OpenHarmony 设备开发第一版

连志安

目录

第 1 章 初始 OpenHarmony

摘要：本文简单介绍 OpenHarmony、轻量系统、小型系统、标准系统的差异，以及相关的官方资料和文档

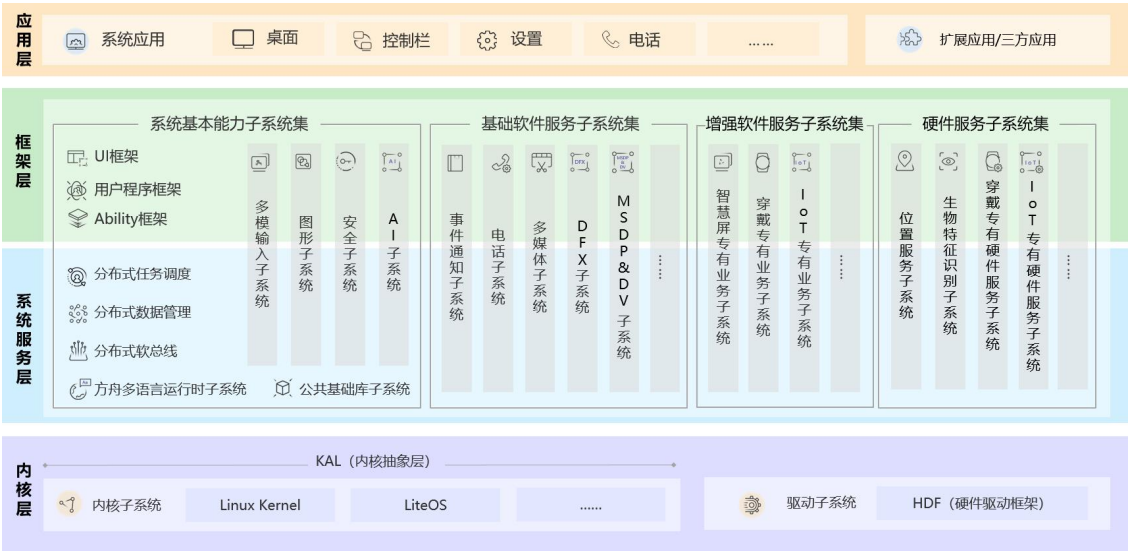
适合群体：第一次接触 OpenHarmony、或者想要对 OpenHarmony 有一个全面的认知的。

1.1 系统类型

OpenHarmony 是由开放原子开源基金会（OpenAtom Foundation）孵化及运营的开源项目，目标是面向全场景、全连接、全智能时代，基于开源的方式，搭建一个智能终端设备操作系统的框架和平台，促进万物互联产业的繁荣发展。

官方 gitee 仓库：https://gitee.com/openharmony

技术架构如下：



关于系统的特性，这里不再过多赘述，开发者可以直接在 官方 gitee 仓库：https://gitee.com/openharmony 中查看。特别是内核层的多内核设计、HDF 驱动框架、分布式能力等。

1.2 系统类型

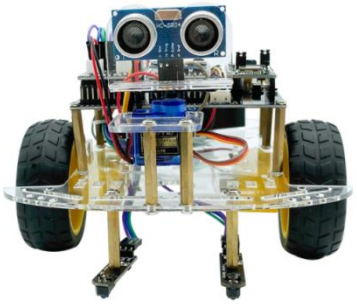
OpenHarmony 是一个面向全场景，支持各类设备的系统。这里的设备就包括像 MCU 单片机这样资源较少的芯片，也支持像 RK3568 这样的多核 CPU。

为了能适应各种硬件，OpenHarmony 提供了像 LiteOS、Linux 这样的不同内核，并基于这些内核形成了不同的系统类型，同时又在这些系统中构建了一套统一的系统能力。

总体来说，目前 OpenHarmony 主要有 3 种系统类型：L0（又称轻量系统）、L1（小型系统）、L2（标准系统）。

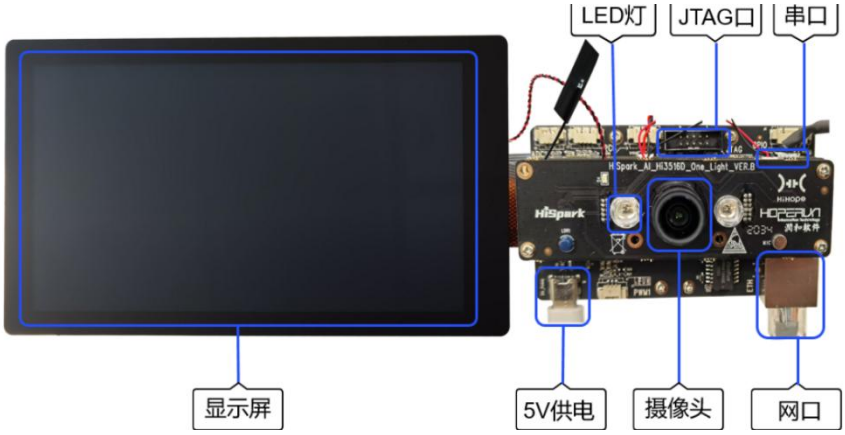
（1）轻量系统（mini system）

面向 MCU 类处理器例如 Arm Cortex-M、RISC-V 32 位的设备，硬件资源极其有限，支持的设备最小内存为 128KiB，可以提供多种轻量级网络协议，轻量级的图形框架，以及丰富的 IOT 总线读写部件等。可支撑的产品如智能家居领域的连接类模组、传感器设备、穿戴类设备等。典型的设备、开发板有 HI3861 鸿蒙小车、Neptune 开发板，如下：



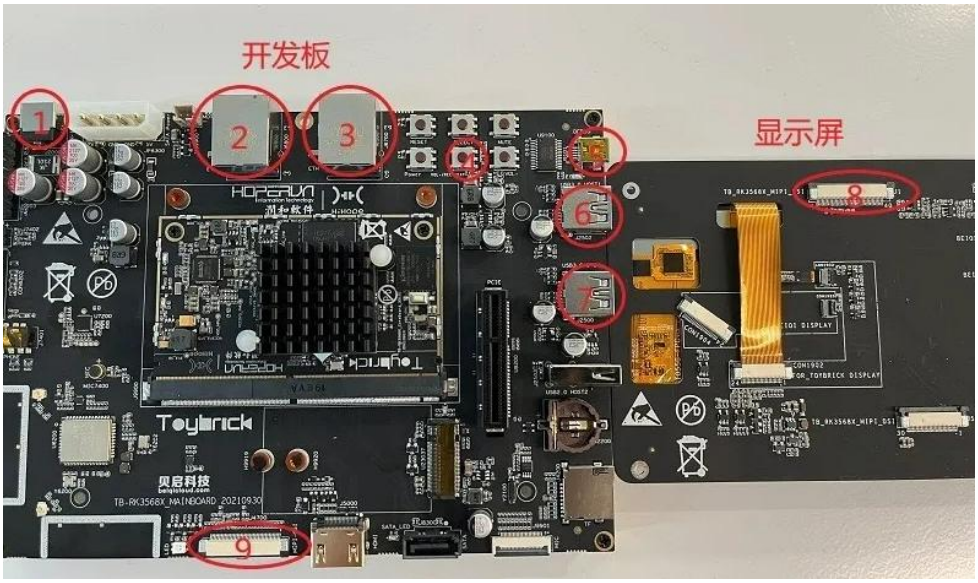
（2）小型系统（small system）

面向应用处理器例如 Arm Cortex-A 的设备，支持的设备最小内存为 1MiB，可以提供更高的安全能力、标准的图形框架、视频编解码的多媒体能力。可支撑的产品如智能家居领域的 IP Camera、电子猫眼、路由器以及智慧出行域的行车记录仪等。典型的开发板有 AI Camera 开发板，如下：



（3）标准系统（standard system）

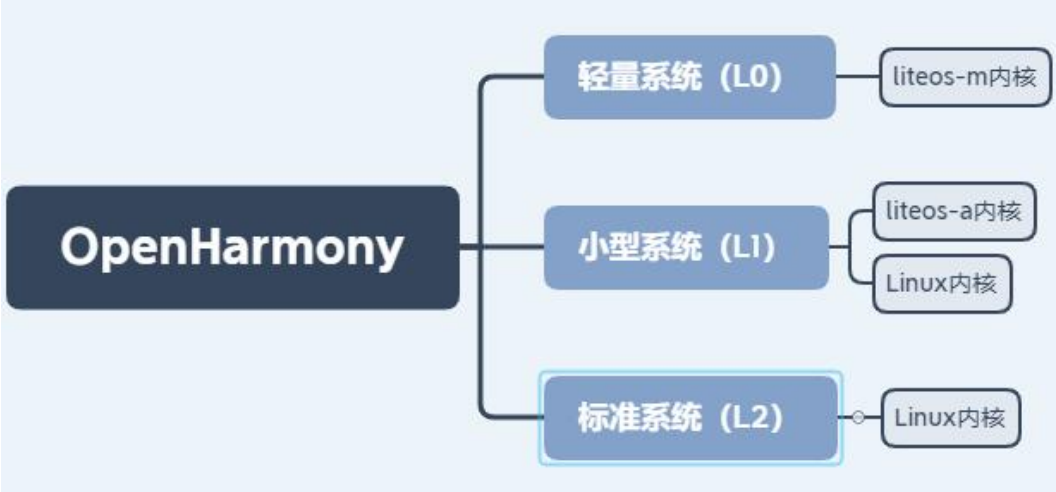
面向应用处理器例如 Arm Cortex-A 的设备，支持的设备最小内存为 128MiB，可以提供增强的交互能力、3D GPU 以及硬件合成能力、更多控件以及动效更丰富的图形能力、完整的应用框架。可支撑的产品如高端的冰箱显示屏。典型的设备有大禹 200 开发板，如下：



1.3 内核类型

OpenHarmony 支持多种内核，目前已适配的内核有 liteos-m、liteos-a、Linux（有 4.19 和 5.10 版本）。

内核与系统类型的对应关系如下图：



轻量系统目前适配了 liteos-m 内核，该内核对硬件资源较少，适用于单片机。

小型系统目前适配了 liteos-a 和 Linux 2 种内核，开发者可以选择合适的内核进行产品开发。

标准系统目前适配了 Linux 内核，开发者可以基于 linux kernel 演进。

1.4 系统差异

轻量系统、小型系统、标准系统的差异主要体现在子系统支持程度上，本文已列出如下，但是建议读者直接查看官网，以便获取最新的特性支持情况。

子系 统	简 介	适用范围
内核	支持适用于嵌入式设备及资源受限设备，具有小体积、高性能、低功耗等特征的 LiteOS 内核；支持基于 linux kernel 演进的适用于标准系统的 linux 内核。	小型系统

		标准系统
分布式文件	提供本地同步 JS 文件接口。	标准系统
图形	主要包括 UI 组件、布局、动画、字体、输入事件、窗口管理、渲染绘制等模块，构建基于轻量 OS 应用框架满足硬件资源较小的物联网设备或者构建基于标准 OS 的应用框架满足富设备（如平板和轻智能机等）的 OpenHarmony 系统应用开发。	所有系统
驱动	OpenHarmony 驱动子系统采用 C 面向对象编程模型构建，通过平台解耦、内核解耦，兼容不同内核，提供了归一化的驱动平台底座，旨在为开发者提供更精准、更高效的开发环境，力求做到一次开发，多系统部署。	所有系统
电源管理服务	电源管理服务子系统提供如下功能：重启系统；管理休眠运行锁；系统电源状态管理和查询；充电和电池状态查询和上报；显示亮灭屏状态管理，包括显示亮度调节。	标准系统
泛 Sensor 服务	泛 Sensor 中包含传感器和小器件，传感器用于侦测环境中所发生事件或变化，并将此消息发送至其他电子设备，小器件用于向外传递信号的设备，包括马达和 LED 灯，对开发者提供控制马达振动和 LED 灯开关的能力。	小型系统
多模输入	OpenHarmony 旨在为开发者提供 NUI（Natural User Interface）的交互方式，有别于传统操作系统的输入，在 OpenHarmony 上，我们将多种维度的输入整合在一起，开发者可以借助应用程序框架、系统自带的 UI 组件或 API 接口轻松地实现具有多维、自然交互特点的应用程序。具体来说，多模输入子系统目前支持传统的输	标准系统

	入交互方式，例如按键和触控。	
启动恢复	启动恢复负责在内核启动之后，应用启动之前的操作系统中间层的启动。并提供系统属性查询、修改及设备恢复出厂设置的功能。	所有系统
升级服务	可支持 OpenHarmony 设备的 OTA(Over The Air)升级。	标准系统
帐号	支持在端侧对接厂商云帐号应用，提供分布式帐号登录状态查询和更新的管理能力。	标准系统
编译构建	编译构建子系统提供了一个基于 Gn 和 ninja 的编译构建框架。	所有系统
测试	开发过程采用测试驱动开发模式，开发者基于系统新增特性可以通过开发者自己开发用例保证，对于系统已有特性的修改，也可通过修改项目中原有的测试用例保证，开发者测试旨在帮助开发者在开发阶段就能开发出高质量代码。	所有系统
数据管理	数据管理支持应用本地数据管理和分布式数据管理： 支持应用本地数据管理，包括轻量级偏好数据库，关系型数据库。 支持分布式数据服务，为应用程序提供不同设备间数据库数据分布式的能力。	标准系统
语言编译运行时	语言运行时提供了 JS、C/C++语言程序的编译、执行环境，提供支撑运行时的基础库，以及关联的 API 接口、编译器和配套工具。	所有系统

分布式任务调度	提供系统服务的启动、注册、查询及管理能力。	所有系统
JS UI 框架	JS UI 框架是 OpenHarmony UI 开发框架，支持类 Web 范式编程。	所有系统
媒体	提供音频、视频、相机等简单有效的媒体组件开发接口，使得应用开发者轻松使用系统的多媒体资源。	所有系统
事件通知	公共事件管理实现了订阅、退订、发布、接收公共事件（例如亮灭屏事件、USB 插拔事件）的能力。	标准系统
杂散软件服务	提供设置时间的能力。	标准系统
用户程序框架	提供包安装、卸载、运行及管理能力。	所有系统
电话服务	提供 SIM 卡、搜网、蜂窝数据、蜂窝通话、短彩信等蜂窝移动网络基础通信能力，可管理多类型通话和数据网络连接，为应用开发者提供便捷一致的通信 API。	标准系统
公共基础类库	公共基础库存放 OpenHarmony 通用的基础组件。这些基础组件可被 OpenHarmony 各业务子系统及上层应用所使用。	所有系统
研发工具链	提供设备连接调试器 hdc；提供了性能跟踪能力和接口；提供了性能调优框架，旨在为开发者提供一套性能调优平台，可以用来分析内存、性能等问题。	标准系统
分布式软总线	分布式软总线旨在为 OpenHarmony 系统提供跨进程或跨设备的通信能力，主要包含软总线和进程间通信两部分。其中，软总线为应用和系统提供近场设备间分布式通信的能力，提供不区分通信方式的设备发现，连接，组网和传输功能；而进程间通信则提供不区	所有系统

	分设备内或设备间的进程间通信能力。	
XTS	XTS 是 OpenHarmony 生态认证测试套件的集合，当前包括 acts（application compatibility test suite）应用兼容性测试套，后续会拓展 dcts（device compatibility test suite）设备兼容性测试套等。	所有系统
系统应用	系统应用提供了 OpenHarmony 标准版上的部分系统应用，如桌面、SystemUI、设置等应用，为开发者提供了构建标准版应用的具体实例，这些应用支持在所有标准版系统的设备上使用。	标准系统
DFX	DFX 是 OpenHarmony 非功能属性能力，包含日志系统、应用和系统事件日志接口、事件日志订阅服务、故障信息生成采集等功能。	所有系统
全球化	当 OpenHarmony 设备或应用在全球不同区域使用时，系统和应用需要满足不同市场用户关于语言、文化习俗的需求。全球化子系统提供支持多语言、多文化的能力，包括资源管理能力和国际化能力。	所有系统
安全	安全子系统包括系统安全、数据安全、应用安全等模块，为 OpenHarmony 提供了保护系统和和用户数据的能力。安全子系统当前开源的功能，包括应用完整性保护、应用权限管理、设备认证、密钥管理服务。	所有系统

第 2 章 源码下载和开发环境

摘要：本文简单介绍 OpenHarmony 开发环境，代码下载、版本更新日志等。
适合群体：想要上手开发 OpenHarmony 设备

搭建环境比较复杂，建议大家使用配置好的环境。
https://gitee.com/lianzhian/OpenHarmony-virtual-machine
虚拟机工具可以使用：VirtualBox
https://www.virtualbox.org/

2.1 源码下载

本文这里做下总结：
（1）注册码云 gitee 账号。

（2）注册码云 SSH 公钥，具体可以百度
生成 key
ssh-keygen -t rsa -C “邮箱地址@”

（3）安装 git 客户端和 git-lfs 并配置用户信息。

```
git config --global user.name "yourname"
git config --global user.email "your-email-address"
git config --global credential.helper store
```

（3）安装码云 repo 工具，可以执行如下命令。

```
curl -s https://gitee.com/oschina/repo/raw/fork_flow/repo-py3 > ./repo
sudo cp repo    /usr/local/bin/repo
#注意，如果没有权限，可下载至其他目录，并将其配置到环境变量中
sudo chmod a+x /usr/local/bin/repo
pip3 install -i https://repo.huaweicloud.com/repository/pypi/simple requests
```

以下是主干代码的下载方式，但是主干代码可能会不稳定，不推荐大家使用，推荐大家使用 TLS 版本。
OpenHarmony 主干代码获取

方式一：通过 repo + ssh 下载（需注册公钥，请参考码云帮助中心）。

```
repo init -u git@gitee.com:openharmony/manifest.git -b master --no-repo-verify
repo sync -c
repo forall -c 'git lfs pull'
```

方式二：通过 repo + https 下载。

```
repo init -u https://gitee.com/openharmony/manifest.git -b master --no-repo-verify
repo sync -c
repo forall -c 'git lfs pull'
```

2.2 Release 版本下载

OpenHarmony 3.x Releases

- OpenHarmony v3.1 Beta (2021-12-31)
- OpenHarmony v3.0.1 LTS (2022-01-12)
- OpenHarmony v3.0 LTS (2021-09-30)

OpenHarmony 2.x Releases

- OpenHarmony v2.2 beta2 (2021-08-04)
- OpenHarmony 2.0 Canary (2021-06-02)

OpenHarmony 1.x Releases

- OpenHarmony v1.1.3 LTS (2021-09-30)
- OpenHarmony v1.1.2 LTS (2021-08-04)
- OpenHarmony 1.1.1 LTS (2021-06-22)
- OpenHarmony 1.1.0 LTS (2021-04-01)
- OpenHarmony 1.0 (2020-09-10)

2.3 开发环境

OpenHarmony 的开发环境主要分为 window、Linux 两个。

其中 window 环境用于编写代码、下载程序等。

Linux 环境用于代码下载、编译等。

这里推荐大家只使用 Linux 环境即可，Linux 可以使用 Ubuntu 20.04 版本。关于 Windows 环境，大家可装可以不装，编写代码可以使用自己喜欢的 IDE、下载的话，不同开发板都会提供不同的下载工具。

Ubuntu 的开发环境可以参考官网：

https://gitee.com/openharmony/docs/blob/master/zh-cn/device-dev/quick-start/quickstart-lite-package-environment.md

本文也会列出来，但是后面官方可能会更新，导致本文不一定适用。

需要注意的是，关于 Ubuntu 的环境主要分为两部分：

（1）OpenHarmony 代码所需的公共部分：这里主要是安装 python、hb 等，这些都是必须的。

（2）具体开发板所需的开发环境：这个跟具体芯片、开发板相关，例如对应的交叉编译器、或者制作文件系统相关的脚本组件等。这些看自己所需的芯片环境是哪些。

2.3.1 OpenHarmony 开发环境

（1）将 Ubuntu Shell 环境修改为 bash。

执行如下命令，确认输出结果为 bash。如果输出结果不是 bash，请根据步骤 2，将 Ubuntu

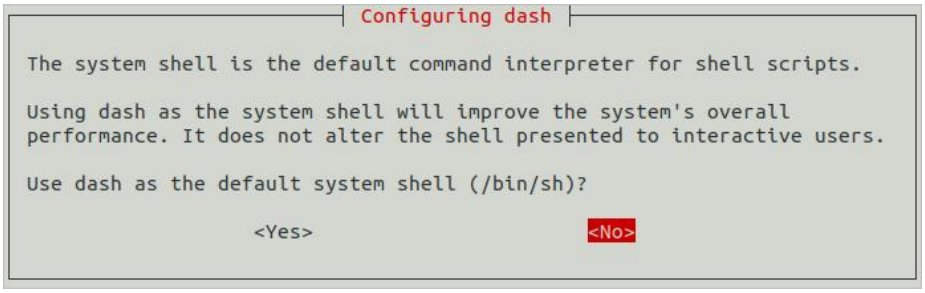
shell 修改为 bash。

ls -l /bin/sh

```
harmonyos@harmonyos-Linux:~$ ls -l /bin/sh
lrwxrwxrwx 1 root root 4 12月 21 19:54 /bin/sh -> bash
```

