配网客户端的调试摸索

- 由于官方仅提供了java语言的安卓端app代码，如果希望自定义app或者使用客户端进行复刻，则需要前往开源配网软件源代码SoftAPTransport.java查看相关逻辑，需要复刻的话，必须清晰http的通信交互/协议交换（官方文档没写，怀疑这个文档是乐鑫官方写给自家公司内部看的），
- 开源app中的 HTTP 请求的核心流程体现在如下方法里：

> `sendPostRequest(String path, byte[] data, ResponseListener listener)`
> 负责构造并发送 *HTTP POST* 请求到指定 *path*（如 `prov-session`, `prov-scan` 等），并处理服务器（*ESP* 设备）响应。
> `sendConfigData(String path, byte[] data, ResponseListener listener)`
> 对外暴露接口，实际通过线程池异步调用 `sendPostRequest` 完成配网通信。

```
//java 代码片段如下（省略部分内容）：
URL url = new URL("http://" + baseUrl + "/" + path);
HttpURLConnection urlConnection = (HttpURLConnection)
url.openConnection();
urlConnection.setRequestMethod("POST");
urlConnection.setRequestProperty("Accept", "text/plain");
urlConnection.setRequestProperty("Content-type",
"application/x-www-form-urlencoded");
...
OutputStream os = urlConnection.getOutputStream();
os.write(data);
os.close();
```

- 其中 `baseUrl` 会配置为设备的 mDNS 地址，比如 `wifi-prov.local`。而esp默认的官方地址为`192.168.4.1`，这一点在esp的调试终端中可以看到

```
2025-06-04 20:22:23 I (928) wifi:mode : softAP
(34:98:7a:0e:af:65)
2025-06-04 20:22:23 I (928) wifi:Total power save buffer
number: 16
2025-06-04 20:22:23 I (928) wifi:Init max length of beacon:
752/752
2025-06-04 20:22:23 I (938) wifi:Init max length of beacon:
752/752
2025-06-04 20:22:23 I (938) esp_netif_lwip: DHCP server
started on interface WIFI_AP_DEF with IP: 192.168.4.1
2025-06-04 20:22:23 I (958) wifi:flush txq
2025-06-04 20:22:23 I (958) wifi:stop sw txq
2025-06-04 20:22:23 I (958) wifi:lmac stop hw txq
2025-06-04 20:22:23 I (978) WIFI: Starting provisioning...
2025-06-04 20:22:23 I (978) wifi:mode : sta
(34:98:7a:0e:af:64)
2025-06-04 20:22:23 I (978) wifi:enable tsf
2025-06-04 20:22:23 I (988) wifi:mode : sta
(34:98:7a:0e:af:64) + softAP (34:98:7a:0e:af:65)
2025-06-04 20:22:23 I (988) wifi:Total power save buffer
number: 16
2025-06-04 20:22:23 I (998) wifi:Init max length of beacon:
752/752
2025-06-04 20:22:23 I (1008) wifi:Init max length of beacon:
752/752
2025-06-04 20:22:23 I (1008) esp_netif_lwip: DHCP server
started on interface WIFI_AP_DEF with IP: 192.168.4.1
2025-06-04 20:22:23 W (1028) wifi:Affected by the ESP-NOW
encrypt num, set the max connection num to 2
2025-06-04 20:22:23 I (1288) wifi:Total power save buffer
number: 16
2025-06-04 20:22:23 I (1288) esp_netif_lwip: DHCP server
started on interface WIFI_AP_DEF with IP: 192.168.4.1
2025-06-04 20:22:23 I (1298) wifi_prov_mgr: Provisioning
started with service name : SMARTHOST_PROV
2025-06-04 20:22:23 I (1308) WIFI: PROVISIONING STARTED
```

- 当设备连接上之后，设备端分配到IP地址。此时就可以进行http的访问请求了

```
2025-06-04 21:11:57 I (2974568) wifi:station:
8c:c6:81:9a:bb:18 join, AID=1, bgn, 20
2025-06-04 21:11:57 I (2974848) esp_netif_lwip: DHCP server
assigned IP to a client, IP is: 192.168.4.2
```

