# DFRobot FireBeetle 2 ESP32-P4 电子工牌服务端实现

## 1. 项目概述

本项目是一套基于 DFRobot FireBeetle 2 ESP32-P4 服务端 + M5Paper/ESP32-C3 客户端的智能电子工牌系统。**DFRobot FireBeetle 2 ESP32-P4 是整个系统的核心枢纽**，承担 BLE 扫描发现、打卡判定、Web 服务管理、数据持久化、配置下发等全部服务端职责。

本文档重点介绍 **DFRobot FireBeetle 2 ESP32-P4 服务端**的实现细节，包括 BLE 扫描与打卡逻辑、SQLite 数据库设计、Web API 接口、以及作为 BLE 中心设备与客户端的交互。M5Paper 和 ESP32-C3 客户端仅作为交互对象简要说明。

项目的源码，可以直接访问：https://github.com/HonestQiao/e-badge

---

## 2. 硬件平台

服务端采用 **DFRobot FireBeetle 2 ESP32-P4** 开发板，核心硬件参数如下：

| 参数 | 规格 |
| --- | --- |
| 主控 | ESP32-P4（双核 RISC-V，400MHz） |
| 协处理器 | 集成 ESP32-C6（WiFi 6 + BLE 5） |
| Flash | 16MB |
| WiFi | 2.4GHz，802.11 b/g/n/ax |
| 蓝牙 | BLE 5.0 |
| USB | USB-Serial/JTAG |

### 2.1 ESP32-C6 协处理器

ESP32-P4 本身无内置 WiFi/BLE，FireBeetle 2 开发板通过板载 ESP32-C6 协处理器提供无线连接能力。在 Arduino 环境中，直接使用标准 `WiFi.h` 、`BLEDevice.h` 等库即可，底层由 ESP-IDF 自动通过 SPI/内部总线与 C6 通信，开发者无需关心协处理器细节。

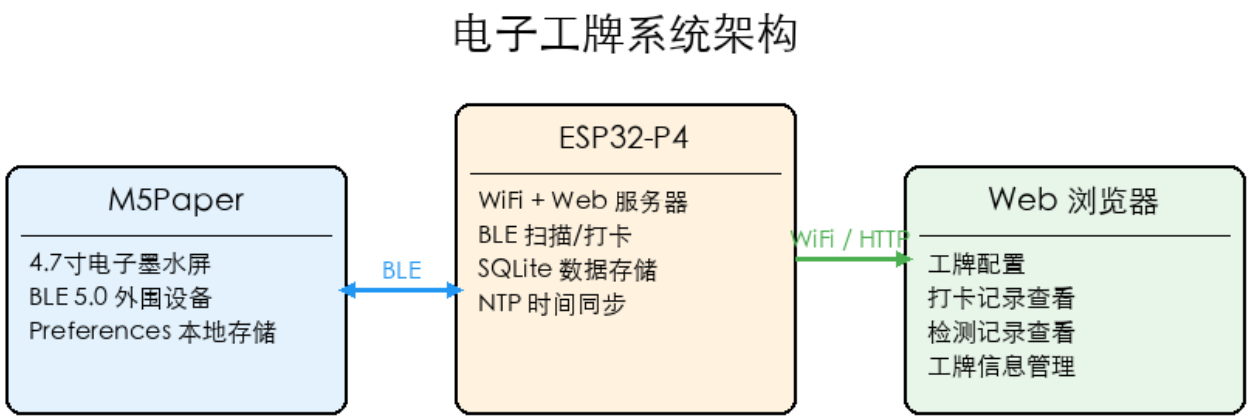
### 2.2 分区方案

ESP32-P4 固件体积较大（约 1.5MB，含 SQLite + WebServer + BLE + WiFi），**必须至少使用 8MB 分区方案**：

- 分区方案：`8M with spiffs (3MB APP/1.5MB SPIFFS)`
- 固件占用：约 44%（1.5MB / 3.4MB）
- SPIFFS 用于 SQLite 数据库存储