（2）打开终端工具，执行如下命令，输入密码，然后选择 No，将 Ubuntu shell 由 dash 修改为 bash。

sudo dpkg-reconfigure dash



（3）使用如下 apt-get 命令安装编译所需的必要的库和工具：

sudo apt-get install gcc

sudo apt-get install build-essential gcc g++ make zlib* libffi-dev e2fsprogs pkg-config flex bison perl bc openssl libssl-dev libelf-dev libc6-dev-amd64 binutils binutils-dev libdwarf-dev u-boot-tools mtd-utils gcc-arm-linux-gnueabi cpio device-tree-compiler

（4）安装 hb

需要先下载源码在源码根目录下载运行如下命令安装 hb

python3 -m pip install build/lite

设置环境变量

vim ~/.bashrc

将以下命令拷贝到.bashrc 文件的最后一行，保存并退出。

export PATH=~/.local/bin:$PATH

执行如下命令更新环境变量。

source ~/.bashrc

执行"hb -h"，界面打印以下信息即表示安装成功：

usage: hb

OHOS build system

positional arguments:
{build,set,env,clean}
build Build source code
set OHOS build settings
env Show OHOS build env
clean Clean output

optional arguments:
-h, --help show this help message and exit

2.3.2 Hi3861 开发环境

最好参考官方文档：
https://gitee.com/openharmony/docs/blob/master/zh-cn/device-dev/quick-start/quickstart-lite-steps-hi3861-setting.md
这里做下记录

（1）安装编译依赖基础软件（仅 Ubuntu 20+需要）
执行以下命令进行安装：

sudo apt-get install build-essential gcc g++ make zlib* libffi-dev

（2）安装 Scons
运行如下命令，安装 SCons 安装包。

python3 -m pip install scons
运行如下命令，查看是否安装成功。如果安装成功，查询结果下图所示。

scons -v
图 1 SCons 安装成功界面，版本要求 3.0.4 以上

```
SCons by Steven Knight et al.:
        script: v3.1.2.bee7caf9defd6e108fc2998a2520ddb36a967691, 2019-12-17 02:07:09, by bdeegan on octoc
og
        engine: v3.1.2.bee7caf9defd6e108fc2998a2520ddb36a967691, 2019-12-17 02:07:09, by bdeegan on octoc
og
        engine path: ['/usr/local/lib/python3.7/site-packages/scons/SCons']
Copyright (c) 2001 - 2019 The SCons Foundation
```

（3）安装 python 模块
运行如下命令，安装 python 模块 setuptools。

pip3 install setuptools

（4）安装 GUI menuconfig 工具（Kconfiglib），建议安装 Kconfiglib 13.2.0+版本，任选如

下一种方式。

命令行方式：

sudo pip3 install kconfiglib

（5）安装 pycryptodome，任选如下一种方式。

安装升级文件签名依赖的 Python 组件包，包括：pycryptodome、six、ecdsa。安装 ecdsa 依赖 six，请先安装 six，再安装 ecdsa。

命令行方式：

sudo pip3 install pycryptodome

（6）安装 six，任选如下一种方式。

命令行方式：

sudo pip3 install six --upgrade --ignore-installed six

（7）安装 ecdsa，任选如下一种方式。

命令行方式：

sudo pip3 install ecdsa

（8）安装 gcc_riscv32（WLAN 模组类编译工具链）
下载以下交叉编译工具链：
https://gitee.com/link?target=https%3A%2F%2Frepo.huaweicloud.com%2Fharmonyos%2Fcompiler%2Fgcc_riscv32%2F7.3.0%2Flinux%2Fgcc_riscv32-linux-7.3.0.tar.gz
请先执行以下命令将压缩包解压到根目录：

tar -xvf gcc_riscv32-linux-7.3.0.tar.gz -C ~
vim ~/.bashrc
将以下命令拷贝到.bashrc 文件的最后一行，保存并退出。

export PATH=~/gcc_riscv32/bin:$PATH
生效环境变量。

source ~/.bashrc
Shell 命令行中输入如下命令，如果能正确显示编译器版本号，表明编译器安装成功。

riscv32-unknown-elf-gcc -v

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
    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，建议 123456 即可
sudo smbpasswd -a harmony

重启 samba 服务
sudo service smbd restart

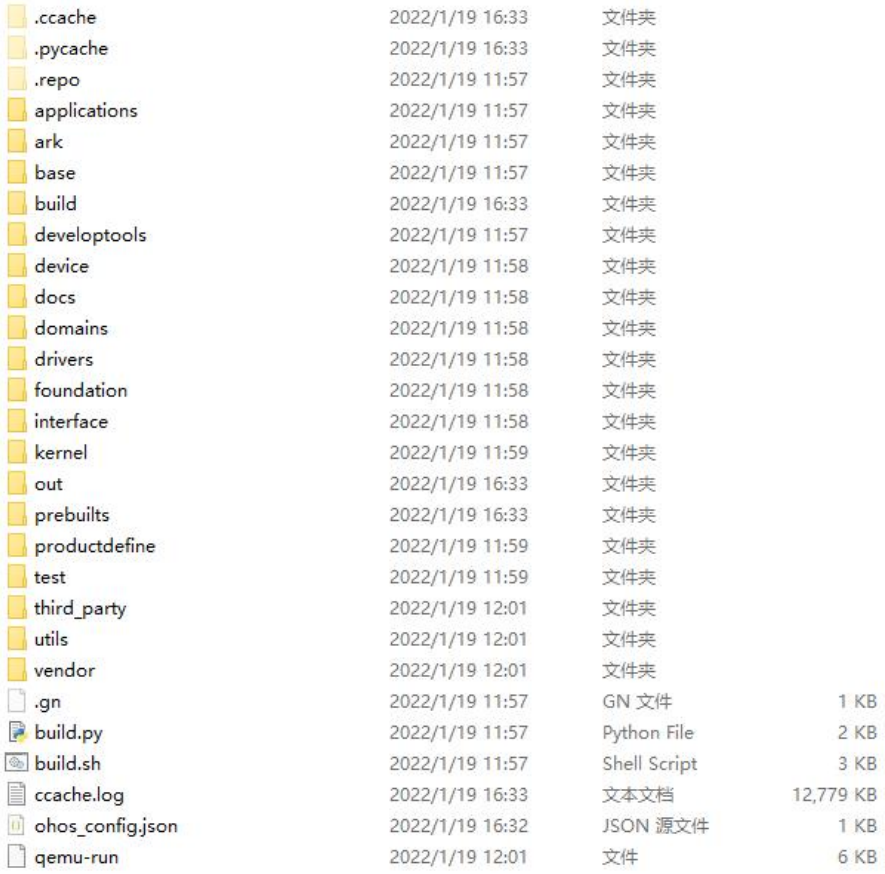
第 3 章 代码编译和烧录

摘要：本文简单介绍 OpenHarmony 最新版本代码目录简单解读、编译、烧录
适合群体：适用于 Hi3861 开发板

本手册所有例程代码均在
https://gitee.com/hihope_iot/HiHope_Pegasus_Doc/tree/master/%E3%80%902%E3%80%91%E8%AF%BE%E7%A8%8B%E6%96%87%E6%A1%A3%E9%85%8D%E5%A5%97%E4%BE%8B%E7%A8%8B

3.1 源码目录

下载完代码后，大家可以进入代码目录：



这里重点介绍几个比较重要的文件夹：

1 vendor 文件夹

该文件夹存放的是厂商相关的配置，包括组件配置、HDF 相关配置，代码目录如下：

```
├── hisilicon
│   ├── Hi3516DV300
│   │   ├── hals
│   │   ├── hdf_config
│   │   └── usb
│   ├── hispark_aries
│   │   ├── hals
│   │   ├── hdf_config
│   │   ├── init_configs
│   │   └── kernel_configs
│   ├── hispark_pegasus
│   │   └── hals
│   ├── hispark_taurus
│   │   ├── hals
│   │   ├── hdf_config
│   │   ├── init_configs
│   │   └── kernel_configs
│   ├── hispark_taurus_linux
│   │   ├── hals
│   │   ├── hdf_config
│   │   └── init_configs
│   └── watchos
│       ├── hals
│       └── hdf_config
└── huawei
    └── hdf
        └── sample
```

可以看到有 hisilicon 文件夹，下面有 Hi3516DV300、hispark_aries 等，其中 hi3861 开发板对应的是 hispark_pegasus

里面有如下文件：

hals	2022/1/19 12:01	文件夹	
BUILD.gn	2022/1/19 12:01	GN 文件	1 KB
config.json	2022/1/19 12:01	JSON 源文件	4 KB
ohos.build	2022/1/19 12:01	BUILD 文件	1 KB

其中比较重要的是 config.json 配置文件，里面定义了内核类型，和使用了哪些子系统。具体我们后再做具体解读。

2 device 文件夹

该文件夹存放的是具体开发板、芯片相关的源码。这里 OpenHarmony 又分为 SoC 和 Board 两大块。其中 SoC 里面是具体芯片相关的代码、包括该芯片相关的驱动； board 是开发板相关的代码，具体跟开发板相关。

之所以这样设计，是为将 SoC 和 board 区分出来，实现 soc 相关代码可复用。因为后续可能存在一个 soc 多个 board 的情况。

```
├── board
│   ├── bearpi
│   │   └── bearpi_hm_nano
│   ├── fnlink
│   │   ├── doc
│   │   ├── drivers
│   │   ├── hcs
│   │   ├── shields
│   │   └── v200zr
│   ├── goodix
│   │   ├── drivers
│   │   ├── gr5515_sk
│   │   └── hcs
│   └── hisilicon
│       ├── hispark_aries
│       ├── hispark_pegasus
│       └── hispark_taurus
└── soc
    ├── bestechnic
    │   ├── bes2600
    │   └── hals
    ├── goodix
    │   └── gr551x
    └── hisilicon
        ├── common
        ├── hi3516dv300
        ├── hi3518ev300
        └── hi3861v100
```

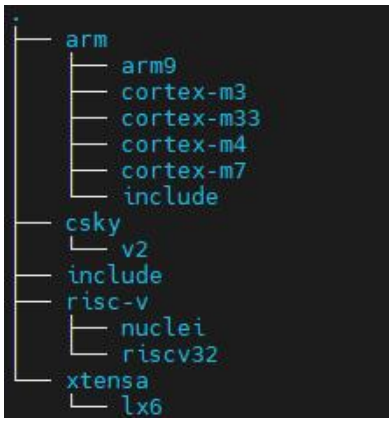
其中，润和的 WiFi IoT 开发板对应的 soc 是 hi3861v100 文件夹，对应的 board 是 hispark_pegasus

3 arch 文件夹

该文件夹存放具体芯片架构的代码，文件夹路径：

kernel/liteos_m/arch

代码路径如下：



于是 OpenHarmony 实现了 ARCH（架构）、soc（芯片）、board（开发板）3 层隔离，降低了代码的耦合性。设计比较合理。

4 applications 文件夹

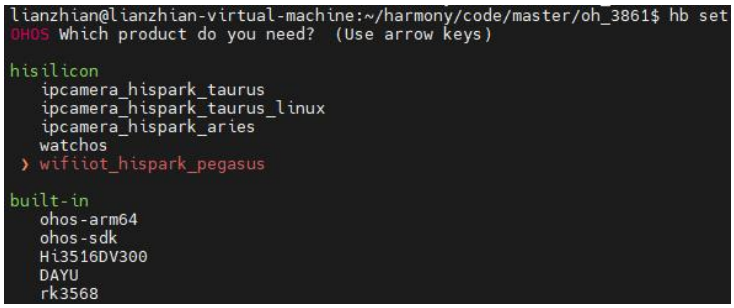
该文件夹存放应用相关代码，后续我们编写代码需要在该文件夹下添加。

3.2 编译

轻量系统采用 hb 编译，在代码根目录下执行

hb set

首次输入可以会提示需要输入路径，直接输入 . （英文的点号）即可



这里我们选择 wifiiot_hispark_pegasus

之后输入：

hb build -f

开始编译

看到如下即表示编译成功：

```
[OHOS INFO] ccache summary:
[OHOS INFO] cache hit (direct)  : 435
[OHOS INFO] cache hit (preprocessed)  : 2
[OHOS INFO] cache miss  : 1
[OHOS INFO] hit rate:  99.77%
[OHOS INFO] mis rate: 0.23%
[OHOS INFO] ------------------------------------------
--------------------------------------------------------------------------------
No pycache actions in pycache, skip statistics
--------------------------------------------------------------------------------
[OHOS INFO] c targets overlap rate statistics
[OHOS INFO] subsystem           files NO.       percentage      builds NO.      percentage      overlap rate
[OHOS INFO] communication           140         32.0%           140         32.0%    1.00
[OHOS INFO] distributedschedule      15         3.4%             15         3.4%     1.00
[OHOS INFO] hiviewdfx                15         3.4%             15         3.4%     1.00
[OHOS INFO] security                170         38.8%           170         38.8%    1.00
[OHOS INFO] startup                   5         1.1%              5         1.1%     1.00
[OHOS INFO] test                     60         13.7%            60         13.7%    1.00
[OHOS INFO] third_party               3         0.7%              3         0.7%     1.00
[OHOS INFO] utils                     4         0.9%              4         0.9%     1.00
[OHOS INFO] xts                      60         13.7%            60         13.7%    1.00
[OHOS INFO]
[OHOS INFO] c overall build overlap rate: 1.00
[OHOS INFO]
[OHOS INFO]
[OHOS INFO] wifiiot_hispark_pegasus build success
[OHOS INFO] cost time: 0:00:34
lianzhian@lianzhian-virtual-machine:~/harmony/code/master/oh_3861$
```

编译出来的固件位于：out/hispark_pegasus/wifiiot_hispark_pegasus/

build_configs	2022/1/19 17:34	文件夹	
etc	2022/1/19 17:34	文件夹	
gen	2022/1/19 17:34	文件夹	
libs	2022/1/19 17:34	文件夹	
NOTICE_FILE	2022/1/19 17:34	文件夹	
obj	2022/1/19 17:34	文件夹	
packages	2022/1/19 17:34	文件夹	
suites	2022/1/19 17:34	文件夹	
.ninja_deps	2022/1/19 17:34	NINJA_DEPS 文件	1 KB
.ninja_log	2022/1/19 17:34	NINJA_LOG 文件	94 KB
all_parts_info.json	2022/1/19 17:34	JSON 源文件	14 KB
args.gn	2022/1/19 17:34	GN 文件	1 KB
binary_installed_parts.json	2022/1/19 17:34	JSON 源文件	1 KB
build.1642584840.5396714.log	2022/1/19 17:34	文本文档	1 KB
build.log	2022/1/19 17:34	文本文档	229 KB
build.ninja	2022/1/19 17:34	NINJA 文件	62 KB
build.ninja.d	2022/1/19 17:34	D 文件	14 KB
build.trace.gz	2022/1/19 17:34	WinRAR 压缩文...	1 KB
Hi3861_boot_signed.bin	2022/1/19 17:34	BIN 文件	24 KB
Hi3861_boot_signed_B.bin	2022/1/19 17:34	BIN 文件	24 KB
Hi3861_loader_signed.bin	2022/1/19 17:34	BIN 文件	15 KB
Hi3861_wifiiot_app.asm	2022/1/19 17:34	ASM 文件	19,316 KB
Hi3861_wifiiot_app.map	2022/1/19 17:34	MAP 文件	4,124 KB
Hi3861_wifiiot_app.out	2022/1/19 17:34	OUT 文件	1,856 KB
Hi3861_wifiiot_app_allinone.bin	2022/1/19 17:34	BIN 文件	974 KB
Hi3861_wifiiot_app_burn.bin	2022/1/19 17:34	BIN 文件	959 KB
Hi3861_wifiiot_app_flash_boot_ota.bin	2022/1/19 17:34	BIN 文件	25 KB
Hi3861_wifiiot_app_ota.bin	2022/1/19 17:34	BIN 文件	496 KB
Hi3861_wifiiot_app_vercfg.bin	2022/1/19 17:34	BIN 文件	1 KB
sorted_action_duration.txt	2022/1/19 17:34	文本文档	1 KB
src_installed_parts.json	2022/1/19 17:34	JSON 源文件	13 KB
src_sa_infos_tmp.json	2022/1/19 17:34	JSON 源文件	1 KB
toolchain.ninja	2022/1/19 17:34	NINJA 文件	124 KB

其中，Hi3861_wifiiot_app_allinone.bin 是我们要烧录到开发板的。

3.3 烧录

（1）基于 vscode 方式烧录

OpenHarmony 可以基于 vscode 的方式进行烧录，但是该方式比较复杂，这里暂时不推荐。

如果读者感兴趣可以参考：

https://device.harmonyos.com/cn/docs/documentation/guide/quickstart-lite-steps-hi3861-burn-0000001190053075

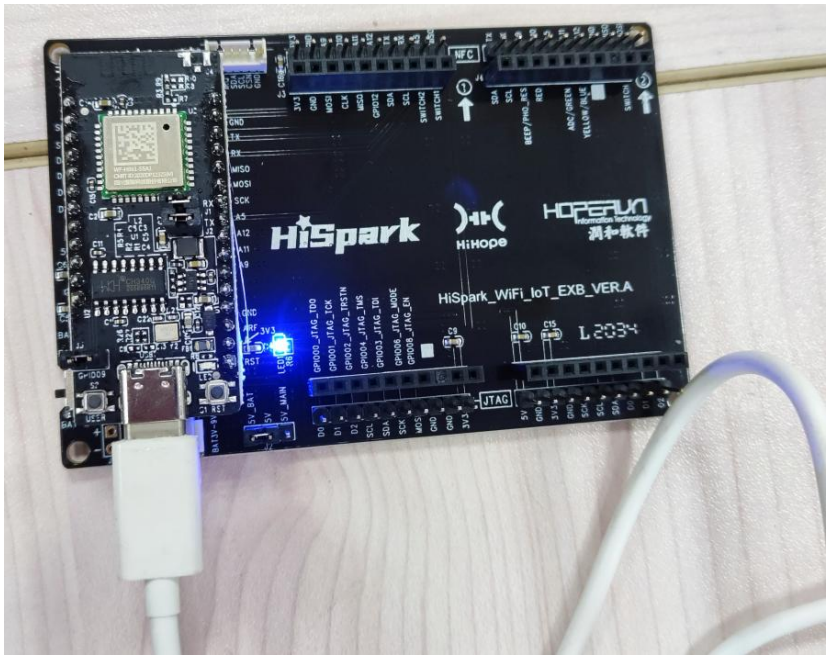
（2）基于 hiburn 工具烧录

推荐读者采用此方式，比较简单便捷。

首先下载 HiBurn 工具，下载链接：

https://harmonyos.51cto.com/resource/29

使用 USB 线连接到 3861 开发板，如图：

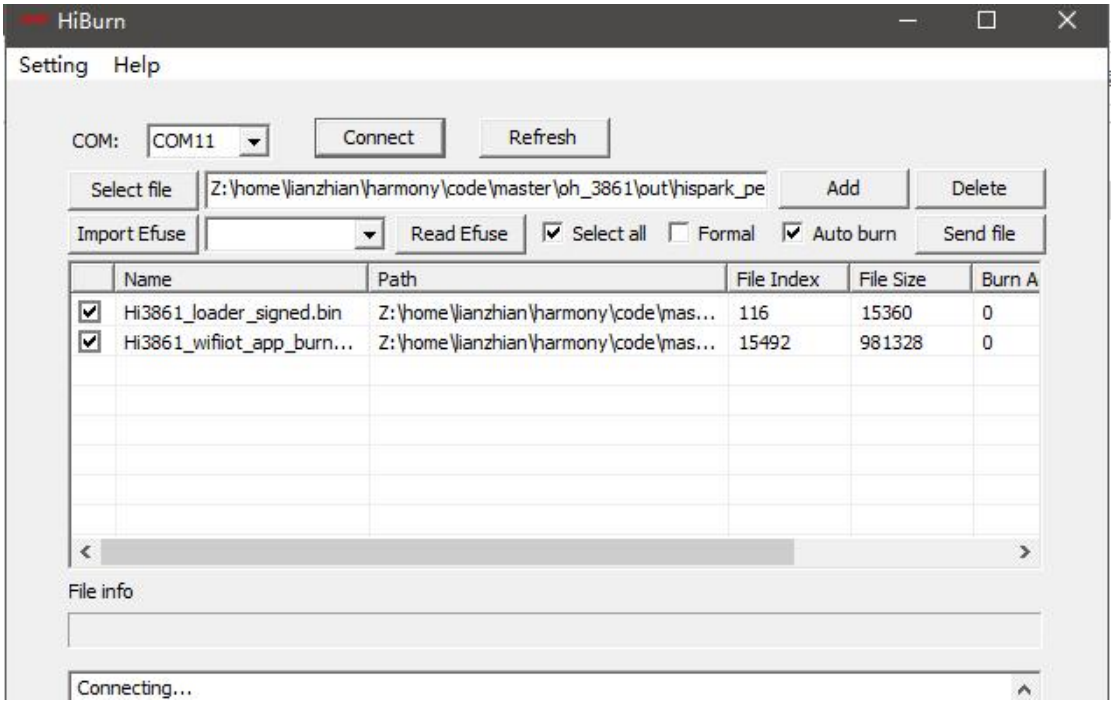


（1）打开 HiBurn 工具，

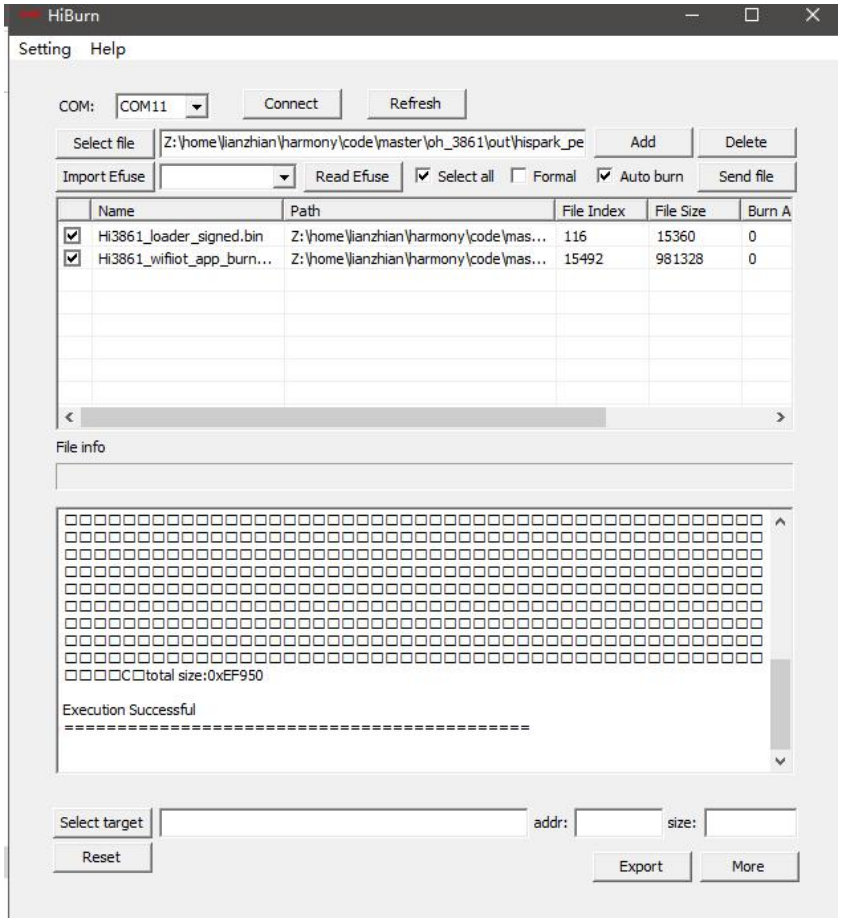
（2）点击 select file 选择要下载的 Hi3861_wifiiot_app_allinone.bin 文件，

