浙江大学

嵌入式系统

大程报告

项目名称： 基于 VS Code 平台的 MicroPython on ESP32 开发插件

组　　名： 8 越界数组

指导老师： 翁恺

组员信息	
姓名	**学号**
杨苏洋	319010
潘恩皓	319010
张沛全	319010
包德政	319010

2022 年 6 月 6 日

1 项目背景

Python，是一种面向对象的解释型计算机程序设计语言，它是纯粹的自由软件，源代码和解释器CPython遵循GPL（GNU General Public License）协议。Python的设计目标之一是让代码具备高度的可阅读性。它设计时尽量使用其它语言经常使用的标点符号和英文单字，让代码看起来整洁美观。它不像其他的静态语言如C、Pascal那样需要重复书写声明语句，也不像它们的语法那样经常有特殊情况和意外。总之，Python是一种简单易用的、能够运行在多个平台下的计算机编程语言。而MicroPython，是跑在MCU上的Python，是嵌入式开发中非常常用的开发语言。它通过内置的解释器执行py文件或者py命令，就可以让微控制器运行想要的功能。MicroPython和Python编程语言一样，在任何板子上都可以使用通用的API控制硬件底层，比如点亮LED灯，读取传感器信息，LCD显示字符串、控制电机、连接网络、连接蓝牙等等。

虽然MicroPython相较于C在嵌入式开发上具有更高可读性和使用更便捷的优点。但由于它出现的时间较晚，到现在仍然没有形成良好的开发生态。其中一个明显的特征就是，目前还没有一个功能较为完善且方便易用的开发MicroPython的IDE，这反而导致了MicroPython在嵌入式系统上的开发并不好用。基于这样的现状，我们打算制作一款基于VS Code平台的可视化插件，实现在ESP32上MicroPython的编程、调试、部署环境等集成功能。

2 项目内容

本项目将尝试实现一个基于VS Code平台的可视化插件，实现在ESP32上MicroPython的编程、调试、部署环境等集成功能。

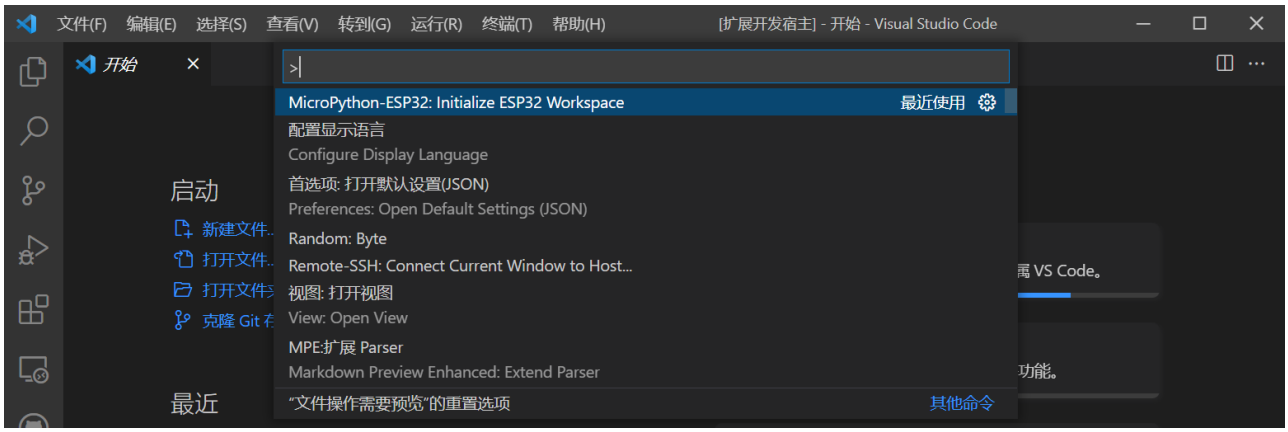
该插件将基于TypeScript进行开发，并参考pyboard.py、mpfshell等成熟的MicroPython连接工具，达到可以在ESP32进行便捷的MicroPython运行环境配置、Wi-Fi连接配置、代码运行与调试、文件系统概览等功能。

