



EdgerOS 应用

开发指南

文档版本 V1.0 / 发布日期 2021-10-28

| | |
|----------------------|----|
| EdgerOS 简介..... | 1 |
| 爱智应用简介..... | 4 |
| JSRE 简介..... | 6 |
| 环境准备..... | 8 |
| 创建应用..... | 9 |
| 部署应用..... | 18 |
| 概述..... | 22 |
| 开发一个爱智应用..... | 23 |
| 首个爱智应用示例..... | 23 |
| 集成 Web SDK..... | 29 |
| 完成服务端认证..... | 32 |
| 开发 RESTful..... | 35 |
| 优化 UI 展示效果..... | 39 |
| 优化性能..... | 48 |
| 上传文件..... | 50 |
| 扩展前端通信..... | 53 |
| 服务端安全认证..... | 53 |
| 开发 Socket.IO..... | 57 |
| 开发 WebSyncTable..... | 60 |
| 数据库组件..... | 63 |
| SQLite3..... | 63 |
| LightKV..... | 66 |
| SyncTable..... | 68 |
| Redis..... | 70 |
| 扩展功能..... | 71 |

| | |
|-------------------------|-----|
| 多任务组件..... | 71 |
| 设备管理框架..... | 76 |
| AI 框架..... | 82 |
| 多媒体框架..... | 95 |
| 使用 TypeScript 语言开发..... | 102 |
| 离线下载..... | 104 |
| 智能开关..... | 111 |
| 甲醛检测器..... | 116 |
| 智能插座..... | 124 |
| IoT Pi 控制..... | 129 |
| 智能摄像头..... | 134 |
| 智能门锁..... | 137 |
| 人脸识别..... | 140 |
| 开发者社区..... | 142 |
| 代码示例..... | 143 |
| EPM..... | 144 |
| 其他资源..... | 147 |
| 概述..... | 148 |
| 设计准则..... | 149 |
| 设计风格..... | 150 |
| 控件..... | 161 |
| 常见问题..... | 167 |

EdgerOS 简介

什么是 EdgerOS

EdgerOS 是翼辉信息自主研发，面向“万物互联”时代的新一代智能边缘计算操作系统。她继承了翼辉信息自主知识产权嵌入式实时操作系统 SylixOS 的稳定、安全、可靠、开放等特点，聚焦边缘计算领域，致力于解决行业发展的核心问题。EdgerOS 主要部署在应用场景的边缘侧，靠近用户业务数据源头，能够联合物联网产业链上下游，为用户提供近端边缘计算服务，满足行业在低时延、高带宽、安全与隐私保护等方面的需求。

产品特点

- 自主内核

EdgerOS 采用完整自主知识产权的 SylixOS 内核，是一款完全自主设计的智能边缘计算操作系统。自主内核让 EdgerOS 拥有了强大的系统演进能力，可从底层或其他更深度的维度进行整合优化，为合作伙伴提供更加深度的技术支持。同时自主内核可以有效摆脱国外的技术控制，打破技术壁垒，拥有更广阔的发展空间。

- 极简开发

EdgerOS 不仅在内核上打造了一个高性能的 JSRE 引擎，还提供了功能非常丰富的开发框架，包括 App 框架、流媒体框架、AI 框架、设备管理框架等。同时，EdgerOS 提供了非常强的设备抽象能力，设备开发者无论通过标准协议还是私有协议，都可以将设备能力接入爱智系统，并通过爱智设备框架的标准接口被上层的应用开发者使用，真正实现了软硬件的解耦。此外，爱智还提供了 AI 接口和爱智云服务，开发者无需关心用户账号、设备远程访问、应用的分发、支付等技术问题，只需要专注于业务功能的开发，有效降低应用开发的难度和成本，提高开发效率。

- 开放生态

EdgerOS 通过在操作系统层面对物联网设备和服务的抽象和标准化，实现了物联网软硬件的完全解耦，能够有效促进产业分工，推动整个产业进行重构。秉承开放的心态和理念，EdgerOS 不限制智能设备的厂家或者品牌，能够为所有的开发者提供完备的技术支持，设备和应用之间可以随意组合，应用开发者和设备开发者可以随意组合。开发者可以自由选择标准协议或私有协议，或从爱智应用市场、设备中心中获取海量的开发资源。EdgerOS 不重新发明标准，而是将硬件、应用和云等关键元素进行连接，为开发者提供广阔的开发土壤。

- 随心访问

EdgerOS 通过爱智云为开发者和消费者提供远程访问能力。无需新增任何代码，也无需任何后端配置，随时随地即可远程访问系统底层设备。

- 安全无忧

EdgerOS 具备强大的安全保障能力。在底层安全方面，EdgerOS 内核来自翼辉信息嵌入式实时操作系统 SylixOS，SylixOS 已通过操作系统 SIL3/SIL4 国际安全等级认证，此认证是操作系统领域最高等级安全认证，通过 SIL 等级认证的操作系统则可以应用在极高安全性要求的轨道

交通牵引系统、信号系统、车身控制系统等高安全的行业领域。在上层安全方面，EdgerOS 使用了基于安全容器的沙箱技术和基于数字证书的多域安全认证，对访问的安全性做到有效防护。同时 EdgerOS 拥有多维度权限管理功能，支持配置用户-应用-设备间的访问权限，对系统能力进行组合，保证使用的安全性。

产品架构

EdgerOS 为操作系统产品，整个系统架构共分为五层，从下向上分别为：内核层、JSRE、系统服务层、应用框架层和应用层。



内核层

EdgerOS 内核来自翼辉信息嵌入式实时操作系统 SylinxOS，SylinxOS 已历经数十年持续开发，拥有超百万行代码，内核自主化率 100%；并且经过航天、军工、轨交等众多行业的验证，首批搭载设备已累计八万小时不掉电运行，迄今已稳定承载上千个项目，真正做到了坚若磐石，万无一失。

JSRE

EdgerOS 的高性能 JavaScript 运行时把互联网技术栈带入了物联网和边缘计算领域，在封装了大量基础库的同时，为上层框架提供了丰富的 API，EdgerOS 的基础库包含了安全加密、AI、多媒体、数据库和多种通信协议等核心能力。

系统服务层

EdgerOS 在系统服务层提供了丰富的系统服务和基础组件，基础组件包含了 General（通用组件）、Multi-Task（多任务）、Network（网络）、IoT Protocol（物联网协议）、File System（文件系统）等；系统服务包含了 Account（账户服务）、Master（桌面服务）、Notify（系统通知服务）、Permission（权限管理服务）和爱智云服务等众多服务。

应用框架层

应用框架层涵盖了 App 框架、流媒体框架、AI 框架和设备管理框架四大能力集合。其中 App 框架提供了丰富的基础组件，降低了应用开发的复杂度；流媒体框架和 AI 框架，为开发者提供了开箱即用的人工智能接口和流媒体处理能力；设备管理框架实现了对智能设备的高度抽象和封装，开发者不需要了解智能设备的底层通信协议细节，只需要调用高度抽象的设备接口，就可以快速发现和连接设备，并调用设备的各项功能。

应用层

EdgerOS 应用层提供了几款内置应用，最终用户可以通过设置、设备和网络等内置应用管理自己的智能设备、用户分组、权限和无线网络等设置，同时可以通过爱智世界下载和安装丰富多彩的爱智应用。

爱智应用简介

什么是爱智应用

爱智应用是指运行在 EdgerOS 操作系统上的智能应用程序。除了 EdgerOS 系统内置的应用外，开发者可以通过爱智世界向广大爱智设备用户分发和销售自己的作品。此外，通过在爱智应用内集成翼辉支付，开发者也可以通过应用内购买盈利。EdgerOS® 是为万物互联时代而生的智能操作系统。它为广大开发者提供了基于互联网技术栈的开发平台，极大简化了应用的开发难度，提高开发效率。

产品优势

- 低代码

EdgerOS App 使用 JavaScript / TypeScript 语言开发，无须驱动程序等底层嵌入式开发。使用应用开发模板和完整的应用框架，开发者只需编写应用本身的代码即可。

- 平台赋能

爱智开发者平台提供完整的文档、SDK、[开发套件](#) 甚至人工智能接口，助力开发者快速开发出功能强大的智能应用，投身全新的“万物互联”时代。

- 完整生态

除了 EdgerOS 系统本身，爱智云为广大用户提供了物联网与互联网无缝互通的使用体验。通过开源开放的通讯协议，设备开发者使用 FreeRTOS，MS-RTOS® 等面向物联网的操作系统可以轻松地构建出与 EdgerOS 自动接入的万千智能设备，与爱智云、爱智设备一起形成云、边、端一体的产品体系，衍生出充满无限想象的各类智慧场景。

- 商业闭环

爱智世界作为爱智应用的分发渠道，将开发者与广大用户相连。将爱智应用发布在爱智世界中，不仅可以直接销售应用本身，还可以通过应用内购买为用户提供更为丰富的功能和服务。

产品生态

爱智应用运行于爱智设备上，扩展了设备的能力，为设备用户提供了更丰富的功能和体验。爱智设备是 EdgerOS 智能操作系统的载体，它为操作系统提供了各种计算所需的硬件资源和网络连接能力。爱智手机应用是连接用户与爱智设备的桥梁，是用户使用和体验爱智智能设备的窗口。

- 爱智设备

爱智设备泛指翼辉信息等智能设备厂商研制的，搭载 EdgerOS 操作系统的智能边缘计算机。爱智设备具有全面的互联网和物联网连接能力，内置多种人工智能算法，能够出色地完成各种智能计算任务。

- 爱智手机应用

爱智手机应用安装在手机、Pad、电脑等终端上，使这些终端成为 EdgerOS 的显示器。多个用户可以通过爱智手机应用同时使用同一台爱智设备。无论是通过本地无线网络连接还是远程的

爱智云服务，爱智手机应用可以在多台设备间切换，为用户提供无缝的使用体验。

edgeros.com

edgeros.com

JSRE 简介

什么是 JSRE

JSRE (JavaScript Runtime Environment) 是 EdgerOS 智能操作系统中的 JavaScript 运行环境，它把互联网技术栈带入了物联网和边缘计算领域，在封装了大量基础库的同时，为上层框架提供了丰富的 API。EdgerOS 的基础库包含了安全加密、AI、多媒体、数据库和多种通信协议等核心能力，极大地降低了应用程序开发难度。

产品特点

- 极简开发

JSRE 打通了互联网和物联网技术栈，开发者可以使用熟悉的互联网技术开发物联网应用，同时系统提供功能丰富、开箱即用的开发框架和人工智能接口，有效降低应用开发的难度和成本，提高开发效率。

- 多线程

JSRE 对 JavaScript 语言风格做了重大改进，提供了一个标准的多线程环境，您可以在 JSRE 中创建多个线程处理不同的任务，来提高程序的并发处理能力。同时，JSRE 还保留了 JavaScript 异步特性，您可以使用任何喜欢的机制(同步或异步)开发应用程序，因此基于 JSRE 开发应用程序非常灵活。

- 轻量化

相较 Node.js 而言 JSRE 消耗更少的系统内存，使用起来更轻量化，更适合在嵌入式开发领域使用。

- 设备和协议抽象

基于 JSRE 的 EdgerOS 提供了强大的设备抽象能力，设备开发者无论通过标准协议还是私有协议，都可以将设备能力接入爱智系统，并通过爱智设备框架的标准接口被上层的应用开发者使用，上层应用开发者不需要了解智能设备的底层通信协议细节，只需要调用高度抽象的设备接口，就可以快速发现和连接设备，并调用设备的各项功能。

API 概览

JSRE 中除了常见的系统 API 外，还内置了一些框架和扩展 API：

- Web 框架
- 数据库
- 多任务，多进程
- 文件系统
- 网络
- 路由
- IoT 设备

- 多媒体
- AI
- 其他通用模块

关于 JSRE 的更多信息，请参考 [JSRE 使用方法](#) 和 [JSRE API](#)。

edgeros.com

edgeros.com

环境准备

本章主要介绍开发爱智应用的工具准备和工具配置。

操作步骤

1. 安装 Visual Studio Code

下载并安装 [Visual Studio Code](#) 工具。

2. 安装 EdgerOS 扩展插件

在 VSCode 中 [下载 EdgerOS 扩展插件](#) 并安装。开发者可通过插件获取爱智应用开发模板，以及在爱智设备上打包、上传、安装或更新爱智应用的能力。

3. 安装 EdgerOS 安全证书

在开发环境中 [安装 EdgerOS 安全证书](#)。

创建应用

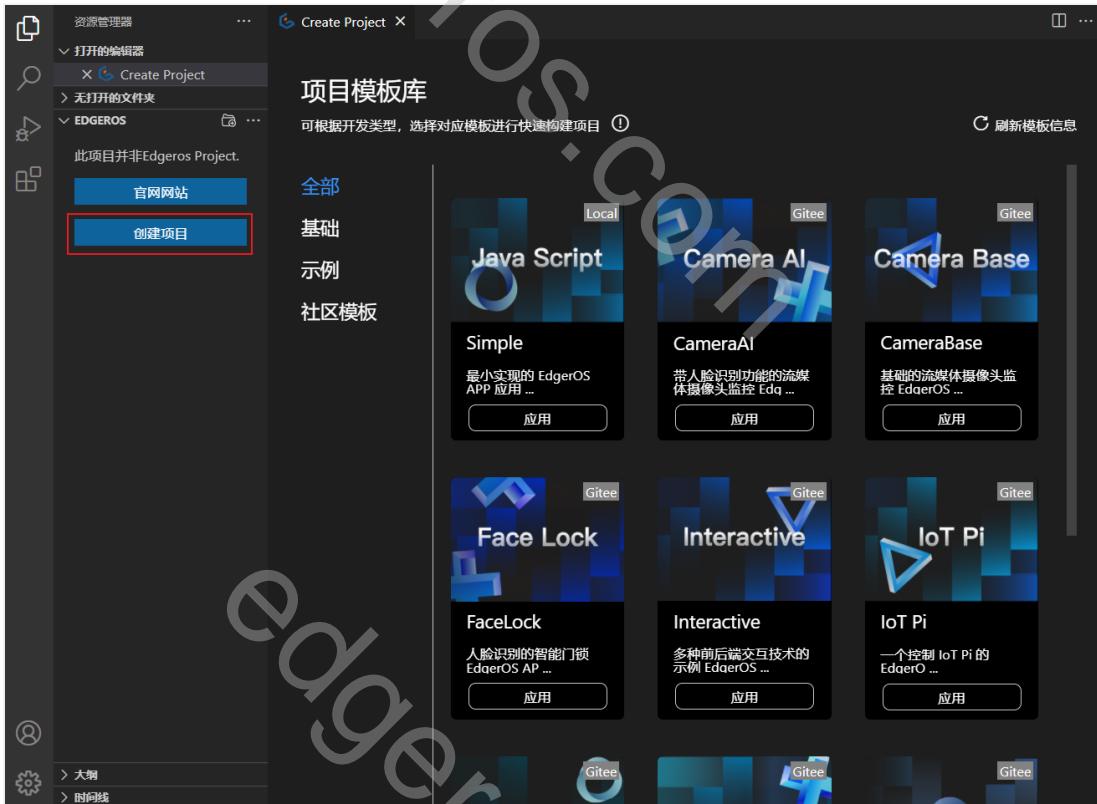
本章主要介绍如何使用 EdgerOS 扩展插件快速创建出爱智应用，以及工程文件详解。

前提条件

完成 环境准备。

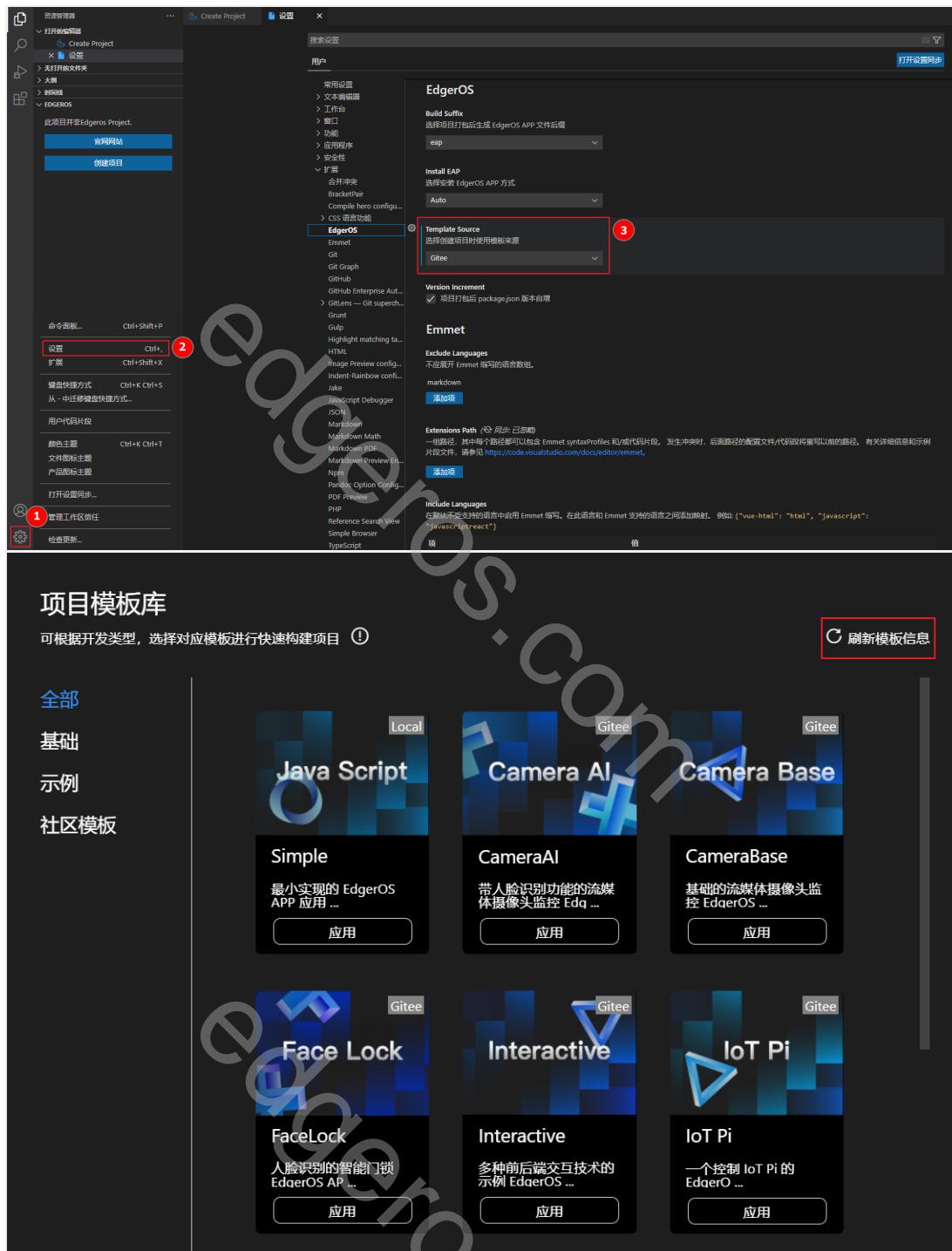
操作步骤

1. 打开 Visual Studio Code，点击插件 **EdgerOS>创建项目**，选择所需的模板后点击应用。

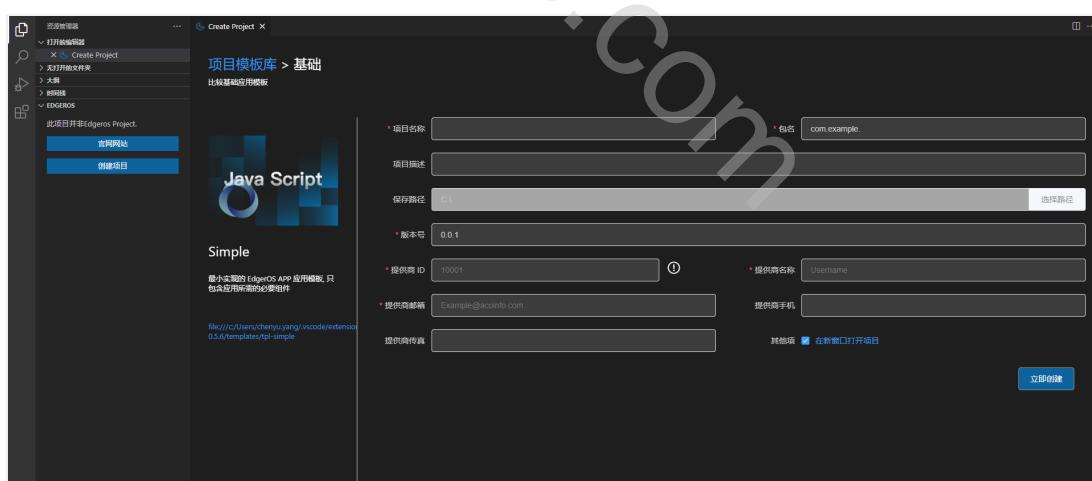


注意：

默认模板源为 *github*，若模板无法加载可按照以下方式切换模板源：点击 VSCode 左下方的管理按钮，选择**设置>扩展>EdgerOS**，将 *Template Source* 切换为 *Gitee*，然后在创建模板页面点击**刷新模板信息**按钮即可。



2. 参考表 1 填写模板参数，填写完成后点击立即创建按钮，完成应用创建。



注意：

创建完成后需进入创建的项目根目录，执行 `npm install` 命令安装依赖包。

表 1 创建应用参数模板

| 参数 | 说明 | 取值样例 |
|----------------------|-----------------------|----------------------------------|
| 项目名称 | 该项目的工程文件名称 | test |
| 包名 | 软件包名称 | com.example.myapp |
| 项目描述 | 简要说明项目用途 | 测试项目 |
| 保存路径 | 项目在本地保存路径 | - |
| 版本号 | 应用的版本号 | 0.0.1 |
| 提供商 ID | 开发者 ID (从开发者网站个人信息查询) | 479ace68109611ecbf6b00163e163bca |
| 提供商名称 | 开发者 ID 对应的用户名 | acoUsername |
| 提供商邮箱 | 开发者的邮箱 | example@example.com |
| 提供商手机 | 开发者的手机号 | 123456789 |
| 其他项： 在新窗口 打开项目 | 是否在新的 VScode 窗口打开项目 | - |

创建示例

- 以创建 Vue 应用为例，打开已创建的项目工程，在 public 目录下打开 index.html，修改其内容如下：

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Hello EdgerOS</title>
    <script src=".//vue/vue.min.js"></script>
  </head>
  <body>
    <h1>Hello EdgerOS</h1>
  
```

```

<div id="app">
    Counter: {{ counter }}
</div>
<script>
    var app = new Vue({
        el: "#app",
        data() {
            return {
                counter: 0,
            };
        },
        mounted() {
            setInterval(() => this.counter++, 1000);
        },
    });
</script>
</body>
</html>

```

2. 修改完毕并保存后，一个最简单的 Vue 示例就创建完成了。

补充说明

文件结构介绍

创建应用项目后会在项目目录下生成一个文件夹（文件夹名称为项目名称），作为项目根目录。这个文件夹中已经包含了项目配置文件与示例页面的初始代码，主要结构如下：

| | |
|---------------|----------------|
| assets | 资源文件夹 |
| routers | 路由信息 |
| public | 静态页面文件 |
| views | 模板页面 |
| eslintrc.json | eslint 配置文件 |
| edgeros.json | edgeros 应用配置文件 |
| main.js | 程序入口 |
| jsconfig.json | 代码补全配置文件 |
| package.json | 依赖包的管理 |

edgeros.json 介绍

edgeros.json 示例

```
{
    "name": "default",
    "bundleid": "com.example.myapp",
    "test": "./test/index.js",
}
```

