

# **NB-IoT 实训平台**

# **说明书**

## **V1.0**

# 目录

1.	NB-IoT 技术简介.....	1
2.	硬件设备说明 .....	3
2.1.	NB-IoT 节点.....	3
2.1.1.	NB-IoT 终端节点.....	3
2.1.2.	NB-IoT 开发板.....	5
2.2.	NB-IoT 本地服务器 .....	7
2.3.	加速度传感器模块 .....	8
2.4.	光照传感器模块.....	9
2.5.	超声波传感器模块 .....	10
2.6.	磁阻传感器模块.....	11
2.7.	温湿度传感器模块 .....	12
2.8.	GPS 模块 .....	13
2.9.	NB-IoT 通信模块 .....	14
2.10.	蜂鸣器模块 .....	15
2.11.	其他配件.....	16
3.	STM32 基础应用 .....	17
3.1.	工程配置简介 .....	17
3.2.	时钟配置示例 .....	19
3.3.	通用 I/O 配置示例.....	21
3.4.	实时时钟配置示例 .....	22
3.5.	独立看门狗配置示例.....	24
4.	传感器及响应模块应用 .....	26
4.1.	工程配置简介 .....	26
4.2.	加速度传感器应用示例 .....	28
4.3.	光照传感器应用示例.....	30
4.4.	超声波传感器应用示例 .....	33
4.5.	磁阻传感器应用示例.....	35
4.6.	蜂鸣器应用示例.....	36
4.7.	三色 LED 灯应用示例.....	37
5.	NB-IoT 技术应用.....	39
5.1.	NB-IoT 协议应用简介.....	39
5.1.1.	网络 .....	39
5.1.2.	物理层 .....	41
5.1.3.	小区接入.....	50
5.1.4.	数据传输.....	53
5.2.	NB-IoT 节点上行通讯示例 .....	55
5.2.1.	AT 指令调试 .....	55
5.2.2.	应用代码介绍.....	58
5.3.	NB-IOT 节点下行通讯示例 .....	66
5.3.1.	AT 指令调试 .....	66
5.3.2.	应用代码介绍.....	70

---

5.4.	NB-IoT Web 客户端配置及使用示例 .....	74
5.4.1.	Web 客户端简介 .....	74
5.4.2.	Web 客户端配置使用 .....	76
5.5.	基于 NB-IoT 技术的设备控制示例 .....	79
5.5.1.	节点配置使用 .....	79
5.5.2.	应用代码介绍 .....	82
5.6.	NB-IoT 温湿度监控及定位应用示例 .....	86
5.6.1.	节点配置使用 .....	86
5.6.2.	应用代码介绍 .....	88
6.	NB-IoT 本地服务器 API 应用参考 .....	90
6.1.	MQTT 协议应用参考示例 .....	90
6.1.1.	MQTT 协议介绍 .....	90
6.1.2.	MQTT 协议操作 .....	93
6.2.	本地数据存储应用参考示例 .....	99
6.2.1.	工程配置简介 .....	99
6.2.2.	数据获取和存储 .....	101
6.3.	本地数据获取应用参考示例 .....	107
6.3.1.	express 简介 .....	107
6.3.2.	安装 express .....	107
6.3.3.	express 框架实例 .....	109
6.4.	NB-IoT 节点数据解析及显示示例 .....	112
6.4.1.	工程配置简介 .....	112
6.4.2.	NB-IoT 节点数据获取 .....	113
6.4.3.	NB-IoT 节点数据解析及显示 .....	115

# 1. NB-IoT 技术简介

移动通信正在从人和人的连接，向人与物以及物与物的连接迈进，万物互联是必然趋势。然而当前的 4G 网络在物与物连接上能力不足。事实上，相比蓝牙、ZigBee 等短距离通信技术，移动蜂窝网络具备广覆盖、可移动以及大连接数等特性，能够带来更加丰富的应用场景，理应成为物联网的主要连接技术。作为 LTE 的演进型技术，4.5G 除了具有高达 1Gbps 的峰值速率，还意味着基于蜂窝物联网的更多连接数，支持海量 M2M 连接以及更低时延，将助推高清视频、VoLTE 以及物联网等应用快速普及。蜂窝物联网正在开启一个前所未有的广阔市场。

对于电信运营商而言，车联网、智慧医疗、智能家居等物联网应用将产生海量连接，远远超过人与人之间的通信需求。人与人之间的通讯规模已近天花板，物与物的则刚刚进入增长快车道。随着可穿戴、车联网、智能抄表等新兴市场的开启，工业 4.0、智慧城市、智慧农业等理念照进现实，万物互联的时代正加速到来。

物联网 (IoT) 的未来充满想象空间。华为认为，到 2025 年全球将有 1000 亿个连接，其中大部分与物联网有关。物联网对连接的要求与传统蜂窝网络有着很大不同，窄带蜂窝物联网 (NB-IoT) 由此应运而生。这一由电信行业推动的新兴技术拥有覆盖广、连接多、速率低、成本低、功耗少、架构优等特点，极具商用潜力。

基于蜂窝的窄带物联网 (Narrow Band Internet of Things, NB-IoT) 成为万物互联网络的一个重要分支。NB-IoT 构建于蜂窝网络，只消耗大约 180KHz 的带宽，可直接部署于 GSM 网络、UMTS 网络或 LTE 网络，以降低部署成本、实现平滑升级。

NB-IoT 具备四大特点：一是广覆盖，将提供改进的室内覆盖，在同样的频段下，NB-IoT 比现有的网络增益 20dB，覆盖面积扩大 100 倍；二是具备支撑海量连接的能力，NB-IoT 一个扇区能够支持 10 万个连接，支持低延时敏感度、超低的设备成本、低设备功耗和优化的网络架构；三是更低功耗，NB-IoT 终端模块的待机时间可长达 10 年；四是更低的模块成本，企业预期的单个接连模块不

超过 5 美元。

随着智能城市、大数据时代的来临，无线通信将实现万物连接。很多企业预计未来全球物联网连接数将是千亿级的时代。目前已经出现了大量物与物的联接，然而这些联接大多通过蓝牙、Wi-Fi 等短距通信技术承载，但非运营商移动网络。为了满足不同物联网业务需求，根据物联网业务特征和移动通信网络特点，3GPP 根据窄带业务应用场景开展了增强移动通信网络功能的技术研究以适应蓬勃发展的物联网业务需求。

Machina 预测，NB-IoT 未来将覆盖 25%的物联网连接。对面临用户饱和、OTT 冲击的运营商来说，NB-IoT 将叩开广袤的新市场，带来三倍以上的连接增长；而对正积极转型升级的传统行业从业者而言，它在适应场景、网络性能、可管可控及可靠性等方面亦具备运营商网络的先天优势。

## 2. 硬件设备说明

### 2.1. NB-IoT 节点

NB-IoT 节点为 NB-IoT 网络中的终端设备。

实训平台根据实验的操作性,把 NB-IoT 节点分成两类,一类主要用来测试、使用 NB-IoT 相关协议的 NB-IoT 终端节点。一类是用来开发、调试设备的 NB-IoT 开发板。功能上 NB-IoT 终端节点和 NB-IoT 开发板相同。

#### 2.1.1.NB-IoT 终端节点

NB-IoT 终端节点,主要用来测试 NB-IoT 协议,并支持低功耗模式。

NB-IoT 终端节点提供一个下载口,一个 USB 调试接口,其上配有 NB-IoT 通信模块,是一个标准的 NB-IoT 终端设备。



图 1-1 NB-IoT 终端节点

#### 主要参数:

1. 硬件开发平台: ARM Cortex M0; ROM: 64KB; RAM: 8KB
2. 软件开发平台: 基于标准 C 语言开发
3. 支持 NB-IoT 协议
4. 发射功率: 23dBm $\pm$ 2dB
5. 供电: 3.1~4.2V DC, 典型值: 3.6V DC
6. 灵敏度: -129dBm
7. USIM/ESIM 接口: USIM, 3.0V DC

8. 通信接口：UART
9. UART 速率：9600 波特率
10. 支持 850MHz、900MHz 频段
11. 支持 SMA 接口天线

该 NB-IoT 终端节点，可以通过 USB 口供电，也可以通过配备的电源扩展板供电，电源扩展板分为插接式和板载式两种如图所示，实训平台中一般配备其中一种。

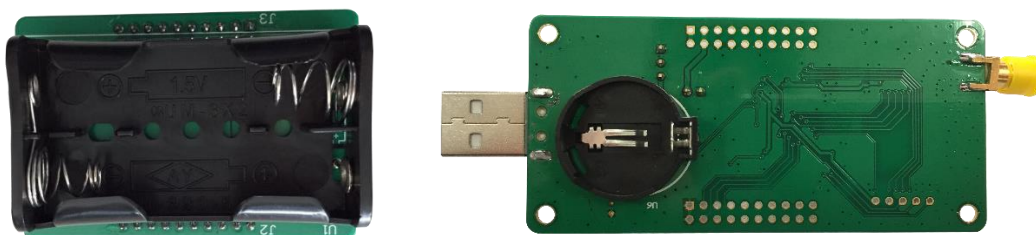


图 2-2 电源扩展板

针对该类节点配备了 JLink-SWD 转接头，以方便程序下载，如图所示。

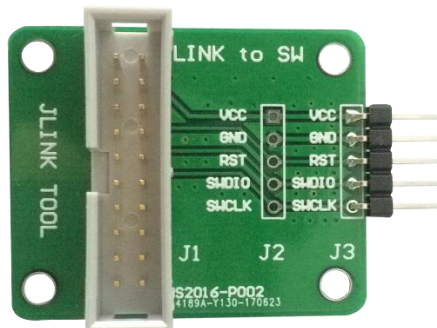


图 2-3 转接头

下载程序时保持下载器插针和 NB-IoT 终端节点的下载器插孔的紧密连接即可。

#### 引脚连接：

下载器转接板对应节点：VCC 接 3.3V；GND 接 GND；RST 接 RST；SWDIO 接 DIO；SWCLK 接 CLK。

该 NB-IoT 终端节点只支持第五章的源码应用调试。

## 2.1.2.NB-IoT 开发板

### ➤ 开发板简介

NB-IoT 开发板，主要用于系统资源开发、集成传感器应用、NB-IoT 协议开发、低功耗测试等。

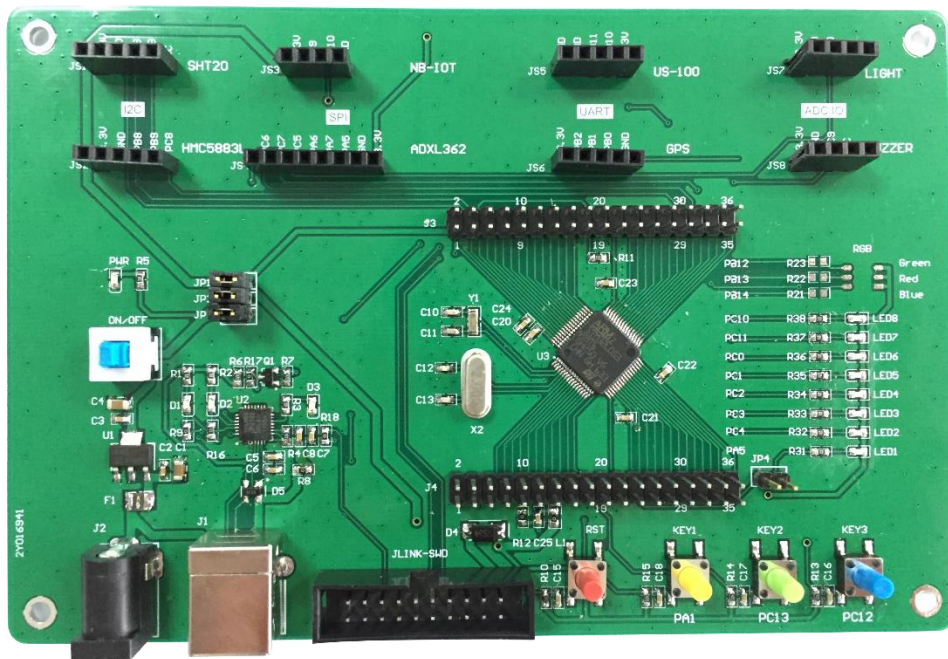


图 2-4 NB-IoT 开发板

### 主要参数：

1. 硬件开发平台：ARM Cortex M0；ROM：64KB；RAM：8KB
2. 软件开发平台：基于标准 C 语言开发
3. 支持 NB-IoT 模块
4. 工作电压：3.3V DC
5. 支持 JLink 下载调试
6. 支持传感器调试
7. 支持 I/O 扩展口应用
8. 支持板载 LED 灯调试

该 NB-IoT 开发板支持本实训平台中全套源码的应用调试。

### ➤ 功能介绍



接口	功能	描述
J1	USB 调试接口	<b>printf 默认输出口</b> ，支持给开发板供电
J2	电源接口	支持给开发板供电
J3	I/O 扩展口	
J4	I/O 扩展口	
JP1	NB-IoT 通信模块 电源端子	给 NB-IoT 通信模块单独供电，接跳线帽，供电。可串接电流表，测试 NB-IoT 通信模块功耗， <b>测功耗时必须保证 JP2 接跳线帽</b>
JP2	MCU 电源端子	给 MCU、NB-IoT 通信模块供电，接跳线帽，供电。可串接电流表，测试 MCU 功耗。 <b>若 JP1 未接跳线帽，则只测试 MCU 功耗；若 JP1 已接跳线帽，则测试 MCU 和 NB-IoT 通信模块的总功耗。低功耗测试时要确保运行 Class_A，建议关闭 printf 功能。</b>
JP3	传感器电源端子	给接入 JS1、JS2、JS4、JS5、JS6、JS7、JS8 的模块供电
JP4	LED1 端子	使用 LED1 时，接跳线帽。 <b>在使用 SPI 资源时必须去除跳线帽。</b>
JS1		接温湿度传感器模块
JS2		接磁阻传感器模块
JS3		接 NB-IoT 通信模块
JS4		接加速度传感器模块
JS5		接超声波传感器模块
JS6		接 GPS 模块
JS7		电压采样接口，可接光照传感器模块
JS8		I/O 控制接口，可接蜂鸣器模块
JLINK-SWD	下载接口	支持程序下载，支持板载供电， <b>不受 ON/OFF 开关控制</b>

ON/OFF	电源按键	控制 J1、J2 接入的电源
RST	复位按键	
Y1	低速外部晶振	≈32KHz
X2	高速外部晶振	≈8MHz
PWR	电源指示灯	开发板供电正常，常亮
D1	指示灯	板载 USB 口接收指示灯
D2	指示灯	板载 USB 口发送指示灯
D3	指示灯	USB 挂起，常亮；USB 开启，关闭
LED 模组		LED1~8、RGB（Green、Red、Blue），所有 LED 灯的控制引脚都是低电平有效，即引脚低电平点亮 LED 灯
KEY 模组		KEY1、KEY2、KEY3，板载按键，下降沿触发

由于现阶段 NB-IoT 通信模块的低功耗功能（有模块厂商决定）并不完善，上表中的低功耗测试无法实现。

## 2.2. NB-IoT 本地服务器

本实训平台中，NB-IoT 本地服务器实现 NB-IoT 节点数据的本地化存储，同时支持 API 的调用。通过本地服务器，不但可以在 Web 客户端查看 NB-IoT 节点的应用数据，还可以在 Web 客户端调用 API 接口，实现与 NB-IoT 节点的下行数据交互。除此之外，NB-IoT 本地服务器丰富的 API 接口以及详尽的说明文档和示例，可以满足各种应用场景的定制化需求。

使用时，只需用 **micro USB** 连接线给本地服务器供电，并**接入网线**即可（micro USB 接口在侧面）。

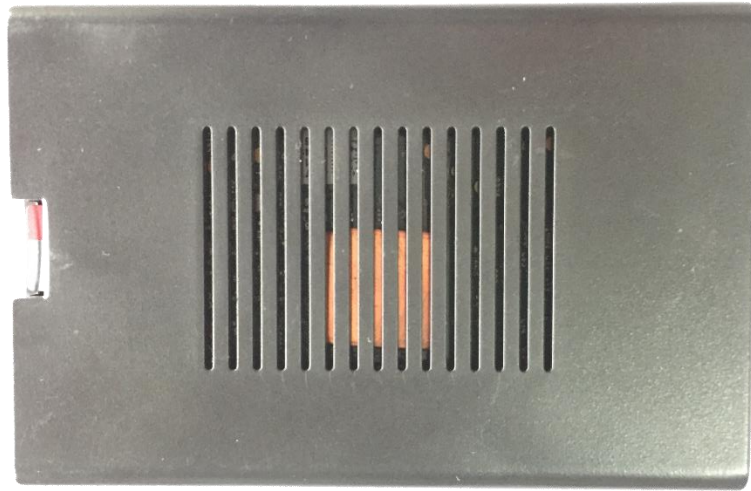


图 2-5 NB-IoT 本地服务器

主要参数:

1. 硬件开发平台: 采用 ARM Cortex A53 64 位 4 核处理器, 主频 1.2GHz, ROM: 16G, RAM: 1G
2. 操作系统: Linux

### 2.3. 加速度传感器模块

加速度传感器采用 ADXL362, 如下图所示。

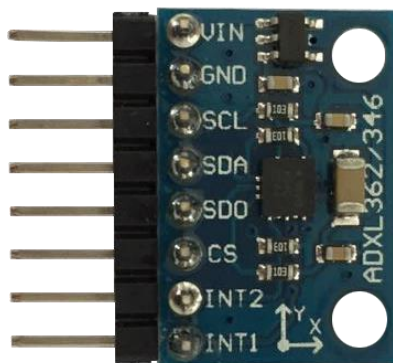


图 2-6 加速度传感器模块

主要参数:

1. 测量范围： $\pm 2 \sim \pm 8g$
2. 工作电压：1.6~3.3V DC
3. 通信接口：SPI
4. 支持 3 轴测量

配套开发板的使用接线图如下。

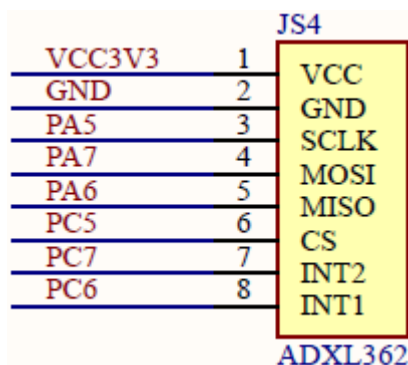


图 2-7 引脚图

引脚连接：

VIN 接 VCC；GND 接 GND；SCL 接 PA5；SDA 接 PA7；SDO 接 PA6；CS 接 PC5；INT2 接 PC7；INT1 接 PC6。

## 2.4. 光照传感器模块

光照传感器采用光敏器件，如下图所示。



图 2-8 光照传感器模块

主要参数：

1. 响应波长：320~730nm
2. 通信接口：模拟输出

配套开发板的使用接线图如下，开发板中采用电压采样的方式获取光照强度的变化。

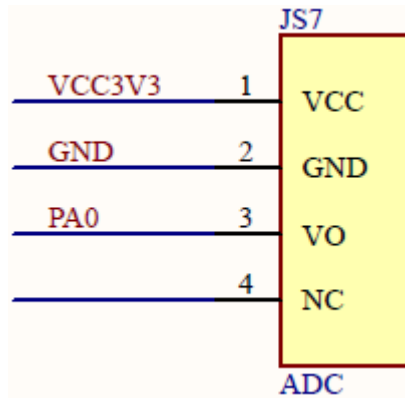


图 2-9 引脚图

引脚连接：

VCC 接 VCC；GND 接 GND；ADC 接 PA0；NC 不做连接。

## 2.5. 超声波传感器模块

超声波传感器，如下图所示。

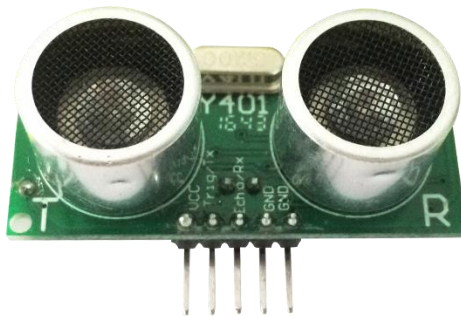


图 2-10 超声波传感器模块

主要参数：

1. 探测距离：20~4500mm
2. 探测角度：<15°
3. 工作电压：2.4~5.5V DC
4. 静态电流：2mA
5. 通信接口：数字接口
6. 支持 UART 接口

配套开发板的使用接线图如下。

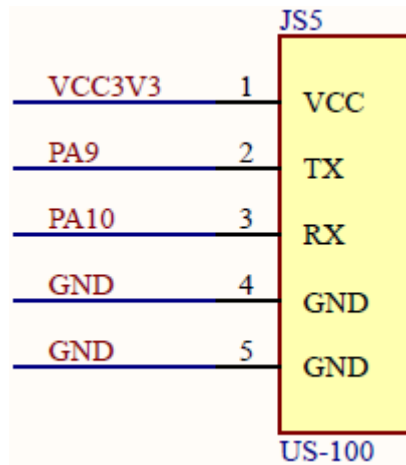


图 2-11 引脚图

引脚连接：

VCC 接 VCC；Trig/Tx 接 PA9；Echo/Rx 接 PA10；GND 接 GND。

## 2.6. 磁阻传感器模块

磁阻传感器采用 HMC5883L，如下图所示。

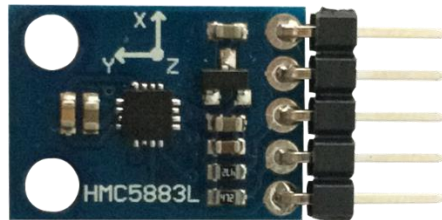


图 2-12 磁阻传感器模块

主要参数：

1. 测量范围： $\pm 8\text{G}$ as
2. 工作电压：2.16~3.6V DC
3. 通信接口：I2C

配套开发板的使用接线图如下。

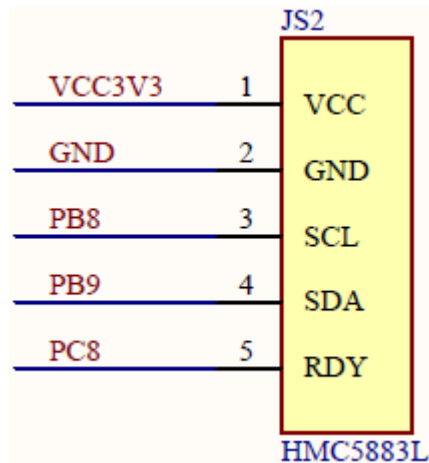


图 2-13 引脚图

引脚连接:

VCC 接 VCC; GND 接 GND; SCL 接 PB8; SDA 接 PB9; DRDY 接 PC8。

## 2.7. 温湿度传感器模块

温湿度传感器模块使用 sht20，如下图所示。

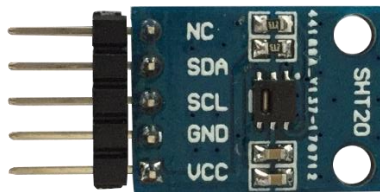


图 2-14 温湿度传感器模块

主要参数:

1. 温度测量范围:  $-40\sim 125^{\circ}\text{C}$
2. 湿度测量范围:  $0\sim 100\%\text{RH}$
3. 温度精度:  $\pm 0.3^{\circ}\text{C}$
4. 湿度精度:  $\pm 3.0\%\text{RH}$
5. 工作电流:  $\leq 330\mu\text{A}$
6. 电源电压:  $2.1\sim 3.6\text{V}$
7. 通信接口: I2C

配套开发板的使用接线图如下。

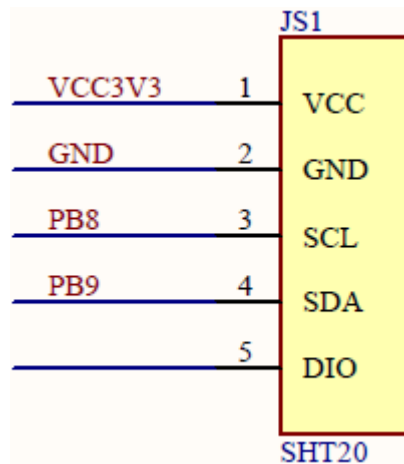


图 2-15 引脚图

引脚连接：

VCC 接 VCC；GND 接 GND；SCL 接 PB8；SDA 接 PB9，NC 不做连接。

## 2.8. GPS 模块

GPS 使用 SKG09A 模块，如下图所示。



图 2-16 GPS 模块

主要参数：

1. 灵敏度：跟踪：-165dBm；捕获：-148dBm
2. 首次定位时间：冷启动：23s（开阔天空）；温启动：2~3s；热启动：< 1s
3. 定位精度：3m（CEP50 without SA）;2.5m（SBAS）
4. 速度精度：0.1m/s
5. 最大更新速率：10Hz
6. 运行限制：高度：<18000m；速度：<515m/s；加速度：<4g



7. 电源：3~4.2V
8. 功耗：50mW
9. 支持 UART 通讯接口
10. 支持 22 个跟踪通道
11. 支持 66 个捕获通道

配套开发板的使用接线图如下。

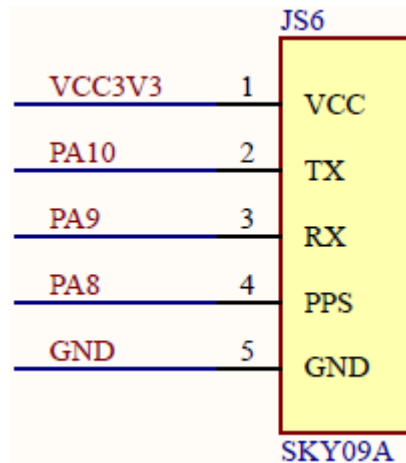


图 2-17 引脚图

引脚连接：

VCC 接 VCC；TXD 接 PA10；RXD 接 PA9；PPS 接 PA8；GND 接 GND。

## 2.9. NB-IoT 通信模块

NB-IoT 通信模块，如下图所示。

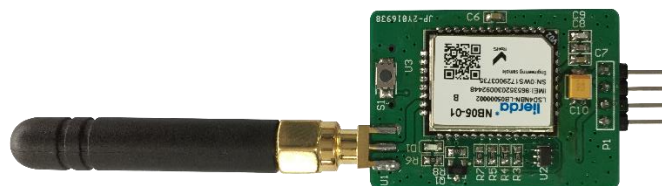


图 2-18 NB-IoT 通信模块

主要参数：

1. 发射功率：23dBm±2dB
2. 供电：3.1~4.2V DC，典型值：3.6V DC
3. 灵敏度：-129dBm

4. USIM/ESIM 接口：USIM, 3.0V DC
5. 通信接口：UART
6. UART 速率：9600 波特率
7. 支持 AT 指令
8. 支持 850MHz、900MHz 频段
9. 支持 SMA 接口天线

配套开发板的使用接线图如下。

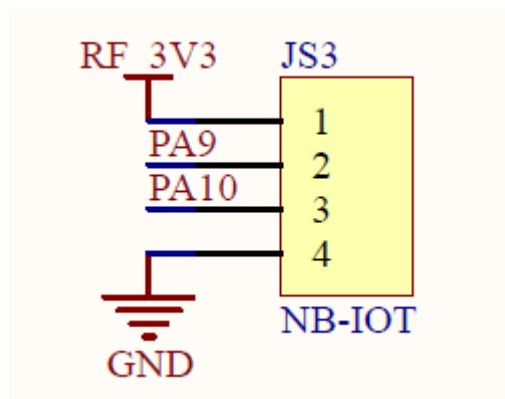


图 2-19 引脚图

引脚连接：

VCC 接 3V3；RX 接 PA9；TX 接 PA10；GND 接 GND。

## 2.10. 蜂鸣器模块

蜂鸣器模块如下图所示。



图 2-20 蜂鸣器模块

主要参数：

1. 供电电压：3.3V

## 2. 电平控制：高电平有效（打开蜂鸣器）

配套开发板的使用接线图如下。

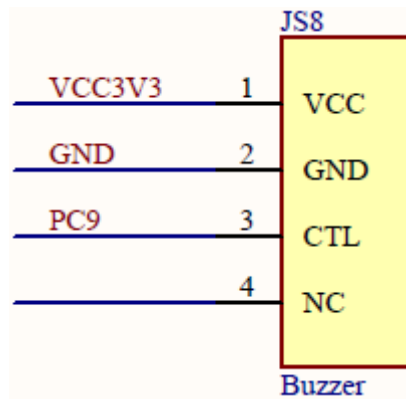


图 2-21 引脚图

引脚连接：

VCC 接 VCC；GND 接 GND；CTL 接 PC9。

## 2.11. 其他配件

1. JLink 下载器
2. NB-IoT 棒状天线
3. GPS 天线
4. micro USB 连接线
5. USB-B 连接线
6. USB 适配器器
7. NB-IoT 终端节点电源扩展板
8. NB-IoT 开发板电源适配器
9. NB-IoT 运营商卡