- 初次尝试，在客户端或者浏览器/终端，进行`GET`方法请求，出现以下报错`Specified method is invalid for this resource`，说明当前端点，并不支持`GET`方法。

```
2025-06-04 20:15:48 I (164858) wifi:station:
8c:c6:81:9a:bb:18 join, AID=1, bgn, 20
2025-06-04 20:15:49 I (165068) esp_netif_lwip: DHCP server
assigned IP to a client, IP is: 192.168.4.2
2025-06-04 20:15:49 W (165278) wifi:<ba-add>idx:2 (ifx:1,
8c:c6:81:9a:bb:18), tid:0, ssn:22, winSize:64
2025-06-04 20:15:50 W (166438) wifi:<ba-add>idx:3 (ifx:1,
8c:c6:81:9a:bb:18), tid:6, ssn:1, winSize:64
2025-06-04 20:16:23 W (199578) httpd_uri: httpd_uri: Method
'1' not allowed for URI '/proto-ver'
2025-06-04 20:16:23 W (199578) httpd_txrx:
httpd_resp_send_err: 405 Method Not Allowed - Specified
method is invalid for this resource
```

- 再次尝试，使用`POST`方法请求，但是报错`Content length not found`，此时说明POST 请求没有携带 body，也没有自动携带 `Content-Length` 头。

```
curl -v -X POST -H "Accept: application/json" -d '{}'
http://192.168.4.1/proto-ver
```

```
2025-06-04 21:10:34 W (2891798) protocomm_httpd: Closing
session with ID: 996713560
2025-06-04 21:10:34 E (2891798) protocomm_httpd: Content
length not found
2025-06-04 21:10:34 W (2891798) httpd_uri: httpd_uri: uri
handler execution failed
2025-06-04 21:10:34 W (2891808) protocomm_httpd: Resetting
socket session id as socket 57was closed
```

- 而ESP 侧的 protocomm 实现要求所有 POST 请求必须有 `Content-Length` 字段（哪怕 body 为空）。

```
curl -v -X POST -H "Content-Type: application/json" -d ''
http://192.168.4.1/proto-ver
```

- 连接后，客户端应用程序可以立即从 `proto-ver` 端点获取版本或功能信息。所有与此端点的通信均未加密，因此在建立安全会话之前，可以检索相关必要信息，确保会话兼容。响应数据采用 JSON 格式，示例如下：`prov: { ver: v1.1, sec_ver: 1, sec_patch_ver: 0, cap: [no_pop] }, my_app: { ver: 1.345, cap: [cloud, local_ctrl] },....`。此时在终端进行访问成功之后，终端信息如下所示，拿到了`prov`的json文本标签即视为成功。

```
C:\Users\odddouglas>curl -v -X POST -H "Content-Type:
application/json" -d '' http://192.168.4.1/proto-ver
Note: Unnecessary use of -X or --request, POST is already
inferred.
*   Trying 192.168.4.1:80...
* Connected to 192.168.4.1 (192.168.4.1) port 80
* using HTTP/1.x
> POST /proto-ver HTTP/1.1
> Host: 192.168.4.1
> User-Agent: curl/8.12.1
> Accept: */*
> Content-Type: application/json
> Content-Length: 2
```

```
>
* upload completely sent off: 2 bytes
< HTTP/1.1 200 OK
< Content-Type: text/html
< Content-Length: 73
< Set-Cookie: session=2252508986
<
{
        "prov": {
                "ver":  "v1.1",
                "sec_ver":      1,
                "cap":  ["wifi_scan"]
        }
}* Connection #0 to host 192.168.4.1 left intact
```

具体配网步骤

- softap配网相关http逻辑：在原生框架中，有一段提示。`typings\types\wx\lib.wx.api.d.ts`中

```
* 发起 HTTPS 网络请求。使用前请注意阅读[相关说明]
(https://developers.weixin.qq.com/miniprogram/dev/framework/
ability/network.html)。
*
* **data 参数说明**
*
*
* 最终发送给服务器的数据是 String 类型，如果传入的 data 不是 String
类型，会被转换成 String 。转换规则如下：
* - 对于 `GET` 方法的数据，会将数据转换成 query string
(`encodeURIComponent(k)=encodeURIComponent(v)&encodeURICompo
nent(k)=encodeURIComponent(v)...`)
* - 对于 `POST` 方法且 `header['content-type']` 为
`application/json` 的数据，会对数据进行 JSON 序列化
* - 对于 `POST` 方法且 `header['content-type']` 为
`application/x-www-form-urlencoded` 的数据，会将数据转换成 query
string `
(encodeURIComponent(k)=encodeURIComponent(v)&encodeURICompon
ent(k)=encodeURIComponent(v)...)`
*
```