```

"ignore_modules": ["@edgeros/eslint-plugin-jsre", "@edgeros/jsre-
types"],
"native_modules": ["dayjs", "lodash"],
"ignore_path": ["./test/**", "./temp/**"],
"assets": {
    "ico_big": "assets/big.png",
    "ico_small": "assets/small.png",
    "ico_widget": "assets/widget.png",
    "splash": "assets/splash.png"
},
"program": {
    "gss": false,
    "log": "console",
    "will": false,
    "reside": false,
    "mesv": "1.0.0",
    "resource": "public",
    "experimental": false
},
"loading": {
    "splash": "splash",
    "background": "#000000",
    "animation": "enlarge"
},
"vendor": {
    "id": "479ace68109611ecbf6b00163e163bca",
    "name": "mycompany",
    "email": "example@example.com",
    "phone": "123456789",
    "fax": ""
},
"update": {
    "remove": ["public"]
},
"widget": [
{
    "ico": "ico_widget",
    "name": "My Widget",
    "path": "widget",
    "rows": 2,
    "columns": 2,
    "category": "Demo"
}
]
}

```

注意：

`edgeros.json` 是 VScode EdgerOS 开发插件中的一项配置文件，用于表明此文件的根目录下是一个爱智应用项目。在编译过程中，VSCode 插件会根据 `edgeros.json` 的配置进行动态编译，并生成 `desc.json` 文件。`desc.json` 是编译后产生的文件，仅保留了爱智应用的必要描述。

edgeros.json 字段

| 字段 | 类型 | 描述 | 用途 |
|----------------|-------|-----------|---|
| name | 字符串 | App 名称 | 在爱智世界中展示时的应用名称 |
| bundleid | 字符串 | App 的包名 | App 包名 (bundleid)，建议使用反域名序列，例如 com.example.myapp |
| test | 字符串 | 测试脚本的入口路径 | 用以指定 EdgerAPP 项目中单测脚本文件的入口路径，例如 ./test/index.js |
| ignore_modules | 字符串数组 | module 名称 | 过滤不参与打包的 module 名称 |
| native_modules | 字符串数组 | JS 模块名称 | 用以声明当前 APP 使用的非 EdgerOS 官方包，构建时会将这些包打进构建产物以正常使用。需使用者自行判断该包是否可以在 JSRE 中运行。一般原生的 JS 非 Node 模块都可以在 JSRE 中正常运行。例如 ['dayjs', 'lodash'] |

ignore_path

| 字段 | 类型 | 描述 | 用途 |
|-------------|-------|-------|--------------|
| ignore_path | 字符串数组 | 字符串路径 | 过滤不参与打包的文件路径 |

assets

| 字段 | 类型 | 描述 | 用途 | 注意事项 |
|-----------|-----|------------------|--------------------|---|
| ico_big | 字符串 | App 必须的高清图标的资源路径 | 暂未使用 | 必须存在于软件包的根目录下且为 PNG/JPG/JPEG 格式； 在深色模式和浅色模式下必须保持清晰； 图标尺寸为 160*160(px) 至 240*240(px)； 文件大小不超过 100 KB。 |
| ico_small | 字符串 | App 必须的小图标的资源路径 | 用于 App 桌面图标的圆角矩形图片 | 必须存在于软件包的根目录下且为 PNG/JPG/JPEG 格式； |

| | | | | |
|------------|-----|-----------------|------------------------------------|---|
| | | | | 在深色模式和浅色模式下必须保持清晰；图标尺寸为 160*160(px) 至 180*180(px)；文件大小不超过 100 KB。 |
| ico_widget | 字符串 | 可自定义名称的其它可引用的资源 | 此处 ico_widget 定义 widget 中图标图片文件的路径 | - |
| splash | 字符串 | 可自定义名称的其它可引用的资源 | 此处 splash 定义欢迎页图片文件的路径 | 格式为 PNG/JPG/JPEG；文件大小不超过 200 KB。 |

program

| 字段 | 类型 | 默认值 | 权限申请 | 描述 | 用途 |
|----------|---------|---------|------|--|--------------------|
| gss | 选填，布尔值 | false | - | 相同开发商 (vendor.id) 发布的所有 App 可通过 gss 通信，不同开发商的 App 通过 zone 隔离；系统服务的 zone 为空；App 无法创建空的 zone | 用于控制同一开发商 App 间的通信 |
| log | 选填，字符串 | console | - | "file"：保存日志到文件；日志文件有两个，写满后自动滚动输出到另外一个；"null"：不输出任何日志；"console"：仅允许在开发调试阶段使用 | 用于控制日志的输出方式 |
| will | 选填，布尔值 | false | 需要 | App 不会被立即杀死，以便完成自己的任务。在应用关闭时，该应用可以通过 Process 模块监听 will 事件，详细信息请参见 Process Events | 用于控制 App 的遗嘱权限 |
| reside | 选填，布尔值 | false | 需要 | 常驻内存权限，比如说提供 widget 的 App | 用于控制 App 的常驻内存权限 |
| mesv | 必填，整数数组 | N/A | N/A | 整数数组 Minimal Edger system version，即最低兼容的 EdgerOS 版本 | 填写最低兼容的 EdgerOS 版本 |
| resource | 必填， | public | N/A | resource 字段的值对应的文件夹用来存放前端缓存资源内容，该 | EdgerOS 将提供 |

| | | | | | |
|--------------|--------|-------|-----|-----------------|---------------------|
| | 字符串 | | | 文件夹必须存在且内容不能为空 | 预加载服务，加速应用启动速度 |
| experimental | 选填，布尔值 | false | N/A | 表示该应用是否是一个实验室应用 | 用于控制 App 是否为一个实验室应用 |

loading

| 字段 | 类型 | 描述 | 用途 |
|------------|-----|-------------------------|---------------------------------------|
| splash | 字符串 | splash 引用 assets 字段中的资源 | App 加载首屏的图片文件，引用 assets 字段内定义资源文件的字段名 |
| background | 字符串 | #000000 格式的颜色编码值 | App 加载首屏的图片文件的背景色 |
| animation | 字符串 | enlarge 动画效果：扩大 | App 加载完成后，首屏图片的退出动画效果 |

update

| 类型 | 字段 | 可选字段 | 描述 | 用途 |
|----|--------|------|--------|-------------------|
| 数组 | remove | - | 自动清理目录 | 应用升级时，系统自动清理的目录集合 |

widget

| 字段 | 类型 | 字段说明 | 描述 | 用途 |
|---------|-----|------|--|--------------------------------|
| ico | 字符串 | 必填 | widget 图标，例如：ico_widget | widget 图标，引用 assets 字段内定义的资源文件 |
| name | 字符串 | 必填 | widget 显示名称 | 用于记录 widget 图标名称 |
| path | 字符串 | 必填 | widget page URL 路径，最终会与 App 的 hostname 拼接成完整 URL | 用于记录 widget 的加载路径 |
| rows | 数字 | 必填 | widget 占用行数 | 用于记录 widget 占用桌面栅格的行数 |
| columns | 数字 | 必填 | widget 占用列数 | 用于记录 widget 占用桌面 |

| | | | | 栅格的列数 |
|----------|-----|----|---|--|
| category | 字符串 | 可选 | widget 所述分类，多个 widget 可属于一个组，EdgerOS 会预定义一些分类，没有指定分类则分类名以 name 代替 | 记录 widget 的分类，可以从 EdgerOS 的分类里选择，如果没有指定分类则分类名以 name 代替 |

vendor

| 字段 | 类型 | 字段说明 | 描述 | 用途 |
|-------|-----|------|--|---------------------------|
| id | 字符串 | 必填 | 系统为用户生成的 ID,可以在爱智开发者平台的个人信息中查看，例如： 479ace68109611ecbf6b00163e163bca | |
| name | 字符串 | 必填 | 软件开发商名称 | 用于记录 App 开发者的用户名或者开发厂商的名称 |
| email | 字符串 | 必填 | 软件开发商的联系电子邮件地址 | 用于记录 App 开发商的电子邮件地址 |
| phone | 字符串 | 可选 | 软件开发商的联系电话 | 用于记录 App 开发商的联系电话 |
| fax | 字符串 | 可选 | 软件开发商的传真号码 | 用于记录 App 开发商的传真号码 |

部署应用

本章主要介绍如何将爱智应用部署到爱智设备。

前提条件

- 准备一台 Spirit 1 并激活，具体步骤请参考 Spirit 1 包装盒内附带的安装指南。
- 完成 [环境准备](#)。
- 完成 [创建应用](#)。

操作步骤

步骤 1：开启开发模式

- 在 PC 端连接“EOS-xxx”无线网络(网络名称请从 Spirit 1 设备底部的标签获取)。
- 在浏览器输入 <https://192.168.128.1/>，使用管理员身份的翼辉 ID 或设备密码进行登录。
- 在爱智桌面点击[设置>开发模式](#)，打开开发模式开关，并获取开发密码。



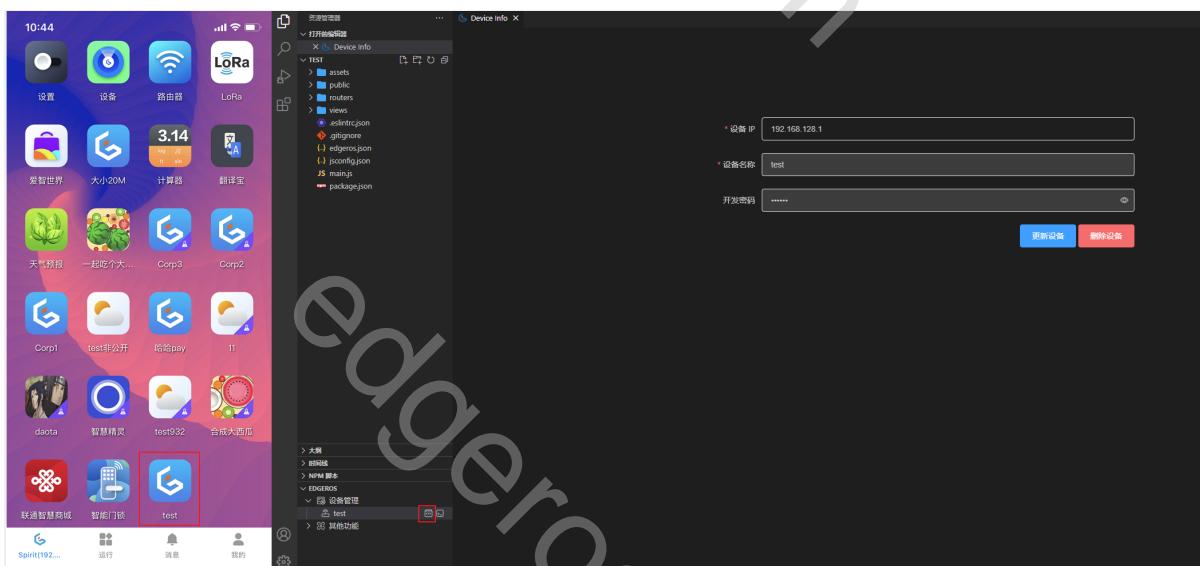
步骤 2：添加设备

- 在 VSCode 中打开已创建的应用项目，点击**EDGEROS>设备管理>添加设备按钮**（爱智设备的默认局域网 IP : 192.168.128.1）。
- 填写**设备 IP、设备名称和开发密码**，然后点击**添加设备按钮**。



步骤 3：部署应用

点击设备菜单右侧的**安装 EdgerOS App**按钮，应用将自动打包并发布至设备桌面，在设备桌面可以看到爱智应用的桌面图标，点击即可进行访问。



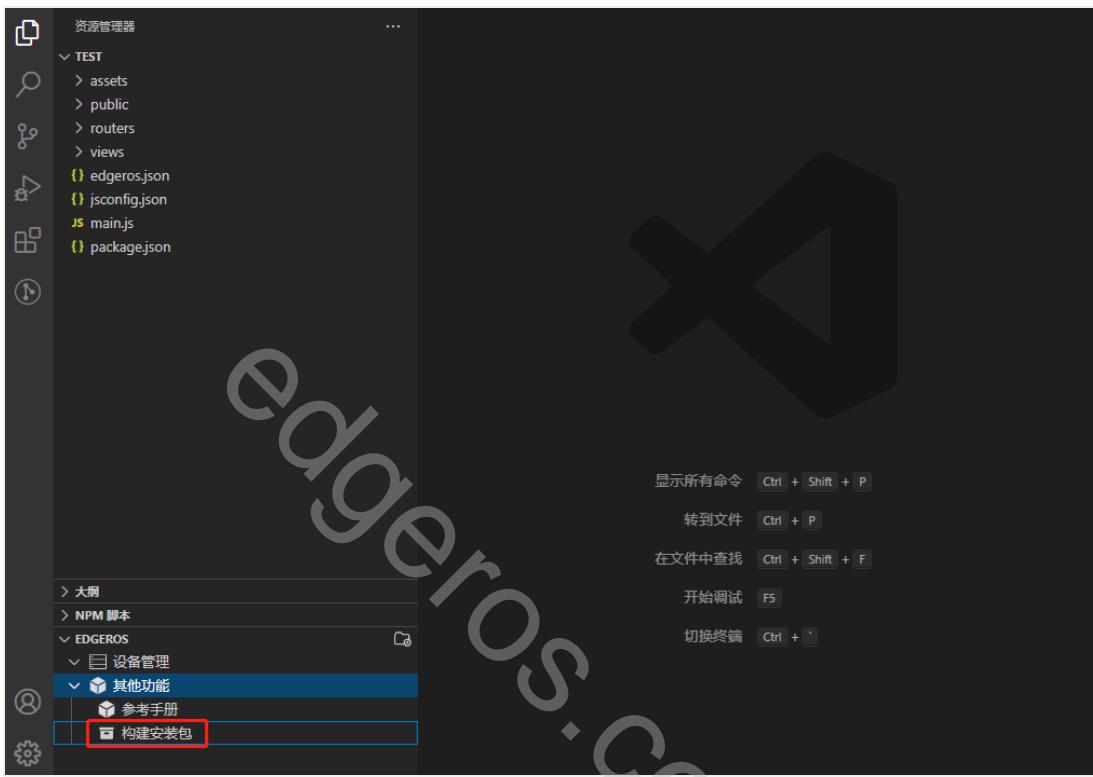
注意：

通过 IDE 安装的 App，享有爱智云远程访问权限，公开组成员除外。如果您的 App 已完成开发和测试，建议您通过爱智开发者平台进行上架。详情请参见[应用管理指南](#)。

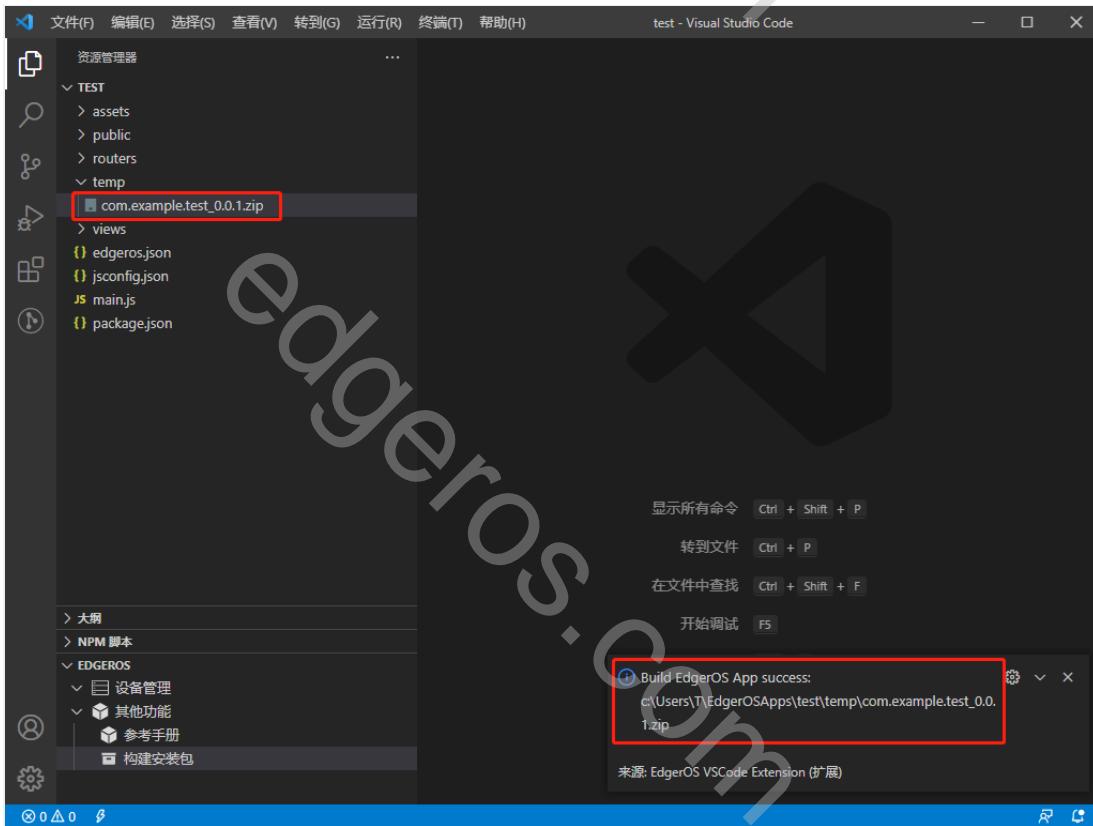
步骤 4：应用打包流程（可选）

如果您的 App 已开发配置完成，需要使用 EdgerOS 扩展插件进行打包，请按照以下步骤操作。

- 在 VSCode 中打开应用，点击**EDGEROS > 其他功能 > 构建安装包**，即可完成打包。

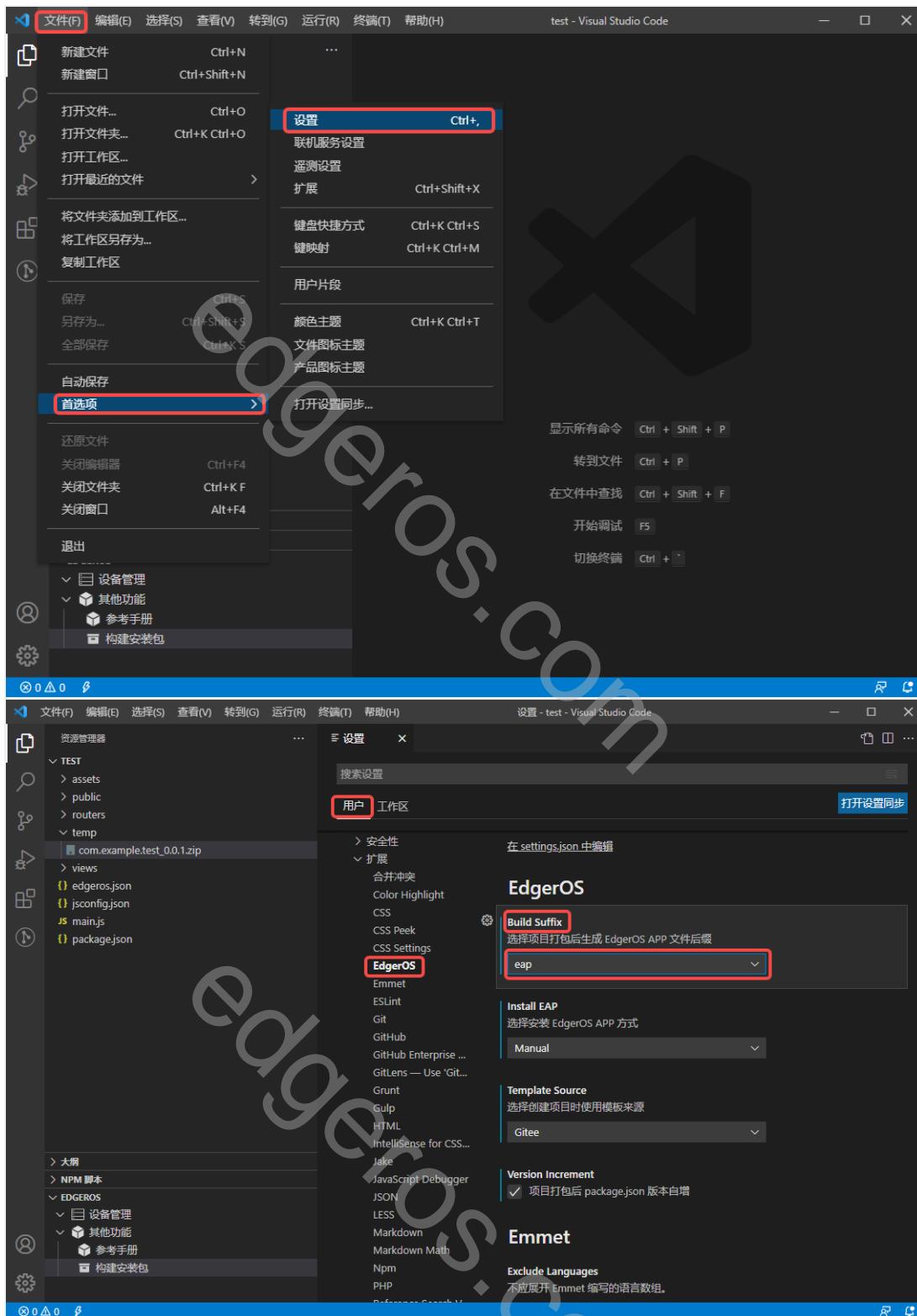


2. 完成打包后在应用项目文件中点击 temp，即可找到应用的包，同时在本地按照右下角提示的路径也可以找到应用的包。



注意：

爱智开发者平台支持上传 eap 和 zip 两种格式的包，默认为 eap 格式。如果要更改插件打包的格式请打开文件>首选项>设置，在设置窗口中点击用户>扩展> **EdgerOS**，打开 EdgerOS 插件的 **Build Suffix** 选项，在下拉框中选择要更改的文件后缀即可。



概述

EdgerOS 在操作系统层面对设备和服务进行抽象和标准化，实现了软硬件的完全解耦，开发者可以使用任意 Web 框架进行前后端分离开发。同时，EdgerOS 拥有功能非常丰富的开发框架，包括 App 框架、流媒体框架、AI 框架、设备管理框架等，让开发者专注于业务功能，提升开发效率。

本章节主要介前后端开发框架和功能模块，为开发者介绍如何构建完整的爱智应用，通过本章节可以学习以下内容：

- 使用 Vue 框架构建一个可以前后端交互的爱智应用
- 集成 EdgerOS 的 Web SDK
- 保障产品接口的安全性
- UI 适配，包括安全区域适配、侧滑操作和应用主题适配
- 爱智应用的性能要求，以及如何优化应用的性能
- 通讯协议接口开发
- 使用 TypeScript 开发爱智应用
- 爱智应用开发常用数据库
- 爱智应用扩展功能模块，包括设备控制、多媒体、AI 和多任务

首个爱智应用示例

本章将以 Vue 为代码示例，介绍如何构建一个可以前后端交互的爱智应用。

概述

开发者在构建应用时可使用任意的 Web 开发框架进行前后端分离开发。前后端分离开发是指将前端页面和后端服务拆开来单独进行开发，全部完成后再联调打包，实现高内聚低耦合，提升研发效率。

Vue 是常用的前端框架之一，阅读本章可了解如下技术：

- 基于 Vue 框架，独立进行爱智前、后端开发。
- 使用现代前端框架的 Proxy 机制进行数据代理，实现可实时预览的 UI 开发。
- 如何将前后端结合为一个可运行在爱智上的应用。

操作步骤

步骤 1：创建爱智应用

参考 [创建应用](#) 章节，创建一个爱智应用，创建好的目录结构如下：

| | |
|------------------|----------------|
| └─ assets | 资源文件夹 |
| └─ routers | 路由信息 |
| └─ public | 静态页面文件 |
| └─ views | 模板页面 |
| └─ eslintrc.json | eslint 配置文件 |
| └─ edgeros.json | edgeros 应用配置文件 |
| └─ main.js | 程序入口 |
| └─ jsconfig.json | 代码补全配置文件 |
| └─ package.json | 依赖包的管理 |

步骤 2：定义 REST

1. 打开已创建的爱智应用工程，在 main.js 文件中添加如下代码：

```
console.log(sys.appinfo().ports);
console.log(sys.appinfo().appid);
console.log(sys.appinfo().port);

let times = 0;
app.get("/times", function(req, res) {
  res.json({ times: times++ });
});

/* Event loop */
require("iosched").forever();
```

注意：请确保 `require('iosched').forever();` 字段处于文件末尾，可参考 [IOSched : I/O events scheduler](#)。

2. 添加完成后，这段代码会返回该接口被调用的总次数，三个 `console.log` 语句会在插件打印该应用启动时的日志，包含以下重要信息：

```
ports - https 端口号
appid - 应用 ID
port - http 端口号
```

步骤 3：部署应用

参考 [部署应用](#) 章节，将应用部署至爱智桌面。

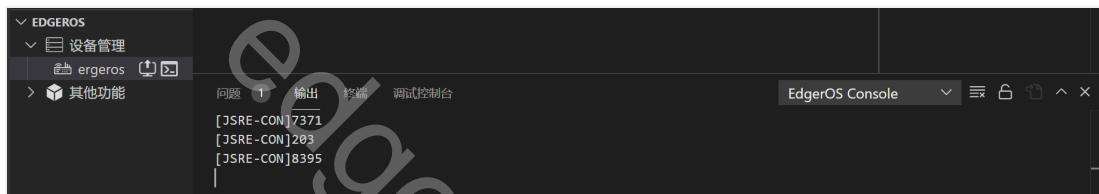
步骤 4：启动应用

1. 部署完成后，点击 **EDGEROS>设备管理>打开控制台输出按钮**，用于获取爱智应用的启动信息。



2. 打开浏览器输入 <https://192.168.128.1> 进入爱智桌面，点击启动刚才安装部署的应用。

3. 查看控制台，爱智应用启动后将输出以下启动信息，应用启动成功。



步骤 5：创建 Vue 应用

1. 请确保您的环境已安装 Node.js 和 Vue 命令行工具(CLI)，之后在控制台或终端执行以下命令，创建一个 Vue 应用。

```
vue create demo-app
```

2. 在 Visual Studio Code 中打开该工程，修改 `src>components>HelloWorld.vue` 文件如下：

```
<template>
<div class="hello">
  <h1>{{ msg }}</h1>
  <h2>Say hello {{ reply.times }} times</h2>
  <button @click="sayHello">Hello EdgerOS</button>
</div>
</template>
```

```

<script>
export default {
  name: "HelloWorld",
  props: {
    msg: String,
  },
  data() {
    return {
      reply: {},
    };
  },
  methods: {
    async sayHello() {
      try {
        const response = await fetch("/times");

        this.reply = await response.json();
      } catch (error) {
        console.error("error is: ", error);
      }
    },
  },
};
</script>

```

3. 修改完成后，这段代码将在界面上增加一个可点击的按钮和一个文本展示区域，文本展示区域的内容由步骤 2 中的 REST 调用结果填充。

步骤 6：连接前后端代码

以上步骤分别创建了前端和后端应用，需要启用 webpack 的 proxy 机制将分离的前后端代码进行连接。对 Vue 来说，需要配置 vue.config.js 文件如下：

```

module.exports = {
  assetsDir: "static",
  productionSourceMap: false,

  devServer: {
    disableHostCheck: true,
    proxy: {
      "/": {
        target: "https://192.168.128.1:7374",
        changeOrigin: true,
        secure: false,
      },
    },
  },
}

```

```
},  
};
```

注意：

- *target* 字段的 URL 指向了爱智应用在爱智设备中的地址，需注意写法。
- URL 必须是 https 格式。
- 应用与爱智必须通过 Wi-Fi 直连，处于开发状态的爱智应用脱离直连模式将无法使用。
- 端口信息来自步骤 4 中控制台的信息。

步骤 7：启动完整应用

1. 在项目控制台输入以下命令，启动完整应用。

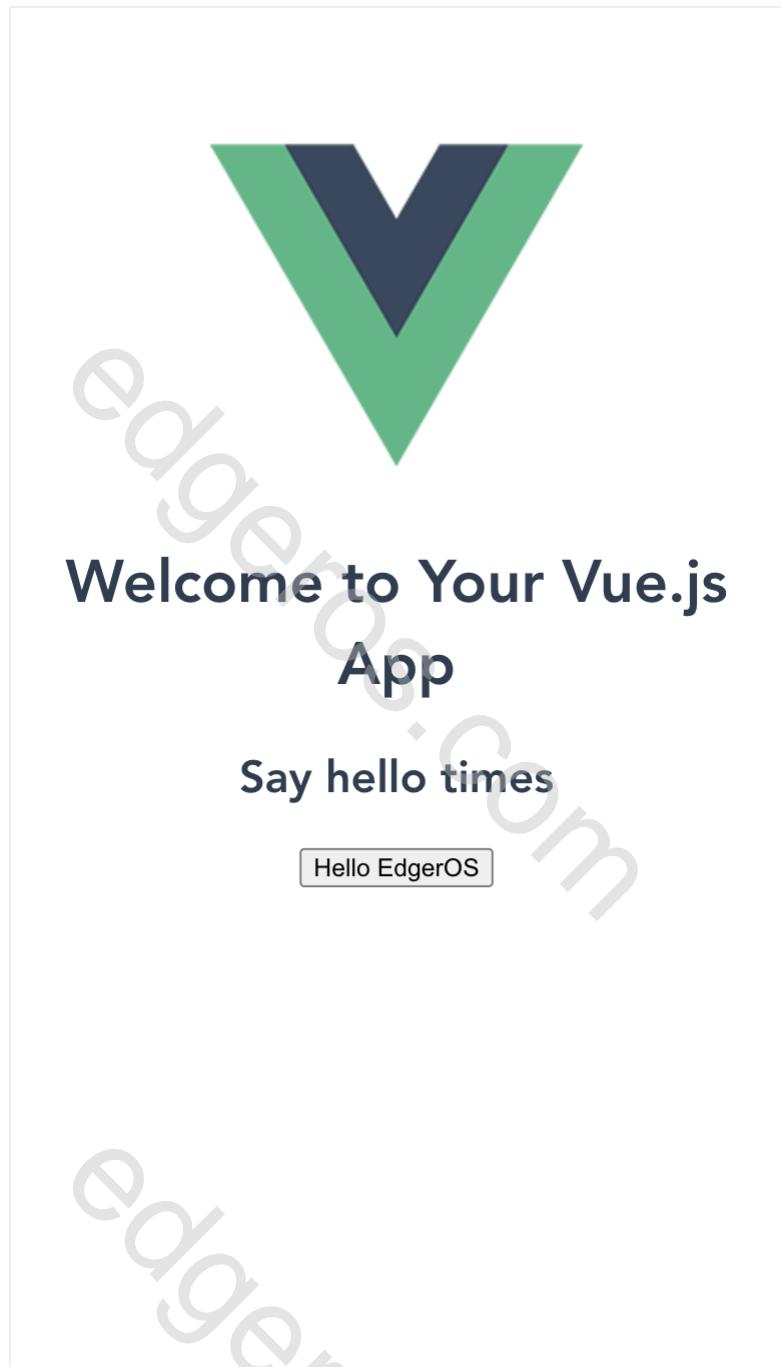
```
npm run serve
```

2. 启动成功后将展示以下输出：

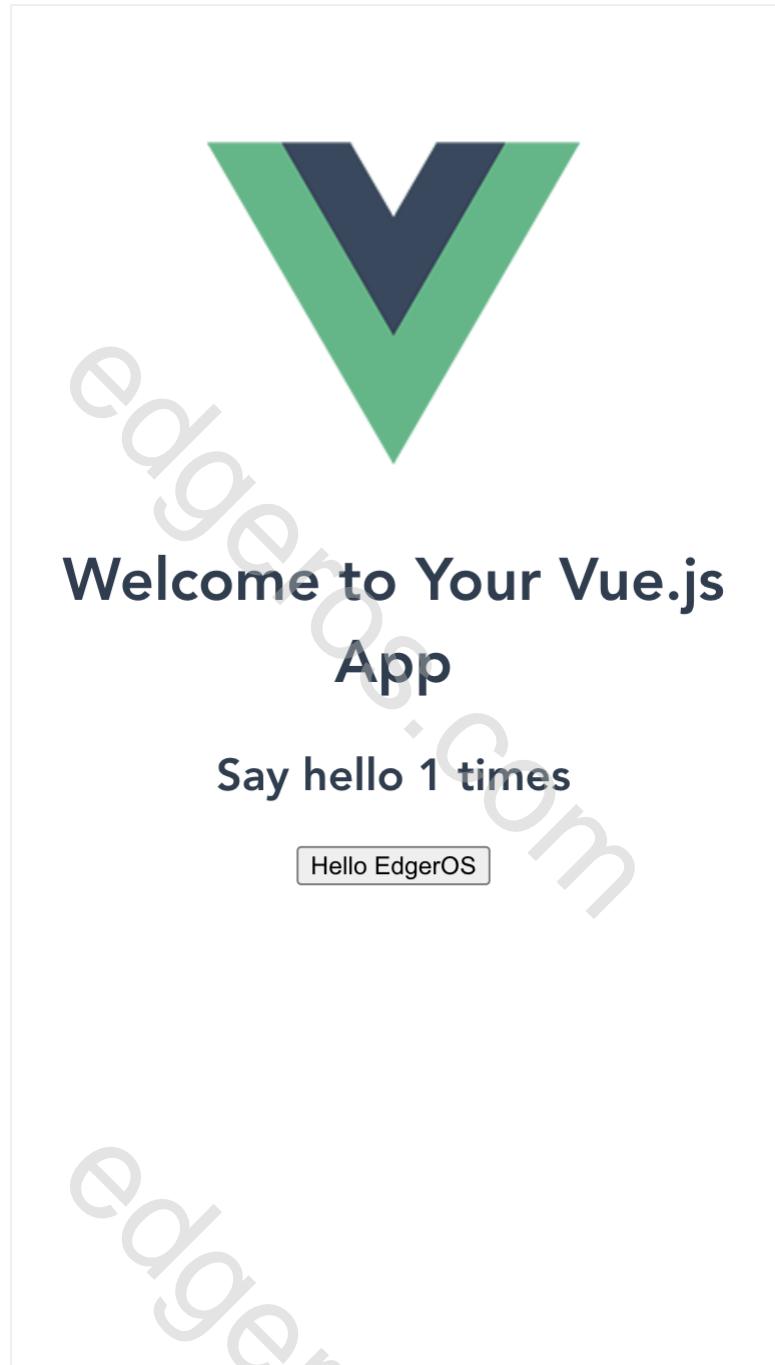