## 3. STM32 基础应用

### 3.1. 工程配置简介

STM32L0xx 系统资源应用示例在如下文件夹：

NB-IoT\_training\_platform/ARM/Projects/STM32L0XX/Examples/

例程如下：

- RCC：时钟配置使用
- GPIO：通用 I/O 口的配置使用
- RTC：系统实时时钟的配置使用
- IWDG：系统独立看门狗配置使用

工程中，文件夹、文件架构如下：

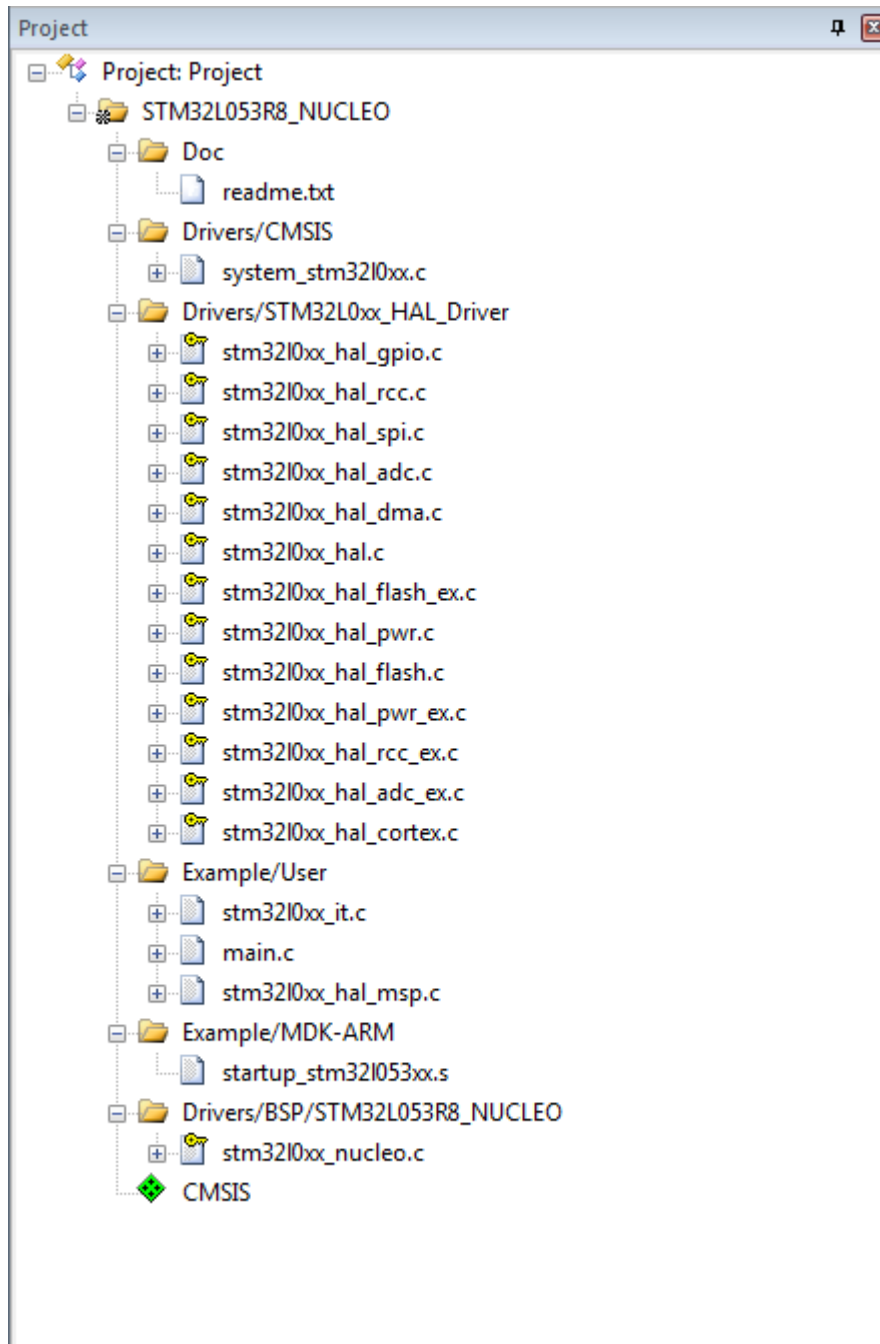


图 3-1 工程文件目录

目录结构:

- STM32L053R8\_NUCLEO:工程文件夹
- Doc/:存放相关的说明文档
- Driver/CMSIS/: ARM Cortex™ 微控制器软件接口标准的软件抽象层文件夹
- Driver/STM32L0xx\_HAL\_Driver:STM32L0/系类的官方驱动库文件夹
- Example/User/:存放可供用户修改、编辑的文件

- Example/MDK-ARM/:存放 ARM 启动文件
- Drivers/BSP/STM32L053R8\_NUCLEO/:存放与开发板相关的文件

本章所有的例程都符合以上文件架构规范，本章的例程修改、编译一般只涉及 Example/User 下的文件，该文件夹下一般包含如下三个文件：

- main.c:主文件，包含该程序运行的实例
- stm32l0xx\_it.c:中断响应文件
- stm32l0xx\_hal\_msp.c:系统相关资源配置文件

每个例程的具体应用说明见 Doc 文件夹下的 readme.txt 文件。本文示例采用的集成开发工具为 KEI15。

本实训平台推荐使用 Keil MDK，该软件可以从官方网站或论坛下载。安装完成软件界面如下。

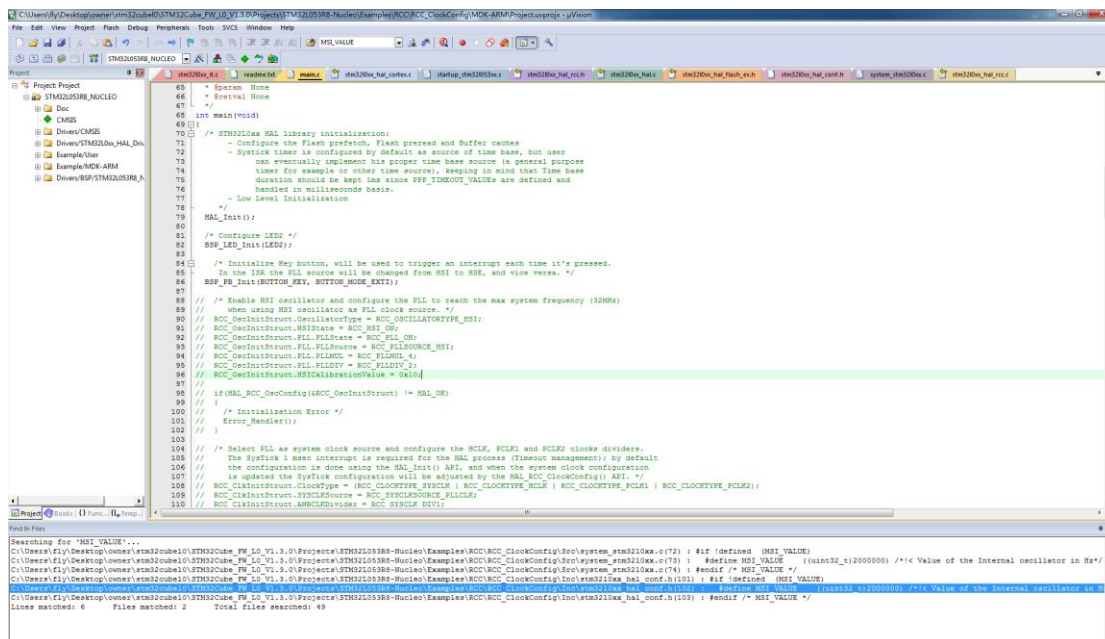


图 3-2 开发软件

### 3.2. 时钟配置示例

该例程介绍了如何配置系统时钟，以及在系统运行时修改运行时钟。具体描述见 Doc/readme.txt 文档。

#### ➤ main.c

系统基础资源初始化、板载 LED2、板载按键配置：

```
int main(void)
```

```

{
    HAL_Init(); //底层硬件初始化
    BSP_LED_Init(LED2); //LED2 初始化配置
    BSP_PB_Init(BUTTON_KEY, BUTTON_MODE_EXTI); //按键配置
    .....
}

```

系统时钟设置：选择内部 16MHz HSI，并把其作为 PLL 输入源。PLL 经过分频、倍频配置到 32MHz。

```

RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
RCC_OscInitStruct.HSIState = RCC_HSI_ON;
RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
RCC_OscInitStruct.PLL.PLLMUL = RCC_PLLMUL_4;
RCC_OscInitStruct.PLL.PLLDIV = RCC_PLLDIV_2;
RCC_OscInitStruct.HSICalibrationValue = 0x10;

if(HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
{
    Error_Handler();
}

```

选择 PLL 作为系统时钟，并配置 HCLK、PCLK1、PCLK2 时钟。

```

RCC_ClkInitStruct.ClockType=(RCC_CLOCKTYPE_SYSCLK|
RCC_CLOCKTYPE_HCLK |
RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2);
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;

if(HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1) !=
HAL_OK)
{
    Error_Handler();
}

```

把系统时钟通过 PA8 口输出。

```

HAL_RCC_MCOConfig(RCC_MCO1,RCC_MCO1SOURCE_SYSCLK,
RCC_MCODIV_1);

```

main.c 中还有中断的回调函数；外部时钟（HSE）、内部时钟（HSI）配置函数；错误处理函数。

```

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin);
void SystemClockHSE_Config(void);

```

```
void SystemClockHSI_Config(void);
```

➤ **stm32l0xx\_it.c**

外部中断函数配置，进入中断程序后回调 main.c 中的中断回调函数。

```
void EXTI4_15_IRQHandler(void)
{
    HAL_GPIO_EXTI_IRQHandler(KEY_BUTTON_PIN);
}
```

### 3.3. 通用 I/O 配置示例

该例程介绍如何配置 GPIO 口，配置完成后实现对板载 LED（PA5）灯的控制，实现交替闪烁。具体描述见 Doc/readme.txt 文档。

main 函数的初始化配同 3.2.1。关于 GPIO 的配置如下：

➤ **main.c**

控制 LED 灯的 GPIO 口选用 PA5，上拉输出、高速模式。

```
__HAL_RCC_GPIOA_CLK_ENABLE();
GPIO_InitStruct.Pin = (GPIO_PIN_5);
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_PULLUP;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH ;

HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

系统时钟选用内部时钟（MSI）：≈2MHz。

```
static void SystemClock_Config(void)
{
    .....
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_MSI;
    RCC_OscInitStruct.MSISState = RCC_MSI_ON;
    /* 设置 MSI = 2.097 MHz */
    RCC_OscInitStruct.MSIClockRange = RCC_MSIRANGE_5;
    RCC_OscInitStruct.MSICalibrationValue=0x00;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
    .....
    /* MSI 作为系统时钟（system clock）*/
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_MSI;
    .....
}
```

LED 灯每隔 200ms 翻转一次状态，即闪烁。

```
while (1)
{
```



```

    HAL_Delay(100);           //延迟 100ms
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5); //翻转 PA5 状态
    /* 100 ms 延时 */
    HAL_Delay(100);
}

```

### 3.4. 实时时钟配置示例

该例程介绍如何配置 RTC，并应用 RTC 的 alarm 功能定时点亮 LED 灯。  
具体描述见 Doc/readme.txt 文档。

main 函数的初始化以及中断配置方式同 3.2.1。关于 RTC、Alarm 的配置如下：

#### ➤ *main.c*

初始化 RTC，设置闹钟（Alarm）时间，设置初始时间（年、月、日、时、分、秒）。

```

static void RTC_AlarmConfig(void)
{
    .....
    /*
    * 1、配置 RTC 的同步异步分频系数
    * 2、关闭 RTC 输出
    */
    RtcHandle.Instance = RTC;
    RtcHandle.Init.HourFormat = RTC_HOURFORMAT_24;
    RtcHandle.Init.AsynchPrediv = RTC_ASYNC_PREDIV;
    RtcHandle.Init.SynchPrediv = RTC_SYNCH_PREDIV;
    RtcHandle.Init.OutPut = RTC_OUTPUT_DISABLE;
    /* 设置闹铃时间：02：20：30 */
    salarmstructure.AlarmTime.Hours = 0x02;
    salarmstructure.AlarmTime.Minutes = 0x20;
    salarmstructure.AlarmTime.Seconds = 0x30;
    salarmstructure.AlarmTime.SubSeconds = 0x56;
    /* 设置时间：02：20：00 */
    stimestructure.Hours = 0x02;
    stimestructure.Minutes = 0x20;
    stimestructure.Seconds = 0x00;
    stimestructure.TimeFormat = RTC_HOURFORMAT12_AM;
    stimestructure.DayLightSaving = RTC_DAYLIGHTSAVING_NONE ;
    stimestructure.StoreOperation = RTC_STOREOPERATION_RESET;
    .....
}

```

```
}

```

RTC 的运行时间实时更新函数如下，且可以在调试模式下，通过 showtime 变量查看系统运行时间。

```
static void RTC_TimeShow(uint8_t* showtime)
{
    .....
    HAL_RTC_GetTime(&RtcHandle, &stimestructureget,
    RTC_FORMAT_BIN);
    .....
    sprintf((char*)showtime,"%02d:%02d:%02d",
    stimestructureget.Hours,
    stimestructureget.Minutes, stimestructureget.Seconds);
    .....
}
```

#### ➤ *Stm32l0xx\_hal\_msp.c*

配置 RTC 选用的时钟源及相关参数，并使能 RTC。这里以 LSI 配置介绍，LSE 的配置与之类似，至于 LSE 和 LSI 的切换在 main.h 中选择。

```
void HAL_RTC_MspInit(RTC_HandleTypeDef *hrtc)
{
    .....
    /* LSI 作为 RTC 的时钟 */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_LSI |
    RCC_OSCILLATORTYPE_LSE;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
    RCC_OscInitStruct.LSIState = RCC_LSI_ON;
    RCC_OscInitStruct.LSEState = RCC_LSE_OFF;
    .....
    /* 备份域设置 */
    PeriphClkInitStruct.PeriphClockSelection = RCC_PERIPHCLK_RTC;
    PeriphClkInitStruct.RTCClockSelection = RCC_RTCCLKSOURCE_LSI;
    .....
    /*
    * 1, RTC 使能
    * 2, 设置 RTC 中断级，并使能中断
    */
    __HAL_RCC_RTC_ENABLE();
    HAL_NVIC_SetPriority(RTC_IRQn, 0x0, 0);
    HAL_NVIC_EnableIRQ(RTC_IRQn);
}
```

### 3.5. 独立看门狗配置示例

该例程介绍如何配置并使用独立看门狗。该程序设置看门狗的超时时间为 250ms，一旦看门狗超时，则会导致系统复位。具体描述见 Doc/readme.txt 文档。

main 函数的初始化以及中断配置方式同 3.2.1。关于独立看门狗的配置如下：

#### ➤ *main.c*

配置独立看门狗参数。预分频系数：32，看门狗时钟为 LSI（频率：uwLsiFreq），超时时间 250ms，可以计算出 Reload 的值如下：

$$\text{Reload} = 0.25 / (32 / \text{uwLsiFreq});$$

$$\text{Reload} = \text{uwLsiFreq} / 128;$$

```
IwdgHandle.Instance = IWDG;
IwdgHandle.Init.Prescaler = IWDG_PRESCALER_32;
IwdgHandle.Init.Reload = uwLsiFreq/128;
IwdgHandle.Init.Window = IWDG_WINDOW_DISABLE;
```

在 while（1）中重载计数器（喂狗），确保不会导致看门狗超时而导致系统复位。

```
while (1)
{
    .....
    if(HAL_IWDG_Refresh(&IwdgHandle) != HAL_OK)
    {
        Error_Handler();
    }
}
```

用定时器 TM21 测量 LSI 的时钟频率，从而精确独立看门的超时时间计算。

```
static uint32_t GetLSIFrequency(void)
{
    .....
    /* 配置 TIM 的时钟捕获模式 */
    Input_Handle.Init.Prescaler      = 0;
    Input_Handle.Init.CounterMode    = TIM_COUNTERMODE_UP;
    .....
    TIMInput_Config.ICPolarity        = TIM_ICPOLARITY_RISING;
    TIMInput_Config.ICSelection        = TIM_ICSELECTION_DIRECTTI;
```

```

TIMInput_Config.ICPrescaler      = TIM_ICPSC_DIV8;
TIMInput_Config.ICFilter         = 0;

.....
/* 直到获取到 2 次 LSI 信号的边沿 */
while(uwCaptureNumber != 2)
{
}

.....
/* 返回测量出的 LSI 时钟频率值 */
return uwLsiFreq;
}

```

TIM21 的中断返回函数，在该函数中计算 LSI 的时钟频率，其中 uwLsiFreq 是定义的全局变量。

```

void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
{
    tmpCC4[uwCaptureNumber++] =
    HAL_TIM_ReadCapturedValue(&Input_Handle, TIM_CHANNEL_1);
    if (uwCaptureNumber >= 2)
    {
        uwPeriodValue = (uint16_t)(0xFFFF - tmpCC4[0] +
        tmpCC4[1] + 1);
        uwLsiFreq = (uint32_t) SystemCoreClock / uwPeriodValue;
        uwLsiFreq *= 8;
    }
}

```

#### ➤ *Stm32l0xx\_hal\_msp.c*

配置独立看门狗、定时器的相关资源，并使能。设置中断优先级并使能。

```

void HAL_IWDG_MspInit(IWDG_HandleTypeDef* hiwdg)
{
    __HAL_RCC_PWR_CLK_ENABLE();
    HAL_PWR_EnableBkUpAccess();
}

void HAL_TIM_IC_MspInit(TIM_HandleTypeDef *htim)
{
    __HAL_RCC_LSI_ENABLE();
    while (__HAL_RCC_GET_FLAG(RCC_FLAG_LSIRDY) == RESET)
    {
    }
    __HAL_RCC_TIM21_CLK_ENABLE();
    HAL_NVIC_SetPriority(TIM21_IRQn,0,0);
    HAL_NVIC_EnableIRQ(TIM21_IRQn);
}

```

## 4. 传感器及响应模块应用

### 4.1. 工程配置简介

STM32L0xx 传感器应用示例在如下文件夹：

NB-IoT\_training\_platform/ARM/Projects/STM32L0XX/Applications/MDK

工程中，文件夹、文件架构如下：

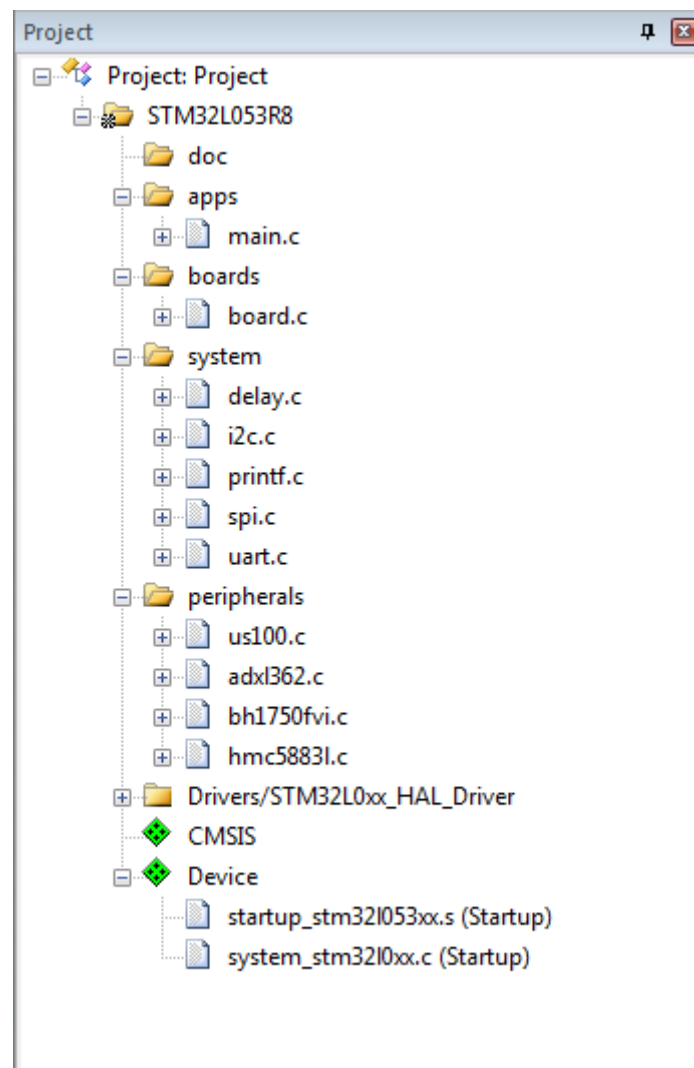


图 4-1 工程文件目录

目录结构：

- STM32L053R8:工程文件夹
- doc/:存放相关的说明文档
- apps/:应用文件

- boards/:依赖于开发板的文件
- system/:系统资源文件
- peripherals/:外围设备文件
- Driver/STM32L0xx\_HAL\_Driver/:STM32L0 系类的官方驱动库文件夹

本章针对传感器的应用主要在于：

main.c:应用文件

main.h:配置文件

除 Driver/STM32L0xx\_HAL\_Driver/官方驱动文件外，根据传感的深度应用，其它文件均可有适度修改。

#### ➤ **main.h**

板载模块、传感器启用设定。

#define	NORMAL_LED	//启用普通 LED 功能
#define	RGB_LED	//启用 RGB_LED 功能
#define	BUTTON	//启用按键功能
#define	BUZZER	//启用蜂鸣器功能
#define	PRINTF	//启用 printf 功能
#define	ADC_1	//启用 ADC_1，电压采样功能
#define	ADXL362	//启用加速度传感器：ADXL362
#define	HMC5883L	//启用磁阻传感器：HMC5883L
#define	US100	//启用超声波传感器：US100

示例代码都可以用 printf 功能实时打印传感器数据，因此 printf 功能默认开启，如有其他自定义的调试方式，可以关闭该功能。

printf 功能默认使用了系统 uart2 资源，串口设置（com12 按实际情况修改）如下：



图 4-2 串口工具配置

## 4.2. 加速度传感器应用示例

加速度传感器选用的是 ADXL362，驱动文件见 adxl362.c

加速度传感器的使用如下

首先在 main.h 中开启加速度传感器功能

### ➤ main.h

```
#define ADXL362           //启用加速度传感器：ADXL362
```

在 main.c 文件中的实现如下：

### ➤ main.c

配置加速度传感器并运行。

```
#ifndef ADXL362
/* 系统中断引脚配置，对应加速度中断引脚：INT1*/
io_interrupt_config();
/* 配置加速度传感器并运行 */
adxl362_config();
```

```
#endif
```

加速度传感器详细配置。

```
void adxl362_config()
{
    /* 初始化 SPI 硬件资源 */
    spi_gpio_init(&SpiHandle);
    /* 初始化加速传感器，并配置 SPI 的模式 */
    ADXL362_Init();
    ADXL362_SoftwareReset();
    ADXL362_SetPowerMode(0);
    /* 设置加速的测量范围：-2g~+2g */
    ADXL362_SetRange(ADXL362_RANGE_2G);
    /* 设置加速传感器的采样速率：400Hz */
    ADXL362_SetOutputRate(ADXL362_ODR_400_HZ);
    /* 开启运动检测，并关闭 FIFO 功能 */
    ADXL362_SetRegisterValue(ADXL362_ACT_INACT_CTL_ACT_EN,
    ADXL362_REG_ACT_INACT_CTL,1);
    ADXL362_FifoSetup(ADXL362_FIFO_DISABLE,0,0);
    /* 数据就绪状态映射到中断 1：INT1；并开启运行模式 */
    ADXL362_INTMAP1();
    ADXL362_SetPowerMode(1);
}
```

读取加速度传感器的数据，并用 printf 打印实时数据。

```
#ifndef ADXL362
ADXL362_GetXyz(&x_axis,&y_axis,&z_axis);
#endif
#ifdef PRINTF
printf("x_axis=%d y_axis=%d z_axis=%d\n",x_axis,y_axis,z_axis);
DelayMs(200); //打印间隔 200ms
#endif#endif
```

系统中断引脚（对应加速度中断引脚：INT1）配置如下：

#### ➤ **board.c**

```
void io_interrupt_config(void)
{
    /* 配置 PC6 引脚为中断引脚：上升沿触发 */
    GPIO_InitStructure.Mode = GPIO_MODE_IT_RISING;
    GPIO_InitStructure.Pull = GPIO_PULLDOWN;
    GPIO_InitStructure.Pin = GPIO_PIN_6;
    GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(GPIOC, &GPIO_InitStructure);
    /* 配置中断优先级并使能中断 */
    HAL_NVIC_SetPriority(EXTI4_15_IRQn, 3, 0);
    HAL_NVIC_EnableIRQ(EXTI4_15_IRQn);
}
```



}

加速度传感器打印数据如下，字符形式显示，格式为：x（x\_axis）轴数据、y（y\_axis）轴数据、z（z\_axis）轴数据。

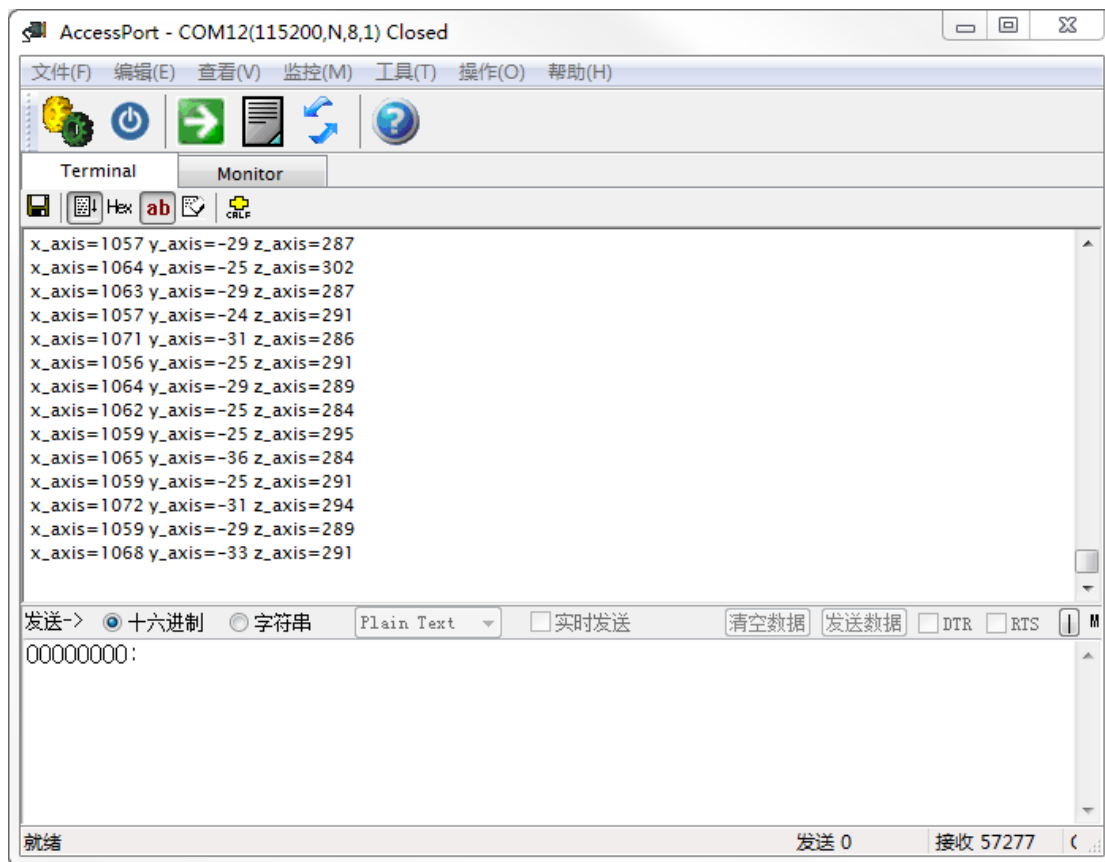


图 4-3 加速度传感器数据串口显示

### 4.3. 光照传感器应用示例

光照传感器，其原理是利用了光敏电阻的特性感知光源，然后通过采集传感器的输出电压，再折算出光照强度，因此该驱动的实现既是通过 ADC 采样电压来完成工作。

光照传感器的使用如下：

首先在 main.h 中开启 ADC 功能。

#### ➤ main.h

```
#define ADC_1 //开启 ADC 采样功能
```

然后初始化 ADC 配置。

#### ➤ main.c

```
#ifndef ADC_1
    adc1_init(); //初始化配置 ADC1
#endif
```

```

adc1_set_channel0(); //选测采样通道：通道 0
adc1_start();        //开始 ADC 采样
#endif

```

配置 ADC 中断回调函数，并获取 ADC 转换完成后的采样值。

