用C语言构建复杂而灵活的系统架构

原创 Psyducking 嵌入式软件客栈 2025年11月14日 08:11 广东

在复杂的嵌入式系统（如多设备驱动框架、协议栈、状态机）往往需要良好的代码组织。通过结构体和函数指针，可以在C语言中模拟面向对象的三大特性：**封装**、**继承**和**多态**。

面向对象在C中的实现

封装（Encapsulation）

原理：将数据和对数据的操作封装在一起，隐藏内部实现细节。

C语言实现：

- 使用结构体存储数据成员
- 将结构体定义在 `.c` 文件中，头文件只提供前向声明
- 通过函数接口访问和操作数据

```c
// device.h - 对外接口
typedefstruct Device Device;// 不透明指针

Device* device_create(void);
voiddevice_destroy(Device* dev);
intdevice_init(Device* dev, uint32_t config);
intdevice_read(Device* dev, uint8_t* buffer, size_t len);
intdevice_write(Device* dev, const uint8_t* data, size_t len);

// device.c - 内部实现
struct Device {
    uint32_t id;
    uint32_t state;
    void* private_data;  // 私有数据
};

Device* device_create(void) {
    Device* dev = malloc(sizeof(Device));
    if (dev) {
        dev->id = 0;
        dev->state = DEVICE_STATE_INIT;
        dev->private_data = NULL;
    }
    return dev;
}
```

继承（Inheritance）

原理：子类继承父类的属性和方法，可以扩展或重写。

C语言实现：

- 将父类结构体作为子类结构体的第一个成员（结构体布局兼容）
- 子类可以安全地转换为父类指针
- 通过函数指针实现方法重写

```
// 基类
typedefstruct {
    uint32_t type;
    uint32_t state;
    int (*init)(void* self);
    int (*read)(void* self, uint8_t* buf, size_t len);
    int (*write)(void* self, constuint8_t* data, size_t len);
} BaseDevice;

// 派生类：SPI设备
typedefstruct {
    BaseDevice base;      // 基类作为第一个成员
    SPI_HandleTypeDef* spi_handle;
    GPIO_TypeDef* cs_port;
    uint16_t cs_pin;
} SPIDevice;

// 派生类：I2C设备
typedefstruct {
    BaseDevice base;      // 基类作为第一个成员
    I2C_HandleTypeDef* i2c_handle;
    uint8_t device_addr;
} I2CDevice;
```

多态（Polymorphism）

原理：同一接口可以有不同的实现，运行时根据对象类型调用相应方法。

C语言实现：

- 使用函数指针作为"虚函数表"
- 每个对象实例包含指向其方法的函数指针
- 通过函数指针调用，实现运行时多态

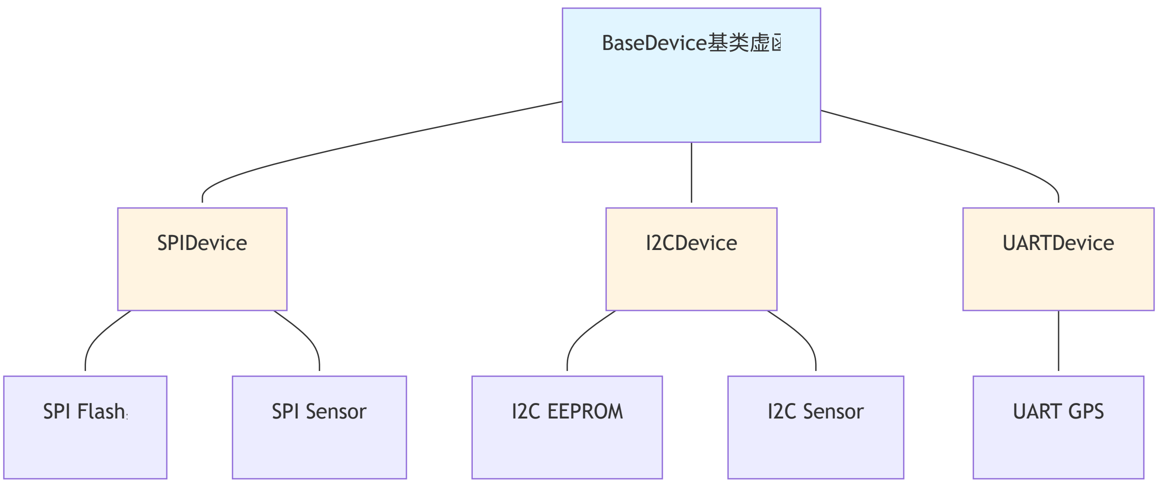
```
// 基类方法（虚函数）
intbase_device_read(void* self, uint8_t* buf, size_t len) {
    BaseDevice* dev = (BaseDevice*)self;
    // 基类默认实现或抽象方法
```

```c
    return-1;  // 未实现
}

// SPI设备的方法实现
intspi_device_read(void* self, uint8_t* buf, size_t len) {
    SPIDevice* spi_dev = (SPIDevice*)self;
    // SPI特定的读取实现
    HAL_GPIO_WritePin(spi_dev->cs_port, spi_dev->cs_pin, GPIO_PIN_RESET);
    HAL_SPI_Receive(spi_dev->spi_handle, buf, len, HAL_MAX_DELAY);
    HAL_GPIO_WritePin(spi_dev->cs_port, spi_dev->cs_pin, GPIO_PIN_SET);
    return len;
}

// I2C设备的方法实现
inti2c_device_read(void* self, uint8_t* buf, size_t len) {
    I2CDevice* i2c_dev = (I2CDevice*)self;
    // I2C特定的读取实现
    HAL_I2C_Master_Receive(i2c_dev->i2c_handle,
                          i2c_dev->device_addr << 1,
                          buf, len, HAL_MAX_DELAY);
    return len;
}

// 多态调用
intdevice_read(BaseDevice* dev, uint8_t* buf, size_t len) {
    return dev->read(dev, buf, len);  // 通过函数指针调用
}
```