3 项目成果

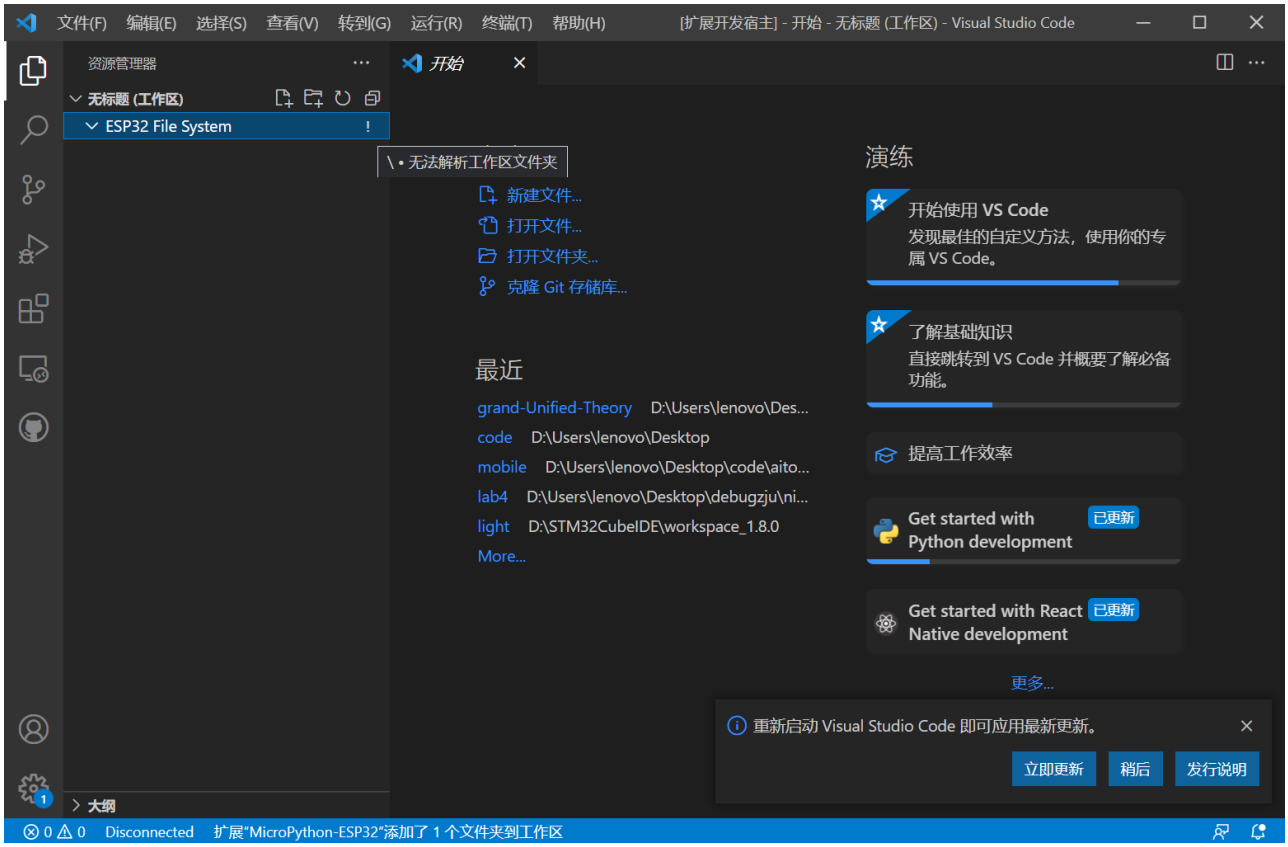
3.1 文件系统

3.1.1 效果展示

利用工作空间，实现了一个VS Code中的文件系统。如图呼出命令面板，输入初始化工作空间的命令。



可以看到命令执行完毕后左侧工作区出现了ESP32的文件系统。但是由于此时还未连接上ESP32，工作区提示无法解析工作区文件夹。



3.1.2 源码分析

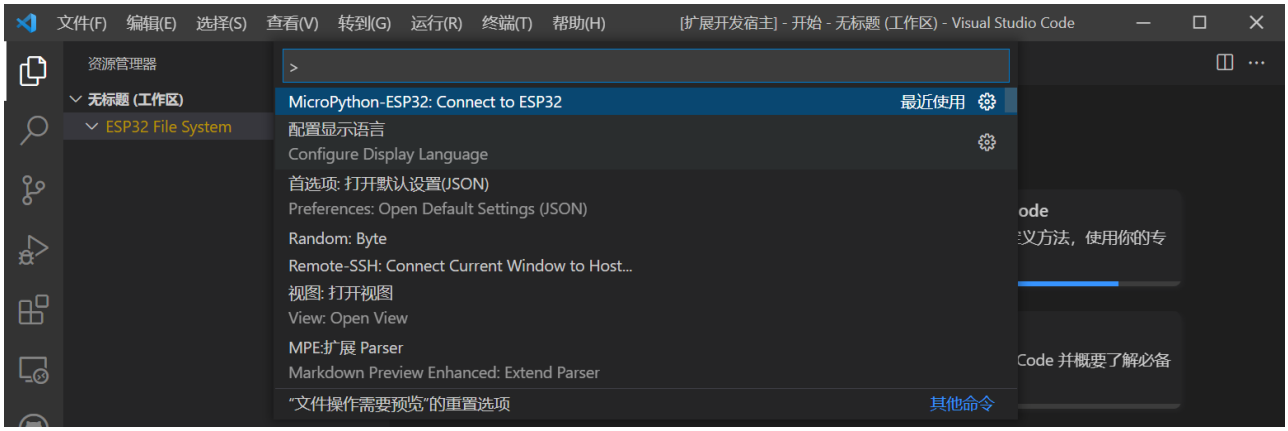
文件系统的显示功能主要是使用了`VS Code`的文件系统相关接口和函数，主要包括了`vscode.FileStat`，`vscode.FilrSystemProvider`，而文件目录信息的获取，则是利用`MicroPython`中`os.ilistdir`与`os.stst`两个函数的返回结果，并进行字符串解析得到的。

相关代码放置在`fileSystem.ts`中。

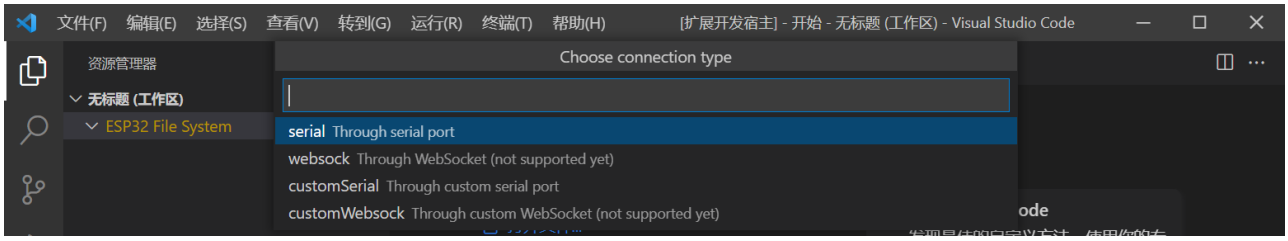
3.2 串口连接

3.2.1 效果展示

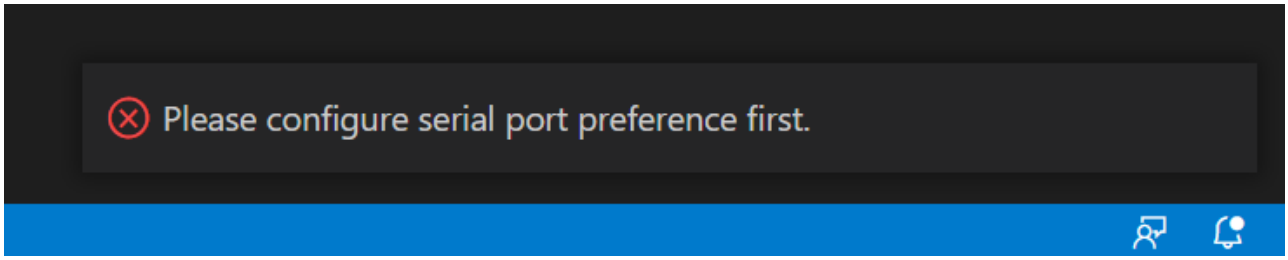
如图呼出命令面板，选择连接到`ESP32`命令。



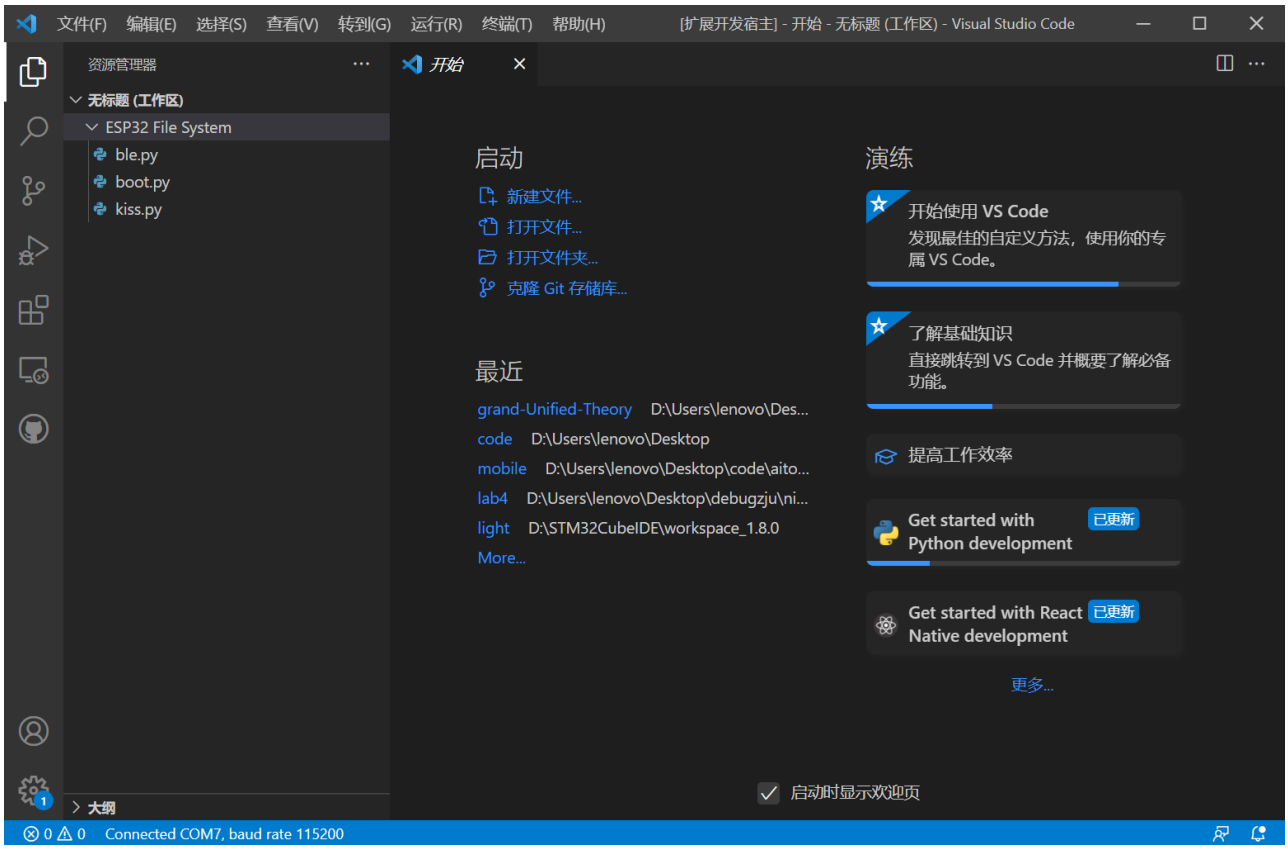
第一个选项为使用配置文件中的默认配置连接到`ESP32`，第三个选项需要用户手动输入端口以及码率以连接到`ESP32`。