```
DONE Compiled successfully in 191ms
Message (⌘Enter to...)  
v vue.  
1—  
App running at:  
- Local: http://localhost:8080/  
- Network: http://192.168.128.100:8080/
```

3. 通常会有多个 URL 可供使用，需要选择非 localhost 的一项，复制 <http://192.168.128.100:8080> 到浏览器地址栏，进入以下界面：



4. 点击 Hello EdgerOS 按钮，UI 界面将显示按钮被点击的次数，应用创建成功。



步骤 8：发布应用

1. 在 Vue 工程中执行 `npm run build`，构建前端应用。
2. 构建完成后，将构建好的内容从 `dist` 目录拷贝至爱智应用的 `public` 目录下。
3. 参考 [发布应用](#) 章节，完成应用发布。

集成 Web SDK

本章节介绍 Web-SDK 的常用功能和集成方法。

概述

Web-SDK 是 EdgerOS 提供的前端开发工具包，为开发者提供一系列与爱智应用交互的接口，支持 JavaScript 和 TypeScript 语言，更多信息请参考 [Web-SDK API 文档](#)。

使用说明

安装方法

在开发环境中执行以下命令，安装 Web-SDK。

```
npm install @edgeros/web-sdk
```

集成方法

Web-SDK 在 JavaScript 和 TypeScript 中的集成方法如下：

- TypeScript

```
import { edger } from "@edgeros/web-sdk";
```

- JavaScript

```
<script src="\PATH/sdk.min.js"></script>
<script>
  const edger = window.edger
  .....
</script>
```

注意：

若使用 script 标签引入 Web-SDK，edger 对象将会挂载在 window 对象上，调用时需使用 window.edger 参数。

常用功能

Web-SDK 为开发者提供便捷的开发接口，以下列举部分常用功能：

1. 获取用户信息

通过 `edger.user()` 接口，获取当前登录者的翼辉 ID、昵称和头像信息。

```
edger
  .user()
  .then((data) => {
    var { acoid, nickname, profile } = data;
    console.log("User:", acoid, nickname, profile);
  })
  .catch((error) => {
    console.error(error);
  });
}
```

2. 添加扫码功能

调用 API 为爱智应用添加扫码功能。

```
edger.mobile
  .qrscan()
  .then((data) => {
    var { format, text } = data;
  })
  .catch((error) => {
    // Maybe the user cancelled this scan.
    console.error(error);
  });
}
```

3. 权限功能

- **查询权限**

通过 `edger.permission.fetch()` 接口，查询当前用户权限，具体请参考 [权限详情](#)。

```
edger.permission
  .fetch()
  .then((data) => {
    // data contains a complete permission table
    if (data.share) {
      console.log("We have share permission");
    }
    if (data.mediacenter.readable) {
      console.log("We have mediacenter.readable permission");
    }
    if (data.devices.includes("xxxx")) {
      console.log("We have device xxx permission");
    }
  })
  .catch((error) => {
    console.error(error);
  });
}
```

- 申请权限

以下代码将在应用界面弹出权限申请对话框，要求使用者赋予对应权限。

```
edger.permission
  .request({
    code: ["network"],
    type: "permissions",
  })
  .then((data) => {
    // data.success is 'true' means pops up successfully
  })
  .catch((error) => {
    console.error(error);
  });
}
```

- 权限变更

监听当前权限，在权限发生变化时触发此事件，以下两种代码方式均可实现：

```
edger.onAction("permission", (data) => {
  // This data is same with edger.permission.fetch()
  if (data.share) {
    console.log("We have share permission");
  }
});
```

```
edger.addEventListener("permission", (data) => {
  // This data is same with edger.permission.fetch()
  if (data.share) {
    console.log("We have share permission");
  }
});
```

完成服务端认证

本章主要介绍服务端如何通过 *RESTful* 接口与爱智进行安全通信。

概述

爱智为通信安全提供了基于 Token 和随机数 Strand 的双重保障，两项验证均通过才能完成通信。随机数以分钟级进行刷新，过期随机数无法通过安全验证。

使用说明

获取 Token 和 Strand

爱智提供以下两种方式获取 Token 与 Strand：

- 主动获取

```
import { edger } from "@edgeros/web-sdk";
import { tokenRef } from "@/service/api";

// 主动获取 token 并设置
edger
  .token()
  .then((data) => {
    const { token, strand } = data;
    tokenRef.token = token;
    tokenRef.strand = strand;
  })
  .catch((error) => {
    console.log(error);
  });

```

- 被动获取

```
import { edger } from "@edgeros/web-sdk";
import { tokenRef } from "@/service/api";

// 监听 token 变化并更新
edger.onAction("token", (data) => {
  const { token, strand } = data;
  tokenRef.token = token;
  tokenRef.strand = strand;
});
```

集成 Token 和 Srand

下面以 Vue 2 + Typescript 开发为例介绍 REST API 集成 Token 和 Srand 的方法：

1. 在开发环境执行以下命令，安装 @edgeros/web-sdk 包。

```
npm install @edgeros/web-sdk
```

2. 在 src 目录下创建服务文件夹 src/service，并在 service 文件下创建文件 api.ts 和 edger.ts。

3. 参考以下示例，在 api.ts 中创建 Token 和 Srand 变量并导出。

```
//api.ts 示例代码
export const tokenRef = { token: "", srand: "" };
```

4. 参考以下示例，在 Vue 工程下的 main.ts(/项目名/src/main.ts) 文件中引入 @edgeros/web-sdk。

```
//main.ts 示例代码
import { edger } from "@edgeros/web-sdk";
import { tokenRef } from "@/service/api";

// 主动获取 token 并设置
edger.token().then((data) => {
  const { token, srand } = data;
  tokenRef.token = token;
  tokenRef.srand = srand;
});

// 监听 token 变化并更新
edger.onAction("token", (data) => {
  const { token, srand } = data;
  tokenRef.token = token;
  tokenRef.srand = srand;
});
```

5. 参考以下示例，将 api.ts 引入到 edger.ts 中，读取 Token 和 Srand 应用到请求头的配置中。

```
//edger.ts 示例代码
import axios from "axios";
import { tokenRef } from "./api";

const axiosConfig = () => {
  return {
    // headers 中的 'edger-token' 和 'edger-srand' 为平台约定必传字段
```

```
headers: {
    "edger-token": tokenRef.token,
    "edger-srand": tokenRef.srand,
},
};

export default {
    /**
     * get 请求示例
     */
    getAppInfo(): Promise<ApiResult> {
        return axios.get("/app/get/info", { ...axiosConfig() });
    },
    /**
     * post 请求示例
     */
    updateAppInfo(name: string): Promise<ApiResult> {
        return axios.post("/app/post/info", { name }, { ...axiosConfig() });
    },
};
```

开发 RESTful

本章介绍 RESTful 接口的安装导入和使用方法。

概述

EdgerOS 后端程序是基于 RESTful 风格的 Web 应用程序框架构建的，支持 GET、POST、DELETE、HEAD、PUT 等各种请求方法，以下介绍 RESTful 的安装导入和请求方法。

安装导入

前端技术种类众多，开发者需安装相对应的 HTTP 插件。本章以 Vue 的 Axios 插件为例，通过 script 标签引入静态 js 文件。

1. 从 [vue.min.js](#) 获取 vue.min.js，从 [axios.min.js](#) 获取 axios.min.js。
2. 参考以下示例，导入 vue.js 和 vue.js axios。

```
<!--引入vue.js-->
<script src="./vue/vue.min.js"></script>
<!--引入vue.js axios插件-->
<script src="./vue/axios.min.js"></script>
```

请求方法

本章以 GET 和 POST 为例，介绍 RESTful 的请求方法。

GET 请求

1. 请参考以下示例，查询用户信息列表。

```
getUsers: function () {
  const auth = {
    'edger-token': this.token,
    'edger-srand': this.srand
  };
  axios
    .get('/api/user', {}, {headers: auth})
    .then(res =>{
      console.log(res.body);
    })
    .catch(function (error) {
      console.log(error);
    });
}
```

2. 请参考以下示例，在后端处理 GET 请求。

```
router.get('/api/user', function (req, res) {
  const userInfo = req.eos.user // 提取用户信息
  console.log('/api/user', userInfo.acoid)

  res.json({
    result: true,
    message: 'success',
    data: users
  })
})
```

POST 请求

1. 请参考以下示例，在前端通过 post 请求，提交用户填写的个人信息。

```
addUser: function () {
  const auth = {
    'edger-token': this.token,
    'edger-srand': this.srand
  };
  axios
    .post('/api/user', { name: this.name, phone: this.phone }, {headers: auth})
    .then(res => {
      console.log(res.body);
    })
    .catch(function (error) {
      console.log(error);
    });
}
```

2. 请参考以下示例，在后端处理 POST 请求。

```
router.post("/api/user", function(req, res) {
  // 数据库或其他操作
  res.json({
    result: true,
    message: "success",
  });
})
```

安全性说明

如果需要对某个 API 进行安全保护，可以在声明该 API 时增加前缀“/api”，以增加 API 的安全性。

缓存能力说明

爱智 App 会为爱智应用程序提供缓存能力，静态文件资源请求可以直接从缓存中获取，为了保证你的 REST API 的 GET 请求不会被缓存，你可以遵守以下任意的规则：

- 以“/api”开头。
- REST 的请求头包含 `edger-token` 和 `edger-srand`。
- REST 的请求头包含 `content-type`。

示例

```
// Server
const fs = require('fs')
const WebApp = require('webapp');
const app = WebApp.createApp();

app.use(WebApp.static('./public'));
// 通过'/api/image'获取的资源不会被缓存
app.use('/api/image', (req,res)=>{
    const imgStream=fs.createReadStream('api_image.png')
    imgStream.pipe(res)
});

app.start();

require('iosched').forever();

// Client
// 以下资源请求均不会被缓存
fetch('/images/demo.jpg', {
    headers: { 'edger-token': 'xxx', 'edger-srand': 'xxx' }
});

fetch('/images/demo.jpg', {
    headers: { 'content-type': 'image/jpg' }
});

fetch('/api/image');
```

userInfo 数据结构

| 字段名称 | 类型 | 说明 |
|----------|--------|---------------|
| nickname | string | 用户昵称（默认为手机号码） |

| | | |
|-------|--------|--------|
| exp | number | 用户过期时间 |
| acoid | string | 翼辉 ID |

edgeros.com

优化 UI 展示效果

本章主要介绍如何优化爱智应用的 UI 展示效果。

概述

为了优化爱智应用的 UI 展示效果，提升用户使用体验，需要对爱智应用进行 UI 适配。本章为您介绍以下内容：

- 安全区域适配
- 侧滑操作适配
- 应用主题适配

安全区域适配

安全区域说明

安全区域是指一个可视窗口范围，处于安全区域的内容不受圆角、齐刘海、黑边框等影响。



以 iOS 设备为例，下图中右图未添加安全区域适配，显示的内容区域将与顶部（状态栏）或底部（操作条）的显示区域重叠，影响使用体验。左图添加了安全区适配的显示，使用体验良好。



适配方法

1. 页面布局设置

- 设置页面在可视窗口的布局方式

在 index 文件的代码头部新增 viewport-fit 属性，页面内容将完全覆盖整个窗口。

```
<meta name="viewport" content="width=device-width, viewport-fit=cover"
/>
```

- 设置页面主体内容限定在安全区域内

在 CSS 文件中添加 --edger-safe-area-inset-top 和 --edger-safe-area-inset-bottom 的显示支持，页面主体内容将限定在安全区域内。如果不设置这个值，可能存在小黑条遮挡页面最底部内容的情况。

3. 在开发环境执行以下命令，安装 @edger/web-sdk。

```
npm install @edgeros/web-sdk
```

4. 参考以下示例，在项目中引入 @edger/web-sdk。

```
import { edger } from "@edgeros/web-sdk";

// 监听屏幕方向的变化，动态修改安全区的属性值
edger.onAction("orientation", (data) => {
    // 屏幕方向变化，执行 `edger.layout.safeArea` 方法
    edger.layout.safeArea();
    console.log("onAction Current orientation111:", data);
});
```

5. 参考以下示例，在 CSS 文件中使用 @edger/web-sdk 配置安全区属性。

```
<style>
.page {
    ...
    bottom: 50px(假设值);
    padding-top: 0;
    ...
    /* 适配安全区 */
    bottom: calc(50px(假设值) + var(--edger-safe-area-inset-bottom));
    padding-top: var(--edger-safe-area-inset-top);
}
</style>
```

侧滑适配

侧滑说明

随着全面屏手机的普及，物理返回按键逐渐消失，侧滑操作为用户带来了更好的使用体验，尤其对于习惯单手操作用户，侧滑操作已经成为手机应用必备的配置之一。

适配方法

爱智提供以下两种方式适配侧滑功能：

方式一

1. 在开发环境执行以下命令，安装 @edger/web-sdk。

```
npm install @edgeros/web-sdk
```

2. 参考以下示例，在开发单页面应用时：初始化完成后调用

```
edger.enableHistoryNavigation()，实现 history 模式的页面回退；
```

```
import { edger } from "@edgeros/web-sdk";

edger.enableHistoryNavigation();
```

3. 参考以下示例，在开发多页面应用时：在每个页面中调用

```
edger.enableHistoryNavigation()，实现 history 模式的页面回退。
```

注意：需要在项目的 index.html 里进行首页重定向。

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>EdgerOS App</title>
  </head>
  <body></body>
  <script>
    window.location.href = "./html/index.html"; // 自己定义的项目的首页
  </script>
</html>
```

方式二

本方式以 Vue 项目为例：

1. 在开发环境执行以下命令，安装 @edgeros/web-sdk。

```
npm install @edgeros/web-sdk
```

2. 参考以下示例，在路由配置文件中需要增加额外参数 `meta: { index: number }`，以供滑动触发路由跳转。

```
import Vue from 'vue';
import VueRouter, { RouteConfig } from 'vue-router';
import Index from '@/views/Index.vue';
import Page1 from '@/views/Page1.vue';
import Page2 from '@/views/Page2.vue';

Vue.use(VueRouter);

const routes: Array<RouteConfig> = [
  {
    path: '/',
    component: Index,
    children: [
      ...
      {
        path: 'page1',
        name: 'page1',
        meta: { index: 10 // 假设值 },
        component: Page1
      },
      {
        path: 'page1',
        name: 'page2',
        meta: { index: 20 // 假设值 },
        component: Page2
      }
      ...
    ]
  }
]

const router = new VueRouter({ routes });

export default router;
```

3. 参考以下示例，在 Vue 工程下的 App.vue 文件中调用 `onAction('swipe')` 监听滑动事件，开启滑动事件监听，启用侧滑触发路由跳转。

```
//App.vue 示例代码
<template>
  <div id="index-page">
    <router-view />
  </div>
</template>

<script lang="ts">
import { Component, Vue, Watch } from "vue-property-decorator";
import { Route } from "vue-router";
import { edger } from "@edgeros/web-sdk";

interface HistoryPath {
  index: number;
  fullPath?: string;
}

@Component()
export default class App extends Vue {
  private historyRouter: Array<HistoryPath> = [];
  get routeChange() {
    return this.$route;
  }

  // 监听路由变化，更新路由存储数据
  @Watch("routeChange") private onRouteChange(to: Route, from: Route) {
    const { fullPath, meta } = from;
    if (to.meta.index > from.meta.index) {
      this.historyRouter.push({ fullPath, index: meta.index });
    } else {
      this.historyRouter.pop();
    }
  }

  created() {
    edger.onAction("swipe", (data) => {
      if (data.payload.direction === "left") {
        this.pathBackward();
      }
    });
  }

  private pathBackward() {
    if (this.historyRouter.length) {
      const { fullPath } = this.historyRouter[this.historyRouter.length - 1];
      this.$router.push({ path: fullPath });
    }
  }
}
```

```

        }
    }
}

</script>

```

关闭侧滑

当侧滑功能与其他页面滑动操作冲突时，请保持默认不开启侧滑状态，确保应用能够正常使用。

应用主题适配

应用主题说明

爱智应用通过 CSS 的自定义属性实现应用主题适配。CSS 自定义属性由 CSS 作者定义，包含的值可以在整个代码中重复使用。自定义属性受级联约束，从父级继承，是实现主题切换的关键步骤。

- CSS 属性的基本用法

声明一个自定义属性，属性名需要以两个减号 “--” 开始，属性值则可以是任何有效的 CSS 值。和其他属性一样，自定义属性也是写在规则集之内的，示例如下：

```

element {
    --main-bg-color: brown;
}

```

- CSS 自定义属性的继承性

自定义属性会继承，如果一个给定的元素上没有设置自定义属性的值，该元素将继承其父元素的 CSS 属性。

适配方法

1. 在开发环境执行以下命令，安装 @edgeros/web-sdk。

```
npm install @edgeros/web-sdk
```

2. 参考以下示例，在 main.ts 中监听 theme 事件，实现深浅色变量设置。

```

import { edger } from "@edgeros/web-sdk";

edger.onAction("theme", (data: any) => {
    if (data.payload === "light") {
        document.body.style.setProperty("--priamryColor", "#333");
        document.body.style.setProperty("--priamryBgColor", "#FF9300");
    } else {
        document.body.style.setProperty("--priamryColor", "#eee");
        document.body.style.setProperty("--priamryBgColor", "#FD9507");
    }
});

```

```
    }  
});
```

3. 参考以下示例，在 CSS 中使用定义的变量。

```
<style>  
.box {  
    ...  
    background-color: var(--priamryBgColor);  
    color: var(--priamryColor);  
    ...  
}  
</style>
```

4. 按照以上步骤，即可配置深浅色主题的显示模式。





优化性能

本章主要介绍应用的性能要求，以及如何优化应用的性能。

基本要求

- 首屏加载时间不超过 15 秒

为了提高用户体验，爱智要求每个 App 启动时从加载资源到首屏显示不能超过 15 秒，如果超时系统会弹出提醒。如果应用的启动时间过长，需要优化应用的首屏加载时间，以保证 App 的正常启动。

- 支持遗嘱权限

App 申请遗嘱 (will) 权限后，当用户关闭该 App 时，系统不会立即关闭该应用，而是会保留一些时间处理任务，开发者需要保证尽可能快的进行数据保存，否则进程将强制结束。

优化方法

开发者可以通过以下两种方式，提高首屏响应速度：

- 基础优化：优化基础代码，减少影响首屏响应的因素。
- 项目优化：通过项目优化提高首屏的响应速度，保证项目正常启动，提高用户体验。

基础优化

1. 参考以下示例，使用 link 标签将样式表放在 html 文件的 head 中，可以有效避免白屏和无样式内容的闪烁。

```
<head>
  <link rel="stylesheet" href="example.css" />
</head>
```

2. 参考以下示例，将脚本放在 html 文件的底部，这样不会阻塞页面内容的呈现，而且页面中的可视组件可以尽早下载。

```
<body>
  <!-- 将脚本放在底部 -->
  <script src="example.js"></script>
</body>
```

注意：

将脚本放在顶部会造成的影响：脚本会阻塞对其后面内容的显示和组件的下载。

3. 使用矢量图标，相较于图片而言资源占用更小，如果你的项目里有小图标，请使用矢量图标。

说明:

具体的使用细节请参考 [使用帮助](#)。

4. 对代码进行压缩，例如：使用 UglifyJS 进行代码压缩，UglifyJS 是一个集 js 解释器、最小化器、压缩器和美化器的工具集。

5. CDN 缓存优化

CDN 指的是一组分布在各个地区的服务器。这些服务器存储着数据的副本，因此服务器可以根据哪些服务器与用户距离最近，来满足数据的请求。CDN 提供快速服务，较少受高流量影响。相较于其他的缓存（优化页面流畅程度），CDN 缓存更多的是为了优化首屏加载速度。我们建议开发者提供本地资源包，以便 App 离线工作。

项目优化

利用 PWA 技术优化项目代码，可以有效降低首屏响应时间。PWA 全称 Progressive Web App，即渐进式 Web 应用。一个 PWA 应用首先是一个页面，可以通过 Web 技术编写出一个页面应用，随后添加 App Manifest 和 Service Worker 来实现 PWA 的安装和离线等功能。Service Worker 是为解决“Web App 的用户体验不如 Native App”的普遍问题而提供的一系列技术集合，Service Worker 是 PWA 的核心技术，它能够为 web 应用提供离线缓存功能，下面列举了一些 Service Worker 的特性：

- 基于 HTTPS 环境，这是构建 PWA 的硬性前提
- 是一个独立的 worker 线程，独立于当前页面进程，有自己独立的 worker context
- 可拦截 HTTP 请求和响应，可缓存文件，缓存的文件可以在网络离线状态时取到
- 能向客户端推送消息
- 不能直接操作 DOM
- 异步实现，内部大都是通过 Promise 实现

若要在代码中使用 PWA 进行项目优化，请参考以下链接：

[Vue 兼容 PWA](#)

[React 兼容 PWA](#)

[Workbox](#)

注意：

- 客户端通过爱智云向爱智设备上传文件时，文件大小不能超过 16 M。
- 客户端本地直连爱智设备上传文件时，不限制文件大小。

上传文件

本文介绍如何通过爱智 App 将本地图片、视频、音频等文件上传到爱智。

概述

文件上传应用非常广泛，大部分网站或应用都有文件上传功能。文件上传的场景包括单文件上传、多文件上传、目录上传和服务端上传等，通过文件上传，我们可以将本地的资源文件发送到服务器上，有效地实现资源共享和信息通讯。

本文介绍如何通过爱智 App 上传文件到爱智。在爱智 App 中，目前支持 Ajax 和 Fetch 两种文件上传方式。

注意：

爱智 App 中暂不支持 Form 表单提交方式。

上传案例

参考以下示例，使用 Ajax 和 Fetch 方式上传文件到爱智。

爱智应用前端部分

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>EdgerOS Upload File</title>
</head>
<body style="text-align: center;">
    <input type="file" id="file">
    <button type="button" onclick="fetchUpload()">FETCH</button>
    <button type="button" onclick="ajaxUpload()">AJAX</button>
</body>
<script type="text/javascript">
    /**
     * fetch upload file
     */
    async function fetchUpload() {
        const form = _createFormData()
        try {
            console.log('fetch upload start')
            const response = await fetch('/api/upload', {
                method: 'POST',
                body: form,
            })
            console.log('fetch upload success:', response.status);
        }
    }
</script>
```

```

    } catch (error) {
        console.error('Error:', error);
    }

}

/**
 * ajax upload file
 */
function ajaxUpload() {
    const form = _createFormData()
    const xhr = new XMLHttpRequest()
    xhr.open('post', '/api/upload', true)
    xhr.onload = (evt) => console.log('ajax upload
success:', evt.target.responseText)
    xhr.onerror = (evt) => console.error(evt.target.responseText)
    xhr.upload.onprogress = (evt) => console.log('ajax upload
progress:', Math.round(evt.loaded / evt.total * 100) + "%")
    xhr.upload.onloadstart = (evt) => console.log('ajax upload start')
    xhr.send(form)
}

/**
 * create FormData
 */
function _createFormData() {
    const fileInfo = document.getElementById("file").files[0]
    if (!fileInfo) {
        console.log('No selection file');
        throw new Error('No selection file')
    }
    const form = new FormData()
    form.append('filename', fileInfo.name)
    form.append('file', fileInfo)
    return form
}
</script>
</html>

```

爱智应用后端部分

```

/* Import system modules */
const WebApp = require('webapp');
const multer = require('middleware').multer;

/* Create App */
const app = WebApp.createApp();

```

```
/* Set static path */
app.use(WebApp.static('./public'));

/* Set upload path */
const storage = multer.diskStorage({
  destination: '/file-upload',
  filename: (req, file, cb) => cb(null, Date.now() + '-' +
req.body.filename)
});
const upload = multer({ storage: storage });
app.post('/api/upload', upload.single('file'), (req, res) => {
  console.log('uploaded file saved: ', req.file.path, req.file.size)
  res.json({ errcode: 0, errmsg: 'ok', })
})

/* Start App */
app.start();
/* Event loop */
require('iosched').forever();
```

无论是使用 Ajax 还是 Fetch，都能将文件上传到爱智。其中 Ajax 能够快速地将增量更新呈现在用户界面上，无需刷新整个页面；Fetch 采用 Promise 的异步处理机制，提供了丰富的 API，写法和使用更加简单。

服务端安全认证

本章主要介绍服务端通过 Socket.IO 和 WebSyncTable 接口与爱智通信时如何进行安全认证。

概述

为保证通信安全，爱智提供了基于 Token 和随机数 Strand 的双重保障，两项认证均通过才能完成通信。随机数以分钟级进行刷新，过期随机数无法通过安全验证。

获取 Token 和 Strand

具体请参见 [获取 Token 和 Strand](#)。

集成 Token 与 Strand

不同通讯协议的集成方法如下：

Socket.IO 集成 Token 和 Strand

- 在开发环境执行以下命令，安装 socket.io-client 和 vue-socket.io-extended 包。

```
npm install socket.io-client vue-socket.io-extended
```

- 参考以下示例，在 src 目录下创建 socket.ts 文件，引入安装包，编写 socket.io 的相关代码。

```
//示例代码：示例代码实现了 socket.io 在异常情况下的重试机制
import Vue from "vue";
import VueSocketIOExt from "vue-socket.io-extended";
import SocketIO from "socket.io-client";
import { tokenRef } from "./service/api";

export let socket!: SocketIOClient.Socket;

const addListener = (
  client: SocketIOClient.Socket,
  evt: string,
  fn: () => void
) => {
  if (!client.hasListeners(evt)) {
    client.on(evt, fn);
  }
};

export const initSocket = (data: TokenResponse) => {
  // 创建 socketio 连接
}
```

```

socket = SocketIO({
  query: { "edger-token": data.token, "edger-srand": data.srand },
  transports: ["websocket"],
});

const errorHandler = () => {
  // 如果 token 或 srand 不存在, 则终止 socketio 连接
  if (!tokenRef.token || !tokenRef.srand) {
    return;
  }
  socket.io.opts.query = {
    ...socket.io.opts.query,
    "edger-token": tokenRef.token,
    "edger-srand": tokenRef.srand,
  } as Record<string, any>;
};

// 异常断联后, 重新尝试 socket 连接
addListener(socket, "error", errorHandler);
addListener(socket, "connect_error", errorHandler);
addListener(socket, "connect_timeout", errorHandler);
addListener(socket, "reconnect_error", errorHandler);
addListener(socket, "reconnect_failed", errorHandler);
addListener(socket, "disconnect", errorHandler);

Vue.use(VueSocketIOExt, socket);
};

```

3. 参考以下示例，在 main.ts 文件中引入 socket.ts 文件，并初始化 socket.io 连接。

```

//main.ts 示例代码
...
import { initSocket } from './socket'

const updateToken = (data: TokenResponse) => {
  if (data) {
    initSocket(data)
  }
}
...

```

WebSyncTable 集成 Token 和 Strand

1. 在开发环境执行以下命令，安装 @edgeros/synctable 包。

```
npm install @edgeros/web-synctable
```

2. 参考以下示例，在需要使用 SyncTable 的文件中集成 Token 和 Srand。

```
//示例代码
<template>
...
</template>

<script lang='ts'>
import { Component, Vue } from 'vue-property-decorator';
import { tokenRef } from '@/service/api';
import { SyncTable } from '@edgeros/web-synctable';

@Component({
  components: {}
})
export default class TestApp extends Vue {

  private table: any

  // 实例创建完成之后创建 SyncTable 连接
  created () {
    if (!this.table) {
      this.connectSyncTable()
    }
  }

  // 实例销毁后断开 SyncTable 连接
  destroyed () {
    this.disconnectSyncTable()
  }

  private connectSyncTable () {
    const signboard = 'xxx.xxx' // SyncTable 表名
    // 创建的 SyncTable 是一个类似 Map 的对象，后续操作可以通过后台约定的 key 值获取对应数据
    const table = new window.SyncTable(`wss://${window.location.host}`,
signboard, {token: tokenRef.token, srand: tokenRef.srand});
    this.table = table
    table.addEventListener('update', (key: string, value: number[]) => {
      // 在监听到更新后，对数据进行一些处理
    })
  }

  private disconnectSyncTable () {
    if (this.table) {
      this.table.close()
      this.table = undefined
    }
  }
}
```

```
    }
}
</script>
```

edgeros.com

edgeros.com

开发 Socket.IO

本章介绍 Socket.IO 接口的安装导入和使用方法。

概述

Socket.IO 在客户端和服务器之间传递事件，实现低延迟的双向通信，以下介绍 Socket.IO 的安装导入和使用步骤。

安装导入

1. 以 Vue 为例，从 [vue.min.js](#) 获取 vue.min.js，从 [socket.io.js](#) 获取 socket.io.js。
2. 参考以下示例，在前端引入 vue.js 和 Socket.IO。

```
<!-- 引入vue-->
<script src=".//vue/vue.min.js"></script>
<!-- 引入socketio client-->
<script src=".//socketio/socket.io.js"></script>
```

说明：

Socket.IO 使用请参考 [Socket.IO](#)。

3. 参考以下示例，在后端导入 Socket.IO。

```
var io = require("socket.io");
```

注意：

JSRE 目前支持的 Socket.IO 是 v2.2 版本，socket.io-client 需要安装 v2.x 版本。

操作步骤

以下介绍 Socket.IO 的使用步骤。

步骤 1：创建 Socket.IO 连接

1. 参考以下示例，创建前端连接。