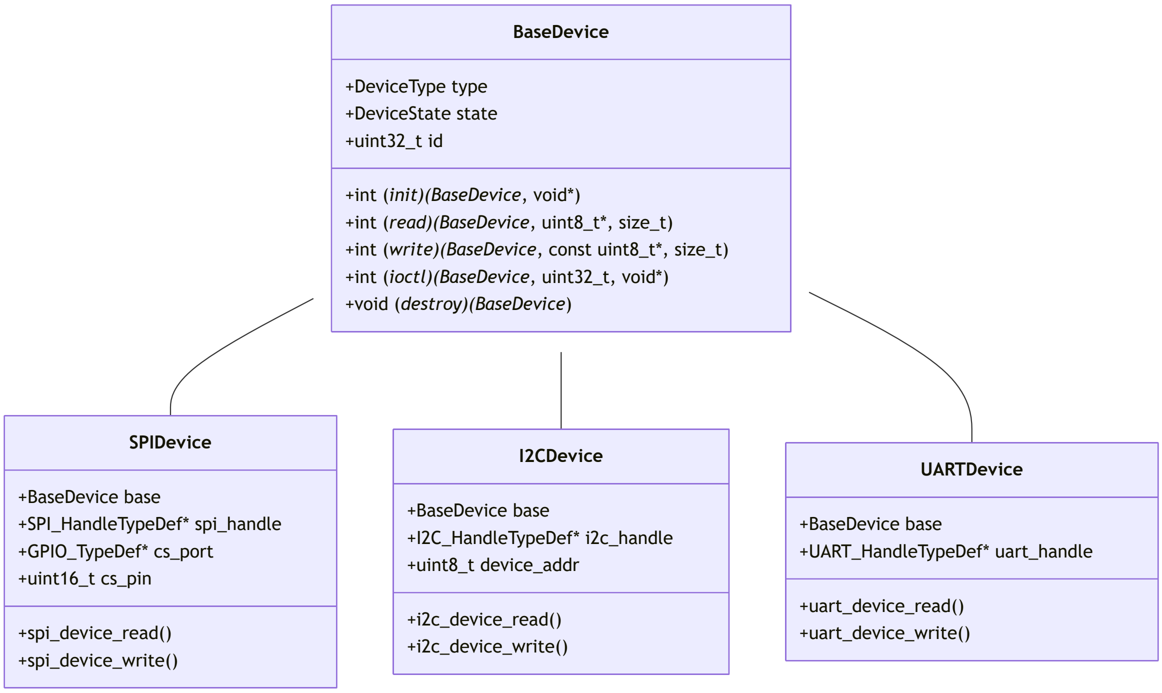
设备驱动框架设计

框架架构设计

设计一个通用的设备驱动框架，支持多种通信接口（SPI、I2C、UART），并可以轻松扩展新的设备类型。



类关系图



核心数据结构定义

```
// device_driver.h
#ifndef DEVICE_DRIVER_H
#define DEVICE_DRIVER_H

#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>

// 设备状态枚举
typedefenum {
    DEVICE_STATE_UNINIT = 0,
    DEVICE_STATE_INIT,
    DEVICE_STATE_READY,
    DEVICE_STATE_BUSY,
    DEVICE_STATE_ERROR
} DeviceState;

// 设备类型枚举
typedefenum {
    DEVICE_TYPE_SPI = 0,
    DEVICE_TYPE_I2C,
    DEVICE_TYPE_UART,
    DEVICE_TYPE_MAX
} DeviceType;

// 前向声明
typedefstruct BaseDevice BaseDevice;
```

```
// 基类结构体（虚函数表 + 数据成员）
struct BaseDevice {
    // 数据成员
    DeviceType type;
    DeviceState state;
    uint32_t id;
    uint32_t error_code;

    // 虚函数表（函数指针）
    int (*init)(BaseDevice* self, void* config);
    int (*deinit)(BaseDevice* self);
    int (*read)(BaseDevice* self, uint8_t* buffer, size_t length);
    int (*write)(BaseDevice* self, constuint8_t* data, size_t length);
    int (*ioctl)(BaseDevice* self, uint32_t cmd, void* arg);
    void (*destroy)(BaseDevice* self);

    // 私有数据指针（用于存储派生类特定数据）
    void* private_data;
};

// 公共接口函数
BaseDevice* device_create(DeviceType type, void* config);
voiddevice_destroy(BaseDevice* device);
intdevice_init(BaseDevice* device, void* config);
intdevice_read(BaseDevice* device, uint8_t* buffer, size_t length);
intdevice_write(BaseDevice* device, const uint8_t* data, size_t length);
intdevice_ioctl(BaseDevice* device, uint32_t cmd, void* arg);
DeviceState device_get_state(BaseDevice* device);
constchar* device_get_type_string(BaseDevice* device);

#endif // DEVICE_DRIVER_H
```

SPI设备实现

```
// spi_device.h
#ifndef SPI_DEVICE_H
#define SPI_DEVICE_H

#include "device_driver.h"
#include "stm32f4xx_hal.h"

// SPI设备配置结构体
typedefstruct {
    SPI_HandleTypeDef* spi_handle;
    GPIO_TypeDef* cs_port;
    uint16_t cs_pin;
    uint32_t timeout_ms;
```

```
} SPIConfig;

// SPI设备结构体（继承自BaseDevice）
typedefstruct {
    BaseDevice base;           // 基类作为第一个成员
    SPI_HandleTypeDef* spi_handle;
    GPIO_TypeDef* cs_port;
    uint16_t cs_pin;
    uint32_t timeout_ms;
} SPIDevice;

// SPI设备创建函数
BaseDevice* spi_device_create(SPIConfig* config);

#endif // SPI_DEVICE_H

// spi_device.c
#include "spi_device.h"
#include <stdlib.h>
#include <string.h>

// SPI设备的方法实现
staticintspi_device_init(BaseDevice* self, void* config) {
    SPIDevice* spi_dev = (SPIDevice*)self;
    SPIConfig* cfg = (SPIConfig*)config;

    if (!spi_dev || !cfg) {
        return-1;
    }

    // 初始化SPI设备特定数据
    spi_dev->spi_handle = cfg->spi_handle;
    spi_dev->cs_port = cfg->cs_port;
    spi_dev->cs_pin = cfg->cs_pin;
    spi_dev->timeout_ms = cfg->timeout_ms;

    // 初始化CS引脚
    HAL_GPIO_WritePin(spi_dev->cs_port, spi_dev->cs_pin, GPIO_PIN_SET);

    self->state = DEVICE_STATE_READY;
    return0;
}

staticintspi_device_deinit(BaseDevice* self) {
    SPIDevice* spi_dev = (SPIDevice*)self;

    if (!spi_dev) {
        return-1;
    }
```

```
    // 释放CS引脚
    HAL_GPIO_WritePin(spi_dev->cs_port, spi_dev->cs_pin, GPIO_PIN_SET);

    self->state = DEVICE_STATE_UNINIT;
    return0;
}

staticintspi_device_read(BaseDevice* self, uint8_t* buffer, size_t length) {