（3）勾选上 Auto burn 选项

（4）单击 Connect 按钮



（5）此时按下开发板上面的 RST 复位按钮，即可看到程序已经开始下载：



（6）注意下载后，我们点击 disconnect 按钮，不然我们再次复位会重新烧录。我们也可以关闭掉 HiBurn 程序。

3.4 运行效果

烧录完后，我们可以打开串口工具，查看串口打印:

```
----------------------
2 Tests 0 Failures 0 Ignored
OK
Start to run test suite:HksBnExpModTest
Run test suite 1 times
setup
HksInitialize Begin!
HksInitialize End!
HksBnExpModTestSetUp End2!
../../../test/xts/acts/security_lite/huks/liteos_m_adapter/hks_bn_exp_mod_test.c:126:HksBnExpModTest001:PASS
tearDown

----------------------
1 Tests 0 Failures 0 Ignored
OK
Start to run test suite:HksCipherTest
Run test suite 1 times
setup
HksInitialize Begin!
HksInitialize End!
HksCipherTestSetUp End2!
HksCipherTest001 Begin!
HksCipherTest001 End!
HksCipherTest001 End2!
../../../test/xts/acts/security_lite/huks/liteos_m_adapter/hks_cipher_test.c:190:HksCipherTest001:PASS
HksCipherTest002 Begin!
HksCipherTest002 End!
HksCipherTest002 End2!
../../../test/xts/acts/security_lite/huks/liteos_m_adapter/hks_cipher_test.c:209:HksCipherTest002:PASS
HksCipherTest003 Begin!
HksCipherTest003 End!
HksCipherTest003 End2!
../../../test/xts/acts/security_lite/huks/liteos_m_adapter/hks_cipher_test.c:227:HksCipherTest003:PASS
HksCipherTest004 Begin!
HksCipherTest004 End!
HksCipherTest004 End2!
../../../test/xts/acts/security_lite/huks/liteos_m_adapter/hks_cipher_test.c:244:HksCipherTest004:PASS
HksCipherTest005 Begin!
HksCipherTest005 End!
```

这是因为默认打开了 xts 测试，开发板跑起来系统后会做 xts 测试。

看到如下提示，则表示 xts 测试通过

```
-----------------------
11 Tests 0 Failures 0 Ignored
OK
All the test suites finished!
```

第 4 章 编写第一个程序、启动流程分析

摘要：本文简单介绍如何编写第一个 hello world 程序，以及程序是被执行的
适合群体：适用于 Hi3861 开发板，启动流程分析

4.1 编写第一个程序

编写一个 hello world 程序比较简单，可以参考官网：

https://gitee.com/openharmony/docs/blob/master/zh-cn/device-dev/quick-start/quickstart-lite-steps-hi3861-application-framework.md

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：

```
.
└── applications
    └── sample
        └── wifi-iot
            └── app
                ├── my_first_app
                │   ├── hello_world.c
                │   └── BUILD.gn
                └── BUILD.gn
```

（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 <stdio.h>
#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。

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")。

4.2 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 平台配置文件位于：
vendor\hisilicon\hispark_pegasus\config.json

```
{
    "product_name": "wifiiot_hispark_pegasus",
    "type": "small",
    "version": "3.0",
    "ohos_version": "OpenHarmony 1.0",
    "device_company": "hisilicon",
    "board": "hispark_pegasus",
    "kernel_type": "liteos_m",
    "kernel_is_prebuilt": true,
    "kernel_version": "",
    "subsystems": [
      {
        "subsystem": "applications",
        "components": [
          { "component": "wifi_iot_sample_app", "features":[] }
        ]
      },
      {
        "subsystem": "iot_hardware",
        "components": [
          { "component": "iot_controller", "features":[] }
        ]
      },
      {
        "subsystem": "hiviewdfx",
        "components": [
          { "component": "hilog_lite", "features":[] },
          { "component": "hievent_lite", "features":[] },
          { "component": "blackbox", "features":[] },
          { "component": "hidumper_mini", "features":[] }
        ]
      },
      {
        "subsystem": "distributedschedule",
        "components": [
          { "component": "samgr_lite", "features":[] }
        ]
      },
      {
        "subsystem": "security",
        "components": [
          { "component": "hichainsdk", "features":[] },
          { "component": "deviceauth_lite", "features":[] },
          { "component": "huks", "features":
            [
```


可以看到该配置文件有很多内容，
第一段这里指定了产品名称、版本、使用的内核类型

```
"product_name": "wifiiot_hispark_pegasus",
"type": "small",
"version": "3.0",
"ohos_version": "OpenHarmony 1.0",
"device_company": "hisilicon",
"board": "hispark_pegasus",
"kernel_type": "liteos_m",
"kernel_is_prebuilt": true,
"kernel_version": "",
```

下面这里都是子系统：

```
"subsystems": [
  {
    "subsystem": "applications",
    "components": [
      { "component": "wifi_iot_sample_app", "features":[] }
    ]
  },
  {
    "subsystem": "iot_hardware",
    "components": [
      { "component": "iot_controller", "features":[] }
    ]
  },
  {
    "subsystem": "hiviewdfx",
    "components": [
      { "component": "hilog_lite", "features":[] },
      { "component": "hievent_lite", "features":[] },
      { "component": "blackbox", "features":[] },
      { "component": "hidumper_mini", "features":[] }
    ]
  },
  {
    "subsystem": "distributedschedule",
    "components": [
      { "component": "samgr_lite", "features":[] }
    ]
  },
```

其中我们重点关注这几个模块：

（1）applications：应用子系统

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。

（2）iot_hardware：硬件驱动子系统

头文件路径： base\iot_hardware\peripheral\interfaces\kits

具体代码路径，由 device\board\hisilicon\hispark_pegasus\liteos_m\config.gni 文件中指定：

```
117   # Board adapter dir for OHOS components.
118   board_adapter_dir = "//device/soc/hisilicon/hi3861v100/hi3861_adapter"
119
```

config.gni 文件内容较多，后续会一一解读

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

（3）xts：xts 测试子系统。

这里我们先不要 xts 子系统，不然每次开机后，系统都要跑 xts 认证程序，影响我们后面测试，我们先注删除，如下：

```
99        {
100         "subsystem": "vendor",
101         "components": [
102           { "component": "hi3861_sdk", "target": "//device/soc/hisilicon/hi3861v100/sdk_liteos:wifiiot_sdk", "features":[] }
103         ]
104       },
105       {
106         "subsystem": "xts",
107         "components": [
108           { "component": "xts_acts", "features":
109             [
110               "enable_ohos_test_xts_acts_use_thirdparty_lwip = false"
111             ]
112           },
113           { "component": "xts_tools", "features":[] }
114         ]
115       }
116     ],
117     "third_party_dir": "//device/soc/hisilicon/hi3861v100/sdk_liteos/third_party",
118     "product_adapter_dir": "//vendor/hisilicon/hispark_pegasus/hals"
119   }
120
```

这里的逗号也要删除

4.3 Hi3861 启动流程

由于 hi3681 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：

device\soc\hisilicon\hi3861v100\sdk_liteos\app\wifiiot_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();
        peripheral_init_no_sleep();

    #ifndef CONFIG_FACTORY_TEST_MODE
        hi_lpc_register_wakeup_entry(peripheral_init);
    #endif

        hi_u32 ret = hi_factory_nv_init(HI_FNV_DEFAULT_ADDR, HI_NV_DEFAULT_TOTAL_SIZE,
HI_NV_DEFAULT_BLOCK_SIZE);
        if (ret != HI_ERR_SUCCESS) {
            printf("factory nv init fail\r\n");
        }

        /* partion table should init after factory nv init. */
        ret = hi_flash_partition_init();
        if (ret != HI_ERR_SUCCESS) {
            printf("flash partition table init fail:0x%x \r\n", ret);
        }
        ptable = hi_get_partition_table();

        ret = hi_nv_init(ptable->table[HI_FLASH_PARTITON_NORMAL_NV].addr,
ptable->table[HI_FLASH_PARTITON_NORMAL_NV].size,
            HI_NV_DEFAULT_BLOCK_SIZE);
        if (ret != HI_ERR_SUCCESS) {
            printf("nv init fail\r\n");
        }

    #ifndef CONFIG_FACTORY_TEST_MODE
        hi_upg_init();
    #endif

        /* if not use file system, there is no need init it */
        hi_fs_init();

        (hi_void)hi_event_init(APP_INIT_EVENT_NUM, HI_NULL);
        hi_sal_init();
        /* 此处设为 TRUE 后中断中看门狗复位会显示复位时 PC 值,但有复位不完全风险,
量产版本请务必设为 FALSE */
        hi_syserr_watchdog_debug(HI_FALSE);
        /* 默认记录宕机信息到 FLASH，根据应用场景，可不记录，避免频繁异常宕机情
况损耗 FLASH 寿命 */
        hi_syserr_record_crash_info(HI_TRUE);

        hi_lpc_init();
```

```
    hi_lpc_register_hw_handler(config_before_sleep, config_after_sleep);

#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

    /* 如果不需要使用 Histudio 查看 WIFI 驱动运行日志等，无需初始化 diag */
    /* if not use histudio for diagnostic, diag initialization is unnecessary */
    /* Shell and Diag use the same uart port, only one of them can be selected */
#ifndef CONFIG_FACTORY_TEST_MODE

#ifndef ENABLE_SHELL_DEBUG
#ifdef CONFIG_DIAG_SUPPORT
    (hi_void)hi_diag_init();
#endif
#else
    (hi_void)hi_shell_init();
#endif

    tcpip_init(NULL, NULL);
#endif

    ret = hi_wifi_init(APP_INIT_VAP_NUM, APP_INIT_USR_NUM);
    if (ret != HISI_OK) {
        printf("wifi init failed!\n");
    } else {
        printf("wifi init success!\n");
    }
    app_demo_task_release_mem(); /* 释放系统栈内存所使用任务 */

#ifndef CONFIG_FACTORY_TEST_MODE
    app_demo_upg_init();
#ifdef CONFIG_HILINK
    ret = hilink_main();
    if (ret != HISI_OK) {
        printf("hilink init failed!\n");
    } else {
        printf("hilink init success!\n");
    }
#endif
#endif
```

```
        OHOS_Main();
    }
```

app_main 一开始打印了 SDK 版本号，中间还会有一些初始化动作，最后一行会调用 OHOS_Main();

该函数原型如下：

```
    void OHOS_Main()
    {
    #if defined(CONFIG_AT_COMMAND) || defined(CONFIG_FACTORY_TEST_MODE)
        hi_u32 ret;
        ret = hi_at_init();
        if (ret == HI_ERR_SUCCESS) {
            hi_u32 ret2 = hi_at_register_cmd(G_OHOS_AT_FUNC_TBL,
OHOS_AT_FUNC_NUM);
            if (ret2 != HI_ERR_SUCCESS) {
                printf("Register ohos failed!\n");
            }
        }
    #endif
        OHOS_SystemInit();
    }
```

最后，OHOS_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\bootstrap_lite\services\source\system_init.c

```
    void OHOS_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：

```
19  oid OHOS_SystemInit(void)
20
21      printf("_____>>>>>>> lza %s %d\r\n", __FILE__, __LINE__);
22      MODULE_INIT(bsp);
23      MODULE_INIT(device);
24      MODULE_INIT(core);
25      SYS_INIT(service);
26      SYS_INIT(feature);
27      printf("_____>>>>>>> lza %s %d\r\n", __FILE__, __LINE__);
28      MODULE_INIT(run);
29      printf("_____>>>>>>> lza %s %d\r\n", __FILE__, __LINE__);
30      SAMGR_Bootstrap();
```

我们重新编译后烧录。打开串口查看打印信息，如下：

```
ready to OS start
sdk ver:Hi3861V100R001C00SPC025 2020-09-03 18:10:00
_____>>>>>>> lza app/wifiiot_app/src/app_main.c 406
formatting spiffs...
FileSystem mount ok.
wifi init success!
_____>>>>>>> lza ../../../base/startup/bootstrap_lite/services/source/system_init.c 21
hilog will init.

hievent will init.

hievent init success.
_____>>>>>>> lza ../../../base/startup/bootstrap_lite/services/source/system_init.c 27
___________>>>>>>>>>>>>>>>>>>>> [DEMO] Hello world.
_____>>>>>>> lza ../../../base/startup/bootstrap_lite/services/source/system_init.c 29

hiview init success.
```

可以看到在 27 行之后，就打印 hello world 的信息。符合预期。

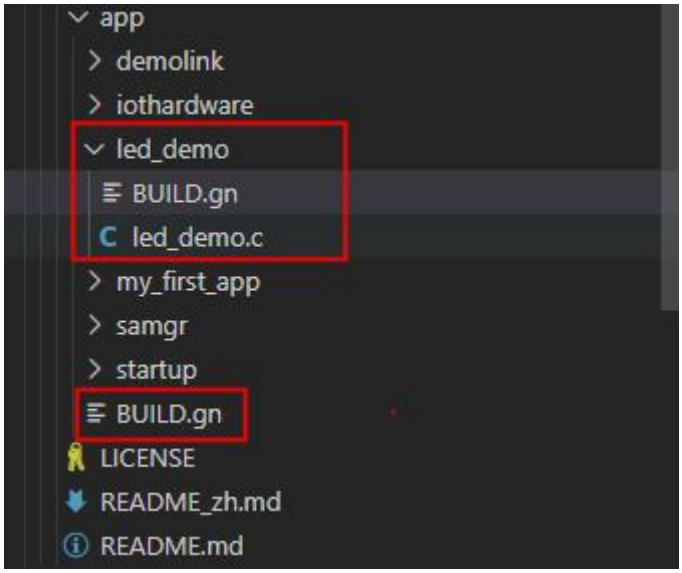
第 5 章 驱动之 GPIO 点灯

摘要：本文简单介绍如何操作 GPIO 去点灯
适合群体：适用于 Hi3861 开发板，L0 轻量系统驱动开发

5.1 点灯例程源码

先看最简单得 LED 灯闪烁操作

源码结构如下：



第一个 BUILD.gn 文件内容：

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

    include_dirs = [
        "//utils/native/lite/include",
        "//kernel/liteos_m/components/cmsis/2.0",
        "//base/iot_hardware/peripheral/interfaces/kits",
    ]
}
```

第二个 BUILD.gn 内容：

```
# Copyright (c) 2020 Huawei Device Co., Ltd.
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
```

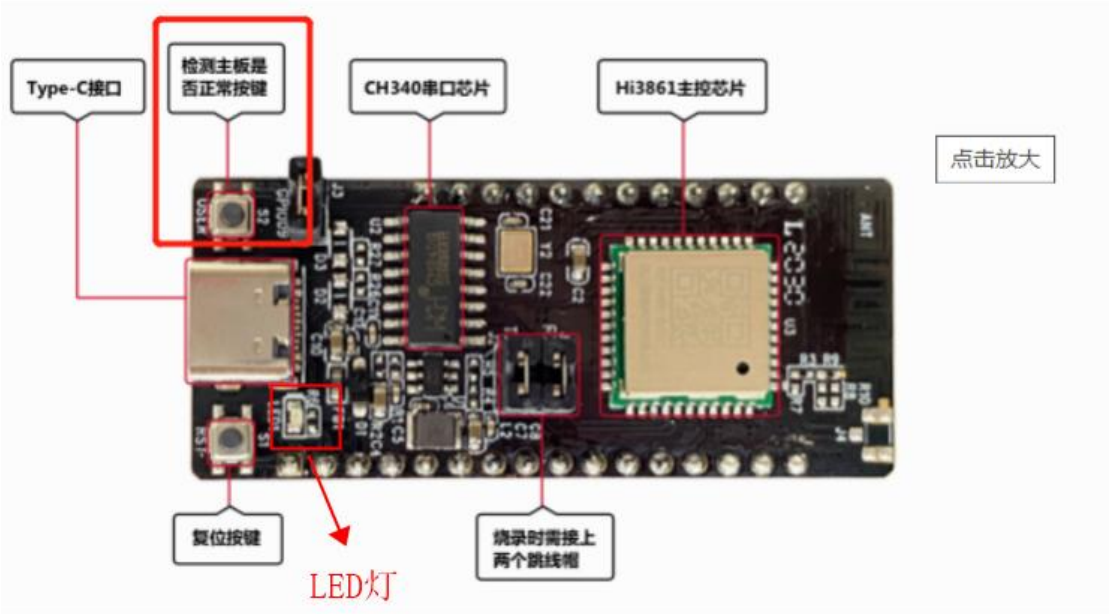
```
#
#         http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

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

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

led_demo.c 内容：

```
#include <unistd.h>
#include "stdio.h"
#include "ohos_init.h"
#include "cmsis_os2.h"
#include "iot_gpio.h"

#define LED_TEST_GPIO 9 // for hispark_pegasus

void *LedTask(const char *arg)
{
        //初始化 GPIO
        IoTGpioInit(LED_TEST_GPIO);

        //设置为输出
        IoTGpioSetDir(LED_TEST_GPIO, IOT_GPIO_DIR_OUT);

        (void)arg;
        while (1)
        {
                //输出低电平
                IoTGpioSetDir(LED_TEST_GPIO, 0);
                usleep(300000);
                //输出高电平
                IoTGpioSetDir(LED_TEST_GPIO, 1);
                usleep(300000);
```

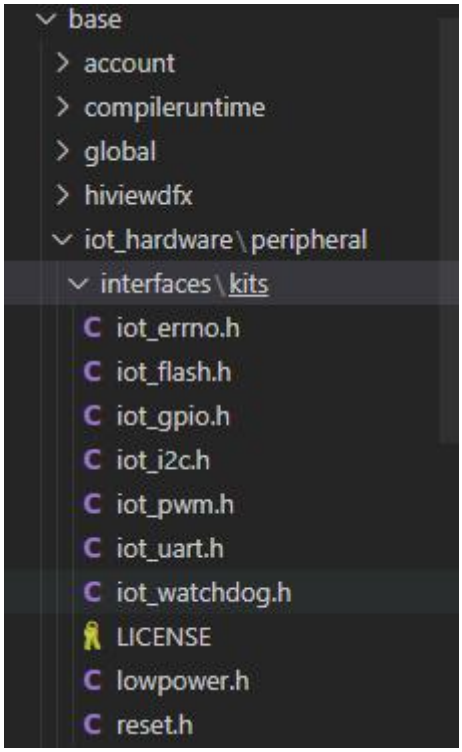
```
        }

        return NULL;
}

void led_demo(void)
{
        osThreadAttr_t attr;

        attr.name = "LedTask";
        attr.attr_bits = 0U;
        attr.cb_mem = NULL;
        attr.cb_size = 0U;
        attr.stack_mem = NULL;
        attr.stack_size = 512;
        attr.priority = 26;

        if (osThreadNew((osThreadFunc_t)LedTask, NULL, &attr) == NULL) {
                printf("[LedExample] Falied to create LedTask!\n");
        }

}


SYS_RUN(led_demo);
```

