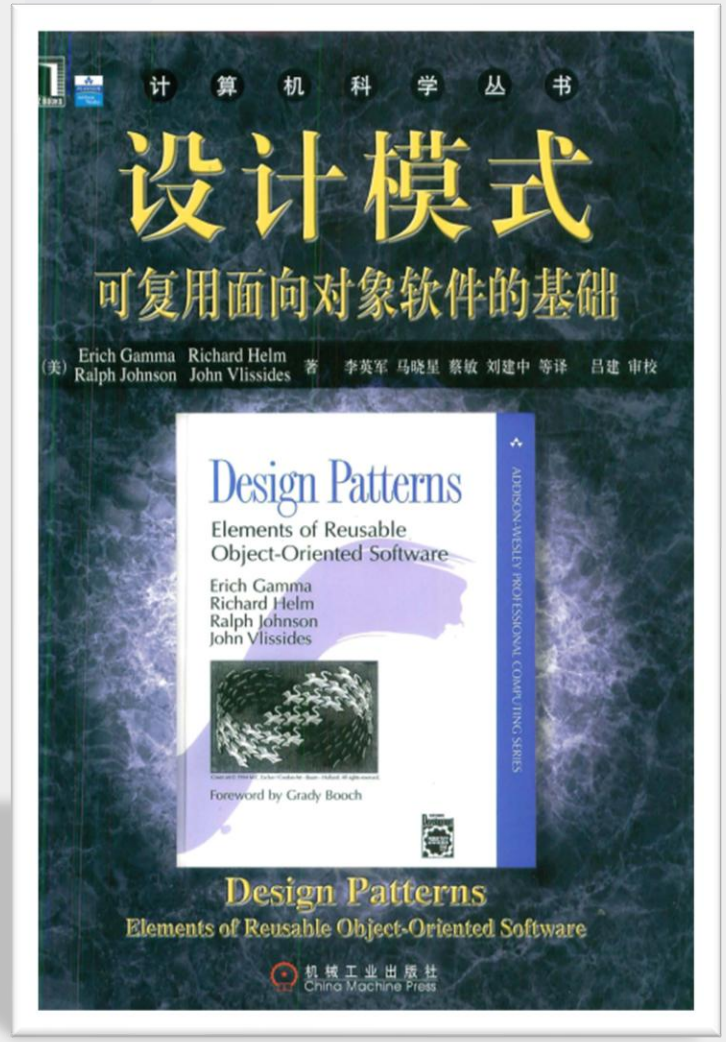

设计模式

面向嵌入式软件开发

ZHU ZHU
ENGINEER

针对C语言及嵌入式软件开发的伴读教程，陪伴学习设计模式这一进阶知识点。

第一章 引言

设计面向对象软件比较困难，而设计可复用的面向对象软件就更加困难。

C++ /Java ？

虽然设计模式描述的是面向对象设计，但它们都基于实际的解决方案，这些方案的实现语言是Smalltalk 和C++等主流面向对象编程语言，而不是过程式语言(Pascal、C、Ada)或更具动态特性的面向对象语言(CLOS、Dylan、Self)。我们从实用角度出发选择了Smalltalk 和C++，因为在这些语言的使用上，我们积累了许多经验，况且它们也变得越来越流行。

C！！！

1.1 什么是设计模式

设计模式是总结出来的、可以复用的“软件结构思路”

你有一个温度采集系统，可以使用不同的传感器（如 DS18B20、NTC、I2C温度计）；ADC 、DAC 、LED都可以

1.1 什么是设计模式

```c
typedef struct {
    float (*read_temp)(void);
} TempSensor;

float ds18b20_read(void);
float ntc_read(void);
float i2c_temp_read(void);

TempSensor ds18b20_sensor = { .read_temp = ds18b20_read };
TempSensor ntc_sensor = { .read_temp = ntc_read };
TempSensor i2c_sensor = { .read_temp = i2c_temp_read };
```

设计模式是经验的抽象，是让软件更容易改、更容易扩、更容易理解的结构套路。

1.1 什么是设计模式

策略模式

1. **模式名称**（pattern name） 一个助记名，它用一两个词来描述模式的问题、解决方案和效果。命名一个新的模式增加了我们的设计词汇。设计模式允许我们在较高的抽象层次上进行设计。基于一个模式词汇表，我们自己以及同事之间就可以讨论模式并在编写文档时使用它们。模式名可以帮助我们思考，便于我们与其他人交流设计思想及设计结果。找到恰当的模式名也是我们设计模式编目工作的难点之一。

2. **问题**(problem) 描述了应该在何时使用模式。它解释了设计问题和问题存在的前因后果，它可能描述了特定的设计问题，如怎样用对象表示算法等。也可能描述了导致不灵活设计的类或对象结构。有时候，问题部分会包括使用模式必须满足的一系列先决条件。

3. **解决方案**(solution) 描述了设计的组成成分，它们之间的相互关系及各自的职责和协作方式。因为模式就像一个模板，可应用于多种不同场合，所以解决方案并不描述一个特定而具体的设计或实现，而是提供设计问题的抽象描述和怎样用一个具有一般意义的元素组合（类或对象组合）来解决这个问题。

4. **效果**(consequences) 描述了模式应用的效果及使用模式应权衡的问题。尽管我们描述设计决策时，并不总提到模式效果，但它们对于评价设计选择和理解使用模式的代价及好处具有重要意义。软件效果大多关注对时间和空间的衡量，它们也表述了语言和实现问题。因为复用是面向对象设计的要素之一，所以模式效果包括它对系统的灵活性、扩充性或可移植性的影响，显式地列出这些效果对理解和评价这些模式很有帮助。