```

#ifdef ADC_1
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* AdcHandle)
{
    adc1_stop();           //关闭电压转换中断
    /* 获取 ADC 转换完成后的采样值 */
    adc1_converted_value = HAL_ADC_GetValue(AdcHandle);
    adc1_flag = SET;       //转换完成标志
}
#endif

```

每当 ADC 转换完成后，用 printf 工具打印经过公式计算过的电压值。

电压转换公式：电压值 = 采样值 \* 3.3V / 4096。

```

#ifdef ADC_1
    if(adc1_flag == SET)
    {
        /* 计算电压值 */
        voltage = (float)adc1_converted_value * 3.3 / 4096;
#ifdef PRINTF
        printf("ADC1:%.2fV\n",voltage);
#endif
        DelayMs(100);
        adc1_flag = RESET; //清除转换完成标志
        adc1_start();      //开器电压采样功能
    }
#endif

```

ADC 功能配置，具体如下：

#### ➤ *board.c*

设置 ADC 的基本参数，关键参数如下：

```

void adc1_init()
{
    .....
    /*
    *配置 ADC
    *1、12 位电压采样
    *2、采样时间 7.5 个时钟周期
    *3、数据右对齐
    *4、转换结束标志 EOC
    *5、关闭 DMA
    */
}

```

```

    */
    AdcHandle.Init.Resolution      = ADC_RESOLUTION_12B;
    AdcHandle.Init.SamplingTime    = ADC_SAMPLETIME_7CYCLES_5;
    .....
    AdcHandle.Init.DataAlign       = ADC_DATAALIGN_RIGHT;
    .....
    AdcHandle.Init.EOCSelection     = ADC_EOC_SINGLE_CONV;
    AdcHandle.Init.DMAContinuousRequests = DISABLE;
    .....
    adc1_gpio_init(&AdcHandle);    //配置对应的 ADC 采样引脚
    .....
}

```

配置 PA0 作为 ADC 采样引脚。

```

void adc1_gpio_init(ADC_HandleTypeDef *hadc)
{
    .....
    GPIO_InitStruct.Pin = GPIO_PIN_0;
    GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
    .....
}

```

启用通道 0 作为采样通道。

```

void adc1_set_channel0()
{
    sConfig.Channel = ADC_CHANNEL_0;
    HAL_ADC_ConfigChannel(&AdcHandle, &sConfig);
}

```

开启与关闭 ADC 采样。

```

void adc1_start()
{
    HAL_ADC_Start_IT(&AdcHandle);
}

void adc1_stop()
{
    HAL_ADC_Stop_IT(&AdcHandle);
}

```

打印信息如下，字符形式显示，格式为：ADC1 采样的电压值，单位为 V。

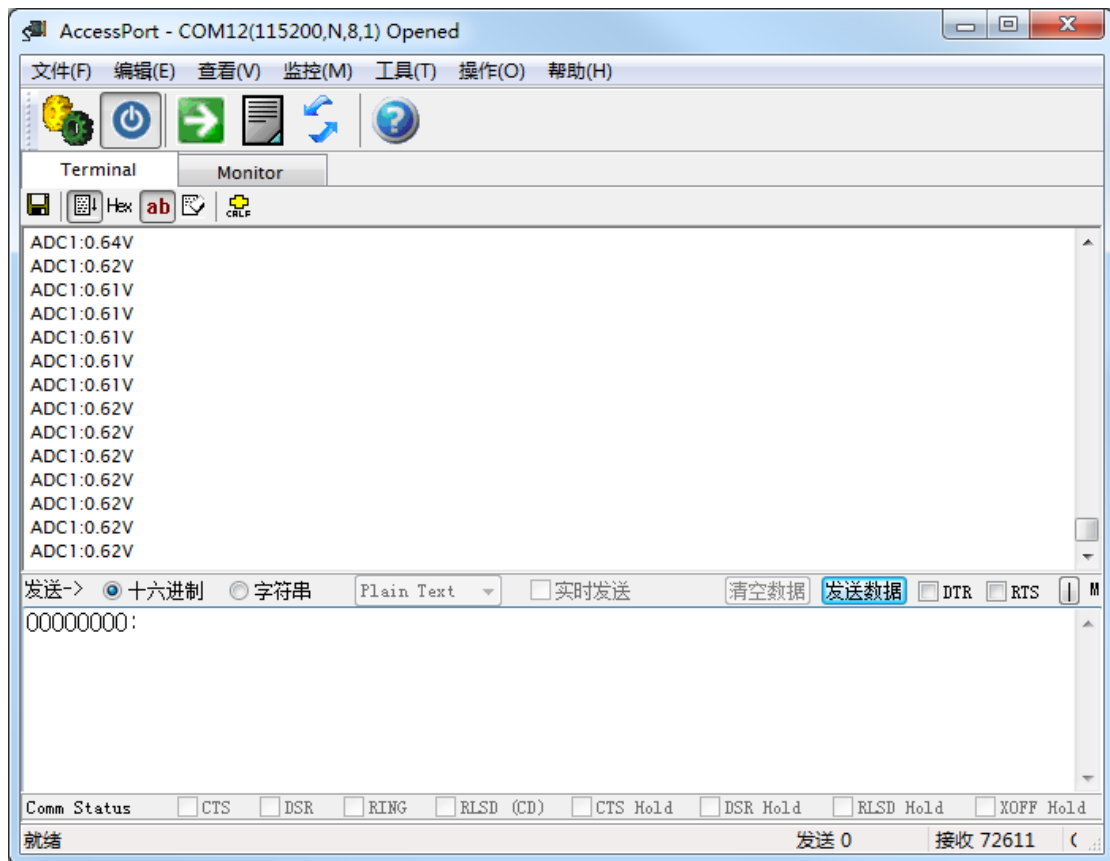


图 4-4 光照传感器数据串口显示

该传感器的原理是光敏电阻，随着光强的改变光敏电阻的阻值随之改变，导致采样的电压改变：光强越强，阻值越大，采样电压越大。

#### 4.4. 超声波传感器应用示例

超声波传感器选用的是 US100，驱动文件见 us100.c。

超声波传感器的使用如下。**该示例中，超声波使用的是 I/O 模式，使用时去除传感器的跳线帽。**

首先在 main.h 中开启超声波传感器功能。

##### ➤ main.h

```
#define US100 //启用超声波传感器：US100
```

然后在 main.c 文件中初始化并运行，通过 printf 输出实时数据。

##### ➤ main.c

初始化超声波传感器：

```
#ifndef US100
    us100_init();
#endif
```

读取超声波传感器数据：

数据有效范围：25~4500mm。

```
#ifndef US100
    us100_data = us100_get_distance();
    DelayMs(50);    //测量打印延迟
    if(us100_data >= 25 || us100_data <= 4500)
    {
#ifdef PRINTF
        printf("distance:%dmm\n",us100_data);
#endif
    }
    else
    {
#ifdef PRINTF
        printf("error\n");
#endif
    }
}
#endif
```

超声波传感器打印数据如下，字符形式显示，格式为：距离数值，单位毫米。

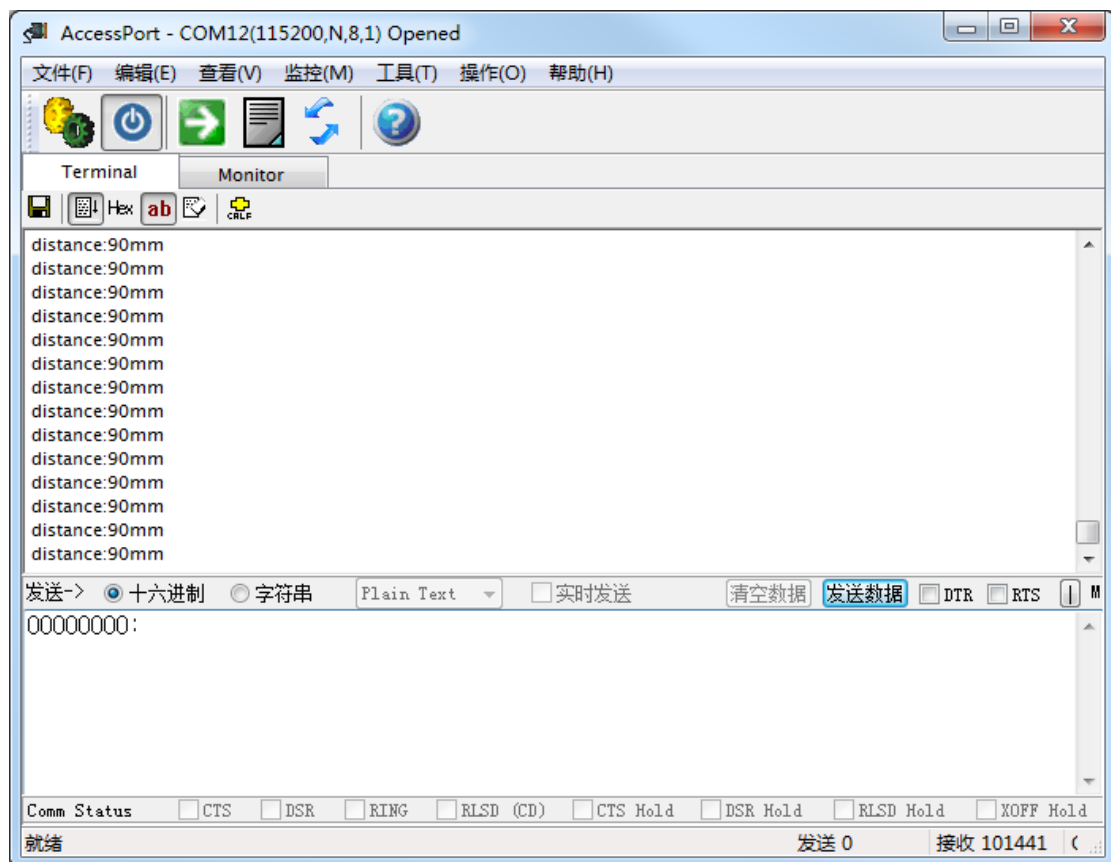


图 4-2 超声波传感器数据串口显示

## 4.5. 磁阻传感器应用示例

磁阻传感器选用的是 HMC5883L，驱动文件见 hmc5883l.c。

磁阻传感器的使用如下：

首先在 main.h 中开启磁阻传感器功能。

### ➤ main.h

```
#define HMC5883L    //启用磁阻传感器：HMC5883L
```

然后在 main.c 文件中初始化并运行，通过 printf 输出实时数据。

### ➤ main.c

初始化磁阻传感器：

```
#ifndef HMC5883L
    hmc5883l_init();
#endif
```

读取磁阻传感器数据，并通过 printf 实时打印出来：

```
#ifndef HMC5883L
    DelayMs(1000);
    hmc5883l_angle = hmc5883l_get_data(hmc5883l_data);
#endif
#ifdef PRINTF
    printf("x:%d y:%d z:%d  angle:%.2f\n",
        hmc5883l_data[0],hmc5883l_data[1],hmc5883l_data[2],hmc5883l_angle);
#endif
#endif
```

磁阻传感器的初始化配置如下：

### ➤ hmc5883l.c

配置系统 I2C 接口，并配置磁阻传感器的寄存器。

```
void hmc5883l_init()
{
    /* 配置 I2C 的 I/O 并初始化 */
    i2c_gpio_init(&I2CxHandle_hmc5883l);
    /* I2C 初始化 */
    i2c_init();
    /* 配置磁阻传感器的寄存器 A 和 B */
    tx_buffer[0] = 0x00; //寄存器 A
    tx_buffer[1] = 0x70; //寄存器 A 的值
    tx_buffer[2] = 0x20; //寄存器 B 的值，地址自增
    .....
}
```

磁阻传感器数据打印如下，字符形式显示，打印格式为：x 轴数据；y 轴数据；z 轴数据；磁场角度。

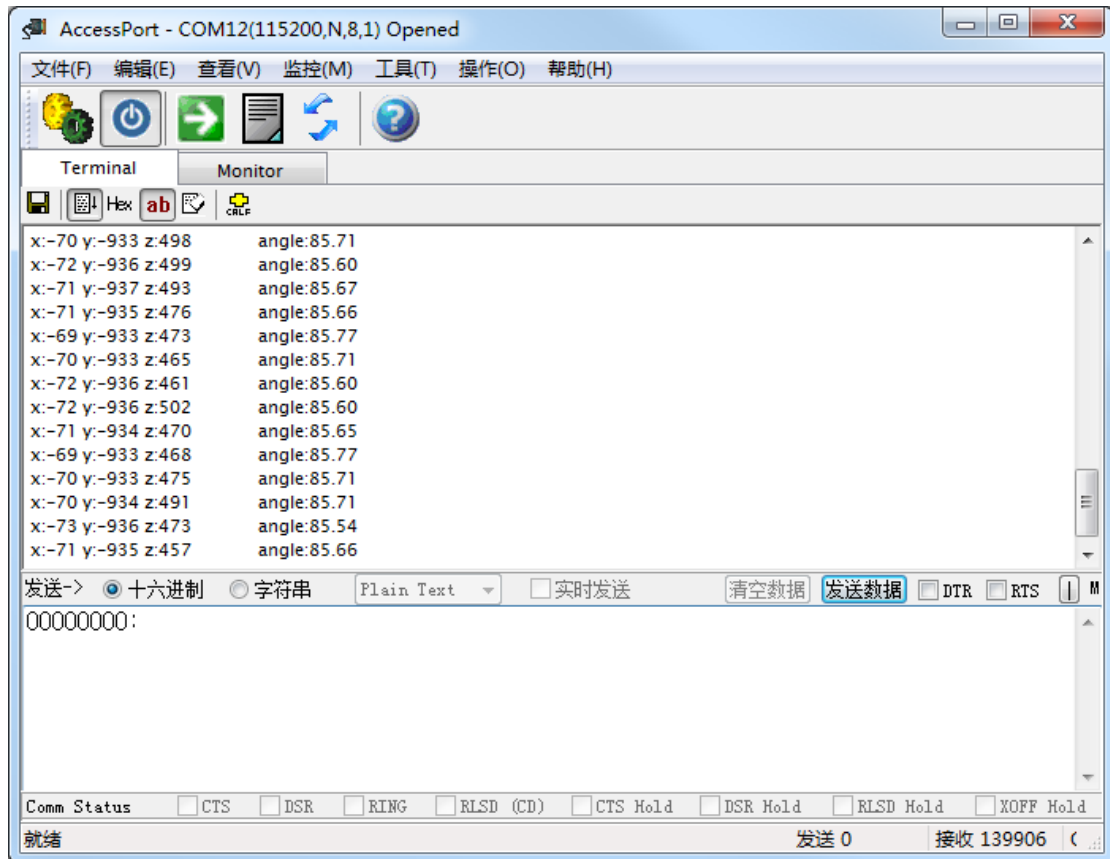


图 4-3 磁阻传感器数据串口显示

## 4.6. 蜂鸣器应用示例

蜂鸣器模块接入 NB-IoT 开发版，下载源码例程后，蜂鸣器间歇性工作鸣叫（时间间隔：300ms）。

蜂鸣器驱动文件见 `buzzer.c`。

蜂鸣器的使用如下：

首先在 `main.h` 中开启蜂鸣器功能。

### ➤ `main.h`

```
#define BUZZER //启用蜂鸣器
```

然后在 `main.c` 文件中初始化并运行。

### ➤ `main.c`

初始化蜂鸣器

```
#ifndef BUZZER
buzzer_init();
```

```
#endif
```

用一个延迟，开启、关闭蜂鸣器，实现蜂鸣器的间歇性鸣叫。

```
#ifndef BUZZER
    buzzer_on();
    DelayMs(300);
    buzzer_off();
#endif
```

蜂鸣器的驱动程序如下：

#### ➤ *buzzer.c*

驱动蜂鸣器，只需操控 CTL 引脚。该引脚低电平，蜂鸣器关闭；该引脚高电平，蜂鸣器开启。

```
void buzzer_on()
{
    HAL_GPIO_WritePin(BUZZER_GPIO_PORT,BUZZER_CTL,GPIO_PIN_SET);
}

void buzzer_off()
{
    HAL_GPIO_WritePin(BUZZER_GPIO_PORT,BUZZER_CTL,GPIO_PIN_RESET);
}
```

## 4.7. 三色 LED 灯应用示例

NB-IoT 开发板下载源码例程后，板载三色 LED 灯以红、绿、蓝的颜色循环点亮。

三色 LED 灯驱动文件见 led\_board.c。

三色 LED 灯的使用如下，LED 灯为低电平驱动，即引脚高电平关闭，引脚低电平点亮。

首先在 main.h 中开启三色 LED 灯的功能。

#### ➤ *main.h*

```
#define RGB_LED    //启用三色 LED 灯
```

然后在 main.c 文件中初始化并运行。

#### ➤ *main.c*

初始化三色 LED 灯功能。