---

## 3. 系统架构



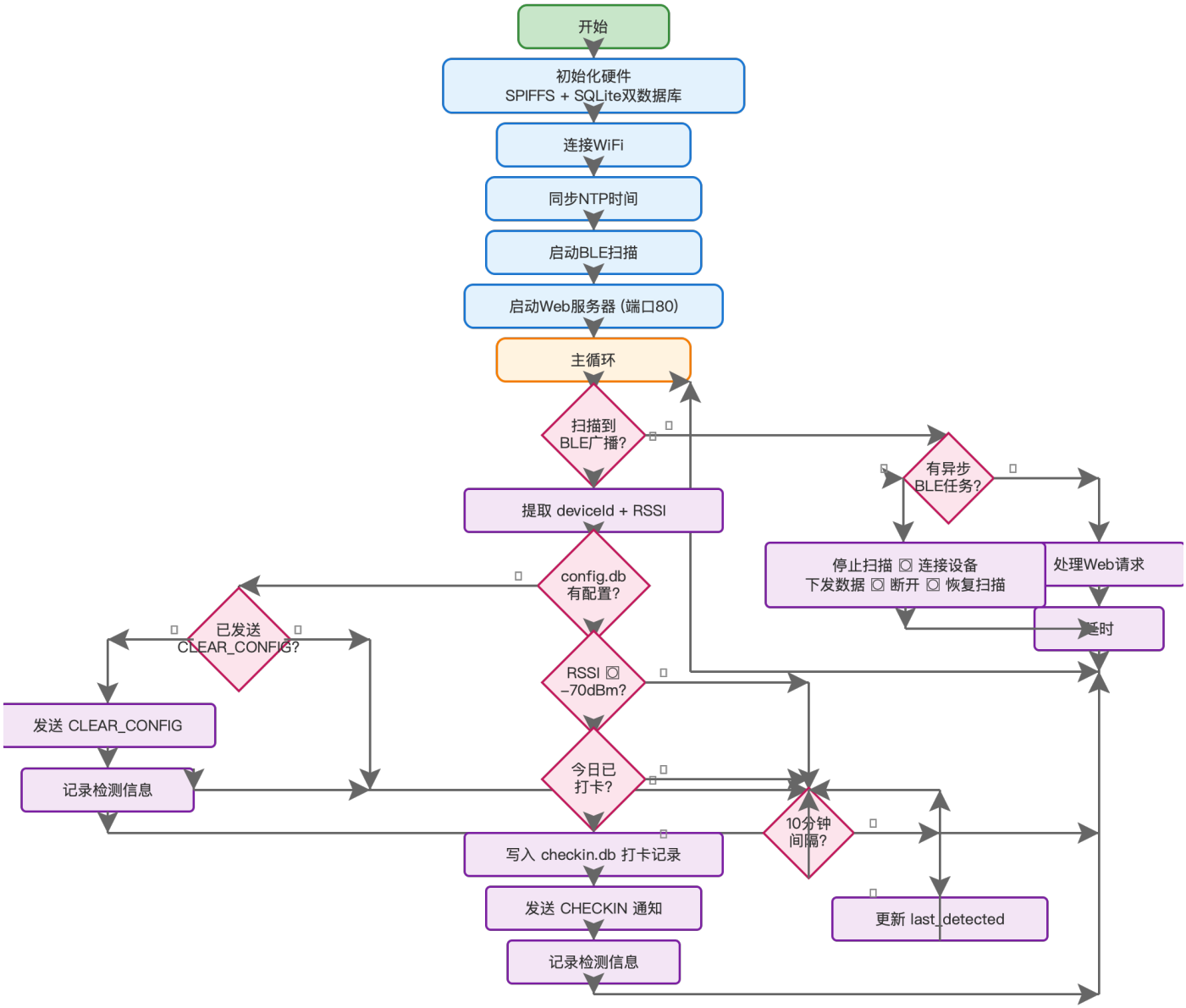

M5Paper 作为 BLE 外围设备 + 墨水屏显示终端
通过 P4 服务端与 Web 管理页面交互

ESP32-P4 在系统中承担四个核心角色：

1. **BLE 中心设备（Central）**：持续扫描周围 BLE 广播，发现 M5Paper/C3 客户端
2. **打卡判定引擎**：根据 RSSI 阈值和数据库记录，判定是否触发打卡
3. **Web 服务器**：提供管理页面和 REST API，供浏览器访问
4. **数据持久化中心**：SQLite 存储打卡记录和工牌配置

核心运行逻辑：

下图展示了 ESP32-P4 服务端从启动到主循环的完整运行逻辑，包括 BLE 扫描、打卡判定、新设备处理、异步任务调度和 Web 请求处理等核心流程：




**流程说明**：

- **初始化阶段**：上电后依次完成 SPIFFS/SQLite 初始化、WiFi 连接、NTP 时间同步、BLE 扫描和 Web 服务器启动
- **主循环**：ESP32-P4 的运行主体，持续执行以下逻辑：
  - **设备发现**：扫描到 `M5B_` / `C3B_` 广播后，提取 deviceId 和 RSSI
  - **新设备处理**：无配置的设备发送 `CLEAR_CONFIG`（每个设备仅一次）
  - **打卡判定**：有配置且 RSSI ≥ -70dBm 的设备，检查今日是否已打卡，未打卡则写入 `checkin.db` 并通知客户端
  - **已打卡更新**：已打卡设备每 10 分钟更新一次 `last_detected`
  - **异步 BLE 任务**：处理配置下发、打卡通知等需要主动连接设备的操作（扫描和连接不能同时进行）
  - **Web 请求处理**：响应浏览器 API 请求