编译后烧录进去，应该可以看到复位按键旁边的 LED 灯一直在闪烁。



5.2 驱动框架

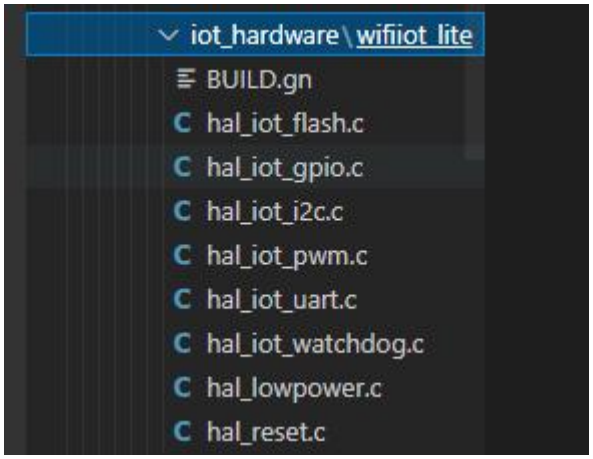
OpenHarmony 为轻量系统提供了一套简单的驱动封装接口，函数的定义相关头文件位于“base\iot_hardware\peripheral\interfaces\kits”



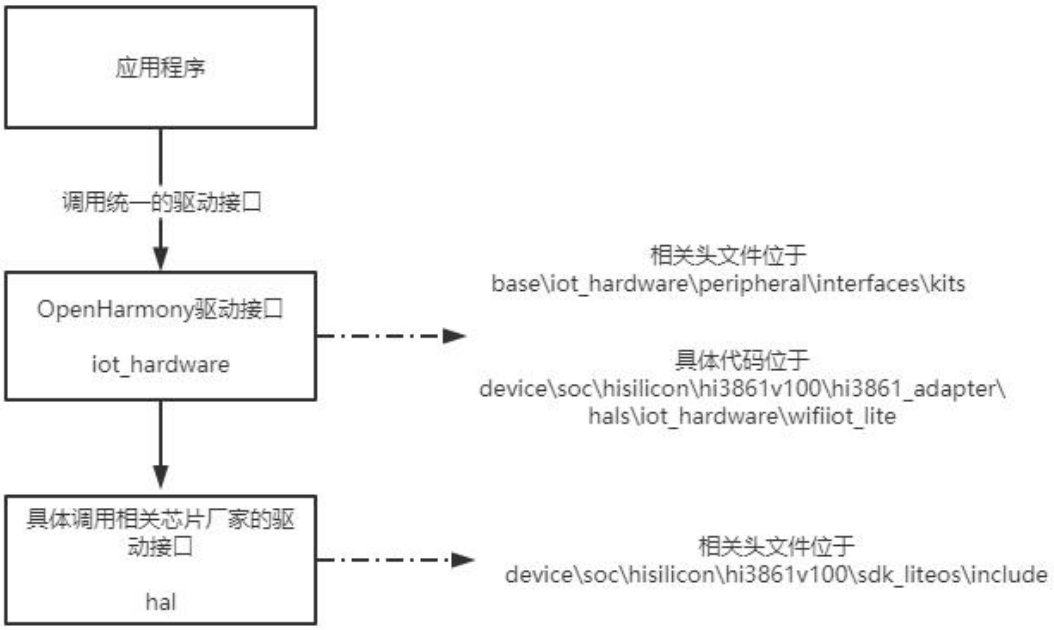
这里只有头文件，具体的函数实现，需要在对应的 soc 中，具体路径定义由 device\board\hisilicon\hispark_pegasus\liteos_m\config.gni 文件中定义：

```
117   # Board adapter dir for OHOS components.
118   board_adapter_dir = "//device/soc/hisilicon/hi3861v100/hi3861_adapter"
119
```

所以我们可以知道，具体的路径就是
“device\soc\hisilicon\hi3861v100\hi3861_adapter\hals\iot_hardware\wifiiot_lite”
相关文件如下：



这里是代码实现，具体是将 hi3861 相关的驱动接口封装成鸿蒙的驱动接口。
所以我们可以总结如下：



5.3 GPIO 相关接口函数

（1）相关枚举：

```
/**
  * @brief 枚举 GPIO 电平值。
  */
类型定义枚举 {
    /** 低 GPIO 电平 */
    IOT_GPIO_VALUE0 = 0，
    /** 高 GPIO 电平 */
    IOT_GPIO_VALUE1
} IotGpioValue;

/**
  * @brief 枚举 GPIO 方向。
  */
类型定义枚举 {
    /** 输入 */
    IOT_GPIO_DIR_IN = 0，
    /** 输出 */
    IOT_GPIO_DIR_OUT
} IotGpioDir;

/**
  * @brief 枚举 GPIO 中断触发模式。
  */
类型定义枚举 {
    /** 电平敏感中断 */
```

```
        IOT_INT_TYPE_LEVEL = 0，
        /** 边缘敏感中断 */
        IOT_INT_TYPE_EDGE
} IotGpioIntType;

/**
  * @brief 枚举 I/O 中断极性。
  */
类型定义枚举 {
        /** 低电平或下降沿中断 */
        IOT_GPIO_EDGE_FALL_LEVEL_LOW = 0，
        /** 高电平或上升沿中断 */
        IOT_GPIO_EDGE_RISE_LEVEL_HIGH
} IotGpioIntPolarity;
```

（2）普通 GPIO 相关 API

```
/**
 * @brief 表示 GPIO 中断回调。
 *
 */
typedef void (*GpioIsrCallbackFunc) (char *arg);

/**
 * @brief 初始化一个 GPIO 设备。
 *
 * @param id 表示 GPIO 引脚号。
 * @return 如果 GPIO 设备已初始化，则返回 {@link IOT_SUCCESS}；
 * 否则返回 {@link IOT_FAILURE}。其他返回值详见芯片说明。
 * @从 2.2 开始
 * @2.2 版
 */
unsigned int IoTGpioInit（unsigned int id）；

/**
 * @brief 取消初始化 GPIO 设备。
 *
 * @param id 表示 GPIO 引脚号。
 * @return 如果 GPIO 设备被取消初始化，则返回 {@link IOT_SUCCESS}；
 * 否则返回 {@link IOT_FAILURE}。其他返回值详见芯片说明。
 * @从 2.2 开始
 * @2.2 版
 */
unsigned int IoTGpioDeinit（unsigned int id）；
```

```
/**
 * @brief 设置 GPIO 引脚的方向。
 *
 * @param id 表示 GPIO 引脚号。
 * @param dir 指示 GPIO 输入/输出方向。
 * @return 如果设置了方向，则返回 {@link IOT_SUCCESS}；
 * 否则返回 {@link IOT_FAILURE}。其他返回值详见芯片说明。
 * @从 2.2 开始
 * @2.2 版
 */
unsigned int IoTGpioSetDir(unsigned int id, IotGpioDir dir);

/**
 * @brief 获取 GPIO 引脚的方向。
 *
 * @param id 表示 GPIO 引脚号。
 * @param dir 指示指向 GPIO 输入/输出方向的指针。
 * @return 如果获取到方向，则返回 {@link IOT_SUCCESS}；
 * 否则返回 {@link IOT_FAILURE}。其他返回值详见芯片说明。
 * @从 2.2 开始
 * @2.2 版
 */
unsigned int IoTGpioGetDir(unsigned int id, IotGpioDir *dir);

/**
 * @brief 设置 GPIO 引脚的输出电平值。
 *
 * @param id 表示 GPIO 引脚号。
 * @param val 表示输出电平值。
 * @return 如果设置了输出级别值，则返回 {@link IOT_SUCCESS}；
 * 否则返回 {@link IOT_FAILURE}。其他返回值详见芯片说明。
 * @从 2.2 开始
 * @2.2 版
 */
unsigned int IoTGpioSetOutputVal（unsigned int id，IotGpioValue val）；

/**
 * @brief 获取 GPIO 引脚的输出电平值。
 *
 * @param id 表示 GPIO 引脚号。
 * @param val 表示指向输出电平值的指针。
 * @return 如果获得输出电平值，则返回 {@link IOT_SUCCESS}；
 * 否则返回 {@link IOT_FAILURE}。其他返回值详见芯片说明。
```

```
 * @从 2.2 开始
 * @2.2 版
 */
unsigned int IoTGpioGetOutputVal(unsigned int id, IotGpioValue *val);

/**
 * @brief 获取 GPIO 引脚的输入电平值。
 *
 * @param id 表示 GPIO 引脚号。
 * @param val 表示指向输入电平值的指针。
 * @return 如果获得输入电平值，则返回 {@link IOT_SUCCESS}；
 * 否则返回 {@link IOT_FAILURE}。其他返回值详见芯片说明。
 * @从 2.2 开始
 * @2.2 版
 */
unsigned int IoTGpioGetInputVal(unsigned int id, IotGpioValue *val);
```

（3）按键中断相关 API

```
/**
 * @brief 启用 GPIO 引脚的中断功能。
 *
 * 该函数可用于设置 GPIO 引脚的中断类型、中断极性和中断回调。
 *
 * @param id 表示 GPIO 引脚号。
 * @param intType 表示中断类型。
 * @param intPolarity 指示中断极性。
 * @param func 表示中断回调函数。
 * @param arg 表示指向中断回调函数中使用的参数的指针。
 * @return 如果启用中断功能，则返回 {@link IOT_SUCCESS}；
 * 否则返回 {@link IOT_FAILURE}。其他返回值详见芯片说明。
 * @从 2.2 开始
 * @2.2 版
 */
unsigned int IoTGpioRegisterIsrFunc（unsigned int id，IotGpioIntType intType，
IotGpioIntPolarity intPolarity，
                                    GpioIsrCallbackFunc fun，char *arg);

/**
 * @brief 禁用 GPIO 引脚的中断功能。
 *
 * @param id 表示 GPIO 引脚号。
 * @return 如果中断功能被禁用，则返回 {@link IOT_SUCCESS}；
 * 否则返回 {@link IOT_FAILURE}。其他返回值详见芯片说明。
 * @从 2.2 开始
```

```
     * @2.2 版
     */
    unsigned int IoTGpioUnregisterIsrFunc(unsigned int id);

    /**
     * @brief 屏蔽 GPIO 引脚的中断功能。
     *
     * @param id 表示 GPIO 引脚号。
     * @param mask 表示中断函数是否被屏蔽。
     * 值<b>1</b>表示屏蔽中断功能，<b>0</b>表示不屏蔽中断功能。
     * @return 如果中断功能被屏蔽，则返回 {@link IOT_SUCCESS}；
     * 否则返回 {@link IOT_FAILURE}。其他返回值详见芯片说明。
     * @从 2.2 开始
     * @2.2 版
     */
    unsigned int IoTGpioSetIsrMask(unsigned int id, unsigned char mask);

    /**
     * @brief 设置 GPIO 引脚的中断触发模式。
     *
     * 此函数根据中断类型和中断极性配置 GPIO 引脚。
     *
     * @param id 表示 GPIO 引脚号。
     * @param intType 表示中断类型。
     * @param intPolarity 指示中断极性。
     * @return 如果设置了中断触发模式，则返回 {@link IOT_SUCCESS}；
     * 否则返回 {@link IOT_FAILURE}。其他返回值详见芯片说明。
     * @从 2.2 开始
     * @2.2 版
     */
    unsigned int IoTGpioSetIsrMode(unsigned int id, IotGpioIntType intType, IotGpioIntPolarity
intPolarity);
```

第 6 章 驱动之 ADC 按键

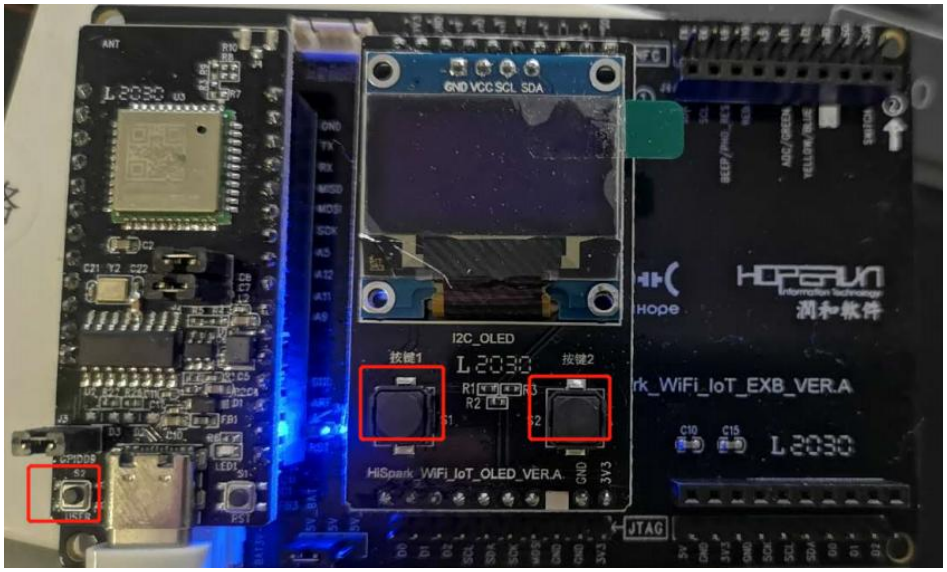
摘要：本文简单介绍如何操作 ADC 去读取电压，并且实现开发板上 3 个 ADC 按键检测的功能

适合群体：适用于润和 Hi3861 开发板，L0 轻量系统驱动开发

文中所有代码仓库：https://gitee.com/qidiyun/hihope-3861-smart-home-kit

6.1 实验效果

查看开发板，可以看到除了复位按键之外，还有 3 个按键。而查看原理，我们可以看到这个 3 个按键其实都是接的 GPIO5 引脚，而 GPIO5 引脚又可复用为 ADC2 引脚。



故而，我们可以猜测出来我们可以使用 ADC 检测电压，判断出来是哪个引脚被按下了。

看下效果：

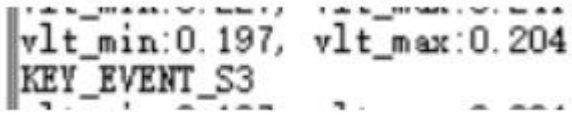
当我按下 按键 1 的时候，串口会打印：

```
vlt_min:0.563, vlt_max:0.577
KEY_EVENT_S1
```

当我按下按键 2 的时候串口会打印：

```
vlt_min:0.963, vlt_max:0.970
KEY_EVENT_S2
```

当我按下 USER 按键的时候串口会打印



其中 vlt_min 表示读取到 ADC 值的最小值，

vlt_max 表示读取到 ADC 值的最大值。

由此我们可以看到，按键 1 被按下的时候，ADC 值得范围在 0.563 ~ 0.577

按键 2 按下后，ADC 值在 0.963 ~ 0.970

USER 按键按下后 ADC 值 在 0.197 ~ 0.204

如果没有按键按下，则 ADC 值在 3.227 ~ 3.241

vlt_min:3.227, vlt_max:3.241

6.2 代码实现

代码实现其实很简单。

（1）引脚初始化

这里由于 GPIO5 默认被复用为串口引脚，这里我们重新修改为普通 GPIO 引脚。初始化代码如下：

```
(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;
}
```

（2）读取 ADC 值

读取 ADC 值的代码页相对简单，这里，我是重复读取 64 次，减少误判。

```
    memset_s(g_adc_buf, sizeof(g_adc_buf), 0x0, sizeof(g_adc_buf));

    for (i = 0; i < ADC_TEST_LENGTH; i++) {
        ret = hi_adc_read((hi_adc_channel_index)HI_ADC_CHANNEL_2, &data,
HI_ADC_EQU_MODEL_1, HI_ADC_CUR_BAIS_DEFAULT, 0);
        if (ret != HI_ERR_SUCCESS) {
            printf("ADC Read Fail\n");
            return;
        }
        g_adc_buf[i] = data;
    }
```

（3）对读出来的 ADC 值进行判断处理

S1 对应的是按键 1 、 S2 对应的是按键 2 、 S3 对应的是 USER 按键

```
    for (i = 0; i < data_len; i++) {
        vlt = g_adc_buf[i];
        float voltage = (float)vlt * 1.8 * 4 / 4096.0;    /* vlt * 1.8 * 4 / 4096.0: Convert code
into voltage */
        vlt_max = (voltage > vlt_max) ? voltage : vlt_max;
        vlt_min = (voltage < vlt_min) ? voltage : vlt_min;
    }
    //printf("vlt_min:%.3f, vlt_max:%.3f \n", vlt_min, vlt_max);

    vlt_val = (vlt_min + vlt_max)/2.0;

    if((vlt_val > 0.4) && (vlt_val < 0.6))
    {
        if(key_flg == 0)
        {
            key_flg = 1;
            key_status = KEY_EVENT_S1;
        }
    }
    if((vlt_val > 0.8) && (vlt_val < 1.1))
    {
        if(key_flg == 0)
        {
            key_flg = 1;
```

```
            key_status = KEY_EVENT_S2;
        }
    }

    if((vlt_val > 0.01) && (vlt_val < 0.3))
    {
        if(key_flg == 0)
        {
            key_flg = 1;
            key_status = KEY_EVENT_S3;
        }
    }

    if(vlt_val > 3.0)
    {
        key_flg = 0;
        key_status = KEY_EVENT_NONE;
    }
```

（4）使用

编写好上面代码后，就可以直接在 while 循环中判断按键值了：

```
    while(1)
    {
        //读取 ADC 值
        app_demo_adc_test();

        switch(get_key_event())
        {
            case KEY_EVENT_NONE:
            {

            }
            break;

            case KEY_EVENT_S1:
            {
                printf("KEY_EVENT_S1 \r\n");
            }
            break;
```

```
                case KEY_EVENT_S2:
                {
                    printf("KEY_EVENT_S2 \r\n");
                }
                break;

                case KEY_EVENT_S3:
                {
                    printf("KEY_EVENT_S3 \r\n");
                }
                break;

            }

            usleep(30000);
        }
```

第 7 章 驱动之 I2C 显示 OLED 屏幕

摘要：本文简单介绍如何操作 I2C 去显示 OLED 屏幕，并且实现动画播放、中文英文显示、绘图等功能

适合群体：适用于润和 Hi3861 开发板，L0 轻量系统驱动开发

文中所有代码仓库：https://gitee.com/qidiyun/hihope-3861-smart-home-kit

本文参考许思维老师的文章，许思维老师主页是：

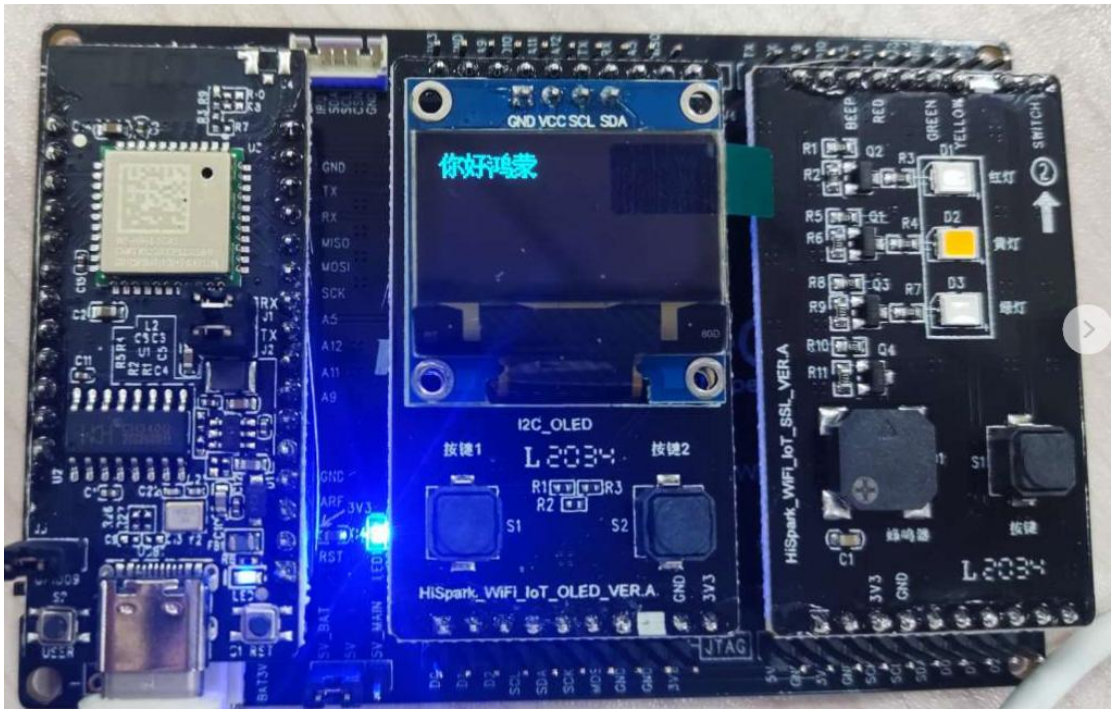
https://harmonyos.51cto.com/user/posts/6631823