```
const auth = {
  "edger-token": this.token,
  "edger-srand": this.srand,
};

this.socket = io({
```

```
    query: auth,
}) ;
```

2. 参考以下示例，创建后端服务，并监听已连接事件。

```
var socketio = io(app, {
  serveClient: false,
  pingInterval: 10000,
  pingTimeout: 5000,
}) ;
```

参数说明

| 参数 | 说明 | 取值样例 |
|--------------|----------------------------|-------|
| app | 需要绑定的服务器 | - |
| serveClient | 是否提供客户端文件 | false |
| pingTimeout | 没有 ping 数据包需要多少毫秒才能认为连接已关闭 | 60000 |
| pingInterval | 发送新的 ping 数据包前需要多少毫秒 | 25000 |

步骤 2：监听 Socket.IO 连接事件

1. 参考以下示例，在前端监听连接事件，连接事件中可以进行一些初始化操作，以及监听用户自定义事件。

```
this.socket.on("connect", () => {
  console.log("已连接！");
}) ;
```

2. 参考以下示例，在后端监听前端用户请求事件。

```
socketio.on("connection", function(socket) {
  // ...
}) ;
```

步骤 3：收发消息

1. 参考以下示例，前端可以使用 `socket.emit()` 方法向后端发送消息。

```
send: function() {
  this.socket.emit('message', this.message, (response) => {
```

```

    this.response = response;
  });
}

```

说明：

- *message* 是用户自定义事件。
- *this.message* 是事件参数，*emit* 事件参数可以是不定参数。
- *emit* 最后一个参数如果是函数，则表示该事件将等待对方回复消息。

2. 参考以下示例，在后端监听事件接收消息，`socket.on()` 参数与前端程序参数保持对应。

```

socketio.on('connection', function (socket) {
  const userInfo = socket.handshake.eos.user // 提取用户信息

  socket.on('message', (msg, callback) => {
    callback(`server: ${msg}`)
  })
})

```

userInfo 数据结构

| 字段名称 | 类型 | 说明 |
|----------|--------|---------------|
| nickname | string | 用户昵称（默认为手机号码） |
| exp | number | 用户过期时间 |
| acoid | string | 翼辉 ID |

开发 WebSyncTable

本章介绍 *WebSyncTable* 的安装导入和使用步骤。

概述

WebSyncTable 模块是专为 *SyncTable* 服务的，它在内部封装了 *WebSocket* 用于客户端服务端的通信，主要用于完成客户端和服务端的本地表数据同步，以下介绍 *WebSyncTable* 的安装导入和操作方法。

安装导入

1. 以 Vue 为例，从 `vue.min.js` 获取 `vue.min.js`，从 `synctable.min.js` 获取 `synctable.min.js`。
2. 参考以下示例，在前端通过 `script` 标签引入 `vue.js` 和 *WebSyncTable*。

```
<!-- 引入vue-->
<script src="./vue/vue.min.js"></script>
<!-- 引入webSyncTable-->
<script src="./synctable/synctable.min.js"></script>
```

3. 参考以下示例，在后端引入 *SyncTable* 和 *WebSyncTable*。*SyncTable* 用于创建本地同步表，*WebSyncTable* 用于客户端服务端的表同步。

```
var WebSyncTable = require("websynctable");
var SyncTable = require("synctable");
```

操作步骤

以下介绍 *WebSyncTable* 的使用步骤。

步骤 1：创建同步表

1. 参考以下示例，在前端获取协议类型拼接服务端访问地址，创建一个名为 `table1` 的表。

```
var proto = location.protocol === "http:" ? "ws:" : "wss:";
var server = `${proto}//${window.location.host}`;
const auth = {
  token: this.token,
  srand: this.srand,
};
this.table = new SyncTable(server, "table1", auth);
```

2. 参考以下示例，在后端同步创建一个与客户端相同的 *SyncTable*，然后创建 *WebSyncTable* 同步前后端的表。

```

const table = new SyncTable('table1')
// Create SyncTable server
const server = new WebSyncTable(app, table, {
    onclient: function (event, client) {
        const userInfo = client.eos.user // 提取用户信息
        return true
    }
})

```

步骤 2：添加变量

参考以下示例，在前端通过 `table.set()` 方法向表中设置属性，后端会自动同步数据，服务也会有一份相同的数据。

```

this.table
.set(this.key, this.value)
.then((v) => {
    console.log("ok");
})
.catch((e) => {
    console.error("set error!");
});

```

步骤 3：获取变量

参考以下示例，在前端获取变量。此方法会优先从本地表中拿数据，所以不会去请求后端，这样可以节省网络资源。

```

this.table
.get(this.key)
.then((value) => {
    console.log("k1 value is:", value);
    this.value = value;
})
.catch((error) => {
    console.error("get k1 value error!");
});

```

userInfo 数据结构

| 字段名称 | 类型 | 说明 |
|----------|--------|---------------|
| nickname | string | 用户昵称（默认为手机号码） |

| | | |
|-------|--------|--------|
| exp | number | 用户过期时间 |
| acoid | string | 翼辉 ID |

edgeros.com

SQLite3

本章主要介绍 SQLite3 的使用方法。

概述

SQLite3 是一种轻量型、进程内的关系型数据库，是 EdgerOS 内置的一种自包含的、无服务器的、零配置的事务性 SQL 数据库引擎。

使用方法

1. 使用以下命令导入 SQLite3 数据库。

```
var Sqlite3 = require("sqlite3");
```

2. 使用以下命令建库。

```
var db = Sqlite3.open(":memory:");
```

说明：

`open()` 字段将根据指定的路径创建或打开数据库文件，如果路径参数为 `:memory:`，则创建一个匿名的内存数据库。

3. 使用以下命令在数据库中建表。

```
db.run("CREATE TABLE user(name text, age int);");
```

说明：

`db.run(arg1)` 表示执行一段 sql 脚本，示例语句创建了一个名为 `user` 的表，字段有 `name`，字段类型为 `text`，`age` 字段的类型为 `int`。sql 脚本的写法参考 SQLite3 的语法。

功能介绍

查询数据

请参考以下示例，该程序监听客户端发来的查询 `user` 数据请求，通过 `db.prepare(arg1)` 执行了查询 `user` 表的 sql 脚本，之后遍历查询出的结果集，最后通过 `stmt.finalize()` 结束查询。

```
router.get("/list", function(req, res) {
  var data = [];
  // 查询user表的所有记录
  db.run("SELECT * FROM user;", (row) => {
```

```

    data.push(row);
  });
  res.json(data);
})();

```

以下为前端发送查询请求示例。

```

getUsers: function () {
  const auth = {
    'edger-token': this.token,
    'edger-srand': this.srand
  };
  axios.get('/api/sqlite/list', {}, {headers: auth})
    .then(res => {
      // ...
    })
    .catch(function (error) {
      console.log(error);
    });
}

```

添加数据

请参考以下示例，该程序将前端提交的 user 数据插入到 SQLite3 数据库的 user 表中。

```

router.post("/add", function(req, res) {
  // 拿到提交的数据
  var user = req.body;
  // 运行sql脚本插入到数据库user表中
  db.run("INSERT INTO user VALUES(?, ?);", user.name, user.age);
  res.json({ ret: true });
});

```

以下为前端提交数据示例。

```

addUser: function () {
  const auth = {
    'edger-token': this.token,
    'edger-srand': this.srand
  };
  axios.post('/api/sqlite/add', { name: this.name, age: this.age },
  {headers: auth})
    .then(res => {
      // ...
    })
    .catch(function (error) {

```

```
    console.log(error);
  });
}
```

edgeros.com

LightKV

本章主要介绍 LightKV 的使用方法。

概述

LightKV 是 EdgerOS 内置的一种事务性 Key/Value 型数据库，拥有轻量级、高性能、高可靠的特点，提供更快的数据保存速度，同时也会消耗大量的内存缓存。

使用方法

1. 使用以下命令导入 LightKV 数据库。

```
var LightKV = require("lightkv");
```

2. 使用以下命令建库。

```
var kv = new LightKV("./lightkv.db", "c+", LightKV.OBJECT);
```

说明：

- *arg1*：数据库文件名。
- *arg2*：打开标志，默认值：`'c+'`。
- *arg3*：表示数据所在数据库存储的类型，默认值：`LightKV.BUFFER`。

功能介绍

初始化计数器

使用以下命令，初始化计数器，如果 `access` 不存在则初始化为 0。

```
if (!kv.has("access")) {
    kv.set("access", { count: 0 });
}
```

计数变量自增

1. 参考以下示例，接收前端请求后，`access` 的 `count` 属性自增 1。

```
router.get("/access", function(req, res) {
    var count = kv.get("access").count;
    count++;
    kv.set("access", { count: count });
    res.json({ count: count });
```

```
    }  
);
```

2. 参考以下示例，在前端页面上构建一个按钮和一个数值展示，用户点击按钮即可发送 access 请求到服务端，本示例通过 Vue 的 Axios 库来向服务端发送 HTTP 请求。

```
onAccess: function () {  
    const auth = {  
        'edger-token': this.token,  
        'edger-srand': this.srand  
    };  
    axios.get('/api/lightkv/access', {}, {headers: auth})  
        .then(res => {// ...  
    })  
        .catch(function (error) {console.log(error);});  
}
```

SyncTable

本章主要介绍 SyncTable 的使用方法。

概述

SyncTable 是一种 Key/Value 型数据库，使用该数据库可以在 EdgerOS 的内存中保存和修改数据，为处于多任务状态的数据提供同步功能。

使用方法

1. 使用以下命令导入 SyncTable 数据库。

```
var SyncTable = require("synctable");
```

2. 使用以下命令建表，本示例创建了一个名称为 table1 的同步表。

```
var table = new SyncTable("table1");
```

功能介绍

初始化计数器

参考以下命令，如果 count 变量不存在，使用命令直接进行初始化。

```
if (!table.has("count")) {
    table.set("count", 0);
}
```

计数变量自增

1. 参考以下示例，接收前端发送 access 请求，获取到 table 里的 count 计数器，自增一，然后更新到 table 中。

```
router.get("/access", function(req, res) {
    // 获取count变量
    var count = table.get("count");
    count++;
    // 更新count
    table.set("count", count);
    res.json({ count: count });
});
```

2. 参考以下示例，在前端发送 aceess 请求。

```
onAccess: function () {
    const auth = {
        'edger-token': this.token,
        'edger-srand': this.srand
    };
    axios.get('/api/synctable/access', {}, {headers: auth})
        .then(res => {
            // ...
        })
        .catch(function (error) {
            console.log(error);
        });
}
```

Redis

本章主要介绍 Redis 的使用方法。

概述

Redis (Remote Dictionary Server)，即远程字典服务，是一个开源、支持网络、可基于内存和持久化的日志型 Key/Value 数据库。

使用方法

EdgerOS 支持 Redis 客户端，请参考以下示例，直接连接 Redis 服务器，更多 Redis 使用方法请参考 [Database/Redis](#)。

```
const iosched = require('iosched');
const redis = require('redis');
const client = redis.createClient({ host: /* Your redis server host*/,
port: /*Your redis server port*/ });

client.on('error', function(error) {
  console.error(error);
});

client.set('key', 'value', redis.print);
client.get('key', redis.print);

while (true) {
  iosched.poll();
}
```

多任务组件

本章主要介绍多任务组件。

概述

EdgerOS 每个应用程序目前只允许一个进程，不排除未来允许应用内部包含多进程。当应用需要高性能运算或并行运行时，可以在应用进程内创建多任务。

使用方法

在 [人脸识别](#) 示例 app-demo-camera-ai 项目主任务中集成了 FaceNN 模块，实现了人脸识别功能。由于人脸侦测占用较多运算资源，有时会影响到主任务的响应速度，可以使用多任务的方案解决这个问题：

- 在主任务中创建子任务，将人脸侦测功能迁移到子任务中处理。
 - 主任务与子任务之间通过 Sigslot 进行通信，子任务将识别的人脸信息发送给主任务。
 - 主任务接收子任务人脸信息后，分发给客户端。
1. 参考以下示例，新建 Task 对象即为新建任务。

```
var task = new Task("./task.js", "test_arg_string");
```

说明：

任务之间可以通过 `Task.send()`, `Task.recv()` 发送和接收消息，更多内容请参考 [Multi-Task/Task](#)。

2. 如果任务间需要进行更灵活地通信时，可使用 SigSlot 信号槽，参考以下示例引入模块。

```
var SigSlot = require("sigslot");
var sigslot = new SigSlot("Test");
```

3. 参考以下示例，在同名信号槽对象上可以在同任务或不同任务之间像事件一样注册和监听信号槽。

```
sigslot.slot("event1", (msg) => {
    // ...
}) ;
// ...
sigslot.emit("event1", msg);
```

说明：

关于信号槽更多内容请参考 [Multi-Task/SigSlot](#)。

功能介绍

创建子任务

- 参考以下示例，`start()` 接口实现中，当 `MediaDecoder` 新建对象连接 `rtsp` 流并启动后，创建一个子任务。

```
start() {
    var netcam = new MediaDecoder();
    // ...
    new Promise((resolve, reject) => {
        netcam.open(url, { proto: 'tcp', name: self.name }, 10000, (err) =>
    {
        // ...
    })
    .then((netcam) => {
        super.start.call(self);
        netcam.start();
        // ...
        self.aiTask = new Task('./src_ai.js', {
            magic: 'tasks-face-flv',
            name: self.name,
        }, {
            directory: module.directory
        });
    })
    // ...
}
```

说明：

`./src_ai.js` 是新任务执行程序，注意创建任务时，设置了 `directory: module.directory` 选项，表示文件路径是相对当前模块的。

- 创建子任务时传入了 `name: self.name` 选项，参考以下示例，`self.name` 在构造函数中定义如下。

```
constructor(serv, mode, inOpts, outOpts) {
    this.name = `${input.host}:${input.port}${input.path}`;
}
```

- 主任务中新建 `MediaDecoder` 对象，使用 `netcam.open()` 打开一个命名的 `rtsp` 流，参考以下示例，在子任务中创建 `MediaDecoder` 同名副本对象。

```
class SrcAI {
    constructor(name) {
```

```

// ...
this.start();
}

start() {
// ...
this.netcam = new MediaDecoder().open(this.main);
this.netcam.on("video", this.onVideo.bind(this));
this.netcam.start();
}
// ...
}

var name = ARGUMENT.name;
var srcAI = new SrcAI(name);

```

4. 监听 netcam 的 video 事件。获取视频帧数据。

使用 Sigslot 通信

1. 参考以下示例，引入 Sigslot 模块。

```
var Sigslot = require("sigslot");
```

2. 参考以下示例，在主任务中创建一个命名的 Sigslot 对象。

```

constructor(serv, mode, inOpts, outOpts) {
// ...
this.aiSlot = new Sigslot(this.name);
var self = this;
this.aiSlot.slot('error', (type, msg) => {
console.error('AI task err, ai to be close:', msg);
self.aiTask.cancel();
self.aiTask = null;
}, 'error');
this.aiSlot.slot('face', (type, msg) => {
self.onVideo(msg);
}, 'face');
}

```

说明：

`new Sigslot(this.name)` 的 `name` 即为传入子任务的 `name` 参数。

3. 参考以下示例，在子任务中创建一个同名的 Sigslot 对象。

```

constructor(name) {
    // ...
    var slot = new SigsSlot(name);
}

```

说明：

- 主任务与子任务之间就可以发送和接收消息。
- 以上代码表示主任务 `aiSlot` 对象监听了 `face` 信号槽事件，因此主任务可以接收子任务 `face` 信号槽消息。

4. 参考以下示例，子任务在处理视频帧数据并通过信号槽向主任务发送消息。

```

onVideo(frame) {
    var buf = new Buffer(frame.arrayBuffer);
    const view = DEF_DETECT_VIEW;
    var faceInfo = facenn.detect(buf, { width: view.width, height: view.height, pixelFormat: FACENN_PIXEL_FORMAT });
    var ret = [] /* Empty array - clear. */

    for (var i = 0; i < faceInfo.length; i++) {
        var info = {};
        info.x0 = Math.max(faceInfo[i].x0 - 10, 0);
        info.x1 = Math.min(faceInfo[i].x1 + 10, view.width - 1);
        info.y0 = Math.max(faceInfo[i].y0 - 10, 0);
        info.y1 = Math.min(faceInfo[i].y1, view.height - 1);
        ret.push(info);
    }

    /* sigslot send to src. */
    if (this.slot) {
        this.slot.emit('face', ret);
    }
}

```

说明：

- 子任务中人脸侦测处理与前一章基本相同，只是人脸定位数据未经转换。
- `this.slot.emit('face', ret)` 在 `face` 信号槽上发送以上数据。

5. 参考以下示例，主任务接收人脸侦测数据。

```

onVideo(infos) {
    const view = DEF_DETECT_VIEW;
    for (var i = 0; i < infos.length; i++) {
        var info = infos[i];
    }
}

```

```
info.x0 = Math.round(info.x0 * this.mediaInfo.width / view.width);
info.x1 = Math.round(info.x1 * this.mediaInfo.width / view.width);
info.y0 = Math.round(info.y0 * this.mediaInfo.height / view.height);
info.y1 = Math.round(info.y1 * this.mediaInfo.height / view.height);
}

var cliMgr = this.getCliMgr();
cliMgr.iter(function(cli) {
cli.sendData({type: 'face'}, infos);
});
}
```

说明：

主任务接收到人脸侦测数据，将转换定位后的数据分发给当前客户端。

设备管理框架

本章主要介绍如何基于爱智设备管理框架进行设备控制。

概述

爱智应用开发中可以直接使用 EdgerOS 设备管理框架进行设备控制，EdgerOS 支持接入 ZigBee 设备和 SDDC 设备，并通过统一接口进行访问，具体请参考：

- ZigBee 设备控制 [IoT Device/ZigBeeDev](#)。
- SDDC 设备控制 [IoT Device/SddcDev](#)。

应用示例

爱智提供以下设备控制应用示例，供开发者参考：

- [智能插座](#)
- [IoT Pi 控制](#)

功能介绍

导入模块

设备控制需要使用 Device 模块来发现和管理设备，爱智模板中的 app-demo-iotpi 和 app-demo-plug 项目集成了 Device 模块，代码位于 main.js 中。

```
var Device = require("device");
```

参考以下 IoT Pi 示例，设置两个变量，用于保存发现的 IoT Pi 设备列表和当前访问的 IoT Pi 设备。

```
/* IoT Pi device */
var iotpi = undefined;
/* IoT Pi devices */
var iotpis = new Map();
```

设备发现

1. 参考以下 IoT Pi 示例，爱智应用启动时，初始化获取设备列表。

```
/*
 * Get All Iot Pi device
 */
Device.list(true, function(error, list) {
    if (list) {
        list.forEach(function(dev) {
            Device.info(dev.devid, function(error, info) {
```

```
    if (info && info.report.name === "IoT Pi") {  
        iotpis.set(dev.devid, {  
            devid: dev.devid,  
            alias: dev.alias,  
            report: info.report,  
        });  
    }  
});  
});  
}  
});
```

2. `Device.list()` 获取设备列表后，`Device.info()` 将进一步获取设备详细信息，然后根据设备筛选出 IoT Pi 设备对象并保存在 `iotpis` 数组中。
 3. 参考以下示例，引入 `Socket.IO` 与前端实时交互，创建 `Socket.IO`，用于将新设备或设备离线消息及时通知前端。

```
/* Socket IO */
var io = require("socket.io")(app, {
  path: "/iotpi",
  serveClient: false,
  pingInterval: 10000,
  pingTimeout: 5000,
  cookie: false,
});
```

4. 参考以下示例，在前端 Vue 项目中安装 socket.io-client，vue-socket.io-extended，用于使用 Socket.IO。

```
npm install socket.io-client@2.3  
npm install vue-socket.io-extended
```

5. 参考以下示例，在 socket 连接时加入 auth 参数作为前端认证令牌。

```
const socket = SocketIO({
  path: "/",
  query: auth,
  transports: ["websocket"],
});
```

注意：

- *auth* 对象来源于 EdgeOS 提供的安全机制，详情请参考 [SDK/Security](#)。
 - *path* 选项前端后端应保持一致。

- *socket.io-client* 只能安装 2.x 版本，推荐使用 [socket.io-client@2.3](#) 版。

6. 参考以下示例，当 Socket.IO 前端建立连接时，监听前端获取设备列表的请求。

```
io.on("connection", function(sockio) {
    // ...
    sockio.on("iotpi-list", function(result) {
        var devs = [];
        iotpis.forEach(function(iotpi) {
            devs.push(iotpi);
        });
        result(devs);
    });
});
```

7. 使用以下命令，前端发送设备列表请求，获取后端设备列表。

```
this.$socket.client.emit("iotpi-list", (data) => {
    this.iotpis = data;
    // ...
});
```

爱智应用运行过程中，通过 *join* 事件可以监听新设备加入。

```
Device.on("join", function(devid, info) {
    if (info.report.name === "IoT Pi") {
        var devobj = {
            devid: devid,
            alias: info.alias,
            report: info.report,
        };
        iotpis.set(devid, devobj);
        io.emit("iotpi-join", devobj);
    }
});
```

8. 参考以下示例，将新加入设备保存到设备列表 *iotpis* 中，同时 *socket.io* 对象将新设备事件 *iotpi-join* 通知给所有前端，前端 *socket* 会通过 *iotpi-join* 事件来监听后端的推送，如有新加入设备，更新本地设备列表。

```
this.$socket.$subscribe("iotpi-join", (iotpi) => {
    // ...
    this.iotpis.push(iotpi);
});
```

参考以下示例，lost 事件可以监听设备离线，离线的设备通过 Socket.IO 消息 iotpi-lost 通知给所有前端。

```
Device.on("lost", function(devid) {
    if (iotpis.has(devid)) {
        iotpis.delete(devid);
        if (iotpi && iotpi.devid === devid) {
            iotpiRemove();
        }
        io.emit("iotpi-lost", devid);
    }
});
```

9. 参考以下示例，前端 socket 通过 iotpi-lost 事件来监听后端的推送，设备离线将处罚本地设备列表更新。

```
this.$socket.$subscribe("iotpi-lost", (devid) => {
    // ...
    this.iotpis = this.iotpis.filter((iotpi) => {
        return iotpi.devid !== devid;
    });
});
```

创建设备

1. 每个设备有唯一的设备 ID (devid)，前端从设备列表中选择某个设备时，通过 REST 接口将设备 ID 发送给后端。参考以下示例，爱智应用后端获取有效 devid 后便可创建设备对象。

```
app.post("/api/select/:devid", function(req, res) {
    // ...
    iotpi = new Device();
    iotpi.request(req.params.devid, function(error) {
        // ...
        iotpi.send(
            { query: true },
            function(error) {
                if (error) {
                    console.error("Query IoT Pi error:", error.message);
                } else {
                    console.log("Query IoT Pi Ok!");
                }
            },
            3
        );
    });
});
```

2. 参考以下示例，创建 IoT Pi 设备对象，通过 `iotpi.request()` 与设备关联成功后，`iotpi.send()` 将检查设备是否可用。前端会发送切换设备请求，根据后端回复进入设备详情页面。

```
axios
  .post(`/api/select/${iotpi.devid}`, {}, { headers: getHeaders() })
  .then(() => {
    this.$router.push({ name: "Details", params: iotpi });
  })
  .catch((error) => {
    // ...
  });
});
```

注意：

在创建 IoT Pi 设备之前爱智应用需要获取设备权限，否则执行 `iotpi.request()` 时会失败。

设备控制

1. 参考以下 IoT Pi 示例，前端可以通过 Socket.IO 发送 `iotpi` 指示灯开关控制命令。

```
this.$socket.client.emit("iotpi-control", msg);
```

2. 参考以下示例，在后端监听 `iotpi-control` 事件。

```
io.on("connection", function(sockio) {
  sockio.on("iotpi-control", function(msg) {
    if (iotpi && iotpi.devid) {
      console.log("Client send message:", JSON.stringify(msg));
      iotpi.send(
        msg,
        function(error) {
          if (error) {
            console.error("Send message to IoT Pi error:",
error.message);
          }
        },
        3
      );
    } else {
      sockio.emit("iotpi-error", { error: "No device!" });
    }
  });
  // ...
});
```

3. 参考以下示例，通过 `iotpi.send()` 向设备发送开关命令。

```
{"led1": true} // 开led1
{"led1": false} // 关led1
{"led2": true} // 开led2
{"led2": false} // 关led2
{"led3": true} // 开led3
{"led3": false} // 关led3
```

4. 参考以下示例，IoT Pi 设备对象监控 message 消息可获得 IoT Pi 的 led 开关控制状态变化，将结果同步给前端，该事件在 `iotpi` 对象创建时监听。

```
iotpi.request(req.params.devid, function(error) {
    iotpi.on("message", function(msg) {
        io.emit("iotpi-message", msg);
    });
});
```

5. 参考以下示例，前端 socket 监听 message 事件获取后端推送来的设备状态。

```
this.$socket.$subscribe("iotpi-message", (msg) => {
    if (typeof msg.led1 !== "undefined") {
        this.iotpi.led1 = msg.led1;
    }
    if (typeof msg.led2 !== "undefined") {
        this.iotpi.led2 = msg.led2;
    }
    if (typeof msg.led3 !== "undefined") {
        this.iotpi.led3 = msg.led3;
    }
});
```

iotpi-message 消息：

```
{"led1": true} // 开led1
{"led1": false} // 关led1
{"led2": true} // 开led2
{"led2": false} // 关led2
{"led3": true} // 开led3
{"led3": false} // 关led3
```

AI 框架

本章主要介绍爱智 AI 框架。

概述

爱智具备强大的边缘 AI 处理能力，提供 AI 组件库和便捷的 JS API，方便开发者接入和使用 AI。同时提供多种灵活的 AI 模型安装机制，让更多的模型运行在爱智上，适配不同的用户场景，EdgerOS AI 包含以下框架：

- NCNN 是一个针对嵌入式平台优化的高性能神经网络计算框架，详情请参考 [AI Engine/NCNN](#)。
- FaceNN 模块提供人脸检测和识别功能，详情请参考 [AI Engine/FaceNN](#)。
- HandNN 模块提供手部检测识别功能，详情请参考 [AI Engine/HandNN](#)。
- ThingNN 模块提供物体检测和识别功能，详情请参考 [AI Engine/ThingNN](#)。
- LicPlateNN 模块提供车牌检测和识别功能，详情请参考 [AI Engine/LicPlateNN](#)。

应用示例

爱智提供以下 AI 应用示例，供开发者参考：

- [智能门锁](#)
- [人脸识别](#)

功能介绍

人脸识别

用户信息记录与人脸识别等 AI 功能由 faceai (facelock/face.js) 模块提供。

```
const faceai = require("./face");
```

1. 前后端使用 WebSyncTable 技术进行交互，参考以下示例，应用初始化时首先创建 WebSyncTable 对象。

```
const SyncTable = require("synctable");
const WebSyncTable = require("websynctable");
// ...
var main = new SyncTable("main");
var server = new WebSyncTable(app, main);
```

说明：

WebSyncTable 对象的使用请参考 [WebSyncTable 开发](#)。

2. 参考以下示例，在前端引入 @edgeros/web-synctable 模块。

```
<script src="static/js/synctable.min.js"></script>
```

3. 参考以下示例，在前端创建 WebSyncTable 对象。

```
var link = null;
app.security.init(function() {
    if (link == null) {
        var proto = location.protocol === "http:" ? "ws:" : "wss:";
        var server = `${proto}//${window.location.host}`;
        link = new SyncTable(server, "main", app.security);
        // ...
    }
});
```

说明：

- WebSyncTable 需要使用 EdgerOS 安全机制，app 模块维护了安全机制的 token 等信息。
- EdgerOS 安全机制请参考[服务端安全认证](#)。

4. 数据采集与解锁通过 ESP32 SDDC 设备实现，ESP32 SDDC 设备与 EdgerOS 之间通过 SDDC 协议传送数据以及收发指令。参考以下示例，在应用初始化时创建一个设备对象。

```
const Device = require("device");
const dev = new Device();
```

人脸解锁

ESP32 SDDC 设备从人体传感器获取一路输入信号，当用户经过人体传感器时，触发一次解锁过程，设备向应用发送 recv 消息，参考以下示例，实现应用中设备对象监听消息 (main.js)。

```
dev.on("message", function(msg) {
    if (msg && msg.cmd === "recv" && msg.size > 0) {
        recv(false, msg.size)
            .then((chunk) => {
                lastpic = chunk;
                var name = undefined;
                var info = faceai.detect(lastpic);
                if (info) {
                    for (var face of info.faces) {
                        var feature = faceai.feature(face, info.bitmap);
                        if (feature) {
                            name = faceai.best(feature);
                            if (name) {
                                dev.send({ cmd: "unlock", timeout: 5000 });
                            }
                        }
                    }
                }
            })
    }
});
```

```

        break;
    }
}
} else {
    console.log("No face detect!");
}
server.reverse("recv", { file: RECV_FILE, name });
})
.catch((error) => console.error(error));
}
);
}
);

```

人脸解锁过程：

1. `recv()` 方法从设备获取图像并更新到缓存。
2. `faceai.detect()` 从缓存侦测人脸信息。
3. `faceai.feature()` 提取人脸特征值。
4. `faceai.best()` 从数据库中比对人脸特征值并返回用户识别结果。
5. 识别到合法用户时，`dev.send()` 向设备发起解锁命令，其中选项 `timeout` 表示开锁持续时间。
6. `server.reverse()` 向前端反馈结果，前端更新识别的图像。

设备发现

参考以下示例，在 `devreq()` 方法中首先获取设备列表，从列表中筛选我们需要的设备，然后将设备对象与有效的设备信息关联起来。

