1.6 设计模式怎样解决设计问题

1.6.1 寻找合适的对象

1 设计模式的“寻找合适的对象”核心思想

原书中这一节主要意思是：在设计系统时，你要清楚 **哪个对象应该承担某个职责**，也就是 **责任归属**。

- 不是把所有功能都堆到一个对象上，也不是随意分配职责。
- 要考虑对象的 **自然性（自然地承担某些行为）** 和 **封装性**。

2 嵌入式 C 的特点

在嵌入式系统里，我们常见的特点：

- **资源有限**：内存小、CPU慢。
- **模块化强**：代码通常按功能模块划分，硬件驱动、应用逻辑、通信协议分开。
- **硬件绑定紧密**：对象通常与外设或数据结构一一对应。
- **面向结构而非对象**：C 没有类和继承，更多用 `struct` + 函数指针来模拟对象行为。

所以，我们在嵌入式 C 里解读“寻找合适的对象”时，更多是 **寻找合适的结构体/模块承担职责**。

3️⃣ 如何在嵌入式 C 中寻找合适对象

可以按以下思路：

(1) 按职责划分结构体

- 每个模块对应一个“对象”，封装它的状态和行为。

(2) 模块自然性原则

- 谁最自然处理这件事，就把职责给谁。
- 例如：
  - UART 数据收发 → UART 驱动模块。
  - LED 灯闪烁 → LED 模块。
- 避免把 LED 闪烁逻辑放到 UI 模块里，这就是“职责不自然”。

(3) 接口与回调模拟多态

- C 没有类，但函数指针可以让模块暴露行为。

(4) 考虑硬件约束

- 一个对象不应该跨越多个硬件模块职责。
- 例如：
  - I2C 总线管理 → I2C 控制器对象
  - 每个传感器 → 传感器对象
- 避免一个对象直接操作 SPI、UART、GPIO 混乱。

假设我们做一个 小型风扇控制系统：

- 按键控制风扇开关和转速。
- 电机驱动负责风扇运转。
- LED 指示灯显示风扇状态（开/关/转速等级）。
- 我们要在 C 里设计模块，让每个模块承担最自然的职责。

(1) 按键对象

职责：负责读取按键状态、去抖动、发送事件。

(2) 电机对象

职责：控制风扇开关与转速

(3) LED 指示对象

职责：显示风扇状态

主程序：负责模块间的协调，而不是直接去操作 GPIO 或寄存器。

这样每个对象都有 **自然职责**，符合设计模式里"寻找合适对象"的思想。

总结　道法自然，不要强求。

1. **模块化结构体设计**：每个 `struct` 对应自然承担职责的功能块。
2. **封装行为与状态**：用函数指针封装方法，实现"对象化"。
3. **职责自然分配**：把功能交给最"合理"的模块，避免耦合过高。
4. **遵守硬件边界**：对象不跨越多个硬件域，便于维护和扩展。

1.6.2 决定对象的粒度

嵌入式里面怎么去区分项目内的对象颗粒度，这个事情比较困难。

2️⃣ 具体模式解析

1. **Facade（外观模式，4.5）**
   - 描述如何用 **一个对象表示完整子系统**。
   - 嵌入式理解：比如把风扇系统的"按键+电机+LED"封装成一个 `FanController` 外观对象，让主程序只调用一个接口即可操作整个系统。
   - 特点：粒度偏大，职责集中，简化调用。
2. **Flyweight（享元模式，4.6）**
   - 描述如何支持 **大量最小粒度的对象**，共享公共状态。
   - 嵌入式理解：比如大量相同 LED 或传感器对象，内部共享寄存器映射表或配置数据，避免重复开销。
   - 特点：粒度偏小，但通过共享管理大量对象。
3. **其他将对象分解成小对象的模式**
   - 例如组合模式（Composite）、策略模式（Strategy）等
   - 让一个大对象被拆分成若干小对象，每个小对象专注某个职责。
   - 嵌入式理解：一个复杂外设（如 UART 模块）可以拆成 TX、RX、Buffer 等小对象，各自处理对应行为。
4. **Abstract Factory（抽象工厂，3.1） & Builder（建造者，3.2）**
   - 产生 **专门负责生成其他对象的对象**。

- 嵌入式理解：比如有一个 `PeripheralFactory`，根据不同硬件生成 UART、SPI、I2C 对象，方便扩展和复用。

5. **Visitor（访问者，5.10） & Command（命令，5.2）**
   - 生成的对象专门 **负责对其他对象或对象组发起请求**。
   - 嵌入式理解：比如有一个 `Command` 对象集合，用于批量操作风扇、LED、传感器等，每个对象只关注自己的行为，实现解耦。

3️⃣ 嵌入式 C 的启示

- **粒度控制**：设计模式告诉我们有大对象（Facade）、小对象（Flyweight）、生成对象（Factory/Builder）等策略。
- **职责清晰**：每个对象都有“自然职责”，不要乱堆功能。
- **管理大量对象**：Flyweight、Command、Visitor 等模式提供了管理小对象或批量操作的方法。
- **适配硬件约束**：通过模式可以在嵌入式资源有限的情况下，实现模块化、可复用、易维护的对象设计。

同样的项目，不同的团队或者个人，使用的对象颗粒度就是不一样。

场景：

- 功能：按钮按下控制 LED
- 新增需求：长按不亮，短按亮，双击闪烁，长按呼吸灯

不同团队设计的对象颗粒度：

团队人数	对象设计	特点	新需求修改难度
1人	按钮检测电平 + 点灯逻辑在一个模块	粒度粗	每次新增功能都要改同一模块，改动大，容易出错
2人	按钮+LED 驱动模块，逻辑处理模块	粒度中	修改逻辑处理模块即可，驱动模块无需改动，修改局部，风险小
3人	按键驱动、LED驱动、逻辑处理模块	粒度细	新需求只修改逻辑处理模块或增加新对象，最清晰，速度快，扩展性高，但对象数量多，需要团队协调