- 因此在消息发送的过程中，我们要注意消息体的data类型，这很重要。我们可以使用`request`关键词，进行基本的请求，得到响应。首先我们先进行/proto-ver的端点请求以获得设备端当前的版本和安全性等信息。

```
wx.request({
  url: 'http://192.168.4.1/proto-ver',
  method: 'POST',
  header: {
    'Content-Type': 'application/json' //请求头
  },
  data: '{}',
  success: function(res) {
    console.log('请求成功：', res.data);
  },
  fail: function(err) {
    console.error('请求失败：', err);
```

```
  }
});
```

- 接下来是十分重要的几步，根据官方文档寥寥几句，因此猜测需要将.proto文件通过 protobufjs 库转换成客户端所能调用的 JS 模块文件，推荐使用如下命令将 .proto 文件转换为 JavaScript 及类型定义文件：

```
# 安装 protobufjs 工具
npm install -g protobufjs

# 将 proto 文件转换为 js 模块
pbjs -t static-module -w commonjs -o proto_bundle.js
session.proto sec1.proto wifi_config.proto

# 生成类型定义（可选）
pbts -o proto_bundle.d.ts proto_bundle.js
```

- 而这个所谓的 `proto` 文件，实质上定义了配网过程中的数据结构和通信协议。常见文件包括 `session.proto`、`sec0.proto`、`wifi_config.proto` 等，它们被 ESP 端与客户端共同使用以确保通信协议一致性。需要在espidf中进行寻找proto文件，以下两个为例，一个是安全性proto，一个是wifi配网的配置proto

```
esp-idf\components\protocomm\proto\sec0.proto
esp-idf\components\wifi_provisioning\src\wifi_config.proto
```

- 生成后，我们就可以通过 `protobufjs` 提供的方式进行序列化/反序列化操作。我们可以在生成的js文件中往下翻找可以看到 `[WiFiConfigPayload]`，这是一个接口（我翻了好久，没想到不在最顶层），定义了 WiFi 配置相关的消息类型及其命令和响应的属性结构。

```
/**
     * Properties of a WiFiConfigPayload.
     * @exports IWiFiConfigPayload
     * @interface IWiFiConfigPayload
     * @property {WiFiConfigMsgType|null} [msg]
WiFiConfigPayload msg
     * @property {ICmdGetStatus|null} [cmdGetStatus]
WiFiConfigPayload cmdGetStatus
     * @property {IRespGetStatus|null} [respGetStatus]
WiFiConfigPayload respGetStatus
     * @property {ICmdSetConfig|null} [cmdSetConfig]
WiFiConfigPayload cmdSetConfig
     * @property {IRespSetConfig|null} [respSetConfig]
WiFiConfigPayload respSetConfig
     * @property {ICmdApplyConfig|null} [cmdApplyConfig]
WiFiConfigPayload cmdApplyConfig
     * @property {IRespApplyConfig|null} [respApplyConfig]
WiFiConfigPayload respApplyConfig
     */
```

- 我们首先在客户端中引入模块

```
import * as sec0 from '../../../common/sec0.js';
import * as wificonfig from
'../../../common/wifi_config.js';
```

- 不过需要注意`sec0.js`和`wifi_cconfig.js`文件的export部分，通常会由于`protobuf.js`的全局命名空间`[$protobuf.roots["default"]]`导致的。`protobuf.js`默认会将所有定义的消息存储在`[$protobuf.roots["default"]]`中，因此两个文件导入时会共享同一个命名空间`[$root]`，导致内容冲突或覆盖。

```
// Exported root namespace 两个js文件的置顶导出，均为这个，
/*eslint-disable block-scoped-var, id-length, no-control-
regex, no-magic-numbers, no-prototype-builtins, no-
redeclare, no-shadow, no-var, sort-vars*/
"use strict";

var $protobuf = require("protobufjs/minimal");

// Common aliases
var $Reader = $protobuf.Reader, $Writer = $protobuf.Writer,
$util = $protobuf.util;

// Exported root namespace
var $root = $protobuf.roots["default"] ||
($protobuf.roots["default"] = {});
```

