

HarmonyOS 设备开发入门

更多鸿蒙技术文章、课程、直播，都在 [HarmonyOS社区](#)

版本： 1.0
作者： 连志安
时间： 2020 年 10 月



目录

目录	1
第 1 章 HarmonyOS 介绍.....	1
1.1 鸿蒙系统与 Linux、Android 的不同.....	1
1.2 LiteOS 内核.....	2
1.3 相关资料.....	2
第 2 章 开发环境搭建.....	3
2.1 Linux 环境搭建.....	3
2.2 Windows 访问 ubuntu 文件.....	5
2.3 Windows 环境搭建.....	8
2.4 烧录.....	8
第 3 章 Hi3861 开发.....	8
3.1 编写一个简单的 hello world 程序.....	8
3.2 Hi3861 相关代码结构.....	10
3.3 Hi3861 启动流程.....	12
3.4 Hi3861 AT 指令源码分析，如何添加一条自己的 AT 指令	14
3.5 Hi3861 WiFi 操作，热点连接.....	17
3.6 Hi3861 OLED 驱动.....	21
3.7 Hi3861 实现 APP 一键配网功能.....	24

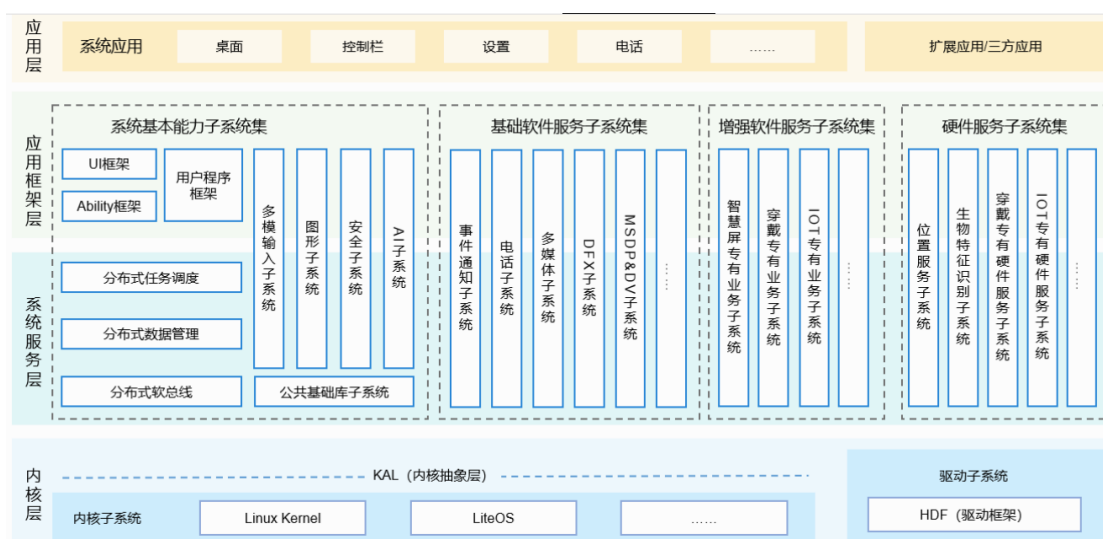
第 1 章 HarmonyOS 介绍

1.1 鸿蒙系统与 Linux、Android 的不同

HarmonyOS 是一款“面向未来”、面向全场景（移动办公、运动健康、社

交通、媒体娱乐等）的分布式操作系统。在传统的单设备系统能力的基础上，HarmonyOS 提出了基于同一套系统能力、适配多种终端形态的分布式理念，能够支持多种终端设备。

HarmonyOS 整体遵从分层设计，从下向上依次为：内核层、系统服务层、框架层和应用层。系统功能按照“系统 > 子系统 > 功能/模块”逐级展开，在多设备部署场景下，支持根据实际需求裁剪某些非必要的子系统或功能/模块。HarmonyOS 技术架构如图所示。



我们可以看到，鸿蒙系统不单单是一个内核，它还包含了整个操作系统的所有框架，更像是 Windows 和 Android。

而鸿蒙系统的内核支持 Linux 和 LiteOS。

1.2 LiteOS 内核

LiteOS 是一个内核，相比其 Linux 来说，它更精简，启动时间更快。同时 liteOS 内核有 liteOS-a 和 liteOS-m 。

liteOS-a 通常运行支持 MMU 的芯片上，支持内核/APP 空间隔离。ARM cotex-A 系列

liteOS-m 运行在没有 MMU 的芯片上，也就是 MCU，例如我们常见的 STM32 芯片。所以鸿蒙 OS 也是支持 STM32 系列单片机的，但是目前还没有完成移植工作。

1.3 相关资料

鸿蒙官方文档: <https://www.harmonyos.com/cn/develop>

鸿蒙 gitee: <https://openharmony.gitee.com/openharmony>

鸿蒙 OS 代码下载:

https://device.harmonyos.com/cn/docs/start/get-code/oem_sourcecode_guide-0000001050769927

第 2 章 开发环境搭建

关于开发环境的搭建,可以参考华为官网说明。

https://device.harmonyos.com/cn/docs/start/introduce/oem_quickstart_3861_build-0000001054781998。

目前鸿蒙系统的开发方式是在 Linux 系统上面编译源码, Windows 系统上编写、烧录。

故而需要搭建两个开发环境。

2.1 Linux 环境搭建

关于 Linux 系统的环境搭建,个人建议使用 ubuntu 20.04。当然我们也提供了搭建好环境的 ubuntu 20.04 镜像,大家可以直接下载,直接编译代码,不需要再按官网的操作再重新搭建环境。

目测个人第一次搭建至少需要几个小时的时间,还可能会出错。

由于百度网盘经常封链接,如果发现链接失效,可以联系我, VX 13510979604

腾讯云盘

链接: <https://share.weiyun.com/6suCAhNN>

百度网盘(以下几个链接,选一个能用的下载就行):

1、链接: <https://pan.baidu.com/s/1sT3ASuqRbh3zH3WFdxw6AA>

提取码: iaap

2、链接: <https://pan.baidu.com/s/1j8jLF0QZmiWhriiwzMPCMg>

提取码: zgew

说明:

