设计模式学习笔记

Reference

- *https://refactoringguru.cn/design-patterns/catalog*
- *大话设计模式*
- *深入设计模式*

一、设计模式主要原则

1）单一职责原则：

一个类负责实现一个功能

2）迪拉米特法则：

如果两个类不必彼此直接通信，那么这两个类就不应该发生直接的相互作用。如果出现调用情况，使用第三方转发调用。

- 每一个类尽量降低外部成员的访问权限

3）开放-封闭原则：

面对需求，对程序的改动时通过增加新代码进行，而不是更改现有的代码。

- 开放-封闭中的假设原则
  - 假设变化不会发生
  - 当变化发生时，创建抽象隔离同类变化（与里氏代换的关系）

4）依赖倒转原则：

- 高层模块不应该依赖底层模块，两者都应依赖抽象
- 抽象不应该依赖细节，细节应该依赖抽象

5）里氏代换原则：

- 抽象声明接口，为子类替换父类提供可能性。同时由于依赖抽象，满足了依赖倒转原则，解开了高层和底层模块间的耦合。

6）合成/聚合复用原则：

拆分类，构建类间的合成和聚合关系，避免无限继承导致规模不可控的继承树

- 聚合关系：大雁和雁群（弱拥有关系）
- 合成关系：大雁和翅膀（严格的部分和整体的强拥有关系）

二、类间关系展示

1）合成（组合）关系：

- 实心菱形

2）聚合关系：

- 空心菱形

3）关联关系（季节与蝉）

- 实线+箭头 ->

4）继承关系：

- 虚线+空心箭头

5）依赖关系：

- 虚线+普通箭头

6）接口声明：

- 矩形表示法《interface》

二、创建型模式

01-工厂方法模式

1）模式定义：

创建型设计模式，在父类提供一个创建对象的方法，允许子类决定实例化对象的类型。

2）模式结构（主要成员）

2.1）产品（Product）

2.2）具体产品（Concrete Product）

2.3）创建者（Creator）

2.4）具体创建者（Concrete Creator）重写基础工厂方法

3）适用场景

3.1）编写代码时，无法预知对象确切类别与·依赖关系

3.2）希望用户能够拓展软件库或内部组件

3.3）复用现有对象节省资源（处理大型资源密集型对象，数据库连接，文件系统和网络资源时候。复用现有对象，同时创建新对象）

4）优缺点

- 优点
  - 避免创建者和具体产品之间的耦合性
  - 单一职责原则：单独维护产品的创建代码
  - 开闭原则：通过新增代码引入新产品类型
- 缺点
  - 引入许多新的子类与子类选择的条件判断代码（增加了复杂性）

5）与其他模式的关系

5.1）开发初期会使用工厂方法模式，开发简单灵活，随后演化适用抽象工厂模式、原型模式或生成器模式

5.2）抽象工厂模式一般给予一组工厂方法or原型方法

5.3）同时使用工厂方法和迭代器模式

5.4）工厂方法是模板方法模式的一种特殊形式

02-抽象工厂模式

1）模式定义（创建型设计模式）：

无需制定其具体类，就能创建一些相关对象。抽象工厂提供一个接口用于创建每个系列产品的对象。

模式来源：

不同数据库之间的切换，利用抽象工厂模式抽象出数据库

2）模式结构（主要成员）

2.1）抽象产品（Abstract Product）：声明产品的创建接口

2.2）具体产品（Concrete Product）：实现产品的创建接口

2.3）抽象工厂（Abstract Factory）：声明抽象接口

2.4）具体工厂（Concrete Factory）：创建特定产品变体

3）适用场景

3.1）需要与多个不同系列的相关产品交互，由于无法提前获取相关信息，出于对未来拓展性的考虑。不希望代码给予产品的具体类进行构建。

3.2）如果本身已经有一组抽象方法类，由于新需求导致功能不明确，可以使用抽象工厂模式。

4）优缺点

- 优点
  - 具体工厂与特定产品一对一绑定，确保产品类型一定能被生产出来
  - 避免客户端和具体产品类耦合
  - 单一职责原则：将产品生成的代码抽取到同一位置，易于维护

    - 开闭原则：引入新产品变体时，无需修改客户端代码
- 缺点
    - 一旦需要新增产品，需要新增与产品对应的工厂
    - 此模式需要向应用中引入众多接口和类（增加了复杂性）