- 为了让 `sec0.js`和 `wifi_config.js` 独立工作，可以避免使用全局命名空间 `$protobuf.roots["default"]`，改为为每个文件定义独立的 `$root` 命名空间将 `$root` 定义为一个独立的命名空间，而不是使用 `$protobuf.roots["default"]`

```
/*eslint-disable block-scoped-var, id-length, no-control-
regex, no-magic-numbers, no-prototype-builtins, no-
redeclare, no-shadow, no-var, sort-vars*/
"use strict";

var $protobuf = require("protobufjs/minimal");

// Common aliases
var $Reader = $protobuf.Reader, $Writer = $protobuf.Writer,
$util = $protobuf.util;

// Exported root namespace
var $root ={};
```

- 解决以上问题之后，为了wifi的正确配置，我们还需要首先一次安全会话的建立（握手），为了方便，在这里我们在设备端和客户端都约定了`sec0`的方案（即安全性0，无密钥交换）。

```
 if (!provisioned)
    {
        is_provisioning = true;
        ESP_LOGI(TAG, "Starting provisioning...");

        const char *service_name = "SMARTHOST_PROV";
        const char *service_key = "abcd1234"; // SoftAP 密码
        const char *pop = NULL;          // Proof of
possession

        wifi_prov_security_t security =
WIFI_PROV_SECURITY_0;


 ESP_ERROR_CHECK(wifi_prov_mgr_start_provisioning(security,
pop, service_name, service_key));
    }
```

- 接下来就是进行串行的流程，按顺序执行 `proto-ver → prov-session → prov-config → apply-config`，这里写的不够完善，建议使用 `async/await` 或 `Promise` 链式写法。

```
// 创建会话
// 创建消息对象
const message = sec0.Sec0Payload.create({
  msg: sec0.Sec0MsgType.S0_Session_Command, // 假设 0 是有效的
枚举值
  sc: {}  // 假设 sc 是一个嵌套消息，符合 S0SessionCmd 的定义
});
// 序列化为 Uint8Array
// const buffer = S0SessionCmd.encode(message).finish();
const buffer = sec0.Sec0Payload.encode(message).finish();
// Log the serialized buffer
```

```
console.log(buffer);
const newbuffer = Uint8Array.from(buffer).buffer;
console.log(newbuffer);
// 使用 wx.request 或 wx.connectSocket 发送 buffer
uni.request({
    url: 'http://192.168.4.1/prov-session', // 替换为设备的实际IP地址和端点
    method: 'POST',
    header: {
        'content-type': 'application/octet-stream' // Protobuf消息的MIME类型
    },
    data: newbuffer, // 发送编码后的Protobuf消息
    success: (res) => {
        console.log('会话成功',res);
        const uint8arr = new Uint8Array(res);

 console.log(wificonfig.WiFiConfigPayload.decode(uint8arr));
    },
    fail: (err) => {
        console.log('会话失败',err)
    }
});

//接着进行wifi信息的配置

// this.wifiInfo.SSID = this.stringToUTF8Array(this.wifiInfo.SSID)
// this.wifiInfo.password = this.stringToUTF8Array(this.wifiInfo.password)
// console.log('WiFi名',this.wifiInfo.SSID)
// console.log('密码',this.wifiInfo.password)

// 创建 WiFi 配置消息对象
const wifimessage =  wificonfig.WiFiConfigPayload.create({
    msg: wificonfig.WiFiConfigMsgType.TypeCmdSetConfig, // 设置 WiFi 配置的枚举值
    cmdSetConfig: {
```

```
        ssid: this.stringToUTF8Array(this.wifiInfo.SSID), //
"Hello Wi-2-2" 的 UTF-8 编码
        passphrase:
this.stringToUTF8Array(this.wifiInfo.password), //
"Password!" 的 UTF-8 编码
    }

});

// // 序列化为 Uint8Array
const wifibuffer =
 wificonfig.WiFiConfigPayload.encode(wifimessage).finish();

// Log the serialized buffer
console.log('wifi信息',wifibuffer);
const newwifibuffer = Uint8Array.from(wifibuffer).buffer;
console.log(newwifibuffer);

// 使用 uni.request 发送 buffer
uni.request({
    url: 'http://192.168.4.1/prov-config', // 替换为设备的实际
IP地址和端点
    method: 'POST',
    header: {
        'content-type': 'application/octet-stream' //
Protobuf 消息的 MIME 类型
    },
    data: newwifibuffer, // 发送编码后的 Protobuf 消息
    success: (res) => {
        console.log('WiFi 配置成功', res);
        const uint8arr = new Uint8Array(res);

 console.log(wificonfig.WiFiConfigPayload.decode(uint8arr));
    },
    fail: (err) => {
        console.log('WiFi 配置失败', err);
    }
});
```

```
// 接着我们需要对配置好的wifi信息进行apply应用

// 发送 ApplyConfig
const applyConfigMessage =
wificonfig.WiFiConfigPayload.create({
  msg: wificonfig.WiFiConfigMsgType.TypeCmdApplyConfig,
  cmdApplyConfig: {} // 空对象即可
});

const applyBuffer =
wificonfig.WiFiConfigPayload.encode(applyConfigMessage).fini
sh();
const applyArrayBuffer =
Uint8Array.from(applyBuffer).buffer;

uni.request({
  url: 'http://192.168.4.1/prov-config',
  method: 'POST',
  header: {
    'content-type': 'application/octet-stream',
  },
  data: applyArrayBuffer,
  success: (res) => {
    console.log('ApplyConfig 成功',res);
    const uint8arr = new Uint8Array(res);

 console.log(wificonfig.WiFiConfigPayload.decode(uint8arr));
    uni.showToast({
      title: '配网完成',
      icon: 'success',
    });
  },
  fail: (err) => {
    console.error('ApplyConfig 失败', err);
    uni.showToast({
      title: '发送 ApplyConfig 失败',
      icon: 'none',
```

```
    });
  }
});
```