# 4. 核心功能模块

## 4.1 BLE 扫描与设备检测

ESP32-P4 以 BLE 中心设备（Central）身份持续扫描周围广播，只关注名称前缀为 `M5B_`（M5Paper）或 `C3B_`（ESP32-C3）的设备。

### 4.1.1 扫描参数配置

```
#define BLE_SCAN_INTERVAL    100     // 扫描间隔 (ms)
#define BLE_SCAN_WINDOW      99      // 扫描窗口 (ms)
```

扫描间隔和窗口几乎相等，意味着 ESP32-P4 几乎持续在扫描状态，确保不遗漏设备广播。

### 4.1.2 扫描回调与过滤

```
class MyAdvertisedDeviceCallbacks : public BLEAdvertisedDeviceCallbacks
{
  void onResult(BLEAdvertisedDevice advertisedDevice) {
    String name = advertisedDevice.getName().c_str();
    int rssi = advertisedDevice.getRSSI();

    // 只处理 M5B_ 或 C3B_ 开头的设备
    if (!name.startsWith(DEVICE_NAME_PREFIX) &&
        !name.startsWith(DEVICE_NAME_PREFIX_C3)) return;

    String deviceId = gScanner->getDeviceIdFromName(name);
    BLEAddress address = advertisedDevice.getAddress();
    unsigned long now = getUnixTime();

    // 更新内存检测记录（按设备去重）
    gScanner->updateDetection(deviceId, name, address, rssi, now);
    // 打卡/检测逻辑
    gScanner->handleDetection(deviceId, name, rssi, address, now);
  }
};
```

**关键设计**：扫描回调中只做两件事——更新内存检测记录、触发打卡逻辑。所有耗时操作（如 BLE 连接、数据库写入）都异步处理，避免阻塞扫描循环。

### 4.1.3 内存检测记录

ESP32-P4 维护一个内存数组记录检测到的设备（最多 100 个，按设备去重）：

```
struct DetectionRecord {
  String deviceId;         // 设备 ID（8 位十六进制）
  String deviceName;       // 广播名称（如 M5B_D71F8A3C）
  BLEAddress address;      // BLE 物理地址
  int rssi;                // 信号强度
  unsigned long timestamp;// 最近一次检测的 UNIX 时间戳
  bool clearConfigSent;   // 是否已发送过清空配置通知
};
```

该数组用于：

- Web 页面的"检测记录"和"在线设备列表"实时展示
- 避免对同一设备重复发送 `CLEAR_CONFIG`
- 配置下发时查找目标设备的 BLE 地址

---

## 4.2 打卡判定逻辑

打卡判定是 ESP32-P4 的核心业务逻辑，在 `handleDetection()` 中实现。

### 4.2.1 判定流程

```
检测到设备广播
    |
    ▼
提取 deviceId + RSSI
    |
    ▼
查询数据库：该设备是否有工牌配置？
    |
    ├── 无配置 ──→ 发送 CLEAR_CONFIG（新设备处理）
    |                    └── 标记已发送，避免重复
    |
    └── 有配置 ──→ RSSI ≥ -70dBm？
                        |
                        ├── 否 ──→ 仅记录检测，不打卡
                        |
                        └── 是 ──→ 今日是否已打卡？
                                        |
                                        ├── 已打卡 ──→ 更新 last_detected
时间
                                        |                    （10 分钟间隔，
减少写操作）
                                        |
                                        └── 未打卡 ──→ 写入打卡记录到
SQLite
```

```
                                                    └── 发送
    CHECKIN 通知到设备
```

### 4.2.2 核心代码

```
void BLEScanner::handleDetection(const String& deviceId, const String&
name,
                                 int rssi, BLEAddress address, unsigned
long now) {
  // 获取今日日期
  String today = getTodayDate();

  // 从数据库查工牌配置
  BadgeConfigDBRecord config;
  bool hasConfig = pDb->getBadgeConfig(deviceId, config);

  // ===== 新设备检测：无配置记录的设备，发送清空配置通知 =====
  if (!hasConfig && rssi >= -80) {
    // 检查是否已发送过（避免重复通知）
    bool alreadySent = false;
    for (int i = 0; i < detectionCount; i++) {
      if (detections[i].deviceId == deviceId &&
detections[i].clearConfigSent) {
        alreadySent = true;
        break;
      }
    }
    if (!alreadySent) {
      // 标记为已发送，发送 CLEAR_CONFIG
      notifyClearConfig(address);
    }
  }

  // 未配置设备不打卡
  if (!hasConfig) return;

  // ===== 打卡逻辑 =====
  if (rssi >= BLE_RSSI_THRESHOLD) {  // -70dBm
    if (!pDb->hasCheckInToday(deviceId, today)) {
      // 今日首次打卡
      String checkinTime = formatTime(now);
      bool ok = pDb->addCheckIn(deviceId, config.name, config.dept,
                      config.title, config.badgeId, today, checkinTime,
now, rssi);
      if (ok) {
        // 通知客户端
        notifyCheckIn(address, checkinTime);
      }
    } else {
```

```
        // 已打卡过，仅更新最近检测时间（10 分钟间隔）
        unsigned long lastDetected = pDb->getLastDetected(deviceId,
today);
        if (now - lastDetected > 600) {  // 10 分钟 = 600 秒
          pDb->updateLastDetected(deviceId, today, now);
        }
      }
    }
  }
```