5）与其他模式的关系

5.1）与生成器模式的区别

- 生成器模式关注分步生成复杂对象，且允许在获取产品前补充额外步骤。
- 抽象工厂专门用于生产一些列相关对象。

5.2）使用简单工厂优化抽象工厂模式

- 利用反射+配置文件替换抽象工厂模式，消除使用条件逻辑进行产品选择的操作。

5.3）抽象工厂模式，通过提供一组工厂方法，可以使用【原型模式】替换

5.4）搭配桥接模式，由桥接模式定义的抽象只能与特定工厂实现合作。抽象工厂对特定关系进行封装

5.5）抽象工厂、生成器、原型都可以通过单例模式来实现

03-生成器模式

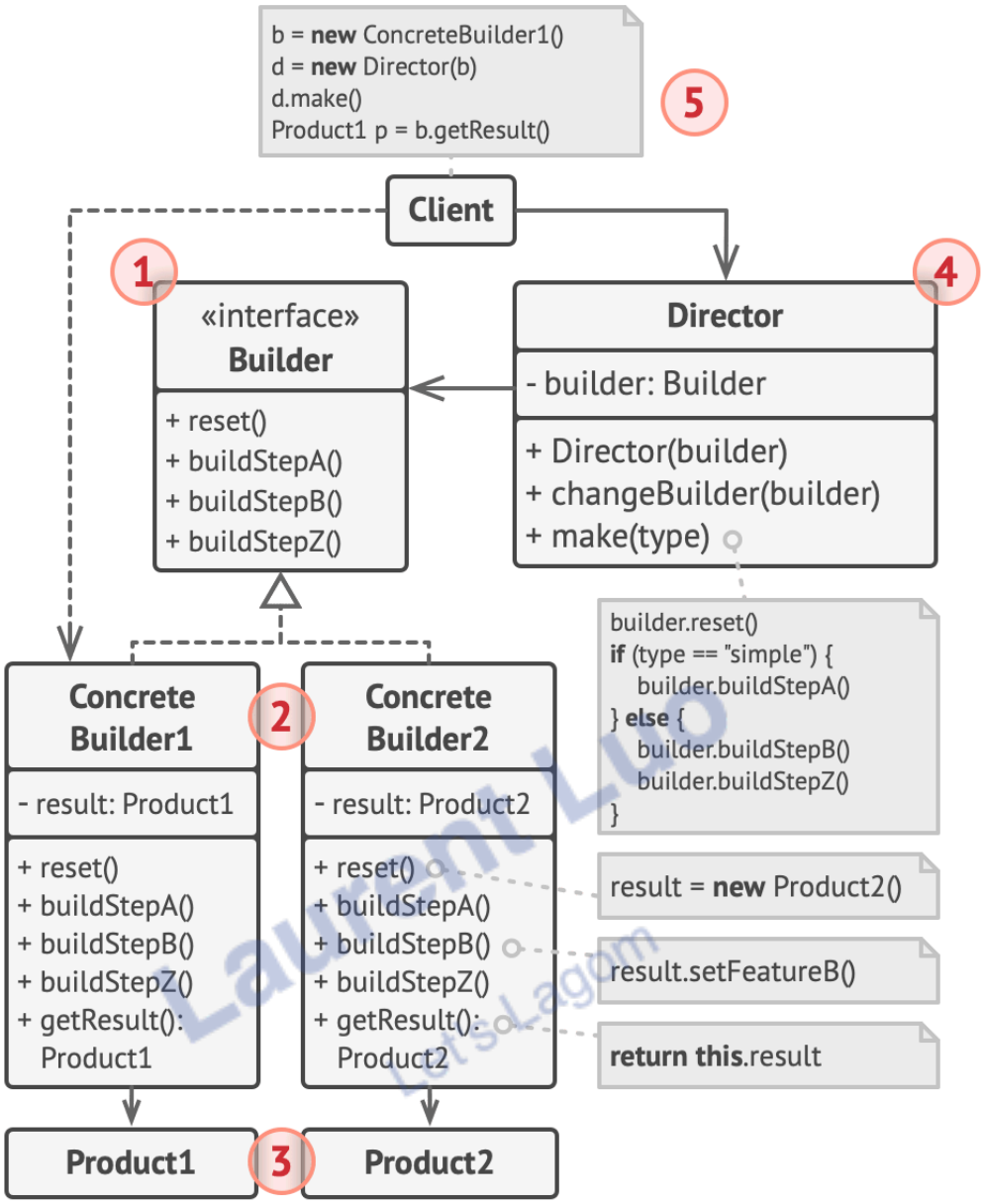
1）模式定义（创建型设计模式）：

定义分步骤实现复杂对象的模块创建流程。 该模式允许你使用相同的创建代码生成不同类型和形式的对象。

模式来源：

面对一个复杂对象（房子），构造时需要对多个成员变量和嵌套对象进行繁复的初始化工作。

2）模式结构（主要成员）



2.1）生成器（Builder）：接口声明在所有类型生成器中通用的产品构造步骤

2.2）具体生成器（Concrete Builders）提供构造过程的不同实现。具体生成器也可以构造不遵循通用接口的产品

2.3）产品（Products）：最终产品

2.4）主管（Directors）：定义调用构造步骤的顺序

2.5）客户端（Client）与某个生成器对象进行关联。通过主管类构造函数的参数进行一次性关联。

3）适用场景

3.1）避免构造不同参数的类构造函数

3.2）使用同一套流程创建不同形式的产品

3.3）构造组合树或其他复杂对象

4）优缺点

- 优点
  - 分步、递归运行创建步骤
  - 生成不同形式的产品时，可以复用相同的制造代码
  - 单一职责原则：将复杂对象的内部对象的构造代码从产品业务逻辑分离
- 缺点
  - 此模式回新增多个类，导致整体复杂度的增加

