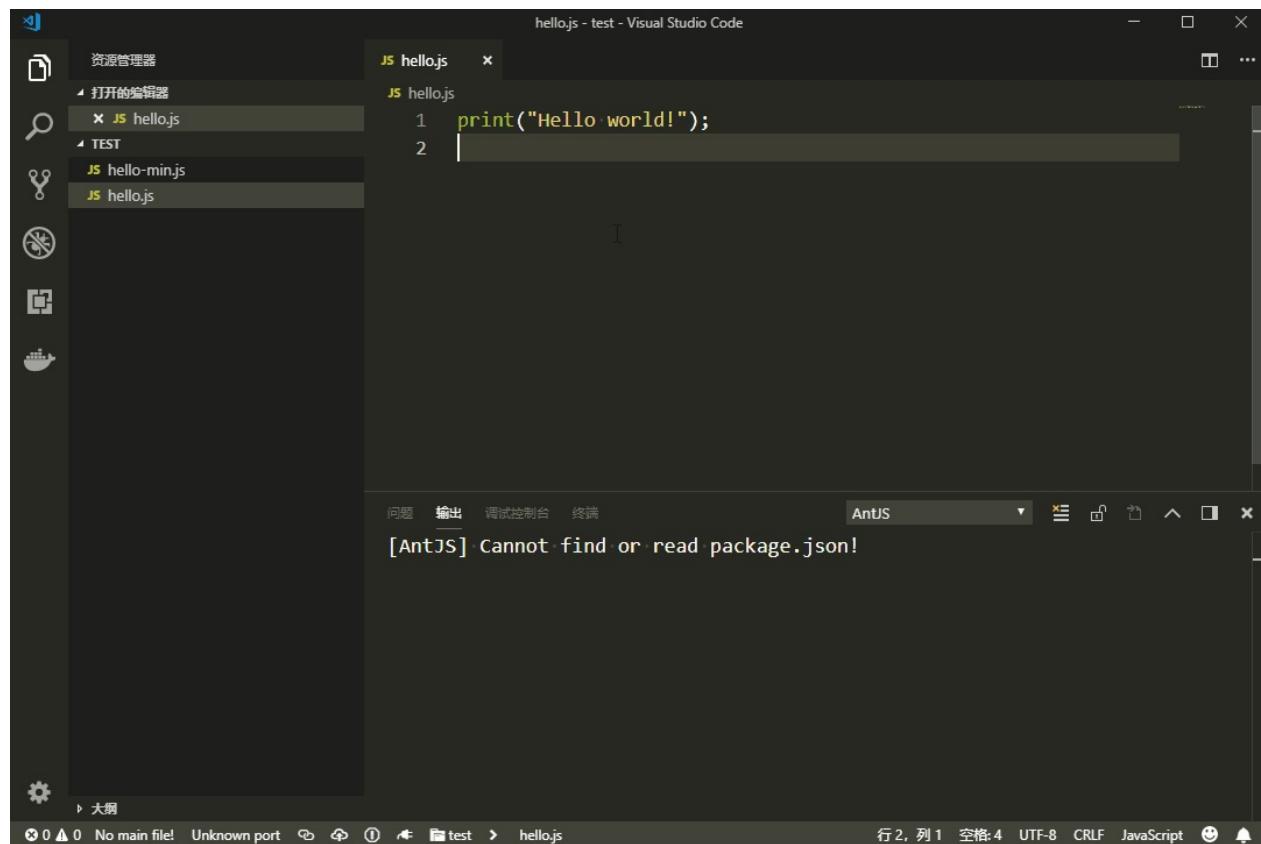
注意：此功能仅适用于循环结构（for循环、while循环等）

2.2.4.6 重启设备

使用按钮：点击 按钮，若已设置好串口，且已连接，则会向设备发送reboot命令使设备reboot，否则会进行相关提示



使用命令：使用 `Ctrl+Shift+p`，在弹出的命令列表中选择 `AntJS: AntJS Reboot Device`，若已设置好串口，且已连接，则会向设备发送reboot命令使设备reboot，否则会进行相关提示



2.3 初始JS文件加载工具

该工具可将JS文件加载至bin文件，生成的bin文件烧录到芯片即可实现JS文件内置到芯片中。

MapleJS内置JS脚本有如下两个特性：

1. 支持乒乓缓冲机制，可内置ping.js和pang.js两个脚本，MapleJS在启动时会依据配置文件加载ping.js或pang.js，其中配置文件将由该工具自动加载至bin文件中。乒乓缓冲机制在内部存在两个缓冲区，其中一个用来存储当前正在运行的JS脚本，当部署新JS脚本或升级JS脚本时会存储至另外一个缓冲区，从而在接收的同时保证当前运行的JS脚本不受影响，当接收完整之后，新接收的JS脚本将切换成正在运行的JS脚本，从而实现部署或升级时平滑过渡。
2. 支持两种方式存储内置JS脚本：
 - 基于raw flash存储内置JS脚本。（注：MapleJS目前版本不支持raw flash方案存储JS文件，但是之后版本将支持，将来工具也无需更新即可使用）
 - 基于spiffs文件系统储存内置JS脚本。

该工具需通过命令行参数输入MapleJS内置JS脚本特性及其他参数，执行成功时将生成bin文件并打印烧录的起始地址，可使用ImageTool工具将bin文件烧录至芯片。

2.3.1 工具使用步骤

步骤1： 打开命令提示符(cmd)并切换至init-js-load-tool.exe工具目录(在Windows文件浏览器地址栏输入cmd并回车可快速打开cmd并切到该目录)。



步骤2： 根据实际情况输入相应命令及参数生成bin文件。

下面举例说明

MapleJS基于spiffs文件系统存储内置JS文件，需将本目录下test.js做为ping.js、test2.js作为pang.js，启动时最终加载ping.js：

```
init-js-load-tool.exe -p spiffs -ping test.js -pang test2.js -u ping -s 65536 spiffs_test.bin
```



生成成功时，将打印：

```
Success!
Please Download XXXX.bin to the 0XXXX address
```

具体命令参数详情解释如下：

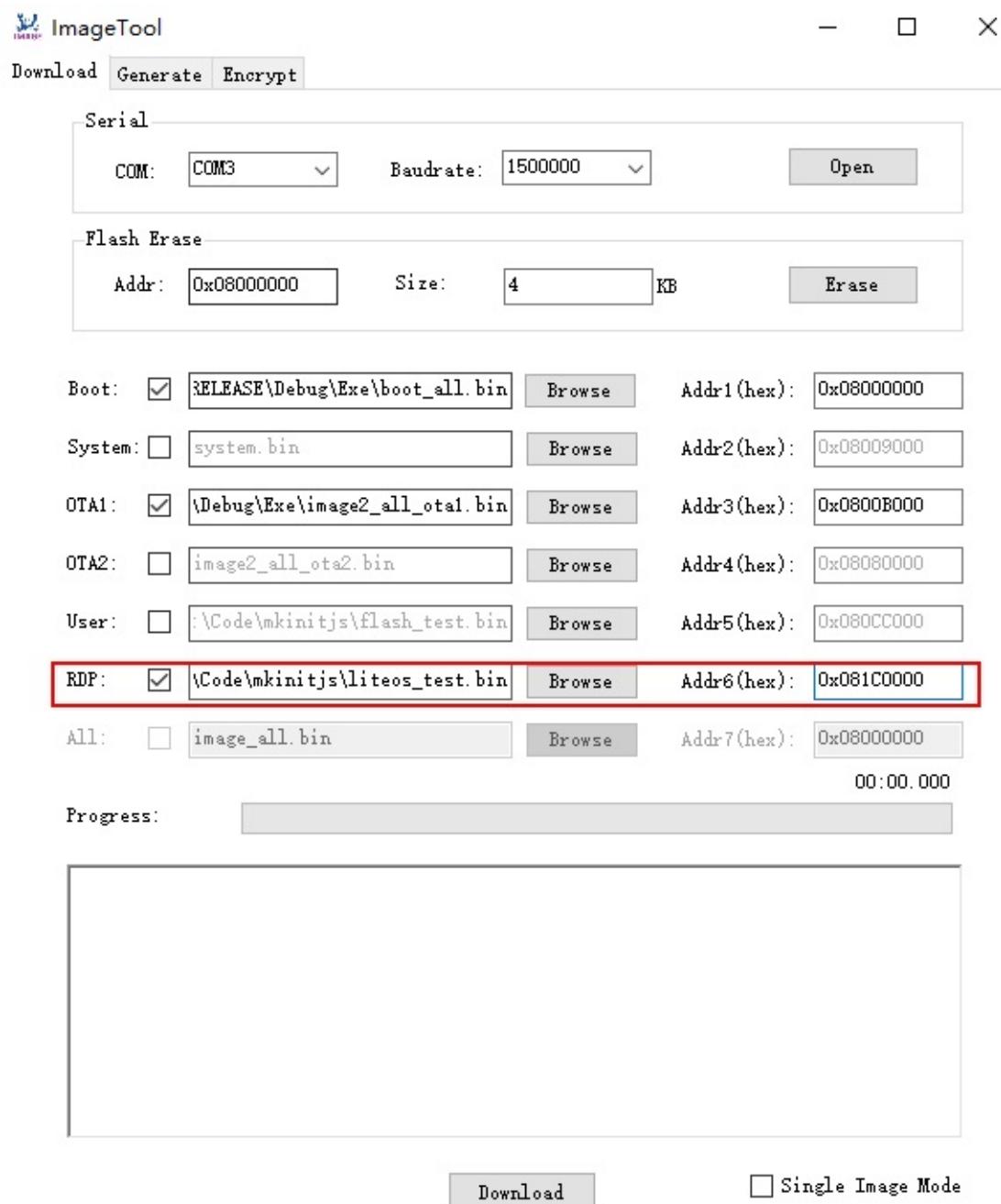
```
-p: 为芯片的存储文件的方式，选项只可为flash或spiffs，flash代表Raw Flash存储方式，spiffs代表spiffs文件系统方式，目前版本只能选择spiffs，选项必填
-ping: 指定目标文件ping.js的原始js文件路径，此选项必填
-pang: 指定目标文件pang.js的原始js文件路径，此选项非必填且无默认值
-u: 指定启动时加载ping.js还是pang.js，选项只可为ping或pang，其中ping表示加载ping.js，pang表示加载pang.js，此选项默认值为ping
-s: 指定spiffs模式时输出文件的大小，此参数须为4096的倍数，此选项有默认值为65536(64K)，取值范围为[12288, 262144]
-c: 指定芯片，当前只支持RTL8710芯片，此参数只可为RTL8710，RTL8710表示芯片为RTL8710，此选项默认值为RTL8710
最后一个参数：为目标输出文件，此选项必填且无默认值
```

步骤3: 选择需烧录的文件及输入地址信息。

在RDP栏中（或其他栏）选择步骤2生成的文件，并填入步骤2打印的烧录起始地址，具体如下图：

```
E:\初始JS文件加载工具>init-js-load-tool.exe -p flash -ping test.js -pang test2.js -u ping flash_test.bin
Success!
Please Download flash_test.bin to the [0x080CC000] address
```