```
#ifndef RGB_LED
```



```
    led_init(RGB);  
#endif
```

用一个延迟，分别驱动红、绿、蓝 LED 灯，使其循环点亮。

```
#ifdef RGB_LED  
    rgb_red(RGB);    //点亮红灯  
    DelayMs(100);  
    rgb_green(RGB);  //点亮绿灯  
    DelayMs(100);  
    rgb_blue(RGB);   //点亮蓝灯  
    DelayMs(100);  
#endif
```

## 5. NB-IoT 技术应用

### 5.1. NB-IoT 协议应用简介

#### 5.1.1. 网络

##### 5.1.1.1. 核心网

为了将物联网数据发送给应用，蜂窝物联网（CIoT）在 EPS 定义了两种优化方案：

- CIoT EPS 用户面功能优化（User Plane CIoT EPS optimisation）
- CIoT EPS 控制面功能优化（Control Plane CIoT EPS optimisation）

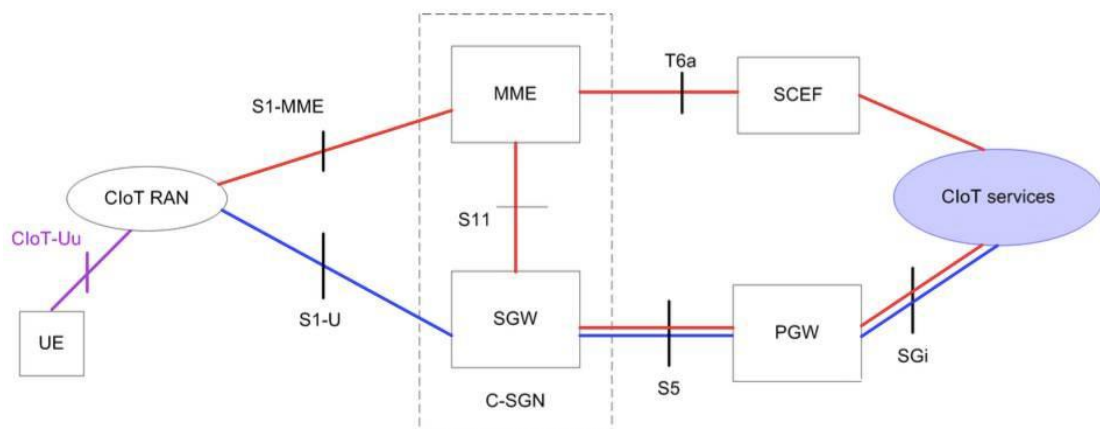


图 5-1 优化方案

如上图所示，红线表示 CIoT EPS 控制面功能优化方案，蓝线表示 CIoT EPS 用户面功能优化方案。

对于 CIoT EPS 控制面功能优化，上行数据从 eNB（CIoT RAN）传送到 MME，在这里传输路径分为两个分支：或者通过 SGW 传送到 PGW 再传送到应用服务器，或者通过 SCEF（Service Capability Exposure Function）连接到应用服务器（CIoT Services），后者仅支持非 IP 数据传送。下行数据传送路径一样，只是方向相反。

这一方案无需建立数据无线承载，数据包直接在信令无线承载上发送。因此，这一方案极适合非频发的小数据包传送。

SCEF 是专门为 NB-IoT 设计而新引入的，它用于在控制面上传送非 IP 数据包，并为鉴权等网络服务提供了一个抽象的接口。

对于 CIoT EPS 用户面功能优化，物联网数据传送方式和传统数据流量一样，在无线承载上发送数据，由 SGW 传送到 PGW 再到应用服务器。因此，这种方案在建立连接时会产生额外开销，不过，它的优势是数据包序列传送更快。这一方案支持 IP 数据和非 IP 数据传送。

### 5.1.1.2. 接入网

NB-IoT 的接入网构架与 LTE 一样。

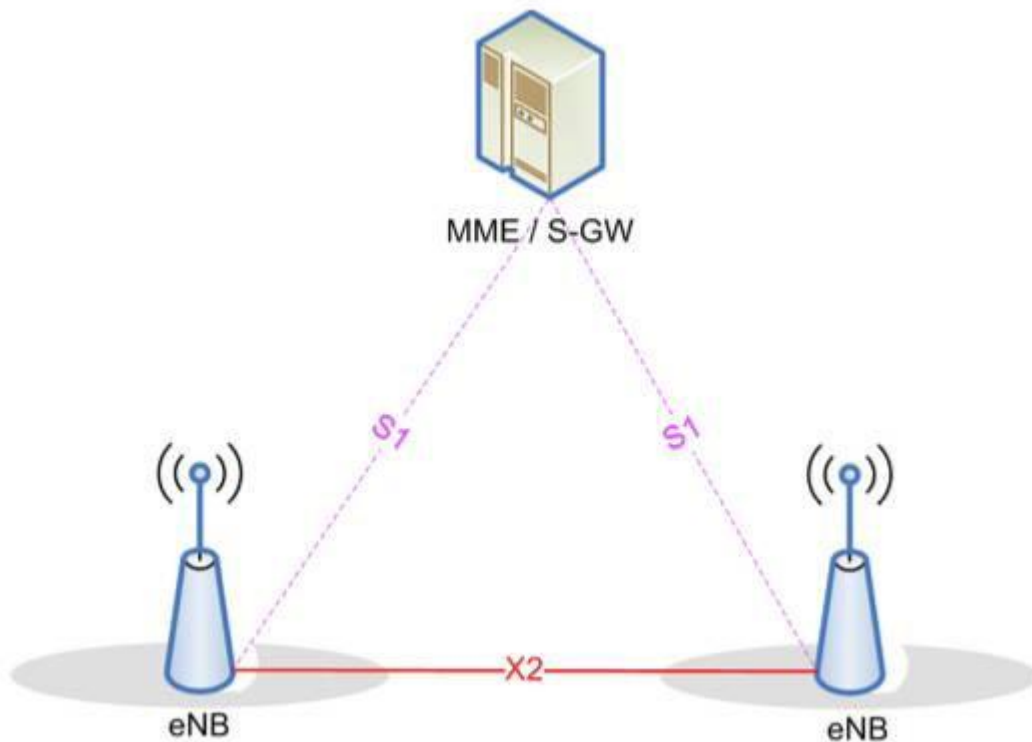


图 5-2 网络架构

eNB 通过 S1 接口连接到 MME/S-GW，只是接口上传送的是 NB-IoT 消息和数据。尽管 NB-IoT 没有定义切换，但在两个 eNB 之间依然有 X2 接口，X2 接口使能 UE 在进入空闲状态后，快速启动 resume 流程，接入到其它 eNB（resume 流程将在本文后面详述）。

### 5.1.1.3. 频段

NB-IoT 沿用 LTE 定义的频段号，Release 13 为 NB-IoT 指定了 14 个频段。

Band Number	Uplink frequency range / MHz	Downlink frequency range / MHz
1	1920 - 1980	2110 - 2170
2	1850 - 1910	1930 - 1990
3	1710 - 1785	1805 - 1880
5	824 - 849	869 - 894
8	880 - 915	925 - 960
12	699 - 716	729 - 746
13	777 - 787	746 - 756
17	704 - 716	734 - 746
18	815 - 830	860 - 875
19	830 - 845	875 - 890
20	832 - 862	791 - 821
26	814 - 849	859 - 894
28	703 - 748	758 - 803
66	1710 - 1780	2100-2200

图 5-3 频段

### 5.1.2. 物理层

物理层设计	下行	上行
多址技术	OFDMA	SC-FDMA
子载波带宽	15KHz	3.75KHz/15KHz
发射功率	43dBm	23dBm
帧长度	1ms	1ms
TTI长度	1ms	1ms/8ms
SCH低阶调制	QPSK	BPSK
SCH高阶调制	QPSK	QPSK
符号重复最大次数	32	32

图 5-4 物理层参数

### 5.1.2.1. 工作模式

#### 1. 部署方式（Operation Modes）

NB-IoT 占用 180KHz 带宽，这与在 LTE 帧结构中一个资源块的带宽是一样的。所以，以下三种部署方式成为可能：

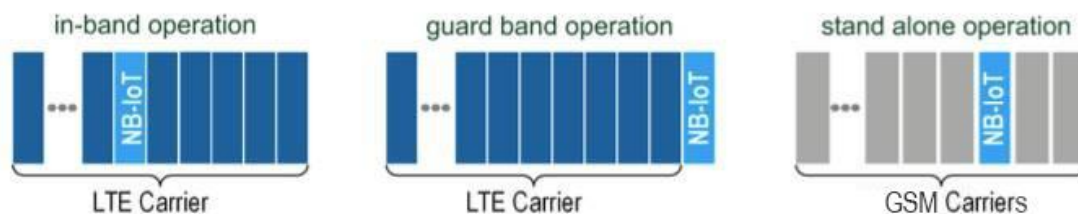


图 5-5 部署方式

- 独立部署（Stand alone operation）

适合用于重耕 GSM 频段，GSM 的信道带宽为 200KHz，这刚好为 NB-IoT 180KHz 带宽辟出空间，且两边还有 10KHz 的保护间隔。

- 保护带部署（Guard band operation）

利用 LTE 边缘保护频带中未使用的 180KHz 带宽的资源块。

- 带内部署（In-band operation）

利用 LTE 载波中间的任何资源块。

#### 2. CE Level

CE Level，即覆盖增强等级（Coverage Enhancement Level）。从 0 到 2，CE Level 共三个等级，分别对应可对抗 144dB、154dB、164dB 的信号衰减。基站与 NB-IoT 终端之间会根据其所在的 CE Level 来选择相对应的信息重发次数。

#### 3. 双工模式

Release 13 NB-IoT 仅支持 FDD 半双工 type-B 模式。

FDD 意味着上行和下行在频率上分开，UE 不会同时处理接收和发送。

半双工设计意味着只需多一个切换器去改变发送和接收模式，比起全双工所需的元件，成本更低廉，且可降低电池能耗。

在 Release 12 中，定义了半双工分为 type A 和 type B 两种类型，其中 type B 为 Cat.0 所用。在 type A 下，UE 在发送上行信号时，其前面一个子帧的下行

信号中最后一个 Symbol 不接收，用来作为保护时隙（Guard Period, GP），而在 type B 下，UE 在发送上行信号时，其前面的子帧和后面的子帧都不接收下行信号，使得保护时隙加长，这对于设备的要求降低，且提高了信号的可靠性。

### 5.1.2.2. 下行链路

对于下行链路，NB-IoT 定义了三种物理信道：

- NPBCH，窄带物理广播信道。
- NPDCCH，窄带物理下行控制信道。
- NPDSCH，窄带物理下行共享信道。

还定义了两种物理信号：

- NRS，窄带参考信号。
- NPSS 和 NSSS，主同步信号和辅同步信号。

相比 LTE，NB-IoT 的下行物理信道较少，且去掉了 PMCH（Physical Multicast channel，物理多播信道），原因是 NB-IoT 不提供多媒体广播/组播服务。

下图是 NB-IoT 传输信道和物理信道之间的映射关系。

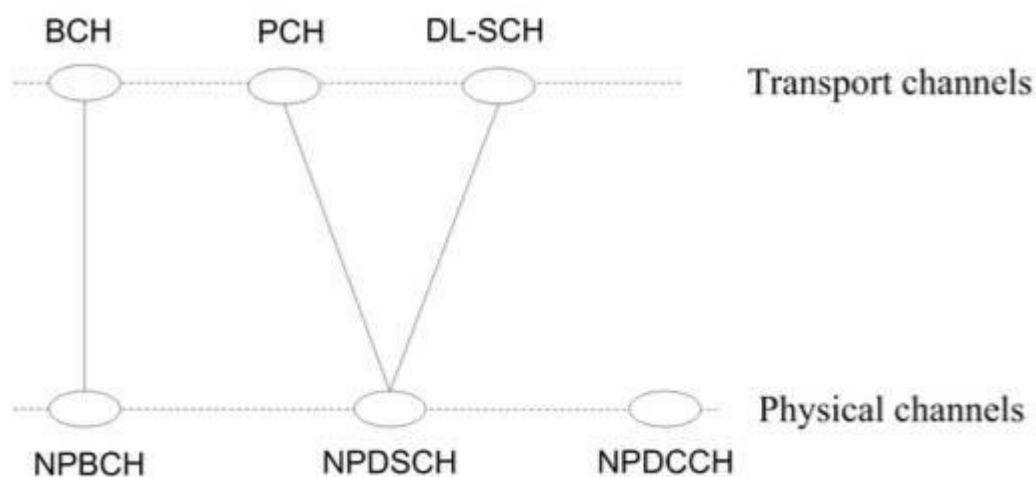


图 5-6 映射关系图

MIB 消息在 NPBCH 中传输，其余信令消息和数据在 NPDSCH 上传输，NPDCCH 负责控制 UE 和 eNB 间的数据传输。

NB-IoT 下行调制方式为 QPSK。NB-IoT 下行最多支持两个天线端口（Antenna Port），AP0 和 AP1。和 LTE 一样，NB-IoT 也有 PCI（Physical Cell ID，物理小区标识），称为 NCellID（Narrowband physical cell ID），一共定义了 504 个 NCellID。

### 1. 帧和时隙结构

和 LTE 循环前缀（Normal CP）物理资源块一样，在频域上由 12 个子载波（每个子载波宽度为 15KHz）组成，在时域上由 7 个 OFDM 符号组成 0.5ms 的时隙，这样保证了和 LTE 的相容性，对于带内部署方式至关重要。

每个时隙 0.5ms，2 个时隙就组成了一个子帧（SF），10 个子帧组成一个无线帧（RF）。下图是 NB-IoT 的帧结构，依然和 LTE 一样。

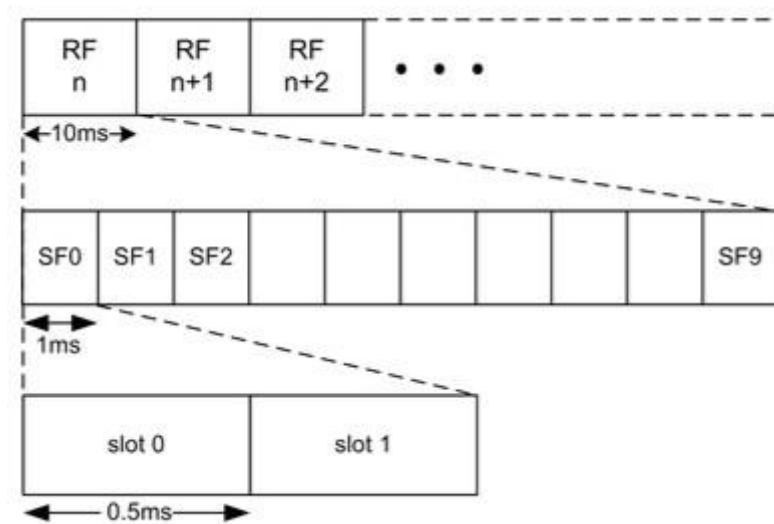


图 5-7 帧结构

### 2. NRS（窄带参考信号）

NRS（窄带参考信号），也称为导频信号，主要作用是下行信道质量测量估计，用于 UE 端的相干检测和解调。在用于广播和下行专用信道时，所有下行子帧都要传输 NRS，无论有无数据传送。

NB-IoT 下行最多支持两个天线端口，NRS 只能在一个天线端口或两个天线端口上传输，资源的位置在时间上与 LTE 的 CRS（Cell-Specific Reference Signal，小区特定参考信号）错开，在频率上则与之相同，这样在带内部署（In-Band Operation）时，若检测到 CRS，可与 NRS 共同使用来做信道估测。

### 3. 同步信号

NPSS 为 NB-IoT UE 时间和频率同步提供参考信号，与 LTE 不同的是，NPSS 中不携带任何小区信息，NSSS 带有 PCI。NPSS 与 NSSS 在资源位置上避开了 LTE 的控制区域。

NPSS 的周期是 10ms，NSSS 的周期是 20ms。NB-IoT UE 在小区搜索时，会先检测 NPSS，因此 NPSS 的设计为短的 ZC(Zadoff-Chu)序列，这降低了初步信号检测和同步的复杂性。

#### 4. NBPBCH

NBPBCH 的 TTI 为 640ms，承载 MIB-NB（Narrowband Master Information Block），其余系统信息如 SIB1-NB 等承载于 NPDSCH 中。SIB1-NB 为周期性出现，其余系统信息则由 SIB1-NB 中所带的排程信息做排程。和 LTE 一样，NB-PBCH 端口数通过 CRC mask 识别，区别是 NB-IOT 最多只支持 2 端口。NB-IOT 在解调 MIB 信息过程中确定小区天线端口数。

在三种 operation mode 下，NB-PBCH 均不使用前 3 个 OFDM 符号。In-band 模式下 NBPBCH 假定存在 4 个 LTE CRS 端口，2 个 NRS 端口进行速率匹配。

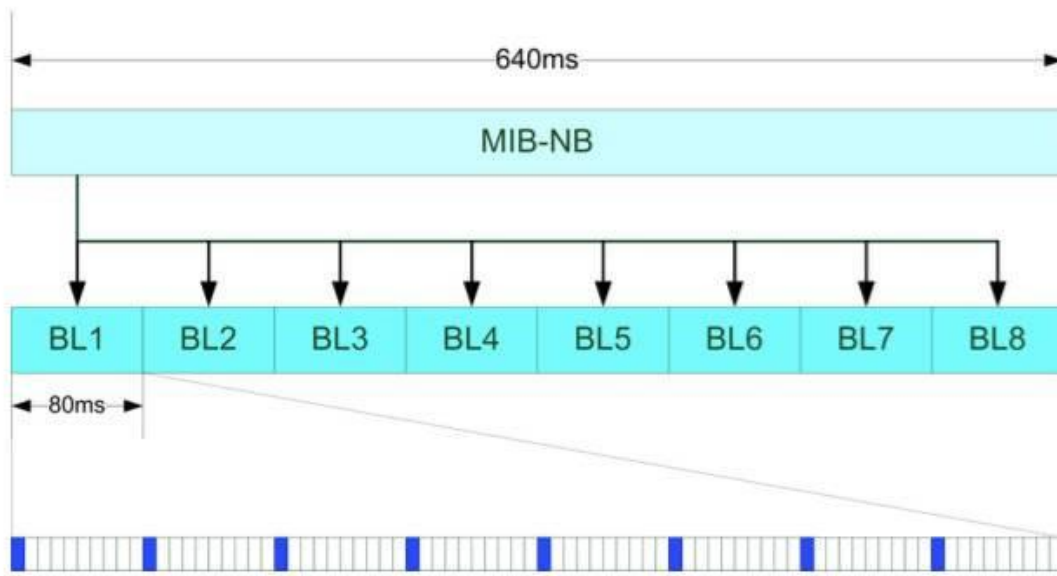


图 5-8 NPBCH 映射到子帧

#### 5. NPDCCH

NPDCCH 中承载的是 DCI（Downlink Control Information），包含一个或多个 UE 上的资源分配和其他的控制信息。UE 需要首先解调 NPDCCH 中的 DCI，然后才能够在相应的资源位置上解调属于 UE 自己的 NPDSCH（包括广



播消息，寻呼，UE 的数据等）。NPDCCH 包含了 UL grant，以指示 UE 上行数据传输时所使用的资源。

NPDCCH 子帧设计如下图所示：

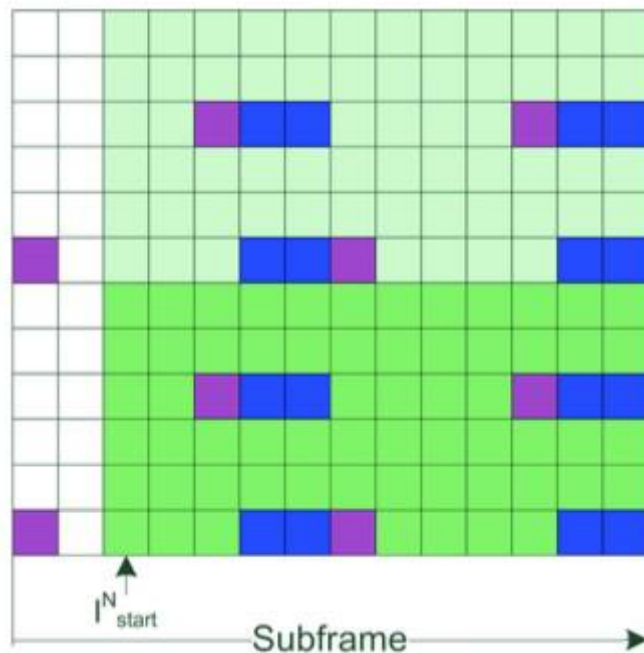


图 5-9 NPDCCH 子帧设计

图中浅绿色和深绿色代表 NPDCCH 使用的 RE，紫色代表 LTE CRS，蓝色代表 NRS。上图表示在 LTE 单天线端口和 NB-IoT2 天线端口下 in-band 模式的映射。

NPDCCH 的符号起始位置：对于 in-band，如果是 SIB 子帧，起始位置为 3，非 SIB 子帧，起始位置包含在 SIB2-NB 中；对于 stand-alone 和 Guard band，起始位置统一为 0。

NPDCCH 有别于 LTE 系统中的 PDCCH 的是，并非每个 Subframe 都有 NPDCCH，而是周期性出现。NPDCCH 有三种搜索空间(Search Space)，分别用于排程一般数据传输、Random Access 相关信息传输，以及寻呼(Paging)信息传输。

各个 Search Space 有无线资源控制(RRC)配置相对应的最大重复次数  $R_{max}$ ，其 Search Space 的出现周期大小即为相应的  $R_{max}$  与 RRC 层配置的一参数的乘积。

RRC 层也可配置一偏移(Offset)以调整 Search Space 的开始时间。在大部分的搜索空间配置中,所占用的资源大小为一 PRB,仅有少数配置为占用 6 个 Subcarrier。

一个 DCI 中会带有该 DCI 的重传次数,以及 DCI 传送结束后至其所排程的 NPDSCH 或 NPUSCH 所需的延迟时间,NB-IoT UE 即可使用此 DCI 所在的 Search Space 的开始时间,来推算 DCI 的结束时间以及排程的数据的开始时间,以进行数据的传送或接收。

## 6. NPDSCH

NPDSCH 的子帧结构和 NPDCCH 一样。

NPDSCH 是用来传送下行数据以及系统信息,NPDSCH 所占用的带宽是整个 PRB 大小。一个传输块 (Transport Block, TB) 依据所使用的调制与编码策略(MCS),可能需要使用多于一个子帧来传输,因此在 NPDCCH 中接收到的 Downlink Assignment 中会包含一个 TB 对应的子帧数目以及重传次数指示。

### 5.1.2.3. 上行链路

对于上行链路,NB-IoT 定义了两种物理信道:

- NPUSCH, 窄带物理上行共享信道。
- NPRACH, 窄带物理随机接入信道。

还有:

- DMRS, 上行解调参考信号。

#### 1. 时隙结构

NB-IoT 上行使用 SC-FDMA,考虑到 NB-IoT 终端的低成本需求,在上行要支持单频(Single Tone)传输,子载波间隔除了原有的 15KHz,还新制订了 3.75KHz 的子载波间隔,共 48 个子载波。

当采用 15KHz 子载波间隔时,资源分配和 LTE 一样。

15KHz 为 3.75KHz 的整数倍,所以对 LTE 系统干扰较小。由于下行的帧结构与 LTE 相同,为了使上行与下行相容,子载波空间为 3.75KHz 的帧结构中,一个时隙同样包含 7 个 Symbol,共 2ms 长,刚好是 LTE 时隙长度的 4 倍。

此外，NB-IoT 系统中的采样频率(Sampling Rate)为 1.92MHz，子载波间隔为 3.75KHz 的帧结构中，一个 Symbol 的时间长度为 512Ts(Sampling Duration)，加上循环前缀(Cyclic Prefix, CP)长 16Ts，共 528Ts。因此，一个时隙包含 7 个 Symbol 再加上保护区间(Guard Period)共 3840Ts，即 2ms 长。

## 2. NPUSCH

NPUSCH 用来传送上行数据以及上行控制信息。NPUSCH 传输可使用单频或多频传输。

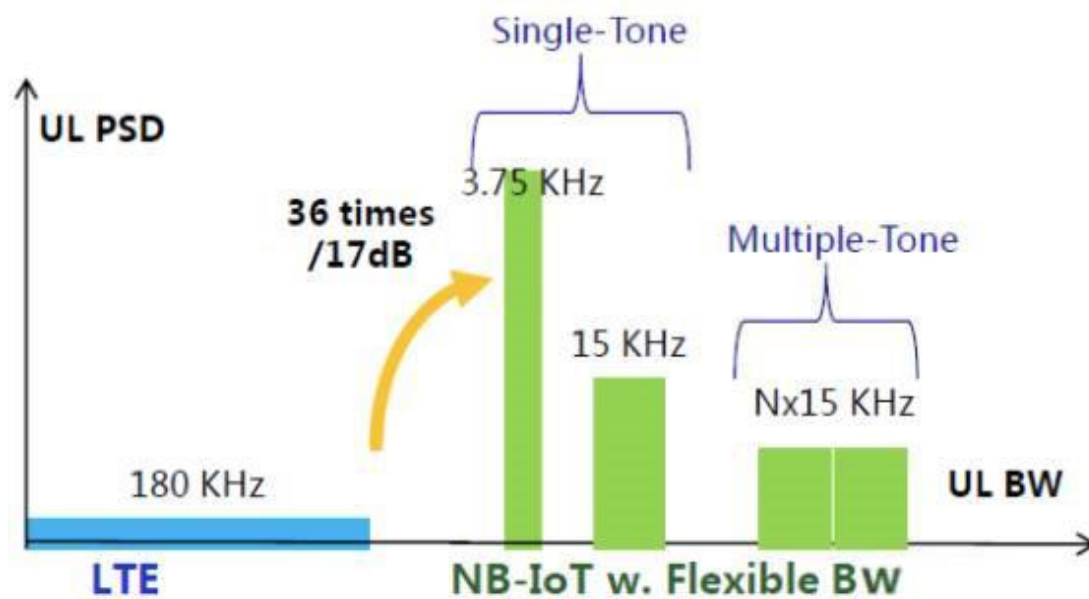


图 5-10 单频与多频传输

在 NPUSCH 上，定义了两种格式：format 1 和 format 2。NPUSCH format 1 为 UL-SCH 上的上行信道数据而设计，其资源块不大于 1000 bits；NPUSCH format 2 传送上行控制信息（UCI）。

映射到传输块的最小单元叫资源单元（RU，resource unit），它由 NPUSCH 格式和子载波空间决定。

有别于 LTE 系统中的资源分配的基本单位为子帧，NB-IoT 根据子载波和时隙数目来作为资源分配的基本单位。

当子载波空间为 3.75 kHz 时，只支持单频传输，一个 RU 在频域上包含 1 个子载波，在时域上包含 16 个时隙，所以，一个 RU 的长度为 32ms。

当子载波空间为 15kHz 时，支持单频传输和多频传输，一个 RU 包含 1 个子载波和 16 个时隙，长度为 8ms；当一个 RU 包含 12 个子载波时，则有 2 个

时隙的时间长度，即 1ms，此资源单位刚好是 LTE 系统中的一个子帧。资源单位的时间长度设计为 2 的幂次方，是为了更有效的运用资源，避免产生资源空隙而造成资源浪费。

对于 NPUSCH format 2，RU 总是由 1 个子载波和 4 个时隙组成，所以，当子载波空间为 3.75 kHz 时，一个 RU 时长为 8ms；当子载波空间为 15kHz 时，一个 RU 时长为 2ms。

对于 NPUSCH format 2，调制方式为 BPSK。

对于 NPUSCH format 1，调制方式分为以下两种情况：

- 包含一个子载波的 RU，采用 BPSK 和 QPSK。
- 其它情况下，采用 QPSK。

由于一个 TB 可能需要使用多个资源单位来传输，因此在 NPDCCH 中接收到的 Uplink Grant 中除了指示上行数据传输所使用的资源单位的子载波的索引 (Index)，也会包含一个 TB 对应的资源单位数目以及重传次数指示。

NPUSCH Format 2 是 NB-IoT 终端用来传送指示 NPDSCH 有无成功接收的 HARQ-ACK/NACK，所使用的子载波的索引(Index)是在由对应的 NPDSCH 的下行分配(Downlink Assignment)中指示，重传次数则由 RRC 参数配置。

### 3. DMRS

根据 NPUSCH 格式，DMRS 每时隙传输 1 个或者 3 个 SC-FDMA 符号。

### 4. NPRACH

与 LTE 的 Random Access Preamble 使用 ZC 序列不同，NB-IoT 的 Random Access Preamble 是单频传输（3.75KHz 子载波），且使用的 Symbol 为一定值。一次的 Random Access Preamble 传送包含四个 Symbol Group，一个 Symbol Group 是 5 个 Symbol 加上一 CP。

每个 Symbol Group 之间会有跳频。选择传送的 Random Access Preamble 即是选择起始的子载波。

基站会根据各个 CE Level 去配置相应的 NPRACH 资源。

Random Access 开始之前，NB-IoT 终端会通过 DL measurement（比如 RSRP）来决定 CE Level，并使用该 CE Level 指定的 NPRACH 资源。一旦 Random Access Preamble 传送失败，NB-IoT 终端会在升级 CE Level 重新尝试，直到尝试完所有 CE Level 的 NPRACH 资源为止。

### 5.1.3. 小区接入

NB-IoT 的小区接入流程和 LTE 差不多：小区搜索取得频率和符号同步、获取 SIB 信息、启动随机接入流程建立 RRC 连接。当终端返回 RRC\_IDLE 状态，当需要进行数据发送或收到寻呼时，也会再次启动随机接入流程。

#### 5.1.3.1. 协议栈和信令承载

总的来说，NB-IoT 协议栈基于 LTE 设计，但是根据物联网的需求，去掉了一些不必要的功能，减少了协议栈处理流程的开销。因此，从协议栈的角度看，NB-IoT 是新的空口协议。

以无线承载（RB）为例，在 LTE 系统中，SRB（signalling radio bearers，信令无线承载）会部分复用，SRB0 用来传输 RRC 消息，在逻辑信道 CCCH 上传输；而 SRB1 既用来传输 RRC 消息，也会包含 NAS 消息，其在逻辑信道 DCCH 上传输。

LTE 中还定义了 SRB2，但 NB-IoT 没有。

此外，NB-IoT 还定义一种新的信令无线承载 SRB1bis，SRB1bis 和 SRB1 的配置基本一致，除了没有 PDCP，这也意味着在 Control Plane ClIoT EPS optimisation 下只有 SRB1bis，因为只有在这种模式才不需要。

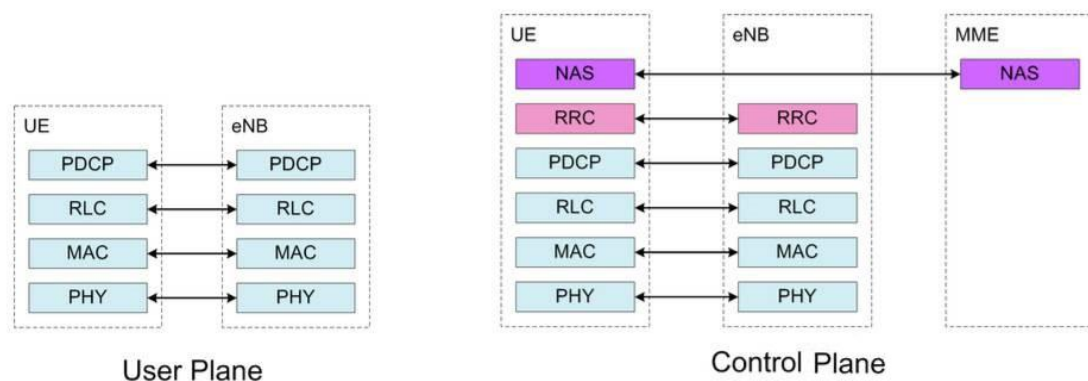


图 5-11 NB-IoT 协议栈

#### 5.1.3.2. 系统信息

NB-IoT 经过简化，去掉了一些对物联网不必要的 SIB，只保留了 8 个：

System Information Block	Content
MIB-NB	Essential information required to receive further system information
SIBType1-NB	Cell access and selection, other SIB scheduling
SIBType2-NB	Radio resource configuration information
SIBType3-NB	Cell re-selection information for intra-frequency, inter-frequency
SIBType4-NB	Neighboring cell related information relevant for intra-frequency cell re-selection
SIBType5-NB	Neighboring cell related information relevant for inter-frequency cell re-selection
SIBType14-NB	Access Barring parameters
SIBType16-NB	Information related to GPS time and Coordinated Universal Time (UTC)

图 5-12 系统信息分类

- SIBType1-NB: 小区接入和选择, 其它 SIB 调度
- SIBType2-NB: 无线资源分配信息
- SIBType3-NB: 小区重选信息
- SIBType4-NB: Intra-frequency 的邻近 Cell 相关信息
- SIBType5-NB: Inter-frequency 的邻近 Cell 相关信息
- SIBType14-NB: 接入禁止(Access Barring)
- SIBType16-NB: GPS 时间/世界标准时间信息

需特别说明的是, SIB-NB 是独立于 LTE 系统传送的, 并非夹带在原 LTE 的 SIB 之中。

### 5.1.3.3. 小区重选和移动性

由于 NB-IoT 主要为非频发小数据包流量而设计, 所以 RRC\_CONNECTED 中的切换过程并不需要, 被移除了。如果需要改变服务小区, NB-IoT 终端会进行 RRC 释放, 进入 RRC\_IDLE 状态, 再重选至其他小区。

在 RRC\_IDLE 状态，小区重选定义了 intra frequency 和 inter frequency 两类小区，inter frequency 指的是 in-band operation 下两个 180 kHz 载波之间的重选。

NB-IoT 的小区重选机制也做了适度的简化，由于 NB-IoT 终端不支持紧急拨号功能，所以，当终端重选时无法找到 Suitable Cell 的情况下，终端不会暂时驻扎(Camp)在 Acceptable Cell，而是持续搜寻直到找到 Suitable Cell 为止。根据 3GPP TS 36.304 定义，所谓 Suitable Cell 为可以提供正常服务的小区，而 Acceptable Cell 为仅能提供紧急服务的小区。

#### 5.1.3.4. 随机接入过程

NB-IoT 的 RACH 过程和 LTE 一样，只是参数不同。

基于竞争的 NB-IOT 随机接入过程：

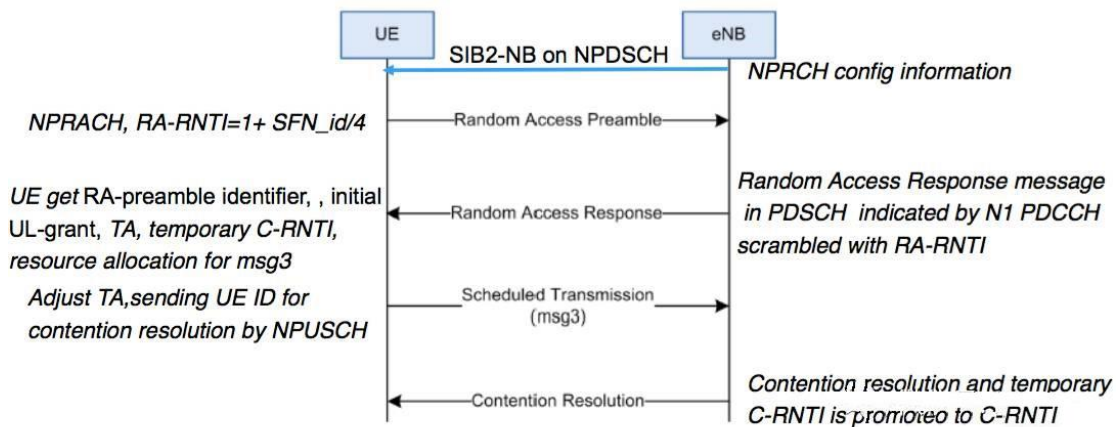


图 5-13 基于竞争的 NB-IoT 随机接入过程

基于非竞争的 NB-IOT 随机接入过程：

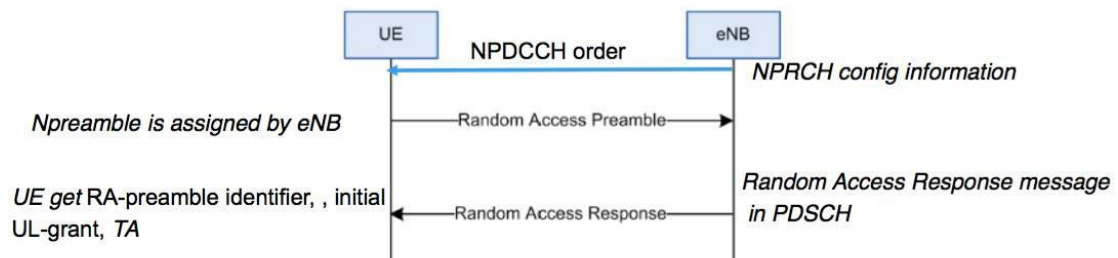


图 5-14 基于非竞争的 NB-IoT 随机接入过程

### 5.1.3.5. 连接管理

由于 NB-IoT 并不支持不同技术间的切换，所以 RRC 状态模式也非常简单。

- RRC Connection Establishment

RRC Connection Establishment 流程和 LTE 一样，但内容却不相同。

很多原因都会引起 RRC 建立，但是，在 NB-IoT 中，RRCConnectionRequest 中的 Establishment Cause 里没有 delayTolerantAccess，因为 NB-IOT 被预先假设为容忍延迟的。

另外，在 Establishment Cause 里，UE 将说明支持单频或多频的能力。与 LTE 不同的是，NB-IoT 新增了 Suspend-Resume 流程。当基站释放连接时，基站会下达指令让 NB-IoT 终端进入 Suspend 模式，该 Suspend 指令带有一组 Resume ID，此时，终端进入 Suspend 模式并存储当前的 AS context。

当终端需要再次进行数据传输时，只需要在 RRC Connection Resume Request 中携带 Resume ID，基站即可通过此 Resume ID 来识别终端，并跳过相关配置信息交换，直接进入数据传输。

简而言之，在 RRC\_Connected 至 RRC\_IDLE 状态时，NB-IoT 终端会尽可能的保留 RRC\_Connected 下所使用的无线资源分配和相关安全性配置，减少两种状态之间切换时所需的信息交换数量，以达到省电的目的。

### 5.1.4. 数据传输

如前文所述，NB-IoT 定义了两种数据传输模式：Control Plane CIoT EPS optimisation 方案和 User Plane CIoT EPS optimisation 方案。对于数据发起方，由终端选择决定哪一种方案。对于数据接收方，由 MME 参考终端习惯，选择决定哪一种方案。

#### 5.1.4.1. Control Plane CIoT EPS Optimisation

对于 Control Plane CIoT EPS Optimisation，终端和基站间的数据交换在 RRC 级上完成。对于下行，数据包附带在 RRCConnectionSetup 消息里；对于



上行，数据包附带在 `RRCCConnectionSetupComplete` 消息里。如果数据量过大，RRC 不能完成全部传输，将使用 `DLInformationTransfer` 和 `ULInformationTransfer` 消息继续传送。

这两类消息中包含的是带有 NAS 消息的 byte 数组，其对应 NB-IoT 数据包，因此，对于基站是透明的，UE 的 RRC 也会将它直接转发给上一层。

在这种传输模式下，没有 RRC connection reconfiguration 流程，数据在 RRC connection setup 消息里传送，或者在 RRC connection setup 之后立即 RRC connection release 并启动 resume 流程。

#### 5.1.4.2. User Plane CIoT EPS optimisation

在 User Plane CIoT EPS optimisation 模式下，数据通过传统的用户面传送，为了降低物联网终端的复杂性，只可以同时配置一个或两个 DRB。

此时，有两种情况：

##### 1. 情景一：

当 RRC 连接释放时，RRC 连接释放会携带 Resume ID，并启动 resume 流程，如果 resume 成功，更新密钥安全建立后，保留了先前 RRC\_Connected 的无线承载也随之建立。

##### 2. 情景二：

当 RRC 连接释放时，如果 RRC 连接释放没有携带 Resume ID，或者 resume 请求失败。首先，通过 `SecurityModeCommand` 和 `SecurityModeComplete` 建立 AS 级安全。

在 `SecurityModeCommand` 消息中，基站使用 SRB1 和 DRB 提供加密算法和对 SRB1 完整性保护。LTE 中定义的所有算法都包含在 NB-IoT 里。

当安全激活后，进入 RRC connection reconfiguration 流程建立 DRBs。

在重配置消息中，基站为 UE 提供无线承载，包括 RLC 和逻辑信道配置。PDCP 仅配置于 DRBs，因为 SRB 采用默认值。在 MAC 配置中，将提供 BSR、SR、DRX 等配置。最后，物理配置提供将数据映射到时隙和频率的参数。

### 5.1.4.3. 多载波配置

在 RRCConnectionReconfiguration 消息中，可在上下行设置一个额外的载波，称为非锚定载波（non-anchor carrier）。

基于多载波配置，系统可以在一个小区里同时提供多个载波服务，因此，NB-IoT 的载波可以分为两类：提供 NPSS、NSSS 与承载 NPBCH 和系统信息的载波称为 Anchor Carrier，其余的载波则称为 Non-Anchor Carrier。

当提供 non-anchor 载波时，UE 在此载波上接收所有数据，但同步、广播和寻呼等消息只能在 Anchor Carrier 上接收。

NB-IoT 终端一律需要在 Anchor Carrier 上面 Random Access，基站会在 Random Access 过程中传送 Non-Anchor Carrier 调度信息，以将终端卸载至 Non-Anchor Carrier 上进行后续数据传输，避免 Anchor Carrier 的无线资源吃紧。

另外，单个 NB-IoT 终端同一时间只能在一个载波上传送数据，不允许同时在 Anchor Carrier 和 Non-Anchor Carrier 上传送数据。

## 5.2. NB-IoT 节点上行通讯示例

NB-IoT 节点定时向指定服务器发送数据，服务器需要提供公网 IP，通讯协议采用 UDP 协议。测试时使用串口工具连接 NB-IoT 通信模块，采用 AT 指令调试。

### 5.2.1. AT 指令调试

打开串口调试工具设置：9600.8.1.N.N 并输入以下 AT 指令(指令后面需要加回车符)：

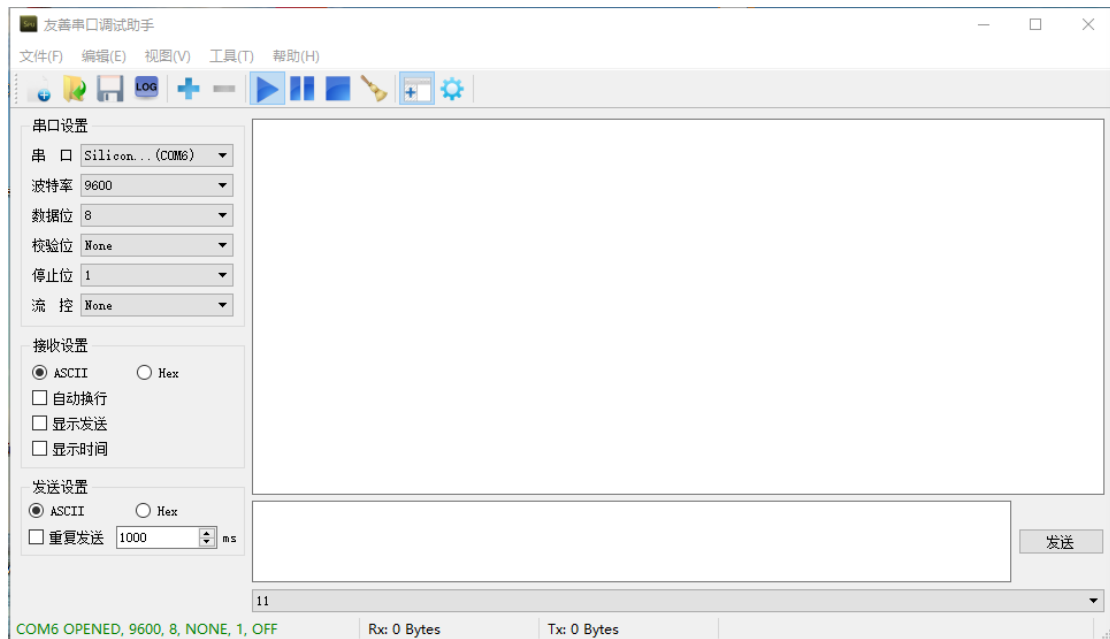


图 5-15 串口工具

1. 查看模块能否通讯。