```

async function devreq() {
    var list = await devlist();
    for (var device of list) {
        var info = await devinfo(device.devid);
        if (info && info.report.name === "IoT Camera") {
            request(device.devid);
            break;
        }
    }
}

```

设备发现步骤：

1. 在 `devlist()` 方法中调用 `Device.list()` 获取设备列表。
2. 获取设备列表后，用 `devinfo()` 方法遍历每个设备，`devinfo()` 方法调用 `Device.info()` 获取每个设备信息。
3. 筛选出所需的 AI 设备。

说明：

`Device` 模块的使用请参考 [Device](#)。

用户注册

1. 注册用户后才能通过设备采集用户人脸数据，参考以下示例，由前端发起请求。

```
recv: async function() {
  try {
    var reply = await link.fetch('recv', {
      cipher: bindings.data.cipher
    }, true, 4000);
    bindings.data.response = reply.res;
    if (reply.res === 'ok') {
      bindings.data.picture = reply.file + `?t=${Date.now()}`;
    }
    // ...
  }
  // ...
}
```

说明：

- `link.fetch()` 发起 `recv` 请求时附带了 `cipher` 选项，它决定了应用与设备是否加密通信。
- 将返回的图片 URL 地址 (`reply.file`) 更新到 `bindings.data.picture` 上，前端可预览用户图像。

2. 后端接收到请求，调用 `recv()` 过程获取用户图像，并返回图像信息。

```
server.on("recv", (msg, client, reply) => {
  // ...
  recv(msg.cipher).then((chunk) => {
    lastpic = chunk;
    reply({ res: "ok", file: RECV_FILE, size: chunk.length });
  });
  // ...
});
```

说明：

`lastpic = chunk` 将获取的图像缓存起来，后续从缓存中直接读取图像数据。

3. 参考以下示例，配置 `recv()` 实现。

```
async function recv(cipher, size) {
  return new Promise((resolve, rejects) => {
    var chunks = [];
    var connector = new Device.Connector(dev, cipher);
```

```

dev.send({ cmd: "recv", connector, size });
// ...
connector.on("data", function(data) {
  chunks.push(data);
});
// ...
connector.on("close", function() {
  if (chunks.length) {
    var chunk = Buffer.concat(chunks);
    if (!size || chunk.length === size) {
      return resolve(chunk);
    }
  }
  rejects("Recv error!");
});
});
}

```

说明：

`dev` 对象向设备发送 `recv` 消息请求图像数据，`dev.send()` 附加了一个新建的 `Device.Connector` 对象，`Device.Connector` 是一个 `stream.Duplex` 类，可以接收数据流。

4. 前端获取图像地址后，预览图像，当采集到合适的人脸图像后，参考以下示例，前端发出保存请求。

```

save: async function() {
  // ...
  try {
    var reply = await link.fetch('save', {
      name: bindings.data.name
    }, true, 4000);
    // ...
  }
  // ...
}

```

说明：

请求附带了注册用户的名称。

5. 参考以下示例，后端接收保存请求。

```

server.on("save", (msg, client, reply) => {
  // ...
  var info = faceai.detect(lastpic);

```

```

if (info == undefined) {
    reply({ res: "error", info: "No face detect!" });
} else if (info.faces.length > 1) {
    reply({ res: "error", info: "Too many face detect!" });
} else {
    var feature = faceai.feature(info.faces[0], info.bitmap);
    if (feature) {
        faceai.save(msg.name, feature);
        reply({ res: "ok" });
    } else {
        reply({ res: "error", info: "Facial features are not obvious!" });
    }
}
);

```

说明：

lastpic 是之前缓存的图像数据，后端接收到保存请求后进行以下处理：

1. `faceai.detect(lastpic)` 值检测图像中人脸信息。
2. `faceai.feature()` 提取人脸特征值。
3. `faceai.save()` 保存用户信息，完成用户注册。

补充说明

faceai 模块介绍

在以上用户注册与人脸解锁过程中都使用了 `faceai` 模块，`faceai` 模块实现了一个 `FaceDatabase` 类，用来保存用户注册信息，然后提供了两组接口，一组是数据库接口，另一组是 AI 接口。

`FaceDatabase` 类封装了一个 `LightKV` 对象和一个 `Map` 对象。

```

class FaceDatabase {
    constructor() {
        this.lkv = new LightKV("./faces.lkv", "c++", LightKV.OBJECT);
        this.map = this.lkv.toMap();
    }
    // ...
}

```

说明：

- `LightKV` 用于数据持久化记录，关于 `LightKV` 数据库使用请参考 [LightKV 示例](#)。
- `Map` 对象从 `LightKV` 同步数据，作为数据缓存，加快数据访问。
- `faceai` 模块创建了一个 `FaceDatabase` 类全局对象 `facedb`，并提供了两个接口访问数据库：
 - `save(name, feature)` 保存用户信息（用户名 + 识别的人脸特征值）。

- `remove(name)` 从数据库中删除一个用户记录。

faceai 主要包括以下接口：

- `faceai.detect()` 接口

```
function detect(picture) {
    try {
        var bitmap = imagecodec.decode(picture, {
            components: imagecodec.COMPONENTS_RGB,
        });
        if (bitmap == undefined) {
            throw new Error("Unknown picture format!");
        }
        var faces = facenn.detect(
            bitmap.buffer,
            {
                width: bitmap.width,
                height: bitmap.height,
                pixelFormat: facenn.PIX_FMT_RGB24,
            },
            true
        );
        if (faces) {
            for (var i = 0; i < faces.length; i++) {
                if (faces[i].score < DETECT_CONFIDENCE_RATE) {
                    faces.splice(i, 1);
                    i--;
                }
            }
            if (faces.length) {
                console.log("Detect face num:", faces.length);
                return { faces, bitmap };
            }
        }
    } catch (error) {
        console.error("Image decode error", error);
    }
    return undefined;
}
```

`faceai.detect()` 实现图像数据接收，侦测图像中人脸，并返回侦测结果，操作步骤如下：

1. `imagecodec.decode()` 对图片进行解码，`imagecodec` 模块请参考 [imagecodec](#)。
2. `facenn.detect()` 侦测图像中人脸，返回侦测到的一组人脸信息，`facenn` 模块请参考 [facenn](#)。

3. 在一个循环中遍历侦测到的人脸信息，`faces[i].score < DETECT_CONFIDENCE_RATE` 实现按默认阈值过滤掉辨识度低的人脸信息。
 4. 返回已识别的人脸信息。
- `faceai.feature()` 接口

```
function feature(face, bitmap) {
    var feature = facenn.feature(
        bitmap.buffer,
        {
            width: bitmap.width,
            height: bitmap.height,
            pixelFormat: facenn.PIX_FMT_RGB24,
        },
        face,
        { live: true }
    );
    if (feature.live < LIVING_BODY_CONFIDENCE_RATE) {
        console.log("No live:", feature.live);
        return undefined;
    } else {
        return feature.keys;
    }
}
```

`faceai.feature()` 接口用于提取人脸特征值，它接收从 `faceai.detect()` 返回的人脸数据及解码的图像数据，操作步骤如下：

1. `faceai.feature()` 使用 `facenn.feature()` 提取人脸特征信息。
2. 提供特征值之后，`feature.live < LIVING_BODY_CONFIDENCE_RATE` 按默认阈值判断活体对象。
3. 返回满足活体检测的人脸特征值。

- `faceai.best()` 接口

```
function best(feature) {
    https://www.edgeros.com/edgeros/guide/extensi.html  var iter =
facedb.map[Symbol.iterator]();
    var name = undefined;
    var near = 0;
    for (var item of iter) {
        var same = facenn.compare(feature, item[1]);
        if (same > near) {
            near = same;
            name = item[0];
        }
    }
}
```

```

if (name) {
    console.info("Best mach with:", name, "prob:", near);
}
return near >= RECONGNITION_CONFIDENCE_RATE ? name : undefined;
}

```

`faceai.best()` 用于查询用户，接口接收一个人脸特征值，从数据库中检索出与特征值最匹配的用户。操作步骤如下：

<https://www.edgeros.com/edgeros/guide/extensi.html1>. `facedb.map[Symbol.iterator]()` 从数据库缓存中获取迭代器，检索每个用户记录。2. `facenn.compare(feature, item[1])` 将待匹配的特征值与每个用户数据比对，最后获得与特征值最相似的用户。3. 最后还要进行阈值判断，如果匹配成功返回用户名。

说明：

`faceai` 模块还提供了一个简单的查询用户接口 `faceai.search()`，这个方法返回查询到的第一个用户。

扩展 CameraSource

1. 在 `CameraSource` 接口 `start()` 中，新创建的 `MediaDecoder` 需监听 `video` 事件获取视频帧数据。

```

start() {
    var netcam = new MediaDecoder();
    // ...
    new Promise((resolve, reject) => {
        netcam.open(url, { proto: 'tcp', name: name }, 10000, (err) => {
            // ...
            netcam.on('video', self.onVideo.bind(self));
            resolve(netcam);
        });
    });
    // ...
}

```

2. 参考以下示例，为了准确定位人脸在视频中的相对位置，需要将原始视频分辨率信息反馈给前端。

```

start() {
    // ...
    super.start.call(self);
    netcam.start();
    var info = netcam.srcVideoFormat();
    self.mediaInfo = { width: info.width, height: info.height, fps: Math.round(info.fps) };
}

```

```

self.sendDataHeader({ type: 'media' }, self.mediaInfo); /* {width,
height, fps} */
// ...
}

```

说明：

`netcam.start()` 之后，`netcam.srcVideoFormat()` 获取视频信息，通过 `MeidaSource` 接口 `sendDataHeader()` 将视频分辨率信息通知给客户端，同时视频信息还将缓存起来，确保每一个连接到流媒体服务的客户端都能在连接握手成功后立即接收到这一信息。

3. 参考以下示例，为了识别人脸信息，引入 `facenn` 模块。

```
var facenn = require("facenn");
```

4. 参考以下示例，在注册的事件中接收并处理视频数据。

```

onVideo(frame) {
    var buf = new Buffer(frame.arrayBuffer);
    const view = DEF_DETECT_VIEW;
    var faceInfo = facenn.detect(buf, { width: view.width, height:
view.height, pixelFormat: FACENN_PIXEL_FORMAT });
    var ret = []; /* Empty array - clear. */

    for (var i = 0; i < faceInfo.length; i++) {
        var info = {};
        info.x0 = Math.max(faceInfo[i].x0 - 10, 0);
        info.x1 = Math.min(faceInfo[i].x1 + 10, view.width - 1);
        info.y0 = Math.max(faceInfo[i].y0 - 10, 0);
        info.y1 = Math.min(faceInfo[i].y1, view.height - 1);

        info.x0 = Math.round(info.x0 * this.mediaInfo.width / view.width);
        info.x1 = Math.round(info.x1 * this.mediaInfo.width / view.width);
        info.y0 = Math.round(info.y0 * this.mediaInfo.height / view.height);
        info.y1 = Math.round(info.y1 * this.mediaInfo.height / view.height);
        ret.push(info);
    }

    var cliMgr = this.getCliMgr();
    cliMgr.iter(function (cli) {
        cli.sendData({ type: 'face' }, ret);
    });
}

```

说明：

- `facenn.detect()` 探测视频中人脸并返回人脸相对视频位置信息。

- 输出的视频帧经过了转换，客户端接收的视频流分辨率与原始流相同，还需要将获取的人脸位置转换为相对原始视频中的位置，代码中间的迭代计算便是重新计算每个已识别人脸的坐标位置。

5. 通过 cliMgr 遍历每个客户端，将识别的人脸信息分发给每个已连接的客户端。

`cli.sendData()` 设计为利用数据通道向客户端推送结构化数据的接口。

前端扩展

本节示例前端基于[智能摄像头](#)示例前端工程`app-demo-camera-base/web`扩展实现，工程位于`app-demo-camera-ai/web`目录下。本节使用`@edgeros/web-mediachannel`数据通道接口，接收流媒体服务器的人脸识别信息。

1. 为了便于处理，继承`MediaClient`实现了一个扩展的类`ClientWrap`，代码位于`app-demo-camera-ai/web/src/lib/mediachannel.js`。

```
export default function createMediaClient(
  ClientType,
  origin,
  canvas,
  opts,
  shakeHandle
) {
  class ClientWrap extends ClientType {
    constructor(origin, canvas, opts, shakeHandle) {
      super(origin, shakeHandle, opts);
      // ...
    }
  }
  // ...
  return new ClientWrap(origin, canvas, opts, shakeHandle);
}
```

2. 参考以下示例，在`Player.vue`中创建`mediaClient`对象。

```
import createMediaClient from "@lib/mediachannel";
// ...
const { w, h } = this.getPageSize();
this.np = new NodePlayer();
// ...
var canvas = document.getElementById("layout");
var proto = location.protocol === "http:" ? "ws:" : "wss:";
var host = `${proto}//${window.location.host}`;
var mediaClient = createMediaClient(
  MediaClient,
  host,
  canvas,
```

```

{
  canvaw: w,
  canvah: h,
  path: this.dev.path,
},
(client, path) => {
  this.np.start(host + path);
}
);

```

说明：

创建 `mediaClient` 时需传入一个 `canvas` 参数，`canvas` 是用于绘制识别人脸可视化数据的画布，对应 `layout` 组件。

```

<div
  class="canvas-wrapper"
  :style="{position:'relative', width: width + 'px', height: height +
'px'}"
>
  <canvas
    id="video"
    :style="{position:'absolute', width: width + 'px', height: height +
'px'}"
  />
  <canvas
    id="layout"
    :style="{position:'absolute'}"
    :width="width"
    :height="height"
  />
</div>

```

3. 将 `layout` 画布置于 `video` 视频画布之上与之重合，实现人脸识别与视频同步。
4. 处理 `ClientWrap` 内部实现，参考以下示例。通过监听 `MediaClient` 的 `data` 事件，接收流媒体服务器数据通道推送数据。

```

constructor(origin, canvas, opts, shakeHandle) {
  super(origin, shakeHandle, opts);
  // ...
  this.on('data', this.onData.bind(this));
}

```

5. 参考以下示例，`onData` 接收并处理 `data` 事件。

```
onData(self, opts, data) {
    var type = opts && opts.type ? opts.type : null;
    if (type === 'media') {
        this.videoow = data.width;
        this.videooh = data.height;
        this.rw = this.canvaw / this.videoow;
        this.rh = this.canvah / this.videooh;

    } else if (type === 'face') {
        this.draw(data);

    } else {
        console.error('Data invalid.');
    }
}
```

多媒体框架

本章主要介绍爱智多媒体框架。

概述

爱智提供多种多媒体处理模块，用于处理多媒体应用的解码、视频和流媒体服务，具体请参考：

- MediaDecoder：多媒体解码模块，请参考 [Multi-Media/MediaDecoder](#)。
- VideoOverlay：视频预览模块，请参考 [Multi-Media/VideoOverlay](#)。
- WebMedia：流媒体服务器框架，请参考 [Multi-Media/WebMedia](#)。

应用示例

爱智提供以下多媒体应用示例，供开发者参考：

[智能摄像头](#)

功能介绍

定制 MediaSource

1. 以流媒体服务为例，需要自定义 MediaSource 接口，具体请参考 [Multi-Media/WebMedia](#)，新实现的 CameraSource 位于 camera_src.js 文件中。

参考以下示例，CameraSource 直接继承至 JSRE 内置的 FlvSrc 类（注册名称：flv），在 FlvSrc 基础上已经具备基础的流解析和分发功能。

```
var FlvSrc = require("webmedia/source/flv");
class CameraSource extends FlvSrc {
    constructor(serv, mode, inOpts, outOpts) {
        super(serv, mode, inOpts, outOpts);
        // ...
    }
    // ...
}
```

2. 参考以下示例，重写 start() 接口，将 MediaDecoder 对象封装到 CameraSource 中，使 CameraSource 类能够完整地接收和处理 rtsp 流。

```
var MediaDecoder = require('mediadecoder');
// ...
start() {
    var netcam = new MediaDecoder();
    this._netcam = netcam;
    var self = this;
```

```
var input = this.inOpts;
var url =
`rtsp://${input.user}:${input.pass}@${input.host}:${input.port}${input.pat

var name = `${input.host}:${input.port}${input.path}`;
new Promise((resolve, reject) => {
  netcam.open(url, { proto: 'tcp', name: name }, 10000, (err) => {
    // ...
    netcam.on('remux', self.onStream.bind(self));
    netcam.on('header', self.onStream.bind(self));
    netcam.on('eof', self.onEnd.bind(self));
  }
})
```

说明：

rtsp 流地址根据 *CameraSource* 创建时传入参数 *inOpts* 生成。

3. 参考以下示例，netcam 对象将转换后的流交由 `onStream()` 方法处理，由于 FlvSrc 实现了 `pushStream()` 接口，只需直接将流推送给 FlvSrc 处理即可，CameraSource 实现了完整的获取、转换、分发流功能。

```
onStream(frame) {
  if (!this._netcam) {
    return;
  }
  var buf = Buffer.from(frame.arrayBuffer);
  try {
    this.pushStream(buf);
  } catch (e) {
    console.error(e);
    this.stop();
  }
}
```

创建流媒体服务器

- 参考以下流媒体服务示例，先在 WebMedia 框架中注册 CameraSource(main.js)，注册完成后，就可以使用名称为 camera-fly 的 MediaSource。

```
var WebMedia = require("webmedia");
var CameraSource = require("./camera_src");

const sourceName = "camera-flv";
WebMedia.registerSource(sourceName, CameraSource);
```

2. 参考以下示例，创建流媒体服务器。

方式一

引用 @edgeros/jsre-medias 模块创建流媒体服务器。

```
const { Manager } = require("@edgeros/jsre-medias");
```

@edgeros/jsre-medias 主要由 Media 和 Manager 构成：

- Manager 集成了 onvif 模块，可以自动发现、记录设备，创建和管理摄像头对象，创建和管理 Media 对象。
- Media 是对 WebMedia 服务器的封装，Manager 服务可以同时接入多路流媒体。

方式二

使用 Manager 创建流媒体服务器。

```
var server = undefined;
// ...
function createMediaSer() {
  console.log("Create media server.");
  if (server) {
    return server;
  }

  var opts = {
    mediaTimeout: 1800000,
    searchCycle: 20000,
    autoGetCamera: false,
  };
  server = new Manager(app, null, opts, (opts) => {
    return {
      source: sourceName,
      inOpts: opts,
      outOpts: null,
    };
  });
  // ...
}
```

Manager 创建的流媒体服务器是双通道模式，流媒体通道与数据通道都是基于 WebSocket 协议，Manager 内部自动创建 WebSocket 服务器，并动态生成流媒体通道访问路径。

说明：

- *mediaTimeout*：每路流媒体服务的超时时长，当没有任何客户端访问服务，超时后服务将会关闭并回收资源，避免长时间占用系统资源。
- *searchCycle*：搜索设备周期。

- *autoGetCamera* : 默认情况下 Manager 搜索到一个设备后会尝试将设备升级为摄像头对象并获取其流地址，如果失败了将摄像头对象降级为普通设备对象。本示例中我们禁用此功能。
- 创建 Manager 时需传入一个回调函数，这个回调函数返回一个对象，这个对象将作为创建 WebMedia 流媒体服务器的参数。其中我们可以看到它指定了之前创建的 CameraSource，并且动态地传入了 *inOpts*，*inOpts* 中包含搜索到的设备返回的 rtsp 流地址参数。

获取设备列表

参考以下流媒体示例，前端通过 REST 接口请求设备列表，交由后端处理。

```
app.get("/api/list", (req, res) => {
  // ...
  var devs = [];
  server.iterDev((key, dev) => {
    var info = dev.dev;
    var stream = dev.mainStream;
    var media = stream ? stream.media : null;
    devs.push({
      devId: key,
      alias: `${info.hostname}:${info.port}${info.path}`,
      report: info.urn,
      path: media ? "/" + media.sid : "",
      status: media ? true : false,
    });
  });
  res.send(JSON.stringify(devs));
});
```

说明：

server.iterDev 作用是获取搜索到的设备对象，*media* 是已识别为 rtsp 摄像头设备并创建的流媒体服务器对象，该对象中已经包含 *path* 相关信息，前端根据此路径便可连接到流媒体服务器。

连接设备

1. 参考以下流媒体示例，前端获取到设备列表后，对于未连接的设备（*media* 为空）需提供账号密码尝试连接设备，用于创建流媒体服务。

```
app.post("/api/login", bodyParser.json(), (req, res) => {
  // ...
  var ret = { result: false, msg: "error" };
  var info = req.body;
  try {
    connectMedia(info, (media) => {
      if (!media || media instanceof Error) {
        ret.msg = `Device ${info.devId} login fail.`;
```

```

        console.warn(media ? media.message : ret.msg);
    } else {
        ret.result = true;
        ret.msg = "ok";
        ret.path = "/" + media.sid;
    }
    res.send(JSON.stringify(ret));
});
}
// ...
);

```

说明：

info 包含前端传入的设备 ID 与账号密码。

2. 参考以下示例，`connectMedia()` 是创建流媒体服务过程，回调返回 `media` 对象，这个过程可能会失败，失败原因可能是账号密码不正确，也可能是该设备不是 rtsp 设备。

- 识别摄像头设备：

```

function connectMedia(info, cb) {
    var devId = info.devId;
    var dev = server.findDev(devId);
    // ...
    var stream = dev.mainStream;
    if (!stream) {
        server.createStream(devId, info, (err, streams) => {
            if (err) {
                cb(err);
            } else {
                getMedia(cb);
            }
        });
    }
    // ...
}

```

说明：

`server.createStream` 识别摄像头设备，并将获取的视频流信息保存在内部 `streams` 对象中。

- 创建流媒体服务：

```

function connectMedia(info, cb) {
    // ...
    function getMedia(cb) {
        var stream = dev.mainStream;

```

```

if (!stream) {
    return cb();
} else if (stream.media) {
    return cb(stream.media);
}
server.createMedia(devId, stream.token, info, (err, media) => {
    if (err) {
        cb(err);
    } else {
        cb(media);
    }
});
}
}

```

说明：

从设备上获取一路主媒体流 `dev.mainStream`，然后调用 `server.createMedia` 创建流媒体服务器，返回 `media` 对象。

连接流媒体

本示例前端采用 Vue 实现，位于前端项目的 web 目录下，以下代码在 Player.vue 组件中。

1. 前端连接流媒体服务时，双通道有握手过程，使用 `@edgeros/web-mediaclient` 模块处理连接过程，参考以下示例在页面中引用模块。

```
<script type="text/javascript" src="./mediaclient.min.js"></script>
```

2. 参考以下示例，页面初始化时创建 `MediaClient` 对象。

```

this.np = new NodePlayer();
// ...
var proto = location.protocol === "http:" ? "ws:" : "wss:";
var host = `${proto}//${window.location.host}`;
var mediaClient = new MediaClient(
    host,
    (client, path) => {
        this.np.start(host + path);
    },
    { path: this.dev.path }
);

```

说明：

- 创建 `MediaClient` 对象时传入参数 `this.dev.path`（摄像头设备的流媒体服务路径）。

- `new MediaClient()` 回调函数是 `mediaClient` 对象握手成功后的回调处理，返回 `ws-flv` 流访问地址，在回调播放器中即可使用这个地址播放音视频。
- `NodePlayer` 是一款性能优秀的支持 `flv` 流媒体的播放器，本示例中使用的是试用版，可持续播放 10 分钟。`NodePlayer` 播放器的使用请参考 [NodePlayer 文档](#)。

3. 参考以下示例，可使用 `mediaClient` 打开连接。

```
startPlay: function () {
    console.log('Start play.');
    if (!this.isStarting) {
        console.log('Start.');
        this.isStarting = true;
        this.mediaClient.open(getAuth());
    }
}
```

说明：

- 调用 `mediaClient.open(auth)` 方法打开 `mediaClient` 连接，连接成功后会回调上述握手函数。
- `getAuth()` 传入的是实时更新的 `{token, srand}` 安全访问参数，关于安全机制请参考 [安全机制](#)。

4. 参考以下示例，可使用 `mediaClient` 关闭连接。

```
stopPlay: function() {
    if (this.isStarting) {
        console.log('Stop.');
        this.mediaClient.close();
    }
}

mediaClient.on('close', () => {
    console.log('MediaClient on close');
    this.np.stop();
    this.isStarting = false;
});
```

说明：

- 调用 `mediaClient.close()` 方法关闭 `mediaClient` 连接。
- 关闭连接会触发 `close` 事件，在事件中停止播放器。
- `mediaClient` 关闭后可重新建立连接。

使用 TypeScript 语言开发

本文介绍如何使用 *TypeScript* 语言开发爱智应用。

准备工作

- 完成 [环境准备](#)。
- 阅读 [创建应用](#)，了解创建应用的方法。

操作步骤

创建应用

可通过以下方式获取应用模板：

方式一

使用 git clone 命令，从 github 上克隆应用模板。

```
git clone https://github.com/edgeros/tpl-typescript.git [project_name]
```

方式二

在 Visual Studio Code 中使用应用模板进行创建，请参考 [创建应用](#) 步骤，选择 TypeScript 模版即可。

创建出的目录结构如下：

| | |
|------------------|----------------|
| └─ assets | 资源文件夹 |
| └─ src | 项目源文件 |
| └─ routers | 路由信息 |
| └─ main.ts | 程序入口 |
| └─ public | 静态页面文件 |
| └─ eslintrc.json | eslint 配置文件 |
| └─ edgeros.json | edgeros 应用配置文件 |
| └─ tsconfig.json | tsconfig 配置文件 |
| └─ package.json | 依赖包的管理 |

相对于 JavaScript 工程项目，该模板中将所有文件放入到了 src 文件目录下，需要注意 package.json 文件中相应的配置。默认情况下 tsconfig.json 中的 outDir 字段配置为 dist，因此打包后的 TypeScript 项目入口文件在 package.json 中需要做如下改变：

```
main: "dist/main.js"
```

编译 TypeScript

参考以下示例，编译 TypeScript，运行编译之后会生成对应的 JavaScript 代码文件夹 dist。

```
npm run compile
```

部署与发布应用

编译完成后，可参考部署应用和发布应用文档，完成部署与发布。

- 部署应用
- 发布应用

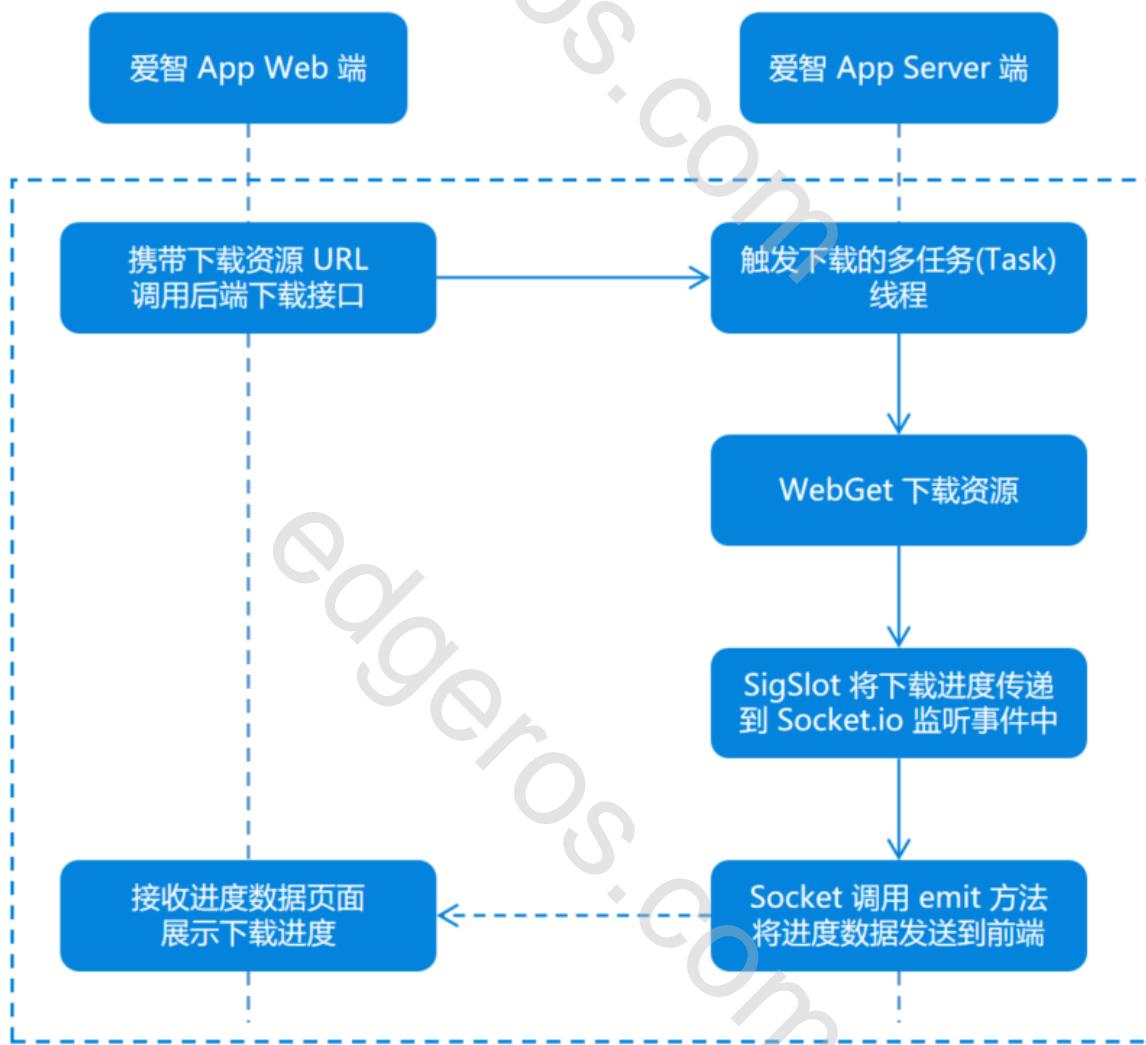
离线下载

本章主要介绍如何使用爱智设备 (Spirit 1) 实现离线下载功能。离线下载可将互联网上的资源 (如图片、音视频、文档等) 直接通过爱智服务器下载至爱智设备上，突破用户带宽速度限制，节约用户时间。

前提条件

- 准备一台爱智设备 (Spirit 1 智能边缘计算机)。
- 激活爱智设备，具体操作可参考设备包装盒内说明书。

开发原型



需求分析

- 因为是离线下载，所以爱智 APP Web 端向爱智 APP Server 端发送一个请求，爱智后端启动多任务线程完成下载任务，这时候前端就可以关闭了。
- 需要将下载的进度实时返回到前端，使用 Socket.IO 将进度推送到前端。
- 下载任务和 Socket.IO 之间需要异步通信，使用 SigSlot。

操作步骤

步骤 1：前端实现

前端需实现发送请求以及监听进度部分，在 onMounted 生命周期中接收 Socket.IO 传递过来的下载进度的数据，然后更新到对应的 DataView 中。这里给出大致的代码段，详细的内容可以参考文末附上的代码仓库。