    SPIDevice* spi_dev = (SPIDevice*)self;
    HAL_StatusTypeDef status;

    if (!spi_dev || !buffer || length == 0) {
        return-1;
    }

    if (self->state != DEVICE_STATE_READY) {
        return-2;
    }

    self->state = DEVICE_STATE_BUSY;

    // 片选拉低
    HAL_GPIO_WritePin(spi_dev->cs_port, spi_dev->cs_pin, GPIO_PIN_RESET);

    // SPI读取
    status = HAL_SPI_Receive(spi_dev->spi_handle, buffer, length,
                             spi_dev->timeout_ms);

    // 片选拉高
    HAL_GPIO_WritePin(spi_dev->cs_port, spi_dev->cs_pin, GPIO_PIN_SET);

    self->state = DEVICE_STATE_READY;

    if (status != HAL_OK) {
        self->state = DEVICE_STATE_ERROR;
        self->error_code = status;
        return-3;
    }

    return length;
}

staticintspi_device_write(BaseDevice* self, const uint8_t* data, size_t length) {
    SPIDevice* spi_dev = (SPIDevice*)self;
    HAL_StatusTypeDef status;

    if (!spi_dev || !data || length == 0) {
        return-1;
```

```
    }

    if (self->state != DEVICE_STATE_READY) {
        return-2;
    }

    self->state = DEVICE_STATE_BUSY;

    // 片选拉低
    HAL_GPIO_WritePin(spi_dev->cs_port, spi_dev->cs_pin, GPIO_PIN_RESET);

    // SPI写入
    status = HAL_SPI_Transmit(spi_dev->spi_handle, (uint8_t*)data, length,
                              spi_dev->timeout_ms);

    // 片选拉高
    HAL_GPIO_WritePin(spi_dev->cs_port, spi_dev->cs_pin, GPIO_PIN_SET);

    self->state = DEVICE_STATE_READY;

    if (status != HAL_OK) {
        self->state = DEVICE_STATE_ERROR;
        self->error_code = status;
        return-3;
    }

    return length;
}

staticintspi_device_ioctl(BaseDevice* self, uint32_t cmd, void* arg) {
    SPIDevice* spi_dev = (SPIDevice*)self;

    switch (cmd) {
        case SPI_IOCTL_SET_TIMEOUT:
            if (arg) {
                spi_dev->timeout_ms = *(uint32_t*)arg;
                return0;
            }
            break;
        case SPI_IOCTL_GET_TIMEOUT:
            if (arg) {
                *(uint32_t*)arg = spi_dev->timeout_ms;
                return0;
            }
            break;
        default:
            return-1;
    }
    return-1;
```

```c
}

staticvoidspi_device_destroy(BaseDevice* self) {
    if (self) {
        spi_device_deinit(self);
        free(self);
    }
}

// SPI设备创建函数（构造函数）
BaseDevice* spi_device_create(SPIConfig* config) {
    SPIDevice* spi_dev = (SPIDevice*)malloc(sizeof(SPIDevice));

    if (!spi_dev) {
        returnNULL;
    }

    // 初始化基类成员
    memset(spi_dev, 0, sizeof(SPIDevice));
    spi_dev->base.type = DEVICE_TYPE_SPI;
    spi_dev->base.state = DEVICE_STATE_UNINIT;
    spi_dev->base.id = 0;
    spi_dev->base.error_code = 0;

    // 绑定虚函数（方法重写）
    spi_dev->base.init = spi_device_init;
    spi_dev->base.deinit = spi_device_deinit;
    spi_dev->base.read = spi_device_read;
    spi_dev->base.write = spi_device_write;
    spi_dev->base.ioctl = spi_device_ioctl;
    spi_dev->base.destroy = spi_device_destroy;

    // 初始化SPI特定数据
    if (config) {
        spi_dev->spi_handle = config->spi_handle;
        spi_dev->cs_port = config->cs_port;
        spi_dev->cs_pin = config->cs_pin;
        spi_dev->timeout_ms = config->timeout_ms;
    }

    return (BaseDevice*)spi_dev;
}
```