5）与其他模式的关系

5.1）初期会使用工厂方法模式，后续随着业务层级复杂度的增加，演化为抽象工厂模式、原型模式和生成器模式

5.2）重点关注如何分步生成复杂对象，相比抽象工厂对创建产品的模板性，生成器模式能够在此基础上执行额外的构造步骤

5.3）配合组合模式配合构造复杂组合模式树，使得构造步骤以递归方式运行

5.4）配合桥接模式，主管类负责抽象，不同生成器负责实现

5.5）抽象工厂、生成器和原型模式都可以使用单例模式实现。

04-原型模式

1）模式定义（创建型设计模式）：

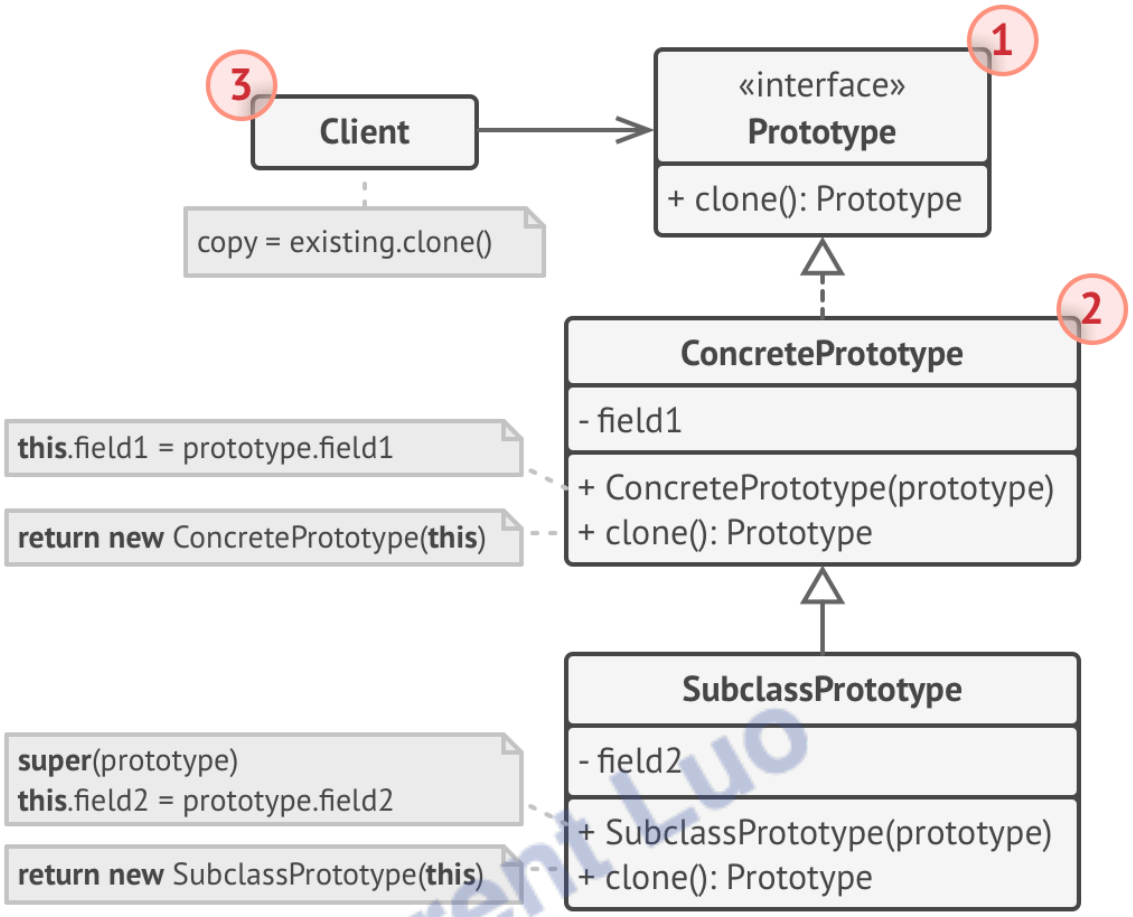
复制已有对象，又无需使代码依赖他们所属的类。

模式来源：

复制一个复杂对象，暴力遍历所有成员和属性，并将成员变量复制到新对象中是不可行的。因为对象中部分组件类实例属性可能是私有的。且部分组件类的接口是抽象类声明的，所以无法知道是什么具体类。

- python中的深拷贝、浅拷贝（`__copy__\__deepcopy__`），对于对象中的可变对象进行深拷贝。

2）模式结构（主要成员）



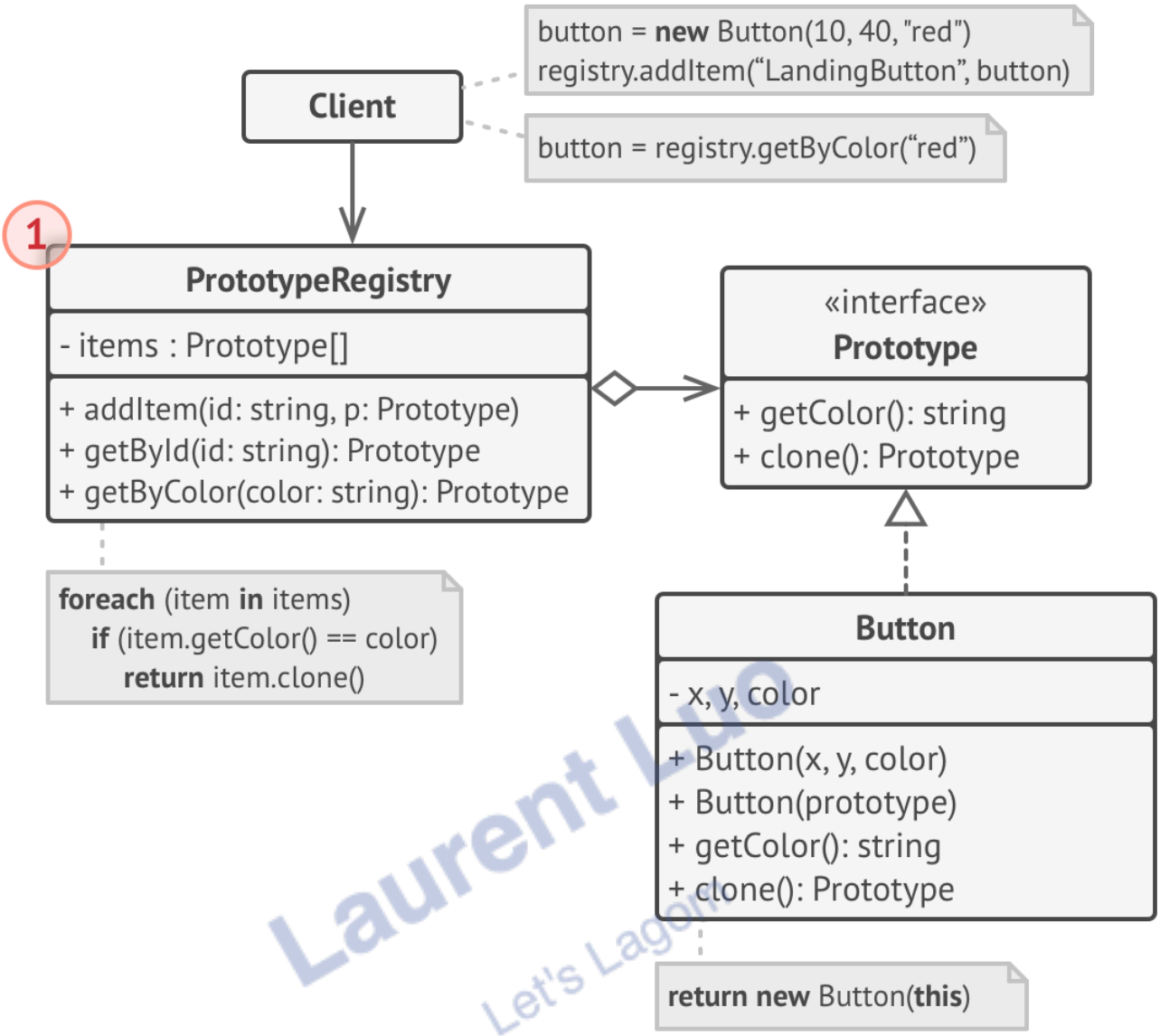
2.1）原型（Prototype）：声明克隆方法接口。 其中只会有一个名为 `clone` 克隆的方法

2.2）具体原型（Concrete Builders）类将实现克隆方法。

- 将原始对象的数据复制到克隆体中之外，
- 还需处理克隆过程中的极端情况， 例如克隆关联对象和梳理递归依赖等等

2.3）客户端（Client）可以复制实现了原型接口的任何对象

2.4）原型注册表：提供了访问常用原型的简单方法，存储了一系列可供随时复制的预生成对象。



3）适用场景

3.1）需要复制一些对象，同时代码独立于这些对象所属的具体类。

3.2）如果子类的区别仅在对象初始化方式，可以使用原型模式

3.3）构造组合树或其他复杂对象

4）优缺点

- 优点
  - 克隆对象但无需与他们所属的具体类耦合
  - 克隆预生成原型，避免反复运行初始化代码
  - 便于生成复杂对象
  - 可以用继承之外的方式处理复杂对象的不同配置
- 缺点