```
// 前端以 vue3 为例实现
// home.jsx
import { defineComponent, ref, onMounted } from "vue";
import { useRouter } from "vue-router";
import { download } from "../apis/index";

export default defineComponent({
  name: "Home",
  setup(props, ctx) {
    const router = useRouter();
    const downloadList = ref([]);

    onMounted(() => {
      socketio.socket.on("progress", (data) => {
        downloadList.value.forEach((item) => {
          if (item.hash === data.hash) {
            item.progress = data.data;
            item.name = data.name;
          }
        });
        if (data.data === 100) {
          console.log(">>> download over <<<<");
        }
      });
    });
  },
});
```

步骤 2：Server 端实现

1. router 中触发下载任务

在路由开始时运行下载的线程，即可在其内部监听一个 download 的 Sigslot 任务。路由接收到下载的请求后，将参数通过 Sigslot 发送到 Task 线程。

```
// routers/rest.js
const Router = require("webapp").Router;
const SigSlot = require("sigslot");
const fs = require("fs");
```

```

const sigslot = new SigSlot("download");

/* Create router */
const router = Router.create();

// download task
const task = new Task("./download-task.js", "download-task", {
  directory: module.directory,
});

// 下载文件接口
router.post("/download", function(req, res) {
  sigslot.emit("download", {
    url: req.body.url,
    hash: req.body.hash,
  });
  res.json({
    code: 0,
    msg: "success",
  });
});

// 获取下载文件的资源列表
router.get("/download-list", function(req, res) {
  res.json({
    code: 0,
    msg: "ok",
    data: fs.readdir("./public/download"),
  });
});

/* Export router */
module.exports = router;

```

2. Task 线程中的内容

Task 实现外层 SigSlot 订阅下载任务后，使用 WebGet.file 进行资源下载，以及按照需求配置分片以及断点续传等功能。WebGet.file 的回调函数返回 WebGet 对象，可以在 WebGet 对象的 data 事件中获取计算进度的数据，再通过 SigSlot 发布到 Socket.IO 的订阅事件中。

```

// routers/download-task.js
const WebGet = require("webget");
const SigSlot = require("sigslot");
const fs = require("fs");

const sigslot = new SigSlot("download");

```

```

fs.mkdir("./public/download", 0o666, true);

// Subscribe
sigslot.slot("download", (msg) => {
    const url = msg.url;
    const fileNameArr = url.split("/");
    const fileName = fileNameArr[fileNameArr.length - 1];
    const path = `./public/download/${fileName}`;
    let totalSize = 0;

    WebGet.file(
        url,
        path,
        {
            limits: 512,
            lines: 2,
            reload: true,
        },
        (loader) => {
            loader.on("response", (info) => {
                totalSize = info.requestSize;
                // console.log(`Download begin, original=${info.originalSize},
total=${totalSize}, loaded=${info.loadedSize}`);
            });

            loader.on("data", (chunk, info) => {
                const progress = (info.completeSize / totalSize) * 100;
                // console.log(`Recv data, size=${chunk.byteLength},
offset=${info.offset}, progress=${progress}%`);
                sigslot.emit("progress", {
                    data: Math.round(progress),
                    hash: msg.hash,
                    name: fileName,
                });
            });
        });

        loader.on("end", () => {
            console.log(`Download finish, file: ${path}`);
        });

        loader.on("error", (e) => {
            console.log("Download error:", e.message);
        });
    }
);
}
);

```

```
require("iosched").forever();
```

3. 返回给前端的下载进度

在入口文件中建立 Socket 连接，然后订阅在上一步的 Task 中发布的下载进度数据，再用 Socket 发布到前端。

```
// main.js
/* Import system modules */
const WebApp = require("webapp");
const io = require("socket.io");
const SigSlot = require("sigslot");
const bodyParser = require("middleware").bodyParser;

const sigslot = new SigSlot("download");

/* Import routers */
const myRouter = require("./routers/rest");

/* Create App */
const app = WebApp.createApp();

/* Set static path */
app.use(WebApp.static("./public"));

// body parser
app.use(bodyParser.json()); // for parsing application/json
app.use(bodyParser.urlencoded()); // for parsing application/x-www-form-urlencoded

/* Set test rest */
app.use("/api", myRouter);

/* Rend test */
app.get("/temp.html", function(req, res) {
    res.render("temp", { time: Date.now() });
});

/* Start App */
app.start();

const socketio = io(app, {
    serveClient: false,
    pingInterval: 10000,
    pingTimeout: 5000,
```

```

} );

socketio.on("connection", (socket) => {
    console.log(">> socket connected <<");
    // Subscribe
    sigslot.slot("progress", (msg) => {
        // console.log('>> main download arrived:', msg.data);
        socket.emit("progress", msg);
    });
});

/* Event loop */
require("iosched").forever();

```

结果验证

发送下载请求，爱智即可完成离线下载操作，离线下载示例验证成功。



补充说明

技术点

- Task 多任务
- SigSlot 异步通信
- Socket.IO 订阅/发布
- WebGet 下载

SigSlot 和 WebGet 说明

SigSlot 和 WebGet 两个模块都是 JSRE 已经提供好的 API，可直接引用。

- SigSlot

SigSlot 是一个事件驱动的异步通信组件，支持多任务和多进程，这也是为什么我们这里选择使用 SigSlot 在 Task 中进行通信的原因。它继承自 EventEmitter，是一个典型的订阅和发布通信机制。SigSlot 的功能还远不止于此，当应用申请开启 GSS 支持后，来自同一开发供应商的应用程序可以通过 GSS 的功能互相订阅和发布消息。本示例只在同一应用中的多个线程中使用它的异步通信功能。

- WebGet

WebGet 模块用于获取 http 数据。它支持分段请求数据，以及断点续传等。通过调用 WebGet 上的 file 方法来将数据保存到文件中。可以在选项中指定数据的起始位置、每段数据的大小以及并行请求的数量。如果文件存在并且设置了 reload 为 true，WebGet 对象将检查日志并启动断点恢复过程。

源码链接

[离线下载源码](#)

智能开关

本章主要介绍如何制作一个可通过应用程序控制的智能开关，实现灯泡的远程开关、亮度调节和颜色变化等功能。

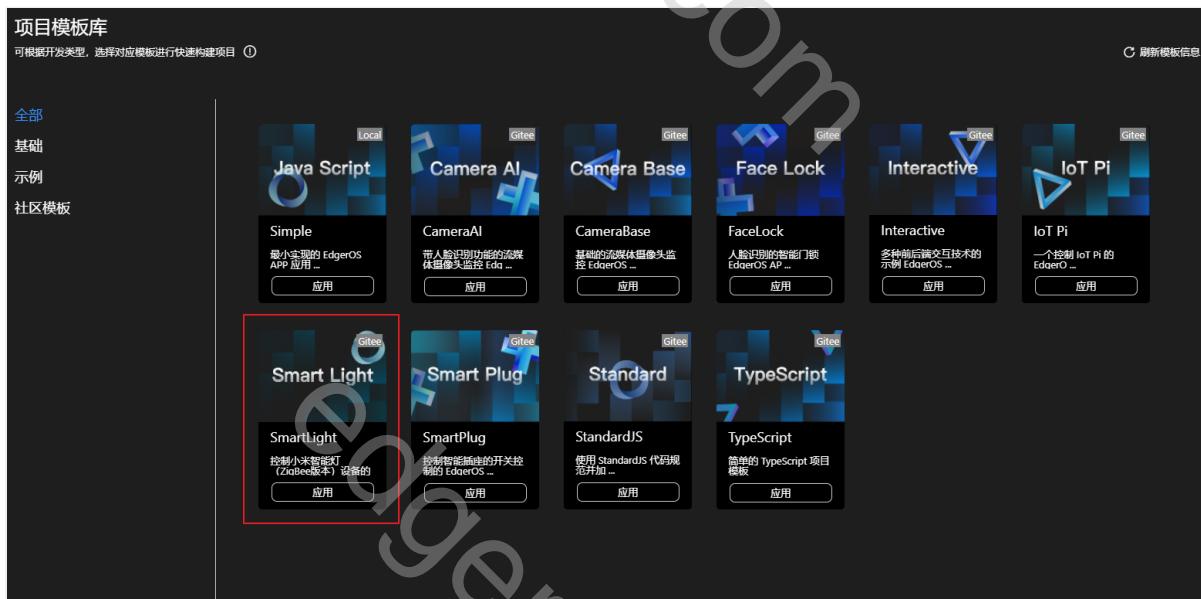
前提条件

- 准备一台爱智设备（Spirit 1 智能边缘计算机）。
- 智能灯泡（本示例选择 ZigBee 智能灯，支持调节亮度和颜色）。
- 代码模板（爱智项目模板中已内置）。

操作步骤

步骤 1：构建项目

在 VSCode 插件的项目模板库找到以下模板，直接进行快速构建。



步骤 2：项目打包

- 进入已创建项目的 web 目录，执行 `npm install` 完成依赖文件的下载，再执行 `npm run build` 命令将前端项目打包。

```
PS C:\Users\dengshihua\Desktop\app-demo-smart-light> cd .\web\
PS C:\Users\dengshihua\Desktop\app-demo-smart-light\web> npm run build

> smart-light@0.1.0 build
> vue-cli-service build
```

- 已打包的内容在 dist 目录中，如下图所示则表示打包完成。

```

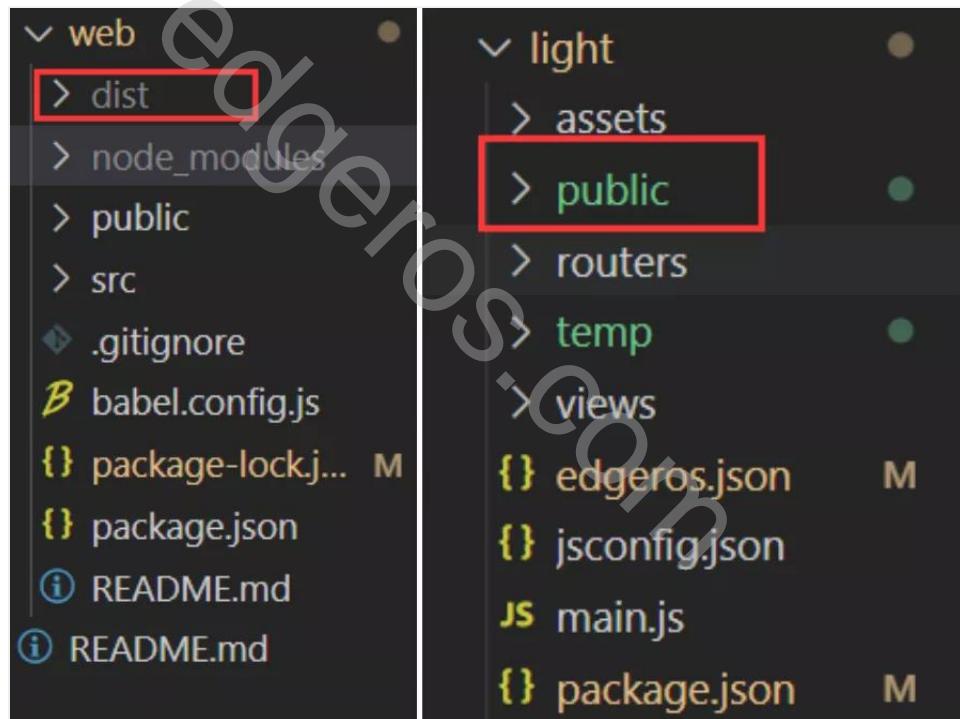
dist\js\chunk-vendors.ee4e137b.js      270.66 KiB
dist\js\app.3ffe106.js                  7.22 KiB
dist\js\chunk-3419f65c.72cad08f.js     4.69 KiB
dist\css\chunk-vendors.def3fd65.css    140.33 KiB
dist\css\app.b2aaafc8.css              0.34 KiB
dist\css\chunk-3419f65c.209ca9b9.css   0.33 KiB

Images and other types of assets omitted.

DONE Build complete. The dist directory is ready to be deployed.
INFO Check out deployment instructions at https://cli.vuejs.org/guide/deployment.html

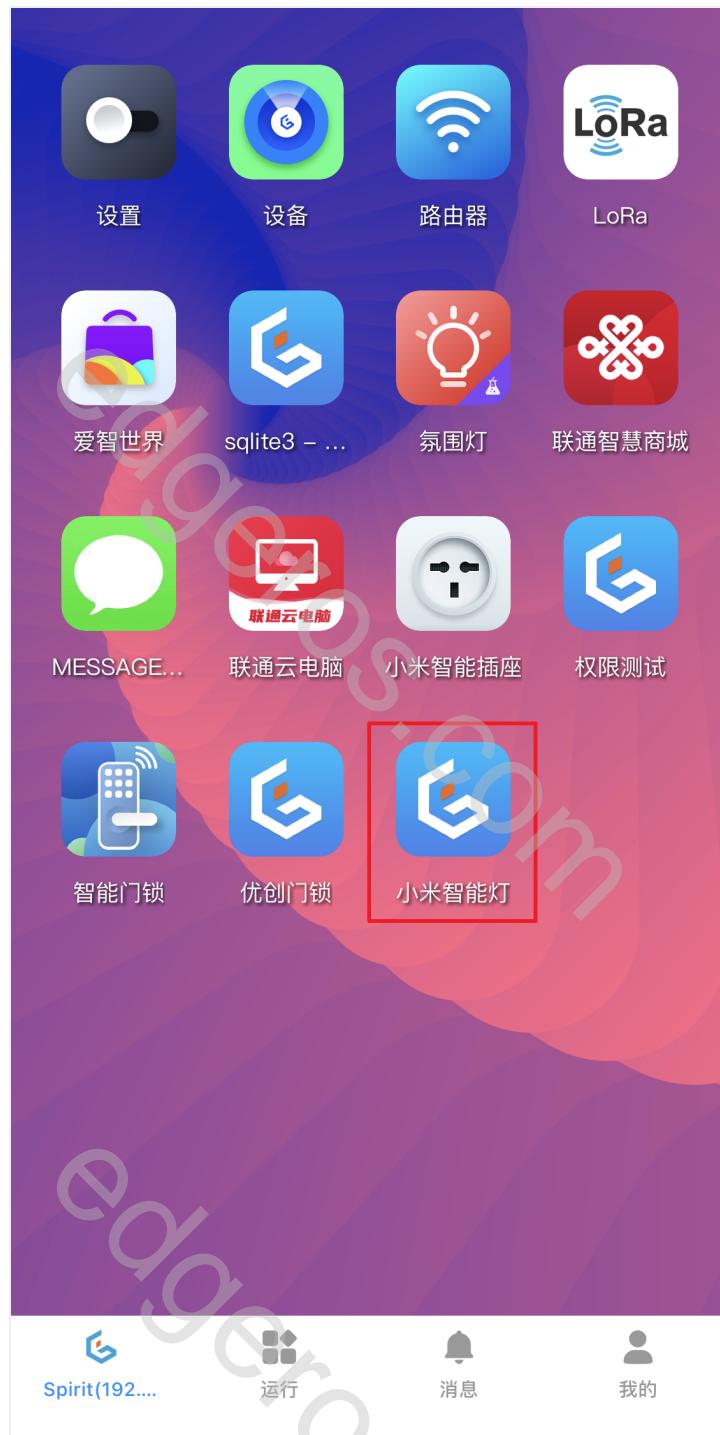
```

3. 将 dist 目录的内容复制到 public 目录中。



步骤 3：应用部署

将应用安装部署到 Spirit 1 上，登录设备即可发现小米智能灯应用已安装成功。



步骤 4：连接设备

进入应用并按照提示进行设备连接。本示例采用绿米 Aqara 智能 LED 灯泡，按照说明连续开关灯座，开关三次直到智能灯开始闪烁完成初始化，然后将设备添加到 Spirit 1 中。



注意：需要在设置-隐私设置-应用权限中为小米智能灯进行授权，否则 APP 无法控制设备。

结果验证

1. 进入小米智能灯应用，在 APP 上即可进行开关灯泡操作。



2. 调节亮度和色温之后，灯泡也随之产生亮度和颜色变化，智能开关示例验证成功。

补充说明

技术点

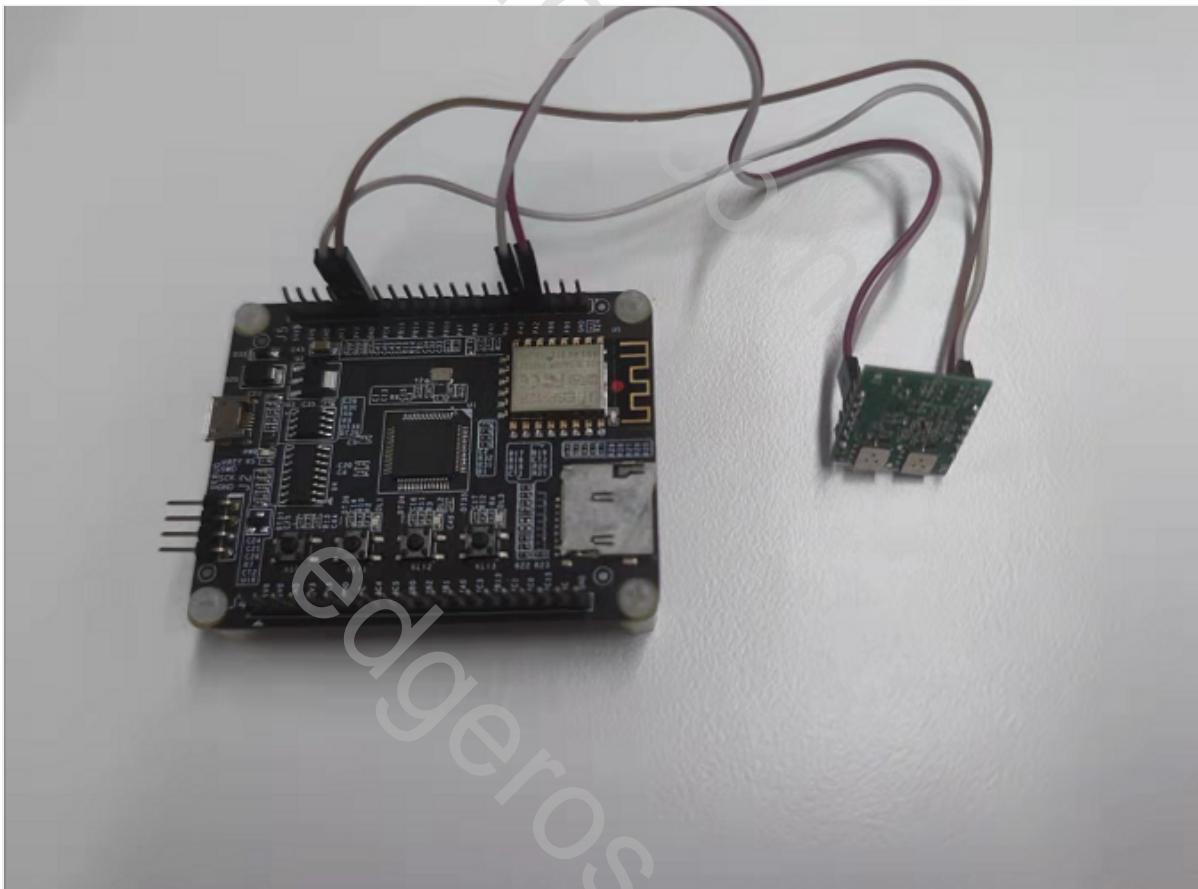
- Device 设备控制
- SDDC 设备发现控制协议

甲醛检测器

本章主要介绍基于 IoT Pi 和 MS-RTOS 制作一个可通过应用程序控制的智能甲醛检测器。

前提条件

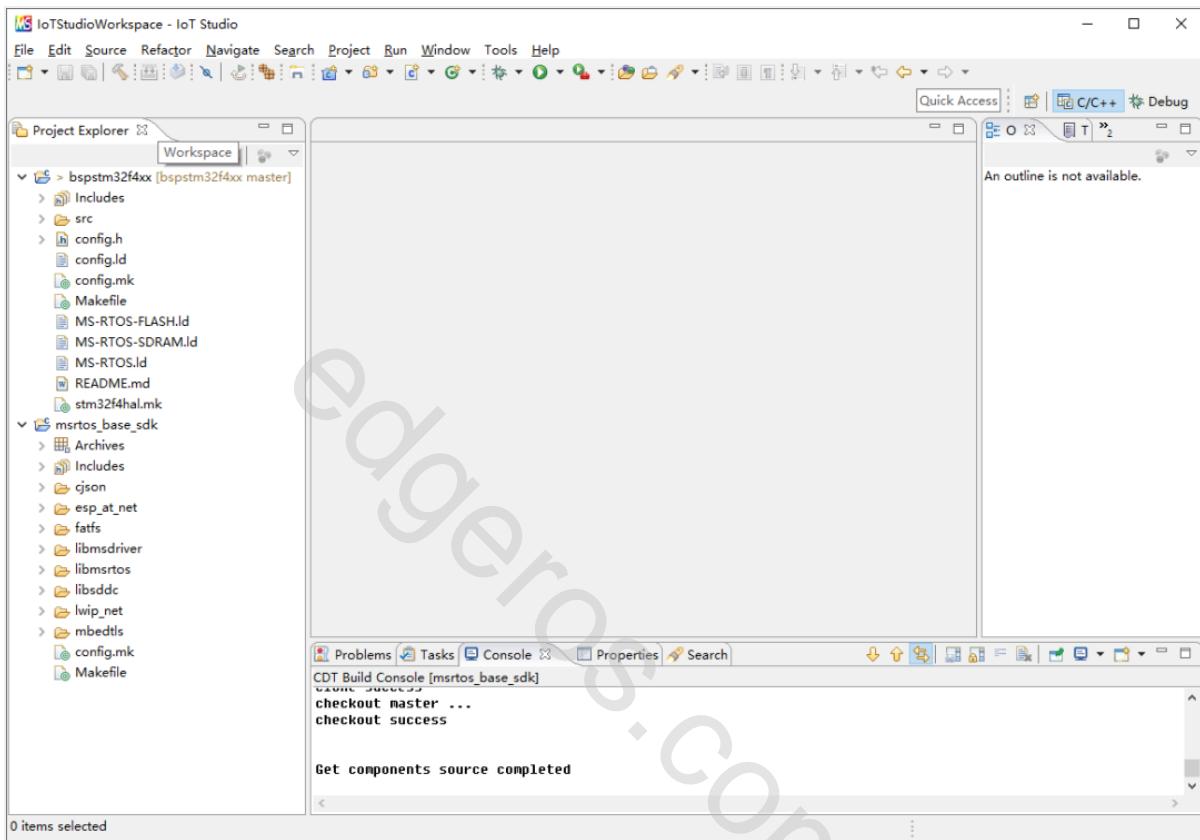
- 下载 MS-RTOS 对应的 IDE 开发环境，包括 IoT Studio 工具和 MS-RTOS AutoTester 工具。具体请参考 [官网教程](#)。
- 准备一块 IoT Pi 开发板。
- 准备一个配合烧写的 Jlink ARM 仿真器。
- 准备一个甲醛检测传感器，管脚连接如下：3V3 - 3V3、GND - GND、RXD - PA2、TXD - PA3。



操作步骤

步骤 1：工程配置

1. 在 MS-RTOS 云开发平台上完成 msrtos_base_sdk 配置和下载。
2. 在 IoT Studio 上完成 msrtos_base_sdk 工程导入和编译。
3. 在 IoT Studio 上完成 bspstm32f4xx 工程的下载和导入，如下图所示：



步骤 2：Wi-Fi AP 列表配置

`bspstm32f4xx/src/board/IOT_PI/iot_pi_init.c` 为 IoT Pi 开发板的初始化源文件，在此源文件中的 `ap_list[]` 变量用于指定手动连接模式下尝试连接到的 Wi-Fi AP 列表：

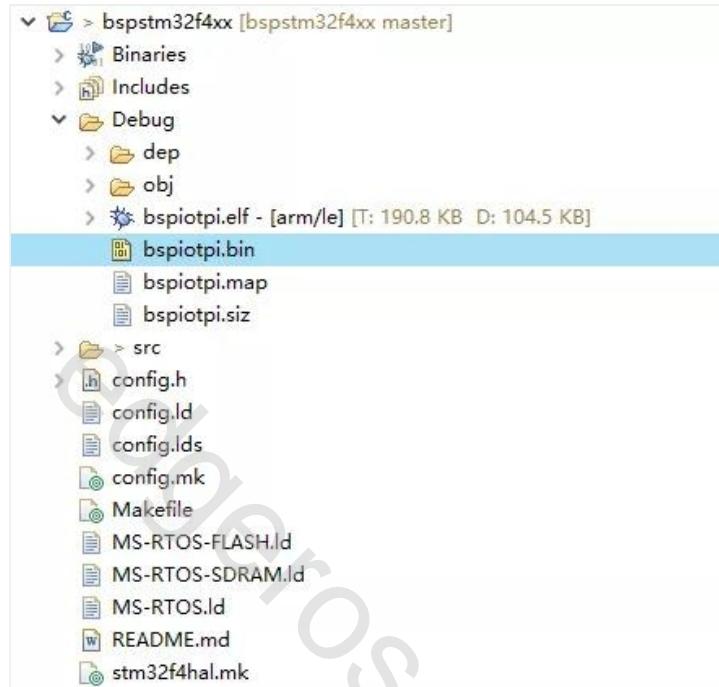
```

/**
 * WiFi AP list.
 */
static const ms_esp_at_net_ap_t ap_list[] = {
    { "EOS-00000F",      "123456789" }, // Spirit 1 的 Wi-Fi AP SSID 与密码
};

```

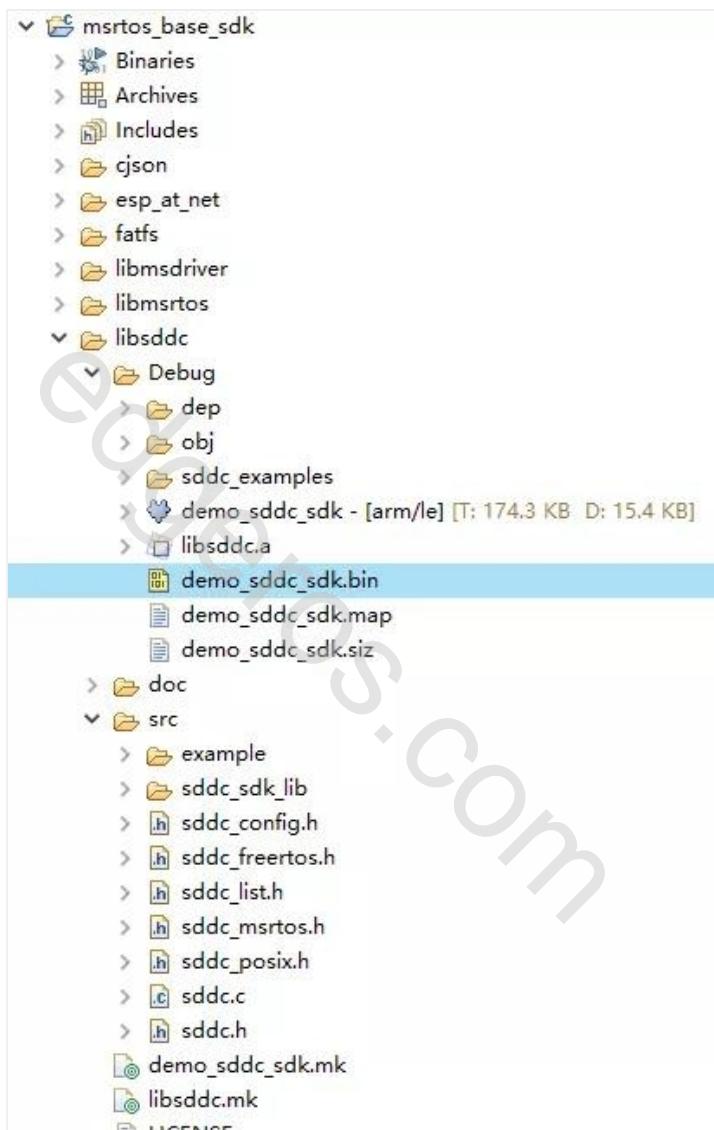
步骤 3：bspstm32f4xx 编译

选中 `bspstm32f4xx` 工程，点击编译按钮，编译完成后会在 Debug 目录生成 `bspiotpibin` 文件。



步骤 4：libsddc 配置及编译

1. 下载 libsddc，将整个目录替换 msrtos_base_sdk 工程中的 libsddc 目录。
2. 选中 msrtos_base_sdk 工程，点击编译按钮，编译 msrtos_base_sdk 工程的组件，编译完成后，会在 libsddc/Debug/sddc_examples/src/example/demo 目录生成 sddc_demo.bin 文件。



步骤 5：烧录代码

1. 使用 MS-RTOS AutoTester 烧写镜像，参考 [IoT Pi 快速入门](#) 完成 bspiotpi.bin 和 demo_sddc_sdk.bin 镜像烧写，注意不同的镜像需要烧写到不同的地址。

| 镜像 | 烧写地址 |
|-------------------|------------|
| bspiotpi.bin | 0x08000000 |
| demo_sddc_sdk.bin | 0x08040000 |

2. 按下 IoT Pi 开发板的 RESET 按键，MS-RTOS 操作系统启动后，将自动运行 0x08040000 地址处的 SDDC demo 程序：



结果验证

1. 打开嗅探器应用并进行测试，看到甲醛浓度只有 0.01，甲醛检测器验证成功。

设备通信测试

通信设备: ch2o

发送数据:

Object Array

obj: ['ch2o']

method: get

复杂数据拷贝至此:
仅支持JSON格式数据

发送数据预览:

