经典设计模式浅析

《经典设计模式浅析》总纲——用白话把 GoF 23 种设计模式讲清楚：每个模式解决什么痛点、怎么实现、有哪些坑、开源项目里怎么用。主用 Go，辅以 C++/Java。

全 24 篇 · 作者 David

写在前面 · 模式是给痛点立的牌子

先说一句可能得罪人的大实话：大部分人是先会背 23 个模式的名字，再慢慢忘光，最后在某次重构里自己重新发明了其中几个——然后才真正懂了它们。

这个系列想把顺序倒过来：先让你看见痛，再告诉你前人是怎么止痛的。

设计模式到底是个啥

把话说白了：设计模式就是前人在某个坑里摔多了，爬出来之后在坑边立的一块牌子，上面写着“这么走能绕过去”。

它不是语法，不是框架，不是你 `import` 进来就能用的库。它是一种在特定场景下反复被验证有效的对象组织套路。1994 年，四位作者(业内昵称 GoF,Gang of Four,“四人帮”)把当时面向对象界反复出现的 23 种套路收集成册，写了本《Design Patterns》，从此这 23 个就成了程序员的“普通话”——你说“这里用个观察者”，对面立刻懂，不用画半小时图。

> 所以模式最大的价值有两层：一层是解决方案本身(怎么写)，另一层是词汇表(怎么沟通)。后者常被低估——团队里能用一个词对齐一整套设计，是极高效的事。

为什么会有这些痛点：一切都因为“变”

如果需求永远不变，世界上就不需要设计模式——一把梭把功能写完，能跑就行。

问题是需求一定会变。今天只支持支付宝，明天加微信，后天老板要加数字货币；今天导出 PDF，下周要导出 Excel。几乎所有设计模式，本质上都在回答同一个问题：当某个东西要变的时候，怎么让改动的影响范围最小？

GoF 有一句纲领性的话，值得刻在脑子里：

> *“Encapsulate the concept that varies.(把“会变的那部分”封装隔离起来。)”*

你会发现，23 个模式几乎都是这句话在不同场景下的变体：策略模式隔离“会变的算法”，观察者模式隔离“会变的通知对象”，工厂模式隔离“会变的具体类”……看模式时别只盯着类图，要问一句：它到底在隔离什么变化？ 抓住这条线，23 个模式就不再是 23 条孤立的口诀，而是一张网。

三大类，一句话分清

GoF 把 23 个模式按“管什么”分成三类，这个分法非常好记：

类别	管什么	一句话	典型代表
创建型	对象怎么生	把 `new` 这件事管起来，别到处乱 new	单例、工厂、建造者
结构型	对象怎么拼	把小对象组装成大结构，还不别扭	适配器、装饰器、代理
行为型	对象怎么处	对象之间怎么分工、怎么通信	策略、观察者、命令

创建型管“生”，结构型管“拼”，行为型管“处对象关系”——生、拼、处，三个字记一辈子。

每篇怎么讲

为了不写成第 101 本“设计模式速查手册”，本系列每个模式都按同一条线索展开，力求把“背名词”变成“懂痛点”：

1. 这是什么 —— 先用一个现实生活里的例子把直觉建立起来(遥控器、咖啡加料、公司审批流……)，再回到代码。
2. 解决什么痛 —— 不先讲解法，先把“不用这个模式会有多难受”演示给你看。痛点不清，解法无感。
3. 怎么实现 —— 给出 Go 的惯用实现(必要时对比 C++ / Java)，不堆代码，只讲关键结构。
4. 有哪些坑 —— 模式用错比不用还糟。这一节专讲过度设计、误用、以及“看起来像其实不是”的陷阱。
5. 开源里怎么用 —— 到真实的知名项目(Go 标准库、Kubernetes、Java JDK、Nginx……)里，把这个模式活生生指给你看。

> ⚠️ 一句提醒贯穿全系列：**模式是解药，不是维生素**。没病别乱吃。见到痛点才用对应的模式，是工程师；为了用模式而制造痛点，是“模式病”。本系列每一篇的“坑”小节，有一半篇幅在劝你别乱用。

关于代码

- 主语言 Go：用 Go 的惯用法(小接口、组合、函数值、`error` 返回)来写，你会发现不少“经典 OOP 模式”在 Go 里轻得几乎看不见——这本身就很说明问题。
- 辅以 C++ / Java：在能体现差异的地方，顺手对比一下 C++ 和 Java 的写法，看看同一个痛点在不同类型系统下怎么止。
- 完整可运行代码都在配套开源仓库，9 种语言对照实现，建议边读边跑：👉 github.com/davidapp/GoF

准备好了就从单例模式开始——它最简单，却也是争议最大、被骂最多的一个，适合用来热身。

01 单例模式 · Singleton（创建型）

我们从最简单的模式开始热身。简单到什么程度呢？一句话就能说完：让某个类型在整个程序里只有一个实例，并且提供一个统一的地方去拿它。

但别被“简单”骗了——单例是 23 个模式里争议最大、被骂得最多的一个。有人天天用，有人见了就皱眉说“这不就是全局变量化了个妆嘛”。热身归热身，这里面的门道不少。

一、这是什么：公司只有一个 CEO

现实里的单例随处可见：

- 一家公司，CEO 只有一个。全公司上下要找“最终拍板的人”，指向的都是同一个。你不会说“给我 new 一个新 CEO”。
- 一个国家，中央政府只有一个。
- 你电脑上，打印任务队列通常只有一个——不然两个队列各排各的，纸就打架了。

这些东西的共同点是：它天然就该只有一份，而且大家得能方便地找到这一份。 这就是单例。

二、解决什么痛：两个需求捆在一起

单例其实同时回答了两个问题，这两个问题必须一起满足，才轮得到单例出场：

1. “只能有一个” —— 如果 new 出好几个日志器、好几个配置管理器，各自维护一份状态，那就乱套了。想象配置管理器有三个实例，A 改了配置 B 看不见，这种 bug 能让人查到怀疑人生。
2. “到处都要用” —— 这个唯一实例得有个全局、方便的访问点，不能让调用方费劲地层层传递。

> ⚠️ 划重点：必须两个条件同时成立。 如果只是“想全局访问”，那是全局变量的活儿；如果只是“想控制数量”，可能对象池更合适。单例的独特价值在于把“唯一性”和“全局访问”打包保证。很多人滥用单例，恰恰是因为只想要第 2 点(图方便)，却顺手把第 1 点(强唯一)也焊死了——埋雷就是从这儿开始的。

三、怎么实现：Go 的 `sync.Once`

在 Go 里，单例的惯用写法干净得几乎不像个“模式”：一个包级变量存实例，一个 `sync.Once` 保证初始化只跑一次。

go/singleton/main.go（核心）

```go
var (
    instance *Logger
    once     sync.Once
)

// GetLogger 返回全局唯一的 Logger 实例。
// sync.Once 保证懒加载且并发安全——这是 Go 里实现单例的标准答案。
func GetLogger() *Logger {
    once.Do(func() {
        fmt.Println("(创建 Logger 实例...)")
        instance = &Logger{level: LevelInfo}
    })
    return instance
}
```

`once.Do(f)` 的语义是：无论多少个 goroutine 同时调用 `GetLogger()`，里头那个初始化函数 `f` 保证只执行一次，其余调用会阻塞到第一次执行完再返回。于是“创建 Logger 实例...”这行，你跑多少遍都只打印一次。

对比一下你可能在别的语言里见过的、著名的“双重检查锁定”(Double-Checked Locking)，就知道 Go 有多省心了：

Java —— 经典但容易写错的双重检查锁

```java
class Logger {
    private static volatile Logger instance;    // volatile 一个都不能少
    private Logger() {}
    static Logger getInstance() {
        if (instance == null) {                 // 第一次检查(不加锁)
            synchronized (Logger.class) {
                if (instance == null) {         // 第二次检查(加锁后)
                    instance = new Logger();
                }
            }
        }
        return instance;
    }
}
```

这段 Java 代码是无数面试题的重灾区：`volatile` 忘了会因指令重排拿到半初始化对象，两次 `null` 检查的意义各不相同.....Go 用 `sync.Once` 把这一整套心智负担全吃掉了。 这也是为什么在 Go 社区，单例几乎从不被当成一个“需要专门学”的模式——语言层面已经给好了。

> 更 Go 的观点是：很多所谓单例，其实一个包级变量 + `init()` 函数(饿汉式，程序启动即创建)就够了，连 `sync.Once` 都省了。只有当初始化开销大、希望“用到才创建”(懒汉式)时，`sync.Once` 才真正上场。

C++ 则在 C++11 之后靠“魔法静态变量”(Meyers' Singleton)达到同样的并发安全：

C++11 —— 函数内 static,标准保证线程安全初始化

```
Logger& getLogger() {
    static Logger instance;    // C++11 起,静态局部变量的初始化是线程安全的
    return instance;
}
```

三种语言，同一个痛点，止痛方式一个比一个轻。语言进化的一个方向，就是把过去要靠“模式”手写的东西，变成语言自带的保证。

四、有哪些坑：它凭什么被骂

单例是“模式病”的头号病灶，几个经典槽点必须拎清：

- 它是伪装的全局状态。 全局可变状态是万恶之源这句话你肯定听过。单例把全局变量包了层皮，但本质没变：任何地方都能改它，改坏了你很难定位是谁干的。
- 它让单元测试变地狱。 测试要的是“隔离”和“可替换”。而单例是硬编码的全局引用，你想在测试里塞个假的(mock)进去？难。它把依赖藏在了 `GetLogger()` 这种静态调用背后，而不是老老实实从参数传进来——这叫“隐藏依赖”。
- “唯一”的假设经常被现实打脸。 今天你笃定“数据库连接只要一个”，明天要读写分离、要分库，就得有俩。当初焊死的单例，现在得连根拔。

> ⚠ 实战建议：优先考虑“依赖注入 + 单一实例”，而不是“单例模式”。 也就是说——在程序启动处创建一个实例，然后把它当普通参数显式传给需要的人。这样“只有一个”由你的组装逻辑保证(你就 new 了一次)，而每个使用者拿到的是一个普通对象，能测、能换、依赖关系一目了然。你既得到了“唯一”，又躲开了“全局访问”带来的所有毛病。单例真正不可替代的场景，比大多数人以为的要少得多。

五、开源里怎么用

抛开争议，单例(以及它的温和替代——包级单一实例)在真实项目里确实无处不在：

- Go 标准库 `http.DefaultClient` / `http.DefaultServeMux`：包级的默认单一实例，`http.Get()` 底层用的就是 `DefaultClient`。它没上锁保护(是导出变量),Go 团队用“约定”而非“强制”来维持其单一性——很 Go 的取舍。
- Go `database/sql` 的 `*sql.DB`：官方明确建议一个数据库全程共享一个 `*sql.DB`(它内部是连接池)，典型的“进程内单例”用法——注意它单例的是“池”，不是“连接”。
- Java `java.lang.Runtime`：`Runtime.getRuntime()` 是 JDK 里教科书级的单例。
- Spring 框架：Bean 默认作用域就是 `singleton`——但它是“每个容器一个”，而且由容器统一管理生命周期和注入。这其实正是上面推荐的“依赖注入 + 单一实例”，Spring 把单例做对了。

从 `Runtime` 的硬单例，到 Spring 容器管理的软单例，你能清楚看到工程界这些年的态度转变：“唯一”这件事，越来越倾向于交给外层的组装/容器来保证，而不是让类自己焊死。

热身完毕。下一篇工厂方法，我们进入创建型的正题：当“到底 new 哪个类”本身都会变的时候，怎么办。

02 工厂方法 · Factory Method（创建型）

上一篇的单例管的是“只造一个”。这一篇更进一步：造哪个，我现在还不知道。

一、这是什么：你点外卖，不关心是谁在颠勺

你在 App 上点了份宫保鸡丁。你关心的是“我要一份宫保鸡丁”，至于是张师傅还是李师傅炒的、用的是燃气灶还是电磁炉——你不关心，也不应该关心。App 只管把“做一份菜”的请求发出去，具体哪个厨房、哪口锅，由后厨自己决定。

工厂方法就是干这个的：它定义一个“创建产品”的方法，但把“到底创建哪个具体产品”的决定权，交给子类(或具体实现)。调用方拿到的永远是一个抽象的“产品”，不碰具体类型。

配套代码用的是物流的例子：你要“运货”，至于用卡车还是轮船，交给具体的物流公司去定。

二、解决什么痛：满地都是 `new ConcreteClass()`

先看不用工厂方法有多难受。假设你的业务代码里直接这么写：

```
// 痛点代码:业务逻辑和"具体用哪个运输工具"死死焊在一起
func planDelivery(mode string) string {
    var t Transport
    if mode == "road" {
        t = &Truck{}        // ← 硬编码具体类型
    } else if mode == "sea" {
        t = &Ship{}         // ← 硬编码具体类型
    }
    // ...一堆基于 t 的通用逻辑...
    return t.Deliver()
}
```

这段代码的病根在于：它既负责“业务流程”，又负责“决定造哪个类”。于是——

- 哪天要加“空运”(`Plane`)，你得回来改这个 `if-else`。
- 类似的 `if mode == ...` 判断，往往在代码库里散落好几处，加一个类型要改一圈，漏一处就是 bug。
- 这直接违反了开闭原则(对扩展开放，对修改关闭)：加新类型本该“只增不改”，现在却要动老代码。

> 一句话抓住本质：工厂方法隔离的“会变的东西”，就是“具体产品类”。业务流程是稳定的(总是“规划路线→运输”)，会变的是“用哪种运输工具”。把后者拎出去单独封装，业务流程就再也不用因为多了个运输方式而改动。

三、怎么实现：抽象创建者 + 具体创建者

Go 的做法是把“创建”抽象成一个接口方法。业务逻辑只依赖这个接口，不认识任何具体产品：

go/factory-method/main.go（核心）

```
// 抽象创建者:只声明工厂方法,造哪个 Transport 由实现者决定
type Logistics interface {
    CreateTransport() Transport
}

// 通用业务逻辑:完全不依赖具体产品,只通过工厂方法拿抽象产品
func PlanDelivery(l Logistics) string {
    transport := l.CreateTransport()      // ← 造谁?我不知道,也不想知道
    return "规划运输路线 -> " + transport.Deliver()
}

// 具体创建者:各自决定造什么
type RoadLogistics struct{}
func (r *RoadLogistics) CreateTransport() Transport { return &Truck{} }

type SeaLogistics struct{}
func (s *SeaLogistics) CreateTransport() Transport { return &Ship{} }
```

现在加“空运”是什么体验？新增一个 `AirLogistics` + `Plane`，一行老代码都不用改。`PlanDelivery` 岿然不动。这就是开闭原则落地的样子。

Java 里这个模式的形态更“标准”(因为 Java 靠继承)，抽象创建者是个抽象类：

Java —— 工厂方法常以抽象类形式出现

```
abstract class Logistics {
    abstract Transport createTransport();       // 工厂方法,交给子类实现
    String planDelivery() {                      // 复用的业务骨架
        return "规划路线 -> " + createTransport().deliver();
    }
}
class RoadLogistics extends Logistics {
    Transport createTransport() { return new Truck(); }
}
```

注意这里的味道差异：Java 用“继承 + 覆写抽象方法”来下放决定权，而 Go 用“实现接口”。Go 没有继承，反而让这个模式显得更轻——工厂方法的精髓从来不是“继承”，而是“把创建行为抽象成一个可替换的点”。

四、有哪些坑

- 别和“简单工厂”搞混。 你可能见过一个 `NewTransport(mode string) Transport` 函数，内部一个 `switch` 造不同对象——那叫简单工厂(Simple Factory)，它不是 GoF 的工厂方法，甚至不算正式模式。区别在于：简单工厂把 `switch` 收拢到一个函数里(治标，加类型还是要改那个 switch)；工

厂方法则通过多态彻底消灭了 `switch`(治本，加类型只需新增类)。简单工厂胜在简单，小场景够用；别看见“工厂”俩字就以为是同一个东西。

- 别为了模式而模式。 如果你就一种产品，而且看不到未来会有第二种，直接 `new` 它、或者写个朴素的 `NewXxx()` 构造函数就好。凭空引入抽象创建者接口，是拿真实的复杂度去换想象中的灵活性，纯亏。
- Go 味提醒： 大量场景下，Go 根本不需要“抽象创建者”这一层。一个返回接口的普通函数 `func NewTransport(kind Kind) Transport` 就解决了绝大多数需求。只有当“创建逻辑本身要随上下文多态变化”(比如不同物流公司有各自成套的创建策略)时，完整的工厂方法才真正值回票价。

> ⚠️ 判断要不要上工厂方法，就问一句：“未来会不会冒出新的产品类型，而我不想每次都改调用方？”答“会”，用；答“不确定”或“不会”，先别用，等痛了再重构进来也不迟。

五、开源里怎么用

- Go `database/sql`：堪称工厂方法的活教材。驱动通过 `sql.Register("mysql", &MySQLDriver{})` 注册，你调 `sql.Open("mysql", dsn)` 时，标准库根据名字找到对应驱动，由驱动自己创建连接。你的业务代码只认 `*sql.DB` 和 `driver.Conn` 接口，换成 postgres 只改一个字符串——具体连接对象由谁造？驱动这个“具体创建者”造。
- Go `image` 包：`image.Decode` 依据注册的格式(png/jpeg/gif)把字节流解码成 `image.Image`，各格式包在 `init()` 里注册自己的解码器。同一个“注册 + 按需创建”的骨架。
- Java JDK: `Collection.iterator()` 是教科书级工厂方法——每种集合返回自己的 `Iterator` 实现；`Calendar.getInstance()` 依据地区/时区返回具体的 `Calendar` 子类。
- 日志门面(SLF4J / Go 的 `log/slog`)：你面向抽象的 `Logger` 接口编程，具体产出哪个后端的 logger 实例，由工厂决定。