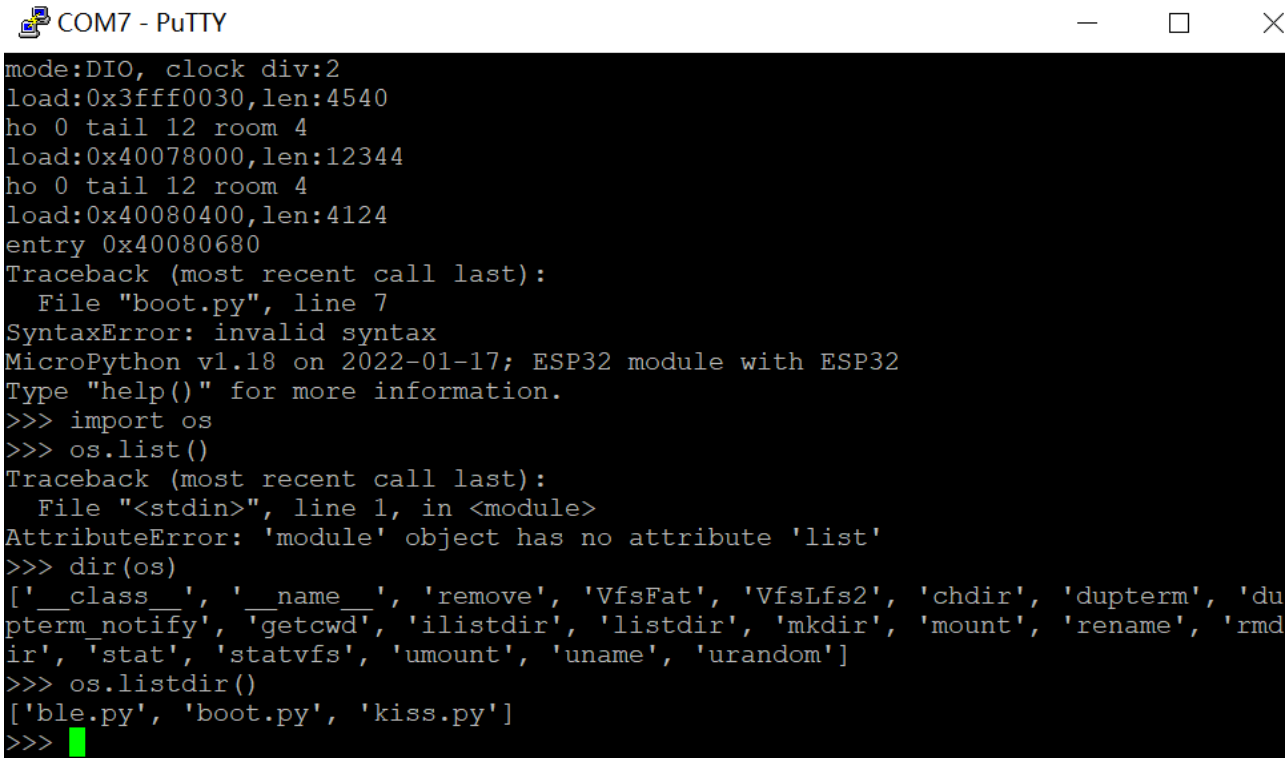
如果使用配置文件缺少配置，会提示配置端口数据。



选择第三个选项并手动输入端口`COM7`以及码率 115200，可以看到左下角提示连接成功。工作空间也出现了`ESP32`上的三个文件。



用`PuTTY`连接到端口查看文件，可以验证这三个文件确实存在于`ESP32`上。



3.2.2 源码分析

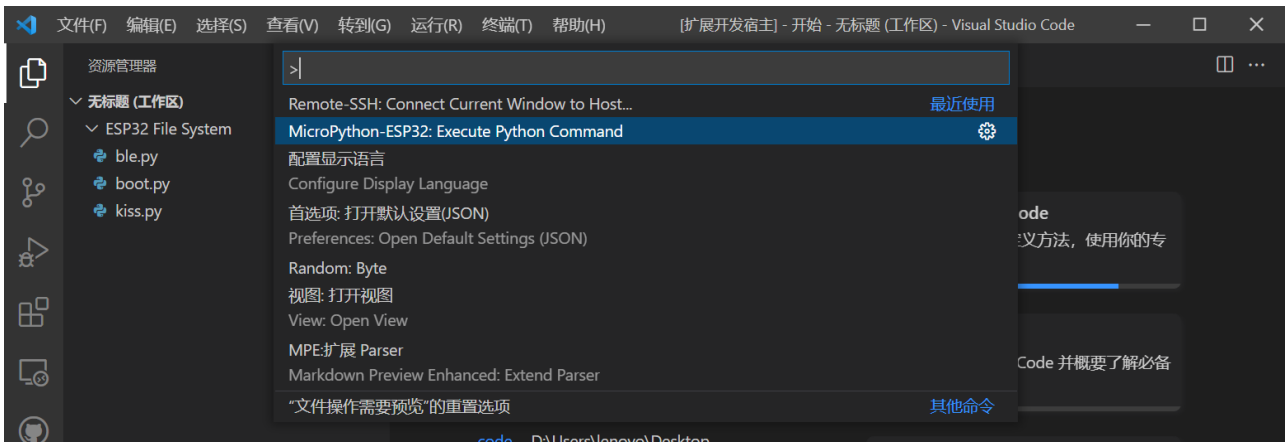
串口连接主要使用了`serialport`包中的`SerialPort`类，在已知串口与波特率的情况下，使用该类的`read`，`write`，`drain`等功能，封装成我们需要的`read`，`write`与`close`等功能。