注：请以步骤2打印的地址为准，因采用不同方式存储JS文件时其烧录的起始地址不同。



步骤4: 烧录到芯片，烧录方法和烧录其他固件一样，此处省略。

- 3 语法规格
 - 3.1 语法标准
 - 3.2 裁剪规格
 - 3.2.1 裁剪delete操作符 (ES 5.1 11.4.1)
 - 3.2.2 裁剪void运算符 (ES 5.1 11.4.2)
 - 3.2.3 裁剪typeof运算符 (ES 5.1 11.4.3)
 - 3.2.4 裁剪instanceof运算符 (ES 5.1 11.8.6)
 - 3.2.5 裁剪in运算符 (ES 5.1 11.8.7)
 - 3.2.6 裁剪do...while语句 (ES 5.1 12.6.1)
 - 3.2.7 裁剪for...in语句 (ES 5.1 12.6.4)
 - 3.2.8 裁剪with语句 (ES 5.1 12.10)
 - 3.2.9 裁剪label语句 (ES 5.1 12.12)
 - 3.2.10 裁剪throw语句 (ES 5.1 12.13)
 - 3.2.11 裁剪try语句 (ES 5.1 12.14)
 - 3.2.12 裁剪debugger语句 (ES 5.1 12.15)
 - 3.2.13 裁剪NaN属性 (ES 5.1 15.1.1.1)
 - 3.2.14 裁剪Infinity属性 (ES 5.1 15.1.1.2)
 - 3.2.15 裁剪parseInt函数 (ES 5.1 15.1.2.2)
 - 3.2.16 裁剪parseFloat函数 (ES 5.1 15.1.2.3)
 - 3.2.17 裁剪isNaN函数 (ES 5.1 15.1.2.4)
 - 3.2.18 裁剪isFinite函数 (ES 5.1 15.1.2.5)
 - 3.2.19 裁剪decodeURI函数 (ES 5.1 15.1.3.1)
 - 3.2.20 裁剪decodeURIComponent函数 (ES 5.1 15.1.3.2)
 - 3.2.21 裁剪encodeURI函数 (ES 5.1 15.1.3.3)
 - 3.2.22 裁剪encodeURIComponent函数 (ES 5.1 15.1.3.4)
 - 3.2.23 裁剪String对象原型substr方法 (ES 5.1 B.2.3)
 - 3.2.24 裁剪Error对象 (ES 5.1 15.11)
 - 3.2.25 不带parser时不支持eval语句 (ES 5.1 15.1.2.1)
 - 3.2.26 裁剪escape函数 (ES 5.1 B.2.1)
 - 3.2.27 不带parser时不支持Function构造函数 (ES 5.1 15.3.1-15.3.2)
 - 3.2.28 裁剪unescape函数 (ES 5.1 B.2.2)
 - 示例:
 - 3.2.29 裁剪Date对象 (ES 5.1 15.9)
 - 示例:
 - 3.2.30 裁剪正则表达式 (ES 5.1 7.8.5 15.10)
 - 示例:

3 语法规格

本章介绍MapleJS支持的JavaScript语法规格。整体上，为了实现引擎轻量化，结合IoT设备侧的使用场景，MapleJS基于通用语法规格进行了部分语法裁剪，由于通用规格可参考业界标准，因此本章着重介绍裁剪部分。

3.1 语法标准

`JavaScript` 是一种动态类型、弱类型、基于原型的脚本语言。变量使用之前不需要类型声明，通常变量的类型是被赋值的那个值的类型。计算时可以不同类型之间对使用者透明地隐式转换，即使类型不正确，也能通过隐式转换来得到正确的类型。新对象继承对象（作为模版），将自身的属性共享给新对象，模版对象称为原型。这样新对象实例化后不但可以享有自己创建时和运行时定义的属性，而且可以享有原型对象的属性。

`JavaScript` 的核心是 `ECMAScript`，而 `ECMAScript` 是一个由 ECMA International 进行标准化，TC39 委员会进行监督的语言。它规定了语言的组成部分：语法、类型、语句、关键字、保留字、操作符、对象。我们支持的语法规范为 `ECMAScript 5 (ES5)`。它是 `ECMAScript` 的第五版修订，于 2009 年完成标准化。这个规范在 Web 领域，已经被所有现代浏览器相当完全

的实现了。

基于目前 IoT 设备的特点，我们基于 ES 5.1 进行了一些语法的裁剪，即 3.2 裁剪规格 展示的裁剪项不被支持。如果使用 3.2 节展示的语法，则会得到未定义错误。

3.2 裁剪规格

3.2.1 裁剪 delete 操作符 (ES 5.1 11.4.1)

`delete` 操作符用于删除对象的某个属性；如果没有指向这个属性的引用，那它最终会被释放。

示例：

```
var o = {};
o.x = new Object();
delete o.x;
```

3.2.2 裁剪 void 运算符 (ES 5.1 11.4.2)

`void` 运算符对给定的表达式进行求值，然后返回 `undefined`。

示例：

```
void(0);
```

3.2.3 裁剪 typeof 运算符 (ES 5.1 11.4.3)

`typeof` 操作符返回一个字符串，表示未经计算的操作数的类型。

示例：

```
print(typeof 42)           // expected output: number
print(typeof 'balabala') // expected output: string
print(typeof true)        // expected output: boolean
Function("return typeof this;")
```

3.2.4 裁剪 instanceof 运算符 (ES 5.1 11.8.6)

`instanceof` 运算符用来检测 `constructor.prototype` 是否存在于参数 `object` 的原型链上。

示例：

```
function C(){}
var o = new C();
o instanceof C; // expected output: true
                // 因为Object.getPrototypeOf(o) === C.prototype
```

3.2.5 裁剪 in 运算符 (ES 5.1 11.8.7)

如果指定的属性在指定的对象或其原型链中，则 `in` 运算符返回 `true`。

示例1：

```
var arr = [1,2,3];
2 in arr // true
3 in arr // true
```

示例2:

```
var evalStr =
'for (var x in this) {\n' + '}\n';
eval(evalStr);
```

3.2.6 裁剪do...while语句 (ES 5.1 12.6.1)

`do...while` 语句创建一个执行指定语句的循环，直到 `condition` 值为 `false`。在执行 `statement` 后检测 `condition`，所以指定的 `statement` 至少执行一次。

```
do
  statement
while (condition);
```

statement

执行至少一次的语句，并在每次 `condition` 值为真时重新执行。想执行多行语句，可使用 `block` 语句（`{ ... }`）包裹这些语句。

condition

循环中每次都会计算的表达式。如果 `condition` 值为真，`statement` 会再次执行。当 `condition` 值为假，则跳到 `do...while` 之后的语句。

示例：

```
var i = 0;
do {
  i += 2;
} while( i<10 ); //do-while
```

3.2.7 裁剪for...in语句 (ES 5.1 12.6.4)

`for...in` 语句以任意顺序遍历一个对象的可枚举属性。对于每个不同的属性，语句都会被执行。

```
for (variable in object) {...}
```

variable

在每次迭代时，将不同的属性名分配给变量。

object

被迭代枚举其属性的对象。

示例：

```
var arr = [1,2,3];
for (var i in arr) { //for in
    var j = i;
}
```

3.2.8 裁剪with语句 (ES 5.1 12.10)

with 语句 扩展一个语句的作用域链。

```
with (expression) {
    statement
}
```

expression

将给定的表达式添加到在评估语句时使用的作用域链上。表达式周围的括号是必需的。

statement

任何语句。要执行多个语句，请使用一个块语句 ({ ... }) 对这些语句进行分组。

示例:

```
var obj2 = { x: 2 };
with (obj2) { //with
    var t = x;
}
```

3.2.9 裁剪 label 语句 (ES 5.1 12.12)

label语句可以和 break 或 continue 语句一起使用。标记就是在一条语句前面加个可以引用的标识符。

```
label :
    statement
```

label

任何不是保留关键字的 JavaScript 标识符。

statement

语句。break 可用于任何标记语句，而 continue 可用于循环标记语句。

示例:

```

var i, j;

loop1:
for (i = 0; i < 3; i++) {      //The first for statement is labeled "loop1"
    loop2:
    for (j = 0; j < 3; j++) {  //The second for statement is labeled "loop2"
        if (i == 1 && j == 1) {
            continue loop1;
        }
        print("i = " + i + ", j = " + j);
    }
}

// expected output:
// "i = 0, j = 0"
// "i = 0, j = 1"
// "i = 0, j = 2"
// "i = 1, j = 0"
// "i = 2, j = 0"
// "i = 2, j = 1"
// "i = 2, j = 2"

```

3.2.10 裁剪throw语句 (ES 5.1 12.13)

`throw` 语句用来抛出一个用户自定义的异常。当前函数的执行将被停止 (`throw` 之后的语句将不会执行)，并且控制将被传递到调用堆栈中的第一个`catch`块。如果调用者函数中没有 `catch` 块，程序将会终止。

示例：

```

function getRectArea(width, height) {
    if (isNaN(width) || isNaN(height)) {
        throw "Parameter is not a number!";
    }
}

try {
    getRectArea(3, 'A');
}
catch(e) {
    print(e);
    // expected output: "Parameter is not a number!"
}

```

3.2.11 裁剪try语句 (ES 5.1 12.14)

`try...catch` 语句将能引发错误的代码放在 `try` 块中，并且对应一个响应，然后有异常被抛出。

```

try {
    try_statements
}
[catch (exception_var_1 if condition_1) { // non-standard
    catch_statements_1
}]
...
[catch (exception_var_2) {
    catch_statements_2
}]
[finally {
    finally_statements
}]

```

try_statements

需要被执行的语句。

catch_statements_1, catch_statements_2

如果在 `try` 块里有异常被抛出时执行的语句。

`exception_var_1, exception_var_2`

用于保存关联`catch`子句的异常对象的标识符。

`condition_1`

一个条件表达式。

`finally_statements`

在 `try` 语句块之后执行的语句块。无论是否有异常抛出或捕获这些语句都将执行。

示例：

```
try {
    throw "myException"; // generates an exception
}
catch (e) {
    // statements to handle any exceptions
    logMyErrors(e); // pass exception object to error handler
}
```

3.2.12 裁剪`debugger`语句 (ES 5.1 12.15)

`debugger` 语句调用任何可用的调试功能，例如设置断点。如果没有调试功能可用，则此语句不起作用。

示例：

```
debugger; // do potentially buggy stuff to examine, step through, etc.
```

3.2.13 裁剪`NaN`属性 (ES 5.1 15.1.1.1)

`NaN` 属性用于引用特殊的非数字值。

示例：

```
var i = NaN;
```

3.2.14 裁剪`Infinity`属性 (ES 5.1 15.1.1.2)

全局属性 `Infinity` 是一个数值，表示无穷大。

示例：

```
var i = Infinity;
```

3.2.15 裁剪`parseInt`函数 (ES 5.1 15.1.2.2)

`parseInt()` 函数可解析一个字符串，并返回一个整数。

示例：

```
var i = parseInt("8", 10);
```

3.2.16 裁剪parseFloat函数 (ES 5.1 15.1.2.3)

`parseFloat()` 函数可解析一个字符串，并返回一个浮点数。

示例：

```
var i = parseFloat("1.2");
```

3.2.17 裁剪isNaN函数 (ES 5.1 15.1.2.4)

`isNaN()` 函数用于检查其参数是否是非数字值。

示例：

```
isNaN(3); // false
```

3.2.18 裁剪isFinite函数 (ES 5.1 15.1.2.5)

该全局 `isFinite()` 函数用来判断被传入的参数值是否为一个有限数值（`finite number`）。在必要情况下，参数会首先转为一个数值。

示例：

```
isFinite(3); // true
```

3.2.19 裁剪decodeURI函数 (ES 5.1 15.1.3.1)