#### 4.2.3 设计要点

| 设计 | 说明 |
| --- | --- |
| RSSI 阈值 -70dBm | 约 1~2 米距离，确保设备真正靠近 ESP32-P4 才打卡 |
| 每日一次打卡 | 同一设备同一天只打卡一次，跨天自动重置 |
| 10 分钟更新间隔 | 已打卡设备仅每 10 分钟更新 `last_detected`，减少数据库写操作 |
| 异步 BLE 通知 | 扫描回调中不直接连接设备，而是设置标志位，在 `loop()` 中异步处理 |

---

### 4.3 新设备处理（CLEAR_CONFIG）

当检测到无配置记录的设备时，ESP32-P4 需要通知该设备清空本地配置并进入等待配置状态。

#### 4.3.1 处理流程

1. ESP32-P4 扫描到 `M5B_XXXXXXXX` 或 `C3B_XXXXXXXX`
2. 查询 `config.db`：无该设备的工牌配置
3. 检查内存检测记录：是否已发送过 `CLEAR_CONFIG`
4. 若未发送，设置异步标志 `pendingClearConfig`
5. `loop()` 中检测到标志，停止扫描 → 连接设备 → 发送 `CLEAR_CONFIG` → 断开连接 → 恢复扫描

#### 4.3.2 避免重复通知

通过 `clearConfigSent` 标志确保只发送一次：

```
if (!alreadySent) {
```

```
    detections[i].clearConfigSent = true;  // 标记已发送
    notifyClearConfig(address);             // 设置异步通知标志
}
```

这样即使设备持续在扫描范围内广播，也只会收到一次 `CLEAR_CONFIG`。

---

## 4.4 BLE 客户端连接与配置下发

ESP32-P4 作为 BLE 中心设备，需要主动连接客户端（M5Paper/C3）来发送通知或配置。这部分由 `BLEClientManager` 类实现。

### 4.4.1 连接流程

```
bool BLEClientManager::doConnect(BLEAddress address) {
  // BLE 扫描和连接不能同时进行，先停止扫描
  BLEScan* pScan = BLEDevice::getScan();
  if (pScan && pScan->isScanning()) {
    pScan->stop();
    delay(300);
  }

  // 重试 3 次
  for (int retry = 0; retry < 3; retry++) {
    // 先尝试 PUBLIC 地址类型
    if (pClient->connect(address, BLE_ADDR_PUBLIC)) {
      // 等待连接回调确认
      unsigned long start = millis();
      while (!connected && millis() - start < 10000) {
        delay(100);
      }
      if (connected) return true;
    }
    // 失败则尝试 RANDOM 地址类型
    if (pClient->connect(address, BLE_ADDR_RANDOM)) {
      // ...
    }
  }
  return false;
}
```

**关键设计**：

- 连接前必须停止 BLE 扫描（扫描和连接不能同时进行）
- 同时尝试 `PUBLIC` 和 `RANDOM` 两种地址类型
- 连接后等待回调确认（最长 10 秒超时）

- 最多重试 3 次

### 4.4.2 数据写入

连接成功后，向特征值写入数据：

```
bool BLEClientManager::writeToCharacteristic(BLEAddress address,
                                             const String& data,
                                             const String& extra) {
  // 1. 连接设备
  if (!doConnect(address)) return false;

  // 2. 查找服务和特征值
  BLERemoteService* pService = pClient->getService(SERVICE_UUID);
  BLERemoteCharacteristic* pChar = pService->getCharacteristic(CHARACTERISTIC_UUID);

  // 3. 写入主数据
  pChar->writeValue(data.c_str(), data.length());

  // 4. 如有额外数据（如时间同步），延迟 200ms 后写入
  if (!extra.isEmpty()) {
    delay(200);
    pChar->writeValue(extra.c_str(), extra.length());
  }

  // 5. 延迟 500ms 后断开连接
  delay(500);
  pClient->disconnect();
  return true;
}
```