- 克隆模式包含循环引用的复杂对象的实现难度较高

5）与其他模式的关系

5.1）初期会使用工厂方法模式，后续随着业务层级复杂度的增加，演化为抽象工厂模式、原型模式和生成器模式

5.2）抽象工厂模式通常基于一组工厂方法，可以使用原型模式生成这些类的方法

5.3）用于保存命令模式的历史记录

5.4）配合组合模式与装饰模式，原型模式承担复杂结构的复制，避免从重新构造（接口）

5.5）原型不基于即成，工厂方法基于即成，但不需要初始化步骤

5.6）可以作为备忘录模式的简化版本。

5.7）抽象工厂、生成器、原型可以用单例模式实现

05-单例模式

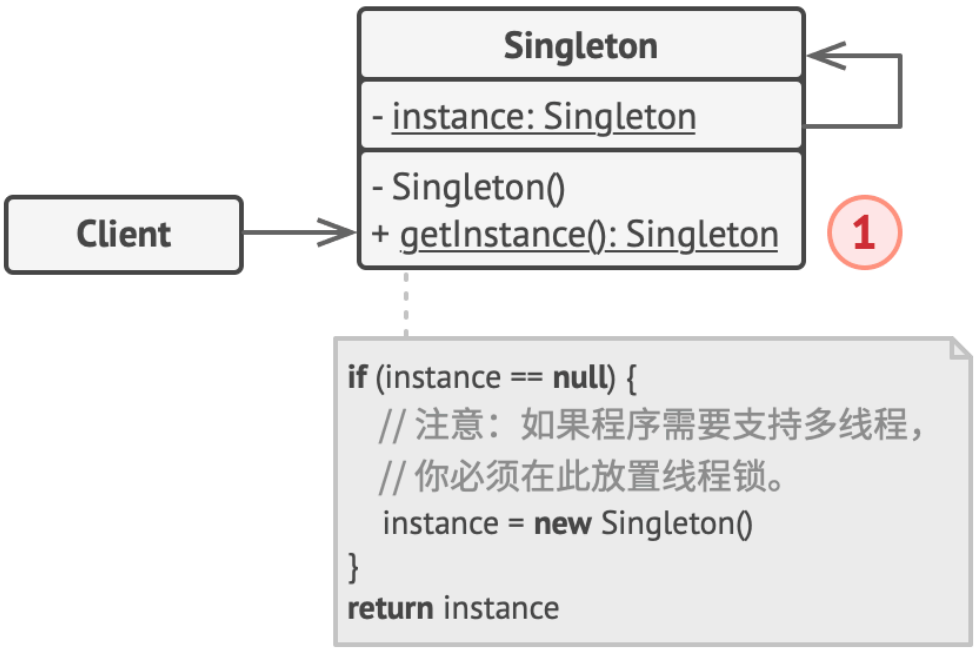
1）模式定义：

创建型设计模式，类自身封装唯一实例，保证一个类只有一个实例。类提供单一实例的受控访问（构造函数私有化）

模式来源

保证一个类只有一个实例，控制共享资源的访问权限

2）模式结构（主要成员）



2.1）单例（Singleton）类

- 构造函数必须对客户端隐藏（私有），调用获取实例方法是获取单例对象的唯一方法
- 声明了一个名为getInstance 获取实例的静态方法来返回其所属类的一个相同实例；
- 多线程模式下，构造函数创建实例时，使用双重锁定机制，避免多线程生成多个实例。

3）适用场景

3.1）如果程序中的某个类对所有的客户端只有一个可用的实例

3.2）需要严格控制全局变量的情况

4）优缺点

- 优点
  - 保证一个类只有一个实例
  - 获得了一个指向该实例的全局访问节点
  - 仅在首次请求单例时对其进行初始化
- 缺点
  - 违反了单一职责原则
  - 单例模式导致程序组件之间相互了解过多
  - 在多线程环境下需要设置双重锁机制
  - 单元测试可能比较困难，因为测试框架是以基于即成方式创建模拟对象

5）与其他模式的关系

5.1）外观模式通常可以转换为单例模式，大部分情况下只需要一个外观对象

5.2）当对象所有共享状态简化为一个享元对象时，享元模式近似于单例

- 但是享元类可以有多个实体，各实体内在状态可以不同
- 单例对象是可变的，享元对象是不可变的

5.3）抽象工厂、原型、生成器都可以用单例实现

二、结构型模式

06-适配器模式

1）模式定义：

结构型设计模式，适配原本不兼容接口。

模式来源

- .NET框架的应用
- 现有系统模块接口和客户期待接口不同，需要添加适配器进行转换。

2）模式结构（主要成员）

2.1）客户端（Client）包含当前程序业务逻辑的类

2.2）客户端接口（Client Interface）描述了其他类于客户端代码合作时必须遵守的协议

2.3）服务（Service/Adaptee）不兼容的功能接口

2.4）适配器（Adapter）同时与客户端和服务交互的类

3）适用场景

3.1）接口不兼容

3.2）希望复用同一个继承体系下不同类的特有接口

4）优缺点

- 优点
  - 单一职责原则：分离数据转换代码于程序主要逻辑代码
  - 开闭原则：通过新增代码引入新产品类型
- 缺点
  - 增加适配器模块，导致系统引入额外复杂度（因此使用前先评估，是否可以通过重构接口实现需求）