`decodeURI()` 函数可对 `encodeURI()` 函数编码过的 `URI` 进行解码。

示例：

```
var i = "http://www.huawei.com";
decodeURI(i);
```

3.2.20 裁剪decodeURIComponent函数 (ES 5.1 15.1.3.2)

`decodeURIComponent()` 函数可对 `encodeURIComponent()` 函数编码的 `URI` 进行解码。

示例：

```
var i = "http://www.huawei.com";
decodeURIComponent(i);
```

3.2.21 裁剪encodeURIComponent函数 (ES 5.1 15.1.3.3)

`encodeURIComponent()` 函数可把字符串作为 `URI` 进行编码。

示例：

```
var i = "http://www.huawei.com";
encodeURI(i);
```

3.2.22 裁剪encodeURIComponent函数 (ES 5.1 15.1.3.4)

`encodeURIComponent()` 函数可把字符串作为 `URI` 组件进行编码。

示例：

```
var i = "http://www.huawei.com";
encodeURIComponent(i);
```

3.2.23 裁剪String对象原型substr方法 (ES 5.1 B.2.3)

`substr` 方法有两个参数 `start` 和 `length`，用于将`this`对象转换为一个字符串，并返回这个字符串中从 `start` 位置一直到 `length` 位置（或如果 `length` 是 `undefined`，就一直到字符串结束位置）的字符组成的子串。

示例：

```
var str = "Hello world!";
var substr = str.substr(3,7);
```

3.2.24 裁剪Error对象 (ES 5.1 15.11)

`Error` 对象的实例在运行时遇到错误的情况下会被当做异常抛出。`Error` 对象也可以作为用户自定义异常类的基对象。

```
15.11 Error Objects
  15.11.1 The Error Constructor Called as a Function
  15.11.2 The Error Constructor
  15.11.3 Properties of the Error Constructor
  15.11.4 Properties of the Error Prototype Object
  15.11.5 Properties of Error Instances
  15.11.6 Native Error Types Used in This Standard
    15.11.6.1 EvalError
    15.11.6.2 RangeError
    15.11.6.3 ReferenceError
    15.11.6.4 SyntaxError
    15.11.6.5 TypeError
    15.11.6.6 URIError
  15.11.7 NativeError Object Structure
```

除了通用的 `Error` 构造函数外，JavaScript 还有6个其他类型的错误构造函数：`EvalError`：`eval` 函数没有被正确执行时抛出此错误，该错误类型已经不再在ES5中出现了，只是为了保证与以前代码兼容才继续保留。`RangeError`：当一个值超出有效范围时发生的错误，主要有数组长度为负数、`number` 对象的方法参数超出范围、函数堆栈超过最大值。`ReferenceError`：引用一个不存在的变量时发生的错误或者将一个值分配给无法分配的对象，比如对函数的运行结果或者 `this` 赋值。

`SyntaxError`：解析代码时发生的语法错误，比如变量名错误、缺少括号等。`TypeError`：当变量或参数不是预期类型时发生的错误，比如对字符串、布尔值、数值等原始类型的值使用 `new` 命令就会抛出这种错误。`URIError`：当 `URI` 相关函数的参数不正确时抛出的错误，主要涉

及 `encodeURI`、`decodeURI`、`encodeURIComponent`、`decodeURIComponent`、`escape`、`unescape` 这六个函数。

示例1：

```
try {
  throw new Error("Whoops!");
} catch (e) {
  print(e.name + ": " + e.message);
}
```

示例2:

```
var x = new Error("This is an error");
if (x.constructor == Error)
    print("Object is an error.");
```

示例3:

```
var check = function(num) {
    if (num < 0 || num > 100) {
        throw new RangeError('Parameter must be between ' + 0 + ' and ' + 100);
    }
};

try {
    check(500);
}
catch (e) {
    if (e instanceof RangeError) {
        print(e.name + ": " + e.message);
    }
}
```

示例4:

```
try {
    var a = b;
}
catch (e) {
    print(e instanceof ReferenceError);
}
```

3.2.25 不带parser时不支持eval语句 (ES 5.1 15.1.2.1)

`eval()` 是全局对象的一个函数属性。`eval()` 的参数是一个字符串。如果字符串表示的是表达式，`eval()` 会对表达式进行求值。如果参数表示一个或多个 JavaScript 语句，那么 `eval()` 就会执行这些语句。

示例:

```
eval('1+1');
eval('var a = 1;')
```

3.2.26 裁剪escape函数 (ES 5.1 B.2.1)

`escape` 函数是全局对象的一个属性。它通过将一些字符替换成十六进制转义序列，计算出一个新字符串值。对于代码单元小于等于0xFF的被替换字符，使用%xx格式的两位数转义序列。对于代码单元大于0xFF的被替换字符，使用%uxxxx格式的四位数转义序列。

示例:

```
escape("hello!"); //expected output: "hello%21"
```

3.2.27 不带parser时不支持Function构造函数 (ES 5.1 15.3.1-15.3.2)

`Function` 构造函数创建一个新的`Function`对象。

示例:

```
var f = new Function('a', 'b', 'return a + b');
```

3.2.28 裁剪unescape函数(ES 5.1 B.2.2)

unescape 函数可对 escape 编码的字符串进行解码。

示例：

```
unescape("hello%21"); //expected output: "hello!"
```

3.2.29 裁剪Date对象 (ES 5.1 15.9)

Date 对象用于处理日期与时间。

示例：

```
var today = new Date()
var d1 = new Date("September 29, 2018 19:30:00")
var d2 = new Date(18,9,29)
var d3 = new Date(18,9,29,19,30,0)
```

3.2.30 裁剪正则表达式 (ES 5.1 7.8.5 15.10)

正则表达式描述了一种字符串匹配的模式。

示例：

```
var re = /.at/i; //正则表达式字面量，匹配第一个以"at"结尾的3个字符的组合，不区分大小写
var r = RegExp ("a"); //正则表达式对象
```

- 4 模块接口介绍

4 模块接口介绍

模块接口主要是为了给开发者提供可以直接操作硬件、系统功能、网络连接的能力。开发者可以在使用具体某些模块的时候，在JS代码中加载相应模块，从而获得该模块接口所提供的能力。

以 `gpio` 为例，当开发者需要操作该系列硬件的时候，只需要在js中执行 `var gpio = require('gpio')`，即可通过访问该 `gpio` 变量的 API 去操作硬件。

本章描述中所有与硬件相关的例子，都是以realtek 8710板子为例。

4.1 Timer

4.1.1 介绍

Timer模块可以提供异步的周期性和一次性定时器功能，以及同步的延时功能。使用Timer模块，首先需要使用require语句获得该模块句柄：`var timerHandler = require('timer');`。

4.1.2 模块接口介绍

4.1.2.1 异步周期性定时器接口

```
timerID setInterval (func, interval, arg);
```

功能

提供异步的，周期性的定时器功能

参数列表

- `func`: 定时器超时后的回调函数；
- `interval`: 定时器的周期长度，以毫秒（ms）为单位，类型为整数number（`number>=0`）；注意：当interval过小时，会导致引擎来不及处理，导致引擎的事件队列溢出，所以建议`number>=10`。
- `arg`: 传递给回调函数的参数。只允许一个参数，如果需要传递多个参数，需要把多个参数打包在一个结构体里。如果没有参数，该接口将默认传递`undefined`；
- `timerID`: 返回值，timer的ID，可以用于timer的关闭。

4.1.2.2 异步一次性定时器接口

```
timerID setDelay (func, delay, arg);
```

功能

提供异步的，一次性的定时器功能，即超时后func只被调用一次，timer随后被关闭。

参数列表

- `func`: 定时器超时后的回调函数；
- `delay`: 定时器的周期长度，以毫秒（ms）为单位，类型为整数number（`number>=0`）；
- `arg`: 传递给回调函数的参数。只允许一个参数，如果需要传递多个参数，需要把多个参数打包在一个结构体里。如果没有参数，该接口将默认传递`undefined`；
- `timerID`: 返回值，timer的ID，可以用于timer的关闭。

4.1.2.3 同步延时接口

```
void setDelay (delay);
```

功能

提供同步的延时功能。

参数列表

- `delay`: 延时的时间长度，以毫秒（ms）为单位，类型为整数number（`number>=0`）；

- 无返回值。

4.1.2.4 定时器关闭

```
void stopTimer (tiemrID);
```

功能

停止异步的定时器，包括周期性和一次性的。

参数列表

- tiemrID: 定时器的id;
- 无返回值。

4.1.3 约束

同时生效的异步定时器最大数为10。

4.1.4 Sample

用例1：从某端口读取数据（大于**100**字节），每隔**100ms**查询一次，读完退出；

```
var tim = require('timer');
var len = 0;
for (;;) {
  /* pseudo code, read data from a port. */
  len += port.read(...);

  if (len < 100) {
    tim.setDelay(100);
  } else {
    break;
  }
}
```

用例2：使用异步**timer**实现相同功能；

```
var tim = require('timer');
var len = 0;
var id = tim.setInterval(function() {
  /* pseudo code, read data from a port. */
  len += port.read(...);

  if (len > 100) {
    tim.stopTimer(id);
  }
}, 100);
```

4.2 GPIO

4.2.1 介绍

GPIO是一种通用的I/O端口，GPIO模块可以允许用户通过API调用驱动GPIO端口进行读、写、监听等功能。使用GPIO模块，首先需要使用require语句获得该模块句柄：`var gpio = require('gpio');`

4.2.2 模块接口介绍

4.2.2.1 GPIO端口方向的枚举定义

- `gpio.DIR_IN`: input方向;
- `gpio.DIR_OUT`: output方向;
- `gpio.DIR_INOUT`: input/output方向;

4.2.2.2 GPIO端口状态的枚举定义

- `gpio.PULLNONE`: 不拉输入/输出;
- `gpio.PULLUP`: 上拉输入/输出;
- `gpio.PULLDOWN`: 下拉输入/输出;
- `gpio.OPENDRAIN`: 开漏输出，仅在端口方向为输出时可用;

4.2.2.3 GPIO监听事件的枚举定义

- `gpio.INT_RISING`: 上升沿监听;
- `gpio.INT_FALLING`: 下降沿监听;
- `gpio.INT_ANY`: 任意边沿监听;

4.2.2.4 打开一个GPIO端口

`GPIOPin gpio.open(config)`

功能

根据配置打开GPIO端口

参数列表

- `config`: GPIO端口的配置，包括四个属性：
 - `pin`: 设置GPIO端口对应的管脚号，以realtek为例，开发板有效的GPIO管脚为0, 5, 12, 18, 19, 22, 23, 29, 30;
 - `mode`: 设置端口输入/输出模式，具体值参考4.2.2.2;
 - `dir`: 设置端口的方向，具体值参考4.2.2.1;
- `GPIOPin`: 返回值，GPIO接口对象;

4.2.2.5 向GPIO端口写出

`void gpioPin.write (number val)`

功能

向GPIO端口写入值。

参数列表

- **val:** 向GPIO端口写入的值，取值为1，或者0；如果写入值不是0或1，输入值大于0视为1，小于0的值视为0；
- 无返回值。

4.2.2.6 从GPIO端口读入

```
number gpioPin.read ()
```

功能

从GPIO端口读入值。

参数列表

- 无参数；
- 返回值：number类型，当端口处于激活态时返回1，否则返回0；

4.2.2.7 关闭一个GPIO端口

```
void gpioPin.close ()
```

功能

关闭GPIO端口。

参数列表

- 无参数；
- 无返回值。

4.2.2.8 GPIO端口监听

```
void gpioPin.on (event, func, arg)
```

功能

监听GPIO端口的事件，比如上升沿、下降沿或者二者兼顾，从而调用相应的回调函数。新增回调函数会使旧的回调函数失效。

参数列表

- **event:** 监听的事件类型，具体值参考4.2.2.3；
- **func:** 回调函数，当监听事件发生时调用该函数；
- **arg:** 传递给回调函数的参数，只允许一个参数，如果需要传递多个参数，需要把多个参数打包在一个结构体里。如果没有参数，该接口将默认传递undefined；
- 无返回值；