- 1、已配置好开发环境，可直接编译代码，编译可以正常运行
- 2、配置好 sftp ，可远程传输文件

账号：harmony

密码：123456

代码路径：~/harmony/code/code-1.0

相关的环境工具路径（可以不用管了，已经配置好了，直接可以编译）：

~/harmony/tools

编译命令：

（1）对应开发板： hi3516 IPC 开发板

python build.py ipcamera_hi3516dv300

（2）对应开发板： hi3518 IPC 开发板

python build.py ipcamera_hi3518ev300

（3）对应开发板： hi3861 智能家居 开发板

python build.py wifiot

编译结果，可以看到已经编译成功了

```
[section sign][31]=[0x48]
[image_id=0x3c78961e][struct_version=0x0]]
[hash_alg=0x0][sign_alg=0x3f][sign_param=0x0]
[section_count=0x1]
[section0_compress=0x1][section0_offset=0x3c0][section0_len=0x69448]
[section1_compress=0x0][section1_offset=0x0][section1_len=0x0]
-----output/bin/Hi3861_wifiot_app_ota.bin image info print end-----
-----
< ~~~~~~>
BUILD SUCCESS
< ~~~~~~>
See build log from: /home/harmony/harmony/code/code-hi3516/vendor/hisi/hi3861/h
i3861/build/build_tmp/logs/build_kernel.log
[197/197] STAMP obj/vendor/hisi/hi3861/hi3861/run_wifiot_scons.stamp
ohos wifiot build success!
harmony@harmony-virtual-machine:~/harmony/code/code-hi3516$
```

```
[1152/1164] SOLINK ./libaudio_api.so
[1153/1164] SOLINK ./libui.so
[1154/1164] LLVM LINK ./bin/abilityMain
[1155/1164] STAMP obj/foundation/aafwk/frameworks/ability_lite/aafwk_abilityMain_lite.stamp
[1156/1164] LLVM LINK dev_tools/bin/aa
[1157/1164] STAMP obj/foundation/graphic/lite/frameworks/ui/liteui.stamp
[1158/1164] STAMP obj/foundation/aafwk/services/abilitymgr_lite/tools/tools_lite.stamp
[1159/1164] STAMP obj/foundation/aafwk/services/abilitymgr_lite/aafwk_services_lite.stamp
[1160/1164] SOLINK ./libace_lite.so
[1161/1164] STAMP obj/foundation/ace/frameworks/lite/jsfwk.stamp
[1162/1164] STAMP obj/build/lite/ohos.stamp
[1163/1164] ACTION //build/lite:gen_rootfs(//build/lite/toolchain:linux_x86_64_clang)
[1164/1164] STAMP obj/build/lite/gen_rootfs.stamp
ohos ipcamera_hi3516dv300 build success!
harmony@harmony-virtual-machine:~/harmony/code/code-hi3516$
harmony@harmony-virtual-machine:~/harmony/code/code-hi3516$
```

2.2 Windows 访问 ubuntu 文件

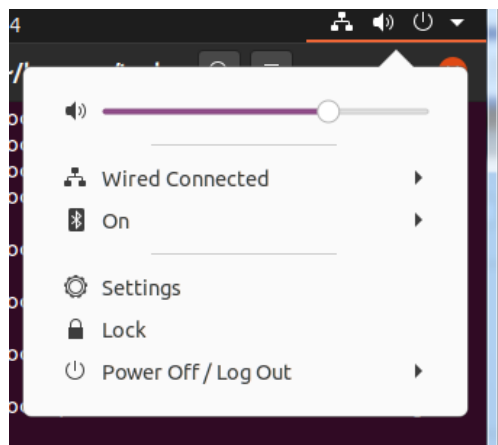
由于我们后面需要在 Windows 上直接编辑 ubuntu 系统里面的鸿蒙源码，故而我们需要使用 samba 服务，让 Windows 能访问到 ubuntu。

操作如下：

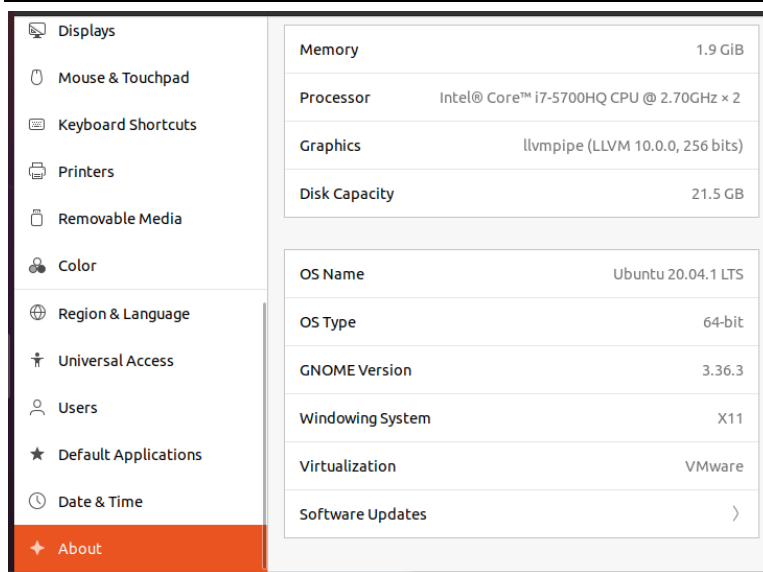
1.设置 apt-get 源

可以更快地下载 samba。设置如下

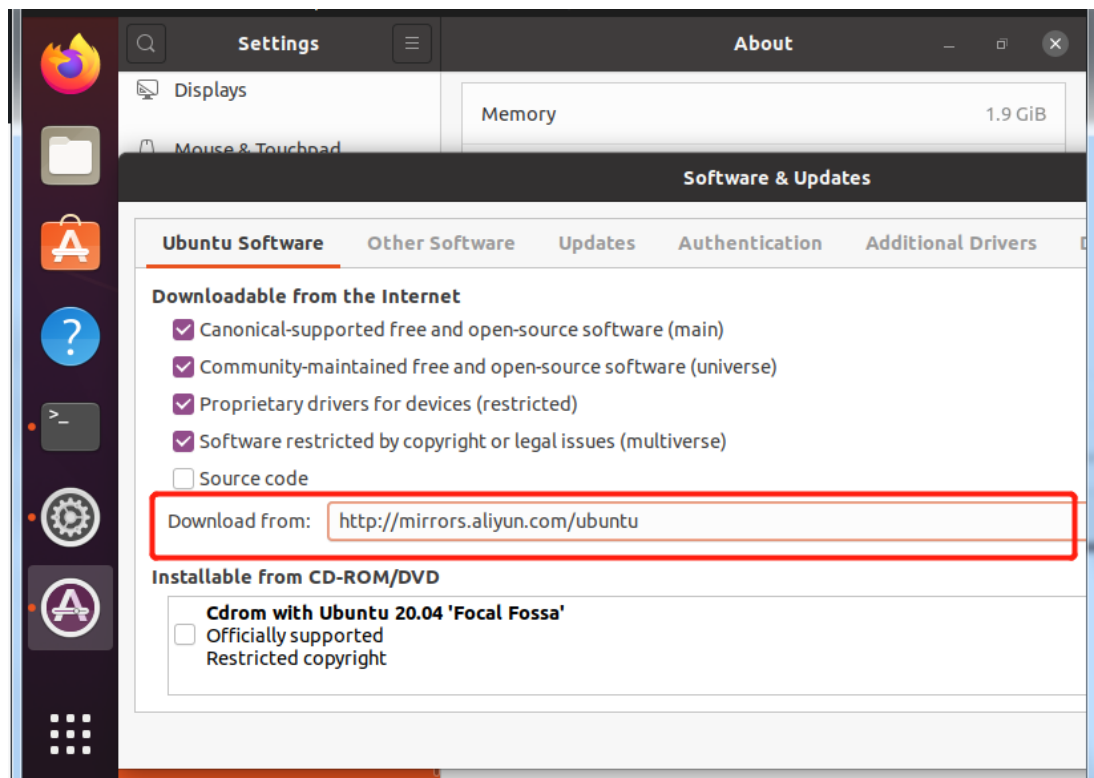
(1) 在桌面右上角点击打开菜单，点击 setting 选项。



(2) 在设置选项右侧下拉找到“关于”，点击 Software Updates。



(3) 在软件和更新界面里可以看到“下载自”，我们可以进行修改。



(4) 推荐选择 [mirros.aliyun.com](http://mirrors.aliyun.com) 或者 mirrors.tuna.tsinghua.edu.cn，你也可以点击选择最佳服务器，测

(5) 试连接最快的软件源（测试时间较长）。

(6) 最后，退出软件与更新界面时，会提示更新软件列表信息，点击重新载入即可。

2.安装 samba

输入如下命令：

```
sudo apt-get install samba
```

```
sudo apt-get install samba-common
```

修改 samba 配置文件

```
sudo vim /etc/samba/smb.conf
```

在最后加入如下内容：

```
[work]
    comment = samba home directory
    path = /home/harmony/
    public = yes
    browseable = yes
    public = yes
    writeable = yes
    read only = no
    valid users = harmony
    create mask = 0777
    directory mask = 0777
    #force user = nobody
    #force group = nogroup
    available = yes
```

