



设计模式

面向嵌入式软件开发



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。

1.6.5 运用复用机制（嵌入式 C 精简伴读）

本节核心观点只有两个：

1. 软件复用≠复制代码，而是通过结构让代码可扩展、可替换、减少修改。
2. 复用方式中，“组合 + 委托”是最推荐的方案。

1. 什么是复用机制？

复用机制是指：

让新功能复用已有代码，不要强依赖内部细节，也不要到处修改老代码。

嵌入式 C 中常见复用手段：

- 组合（推荐）
- 伪继承（不推荐）
- 复制/修改（临时用）

2. 为什么推荐“组合”？

组合 = 结构体里放子模块 + 通过统一接口调用它们

在 C 中等价于：

- 模块内封装数据和行为
- 上层只通过函数指针表 / API 调用
- 新模块只换实现，不改旧代码

简单例子：统一通信接口

```
typedef struct {
    int (*send)(uint8_t*, int);
    int (*recv)(uint8_t*, int);
} io_ops_t;

typedef struct {
    io_ops_t* ops;    // 组合：放进去即可
} comm_t;
```

新增 UART / SPI / USB → 只需要新的 `ops`
上层协议无需修改。

3. 委托：组合的核心技巧

委托（Delegation）就是：

一个对象把工作交给另一个对象做。

在 C 中是：

- 主模块不直接实现行为
- 把行为“委托”给函数指针 / 回调

例子：

```
void protocol_send(comm_t* c, uint8_t* buf, int len) {
    c->ops->send(buf, len);    // 委托底层实现
}
```

上层不关心底层是 UART、SPI 还是虚拟链路。

4. 为什么不推荐“继承/伪继承”？

结构体嵌套模拟继承：

```
typedef struct { int id; } base_t;
typedef struct { base_t base; int extra; } child_t;
```

缺点：

- 耦合高
- 上层容易依赖内部结构
- 扩展困难

适合少量情况，不是主流复用手段。

1.6.6 关联运行时和编译时的结构（嵌入式 C 伴读）

本节原书讲的是：

类图（静态结构）和对象图（运行时结构）如何对应。

但在嵌入式 C 中：

没有类、没有对象生命周期管理、没有自动构造/析构机制。

因此这节的重点可以压缩为一句话：

C 里所有运行时关系都归结为“内嵌”或“指针引用”。

1. 编译时结构：就是你的 struct 定义

C 的“类图” = 结构体的定义方式。

示例：

```
typedef struct {  
    motor_t x;  
    motor_t y;  
} axis_t;
```

2. 运行时结构：你如何实例化 + 如何建立引用

对象图等价于：

- 哪些对象被创建
- 它们之间用什么指针链接

示例（运行时建立关系）：

```
motor_t motor_x, motor_y;  
  
axis_t axis = {  
    .x = motor_x,  
    .y = motor_y,  
};
```

运行时对象关系非常“物理”：

- 内嵌的就是内嵌
- 指针指向谁就是谁
- 没有隐藏行为

3. 聚合(Aggregation) vs 相识(Association) —— C 中基本无差别

在本节原书中，两个概念的区别是：

关系	意义
相识	A 只需要知道 B
聚合	A 拥有一个“弱成员” B（不负责生命周期）

但在嵌入式 C 中：

两者都是“指针引用”，语义上无法区分。

例子（相识 or 聚合？你看不出来）：

```
typedef struct {
    driver_t* drv;    // 指向外部，不负责分配
} protocol_t;
```

原书的“聚合”和“相识”概念，在 C 中都只是：

- 我知道你，但
- 你的生命周期不归我管
- 我只是一个指针

4. 嵌入式 C 中真正需要区分的只有两类关系

① 组合（Composition）= 内嵌结构体

```
typedef struct {
    sensor_t s;        // 我拥有你
} board_t;
```

特点：

- 生命周期绑定
- 内存布局固定（编译期确定）
- 强关系

② 引用（Reference）= 指针指向外部结构体

```
typedef struct {
    sensor_t* s;        // 我指向你，但不负责你
} board_t;
```

特点：

- 关系由运行时建立
- 生命周期不确定
- 弱关系（无法区分聚合 or 相识）

1.6.7 设计应支持变化（嵌入式 C 简明版）

这一小节的核心观点很简单：

设计要能承受未来变化，不要把变化写死，把稳定的部分抽出来、不稳定的部分隔离起来。

在 C 中，它的落地方式通常是：

- 用函数指针接口隔离变化
- 用组合 + 委托让行为可替换
- 把常变和不常变的东西拆开

1. 不要让变化渗入整个系统

嵌入式环境常见的“变化点”：

- 不同硬件驱动
- 不同通信链路（UART/SPI/USB）
- 不同协议版本
- 不同策略（校准算法、滤波算法）
- 不同业务流程（状态机）

错误做法是：

```
// 在上层写死底层细节
if (use_uart) {
    uart_send(...);
} else {
    spi_send(...);
}
```

这种写法导致：

底层一变，上层全改。

2. 正确做法：把变化部分抽象成接口（函数指针表）

例子：通信接口抽象

```
typedef struct {
    int (*send)(uint8_t*, int);
    int (*recv)(uint8_t*, int);
} io_ops_t;
```

上层不管是 UART、SPI、USB，只使用 `ops->send()`。

上层只依赖 **稳定的接口**，底层是变化的，实现可替换。

3. 使用组合 + 委托，把变化封装进去

```
typedef struct {
    io_ops_t* ops;        // 组合
} protocol_t;

int proto_send(protocol_t* p, uint8_t* buf, int len) {
    return p->ops->send(buf, len); // 委托给底层
}
```

未来要支持：

- SPI
- 两路 UART
- 无线链路

只要换 `ops`，上层完全不动。

这就是本节要表达的“支持变化”。

4. 关于“稳定 vs 不稳定部分”的拆分

嵌入式项目里的 **稳定部分**：

- 数据结构
- 公共接口
- 驱动抽象层
- 状态机框架
- 中断管理框架
- 日志、队列、协议框架

不稳定部分：

- 某个具体芯片驱动
- 某个协议细节
- 校准/滤波算法
- 项目特定业务流程

本节重点是：

把不稳定部分放在框架外侧，通过接口隔离开。

稳定部分形成平台 / 底座。