

AI VOX3 自定义 MCP API 使用指南

概述

本指南说明如何在 `main.cpp` 中轻松添加自定义的 MCP (Model Context Protocol) 工具, 让 AI 引擎能够调用您自己的函数控制设备、读取传感器数据等。

核心概念

两个步骤

- 1. **声明工具 (Declarator)** - 向 AI 引擎注册工具的名称、描述和参数 schema
- 2. **处理调用 (Handler)** - 当 AI 引擎调用该工具时, 处理调用并返回结果

API 简介

```
// 1. 注册工具声明器 (在 InitMcpTools 中被调用)
void RegisterUserMcpDeclarator(
    const std::function<void(ai_vox::Engine*)>& declarator
);

// 2. 注册工具处理器 (当收到工具调用时被触发)
void RegisterUserMcpHandler(
    const std::string& name,
    const std::function<void(const ai_vox::McpToolCallEvent*)>& handler
);
```

优化后的AI-VOX3示例工程中, 仅需要在 `main.cpp` 中添加上述两个步骤的代码, 即可实现自定义 MCP 工具功能。优化后的代码框架如下:

```
#include <Arduino.h>
#include "ai_vox3_device.h"
#include "ai_vox_engine.h"
#include <ArduinoJson.h>

// =====MCP工具示例
// =====

/**
 * @brief MCP工具
 *
 * 该函数注册一个名为 "user.mcp_tool_example" 的MCP工具
 */
void RegisterMyCustomTool()
```

```

{
    // 注册工具声明器，定义工具的名称和描述
    RegisterUserMcpDeclarator([](ai_vox::Engine &engine)
    { engine.AddMcpTool("user.mcp_tool_example",    //
工具名称
                                "Tool description", // 工具描述
                                {
                                    {"param_name1",
                                        ai_vox::ParamSchema<bool>{
                                            .default_value =

std::nullopt, // 方向参数，默认值为空

                                    }},
                                    {"param_name2",

ai_vox::ParamSchema<int64_t>{
                                            .default_value =

std::nullopt, // 速度参数，默认值为空

                                            .min = 0,

// 最小速度为0%

                                            .max = 100,

// 最大速度为100%

                                    }}}); });

    // 注册工具处理器，收到调用时
    RegisterUserMcpHandler("user.mcp_tool_example", [])(const
ai_vox::McpToolCallEvent &ev)
    {
        // 解析参数
        const auto param_name1_ptr = ev.param<bool>("param_name1");
        const auto param_name2_ptr = ev.param<int64_t>("param_name2");
        // 检查必需参数是否存在
        if (param_name1_ptr == nullptr || param_name2_ptr == nullptr) {
            ai_vox::Engine::GetInstance().SendMcpCallError(ev.id, "Missing
required arguments: param_name1 and/or param_name2");
            return;
        }

        // 获取参数值
        bool param_name1 = *param_name1_ptr;
        int64_t param_name2 = *param_name2_ptr;

        bool result = true; // 假设执行成功

        // 创建响应
        DynamicJsonDocument doc(256);
        doc["status"] = "success";
        doc["param_name1"] = param_name1;
        doc["param_name2"] = param_name2;
        doc["description"] = result ? "success" : "failed";

        // 将 JSON 文档转换为字符串
        String jsonString;
        serializeJson(doc, jsonString);
    }
}

```

```

        // 发送响应
        ai_vox::Engine::GetInstance().SendMcpCallResponse(ev.id,
        jsonString.c_str()); });
    }

// ===== Setup 和 Loop =====
void setup()
{
    Serial.begin(115200);
    delay(500); // 等待串口初始化

    // 注册MCP工具
    RegisterMyCustomTool();

    // 初始化设备服务，包括硬件和AI引擎，必备步骤
    InitializeDevice();
}

void loop()
{
    // 处理设备服务主循环事件， 必备步骤
    ProcessMainLoop();
}