保存退出后，输入如下命令，设置 samba 密码，建议 123456 即可

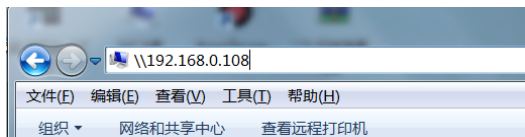
```
sudo smbpasswd -a harmony
```

重启 samba 服务

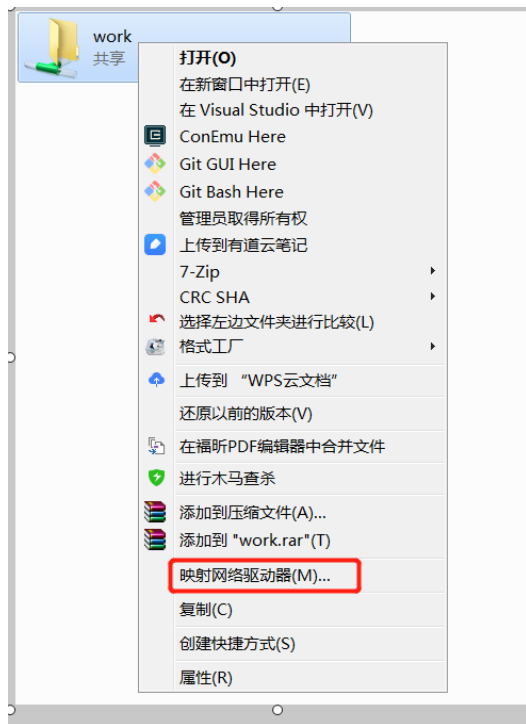
```
sudo service smbd restart
```

3.windows 映射

在文件夹路径输入虚拟机的 IP 地址



最后映射成网络驱动器即可



2.3 Windows 环境搭建

Windows 的环境搭建，官网已经有了，这里就不在赘述。

https://device.harmonyos.com/cn/docs/ide/user-guides/tool_install-0000001050164976

2.4 烧录

烧录也可以参考官方文档：

https://device.harmonyos.com/cn/docs/ide/user-guides/riscv_upload-0000001051668683

第3章 Hi3861 开发

3.1 编写一个简单的 hello world 程序

编写一个 hello world 程序比较简单，可以参考官网：

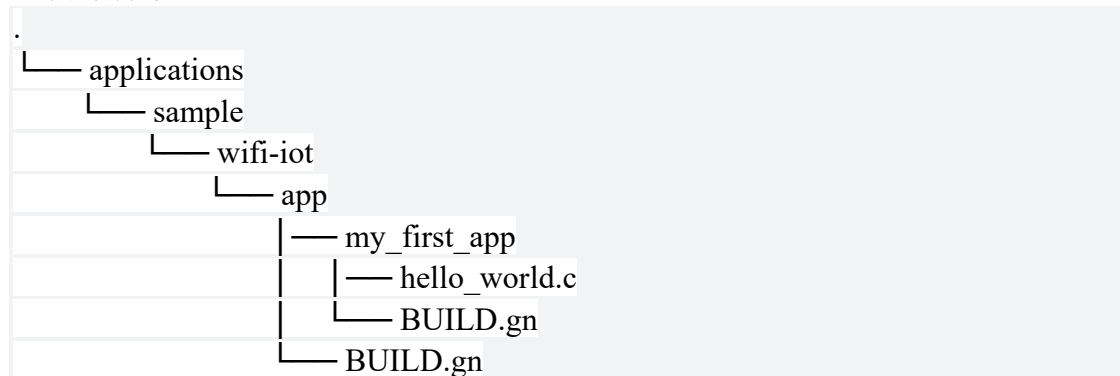
https://device.harmonyos.com/cn/docs/start/introduce/oem_wifi_start-0000001050168544

本文在这里做下总结：

（1）确定目录结构。

开发者编写业务时，务必先在 `./applications/sample/wifi-iot/app` 路径下新建一个目录（或一套目录结构），用于存放业务源码文件。

例如：在 `app` 下新增业务 `my_first_app`，其中 `hello_world.c` 为业务代码，`BUILD.gn` 为编译脚本，具体规划目录结构如下：



（2）编写业务代码。

在 `hello_world.c` 中新建业务入口函数 `HelloWorld`，并实现业务逻辑。并在代码最下方，使用 HarmonyOS 启动恢复模块接口 `SYS_RUN()` 启动业务。（`SYS_RUN` 定义在 `ohos_init.h` 文件中）

```
#include "ohos_init.h"
#include "ohos_types.h"

void HelloWorld(void)
{
    printf("[DEMO] Hello world.\n");
}

SYS_RUN(HelloWorld);
```

（3）编写用于将业务构建成为静态库的 `BUILD.gn` 文件。

如步骤 1 所述，`BUILD.gn` 文件由三部分内容（目标、源文件、头文件路径）构成，需由开发者完成填写。以 `my_first_app` 为例，需要创建 `./applications/sample/wifi-iot/app/my_first_app/BUILD.gn`，并完如下配置。

```
static_library("myapp") {
    sources = [
        "hello_world.c"
    ]
    include_dirs = [
        "../utils/native/liteos/include"
    ]
}
```

`static_library` 中指定业务模块的编译结果，为静态库文件 `libmyapp.a`，开发者根据实际情况完成填写。

`sources` 中指定静态库.a 所依赖的.c 文件及其路径，若路径中包含 `"/"` 则表示绝对路径（此处为代码根路径），若不包含 `"/"` 则表示相对路径。

`include_dirs` 中指定 `source` 所需要依赖的.h 文件路径。

（4）编写模块 `BUILD.gn` 文件，指定需参与构建的特性模块。

配置 `./applications/sample/wifi-iot/app/BUILD.gn` 文件，在 `features` 字段中增加索引，使目标模块参与编译。`features` 字段指定业务模块的路径和目标，以 `my_first_app` 举例，`features` 字段配置如下。

```
import("//build/lite/config/component/lite_component.gni")

lite_component("app") {
    features = [
        "my_first_app:myapp",
    ]
}
```

`my_first_app` 是相对路径，指向 `./applications/sample/wifi-iot/app/my_first_app/BUILD.gn`。

`myapp` 是目标，指向 `./applications/sample/wifi-iot/app/my_first_app/BUILD.gn` 中的 `static_library("myapp")`。

3.2 Hi3861 相关代码结构

目前 hi3861 用的是 `liteos-m` 内核，但是目前 hi3681 的 `liteos-m` 被芯片 rom 化了，固化在芯片内部了。所以在 `harmonyOS` 代码是找不到 hi3861 的内核部分。但是这样不妨碍我们去理清 hi3861 的其他代码结构。

hi3861 平台配置文件

`build\lite\platform\hi3861v100_liteos_riscv\platform.json`

该文件描述了 hi3681 平台相关的代码路径，例如 `application`、`startup` 等。

```
{
  "subsystems": [
    {
      "subsystem": "applications",
      "optional": "true",
      "components": [
        {
          "component": "wifi_iot_sample_app",
          "optional": "true",
          "targets": [
            "//applications/sample/wifi-iot/app"
          ],
          "features": [],
          "deps": {}
        }
      ]
    },
    {
      "subsystem": "startup",
      "optional": "true",
      "components": [
        {
          "component": "syspara",
          "optional": "false",
          "targets": [
            "//base/startup/frameworks/syspara_lite/parameter:parameter"
          ]
        }
      ]
    }
  ]
}
```