```
/*发送*/
AT
/*接收*/
OK
```

2. 重启模块。

```
/*发送*/
AT+NRB
/*接收*/
REBOOTING
?0
REBOOT_CAUSE_APPLICATION_AT
Neul
OK
```

3. 查看 IMEI 号（如果返回 ERROR，说明模块没有设置 IMEI 需要手动设置）。

```
/*发送*/
AT+CGSN=1
/*接收*/
+CGSN:865352030093883
OK
```

4. 设置 IMEI，如果已经设置则跳过此步骤。

```
/*发送*/
AT+NTSETID=1, 201612091450303
```

```
/*接收*/
OK
```

5. 设置自动连接网络。

```
/*发送*/
AT+NCONFIG=AUTOCONNECT,TRUE
/*接收*/
OK
```

6. 打开模块全功能模式。

```
/*发送*/
AT+CFUN=1
/*接收*/
OK
```

7. 查询 SIM 卡号（如果没有卡号返回，说明没检测到 SIM 卡）。

```
/*发送*/
AT+CIMI
/*接收*/
460111174764199
OK
```

8. 查询网络附着状态（返回 1 代表附着成功，如果返回 0 则多试几次）。

```
/*发送*/
AT+CGATT?
/*接收*/
+CGATT:1
OK
```

9. 创建 UDP socket（DGRAM 是 socket 类型，是 UDP 协议，8091 是监听的端口）。

```
/*发送*/
AT+NSOCR=DGRAM,17,8091,1
/*接收*/
0
OK
```

10. 发送 UDP 数据（0 是 socket 号,120.55.62.150 是远程服务器 IP，50000 是端口号，5 是数据长度，4142434445 代表**十六进制** 0x41，0x42，0x43，0x44，0x45，即**ASCLL 码**）。

**其中远程服务器的 IP、端口为公网地址，或者是经过端口映射的地址，示例中的 IP、端口仅供参考！**

```
/*发送*/
AT+NSOST=0,120.55.62.150,50000,5,4142434445
```

```
/*接收*/
```

```
0,5
```

```
OK
```

按照以上操作登录服务器端就可以看到数据，使用 nc（Linux 系统下端口监听命令）工具监听相应端口，示例如下：

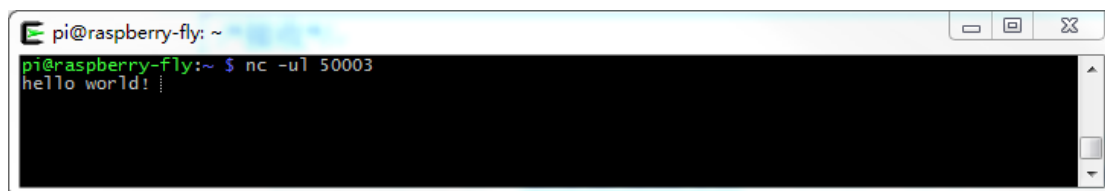


图 5-16 服务器接受到的数据

### 5.2.2. 应用代码介绍

打开工程 NB-IoT\_training\_platform\NB-IoT\keil\nb-node\nb-node.uvprojx，工程目录如下：

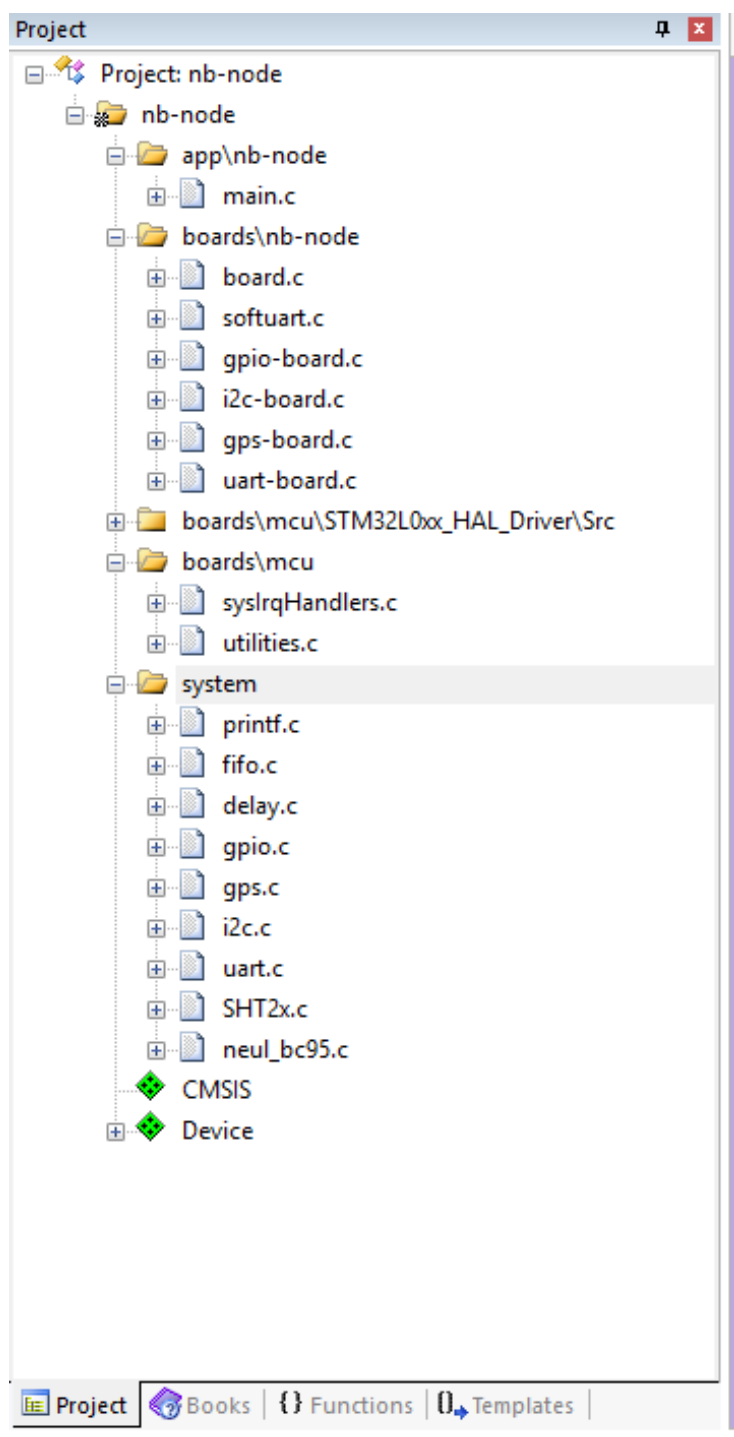


图 5-17 工程目录

目录结构:

- nb-node:工程文件夹
- app\nb-node\:应用文件
- boards\nb-node\:依赖于开发板的文件

- boards\mcu\stm32L0xx\_HAL\_Driver\src:STM32L0 系类的官方驱动库文件夹
- boards\mcu\stm32\;STM32 系列的共用系统文件
- system\;系统资源文件

## 1. 打开终端节点 nb-node 示例程序

NB-IoT\_training\_platform\NB-IoT\keil\nb-node\nb-node.uvprojx

## 2. 配置节点

### ➤ config.h

使用节点前需要对 config.h 文件进行配置，各个参数说明如下：

**REMOTE\_IPV4:** 此参数为转发服务器公网 IP,NB-IOT 终端节点的数据首先会发往这个转发服务器，转发服务器会将数据转发给各个用户的 NB-IoT 本地服务器，供用户使用。**选择默认值即可，若修改了此 IP，需要保证对应服务器有数据转发服务，否则数据转发功能失效，本地服务将获取不到终端节点的数据。**

**REMOTE\_PORT:** 转发服务器的端口，**选择默认值即可。若修改参数，需确保对应端口具备数据转发功能。**

**SERVER\_ADDR:** NB-IoT 本地服务器 MAC 地址，需要用户根据各自的本地服务器 MAC 地址进行修改。

**NODE\_ID:** 节点 ID，用户可随意设置，但要确保**同一个 NB-IoT 本地服务器的各个节点地址都是唯一的。**

```
/*服务器 ip,需要公网 ip （默认，无需修改）*/
#define REMOTE_IPV4 "120.55.62.150"

/*转发服务器端口（默认，无需修改）*/
#define REMOTE_PORT 50003

/*用户终端服务器地址（需要修改）*/
#define SERVER_ADDR {0x00,0x00,0x00,0xff,0xff,0xff}

/*节点 ID（需要修改）*/
#define NODE_ID {0x00, 0x00, 0x00, 0x05}

/*数据上报周期（需要修改，单位为 ms，该示例中周期需要大于 30000ms）*/
/*
```

```
#define APP_TX_DUTYCYCLE 60000
```

### 3. 工程配置

打开工程配置界面并配置相关信息。

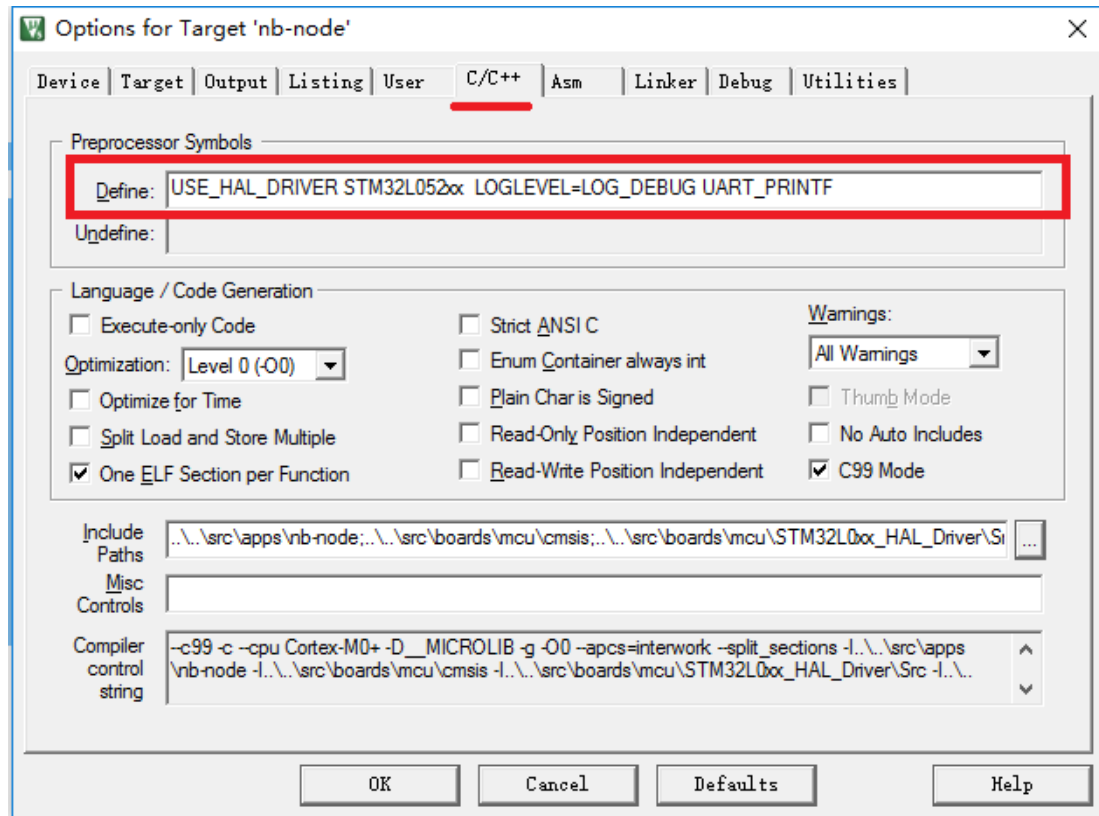


图 5-18 宏配置界面

宏说明如下：

**USE\_HAL\_DRIVER:**使用官方驱动库文件。（默认，无需修改）

**STM32L052xx :**选择芯片系列。（默认，无需修改）

**LOGLEVEL=LOG\_DEBUG:**printf 打印等级，根据实际情况确定如何配置。

共分四种情况：LOG\_ERR、LOG\_WARN、LOG\_INFO、LOG\_DEBUG。  
打印数量由少到多。（调试时选择 **LOG\_DEBUG** 即可）

**UART\_PRINTF:**开启串口 printf 功能，通过 uart2 打印数据。（默认开启）

### 4. JLINK 下载配置

打开工程配置，选择 Debug->Setting,选择 SWD 接口下载：



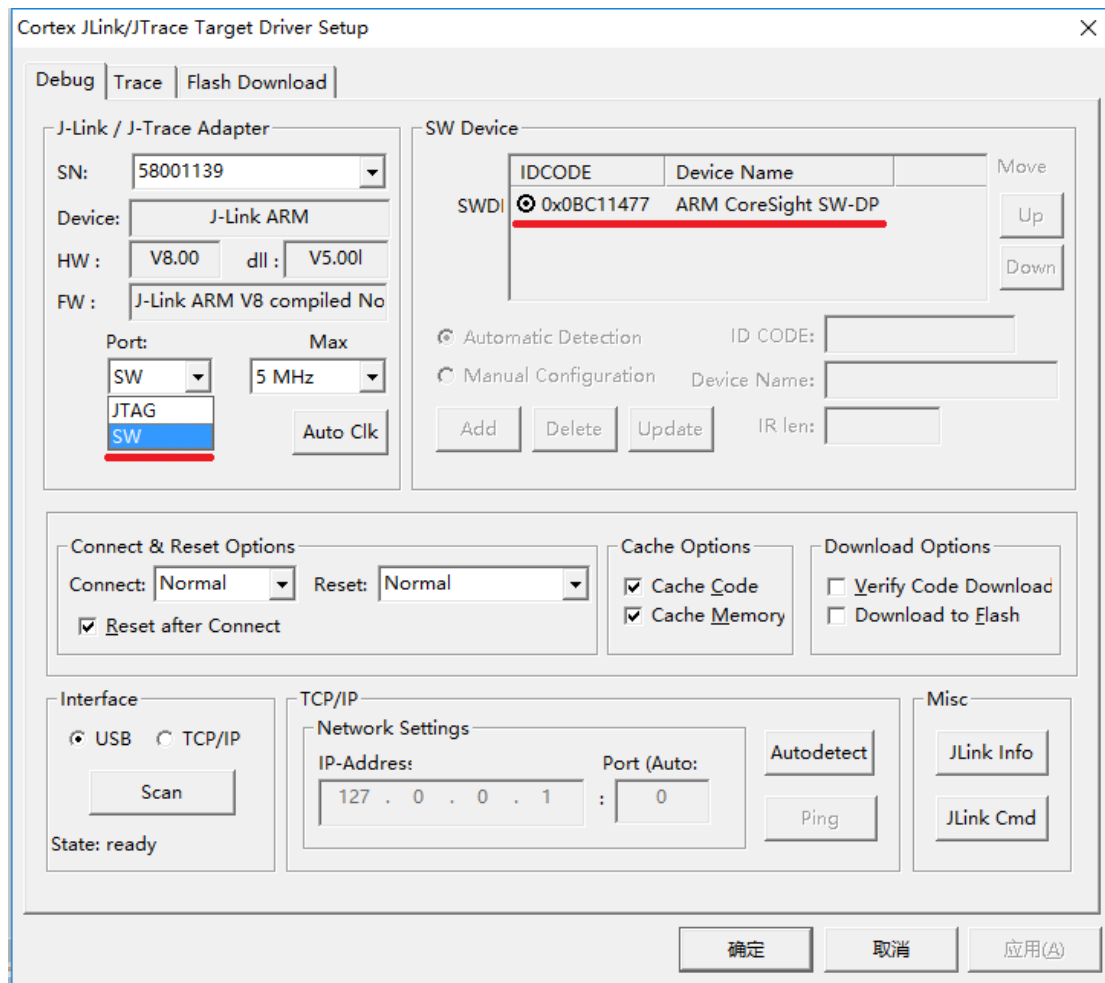


图 5-19 下载器设置

### ➤ Config.h

配置节点参数：

```
/*转发服务器 ip,需要公网 ip*/
#define REMOTE_IPV4 "120.55.62.150"
/*转发服务器端口*/
#define REMOTE_PORT 50000
/*本地服务器地址*/
#define SERVER_ADDR {0x00,0x00,0x00,0xff,0xff,0xff}
/*节点 ID，用来区分同一个转发服务器中的各个节点*/
#define NODE_ID {0x00, 0x00, 0x00, 0x01}
/*修改数据上报周期*/
#define APP_TX_DUTYCYCLE 60000
```

### ➤ Main.c

主程序工作流程：

```
int main(){
    /*初始化时钟 UART */
    BoardInitMcu();
```

```

/*初始化看门狗，每 3s 喂狗一次 */
iwdog_init();
timer_create(&iwdg_refresh_timer,          IWDG_REFRESH_DUTYCYCLE,
OPT_TMR_PERIODIC, iwdg_refresh_event);
timer_start(&iwdg_refresh_timer);

/*创建一个周期为 APP_TX_DUTYCYCLE 的定时器，用来周期上报数据
*/
timer_create(&appdata_send_timer,APP_TX_DUTYCYCLE,OPT_TMR_PERIODIC, appdata_send_event);
/*初始化 NB-IOT 模块，通过发送 AT 指令和模块通讯使其初始化*/
while((ret = neul_bc95_hw_init()) != 0)
{
    debug("init fail,reboot.\n");
}
debug("init success.\n");

/*附着网络*/
net_attch();

/*发送 AT+NSOCR=DGRAM,17,8091,1 创建 UDP SOCKET*/
debug("creat udp socket...\n");
if((ret = neul_bc95_create_udpsocket(8091)) >= 0)
{
    udpsocket = ret;
    debug("ok.\n");
}
while(1)
{
    .....
    /*发送 UDP 数据*/
}
}

```

查询附着网络：

```

void net_attch()
{
    uint8_t try_cnt = 0;
    int ret = 0;

    while(1)
    {
        /*AT+NRB 重启*/
        neul_bc95_reboot();
    }
}

```

```

/*AT+CGATT 查询网络*/
do{
    log("get net status:%d\n", try_cnt);
    ret = neul_bc95_get_netstat();
    if(!ret)
    {
        log("connected.\n");
        return ;
    }
    neul_bc95_sleep(5000);
} while(ret < 0 && ++try_cnt < GET_NETSTAT_RETRY_MAX);
try_cnt = 0;
log("no network connect.reboot!\n");
}
}

```

发送 UDP 数据：

```

bool udp_uplink()
{
    int ret = 0;
    uint8_t i;

    /*是否准备发送*/
    if(is_appdata_prepare == false)
        return false;

    /*数据打包*/
    prepare_data();

    /*log 打印*/
    log("udp send...\n");
    for(i = 0; i < APP_DATA_SIZE; i++)
    {
        log("data[%i]:%x\n", i, app_data[i]);
    }

    /*通过 AT+NSOST 发送 UDP 数据*/
    ret = neul_bc95_udp_send(udpsocket, (const char *)app_data,
APP_DATA_SIZE);
    if(ret == 0)
    {
        log("udp send ok.\n");
        is_appdata_prepare = false;
        return true;
    }
}

```

```

    }
    else
    {
        log("udp send fail.\n");

        /*附着网络*/
        net_attch();

        /*创建 UDP socket*/
        log("creat udp socket...\n");
        if((ret = neul_bc95_create_udpsocket(8091)) >= 0)
        {
            udpsocket = ret;
            log("ok.\n");
        }
        return false;
    }
}

```

#### ➤ *Neul\_bc95.c*

UDP 数据上报函数：

```

int neul_bc95_udp_send(int socket, const char *buf, int sendlen)
{
    char *cmd = "AT+NSOST=";
    int ret = 0;
    char *str = NULL;

    .....
    /*参数初始化*/
    .....
    /*参数判断*/

    /*将 hex 转为 string*/
    neul_bc95_hex_to_str((unsigned char *)buf, sendlen, neul_bc95_tmpbuf);

    /*AT 指令打包、发送*/
    sprintf(neul_dev.wbuf, "%s%d,%s,%d,%d,%s\r", cmd, socket,
        (neul_dev.addrinfo+socket)->ip,
        (neul_dev.addrinfo+socket)->port,
        sendlen,
        neul_bc95_tmpbuf);
    ret = neul_dev.ops->dev_write(neul_dev.wbuf, strlen(neul_dev.wbuf), 0);

    /* AT 指令发送失败*/
}

```

```

if (ret < 0)
{
    //write data to bc95 failed
    return -1;
}
memset(neul_dev.rbuf, 0, 32);

/*读取返回结果*/
ret = neul_dev.ops->dev_read(neul_dev.rbuf, 32, 0, 2000);

/*未收到返回数据*/
if (ret <= 0)
{
    //read bc95 read set return value info failed
    log("time out.\n");
    return -1;
}

/*返回数据错误*/
str = strstr(neul_dev.rbuf,"OK");
if (!str)
{
    log("error.\n");
    return -1;
}
return 0;
}

```

## 5.3. NB-IOT 节点下行通讯示例

NB-IOT 终端节点定时向指定服务器发送数据，服务器接收到节点数据后返回一个数据给节点。服务器需要提供公网 IP，通讯协议采用 UDP。同样采用串口工具调试结合 AT 指令调试。

### 5.3.1. AT 指令调试

#### 1. 查看模块能否通讯

```

/*发送*/
AT
/*接收*/
OK

```

## 2. 重启模块

```
/*发送*/  
AT+NRB  
/*接收*/  
REBOOTING  
?0  
REBOOT_CAUSE_APPLICATION_AT  
Neul  
OK
```

## 3. 查看 IMEI 号（如果返回 ERROR，说明模块没有设置 IMEI 需要手动设置）

```
/*发送*/  
AT+CGSN=1  
/*接收*/  
+CGSN:865352030093883  
OK
```

## 4. 设置 IMEI，如果已经设置则跳过此步骤

```
/*发送*/  
AT+NTSETID=1, 201612091450303  
/*接收*/  
OK
```

## 5. 设置自动连接网络

```
/*发送*/  
AT+NCONFIG=AUTOCONNECT,TRUE  
/*接收*/  
OK
```

## 6. 打开模块全功能模式

```
/*发送*/  
AT+CFUN=1  
/*接收*/  
OK
```

## 7. 查询 SIM 卡号（如果没有卡号返回，说明没检测到 SIM 卡）

```
/*发送*/  
AT+CIMI  
/*接收*/  
460111174764199  
OK
```

## 8. 查询网络附着状态（返回 1 代表附着成功，如果返回 0 则多试几次）

```
/*发送*/  
AT+CGATT?
```

```
/*接收*/
+CGATT:1
OK
```

9. 创建 UDP socket (DGRAM 是 socket 类型, 是 UDP 协议, 8091 是监听的端口)

```
/*发送*/
AT+NSOCR=DGRAM,17,8091,1
/*接收*/
0
OK
```

10. 编写服务器收发脚本

登录服务器端编写一个简单 UDP 收发 python 脚本 (仅供参考) 用于给节点发送 UDP 下行数据。

➤ *socketudp\_server.py*

UDP 收发 python 脚本代码

```
#!/usr/bin/env python

import socket
/*创建 socket*/
address = ('120.55.62.150', 50004)
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(address)

/*循环接收 UDP 数据*/
while True:
    data, addr = s.recvfrom(2048)
    if not data:
        print "client has exist"
        break
    print "received:", data, "from", addr
    s.sendto("1", addr)
    print "send", "1", "to", addr
```

```
s.close
```

通过 VIM 编辑器编辑代码，然后输入 wq（保存退出）。



```

1 #!/usr/bin/env python
2
3 import socket
4
5 address = ('101.37.27.74', 50004)
6 s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
7 s.bind(address)
8
9 while True:
10     data, addr = s.recvfrom(2048)
11     if not data:
12         print "client has exist"
13         break
14     print "received:", data, "from", addr
15     s.sendto("1",addr)
16     print "send", "1", "to", addr
17
18 s.close()

```

图 5-20 运行脚本

执行 python socketudp\_server.py 运行脚本。

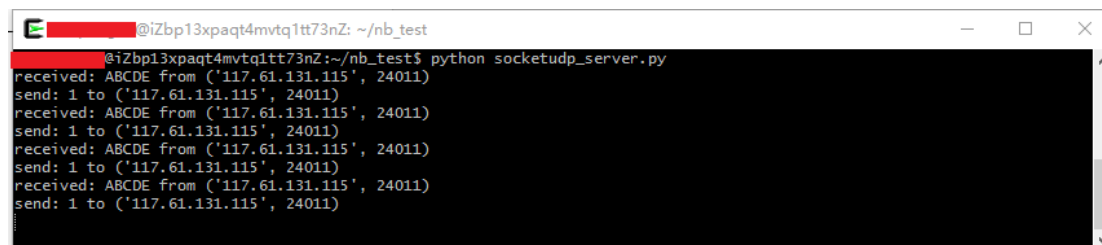
11. NB-IoT 节点端发送 UDP 数据（0 是 socket 号, 120.55.62.150 是服务器号, 50004 是端口号, 5 是数据长度, 4142434445 代表十六进制 0x41 0x42 0x43 0x44 0x45）

```

/*发送*/
AT+NSOST=0 120.55.62.150,50004,5,4142434445
/*接收*/
0,5 OK
/*等待一段时间，返回以下信息代码节点收到服务器返回的数据*/
+NSONMI:0,1

```

同时，服务器端会收到节点上行数据，并给节点发送下行数据：