这些例子有个共同气质：调用方永远只跟“抽象产品”打交道，“具体是谁”这件事被牢牢关在工厂里。 记住这个气质，你在任何代码库里都能一眼认出工厂方法。

工厂方法一次造“一个”产品。下一篇抽象工厂，我们要一次造“一整族”配套的产品——并保证它们不会串味。

03 抽象工厂 · Abstract Factory（创建型）

上一篇的工厂方法，一次造一个产品。这一篇要解决的是升级版难题：我要一次造一整套配套的产品，而且这套东西之间不能串味。

一、这是什么：宜家的整套家具风格

去宜家买家具，你选的往往不是单件，而是一个系列：北欧现代风的沙发、茶几、电视柜是一套，配一起协调；美式复古风又是另一套。你不会拿北欧沙发去配美式复古茶几——风格得统一。

“现代风工厂”能给你造出成套的现代家具，“复古风工厂”能给你造出成套的复古家具。你只要选定一个工厂，它交付的一整族产品自动是配套的。这就是抽象工厂。

配套代码是经典的跨平台 GUI：一个“Windows 工厂”造出成套的 Windows 按钮 + Windows 复选框；一个“macOS 工厂”造出成套的 macOS 控件。你选定平台工厂，拿到的整套控件风格自动统一。

二、解决什么痛：一族对象，必须“成套”

工厂方法解决“造哪个”的问题；抽象工厂在此之上，多压了一个更狠的约束：这些对象之间有“家族”关系，必须来自同一族，混搭就出事。

不用抽象工厂，痛在哪？你会写出这种代码：

```
// 痛点:平台判断散落各处,而且极易"串味"
button := &WindowsButton{}
checkbox := &MacCheckbox{}      // ← 手滑!Windows 按钮配了 macOS 复选框
```

编译器不会拦你——它俩都满足各自的接口。但运行起来，界面风格一半 Windows 一半 macOS，活见鬼。问题的根源是：“保证同族”这件事，没有任何机制来强制，全靠程序员自觉。 一旦平台判断逻辑散落在几十个地方，串味是迟早的事。

> 抽象工厂隔离的“会变的东西”，是“整个产品族”。会变的不是单个控件，而是“这一整套用哪个平台的实现”。把“造一族产品”的责任收拢到一个工厂对象里，同族一致性就由这个工厂从结构上保证了——你根本没机会串味，因为一个 `WindowsFactory` 压根造不出 macOS 复选框。

三、怎么实现：一个工厂，一族创建方法

关键区别就一点：工厂方法接口只有一个创建方法，抽象工厂接口有一组——每个方法造这一族里的一种产品。

go/abstract-factory/main.go（核心）

```go
// 抽象工厂:声明一组创建方法,生产成套的相关产品
type GUIFactory interface {
    CreateButton() Button        // 造这一族的按钮
    CreateCheckbox() Checkbox    // 造这一族的复选框
}

// 具体工厂:Windows 一族
type WindowsFactory struct{}
func (f *WindowsFactory) CreateButton() Button     { return &WindowsButton{} }
func (f *WindowsFactory) CreateCheckbox() Checkbox { return &WindowsCheckbox{} }

// 客户端:只依赖抽象工厂 + 抽象产品,拿到的整族自动配套
func renderUI(factory GUIFactory) {
    button := factory.CreateButton()
    checkbox := factory.CreateCheckbox()
    // button 和 checkbox 保证同族——因为它俩出自同一个 factory
    fmt.Println(button.Render(), checkbox.Render())
}
```

`renderUI(&WindowsFactory{})` 交付纯 Windows 套装，`renderUI(&MacFactory{})` 交付纯 macOS 套装。“同族一致”不再靠自觉，而是靠“它们出自同一个工厂对象”这个事实来保证。 想支持 Linux？加一个 `LinuxFactory` 实现这仨方法，客户端代码一行不改。

四、有哪些坑

- 和工厂方法怎么分？ 一句口诀：工厂方法是“一个方法造一个产品的变体”，抽象工厂是“一个对象造一族产品的变体”。 前者关注“这一个东西造哪种”，后者关注“这一整套用哪个系列”。实现上，抽象工厂的每个方法，单拎出来看都像一个工厂方法——抽象工厂常被看作“工厂方法的集合”。
- 著名的软肋：加“产品种类”是噩梦。 这是抽象工厂最该被记住的缺点。加一个产品族(比如新增 Linux 平台)很爽——只需新增一个工厂类。但加一个产品种类(比如所有平台都要新增一个 `Slider` 控件)是灾难——你得去改 `GUIFactory` 接口，然后每一个具体工厂都得跟着实现 `CreateSlider()`。产品维度越稳定，抽象工厂越合适；产品种类老是增删，它就是负担。
- 别为“可能有第二个平台”就上抽象工厂。 如果你现在只有一个平台，而且第二个八字没一撇，这套接口 + 多工厂的架子就是纯开销。抽象工厂的复杂度不低，值不值得，取决于“产品族真的会有多个”这个前提是否成立。

> ⚠ 抽象工厂是创建型里“最重”的模式之一。上它之前先确认两点：(1) 你真的有“成套配套”的一族产品；(2) 这个“族”真的会有多个变体需要整体切换。 两个都满足，它是利器；缺一个，它就是过度设计的典型。

五、开源里怎么用

- Java JDK: `javax.xml.parsers.DocumentBuilderFactory`、`SAXParserFactory`——它们生产一族 XML 解析相关对象，具体实现可通过配置整体替换；`java.awt.Toolkit` 也是按平台生产成

套 AWT 组件的抽象工厂。

- 数据库/驱动生态：一个“方言工厂”对应一个数据库，成套地生产该数据库的连接、方言 SQL 生成器、类型映射器——切换数据库 = 切换整个工厂族。
- 跨平台 UI 框架：Qt、GTK、以及各类游戏引擎的渲染后端(DirectX 一族 / OpenGL 一族 / Vulkan 一族)，都在用抽象工厂的思路，让“切换整套底层实现”变成“换一个工厂”。
- Go 里为什么少见“纯抽象工厂”？ 因为 Go 用接口组合和返回接口的函数就能自然表达“一族相关构造”，很少需要把它郑重其事地封成一个 `XxxFactory` 接口。这再次印证那句话：模式是语言能力的补丁，语言越强，某些模式就越“隐形”。

创建型到这里，我们已经会造“一个”(工厂方法)和“一族”(抽象工厂)。下一篇建造者，换个维度：当单个对象本身复杂到“零件又多又杂、还有一堆可选项”时，怎么优雅地把它拼出来。

04 建造者 · Builder（创建型）

前两篇解决“造哪个类”。这一篇的产品只有一种，但它复杂：零件多、可选项多、还得保证组装出来是个完整合法的东西。

一、这是什么：点一份 Subway 三明治

去 Subway 点餐是建造者模式最好的现实注解：你不是“买一个做好的三明治”，而是一步步组装——先选面包，再选肉，加不加芝士，要哪些蔬菜，最后淋什么酱。店员按你的指令分步骤搭，最后递给你一个成品。

关键在于：同样的组装流程(选面包→加料→淋酱)，换一组选择，就产出完全不同的三明治。 建造者模式就是把“分步骤构造”这件事，和“最终产品长什么样”解耦开。

配套代码用组装电脑演示：一步步 `SetCPU`、`SetMemory`、`SetGPU`，最后 `Build()` 出一台完整的机器。

二、解决什么痛：参数排到天边的构造函数

假设电脑有 4 个部件，其中显卡可选。不用建造者，你大概会走上这条不归路：

Java —— 臭名昭著的

```
new Computer("i5", "16GB", "512GB SSD");                          // 没显卡
new Computer("i9", "32GB", "2TB SSD", "RTX 4090");                // 有显卡
new Computer("i9", "32GB", "2TB SSD", "RTX 4090", true, 750);  // ...还要加电源?
// 调用处:new Computer("i9", "32GB", "2TB", "RTX4090", true, 750, false, ...)
//          ↑ 第 6 个参数 true 是啥?第 7 个呢?鬼知道。
```

痛点清清楚楚：

- 参数一多，可读性归零。 一排 `true, false, 750, null` 摆在那，谁看得懂哪个是哪个？
- 可选参数组合爆炸。 有的要显卡有的不要，你得重载出一堆构造函数，或者传一堆 `null`。
- 对象可能中途“半成品”。 要是改用一堆 setter 来配置，那么在“new 出来”到“setter 设完”之间，这个对象是不完整、不合法的——别人这时候拿去用就出事。

> 建造者隔离的“会变的东西”，是“复杂对象的组装过程”。 它把“怎么一步步搭”和“最终产品”拆开，让你能用可读的分步调用替代天书般的长构造函数，并且把“完整性校验”推迟到最后一刻统一做。

三、怎么实现：分步 + 链式 + 最后 Build 校验

Go 的建造者：每个 `SetXxx` 配一个部件并返回自身(于是能链式调用)，最后 `Build()` 统一校验并交出成品。

go/builder/main.go（核心）

```
type ComputerBuilder interface {
    SetCPU(cpu string) ComputerBuilder
    SetMemory(memory string) ComputerBuilder
    SetStorage(storage string) ComputerBuilder
    SetGPU(gpu string) ComputerBuilder
    Build() (*Computer, error)      // ← 校验在这一步集中发生
}

func (b *computerBuilder) Build() (*Computer, error) {
    if b.computer.CPU == "" {
        return nil, errors.New("缺少 CPU，无法组装电脑")    // 完整性校验
    }
    if b.computer.Memory == "" {
        return nil, errors.New("缺少内存，无法组装电脑")
    }
    // ...
    return b.computer, nil
}
```

调用处一下子就读得懂了，可选的显卡加不加随意：

```
pc, err := NewComputerBuilder().
    SetCPU("AMD Ryzen 9").
    SetMemory("64GB").
    SetStorage("4TB NVMe").
    SetGPU("AMD RX 7900").      // 想加就加，不加就省略，不用传 nil
    Build()                     // 到这里才校验+交货，拿到的一定是完整合法的电脑
```

代码里还有个 Director(指挥者) 角色：它把“办公机 = i5 + 16G + 无显卡”“游戏机 = i9 + 32G + RTX4090”这些常见配方固化下来，让调用方一句 `BuildGamingPC()` 就拿到预设配置。Director 是可选的——它封装的是“常用组装顺序/配方”，没有常用配方时大可省略。

Go 味插播：functional options

必须诚实地说：上面这种“链式 builder”虽然能用，但在 Go 社区里，更地道的做法常常是 functional options(函数选项):

Go 惯用法 —— functional options,可选项的另一种解法

```go
type Option func(*Computer)

func WithGPU(gpu string) Option { return func(c *Computer) { c.GPU = gpu } }
func WithMemory(m string) Option { return func(c *Computer) { c.Memory = m } }

func NewComputer(cpu string, opts ...Option) *Computer {
    c := &Computer{CPU: cpu, Memory: "8GB"}   // CPU 必填,其余给默认值
    for _, opt := range opts { opt(c) }
    return c
}

pc := NewComputer("i9", WithMemory("32GB"), WithGPU("RTX 4090"))  // 想配啥配啥
```

两者解决的是同一个痛(可选参数太多)，取舍在于：functional options 胜在轻、天然支持默认值、加选项不破坏调用方；经典 builder 胜在“必须分多步才能拿到成品”和“Build 时集中校验、产出不可变对象”的场景。Go 里可选参数优先想想 options，真需要严格的分步校验流程时再上完整 builder。

四、有哪些坑

- 别用建造者造简单对象。 就俩字段还搞 builder，纯属仪式感。字段少、没可选项，老老实实写构造函数。
- Build 之前的中间态别泄露出去。 builder 内部那个半成品对象，只应在 `Build()` 成功后才交给外界。别提前把内部指针递出去，否则又回到“半成品被使用”的老问题。
- 校验放哪要想清楚。 建造者的一大价值是“最后集中校验”。如果你把校验散在每个 `SetXxx` 里，就丢了这个好处——而且分步设置时，先设的字段没法校验和后设字段的关系。让 `Build()` 做最终的、整体的校验。
- 链式返回类型的小陷阱。 链式调用要求每步返回 builder 本身。Go 里若用了内嵌/组合来复用，注意返回的是不是你想要的那个具体类型，别让链断在半路。

五、开源里怎么用

- Go `strings.Builder`：标准库里顶着 Builder 名字的家伙。严格说它更偏“高效拼接”(避免字符串反复分配)，但“分步 `WriteString` → 最后 `String()` 取成品”的骨架，正是建造者的味道。
- Go gRPC / 各类客户端的 functional options: `grpc.Dial(target, grpc.WithInsecure(), grpc.WithBlock())` ——上面讲的 options 流派，是 Go 生态里“建造者痛点”的主流解法，`DialOption` 满天飞。
- Java 生态的重灾区(褒义): `StringBuilder`、`Stream.Builder`、OkHttp 的 `Request.Builder`、Lombok 的 `@Builder` 注解、protobuf 生成代码里的 `XxxBuilder` ——Java 因为没有具名/可选参数，建造者用得极其广泛，几乎是“多可选参数对象”的标配。

- Kubernetes / Docker 客户端：构造复杂的配置对象(Pod spec、容器配置)时，大量使用建造者或 options 风格，让层层嵌套的配置能一步步、可读地搭起来。

> ⚠️ 一个横向观察：“可选参数太多”这个痛，在有具名/默认参数的语言(Python、Kotlin、Swift)里，大半被语言特性直接消化了(直接 `Computer(cpu="i9", gpu="RTX4090")`)。建造者在 Java 里如此流行，恰恰因为 Java 长期缺这个特性。又一次：模式的流行度，反映的是语言的短板。

创建型还剩最后一位。下一篇原型：当“从头造一个”太贵时，我们换个思路——照着现成的，拷一个。

05 原型 · Prototype（创建型）

创建型的最后一位。前四个模式都在琢磨“怎么把一个对象造出来”，原型模式偏不：别造了，照着现成的拷一个不就完了？

一、这是什么：复印一份再改，别重写

你要写一份合同。是打开空白文档从头敲，还是找一份去年的旧合同，复制一份，改几个关键字段？显然后者快得多——绝大部分条款是现成的，你只需在副本上动几处。

原型模式就是这个思路：系统里已经有一个配置好的对象(原型)，需要新对象时，不从零构造，而是克隆这个原型，再在副本上做少量修改。

配套代码克隆图形：有一个配置好的 `Circle`，要新圆时直接 `Clone()` 出一份，改改位置就用，不用重新设置颜色、半径等一堆属性。

二、解决什么痛：新建太贵，或者你不想认识它的类

原型模式主要止两种痛：

1. “造一个”的成本很高。 有些对象初始化要查数据库、跑复杂计算、读网络、解析大文件。如果你手头已经有一个初始化好的同类对象，拷贝内存几乎总比重新初始化便宜得多。游戏里刷一片一模一样的小怪，与其每只都重新加载模型和属性，不如以一只为模板疯狂复制。
2. 你想要副本，却不想依赖它的具体类。 你手上是个抽象的 `Shape`，想复制它。但它到底是 `Circle` 还是 `Rectangle`？你不知道，也不想用一堆 `if type == Circle` 去判断再分别 new。让对象自己提供 `Clone()`，复制这件事就交给它自己——它最清楚怎么拷贝自己。

> 原型隔离的“会变的东西”，是“具体要复制哪个类”。 客户端只对着 `Clone()` 喊一声，至于复制出的是圆是方，由对象自身负责。这一点和工厂方法异曲同工，只是手段从“new”换成了“copy”。

三、怎么实现：Go 结构体天生会拷贝

这是 Go 占尽便宜的一个模式。因为 Go 的结构体赋值(`cp := *c`)本身就是逐字段的值拷贝，对只含值类型字段的结构体来说，一行就完成了克隆：

go/prototype/main.go（核心）

```go
type Shape interface {
    Clone() Shape
    Info() string
    SetPosition(x, y int)
}

type Circle struct {
    Color  string
    X, Y   int
    Radius int
}

// Clone 返回自身的一份拷贝。Go 中 *c 解引用再赋值，
// 就是逐字段值拷贝——对全值类型字段的结构体，这即是深拷贝。
func (c *Circle) Clone() Shape {
    cp := *c          // ← 一行搞定
    return &cp
}
```

用起来，改副本完全不影响原型：

```go
original := &Circle{Color: "红色", X: 10, Y: 20, Radius: 5}
cloned := original.Clone()
cloned.SetPosition(100, 200)   // 只动副本
// original 依旧是 (10,20)，纹丝不动
```

Java 里就没这么轻松了，得实现 `Cloneable` 接口、覆写 `clone()`、处理受检异常——而且默认还是浅拷贝，坑一堆(见下)。

四、有哪些坑：深拷贝 vs 浅拷贝，头号杀手

这是原型模式最危险、最必须讲清的一节。上面 `cp := *c` 之所以是完整拷贝，全靠 `Circle` 的字段都是值类型(string、int)。一旦结构体里含有指针、slice、map、channel 这类引用类型，`cp := *c` 就只拷贝了“引用”本身——副本和原型会共享底层数据：

浅拷贝陷阱 —— 副本和原型偷偷共享了同一个 slice

```
type Config struct {
    Name  string
    Tags  []string        // ← 引用类型!
}

func (c *Config) Clone() *Config {
    cp := *c               // 浅拷贝:cp.Tags 和 c.Tags 指向同一个底层数组
    return &cp
}

orig := &Config{Name: "prod", Tags: []string{"a", "b"}}
clone := orig.Clone()
clone.Tags[0] = "HACKED"    // 想改副本...
// 结果 orig.Tags[0] 也变成了 "HACKED"!两个对象共享同一个数组。
```