说个最简单 但是又很抽象的例子

一个按钮点一个led，按钮按下点亮led

1个人，检测电平->点灯

2个人，按钮和led 驱动；逻辑处理

3个人，按键驱动；LED驱动：逻辑处理

这个新增需求，要求长按灯不亮，短按才亮，1个人的团队就得更改比较多； 2.3人团队就一个修改按钮驱动 一个新增处理逻辑

又新增需求 双击按钮 灯闪烁，长按 灯呼吸。短按 长亮；请问那个团队修改最清晰 ，速度最快 代码维护更容易。

这个例子只是告诉大家 ，适合自己团队的才是最好的；虽然更细的颗粒度，扩展性 ，代码层级哪些都更好，但是也是和付出的人力相关的

1.6.3 指定对象接口

1 “指定对象接口”

- **意思**：给每个对象定义一组公开的操作或方法，这些就是对象的“接口”。
- **嵌入式 C 对应**：
  - 可以用函数 + 结构体封装（`struct + 函数指针`），或直接在头文件暴露 API
  - 例子：LED 对象提供 `led_on()`、`led_off()` 接口，外部代码调用这些接口，而不直接操作寄存器。

---

2 “暴露接口而隐藏实现”

- **意思**：对象内部的实现细节对外部不可见，调用者只关心接口功能。
- **嵌入式 C 对应**：
  - 头文件只声明接口函数
  - 驱动文件内部实现具体操作寄存器、定时器等
  - 优点：实现可以改动，调用者无需修改。

---

3 “接口的作用”

- **模块化**：每个对象内部逻辑可以随时改动，不影响外部调用。
- **解耦**：调用者只关心“做什么”，不关心“怎么做”。
- **可替换性**：同一个接口可以有多种实现（不同硬件或算法）。
- **嵌入式 C 对应**：
  - 通过 `struct + 函数指针` 或统一 API，可以替换不同硬件驱动，而不改调用代码。

---

4 与过程调用的对比

特点	直接过程调用	指定对象接口
可读性	简单	稍复杂但逻辑清晰
耦合	高	低

特点	直接过程调用	指定对象接口
扩展性	差	好
多硬件支持	麻烦	容易

- 在嵌入式 C 里，“指定对象接口”就是 **在有限资源下模拟面向对象设计**，让模块职责清晰、可替换、易维护。
- 当然，对于简单硬件、短生命周期项目，直接过程调用也完全可行。

1.6.4 描述对象的实现

1️⃣ 原文大意拆解

1. **抽象类（Abstract Class）**
   - 提供 **部分实现 + 接口声明**
   - 调用者可以依赖抽象类接口，而不必关心具体实现
2. **混入类（Mixin Class）**
   - 提供一些额外功能，可与其他类组合
   - 不独立存在，只是为了扩展已有类的行为
3. **类继承 vs 接口继承**
   - **类继承**：继承父类的实现和接口
     - 调用者可能依赖父类实现细节 → 耦合度高
   - **接口继承**：只继承方法签名（接口），不继承具体实现
     - 调用者只依赖接口 → 低耦合，可替换不同实现
4. **核心思想**：

> 对接口编程，而不是对实现编程

- 调用者只依赖对象提供的接口，而不依赖具体实现或父类内部逻辑
- 便于模块替换、扩展和维护

2️⃣ 嵌入式 C 的对应实现

C 语言没有类、继承和混入类，但可以模拟接口和抽象概念：

1. **抽象类对应 C**
   - 用 `struct + 函数指针` 定义接口和默认实现
   - 外部模块依赖接口函数，不关心内部实现

```
typedef struct Motor Motor;

struct Motor {
    void (*start)(Motor* self);
    void (*stop)(Motor* self);
    // 默认实现可以提供空函数或基础功能
};
```

1. **接口继承对应 C**
    - 通过 **统一接口 struct**，不同模块实现同样方法
    - 调用者只依赖接口，不管具体实现

```
typedef struct {
    void (*start)(void*);
    void (*stop)(void*);
} MotorOps;

typedef struct {
    int id;
    MotorOps* ops;
} Motor;
```

- 不同硬件可实现不同 `MotorOps`，调用者只用 `ops->start(&motor)`

1. **混入类对应 C**
    - 通过额外的函数或辅助 struct 实现额外功能
    - 可组合到多个对象中使用

```
typedef struct {
    void (*log_status)(void*);
} LoggerMixin;

// Motor 对象可以组合 LoggerMixin，实现状态记录
```

1. **接口编程 vs 实现编程**

- **接口编程**：调用 `ops->start(&motor)`，不依赖寄存器配置或算法
- **实现编程**：直接调用 `GPIOx->CR = 1` 等硬件寄存器操作 → 高耦合，修改成本大

---

3 总结

- **抽象类/接口** → 提供调用接口，隐藏实现
- **混入类** → 为对象增加可复用功能
- **接口继承**比类继承更灵活 → 调用者只依赖接口，耦合低
- **嵌入式 C 实现方式**：
    - `struct + 函数指针` 模拟对象接口

- 通过组合 struct 实现“混入”
- 调用者通过接口编程，而不是直接操作内部实现或寄存器

> 核心思想：**写代码依赖接口而不是实现** → 提高模块可替换性、扩展性和维护性。