I2C设备实现

```c
// i2c_device.h
#ifndef I2C_DEVICE_H
#define I2C_DEVICE_H
```

```c
#include "device_driver.h"
#include "stm32f4xx_hal.h"

// I2C设备配置结构体
typedefstruct {
    I2C_HandleTypeDef* i2c_handle;
    uint8_t device_addr;
    uint32_t timeout_ms;
} I2CConfig;

// I2C设备结构体
typedefstruct {
    BaseDevice base;
    I2C_HandleTypeDef* i2c_handle;
    uint8_t device_addr;
    uint32_t timeout_ms;
} I2CDevice;

BaseDevice* i2c_device_create(I2CConfig* config);

#endif // I2C_DEVICE_H

// i2c_device.c
#include "i2c_device.h"
#include <stdlib.h>
#include <string.h>

staticinti2c_device_init(BaseDevice* self, void* config) {
    I2CDevice* i2c_dev = (I2CDevice*)self;
    I2CConfig* cfg = (I2CConfig*)config;

    if (!i2c_dev || !cfg) {
        return-1;
    }

    i2c_dev->i2c_handle = cfg->i2c_handle;
    i2c_dev->device_addr = cfg->device_addr;
    i2c_dev->timeout_ms = cfg->timeout_ms;

    self->state = DEVICE_STATE_READY;
    return0;
}

staticinti2c_device_deinit(BaseDevice* self) {
    if (!self) {
        return-1;
    }
```

```
    self->state = DEVICE_STATE_UNINIT;
    return0;
}

staticinti2c_device_read(BaseDevice* self, uint8_t* buffer, size_t length) {
    I2CDevice* i2c_dev = (I2CDevice*)self;
    HAL_StatusTypeDef status;

    if (!i2c_dev || !buffer || length == 0) {
        return-1;
    }

    if (self->state != DEVICE_STATE_READY) {
        return-2;
    }

    self->state = DEVICE_STATE_BUSY;

    // I2C读取
    status = HAL_I2C_Master_Receive(i2c_dev->i2c_handle,
                                    i2c_dev->device_addr << 1,
                                    buffer, length,
                                    i2c_dev->timeout_ms);

    self->state = DEVICE_STATE_READY;

    if (status != HAL_OK) {
        self->state = DEVICE_STATE_ERROR;
        self->error_code = status;
        return-3;
    }

    return length;
}

staticinti2c_device_write(BaseDevice* self, const uint8_t* data, size_t length) {
    I2CDevice* i2c_dev = (I2CDevice*)self;
    HAL_StatusTypeDef status;

    if (!i2c_dev || !data || length == 0) {
        return-1;
    }

    if (self->state != DEVICE_STATE_READY) {
        return-2;
    }

    self->state = DEVICE_STATE_BUSY;
```

```
    // I2C写入
    status = HAL_I2C_Master_Transmit(i2c_dev->i2c_handle,
                                     i2c_dev->device_addr << 1,
                                     (uint8_t*)data, length,
                                     i2c_dev->timeout_ms);

    self->state = DEVICE_STATE_READY;

    if (status != HAL_OK) {
        self->state = DEVICE_STATE_ERROR;
        self->error_code = status;
        return-3;
    }

    return length;
}

staticinti2c_device_ioctl(BaseDevice* self, uint32_t cmd, void* arg) {
    I2CDevice* i2c_dev = (I2CDevice*)self;

    switch (cmd) {
        case I2C_IOCTL_SET_ADDR:
            if (arg) {
                i2c_dev->device_addr = *(uint8_t*)arg;
                return0;
            }
            break;
        case I2C_IOCTL_GET_ADDR:
            if (arg) {
                *(uint8_t*)arg = i2c_dev->device_addr;
                return0;
            }
            break;
        default:
            return-1;
    }
    return-1;
}

staticvoidi2c_device_destroy(BaseDevice* self) {
    if (self) {
        i2c_device_deinit(self);
        free(self);
    }
}

BaseDevice* i2c_device_create(I2CConfig* config) {
    I2CDevice* i2c_dev = (I2CDevice*)malloc(sizeof(I2CDevice));
```

```c
    if (!i2c_dev) {
        returnNULL;
    }

    memset(i2c_dev, 0, sizeof(I2CDevice));
    i2c_dev->base.type = DEVICE_TYPE_I2C;
    i2c_dev->base.state = DEVICE_STATE_UNINIT;

    // 绑定虚函数
    i2c_dev->base.init = i2c_device_init;
    i2c_dev->base.deinit = i2c_device_deinit;
    i2c_dev->base.read = i2c_device_read;
    i2c_dev->base.write = i2c_device_write;
    i2c_dev->base.ioctl = i2c_device_ioctl;
    i2c_dev->base.destroy = i2c_device_destroy;

    if (config) {
        i2c_dev->i2c_handle = config->i2c_handle;
        i2c_dev->device_addr = config->device_addr;
        i2c_dev->timeout_ms = config->timeout_ms;
    }

    return (BaseDevice*)i2c_dev;
}
```