正确的深拷贝必须手动把引用类型字段也复制一份：

深拷贝 —— 引用类型字段要逐个

```
func (c *Config) Clone() *Config {
    cp := *c
    cp.Tags = append([]string(nil), c.Tags...)    // 复制一份新的底层数组
    return &cp
}
```

> ⚠ 这是原型模式踩坑率最高的地方，没有之一。 记死一句话：`cp := *c` 只拷贝“第一层”。 结构体里凡是指针、slice、map，浅拷贝后都和原型共享同一份底层数据，改一个动全家。深拷贝要不要做、做到几层，取决于你的对象结构——嵌套引用越深，越要小心，必要时逐层手写或借助序列化(如 `json.Marshal` 再 `Unmarshal`，虽慢但省心)。循环引用更是深拷贝的经典噩梦，遇到要专门处理。

另外两个小坑：

- Java 的 `Cloneable` 是反面教材。 《Effective Java》专门开一条批判它：`clone()` 机制设计得别扭(受检异常、`Object.clone()` 的浅拷贝默认、和构造函数不一致)。Java 社区更推荐用拷贝构造函数或拷贝工厂代替 `Cloneable`。这说明：即便是 GoF 钦定的模式，具体语言的实现也可能是个坑，别盲从。
- 别为了用而用。 如果你的对象构造成本很低，直接 new 一个新的、干干净净，比“克隆再修改”更清晰，也没有共享数据的风险。原型的价值，建立在“新建确实贵”或“确实要摆脱具体类”之上。

五、开源里怎么用

- Go 的 `Clone()` / `Copy()` 家族：`proto.Clone()`(Protocol Buffers 深拷贝一个 message)、很多库给复杂配置对象提供的 `DeepCopy()`。Kubernetes 尤其典型——它的 API 对象(Pod、

Deployment 等)几乎每个都自动生成了 `DeepCopy()` 方法(`zz_generated.deepcopy.go`),因为 K8s 内部大量传递对象副本以避免并发修改共享状态,这是原型模式在超大型项目里的工业级应用。

- Go 的值语义:`time.Time`、`net/url.URL` 等被设计成值类型,赋值即安全拷贝——语言层面把“原型”变成了默认行为。
- JavaScript `structuredClone()`:浏览器/Node 内置的深拷贝函数,一行完成复杂对象(含嵌套、循环引用)的深克隆,是原型思路的现代标配。
- 各类编辑器/设计工具的“复制粘贴”:复制一个图层、一个组件、一个节点,底层几乎都是“克隆原型对象再改属性”,和配套代码的图形克隆一模一样。

创建型五连,收工。 回头看这一组的主线:它们全在管一件事——别让“创建对象”这个动作,把你的业务代码绑死在具体类型上。 单例管数量,工厂管“造哪个”,抽象工厂管“造哪族”,建造者管“怎么拼”,原型管“照着拷”。

下一章进入结构型:对象已经造好了,该研究怎么把它们拼装成更大的结构,还不别扭。第一个出场的是最实用的老黄牛——适配器。

06 适配器 · Adapter（结构型）

进入结构型。这一章研究“怎么把对象拼装成更大的结构”，而开门第一位是最接地气的老黄牛——适配器。你几乎每天都在用它，只是没意识到。

一、这是什么：出国带的那个电源转换头

你去欧洲旅行，带了国标插头的充电器，墙上却是欧标插孔。插不进去。你不会去砸墙改插座，也不会拆了充电器——你买个电源转换头，一头接你的国标插头，一头插欧标插座，中间转换一下。

适配器模式就是这个转换头：你有一个现成的东西(充电器)，接口却和目标环境(欧标插座)不匹配，又都不能改。于是加一个中间层，把一方的接口“翻译”成另一方期望的样子。

配套代码接第三方支付：你的系统统一用 `Pay(元)` 接口，但第三方 Stripe SDK 只有 `ChargeInCents(分)`——方法名不同、单位不同。你不能改 Stripe 的源码，于是写个适配器夹在中间翻译。

二、解决什么痛：接口对不上，双方还都改不了

痛点非常具体：

- 一方是你控制不了的。 Stripe SDK 是人家的，你无权改。你的系统接口也已经被一堆代码依赖了，不好动。
- 强行让业务代码去迁就第三方，会污染整个代码库。 如果每处调用都写 `stripe.ChargeInCents(int(yuan*100))`，那么“元转分”“方法名不同”这些脏活就散得到处都是。哪天换个支付商，又要满地改。

> 适配器隔离的“会变的东西”，是“第三方/遗留代码那套不兼容的接口”。 它把“接口不匹配”这个脏活收进一个专门的类里，让业务代码永远只面对干净统一的目标接口。第三方怎么变、换成谁，只影响适配器一个文件。

三、怎么实现：包一层，翻译过去

Go 用对象适配器(组合)：适配器持有被适配对象，实现目标接口，在方法里做翻译后转调。

go/adapter/main.go（核心）

```go
// 目标接口:系统统一期望的样子(以"元"为单位)
type PaymentProcessor interface {
    Pay(yuan float64) (string, error)
}

// 被适配者:第三方 SDK,接口不兼容(以"分"为单位,方法名也不同)
type StripePayment struct{}
func (s *StripePayment) ChargeInCents(cents int) string { /* ... */ }

// 适配器:实现目标接口,内部把"元"翻译成"分"再转调被适配者
type StripeAdapter struct {
    stripe *StripePayment
}
func (a *StripeAdapter) Pay(yuan float64) (string, error) {
    if yuan <= 0 {
        return "", fmt.Errorf("金额必须为正")
    }
    cents := int(yuan * 100)                    // ← 翻译:元 → 分
    return a.stripe.ChargeInCents(cents), nil // ← 转调被适配者
}
```

于是客户端 `checkout()` 只认 `PaymentProcessor` 接口，原生的支付宝、包装过的 Stripe，在它眼里长得一模一样：

```go
checkout(&NativeAlipay{}, 99.9)                          // 原生实现
checkout(NewStripeAdapter(&StripePayment{}), 199.5)      // 适配后的第三方
```

> 经典 GoF 还分类适配器(靠多重继承同时继承目标和被适配者)和对象适配器(靠组合持有被适配者)。Go 没有继承，天然只走对象适配器这条路——而这恰恰是 GoF 更推荐的一种(组合优于继承)。

四、有哪些坑

- 它是“事后补救”，不是“事前设计”。 适配器的存在，往往意味着两套代码当初没约定好接口。它是很好的“止损”工具，但如果你在新设计里就到处需要适配器，说明接口规划出了问题——那该统一接口，而不是狂加适配层。
- 和几个“长得像”的模式分清楚，这是结构型最容易混的地方，一张表记牢：

模式	它对接口做什么	一句话意图
适配器	改接口(A 变 B)	让不兼容的能一起用
装饰器	不改接口，加功能	动态增强
代理	不改接口，控访问	挡在前面管访问
外观	简化一组接口	给复杂子系统一个简单入口

四个都是“包一层”，区别全在意图。适配器的意图独一份：转换接口。

- 别让适配器承担过多逻辑。 适配器该薄——只做接口翻译。如果你发现它里面塞了业务规则、状态管理，那它已经不是适配器了，该拆。

五、开源里怎么用

- Go `http.HandlerFunc` ——神来之笔，必看。 `http.Handler` 是个接口(要求有 `ServeHTTP` 方法)，但你写 handler 时明明只想写个普通函数。标准库定义了：

  ```
  type HandlerFunc func(ResponseWriter, *Request)
  func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) { f(w, r) }
  ```

  这就是把“函数”适配成“接口”！ `http.HandlerFunc(myFunc)` 一转，你的普通函数就满足了 `Handler` 接口。同款手法还有 `sort.Reverse`(把一个 `Interface` 适配成“反着比”的 `Interface`)。
- Go `io` 包： `io.NopCloser(r)` 把一个只有 `Read` 的 `Reader` 适配成带空 `Close` 的 `ReadCloser` ——很多 API 要求 `ReadCloser` ，而你手头只有 `Reader` ，一个适配器搞定。
- Java I/O: `InputStreamReader` 把字节流(`InputStream`)适配成字符流(`Reader`)，是 JDK 里教科书级的适配器； `Arrays.asList()` 把数组适配成 `List` 。
- 各类 SDK 的“兼容层”：接入多家云厂商/多个支付渠道/多个短信服务商时，几乎必然为每一家写一个适配器，对上暴露统一接口。这是适配器在真实业务里最高频的用法。

适配器转换接口。下一篇装饰器：同样是“包一层”，但它不改接口，而是层层叠加功能——顺便你会见到 Go 的 `io` 包这个装饰器天堂。

07 装饰器 · Decorator（结构型）

上一篇适配器“包一层”是为了改接口。这一篇同样“包一层”，目的却是在不改接口的前提下，给对象层层叠加新功能。

一、这是什么：咖啡加料，加一样算一样钱

星巴克点单：一杯 Espresso 15 块。加份牛奶，+3；再加份糖，+1.5；再加一份糖，又 +1.5。每加一“层”料，描述变长一点，价钱涨一点，但它始终还是一杯“饮品”——你还是能对它问“你叫啥”“多少钱”。

装饰器模式就是这个“加料”过程：用一个装饰对象把原对象包起来，包一层加一点功能，而包出来的东西和原对象长得一样(实现同一个接口)，于是可以继续往外包。 俄罗斯套娃、穿衣服一层层加，都是一个道理。

配套代码就是这杯咖啡：`Espresso` 外面包 `MilkDecorator`，再包 `SugarDecorator`，每层的 `Cost()` 都是“被包对象的价 + 自己这层的价”。

二、解决什么痛：继承会把你淹死

“给对象加功能”，第一反应往往是继承。但继承在这里会引发组合爆炸：

```
要"加牛奶的咖啡"⟶ 写个 MilkCoffee 子类
要"加糖的咖啡"⟶ 写个 SugarCoffee 子类
要"加牛奶又加糖的"⟶ 写个 MilkSugarCoffee 子类
要"加两份糖的"⟶ ......
```

三种料的自由组合就是 2^3 种子类，再来一种料直接翻倍。而且继承是编译期就定死的——你没法在运行时，根据用户临时的选择动态组装。这是继承的死穴：它是静态的、组合会爆炸的。

> 装饰器隔离的“会变的东西”，是“对象要叠加哪些额外职责，以及叠加的顺序”。 它用组合 + 运行时包裹代替继承 + 编译期固定，把 2^n 的子类爆炸，压成 n 个可自由拼接的装饰器。要什么料，运行时现包。

这正是 GoF 那条著名原则的活例：优先使用对象组合，而非类继承(Favor composition over inheritance)。

三、怎么实现：持有同接口对象，转调 + 增强

关键就一条：装饰器自己也实现被装饰的那个接口，并持有一个该接口的对象。方法里先转调内部对象，再叠加自己的增强。

go/decorator/main.go（核心）

```go
type Beverage interface {
    Description() string
    Cost() float64
}

type Espresso struct{}   // 具体组件:最里层的"本体"
func (e *Espresso) Description() string { return "Espresso 意式浓缩" }
func (e *Espresso) Cost() float64       { return 15.0 }

// 装饰器持有一个 Beverage(组合),自己也是 Beverage
type MilkDecorator struct {
    beverageDecorator          // 内嵌:复用"持有 wrapped"的能力
}
func (m *MilkDecorator) Description() string {
    return m.wrapped.Description() + " + 牛奶"    // 转调 + 增强
}
func (m *MilkDecorator) Cost() float64 {
    return m.wrapped.Cost() + 3.0                 // 转调 + 增强
}
```

因为包出来的还是 `Beverage`，所以能一层层套下去，顺序随便你：

```go
var b Beverage = &Espresso{}
b = NewMilkDecorator(b)     // 加奶
b = NewSugarDecorator(b)    // 加糖
b = NewSugarDecorator(b)    // 再加一份糖
// b.Description() => "Espresso 意式浓缩 + 牛奶 + 糖 + 糖"
// b.Cost()        => 15 + 3 + 1.5 + 1.5 = 21
```

四、有哪些坑

- 装饰器必须和本体“接口透明”。 包一层之后，外界不该看出它被包过——还是同一个接口。一旦装饰器加了本体没有的新方法，而调用方需要那个方法，透明性就破了(多层包裹后，里层的新方法会被“埋”住)。
- 顺序有时有意义。 “先加密再压缩”和“先压缩再加密”结果天差地别。装饰器允许自由组合，但不代表顺序无所谓——这是灵活性的代价。
- 多层包裹，调试栈很深。 一个请求穿过十层装饰器，报错时调用栈能拉很长。灵活是有代价的，层数别为炫技而堆。
- 和代理长得几乎一样，靠意图区分。 两者都“持有同接口对象再转调”。装饰器的意图是加功能(Cost 变了、Description 变了)；代理的意图是控制访问(功能不变，管的是能不能访问、何时访问)。下一篇细说。

五、开源里怎么用

- Go `io` 包 —— 全世界最好的装饰器教材，没有之一。`io.Reader` 是那个统一接口，然后你可以疯狂套娃：

```
r := os.Open("data.gz")              // 原始文件流(Reader)
r = bufio.NewReader(r)               // 装饰:加缓冲
gz, _ := gzip.NewReader(r)           // 装饰:加 gzip 解压
r = io.LimitReader(gz, 1<<20)        // 装饰:限制最多读 1MB
```

  每一层都接收一个 `Reader`、返回一个 `Reader`，和咖啡加料一模一样。读这段代码，你就彻底懂装饰器了。

- Go HTTP 中间件：`func(http.Handler) http.Handler` 这个签名就是装饰器——日志中间件、鉴权中间件、限流中间件，一层层把 handler 包起来，每层加一点横切功能。整个 Go Web 生态的中间件都建立在装饰器之上。
- Java `java.io`：`new BufferedInputStream(new FileInputStream(f))` 是 JDK 里最经典的装饰器嵌套，和 Go 的 `io` 同源同宗(装饰器模式当年很大程度就是为 I/O 流设计的)。
- Python `@decorator` 语法：虽然实现机制不同，但“用一个函数包住另一个函数、加点前后处理”的思想一脉相承，`@lru_cache`、`@log` 都是这个味。

装饰器加功能，代理控访问。它俩结构像双胞胎，意图却南辕北辙。下一篇代理，我们把这对双胞胎彻底分清。

08 代理 · Proxy（结构型）

上一篇结尾埋了个扣：代理和装饰器结构几乎一样，凭什么算两个模式？这一篇解开它。

一、这是什么：你找明星，得先过经纪人

你想请某位明星站台。你能直接打他电话吗？不能。你得先联系他的经纪人。经纪人和明星“接口一致”(对外都代表这位明星)，但他站在明星前面，替明星控制访问：帮你排期、挡掉不靠谱的邀约、代收合同、明星忙时先应付着.....你以为在跟明星打交道，其实一直在跟经纪人打交道。

代理模式就是这位经纪人：给目标对象配一个“替身”，替身和本尊实现同一个接口，调用方对着替身说话，替身在中间控制对本尊的访问。

配套代码是图片懒加载：`ImageProxy` 是替身，`RealImage`(加载很慢)是本尊。你拿到一堆代理时，一张图都还没真加载；直到某张图第一次被 `Display()`，它的代理才偷偷把真图 load 进来。

二、解决什么痛：我想在“访问”这件事上做文章

有时候，问题不在对象本身的功能，而在**“访问它”这个动作**上，你想加点控制：

- 它创建太贵，想用到才创建(虚拟代理/懒加载)——配套代码这种。
- 不是谁都能访问，想加权限检查(保护代理)。
- 它在另一台机器上，想让远程调用看起来像本地调用(远程代理)。
- 它每次算都很慢，想把结果缓存起来(缓存代理)。

这些需求的共同点：你不想改目标对象(它可能是第三方，或者职责单一不该掺和这些)，也不想让调用方感知(调用方就想正常用，不管你懒不懒加载)。于是在中间塞个替身。

> 代理隔离的“会变的东西”，是“访问目标对象的方式与时机”。目标对象只管做好它本职的事，而“何时创建、能不能访问、要不要走网络、结果缓不缓存”这些访问控制策略，全被收进代理里，两边都干净。

三、怎么实现：同接口替身 + 按需转调本尊

go/proxy/main.go（核心）

```
type Image interface { Display() string }

type RealImage struct { filename string }    // 本尊:创建代价高
func NewRealImage(f string) *RealImage {
    img := &RealImage{filename: f}
    img.loadFromDisk()                        // ← 构造即"加载",很慢
    return img
}

// 代理:和本尊同接口,持有本尊引用,延迟到首次访问才创建它
type ImageProxy struct {
    filename  string
    realImage *RealImage
}
func (p *ImageProxy) Display() string {
    if p.realImage == nil {                   // ← 访问控制:第一次才真加载
        p.realImage = NewRealImage(p.filename)
    }
    return p.realImage.Display()              // 之后复用,不再重复加载
}
```

结构上，它和装饰器几乎一字不差：都实现同接口、都持有一个同接口对象、都在方法里转调。区别只在意图——见下一节。

四、有哪些坑：和装饰器到底差在哪

这是本篇的重点。代理和装饰器是结构型里最像的一对，面试常问，记牢这张对照：

	装饰器	代理
意图	增强功能(结果变了)	控制访问(结果不变)
典型标志	层层叠加，常多层套娃	通常一层，替身管着一个本尊
谁决定包不包	客户端主动一层层包	代理自己决定何时/是否接触本尊
例子	咖啡加料、`bufio` 包 `Reader`	懒加载、权限校验、RPC stub

一句话：装饰器让对象“变强”，代理让对象“变得可控”。 装饰器改的是功能本身(咖啡变贵了、流被压缩了)；代理不碰功能，管的是“能不能用、什么时候用、用起来多贵”。