4.2.3 约束

无。

4.2.4 样例

用例1： 初始化GPIO端口，并写入1；

```
var gpio = require('gpio');
var config={
  pin:5,
  dir:gpio.DIR_OUT,
  mode:gpio.PULLNONE
};

var pin= gpio.open(config);
pin.write(1);
```

用例2：监听**GPIO**端口的上升沿事件，当收到该事件时点亮**led**灯；

```
var gpio = require('gpio');
var config = {
  pin: 5,
  dir: gpio.DIR_IN,
  mode: gpio.PULLNONE
};

var pin = gpio.open(config);

config = {
  pin: 12,
  dir: gpio.DIR_OUT,
  mode: gpio.PULLNONE
};
var led = gpio.open(config);

pin.on(gpio.INT_RISING, function(){
  led.write(1);
});
```

4.3 UART

UART模块可以使设备通过串口发送/接收数据。

4.3.1 介绍

UART.open

UART.open根据 config 配置打开 pin 引脚。如果调用成功，返回UARTPin对象。`UARTPin open(config)`

- `config` 为uart端口的配置对象，它包括四个属性：`uartNumber`、`baudRate` 和 `dataBits`、`stopBits`。
 - `uartNumber`：内置uart编号，值为0或1。该属性是可选属性，默认为1(0为调试端口，调试时用于部署脚本，此时，由于io29与io30同usb接口有复用关系，因此不能进行相关操作)。
 - 0: 接收引脚为IO29(即29)，发送引脚为IO30(即30)。
 - 1: 接收引脚为IO18(即18)，发送引脚为IO23(即23)。
 - `baudRate`：波特率，取值范围是[110, 6000000]。可选，默认为115200。
 - `dataBits`：数据位，这是衡量通信中实际数据位的参数，取值为7或8。如何设置取决于你想传送的信息。比如，标准的ASCII码是0~127(7位)。扩展的ASCII码是0~255(8位)。实际数据位取决于通信协议的选取。该属性是可选属性，默认值是8。如果输入数值不是7和8，按照数据位为默认值处理。
 - `stopBits`：停止位，用于表示单个包的最后一一位。取值为1或2位。由于数据是在传输线上定时的，并且每一个设备有其自己的时钟，很可能在通信中两台设备间出现了小小的不同步。因此停止位不仅仅是表示传输的结束，并且提供计算机校正时钟同步的机会。适用于停止位的位数越多，不同步的容忍程度越大，但是数据传输率同时也越慢。该属性是可选属性，默认为1。如果输入数值不是1或2，按照停止位为默认值处理。

例如：

- `config1={uartNumber:0, baudRate:9600, dataBits:8, stopBits:1}` 表示 uart接收引脚是IO29，发送引脚是IO30，波特率是9600，数据位是8位，停止位是1位。

UARTPin.write

UARTPin.write向UART发送引脚发送Buffer中的数据。

`void write(Buffer data)`

- `data` 发送的数据，类型为Buffer。Buffer模块可参考 [4.8 Buffer](#)

UARTPin.read

UARTPin.read从UART接受引脚接收size字符的数据，存放到Buffer中，如果 `size` 大于 `buf` 的长度，那么只读取 `buf` 长度的数据。如果提供了超时返回时间，那么达到超时时间后，返回已经读取到的字节数。它是一个同步读取接口。

`void read(Buffer buf, size, timeout)`

- `buf` 为存放读取的数据的Buffer，类型为Buffer。
- `size` 为要读取的字符的个数，类型为整数number (`number>=0`)。
- `timeout` 可选参数，超时返回时间，类型为整数number (`number>=0`)。如果timeout为零或者不提供该参数，那么只有接收size字符的数据才返回。

UARTPin.on

UARTPin.on为UART event设置回调函数。当前仅支持UART.DATA事件。新增回调函数会使旧的回调函数失效。

`uart.on(method, [number/end_char], [function])`

- `method` 目前只支持 UART.DATA event。
- `number/end_char` 可以是一个大于0的数字或者一个字符
 - 如果是大于零的数字n，那么当每次接收到n个字节时，回调函数被调用。
 - 如果是字符c，那么当每次接收到字符c时 或者当接收到最大长度为255个字符时，回调函数被调用。
- `function` 回调函数，event UART.DATA 的回调函数形如: `function(data){...}`, data是一个buffer，表示已经被uart接收到的数据。

当只提供method参数时，可以注销该event的回调函数。例如： `UARTPin.on(UART.DATA)` 可以注销UART.DATA event的回调函数。

4.3.2 样例

- UART的示例代码: 它展示了UART读入用户输入的3个字符然后输出。

```
var uart = require('uart');
var config={uartNumber:1};
var port=uart.open(config); //打开uart number 1 端口, 即 接收引脚为IO18 (即18), 发送引脚为IO23 (即23)。
buf = new Buffer(100); // 申请一个大小为100的buffer
port.read(buf,3); //从uart 接收引脚读取3个字节的内容到buf中
port.write(buf); // 将buf的内容发送到uart 发送引脚
```

- UART的示例代码: 它注册了一个回调函数，每读4个字节，输出读到的内容。当读到quit时，注销回调函数。

```
var uart = require('uart');
var config={uartNumber:1};
var port=uart.open(config); //打开uart number 1 端口, 即 接收引脚为IO18 (即18), 发送引脚为IO23 (即23)。
port.on(uart.DATA, 4, function(data) { // 设置回调函数, 每4个字节调用一次,
  print('receive from uart:', data);
  if(data.toString() == '71756974') //hex code of "quit"
  {
    port.on(uart.DATA) //注销回调函数
  }
});
```

- UART的示例代码: 它注册了一个回调函数，每读到'\r'字符(或者255个字符)，输出读到的内容。当读到'quit\r'时，注销回调函数。

```
var uart = require('uart');
var config={uartNumber:1};
var port=uart.open(config);
port.on(uart.DATA, '\r', function(data) {
  print('receive from uart:', data);
  if(data.toString() == '717569740d') //hex code of "quit\r"
  {
    port.on(uart.DATA)
  }
});
```

4.3.3 约束

串口监听数据的频率太高会导致事件队列溢出，当出现队列溢出时可做如下调整：

- 1.需降低串口发送数据的频率；
- 2.缩短数据的处理时间；
- 3.调大queue的数量。

做上面的几个调整时，需要遵循以下原则之一：

- 1.每个事件的处理时间要小于2个监听事件之间的间隔时间；
- 2.需要处理的事件数量要小于当前空闲queue的数量。

4.4 PWM

PWM模块用于控制支持脉冲宽度调制的引脚产生重复信号，即方波脉冲，其中信号在特定时间上下移动。例如，您可以通过PWM模块将pin IO3引脚每3ms脉冲1ms，它会无限期地执行此操作，直到再次进行设置。

通过设置占空比来控制PWM器件，即信号“开启”的时间百分比。例如，控制一个带有PWM的LED，并给它一个50%的占空比（例如1ms开，1ms关），它将以半亮度发光；如果给它一个10%的占空比（例如1ms开启,9ms关闭），它将以10%的亮度发光。

4.4.1 介绍

PWM.open

PWM.open根据 config 配置打开 pin 引脚并设置脉冲周期和占空比。如果调用成功，返回PWMPin对象。

```
PWMPin open(config)
```

- config 为PWM端口的配置对象，它包括三个属性： pin 、 period 和 duty 。
 - pin : 引脚号，类型为整数number (number>=0)，支持PWM的引脚包括: IO0(即0)、IO5(即5)、IO12(即12)、IO14(即14)、IO15(即15)、IO22(即22)和IO23(即23);
 - period : 脉冲周期，类型为整数number (number>=0)，单位是μs;
 - duty : 占空比，类型是浮点number，取值范围是[0,1]。例如:
- config1={pin:5, period:20000, duty:0} 表示打开IO5引脚，设置它的脉冲周期是20000μs (20ms)，占空比是0(20000μs开,0μs关)。如果是控制LED，它将不发光。
- config2={pin:12, period:10000, duty:0.5} 表示打开IO12引脚，设置它的脉冲周期是10000μs (10ms)，占空比是0.5(5000μs开,5000μs关)。如果是控制LED，它将半亮度发光。

PWMPin.set_period

PWMPin.set_period设置PWMPin对象的脉冲周期。

```
void set_period(number)
```

- number 为设置的脉冲周期，类型为整数number (number>=0)，单位是μs。
 - number>0 ，脉冲周期值将正常设置，不会对占空比设置产生影响。
 - number<0 ，函数会上报错误并返回。
 - number=0 ，脉冲周期被设置为0，此时无论设置占空比的值是多少，返回的占空比的值都将为0，之后若重新更改脉冲周期使 number>0 ，需要对占空比进行重新赋值，否则将默认输出占空比为0。

PWMPin.set_duty

PWMPin.set_duty设置PWMPin对象的占空比。

```
void set_duty(number)
```

- number 为设置的占空比，类型是浮点number，取值范围是[0,1]。
 - number>1 ，则设置占空比为1。
 - number<0 ，则设置占空比为0。
 - 占空比的值为0表示信号不“开启”，1表示信号在脉冲周期内完全“开启”。如果是控制LED，占空比的值为0表示不发光，1表示以100%的亮度发光。

PWMPin.get_duty

PWMPin.get_duty获取PWMPin对象的占空比，返回值是[0,1]的浮点数。

```
number get_duty()
```

4.4.2 样例

下面例子是PWM的示例代码。它需要1-3个LED灯或者1个三色(RGB)的LED灯，根据下面的接线方式连接LED和PWM引脚后，LED将逐渐变亮，然后逐渐变暗(感觉好像是人在呼吸，所以也叫呼吸灯)。

- 1-3个LED灯
 - 连接一个LED到IO_23并接地
 - 连接一个LED到IO_5并接地
 - 连接一个LED到IO_12并接地
- 3色LED灯
 - 连接R引脚到IO_23
 - 连接G引脚到IO_5
 - 连接B引脚到IO_12
 - 接地

```
var pwm = require('pwm');
var timer = require('timer');
var pwm_period=20000;
var config1={pin:23, period:pwm_period,duty:0};
var config2={pin:5, period:pwm_period,duty:0};
var config3={pin:12, period:pwm_period,duty:0};
var ports=new Array(3)
ports[0]=pwm.open(config1);
ports[1]=pwm.open(config2);
ports[2]=pwm.open(config3);

var PWM_STEP=0.01;
var steps=new Array(PWM_STEP, PWM_STEP, PWM_STEP);
var pwms=new Array(0.0, 0.0, 0.0);

while(1) {
  for(var i=0;i<3;i++) {
    ports[i].set_duty(pwms[i]);
    pwms[i]=pwms[i]+steps[i];
    if(pwms[i]>=1.0) {
      steps[i]=-PWM_STEP;
      pwms[i]=1.0;
    }
    if(pwms[i]<0.0) {
      steps[i]=PWM_STEP;
      pwms[i]=0.0;
    }
  }
  timer.setDelay(20);
}
```