5）与其他模式的关系

5.1）开发设计前期尽量通过【桥接模式】，对程序模块化开发。在重构接口与适配器模式之间做取舍。

5.2）与装饰模式的区别

- 适配器对已有对象的接口进行修改。
- 装饰模式能在不改变对象接口的前提下强化对象功能。

5.3）与代理模式区别

- 适配器为被封装对象提供不同接口。
- 代理模式则是为不同对象提供相同接口，装饰器模式则是为对象提供加强的接口。

5.4）外观模式区别

- 外观模式为现有对象定义了一个新接口。
- 适配器则尝试运用现有接口。而且适配器通常只封装一个对象。但是外观模式通常会作用于整个对象的子系统上

5.5）桥接模式、状态模式、策略模式（包含某种程度的适配器）的接口处理类似。本质上这几个模式是基于组合模式（将工作委派给其他对象）

07-桥接模式

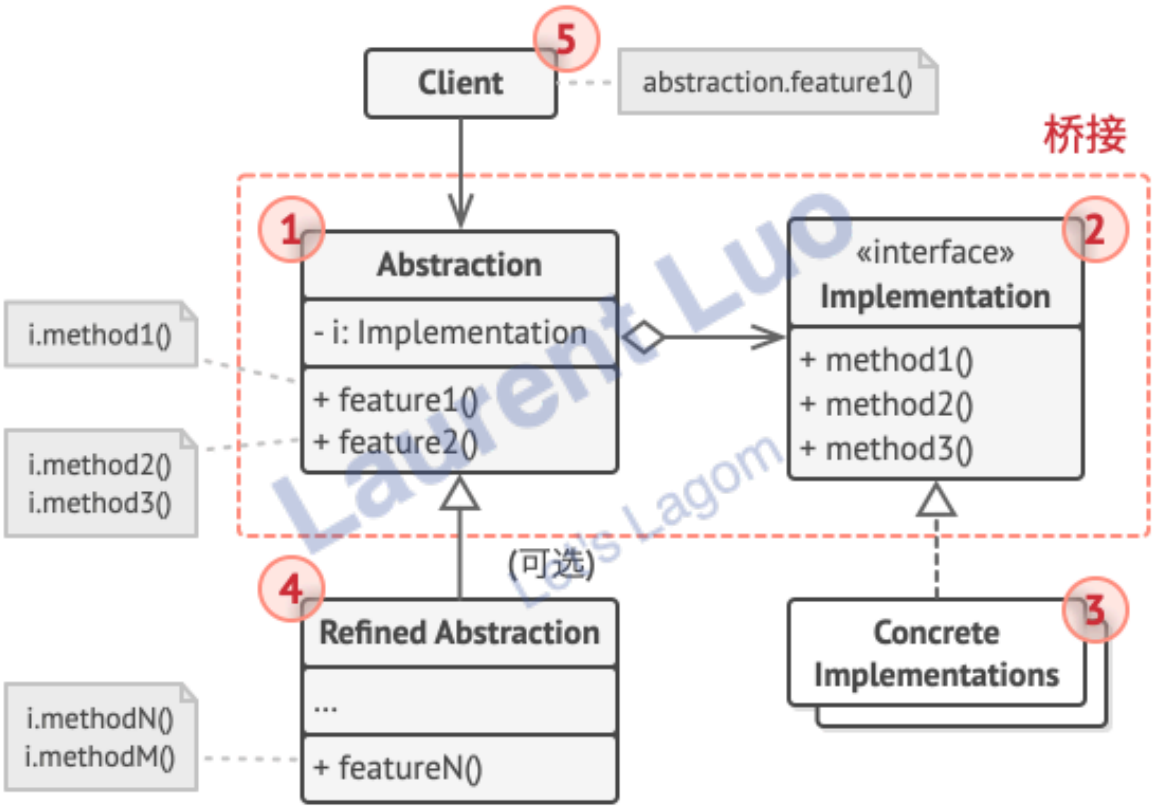
1）模式定义：

解决单类继承导致的子类数量不可控的情况，通过桥接模式替换强耦合的继承结构。将一个大类或一些列紧密相关的类拆分为抽象和实现两个独立的层次结构。

模式来源

处理跨平台应用、支持多种类型数据库服务器或多个特定种类。同一个GUI能够在Windows 和linux下运行，且无需改动API相关的类就可以修改GUI类

2）模式结构（主要成员）



2.1）抽象部分（Abstraction）

- 声明绑定具体实现类的方法
- 调用抽象实现类的通用接口

提供高层控制逻辑，依赖于完成底层实际工作的实现对象

2.2）实现部分（Implementation）：

- 为所有具体类实现声明通用接口。
- 抽象部分仅能通过在这里声明的方法实现对象交互

2.3）具体实现（Concrete Implementation）实现通用接口的具体业务逻辑

2.4）精确抽象（Refined Abstraction）提供控制逻辑的变体

2.5）客户端：将特定类型的抽象对象与特定类型的实现对象连接

3）适用场景

3.1）替换庞大的继承类结构

3.2）拆分或充足一个具有多重功能的庞杂类（能与多个数据库服务器进行交互的类）

3.3）多个独立维度扩展一个类，可以使用此模式

3.4）需要在运行时切换不同的实现方法，使用桥接模式

4）优缺点

- 优点
  - 创建与平台无关的类和程序
  - 客户端代码仅与高层抽象部分进行互动
  - 开闭原则：抽象部分和实现部分之间的新增不会相互影响
  - 单一职责原则：抽象部分专注于处理高层逻辑，实现部分处理平台细节
- 缺点
  - 对高内聚的类使用此模式可能会增加代码的复杂度