其它坑：

- 代理链路要透明。 调用方不该因为多了个代理就得改代码。一旦透明性破了(比如代理悄悄改了返回值语义)，就成了 bug 温床。

- 懒加载的并发安全。 配套代码的 `if realImage == nil` 在多 goroutine 下有竞态——真实项目要用 `sync.Once` 或加锁保护“只创建一次”。
- 远程代理的“本地假象”是把双刃剑。 让 RPC 调用看起来像本地调用很爽，但它掩盖了网络的延迟和失败。调用方以为在调本地方法，忘了处理超时/重试——这是分布式系统的经典陷阱。好用不等于没代价。

五、开源里怎么用

- RPC 桩代码(gRPC / Thrift stub)——远程代理的头号应用。 你调 `client.GetUser(ctx, req)` 像调本地方法，底层 stub(代理)把它序列化、走网络、等响应、反序列化。本尊在另一台机器上，代理制造了“本地”的假象。
- Java 动态代理(`java.lang.reflect.Proxy`)——整个 Java 企业生态的基石。 Spring 的声明式事务(`@Transactional`)、AOP 切面、MyBatis 的 Mapper 接口，全靠运行时生成代理：你调接口方法，代理在前后插入“开启事务/提交回滚”“日志”"SQL 执行“。没有代理，就没有半个 Spring。
- ORM 懒加载：Hibernate 查出一个对象，它的关联集合往往是代理，你不访问就不发 SQL，一访问才去查库——虚拟代理的教科书用法。
- Go `net/http/httputil.ReverseProxy` ：反向代理，接收请求、转发给后端、把响应带回来，中间可插改写/负载均衡/缓存。Nginx 反向代理是同一思想的基础设施级实现。

到这里，四个“包一层”的模式(适配器改接口、装饰器加功能、代理控访问)集齐三个，还差一个外观——它包的不是一个对象，而是一整个子系统。下一篇外观。

09 外观 · Facade（结构型）

前三个结构型模式都在包“一个对象”。外观不一样——它包的是一整个子系统。

一、这是什么：家庭影院的“一键观影”

你家客厅堆着投影仪、功放、灯光、流媒体盒子。想看场电影，标准流程是：把灯调暗到 20%、开投影仪、切到流媒体输入源、开功放、音量调到 60、启动播放器选片……七八步，一步不能少，顺序还不能乱。

太累了。于是你装了个智能面板，上面就一个按钮：「观影模式」。一按，上面那七八步自动全做完。这个按钮就是外观：它背后是复杂的子系统，对你只暴露一个简单入口。

配套代码就是这个：`HomeTheaterFacade.WatchMovie("盗梦空间")` 一句话，内部替你协调投影仪、功放、灯光、播放器四个子系统，按正确顺序全部就位。

二、解决什么痛：客户端被子系统的细节淹没

一个功能强大的子系统，内部往往有很多类、很多步骤、很多“必须按这个顺序调”的隐含规则。如果让每个调用方都去直接对付这些细节：

- 调用方被迫理解一大堆内部知识——它明明只想“看个电影”，却要懂投影仪的输入源、功放的音量单位。
- 调用顺序的规则散落在各处，谁都可能调错(先开播放器后开投影仪，黑屏)。
- 子系统一改，所有调用方跟着遭殃——内部多加个“预热”步骤，几十处调用全要改。

> 外观隔离的“会变的东西”，是“子系统的内部结构与协作细节”。它在客户端和子系统之间架一道“墙”，墙上开一扇简单的门(高层方法)。客户端只跟门打交道，子系统内部怎么折腾、加几个步骤、换个实现，都被挡在墙后，伤不到客户端。这直接呼应了迪米特法则(最少知识原则)：别跟陌生人说话，只跟你的直接朋友(外观)说话。

三、怎么实现：一个高层方法，协调一群子系统

外观本身没什么玄机——它就是持有各子系统，在高层方法里按正确顺序把它们串起来。

go/facade/main.go（核心）

```go
// 外观:持有一群子系统
type HomeTheaterFacade struct {
    projector *Projector
    amplifier *Amplifier
    lights    *Lights
    player    *StreamingPlayer
}

// 一个高层方法,把"观影"需要的所有子系统操作按序封装好
func (h *HomeTheaterFacade) WatchMovie(movie string) {
    h.lights.Dim(20)
    h.projector.On()
    h.projector.SetInput("流媒体")
    h.amplifier.On()
    h.amplifier.SetVolume(60)
    h.player.Play(movie)       // ← 客户端这一切都不用知道
}
```

客户端的世界一下子清净了：

```go
theater := NewHomeTheaterFacade()
theater.WatchMovie("盗梦空间")    // 一句话,七八步全搞定
theater.EndMovie()               // 一句话,全部关闭复位
```

四、有哪些坑

- 外观不是“墙”，是“门”——它不禁止你直接进子系统。 这是外观和“封装/黑箱”的关键区别：提供了 `WatchMovie` 的便捷入口，不代表禁止高级用户绕过它、直接操作投影仪做精细控制。外观是“给大多数人的方便法门”，不是“唯一法门”。留这个口子很重要。
- 警惕外观膨胀成“上帝对象”。 图省事，什么方法都往外观里塞，最后它变成一个几千行、无所不知、无所不管的巨型类——这比没有外观还糟。一个外观对应一个清晰的使用场景/子系统，别让它长成怪物。真需要多个场景，就拆多个外观。
- 和适配器分清： 适配器是转换一个接口(让不兼容的能对接)，意图是“兼容”；外观是简化一组接口(给复杂子系统一个简单入口)，意图是“省事”。适配器改的是“接口长相”，外观改的是“使用难度”。

五、开源里怎么用

- Go `http.Get(url)` ——你天天用的外观。 这一句背后，是 `http.Client`、`http.Transport`、`http.Request`、连接池、重定向处理一大套子系统。`http.Get` 把它们全封在身后，给你一个极简入口。同理 `os.ReadFile`、`os.WriteFile` 都是文件操作子系统的外观。
- 各类 SDK 的 `Client`：云服务/支付/短信 SDK 里那个 `client.SendSMS(...)`，内部封装了鉴权签名、构造请求、重试、解析响应一整套流程。你调一个方法，它替你跑完全程——这是外观在商业 SDK 里最普遍的形态。

- 日志门面(SLF4J、Go `log/slog`)：名字里就带“门面”(facade)。它给五花八门的日志后端提供统一的高层 API，你面向门面写日志，底层用 logback 还是 zap，门面替你隔离。
- 框架的“启动器”(Spring Boot Starter、各类 `bootstrap()`)：一句 `SpringApplication.run()` 背后是扫描、装配、启动容器、起 Web 服务器无数步骤——外观思想在框架层面的极致体现。

结构型过半。已出场的四位(适配器、装饰器、代理、外观)都在管“包装与简化”。接下来两位换个主题——管“多维度”和“树形结构”。下一篇桥接：当一个东西有两个独立变化的维度时，怎么不让类的数量乘起来爆炸。

10 桥接 · Bridge（结构型）

桥接是结构型里公认最难理解的一个，倒不是因为它复杂，而是因为 GoF 给它起的名字(“抽象部分”“实现部分”)极其抽象，劝退了一批人。别怕，我们用大白话把它拆开。

一、这是什么：遥控器 × 设备，两个能各自进化的维度

你有遥控器，也有被控的设备。现在注意，这里有两个会独立变化的维度：

- 遥控器这一维会变：有基础遥控器，有带静音的高级遥控器，以后可能有带语音的.....
- 设备这一维也会变：能控电视，能控收音机，以后可能控空调、控音箱.....

关键洞察：这两个维度是正交的、各自独立进化的。“高级遥控器”应该能控任何设备，“电视”应该能被任何遥控器控。它们不该被绑死。

桥接模式就是：把这两个维度拆成两套独立的类层次，让“遥控器”持有一个“设备”(这就是那座“桥”)，两边各自随便扩展，谁也不拖累谁。

二、解决什么痛：继承会让类的数量“乘起来”

假设你头铁，用继承来做。你会得到：

```
BasicRemoteForTV、BasicRemoteForRadio、
AdvancedRemoteForTV、AdvancedRemoteForRadio.....
```

2 种遥控器 × 2 种设备 = 4 个类。现在加一种设备(空调)，变成 2×3=6；再加一种遥控器，3×3=9.....类的数量是两个维度相乘的。这就是继承的“笛卡尔积爆炸”：每个维度增长，总数是乘法级膨胀。

桥接把它从乘法变回加法：

	继承(维度耦合)	桥接(维度解耦)
2 遥控 × 2 设备	2×2 = 4 个类	2 + 2 = 4 个类
3 遥控 × 4 设备	3×4 = 12 个类	3 + 4 = 7 个类
M 遥控 × N 设备	M×N	M + N

维度一多，M+N 对 M×N 是碾压性的优势。

> 桥接隔离的“会变的东西”，是“两个正交维度各自的变化”。 它认定这两维互不相干，于是干脆拆成两条平行的继承线，中间用一个“持有”关系(组合)搭桥。任一维度加东西，都是给那条线加一个类(加法)，而不是给整个矩阵填一格(乘法)。

三、怎么实现：抽象持有实现，搭一座桥

先破除名词障碍。GoF 说的：

- “抽象部分”(Abstraction) = 遥控器这一维(高层控制逻辑，面向用户)。
- “实现部分”(Implementor) = 设备这一维(底层具体干活)。

注意这里的“实现”不是“接口的实现类”那个意思，而是“另一个维度里真正干活的那方”。这个词是所有误解的源头，记住它=设备维就行。

go/bridge/main.go（核心）

```go
// 实现维:设备(TV/Radio 各自实现)
type Device interface {
    Name() string; IsEnabled() bool; Enable(); Disable()
    GetVolume() int; SetVolume(percent int)
}

// 抽象维:遥控器,持有一个 Device —— 这个"持有"就是那座"桥"
type RemoteControl struct {
    device Device         // ← 桥:遥控器这一维,通过它连到设备那一维
}
func (r *RemoteControl) TogglePower() string {
    if r.device.IsEnabled() { r.device.Disable(); return r.device.Name()+" 已关闭
    r.device.Enable(); return r.device.Name()+" 已开启"
}

// 扩展抽象维:高级遥控器,组合复用基础遥控器,再加新功能
type AdvancedRemoteControl struct {
    RemoteControl         // 内嵌复用
}
func (r *AdvancedRemoteControl) Mute() string {
    r.device.SetVolume(0); return r.device.Name()+" 已静音"
}
```

于是任意遥控器 × 任意设备自由组合，加维度只需加一个类：

```go
basic := &RemoteControl{device: NewTV()}                  // 基础遥控 控 电视
adv := NewAdvancedRemoteControl(NewRadio())               // 高级遥控 控 收音机
adv.Mute()                                                // 高级功能,对任何设备都好使
```

四、有哪些坑

- 最大的坑是“看不出该用它”。 桥接的收益，建立在你**提前识别出“这里有两个独立维度”**之上。识别对了，它优雅无比；识别错了(其实只有一个维度，或者两维根本纠缠在一起)，硬套桥接就是给自己找麻烦。经验：当你发现类名开始出现 `AForB`、`XxxWithYyy` 这种“两个词拼起来”的组合命名时，警报响起——大概率该上桥接了。
- 和策略模式结构像，意图不同。 两者都是“持有一个接口对象”。但策略是“换算法”(同一件事的不同做法，运行时替换)，桥接是“解耦两个维度”(两件正交的事，各自演化)。策略通常一个维度，桥接明确是两个。
- 和适配器分清： 适配器是事后补救(两套接口没对上，加层翻译)，桥接是事前规划(设计之初就预见双维度，主动拆开)。一个救火，一个防火。
- 别过早桥接。 只有一个维度会变的时候，引入桥接纯属自缚手脚。等第二个独立维度真的冒头，再重构成桥接不迟。

五、开源里怎么用

- Go `database/sql` —— 一座教科书级的桥。 `sql.DB` / `sql.Rows` 这套是“抽象维”(统一的数据库操作 API，你的业务代码用它), `database/sql/driver` 定义的接口由各家驱动实现，是“实现维”(MySQL 驱动、Postgres 驱动各自干活)。你的 SQL 代码(抽象维)和具体数据库(实现维)独立变化：换数据库只换驱动，业务代码不动；标准库升级 API，驱动也不用重写。这就是桥接。
- JDBC:Java 世界的同款——`java.sql` 是抽象维，各厂商的 `Driver` 是实现维， `DriverManager` 负责搭桥。GoF 书里桥接的原型场景之一。
- 日志门面 × 后端：SLF4J(抽象维)× logback/log4j(实现维);Go `log/slog` 的 `Logger`(抽象维)× `Handler`(实现维)。你写日志的代码和“日志往哪写、什么格式”两个维度彻底解耦。
- 跨平台绘图/GUI：一个 `Shape` / `Window` 抽象维，配一套各平台的绘制“实现维”(Windows GDI、macOS Quartz、Linux X11)。图形逻辑和平台绘制各自演化——GoF 原书举的正是这个例子。

桥接管“两个维度”，下一篇组合管“树形结构”：当对象自己能包含同类对象、层层嵌套成一棵树时，怎么让“单个”和“一组”被一视同仁地对待。

11 组合 · Composite（结构型）

结构型最后两位。这一篇讲树：当对象能装下同类对象、层层嵌套时，怎么让“一个”和“一堆”用起来毫无差别。

一、这是什么：文件夹里能装文件，也能装文件夹

看你电脑的文件系统：一个文件夹里，能放文件，也能放子文件夹；子文件夹里又能放文件和更深的文件夹……无限递归下去，长成一棵树。

现在你要算“这个文件夹总共多大”。你会怎么想？——“把里面每一项的大小加起来”。而“每一项”可能是文件(直接有大小)，也可能是子文件夹(它的大小又是它里面所有项之和)。神奇的是：你处理它们的方式一模一样——都是问一句“你多大？”，不用管它到底是文件还是文件夹。

组合模式就是干这个的：把对象组织成树形结构，并让你用统一的方式对待“叶子”(单个对象)和“容器”(一组对象)。 公司组织架构(部门里有员工也有子部门)、菜单(菜单项里有子菜单)、GUI 控件树，全是这个结构。

配套代码就是文件系统：`File`(叶子)和 `Directory`(容器)都实现同一个 `FileSystemNode` 接口，算总大小时对二者一视同仁。

二、解决什么痛：客户端被迫区分“单个”和“一组”

如果没有统一接口，处理树会痛成这样：

```
// 痛点:每碰一个节点都要先问"你是文件还是文件夹?"
func totalSize(node any) int64 {
    switch n := node.(type) {
    case *File:
        return n.size
    case *Directory:
        var sum int64
        for _, child := range n.children {
            sum += totalSize(child)    // 还得手动递归,还得手动分类型
        }
        return sum
    }
    return 0
}
```

到处是 `if 是文件夹 { 递归 } else { 直接取 }`。这种类型判断散落在每个要遍历树的地方，加一种节点类型(比如“符号链接”)就得把所有这种 `switch` 找出来改一遍。

组合隔离的“会变的东西”，是“某个节点到底是叶子还是容器”。它让叶子和容器实现同一个接口，把“我是谁、我要不要递归”的判断，从客户端收进每个节点自己内部。客户端只管对着统一接口喊 `Size()`，递归由容器节点自己负责——客户端代码里再也不会出现“如果是文件夹就……”。

三、怎么实现：叶子与容器实现同一接口，容器负责递归

go/composite/main.go（核心）

```go
// 统一接口:文件和目录都是"文件系统节点"
type FileSystemNode interface {
    Name() string
    Size() int64
    Print(indent string)
}

// 叶子:文件,大小就是自己的大小
type File struct{ name string; size int64 }
func (f *File) Size() int64 { return f.size }

// 容器:目录,大小 = 递归累加所有孩子(孩子是文件还是目录?不关心!)
type Directory struct {
    name     string
    children []FileSystemNode      // ← 装的是接口,文件目录皆可
}
func (d *Directory) Size() int64 {
    var total int64
    for _, child := range d.children {
        total += child.Size()       // ← 统一调 Size(),递归自然发生
    }
    return total
}
```

于是搭一棵树、算总大小，干净利落，全程没有一次类型判断：

```go
root := NewDirectory("project")
root.Add(src).Add(docs).Add(NewFile("go.mod", 50))   // 目录里混装目录和文件
fmt.Println(root.Size())                              // 一句话递归算完
```

四、有哪些坑

- 透明性 vs 安全性：组合模式最经典的取舍。`Add` / `Remove`(管理孩子)这类方法该放哪？两派：
  - 透明式：放进统一接口，叶子和容器都有 `Add`。好处是客户端完全不用区分二者(最透明)；坏处是“给一个文件 `Add` 一个孩子”在语义上是错的，叶子只能给个空实现或报错(不安全)。
  - 安全式：`Add` / `Remove` 只放在容器(`Directory`)上。好处是叶子压根没有 `Add`，类型安全；坏处是客户端想加孩子时，得先确认“这是个容器”(牺牲一点透明)。

配套代码选了安全式(只有 `Directory` 有 `Add`)，这也更符合 Go 的口味(小接口 + 需要时类型断言)。两条路没有绝对对错，看你更怕“叶子被误用”还是更怕“客户端要判断类型”。

- 递归深了会栈溢出。 树特别深(比如几万层)时，朴素递归会爆栈。真遇到超深树，得改用显式栈的迭代遍历。
- 组合天生和迭代器、访问者是好搭档。 遍历这棵树用迭代器，对树上每个节点执行不同操作用访问者——后面会讲到，它们常常一起出现。