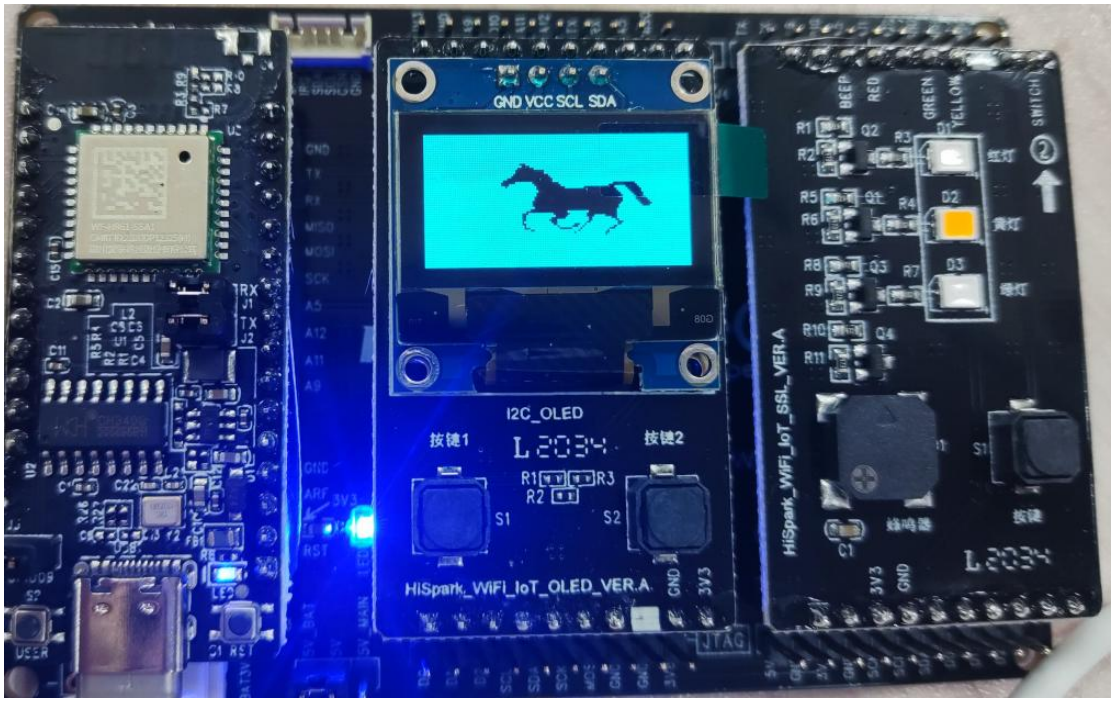
感谢许思维老师~

7.1 实验效果

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。

以下是移植效果：





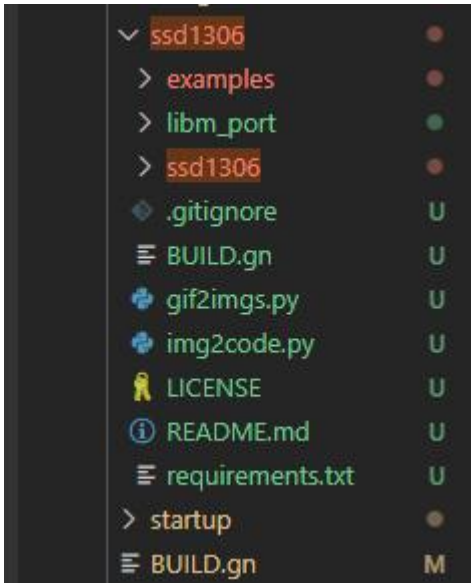
接口：I2C

使用引脚：HI_IO_NAME_GPIO_13 、 HI_IO_NAME_GPIO_14

7.2 代码

这里我直接用许思维老师之前移植好的代码，不过由于鸿蒙版本更新过快，许思维老师之前的代码是基于 1.0 版本，直接拿到 3.0 以上版本编译会出错，这里我修改了编译错误。放在我的仓库中。

整个代码目录如下：



主要是 3 个文件夹：

（1）examples

测试代码，里面有一个 ssd1306_demo.c 文件，是我们的入口函数。

（2）libm_port

从 musl libc 中抽取的`sin`和`cos`的实现。

（3）ssd1306

相关驱动代码部分。

接下来我们来重点看下代码部分：

（1）初始化入口函数

入口函数是 Ssd1306TestDemo ，它创建了 Ssd1306TestTask 线程，所以我们重点看 Ssd1306TestTask 线程。

```
void Ssd1306TestTask(void* arg)
{
    (void) arg;
    //先初始化引脚
    IoTGpioInit(HI_IO_NAME_GPIO_13);
    IoTGpioInit(HI_IO_NAME_GPIO_14);

    //将引脚功能设置为 I2C 引脚
    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);

    //初始化 I2C0
    IoTI2cInit(0, OLED_I2C_BAUDRATE);

    //WatchDogDisable();

    usleep(20*1000);
    //初始化 SSD1306
    ssd1306_Init();
    //全部清空
    ssd1306_Fill(Black);
    ssd1306_SetCursor(0, 0);
    //显示  Hello HarmonyOS!
    ssd1306_DrawString("Hello HarmonyOS!", Font_7x10, White);

    uint32_t start = HAL_GetTick();
    ssd1306_UpdateScreen();
    uint32_t end = HAL_GetTick();
    printf("ssd1306_UpdateScreen time cost: %d ms.\r\n", end - start);

    TestDrawChinese1();
    TestDrawChinese2();

    TestGetTick();
    while (1) {
```

```
            //进行所有用例测试
            ssd1306_TestAll();
            usleep(10000);
        }
    }
```

（2）I2C 发送函数

我们要操作 OLED 屏幕，就需要使用 I2C 发送数据给 OLED 屏幕，代码使用 ssd1306_SendData 函数发送 I2C 数据，该函数原型如下，直接调用的 hi3861 的 i2c 接口函数：

```
static uint32_t ssd1306_SendData(uint8_t* data, size_t size)
{
    int id = SSD1306_I2C_IDX;

    return IoTI2cWrite(id, SSD1306_I2C_ADDR, data, size);
}
```

（3）修改 usr_config.mk 文件

注意，默认代码是没有打开 i2c 功能的，直接编译会提示相关的 i2c 函数没有定义，我们需要打开 i2c 的功能，具体是修改 usr_config.mk 文件 ，目前版本（3.1）的路径是：

device\soc\hisilicon\hi3861v100\sdk_liteos\build\config\usr_config.mk

但是代码结构可能会调整，路径可能会变。

增加 CONFIG_I2C_SUPPORT=y

```
30  #
31  # BSP Settings
32  #
33  CONFIG_I2C_SUPPORT=y
34  # CONFIG_I2S_SUPPORT is not set
35  # CONFIG_SPI_SUPPORT is not set
36  # CONFIG_DMA_SUPPORT is not set
37  # CONFIG_SDIO_SUPPORT is not set
38  # CONFIG_SPI_DMA_SUPPORT is not set
39  # CONFIG_UART_DMA_SUPPORT is not set
40  # CONFIG_PWM_SUPPORT is not set
41  # CONFIG_PWM_HOLD_AFTER_REBOOT is not set
42  CONFIG_AT_SUPPORT=y
43  CONFIG_FILE_SYSTEM_SUPPORT=y
44  CONFIG_UART0_SUPPORT=y
45  CONFIG_UART1_SUPPORT=y
46  # CONFIG_UART2_SUPPORT is not set
47  # end of BSP Settings
```

第 8 章 其它驱动开发示例

摘要：本文简单介绍 Hi3861 其他驱动的开发示例、包括 PWM、SPI、SDIO 等。
适合群体：适用于润和 Hi3861 开发板，L0 轻量系统驱动开发
文中所有代码仓库：https://gitee.com/qidiyun/hihope-3861-smart-home-kit

8.1 代码示例

OpenHarmony 代码中，Hi3861 提供了绝大部分的驱动示例代码，文件路径：device\soc\hisilicon\hi3861v100\sdk_liteos\app\demo\src
开发者可以参考，文件如下：

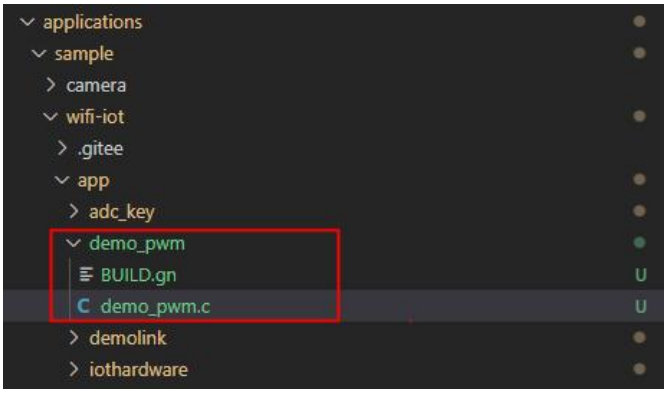
app_demo_adc.c	2022/1/19 16:40	C 源文件	5 KB
app_demo_efuse.c	2022/1/19 16:40	C 源文件	7 KB
app_demo_flash.c	2022/1/19 16:40	C 源文件	4 KB
app_demo_i2c.c	2022/1/19 16:40	C 源文件	4 KB
app_demo_i2s.c	2022/1/19 16:40	C 源文件	8 KB
app_demo_io_gpio.c	2022/1/19 16:40	C 源文件	3 KB
app_demo_nv.c	2022/1/19 16:40	C 源文件	4 KB
app_demo_pwm.c	2022/1/19 16:40	C 源文件	2 KB
app_demo_sdio_device.c	2022/1/19 16:40	C 源文件	6 KB
app_demo_sdio_device.h	2022/1/19 16:40	C Header 源文件	1 KB
app_demo_spi.c	2022/1/19 16:40	C 源文件	32 KB
app_demo_timer_systick.c	2022/1/19 16:40	C 源文件	5 KB
app_demo_tsensor.c	2022/1/19 16:40	C 源文件	5 KB
app_demo_uart.c	2022/1/19 16:40	C 源文件	3 KB
app_demo_upg_verify.c	2022/1/19 16:40	C 源文件	2 KB
app_demo_upg_verify.h	2022/1/19 16:40	C Header 源文件	1 KB
app_http_client.c	2022/1/19 16:40	C 源文件	3 KB
app_main.c	2022/1/19 16:40	C 源文件	18 KB
app_promis.c	2022/1/19 16:40	C 源文件	3 KB
app_promis.h	2022/1/19 16:40	C Header 源文件	1 KB
es8311_codec.c	2022/1/19 16:40	C 源文件	7 KB
es8311_codec.h	2022/1/19 16:40	C Header 源文件	5 KB
netcfg_sample.c	2022/1/19 16:40	C 源文件	11 KB
network_config_sample.c	2022/1/19 16:40	C 源文件	15 KB
SConscript	2022/1/19 16:40	文件	1 KB
wifi_softap.c	2022/1/19 16:40	C 源文件	4 KB
wifi_sta.c	2022/1/19 16:40	C 源文件	5 KB

8.2 如何使用

（1）创建文件夹

一般情况下，我们自己如果需要使用某个驱动，编写的代码要存放在 app 目录下，这里我们以 app_demo_pwm.c 为例。

我们在 app 中新建文件夹 demo_pwm，里面存放代码 demo_pwm.c，然后 app_demo_pwm.c 所有的代码都复制到 demo_pwm.c 中，整个文件夹如下：



（2）编写入口函数

我们需要为 demo_pwm.c 编写一个入口函数，通常情况下，是创建一个线程去执行，通用的代码示例如下：

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

    while(1)
    {
        //调用 app_demo_pwm
        app_demo_pwm();
        usleep(10000);
    }
}


void pwm_demo(void)
{
    osThreadAttr_t attr;

    attr.name = "PWM_Task";
    attr.attr_bits = 0U;
    attr.cb_mem = NULL;
    attr.cb_size = 0U;
    attr.stack_mem = NULL;
    attr.stack_size = 2048;
    attr.priority = 26;

    if (osThreadNew((osThreadFunc_t)PWM_Task, NULL, &attr) == NULL) {
        printf("[PWM_Task] Falied to create PWM_Task!\n");
    }
```

```
}


SYS_RUN(pwm_demo);
```

（3）头文件

此外我们还得修改头文件，首先我们先删除掉原先的 include 的头文件，然后添加如下通用头文件：

```
#include <stdio.h>
#include <unistd.h>
#include "ohos_init.h"
#include "cmsis_os2.h"

#include <hi_types_base.h>
#include <hi_early_debug.h>
```

接着我们可以根据自己使用到的哪个驱动，添加对应的驱动头文件，比如我们用到的是 pwm，那么添加的头文件如下：

```
#include <hi_pwm.h>
```

（3）头文件路径

此外我们还得修改头文件，这里主要修改 BUILD.gn 文件，通常情况下需要增加：

```
"//device/soc/hisilicon/hi3861v100/hi3861_adapter/hals/communication/wifi_lite/wifiservice",
"//device/soc/hisilicon/hi3861v100/hi3861_adapter/kal",
```

修改后如下：

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

    include_dirs = [
        "//utils/native/lite/include",
        "//kernel/liteos_m/components/cmsis/2.0",
        "//base/iot_hardware/peripheral/interfaces/kits",
        "//device/soc/hisilicon/hi3861v100/hi3861_adapter/hals/communication/wifi_lite/wifiservice",
        "//device/soc/hisilicon/hi3861v100/hi3861_adapter/kal",
    ]
}
```

（4）修改 usr_config.mk 文件

此外，如果某个驱动对应的宏我们如果没有打开，那么我们可能还得修改 usr_config.mk 文件，该文件通常路径为：

device\soc\hisilicon\hi3861v100\sdk_liteos\build\config\usr_config.mk

这里我们用到了 PWM，修改前：

```
33  CONFIG_I2C_SUPPORT=y
34  # CONFIG_I2S_SUPPORT is not set
35  # CONFIG_SPI_SUPPORT is not set
36  # CONFIG_DMA_SUPPORT is not set
37  # CONFIG_SDIO_SUPPORT is not set
38  # CONFIG_SPI_DMA_SUPPORT is not set
39  # CONFIG_UART_DMA_SUPPORT is not set
40  # CONFIG_PWM_SUPPORT is not set
41  # CONFIG_PWM_HOLD_AFTER_REBOOT is not set
42  CONFIG_AT_SUPPORT=y
43  CONFIG_FILE_SYSTEM_SUPPORT=y
44  CONFIG_UART0_SUPPORT=y
45  CONFIG_UART1_SUPPORT=y
46  # CONFIG_UART2_SUPPORT is not set
47  # end of BSP Settings
```

修改后：

```
33  CONFIG_I2C_SUPPORT=y
34  # CONFIG_I2S_SUPPORT is not set
35  # CONFIG_SPI_SUPPORT is not set
36  # CONFIG_DMA_SUPPORT is not set
37  # CONFIG_SDIO_SUPPORT is not set
38  # CONFIG_SPI_DMA_SUPPORT is not set
39  # CONFIG_UART_DMA_SUPPORT is not set
40  CONFIG_PWM_SUPPORT=y
41  CONFIG_PWM_HOLD_AFTER_REBOOT=y
42  CONFIG_AT_SUPPORT=y
43  CONFIG_FILE_SYSTEM_SUPPORT=y
44  CONFIG_UART0_SUPPORT=y
45  CONFIG_UART1_SUPPORT=y
46  # CONFIG_UART2_SUPPORT is not set
47  # end of BSP Settings
```

第 9 章 WiFi 之 STA 模式连接热点

摘要：本文简单介绍 Hi3861WiFi 操作，怎么连接到热点，查看 IP，ping 服务器等
适合群体：适用于润和 Hi3861 开发板
文中所有代码仓库：https://gitee.com/qidiyun/hihope-3861-smart-home-kit

9.1AT 指令操作 WiFi

我们可以使用 AT 指令进行 Hi3861 WiFi 操作，连接热点、ping 服务器等。

```
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                      - 检查模组与网关的联通性，其中X.X.X.X需替换为实际的网关地址
```

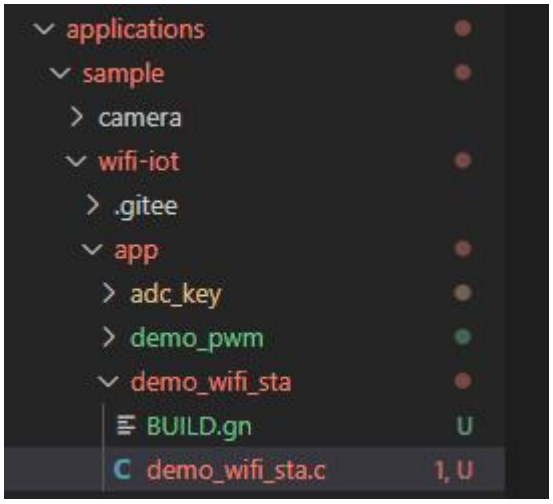
但是很多时候，我们需要实现开机后自动连接到某个热点，光靠 AT 指令不行。

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。

9.2 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 demo_wifi_sta，其中 demo_wifi_sta.c 为业务代码，BUILD.gn 为编译脚本，具体规划目录结构如下：



其中 BUILD.gn 文件内容如下：

```
static_library("demo_wifi_sta") {
    sources = [
        "demo_wifi_sta.c"
```

```
        ]

        include_dirs = [
            "//utils/native/lite/include",
            "//kernel/liteos_m/components/cmsis/2.0",
            "//base/iot_hardware/peripheral/interfaces/kits",

"//device/soc/hisilicon/hi3861v100/hi3861_adapter/hals/communication/wifi_lite/wifiservice",
            "//device/soc/hisilicon/hi3861v100/hi3861_adapter/kal",
            "//device/soc/hisilicon/hi3861v100/sdk_liteos/third_party/lwip_sack/include",
        ]
    }
```

hi_wifi_start_sta 函数：设置 WiFi 参数、扫描热点

```
    int hi_wifi_start_sta(void)
    {
        int ret;
        char ifname[WIFI_IFNAME_MAX_SIZE + 1] = {0};
        int len = sizeof(ifname);
        const unsigned char wifi_vap_res_num = APP_INIT_VAP_NUM;
        const unsigned char wifi_user_res_num = APP_INIT_USR_NUM;
        unsigned int    num = WIFI_SCAN_AP_LIMIT;

        //这里不需要重复进行 WiFi init，因为系统启动后就自己会做 WiFi init
    #if 0
        printf("_______>>>>>>>>>> %s %d \r\n", __FILE__, __LINE__);
        ret = hi_wifi_init(wifi_vap_res_num, wifi_user_res_num);
        if (ret != HISI_OK) {
            return -1;
        }
    #endif
        ret = hi_wifi_sta_start(ifname, &len);
        if (ret != HISI_OK) {
            return -1;
        }

        /* register call back function to receive wifi event, etc scan results event,
          * connected event, disconnected event.
          */
        ret = hi_wifi_register_event_callback(wifi_wpa_event_cb);
        if (ret != HISI_OK) {
            printf("register wifi event callback failed\n");
        }
```

```
    /* acquire netif for IP operation */
    g_lwip_netif = netifapi_netif_find(ifname);
    if (g_lwip_netif == NULL) {
        printf("%s: get netif failed\n", __FUNCTION__);
        return -1;
    }

    /*  开始扫描附件的 WiFi 热点  */
    ret = hi_wifi_sta_scan();
    if (ret != HISI_OK) {
        return -1;
    }

    sleep(5);     /* sleep 5s, waiting for scan result. */

    hi_wifi_ap_info *pst_results = malloc(sizeof(hi_wifi_ap_info) * WIFI_SCAN_AP_LIMIT);
    if (pst_results == NULL) {
        return -1;
    }

    //把扫描到的热点结果存储起来
    ret = hi_wifi_sta_scan_results(pst_results, &num);
    if (ret != HISI_OK) {
        free(pst_results);
        return -1;
    }

    //打印扫描到的所有热点
    for (unsigned int loop = 0; (loop < num) && (loop < WIFI_SCAN_AP_LIMIT); loop++) {
        printf("SSID: %s\n", pst_results[loop].ssid);
    }
    free(pst_results);

    /*  开始接入热点  */
    ret = hi_wifi_start_connect();
    if (ret != 0) {
        return -1;
    }
    return 0;
}
```

连接热点：

```
int hi_wifi_start_connect(void)
```

```
    {
        int ret;
        errno_t rc;
        hi_wifi_assoc_request assoc_req = {0};

        /* copy SSID to assoc_req */
        rc = memcpy_s(assoc_req.ssid, HI_WIFI_MAX_SSID_LEN + 1, "RedmiK40", 8); /* 9:ssid
length */
        if (rc != EOK) {
            return -1;
        }

        //热点加密方式
        assoc_req.auth = HI_WIFI_SECURITY_WPA2PSK;

        /* 热点密码 */
        memcpy(assoc_req.key, "07686582488", 11);

        ret = hi_wifi_sta_connect(&assoc_req);
        if (ret != HISI_OK) {
            return -1;
        }

        return 0;
    }

    热点连接结果回调函数
    void wifi_wpa_event_cb(const hi_wifi_event *hisi_event)
    {
        if (hisi_event == NULL)
            return;

        switch (hisi_event->event) {
            case HI_WIFI_EVT_SCAN_DONE:
                printf("WiFi: Scan results available\n");
                break;
            case HI_WIFI_EVT_CONNECTED:
                printf("WiFi: Connected\n");
                netifapi_dhcp_start(g_lwip_netif);
                break;
            case HI_WIFI_EVT_DISCONNECTED:
                printf("WiFi: Disconnected\n");
                netifapi_dhcp_stop(g_lwip_netif);
                hi_sta_reset_addr(g_lwip_netif);
```

```
                break;
            case HI_WIFI_EVT_WPS_TIMEOUT:
                printf("WiFi: wps is timeout\n");
                break;
            default:
                break;
        }
    }
```

hi_sta_reset_addr：重新复位 sta 的地址、网关等参数。

```
/* clear netif's ip, gateway and netmask */
void hi_sta_reset_addr(struct netif *pst_lwip_netif)
{
    ip4_addr_t st_gw;
    ip4_addr_t st_ipaddr;
    ip4_addr_t st_netmask;

    if (pst_lwip_netif == NULL) {
        printf("hisi_reset_addr::Null param of netdev\r\n");
        return;
    }

    IP4_ADDR(&st_gw, 0, 0, 0, 0);
    IP4_ADDR(&st_ipaddr, 0, 0, 0, 0);
    IP4_ADDR(&st_netmask, 0, 0, 0, 0);

    netifapi_netif_set_addr(pst_lwip_netif, &st_ipaddr, &st_netmask, &st_gw);
}
```