```

使用步骤

步骤 1: 定义工具注册函数

在 `main.cpp` 中定义一个函数来注册您的工具：

```

void RegisterMyCustomTool() {
    // 声明：告诉 AI 引擎这个工具的名称、描述和参数
    RegisterUserMcpDeclarator([](ai_vox::Engine& engine) {
        engine.AddMcpTool(
            "my_namespace.tool_name",          // 工具名称
            "Tool description",                // 工具描述
            {
                // 参数 schema (可选)
                {
                    "param_name",
                    ai_vox::ParamSchema<int64_t>{
                        .default_value = std::nullopt,
                        .min = 0,
                        .max = 100,
                    },
                },
            },
        );
    });
}

```

```

});

// 处理：实现工具的实际逻辑
RegisterUserMcpHandler("my_namespace.tool_name",
    [](const ai_vox::McpToolCallEvent& event) {
        // 从 event 中提取参数
        const auto param_ptr = event.param<int64_t>("param_name");

        if (param_ptr) {
            printf("Tool called with param: %" PRIu64 "\n", *param_ptr);
            // 执行实际的逻辑...
            ai_vox::Engine::GetInstance().SendMcpCallResponse(event.id, true);
        } else {
            ai_vox::Engine::GetInstance().SendMcpCallError(event.id, "Missing param");
        }
    }
);
}

```

步骤 2: 在 setup() 中调用注册函数

```

void setup() {
    // 在 InitializeDevice() 之前注册
    RegisterMyCustomTool();

    // 初始化设备 (会执行所有已注册的 declarator)
    InitializeDevice();
}

```

📖 完整示例

示例 1: 无参数工具 (获取传感器值)

```

void RegisterTemperatureSensor() {
    RegisterUserMcpDeclarator([](ai_vox::Engine& engine) {
        engine.AddMcpTool(
            "sensor.get_temperature",
            "Get current temperature from sensor",
            {} // 无参数
        );
    });

    RegisterUserMcpHandler("sensor.get_temperature",
        [](const ai_vox::McpToolCallEvent& event) {
            // 读取传感器
            int temp = readTemperatureSensor();
            printf("Temperature: %d°C\n", temp);
            ai_vox::Engine::GetInstance().SendMcpCallResponse(event.id, temp);
        }
    );
}

```

```

    }
  );
}

```

示例 2: 有参数工具 (GPIO 控制)

```

void RegisterGpioControl() {
  RegisterUserMcpDeclarator([](ai_vox::Engine& engine) {
    engine.AddMcpTool(
      "gpio.set_state",
      "Set GPIO pin to HIGH or LOW",
      {
        {
          "pin",
          ai_vox::ParamSchema<int64_t>{
            .default_value = std::nullopt,
            .min = 0,
            .max = 48,
          },
        },
        {
          "state",
          ai_vox::ParamSchema<bool>{
            .default_value = std::nullopt,
          },
        },
      }
    );
  });

  RegisterUserMcpHandler("gpio.set_state",
    [](const ai_vox::McpToolCallEvent& event) {
      const auto pin_ptr = event.param<int64_t>("pin");
      const auto state_ptr = event.param<bool>("state");

      if (pin_ptr && state_ptr) {
        pinMode(*pin_ptr, OUTPUT);
        digitalWrite(*pin_ptr, *state_ptr ? HIGH : LOW);
        printf("GPIO %lld set to %s\n", *pin_ptr, *state_ptr ? "HIGH" : "LOW");
        ai_vox::Engine::GetInstance().SendMcpCallResponse(event.id, true);
      } else {
        ai_vox::Engine::GetInstance().SendMcpCallError(event.id, "Missing
parameters");
      }
    }
  );
}

```

示例 3: 返回复杂数据

```

void RegisterJsonResponse() {
    RegisterUserMcpDeclarator([](ai_vox::Engine& engine) {
        engine.AddMcpTool(
            "device.get_status",
            "Get device status",
            {}
        );
    });

    RegisterUserMcpHandler("device.get_status",
        [](const ai_vox::McpToolCallEvent& event) {
            // 可以返回各种类型: bool, int64_t, std::string 等
            std::string status = "{\"led\":1, \"temp\":25, \"wifi\": \"connected\"}";
            ai_vox::Engine::GetInstance().SendMcpCallResponse(event.id, status);
        }
    );
}