五、开源里怎么用

- Go `go/ast`(抽象语法树):Go 编译器/工具链把源码解析成一棵 AST，每个节点都实现 `ast.Node` 接口，`ast.Inspect` 递归遍历整棵树——你写 `gofmt`、`golint`、代码生成器时打交道的就是它。教科书级的组合。
- Go `io/fs`：`fs.FS` 抽象出的文件树，`fs.WalkDir` 统一遍历目录和文件，不用区分。
- DOM 树 / 虚拟 DOM：浏览器的 DOM(`Node` 有 `Element`、`Text` 等，元素能嵌套元素)、React/Vue 的虚拟 DOM，都是组合模式的巨型实例——整个前端建立在这棵树上。
- GUI 控件树：几乎所有 UI 框架(Swing 的 `Container` / `Component`、Qt、Flutter 的 Widget 树)都用组合：容器控件里放叶子控件或子容器，布局、绘制、事件分发全靠对这棵树的统一递归。

组合管“树形结构”。结构型最后一篇享元：当树上/场景里的对象多到成千上万、内存扛不住时，怎么靠“共享”把它们塞进有限的内存里。

12 享元 · Flyweight（结构型）

结构型收官。这一篇是唯一一个为“性能/内存”而生的模式——当对象多到以百万计，它教你怎么把它们塞进有限的内存。

> *“"Flyweight“是拳击里的”蝇量级“(最轻量级)。这个名字本身就在说：让每个对象尽可能轻。”*

一、这是什么：游戏地图里的一片森林

你在做一个游戏，地图上要渲染一百万棵树。每棵树对象都存：名称、颜色、纹理贴图(可能几 MB)、加上自己的坐标。一百万棵 × 每棵几 MB = 内存直接爆炸。

但你冷静看一眼：这一百万棵树，其实翻来覆去就那么几种——松树、枫树、橡树。同一种松树，它们的名称、颜色、纹理完全一样，唯一不同的只是长在哪(坐标)。

那还存一百万份纹理干嘛？把“松树的名称/颜色/纹理”只存一份，让所有松树共享；每棵树自己只留一个坐标 + 一个指向“松树数据”的引用。 一百万棵树，纹理数据却只有 3 份。这就是享元。

配套代码正是这个森林：`TreeType`(名称/颜色/纹理)被共享，种 5 棵树只创建了 2 个 `TreeType`。

二、解决什么痛：海量对象，状态大量重复

痛点很硬核，就是内存：

- 对象数量巨大(百万级的树、编辑器里每个字符、地图上每个图标)。
- 这些对象的状态，很大一部分是重复的(同种树的纹理、同个字母的字形数据)。
- 如果每个对象都完整存一份，内存被大量重复数据吃光。

享元的破解思路，是把对象的状态劈成两半：

状态	叫法	特点	例子	存哪
可共享的	内在状态 (intrinsic)	不随实例变、可复用	树的名称/颜色/纹理	存一份，共享
随实例变的	外在状态 (extrinsic)	每个实例都不同	树的坐标 (x,y)	每个实例自己存/外部传入

> 享元隔离的“会变的东西”，恰恰是通过分离“不变的东西”实现的。 它把“到处重复的、不变的内在状态”抽出来集中共享一份，把“每个实例独有的外在状态”剥离出去。核心动作就一个：分清哪些状态能共享、哪些不能——这是用享元最关键、也最烧脑的一步。

三、怎么实现：享元工厂管缓存，复用内在状态

实现有两个要件：一个不可变的享元对象(存内在状态)，一个享元工厂(缓存已创建的享元，相同的直接复用)。

go/flyweight/main.go（核心）

```go
// 享元:内在状态(可共享),相同名称/颜色/纹理的树共用一个实例
type TreeType struct {
    Name, Color, Texture string      // ← 内在状态,不可变
}

// 享元工厂:缓存并复用,相同内在状态绝不重复创建
type TreeFactory struct {
    types map[string]*TreeType
}
func (f *TreeFactory) GetTreeType(name, color, texture string) *TreeType {
    key := name + "|" + color + "|" + texture
    if t, ok := f.types[key]; ok {
        return t                     // ← 已有,直接复用同一个对象
    }
    t := &TreeType{name, color, texture}
    f.types[key] = t                 // ← 没有,创建并缓存
    return t
}

// 树:外在状态(坐标) + 指向共享内在状态的引用
type Tree struct {
    X, Y int                         // ← 外在状态,每棵不同
    Type *TreeType                   // ← 引用共享的内在状态
}
```

结果：种了 5 棵树，`TreeType` 只造了 2 个(松树、枫树各一)，其余全是共享引用。树的数量再翻万倍，`TreeType` 还是那 2 个。内存占用从“棵数 × 完整数据”降到“棵数 × 一个指针 + 极少数几份完整数据”。

四、有哪些坑

- 划分内在/外在状态是全部难点。 划错了，享元就失效甚至出 bug。判据：在实例间会变的，一定是外在状态，不能塞进共享对象；实例间恒定的，才是内在状态。上面若手滑把坐标放进 `TreeType`，那所有树就会共享同一个坐标，全叠在一个点上——灾难。
- 享元对象必须不可变。 既然被无数实例共享，谁都不能改它——改一个等于改所有。共享对象一旦可变，就是并发和逻辑双重噩梦。内在状态创建后只读，是铁律。

- 只有对象数量巨大时才划算。 享元引入了工厂、缓存、内外状态分离的复杂度。对象就几十个？这套开销纯亏，直接每个存一份更清晰。它是“以复杂度换内存”的交易，只有内存真的紧张、对象真的海量时才成立。
- 共享带来的线程安全。 多线程访问同一个享元工厂的缓存 `map`，要加锁(Go 里 `map` 并发写会 panic)。共享的东西越多，并发考量越重。

五、开源里怎么用

- Java `Integer.valueOf()` 的缓存池 —— 面试常客，享元实锤。 Java 对 `-128 ~ 127` 的 `Integer` 做了缓存：`Integer.valueOf(100)` 多次调用返回同一个对象。所以 `Integer a = 100, b = 100; a == b` 是 `true`，而 `200 == 200`(超出缓存范围)却是 `false` ——这个著名的“坑”，本质就是享元池。`Boolean`、`Character`、短 `Long` 都有类似缓存。
- 字符串驻留(String Interning):Java 的字符串常量池、Go/Python 对字符串字面量的驻留，让内容相同的字符串共享一份存储——享元思想在语言运行时层面的落地。
- 文字排版/浏览器渲染：一个字体里字母 `a` 的字形(glyph)轮廓数据只存一份，页面上出现一万个 `a`，共享这一份字形，各自只带位置和大小——GoF 原书举的正是文档编辑器这个例子。
- 游戏引擎：粒子系统、植被系统、地图瓦片(tile)，海量重复元素共享网格/纹理/材质(内在状态)，各自只带变换矩阵(外在状态)。没有享元，开放世界游戏的内存根本扛不住。

结构型七连，收工。 回看这一组主线：它们全在管“怎么把对象拼成更大的结构还不别扭”——适配器/装饰器/代理/外观管“包装”，桥接管“多维度”，组合管“树”，享元管“海量共享”。

下一章进入最大的一族行为型(11 个)：对象都造好、也拼好了，现在研究它们之间怎么分工、怎么通信、怎么协作。第一个出场的是行为型里最实用、也最能体现“面向接口编程”的——策略模式。

13 策略 · Strategy（行为型）

进入最大的一族——行为型，11 个，管的是“对象之间怎么分工、怎么协作”。打头阵的策略模式，是最实用、也最能体现“面向接口编程”的一个。

一、这是什么：导航 App 里的“路线偏好”

打开高德/Google Maps 导航，同样是从 A 到 B，它给你几个选项：最快路线、最短路线、避开高速、少走收费。目的地没变，变的是“怎么算这条路”的算法。你点一下，算法就换一套，重新给你规划。

策略模式就是这个“路线偏好”：把“完成某件事的不同做法(算法)”各自封装成一块，让它们能互相替换，而使用者可以在运行时自由切换，用哪块就换哪块。

配套代码是支付：同样是“付 299 元”，可以用信用卡、PayPal、加密货币——每种是一个策略，购物车运行时想换哪个换哪个。

二、解决什么痛：一个函数里塞满 if-else

不用策略，多种做法往往挤在一个函数里：

```
// 痛点:每加一种支付方式,就得回来改这个函数,它只会越来越肥
func (c *Cart) Checkout(method string) string {
    if method == "creditcard" {
        // ...信用卡的一堆逻辑...
    } else if method == "paypal" {
        // ...PayPal 的一堆逻辑...
    } else if method == "crypto" {
        // ...加密货币的一堆逻辑...
    }
    // 再加一种?继续往下堆 else if......
}
```

这段代码的病：它把“所有做法”焊死在一个函数里。加一种支付方式要改它，改一种做法也要动它，几种算法的代码还搅在一起互相干扰。经典的开闭原则违反现场。

> 策略隔离的“会变的东西”，就是“算法本身”。把每种算法拎成独立的一块，彼此隔离、可插拔。加算法 = 加一块新策略，老代码一行不动；用哪个 = 运行时把那块塞进去。`Checkout` 从此只负责“调用当前策略”，不再关心到底有几种支付方式。

三、怎么实现：Go 用函数值，轻到看不见

经典 OOP(Java)里，策略是“一个策略接口 + 每种算法一个实现类”。但在 Go 里，一个函数类型就是最好的策略载体——算法本来就是“输入→输出”的函数，何必包成类？

go/strategy/main.go（核心）

```go
// 策略就是一个函数类型:收金额,返回结果。不需要接口 + 一堆类。
type PaymentStrategy func(amount float64) string

// 每种"具体策略"是一个返回该函数的构造(闭包绑定各自的数据)
func CreditCardPay(cardNumber string) PaymentStrategy {
    return func(amount float64) string {
        return fmt.Sprintf("信用卡尾号%s 支付 %.2f 元", cardNumber[len(cardNumber)
    }
}

// 上下文:持有一个可随时替换的策略
type ShoppingCart struct {
    amount   float64
    strategy PaymentStrategy
}
func (c *ShoppingCart) SetPaymentStrategy(s PaymentStrategy) { c.strategy = s }
func (c *ShoppingCart) Checkout() string { return c.strategy(c.amount) }
```

运行时切换，行云流水：

```go
cart := NewShoppingCart(299.5)
cart.SetPaymentStrategy(CreditCardPay("4111111111111234"))
cart.Checkout()                                    // 走信用卡
cart.SetPaymentStrategy(PayPalPay("alice@x.com")) // 换 PayPal
cart.Checkout()
```

> 对比一下就懂 Go 的取舍了：Java 要为策略写 `interface PaymentStrategy` + `CreditCardStrategy`、`PayPalStrategy` 好几个类；Go 用 `type PaymentStrategy func(...)` 加闭包，把整个模式压成几行。当算法是“无状态的一个动作”时，函数值几乎总比策略接口更地道。只有当策略需要携带较多状态/多个方法时，才值得上接口。

四、有哪些坑

- 和状态模式长得一模一样，务必分清。 两者都是“上下文持有一个可替换的接口对象”。区别在意图和驱动方：策略的多个算法互不知情，由客户端从外部主动选(“我要用信用卡”)；状态的多个状态彼此知道、会自动流转，由内部驱动(“播放中→按暂停→自动变成已暂停”)。结构双胞胎，灵魂两个人。第 18 篇状态会再对照一次。
- 策略太多也是负担。 几十种策略散在几十个地方，“该用哪个”本身就成了问题。有时配一个“策略工厂/注册表”按 key 取策略，能收敛这种混乱。

- 客户端得知道策略的存在。 选择的责任在调用方——它得知道有哪些策略、什么场景用哪个。策略模式把算法解耦了，但没消灭“选择”这件事。

五、开源里怎么用

- Go `sort.Slice(s, less)`：那个 `less func(i, j int) bool` 就是比较策略。同一份排序算法，你传不同的 `less`，就能按不同规则排——策略模式在标准库里最优雅的一次现身，而且正是用函数值实现的。
- Go `http.Client.Transport`：`RoundTripper` 接口是可替换的“怎么发这个请求”策略——默认走网络，测试时换成 mock，加个带重试/日志的包装也行。
- 可插拔的算法族：压缩(gzip/zstd/snappy)、哈希(md5/sha256)、负载均衡(轮询/随机/最少连接)、限流(令牌桶/漏桶)——凡是“同一件事有多种算法、想运行时切换”的地方，底层几乎都是策略。
- Java `Comparator`：`Collections.sort(list, comparator)` 和 Go 的 `sort.Slice` 是一回事，Java 世界里策略模式最常见的入口。

策略让“算法”可插拔。下一篇观察者：当一个对象变了、一大群对象要跟着响应时，怎么让它们松耦合地联动——这是“事件驱动”整个世界的地基。

14 观察者 · Observer（行为型）

如果说有哪个模式撑起了半个软件世界，那就是观察者：GUI 的事件、前端的响应式、消息队列的发布订阅、微服务的事件驱动……全是它的变体。

一、这是什么：关注了一个公众号

你关注了某个公众号。作者发新文章时，主动推送给每一个关注者；你不用天天去他主页刷“更新了没”。你随时能关注(订阅)，也随时能取关(退订)。作者不认识你是谁，他只管“群发给所有关注者”。

观察者模式就是这套订阅关系：一个“主题”(被观察者)维护一份“观察者”名单，当自己状态变化时，自动通知名单上的每一个观察者。一对多，而且能动态增删订阅者。

配套代码是气象站：`WeatherStation`(主题)温度一变，自动通知所有订阅的显示设备(手机、电子广告牌)，它们各自更新。

二、解决什么痛：主题不该硬编码知道所有观察者

假设气象站要在温度变化时更新手机和广告牌。最直接的写法：

```
// 痛点:气象站被迫认识每一个具体的显示设备
func (w *WeatherStation) SetTemperature(t float64) {
    w.temperature = t
    w.phoneDisplay.Update(t)       // ← 硬编码依赖手机
    w.billboard.Update(t)          // ← 硬编码依赖广告牌
    // 新加一个网页显示?回来改这里。再加一个手表?再改……
}
```

病根：主题和每个观察者死死耦合。加一种显示设备就得改气象站；气象站还得持有所有设备的具体引用。主题这个“发布方”被拖进了“到底有哪些订阅方”的泥潭里，而这本不该是它操心的事。

> 观察者隔离的“会变的东西”，是“到底有哪些对象关心这个变化、以及它们各自怎么响应”。主题只面对一个抽象的“观察者名单”，发通知时挨个 `Update` 一遍，根本不关心名单里具体是谁、它们收到后干什么。订阅方随时增删，主题岿然不动——发布方和订阅方彻底解耦。

三、怎么实现：名单 + 通知

主题维护观察者切片，提供订阅(`Attach`)、退订(`Detach`)、通知(`Notify`)；状态变化时遍历名单挨个通知。

go/observer/main.go（核心）

```go
type Observer interface{ Update(temperature float64) }

type WeatherStation struct {
    observers   []Observer      // ← 订阅者名单,只认抽象接口
    temperature float64
}
func (w *WeatherStation) Attach(o Observer) { w.observers = append(w.observers,
func (w *WeatherStation) Notify() {
    for _, o := range w.observers {
        o.Update(w.temperature)  // ← 挨个通知,不关心具体是谁
    }
}
func (w *WeatherStation) SetTemperature(t float64) {
    w.temperature = t
    w.Notify()                   // ← 状态一变,自动广播
}
```

订阅、退订都是运行时的事：

```go
station.Attach(phone); station.Attach(billboard)  // 两个都订阅
station.SetTemperature(26.5)                      // 俩都收到
station.Detach(phone)                             // 手机退订
station.SetTemperature(18.0)                      // 只剩广告牌收到
```

四、有哪些坑

- 忘记退订 = 内存泄漏。 这是观察者头号事故。主题的名单一直引用着观察者，观察者就永远不会被回收——哪怕它早该销毁了。GUI 里“页面关了监听器还在”“对象没了事件还绑着”，十有八九是忘了 `Detach`。订了就要记得退。
- 别在 `Update` 里改主题状态。 观察者收到通知后又去改主题，可能触发新一轮通知，轻则顺序混乱，重则无限循环。通知过程中，保持观察者“只读响应”。
- 同步通知会被慢观察者拖垮。 `Notify` 挨个同步调用，只要有一个观察者的 `Update` 很慢(比如发网络请求)，整条通知链就被它卡住。观察者多、响应重时，考虑异步通知(下面 channel 版天然适合)。
- 推模型 vs 拉模型。 `Update(temperature)` 直接把数据“推”给观察者(简单，但主题得预判观察者要什么)；另一种是只通知“我变了”，观察者自己回来“拉”需要的数据(灵活，但多一次交互)。按需求选。

> Go 味：很多时候 channel 才是更地道的“观察者”。 Go 的 `chan` + goroutine 天生就是发布-订阅：主题往 channel 发消息，多个订阅者从各自 channel 收。它天然异步、天然并发安全、天然解耦，连 `Attach/Detach` 的名单管理都能用 channel 的关闭来表达。写 Go 时，先想想“这个联动能不能用 channel”，往往比手搓观察者接口更清爽。

五、开源里怎么用

- 前端事件系统：`element.addEventListener('click', handler)` 就是观察者——DOM 元素是主题，你注册的回调是观察者。整个浏览器交互模型建立在此之上。
- 响应式编程(RxJS、Vue 的响应式、React 的状态订阅)：数据变了，依赖它的视图自动重渲染——观察者的工业级放大版。
- 消息队列(Kafka / RabbitMQ 的 pub-sub)：生产者发布到 topic，所有订阅该 topic 的消费者收到——观察者模式在分布式系统里的形态，主题和观察者甚至不在同一台机器上。
- Java: `java.util.Observer`(注意：已被官方废弃，因为设计过于简陋，是个反面教材)、`PropertyChangeListener`、Swing 的各种事件监听器。废弃 `Observer` 这件事本身也说明：观察者的思想永恒，但具体 API 可以很烂，别照抄。