4.5 SPI

SPI(Serial peripheral interface)即串行外围设备接口，是由Motorola首先在其MC68HCxx系列单片机上定义的，基于高速全双工总线的通讯协议。SPI通讯需要使用4条线：3条总线和1条片选。SPI遵循主从模式，3条总线分别是SCLK、MOSI和MISO，片选线为SS(低电平有效)。

- SS (Slave Select): 片选信号线，用于选中SPI从设备。当从设备上的SS引脚被置拉低时表明该从设备被主机选中。
- SCLK (Serial Clock): 时钟信号线，通讯数据同步用。时钟信号由通讯主机产生，它决定了SPI的通讯速率。
- MOSI (Master Output Slave Input): 主机(数据)输出/从设备(数据)输入引脚，即这条信号线上传输从主机到从机的数据。
- MISO (Master Input Slave Output): 主机(数据)输入/从设备(数据)输出引脚，即这条信号线上传输从从机到主机的数据。主从机通过两条信号线来传输数据。

SPI 模块支持SPI协议。

4.5.1 介绍

SPI.open根据 config 配置打开引脚并设置相关属性。如果调用成功，返回SPIPin对象。

```
SPIPin open(config)
```

- config 为SPI端口的配置对象，主设备包括5个配置属性：`mode`、`spi`、`bits`、`speed`、`transmit_mode`。从设备包括4个配置属性：`mode`、`spi`、`bits`、`transmit_mode`。
 - 主设备属性：
 - `mode`: 值为 SPI.MASTER。
 - `spi`: 内置SPI编号，值为0。
 - 0: MOSI引脚为23, MISO引脚为22, SCLK 引脚为18, SS 引脚为19。
 - `speed`: 传输速率，最高到31.25MHz，类型为浮点数number。
 - `bits`: 数据帧大小，值为4-16的整数。有一点需要注意的是，主机和从机数据帧需要一样。
 - `transmit_mode`: SPI有4种工作模式，值分别为 0、1、2和3。工作模式用于设置时钟极性(CPOL)和时钟相位(CPHA)。时钟极性指通讯设备处于空闲状态(SPI开始通讯前、SS线无效)时，SCLK的状态。时钟相位指数据的采样时刻位于SCLK的偶数边沿采样还是奇数边沿采样。
 - 0: CPOL=0(空闲时刻SCLK低电平), CPHA=0(第一个边沿采样(奇))
 - 1: CPOL=0(空闲时刻SCLK低电平), CPHA=1(第二个边沿采样(偶))
 - 2: CPOL=1(空闲时刻SCLK高电平), CPHA=0(第二个边沿采样(奇))
 - 3: CPOL=1(空闲时刻SCLK高电平), CPHA=1(第二个边沿采样(偶))
 - 有一点需要注意的是，主机和从机需要工作在相同的模式下才能正常通讯。
- 从设备属性: 含义与主设备相同。

例如，下面的配置表示：主设备传输速率是2M，数据帧大小为8 bit，传输模式为0。从设备数据帧大小为8 bit，传输模式为0。

```
slave_config={mode:spi.SLAVE, spi:0, bits:8, transmit_mode: 0};
master_config={mode:spi.MASTER, spi:0, speed:2000000, bits:8, transmit_mode: 0};
```

SPIPin.write

如果SPIPin是

- 主设备，SPIPin.write向SS为低电平的从设备发送 `buf` 中的数据，如果发送成功，返回发送的字节数。

```
number write(Buffer buf)
```

- buf: 发送的数据。

- 从设备，`SPIPin.write`向主设备发送数据 `buf` 中的数据，如果发送成功，返回发送的字节数。

```
number write(Buffer buf)
```

- buf: 发送的数据。

SPIPin.read

如果`SPIPin`是

- 主设备，`SPIPin.read`向SS为低电平的从设备接收长度是 `size` 的数据，如果接收成功，返回一个保存数据的 `Buffer` 对象，长度是 `size`。

```
Buffer read(size)
```

- size: 期望接收数据的长度，类型为整数 `number` (`number>=0`)。

- 从设备，`I2CPin.read`从主设备接收长度为 `size` 的数据，返回一个保存数据的 `Buffer` 对象，长度是 `size`。

```
Buffer read(size)
```

- size: 期望接收数据的长度，类型为整数 `number` (`number>=0`)。

4.5.2 样例

下面的例子展示了Slave 发送数据给Master。首先启动从设备，然后启动主设备。

Master

```
var spi = require('spi');
var hz=2000000;
var ma_config={mode:spi.MASTER, spi:0, speed:hz, bits:8, transmit_mode: 3};
var ma_port=spi.open(ma_config);
var size=2048;
buf=ma_port.read(size);
print(buf.toString());
```

Slave

```
var spi = require('spi');
var hz=2000000;
var sl_config={mode:spi.SLAVE, spi:0, bits:8, transmit_mode: 3};
var sl_port=spi.open(sl_config);
var size=2048;
buf=new Buffer(size);
for(var i=0;i<size;i++) {
    buf.writeUInt8(i+1,i);
}
sl_port.write(buf);
```

4.6 I2C

I2C 模块支持I2C协议，允许多个从设备与一个或多个主设备通信。每个I2C总线有两个信号：SDA和SCL，SDA是数据信号，SCL是时钟信号。

4.6.1 介绍

I2C.open

I2C.open根据 config 配置打开引脚并设置相关属性。如果调用成功，返回I2CPin对象。

```
I2CPin open(config)
```

- config为I2C端口的配置对象，主设备包括3个配置属性：`mode`、`i2c` 和 `speed`。从设备包括6个配置属性：`mode`、`i2c`、`speed`、`i2c_index`、`address`、`mask`。

- 主设备属性：

- `mode`: 值为 I2C.MASTER。
- `i2c`: 内置i2c编号，值为0或1。
 - 0: SDA引脚为IO23 (即23),SCL引脚为IO18 (即18)。
 - 1: SDA引脚为IO19 (即19),SCL引脚为IO22 (即22)。
- `speed`: 传输速率，标准模式(0-100kb/s)，快速模式(<400kb/s)。

- 从设备属性：

- `mode`: 值为 I2C.SLAVE。
- `i2c` 和 `speed` 含义与主设备属性相同。速率需要与主设备相同。
- `i2c_index`: 值为0或者1。0表示I2C0 device，1 表示I2C1 device。
- `address`: 从设备地址，仅支持7-bit address。
- `mask`: 地址位掩码，当地址位掩码某位置 1 (= 1) 时，该位即为“无关位”。无论其在地址的相应位中为0还是 1，从模块都会作出响应。例如，mask 为0110000，I2C 从器件将应答并认为地址 0010000 和 0100000 有效。

主设备和从设备的配置需要满足下面的要求：

- 主设备的从设备的 `speed` 相同。

例如，下面的配置表示：主设备的SDA引脚为IO23,SCL引脚为IO18，传输速率是100k; 从设备的SDA引脚为IO19,SCL引脚为 IO22。传输速率与主设备相同，地址是0xAA。

```
config_master={mode:i2c.MASTER,i2c:0,speed:100000};
config_slave={mode:i2c.SLAVE,i2c:1,speed:100000,i2c_index:0,address:0xAA,mask:0xFF};
```

I2CPin.write

如果I2CPin是

- 主设备，I2CPin.write向地址是 `slave_address` 的从设备发送 `buf` 中的数据，如果发送成功，返回发送的字节数。

```
number write(slave_address, reg_addr, value)
```

- `slave_address`: 从设备地址，单字节数值。
- `reg_addr`: 寄存器地址或控制字节，单字节数字或数组或缓存。
- `value`: 发送的数据，单字节数或数组或缓冲。

- 从设备，I2CPin.write向主设备发送数据 `buf` 中的数据，如果发送成功，返回发送的字节数。

```
number write(Buffer buf)
```

- buf: 发送的数据。

需要注意: 单次读写不能超过256 (含256) 字节. 如需发送超过256字节, 请拆分多次发送。

I2CPin.read

如果I2CPin是

- 主设备,I2CPin.read从地址是 slave_address , 如果接收成功, 返回接收的字节数。

```
number read(slave_address, reg_addr, value)
```

- slave_address: 从设备地址, 单字节数值。
- reg_addr: 寄存器地址或控制字节, 单字节数字或数组或缓存。
- value: 发送的数据, 无或数组或缓冲。

*当value不存在时, 函数返回值是读取到的寄存器的值。

- 从设备,I2CPin.read从主设备接收长度为size的数据, 返回一个保存数据的 Buffer 对象, 长度是 size。

```
Buffer read(size)
```

- size: 期望接收数据的长度, 取值范围为[1, 256]的整数。

需要注意: 单次读写不能超过256 (含256) 字节. 如需发送超过256字节, 请拆分多次发送; 并且如果接收到的数据量小于设置的size, 那么对接口的调用将阻塞, 直到收到足够的数据。

4.6.2 样例

下面例子为I2C的示例代码. 它需要2个realtek设备, 一个主设备, 一个从设备. 根据下面的接线方式连接两个设备后, 主设备向从设备发送数据, 从设备接收并输出.

连接方式:

- 主设备 SDA引脚(IO_23) 连接 从设备SDA引脚 (IO_23), 连线之间接10k上拉电阻.
- 主设备 SCL引脚(IO_18) 连接 从设备SCL引脚 (IO_18), 连线之间接10k上拉电阻.
- 两个设备地线相连(共地).
- 连线后, 先打开从设备, 再打开主设备.

主设备代码

```

var i2c = require('i2c');
var tim = require("timer");
var ma_config={mode:i2c.MASTER, i2c:0, speed:100000};
var i2c_port=i2c.open(ma_config);
i2c_port.write(0x40,0x00,0x00);//reset

var oldmode = i2c_port.read(0x40, 0x00);
var newmode = (oldmode&0x7F)|0x10; // sleep
i2c_port.write(0x40,0x00,newmode);//go to sleep
i2c_port.write(0x40,0xFE,132);//set the prescaler
i2c_port.write(0x40,0x00,oldmode);
tim.setDelay(5);
i2c_port.write(0x40,0x00,oldmode|0xa1);
tim.setDelay(5);
i2c_port.write(0x40,0xFA,0x00);
i2c_port.write(0x40,0xFB,0x00);
i2c_port.write(0x40,0xFC,0x08);
i2c_port.write(0x40,0xFD,0x02);//180 degree
var buf = new Buffer([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,20,21,22,23,24,25,26,27,28,29]);

tim.setInterval(function() {
    //i2c_port.write(0x40, 0x06,buf);
    i2c_port.read(0x40, 0x06,buf);
}, 50);

print("js execute done!");

```

从设备代码

```

var i2c = require('i2c');
var hz=100000;
var sl_config={mode:i2c.SLAVE, i2c:0, speed:hz, i2c_index:0, address:170, mask:255};
var sl_port=i2c.open(sl_config);
var size=100;
buf2=sl_port.read(size);
print(buf2.toString());

```

4.7 HiLink

Hilink是华为开发的智能家居开放互联平台， Hilink模块提供设备侧的Hilink接口，使设备具备快速连接到Hilink平台的能力。

4.7.1 介绍

设备侧与Hilink互通时，主要定义了两个动作 `GET`，`PUT` 和 `DEVSTAT`。`PUT` 对应于网络命令的下发，是一条下行路径；`GET` 对应于设备状态的上报，是一条上行路径，`DEVSTAT` 指示设备是否联网。