### 4.4.3 支持的消息类型

| 方法 | 发送内容 | 使用场景 |
| --- | --- | --- |
| `sendConfigToAddress()` | 配置 JSON + TIME:日期 | Web 端下发工牌配置 |
| `sendCheckInNotify()` | CHECKIN:HH:MM:SS + TIME:日期 | 打卡通知 |
| `sendClearConfigNotify()` | CLEAR_CONFIG | 新设备清空配置 |

所有消息都附带 `TIME:YYYY-MM-DD` 作为额外数据，实现时间同步。

---

## 4.5 Web 服务器与 API

ESP32-P4 内置 WebServer（端口 80），提供管理页面和 REST API。

### 4.5.1 API 列表

| 路径 | 方法 | 功能 |
|---|---|---|
| `/` | GET | 管理页面（四标签：打卡记录/检测记录/工牌信息/工牌配置） |
| `/api/badge_list` | GET | 在线设备列表（30 分钟内检测到） |
| `/api/checkin_list` | GET | 打卡记录 JSON（SQLite 查询，最多 50 条） |
| `/api/detection_list` | GET | 蓝牙检测记录 JSON（内存数组） |
| `/api/stats` | GET | 今日打卡数/在线设备数/日期 |
| `/api/config` | POST | 提交工牌配置并下发到设备 |
| `/api/badge_configs` | GET | 所有已保存的工牌配置列表 |
| `/api/badge_config` | GET | 查询单个工牌配置（参数：`deviceId`） |
| `/api/delete_config` | POST | 删除工牌配置（参数：`deviceId`） |
| `/api/debug` | GET | 系统调试信息（运行时间/内存/设备数） |

### 4.5.2 打卡记录 API

```
server.on("/api/checkin_list", HTTP_GET, []() {
  CheckInDBRecord records[50];
  int count = db.getCheckInRecords(records, 50);

  String response = "[";
  for (int i = 0; i < count; i++) {
    if (i > 0) response += ",";
    response += "{";
    response += "\"name\":\"" + jsonEscape(records[i].name) + "\",";
    response += "\"dept\":\"" + jsonEscape(records[i].dept) + "\",";
    response += "\"title\":\"" + jsonEscape(records[i].title) + "\",";
    response += "\"id\":\"" + jsonEscape(records[i].badgeId) + "\",";
    response += "\"device_id\":\"" + jsonEscape(records[i].deviceId) +
"\",";
    response += "\"date\":\"" + records[i].date + "\",";
    response += "\"time\":\"" + records[i].checkinTime + "\",";
    response += "\"rssi\":" + String(records[i].checkinRssi);
```

```
    response += "}";
  }
  response += "]";
  server.send(200, "application/json", response);
});
```

由于 ArduinoJson 在 ESP32 上处理大 JSON 容易内存不足，采用手动字符串拼接方式构建响应。

#### 4.5.3 工牌配置下发 API

```
server.on("/api/config", HTTP_POST, []() {
  // 1. 解析 JSON 请求体
  String body = server.arg("plain");
  // ...

  // 2. 保存到 SQLite
  db.saveBadgeConfig(deviceId, name, dept, title, badgeId,
getUnixTime());

  // 3. 从内存检测数组查找设备 BLE 地址
  DetectionRecord* records = bleScanner.getDetectionRecords();
  int count = bleScanner.getDetectionCount();
  for (int i = 0; i < count; i++) {
    if (records[i].deviceId == deviceId) {
      targetAddress = records[i].address;
      found = true;
      break;
    }
  }

  if (!found) {
    // 设备不在蓝牙范围内
    server.send(400, ..., "{\"success\":false,...}");
    return;
  }

  // 4. 通过 BLE 发送配置到设备
  String configStr = "{\"name\":\"...\",\"dept\":\"...\",...}";
  String today = getTodayDate();
  bool sent = bleClient.sendConfigToAddress(targetAddress, configStr,
today);

  // 5. 返回结果
  server.send(200, ..., sent ? "配置已下发" : "配置已保存但下发失败");
});
```

**关键流程**：保存到数据库 → 查找 BLE 地址 → 连接设备 → 下发配置 → 返回结果。即使 BLE 下发失败，配置也已保存在数据库中，下次设备靠近时会自动处理。

## 4.6 SQLite 数据库设计

ESP32-P4 使用 SQLite 持久化存储数据。由于 ESP32 SPIFFS 对 sqlite3 的支持有限（不支持 `AUTOINCREMENT`、`UNIQUE` 等需要额外索引的操作），采用**双数据库文件**方案：

| 数据库文件 | 用途 | 表名 |
| --- | --- | --- |
| `/spiffs/checkin.db` | 打卡记录 | `checkin_records` |
| `/spiffs/config.db` | 工牌配置 | `badge_configs` |