其中具体实现放置在`connection/serial.ts`内， 对外接口放置在`connection/port.ts`内。

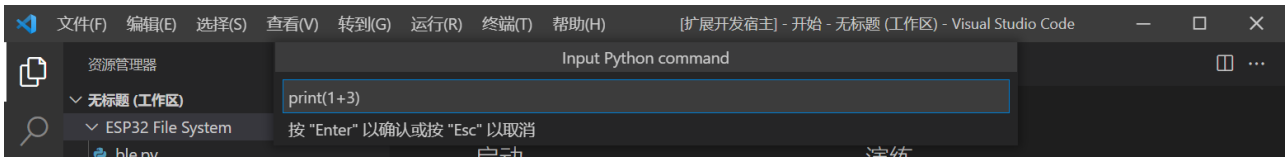
3.3 执行命令

3.3.1 效果展示

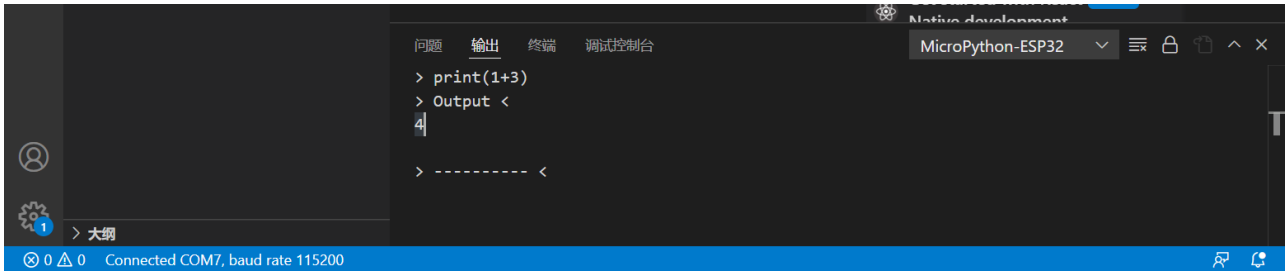
呼出命令面板，选择执行`Python`命令。



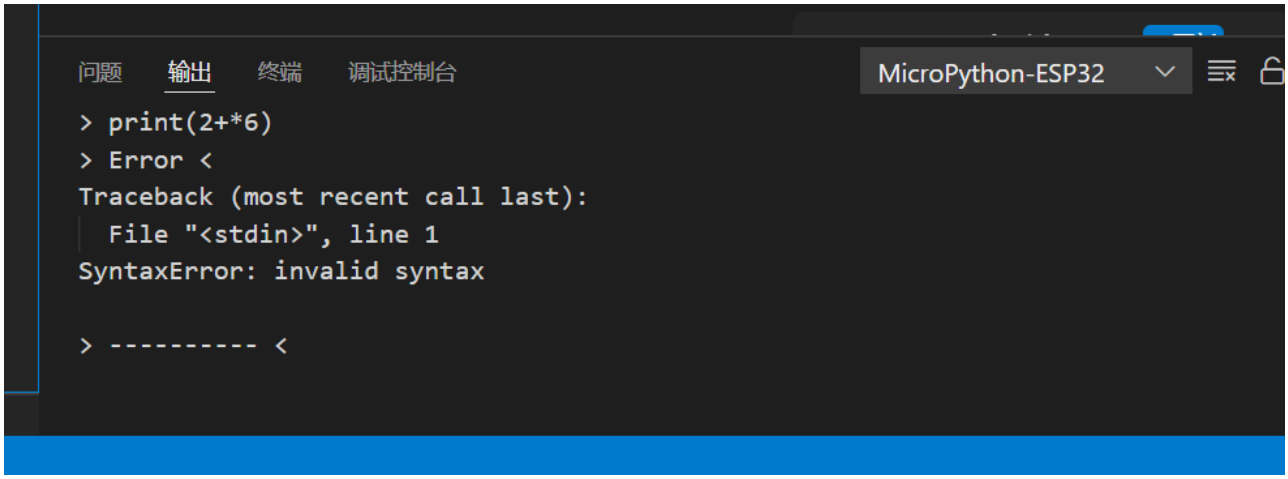
输入一条语句，注意要有`print()`。



可以看到下方产生了输出，并且显示了执行的命令。



如果出错，则会输出错误。



3.3.2 源码分析

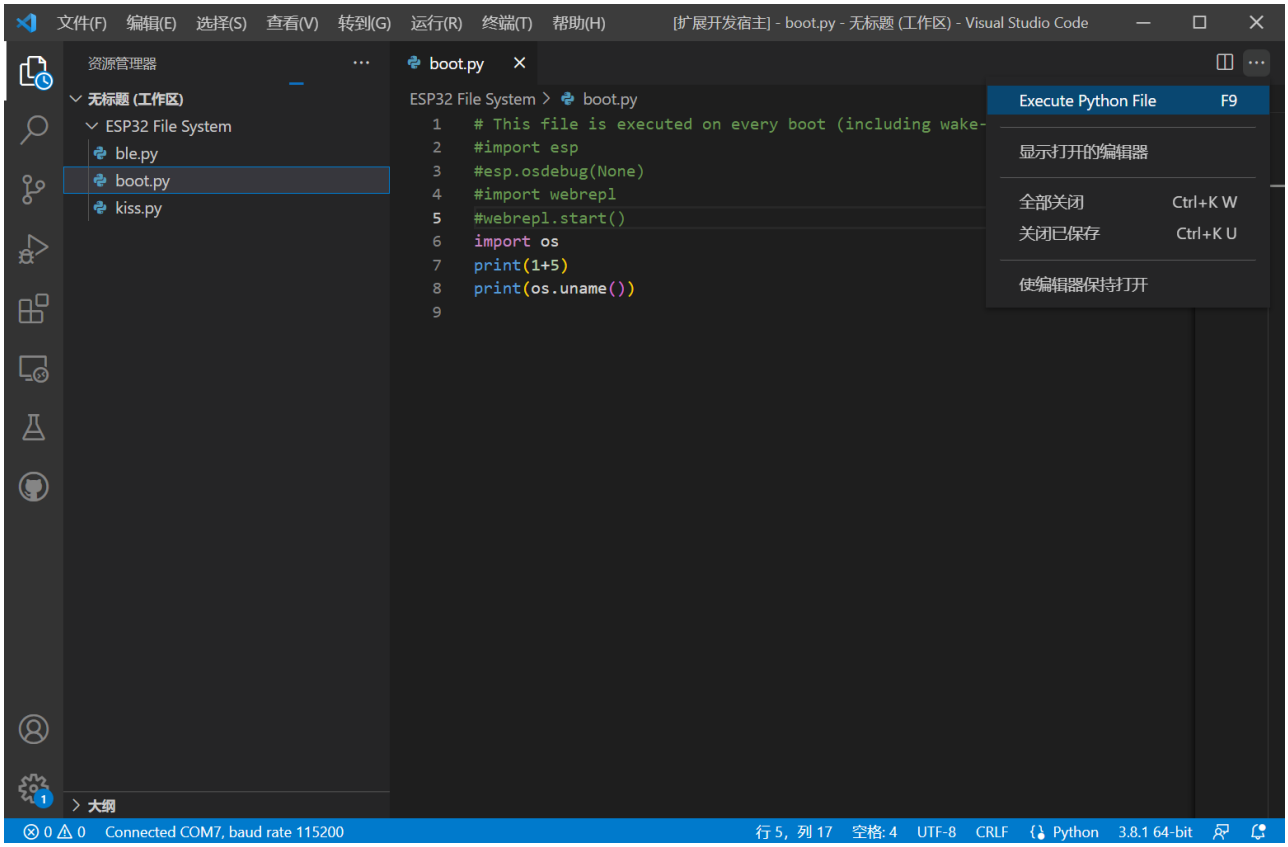
执行命令实际就是向串口发送对应的指令，并加上`\x04`作为指令结束的标志。向串口发送利用了上述提及的`write`函数，并通过`read`获取指令运行的结果。

这一代码放置在`connection/index.ts`的`_exec`函数中。鉴于串口发送结果以`OK`等为结尾表征数据传输结束以及运行结果，因此可以一直读取到`OK`后结束，此后再对结果进行字符串解析。我们加入了一个可选的回调函数。如果没有传入回调函数，则会常规地读入单片机发送的数据以及错误信息并显示。如果传入了回调函数，则可以实现在控制台中交互的效果。