统一接口实现

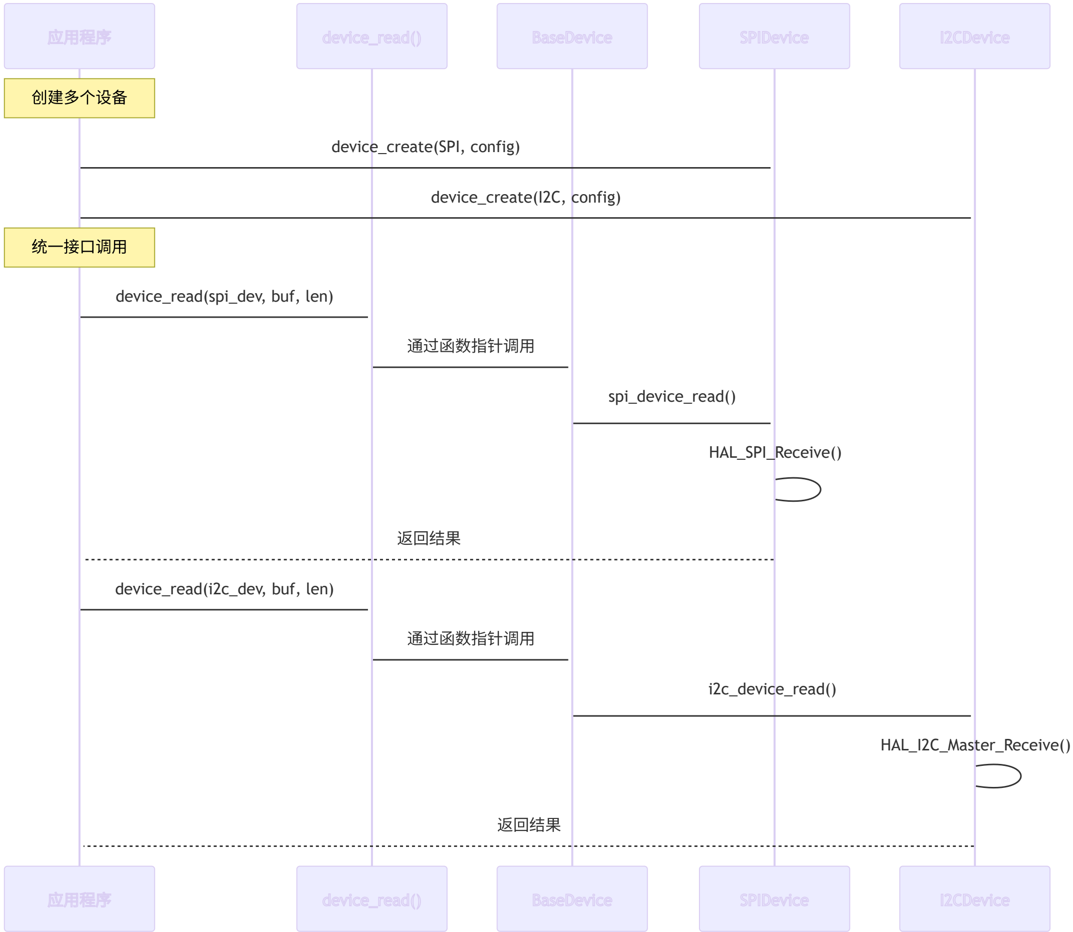
```c
// device_driver.c
#include "device_driver.h"
#include "spi_device.h"
#include "i2c_device.h"
#include <stdlib.h>

// 统一的设备创建接口（工厂模式）
BaseDevice* device_create(DeviceType type, void* config) {
    BaseDevice* device = NULL;

    switch (type) {
        case DEVICE_TYPE_SPI:
            device = spi_device_create((SPIConfig*)config);
            break;
        case DEVICE_TYPE_I2C:
            device = i2c_device_create((I2CConfig*)config);
            break;
        case DEVICE_TYPE_UART:
            // UART设备实现类似
            break;
        default:
```

```
            returnNULL;
        }

    return device;
}

// 统一的设备销毁接口
voiddevice_destroy(BaseDevice* device) {
    if (device && device->destroy) {
        device->destroy(device);
    }
}

// 统一的初始化接口（多态调用）
intdevice_init(BaseDevice* device, void* config) {
    if (!device || !device->init) {
        return-1;
    }
    return device->init(device, config);
}

// 统一的读取接口（多态调用）
intdevice_read(BaseDevice* device, uint8_t* buffer, size_t length) {
    if (!device || !device->read) {
        return-1;
    }
    return device->read(device, buffer, length);
}

// 统一的写入接口（多态调用）
intdevice_write(BaseDevice* device, const uint8_t* data, size_t length) {
    if (!device || !device->write) {
        return-1;
    }
    return device->write(device, data, length);
}

// 统一的控制接口（多态调用）
intdevice_ioctl(BaseDevice* device, uint32_t cmd, void* arg) {
    if (!device || !device->ioctl) {
        return-1;
    }
    return device->ioctl(device, cmd, arg);
}

// 获取设备状态
DeviceState device_get_state(BaseDevice* device) {
    if (!device) {
        return DEVICE_STATE_UNINIT;
```

```c
    }
    return device->state;
}

// 获取设备类型字符串
constchar* device_get_type_string(BaseDevice* device) {
    if (!device) {
        return"UNKNOWN";
    }

    switch (device->type) {
        case DEVICE_TYPE_SPI:
            return"SPI";
        case DEVICE_TYPE_I2C:
            return"I2C";
        case DEVICE_TYPE_UART:
            return"UART";
        default:
            return"UNKNOWN";
    }
}
```