### 4.6.1 打卡记录表

```
CREATE TABLE IF NOT EXISTS checkin_records (
  id INTEGER PRIMARY KEY,
  device_id TEXT NOT NULL,
  name TEXT,
  dept TEXT,
  title TEXT,
  badge_id TEXT,
  date TEXT NOT NULL,
  checkin_time TEXT NOT NULL,
  last_detected INTEGER,
  checkin_rssi INTEGER
);
```

### 4.6.2 工牌配置表

```
CREATE TABLE IF NOT EXISTS badge_configs (
  id INTEGER PRIMARY KEY,
  device_id TEXT NOT NULL,
  name TEXT,
  dept TEXT,
  title TEXT,
  badge_id TEXT,
  updated_at INTEGER
);
```

### 4.6.3 Database 类封装

```
class Database {
public:
  bool init();    // 初始化 SPIFFS + sqlite3 + 建表
```

```
  void close();

  // 打卡记录
  bool addCheckIn(...);                    // 新增打卡记录
  bool updateLastDetected(...);            // 更新最近检测时间
  bool hasCheckInToday(...);               // 查询今日是否已打卡
  unsigned long getLastDetected(...);      // 获取最近检测时间
  int getCheckInRecords(...);              // 获取打卡记录列表
  int getTodayCheckInCount(...);           // 获取今日打卡数

  // 工牌配置
  bool saveBadgeConfig(...);               // 保存/更新配置
  bool getBadgeConfig(...);                // 查询单个配置
  int getAllBadgeConfigs(...);             // 获取所有配置
  bool deleteBadgeConfig(...);             // 删除配置

private:
  sqlite3* dbCheckIn = nullptr;  // 打卡记录数据库
  sqlite3* dbConfig = nullptr;   // 工牌配置数据库
};
```

#### 4.6.4 SPIFFS 初始化

```
bool Database::init() {
  // SPIFFS 挂载，失败则格式化
  if (!SPIFFS.begin(true)) {
    SPIFFS.format();
    SPIFFS.begin(true);
  }

  sqlite3_initialize();

  // 分别打开两个数据库文件
  openDb("/spiffs/checkin.db", &dbCheckIn);
  openDb("/spiffs/config.db", &dbConfig);

  // 分别创建表
  exec(dbCheckIn, "CREATE TABLE IF NOT EXISTS checkin_records (...)");
  exec(dbConfig, "CREATE TABLE IF NOT EXISTS badge_configs (...)");
}
```

**为什么用双文件？** 单文件多表在 SPIFFS 上容易出现 I/O 错误（文件系统碎片、写入冲突等），将两个表拆分到独立文件可以提高稳定性。

---

## 5. Web 管理界面

ESP32-P4 提供完整的 Web 管理页面，用户通过手机或电脑浏览器访问 ESP32-P4 的 IP 地址即可使用。

## 5.1 页面结构

页面采用四标签设计，所有数据通过 AJAX 异步加载：

| 标签 | 数据来源 | 功能 |
| --- | --- | --- |
| 打卡记录 | `/api/checkin_list` | 显示历史打卡记录，含姓名、部门、时间、RSSI |
| 检测记录 | `/api/detection_list` | 显示蓝牙实时检测到的设备 |
| 工牌信息 | `/api/badge_configs` | 显示已保存的所有工牌配置，支持删除 |
| 工牌配置 | `/api/badge_list` | 选择在线设备，填写信息下发到客户端 |

**打卡记录展示**


**检测记录展示**


## 5.2 工牌配置流程

1. 点击"工牌配置"标签，页面自动调用 `/api/badge_list` 获取在线设备
2. 设备以标签形式展示，点击选择
3. 点击设备名称后，自动调用 `/api/badge_config?deviceId=xxx` 加载已有配置填充表单
4. 填写/修改姓名、部门、职位、工号
5. 点击"发送配置到设备"，POST 到 `/api/config`
6. ESP32-P4 保存到数据库，通过 BLE 下发配置，客户端自动重启
7. 客户端重启后 ESP32-P4 重新检测到，自动完成首次打卡


## 5.3 工牌信息管理

"工牌信息"标签页展示所有已配置的设备，每条记录右侧有删除按钮：

- 点击删除 → 调用 `/api/delete_config`（POST，参数 `deviceId`）
- 删除成功后自动刷新列表
- 删除操作仅移除数据库配置，不影响客户端本地存储（客户端仍显示旧工牌，直到收到新的 `CLEAR_CONFIG` 或新配置）


## 6. BLE 通信协议

ESP32-P4 与客户端（M5Paper/ESP32-C3）通过 BLE GATT 通信，基于自定义 UUID 的服务和特征值。

### 6.1 服务定义

| 属性 | UUID | 说明 |
| --- | --- | --- |
| 服务 | `0000ABCD-0000-1000-8000-00805F9B34FB` | 主服务 |
| 特征值 | `00001234-0000-1000-8000-00805F9B34FB` | 读写特征值 |