观察者是“一个变、多个响应”的广播。下一篇命令：我们把“一个操作请求”本身打包成对象，于是它能被存储、排队、记录，甚至撤销和重做。

15 命令 · Command（行为型）

上一篇观察者把“事件”解耦。这一篇更进一步：把“操作”本身变成一个可以攥在手里的对象。

一、这是什么：餐厅的点菜单

你在餐厅点菜，服务员写下一张点菜单：桌号、菜品、数量。这张单子是个“实实在在的东西”——它可以被传到后厨、排进队列、几张单子一起处理、下错了能撤单。而“炒一盘宫保鸡丁”这个动作本身是抓不住的，一旦做了就过去了。点菜单的意义，就是把“要做什么”从“实际去做”里剥离成一个可保存、可传递的凭证。

命令模式就是这张点菜单：把一个请求(要执行的操作 + 需要的参数)封装成一个对象。 于是这个“操作”可以被存起来、排队、记日志、撤销、重做——这些都是“方法调用”这个转瞬即逝的动作做不到的。

配套代码是遥控器：每个按钮绑定一个命令对象(开灯、关灯)，按下就 `Execute()`，而且遥控器记着历史，能一路 `Undo()` 回去。

二、解决什么痛：方法调用“抓不住”

`light.On()` 是个动作，调完就没了。但现实里，你常常想对“操作”这件事做更多文章：

- 撤销/重做：得记住“做过什么”，才能倒回去。
- 排队/延迟执行：把操作攒起来，稍后或异步执行(任务队列)。
- 记录/重放：把操作写进日志，崩溃后重放恢复(事务、WAL)。
- 参数化：让“按钮”这种通用组件，能绑定任意操作，而不用为每个按钮写死逻辑。

这些都要求“操作”是个能存、能传的对象，而不是一次性的方法调用。命令模式就是把动作“对象化”。

> 命令隔离的“会变的东西”，是“具体要执行什么操作”。 它在“发起请求的人”(遥控器/按钮)和“真正干活的人”(灯)之间，插入一个“命令对象”。发起者只管 `Execute()`，完全不知道也不关心这个命令背后是开灯、关门还是发射火箭。于是同一个按钮能绑任意命令，操作还能被存起来秋后算账。

三、怎么实现：把操作 + 接收者打包

命令对象持有“接收者”(真正干活的对象)和“要调的操作”，实现统一的 `Execute()` / `Undo()`。调用者只面对命令接口。

go/command/main.go（核心）

```go
type Command interface {
    Execute() string
    Undo() string
}

// 具体命令:持有接收者(light),把"开灯"打包;Undo 就是反操作"关灯"
type LightOnCommand struct{ light *Light }
func (c *LightOnCommand) Execute() string { return c.light.On() }
func (c *LightOnCommand) Undo() string    { return c.light.Off() }

// 调用者:只认 Command 接口,并记录历史以支持撤销
type RemoteControl struct{ history []Command }
func (r *RemoteControl) PressButton(cmd Command) {
    fmt.Println(cmd.Execute())
    r.history = append(r.history, cmd)      // ← 存起来,为了能撤销
}
func (r *RemoteControl) PressUndo() {
    last := r.history[len(r.history)-1]
    r.history = r.history[:len(r.history)-1]
    fmt.Println("撤销:", last.Undo())         // ← 倒着回去
}
```

因为操作被存进了 `history`，撤销就是把最近的命令挨个 `Undo()`——这正是所有编辑器 Ctrl+Z 的原理骨架。

四、有哪些坑

- 撤销要存够“回退所需的状态”。 `Undo` 有两种实现：一是反操作(开灯↔关灯，像配套代码这样，简单)；二是操作不可逆时，得在 `Execute` 前存快照， `Undo` 时整体恢复——这就和备忘录模式搭上了(第 21 篇)。改一段文字很难“反操作”，只能存下改之前的样子。
- 命令多了，类会膨胀。 每个操作一个命令类，操作一多，类爆炸。Go 里可以用函数(`type Command func()`)或闭包大幅简化，不必每个都建类型——和策略同款思路。
- 宏命令(组合命令)：把多个命令打包成一个“一键执行全部”，本质是命令 + 组合模式。“一键离家”(关灯+锁门+关空调)就是宏命令。
- 和策略分清： 策略是“做同一件事的不同算法”(几种支付方式，可互换)；命令是“把一个请求打包成对象”(强调存储/撤销/排队)。策略关注“怎么做”，命令关注“把‘做什么’变成可管理的对象”。

五、开源里怎么用

- 编辑器/设计工具的 undo/redo:VS Code、Photoshop、Figma 的历史记录，每一步操作都是一个命令对象压进撤销栈——命令模式最闪耀的舞台。
- 任务队列 / 异步作业：把“要干的活”封成命令对象扔进队列，worker 取出来 `Execute` 。Go 里往 channel 里发的 job、各类分布式任务框架(Celery、Sidekiq)的 task，都是命令。

- 事务与预写日志(WAL)：数据库把每个操作记成日志条目(命令)，崩溃后按日志重放恢复，或按逆命令回滚——命令模式在存储引擎里的硬核应用。
- Java `Runnable` / Go 的 `func()` 作任务：把“一段要执行的逻辑”封成对象/闭包交给线程池或 goroutine——这本身就是命令模式最轻量的形态，你可能天天在用却没意识到。

命令把“操作”对象化。下一篇模板方法：当一堆流程“骨架相同、只有个别步骤不同”时，怎么把骨架写一次、把变化的步骤留给各自填。

16 模板方法 • Template Method（行为型）

这是 23 个模式里最朴素的一个，朴素到你八成早就在无意识中用过。它的核心就一句话：把不变的流程骨架固定下来，把会变的步骤挖空留给别人填。

一、这是什么：泡茶和泡咖啡，流程八成一样

泡一杯茶：烧开水 → 泡茶叶 → 倒进杯子 → 加柠檬。 冲一杯咖啡：烧开水 → 冲咖啡粉 → 倒进杯子 → 加奶和糖。

看出来了吗？流程骨架一模一样(烧水→冲泡→倒杯→加料)，只有第 2 步和第 4 步的具体做法不同。你不会为茶和咖啡各写一套完全独立的流程——那“烧水”“倒杯”的逻辑就重复了。

模板方法就是：在一个方法里定义好流程的骨架(步骤顺序固定)，把其中会变的步骤声明成“待填的空”，由具体的子类型去填。 骨架只写一次，变化的步骤各填各的。

配套代码就是这杯饮料：`Prepare()` 是固定骨架，`Brew()`(怎么冲泡)和 `AddCondiments()`(加什么料)是挖空的步骤，`Tea` 和 `Coffee` 各自填。

二、解决什么痛：流程重复，却又不完全一样

如果茶和咖啡各写各的：

```
func (t *Tea) Prepare() {
    boilWater()           // ← 重复
    steepTea()            // 不同
    pourInCup()           // ← 重复
    addLemon()            // 不同
}
func (c *Coffee) Prepare() {
    boilWater()           // ← 和上面一样,重复了
    brewCoffee()          // 不同
    pourInCup()           // ← 又重复
    addMilkSugar()        // 不同
}
```

“烧水”“倒杯”这些公共步骤被抄了两遍。加个“泡奶茶”，再抄一遍。共同的流程骨架散落在每个变体里，改一处骨架(比如“倒杯前先温杯”)要改所有变体。

> 模板方法隔离的“会变的东西”，是“流程中那几个具体步骤”，同时锁死“流程的骨架与顺序”。它把“稳定的编排”和“变化的步骤”分开：骨架收拢到一个模板方法里写一次，变化点挖成“空”交给具体类型填。这样公共流程再不重复，而各变体只需关心“我这几步怎么做”。

三、怎么实现：Go 用组合 + 接口，而非继承

经典模板方法靠继承：父类写模板方法(骨架)和抽象步骤方法，子类覆写抽象步骤。但 Go 没有继承，它用接口委托 + 组合表达同一意图：

go/template-method/main.go（核心）

```go
// 会变的步骤,声明成接口(相当于"挖的空")
type BeverageSteps interface {
    Name() string
    Brew() string
    AddCondiments() string
}

// 模板:持有 steps,Prepare 就是那个"模板方法"——骨架固定,空由 steps 填
type Beverage struct{ steps BeverageSteps }
func (b *Beverage) Prepare() {
    fmt.Println("1. 烧开水")                // 固定
    fmt.Println("2.", b.steps.Brew())      // ← 空:交给具体类型
    fmt.Println("3. 倒入杯中")              // 固定
    fmt.Println("4.", b.steps.AddCondiments()) // ← 空:交给具体类型
}

// 具体步骤:茶,只需填自己那几步
type Tea struct{}
func (t *Tea) Brew() string           { return "用沸水浸泡茶叶" }
func (t *Tea) AddCondiments() string { return "加入柠檬片" }
```

用起来，骨架被复用，步骤各填各的：

```go
NewBeverage(&Tea{}).Prepare()       // 按骨架泡茶
NewBeverage(&Coffee{}).Prepare()    // 同一套骨架冲咖啡
```

> ⚠ 注意 Go 版的微妙之处：它其实介于“模板方法”和“策略”之间——`Beverage` 组合了一个 `BeverageSteps`(像策略)，但 `Prepare` 固定编排这些步骤(像模板方法)。经典 Java 模板方法是“子类继承父类、覆写钩子”，编译期绑定；Go 用组合把“变化的步骤”作为一个对象注入，更灵活。这再次印证：没有继承的语言，模板方法会自然长成组合的样子。

四、有哪些坑

- 它是“好莱坞原则”的化身：“别调用我，我会调用你”(Don't call us, we'll call you)。 控制权在骨架(高层)手里，它在需要时回调你填的步骤(低层)，而不是你去调它。这正是框架和库的根本区别——框架用模板方法调用你的代码(你填 `doGet`，框架的 `service()` 骨架来调你)，库是你调它的代码。
- 别把太多步骤开放为可覆写。 挖的“空”越多，骨架就越不稳定，复用价值越低，甚至退化成“什么都能改”。只挖真正该变的那几步，其余焊死。
- 钩子(hook)方法。 除了必填的步骤，还可以留“可选钩子”(默认空实现，子类想插手才覆写)，比如“要不要加料”由钩子 `wantsCondiments()` 决定。这是模板方法的常见增强。

- 和策略的分野： 模板方法固定整个流程骨架、只让你换其中几步(粒度细，骨架是老大)；策略是把整个算法换掉(粒度粗，一次换一坨)。一个是“填空”，一个是“整段替换”。

五、开源里怎么用

- Go `sort.Sort(data)`：标准库定义了排序的骨架，要求你的类型实现 `Len()` / `Less()` / `Swap()` 三个步骤。排序算法(骨架)标准库写好，“怎么比、怎么换”由你填——浓浓的模板方法味(虽然它也常被归为策略，两者本就相邻)。
- Java `AbstractQueuedSynchronizer`(AQS):JUC 并发包的基石， `ReentrantLock`、`Semaphore`、`CountDownLatch` 全建在它之上。AQS 用模板方法固定了“排队/阻塞/唤醒”的骨架，只让子类填 `tryAcquire` / `tryRelease` 这几步——模板方法的巅峰之作。
- Java `HttpServlet`:`service()` 是骨架，根据请求方法回调 `doGet()` / `doPost()` ——你只填后者。所有 Servlet 框架的核心套路。
- 测试框架：`setUp() ⟶ test() ⟶ tearDown()` 的固定生命周期骨架，你只填中间的测试逻辑——JUnit、Go `testing`、pytest 全是这个结构。

模板方法固定“流程骨架”。下一篇迭代器：怎么顺序访问一个集合的元素，却不用关心它内部到底是数组、链表还是树——顺便看看 Go 1.23 给这个古老模式带来的现代答案。

17 迭代器 • Iterator（行为型）

这个模式你每天都在用，只是语言帮你藏起来了。每次写 `for range`，背后就是迭代器。

一、这是什么：换台不用拆开电视

拿遥控器换台，你按“下一个频道”，电视就跳到下一台。你根本不知道也不需要知道电视内部这些频道是存在数组里、链表里还是哈希表里——你只用“下一个”这个统一动作，就能把所有频道挨个过一遍。

迭代器模式就是这个“下一个”按钮：提供一种统一的方式，顺序访问一个集合里的元素，而不暴露集合的内部结构。 不管集合底层是数组、链表、树，遍历它的姿势都一样。

配套代码是书籍集合：`BookCollection` 内部用切片存书，但对外给一个 `Iterator`（`HasNext` / `Next`），你遍历时只管“还有吗？下一本”，完全不碰它内部的切片。

二、解决什么痛：遍历逻辑和集合结构绑死

如果没有迭代器，遍历就得依赖集合的内部实现：

- 集合底层是切片，你写 `for i := 0; i < len(c.books); i++`——你被迫知道它是切片，还得碰它的内部字段 `books`。
- 哪天它内部改成链表或树，你所有遍历它的代码全得重写。
- 想给同一个集合提供多种遍历方式(正序/倒序/层序)，或者同时开两个游标各遍历各的，朴素 `for` 索引就捉襟见肘了。

> 迭代器隔离的“会变的东西”，是“集合的内部存储结构”。 它在集合和“遍历它的代码”之间加一层：集合负责“怎么造一个能走遍自己的迭代器”，迭代器负责“记住走到哪了、怎么给下一个”。于是遍历代码只依赖 `HasNext/Next` 这个统一接口，集合内部换成什么数据结构，都伤不到它。

三、怎么实现：集合造迭代器，迭代器记位置

集合提供 `CreateIterator()`，返回一个记录着“当前遍历位置”的迭代器对象。

go/iterator/main.go（核心）

```
type Iterator interface {
    HasNext() bool
    Next() *Book
}

// 集合:内部用切片,但对外只暴露"造一个迭代器"
type BookCollection struct{ books []*Book }
func (c *BookCollection) CreateIterator() Iterator {
    return &bookIterator{collection: c, index: 0}
}

// 迭代器:维护当前位置 index,把"走到哪了"这个状态封在自己身上
type bookIterator struct {
    collection *BookCollection
    index      int
}
func (it *bookIterator) HasNext() bool { return it.index < len(it.collection.boo
func (it *bookIterator) Next() *Book {
    book := it.collection.books[it.index]
    it.index++
    return book
}
```

遍历方只面对统一接口，不知道背后是切片：

```
it := collection.CreateIterator()
for it.HasNext() {
    book := it.Next()
    fmt.Printf("《%s》—— %s\n", book.Title, book.Author)
}
```

四、有哪些坑

- 遍历中修改集合，后果自负。 一边迭代一边往集合里加/删元素，是经典 bug 源(索引错位、跳过元素、甚至崩溃)。Java 的迭代器为此设计了 fail-fast——发现集合被并发改动就立刻抛异常；Go 里 `for range` 一个 map 时增删也是未定义行为。迭代期间别动集合。
- 外部迭代 vs 内部迭代。 上面这种“你主动调 `Next()` 往前走”是外部迭代(控制权在你)；另一种是把回调传进去、集合自己遍历并对每个元素调你的回调(内部迭代，如 `collection.Each(func(b){...})`)。前者灵活(能随时停/跳)，后者简洁。
- 多个迭代器互不干扰。 每个迭代器各存各的 `index`，所以能对同一集合同时开好几个游标各走各的。这是“把遍历状态放进独立迭代器对象”带来的好处——别把遍历位置存到集合自己身上，那就只能有一个游标了。

> Go 味：这个模式在 Go 里高度“语言内建”。 绝大多数时候你根本不用手写迭代器——内置的 `for range` 已经能遍历 slice/map/channel。而 Go 1.23 引入的 range-over-func(`iter.Seq`) 更是把自定义迭代器正式变成了语言特性：你写一个 `func(yield func(V) bool)`，就能被

`for v := range seq` 直接遍历。这是迭代器模式在现代 Go 里的官方答案，比手写 `HasNext/Next` 优雅得多。

五、开源里怎么用

- Go `database/sql` 的 `Rows`：`for rows.Next() { rows.Scan(&x) }`——查询结果集就是个迭代器，你一行行取，完全不用管底层游标怎么从数据库拉数据。教科书级实例。
- Go `bufio.Scanner`：`for scanner.Scan() { line := scanner.Text() }`——把一个流按行/按词迭代，`Scan()`/`Text()` 就是 `HasNext()`/`Next()` 的化身。
- Go 1.23 `iter.Seq` / `maps.Keys` / `slices.Values`：标准库开始大量返回迭代器函数，配合 `for range` 使用——现代 Go 的迭代器生态。
- Java `Iterator`/`Iterable`：`for-each` 循环的底层就是它，`Collection` 全家都实现 `Iterable`。几乎所有语言的 `for-each`/`for-in` 语法糖，底下都是迭代器模式。

迭代器让“遍历”与“结构”解耦。下一篇状态：当一个对象的行为随它的内部状态而变(播放器在“播放中”和“已停止”时，同一个按钮干的事不一样)，怎么摆脱满屏的 `if (state == ...)`。

18 状态 · State（行为型）

上一篇迭代器管“遍历”。这一篇管“一个对象在不同状态下，行为完全不同”这件事——它和策略是长得最像的一对，我们这次把它们彻底掰开。

一、这是什么：音乐播放器的那个按钮

音乐播放器上那个播放/暂停按钮，按下去会发生什么，取决于它现在的状态：

- 正在播放时按它 → 暂停。
- 已暂停时按它 → 继续播放。
- 已停止时按它 → 从头播放。

同一个操作(“按按钮”)，在不同状态下行为截然不同。而且状态之间还会流转：播放→暂停→播放→停止……红绿灯(红→绿→黄)、订单(待付款→已付款→已发货)、电梯，全是这种“行为随状态变、状态还会自动跳转”的东西。