HiLink API包括，

```
hilink.on() , hilink.initDevInfo() , hilink.getDevInfo() , hilink.initParam() , hilink.getNetInfo() , hilink.response() , hilink.upload() , hilink.disconnectWiFi()
```

八个接口。

- `hilink.on(option, action function)`，该API用于设置Hilink的两种通信动作。

`option` 为 `hilink.GET` 或 `hilink.PUT` 分别表示对应 `hilink` 的 `get` 或 `put` 动作。

`action function` 为对应于 `get` 和 `put` 的处理函数。

当对应 `option` 为 `hilink.GET` 时，`action function` 的签名为 `function(svc_id, instr)`，返回值为字符串；当对应 `option` 为 `hilink.PUT` 时，`action function` 的签名为 `function(svc_id, payload)`，无返回值。

当对应 `option` 为 `hilink.DEVSTAT` 时，`action function` 的签名为 `function(status)`，无返回值。

- `hilink.initDevInfo()`，该接口用于初始化设备的信息。

以下Hilink设备信息的一个示例：

```
var devinfo={'sn':'','prodId':'112Q','model':'baseboard heater','dev_t':'06A','manu':'0c9','mac':'','hiv':'1.0.0','fwv':'1.0.0','hwv':'1.0.0','swv':'1.0.0','prot_t':1};
```

```
typedef struct {
    char* sn;           /*<设备唯一标识, 比如sn号, 长度范围 [0, 40]*/
    char* prodId;       /*<设备HiLink认证号, 长度范围 [0, 5]*/
    char* model;        /*<设备型号, 长度范围 [0, 32]*/
    char* dev_t;         /*<设备类型, 长度范围 [0, 4]*/
    char* manu;          /*<设备制造商, 长度范围 [0, 4]*/
    char* mac;           /*<设备MAC地址, 固定32字节*/
    char* hiv;           /*<设备HiLink协议版本, 长度范围 [0, 32]*/
    char* fwv;           /*<设备固件版本, 长度范围 [0, 64]*/
    char* hwv;           /*<设备硬件版本, 长度范围 [0, 64]*/
    char* swv;           /*<设备软件版本, 长度范围 [0, 64]*/
    int prot_t;          /*<设备协议类型, 取值范围 [1, 3]*/
} dev_info_t;
```

- `hilink.getDevInfo()`，该接口用于获取设备信息。
- `hilink.initParam()`，该接口用于初始化`dev_info`、`svc_info`、`BI`、`AC`这四个参数。
- `hilink.getNetInfo()`，该接口用于获取网络信息，返回一个json字符串，包含属性有`intensity`，`RSSI`，`SSID`，`BSSID`，`IP`。如果断开网络连接，恢复出厂设置，则需要重启后调用此接口才会显示正确信息。
- `hilink.response()`，`hilink`与引擎之间是用队列通信，该接口用于将引擎的返回值传给给Hilink SDK相关的接口。
- `hilink.upload()`，用于主动上报SVC字段状态给云服务器。
- `hilink.disconnectWiFi()`，该API用于断开wifi连接。

4.7.2 样例

下面例子为某Hilink智能设备的示例代码，主要演示了定义用户逻辑的功能。

```

var hilink=require('hilink');
var uart = require('uart');
var config={uartNumber:1, baudRate:9600, dataBits:8, stopBits:1};
var port=uart.open(config);
var devinfo={'sn':'','prodId':'1000','model':'iot','dev_t':'000','manu':'000','mac':'','hiv':'1.0.0','fwv':'1.0.0','hwv':'1.0.0','swv':'1.0.0','prot_t':1};
var svcinfo="switch,switch;"+
    "netInfo,netInfo;";
var BI= "00000000000000000000000000000000"+
    "3D294E73EE637A1CBC80DB19055F227D"+
    "5C6D60DB4467A1E223FCABA94CDA4A9D"+
    "A492700F7C703E2D372F607810F6DC4C"+
    "3BF2FE6C5D76E32190326D48E2DB4B59"+
    "B9883D73307BEEC83D566661AACB4CDB"+
    "3E8A592ECDD72EE0F1CD06E87B5C14E0"+
    "950C969D5C8C067923CBFDACAA4BD1A1"+
    "CC76931244C517C34FCF4E213B508944"+
    "00000000000000000000000000000000"+
    "89C9EBC41429A477267704EF081B8C8"+
    "23EC28165DC4B9D60B5F6A784D8067F2"+
    "00000000000000000000000000000000"+
    "71DE4C105AF60925FF6AF031E6EF1AF4"+
    "3667A64D94CA75A09293B46892B97D44"+
    "7AD9C92F69A708229E37B8587AE52F7F";
var AC= "00000000000000000000000000000000"+
    "7837F9F02FB5B3B99C690A999E039455"+
    "92DE3268C94247F37B63764A1B007631";
hilink.initParam(JSON.stringify(devinfo), svcinfo, BI, AC);

port.on(uart.DATA, 9, function(data) {
    var val;
    ...
    if(type==0x01 && cmd== 0xF0){
        hilink.disconnectWifi();
        ...
    }
    else if(type == 0x03){
        py.switch=val;
        hilink.upload("switch",JSON.stringify(py));
    }
});
hilink.on(hilink.GET,function(svc_id,instr){
    var ret = "";
    var pass = {};
    ...
    else if(svc_id == "netInfo"){
        hilink.response(hilink.getNetInfo());
        return(hilink.getNetInfo());
    }
    ret = JSON.stringify(pass);
    print(ret);
    hilink.response(ret);
    return ret;
});
hilink.on(hilink.PUT,function(svc_id,payload){
    var pass=JSON.parse(payload);
    var perc=0;
    if(svc_id == "targetLevel")
    ...
    hilink.response(0);
    return 0;
});

```

4.8 Buffer

该Buffer模块与NodeJS中提供的Buffer相似，功能有缩减。其中如果参数为小数，则统一向下取整。

4.8.1 介绍

new Buffer(size or string or array)

申请一个新的Buffer对象。

- size: 分配一个大小为 size 字节的新建的 Buffer，Buffer 所有字节初始化为零。要求满足性质 `0 <= size <= 2147483647(0x7fffffff)`。
- string: 创建一个包含给定字符串 string 的 Buffer，Buffer长度为string的长度。
- number array: 将传入的 number array 数据拷贝到一个新建的 Buffer，如果array中的元素不是number，会当做0处理，若 number值不在0-255范围内，会强转为uint8类型再存储。Buffer长度为array的长度。

buf.length

buf的长度，它是不可修改的数值属性。

buf.readUInt8(offset)

从指定偏移量的buf读取无符号8位整数。

- offset 开始读取之前要跳过的字节数。要求满足性质 `0 <= offset <= buf.length - 1`。
- Returns: 从指定偏移量的buf读取无符号8位整数。

buf.writeUInt8(value, offset)

将值写入指定偏移量的buf。

- value 要写入buf的数字。要求满足性质 `0 <= value <= 255(0xff)`。
- offset 开始写入之前要跳过的字节数。要求满足性质 `0 <= offset <= buf.length - 1`。
- Returns: offset加上写入的字节数。

buf.toString(encoding)

按照指定字符编码将buf解码成字符串。

- encoding 指定字符的编码，支持 `utf8, hex, ascii` 三种字符编码，默认值为 `utf8`。
- Returns 按照指定字符编码将buf解码成字符串。

buf.copy(target[, targetStart[, sourceStart[, sourceEnd]]])

拷贝 buf 中某个区域的数据到 target 中的某个区域。要求满足 `sourceEnd - sourceStart <= target.length - targetStart`。

- target 要拷贝进的 Buffer。
- targetStart target 中开始写入的偏移量。要求满足 `0 <= targetStart <= target.length - 1`，默认为 0。
- sourceStart buf 中开始拷贝的偏移量。要求满足 `0 <= sourceStart <= buf.length - 1`，默认为 0。
- sourceEnd buf 中结束拷贝的偏移量（不包含）。要求满足 `sourceStart < sourceEnd <= buf.length`，默认为 `buf.length`。
- Returns: 拷贝的字节数。

buf.fill(value[, offset[, end [, encoding]]])

用指定的 value 填充 buf指定长度。如果没有指定 offset 与 end，则填充整个 buf。要求满足 `end - offset <= buf.length`

- value 是用来填充 buf 的值, value的类型可以是 string , Buffer , integer 或 Array 。其中如果 value 是一个数值的话, 则会被转换成介于0到255的整数值; value 如果是 Array 的话, 只允许数组的元素全是数值, 同样其中的每个数值都会被转换成介于0到255的整数值。
- offset 开始填充 buf 的偏移量。默认为 0。要求满足 `0 <= offset <= buf.length - 1` 。
- end 结束填充 buf 的偏移量 (不包含) 。默认为 `buf.length` 。要求满足 `offset < end <= buf.length` 。
- encoding 如果 value 是字符串, 则指定 value 的字符编码。默认和目前只支持utf8编码格式。使用 `buf.UTF8` 表示。
- Returns: buf 的引用。

4.8.2 样例

```

buf=new Buffer("hello");           //创建一个包含给定字符串 hello 的 buf
print(buf);                      //依据utf8字符编码打印buf

print(buf.toString());            //依据utf8字符编码打印buf
print(buf.toString("utf8"));      //依据utf8字符编码打印buf
print(buf.toString("hex"));       //依据hex字符编码打印buf
print(buf.toString("ascii"));     //依据ascii字符编码打印buf

buf=new Buffer(10);              //创建一个长度为10的 buf
for (var i=0;i<10;i++) {         //buf的第i个字节写入值i
  buf.writeUInt8(i,i)
}
for (var i=0;i<10;i++) {         //读取buf的第i个字节并打印
  print(buf.readUInt8(i))
}

buf_target=new Buffer(10);        //创建一个长度为10的buf_target
for (var i=0;i<10;i++) {         //将buf_target中的值全部置为0
  buf_target.writeUInt8(0,i)
}
buf.copy(buf_target,3,3,10);      //拷贝buf第3至10字节偏移量的数据到buf_target第3字节偏移量开始
for (var i=0;i<10;i++) {         //读取buf_target的第i个字节并打印
  print(buf_target.readUInt8(i))
}

var val_str='HELLO';
var val_int=0x0b;
var val_arr=[0x01,0x02,0x03,0x04,0x05];

var buf_num=new Buffer([00,00,00,00,00]);
var buf_dest=new Buffer("helloworld");
var buf_src=new Buffer(val_str);

print(buf_num.toString('hex'));    //Buffer原始值: <0 0 0 0 0>

print(buf_num.fill(val_int).toString("hex")); //向Buffer中填充数值0x0b,
                                              //填充后结果:<0b 0b 0b 0b 0b>
print(buf_num.fill(val_arr).toString("hex")); //向Buffer中填充数组,
                                              //填充后结果:<01 02 03 04 05>
print(buf_dest);                  //Buffer原始值: helloworld
print(buf_dest.fill(buf_src, 0, 5)); //向Buffer中填充新Buffer中内容,
                                              //填充后结果:HELLOworld
print(buf_dest.fill('w'));        //向Buffer中填充字符'w',
                                              //填充后结果 :wwwwwww

```

4.9 RTC

RTC (Real-Time Clock) 模块可以让我们获取系统当前时间的一些信息。使用RTC模块，需要使用require语句获得该模块句柄：`var rtc = require('rtc');`。