### 6.2 消息协议

| 消息 | 方向 | 格式 | 说明 |
| --- | --- | --- | --- |
| 广播名称 | 客户端 → ESP32-P4 | `M5B_XXXXXXXX` / `C3B_XXXXXXXX` | BLE 设备名称，含设备 ID |
| 工牌配置 | ESP32-P4 → 客户端 | `{"name":"...","dept":"...",...}` | 下发工牌信息 |
| 打卡通知 | ESP32-P4 → 客户端 | `CHECKIN:HH:MM:SS` | 当日首次打卡 |

| | | | |
|---|---|---|---|
| 时间同步 | ESP32-P4 → 客户端 | `TIME:YYYY-MM-DD` | 日期同步（附带在其他消息中） |
| 清空配置 | ESP32-P4 → 客户端 | `CLEAR_CONFIG` | 清空本地配置，进入等待状态 |

### 6.3 多客户端支持

ESP32-P4 同时支持两种客户端：

| 客户端 | 广播前缀 | 特点 |
|---|---|---|
| M5Paper | `M5B_` | 带 4.7 寸墨水屏，显示工牌信息 |
| ESP32-C3 | `C3B_` | 无屏幕，通过串口输出状态 |

两种客户端在打卡逻辑、配置流程上完全一致，ESP32-P4 通过广播名称前缀区分设备类型。

---

## 7. M5Paper / ESP32-C3 客户端（简要）

客户端作为 BLE 外围设备，职责相对简单：

1. **BLE 广播**：持续广播设备名称（`M5B_XXXXXXXX` 或 `C3B_XXXXXXXX`）
2. **配置接收**：通过 GATT 特征值接收配置 JSON，保存到 Preferences
3. **打卡显示**：收到 `CHECKIN:HH:MM:SS` 后更新显示/串口输出
4. **时间同步**：收到 `TIME:YYYY-MM-DD` 后判断跨天，清除打卡状态
5. **自动重启**：收到新配置后延迟 2 秒重启，让 ESP32-P4 重新检测并自动打卡

M5Paper 额外负责墨水屏 UI 展示（工牌信息、打卡状态、等待配置界面），ESP32-C3 则通过串口输出所有状态。

### M5Paper 界面效果

M5Paper 墨水屏在不同状态下的显示效果：

**等待配置界面**（首次启动或配置被清空后）：


**工牌展示界面**（已配置并打卡后）：


## 8. 完整使用流程

# M5Paper 电子工牌完整使用流程

1

## 步骤1: M5Paper 首次启动

初始化墨水屏、Preferences、BLE
检查本地是否有工牌配置
无配置 → 显示等待配置界面
开始 BLE 广播 M5B_XXXXXXXX

2

## 步骤2: P4 检测到设备

BLE 扫描发现 M5Paper
检查数据库是否有该设备配置
无配置 → 发送 CLEAR_CONFIG
M5Paper 进入等待配置状态

3

## 步骤3: Web 端配置工牌

浏览器访问 P4 管理页面
点击"工牌配置"标签
选择设备 M5B_XXXXXXXX
填写姓名/部门/职位/工号
点击"发送配置到设备"

4

## 步骤4: M5Paper 接收配置

通过 BLE 收到配置 JSON
解析并保存到 Preferences
刷新屏幕显示工牌信息
延迟 2 秒后自动重启

5

## 步骤5: P4 自动打卡

M5Paper 重启后重新广播
P4 再次检测到该设备
检查今日是否已打卡
发送 CHECKIN:HH:MM:SS 通知

6

## 步骤6: 日常使用

持续显示工牌和打卡状态
墨水屏断电后画面保持
跨天时自动清除打卡状态
提示用户重新打卡

## 步骤 1：ESP32-P4 启动

ESP32-P4 上电后：

1. 初始化 SPIFFS 和 SQLite 数据库
2. 连接 WiFi，同步 NTP 时间
3. 启动 BLE 扫描和 Web 服务器
4. 持续扫描周围 `M5B_` / `C3B_` 广播

## 步骤 2：客户端首次启动

M5Paper / C3 上电后：

1. 检查本地 Preferences 是否有工牌配置
2. **无配置**：显示等待配置状态（M5Paper 显示等待页面，C3 串口输出"等待配置"）
3. 开始 BLE 广播