状态模式就是：把每一种状态封装成一个独立的对象，让对象把行为委托给“当前状态对象”。状态变了，行为自然跟着变；状态转换也由状态对象自己驱动。

配套代码是音频播放器：`PlayingState` / `PausedState` / `StoppedState` 各自实现 `Play/Pause/Stop`，`AudioPlayer` 把操作全委托给“当前状态”。

二、解决什么痛：if(state==…) 地狱

不用状态模式，行为随状态变通常写成一堆条件判断，散落在每个方法里：

```
// 痛点:每个方法都要把所有状态判断一遍,状态一多就是灾难
func (p *Player) Play() {
    if p.state == "playing" {
        fmt.Println("已经在播放了")
    } else if p.state == "paused" {
        p.state = "playing"; fmt.Println("恢复播放")
    } else if p.state == "stopped" {
        p.state = "playing"; fmt.Println("开始播放")
    }
}
func (p *Player) Pause() { /* 又把 3 种状态判断一遍 */ }
func (p *Player) Stop()  { /* 再把 3 种状态判断一遍 */ }
```

病灶清清楚楚：同一套状态判断，在 `Play` / `Pause` / `Stop` 里各抄一遍。加一个状态(比如“缓冲中”)，你得找出每一个方法，都加一个 `else if`——漏一个就是 bug。状态和行为的对应关系被打散成一地碎

片。

> 状态隔离的“会变的东西”，是“对象当前处于哪个状态，以及该状态下的行为”。 它把“每个状态下该怎么响应各种操作”从散落的 `if-else` 里收拢，一个状态一个类，该状态的所有行为集中在一处。加状态 = 加一个类，而不是去每个方法里加分支。原本纵横交错的“状态×操作”矩阵，被整齐地切成了一列一列。

三、怎么实现：状态对象化，行为委托给当前状态

每个状态是一个实现了统一接口的对象，上下文持有“当前状态”，把操作转发给它；状态对象在响应时顺便把上下文切到下一个状态。

go/state/main.go（核心）

```
type PlayerState interface {
    Play(player *AudioPlayer)
    Pause(player *AudioPlayer)
    Stop(player *AudioPlayer)
    Name() string
}

// 具体状态:播放中。它知道"在我这个状态下,各操作该怎么响应、该切到哪个状态"
type PlayingState struct{}
func (s *PlayingState) Play(p *AudioPlayer)  { fmt.Println("已经在播放了") }
func (s *PlayingState) Pause(p *AudioPlayer) {
    fmt.Println("暂停播放")
    p.SetState(&PausedState{})      // ← 状态自己驱动转换:播放中→已暂停
}

// 上下文:把操作全委托给"当前状态",自己不写任何 if
type AudioPlayer struct{ state PlayerState }
func (p *AudioPlayer) Play()  { p.state.Play(p) }
func (p *AudioPlayer) Pause() { p.state.Pause(p) }
```

`AudioPlayer` 的方法里再也没有状态判断了——它只是“把活派给当前状态”。加“缓冲中”状态？写个 `BufferingState` 实现接口即可，老状态一行不改。

四、有哪些坑

- 状态模式 vs 策略模式：全系列最经典的辨析，必须记牢。 两者代码结构一模一样(上下文持有一个接口对象，把行为委托出去)。区别在灵魂：

	策略	状态
谁来切换	客户端从外部主动选(“我要用信用卡”)	状态对象自己内部驱动(“暂停后自动进入已暂停”)
彼此知不知情	各策略互不知道对方	各状态知道下一个是谁(要负责转换)
意图	封装可互换的算法	封装随状态而变的行为 + 状态流转

一句话：策略是“你选一个算法用”，状态是“对象自己在几个状态间走来走去”。 结构双胞胎，用途两码事。

- 状态转换逻辑放哪？ 上面是“状态对象自己决定下一个状态”(去中心化，加状态方便，但转换规则散在各状态里)。另一种是把“谁能转到谁”的规则集中放在上下文或一张转换表里(集中，规则清晰，但上下文变重)。状态多、转换复杂时，后者更好维护。
- 状态爆炸。 状态特别多、转换关系特别密时，类的数量和它们之间的跳转会变得难以管理。这时该考虑正经的状态机建模(状态图 + 转换表)，甚至上状态机库/工作流引擎，而不是硬堆状态类。

五、开源里怎么用

- TCP 连接状态机：`LISTEN → SYN_SENT → ESTABLISHED → FIN_WAIT → CLOSED`.....网络协议栈是状态模式(状态机)最硬核的应用，每个状态下对“收到某种包”的响应完全不同。
- 工作流 / 订单流转引擎：订单、审批、工单的状态流转(待处理→处理中→已完成)，各类工作流引擎和 Go 的 FSM 库(如 `looplab/fsm`)就是把状态模式产品化。
- 游戏 AI：敌人的“巡逻/追击/攻击/逃跑”状态机，每个状态下的行为和转换条件各异——游戏开发里状态机无处不在。
- 正则表达式引擎 / 词法分析器：编译原理里的有限状态自动机(DFA/NFA)，本质就是状态模式的数学化身。

状态让对象“随状态改变行为”。下一篇责任链：当一个请求可能由多个处理者之一负责、而你不想让发送者知道到底该谁处理时，把处理者串成一条链，让请求顺着链找到它的归宿——这正是所有 Web 中间件的骨架。

19 责任链 · Chain of Responsibility（行为型）

这个模式你只要写过 Web 后端就一定见过——所有 Web 框架的“中间件”骨架，就是责任链。

一、这是什么：公司的报销审批

你交一张报销单。800 块，经理直接批了；要是 1 万 5，经理权限不够，转给总监；要是 8 万，总监也批不了，再转给 CEO。你只管把单子交给第一个人(经理)，它会自动沿着“经理→总监→CEO”这条链往上走，直到遇到有权批的人。 你不需要知道“这个金额到底该找谁签”，链条自己会把它送到位。

责任链模式就是这条审批链：把多个处理者串成一条链，请求沿着链传递，每个处理者要么处理它、要么甩给下一个，直到有人处理(或走到链尾)。 发送者不需要知道最终是谁处理的。

配套代码正是采购审批：`Manager`(限额 5000)→`Director`(2 万)→`CEO`(10 万)，金额超了就`passToNext`往上转。

二、解决什么痛：发送者被迫知道“该谁处理”

不用责任链，“根据条件找处理者”通常又是一坨`if-else`：

```
// 痛点:发送方被迫掌握全部的分派规则
func handle(req *PurchaseRequest) {
    if req.Amount <= 5000 {
        manager.Approve(req)
    } else if req.Amount <= 20000 {
        director.Approve(req)
    } else if req.Amount <= 100000 {
        ceo.Approve(req)
    }
    // 加一级审批?加一档金额?回来改这里。
}
```

病根：发送方被迫知道整条分派规则——每个处理者的能力边界、谁在谁前面。审批层级一变(加个“VP”档)，这里就得改；而且这套规则和“发起请求”的代码耦合死了。

> 责任链隔离的“会变的东西”，是“一个请求到底该由谁处理、以及处理者的排列顺序”。 它把“分派决策”从发送方剥离，分散到链上每个处理者身上——每个处理者只需回答一个简单问题：“这个我能处理吗？不能就往下传。”发送方只管把请求丢给链头，彻底不关心它最终花落谁家。增删处理者、调整顺序，只是重新接一下链。

三、怎么实现：每个处理者持有“下一个”

每个处理者持有指向“下一个处理者”的引用；处理不了就转交。Go 用组合复用“转交”这段公共逻辑。

go/chain-of-responsibility/main.go（核心）

```
// 基础处理者:封装"转交下一个"的公共逻辑,供各级组合复用
type baseApprover struct{ next Approver }
func (b *baseApprover) SetNext(next Approver) Approver { b.next = next; return n
func (b *baseApprover) passToNext(req *PurchaseRequest) {
    if b.next != nil {
        b.next.Approve(req)          // ← 有下一个,甩给它
        return
    }
    fmt.Println("无人可审批,已拒绝")  // ← 走到链尾的兜底
}

// 具体处理者:能处理就处理,不能就 passToNext
type Manager struct {
    baseApprover
    limit float64
}
func (m *Manager) Approve(req *PurchaseRequest) {
    if req.Amount <= m.limit {
        fmt.Println("经理批准了")      // 我的权限够,处理掉,链终止
        return
    }
    m.passToNext(req)                 // 超我权限,往上转
}
```

组装链就是把它们串起来，发起时只找链头：

```
manager.SetNext(director).SetNext(ceo)   // 串成 经理→总监→CEO
manager.Approve(req)                       // 只管交给链头,自动流转到合适的人
```

四、有哪些坑

- 请求可能走到链尾也没人处理。 必须有兜底(像配套代码那样，到尾了打印“已拒绝”)，否则请求“石沉大海”是很隐蔽的 bug。永远为“没人处理”留个结局。
- 链的顺序即行为。 处理者的排列顺序直接决定结果(先经理还是先总监，天差地别)。责任链很灵活，但这份灵活也意味着“接错链”是一类新 bug。
- 和装饰器“都是链”，区别在意图。 装饰器链上每一环都会执行，层层叠加增强(咖啡每种料都加上)；责任链上通常只有一环真正处理(或部分处理后传递)，重点是“找到那个该处理的人然后停下”。一个“人人有份”，一个“能者上”。
- 别为俩处理者就上责任链。 就两个固定分支，一个 `if-else` 更直白。责任链的价值在“处理者多、顺序可能变、还想动态增删”时才显现。

五、开源里怎么用

- Web 中间件 —— 责任链最闪耀的舞台。 Go 的 `net/http` 中间件、Gin/Echo 的 handler chain、Java 的 Servlet `Filter` chain、Node.js Express 的 `app.use()` ——每个中间件拿到请求，处理自己那部分(鉴权、日志、限流、CORS)，然后 `next()` 传给下一个，或者直接拦截返回(如鉴权失败)。你写的每一个 Web 服务，都跑在一条责任链上。
- Netty 的 `ChannelPipeline`：网络数据在一条 handler 链上流动，每个 handler 处理或传递——高性能网络框架的核心结构。
- 日志框架的 Handler/Appender 链：一条日志记录依次流过多个处理器(格式化、过滤、写文件、发网络)。
- 异常处理：`try-catch` 一层层往上抛，直到某层捕获——语言层面内建的一种“责任链”思想。

责任链让请求“顺着链找到归宿”。下一篇中介者：当一堆对象两两之间互相引用、乱成一张网时，引入一个“中间人”把网状关系收拢成星状——从此大家只跟中间人打交道。

20 中介者 · Mediator（行为型）

上一篇责任链把处理者串成“线”。这一篇处理的是更麻烦的拓扑——一堆对象乱成一张“网”。

一、这是什么：机场塔台

想象一个没有塔台的机场：每架飞机要起降，都得和其他每一架飞机直接通话协调——“我要降落，你让让”“你先起飞我等你”。10 架飞机就是几十条两两通话，乱成一锅粥，还极易出事。

现实中当然不是这样。所有飞机只和塔台通话，由塔台统一协调谁先谁后。飞机之间互不直接联系，全靠塔台这个“中间人”居中调度。10 架飞机，10 条“飞机↔塔台”的线，清清爽爽。

中介者模式就是这座塔台：用一个中介者对象封装一组对象之间的交互，让这些对象不再互相直接引用，而是都只跟中介者打交道。 网状的多对多关系，被收拢成星状的一对多。

配套代码是聊天室：`User` 之间不直接持有对方，发消息全通过 `ChatRoom`(中介者)转发，用户彼此解耦。

二、解决什么痛：N 个对象两两耦合 = N² 张网

一组对象如果两两直接通信，关系数量是平方级增长的：

```
// 痛点:每个用户都得持有其他所有用户的引用
type User struct {
    friends []*User    // ← 我要认识所有人才能给他们发消息
}
func (u *User) Send(msg string) {
    for _, f := range u.friends {
        f.Receive(msg)    // 直接调用每一个
    }
}
```

3 个用户还好，30 个呢？每加一个用户，要把它塞进其他所有人的名单；改一处交互规则(比如“禁言某人”)，得在每个对象里改。对象间的关系变成一张 N×N 的蛛网，牵一发动全身，谁也没法单独复用(一个 `User` 拖着对所有其他 `User` 的依赖)。

> 中介者隔离的“会变的东西”，是“对象之间那套错综复杂的交互逻辑”。 它把散落在各个对象里的“我该怎么和其他人打交道”抽出来，集中到中介者一处。每个对象从此只认识中介者一个，变得干净、可独立复用；而“谁和谁怎么互动”的规则，全收敛到中介者里统一维护。网状 → 星状，平方 → 线性。

三、怎么实现：大家只认中介者

对象持有中介者引用，要和别人交互时，不直接找对方，而是喊中介者；中介者负责协调转发。

go/mediator/main.go（核心）

```go
// 中介者:聊天室,负责协调所有用户之间的消息转发
type ChatRoom struct{ users []*User }
func (r *ChatRoom) Broadcast(sender *User, message string) {
    for _, u := range r.users {
        if u != sender {
            u.Receive(sender.Name, message)    // ← 转发由中介者统一负责
        }
    }
}

// 同事对象:用户,只持有中介者,不直接引用其他用户
type User struct {
    Name     string
    mediator ChatMediator       // ← 只认识"塔台",不认识其他"飞机"
}
func (u *User) Send(message string) {
    u.mediator.Broadcast(u, message)    // ← 我不群发,我告诉中介者,它来
}
```

用户之间零直接引用，加人、改转发规则都只动中介者：

```go
alice.Send("大家好! ")    // Alice 只跟 ChatRoom 说,ChatRoom 转发给 Bob、Carol
```

四、有哪些坑

- 头号大坑：中介者膨胀成“上帝对象”。 这是中介者的阿喀琉斯之踵。你把对象间的复杂度从“四散的网”收进了中介者，但如果不加节制，这些复杂度会全部堆积在中介者内部，让它变成一个几千行、无所不知、无所不管的巨无霸。中介者是把复杂度“搬家”，不是“消灭”——搬进去之后还得好好组织，否则你只是把一团乱麻从外面塞进了一个盒子。真复杂时，中介者内部可能还要再拆分。
- 和观察者的区别。 观察者是单向的一对多广播(主题变→通知观察者，观察者一般不“回话”)；中介者是双向的多对多协调(同事们互相之间的双向交互都经它中转)。聊天室里 A 发 B 收、B 也能发 A 收，是双向的，所以是中介者而非观察者。
- 和外观的区别。 外观是单向简化——客户端通过外观调用子系统，但不管子系统内部之间怎么协作；中介者管的正是同级对象之间的相互协作。外观对着“外面”(简化调用入口)，中介者对着“里面”(协调内部成员)。

五、开源里怎么用

- MVC/MVP 里的 Controller/Presenter：它常常扮演 Model 和 View 之间的中介者——View 的操作交给 Controller,Controller 协调 Model 更新再刷新 View,View 和 Model 不直接对话。

- 事件总线 / Event Bus / 消息总线：组件不互相直接调用，而是把事件发到总线，由总线分发。这是中介者在大型前端(如各种全局 EventBus)和后端(进程内消息总线)里的常见形态。
- UI 对话框协调多个控件：一个复杂表单里，“选了 A 就禁用 B、填了 C 就显示 D”这类联动，若让控件两两直接引用会乱套，通常由对话框本身作中介者统一协调。
- 微服务编排(Orchestration)：一个编排服务协调多个下游微服务的调用顺序与依赖，让下游服务彼此解耦——中介者思想在分布式架构层面的放大(与之相对的“编舞 Choreography”则更接近观察者/事件驱动)。

中介者把“网”收成“星”。下一篇备忘录：怎么保存一个对象某时刻的完整状态，以便之后一键恢复(撤销/读档)，还不破坏它的封装——这是所有“Ctrl+Z”和“游戏存档”的底层原理。

21 备忘录 · Memento（行为型）

所有的“Ctrl+Z”、所有的“游戏存档读档”，底层都是这个模式。

一、这是什么：游戏存档点

打游戏，到一个存档点存个档。之后你浪一波，不小心团灭了——没关系，读档，一切回到存档那一刻。存档这个动作，把你当时的完整状态(等级、装备、位置、血量)拍了个快照封存起来；读档就是拿这个快照把状态整个还原。

备忘录模式就是这套存档/读档机制：在不破坏封装的前提下，捕获一个对象的内部状态并保存在外部，以便之后把对象恢复到这个状态。

配套代码是文本编辑器：`Save()` 把当前内容拍成一个 `EditorMemento`(快照)存进历史，`Restore()` 从某个快照恢复内容——这正是 Ctrl+Z 的原理。

二、解决什么痛：想存档，又不想扒光对象

你想实现撤销，于是需要“保存对象之前的状态”。最粗暴的做法：把对象的内部字段全掏出来，存到外面某个地方。

```
// 痛点:为了存档,把编辑器的内部实现细节全暴露给了外部
type History struct {
    savedContent   string    // 直接存内部字段
    savedCursor    int       // 编辑器内部还有光标...
    savedSelection []int     // ...还有选区...
    // 编辑器一改内部结构,History 跟着改;而且谁都能读写这些字段
}
```

病根：为了存档，破坏了对象的封装。History 现在知道了编辑器的所有内部细节，两者紧耦合；编辑器内部一重构，History 就崩；而且这些本该私有的状态，现在裸露在外，谁都能乱改。

> 备忘录隔离(保护)的，是“对象的内部状态封装”。它让对象自己把状态打包成一个“备忘录”对象(只有它自己看得懂里面是啥)，外部只负责保管这个不透明的包，却看不透、也改不了里面的内容。这样既实现了“存档/恢复”，又没让内部细节泄露出去——鱼和熊掌兼得。