1.2 Smalltalk MVC中的设计模式

模型–视图–控制器模式

层次	职责	示例
Model（模型层）	保存系统状态、执行核心业务逻辑	控制LED、读取温度等
View（视图层）	负责数据输出、反馈显示	串口发送回显、状态报告
Controller（控制层）	负责解析命令、触发动作	解析"LED ON"、"TEMP?"等命令

1.2 Smalltalk MVC中的设计模式

📦 Model层（业务逻辑）

```c

typedef struct {
    bool led_state;
    float temp;
} SystemModel;

void model_set_led(SystemModel *m, bool on) {
    m->led_state = on;
    HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, on ? GPIO_PIN_SET : GPIO_PIN_RESET);
}

float model_read_temp(SystemModel *m) {
    m->temp = read_temperature_sensor();
    return m->temp;
}
```

1.2 Smalltalk MVC中的设计模式

View层（输出显示）

```c

void view_send_status(SystemModel *m) {
    char buf[64];
    sprintf(buf, "LED:%s TEMP:%.1f\r\n", m->led_state ? "ON" : "OFF", m->temp);
    uart_write(buf);
}

void view_send_ok(void) {
    uart_write("OK\r\n");
}

void view_send_error(void) {
    uart_write("ERR\r\n");
}
```

1.2 Smalltalk MVC中的设计模式

Controller层（输入解析）

```c

void controller_handle_command(SystemModel *m, const char *cmd) {
    if (strcmp(cmd, "LED ON") == 0) {
        model_set_led(m, true);
        view_send_ok();
    } else if (strcmp(cmd, "LED OFF") == 0) {
        model_set_led(m, false);
        view_send_ok();
    } else if (strcmp(cmd, "TEMP?") == 0) {
        model_read_temp(m);
        view_send_status(m);
    } else {
        view_send_error();
    }
}
```

1.2 Smalltalk MVC中的设计模式

主循环或RTOS任务

```c

SystemModel sys_model = {0};
char rx_buf[64];

while (1) {
    int len = uart_read(rx_buf);
    if (len > 0) {
        rx_buf[len] = '\0';
        controller_handle_command(&sys_model, rx_buf);
    }
}
```

1.3 描述设计模式

1. 模式名称与分类

- **名称（Name）：** 策略模式（Strategy Pattern）
- **分类（Category）：** 行为型模式（Behavioral Pattern）

2. 意图（Intent）

> 定义一系列算法或行为，将每个算法封装起来，使它们可以互相替换，而不影响使用这些算法的对象。

在嵌入式系统中，这种模式常用于：

- 不同硬件驱动的统一接口封装；
- 不同算法（控制/滤波）可在运行时切换；
- 不同通信协议策略的灵活切换。

1.3 描述设计模式

3. 动机（Motivation）

假设系统支持多种温度传感器：DS18B20、NTC热敏电阻、I²C外设温度计。

4. 适用性（Applicability）

在以下场景中使用策略模式非常合适：

场景	说明
多种算法仅实现不同	如不同滤波算法、通信协议、传感器驱动
希望在运行时切换算法	例如从NTC切换到I2C温度计
不希望控制逻辑中充满条件判断	去掉 `if/switch` 逻辑分支

1.3 描述设计模式

5. 结构（Structure）

C语言中可以用“结构体 + 函数指针”来实现“接口”的概念：

```c

typedef struct {
    float (*read_temp)(void);
} TempSensor;

float ds18b20_read(void);
float ntc_read(void);
float i2c_read(void);

TempSensor sensor_ds18b20 = { ds18b20_read };
TempSensor sensor_ntc     = { ntc_read };
TempSensor sensor_i2c     = { i2c_read };

// 当前使用的策略
TempSensor *sensor = &sensor_ds18b20;
```

1.3 描述设计模式

6. 参与者（Participants）

名称	职责
策略接口（Strategy）	定义统一操作接口，如 `read_temp()`
具体策略（ConcreteStrategy）	实现不同的算法或硬件驱动
上下文（Context）	持有策略对象的指针，并通过接口调用实现功能

7. 协作（Collaboration）

上下文对象（如主控制任务）通过调用策略接口执行温度读取，
而不需要关心实际是哪种传感器。

```c

float t = sensor->read_temp();  // 主程序逻辑不变
```

1.3 描述设计模式

8. 效果（Consequences）

✅ 优点：

- 消除了硬编码的条件判断；
- 新增算法不影响主逻辑；
- 可测试性、扩展性提高；
- 模块化更好。

⚠️ 缺点：

- 增加了一层间接调用；
- 代码结构稍复杂，适合中大型项目。

---

9. 实现（Implementation）

在嵌入式中：

- 通常通过函数指针或回调机制实现；
- 策略实例可通过配置文件、命令或参数选择；
- 若系统支持RTOS，可结合任务消息实现动态切换。

10. 示例代码（Sample Code）

c

```c
#include <stdio.h>

typedef struct {
    float (*read_temp)(void);
} TempSensor;

float ds18b20_read(void) { return 25.5; }
float ntc_read(void)     { return 26.1; }
float i2c_read(void)     { return 24.8; }

TempSensor ds18b20 = { ds18b20_read };
TempSensor ntc     = { ntc_read };
TempSensor i2c     = { i2c_read };

int main(void)
{
    TempSensor *sensor = &ds18b20;
    printf("Temperature = %.2f°C\n", sensor->read_temp());
    sensor = &ntc;
    printf("Temperature = %.2f°C\n", sensor->read_temp());
}
```

1.3 描述设计模式

11. 已知应用（Known Uses）

- STM32 HAL驱动中，UART/ADC/SPI接口均采用类似的回调机制；
- FreeRTOS中 `task_function_t` 函数指针本质上也是策略接口；
- Linux驱动中 `file_operations` 结构体就是一组策略表。