这里我列举出来几个比较重要的：

子系统：applications：

路径：applications/sample/wifi-iot/app

作用：这个路径下存放了 hi3681 编写的应用程序代码，例如我们刚刚写得 hello world 代码就放在这个路径下。

子系统：iot_hardware：

路径：base/iot_hardware/frameworks/wifiot_lite

作用：存放了 hi3681 芯片相关的驱动、例如 spi、gpio、uart 等。

子系统：vendor

路径：vendor/hisi/hi3861/hi3861

作用：存放了 hi3681 相关的厂商 SDK 之类的文件。其中最重要的是

vendor/hisi/hi3861/hi3861/app/wifiot_app/init/app_io_init.c

vendor/hisi/hi3861/hi3861/app/wifiot_app/src/app_main.c

其中，app_io_init.c 是 hi3681 内核启动后的 io 口相关设置，用户需根据应用场景，合理选择各外设的 IO 复用配置。

app_main.c 是内核启动进入的应用程序入口。

3.3 Hi3861 启动流程

由于 hi3861 的 liteos-m 被芯片 rom 化了，固化在芯片内部了。所以我们主要看内核启动后的第一个入口函数。

代码路径：

vendor\hisi\hi3861\hi3861\app\wifiot_app\src\app_main.c

```
hi_void app_main(hi_void)
{
#ifdef CONFIG_FACTORY_TEST_MODE
    printf("factory test mode!\r\n");
#endif

    const hi_char* sdk_ver = hi_get_sdk_version();
    printf("sdk ver:%s\r\n", sdk_ver);
    hi_flash_partition_table *ptable = HI_NULL;

    peripheral_init();

    .....中间省略代码

    HOS_SystemInit();
}
```

app_main 一开始打印了 SDK 版本号，最后一行会调用 HOS_SystemInit(); 函数进行鸿蒙系统的初始化。我们进去看下初始化做了哪些动作。

路径：base/startup/services/bootstrap_lite/source/system_init.c

```
void HOS_SystemInit(void)
{
    MODULE_INIT(bsp);
    MODULE_INIT(device);
    MODULE_INIT(core);
}
```

```
SYS_INIT(service);
SYS_INIT(feature);
MODULE_INIT(run);
SAMGR_Bootstrap();
}
```

我们可以看到主要是初始化了一些相关模块、系统，包括有 bsp、device（设备）。其中最终的是 MODULE_INIT(run);

它负责调用了，所有 run 段的代码，那么 run 段的代码是哪些呢？

事实上就是我们前面 application 中使用 SYS_RUN() 宏设置的函数名。

还记得我们前面写的 hello world 应用程序吗？

```
#include "ohos_init.h"
#include "ohos_types.h"

void HelloWorld(void)
{
    printf("[DEMO] Hello world.\n");
}
SYS_RUN(HelloWorld);
```

也就是说所有用 SYS_RUN() 宏设置的函数都会在使用 MODULE_INIT(run); 的时候被调用。

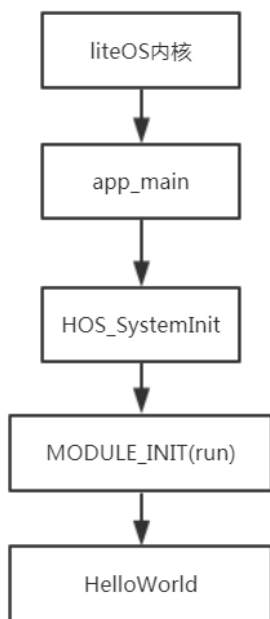
为了验证这一点，我们可以加一些打印信息，如下：

```
00021: void HOS_SystemInit(void)
00022: {
00023:     printf("##### %s %d \r\n", __FILE__, __LINE__);
00024:     MODULE_INIT(bsp);
00025:
00026:     printf("##### %s %d \r\n", __FILE__, __LINE__);
00027:     MODULE_INIT(device);
00028:     printf("##### %s %d \r\n", __FILE__, __LINE__);
00029:     MODULE_INIT(core);
00030:     printf("##### %s %d \r\n", __FILE__, __LINE__);
00031:     SYS_INIT(service);
00032:     printf("##### %s %d \r\n", __FILE__, __LINE__);
00033:     SYS_INIT(feature);
00034:
00035:     printf("##### %s %d \r\n", __FILE__, __LINE__);
00036:     MODULE_INIT(run);
00037:     printf("##### %s %d \r\n", __FILE__, __LINE__);
00038:     SAMGR_Bootstrap();
00039:
00040:     printf("##### %s %d \r\n", __FILE__, __LINE__);
00041: } ? end HOS_SystemInit ?
00042:
```

我们重新编译后烧录。打开串口查看打印信息，如下：

```
____>>>> app/wifiot_app/src/app_main.c 505
____>>>> ../../base/startup/services/bootstrap_lite/source/system_init.c 23
____>>>> ../../base/startup/services/bootstrap_lite/source/system_init.c 26
____>>>> ../../base/startup/services/bootstrap_lite/source/system_init.c 28
____>>>> ../../base/startup/services/bootstrap_lite/source/system_init.c 30
____>>>> ../../base/startup/services/bootstrap_lite/source/system_init.c 32
____>>>> ../../base/startup/services/bootstrap_lite/source/system_init.c 35
[1za][DEMO] Hello world.
____>>>> ../../base/startup/services/bootstrap_lite/source/system_init.c 37
____>>>> ../../base/startup/services/bootstrap_lite/source/system_init.c 40
____>>>> app/wifiot_app/src/app_main.c 508
```

可以看到在 35 行之后，就打印 hello world 的信息。符合预期。



3.4 Hi3861 AT 指令源码分析，如何添加一条自己的 AT 指令

这节主要讲下 hi3861 的 AT 指令相关。先看下 AT 指令在源码中的位置。
上一节已经说到，hi3861 内核启动后的第一个入口函数。

代码路径：

vendor\hisi\hi3861\hi3861\app\wifiot_app\src\app_main.c

hi_void app_main(hi_void)

在 app_main 函数中，会调用 hi_at_init 进行 AT 指令的相关初始化。如果初始化成功，则开始注册各类 AT 指令，代码如下：

```
#if defined(CONFIG_AT_COMMAND) || defined(CONFIG_FACTORY_TEST_MODE)
    ret = hi_at_init();
    if (ret == HI_ERR_SUCCESS) {
        hi_at_sys_cmd_register();
    }
#endif
```

初始化部分暂时先不看，主要是底层相关的。我们重点看下 `hi_at_sys_cmd_register` 注册 AT 指令的函数。

```
hi_void hi_at_sys_cmd_register(hi_void)
{
    printf("____>>>>> %s %d \r\n", __FILE__, __LINE__);

    hi_at_general_cmd_register();
#ifdef CONFIG_FACTORY_TEST_MODE
    hi_at_sta_cmd_register();
    hi_at_softap_cmd_register();
#endif
    hi_at_hipriv_cmd_register();
#ifdef CONFIG_FACTORY_TEST_MODE
#ifdef LOSCFG_APP_MESH
    hi_at_mesh_cmd_register();
#endif
    hi_at_lowpower_cmd_register();
#endif
    hi_at_general_factory_test_cmd_register();
    hi_at_sta_factory_test_cmd_register();
    hi_at_hipriv_factory_test_cmd_register();
    hi_at_io_cmd_register();
}
```

其中，`hi_at_general_cmd_register` 是注册通用指令。代码如下：

```
void hi_at_general_cmd_register(void)
{
    hi_at_register_cmd(g_at_general_func_tbl, AT_GENERAL_FUNC_NUM);
}
```