```

received: ABCDE from ('117.61.131.115', 24011)
send: 1 to ('117.61.131.115', 24011)
received: ABCDE from ('117.61.131.115', 24011)
send: 1 to ('117.61.131.115', 24011)
received: ABCDE from ('117.61.131.115', 24011)
send: 1 to ('117.61.131.115', 24011)
received: ABCDE from ('117.61.131.115', 24011)
send: 1 to ('117.61.131.115', 24011)

```

图 5-21 脚本运行界面

12. 用串口工具读取 UDP 数据包

```

/*发送 0 代表 socket 号 1 代表读取数据长度*/
AT+NSORF=0,1
/*接收 120.55.62.150 代表 IP 50004 代表端口 1 代表接收长度 31 代表数据*/
0, 120.55.62.150,50004,1,31,0

```



OK

### 5.3.2. 应用代码介绍

应用代码的实现如下所示：

#### ➤ *Config.h*

配置节点参数：

```
/*转发服务器 ip,需要公网 ip*/
#define REMOTE_IPV4  "120.55.62.150"

/*转发服务器端口*/
#define REMOTE_PORT  50000

/*本地服务器地址*/
#define SERVER_ADDR  {0x00,0x00,0x00,0xff,0xff,0xff}

/*节点 ID，用来区分同一个转发服务器中的各个节点*/
#define NODE_ID      {0x00, 0x00, 0x00, 0x01}

/*修改数据上报周期*/
#define APP_TX_DUTYCYCLE 60000
```

#### ➤ *Main.c*

主程序工作流程：

```
int main(){
    /*初始化时钟 UART */
    BoardInitMcu();

    /*初始化看门狗，每 3s 喂狗一次 */
    iwdog_init();
    timer_create(&iwdg_refresh_timer,          IWDG_REFRESH_DUTYCYCLE,
OPT_TMR_PERIODIC, iwdg_refresh_event);
    timer_start(&iwdg_refresh_timer);

    /*创建一个周期为 APP_TX_DUTYCYCLE 的定时器，用来周期上报数据
*/
    timer_create(&appdata_send_timer,APP_TX_DUTYCYCLE,OPT_TMR_PER
IODIC, appdata_send_event);

    /*初始化 NB-IOT 模块，通过发送 AT 指令和模块通讯使其初始化*/
```

```

while((ret = neul_bc95_hw_init()) != 0)
{
    debug("init fail,reboot.\n");
}
debug("init success.\n");

/*附着网络*/
net_attch();

/*发送 AT+NSOCR=DGRAM,17,8091,1 创建 UDP SOCKET*/
debug("creat udp socket...\n");
if((ret = neul_bc95_create_udpsocket(8091)) >= 0)
{
    udpsocket = ret;
    debug("ok.\n");
}
while(1)
{
    .....
    /*发送 UDP 数据*/
    .....
    /*接收 UDP 数据*/

}
}

```

查询附着网络：

```

void net_attch()
{
    uint8_t try_cnt = 0;
    int ret = 0;

    while(1)
    {
        /*AT+NRB 重启*/
        neul_bc95_reboot();

        /*AT+CGATT 查询网络*/
        do{
            log("get net status:%d\n", try_cnt);
            ret = neul_bc95_get_netstat();
            if(!ret)
            {
                log("connected.\n");
                return ;
            }
        }while(1);
    }
}

```

```

    }
    neul_bc95_sleep(5000);
} while(ret < 0 && ++try_cnt < GET_NETSTAT_RETRY_MAX);
try_cnt = 0;
log("no network connect.reboot!\n");
}
}

```

接收 UDP 数据：

```

bool udp_downlink()
{
    uint8_t rec_cnt = 0;
    uint8_t i;
    int ret = 0;

    /*最多查询 5 次*/
    while(rec_cnt++ < 5)
    {
        /*发送 AT+NSORF 读取下行数据*/
        ret=neul_bc95_udp_read(udpsocket, (char *)rec_data, APP_DATA_SIZE,
0);

        log("downlink: %d\n", ret);
        if(ret > 0)
        {
            /*数据打印*/
            for(i = 0; i < ret; i++)
            {
                log("data[%i]:%x\n", i, rec_data[i]);
            }

            /*数据解析*/
            parse_downlink(rec_data, APP_DATA_SIZE);
            return true;
        }
        neul_bc95_sleep(5000);
    }
    return false;
}

```

#### ➤ *Neul\_bc95.c*

UDP 数据读取函数：

```

int neul_bc95_udp_read(int socket,char *buf, int maxlen, int mode)
{
    char *cmd = "AT+NSORF=";

```

```

int ret = 0;
char *str = NULL;
int rlen = 0;
int rskt = -1;
int port = 0;
int readleft = 0;
.....
/*参数判断*/

/*数据打包发送*/
sprintf(neul_dev.wbuf, "%s%d,%d\r", cmd, socket, maxlen);
ret = neul_dev.ops->dev_write(neul_dev.wbuf, strlen(neul_dev.wbuf), 0);
if (ret < 0)
{
    return -1;    //写入失败
}

/*读取返回值*/
memset(neul_dev.rbuf, 0, 128);
ret = neul_dev.ops->dev_read(neul_dev.rbuf, 128, 0, 2000);
if (ret <= 0)
{
    return -1;    //读取失败
}

/*判断返回是否正确*/
str = strstr(neul_dev.rbuf, "OK");
if (!str)
{
    return -1;
}

/*数据解析*/
ret = sscanf(neul_dev.rbuf, "%d,%[^,],%d,%d,%[^,],%d\r%s", &rskt,
                neul_bc95_tmpbuf,
                &port,
                &rlen,
                neul_bc95_tmpbuf+15,
                &readleft,
                neul_dev.wbuf);

if (rlen > 0)
{
    neul_bc95_str_to_hex(neul_bc95_tmpbuf+15, rlen*2, buf);
}

return rlen;

```

}

## 5.4. NB-IoT Web 客户端配置及使用示例

Web 客户端是基于 NB-IoT 本地服务器使用的，NB-IoT 本地服务器共享一个 AP，用终端设备接入实现对数据的访问。

接入 AP 账户：“**NB-IoT\_Gateway\_AP\_XXXXXX**”(XXXXXX 为 AP 编号，如：**008001**)，密码：“**1234567890**”。以下所有服务必须在终端接入该 AP 下，才可以使用！

### 5.4.1. Web 客户端简介

#### 1. NB-IoT 应用演示界面

主要完成 NB-IoT 节点数据的可视化展示和控制。网址：<http://test.com>。（**注意！网址必须使用 http！不可使用 https！**）。

界面的整体展示如下图所示。

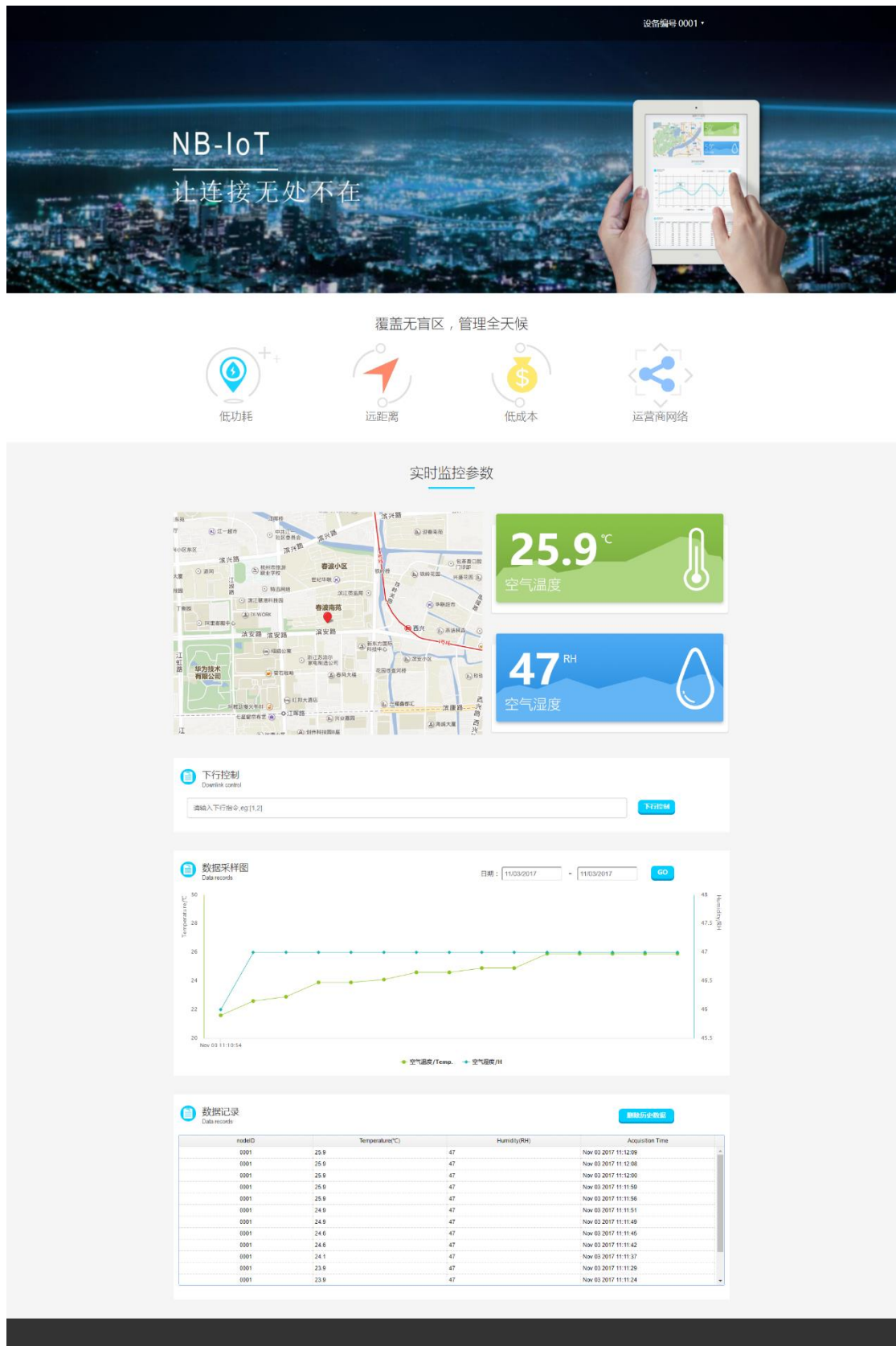


图 5-22 Web 客户端

该 Web 客户端，支持实时参数监控、数据采样图展示、数据记录查询、下行

控制。实时参数监控包括实时 NB-IoT 节点的位置信息展示 (GPS)、温度数据展示、湿度数据展示。下行控制节点提供控制节点参数的功能。数据采样图提供数据的基本分析功能，数据记录提供按时间区间的数据展示。

### 5.4.2. Web 客户端配置使用

1. 在浏览器（谷歌、火狐、ie 等）中输入网址 <http://test.com>，进入该网址访问。
2. 选择 NB-IoT 节点查看数据。

如图下图所示，点击设备编号旁的下拉框，选择要查看的节点编号。



图 5-23 选择节点

接着页面向下滑动至数据采样图，点击采样图边的“GO”按钮，页面数据即刷新为选择查看的节点数据。



图 5-24 数据采样图

### 3. 实时参数监控

页面滚动至实时监控参数，该部分左侧为地图信息，地图上红色的小坐标显示的是节点的位置，如图下图左侧（注意，如果节点没有 GPS 功能或 GPS 暂时出现异常，地图上则不会显示红色的小坐标），点击红色小坐标查看节点信息；右侧显示的是节点最新采集到的空气温度和空气湿度数据。



图 5-25 实时监控参数

### 4. 数据采样图展示和数据记录查询

页面滚动至数据采样图，采样图右上侧有两个选择时间的下拉框，第一个是“起始时间”，第二个是“截止时间”，选择好时间区间后，点击右侧的“GO”按钮，数据采样图和数据记录将会显示该区间内的数据。数据采样图横坐标为时间，曲线为空气温度和空气湿度的时间变化曲线；数据记录以表格形式展示选择区间内的历史数据，显示了节点 ID、温度、湿度、采集时间四样信息。如果需要清除所有数据，请点击“删除历史数据”。



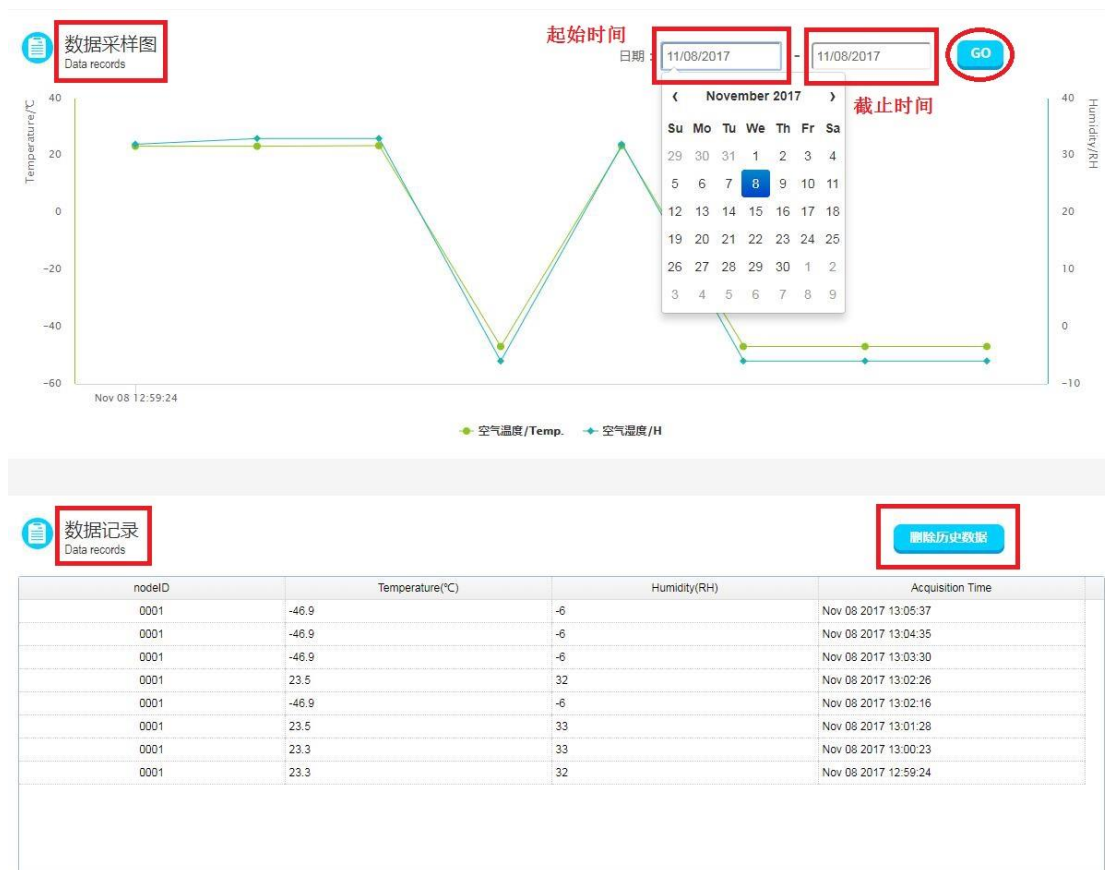


图 5-26 数据采样图和数据记录

## 5. 下行控制

下行控制的功能是在页面上实现对 NB-IoT 节点的控制。

控制前先选择节点，参考上述第 2 步，选择 NB-IoT 节点。选择好节点后，页面滚动至下行控制，如图下图所示，在左侧的输入框里输入控制指令（具体指令请参考 NB-IoT 下行控制指令说明），输入完成后，点击右侧的下行控制按钮，发下行数据。



图 5-27 下行控制窗口

如果页面弹出提示框，显示“发送成功”，如下图所示，则说明下行控制成功。



图 5-28 数据发送成功

如果弹出“请检查输入信息”或“发送失败”，说明下行控制失败，请检查输入指令是否有误。



图 5-29 数据发送失败

## 5.5. 基于 NB-IoT 技术的设备控制示例

NB-IOT 节点与 NB-IoT 本地服务器实现双向通信，节点可以向 Nb-IoT 本地服务器上报告数据，Nb-IoT 本地服务器也可以下行控制数据。

### 5.5.1. 节点配置使用

1. 节点初始配置同上所述，打开串口调试助手设置为 9600.8.1.N.N，重启节点并查看 LOG 信息检查节点是否正常工作,当收到 connect 代表节点网络附着成功，如果长时间不能入网需要检查基站网络覆盖情况。初始化 LOG 信息如下：

```
Printf init
AT...           //发送 AT
ok.
AT+NRB...      //发送 AT+NRB
ok.
```

```
AT...
ok.
AT+NCONFIG=AUTOCONNECT,TRUE...
ok.
AT...
ok.
AT+CGSN=1...
ok
AT+CFUN=1...
ok.
AT+CIMI...
ok.
AT+NBAND=5...
ok.
init success.
get net status:0      //发送 AT+CGATT?  返回 1 代表连接成功
fail
disconnect.
get net status:1
ok
connect.
```

UDP 上报的 LOG 信息如下：

```
creat udp socket...
ok.
udp send...
data[0]:0
data[1]:0
data[2]:0
data[3]:5
data[4]:0
data[5]:0
data[6]:0
data[7]:ff
data[8]:ff
data[9]:ff
data[10]:2a
data[11]:ef
data[12]:c4
data[13]:55
data[14]:7a
data[15]:5a
data[16]:ff
data[17]:f5
```

```
data[18]:6d
data[19]:38
data[20]:6e
data[21]:69
ok
```

2. 连接 NB-IoT 本地服务器 AP, 登录网页 <http://test.com> 查看节点上报数据, 并通过网页下发下行控制命令, 比如发送两个参数[12, 12]。

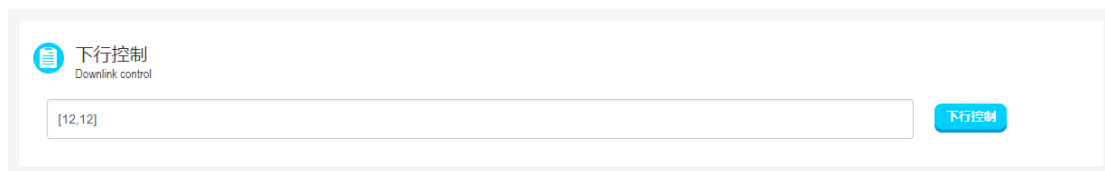


图 5-30 发送下行数据

点击发送后, 数据会随着下一次 NB-IoT 节点上报数据时, 发送给 NB-IoT 节点。

下行数据格式如下:

**[xx,xx,...,xx]:**中括号里包含十进制数据, 每个数据以逗号各开。

每个十进制数据的范围: **0~255**

下行数据长度: **≤512**

如没有按上述格式发送下行数据, 会返回“请检查输入信息”或“发送失败”的提示, 表示发送错误。

3. 观察节点是否收到下行数据

在节点上报数据后, 节点会切换到接收模式, 持续 30 秒, 由于数据下行会有延时, 所以节点端在一段时间内实时查询是否有下行数据。当接收到下行数据时, 串口会打印数据。

下行数据 log, 打印 downlink 代表节点查询一次下行数据, 后面的数值为查到的字节数, 如 **downlink: 3**, 代表查到 3 个字节。

```
downlink: 0      //没有接收到数据
downlink: 3      //收到下行数据, 长度为 3

/*节点端接收到的下行数据*/
data[0]: 1
data[1]: c
data[2]: c
downlink: 0
```

节点端接收到的下行数据格式:

长度：≤512byte，网页中可以填小于 512byte 的下行数据

字节 0：转发服务器返回的 ACK，1：代表节点发送的数据格式正确，0：代表节点发送的数据格式错误或上行数据失败。

字节 1-512：下行数据

### 5.5.2. 应用代码介绍

应用代码实现如下：

#### ➤ *Config.h*

配置节点参数：

```
/*服务器 ip,需要公网 ip*/
#define REMOTE_IPV4 "120.55.62.150"

/*转发服务器端口*/
#define REMOTE_PORT 50003

/*本地服务器地址*/
#define SERVER_ADDR {0x00,0x00,0x00,0xff,0xff,0xff}

/*节点 ID，用来区分同一个转发服务器中的各个节点*/
#define NODE_ID {0x00, 0x00, 0x00, 0x05}

/*节点数据上报周期*/
#define APP_TX_DUTYCYCLE 60000
```

#### ➤ *Main.c*

主程序工作流程：

```
int main(){
    /*初始化时钟 UART */
    BoardInitMcu();

    /*初始化看门狗，每 3s 喂狗一次 */
    iwdog_init();
    timer_create(&iwdg_refresh_timer, IWDG_REFRESH_DUTYCYCLE,
OPT_TMR_PERIODIC, iwdg_refresh_event);
    timer_start(&iwdg_refresh_timer);

    /*创建一个周期为 APP_TX_DUTYCYCLE 的定时器，用来周期上报数据
    */
    timer_create(&appdata_send_timer,APP_TX_DUTYCYCLE,OPT_TMR_PER
```

```

IODIC, appdata_send_event);

/*初始化 NB-IOT 模块，通过发送 AT 指令和模块通讯使其初始化*/
while((ret = neul_bc95_hw_init()) != 0)
{
    debug("init fail,reboot.\n");
}
debug("init success.\n");

/*附着网络*/
net_attach();

/*发送 AT+NSOCR=DGRAM,17,8091,1 创建 UDP SOCKET*/
debug("creat udp socket...\n");
if((ret = neul_bc95_create_udpsocket(8091)) >= 0)
{
    udpsocket = ret;
    debug("ok.\n");
}
while(1)
{
    .....
    /*数据打包*/
    .....
    /*发送 UDP 数据*/
    .....
    /*接收 UDP 数据*/
    .....
    /*打印 UDP 下行数据*/
}
}

```

数据接收：

```

bool udp_downlink()
{
    uint8_t rec_cnt = 0;
    uint8_t i;
    int ret = 0;

    /*最多查询 5 次*/
    while(rec_cnt++ < 5)
    {
        /*发送 AT+NSORF 读取下行数据*/
        ret=neul_bc95_udp_read(udpsocket, (char *)rec_data, APP_DATA_SIZE,

```

```

0);
    log("downlink: %d\n", ret);
    if(ret > 0)
    {
        /*数据打印*/
        for(i = 0; i < ret; i++)
        {
            log("data[%i]:%x\n", i, rec_data[i]);
        }

        /*数据解析*/
        parse_downlink(rec_data, APP_DATA_SIZE);
        return true;
    }
    neul_bc95_sleep(5000);
}
return false;
}

```

数据打包:

```

static void prepare_data()
{
    int32_t latitude, longitude = 0;
    int16_t altitudeGps = 0xFFFF;
    uint16_t humi = 0, temp = 0;
    uint8_t node_id[NODE_ID_SIZE] = NODE_ID;
    GpsGetLatestGpsPositionBinary(&latitude, &longitude);
    altitudeGps = GpsGetLatestGpsAltitude();
    SHT2xGetTempHumi(&temp, &humi);

    /*数据打包*/
    app_data[0] = node_id[0];
    app_data[1] = node_id[1];
    app_data[2] = node_id[2];
    app_data[3] = node_id[3];
    memcpy(&app_data[4], mac, 6);
    app_data[10] = (uint8_t)((latitude >> 16) & 0xFF);
    app_data[11] = (uint8_t)((latitude >> 8) & 0xFF);
    app_data[12] = (uint8_t)(latitude & 0xFF);
    app_data[13] = (uint8_t)((longitude >> 16) & 0xFF);
    app_data[14] = (uint8_t)((longitude >> 8) & 0xFF);
    app_data[15] = (uint8_t)(longitude & 0xFF);
    app_data[16] = (uint8_t)((altitudeGps >> 8) & 0xFF);
    app_data[17] = (uint8_t)(altitudeGps & 0xFF);
}

```

```

app_data[18] = (uint8_t)((temp >> 8) & 0xFF);
app_data[19] = (uint8_t)(temp & 0xFF);
app_data[20] = (uint8_t)((humi >> 8) & 0xFF);
app_data[21] = (uint8_t)(humi & 0xFF);
clear_buf();
}

```

### ➤ *Neul\_bc95.c*

UDP 数据读取函数：

```

int neul_bc95_udp_read(int socket, char *buf, int maxlen, int mode)
{
    char *cmd = "AT+NSORF=";
    int ret = 0;
    char *str = NULL;
    int rlen = 0;
    int rskt = -1;
    int port = 0;
    int readleft = 0;

    .....
    /*参数判断*/

    /*数据打包发送*/
    sprintf(neul_dev.wbuf, "%s%d,%d\r", cmd, socket, maxlen);
    ret = neul_dev.ops->dev_write(neul_dev.wbuf, strlen(neul_dev.wbuf), 0);
    if (ret < 0)
    {
        return -1;    //写入失败
    }
    /*读取返回值*/
    memset(neul_dev.rbuf, 0, 128);
    ret = neul_dev.ops->dev_read(neul_dev.rbuf, 128, 0, 2000);
    if (ret <= 0)
    {
        return -1;    //读取失败
    }
    /*判断返回是否正确*/
    str = strstr(neul_dev.rbuf, "OK");
    if (!str)
    {
        return -1;
    }
    /*数据解析*/
    ret = sscanf(neul_dev.rbuf, "%d,%[^,],%d,%d,%[^,],%d\r%s", &rskt,

```



```

                                neul_bc95_tmpbuf,
                                &port,
                                &rlen,
                                neul_bc95_tmpbuf+15,
                                &readleft,
                                neul_dev.wbuf);

    if (rlen > 0)
    {
        neul_bc95_str_to_hex(neul_bc95_tmpbuf+15, rlen*2, buf);
    }
    return rlen;
}

```

## 5.6. NB-IoT 温湿度监控及定位应用示例

NB-IOT 终端节点采集温湿度、GPS 信息定时向指定服务器发送数据，服务器将数据转发到本地服务器，最终可以通过登录本地服务器的 Web 客户端来查看温湿度、GPS 数据。

### 5.6.1. 节点配置使用

#### 1. 配置 config.h 文件

```

/*转发服务器 ip,需要公网 ip*/
#define REMOTE_IPV4  "120.55.62.150"

/*转发服务器端口*/
#define REMOTE_PORT  50003

/*用户终端服务器地址*/
#define SERVER_ADDR  {0x00,0x00,0x00,0xff,0xff,0xff}

/*节点 ID，用来区分同一个转发服务器中的各个节点*/
#define NODE_ID      {0x00, 0x00, 0x00, 0x05}

/*修改数据上报周期*/
#define APP_TX_DUTYCYCLE 60000

```

2. 烧写程序到终端节点或者开发板，并接上 GPS 天线，上电运行 NB-IoT 节点。
3. 打开串口调试助手设置为 9600.8.1.N.N，重启节点并查看 LOG 信息检查节点是否正常工作。初始化及 UDP 数据的 LOG 信息与上节类似。

### 上报数据的格式及计算公式如下：

长度: 22 字节

字节 0-3:           //节点 ID  
字节 4-9:           //目标服务器地址  
字节 10-12:         //纬度  
字节 13-15:         //经度  
字节 15-17:         //高度  
字节 18-19:         //温度  
字节 20-21:         //湿度

/\*经度计算公式\*/

```
Latitude = ((data[10] << 16) | (data[11] << 8) | (data[12])); if( Latitude >= 0 )
    Latitude = (Latitude * 90) / 8388607;
else
    Latitude = (Latitude * 90) / 8388608;
```

/\*纬度计算公式\*/

```
Longitude = ((data[13] << 16) | (data[14] << 8) | (data[15]));
if( Longitude >= 0 )
    Longitude = (Longitude * 180) / 8388607;
else
    Longitude = (Longitude * 180) / 8388608;
```

/\*高度计算公式\*/

```
Altitude = ((data[16] << 8) | (data[17]));
```

/\*温度计算公式\*/

```
Temperature = ((data[8] << 8) | (data[9]));
Temperature = (Temperature * 175.72) / 65536 - 46.85;
```

/\*湿度计算公式\*/

```
double Humidity = ((AppData[10] << 8) | (AppData[11]));
Humidity = (Humidity * 125) / 65536 - 6;
```

4. 连接 NB-IoT 本地服务器，并登录网页 <http://test.com> 并查看节点数据。



图 5-31 NB-IoT 节点数据显示

### 5.6.2. 应用代码介绍

应用代码实现如下：

#### ➤ *Config.h*

配置节点参数：

```
/*转发服务器 ip,需要公网 ip*/
#define REMOTE_IPV4 "120.55.62.150"