设备端调试

- 如果出现了`config not set`的报错，则说明，串行不严格，流程不对，可能`config`信息还没收到，就进行了`apply`的指令下发。

```
2025-06-06 20:00:20 E (45118) wifi_prov_handlers: Wi-Fi
config not set
```

- 客户端如果出现`reset`的报错，这是设备端HTTP服务器`(protocomm/httpd_main.c)`返回的错误，提示请求内容长度超过了4KB限制，因此请求被拒绝，导致连接被重置(ERR CONNECTION RESET)，这是`data`未正确的`encode`的导致的

```
errno:600001,errMsg:"request:fail -101:net::ERR CONNECTION
RESET")
```

- 如果客户端出现了`aborted`的错误，调试可以发现是设备端出现了硬件崩溃`wifi config.c`， 统统都是因为字段有误导致设备对客户端发来的配置信息访问为空。

```
ferrno:600001，errMsg:"request:fail -103:net::ERR CONNECTION
ABORTED"}
```

```
esp err t cmd set config handler(...)
{
    wifi config t *wifi cfg =(wifi config
t*)malloc(sizeof(wifi config t));

     // 这里很可能出现空指针访问或字段未初始化

    strcpy((char*)wificfg->sta.ssid,...); //第161~162 行
}
```

- 在成功得到wifi信息之后，设备端会打印调试日志，此时已经步入配网尾声，此时出现了`STA Disconnected`，那大概率是因为当前要连接到wifi热点是 5GHz，而 ESP32（多数型号）只支持 2.4GHz，如果不是这个问题，那就是wifi信号太弱了捏。

```
25-06-06 19:38:22 I (23868) WIFI: RECEIVED WI-FI
CREDENTIALS:
2025-06-06 19:38:22     SSID: LabWiFi
2025-06-06 19:38:22     PASSWORD: 6666aaaa
2025-06-06 19:38:26 E (27708) wifi_prov_mgr: STA
Disconnected
2025-06-06 19:38:26 E (27708) wifi_prov_mgr: Disconnect
reason : 201
2025-06-06 19:38:26 E (27708) wifi_prov_mgr: STA AP Not
found
2025-06-06 19:38:26 E (27708) WIFI: PROVISIONING FAILED!
```