```
{ "obj": [ "ch2o" ], "method": "get" }
```

响应数据:

```
{ "method": "report", "data": { "ch2o": 0.01 } }
```

Send

2. 还可以设定警告浓度，达到警告浓度后设备会主动上报当前甲醛浓度。



补充说明

技术点

- Device 设备控制
- SDDC 设备发现控制协议

代码解析

1. 若甲醛传感器模块的协议与本文不一致，需要修改

`libsddc/src/sddc_sdk_lib/SDDC_SDK_UART_DEV.h` 文件中的一个宏值，将 `AIR_INFO_TYPE_1` 改为 `AIR_INFO_TYPE_2` 即可：

```
#define AIR_INFO_TYPE_1    // 改为 AIR_INFO_TYPE_2

#ifndef AIR_INFO_TYPE_1
#define BUF_SIZE 40
#endif

#ifndef AIR_INFO_TYPE_2
#define BUF_SIZE 24
#endif

#ifndef AIR_INFO_TYPE_3
```

```
#define BUF_SIZE 9
#endif

#define FRAME_HEADER_AA 0xaa
#define FRAME_HEADER_2C 0x2c
#define FRAME_HEADER_E4 0xe4

#define UART_AIR_NAME "ch2o"
#define WARN_REPORT_DATA 0.06 //甲醛超标浓度

int uart_dev_init(void);
sddc_bool_t uart_dev_state_get(char *objvalue, int value_len);
sddc_bool_t uart_dev_state_set(const ms_uint64_t value);
```

2. 在 libsddc/src/sddc_sdk/lib/SDDC_SDK_UART_DEV.h 文件中定义了三种类型的数据格式，只有上面的宏定义和函数声明有用到。在 libsddc/src/sddc_sdk/lib/SDDC_SDK_UART_DEV.c 文件中，有着 uart_dev_init 的实现。包括打开串口并配置，并启用一个线程来定期获取甲醛浓度。

```
int uart_dev_init(void)
{
    fd = ms_io_open("/dev/uart2", O_RDWR, 0666);

    ms_uart_param_t param;
    param.baud = 9600;
    param.data_bits = MS_UART_DATA_BITS_8B;
    param.stop_bits = MS_UART_STOP_BITS_1B;
    param.parity = MS_UART_PARITY_NONE;

    param.flow_ctl = MS_UART_FLOW_CTL_NONE;
    param.mode = MS_UART_MODE_TX_RX;
    param.clk_pol = MS_UART_CPOL_LOW;
    param.clk_pha = MS_UART_CPHA_1EDGE;
    param.clk_last_bit = MS_UART_LAST_BIT_DISABLE;

    int ret = ms_io_ioctl(fd, MS_UART_CMD_SET_PARAM, &param);
    if (ret < 0) {
        ms_printf("[error]: set uart param failed!\n");
        ms_io_close(fd);
        return -1;
    }

    warn_data = WARN_REPORT_DATA;
    ret = ms_thread_create("t_uart",
        iot_pi_uart_dev_get_thread,
        MS_NULL,
        2048U,
        30U,
        70U,
```

```

    MS_THREAD_OPT_USER | MS_THREAD_OPT_REENT_EN,
    MS_NULL);
sddc_return_value_if_fail(ret == MS_ERR_NONE, -1);

return 0;
}

```

3. iot_pi_uart_dev_get_thread 线程中定期 3s 获取一次甲醛浓度，校验数据无误后如果超过设置的警告浓度就上报至 Spirit 1 上。

```

void iot_pi_uart_dev_get_thread()
{
    ms_uint8_t buf[BUF_SIZE];

    while(1) {
        usleep(1000 * 3000);
        ms_io_read(fd, &buf, sizeof(buf));
        if(buf[0] == FRAME_HEADER_AA) {
            if (CheckSum(&buf, (BUF_SIZE-1)) != buf[BUF_SIZE-1]) {
                printf("data checksum fail ...\n");
                break;
            } else {
                printf("data checksum success ...\n");
                uart_value_set(&buf);
            }
        }
    }
}

```

参考链接

- [MS-RTOS 云开发平台](#)
- [IoT Pi 快速入门](#)
- [libsddc](#)

智能插座

本章主要实现一个控制京鱼座智能插座(ZigBee 版)的爱智应用，以此展示在 EdgerOS 上如何开发一个控制设备的应用。

前提条件

- 准备京鱼座智能插座(ZigBee 版)。
- 准备 Spirit 1。

操作步骤

步骤 1：获取项目

爱智为开发者提供了智能插座项目模板，请通过链接 [Github](#) 或 [Gitee](#) 获取项目，项目目录结构如下：

```
app-demo-smart-plug
| -- plug: 爱智项目
| -- web: 前端项目
| -- README.MD
| -- res: 存放静态资源
```

步骤 2：创建应用

本示例采用的技术架构如下：

框架：[Vue](#)

UI：[Vant](#)

- 前端构建

1. 在 Visual Studio Code 中打开 web 文件夹，然后在终端依次执行以下命令：

```
npm install //安装项目所有依赖
npm run build //构建项目
```

2. 构建完后会生成一个 dist 文件夹，里面就是构建后的代码。

说明：

socket.io-client：用于与服务端的双向即时通讯。

vue-socket.io-extended：*socket.io-client* 在 Vue 项目里的扩展包，便于开发者开发。

@edgeros/web-sdk：爱智提供与 EdgerOS 交互的前端 API 接口，在此项目中用到了获取用户 token 的接口。

- 应用构建

1. 在 Visual Studio Code 中打开 plug 文件夹，然后在终端执行以下命令：

```
npm install //安装项目所有依赖
```

2. 将前端工程构建生成 dist 文件夹的文件复制到 plug/public 文件夹下。

说明：

device：对智能设备进行控制的 JSRE 的内置模块。

Socket.IO：服务端与客户端双向通讯的 JSRE 的内置模块。

步骤 3：部署应用

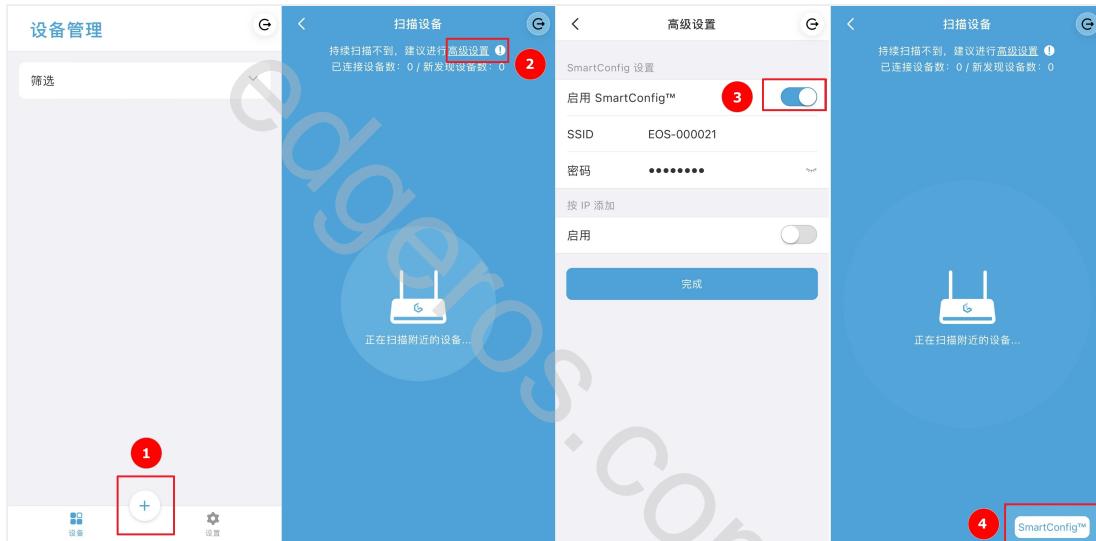
参考 [部署应用](#) 章节，将项目部署至爱智。

步骤 4：环境配置

1. 在爱智桌面点击**设备**，进入设备页面，点击 + 发现设备，进入扫描设备页面。

说明：

- 请按照京鱼座智能插座使用说明书进行操作，确保爱智能够扫描到智能插座设备。
- 设备首次连接 Wi-Fi 热点时，需在扫描设备页面点击**高级设置>启用 SmartConfig**，SmartConfig 时长一般为 30 - 60 秒。



2. 搜索到设备后点击添加，一些设备需要填写密码，添加完成后，在设备列表中可以查看新增的 plug 设备。



3. 在爱智桌面点击设置>隐私设置>设备权限，选择刚连接的智能插座设备，开启 plug 权限。



结果验证

1. 进入 plug 应用，查看已经配置好的智能插座设备，在设备上下线时，爱智自动弹出消息提示。



2. 点击设备名称进入设备详情页，可以查看到设备的控制选项，本示例通过点击按钮来实现对智能插座的开关控制。



补充说明

技术点

- Device 设备控制
- SDDC 设备发现控制协议

示例分析

智能插座后端项目中，需要使用 Device 模块用于发现和管理设备，创建设备对象，代码位于：eap-demo-smart-plug/plug/main.js。

```
var Device = require('device');
```

可参考以下示例，设置两个变量，用于保存发现的智能插座设备和当前访问的智能插座设备。

```
/* Smart plug device */
var plug = undefined;
/* Smart plug devices */
var plugs = new Map();
```

IoT Pi 控制

本章主要将实现一个控制 IoT Pi 的爱智应用，以此展示在 EdgerOS 上如何开发一个控制设备的应用，本示例同样适用于控制 IoT Pi Pro。

前提条件

- 准备 IoT Pi。
- 准备 Spirit 1。

操作步骤

步骤 1：获取项目

爱智为开发者提供了控制 IoT Pi 的项目模板，请通过链接 [Github](#) 或 [Gitee](#) 获取项目，项目目录结构如下：

```
app-demo-iotpi
| -- iotpi: 爱智项目
| -- web: 前端项目
| -- README.md
| -- res: 存放静态资源
```

步骤 2：创建应用

本示例采用的技术架构如下：

框架：[Vue](#)

UI：[Vant](#)

- 前端构建

1. 在 Visual Studio Code 中打开 web 文件夹，然后在终端依次执行以下命令：

```
npm install //安装项目所有依赖
npm run build //构建项目
```

2. 构建完后会生成一个 dist 文件夹，里面就是构建后的代码。

说明：

socket.io-client：用于与服务端的双向即时通讯。

vue-socket.io-extended：*socket.io-client* 在 Vue 项目里的扩展包，便于开发者开发。

@edgeros/web-sdk：爱智提供与 EdgerOS 交互的前端 API 接口，在此项目中用到了获取用户 token 的接口。

• 应用构建

1. 在 Visual Studio Code 中打开 文件夹，然后在终端执行以下命令：

```
npm install //安装项目所有依赖
```

2. 将前端工程构建生成 dist 文件夹的文件复制到 iotpi/public 文件夹下。

说明：

device：对智能设备进行控制的 JSRE 的内置模块。

Socket.IO：服务端与客户端双向通讯的 JSRE 的内置模块。

步骤 3：部署应用

参考 [部署应用](#) 章节，将项目部署至爱智。

步骤 4：环境配置

1. 在爱智桌面点击**设备**，进入设备页面，点击 + 发现设备，进入扫描设备页面。

说明：

- 设备首次连接 Wi-Fi 热点时，需在扫描设备页面点击**高级设置>启用 SmartConfig**，SmartConfig 时长一般为 30 - 60 秒。
- 请将 IoT Pi 按照教程通电并烧入系统，可参考 [IoT Pi 快速入门](#)。



2. 搜索到设备后点击添加，一些设备需要填写密码，添加完成后，在设备列表中可以查看新增的 IoT Pi 设备。



3. 在爱智桌面点击设置>隐私设置>设备权限，选择刚连接的 IoT Pi 设备，开启 iotpi 权限。



结果验证

1. 进入 IoT Pi 应用，查看已经配置好的 IoT Pi 设备，在设备上下线时，爱智自动弹出消息提示。



2. 点击设备名称进入设备详情页，可以查看到设备的控制选项，本示例可以对三个 led 指示灯进行操作，验证成功。



补充说明

技术点

- Device 设备控制
- SDDC 设备发现控制协议

应用目录说明

```
iotpi
|-- assets: 存放图片等一些静态资源
|-- public: 存放前端工程编译后的代码
|-- router: 存放开发者开发的 http 接口服务
|-- views: 存放页面，前后端分离开发可忽略
|-- edgeros.json: 项目的一些基础配置
|-- jsconfig.json: js 运行环境配置
|-- main.js: 应用程序入口
|-- package.json: 包依赖配置
```

示例分析

本示例后端项目中，IoT Pi 需要使用 Device 模块发现和管理设备，创建设备对象，具体代码位于 eap-demo-iotpi/iotpi/main.js。

```
var Device = require('device');
```

参考以下示例，设置两个变量，用于保存发现的 IoT Pi 设备列表和当前访问的 IoT Pi 设备。

```
/* IoT Pi device */
var iotpi = undefined;
/* IoT Pi devices */
var iotpis = new Map();
```

智能摄像头

本章以流媒体摄像头监控应用为示例，介绍 EdgerOS 多媒体应用开发。

前提条件

- 准备一台支持 ONVIF 协议和 RTSP 协议的网络摄像头。
- 准备 Spirit 1。

操作步骤

步骤 1：获取项目

爱智为开发者提供了智能摄像头项目模板，请通过链接 [Github](#) 或 [Gitee](#) 获取项目，项目目录结构如下：

```
app-demo-camera-base
| -- camera1: 爱智应用项目
| -- web: 前端项目
| -- README.md
```

步骤 2：创建应用

- 前端构建

- 前端项目使用 Vue 构建，在 Visual Studio Code 中打开 web 文件夹，然后在终端依次执行以下命令：

```
npm install //安装项目所有依赖
npm run build //构建项目
```

- 构建完后会生成一个 dist 文件夹，里面就是构建后的代码。

说明：

`@edgeros/web-sdk`：爱智提供与 EdgerOS 交互的前端 API 接口，在此项目中用到了获取用户 token 的接口。

`@edgeros/web-mediaclient`：WebMedia 客户端 API 模块，用于连接流媒体服务器并与服务器进行数据交互。

`NodePlayer.js`：播放器，可参考 [开发文档](#)。

- 应用构建

- 在 Visual Studio Code 中打开文件夹，然后在终端执行以下命令：

```
npm install //安装项目所有依赖
```

- 将前端工程构建生成 dist 文件夹的文件复制到 iotpi/public 文件夹下。

说明：

`@edgeros/jsre-onvif` : onvif 协议模块，用于发现设备和获取摄像头设备 rtsp 地址。

`@edgeros/jsre-medias` : WebMedia 服务封装模块，支持管理一组流媒体服务。

步骤 3：部署应用

参考 部署应用 章节，将项目部署至爱智。

步骤 4：环境配置

- 将网络摄像头按照说明书接入 Spirit 1，确保 onvif 功能已开启，且账号密码正确。

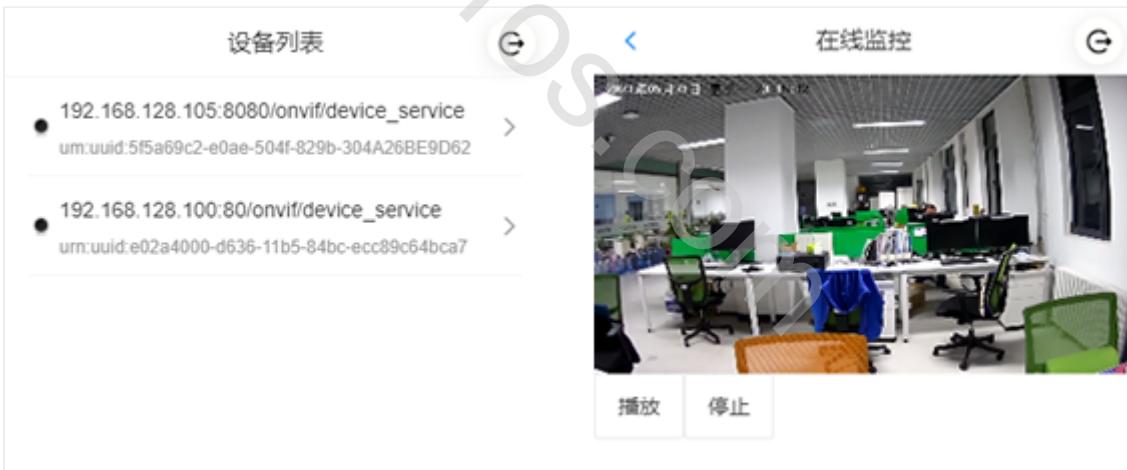
说明：

购买网络摄像头时需确定是否支持 onvif 与 rtsp 协议，我们推荐部分符合要求的网络摄像头产品供参考（网络信息可能有变，购买前请咨询客服）：

| 链接 | 描述 |
|-------|-------------|
| 网络摄像头 | 海康，球形倒挂，带云台 |
| 网络摄像头 | 桌面，带云台 |
| 网络摄像头 | 枪形，不带云台 |
| 网络摄像头 | 方形，不带云台 |

结果验证

进入智能摄像头应用，查看界面正常展示，验证成功。



补充说明

技术点

- WebMedia 流媒体服务

- [VideoOverlay 视频预览](#)
- [MediaDecoder 多媒体解码](#)

注意事项

本示例需要获取网络通信（network - 用于搜索设备）和视频流（rtsp - 获得视频流）权限，获取权限方式请参考 [SDK/ Permission](#)。

智能门锁

本章将借助 EdgerOS AI 模块实现一个支持人脸解锁的智能门锁应用。

前提条件

- 需要 EdgerOS 版本在 1.4.1 及以上。
- 准备 ESP32 SDDC 设备，具体参考 [ESP32 SDDC 设备开发](#)。
- 准备 Spirit 1。

操作步骤

步骤 1：获取项目

爱智为开发者提供了智能门锁项目模板，请通过链接 [Github](#) 或 [Gitee](#) 获取项目，项目目录结构如下：

```
app-demo-facelock
| -- facelock: 爱智应用项目
| -- README.md
```

步骤 2：构建项目

- 前端构建

本示例前端使用 Vue 构建，代码位于项目路径：app-demo-facelock/facelock/public，Vue 代码运行时同步执行，不需要预编译构建。

说明：

`@edgeros/web-sdk`: 爱智提供与 edgeros 交互的前端 API 接口，在此项目中用于获取用户 token 等信息。

`@edgeros/web-synctable`: 爱智提供的 WebSyncTable 模块前端 API 接口。

- 应用构建

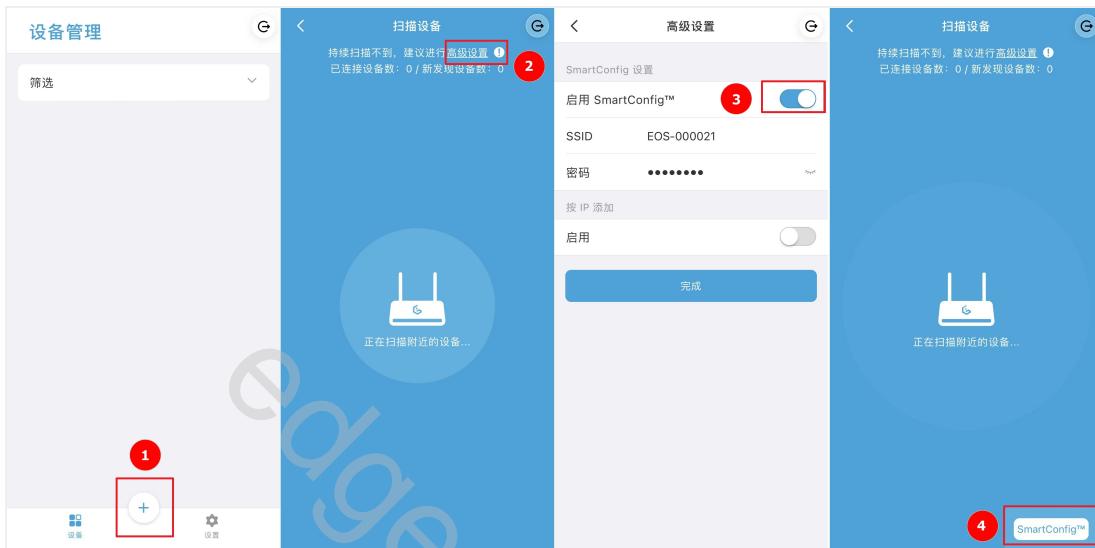
本示例应用可以直接部署，请参考 [部署应用](#) 章节将项目部署至爱智。

步骤 3：环境配置

1. 在爱智桌面点击 **设备**，进入设备页面，点击 + 发现设备，进入扫描设备页面。

说明：

- 设备首次连接 Wi-Fi 热点时，需在扫描设备页面点击 **高级设置>启用 SmartConfig**，SmartConfig 时长一般为 30 - 60 秒。
- ESP32 SDDC 设备配置请参考 [ESP32 SDDC 设备开发>配置工程](#)。



2. 搜索到设备后点击添加，一些设备需要填写密码，添加完成后，在设备列表中可以查看新增的 IoT Camera 设备。



3. 在爱智桌面点击设置>隐私设置>设备权限，选择刚连接的 IoT Camera 设备，开启 Face Lock 权限。



结果验证

在爱智桌面打开刚创建的智能门锁应用，可以查看到摄像头图像，并支持发送指令，验证成功。

Response: ok

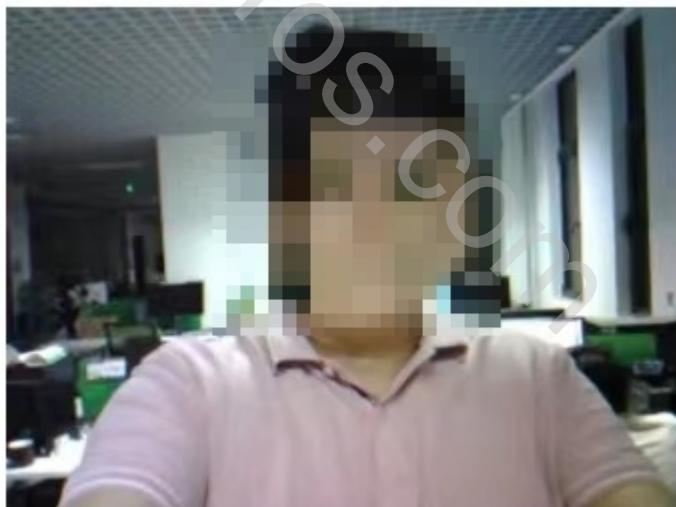
Name:

明文

SHOOT

SAVE

REMOVE



补充说明

技术点

- faceNN 人脸识别
- NCNN 神经网络计算
- Device 设备控制
- SDDC 设备发现控制协议

示例原理

智能门锁的原理是使用一个 ESP32 SDDC 设备采集人脸图像，通过 SDDC 协议将图像上传到 EdgerOS 上进行 AI 分析处理，识别无误后 EdgerOS 向 ESP32 SDDC 设备发送开锁指令，ESP32 SDDC 设备向电磁锁发出开锁信号。

本章着重介绍智能门锁应用程序的开发，关于 ESP32 SDDC 设备及设备开发的更多内容请参考 [ESP32 SDDC 设备开发](#)。

人脸识别

本章将借助 FaceNN 模块实现人脸识别功能。

前提条件

- 准备一台支持 ONVIF 协议和 RTSP 协议的网络摄像头。
- 准备 Spirit 1。

操作步骤

步骤 1：获取项目

爱智为开发者提供了人脸识别项目模板，请通过链接 [Github](#) 或 [Gitee](#) 获取项目，项目目录结构如下：

```
app-demo-camera-ai
| -- camera2: 爱智应用项目
| -- web: 前端项目
| -- README.md
```

步骤 2：创建应用

- 前端构建

1. 在 Visual Studio Code 中打开 web 文件夹，然后在终端依次执行以下命令：

```
npm install //安装项目所有依赖
npm run build //构建项目
```

2. 构建完后会生成一个 dist 文件夹，里面就是构建后的代码。

说明：

`@edgeros/web-sdk`：爱智提供与 edgeros 交互的前端 API 接口，在此项目中用于获取用户 token 等信息。

`@edgeros/web-mediaclient`：WebMedia 客户端 API 模块，用于连接流媒体服务器并与服务器进行数据交互。

`NodePlayer.js`：播放器，可参考 [开发文档](#)。

- 应用构建

1. 在 Visual Studio Code 中打开 camera2 文件夹，然后在终端执行以下命令：