5）与其他模式的关系

5.1）常在开发前期使用，保证各个程序个部分独立，而适配器通常在已有程序中使用，让相互不兼容的类进行合作

5.2）桥接模式、状态模式、策略模式接口相似，本质都是基于组合模式，将工作委派给其他对象。

5.3）搭配抽象工厂模式，桥接模式定义抽象只和特定实现合作。抽象工厂对搭配关系进行封装。

5.4）搭配生成器模式，主管负责抽象工作，不同生成器负责实现工作

08-组合模式

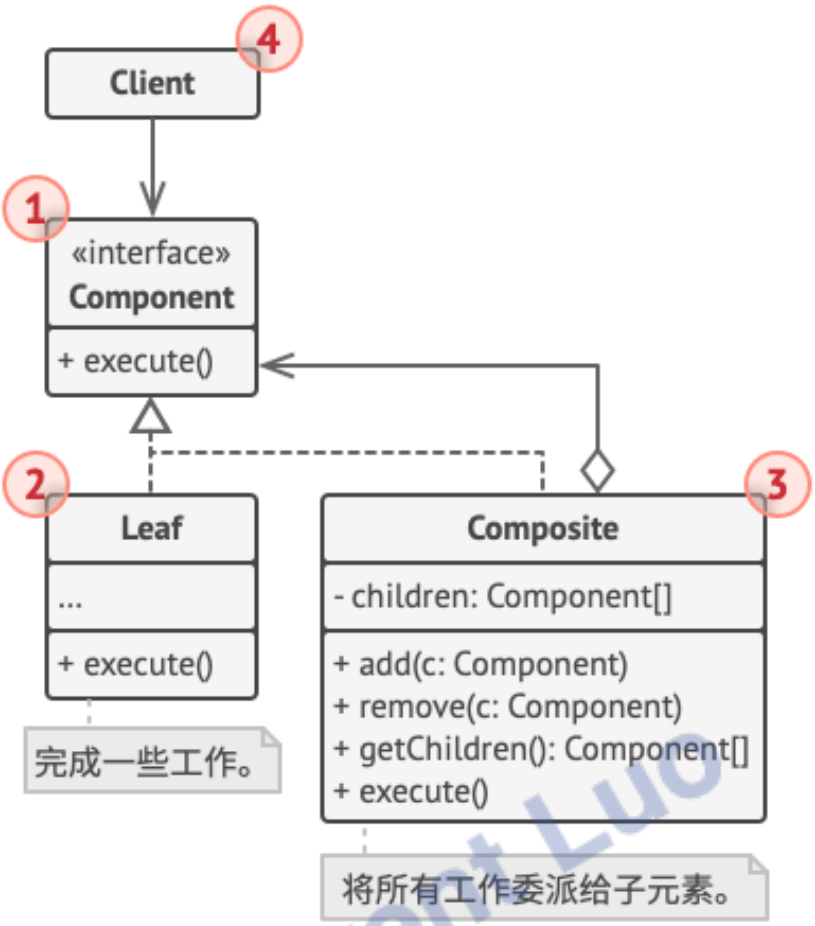
1）模式定义：

将对象组合成树状结构，实现单个对象和组合对象的使用具有一致性

模式来源

模型核心结构需要用树状结构表示，订单中包含各种产品，产品被放在盒子中，盒子被放在更大盒子中，最终打包成一个订单

2）模式结构（主要成员）



2.1）组件（Component）接口描述了简单项目和复杂项目共有的操作

2.2）叶节点（Leaf）树的基本结构，不含子节点

2.3）容器（Container/Composite）枝节点，包含叶节点和其他容器等子项目的单位。接受请求，分配工作给子项目，处理中间结果

2.4）客户端：通过组件接口与所有项目交互，因此可以与组合模式树结构中的枝节点交互

- 叶节点和枝节点的设计可以遵循：透明方式/安全方式

3）适用场景

3.1）需要树状对象结构

3.2）希望客户端代码以相同方式处理简单、复杂元素（基于组合模式所有叶、枝节点的共同接口）

4）优缺点

- 优点
  - 利用多态与递归使用复杂树结构
  - 开闭原则：通过新增枝节点与叶子节点实现需求，不改变原有树结构
- 缺点
  - 对于功能差异较大的类，归纳并定义共同接口困难

5）与其他模式的关系

5.1）组合模式的本质是将工作委派给其他对象，桥接模式、状态模式、策略模式的接口相似，基于组合模式。

5.2）创建负责组合树时，使用生成器模式（生成器模式：定义对复杂对象内部模块的生成顺序）

5.3）搭配职责链模式，叶组件接收到请求后，将请求沿包含全体父组件的链传递到对象树底部（递归）

5.4）组合树的遍历：迭代器模式

5.5）使用访问者模式对整个组合树执行操作

- 具体元素类=叶子/枝节点（具有统一接口处理请求）
- 客户端，请求来源（访问者）

5.6）组合模式和装饰模式区别

- 相同点：结构图很相似，两者都依赖递归则和来组织无限数量的对象。
- 搭配装饰模式，扩展组合树中特定对象的行为（因为基于透明方法设计的组合结构，抽取出了通用业务逻辑接口，因此无法处理非通用请求）