其实就是把 `g_at_general_func_tbl` 数组的 AT 指令都注册进来。我们可以看到这个数组的内容：

```
const at_cmd_func g_at_general_func_tbl[] = {
    {"", 0, HI_NULL, HI_NULL, HI_NULL, (at_call_back_func)at_exe_at_cmd},
    {"+RST", 4, HI_NULL, HI_NULL, (at_call_back_func)at_setup_reset_cmd, (at_call_back_func)at_exe_reset_cmd},
    {"+MAC", 4, HI_NULL, (at_call_back_func)cmd_get_macaddr, (at_call_back_func)cmd_set_macaddr, HI_NULL},
    {"+HELP", 5, HI_NULL, HI_NULL, HI_NULL, (at_call_back_func)at_exe_help_cmd},

#ifdef CONFIG_FACTORY_TEST_MODE
    {"+SYSINFO", 8, HI_NULL, HI_NULL, HI_NULL, (at_call_back_func)at_query_sysinfo_cmd},

```

g_at_general_func_tbl 的结构体原型如下：

```
typedef struct {
    //AT 指令命。前面省略 AT

    hi_char *at_cmd_name;

    //指令的长度

    hi_s8    at_cmd_len;

    //at 测试时调用的回调函数

    at_call_back_func at_test_cmd;

    //at 查询时调用的回调函数

    at_call_back_func at_query_cmd;

    //at 设置时调用的回调函数

    at_call_back_func at_setup_cmd;

    //at 运行时调用的回调函数

    at_call_back_func at_exe_cmd;

} at_cmd_func;
```

看到这个数组，聪明的朋友应该知道怎么增加第一条属于自己的指令了吧~~~~

(1) 增加 AT 指令

```
const at_cmd_func g_at_general_func_tbl[] = {
    {"", 0, HI_NULL, HI_NULL, HI_NULL, (at_call_back_func)at_exe_at_cmd},
    {"+RST", 4, HI_NULL, HI_NULL, (at_call_back_func)at_setup_reset_cmd, (at_call_back_func)at_exe_reset_cmd},
    {"+MYTEST", 7, at_test_mytest_cmd, at_query_mytest_cmd, at_setup_mytest_cmd, at_exe_mytest_cmd},
    {"+MAC", 4, HI_NULL, (at_call_back_func)cmd_get_macaddr, (at_call_back_func)cmd_set_macaddr, HI_NULL},
    {"+HELP", 5, HI_NULL, HI_NULL, HI_NULL, (at_call_back_func)at_exe_help_cmd},

```

(2) 完善相关函数：

hi_u32 at_setup_mytest_cmd(hi_s32 argc, const hi_char *argv[])

```
{
    hi_at_printf("at_setup_mytest_cmd \r\n");

    return HI_ERR_SUCCESS;
}
```

hi_void at_exe_mytest_cmd(hi_s32 argc, const hi_char *argv[])


```
{  
    hi_at_printf("at_exe_mytest_cmd \r\n");  
    return HI_ERR_SUCCESS;  
}  
  
hi_u32 at_query_mytest_cmd(hi_s32 argc, const hi_char* argv[])  
{  
    hi_at_printf("at_query_mytest_cmd \r\n");  
    return HI_ERR_SUCCESS;  
}  
  
hi_u32 at_test_mytest_cmd(hi_s32 argc, const hi_char* argv[])  
{  
    hi_at_printf("at_test_mytest_cmd \r\n");  
    return HI_ERR_SUCCESS;  
}
```

编译后我们开始测试：

发送：AT+MYTEST

接收：at_exe_mytest_cmd

ERROR

发送：AT+MYTEST?

接收：at_query_mytest_cmd

发送：AT+MYTEST=1

接收：at_setup_mytest_cmd

3.5 Hi3861 WiFi 操作，热点连接

之前我们使用 Hi3861 的时候，是使用 AT 指令连接到 WiFi 热点的。例如：

- | | |
|---------------------------------|---------------------------------------|
| 1. AT+STARTSTA | - 启动STA模式 |
| 2. AT+SCAN | - 扫描周边AP |
| 3. AT+SCANRESULT | - 显示扫描结果 |
| 4. AT+CONN="SSID",,2,"PASSWORD" | - 连接指定AP, 其中SSID/PASSWORD为待连接的热点名称和密码 |
| 5. AT+STASTAT | - 查看连接结果 |
| 6. AT+DHCP=wlan0,1 | - 通过DHCP向AP请求wlan0的IP地址 |

查看WLAN模组与网关联通是否正常, 如下图所示。

- | | |
|--------------------|-------------------------------------|
| 1. AT+IFCFG | - 查看模组接口IP |
| 2. AT+PING=X.X.X.X | - 检查模组与网关的连通性, 其中X.X.X.X需替换为实际的网关地址 |

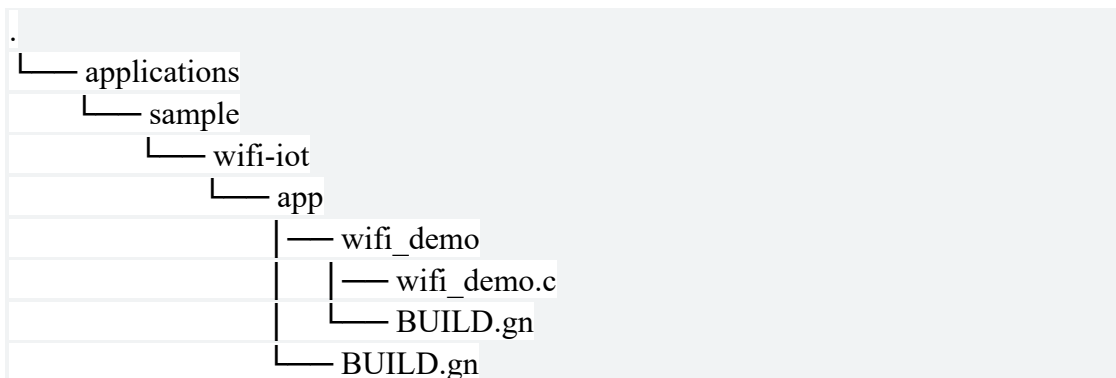
但是很多时候, 我们需要实现开机后自动连接到某个热点, 光靠 AT 指令不行。

Hi3861 为我们提供了 WiFi 操作的相关 API, 方便我们编写代码, 实现热点连接。

1.代码实现

先直接上代码和操作演示。

跟我们最早的 hello world 代码一样, 在 app 下新增业务 wifi_demo, 其中 hello_world.c 为业务代码, BUILD.gn 为编译脚本, 具体规划目录结构如下:



Wifi_demo.c 代码如下:

见附件 doc\05 WiFi 操作\sta_demo\sta_demo.c

Wifi_demo 目录下的 BUILD.gn 文件内容如下:

```
static_library("wifi_demo") {
    sources = [
        "wifi_demo.c"
    ]
}
```


```
include_dirs = [  
    "//utils/native/lite/include",  
    "//kernel/liteos_m/components/cmsis/2.0",  
    "//base/iot_hardware/interfaces/kits/wifiot_lite",  
    "//vendor/hisi/hi3861/hi3861/third_party/lwip_sack/include",  
    "//foundation/communication/interfaces/kits/wifi_lite/wifiservice",  
  
]  
}
```

app 目录下的 BUILD.gn 文件内容修改如下：

```
import("//build/lite/config/component/lite_component.gni")
```

```
lite_component("app") {  
    features = [  
        "wifi_demo:wifi_demo",  
    ]  
}
```