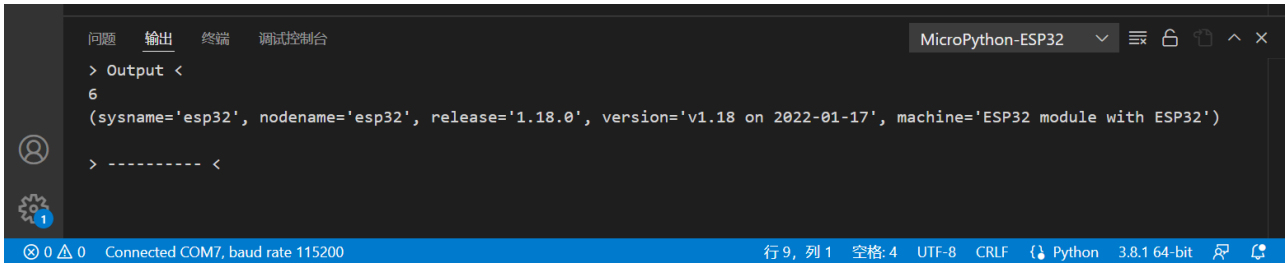
3.4 执行文件

3.4.1 效果展示

打开一个文件，可以按下`F9`或者在右上方点击执行文件。



可以看到产生输出。



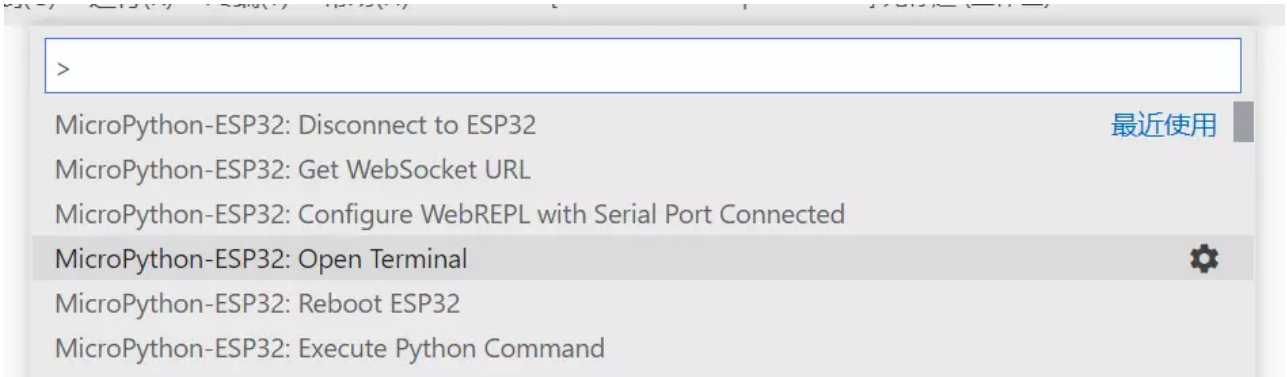
3.4.2 源码分析

执行文件与执行某行命令并无区别，只是将命令为执行某一文件。代码中以`isFile`布尔量作为区别。

3.5 打开终端

3.5.1 效果展示

呼出命令面板，选择打开终端命令。



窗口下方显示打开的终端，可以在里面输入`Python`命令并执行。

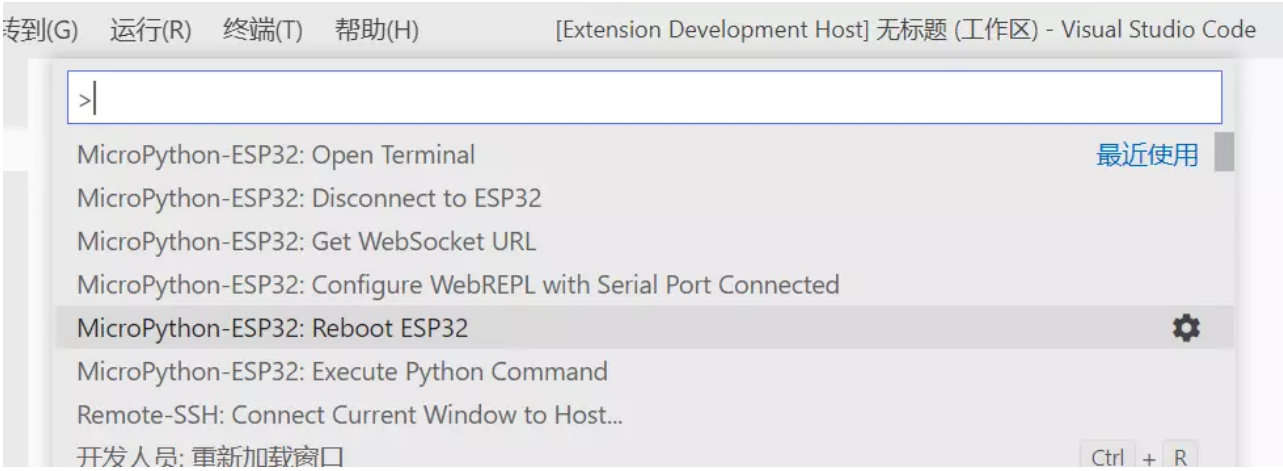
3.5.2 源码分析

在插件功能的实现中一般使用 `MicroPython` 的 raw REPL 模式，这种模式下输入命令不会回显，可以直接获取到命令的执行结果。但这里要打开的是可以与用户交互的终端，所以应该将 `MicroPython` 切换至 normal REPL 模式，这可以通过向板子发送 `\x02` 字符来实现。之后的交互与之前提到的执行命令、文件的交互类似，用回调函数来实现。在用户关闭终端时，需要再向板子发送 `\x01` 字符，切换回 raw REPL 模式，使其他功能正常工作。