```

参数类型支持

根据 `ai_vox::ParamSchema` 的设计，支持以下参数类型：

```

ai_vox::ParamSchema<bool>           // 布尔值
ai_vox::ParamSchema<int64_t>        // 整数
ai_vox::ParamSchema<std::string>    // 字符串

```

每个参数可以定义：

```

{
    "param_name",
    ai_vox::ParamSchema<int64_t>{
        .default_value = std::nullopt, // 可选的默认值
        .min = 0,                     // 可选的最小值
        .max = 100,                    // 可选的最大值
    },
}

```

在 main.cpp 中的实际用法

`main.cpp` 中已包含两个完整示例：

1. `RegisterCustomSensorTool()` - 无参数的传感器读取
2. `RegisterCustomGpioTool()` - 有参数的 GPIO 控制

您可以：

选项 A: 启用示例

取消注释 `setup()` 中的相应行：

```
void setup() {
    // RegisterCustomSensorTool();    // 取消注释启用
    // RegisterCustomGpioTool();      // 取消注释启用
    InitializeDevice();
}
```

选项 B: 基于示例修改

复制示例函数并修改成您需要的功能。

选项 C: 添加新工具

- 1. 在 `main.cpp` 中定义新的工具注册函数
- 2. 在 `setup()` 中调用该函数
- 3. 在初始化前完成所有注册

工具命名约定

建议工具名称遵循以下格式：

```
namespace.category.action

示例：
- my.sensor.get_temperature
- my.gpio.set_pin
- my.storage.save_data
- my.network.send_packet
```

前缀 `self`. 预留给内置工具（音量、LED 等）。

注意事项

- 1. **注册时机**：所有注册必须在 `InitializeDevice()` 之前完成
- 2. **线程安全**：如果在多线程环境中使用，确保注册在初始化时完成
- 3. **错误处理**：使用 `SendMcpCallError()` 来处理错误情况
- 4. **资源清理**：Handler 中使用的资源应该被妥善管理
- 5. **打印输出**：在 Handler 中使用 `printf()` 进行调试

调用流程图

```

AI 引擎决定调用工具
↓
ProcessMainLoop() 中的事件处理
↓
HandleMcpToolCall() 被调用
↓
内置工具处理 (音量、LED 等)
↓
用户自定义 handler 被触发
↓
Handler 执行逻辑, 调用 SendMcpCallResponse/SendMcpCallError
↓
响应返回给 AI 引擎

```

💡 常见模式

模式 1: 简单的 GET 工具

```

RegisterUserMcpHandler("my.device.get_info",
    [](const ai_vox::McpToolCallEvent& event) {
        auto result = getDeviceInfo(); // 获取信息
        ai_vox::Engine::GetInstance().SendMcpCallResponse(event.id, result);
    }
);

```

模式 2: 带参数的 SET 工具

```

RegisterUserMcpHandler("my.device.set_config",
    [](const ai_vox::McpToolCallEvent& event) {
        const auto key_ptr = event.param<std::string>("key");
        const auto value_ptr = event.param<std::string>("value");

        if (key_ptr && value_ptr) {
            setDeviceConfig(*key_ptr, *value_ptr);
            ai_vox::Engine::GetInstance().SendMcpCallResponse(event.id, true);
        } else {
            ai_vox::Engine::GetInstance().SendMcpCallError(event.id, "Invalid params");
        }
    }
);

```

模式 3: 异步操作 (启动后立即返回)

```

RegisterUserMcpHandler("my.device.start_process",
    [](const ai_vox::McpToolCallEvent& event) {

```



```
// 启动异步任务
xTaskCreate(myAsyncTask, "task", 2048, nullptr, 1, nullptr);

// 立即返回响应
ai_vox::Engine::GetInstance().SendMcpCallResponse(event.id, true);
}
);
```

📖 参考

- ai_vox3_device.h- 公共 API 定义
 - ai_vox3_device.cpp - 内部实现
 - main.cpp - 使用示例
-