/*转发服务器端口*/
#define REMOTE_PORT 50000

/*本地服务器地址*/
#define SERVER_ADDR {0x00,0x00,0x00,0xff,0xff,0xff}

/*节点 ID，用来区分同一个转发服务器中的各个节点*/
#define NODE_ID {0x00, 0x00, 0x00, 0x01}

/*修改数据上报周期*/
#define APP_TX_DUTYCYCLE 60000
```

#### ➤ *Main.c*

主程序工作流程：

```
int main(){
    /*初始化时钟 UART */
    BoardInitMcu();
```

```

/*初始化看门狗，每 3s 喂狗一次 */
iwdog_init();
timer_create(&iwdg_refresh_timer,IWDG_REFRESH_DUTYCYCLE,
OPT_TMR_PERIODIC, iwdg_refresh_event);
timer_start(&iwdg_refresh_timer);

/*创建一个周期为 APP_TX_DUTYCYCLE 的定时器，用来周期上报数据
*/
timer_create(&appdata_send_timer,APP_TX_DUTYCYCLE,OPT_TMR_PERIODIC, appdata_send_event);

/*初始化 NB-IOT 模块，通过发送 AT 指令和模块通讯使其初始化*/
while((ret = neul_bc95_hw_init()) != 0)
{
    debug("init fail,reboot.\n");
}
debug("init success.\n");

/*附着网络*/
net_attch();

/*发送 AT+NSOCR=DGRAM,17,8091,1 创建 UDP SOCKET*/
debug("creat udp socket...\n");
if((ret = neul_bc95_create_udpsocket(8091)) >= 0)
{
    udpsocket = ret;
    debug("ok.\n");
}
while(1)
{
    .....
    /*数据解析*/
    .....
    /*发送 UDP 数据*/
}
}

```

## 6. NB-IoT 本地服务器 API 应用参考

本实训平台的 NB-IoT 网络架构为 NB-IoT 节点、云端转发服务器、NB-IoT 本地服务器、Web 客户端。该实训平台实现了 Nb-IoT 网络的去云端化，即云端转发服务器把 NB-IoT 节点的数据转发到本地服务器，在本地网络即可完成 Web 客户端的应用。除此之外，NB-IoT 本地服务器对外提供基于 MQTT 的 API 接口，通过 API 的调用可以实现定制化的功能。

**该章 API 应用参考示例，是在 Linux 系统的基础上，用 NodeJS 实现，仅供参考！**

### 6.1. MQTT 协议应用参考示例

#### 6.1.1. MQTT 协议介绍

##### 6.1.1.1. MQTT 协议简介

MQTT 是一个基于客户端-服务器的消息发布/订阅传输协议。MQTT 协议是轻量、简单、开放和易于实现的，这些特点使它适用范围非常广泛。在很多情况下，包括受限的环境中，如：机器与机器（M2M）通信和物联网（IoT）。其在，通过卫星链路通信传感器、偶尔拨号的医疗设备、智能家居、及一些小型化设备中已广泛使用。

MQTT 协议当前版本为，2014 年发布的 MQTT v3.1.1。除标准版外，还有一个简化版 MQTT-SN，该协议主要针对嵌入式设备，这些设备一般工作于 TCP/IP 网络。

MQTT 协议运行在 TCP/IP 或其他网络协议，提供有序、无损、双向连接。其特点包括：

1. 使用的发布/订阅消息模式，它提供了一对多消息分发，以实现与应用程序的解耦。
2. 对负载内容屏蔽的消息传输机制。
3. 对传输消息有三种服务质量（QoS）：

- 至多一次，这一级别会发生消息丢失或重复，消息发布依赖于底层 TCP/IP 网络。即： $\leq 1$ 。
  - 至少一次，这一级别会确保消息到达，但消息可能会重复。即： $\geq 1$ 。
  - 只有一次，确保消息只有一次到达。即： $= 1$ 。在一些要求比较严格的计费系统中，可以使用此级别。
4. 数据传输和协议交换的最小化（协议头部只有 2 字节），以减少网络流量
  5. 通知机制，异常中断时通知传输双方。

### 6.1.1.2. MQTT 协议原理

#### 1. MQTT 协议实现方式

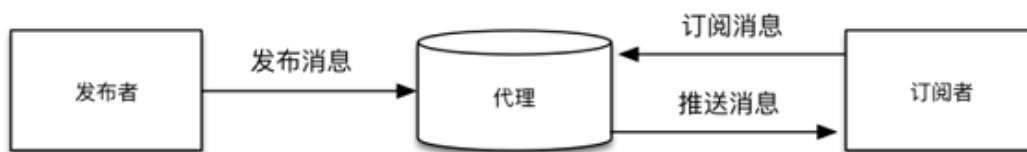


图 6-1 MQTT 协议实现方式

- 1) 实现 MQTT 协议需要：客户端和服务端。
- 2) MQTT 协议中有三种身份：发布者（Publish）、代理（Broker）（服务器）、订阅者（Subscribe）。其中，消息的发布者和订阅者都是客户端，消息代理是服务器，消息发布者可以同时是订阅者。
- 3) MQTT 传输的消息分为：主题（Topic）和负载（payload）两部分：
  - Topic，可以理解为消息的类型，订阅者订阅（Subscribe）后，就会收到该主题的消息内容（payload）。
  - payload，可以理解为消息的内容，是指订阅者具体要使用的内容。

#### 2. 网络传输与应用消息

MQTT 会构建底层网络传输：它将建立客户端到服务器的连接，提供两者之间的一个有序的、无损的、基于字节流的双向传输。

当应用数据通过 MQTT 网络发送时，MQTT 会把与之相关的服务质量（QoS）和主题名（Topic）相关联。

### 3. MQTT 客户端

一个使用 MQTT 协议的应用程序或者设备，它总是建立到服务器的网络连接。客户端可以：

- 发布其他客户端可能会订阅的信息
- 订阅其它客户端发布的消息
- 退订或删除应用程序的消息
- 断开与服务器连接

### 4. MQTT 服务器

MQTT 服务器以称为“消息代理”（Broker），可以是一个应用程序或一台设备。它是位于消息发布者和订阅者之间，它可以：

- 接受来自客户的网络连接
- 接受客户发布的应用信息
- 处理来自客户端的订阅和退订请求
- 向订阅的客户转发应用程序消息

### 5. MQTT 协议中的订阅、主题、会话

#### ● 订阅（Subscription）

订阅包含主题筛选器（Topic Filter）和最大服务质量（QoS）。订阅会与一个会话（Session）关联。一个会话可以包含多个订阅。每一个会话中的每个订阅都有一个不同的主题筛选器。

#### ● 会话（Session）

每个客户端与服务器建立连接后就是一个会话，客户端和服务器之间有状态交互。会话存在于一个网络之间，也可能在客户端和服务器之间跨越多个连续的网络连接。

#### ● 主题名（Topic Name）

连接到一个应用程序消息的标签，该标签与服务器的订阅相匹配。服务器会将消息发送给订阅所匹配标签的每个客户端。

- 主题筛选器（Topic Filter）

一个对主题名通配符筛选器，在订阅表达式中使用，表示订阅所匹配到的多个主题。

- 负载（Payload）

消息订阅者所具体接收的内容。

## 6. MQTT 协议中的方法

MQTT 协议中定义了一些方法（也被称为动作），来于表示对确定资源所进行的操作。这个资源可以代表预先存在的数据或动态生成数据，这取决于服务器的实现。通常来说，资源指服务器上的文件或输出。

Connect，等待与服务器建立连接。

Disconnect，等待 MQTT 客户端完成所做的工作，并与服务器断开 TCP/IP 会话。

Subscribe，等待完成订阅。

UnSubscribe，等待服务器取消客户端的一个或多个 topics 订阅。

Publish，MQTT 客户端发送消息请求，发送完成后返回应用程序线程。

### 6.1.2.MQTT 协议操作

#### 6.1.2.1. MQTT 安装

NodeJS 环境下，使用 npm 安装 MQTT 模块：

```
npm install mqtt --save
```



### 6.1.2.2. MQTT 控制

#### 1. CONNECT – 连接服务端

客户端到服务端的网络连接建立后，客户端发送给服务端的第一个报文必须是 CONNECT 报文，在一个网络连接上，客户端只能发送一次 CONNECT 报文。服务端必须将客户端发送的第二个 CONNECT 报文当作协议违规处理并断开客户端的连接。

```
/*调用 MQTT 模块*/
var mqtt = require('mqtt')
/*连接服务器*/
/*url:mqtt broker 地址,options:连接信息*/
var client = mqtt.connect([url], options)
/*连接服务器实例*/
client = mqtt.connect('mqtt:// github.com:1883',{
    /*用户名*/
    username: mqtt_username,
    /*密码*/
    password: mqtt_password,
    /*客户端 Id*/
    clientId: " + clientId,
    /*客户端断开连接，代理发送的信息*/
    will:{...}
})
```

#### 2. PUBLISH – 发布消息

PUBLISH 从客户端向服务端或者服务端向客户端传输一个应用消息。

```
/*连接服务器*/
var client = mqtt.connect('mqtt:// github.com:1883')
/*发布消息*/
/*topic:发布的主题， string 类型;
```

```

message:发布的消息， string 或 buffer 类型;
options:发布的选项， 包括 qos, return,dup
callback:qos 处理完成时触发*/
client.publish(topic, message, [options], [callback])

```

### 3. SUBSCRIBE - 订阅主题

客户端向服务端发送 SUBSCRIBE 用于创建一个或多个订阅。每个订阅注册客户端关心的一个或多个主题。为了将应用消息转发给与那些订阅匹配的主题，服务端发送 PUBLISH 报文给客户端。

```

/*连接服务器*/
var client  = mqtt.connect('mqtt:// github.com:1883')
/*订阅主题*/
/*topic: 要订阅的主题;
options:订阅的选项， 包括 qos,qos:订阅级别， 默认 0
callback:回调*/
client.subscribe(topic, [options], [callback])

```

### 4. UNSUBSCRIBE –取消订阅

客户端发送 UNSUBSCRIBE 给服务端，用于取消订阅主题。

```

/*连接服务器*/
var client  = mqtt.connect('mqtt:// github.com:1883')
/*取消订阅主题*/
/*topic: 要取消订阅的主题;
callback:回调*/
client.unsubscribe(topic/topic array, [callback])

```

### 5. DISCONNECT –断开连接

客户端与服务端断开连接

```

/*连接服务器*/
var client  = mqtt.connect('mqtt:// github.com:1883')
/*断开连接*/
client.disconnect()

```

### 6.1.2.3. MQTT Client 事件

#### 1. connect 事件

发送成功（重新）连接：

```
/*连接服务器*/  
var client = mqtt.connect('mqtt:// github.com:1883')  
/*发送成功连接*/  
/* connect_callback :成功（重新）连接触发该函数*/  
client.on('connect', connect_callback);
```

#### 2. reconnectt 事件

发送重新连接：

```
/*连接服务器*/  
var client = mqtt.connect('mqtt:// github.com:1883')  
/*发送重新连接*/  
/* reconnect_callback :重新连接触发该函数*/  
client.on('reconnect', reconnect_callback);
```

#### 3. close 事件

断开后发出：

```
/*连接服务器*/  
var client = mqtt.connect('mqtt:// github.com:1883')  
/*发送断开连接*/  
/* close :断开连接触发该函数*/  
client.on('close', close);
```

#### 4. offline 事件

客户端离线时发出：

```
/*连接服务器*/  
var client = mqtt.connect('mqtt:// github.com:1883')  
/*客户端离线*/  
/* offline :成功（重新）连接触发该函数*/
```

```
client.on('offline', offline);
```

## 5. error 事件

客户端无法连接或发生解析错误时发出：

```
/*连接服务器*/  
var client = mqtt.connect('mqtt:// github.com:1883')  
/*客户端无法连接或发生解析错误*/  
client.on('error', function(err) {});
```

## 6. message 事件

客户端收到发布数据包时发出：

```
/*连接服务器*/  
var client = mqtt.connect('mqtt:// github.com:1883')  
/*客户端收到发布数据包*/  
/*topic:收到的主题;  
message:收到的数据包;  
packet :收到的报文*/  
client.on('message', function (topic, message, packet) {});
```

## 7. packetsend 事件

客户端发送任何数据包时发出：

```
/*连接服务器*/  
var client = mqtt.connect('mqtt:// github.com:1883')  
/*客户端发送任何数据包*/  
/*packet :发送的报文*/  
client.on('packetsend', function (packet) {});
```

## 8. packetreceive 事件

客户端发送任何数据包时发出：

```
/*连接服务器*/  
var client = mqtt.connect('mqtt:// github.com:1883')  
/*客户端收到任何数据包*/  
/*packet :收到的报文*/
```

```
client.on('packetreceive', function (packet) {});
```

#### 6.1.2.4. MQTT 操作实例

操作需先在本地配置 nodejs 环境。

客户端 A 的代码如下,该客户端订阅/data/app 主题的消息:

```
var mqtt=require('mqtt');  
var client  = mqtt.connect('mqtt:// github.com:1883');  
/*订阅/data/app 主题*/  
client.subscribe('/data/app');  
client.on('message', function (topic, message, packet) {  
    console.log(topic+message)  
});
```

运行客户端 A 订阅主题/data/app, 如下图:



图 6-2 客户端 A

客户端 B 的代码如下, 该客户端发送主题为/data/app 的消息:

```
var mqtt=require('mqtt');  
var client  = mqtt.connect('mqtt:// github.com:1883');  
/*发送主题为/data/app 的消息*/  
client.publish('/data/app', 'hello world!');
```

运行客户端 B, 发送主题为/data/app 的消息, 如下图:



图 6-3 客户端 B

客户端 A 收到客户端 B 发送的主题和消息，打印出“/data/app hello word!”，如下图：

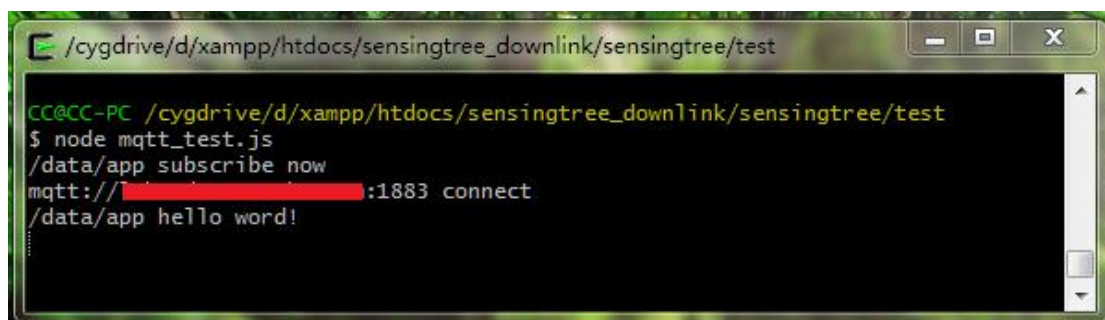


图 6-4 客户端 A 收到消息

## 6.2. 本地数据存储应用参考示例

### 6.2.1. 工程配置简介

工程中，文件夹、文件架构如下：

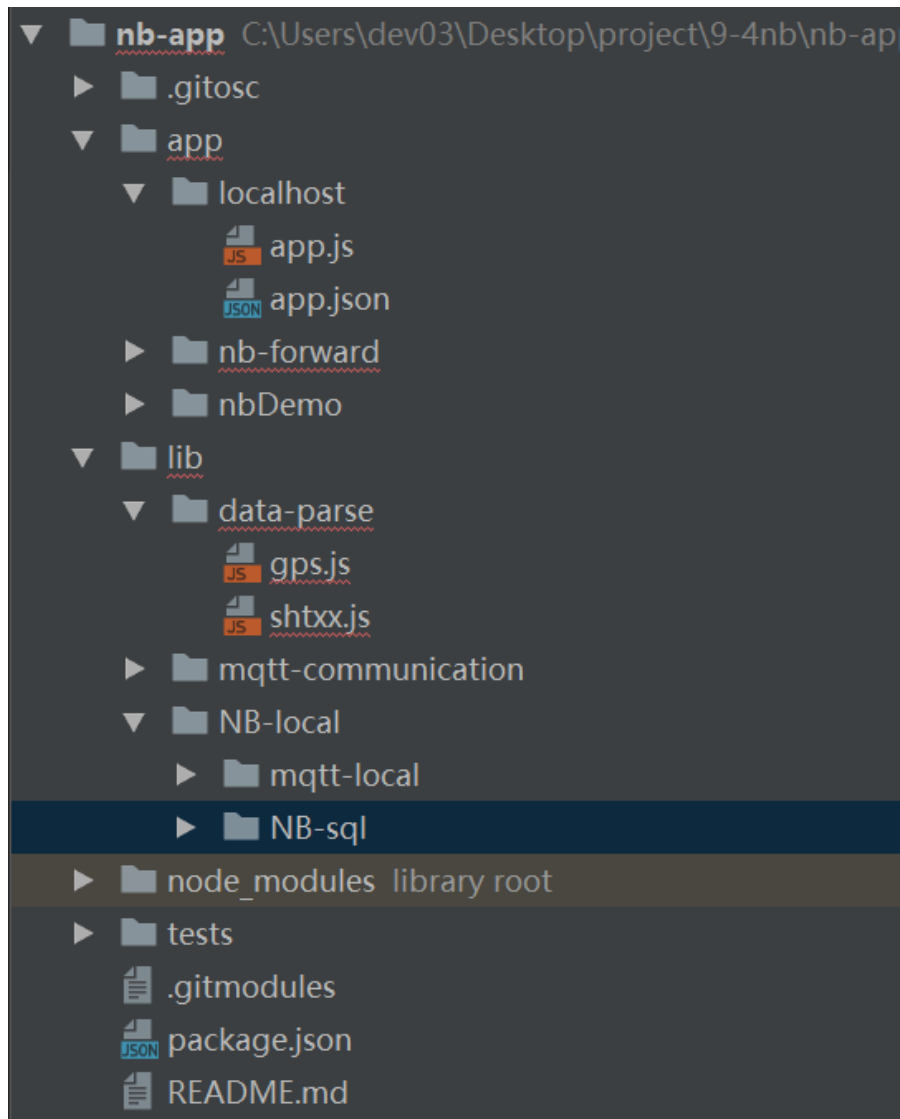


图 6-5 工程文件目录

目录结构：

- Nb-app:工程文件夹
- app\localhost\:本地数据储存文件
- app\localhost\app.js:数据储存功能实现主文件
- app\localhost\app.json:数据储存参数配置文件
- lib\data-parse\:数据解析文件
- lib\mqtt-communication\:mqtt 相关库文件
- lib\NB-local \mqtt-local:mqtt 相关库文件
- lib\NB-local \NB-sql:数据库储存相关文件

### 6.2.2. 数据获取和存储

获取 NB-IoT 节点传送过来的数据主要通过的 mqtt 协议。服务器获取到 NB-IoT 节点发过来的相关数据，然后通过 mqtt 中的 data channel 向本地数据库存储软件发送。本地数据库存储软件只需通过 mqtt 对 data channel 进行相关的订阅即可获取所需 NB-IoT 节点的信息。紧接着通过相应的数据解析之后存储到本地的数据库。

**基于 MQTT 实现的 API 定义：**

**mqtt connect url mqtt://test.com**

**topic /data/nbiot/macurl    macurl 为变量，根据实际情况定义**

**message**

```
JSON.parse(message)
```

```
{
```

**nodeid:节点编号**

**data:数据**

```
}
```

**data 转为 buffer 类型，前三个字节为 latitude,第 4 个到第 6 个字节为 longitude，第七个第八个字节为 altitude，第九第十字节为 temperature，第十一第十二字节为 humidity，如下图所示：**

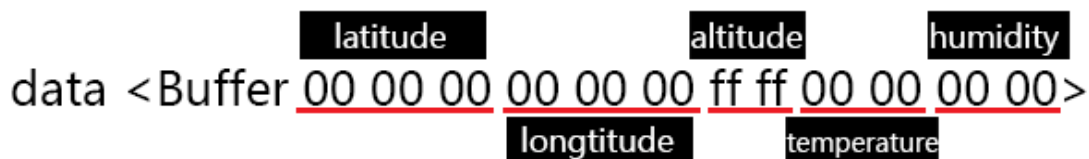


图 6-6 数据格式

#### ➤ 配置 mqtt

```
online_channelDefine=require(__dirname+'../../lib/mqtt-communication/channelDefine')
NB_online_mqtt = new online_channelDefine('nb_online' + Math.random(), 'nbiot')
NB_online_mqtt.connect();
NB_online_mqtt.subscribe(NB_online_mqtt.dataChan() + '/' + macAddress);
/*订阅/data/nbiot/macAddress,其中 macAddress 是个变量，每个 nb 节点对应一个特定的 mac 地址 */
```



➤ 使用 *mqtt* 接受数据

```
/*接受数据的入口函数 */
```

```
NB_online_mqtt.message(Data_parse);
```

```
Data_parse = function(topic, message) {
    var data_parse;
    console.log(topic, message.toString());
    if(topic.match(/.*data.*nbiot/)){
        var use_message = JSON.parse(message);
        var nodeid = use_message.nodeid;
        var app_data = new Buffer(use_message.message.data,'base64');
        console.log('Application data', app_data);
        /*Parse gps, 主要用来解析经纬度以及海拔*/
        /*具体的 gps()解析会在下文中的函数解析中讲到*/
        data_parse = new gps();
        var latitude = ((app_data[0] << 16) | (app_data[1] << 8) | (app_data[2]));
        var longitude = ((app_data[3] << 16) | (app_data[4] << 8) | (app_data[5]));
        var altitude = ((app_data[6] << 8) | (app_data[7]));
        latitude = data_parse.latitude(latitude);
        longitude = data_parse.longitude(longitude);
        altitude = data_parse.altitude(altitude);
        /*Parse shtxx, 主要用来解析温度和湿度*/
        /*具体的 shtxx ()解析会在下文中的函数解析中讲到*/
        data_parse = new shtxx();
        var temperature = ((app_data[8] << 8) | (app_data[9]));
        temperature = data_parse.temperature(temperature);
        var humidity = ((app_data[10] << 8) | (app_data[11]));
        humidity = data_parse.humidity(humidity);

        latestReadings = {
            'nodeId':nodeid,
            'latitude':latitude,
            'longitude':longitude,
            'altitude':altitude,
            'temperature':temperature,
            'humidity':humidity
        }
        /* for debug output */
        console.log('app data', latestReadings);
        /* CloudSave(), 用于将经纬度, 海拔, 温度, 湿度这些数据存储在 appData 表中 */
        CloudSave();
        /* nodeid_save (), 用于将新加的 nodeid 存储在 thirdparty 表中 */
        nodeid_save(nodeid);
```

```
}
}
```

### ➤ 函数解析

#### gps()函数

```
'use strict'
class gps {

  constructor(){
  }
  altitude(raw_data) {
    return raw_data;
  }
  latitude(raw_data) {
    if(raw_data >= 0)
      raw_data = (raw_data * 90) / 8388607;
    else
      raw_data = (raw_data * 90) / 8388608;
    return raw_data;
  }
  longitude(raw_data) {
    if( raw_data >= 0 )
      raw_data = (raw_data * 180) / 8388607;
    else
      raw_data = (raw_data * 180) / 8388608;
    return raw_data;
  }
}
module.exports = gps;
```

#### shtxx()函数

```
'use strict'
class shtxx {
  constructor(){
  }
  temperature(raw_data) {
    /* float accuracy is 0.1 */
    raw_data = parseFloat((((raw_data * 175.72) / 65536 - 46.85).toFixed(1)));
    return raw_data;
  }
  humidity(raw_data) {
    raw_data = (raw_data * 125) / 65536 - 6;
    /* float accuracy is 1 */
    raw_data = parseInt((Math.round(raw_data * 100)/100));
    return raw_data;
  }
}
```

```

    }
}

module.exports = shtxx;

```

### CloudSave()

```

var retry = 0;

CloudSave = function() {

    var sd = new AppData();
    /* 遍历 latestReadings 之后通过 sd.set()往数据表中添加相关的字段和数据，k
    为字段名，latestReading[k]为数据值*/
    _forEach(Object.keys(latestReadings),function (k) {
        console.log(k,latestReadings[k])
        sd.set(k,latestReadings[k])
    });
    /* 通过 sd.save()进行数据储存*/
    sd.save().then(function (sd) {
        console.log('New sensingData created',sd.id);
        retry = 0;
    },function (err) {
        console.error('new AppData create failed ',err);
        /* 假设储存失败，则会进行再次储存，尝试三次之后不再尝试*/
        if(3 > retry) {
            console.log('Re-save AppData at %d times', ++retry);
            setTimeout(CloudSave, 1000);
        }else
            retry = 0;
    })
}

```

### nodeid\_save()

```

var nodeid_save = function(nodeid_add) {
    var sd = new ThirdParty();
    /* sd.equalTo()比较数据表中的字段中是否有符合条件的值*/
    sd.equalTo('password', 'test');
    sd.equalTo('username', 'test');
    /* sd.first()查询结束后返回结果数据中的第一条数据*/
    sd.first().then(function(td) {
        if(typeof td != 'undefined'){
            if(td.nodeids != null)
                var nodeids = td.nodeids;
            else
                var nodeids = [];
        }
    })
}

```

```

        if(nodeids.indexOf(nodeid_add) == -1){
            nodeids.push(nodeid_add);
/* sd.update()更新数据库中的相关数据*/
            sd.update("nodeids",nodeids).then(function (sd) {
                console.log('New nodeids created');
                retry = 0;
            },function (err) {
                console.error('new nodeid create failed ',err);
                if(3 > retry) {
                    console.log('Re-save nodeid at %d times', ++retry);
                    setTimeout(CloudSave, 1000);
                }else
                    retry = 0;
            })
        }
    }
})
}

```

#### ➤ 数据库连接

上文的 CloudSave()和 nodeid\_save()就是主要进行数据库储存和更新的操作，两者分别使用了 appData 和 thirdparty 两张表，使用的 PostgreSQL。

PostgreSQL 是一个自由的对象-关系数据库服务器(数据库管理系统)。PostgreSQL 采用的是比较经典的 C/S (client/server) 结构，也就是一个客户端对应一个服务器端守护进程的模式，这个守护进程分析客户端来的查询请求，生成规划树，进行数据检索并最终把结果格式化输出后返回给客户端。

本 case 主要通过调用 pg 模块来进行对 PostgreSQL 数据库的连接和使用。大致逻辑组成如下,详情可以查看 lib\NB-local \NB-sql 目录下的 base.js:

```

var pg = require('pg');
/* 数据库连接的相关配置信息*/
var pool = new pg.Pool({
    user:'postgres',
    host:'localhost',
    database:'nb_vender',
    password:'postgres'
});

pool.connect(function(){
/* 连接成功之后进行的相关操作*/
})

```

#### ➤ 直接登录数据库进行相关的操作

首先通过 `sudo -u postgres psql` 登录

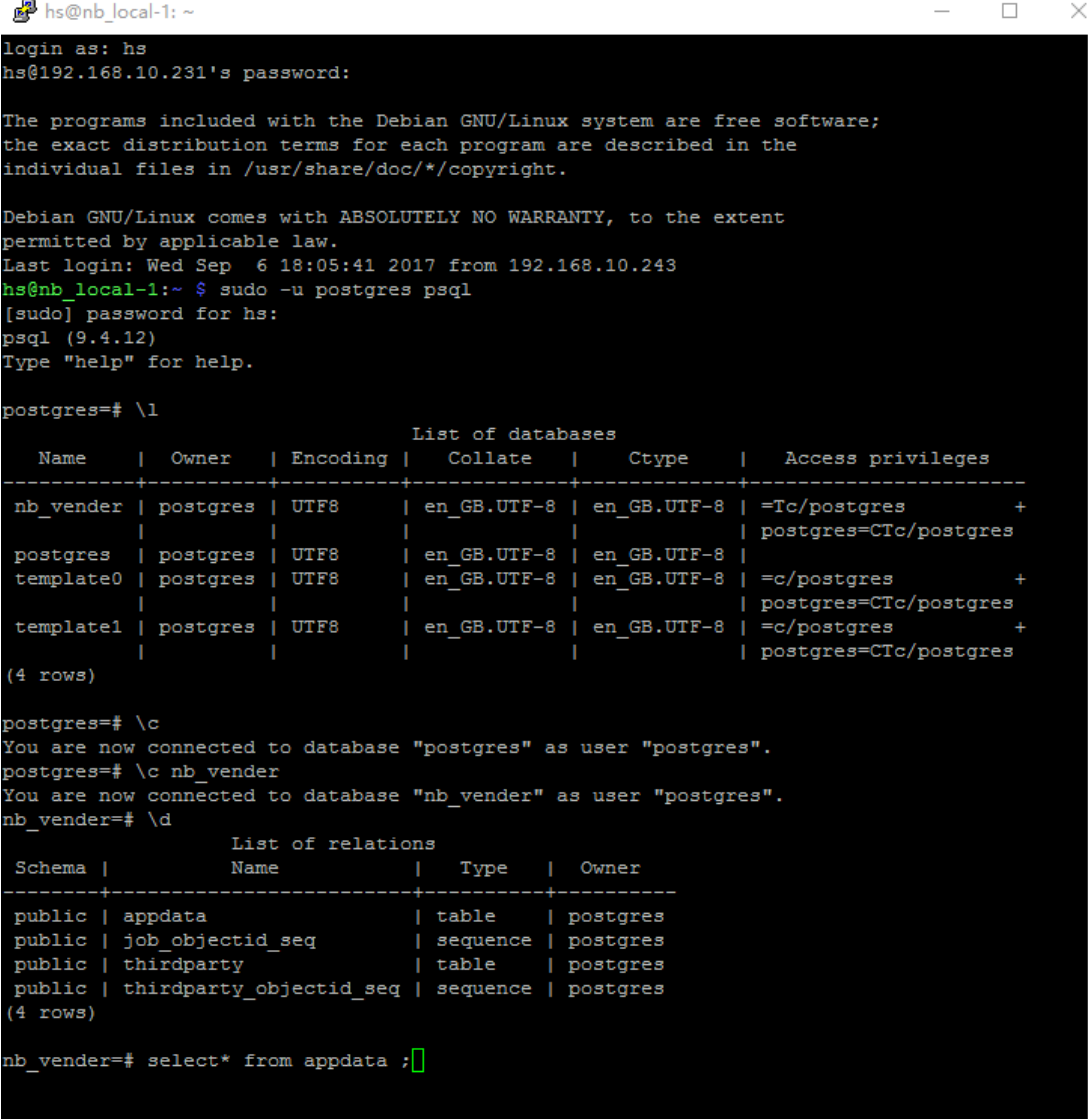
通过 `\l` 可查看当前全部数据库

通过 `\c` 可查看当前数据库

通过 `\c 数据库名` 可切换到所需数据库

通过 `\d` 可查看当前数据库下的所有表

通过 `\q` 退出