编译烧录，查看串口数据：



```
+NOTICE:SCANFINISH  
WiFi: Scan results available  
SSID: 15919500  
SSID: Netcore_FD55A7  
.../applications/sample/wifi-iot/app/wifi_demo/wifi_demo.c 94  
.../base/startup/services/bootstrap_lite/source/system_init.c 40  
.../base/startup/services/bootstrap_lite/source/system_init.c 43  
app/wifiot_app/src/app_main.c 502  
00 00:00:00 0 132 D 0/HIVIEW: hilog init success.  
00 00:00:00 0 132 D 0/HIVIEW: log limit init success.  
00 00:00:00 0 132 I 1/SAMGR: Bootstrap core services(count:3).  
00 00:00:00 0 132 I 1/SAMGR: Init service:0x4ae97c TaskPool:0xf9ecc  
00 00:00:00 0 132 I 1/SAMGR: Init service:0x4ae9ec TaskPool:0xf9eec  
00 00:00:00 0 132 I 1/SAMGR: Init service:0x4aea9c TaskPool:0xf9eac  
00 00:00:00 0 164 I 1/SAMGR: Init service 0x4ae9ec <time: 5570ms> success!  
00 00:00:00 0 220 I 1/SAMGR: Init service 0x4ae97c <time: 5570ms> success!  
00 00:00:00 0 8 D 0/HIVIEW: hiview init success.  
00 00:00:00 0 8 I 1/SAMGR: Init service 0x4aea9c <time: 5570ms> success!  
00 00:00:00 0 8 I 1/SAMGR: Initialized all core system services!  
00 00:00:00 0 220 I 1/SAMGR: Bootstrap system and application services(count:0).  
00 00:00:00 0 220 I 1/SAMGR: Initialized all system and application services!  
00 00:00:00 0 220 I 1/SAMGR: Bootstrap dynamic registered services(count:0).  
WiFi: Connected
```

可以看到有打印扫描到的热点名称：

SSID: 15919500

SSID: Netcore_FD55A7

同时最后打印：WiFi: Connected 成功连接上热点。

2.wifi api 接口说明

Hi3861 提供了非常多的 wifi 相关 API，主要文件是 hi_wifi_api.h

我们这里只列举最重要的几个 API

（1）开启 STA

```
int hi_wifi_sta_start(char *ifname, int *len);
```

（2）停止 STA

```
int hi_wifi_sta_stop(void);
```

（3）扫描附件的热点

```
int hi_wifi_sta_scan(void);
```

（4）连接热点

```
int hi_wifi_sta_connect(hi_wifi_assoc_request *req);
```

其中 hi_wifi_assoc_request *req 结构的定义如下：

```
typedef struct {
    char ssid[HI_WIFI_MAX_SSID_LEN + 1];    /**< SSID. CNcomment: SSID 只支持ASCII字符.CNend */
    hi_wifi_auth_mode auth;                  /**< Authentication mode. CNcomment: 认证类型.CNend */
    char key[HI_WIFI_MAX_KEY_LEN + 1];       /**< Secret key. CNcomment: 秘钥.CNend */
    unsigned char bssid[HI_WIFI_MAC_LEN];    /**< BSSID. CNcomment: BSSID.CNend */
    hi_wifi_pairwise pairwise;               /**< Encryption type. CNcomment: 加密方式,不需指定时置0.CNend */
} hi_wifi_assoc_request;
```

这里需要注意的是，通常加密方式是：HI_WIFI_SECURITY_WPA2PSK

例如我家的热点的连接方式的代码实现如下：

```
/* copy SSID to assoc_req */
//热点名称
rc = memcpy_s(assoc_req.ssid, HI_WIFI_MAX_SSID_LEN + 1, "15919500", 8); /* 9:ssid length */
if (rc != EOK) {
    printf("%s %d \r\n", __FILE__, __LINE__);
    return -1;
}

/*
 * OPEN mode
 * for WPA2-PSK mode:
 * set assoc_req.auth as HI_WIFI_SECURITY_WPA2PSK,
 * then memcpy(assoc_req.key, "12345678", 8).
 */
//热点加密方式
assoc_req.auth = HI_WIFI_SECURITY_WPA2PSK;

/* 热点密码 */
memcpy(assoc_req.key, "11206582488", 11);
```

3.6 Hi3861 OLED 驱动

Hispark WiFi 开发套件又提供一个 OLED 屏幕，但是鸿蒙源码中没有这个屏幕的驱动，我们需要自己去移植。



经过一晚上的调试，现在终于在鸿蒙系统上实现 OLED 屏幕的显示了，效果如下：



这里记录一下移植的过程

(1) 编写驱动代码

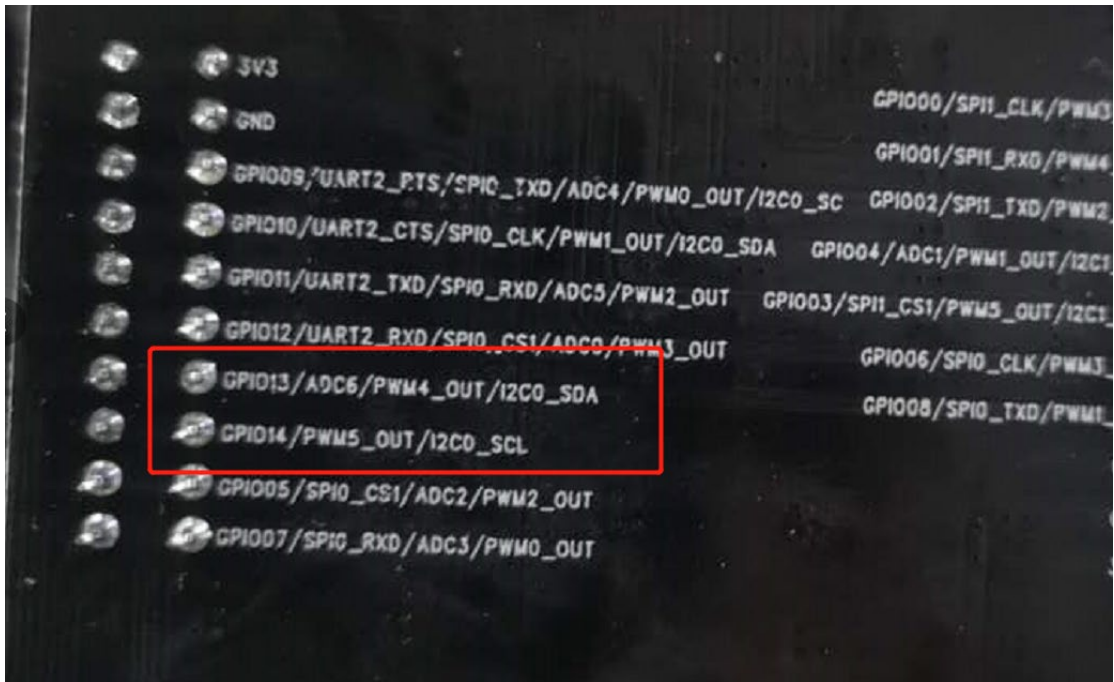
首先在

```
└─ applications
    └─ sample
        └─ wifi-iot
            └─ app
```

新增应用：oled_demo，源码已经放在附件，大家自己下载。

(2) 设置 I2C 引脚复用

确定 i2c 引脚，查看原理图，可以看到 OLED 屏幕使用的是 I2C0，引脚是 GPIO13、GPIO14



所以我们需要修改源码，在

vendor\hisi\hi3861\hi3861\app\wifiot_app\init\app_io_init.c 文件中，初始化 I2C 引脚的代码修改成如下：

```
#ifdef CONFIG_I2C_SUPPORT
/* I2C IO 复用也可以选择 3/4; 9/10, 根据产品设计选择 */
hi_io_set_func(HI_IO_NAME_GPIO_13, HI_IO_FUNC_GPIO_13_I2C0_SDA);
hi_io_set_func(HI_IO_NAME_GPIO_14, HI_IO_FUNC_GPIO_14_I2C0_SCL);
#endif
```