9.3 WiFi 相关 API

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：

```
int hi_wifi_start_connect(void)
{
    int ret;
    errno_t rc;
    hi_wifi_assoc_request assoc_req = {0};

    /* copy SSID to assoc_req */
    rc = memcpy_s(assoc_req.ssid, HI_WIFI_MAX_SSID_LEN + 1, "RedmiK40", 8); /* 9:ssid length */
    if (rc != EOK) {
        return -1;
    }

    //热点加密方式
    assoc_req.auth = HI_WIFI_SECURITY_WPA2PSK;

    /* 热点密码 */
    memcpy(assoc_req.key, "07686582488", 11);

    ret = hi_wifi_sta_connect(&assoc_req);
    if (ret != HISI_OK) {
        return -1;
    }

    return 0;
```

```
    }
```

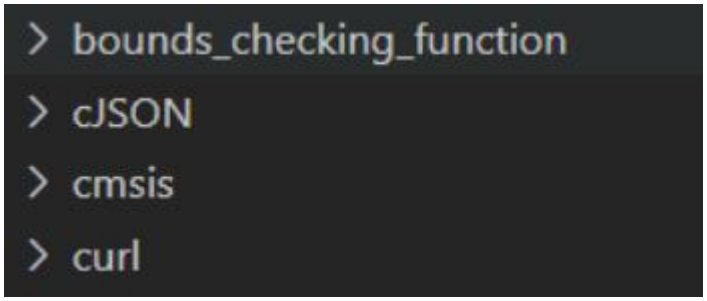
第 10 章 添加软件包

摘要：本文简单介绍 Hi3861WiFi 操作，怎么连接到热点，查看 IP，ping 服务器等
适合群体：适用于润和 Hi3861 开发板
文中所有代码仓库：https://gitee.com/qidiyun/hihope-3861-smart-home-kit

10.1 添加第一个 a_myparty 软件包

打开鸿蒙系统的源码，可以看到有这么一个文件夹：third_party。里面存放的是第三方的代码。

点开我们可以看到有很多第三方代码：



后续我们如果需要往系统中添加、移植任何开源代码，都可以添加到这个文件夹中。接下来，教大家如何添加一个自己的软件包，名字为 a_myparty。

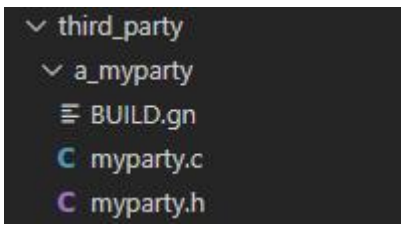
1. 新建一个文件夹 a_myparty

2. 往文件中放置软件包源码

这里我放在的是 myparty.c 和 myparty.h 文件

3. 新建 BUILD.gn 文件

整个代码目录如下：



4. myparty.c 文件内容如下：

其实，我这个只是为了演示的，所以里面代码没什么作用

```
#include <stdio.h>

void myparty_test(void)
{
    printf("first myparty \r\n");
}
```

5. BUILD.gn 文件内容如下：

BUILD.gn 文件主要是描述了软件包的相关信息，包括编译哪些源文件，头文件路径、编译方式（目前 Hi3861 只支持静态加载）

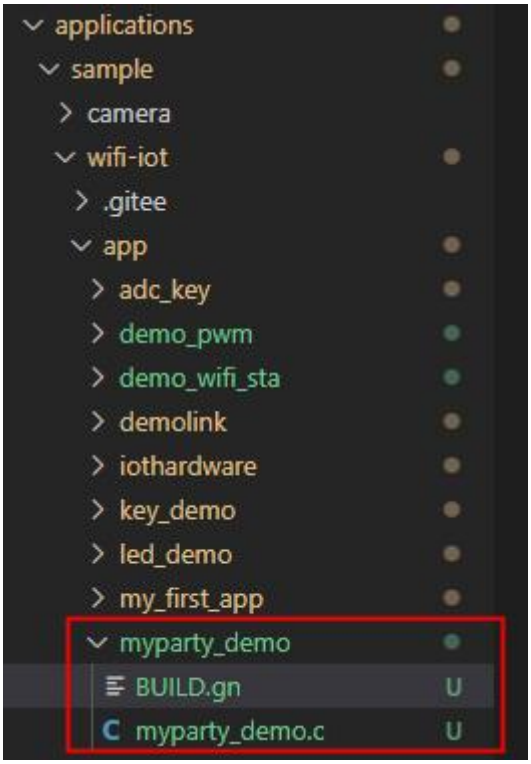
```
import("//build/lite/config/component/lite_component.gni")
import("//build/lite/ndk/ndk.gni")

#这里是配置头文件路径
config("a_myparty_config") {
    include_dirs = [
        ".",
    ]
}

#这里是配置要编译哪些源码
a_myparty_sources = [
    "myparty.c",
]

#这里是静态链接，类似于 Linux 系统的 .a 文件
lite_library("a_myparty_static") {
    target_type = "static_library"
    sources = a_myparty_sources
    public_configs = [ ":a_myparty_config" ]

}

#这里是动态加载，类似于 Linux 系统的 .so 文件
lite_library("a_myparty_shared") {
    target_type = "shared_library"
    sources = a_myparty_sources
    public_configs = [ ":a_myparty_config" ]

}
```

```
#这里是入口，选择是静态还是动态
ndk_lib("a_myparty_ndk") {

    if (kernel_type != "liteos_m") {
        lib_extension = ".so"
        deps = [
            ":a_myparty_shared"
        ]
    } else {
        deps = [
            ":a_myparty_static"
        ]
    }
    head_files = [
        "//third_party/a_myparty"
    ]

}
```

到了这里我们基本上就写完了。
最后我们要让这个第 3 放软件包编译到我们固件中。

10.2 如何使用 a_myparty 软件包

我们在 app 里面新建一个 myparty_demo 的文件夹，目录如下：



其中 BUILD.gn 内容如下：

```
static_library("myparty_demo") {
    sources = [
        "myparty_demo.c"
    ]
#注意需要把 a_myparty 的头文件路径加进来
    include_dirs = [
        "//utils/native/lite/include",
        "//third_party/a_myparty",
    ]

#表示需要 a_myparty  软件包
    deps = [
        "//third_party/a_myparty:a_myparty_static",
    ]
}
```

myparty_demo.c 里面内容如下

```
#include <stdio.h>
#include "ohos_init.h"
#include "ohos_types.h"

#include "myparty.h"

void myparty_demo(void)
{
    myparty_test();
    printf("____________>>>>>>>>>>>>>>>>>>>> [DEMO] Hello world.\n");
}
SYS_RUN(myparty_demo);
```

编译烧录后可以看到 myparty_test 的打印信息：

```
ready to OS start
sdk ver:Hi3861V100R001C00SPC025 2020-09-03 18:10:00
_____>>>>>>> lza app/wifiiot_app/src/app_main.c 406
formatting spiffs...
FileSystem mount ok.
wifi init success!
_____>>>>>>> lza ../../../base/startup/bootstrap_lite/services/source/system_init.c 21
hilog will init.

hievent will init.

hievent init success.
_____>>>>>>> lza ../../../base/startup/bootstrap_lite/services/source/system_init.c 27
first myparty
_____>>>>>>>>>>>>>>>>>>>>[DEMO] Hello world.
_____>>>>>>> lza ../../../base/startup/bootstrap_lite/services/source/system_init.c 29

hiview init success.
No crash dump found!
```

第 11 章 移植 MQTT

摘要：本文简单介绍如何移植 MQTT
适合群体：适用于润和 Hi3861 开发板
文中所有代码仓库：https://gitee.com/qidiyun/hihope-3861-smart-home-kit

10.1 MQTT 介绍

MQTT 是当前最主流的物联网通信协议，需要物联网云平台，例如华为云、阿里云、移动 OneNET 都支持 mqtt。而 Hi3861 则是一款专为 IoT 应用场景打造的芯片。本节主要讲如何在鸿蒙系统中通过移植第 3 方软件包 paho mqtt 去实现 MQTT 协议功能，最后会给出测试验证。为后续的物联网项目打好基础。

友情预告，本节内容较多，源码也贴出来了，大家最好先看一遍，然后再操作一次。

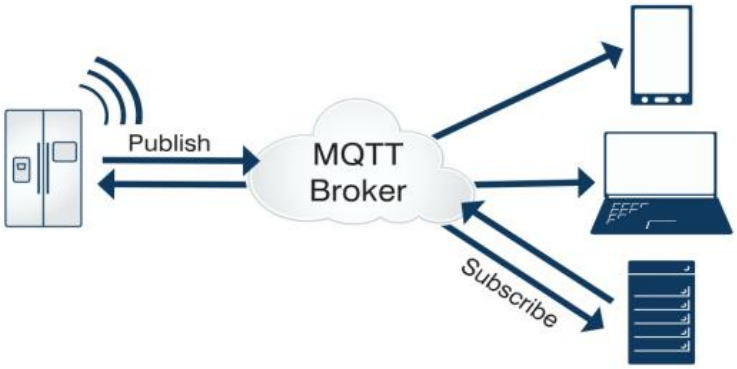
已经移植好的 MQTT 源码：
https://gitee.com/qidiyun/harmony_mqtt

10.2 MQTT 移植

如果不想要自己移植的，可以跳过本节

MQTT 全称为 Message Queuing Telemetry Transport（消息队列遥测传输）是一种基于发布/订阅范式的二进制“轻量级”消息协议，由 IB 公司发布。针对于网络受限和嵌入式设备而设计的一种数据传输协议。MQTT 最大优点在于，可以以极少的代码和有限的带宽，为连接远程设备提供实时可靠的消息服务。作为一种低开销、低带宽占用的即时通讯协议，使其在物联网、小型设备、移动应用等方面有较广泛的应用。MQTT 模型如图所示。

更多 MQTT 协议的介绍见这篇文章： MQTT 协议开发入门



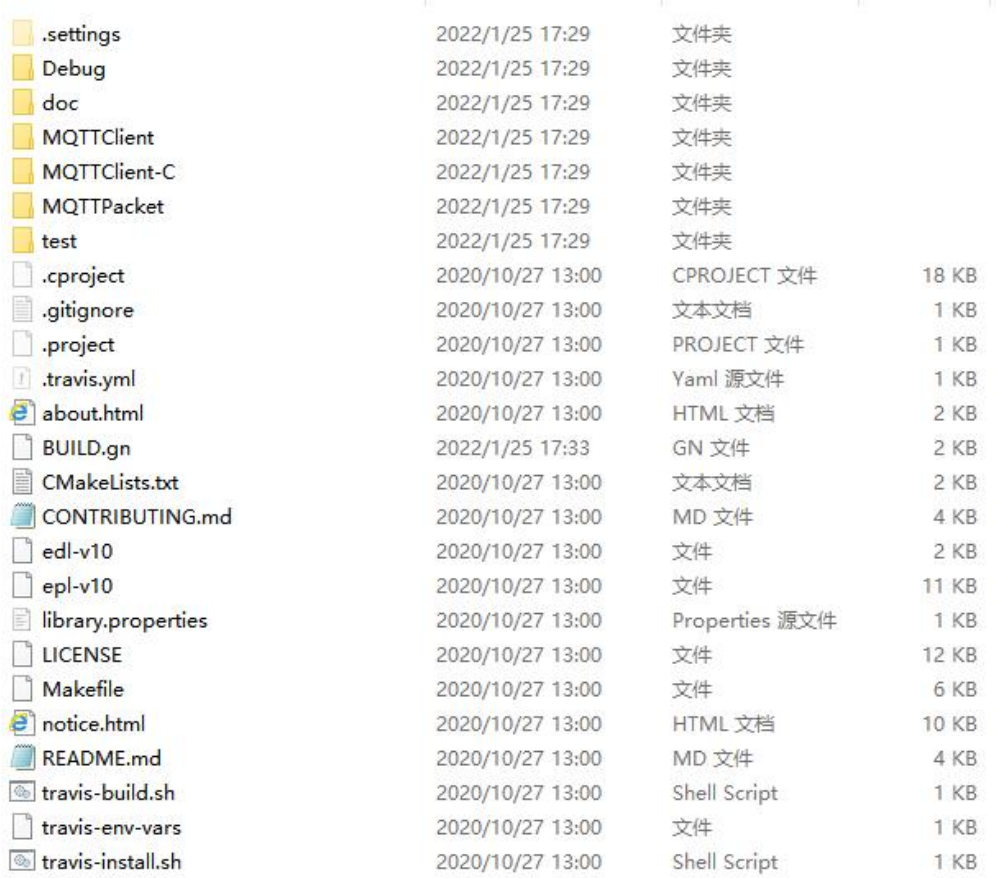
1. 下载 paho mqtt 软件包，添加到鸿蒙代码中

paho mqtt-c 是基于 C 语言实现的 MQTT 客户端，非常适合用在嵌入式设备上。首先下

载源码：

https://github.com/eclipse/paho.mqtt.embedded-c

下载之后解压，会得到这么一个文件夹：



.settings	2022/1/25 17:29	文件夹	
Debug	2022/1/25 17:29	文件夹	
doc	2022/1/25 17:29	文件夹	
MQTTClient	2022/1/25 17:29	文件夹	
MQTTClient-C	2022/1/25 17:29	文件夹	
MQTTPacket	2022/1/25 17:29	文件夹	
test	2022/1/25 17:29	文件夹	
.cproject	2020/10/27 13:00	CPROJECT 文件	18 KB
.gitignore	2020/10/27 13:00	文本文档	1 KB
.project	2020/10/27 13:00	PROJECT 文件	1 KB
.travis.yml	2020/10/27 13:00	Yaml 源文件	1 KB
about.html	2020/10/27 13:00	HTML 文档	2 KB
BUILD.gn	2022/1/25 17:33	GN 文件	2 KB
CMakeLists.txt	2020/10/27 13:00	文本文档	2 KB
CONTRIBUTING.md	2020/10/27 13:00	MD 文件	4 KB
edl-v10	2020/10/27 13:00	文件	2 KB
epl-v10	2020/10/27 13:00	文件	11 KB
library.properties	2020/10/27 13:00	Properties 源文件	1 KB
LICENSE	2020/10/27 13:00	文件	12 KB
Makefile	2020/10/27 13:00	文件	6 KB
notice.html	2020/10/27 13:00	HTML 文档	10 KB
README.md	2020/10/27 13:00	MD 文件	4 KB
travis-build.sh	2020/10/27 13:00	Shell Script	1 KB
travis-env-vars	2020/10/27 13:00	文件	1 KB
travis-install.sh	2020/10/27 13:00	Shell Script	1 KB

我们在鸿蒙系统源码的 third_party 文件夹下创建一个 pahomqtt 文件夹，然后把解压后的所有文件都拷贝到 pahomqtt 文件夹下，目录结构大致如下：

下一步，我们在 pahomqtt 文件夹下面新建 BUILD.gn 文件，用来构建编译。其内容如下：

```
# Copyright (c) 2020 Huawei Device Co., Ltd.
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#        http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

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

config("pahomqtt_config") {
    include_dirs = [
        "MQTTPacket/src",
        "MQTTClient-C/src",
        "MQTTClient-C/src/liteOS",
        "//kernel/liteos_m/components/cmsis/2.0",
    ]
}

pahomqtt_sources = [
"MQTTClient-C/src/liteOS/MQTTLiteOS.c",
"MQTTClient-C/src/MQTTClient.c",

"MQTTPacket/src/MQTTConnectClient.c",
"MQTTPacket/src/MQTTConnectServer.c",
"MQTTPacket/src/MQTTDeserializePublish.c",
"MQTTPacket/src/MQTTFormat.c",
"MQTTPacket/src/MQTTPacket.c",
"MQTTPacket/src/MQTTSerializePublish.c",
"MQTTPacket/src/MQTTSubscribeClient.c",
"MQTTPacket/src/MQTTSubscribeServer.c",
"MQTTPacket/src/MQTTUnsubscribeClient.c",
"MQTTPacket/src/MQTTUnsubscribeServer.c",
]

lite_library("pahomqtt_static") {
    target_type = "static_library"
    sources = pahomqtt_sources
    public_configs = [ ":pahomqtt_config" ]
}

lite_library("pahomqtt_shared") {
    target_type = "shared_library"
    sources = pahomqtt_sources
    public_configs = [ ":pahomqtt_config" ]
}

ndk_lib("pahomqtt_ndk") {
    if (board_name != "hi3861v100") {
        lib_extension = ".so"
```

```
            deps = [
                ":pahomqtt_shared"
            ]
        } else {
            deps = [
                ":pahomqtt_static"
            ]
        }
        head_files = [
            "//third_party/pahomqtt"
        ]
    }
```

2. 移植

我们使用到的是 MQTTClient-C 的代码，该代码支持多线程。

（1）创建 LiteOS 文件夹

MQTT 已经提供了 Linux 和 freertos 的移植，这里我们参考，新建文件夹：

third_party\pahomqtt\MQTTClient-C\src\liteOS

里面存放两个文件：

MQTTLiteOS.c 和 MQTTLiteOS.h

内容如下：

```
#include "MQTTLiteOS.h"

//用来创建线程
int ThreadStart(Thread* thread, void (*fn)(void*), void* arg)
{
    int rc = 0;
    thread = thread;

    osThreadAttr_t attr;

    attr.name = "MQTTTask";
    attr.attr_bits = 0U;
    attr.cb_mem = NULL;
    attr.cb_size = 0U;
    attr.stack_mem = NULL;
    attr.stack_size = 2048;
    attr.priority = osThreadGetPriority(osThreadGetId());

    rc = (int)osThreadNew((osThreadFunc_t)fn, arg, &attr);
```

```
	return rc;
}
//定时器初始化
void TimerInit(Timer* timer)
{
	timer->end_time = (struct timeval){0, 0};
}

char TimerIsExpired(Timer* timer)
{
	struct timeval now, res;
	gettimeofday(&now, NULL);
	timersub(&timer->end_time, &now, &res);
	return res.tv_sec < 0 || (res.tv_sec == 0 && res.tv_usec <= 0);
}


void TimerCountdownMS(Timer* timer, unsigned int timeout)
{
	struct timeval now;
	gettimeofday(&now, NULL);
	struct timeval interval = {timeout / 1000, (timeout % 1000) * 1000};
	timeradd(&now, &interval, &timer->end_time);
}


void TimerCountdown(Timer* timer, unsigned int timeout)
{
	struct timeval now;
	gettimeofday(&now, NULL);
	struct timeval interval = {timeout, 0};
	timeradd(&now, &interval, &timer->end_time);
}


int TimerLeftMS(Timer* timer)
{
	struct timeval now, res;
	gettimeofday(&now, NULL);
	timersub(&timer->end_time, &now, &res);
	//printf("left %d ms\n", (res.tv_sec < 0) ? 0 : res.tv_sec * 1000 + res.tv_usec / 1000);
	return (res.tv_sec < 0) ? 0 : res.tv_sec * 1000 + res.tv_usec / 1000;
}
```

```
void MutexInit(Mutex* mutex)
{
    mutex->sem = osSemaphoreNew(1, 1, NULL);
}

int MutexLock(Mutex* mutex)
{
    return osSemaphoreAcquire(mutex->sem, LOS_WAIT_FOREVER);
}

int MutexUnlock(Mutex* mutex)
{
    return osSemaphoreRelease(mutex->sem);
}

//接受数据
int ohos_read(Network* n, unsigned char* buffer, int len, int timeout_ms)
{
    struct timeval interval = {timeout_ms / 1000, (timeout_ms % 1000) * 1000};
    if (interval.tv_sec < 0 || (interval.tv_sec == 0 && interval.tv_usec <= 0))
    {
        interval.tv_sec = 0;
        interval.tv_usec = 100;
    }

    setsockopt(n->my_socket, SOL_SOCKET, SO_RCVTIMEO, (char *)&interval, sizeof(struct
timeval));

    int bytes = 0;
    while (bytes < len)
    {
        int rc = recv(n->my_socket, &buffer[bytes], (size_t)(len - bytes), 0);
        if (rc == -1)
        {
            if (errno != EAGAIN && errno != EWOULDBLOCK)
                bytes = -1;
            break;
        }
        else if (rc == 0)
        {
            bytes = 0;
```