三、怎么实现：三个角色各司其职

备忘录模式有三个角色，分工是理解它的关键：

- 发起人(Originator)：被存档的对象(`TextEditor`)。只有它能创建备忘录(`Save`)和从备忘录恢复(`Restore`)。
- 备忘录(Memento)：状态快照(`EditorMemento`)。对发起人透明，对外界是个黑盒。
- 管理者(Caretaker)：保管快照的人(`History`)。只负责存取，绝不窥探备忘录内部。

go/memento/main.go（核心）

```
// 备忘录:状态快照,对外只读
type EditorMemento struct{ content string }
func (m *EditorMemento) Content() string { return m.content }

// 发起人:只有它会造快照、用快照恢复
type TextEditor struct{ content string }
func (e *TextEditor) Save() *EditorMemento    { return &EditorMemento{content: e
func (e *TextEditor) Restore(m *EditorMemento) { e.content = m.Content() }

// 管理者:只保管快照栈,不关心快照里装的是什么
type History struct{ snapshots []*EditorMemento }
func (h *History) Push(m *EditorMemento) { h.snapshots = append(h.snapshots, m)
func (h *History) Pop() (*EditorMemento, error) { /* 弹出最近一个 */ }
```

于是撤销就是“取出上一个快照，让编辑器恢复它”：

```
editor.Type("第一段"); history.Push(editor.Save())   // 存档
editor.Type("误操作的第二段")                          // 浪一波
m, _ := history.Pop(); editor.Restore(m)             // 读档,回到第一段
```

> 关键在“管理者不偷看备忘录”。`History` 只把 `EditorMemento` 当成一个不透明的东西存进栈、取出来，从不读它的 `content`(在严格实现里，`content` 对 History 甚至是不可见的)。这正是备忘录保护封装的精髓：保管的人不需要、也不应该知道自己保管的是什么。这也是它和“把字段掏出来存”的本质区别。

四、有哪些坑

- 内存是最大的敌人。每存一次档就是一份状态拷贝。状态大(比如整个文档、整张游戏地图)、存得勤(每敲一个字存一次)，内存很快撑爆。对策：限制历史长度(只留最近 N 步)、增量快照(只存“变化的部分”而非全量)、或用命令模式的逆操作代替全量快照(见下)。
- 深拷贝陷阱，和原型一模一样。如果状态里有引用类型(slice、map、指针)，快照必须深拷贝，否则“快照”和“当前状态”共享底层数据，你改当前状态，快照也跟着变，存了个寂寞。这是备忘录第二高频的坑。
- 和命令模式实现 undo 的分工。两条路：命令模式靠记录“逆操作”来撤销(开灯↔关灯，省内存，但要求操作可逆)；备忘录靠存“整个状态快照”来恢复(操作不可逆时的唯一选择，但费内存)。实战中常二者配合：命令负责“记录做了哪些操作”，在操作难以逆转时，命令内部用备忘录存一份操作前的快照。

- 多态状态的恢复。 若发起人状态复杂(不止一个字符串字段)，备忘录要完整捕获，恢复时也要完整还原，别漏字段。

五、开源里怎么用

- 编辑器/IDE 的 undo/redo 栈：VS Code、IntelliJ、Word 的撤销历史，每一步都是一个备忘录压栈。这是它最直接的应用。
- 游戏存档：存档文件就是游戏世界状态的备忘录，读档即 `Restore`。
- Redux DevTools 的“时间旅行调试”：把每一次 state 变化都存成快照，可以在历史状态间来回跳——备忘录模式的现代明星应用。
- 数据库/虚拟机快照：数据库的一致性快照、VM/容器的 snapshot、文件系统的快照(如 ZFS)，都是“存下某时刻完整状态以便回滚”的备忘录思想在系统层面的体现。Git 的每次 commit 某种意义上也是一份可 `checkout` 回去的状态快照。

备忘录管“状态存档”。行为型只剩最后两位，也是公认最烧脑的两个。下一篇访问者：当元素类型稳定、却要频繁给它们新增各种操作时，怎么把“操作”从元素类里彻底抽出去——代价是理解那个绕人的“双重分派”。

22 访问者 · Visitor（行为型）

公认最烧脑的模式，没有之一。它的机制(双重分派)绕，适用场景也窄。但理解了它，你对“面向对象的边界在哪”会有全新的认识。

一、这是什么：给一批货物做各种检查

海关有一批货物，类型是固定的：液体、固体、危险品。但要对它们做的操作层出不穷：今天称重，明天估税，后天做安检，大后天出口报关.....

注意这里的不对称：货物类型很稳定(就那几种)，但要施加的操作源源不断地新增。 如果每来一种新操作，就往每个货物类里加一个方法(液体加个 `估税()`、固体加个 `估税()`.....)，货物类会越来越臃肿，而且每加一种操作要改所有货物类。

访问者模式反其道而行：把“操作”抽出来，做成一个个独立的“访问者”。货物类只提供一个“接受访问者”的入口，具体做什么由访问者决定。新增操作 = 新增一个访问者，货物类一行不改。

配套代码是图形：`Circle` / `Rectangle` 是稳定的元素，`AreaVisitor`(算面积)、`DrawVisitor`(渲染)是操作。加“导出 SVG”？写个 `SVGVisitor`，图形类不动。

二、解决什么痛：操作频繁新增，元素类不堪重负

把操作直接塞进元素类，痛在“加操作要改所有元素”：

```
// 痛点:每加一个操作,Circle 和 Rectangle 都得加一个方法
type Circle struct{ /*...*/ }
func (c *Circle) Area() float64 { /*...*/ }
func (c *Circle) Draw()         { /*...*/ }
func (c *Circle) ExportSVG()    { /*...*/ }   // 新操作:两个类都得加
// Rectangle 同理,每个操作都要在每个图形类里重复实现一遍
```

更糟的是，这些操作可能职责各异(渲染是 UI 的事、估税是财务的事、报关是物流的事)，全塞进图形类，让图形类变成一个什么都懂的“万金油”，严重违反单一职责。

> 访问者隔离的“会变的东西”，是“施加在一组稳定元素上的操作”。它把“操作”从元素类里彻底搬出去，让每个操作成为一个独立的访问者类。元素只保留一个“接受访问”的钩子，把“具体做什么”完全委托出去。于是加操作 = 加一个访问者(集中在一个类里，还能把同类操作的逻辑聚在一处)，元素类保持稳定和干净。

三、怎么实现：双重分派

访问者的核心机制叫双重分派(double dispatch)，这是它最绕、也最精髓的地方。慢慢看：

go/visitor/main.go（核心）

```
// 元素:提供"接受访问者"的入口
type Shape interface{ Accept(visitor ShapeVisitor) }

// 访问者:为每种具体元素声明一个 Visit 方法
type ShapeVisitor interface {
    VisitCircle(c *Circle)
    VisitRectangle(r *Rectangle)
}

// 具体元素:Accept 里回调访问者中"对应自己类型"的那个方法
func (c *Circle) Accept(v ShapeVisitor)    { v.VisitCircle(c) }      // ⟵ 我是圆
func (r *Rectangle) Accept(v ShapeVisitor) { v.VisitRectangle(r) }  // ⟵ 我是矩

// 具体访问者:实现对每种元素的操作
type AreaVisitor struct{ TotalArea float64 }
func (v *AreaVisitor) VisitCircle(c *Circle)       { v.TotalArea += math.Pi*c.Ra
func (v *AreaVisitor) VisitRectangle(r *Rectangle) { v.TotalArea += r.Width*r.He
```

最终调用哪个方法，由两个类型共同决定——这就是“双重”分派：

1. 第一次分派：`shape.Accept(v)` —— 由 `shape` 的具体类型(圆？矩形？)决定进哪个 `Accept`。
2. 第二次分派：`Accept` 内部 `v.VisitCircle(c)` —— 由 `v` 的具体类型(算面积？渲染？)决定进哪个 `Visit`。

两次一叠加，`(圆，算面积)`、`(矩形，渲染)`……每个组合精确命中对应的方法。用起来：

```
for _, s := range shapes { s.Accept(&AreaVisitor{}) }   // 对所有图形算面积
for _, s := range shapes { s.Accept(&DrawVisitor{}) }   // 换个访问者,全体渲染
```

> ⚠ 为什么要这么绕？因为 Go/Java 这类语言只有单分派(方法调用只根据接收者一个类型动态决定)，没有原生的“根据两个对象的类型选方法”。`Accept` 回调这一手，正是用两次单分派手工模拟出双重分派。理解这一点，你就理解了访问者存在的根本理由——它是在给语言“打补丁”。

四、有哪些坑

- 著名短板：加“操作”易如反掌，加“元素类型”是灾难。 这是访问者最该记住的取舍，和抽象工厂的维度问题异曲同工：
  - 加一个操作(新访问者)：写个新类实现 `ShapeVisitor` 即可，元素类不动。爽。
  - 加一个元素类型(比如新增 `Triangle`)：你得给 `ShapeVisitor` 接口加一个 `VisitTriangle`，然后每一个已有访问者都得实现它。噩梦。

所以判据很硬：只有当元素类型高度稳定、而操作频繁增删时，才用访问者。 反过来(类型常变、操作稳定)，用访问者就是自找苦吃——那种情况下，把方法放进元素类反而对。

- 破坏元素封装。 访问者要干活，往往需要访问元素的内部数据，这就迫使元素把内部状态暴露给访问者(上面 `c.Radius` 得是可读的)。封装性和“操作外置”在这里有天然的张力。
- Go 的小别扭：没有方法重载。 Java 里可以都叫 `visit(Circle)`、`visit(Rectangle)` 靠重载区分，Go 不行，只能起不同名字 `VisitCircle` / `VisitRectangle`。元素类型一多，访问者接口会很长。
- 它很重，别轻易用。 访问者引入的间接和样板不少。除非你确实撞上了“稳定类型 + 爆炸式增长的操作”，否则优先考虑更简单的手段(比如给类型加方法、或用类型 switch)。

五、开源里怎么用

- 编译器 / AST 遍历 —— 访问者的绝对主场。 Go 的 `go/ast` 提供 `ast.Walk(visitor, node)` 和 `ast.Visitor` 接口，让你对语法树的各类节点施加操作：类型检查、代码生成、格式化、静态分析.....每种操作是一个访问者，而 AST 节点类型高度稳定——完美命中访问者的适用条件。你用过的 `gofmt`、`go vet`、各种 linter，底层都在用它。
- Java 注解处理 / `javax.lang.model.element.ElementVisitor`：编译期处理注解、遍历程序元素，标准库直接给了访问者接口。
- Protobuf / Thrift 编译器：遍历 IDL 定义的 AST，为不同目标语言生成代码——每个语言的生成器就是一个访问者。
- 静态分析 / 代码检查工具：ESLint、各类 linter 遍历 AST 施加检查规则，规则即访问者。凡是“稳定的树结构 + 花样繁多的处理操作”，十有八九能看到访问者。

访问者把“操作”从“元素”里剥离。行为型最后一位，也是 GoF 里最冷门的一个。下一篇解释器：怎么给一个简单的小语法(算术表达式、查询条件)定义规则并求值——以及为什么现实中你几乎永远不该手写它。

23 解释器 • Interpreter（行为型）

终章。我们用 GoF 里最冷门、争议也最实在的一个模式收尾——说实话，它是 23 个里你最不可能亲手实现的那个。但理解它，能让你看懂“语言是怎么被执行的”。

一、这是什么：给计算器造一个“小语言”

你要做个计算器，能算 `5 + 3 - 2`。计算机怎么“看懂”这串字符？它得：先把式子拆成一棵树(`5+3` 是一个加法节点，再和 `2` 组成减法节点)，然后从叶子往上递归求值。

解释器模式就是干这个的：给一个简单的语言定义它的文法(每条语法规则对应一个类)，把一个句子表示成一棵抽象语法树(AST)，然后递归地“解释”(求值)这棵树。

配套代码正是这个算术解释器：数字、变量是“终结符”节点，加法、减法是“非终结符”节点，`Parse` 把 `"x + y - 3"` 变成一棵表达式树，`Interpret(ctx)` 递归求值。

二、解决什么痛：一种小语法要反复解析求值

有时你会遇到一种领域特定的小语言，需要反复解析和执行它：

- 计算器的算术表达式。
- 规则引擎的条件：`age > 18 && vip == true`。
- 简单的查询过滤：`status = "active" AND score > 60`。

如果每次都用一堆字符串切割和 `if-else` 硬解析，代码会又臭又长、加一条语法规则就崩。解释器的思路：为文法里的每一条规则定义一个类，让“解析”变成“构造一棵由这些规则类组成的树”，“执行”变成“对树递归调用 `Interpret`”。

> 解释器隔离的“会变的东西”，是“这门小语言的每一条文法规则”。 每条规则(数字、变量、加法、减法.....)封装成一个独立的表达式类，组合成树。加一种运算(比如乘法)= 加一个表达式类。语法的结构被显式地映射成了类的结构。

三、怎么实现：终结符 + 非终结符，递归求值

文法节点分两种：终结符(叶子，不再包含子表达式，如数字、变量)和非终结符(内部节点，由子表达式组合而成，如加法、减法)。所有节点实现同一个 `Interpret` 接口。

go/interpreter/main.go（核心）

```
// 抽象表达式:所有节点都能在上下文中求值
type Expression interface{ Interpret(ctx *Context) int }

// 终结符:数字(叶子,直接返回值)
type NumberExpression struct{ value int }
func (n *NumberExpression) Interpret(ctx *Context) int { return n.value }

// 终结符:变量(叶子,从上下文查值)
type VariableExpression struct{ name string }
func (v *VariableExpression) Interpret(ctx *Context) int { return ctx.GetVar(v.

// 非终结符:加法(内部节点,递归求值左右子表达式再相加)
type AddExpression struct{ left, right Expression }
func (a *AddExpression) Interpret(ctx *Context) int {
    return a.left.Interpret(ctx) + a.right.Interpret(ctx)    // ⟵ 递归
}
```

`Parse` 把中缀表达式串成一棵树，`Interpret` 递归求值，还能带变量上下文：

```
ctx := NewContext()
ctx.SetVar("x", 10); ctx.SetVar("y", 4)
ast, _ := Parse("x + y - 3")
ast.Interpret(ctx)                  // = 11
```

你会发现，这棵表达式树本质就是一棵组合模式的树，而“求值”就是对它的递归遍历——解释器 = 组合模式(AST 结构) + 递归解释。

四、有哪些坑：为什么说它“最鸡肋”

必须实话实说，这是全系列唯一一个我要劝你基本别用的模式：

> ⚠ 解释器只适合“极其简单、且稳定”的小语法。 一旦语法稍微复杂一点(要支持运算符优先级、括号、函数调用、错误恢复.....)，你手写的表达式类和解析逻辑就会爆炸式膨胀，难写、难调、性能还差。GoF 自己都在书里提醒：文法复杂时，别用这个模式。

其它要点：

- 现实中的正确做法，几乎都不是手写解释器。 语法稍复杂，业界用的是解析器生成器(yacc/bison、ANTLR、Go 的 `goyacc`)或成熟的表达式求值库(如 `expr`、`govaluate`)。它们帮你处理优先级、错误、优化等一大堆脏活，比手搓 GoF 解释器专业得多。
- 它和访问者是天生一对。 求值逻辑如果直接写进每个表达式类的 `Interpret` 里，类一多就乱。实践中常把“解释/求值”用访问者从节点类里抽出去——AST 节点只管表示结构，一个 `EvalVisitor` 负责遍历求值，一个 `PrintVisitor` 负责打印，等等。真正的编译器就是这么干的。
- 性能。 树遍历式的解释，每次求值都递归走一遍树，比编译成字节码/机器码慢得多。反复高频执行同一表达式时，考虑编译缓存。

五、开源里怎么用

- `go/parser` + `go/ast`:Go 工具链把源码解析成 AST(解释器模式的“表示”部分)，再由各种工具遍历处理。虽然 Go 编译器最终是编译而非解释，但“源码→AST→遍历”这套结构，正是解释器模式的骨架。
- 模板引擎：Go 的 `text/template` / `html/template`、各类模板语言，解析模板成语法树后求值渲染——解释器的实用形态。
- 表达式/规则引擎：`expr`、`govaluate` 等库让你在运行时求值 `user.age > 18` 这样的表达式，背后是解释器思想(但实现远比 GoF 版精良)。规则引擎、特征开关、告警条件求值都靠它。
- 正则表达式引擎、SQL 解析器、配置语言(HCL)：凡是“把一段文本按文法解析成结构再执行”的地方，都有解释器的影子——但再强调一次，它们几乎无一是手写 GoF 解释器，而是用专业的解析技术。

二十三式，到此收官

恭喜你走到了终点。回望这趟旅程，如果只让你带走三句话：

1. 每个模式都在隔离某种“变化”。 忘掉类图，记住那句纲领——把会变的东西封装隔离起来。策略隔离算法、观察者隔离通知对象、状态隔离状态行为……抓住“它在隔离什么变化”，23 个模式就连成了一张网，而不是 23 条孤立口诀。
2. 模式是解药，不是维生素。 见到痛点才用，别为用而用。本系列每一篇的“坑”都在反复劝一件事：过度设计比不设计更糟。菜鸟堆模式炫技，老手能不用就不用。
3. 模式的流行度，反映语言的短板。 你一路都看到了：单例在 Go 里几乎不算模式(`sync.Once` 内建)、策略退化成一个函数值、迭代器变成 `for range`、建造者在有具名参数的语言里被消解……语言越强，越多模式会“隐形”。 所以学模式，别学成教条——学的是背后那个反复出现的痛点，和止痛的思路。招式会因语言而变，内功不会。

配套的 9 语言完整代码都在 👉 github.com/davidapp/GoF，挑你顺手的语言跑一遍、改一改，比读十遍都管用。

祝你在真实的代码里，痛的时候想起它们，不痛的时候放过它们。

道雾轩

《经典设计模式浅析》