应用示例

多态调用流程



扩展新设备类型

添加新的设备类型非常简单，只需要：

1. 定义新的设备结构体（继承BaseDevice）
2. 实现所有虚函数
3. 在工厂函数中添加创建逻辑

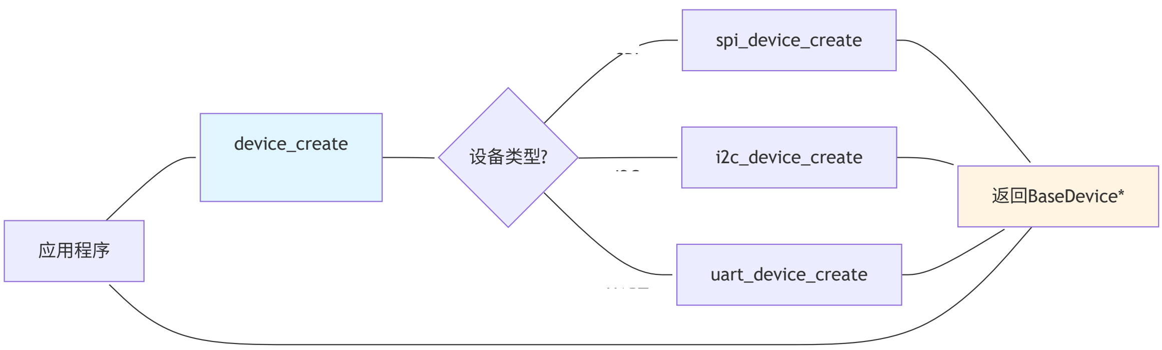
```
// uart_device.h
typedefstruct {
    BaseDevice base;
    UART_HandleTypeDef* uart_handle;
    uint32_t timeout_ms;
} UARTDevice;

// uart_device.c
// 实现所有虚函数...
staticintuart_device_read(BaseDevice* self, uint8_t* buffer, size_t length) {
    UARTDevice* uart_dev = (UARTDevice*)self;
    HAL_StatusTypeDef status;

    status = HAL_UART_Receive(uart_dev->uart_handle, buffer, length,
                              uart_dev->timeout_ms);
    return (status == HAL_OK) ? length : -1;
}
```

```c
// 在device_create中添加:
case DEVICE_TYPE_UART:
    device = uart_device_create((UARTConfig*)config);
    break;
```