4.9.1 介绍

RTC模块目前提供了如下9个API接口

rtc.getTime

`time_t rtc.getTime()` 用于获取当前日期的秒数。这个数是从1970.1.1 00:00:00开始计算的。

rtc.getYear

`time_t rtc.getYear()` 返回日期中的年份。

rtc.getMonth

`time_t rtc.getMonth()` 返回日期中的月份。

rtc.getMday

`time_t rtc.getMday()` 返回日期月份中的第几天。

rtc.getWday

`time_t rtc.getWday()` 返回日期星期中的第几天（其中星期日为第一天，以0表示，依次类推）。

rtc.strToTime

`time_t rtc.strToTime(string time_string)` 用于将输入的 UTC 时间字符串转变为 unix 时间戳。这里输入的时间字符串格式固定为"1970-01-01 00:00:00"，输入非法字符串或非法日期都会上报错误。时间字符串取值范围为 1970-01-01 00:00:00 到 2036-02-07 06:28:15。

rtc.timeToStr

`string rtc.timeToStr(time_t t)` 用于将输入的 unix 时间戳转变为 UTC 时间字符串，时间戳取值范围为0-2085978495。

rtc.setTimeShift

`rtc.setTimeShift(ANTJS_RTC_TIME tz)` 接受一个参数用于设置时间偏置，该参数取值范围为-12到12之间，当设备RTC寄存器保存的并非是UTC时间而是localtime时。这种情况下设置 `rtc.setTimeShift(-8)`（以北京时间为例）来进行校正。

rtc.setTime

`rtc.setTime(time_t t)` 用于以unix时间戳来设置日期，参数设置范围为0-2085978495，对应UTC时间为1970-01-01 00:00:00 到 2036-02-07 06:28:15。

4.9.2 样例

下面例子是RTC的示例代码：

```
var rtc = require('rtc');
var unixtime = rtc.getTime();
var year = rtc.getYear();
var month = rtc.getMonth();
var day = rtc.getMday();
var week = rtc.getWday();
var m = rtc.strToTime("2018-08-17 12:00:00");
var date = rtc.timeToStr(1533744000);
rtc.setTimeShift(-8);
rtc.setTime(1533744000);
```

4.10 System

4.10.1 介绍

System模块允许用户通过调用API来查询引擎和操作系统的堆内存使用状况，以及对board进行复位和恢复出厂设置等操作。使用System模块首先要先获得该模块句柄：`var system = require ('system');`。

4.10.2 查询堆内存

查询引擎堆内存使用情况：`system.engineHeap (type)`

查询操作系统堆内存使用情况：`system.osHeap (type)`

引擎的可分配内存可以是一个静态编译的数组，也可以是通过调用操作系统的接口来分配的系统堆内存，通过一个宏来控制，默认使用静态数组。

4.10.2.1 type的枚举定义以及返回值

- `system.TOTAL`: 总内存大小，`engineHeap`和`osHeap`均支持；
- `systemUSED`: 已使用内存大小，`engineHeap`和`osHeap`均支持；
- `system.MAX_BLOCK`: 内存最大块的大小，`engineHeap`和`osHeap`均支持；
- `system.ALLOC_COUNT`: 内存分配次数，`engineHeap`和`osHeap`均支持；
- `system.FREE_COUNT`: 内存回收次数，`engineHeap`和`osHeap`均支持；
- `system.PEAK_ALLOC`: 内存使用峰值，仅`engineHeap`支持；
- `system.WASTE`: 内存浪费大小，仅`engineHeap`支持；
- `system.PEAK_WASTE`: 内存浪费峰值，仅`engineHeap`支持；
- `system.GC`: 内存GC情况，仅`engineHeap`支持；
- `system.FREE`: 已回收的内存大小，仅`osHeap`支持；

在以上类型中，除GC外，返回值均为number类型；GC返回值类型为object，含有两个属性，一个是size，代表总共GC的内存大小，一个是count，代表GC次数。

4.10.2.2 内存查询特别说明

使用引擎内存查询接口时应当注意，除了debug版本外，其他发布的版本只支持查询TOTAL和USED两种类型，如果查询，则只会返回`undefined`。

内存最大块：OS内存和引擎内存都是以链表的方式来组织的，内存是分为多个块，通过指针来连接，所谓内存最大块是指内存最大的那个块，表示一次申请内存不能超过的最大值。

浪费内存大小：由于在申请引擎的堆内存时，是按8字节对齐的，所以多余申请的字节即为浪费的内存。

4.10.3 重启和恢复出厂设置

重启：`void system.reset ();`

恢复出厂设置：`void system.restore ();` 主要功能是擦除flash中UserData段的数据。

4.10.4 样例

下面是System模块的使用样例，包括了内存查询，与恢复出厂设置。

```
/* 内存查询 */
var system = require ('system');
print ("total heap size of engine:" + system.engineHeap (system.TOTAL));
var gc = system.engineHeap (system.GC);
print ("engine gc times:" + gc.count + ",heap size freed by gc:" + gc.size);
print ("os available heap size:" + (system.osHeap (system.TOTAL) - system.osHeap (systemUSED)));

/* 恢复出厂设置 */
var hilink = require ('hilink');
hilink.disconnectWiFi () /* 断开wifi连接 */
system.restore () /* 恢复出厂设置 */
system.reset () /* 重启 */
```

4.11 OTA

4.11.1 介绍

OTA模块用于JS脚本和MCU的在线升级，它分别提供了面向hilink和JS用户脚本的接口。JS脚本的升级过程不需要用户参与，MCU的升级需要用户调用提供的接口共同完成升级。

4.11.2 MCU升级

在MCU的升级过程中，需要用户在JS脚本中注册相应的hilink回调函数，注册时第一个参数表明升级过程中的事件。而回调函数的主要作用是以函数参数的形式传递相应的MCU数据以供用户处理。

4.11.2.1 事件类型

下面四个事件只用来升级MCU。升级MCU需要全部注册下面四个事件的回调函数；升级JS不需要注册下面事件的回调函数。

- hilink.FILELIST_COMPONENT: 传递MCU的属性，如version, hash value;
- hilink.OTA_START: 传递MCU的大小;
- hilink.OTA_CHUNK: 可能多次调用，分块传递MCU的数据;
- hilink.OTA_END: 传递MCU的升级的状态，hash value来结束此次升级;

4.11.2.2 回调函数参数以及返回值

- FILELIST_COMPONENT 对应 Object/number function (name, data)

data: 对象类型，包含version, hash, value, user_hash, user_value这几个属性，均是string类型，分别表示MCU版本，hash算法，hash值，后两个是可选的。

返回值是一个对象，对象包含两个属性，分别是update(number), fileName(string)，代表升级与否(1代表升级)和要下载的文件名称(若没有则使用默认名称)。

- OTA_START 对应 number function (name, len)

len:number类型，表示MCU长度。返回值是一个对象，有一个属性error表示是否出错，0代表无错，-1表示有错。

- OTA_CHUNK 对应 number function (name, data)

data: Buffer类型，表示MCU升级数据。返回值是一个对象，有一个属性error表示是否出错，0代表无错，-1表示有错。

- OTA_END 对应 number function (name, state, hash)

state: number类型，表示升级状态(0表示正常)。返回值是一个对象，有一个属性error表示是否出错，0代表无错，-1表示有错。

以上参数中的name(string)均代表升级的部件名称。在现在只有一个MCU的情况下，默认是mcu_ota_all.bin。

4.11.3 hislip协议

用户JS脚本可以使用自定协议进行MCU升级，或者可以直接调用hislip模块接口进行升级。

hislip模块主要有两个接口，send和getAck，分别用于通过串口发送数据和接收应答来进行MCU升级。

接口说明：

- send (cmd, data, port): cmd是命令，data是与命令相关数据，port是uart串口对象。
- getAck (port): 返回值是number类型，0代表确认升级或者没有错误，1代表不升级或者发生错误。

发送时主要有四种命令：

- hislip.CMD_QUPG: 请求升级, 数据是MCU版本
- hislip.CMD_SUPG: 确认升级, 数据是MCU长度
- hislip.CMD_TRANSING: 数据传输, 数据是MCU升级块
- hislip.CMD_EUPG: 结束升级, 数据是hash值

以上数据除长度是number类型外, 其余可以是string或者Buffer。

4.11.4 版本信息查询

在hilink模块中, 有两个接口用于用户对当前版本信息进行查询, 分别是getVersion和getJSVersion, 前者获取整体版本号, 后者获取JS脚本版本号, 均是string类型.

4.11.5 例子

```
var hilink = require ('hilink');
var hislip = require ('hislip');
var config={uartNumber:1, baudRate:115200, dataBits:8, stopBits:1};
var port=uart.open(config);

print (hilink.getVersion ()); //整体版本号
print (hilink.getJSVersion ()); //JS脚本版本号
hilink.on (hilink.FILELIST_COMPONENT, function (name, data)
{
    hislip.send (hislip.CMD_QUPG, data.version, port);
    var update = hislip.getAck (port);
    if (update == 0) //need update
        return {"update":1,"fileName":"mcu_ota_all.bin"};
    else
        return {"update":0};
})

hilink.on (hilink.OTA_START, function (name, len)
{
    hislip.send (hislip.CMD_SUPG, len, port);
    var error = hislip.getAck (port);
    return {"error":error};
})

hilink.on (hilink.OTA_CHUNK, function (name, buf)
{
    hislip.send (hislip.CMD_TRANSING, buf, port);
    var error = hislip.getAck (port);
    return {"error":error};
})

hilink.on (hilink.OTA_END, function (name, state, hash)
{
    if (state != 0) return -1;
    hislip.send (hislip.CMD_EUPG, hash, port);
    var error = hislip.getAck (port);
    return {"error":error};
})
```

4.11.6 特殊说明

filelist.json样例

```
{
    "version": "1.0",           //overall version
    "file":      //component name
    {
        "version": "js_1001"     //component name
        "hash": "sha256",
        "value": "123456789...",
        "user_hash": "MD5",
        "user_value": "22345346..."
    },
    "mcu_ota_all.bin":          //component name
    {
        "version": "rtlambaz-fw0100" //component version
        "hash": "sha256",
        "value": "123456789...",
        "user_hash": "MD5",
        "user_value": "22345346..."
    }
}
```

HOTA服务器上filelist.json文件只支持hash/value, user_hash/user_value是客户提供的，用于第三方服务器。

目前允许升级的最大JS脚本为6k。

JS升级不需要注册以上回调函数，MCU升级需要注册全部4个回调函数，如果只注册部分，则回调函数无法使用。同时只要filelist.json里有MCU，则需要注册回调函数。

JS,MCU如果都需要升级，则顺序为先JS,后MCU。如果JS升级成功，会保存JS版本号到FLASH,可以通过getJSVersion读到；之后会用MCU升级会用新的JS脚本进行升级。

只有升级整体成功，即要升级的部件全部升级成功，才会将整体版本号保存，否则通过getVersion读到版本号跟之前一致，不会变化。

Debug版本暂不支持OTA升级。Release和Release_Parser中的pl-cnv版本仅支持JS文件(或者snapshot文件)升级，Release和Release_Parser中的full版本支持JS文件(或者snapshot文件)和MCU升级。

- 5 调试
 - 5.1 使用串口调试
 - 安装
 - 使用
 - 指令
 - 5.2 使用Socket调试

5 调试

本章介绍调试功能。

该调试功能仅限用户在开发调试阶段使用，依赖串口与调试端口连接，产品发布后由于在现网中调试端口关闭，所以功能无法使用，不涉及网上安全问题。

注意：不支持对Bytecode文件的调试。当前本版不支持对含有回调函数的JS脚本的调试。

5.1 使用串口调试

安装

当前版本支持Windows和Linux操作系统，Python版本 2.7 (不支持Python 3)

```
# 仅Windows平台上需要安装pywin32
pip install pywin32

# Windows和Linux平台均需安装pyserial
pip install pyserial
```