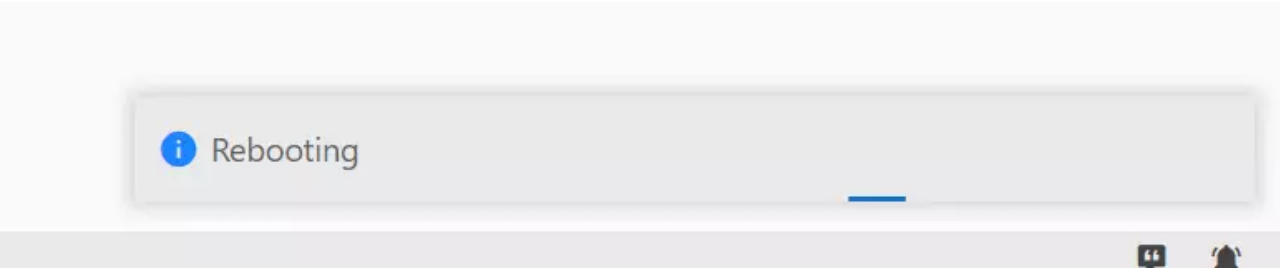
3.6 重启

3.6.1 效果展示

呼出命令面板，选择重启命令。



窗口右下角显示加载条，一段时间后重启完成。



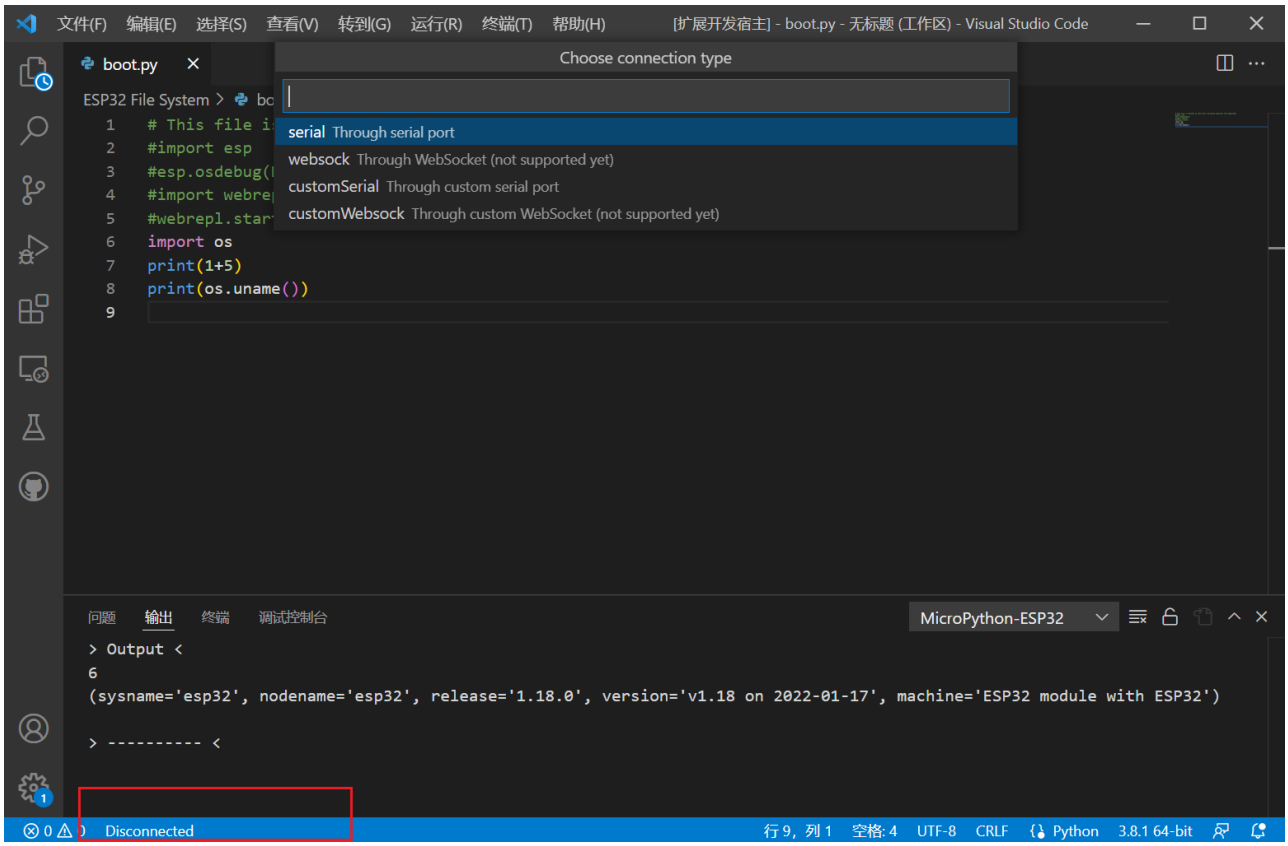
3.6.2 源码分析

重启其实就是执行 `machine.reset()` 命令。为了给用户更好的体验，重启后需要自动重新连接板子。重启命令执行一段时间后，我们会尝试再次与板子建立连接，并执行初始化代码。

3.7 断开连接

3.7.1 效果展示

点击左下方状态即可断开连接。



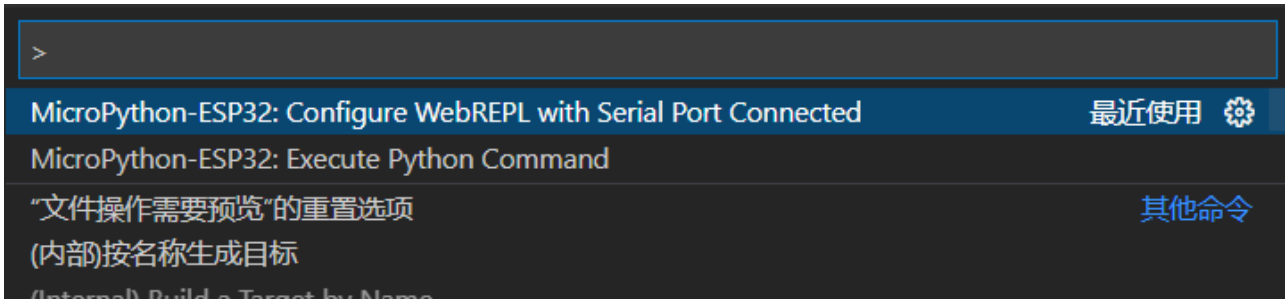
3.7.2 源码分析

即利用上述提及的`close`功能，断开连接。

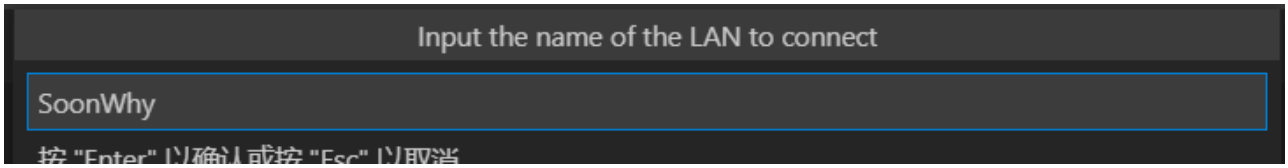
3.8 WebREPL配置

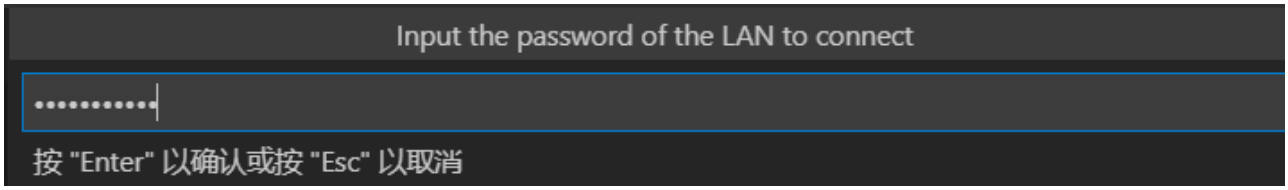
3.8.1 效果展示

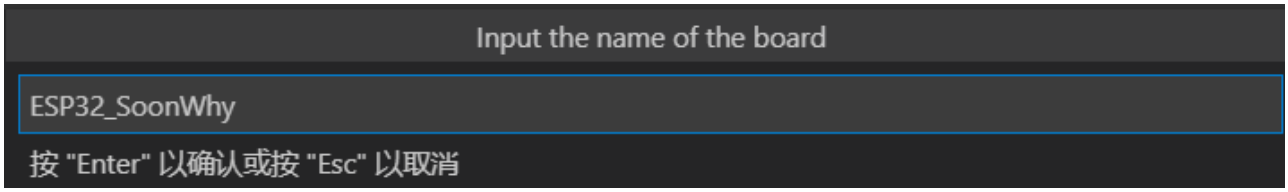
通过串口连接`ESP32`，选择配置`WebREPL`的选项：`Configure WebREPL with Serial Port Connected`