(3) 开启 I2C 功能

修改文件：vendor\hisi\hi3861\hi3861\build\config\usr_config.mk

增加 CONFIG_I2C_SUPPORT=y

以上修改变完成了，重新编译即可看到 OLED 能成功驱动。

(4) OLED 屏幕驱动讲解

入口函数：

```
void my_oled_demo(void)
```

```
{
```

```
    //初始化，我们使用的是 I2C0
```

```
hi_i2c_init(HI_I2C_IDX_0, 100000); /* baudrate: 100000 */

led_init();

OLED_ColorTurn(0); //0 正常显示, 1 反色显示
OLED_DisplayTurn(0); //0 正常显示 1 屏幕翻转显示

OLED_ShowString(8,16,"hello world",16);

OLED_Refresh();
}

I2C 写函数:
hi_u32 my_i2c_write(hi_i2c_idx id, hi_u16 device_addr, hi_u32 send_len)
{
    hi_u32 status;
    hi_i2c_data es8311_i2c_data = { 0 };

    es8311_i2c_data.send_buf = g_send_data;
    es8311_i2c_data.send_len = send_len;
    status = hi_i2c_write(id, device_addr, &es8311_i2c_data);
    if (status != HI_ERR_SUCCESS) {
        printf("==== Error: I2C write status = 0x%x! =====\r\n",
status);

        return status;
    }

    return HI_ERR_SUCCESS;
}
```

3.7 Hi3861 实现 APP 配网功能

本节主要讲如何去实现 Hi3861 配网功能。本节知识有点多，包括 Hi3861 的 WiFi 操作，AP 模式、STA 模式、按键功能、网络编程、JSON 数据格式、手机 APP。

所有源码，还有手机 APP 均提供下载，大家自领。

也可以直接观看视频。

先上原理：

目前主流的 WIFI 配置模式有以下 2 种：

1、智能硬件处于 AP 模式（类似路由器，组成局域网），手机用于 STA 模式

手机连接到处于 AP 模式的智能硬件后组成局域网,手机发送需要连接路由的 SSID 及密码至智能硬件，智能硬件主动去连接指定路由后,完成配网

2、一键配网(smartConfig)模式

智能硬件处于混杂模式下,监听网络中的所有报文;手机 APP 将 SSID 和密码编码到 UDP 报文中,通过广播包或组播报发送,智能硬件接收到 UDP 报文后解码,得到正确的 SSID 和密码,然后主动连接指定 SSID 的路由完成连接。

本文主要讲如何实现第一种 AP 方式。

AP 是 (Wireless) Access Point 的缩写，即 (无线) 访问接入点。简单来讲就像是无线路由器一样，设备打开后进入 AP 模式，在手机的网络列表里面，可以搜索到类似 TPLINK_XXX 的名字（SSID）。

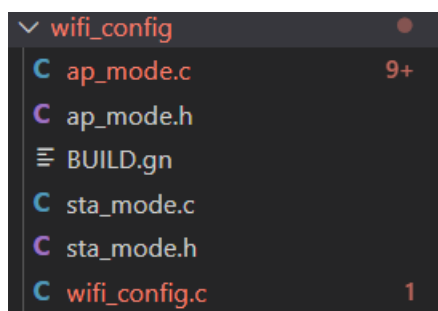
连接步骤：

- 1、Hi3861 上面有一个 user 按键，用户可以按下这个按钮，Hi386 会进入 AP 模式
- 2、手机扫描 WIFI 列表：扫描到 Hi3861 的 SSID（目前是“Hispark-WiFi-IoT”）连接该智能硬件设备，通过手机 APP 发生 我们要连接的热点的 ssid 和密码
- 3、智能硬件设备通过 UDP 包获取配置信息，切换网络模式连接 WIFI 后配网完成

代码实现

（1）代码结构

代码主要由 3 个文件组成



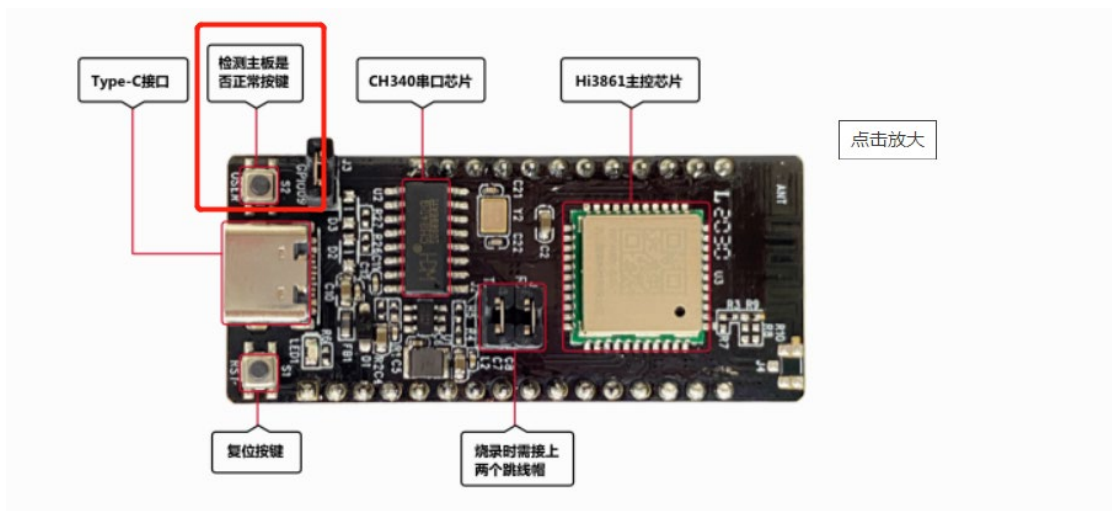
ap_mode.c: 主要实现 AP 模式，并实现一个简单的 UDP 服务器，获取手机 APP 传输过来的热点账号和密码。

sta_mode.c: 实现连接配网的功能。

wifi_config.c: 入口函数，实现按下按键后开始配网的功能。

(2) 按键功能实现

通过查阅原理图，我们可以看到 Hi3861 在 type-C 口附近有一个 user 按钮，如图，主要不要和复位按钮搞错了。user 按钮对应的是 GPIO5 引脚。



于是我们可以使用按键中断编程的方式去实现，代码如下：

```
/* 设置 按键中断响应 */
hi_void my_gpio_isr_demo(hi_void)
{
    hi_u32 ret;

    printf("----- gpio isr demo -----r\n");

    (hi_void)hi_gpio_init();

    hi_io_set_func(HI_IO_NAME_GPIO_5, HI_IO_FUNC_GPIO_5_GPIO); /* uart1 rx */

    ret = hi_gpio_set_dir(HI_GPIO_IDX_5, HI_GPIO_DIR_IN);
    if (ret != HI_ERR_SUCCESS) {
        printf("===== ERROR =====gpio -> hi_gpio_set_dir1 ret:%d\r\n", ret);
        return;
    }

    ret = hi_gpio_register_isr_function(HI_GPIO_IDX_5, HI_INT_TYPE_EDGE,
                                       HI_GPIO_EDGE_RISE_LEVEL_HIGH, my_gpio_isr_func, HI_NULL);
    if (ret != HI_ERR_SUCCESS) {
        printf("===== ERROR =====gpio -> hi_gpio_register_isr_function ret:%d\r\n", ret);
    }
}
```

其中需要主要的是需要使用 `hi_io_set_func(HI_IO_NAME_GPIO_5, HI_IO_FUNC_GPIO_5_GPIO);` 修改 GPIO5 为普通引脚, 否则 GPIO5 默认会被初始化为 串口引脚, 导致无法使用。