使用

uart 选项

该选项必须指定，其值为板子连接的串口号，对大小写敏感，Windows平台下形如 com4，Linux平台下形如 /dev/ttyS0

```
$ python maple-client-serial.py --uart=<port number>
>>>Reboot your board first!<<<
Waiting for UART connection...
```

help 选项

可选，可以列出客户端支持的所有选项，查看每个选项对应的格式和描述

```
$ python maple-client-serial.py --help
usage: maple-client-serial.py [-h] [--uart UART] [--upload UPLOAD] [-v]
                             [--non-interactive] [--color]
                             [--display DISPLAY] [--exception {0,1}]

JerryScript debugger client

optional arguments:
  -h, --help            show this help message and exit
  --uart UART           specify a COM port
  --upload UPLOAD       specify a filename and a COM port via --uart=port
  -v, --verbose         increase verbosity (default: False)
  --non-interactive    disable stop when newline is pressed (default: False)
  --color              enable color highlighting on source commands (default:
                      False)
  --display DISPLAY     set display range
  --exception {0,1}      set exception config, usage 1: [Enable] or 0: [Disable]
```

verbose 选项

可选，可以打印出调试过程中客户端的debug信息， 默认为False

```
$ python maple-client-serial.py --uart=<串口名字 - 例如COM9> --verbose
DEBUG: Debug logging mode: ON
Waiting for UART connection...
```

non-interactive 选项

可选，控制程序运行过程中是否可被打断， 默认为False， 即可以打断

```
$ python maple-client-serial.py --uart=<串口名字 - 例如COM9> --non-interactive
```

color 选项

可选，控制客户端输出信息的颜色显示， 默认为False

注意： color 选项在 cmd 和 PowerShell 中会导致显示异常

```
$ python maple-client-serial.py --uart=<串口名字 - 例如COM9> --color
```

display 选项

可选，控制调试过程中是否显示源码， 其值为显示当前行前后源码的行数

```
$ python maple-client-serial.py --uart=<串口名字 - 例如COM9> --display=10
```

exception 选项

可选，配置是否显示Exception信息， 其值为 0 或 1， 默认为 0

```
$ python maple-client-serial.py --uart=<串口名字 - 例如COM9> --exception=1
```

upload 选项

可选，其值为部署到板子里的文件， 部署完成后，程序退出。

```
$ python maple-client-serial.py --uart=<串口名字 - 例如COM9> --upload=./hello.js
-----
Upload JS with content:
print("Hello, world!");
-----
Finished writing source code to COM port!

Finished restart device through COM port!
```

指令

help 指令

打印帮助信息，无参时打印所有支持的命令，后面可跟参数为调试器支持的命令，打印对应命令的帮助信息

```
(maplejs-debugger) help

Documented commands (type help <topic>):
=====
abort      bt       display   exception  list      next      source
b          c       dump        f      memstats  quit      src
backtrace  continue e       finish     ms       s       step
break      delete   eval      help      n       scroll    throw
```

eval 指令

执行JavaScript源代码，必须跟参数，为JavaScript表达式，返回表达式的计算结果

```
(maplejs-debugger) eval count
1
(maplejs-debugger) eval count+1
2
```

e 指令

同 eval

throw 指令

抛出异常

```
(maplejs-debugger) throw new Error("Throwing an exception")
Stopped at trycatch.js:7
Source: trycatch.js
1  try
2  {
3    print("In try block");
4  }
5 catch(err)
6  {
7 >  print("In catch block");
8  }
9 finally {
10   print("In finally");
11 }
```

abort 指令

抛出异常

```
(maplejs-debugger) abort new Error("Fatal error")
err: Script Error: Error: Fatal error
```

memstats 指令

显示内存使用信息

```
(maplejs-debugger) memstats
Allocated bytes: 0
Byte code bytes: 0
String bytes: 0
Object bytes: 0
Property bytes: 0
```

ms 指令

同 `memstats`

display 指令

设置最多一次显示源码的行数

```
(maplejs-debugger) display 20
```

scroll 指令

向上或向下滚动源码，`w+回车` 向上，`s+回车` 向下，`q+回车` 退出，可与 `display` 配合使用

```
(maplejs-debugger) scroll
w
s
q
```

dump 指令

打印所有调试数据

```
(maplejs-debugger) dump
{325: Function(byte_code_cp:0x145, source_name:'string-load', name:'', line:1, column:1 { string-load:1 })}
```

backtrace 指令

回溯堆栈，列出当前函数调用关系

```
(maplejs-debugger) backtrace
```

bt 指令

同 `backtrace`

step 指令

执行下一条指令，遇到函数会进入函数内部执行，不接受参数

```
(maplejs-debugger) step
```

s 指令

同 `step`

next 指令

可跟一个参数，默认为1，执行下n条指令，不会进入函数体

注意：该命令的参数必须为大于0的正整数，且当其大小大于当前函数的最大行数时，该参数并不会实际作为next命令的执行次数，取而代之的是当前函数的最大行数

```
(maplejs-debugger) next
```

```
(maplejs-debugger) next 3
```

n 指令

同 `next`

finish 指令

继续运行直到当前函数返回，不接受参数

```
(maplejs-debugger) finish
```

f 指令

同 `finish`

continue 指令

继续执行程序，直到遇到下一个断点或者执行结束

```
(maplejs-debugger) continue
```

c 指令

同 `continue`

break 指令

此指令用来在文件中设置断点，目前支持两种方式：

- 通过函数名设置断点：执行到对应函数处暂停执行
- 通过文件名+行号设置断点：执行到对应行时暂停执行

断点分为Active和Pending两种，Pending类型的断点一般是在空行、不存在的函数等处，不会影响文件执行

```
(maplejs-debugger) break main      /* 根据函数名设置断点 */
(maplejs-debugger) break ping.js:10  /* 根据文件名和行号设置断点 */
```

b 指令

同 `break`

list 指令

列出当前所有断点，不接受参数

```
(maplejs-debugger) list
```

delete 指令

删除指定断点，参数可为索引、`all`、`active`、`pending`，索引为可通过list指令查看

```
(maplejs-debugger) delete 1
```

source 指令

显示当前行前后n行的源码

```
(maplejs-debugger) source 5
```

src 指令

同 `source`

exception 指令

配置异常处理程序模块，可以接收的参数为0、1，分别为关闭和开启

```
(maplejs-debugger) exception 0
```

quit 指令

退出串口调试器

注意：当程序运行结束时，必须通过Ctrl+C组合键退出调试器

```
(maplejs-debugger) quit
```

5.2 使用Socket调试

暂不具备socket调试的能力，当前版本仅提供串口调试

- [6 常见问题](#)

6 常见问题

本章介绍常见问题。

1. MapleJS引擎目前分配的堆内存为10KB，由于引擎本身消耗一定内存，所以JS应用实际可使用内存小于该值，当前测试可使用内存估计为8KB左右。

2. 回调与循环的陷阱

引擎采用事件队列运行机制（同Node.js）：先执行当前运行的脚本，当前脚本执行完后依次从事件队列中取下一个事件执行。引擎的timer.setInterval、timer.setDelay、gpio.on、uart.on都是异步接口，当循环中嵌套这些异步接口的回调函数时会遇到一些“问题”。看下面的一个例子：

```
var time = require('timer');
var tid = new Array(2)
var j = 1;

function startTimer(){
    print("this is " + j + " cycle!");
    j++;
}

for(var i=0; i<2; i++){
    print("start timer " + i);
    tid[i] = time.setInterval(startTimer, 100);
}

time.setDelay(120);

for(var i=0; i<2; i++){
    time.stopTimer(tid[i]);
    print("stop timer " + i);
}
```

下面是实际运行起来的输出结果：

```
start timer 0
start timer 1
stop timer 0
stop timer 1
this is 1 cycle!
this is 2 cycle!
```

从运行结果我们发现startTimer在stopTimer之后执行，这是因为引擎采用事件队列运行机制，即引擎首先执行上面的JS脚本，执行完第一个for循环后，引擎会sleep 120 ms，第一个for循环执行完100ms后，tid[0]和tid[1]会向事件队列中发送2个startTimer事件，这两个事件需要等到上面的JS脚本执行完之后才可以执行，所以引擎在120 ms之后会继续运行第二个for循环。当执行完上面的脚本后，引擎从事件队列中取下一个事件执行。因为现在事件队列中已经有两个startTimer事件，所以它们会依次执行。最终得到上面的运行结果。

1. 死循环问题

如果JS脚本有死循环，会导致当前JS脚本一直在引擎上运行，无法运行下一个事件。所以不建议使用死循环以及运行时间比较长的操作。建议使用异步的方式来代替死循环。

如果JS脚本有死循环，而且存在停止执行当前脚本的需求，可通过IDE工具或命令行工具中提供的 `stopEngine` / `stopScript` 功能来实现（仅适用于循环结构）。

注意：向设备发送的stop指令并不会立即得到执行，而是执行到循环语句时才能停止。

- [7 限制与约束](#)
 - [7.1 常见的MapleJS引擎的限制与约束](#)
 - 函数调用参数个数限制
 - 函数定义参数个数限制
 - 字符串长度限制
 - 文件源码最大字节数限制
 - 引用计数限制
 - 栈使用限制
 - [Timer使用限制](#)

7 限制与约束

本章介绍MapleJS引擎的限制与约束

7.1 常见的**MapleJS**引擎的限制与约束

函数调用参数个数限制

函数调用参数的个数限制，MapleJS引擎支持函数调用的参数的最大个数为255，当函数的调用的入参数大于255时，会出现：Script Error: SyntaxError: Maximum number of function arguments reached

函数定义参数个数限制

函数定义参数的个数限制，MapleJS引擎支持函数定义的参数的最大个数为256，当函数定义的入参数大于256时，会出现：Script Error: Maximum number of registers is reached

字符串长度限制

MapleJS引擎支持字符串的长度最大为65535，当字符串的长度超过65535时，会出现：Script Error: SyntaxError: String is too long

文件源码最大字节数限制

MapleJS引擎支持加载js文件的最大字节数为1048576，当源码字符超过最大限制后，后续的字符会被丢弃。

引用计数限制

MapleJS引擎支持对象的最大引用次数为1023，当引用次数超过最大限制后，会引起引擎重启。

引用计数（reference count）增减的基本规则是：

- 1) 当声明了一个变量并将一个引用类型值赋给该变量时，则该值的引用次数就是1；
- 2) 如果同一个值又被赋给另一个变量，则该值的引用次数加1；
- 3) 如果包含对该值引用的变量又取得了另外一个值，则该值的引用次数减1。

栈使用限制

默认MapleJS应用可使用的栈空间为3KB，超过限制会提示：Potential stack overflow, leaving only xx bytes of stack space!

Timer使用限制

定时器的事件触发频率太高会导致事件队列溢出，报错：`Fail to push event to event queue.` 当出现队列溢出时可做出以下调整：

1. 降低定时器触发事件的频率；
2. 缩短数据的处理时间；
3. 调大**Queue**的数量。

做上面的几个调整时，需要遵循以下原则：

1. 每个事件的处理时间要小于2个监听事件之间的间隔时间；
2. 需要处理的事件数量要小于当前空闲queue的数量。