5.7）对组合和装饰设计模式实现的对象，利用原型模式，不需要从零构造

09-装饰模式

基于聚合原则，将对象封装到具有特殊性行为的新对象中（套娃）

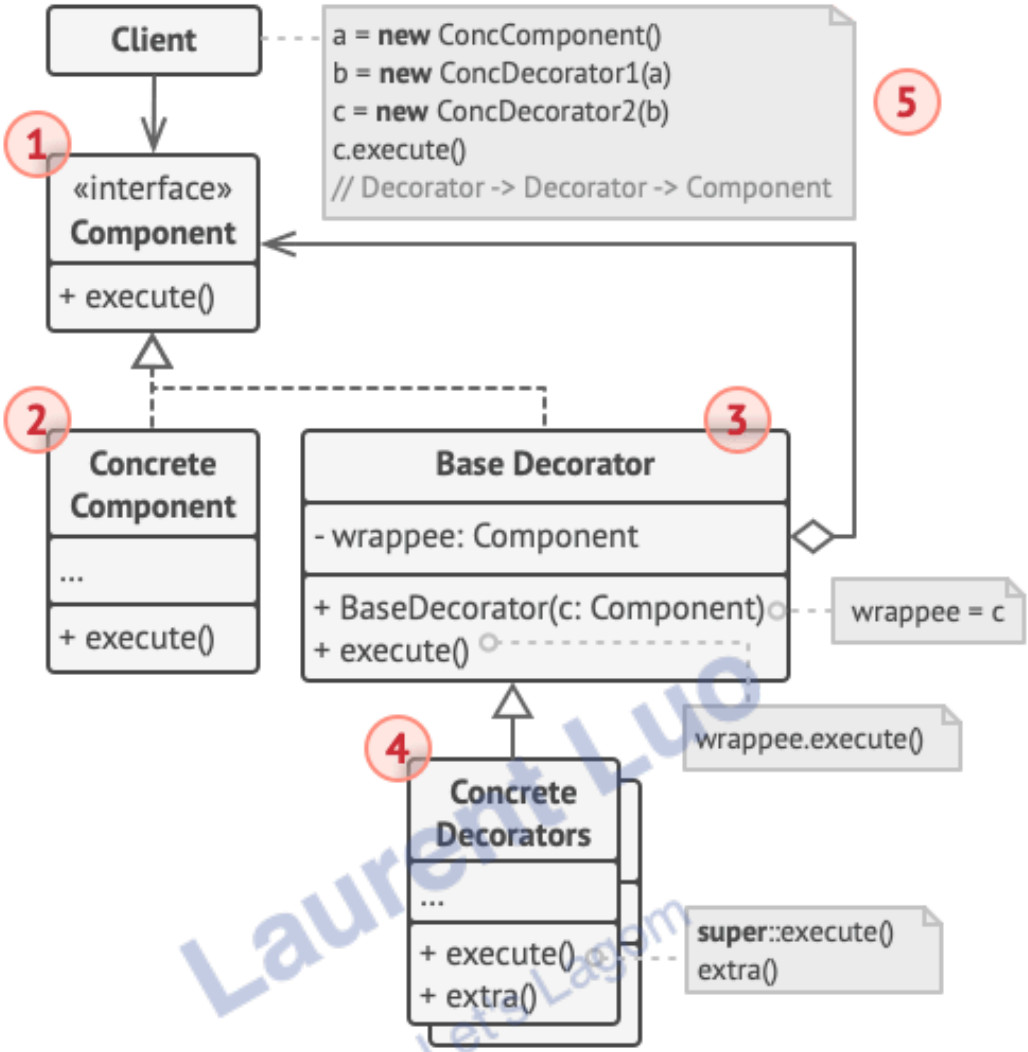
模式来源

需要向不同类型的客户端发送消息，随着客户端类型的增加，需要增加继承对象数量，导致数量上的不可控，管理与维护困难。

1）模式定义：

2）模式结构（主要成员）

- 结构图与组合模式相似（具体元素和具体装饰器/ 叶子节和枝节点）



2.1）部件（Component）声明装饰器和被封装对象的功用接口

2.2）具体部件（Concrete Component）类是被封装对象所属的类。定义了基础行为。

2.3）基础装饰（Base Decorator）

- 拥有一个指向被封装对象的引用成员变量。该变量类型应该被声明为通用部件接口
- 根据引用的具体部件和装饰，基类将操作委派给被封装的对象

2.4）具体装饰类（Concrete Decorator）定义了可动态添加到部件的额外行为。具体装饰类重新装饰基类方法。

2.5）客户端：可以使用多层装饰来封装部件

3）适用场景

3.1）在无需修改代码的情况下使用对象，在运行时为对象新增额外的行为。（基于通用行为的定制化行为）

3.2）当继承拓展方案难以实现/不可行时，尝试使用装饰模式

4）优缺点

- 优点
  - 无需通过继承创建子类就可以拓展对象行为
  - 在运行时添加或删除对象的功能
  - 可以用多个装饰被封装对象，实现行为组合（部件->装饰器1->装饰器2->）
  - 单一职责：将实现了多种不同行为的大类拆分为多个较小的类
- 缺点
  - 在封装器栈中删除特定封装器比较困难
  - 实现行为不受装饰器栈顺序影响的装饰比较难
  - 各层的初始化配置代码可理解性较低