```

hs@nb_local-1: ~
login as: hs
hs@192.168.10.231's password:

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Wed Sep  6 18:05:41 2017 from 192.168.10.243
hs@nb_local-1:~$ sudo -u postgres psql
[sudo] password for hs:
psql (9.4.12)
Type "help" for help.

postgres=# \l
                                List of databases
  Name      | Owner   | Encoding | Collate |  Ctype  | Access privileges
-----+-----+-----+-----+-----+-----
 nb_vender  | postgres | UTF8     | en_GB.UTF-8 | en_GB.UTF-8 | =Tc/postgres +
            |          |          |             |             | postgres=CtC/postgres
 postgres   | postgres | UTF8     | en_GB.UTF-8 | en_GB.UTF-8 |
 template0  | postgres | UTF8     | en_GB.UTF-8 | en_GB.UTF-8 | =c/postgres +
            |          |          |             |             | postgres=CtC/postgres
 template1  | postgres | UTF8     | en_GB.UTF-8 | en_GB.UTF-8 | =c/postgres +
            |          |          |             |             | postgres=CtC/postgres
(4 rows)

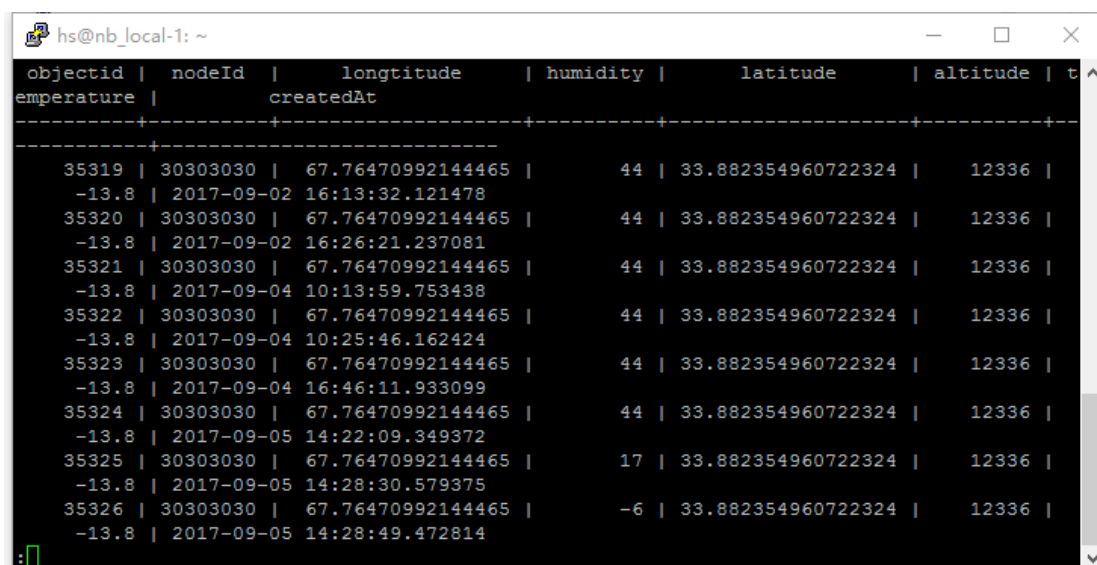
postgres=# \c
You are now connected to database "postgres" as user "postgres".
postgres=# \c nb_vender
You are now connected to database "nb_vender" as user "postgres".
nb_vender=# \d
                        List of relations
 Schema |      Name      | Type  | Owner
-----+-----+-----+-----
 public | appdata         | table | postgres
 public | job_objectid_seq | sequence | postgres
 public | thirdparty      | table | postgres
 public | thirdparty_objectid_seq | sequence | postgres
(4 rows)

nb_vender=# select* from appdata ;

```

图 6-7 登录数据库

如上图中的显示，可以使用相关的 sql 语句查询数据。下图则为查询结果：



objectid	nodeId	longitude	humidity	latitude	altitude	temperature	createdAt
35319	30303030	67.76470992144465	44	33.882354960722324	12336	-13.8	2017-09-02 16:13:32.121478
35320	30303030	67.76470992144465	44	33.882354960722324	12336	-13.8	2017-09-02 16:26:21.237081
35321	30303030	67.76470992144465	44	33.882354960722324	12336	-13.8	2017-09-04 10:13:59.753438
35322	30303030	67.76470992144465	44	33.882354960722324	12336	-13.8	2017-09-04 10:25:46.162424
35323	30303030	67.76470992144465	44	33.882354960722324	12336	-13.8	2017-09-04 16:46:11.933099
35324	30303030	67.76470992144465	44	33.882354960722324	12336	-13.8	2017-09-05 14:22:09.349372
35325	30303030	67.76470992144465	17	33.882354960722324	12336	-13.8	2017-09-05 14:28:30.579375
35326	30303030	67.76470992144465	-6	33.882354960722324	12336	-13.8	2017-09-05 14:28:49.472814

图 6-8 查询数据

## 6.3. 本地数据获取应用参考示例

### 6.3.1. express 简介

express 是一个简洁而灵活的 node.js Web 应用框架, 提供了一系列强大特性帮助你创建各种 Web 应用, 和丰富的 HTTP 工具。

使用 express 可以快速地搭建一个完整功能的网站。express 框架核心特性:

1. 可以设置中间件来响应 HTTP 请求。
2. 定义了路由表用于执行不同的 HTTP 请求动作。
3. 可以通过向模板传递参数来动态渲染 HTML 页面。

### 6.3.2. 安装 express

安装 express 并将其保存到依赖列表中:

```
$ cnpm install express --save
```

以上命令会将 express 框架安装在当前目录的 node\_modules 目录中, node\_modules 目录下会自动创建 express 目录。以下几个重要的模块是需要与 express 框架一起安装的:

- body-parser - node.js 中间件，用于处理 JSON, Raw, Text 和 URL 编码的数据。
- cookie-parser - 这就是一个解析 Cookie 的工具。通过 req.cookies 可以取到传过来的 cookie，并把它们转成对象。

```
$ cnpm install body-parser --save
$ cnpm install cookie-parser --save
```

安装完后，我们可以查看下 express 使用的版本号：

```
$ cnpm list express
/data/www/node
└──express@4.15.2->
/Users/tianqixin/www/node/node_modules/.4.15.2@express
```

项目往往存在 package.json 这个配置文件，用户往往只需要通过 cnpm install 即可安装项目所需的全部依赖，上文中的 cnpm install express --save 只是往 package.json 中添加相关的配置信息而已。

下面为该项目 package.json 文件的相关配置信息：

```
{
  "name": "node-js-getting-started",           //项目名称（必须）
  "version": "1.0.0",                          //项目版本（必须）
  "description": "A sample Node.js app using Express 4", //项目描述（必须）
  "main": "server.js",                        //程序主要项目
  "scripts": {
    "start": "node server.js"
  },
  "keywords": [                                //关键字
    "node",
    "LeanCloud",
    "LeanEngine",
    "express"
  ],
  "license": "MIT",                            //项目许可协议
  "dependencies": {                            //项目依赖包
    "body-parser": "^1.12.3",
    "connect-timeout": "^1.9.0",
    "cookie-parser": "^1.4.3",
    "cors": "^2.8.4",
    "ejs": "^2.3.1",
    "express": "^4.12.3",
    "moment-timezone": "^0.5.13",
    "mqtt": "^1.14.1",
```

```

    "pg": "^6.4.2",
    "q": "^1.5.0",
    "request": "^2.81.0",
    "underscore": "^1.8.3",
    "when": "^3.7.8"
  },
  "engines": {
    "node": "4.x"
  }
}

```

//node 所需版本号

### 6.3.3. express 框架实例

首先配置入口文件 `app.js` 的相关信息，其他所有相关的项目接口都是以 `app.js` 为入口去进行相关的调用。接下来，主要进行 `app.js` 相关的代码分析。

1. 明确自己项目所需要使用的相关的包的依赖，在文件的开头去进行相关的变量声明。

```

'use strict';
var express = require('express');
/*引用 express 框架*/
var timeout = require('connect-timeout');
var path = require('path');
/*引用路径调用的包*/
var cookieParser = require('cookie-parser');
/*引用解析 cookie 的包*/
var bodyParser = require('body-parser');
/*引用解析 body 数据的包*/
// API path
/*引用自己本地抛出的关于相关接口的包*/
var notifyurl = require('./routes/notifyurl');
var notifyids = require('./routes/notifyids');
var history = require('./routes/history');
var fakehistory = require('./routes/fakehistory');
var fakenotifyids = require('./routes/fakenotifyids');
var downlink = require('./routes/downlink');
var app = express();

```

2. 设置相关的模板引擎

默认使用的是 `jade` 模板引擎，该项目主要使用的是 `ejs` 模板，主要用于 `html` 网页的显示。

```
app.set('views', path.join(__dirname, 'views'));
```



```
app.set('view engine', 'ejs');
app.use(express.static('public'));
```

然后设置 views 目录下的相关文件，用于显示。

### 3. 使用相关的中间件

express 应用可使用如下几种中间件：

应用级中间件，路由级中间件，错误处理中间件，内置中间件，第三方中间件。

在下面将通过项目的相关程序进行相关简单的介绍：

#### ➤ 第三方中间件

```
app.use(timeout('15s'));
/*设置默认超时时间*/
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
/*主要用来解析 http request body 中的相关数据*/
```

#### ➤ 路由级中间件

```
/*绑定的对象为 express.Router()*/

/* SET NOTIFY API */
app.use('/api/notifyurl', notifyurl);
app.use('/api/notifyids', notifyids);

/* SET History API */
app.use('/api/history', history);

/* Fake history API, only use on Demo */
app.use('/api/fakehistory', fakehistory);
app.use('/api/fakenotifyids', fakenotifyids);

/* SET DownlinkAPI */
app.use('/api/downlink', downlink);
```

上面的代码使用相关的中间件的主要目的是为了调用相关的接口去进行相应的数据请求。举个例子，history 接口就主要是用来获取相关的历史数据的。

第一步中 require 了 routes 文件夹下面的 history.js 文件。

该文件主题的代码逻辑如下：

```
var router = require('express').Router();
router.get('/', function(req, res) {
```

```
/* 相关路由有各自的调用和处理数据的方式 */
})
module.exports = router;
```

主要实现的是自己写一个中间件进行相应的 http 请求，然后作为一个模块在本地抛出。

#### ➤ 错误处理中间件

```
/*如果任何一个路由都没有返回响应，则抛出一个 404 异常给后续的异常处理器*/
app.use(function(req, res, next) {
  if(!res.headersSent) {
    var err = new Error('Not Found');
    err.status = 404;
    next(err);
  }
});
```

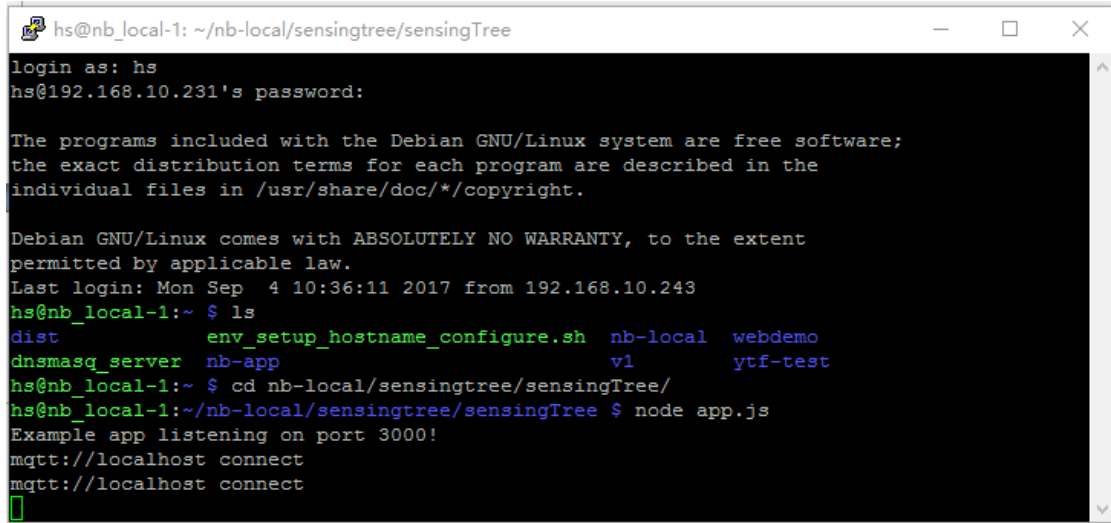
```
app.use(function(err, req, res, next) { // jshint ignore:line
  var statusCode = err.status || 500;
  if(statusCode === 500) {
    console.error(err.stack || err);
  }
  if(req.timedout) {
    console.error('请求超时: url=%s, timeout=%d, 请确认方法执行耗时很长, 或没有正确的 response 回调。', req.originalUrl, err.timeout);
  }
  res.status(statusCode);
  /*默认不输出异常详情*/
  var error = {}
  if (app.get('env') === 'development') {
    /*如果是开发环境，则将异常堆栈输出到页面，方便开发调试*/
    error = err;
  }
  res.render('error', {
    message: err.message,
    error: error
  });
  /* res.render 渲染页面，即传递数据到前台页面*/
});
```

#### 4. 开启服务，监听相关端口

```
app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

```
/* 监听 3000 端口*/
```

如下图所示，出现 Example app listening on port 3000，则说明服务已经开启。

A terminal window titled 'hs@nb\_local-1: ~/nb-local/sensingtree/sensingTree'. The terminal shows the login process for user 'hs' on IP '192.168.10.231'. It displays the Debian GNU/Linux system's free software notice and the last login time. The user runs 'ls' in the directory '~/nb-local/sensingtree/sensingTree', showing files 'dist', 'dnsmasq\_server', 'nb-app', and 'v1'. Then, the user runs 'node app.js', and the terminal outputs 'Example app listening on port 3000!' followed by two 'mqtt://localhost connect' messages.

```
hs@nb_local-1: ~/nb-local/sensingtree/sensingTree
login as: hs
hs@192.168.10.231's password:

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Mon Sep  4 10:36:11 2017 from 192.168.10.243
hs@nb_local-1:~ $ ls
dist      env_setup_hostname_configure.sh  nb-local  webdemo
dnsmasq_server  nb-app                          v1        ytf-test
hs@nb_local-1:~ $ cd nb-local/sensingtree/sensingTree/
hs@nb_local-1:~/nb-local/sensingtree/sensingTree $ node app.js
Example app listening on port 3000!
mqtt://localhost connect
mqtt://localhost connect
█
```

图 6-9 服务器开启

## 6.4. NB-IoT 节点数据解析及显示示例

### 6.4.1. 工程配置简介

演示界面样例地址如下：

<http://NB-IoT.handsometechs.com/>

工程中，文件夹、文件架构如下：

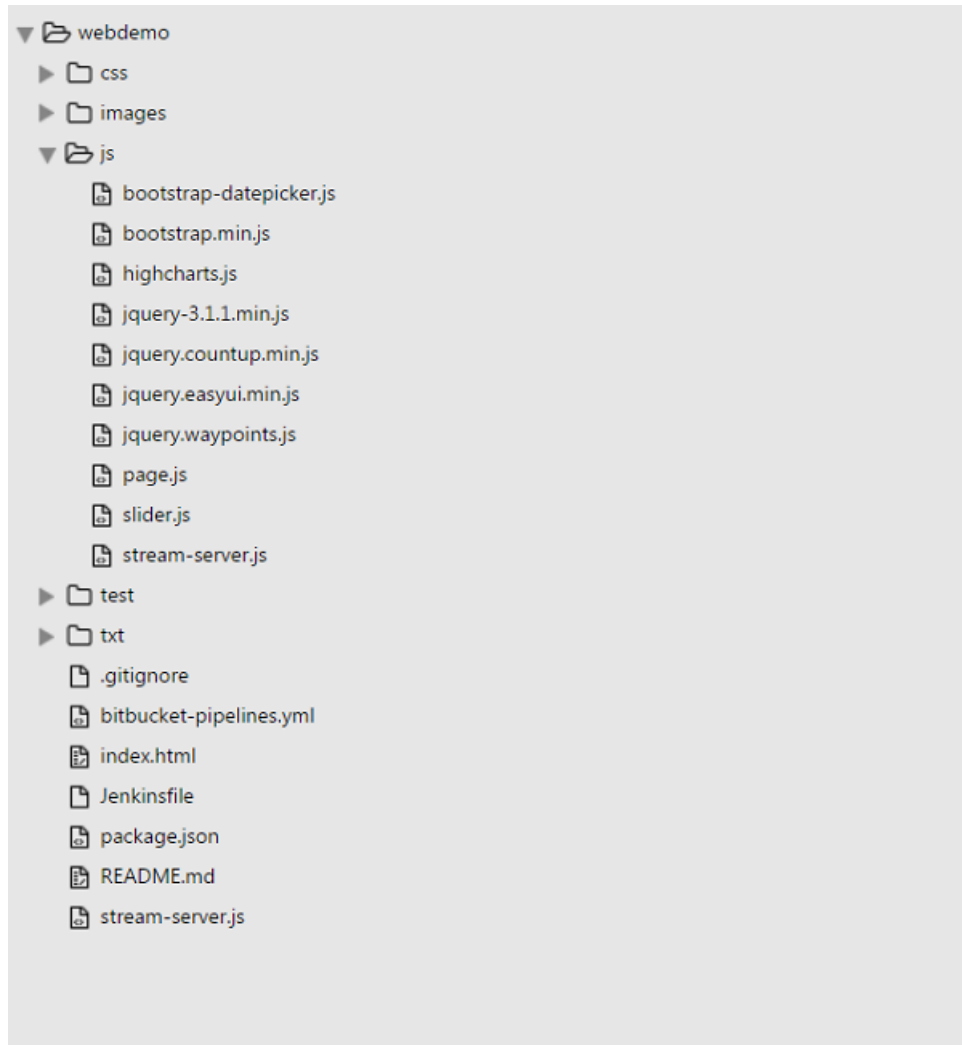


图 6-10 工程文件目录

目录结构：

- Webdemo\：工程文件夹
- Css\：项目样式表
- Images\：图片资源
- js\：依赖的外部库和数据解析与显示代码
- test\：测试代码
- index.html：前端显示界面

### 6.4.2. NB-IoT 节点数据获取

本章针对 NB-IoT 的应用主要在于：

Jquery-3.1.1.min.js: 引用 jquery 库。

Page.js:获取 NB-IoT 节点数据文件。

➤ **Jquery-3.1.1.min.js**

Jquery 资源库，jquery 代码运行的必须依赖。

➤ **Page.js**

首先，通过 http 请求接口 nbdata.handsometechs.com/api/fakenotifyids，获取 NB-IoT 节点 ID:

```
$.ajax({
  /*http 请求 headers 参数*/
  headers: {
    "username": "nbiot",
    "password": "nbiot",
    "Content-Type": "application/json"
  },
  /*http 请求方式为 get*/
  type: "get",
  /*请求数据格式是 JSON*/
  dataType: "json",
  /*http 请求地址*/
  url: "http://nbdata.handsometechs.com/api/fakenotifyids",
  /*http 请求成功*/
  success: function (ids) {
    pointsMap();
  },
  /*http 请求失败*/
  error: function (error) {
  }
});
```

显示界面选择节点 ID，获取该节点的历史数据和最新数据。

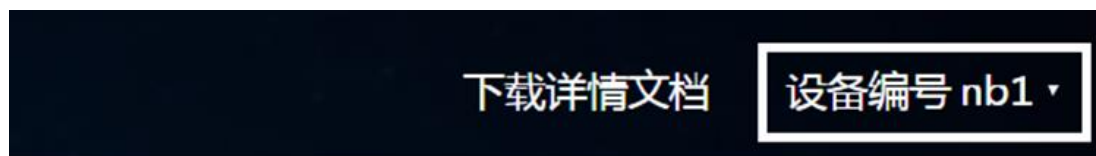


图 6-11 选择 NB-IoT 节点

通过 jquery 中的 ajax 发送 http 请求，获取最新数据和历史数据；若获取数据成功，继续下面的显示流程，若获取数据失败，返回失败状态：

```
/*http 请求参数*/
$.ajax({
  .....
  /*http 请求成功*/
```

```

    success: function (data) {
        Get_Char_Grid(param, callback)
    },
    /*http 请求失败*/
    error: function (error) {
        .....
    }
});

```

通过以上两步可以获取到 NB-IoT 节点的原始数据。

### 6.4.3. NB-IoT 节点数据解析及显示

#### ➤ Page.js

解析原始数据，得到我们需要的数据类型：

```

Get_Char_Grid(param, function (data) {
    /*获取最新数据*/
    var lastdata = data.records[data.records.length - 1];
    /*获取数据后，在图表、地图中显示*/
    bindGridData(data);
    bindCharData(data);
    if (lastdata) {
        BaiduMap(lastdata);
        yibiaoshow(lastdata);
    }
});

```

获取 NB-IoT 节点坐标，在地图中显示节点当前位置，如图 1-3，调用的是百度地图的 API：

```

/*显示节点坐标函数*/
function mapPosition(data, id) {
    /*判断是否已存在该点*/
    if (idArray.indexOf(id) < 0) {
        .....
    } else {
        /*显示节点坐标*/
        var point = new BMap.Point(data.x, data.y);
        map.centerAndZoom(point, 16);
    }
}

```

下图是 NB-IoT 节点在显示界面中展示的效果，调用在线版百度地图，红点为 NB-IoT 节点当前所在位置：



图 6-12 NB-IoT 节点位置信息

获取 NB-IoT 节点最新数据，调用 countUp 显示最新温度和湿度参数：

```
function yibiaoshow(data) {
    /*获取最新数据，显示温度和湿度 */
    $('#map_right .temp .counter').html(data[5]);
    $('#map_right .humi .counter').html(data[6]);
    $('#counter').countUp({
        time:300
    });
}
```

显示界面，最新温度和湿度显示如下图：

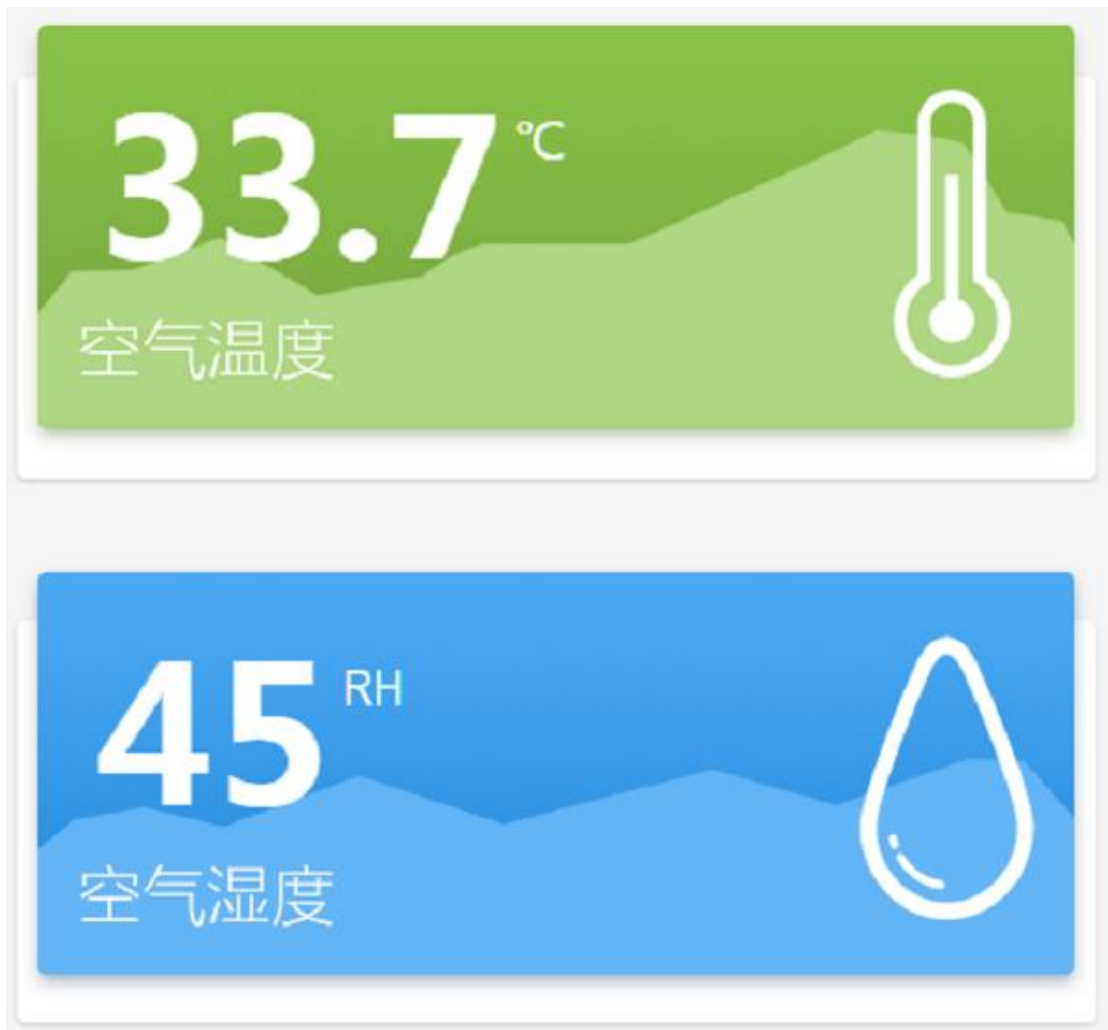


图 6-13 NB-IoT 节点温湿度数据

NB-IoT 节点历史数据曲线以图表显示，调用的是 highcharts 的 type:line 折线图功能：

```
var data_char = [],  
data_x = [];  
.....  
/*历史数据显示到图表 */  
var chart = $('#container').highcharts(options).highcharts();
```

显示界面历史数据曲线的显示效果如下图，横坐标为时间轴，两条曲线分别为温度和湿度的变化曲线：





图 6-14 数据采样图

NB-IoT 节点数据记录以表格显示，调用的是 easyUI 的 datagrid 表格功能：

```
$( '#container1' ).datagrid({
    rowStyler: function (index, row) {
    },
    data: data_grid1,
    columns: [ ..... ]
});
```

显示界面，历史数据记录的显示如下图所示，以表格形式将 NB-IoT 节点数据直观的显示：

nodeID	Temperature(°C)	Humidity(RH)	Acquisition Time
nb1	33.6	46	Aug 26 2017 07:37:09
nb1	33.7	45	Aug 26 2017 07:32:12
nb1	34.2	44	Aug 26 2017 07:27:10
nb1	33.7	45	Aug 26 2017 07:22:02
nb1	33.4	46	Aug 26 2017 07:17:11
nb1	33.1	47	Aug 26 2017 07:12:14
nb1	32.8	47	Aug 26 2017 07:07:17
nb1	32.8	48	Aug 26 2017 07:02:15
nb1	32.7	48	Aug 26 2017 06:57:15
nb1	32.7	48	Aug 26 2017 06:52:18
nb1	32.6	49	Aug 26 2017 06:47:19
nb1	32.4	49	Aug 26 2017 06:42:19

图 6-15 数据记录