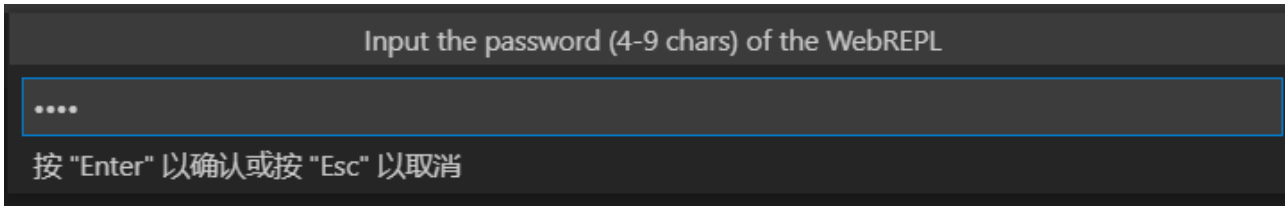


依次输入`WLAN`名、`WLAN`密码、板子的名字（自定义）以及板子的密码：

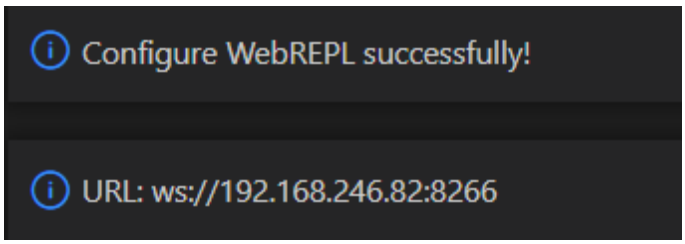




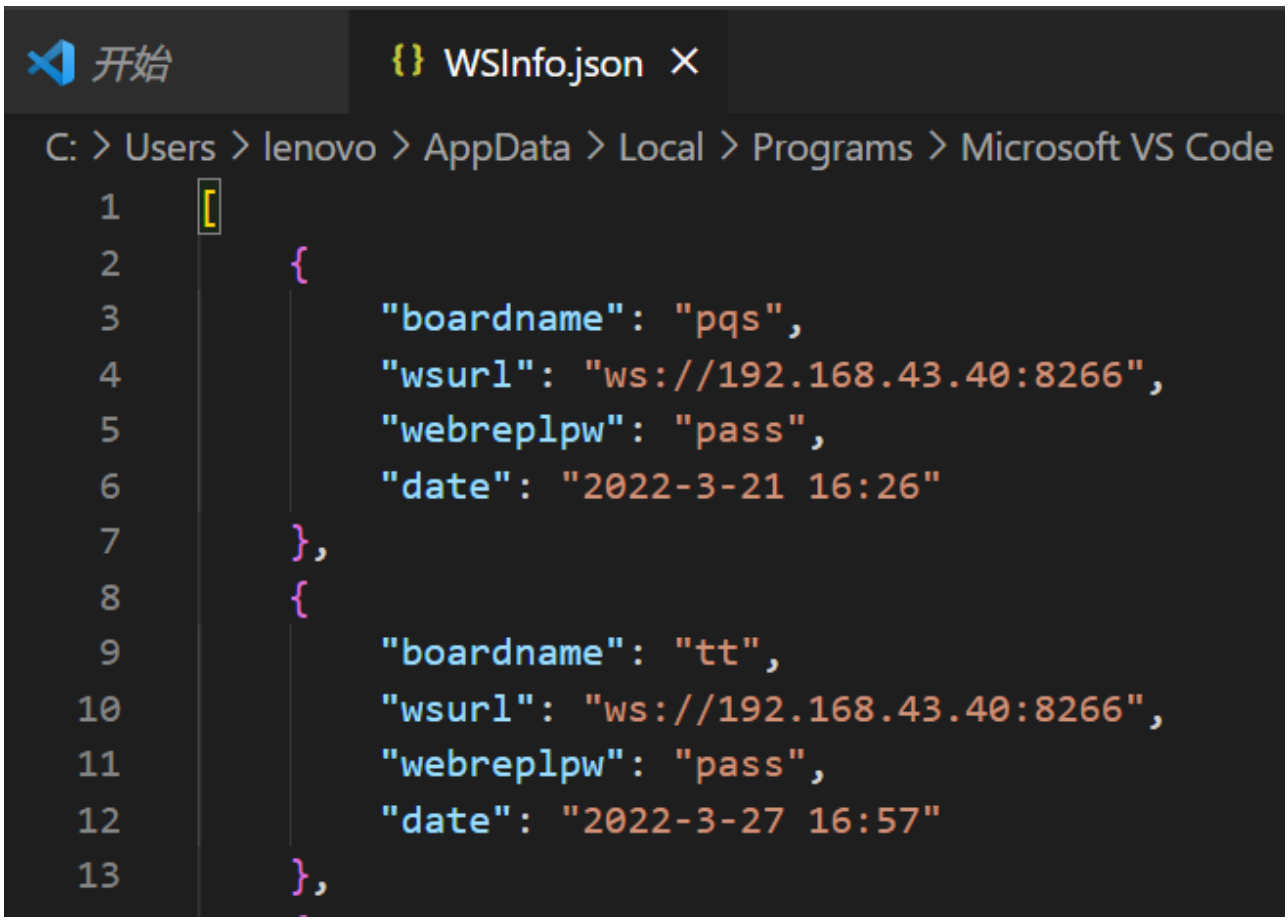




即可完成配置，右下角提示建立的连接的 `IP` 地址：



配置完成后，在本地路径下的WSInfo.json里面可以找到保存的单片机信息。

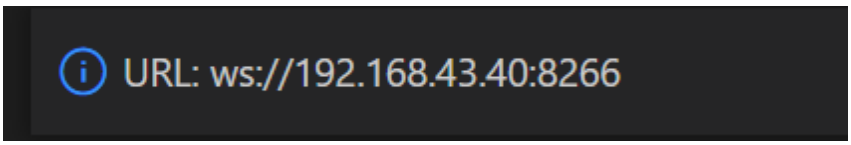


可在 `boot.py` 中发现配置的启动代码：

```
1
2
3   import network
4   import time
5   import webrepl
6   wlan = network.WLAN(network.STA_IF)
7   wlan.active(True)
8   if wlan.isconnected():
9     wlan.disconnect()
10  wlan.connect('SoonWhy', '[redacted]')
11  maxtries = 9
12  while maxtries >0:
13    if wlan.isconnected():
14      webrepl.start()
15      break
16    maxtries = maxtries - 1
17    time.sleep_ms(500)
18
```

在完成配置后，除了可以在本地文件中查看url信息，还可以在保持串口连接的情况下使用命令查看url。

得到当前板子的url：



3.8.2 源码分析

在实验配置的过程中，我们发现如果使用 `import webrepl_setup` 命令，则需要处理许多命令输出的提示语句与字符。我们查看了该命令的源码，发现源码主要做了两件事：一是在 `boot.py` 中加入 `import webrepl` 与 `webrepl.start()` 两条语句（如果被注释的有语句包含 `webrepl` ，则这些语句会被取消注释）；而是新建或修改 `webrepl_cfg.py` 以保存密码。

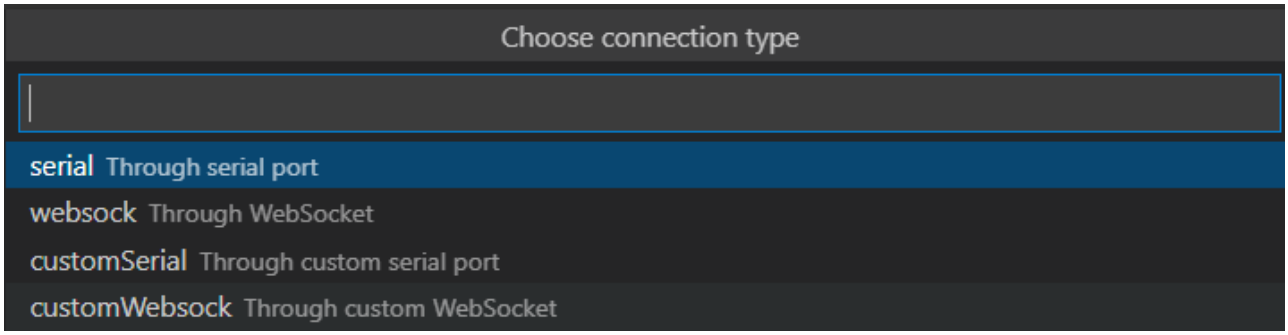
配置时输入的单片机名字以及密码、配置时间等信息会被保存到一个本地的json文件里，方便管理多个单片机。

得知这些信息后，在实现配置时我们不再采用输入 `import webrepl_setup` 命令的方式，免去了处理大量提示语句的麻烦。我们直接修改 `boot.py` 与 `webrepl_cfg.py` 两个文件，并且在启动 `WebREPL` 之前先尝试连接到配置时设置的网络下。如果网络未能成功连接，则不会开启 `WebREPL` 。

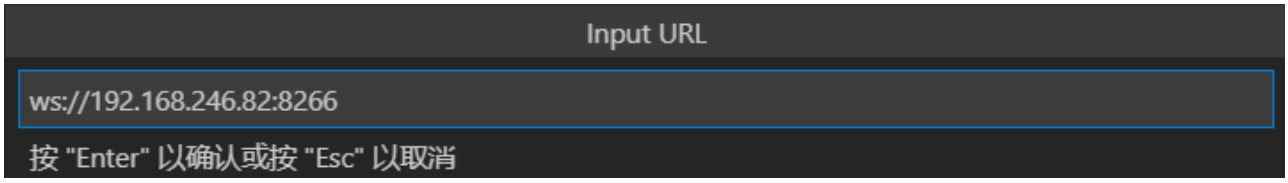
3.9 WebSocket连接

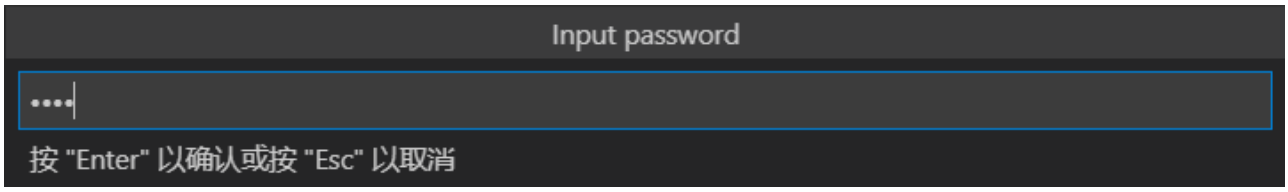
3.9.1 效果展示

在配置好 `WebREPL` 后，使用对应的 `IP` 进行连接，要注意，此时 `PC` 应与 `ESP32` 处在同一网络下。建立 `WebSocket` 连接：