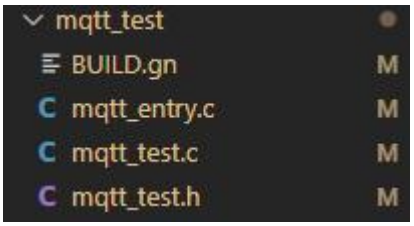
```
                    break;
                }
                else
                    bytes += rc;
            }
            return bytes;
        }

        //写数据
        int ohos_write(Network* n, unsigned char* buffer, int len, int timeout_ms)
        {
            struct timeval tv;

            tv.tv_sec = 0;    /* 30 Secs Timeout */
            tv.tv_usec = timeout_ms * 1000;    // Not init'ing this can cause strange errors

            setsockopt(n->my_socket, SOL_SOCKET, SO_SNDTIMEO, (char *)&tv,sizeof(struct
timeval));
            int    rc = send(n->my_socket, buffer, len, 0);
            return rc;
        }

        //网络初始化
        void NetworkInit(Network* n)
        {
            n->my_socket = 0;
            n->mqttread = ohos_read;
            n->mqttwrite = ohos_write;
        }

        //网络连接
        int NetworkConnect(Network* n, char* addr, int port)
        {
            int type = SOCK_STREAM;
            struct sockaddr_in address;
            int rc = -1;
            sa_family_t family = AF_INET;
            struct addrinfo *result = NULL;
            struct addrinfo hints = {0, AF_UNSPEC, SOCK_STREAM, IPPROTO_TCP, 0, NULL, NULL,
NULL};

            if ((rc = getaddrinfo(addr, NULL, &hints, &result)) == 0)
            {
                struct addrinfo* res = result;
```

```
        /* prefer ip4 addresses */
        while (res)
        {
            if (res->ai_family == AF_INET)
            {
                result = res;
                break;
            }
            res = res->ai_next;
        }

        if (result->ai_family == AF_INET)
        {
            address.sin_port = htons(port);
            address.sin_family = family = AF_INET;
            address.sin_addr = ((struct sockaddr_in*)(result->ai_addr))->sin_addr;
        }
        else
            rc = -1;

        freeaddrinfo(result);
    }

    if (rc == 0)
    {
        n->my_socket = socket(family, type, 0);
        if (n->my_socket != -1)
            rc = connect(n->my_socket, (struct sockaddr*)&address, sizeof(address));
        else
            rc = -1;
    }

    return rc;
}


void NetworkDisconnect(Network* n)
{
    close(n->my_socket);
}
```

至此我们移植基本结束

10.3 测试代码

测试代码比较好写。主要是 3 个文件，内容我都贴出来了：



（1）BUILD.gn 文件内容：

```
static_library("mqtt_test") {
    sources = [
        "mqtt_test.c",
        "mqtt_entry.c"
    ]

    include_dirs = [
        "//utils/native/lite/include",
        "//kernel/liteos_m/components/cmsis/2.0",
        "//base/iot_hardware/interfaces/kits/wifiiot_lite",
        "//vendor/hisi/hi3861/hi3861/third_party/lwip_sack/include",
        "//foundation/communication/interfaces/kits/wifi_lite/wifiservice",
        "//third_party/pahomqtt/MQTTPacket/src",
        "//third_party/pahomqtt/MQTTClient-C/src",
        "//third_party/pahomqtt/MQTTClient-C/src/liteOS",
    ]
#表示需要 a_myparty  软件包
    deps = [
        "//third_party/pahomqtt:pahomqtt_static",
    ]
}
```

（2）mqtt_entry.c 文件主要是进行热点连接，因为我们要使用 MQTT 需要用到网络。热点连接的代码之前在第 9 章已经讲说，这里就不完全贴了，代码仓库也有，主要的代码部分：

```
void wifi_sta_task(void *arg)
{
    arg = arg;

    //连接热点
```

```
    hi_wifi_start_sta();

    while(wifi_ok_flg == 0)
    {
        usleep(30000);
    }


    usleep(2000000);

    //开始进入 MQTT 测试
    mqtt_test();
}
```

（3）mqtt_test.c 文件则是编写了一个简单的 MQTT 测试代码

其中测试用的 mqtt 服务器是我自己的服务器：5.196.95.208
大家也可以改成自己的。

```
#include <stdio.h>
#include <unistd.h>
#include "ohos_init.h"
#include "cmsis_os2.h"

#include "hi_wifi_api.h"
#include "lwip/ip_addr.h"
#include "lwip/netifapi.h"
#include "lwip/sockets.h"

#include "MQTTClient.h"


static MQTTClient mq_client;

  unsigned char *onenet_mqtt_buf;
  unsigned char *onenet_mqtt_readbuf;
  int buf_size;

Network n;
MQTTPacket_connectData data = MQTTPacket_connectData_initializer;

//消息回调函数
void mqtt_callback(MessageData *msg_data)
```

```
    {
        size_t res_len = 0;
        uint8_t *response_buf = NULL;
        char topicname[45] = { "$crsp/" };

        LOS_ASSERT(msg_data);

        printf("topic %.*s receive a message\r\n", msg_data->topicName->lenstring.len,
msg_data->topicName->lenstring.data);

        printf("message is %.*s\r\n", msg_data->message->payloadlen,
msg_data->message->payload);

    }

    int mqtt_connect(void)
    {
        int rc = 0;

        NetworkInit(&n);
        NetworkConnect(&n, "5.196.95.208", 1883);

        buf_size    = 2048;
        onenet_mqtt_buf = (unsigned char *) malloc(buf_size);
        onenet_mqtt_readbuf = (unsigned char *) malloc(buf_size);
        if (!(onenet_mqtt_buf && onenet_mqtt_readbuf))
        {
            printf("No memory for MQTT client buffer!");
            return -2;
        }

        MQTTClientInit(&mq_client, &n, 1000, onenet_mqtt_buf, buf_size,
onenet_mqtt_readbuf, buf_size);

        MQTTStartTask(&mq_client);


        data.keepAliveInterval = 30;
        data.cleansession = 1;
        data.clientID.cstring = "ohos_hi3861";
        data.username.cstring = "123456";
        data.password.cstring = "222222";

        data.keepAliveInterval = 10;
```

```
    data.cleansession = 1;

    mq_client.defaultMessageHandler = mqtt_callback;

    //连接服务器
    rc = MQTTConnect(&mq_client, &data);

    //订阅消息，并设置回调函数
    MQTTSubscribe(&mq_client, "ohossub", 0, mqtt_callback);

    while(1)
    {
        MQTTMessage message;

        message.qos = QOS1;
        message.retained = 0;
        message.payload = (void *)"openharmony";
        message.payloadlen = strlen("openharmony");

        //发送消息
        if (MQTTPublish(&mq_client, "ohospub", &message) < 0)
        {
            return -1;
        }
    }

    return 0;
}


void mqtt_test(void)
{
    mqtt_connect();
}
```

到这里就完成了代码部分，可以开始编译了。

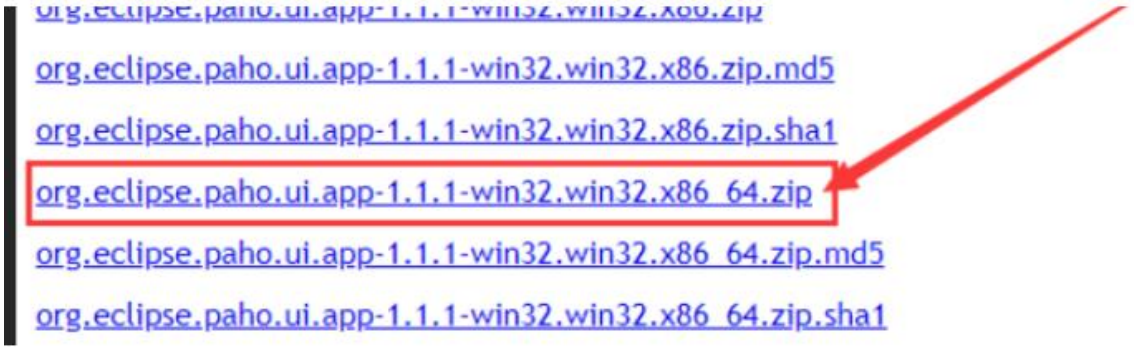
10.4 实验

这里我们需要先下载一个 Windows 电脑端的 MQTT 客户端，这样我们就可以用电脑订

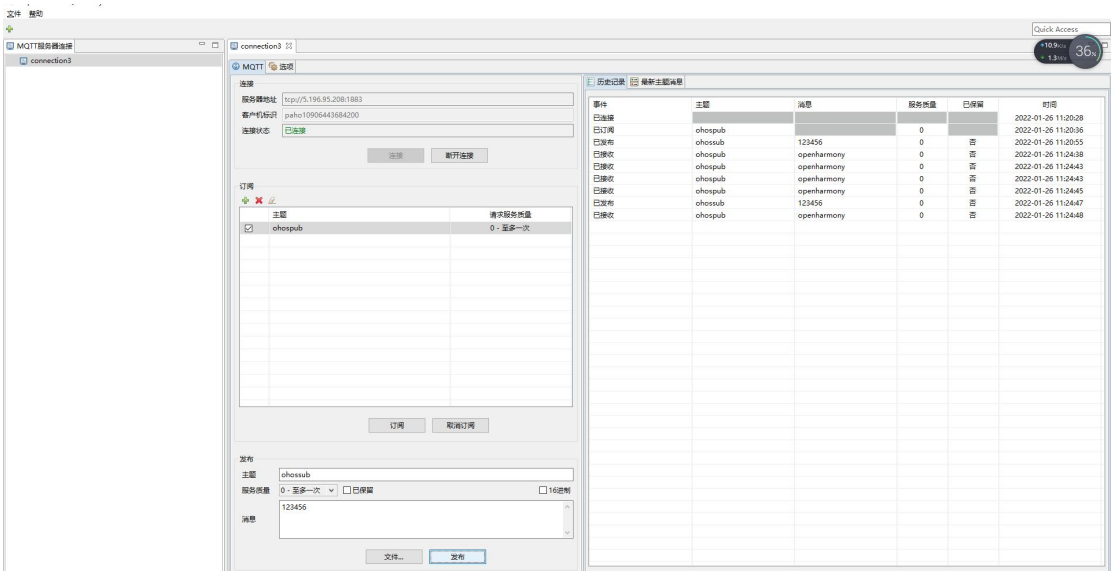
阅开发板的 MQTT 主题信息了。

电脑版的 mqtt 客户端下载链接：
https://repo.eclipse.org/content/repositories/paho-releases/org/eclipse/paho/org.eclipse.paho.ui.app/1.1.1/

我们选择这一个：

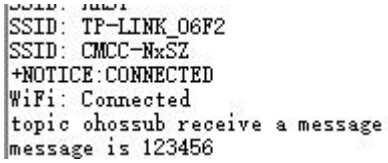


弄完后打开软件，按图操作：



此时我们去查看 我们电脑端的 MQTT 客户端软件，可以看到右边已经有接收 MQTT 信息了，主题未 ohospub，消息内容为 openharmony，说明实验成功。

电脑发送主题为 ohossub，内容为 123456，查看串口打印，可以看到也收到了数据



第 12 章 OneNET 云接入

摘要：本文简单介绍如何接入 OneNET 云平台
适合群体：适用于润和 Hi3861 开发板
文中所有代码仓库：https://gitee.com/qidiyun/hihope-3861-smart-home-kit

12.1 OneNET 云介绍

通常来说，一个物联网产品应当包括设备、云平台、手机 APP。我将在鸿蒙系统上移植 MQTT 协议、OneNET 接入协议，实现手机 APP、网页两者都可以远程（跨网络，不是局域网的）访问开发板数据，并控制开发板的功能。

理论上来说，任何以 MQTT 协议为基础的物联网云平台都可以支持接入。

关于 phomqtt 和 onenet 软件包，已提供下载，声明：所有源码均遵守开源协议~~。

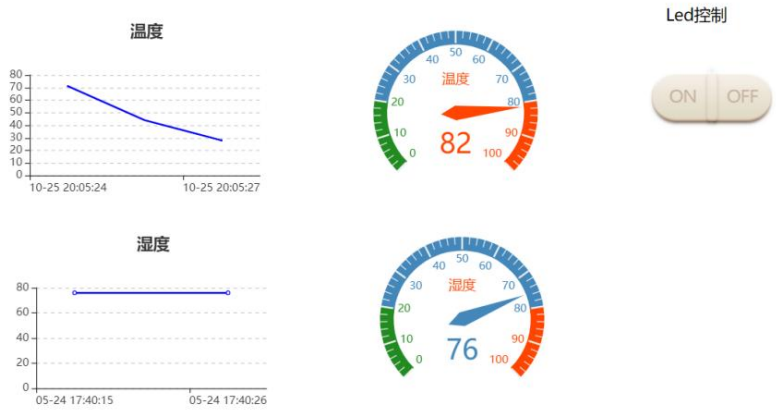
支持鸿蒙系统的 harmony_mqtt 代码仓库:
https://gitee.com/qidiyun/harmony_mqtt

支持鸿蒙系统的 onenet 接入软件包仓库：

https://gitee.com/qidiyun/harmony_onenet

12.2 效果演示

先看下效果，我这边使用的是 OneNET 物联云平台，进入应用，可以看到如下网页界面。该网页的温度、湿度数据由 鸿蒙开发板（Hi3861）上传，同时有一个开关按钮，可以控制开发板的 LED 灯。



另外，也提供一个手机 APP，

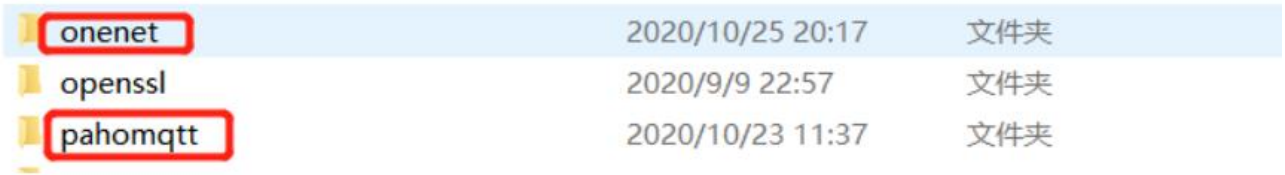
以上界面比较简陋，但不妨碍我们使用，另外选择 OneNET 云平台的主要是原因是接入方式比较简单方便，易于学习，另外一个是 OneNET 提供了物联网云平台、手机 APP，不需要大家自己再去实现，可以更多地将注意力放在鸿蒙系统开发上。

12.3 OneNET 软件包

我这边已经将 mqtt 和 onenet 以软件包的形式发布，两个软件包分别是

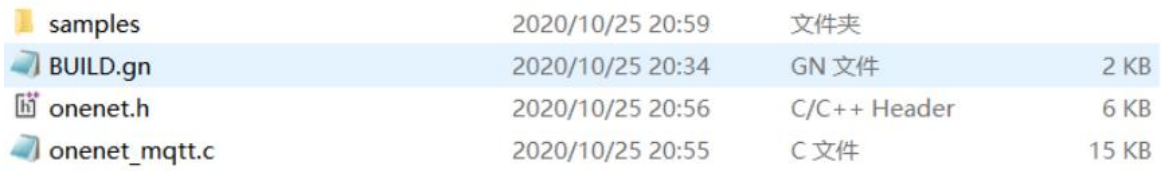
（1）onenet——实现 onenet 接入能力

（2）pahomqtt——实现 MQTT 协议功能



只需要将这两个软件包放到 third_party 文件夹下即可。然后修改

我们来看下 onenet 文件夹：



其中 onenet.h 是头文件

onenet_mqtt.c 是全部源码，它基于 paho mqtt 的 MQTTClient 编程模型。

另外 samples 文件夹下是一个示例代码，代码内容如下：

```
#include <stdio.h>
#include <unistd.h>
#include "MQTTClient.h"
#include "onenet.h"

#define ONENET_INFO_DEVID "597952816"
#define ONENET_INFO_AUTH "202005160951"
#define ONENET_INFO_APIKEY "zgQdlB5y3Bi9pNd2bUYmS8TJHIY="
#define ONENET_INFO_PROID "345377"
#define ONENET_MASTER_APIKEY "gwaK2wJT5wgnSbJYz67CVRGvwkI="

extern int rand(void);


void onenet_cmd_rsp_cb(uint8_t *recv_data, size_t recv_size, uint8_t **resp_data, size_t
*resp_size)
{
    printf("recv data is %.*s\n", recv_size, recv_data);

    *resp_data = NULL;
    *resp_size = 0;
}

int mqtt_test(void)
{

    device_info_init(ONENET_INFO_DEVID, ONENET_INFO_PROID, ONENET_INFO_AUTH,
ONENET_INFO_APIKEY, ONENET_MASTER_APIKEY);
    onenet_mqtt_init();

    onenet_set_cmd_rsp_cb(onenet_cmd_rsp_cb);

    while (1)
    {
        int value = 0;

        value = rand() % 100;

        if (onenet_mqtt_upload_digit("temperature", value) < 0)
```

```
            {
                printf("upload has an error, stop uploading");
                //break;
            }
            else
            {
                printf("buffer : {\"temperature\":%d} \r\n", value);
            }
            sleep(1);
        }
        return 0;
}
```

手机 APP 下载：https://open.iot.10086.cn/doc/art656.html#118

12.4 OneNET 平台使用

首先我们要注册账号，OneNET 平台地址：

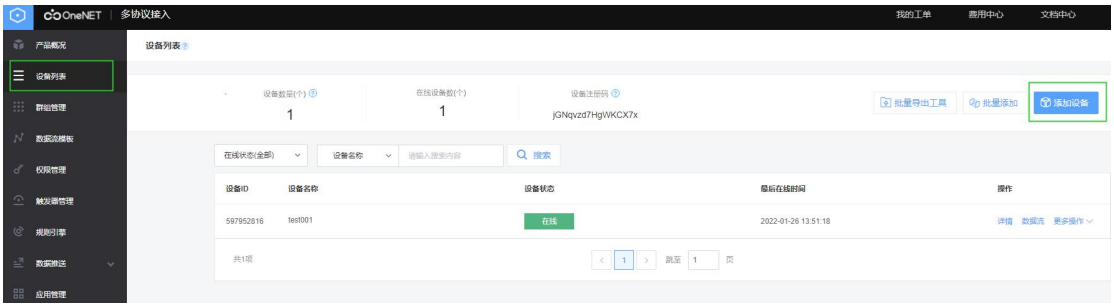
https://open.iot.10086.cn/

然后进入控制台，鼠标放在全部产品服务，选择多协议接入

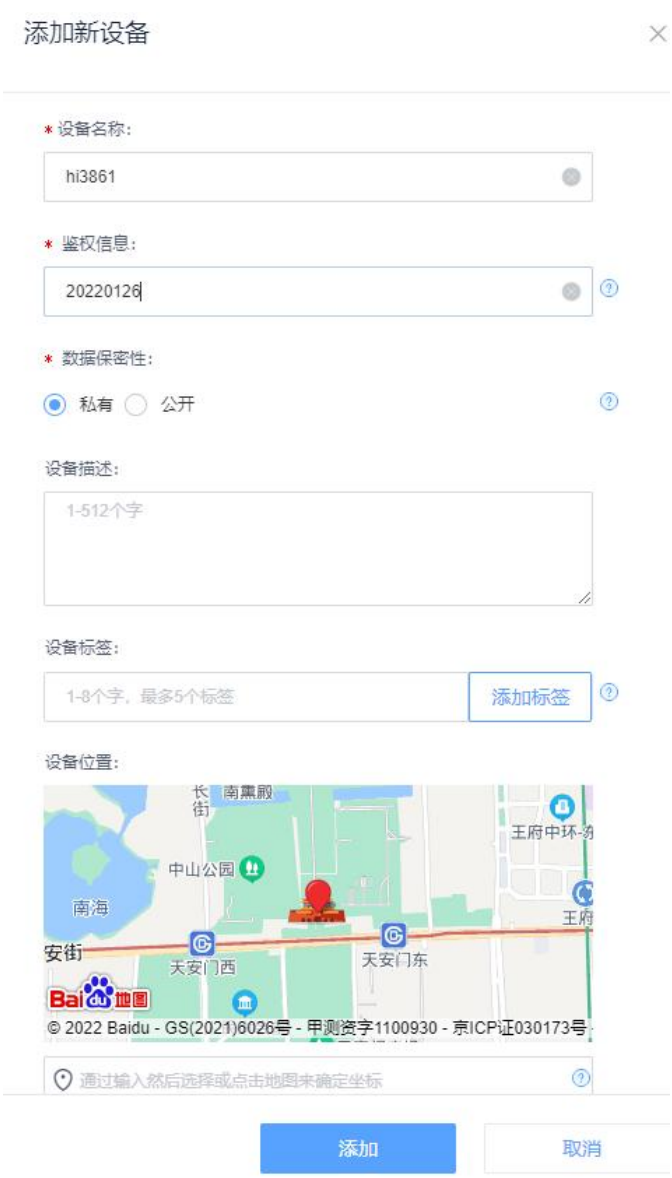
选择 MQTT 旧版，单击创建产品

这里我们可以按照我们的需求填写相关信息即可

之后可以选择添加设备



可以填写信息和填写地理位置，鉴权信息可以随意填写



12.5 OneNET 设备信息

代码中，我们需要填写以下认证信息：

```
#define ONENET_INFO_DEVID "597952816"
#define ONENET_INFO_AUTH "202005160951"
#define ONENET_INFO_APIKEY "zgQdlB5y3Bi9pNd2bUYmS8TJHIY="
#define ONENET_INFO_PROID "345377"
#define ONENET_MASTER_APIKEY "gwaK2wJT5wgnSbJYz67CVRGvwkI="
```

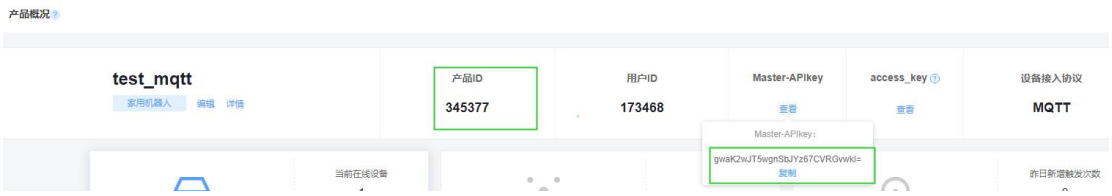
（1）ONENET_INFO_DEVID 和 ONENET_INFO_AUTH
设备 ID 和鉴权信息，可以通过查看设备详情得到：