设计模式应用

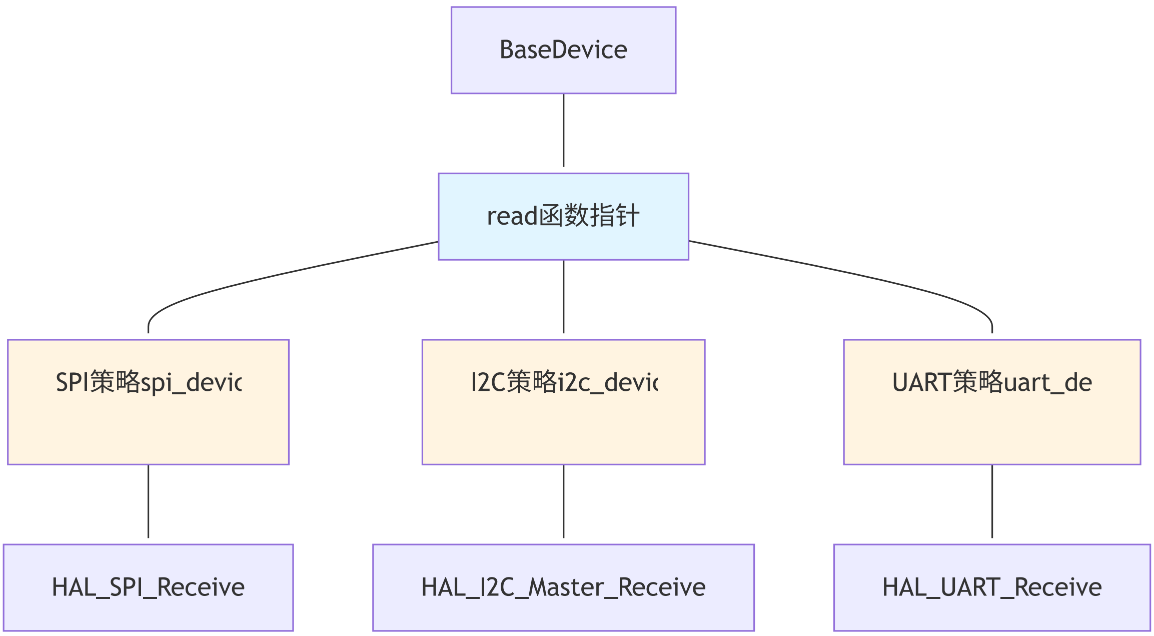
工厂模式（Factory Pattern）

`device_create()` 函数实现了工厂模式，根据设备类型创建对应的设备实例，隐藏了具体创建细节。



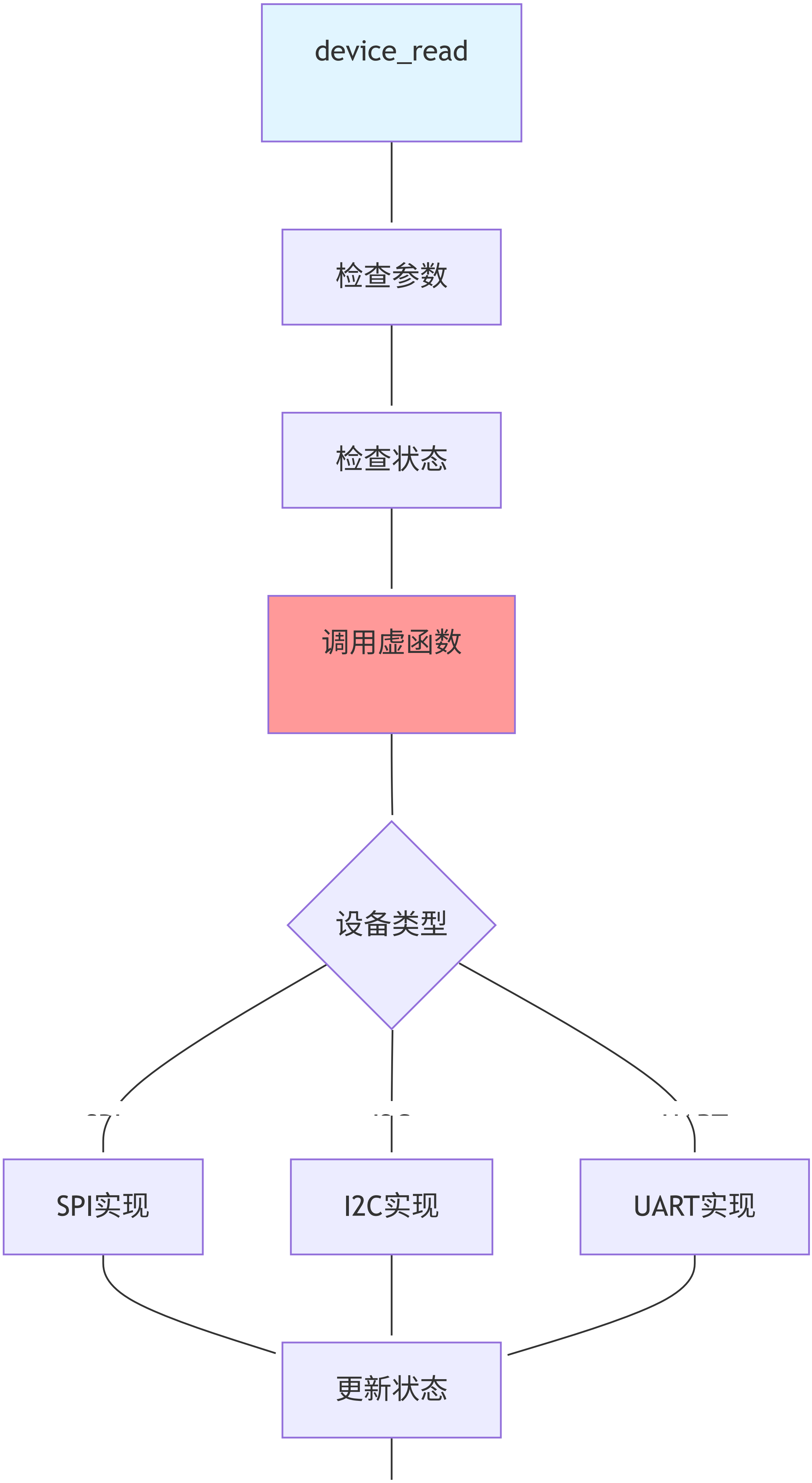
策略模式（Strategy Pattern）

不同的设备类型实现了不同的通信策略（SPI、I2C、UART），通过函数指针在运行时选择策略。



模板方法模式（Template Method Pattern）

基类定义了统一的接口框架，派生类实现具体的操作细节。


device_read
检查参数
检查状态
调用虚函数
设备类型
SPI实现
I2C实现
UART实现
更新状态

总结

1. **封装**：通过不透明指针和接口函数隐藏实现细节
2. **继承**：通过结构体嵌套实现单继承
3. **多态**：通过函数指针实现运行时多态

该设计方法在嵌入式系统中具有优势：

- ✅ **保持C语言的效率**：无虚函数表查找开销，内存占用可控
- ✅ **提高代码可维护性**：清晰的层次结构，易于扩展和修改
- ✅ **增强代码可读性**：统一的接口，自文档化的设计
- ✅ **支持多态编程**：同一接口处理不同设备类型

嵌入式 · 目录

上一篇
深入NVIC：提高MCU性能

下一篇
BLE功耗优化核心策略