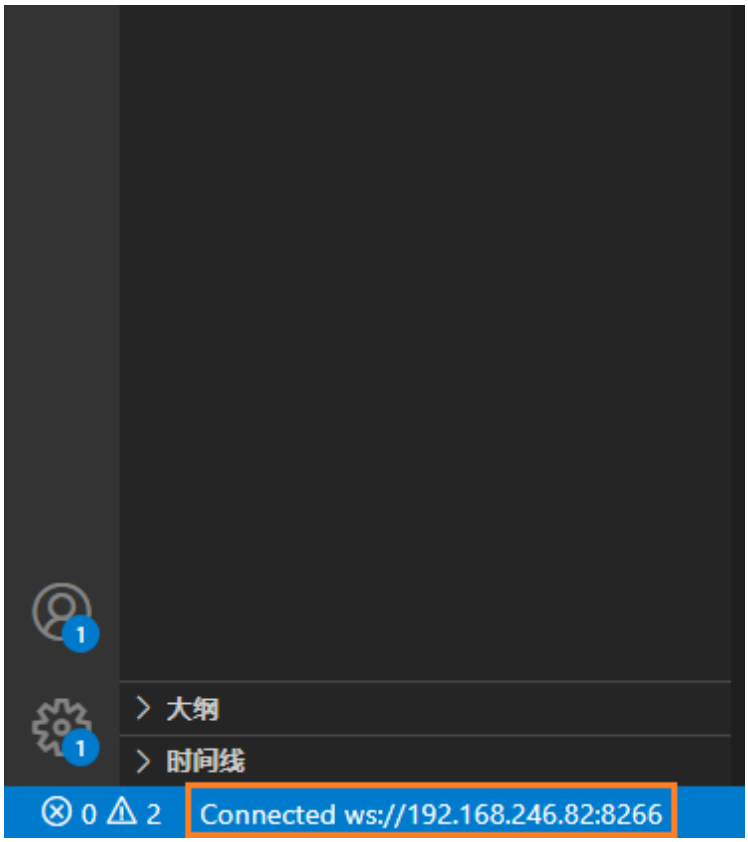


依次输入 `URL` 与板子的密码：

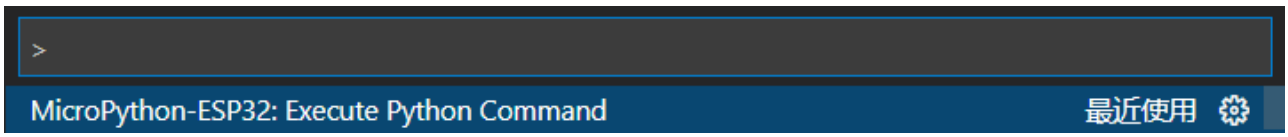


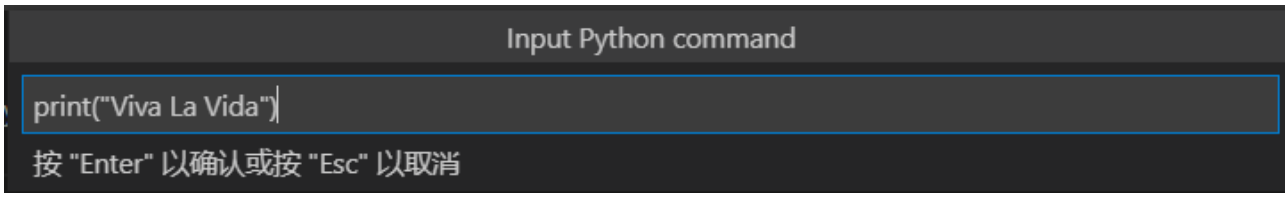


即成功建立连接：



尝试执行 `Python` 语句：





3.9.2 源码分析

`Typescript` 中常用的 `socket` 连接的包主要有 `socket.io` 和 `ws` （即 `websock` ）两种。其中通过 `WireShark` 抓包发现， `socket.io` 并没有完整实现 `WebSocket` 的协议，因此不能成功连接 `ESP32` ，所以最终选择通过 `ws` 实现连接功能，主要代码在 `./connection/websock.ts` 内。

具体的连接功能是以 `URL` 为参数，新建一个 `WebSock` 对象，并通过注册监听回调函数（包括 `onopen` ，`onmessage` ，`onerror` 等）来进行交互。其中 `onopen` 负责监听连接建立成功的时间，而 `onmessage` 负责处理 `ESP32` 发送来的消息，一开始的初始化配置也在这里完成， `onerror` 则负责处理连接上出现的报错。

在此基础上，通过一个全局的 `buffer` 实现了 `read` 操作。使用 `ws.send()` 实现了 `write` 功能。

在对比串口连接的功能后，我们对二者进行了统一化处理。

- 首先是将初始化操作统一以回调函数的方式执行，这是因为网络连接需要一定时间建立，因此需要等到建立完成后才能执行初始化函数，而这一问题在串口连接的时候没有发现是因为串口连接速度很快。
- 其次是统一了二者的错误信息处理函数。
- 最后是通过实验发现在某些网络环境下， `WebSocket` 交互的信息量不能过高，通过实验最终选取了 `128Bytes` 作为包的大小上限。

4 整体测试

对插件的每个模块以及整体进行了多次测试，可以使用插件更为轻松完成 `ESP32` 上的第11次课程实验。

5 感想

本次课程完成了一个非常有意义的项目。尤其是在做到第11次实验时，使用我们自己的插件可以免去使用很多个软件或者工具的麻烦。比较遗憾的是，我们没能圆满地实现利用插件调试 `MicroPython` 的目标。我们在开发中发现， `MicroPython` 本身并没有很好地支持调试的功能，比如 `MicroPython` 在记录调试信息时并没有区分局部变量全局变量，查看源代码也可以从注释中发现这一问题（我们尝试修改源码来解决这一问题，但无疾而终）；又比如官方发布的 `MicroPython` 镜像并不支持使用 `settrace()` ，要使用该函数则需要用户自行编译镜像。在 `MicroPython` 本身难以支持开发完善的调试功能的情况下，我们只好暂时放下这一目标。

```c
mp_obj_t mp_obj_new_code(const mp_module_context_t *context, const mp_raw_code_t *rc) {
    mp_obj_code_t *o = m_new_obj_maybe(mp_obj_code_t);
    if (o == NULL) {
        return MP_OBJ_NULL;
    }
    o->base.type = &mp_type_settrace_codeobj;
    o->context = context;
    o->rc = rc;
    o->dict_locals = mp_locals_get(); // this is a wrong! how to do this properly?
    o->lnotab = MP_OBJ_NULL;
    return MP_OBJ_FROM_PTR(o);
}
```

6 参考资料

[1] Quick reference for the ESP32

[2] MicroPython Libraries

[3] GitHub - wendlers/mpfshell

[4] VS Code 插件创作中文开发文档

[5] TypeScript 中文网

[6] Node SerialPort

[7] Socket.IO

[8] sys --- 系统相关的参数和函数 — Python 3.10.4 文档

[9] GitHub - bobveringa/mpdb

[10] _thread --- 底层多线程 API — Python 3.10.4 文档

[11] GitHub-micropython/webrepl

[12] GitHub-websockets/ws