```
npm install //安装项目所有依赖
```

2. 将前端工程构建生成 dist 文件夹的文件复制到 camera2/public 文件夹下。

说明：

@edgeros/jsre-onvif: onvif 协议模块，发现设备，获取摄像头设备 rtsp 地址。

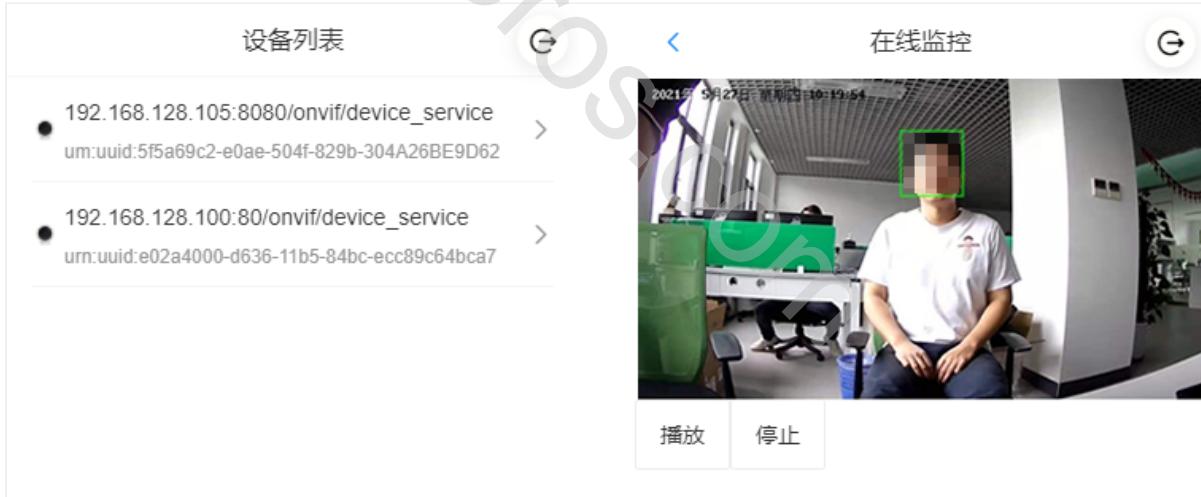
@edgeros/jsre-medias: WebMedia 服务封装模块，支持管理一组流媒体服务。

步骤 3：部署应用

参考 [部署应用](#) 章节，将项目部署至爱智。

结果验证

在爱智桌面打开刚创建的人脸识别应用，应用实现了人脸识别功能，验证成功。

**补充说明****技术点**

- faceNN 人脸识别
- SDDC 设备发现控制协议

示例原理

本示例在 [智能摄像头](#) 基础上增加人脸识别功能，代码上做了以下扩展：

- 扩展 CameraSource，从 MediaDecoder 获取视频帧数据，使用 FaceNN 模块处理视频帧数据，识别人脸信息，并通过 WebMedia 数据通道分发给客户端。
- 前端从 MediaClient 数据通道接口获取人脸识别信息，并将数据可视化绘制在视频之上的画布中。

开发者社区

本章介绍爱智为开发者提供的交流社区。

爱智开发者社区是翼辉爱智为开发者提供的开放交流空间，爱智应用开发者和智能设备开发者可以在这里提问并交流开发资源与信息，请[点此进入](#)。

The screenshot shows the EdgerOS developer community forum interface. The main content area displays three posts:

- 最新发布**: A post by user leecactus0 titled "一文看懂！在 EdgerOS 使用红黑树有多简单轻松！" (A single article to understand how to use Red-Black Trees in EdgerOS). It includes a diagram of a Red-Black Tree and has 17 replies.
- 最新回复**: A post by user leecactus0 titled "一文掌握在 EdgerOS 中使用 SQLite3 数据库引擎" (Master SQLite3 database engine usage in EdgerOS). It includes a diagram of the SQLite logo and has 174 replies.
- 精选**: A post by user ssdaddy titled "如何推广自己的产品呢" (How to promote your own products). It includes a small promotional image for a sales software and has 182 replies.

The right sidebar contains the following information:

- 爱智开发者社区** (Join button)
- 1505 社区成员** | **153 社区内容**
- A brief description: 爱智开发者平台是一个开放的物联网平台，通过爱智世界，应用开发者可以把自己的应用分发到亿万用户的设备上，硬件开发者能够把设备能力开放给海量的开发者，让优质的应用脱颖而出，为用户提供更优秀的使用体验。
- Community manager: 社区管理员 (with icons for edit, delete, and more)
- Feedback section: 你推荐你的朋友来这里加入社区么? (with smiley face rating icons from very bad to very good)
- Community announcements: 社区公告

代码示例

本章介绍爱智应用开发示例代码，供开发者参考。

以下示例推荐使用 Github 仓库，由于一些原因部分仓库无法在码云开源。

| 名称 | Github | Gitee |
|----------------------------|------------------------|-----------------------|
| EdgerOS 支持多种前后端交互技术的示例爱智应用 | Github | Gitee |
| EdgerOS 支持多种的数据库技术的示例爱智应用 | Github | Gitee |
| 一个控制（ZigBee）智能插座的爱智应用 | Github | Gitee |
| 一个控制（ZigBee）智能灯的爱智应用 | Github | Gitee |
| 一个控制 IoT Pi 的爱智应用 | Github | Gitee |
| IoT Pi 应用 Angular 版本前端代码 | Github | Gitee |
| 基础的流媒体摄像头监控爱智应用 | Github | Gitee |
| 完整的摄像头监控爱智应用 | Github | Gitee |
| 人脸解锁的智能门锁爱智应用 | Github | Gitee |
| AI 流媒体应用 | Github | Gitee |

EPM

本章介绍 EPM 仓库的使用方法。

概述

EdgerOS Package Manager (以下简称EPM) 是爱智开源社区为广大开发者免费提供的 JavaScript (以下简称JS) 软件包仓库。开发者仅需要简单的配置就可以使用 npm 命令行工具从 EPM 拉取开源软件包，或者发布自己的开源软件包。

前提条件

- 了解 [命令行开发环境](#)。
- 安装 Node.js，推荐使用最新的 Node.js LTS 发行版，请参考 [Node.js 官方网站](#)。
- 在 [爱智官网](#) 注册翼辉 ID。
- 完成 [NPM 配置](#)。

使用说明

安装软件包

方式一

登录 EPM，使用 npm 命令从 EPM 安装软件包，命令如下：

```
npm install @edgeros/welcome
```

方式二

登录 [EPM 网站](#) 浏览和搜索已经发布的开源软件包并手动下载。

发布软件包

爱智开发者可以将自己的得意之作发布到 EPM 供其它社区成员下载使用。为了保障社区的繁荣发展，开发者在发布自己的软件包之前请先阅读以下说明。

说明：

EPM 原则上仅提供与 EdgerOS® 生态相关的 JS 开源软件包。对于通用的 JS 软件包建议大家直接发布到 [npmjs.com](#) 供更多的开发者使用。

发布方式：

使用 翼辉 ID 登录 EPM 并完成软件包的开发后，按照 npm publish 的标准流程将软件包发布到 EPM。可参考 [npm publish](#)。例如：

```
cd /path/to/your/package
npm publish
```

软件包命名规范：

在 [NPM 命名规范](#)的基础上，为了方便广大开发者查找，EPM 对软件包进行简单的分类，并制定了相应的命名规范：

- @edgeros/jsre-xxx：软件包依赖于 EdgerOS® 操作系统 JSRE 的专有API，脱离 JSRE 可能无法运行。
- @edgeros/web-xxx：可以运行在浏览器中的单纯的 JS 软件包，通常与其它 @edgeros 软件包配合使用。
- 其它软件包名：建议遵循行业常识和最佳实践，例如：
 - @edgeros/cli：爱智命令行开发工具包
 - @edgeros/eslint-plugin-jsre：JSRE 的 eslint 代码辅助提示插件

注意：

- 软件包名中禁止包含侮辱、敏感词汇。
- 软件包不得包含恶意代码。
- EPM 目前不支持 *unpublish*，如果确实需要请 [联系我们](#)。
- 爱智运营团队保留在不通知发布者的情况下直接下架恶意软件包的权力。
- 商业软件开发者请谨慎发布您的软件包。

补充说明

命令行开发环境

- Windows 操作系统用户，可以使用系统自带的 *命令提示符* (*cmd.exe*) 工具。
- Linux 或 Mac 用户可以使用系统自带的 *终端* (*Terminal*) 工具。

在启动 *命令提示符* 或 *终端* 程序后，键入 `npm -v` 确定 npm 程序的版本。否则，如果系统还没有安装或正确配置 Node.js，则会提示“没有此命令”等错误信息。

翼辉 ID

如果您希望发布自己的开源软件包到 EPM，则首先需要使用翼辉 ID 登录 EPM。如果您还没有注册过翼辉 ID，请移步 [爱智官网](#) 进行注册。

配置 NPM

1. 在命令行环境中键入如下命令，并按照提示依次输入您的翼辉 ID、用户名、密码和公共电子邮件地址：

```
npm login --scope=@edgeros --registry=https://registry.epm.edgeros.com

# Username:
# Password:
# Email: (this IS public)
# Logged in as xxx to scope @edgeros on
https://registry.epm.edgeros.com/.
```

- scope=@edgeros : 限定了 NPM 的组织机构范围
 - registry=https://registry.epm.edgeros.com : 指定了 EPM 软件包注册中心的服务地址
2. 登录成功后，npm 将提示 “Logged in as xxx to scope @edgeros on <https://registry.epm.edgeros.com/>”。同时 NPM 会自动地添加您此次登录所配置的 registry , scope , token 等信息保存在 .npmrc 文件中。

您可以通过（Linux）命令 cat ~/.npmrc 查看该文件的内容，请确保文件中已经包含了如下例子中的配置信息：

```
@edgeros:registry=https://registry.epm.edgeros.com/
//registry.epm.edgeros.com/:_password=xxxxxxxxxx
//registry.epm.edgeros.com/:username=翼辉ID
//registry.epm.edgeros.com/:email=xxx@example.com
//registry.epm.edgeros.com/:always-auth=false
```

其中 @edgeros:registry=https://registry.epm.edgeros.com/ 一行说明对于 @edgeros 这个组织内的软件包将从 EPM 站点获取。以下的几行分别保存了你在 EPM 网站使用的翼辉 ID、密码以及公共可见的电子邮件地址。

许可证

EPM 欢迎社区友好的开源许可证，包括但不限于：

- MIT
- 木兰宽松许可证
- Apache License 2.0 (Apache-2.0)
- 3-clause BSD license (BSD-3-Clause)
- 2-clause BSD license (BSD-2-Clause)
- GNU Lesser General Public License (LGPL)

其他资源

本章介绍爱智为开发者提供的其他开发资源。

| 名称 | 介绍 |
|-----------|--|
| Github | Github 和 Github/ms-rtos 是爱智团队维护的开源软件包发布团队。这里将不定期发布爱智团队向社区贡献的各类开源软件包，敬请关注。 |
| 码云 | Gitee.com/ms-rtos 是爱智团队维护的开源软件包发布团队。这里将与 Github/ms-rtos 软件仓库保持同步，方便国内开发者克隆、浏览源代码仓库。 |
| EPM | EdgerOS Package Manager (以下简称EPM) 是爱智开源社区为广大开发者免费提供的 JavaScript (以下简称JS) 软件包仓库。开发者仅需要简单的配置就可以使用 npm 命令行工具从 EPM 拉取开源软件包，或者发布自己的开源软件包。 |
| npmjs.com | npmjs.com/org/edgeros 是爱智团队维护的软件包管理团队。这里将定期发布爱智团队向社区贡献的 JavaScript 软件包。此外，爱智团队也会定期将 EPM 上发布的优秀的开源软件包同步到该组织内。 @edgeros/web-sdk: 爱智应用的浏览器端开发工具包。提供了诸如安全认证、权限申请、设备状态信息查询等功能。 @edgeros/jsre-tape: 兼容 JSRE 的 tape 测试框架。 |

概述

本规范适用于爱智、爱智内置应用和第三方开发者开发的爱智应用的开发和设计。设计稿基于 375*667 屏幕尺寸进行设计，其他分辨率按照比例放大或缩小。

爱智应用的设计规范将不定时更新，请您持续关注。

edgeros.com

edgeros.com

设计准则

本章主要介绍爱智应用的设计准则。

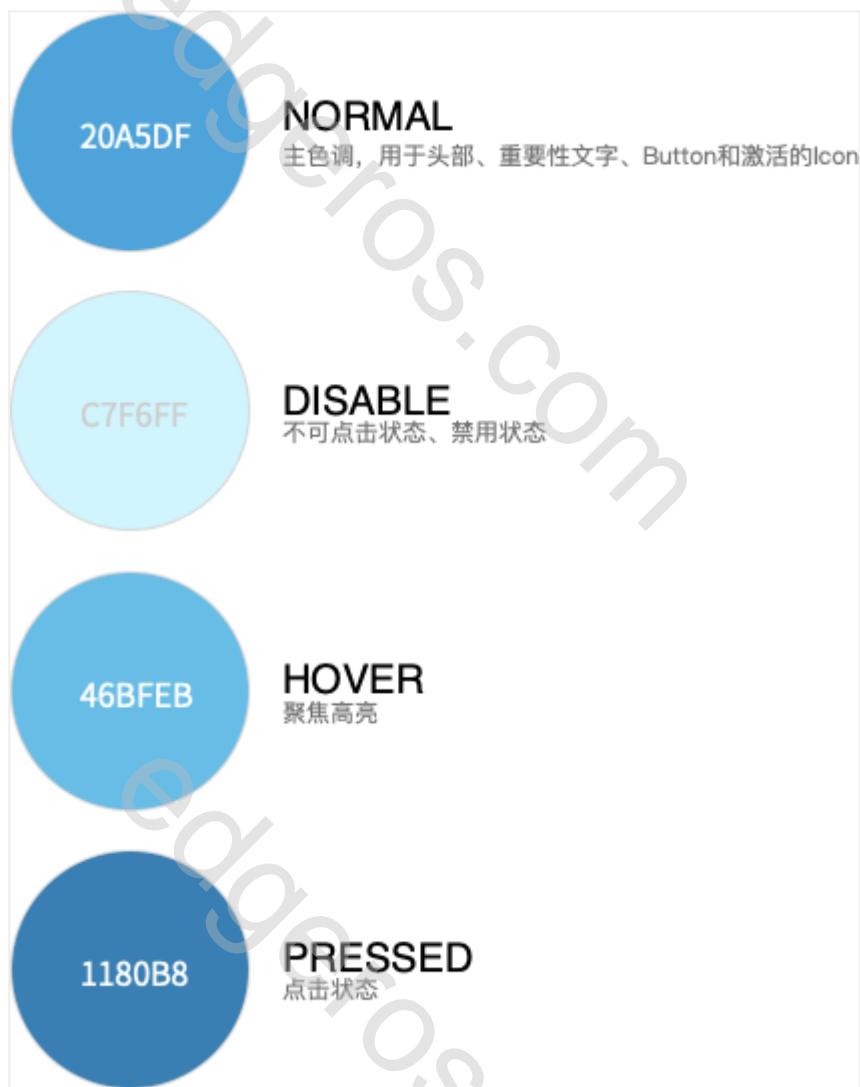
- 简化操作：交互逻辑简洁明了。
- 用户习惯：大部分用户都有固化思维，我们作为设计师应该尊重数据，尊重用户选择。
- 减少干扰：减少界面元素的不确定因素，方便用户快速找到自己想要的。
- 快速响应：加快用户读取的响应速度，能够避免从远程取数据的，就尽量避免。
- 界面友好：除了根据需求提供视觉解决方案以外，在设计的过程中适当地加入一些小细节使交互界面更加友好是设计师的职责所在。

设计风格

本章主要介绍爱智应用的设计风格规范。

标准颜色

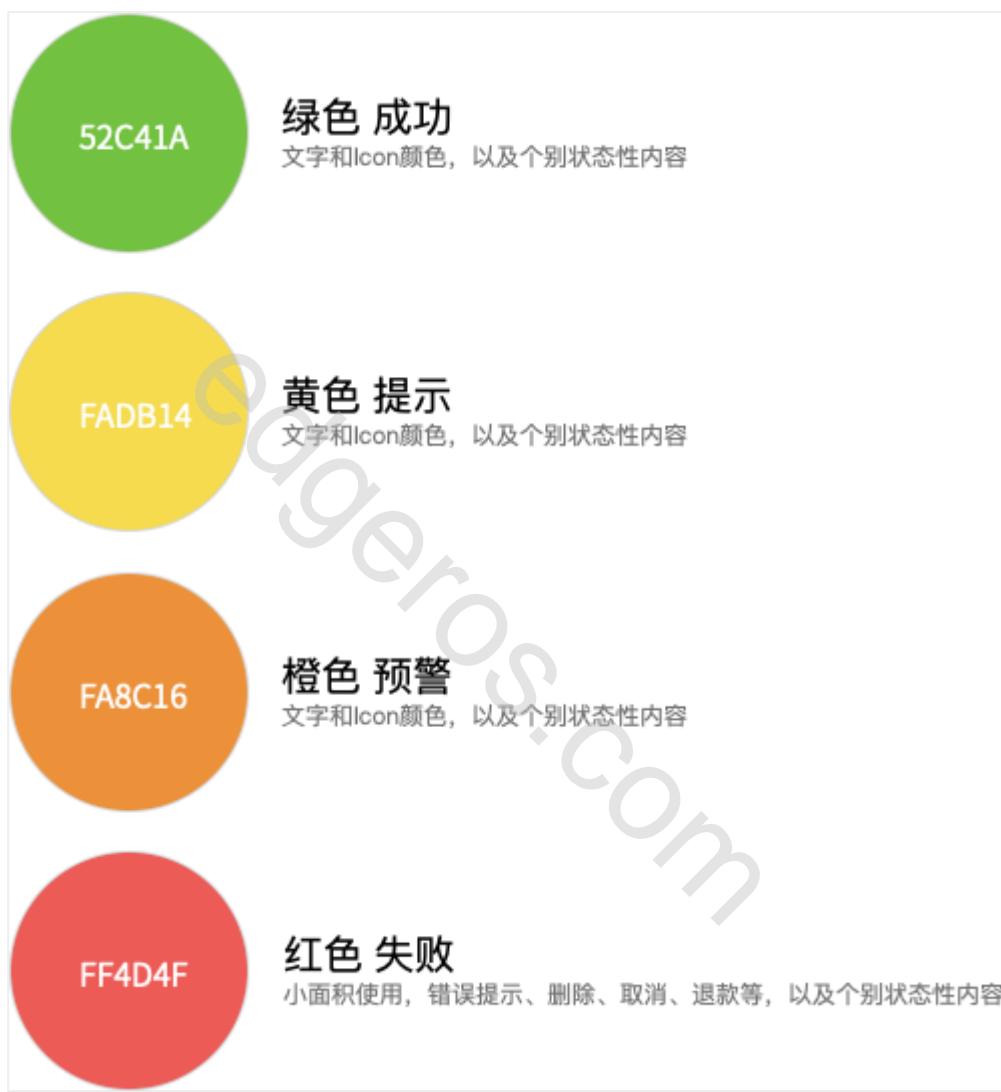
主题颜色



文字颜色



辅助颜色

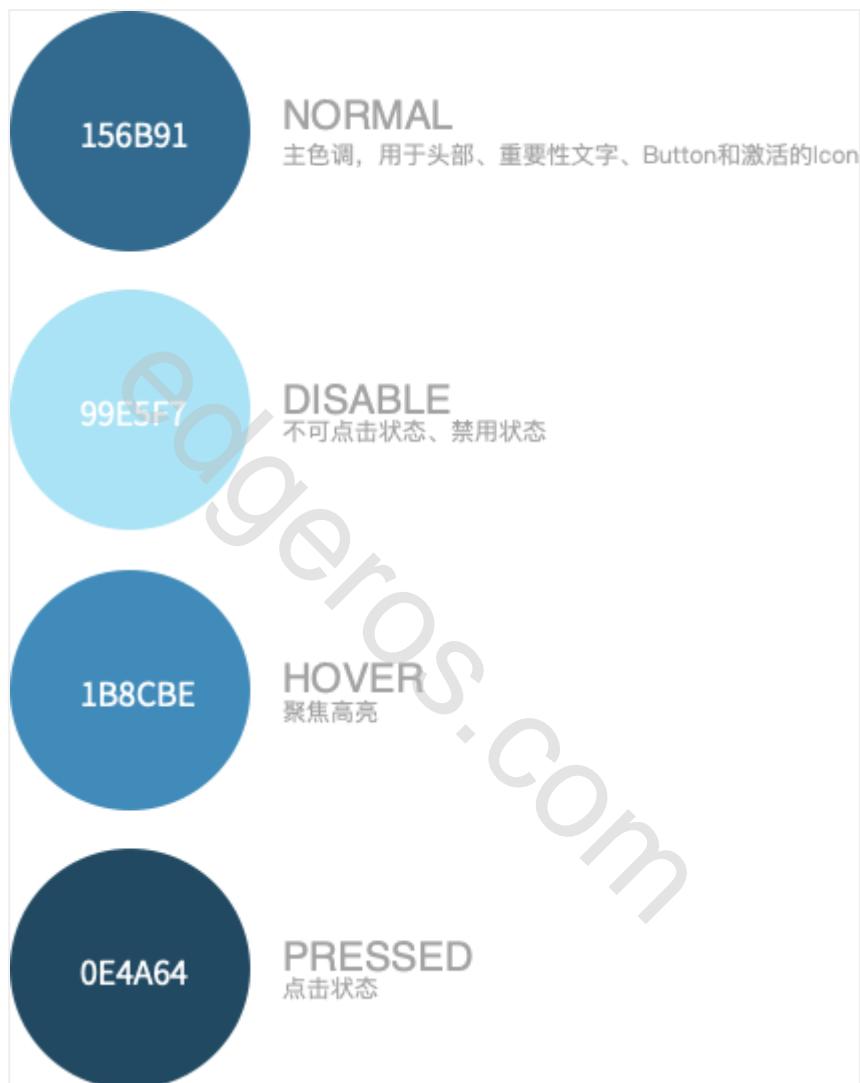


系统权限颜色



深色模式配色

主题颜色



文字颜色



辅助颜色



权限系统色彩



标准字体

字体选择

我们对字体进行统一规范，优先使用系统默认的界面字体，力求在各个操作系统下都有最佳展示效果。

Android 系统

| 语言 | 字体 | 详情 |
|----|----------------------|---|
| 中文 | 思源黑体 / Noto Sans Han | <p>Noto Sans Han 思源黑体 Regular</p> <p>ABCDEFGHIJKLM NOPQRSTUVWXYZ abcdefghijklm nopqrstuvwxyz 1234567890</p> |

英文

Roboto

Roboto**Regular**

ABCDEFGHIJKLM
NOPQRSTUVWXYZ
abcdefghijklm
nopqrstuvwxyz
1234567890

iOS 系统

| 语言 | 字体 | 详情 |
|----|-----------------------------|---|
| 中文 | 苹方 / PingFang SC | <p>PingFang SC 苹方 Regular</p> <p>ABCDEFGHIJKLM NOPQRSTUVWXYZ abcdefghijklm nopqrstuvwxyz 1234567890</p> |
| 英文 | San Francisco Pro/Helvetica | <p>San Francisco Pro Regular</p> <p>ABCDEFGHIJKLM NOPQRSTUVWXYZ abcdefghijklm nopqrstuvwxyz 1234567890</p> <p>Helvetica Regular</p> <p>ABCDEFGHIJKLM NOPQRSTUVWXYZ abcdefghijklm nopqrstuvwxyz 1234567890</p> |

Windows 系统

| 语言 | 字体 | 详情 |
|----|------------------------|----|
| 中文 | 微软雅黑 / Microsoft YaHei | |

| | | |
|----|-------|--|
| | | Microsoft YaHei 微软雅黑 Regular ABCDEFHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz nopqrstuvwxyz 1234567890 |
| 英文 | Arial | Arial Regular ABCDEFHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz nopqrstuvwxyz 1234567890 |

其它

其他智能手机可以优先使用手机品牌自带的设计字体，比如：小米（小米兰亭）、华为（汉仪旗黑/Huawei Font）、OPPO（OPPO sans）等。

目前项目使用 Vant 的方案，支持中英文单独设定，具体方法如下：

```
page {
    font-family: -apple-system, BlinkMacSystemFont, 'Helvetica Neue',
    Helvetica,     Segoe UI, Arial, Roboto, 'PingFang SC', 'miui', 'Hiragino
    Sans GB', 'Microsoft Yahei',     sans-serif;
}
```

字号大小

| | | |
|-----------|------------------------|---------|
| H1 | 24px Extra Extra Large | 行高 32px |
| H2 | 20px Extra Large | 行高 28px |
| H3 | 18px Large | 行高 26px |
| H4 | 16px Medium | 行高 24px |
| H5 | 14px Small | 行高 22px |
| H6 | 12px Extra Small | 行高 20px |
| H7 | 10px Extra Extra Small | 行高 18px |

注意：

• 重要

24字号 最大标题，根据具体情况而定。

20字号 大标题，顶部导航栏，个别重要分类名称。

18字号 标题，较为重要的标题。

• 一般

16字号 正文内容和操作按钮默认值。

• 较弱

14字号 说明性文字、消息详情。

12字号 辅助说明、时间备注。

10字号 最小字号，标签栏文字、消息数量统计，根据具体情况而定。

字体间距

间距均设置为 1 px。字体的默认间距为 0 px，但个别页面（尤其列表页面）文字显得拥挤，故设此值。

图标

- 主页图标为 60 px，圆角半径均为 12 px。
- Edger OS 的 LOGO 使用规范，黑白背景下的颜色统一（Tab Bar、启动页、登录页...）。
- App 内的图标尽可能都使用 SVG 格式的矢量图标。
- Icon 图标为 24 px 或 48 px。
- 触摸目标尺寸均为：48 px。
- 头像尺寸 56 px。
- 统一风格，统一粗细，精致纤细。
- 头像 56 px 圆形、正方形。

控件

本章主要介绍翼辉对于爱智应用控件的 UI 要求。

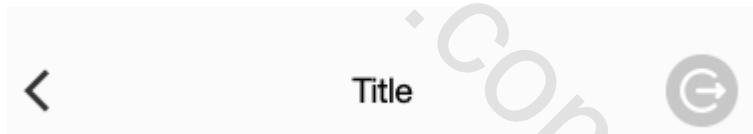
顶部状态栏

默认高度 20 px，特殊屏幕（刘海屏）以实际高度为准（深浅两种颜色模式）。



顶部导航栏

44 px，返回居左、标题居中、退出居右（全局）。

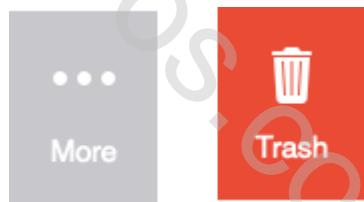


顶部 Tab 标签



列表

- 行高默认 45 px（如果列表有图标，可调整到 50 或者更大）、边框线 1 px。
- 首页的列表类前面增加 Icon 设计，不会显得枯燥单一。
- 列表过长，需要根据逻辑顺序分组展示。
- 需要补充说明的文本，尽可能增加，避免用户无法理解如何操作。
- 列表左滑删除。

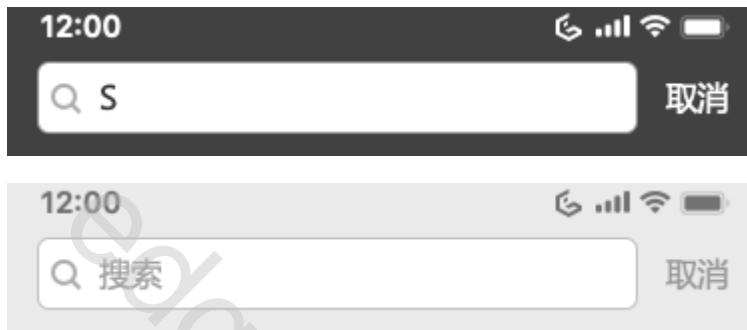


- 列表段落分割的高度为 12 px。
- 留白分割不同模块的间隔为 12 px。
- 模块内相同内容的间隔为 10 px。

输入框

- 列表输入：默认右对齐，通常适用于设置项较多的列表菜单页面。
- 整行列表输入（有标题）：默认左对齐，通常适用于设置项较少的场景，注意给标题预留一定宽度。
- 整行列表输入（无标题）：默认左对齐，通常适用于在二级页面修改名称等场景。

- Input 输出框输入（有标题）：默认左对齐，注意给标题预留一定宽度。
- Input 输出框输入（无标题）：默认左对齐，当输入文本时替换默认提示文本（请输入等）。
- 搜索输入：默认左对齐，注意给搜索 Icon 预留一定宽度。



- 登录注册：默认左对齐。
- 验证码输入：默认左对齐，右侧为获取验证码、倒计时、重新获取。



- 密码输入：有密码明文（显示）和密文（隐藏）区分。
- IP 类输入：有分隔符号，四个字段可以分别聚焦。

按钮

所有按钮的圆角半径均为 6 px。

- 重要按钮：（操作状态：默认 Normal、点击 Pressed、禁用 Disable）保存、新增、应用、删除。

页面主要操作按钮 Normal

按钮

按钮

页面主要操作按钮 Tap

按钮

按钮

页面主要操作按钮 Disable

按钮

按钮

- 次级按钮：取消。

页面次要操作按钮 Normal

页面次要操作按钮 Tap

页面次要操作按钮 Disable

- 危险操作按钮。

页面危险操作按钮 Normal

页面危险操作按钮 Normal

页面危险操作按钮 Normal

- 列表式 button：恢复出厂设置/检查更新/退出登录。





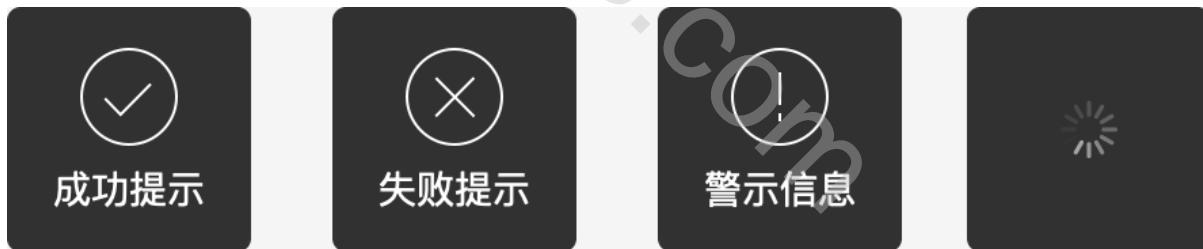
选择组件

- 底部弹出带标题。
- 时间组件（时间、时区）。
- 单选：（圆圈勾选、列表对勾）。
- 多选：方框勾选。
- Switch Button。



通知

- Toast 提示：提示状态（成功 失败 告警）、加载。



- 消息弹框。



- 可操作弹框。

操作失败

账号或密码不一致，请重试

关闭

重试

- 可输入弹框。

操作失败

账号或密码不一致，请重试

请输入密码

关闭

重试

常见问题

我可以在 EPM 发布任意 JS 软件包么？

目前 EPM 仅接收以 `@edgeros/` 为前缀的 JS 软件包。更多信息请参考《软件包发布》章节。

什么是 JSRE？它和 Node.js 有什么不同？

JSRE 是 JavaScript Runtime Environment 的缩写，是 EdgerOS® 智能操作系统中的 JS 运行时。不同于 Node.js，JSRE 目前仅支持基于 SylixOS® 内核的 EdgerOS® 操作系统。如果您的代码中依赖了特定的 JSRE API，则您的代码将无法在其它 JS 运行时中执行。

有没有像 cnpm 一样的专用的 epm 命令行软件包可以直接使用？

出于减少负担的目的，目前推荐大家直接使用 npm。

JSRE 可以使用 TypeScript 开发吗？

支持。爱智团队会陆续推出 TypeScript 项目模板，请关注[模板仓库](#)。

PC 端如何安装 EdgerOS 安全证书？

请参考：[EdgerOS 安全证书安装步骤](#)。