GPIO5 中断回调函数如下:

```
/* gpio callback func */
hi_void my_gpio_isr_func(hi_void *arg)
{
    hi_unref_param(arg);
    printf("----- gpio isr success -----\\r\\n");

    //启动配网功能
    start_wifi_config_flg = 1;
}
```

其实很简单, 就是置某个变量为 1 而已。

(3) 接下来进入 AP 模式

代码如下, 一旦发现 `start_wifi_config_flg` 不为 0, 也就是说发生了按键被按下的事件, 那就会调用 `wifi_start_softap` 函数进入 AP 模式

```
void *wifi_config_thread(const char *arg)
{
    arg = arg;

    my_gpio_isr_demo();

    while(start_wifi_config_flg == 0)
    {
        usleep(300000);
    }

    printf("wifi_start_softap \\r\\n");
    wifi_start_softap();

    osThreadExit();
    return NULL;
}
```

(4) AP 模式

AP 模式的代码部分也很简单，首先我们要先设置好 Hi3861 AP 模式下的 SSID，然后开放网络，不加密。对应的函数是 `wifi_start_softap`

```
rc = memcpy_s(hapd_conf.ssid, HI_WIFI_MAX_SSID_LEN + 1, "Hispark-WiFi-IoT", 16); /* 9:ssid length */
if (rc != EOK) {
    return -1;
}

hapd_conf.authmode = HI_WIFI_SECURITY_OPEN;
hapd_conf.channel_num = 1;

ret = hi_wifi_softap_start(&hapd_conf, ifname, &len);
if (ret != HISI_OK) {
    printf("hi_wifi_softap_start\n");
    return -1;
}
```

接下来设置好 Hi3861 的网段、IP 等，并开启 UDP 服务：

```
IP4_ADDR(&st_gw, 192, 168, 10, 1); /* input your IP for example: 192.168.1.1 */
IP4_ADDR(&st_ipaddr, 192, 168, 10, 1); /* input your netmask for example: 192.168.1.1 */
IP4_ADDR(&st_netmask, 255, 255, 255, 0); /* input your gateway for example: 255.255.255.0 */
netifapi_netif_set_addr(g_lwip_netif, &st_ipaddr, &st_netmask, &st_gw);

netifapi_dhcp_start(g_lwip_netif, 0, 0);

udp_thread();
```

(5) UDP 服务器

UDP 服务器绑定的端口号是 50001，使用 socket 通信接口

```
void udp_thread(void)
{
    int ret;
    struct sockaddr_in servaddr;
    cJSON *recvjson;

    int sockfd = socket(PF_INET, SOCK_DGRAM, 0);

    //服务器 ip port
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(50001);

    bind(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
```

绑定完端口号后，进入接收数据

```
memset(recvline, sizeof(recvline), 0);
ret = recvfrom(sockfd, recvline, 1024, 0, (struct sockaddr*)&addrClient, (&sizeClientAddr));

if(ret>0)
{
    char *pClientIP =inet_ntoa(addrClient.sin_addr);

    printf("%s-%d(%d) says:%s\n",pClientIP,ntohs(addrClient.sin_port),addrClient.sin_port, recvline);

    //进行json解析
    recvjson = cJSON_Parse(recvline);

    printf("ssid : %s\r\n", cJSON_GetObjectItem(recvjson, "ssid")->valuestring);
    printf("passwd : %s\r\n", cJSON_GetObjectItem(recvjson, "passwd")->valuestring);

    memset(ssid, sizeof(ssid), 0);
    memset(passwd, sizeof(passwd), 0);

    strcpy(ssid, cJSON_GetObjectItem(recvjson, "ssid")->valuestring);
    strcpy(passwd, cJSON_GetObjectItem(recvjson, "passwd")->valuestring);

    cJSON_Delete(recvjson);

    //先停止AP模式
    wifi_stop_softap();

    //启动STA模式
    start_sta_connect(ssid, strlen(ssid), passwd, strlen(passwd));
}
```

数据这里我使用 json 格式，由于鸿蒙系统代码中已经自带 cJSON 库，可以直接使用，这一部分的代码也比较简单，大家可以看看。

（6）开启 STA 模式

启动 STA 模式的代码部分也比较简单，我之前有一篇文章有讲，具体代码如下：

```
113 //wifi模块初始化，刚刚我们推出了AP模式。需要重新初始化
114 ret = hi_wifi_init(wifi_vap_res_num, wifi_user_res_num);
115 if (ret != HISI_OK) {
116     printf("%s %d \r\n", __FILE__, __LINE__);
117     //return -1;
118 }
119
120 //启动STA模式
121 ret = hi_wifi_sta_start(ifname, &len);
122 if (ret != HISI_OK) {
123     printf("%s %d \r\n", __FILE__, __LINE__);
124     return;
125 }
126
127 /* 注册wifi事件回调函数，如果成功连接上热点，会有打印信息
128 */
129 ret = hi_wifi_register_event_callback(wifi_wpa_event_cb);
130 if (ret != HISI_OK) {
131     printf("register wifi event callback failed\n");
132 }
133
134 /* acquire netif for IP operation */
135 g_lwip_netif = netifapi_netif_find(ifname);
136 if (g_lwip_netif == NULL) {
137     printf("%s: get netif failed\n", __FUNCTION__);
138     return ;
139 }
140
141 /* 开始进行热点连接 */
142 ret = hi_wifi_start_connect(ssid, ssid_len, passwd, passwd_len);
143 if (ret != 0) {
144     printf("%s %d \r\n", __FILE__, __LINE__);
145     return ;
146 }
```

关键代码已经做了注释。

(7) 连接热点

连接热点时，只需要传入 ssid、加密方式和密码即可。

需要主要的地方是我们通常的 wifi 加密都是 HI_WIFI_SECURITY_WPA2PSK

```
66 int hi_wifi_start_connect(char *ssid, int ssid_len, char *passwd, int passwd_len)
67 {
68     int ret;
69     errno_t rc;
70     hi_wifi_assoc_request assoc_req = {0};
71
72     /* copy SSID to assoc_req */
73     //热点名称
74     rc = memcpy_s(assoc_req.ssid, HI_WIFI_MAX_SSID_LEN + 1, ssid, ssid_len); /* 9:ssid length */
75     if (rc != EOK) {
76         printf("%s %d \r\n", __FILE__, __LINE__);
77         return -1;
78     }
79
80     /*
81      * OPEN mode
82      * for WPA2-PSK mode:
83      * set assoc_req.auth as HI_WIFI_SECURITY_WPA2PSK,
84      * then memcpy(assoc_req.key, "12345678", 8).
85      */
86     //热点加密方式
87     assoc_req.auth = HI_WIFI_SECURITY_WPA2PSK;
88
89     /* 热点密码 */
90     memcpy(assoc_req.key, passwd, passwd_len);
91
92
93     ret = hi_wifi_sta_connect(&assoc_req);
94     if (ret != HISI_OK) {
95         printf("%s %d \r\n", __FILE__, __LINE__);
96         return -1;
97     }
98     printf("%s %d \r\n", __FILE__, __LINE__);
99     return 0;
100 }
```

作者简介: 连志安, 旗点云科技创始人, 广东长虹技术研究所(国企)Android 南美 Android 软件负责人。之前在康佳集团(国企)、CVTE(上市公司)等公司任职。负责过 Android TV、智能网关、路由器、智能家居、安防报警器等项目。熟悉单片机、嵌入式 linux、服务器、Android 系统、手机 APP 开发等。

更多鸿蒙技术文章、课程、直播, 都在 [HarmonyOS社区](#)