（2）ONENET_INFO_APIKEY

Api key，可以通过添加 api key 得到

（3）ONENET_INFO_PROID 和 ONENET_MASTER_APIKEY

这个可以查看产品 ID 和 master key



第 13 章 鸿蒙小车开发

摘要：本文简单介绍鸿蒙系统 + Hi3861 的 WiFi 小车开发

适合群体：适用于润和 Hi3861 开发板

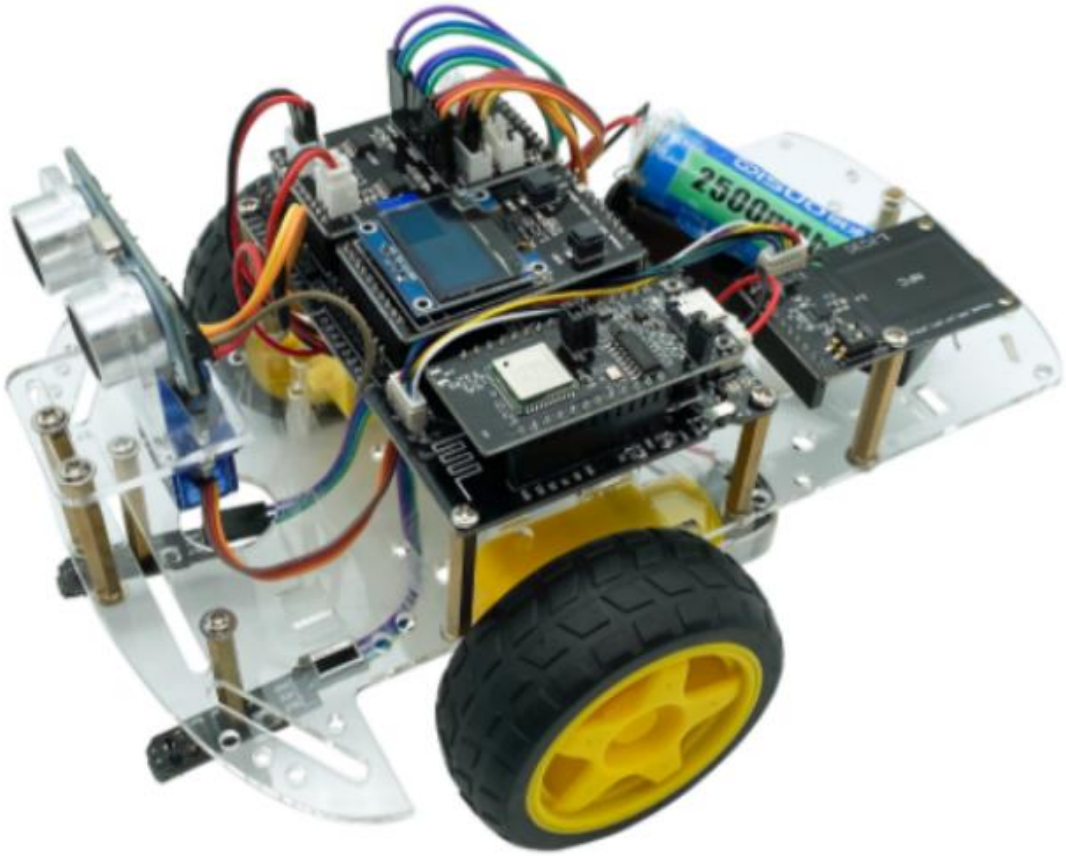
文中所有代码仓库：https://gitee.com/qidiyun/hihope-3861-smart-home-kit

13.1 小车介绍

基于鸿蒙系统 + Hi3861 的 WiFi 小车

首先，我们得有一套 WiFi 小车套件，其实也是 Hi3861 加上电机、循迹模块、超声波等模块。

小车安装完大概是这样：



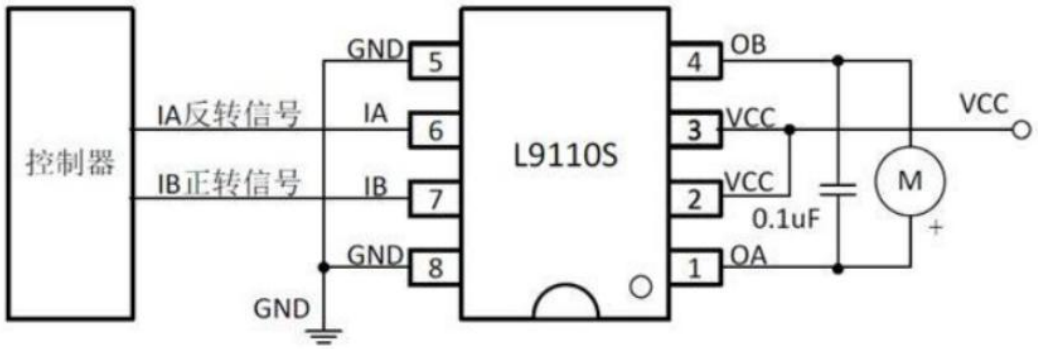
HiSpark智能小车IoT开发板套

13.2 电机驱动

我们这里先只做最简单的，驱动小车的电机，让小车跑起来。

电机的驱动板如下图，目前电机驱动芯片用的是 L9110S 芯片。

典型的应用电路如下图：



我们可以看到，如果要控制电机，我们芯片至少需要 2 路 GPIO 信号，一路用于控制正转，一路用于控制反转。

然后我们小车有两个轮子，需要两个电机，所以我们需要 4 路 GPIO 信号。

查阅小车资料，可以知道，目前 Hi3861 芯片用来控制电机的 4 路 GPIO 分别是：

电机	GPIO 口	PWM 通道
电机 1	GPIO 0	PWM 3
	GPIO 1	PWM 4
电机 2	GPIO 9	PWM 0
	GPIO 10	PWM 1

基于鸿蒙系统 + Hi3861 的 wifi 小车，可以通过电脑、手机控制-鸿蒙 HarmonyOS 技术社区
知道了 PWM 通道和对应的 GPIO 口，我们就可以开始编程了。

首先 PWM 初始化部分：
需要注意，实际代码并没有使用 PWM，而是直接 IO 控制。

```
void pwm_init(void)
{
    IoTGpioInit(IO_NAME_GPIO_0);
    IoTGpioInit(IO_NAME_GPIO_1);
    IoTGpioInit(IO_NAME_GPIO_9);
    IoTGpioInit(IO_NAME_GPIO_10);

    //寮曞彂澶嶄敤
    //hi_io_set_func(IO_NAME_GPIO_0, 0);
    //hi_io_set_func(IO_NAME_GPIO_1, 0);
    //hi_io_set_func(IO_NAME_GPIO_9, 0);
    //hi_io_set_func(IO_NAME_GPIO_10, 0);
}

//鍋滄
void pwm_stop(void)
{
    //鍏堝仠姝   WM
    gpio_control(GPIO0, IOT_GPIO_VALUE1);
    gpio_control(GPIO1, IOT_GPIO_VALUE1);
    gpio_control(GPIO9, IOT_GPIO_VALUE1);
    gpio_control(GPIO10, IOT_GPIO_VALUE1);
}
void car_stop(void)
{
    car_info.cur_status = car_info.go_status;

    printf("pwm_stop \r\n");

    pwm_stop();
```

```
}

//鍖將繘
void pwm_forward(void)
{
    //鍏堝仠姝    WM
    gpio_control(GPIO0, IOT_GPIO_VALUE1);
    gpio_control(GPIO1, IOT_GPIO_VALUE0);
    gpio_control(GPIO9, IOT_GPIO_VALUE1);
    gpio_control(GPIO10, IOT_GPIO_VALUE0);

    //鍚    姘 A 璺痔 WM
    //左轮
    //IoTPwmStart(PWM_PORT_PWM3, 64000, 64000);
    //右轮
    //IoTPwmStart(PWM_PORT_PWM0, 64000, 64000);
}
void car_forward(void)
{
    if(car_info.go_status != CAR_STATUS_FORWARD)
    {
        //鐩存帴闆€鍑?
        return ;
    }
    if(car_info.cur_status == car_info.go_status)
    {
        //鐘舵€佹病鏈夊彉鍖栭鐩存帴鎶ㄥ懡
        return;
    }

    car_info.cur_status = car_info.go_status;

    printf("pwm_forward \r\n");

    pwm_forward();

    step_count_update();
}

//鍚庡€€
void pwm_backward(void)
{
    //鍏堝仠姝    WM
    gpio_control(GPIO0, IOT_GPIO_VALUE0);
```

```
        gpio_control(GPIO1, IOT_GPIO_VALUE1);
        gpio_control(GPIO9, IOT_GPIO_VALUE0);
        gpio_control(GPIO10, IOT_GPIO_VALUE1);

        //鍚　姘 A 璽疿 WM
        //IoTPwmStart(PWM_PORT_PWM4, 64000, 64000);
        //IoTPwmStart(PWM_PORT_PWM1, 64000, 64000);
    }
    void car_backward(void)
    {
        if(car_info.go_status != CAR_STATUS_BACKWARD)
        {
            //鐩存帴闆€鍑?
            return ;
        }
        if(car_info.cur_status == car_info.go_status)
        {
            //鐘舵€侀病鏈夊彉鍖栫鐩存帴鎶ㄥ懡
            return;
        }

        car_info.cur_status = car_info.go_status;

        printf("pwm_backward \r\n");

        pwm_backward();

        step_count_update();
    }

    //宸﹁浆
    void pwm_left(void)
    {
        //鍏堝仠姝　WM
        gpio_control(GPIO0, IOT_GPIO_VALUE0);
        gpio_control(GPIO1, IOT_GPIO_VALUE0);
        gpio_control(GPIO9, IOT_GPIO_VALUE1);
        gpio_control(GPIO10, IOT_GPIO_VALUE0);

        //鍚　姘 A 璽疿 WM
        //IoTPwmStart(PWM_PORT_PWM0, 64000, 64000);


    }
```

```
void car_left(void)
{
	if(car_info.go_status != CAR_STATUS_LEFT)
	{
		//鏈存帴閫€鍑?
		return ;
	}
	if(car_info.cur_status == car_info.go_status)
	{
		//鐘舵€佹病鏈夊彉鍖栵紝鐩存帴鎺ㄩ嚭
		return;
	}

	car_info.cur_status = car_info.go_status;

	printf("pwm_left \r\n");

	pwm_left();

	step_count_update();
}

//鍙宠浆
void pwm_right(void)
{
	//鍏堝仠姝    WM
	gpio_control(GPIO0, IOT_GPIO_VALUE1);
	gpio_control(GPIO1, IOT_GPIO_VALUE0);
	gpio_control(GPIO9, IOT_GPIO_VALUE0);
	gpio_control(GPIO10, IOT_GPIO_VALUE0);

	//鍚    姩 A 疊痔 WM
	//IoTPwmStart(PWM_PORT_PWM3, 64000, 64000);
}
void car_right(void)
{
	if(car_info.go_status != CAR_STATUS_RIGHT)
	{
		//鏈存帴閫€鍑?
		return ;
	}
	if(car_info.cur_status == car_info.go_status)
	{
		//鐘舵€佹病鏈夊彉鍖栵紝鐩存帴鎺ㄩ嚭
```

```
            return;
        }

        car_info.cur_status = car_info.go_status;

        printf("pwm_right \r\n");

        pwm_right();

        step_count_update();
    }
```

如果要使用 pwm 功能，我们需要修改
device/soc/hisilicon/hi3861v100/sdk_liteos/build/config/usr_config.mk
增加这两行，这里是打开 PWM 功能
CONFIG_PWM_SUPPORT=y
CONFIG_PWM_HOLD_AFTER_REBOOT=y

```
33  # CONFIG_I2C_SUPPORT is not set
34  # CONFIG_I2S_SUPPORT is not set
35  # CONFIG_SPI_SUPPORT is not set
36  # CONFIG_DMA_SUPPORT is not set
37  # CONFIG_SDIO_SUPPORT is not set
38  # CONFIG_SPI_DMA_SUPPORT is not set
39  # CONFIG_UART_DMA_SUPPORT is not set
40  CONFIG_PWM_SUPPORT=y
41  CONFIG_PWM_HOLD_AFTER_REBOOT=y
42  CONFIG_AT_SUPPORT=y
43  CONFIG_FILE_SYSTEM_SUPPORT=y
44  CONFIG_UART0_SUPPORT=y
45  CONFIG_UART1_SUPPORT=y
46  # CONFIG_UART2_SUPPORT is not set
47  # end of BSP Settings
```

13.3 WiFi 控制部分

我们在小车上面简单编写一个 UDP 程序，监听 50001 端口号。这里使用的通信格式是 json，小车收到 UDP 数据后，解析 json，并根据命令执行相应的操作，例如前进、后退、左转、右转等，代码如下：

```
if(strcmp("forward", cJSON_GetObjectItem(recvjson, "cmd")->valuestring) == 0)
{
    set_car_status(CAR_STATUS_FORWARD);
    printf("forward\r\n");
}

if(strcmp("backward", cJSON_GetObjectItem(recvjson, "cmd")->valuestring) == 0)
{
    set_car_status(CAR_STATUS_BACKWARD);
    printf("backward\r\n");
}

if(strcmp("left", cJSON_GetObjectItem(recvjson, "cmd")->valuestring) == 0)
{
    set_car_status(CAR_STATUS_LEFT);
    printf("left\r\n");
}

if(strcmp("right", cJSON_GetObjectItem(recvjson, "cmd")->valuestring) == 0)
{
    set_car_status(CAR_STATUS_RIGHT);
    printf("right\r\n");
}

if(strcmp("stop", cJSON_GetObjectItem(recvjson, "cmd")->valuestring) == 0)
{
    set_car_status(CAR_STATUS_STOP);
    printf("stop\r\n");
}
```

电脑端，使用 C#编写一个测试程序，可以手动输入小车的 IP 地址，也可以不输入 IP 地址，这样，电脑端程序会发送广播包给小车，也可以起到控制的功能。

13.4 WiFi 热点连接

注意，我这里 WiFi 小车固件默认连接热点的 ssid 和密码是：

```
20  #define WIFI_SSID       "12-203"
21  #define WIFI_PASSWD     "07686582488"
```

读者需要自己修改成自己的热点。

文件：sta_entry.c

小车的源码，C#控制端的代码均开源，大家可以自由修改，发挥自己的想象，创造出更厉害炫酷的 DIY 产品。

第 14 章 语音控制鸿蒙小车

摘要：本文简单介绍如何使用语音控制鸿蒙小车
适合群体：适用于润和 Hi3861 开发板
文中所有代码仓库：https://gitee.com/qidiyun/hihope-3861-smart-home-kit

14.1 讯飞语音识别

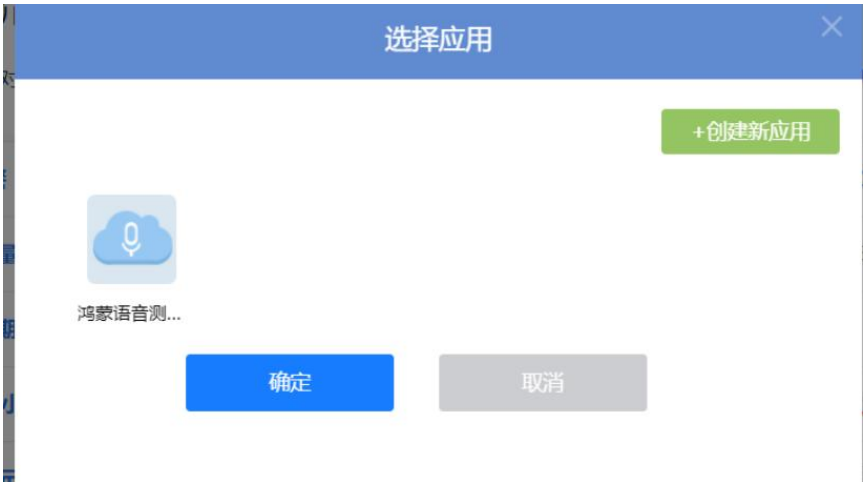
这里我们使用到的是讯飞的语音识别功能，大家可以打开这个网站，申请一个测试账户：
https://www.xfyun.cn/services/lfasr?ch=bd01-b&b_scene_zt=1&renqun_youhua=648371
一般来说我们申请体验包即可，（新用户礼包需要实名认证）：

{ 产品价格 }

以下套餐针对开发者用户调用接口时使用。如果您是个人用

套餐	体验包	新用户礼包
时长量	5小时	最高50小时
有效期	30天	一年
单价（元/小时）	免费	免费
总价（元）	免费	免费
使用服务	立即领取	立即领取

领取完免费使用后，我们创建新应用。



应用名称这些自己根据需求填写

提交后，我们单击应用，查看详情

我们下载 Android SDK 包。

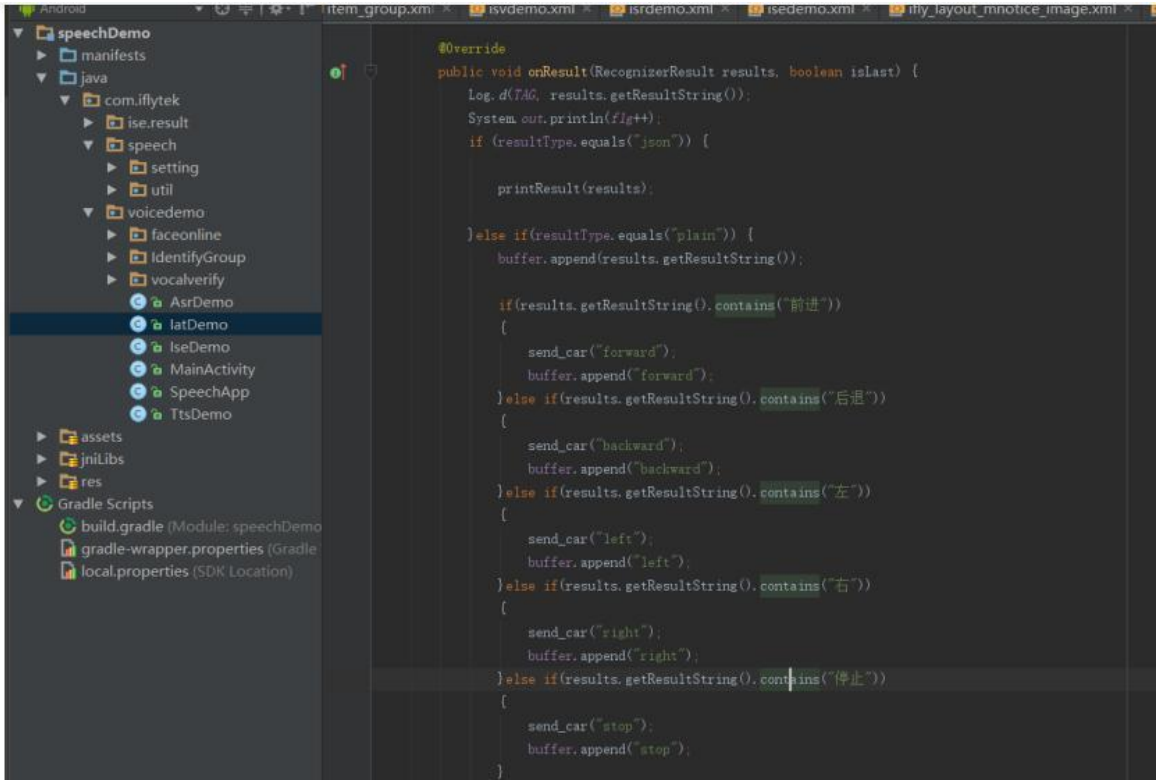
语音听写（流式版）SDK

SDK名称	版本	操作
Android MSC	1140	下载 文档
iOS MSC	1180	下载 文档
Linux MSC	1227	下载 文档
Windows MSC	1126	下载 文档
Java MSC	1021	下载 文档

Android SDK 包的使用可以查看文档。

下载完后，我们在 IatDemo.java 文件的 public void onResult(RecognizerResult results, boolean isLast) 函数中添加我们控制小车的代码，如图：

我这边会提供我修改后的 IatDemo.java 文件 ，大家替换即可。



编译 app，然后得到安装包：speechDemo-debug.apk 。安装到手机。

安装后，我们选择“立即体验语音听写”，然后单击开始，说出关键字“前进”“后退”“向左”“向右”，即可看到小车做出相应的动作


立刻体验语音听写
立刻体验语法识别
立刻体验语义理解
立刻体验语音合成
立刻体验语音评测
立刻体验语音唤醒
立刻体验声纹密码
立刻体验人脸识别


讯飞听写示例
听写结果显示
上传联系人/词表显示
开始
停止
取消
点击设置小语种,当前语种：中文(普通话)
音频流识别

代码解析：

其中比较重要的是发送小车控制指令，指令我们采用的是 json 格式，大家也可以根据自己需求，修改其它指令。

```
void send_car(final String msg)
 {
  clientThread = new Thread(new Runnable() {
   @Override
   public void run() {
    JSONObject address = new JSONObject();
    try {
     address.put("cmd", msg);
     address.put("mode", "step");
    } catch (JSONException e) {
     e.printStackTrace();
    }
```

```
        try {

         InetAddress targetAddress =
InetAddress.getByName("192.168.1.103");



         DatagramPacket packet = new
DatagramPacket(address.toString().getBytes(),
address.toString().length(), targetAddress, 50001);

         client.send(packet);

        } catch (IOException e) {

         e.printStackTrace();

        }

       }

      });

      clientThread.start();

     }
```