## 步骤 3：ESP32-P4 检测新设备

ESP32-P4 扫描到新的客户端广播：

1. 查询数据库：无该设备的工牌配置
2. 发送 `CLEAR_CONFIG` 到客户端
3. 客户端清空本地配置，进入等待配置状态

## 步骤 4：Web 端配置

用户浏览器访问 ESP32-P4 管理页面：

1. 点击"工牌配置"标签
2. 在在线设备列表中选择目标设备
3. 点击设备名称自动加载已有配置（如有）
4. 填写姓名、部门、职位、工号
5. 点击"发送配置到设备"

## 步骤 5：ESP32-P4 下发配置

ESP32-P4 收到配置请求：

1. 保存配置到 SQLite（`config.db`）

2. 从内存检测数组查找设备的 BLE 地址
3. 连接设备 → 发送配置 JSON + `TIME:日期`
4. 客户端收到配置后保存并自动重启

### 步骤 6：自动打卡

客户端重启后重新广播，ESP32-P4 再次检测到：

1. 查询数据库：有工牌配置，今日未打卡
2. RSSI ≥ -70dBm，满足打卡条件
3. 写入打卡记录到 SQLite（`checkin.db`）
4. 发送 `CHECKIN:HH:MM:SS` 通知到客户端
5. 客户端显示/输出打卡状态

---

## 9. 代码与编译烧录

### 9.1 代码结构

完整代码：https://github.com/HonestQiao/e-badge

```
esp32-p4/
├── esp32-p4.ino          # 主程序：WiFi/NTP/WebServer初始化，主循环
├── config.h              # 配置常量：WiFi、BLE、Web、NTP
├── index_html.h          # Web 管理页面 HTML（内嵌到固件中）
├── database.h/cpp        # SQLite 数据库封装（打卡记录 + 工牌配置）
├── ble_scanner.h/cpp     # BLE 扫描：设备发现、打卡判定、新设备处理
├── ble_client.h/cpp      # BLE 客户端：连接设备、下发配置/通知
└── slave/                # ESP32-C6 协处理器固件（预置，通常不需修改）
```

### 各文件职责

| 文件 | 职责 |
| --- | --- |
| `esp32-p4.ino` | 系统入口，初始化 WiFi/NTP/数据库/BLE/Web，主循环调度 |
| `config.h` | 集中管理可配置常量 |
| `index_html.h` | 单页 Web 应用 HTML/CSS/JS，通过 PROGMEM 存储 |
| `database.cpp` | SQLite 双数据库操作，所有 SQL 语句使用预编译防止注入 |

| | |
|---|---|
| `ble_scanner.cpp` | BLE 扫描回调、打卡逻辑、新设备检测、内存记录管理 |
| `ble_client.cpp` | BLE GATT 连接、数据写入、重试机制 |

---

## 9.2 配置参数

`esp32-p4/config.h` 中的可调参数：

| 参数 | 默认值 | 说明 |
|---|---|---|
| `WIFI_SSID` | `OpenBSD` | WiFi 名称 |
| `WIFI_PASSWORD` | ... | WiFi 密码 |
| `NTP_SERVER` | `pool.ntp.org` | NTP 服务器 |
| `NTP_TIME_OFFSET` | 28800 | 北京时间 UTC+8（秒） |
| `BLE_RSSI_THRESHOLD` | -70 | 打卡 RSSI 阈值（dBm） |
| `BLE_SCAN_INTERVAL` | 100 | BLE 扫描间隔（ms） |
| `BLE_SCAN_WINDOW` | 99 | BLE 扫描窗口（ms） |
| `CHECKIN_COOLDOWN` | 5 | 打卡冷却时间（秒） |
| `DEVICE_NAME_PREFIX` | `M5B_` | M5Paper 广播名称前缀 |
| `DEVICE_NAME_PREFIX_C3` | `C3B_` | ESP32-C3 广播名称前缀 |
| `WEB_PORT` | 80 | Web 服务器端口 |
| `SERVICE_UUID` | `0000ABCD-...` | BLE 服务 UUID |
| `CHARACTERISTIC_UUID` | `00001234-...` | BLE 特征值 UUID |

---

## 9.3 编译烧录

可以使用Arduino IDE打开对应的代码文件来编译和上传，也可以使用命令行工具arduino-cli来编译和上传。

如果使用arduino-cli，命令如下：

```
cd esp32-p4
```

```
# 编译（必须指定 8MB 分区方案）
arduino-cli compile --fqbn
esp32:esp32:dfrobot_firebeetle2_esp32p4:PartitionScheme=default_8MB .
# 烧录
arduino-cli upload -p /dev/cu.usbmodem14101 --fqbn
esp32:esp32:dfrobot_firebeetle2_esp32p4:PartitionScheme=default_8MB .
```

**开发板信息：**

| 开发板 | FQBN | 串口 |
| --- | --- | --- |
| FireBeetle 2 ESP32-P4 | `esp32:esp32:dfrobot_firebeetle2_esp32p4` | `/dev/cu.usbmodem14101` |