5）与其他模式的关系

5.1）装饰器相比适配器，能在不改变对象接口的前提下强化对象功能，装饰还支持递归组合

5.2）装饰器模式、代理模式、适配器模式区别

- 装饰器为被封装对象提供接口加强
- 代理模式为对象提供相同接口
- 适配器为被封装对象提供不同接口

5.3）责任链模式区别

- 相同点：结构相似，两者都依赖递归将需要执行的操作传递给一系列对象
- 责任链的管理者可以相互独立执行，随时中断。
- 装饰行为无法中断请求传递

5.4）组合模式区别

- 结构图相似，依赖递归组合来组织无限数量的对象（见UML类图与代码实现）

5.5）搭配原型模式

- 对复杂对象进行复制，避免从零开始构造

5.6）与策略模式区别：

- 装饰器模式可以更改对象外表，
- 策略模式则可以改变封装的对象

5.7）与代理模式区别

- 结构相似。
- 服务对象的生命周期管控权限不同：
  - 代理模式自行管理服务对象的生命周期。
  - 装饰器对象的生成由客户端控制。

10-外观模式

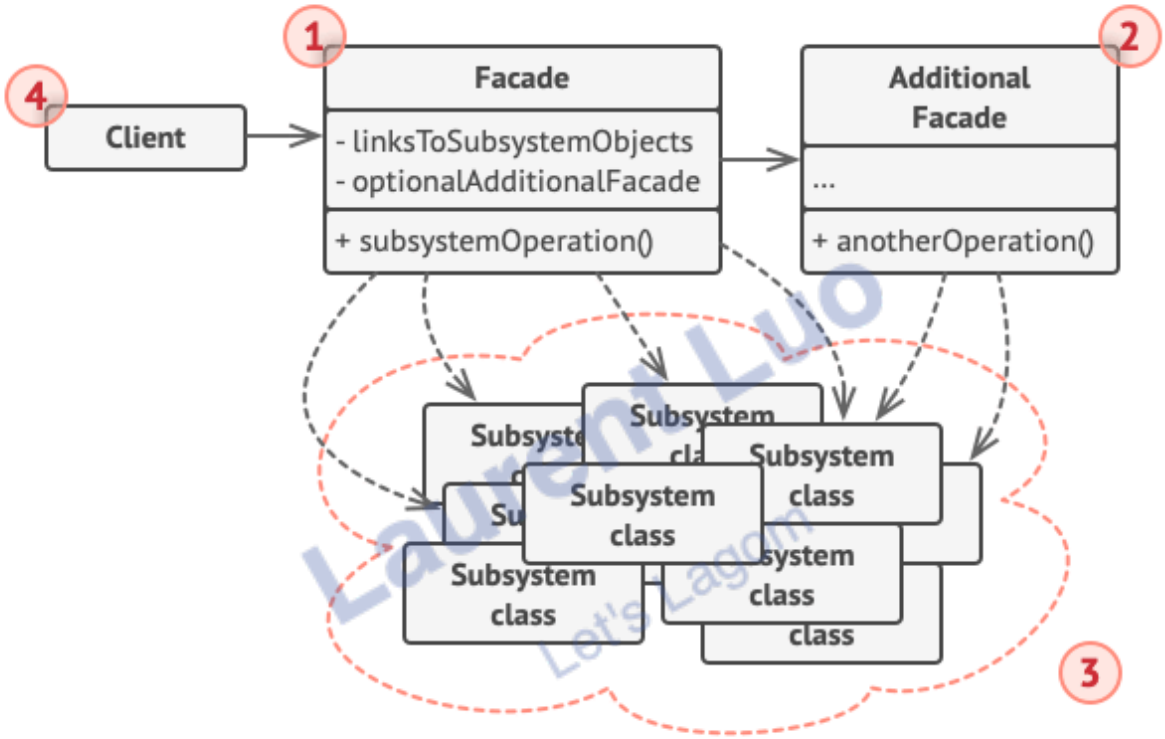
1）模式定义：

基于依赖倒转原则与迪拉米特法则，提供抽象依赖。为程序库、框架或其他复杂类提供一个简单接口

模式来源

程序包含多种功能的复杂库，但业务逻辑只需要使用部分功能

2）模式结构（主要成员）



2.1）外观（Facade）提供访问特定子系统功能接口。**了解如何重定向客户端请求**。

2.2）附加外观（Additional Facade）避免多种不相关的功能污染单一外观

2.3）复杂子系统（Complex Subsystem）由数十个不同对象构成，利用这些对象实现部分业务需求。对于子系统来说只能由外观进行单向通信和调度。

2.4）客户端

3）适用场景

3.1）需要一个指向复杂子系统的直接接口，且需要控制接口访问权限。

3.2）将子系统组织为多层结构：创建外观类抽象子系统各层次的入口。子系统之间也通过外观交互。

3.3）在新系统和旧系统建立外观类

4）优缺点

- 优点
  - 外部封装外观类，避免由于子系统不断重构演化导致调用复杂度的增加
- 缺点
  - 外观类可能成为与程序中所有类都耦合的上帝对象

5）与其他模式的关系

5.1）与适配器模式区别

- 外观模式为现有对象定义了新接口，外观通常会作用于整个对象子系统。
- 适配器模式尝试运用现有接口。适配器只封装一个对象。

5.2）如果子系统只是需要隐藏的构造方法，抽象工厂模式可以替换外观模式

5.3）外观模式和中介者模式，都尝试在紧密耦合的类中组织工作

- 外观模式为子系统所有对象定义了简单接口，但不提供功能，同时子系统不会意识到外观的存在
- 中介者将系统组件之间的沟通行为中心化

5.4）外观类可以转换为单例模式类

5.5）代理模式区别：

- 外观模式暴露给客户的接口可能与子系统的接口不同。
- 代理对象与其服务对象遵循同一接口。

11-享元模式

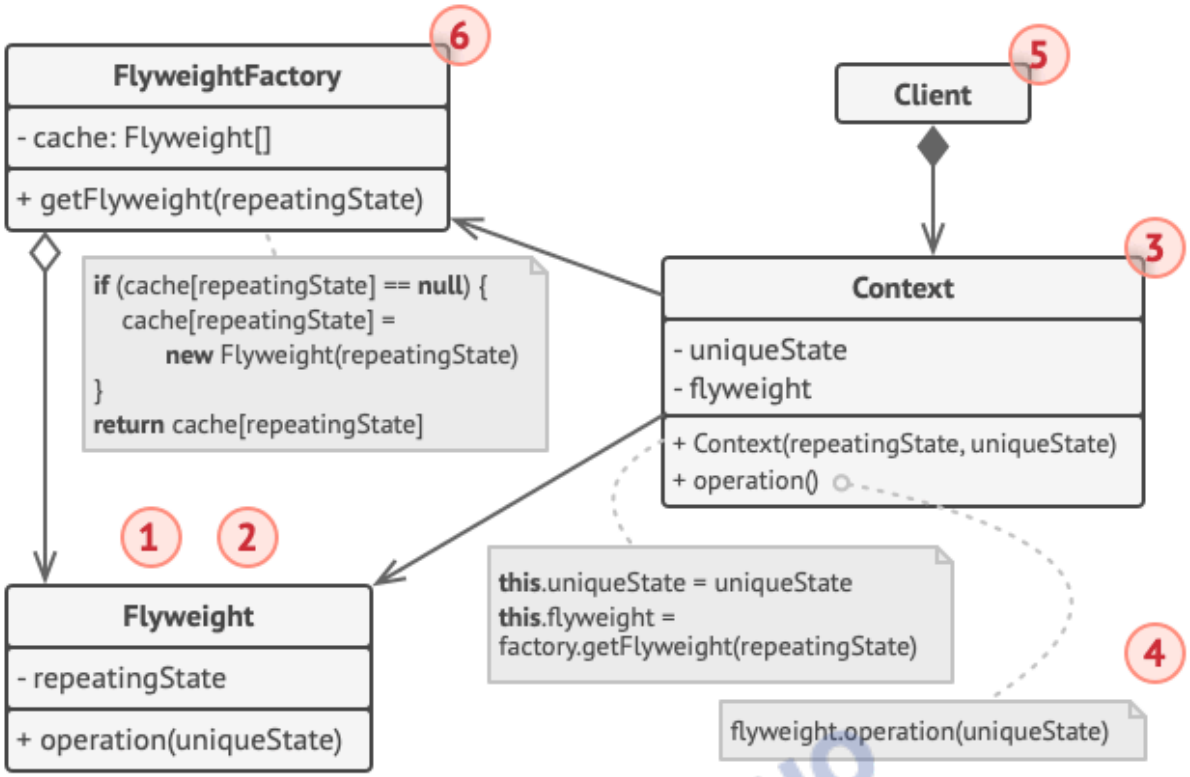
1）模式定义：

结构型模式，通过享元模式，利用共享对象保存多个对象共有的相同状态，

模式来源

程序使用了大量对象，大量类似对象大量内存开销。

2）模式结构（主要成员）



2.1）享元（flyweight）：包含原始对象中部分能在多个对象中共享的状态

2.2）上下文（context）包含原始对象中各不相同的外在状态

- 通常情况下原始对象的行为会保留在享元类中，因此调用享元方法必须提供部分外在状态作为参数

2.3）客户端

2.4）享元工厂（Flyweight Factory）控制享元的创建，管理享元缓存池。客户端调用工厂实现享元创建，传递目标享元的内在状态即可

3）适用场景

3.1）程序支持使用了大量继承同一个类仅在细粒度参数上有区别的对象。且没有足够内存容量。

- 程序需要生成大量相似的复杂对象
- 目标设备内存有限
- 对象中包含可抽取且能在多个对象之间共享的重复状态

4）优缺点

- 优点
  - 如果程序中有很多相似对象， 那么你将可以节省大量内存。
- 缺点
  - 享元方法每次调用需要依赖上下文信息
  - 对原有实体属性进行抽象，代码可读性降低

5）与其他模式的关系

5.1）配合组合模式树共享叶节点节省内存

5.2）与外观模式区别：

- 享元模式展示如何生成大量小型对象
- 外观模式展示如何用一个对象代表整个子系统

5.3）与单例模式区别：当共享状态简化为一个享元对象时候，享元模式近似单例模式

- 享元模式可以有多个实体，且实体内部状态不同
- 单例对象是可变的，享元对象是不可变的

12-代理模式

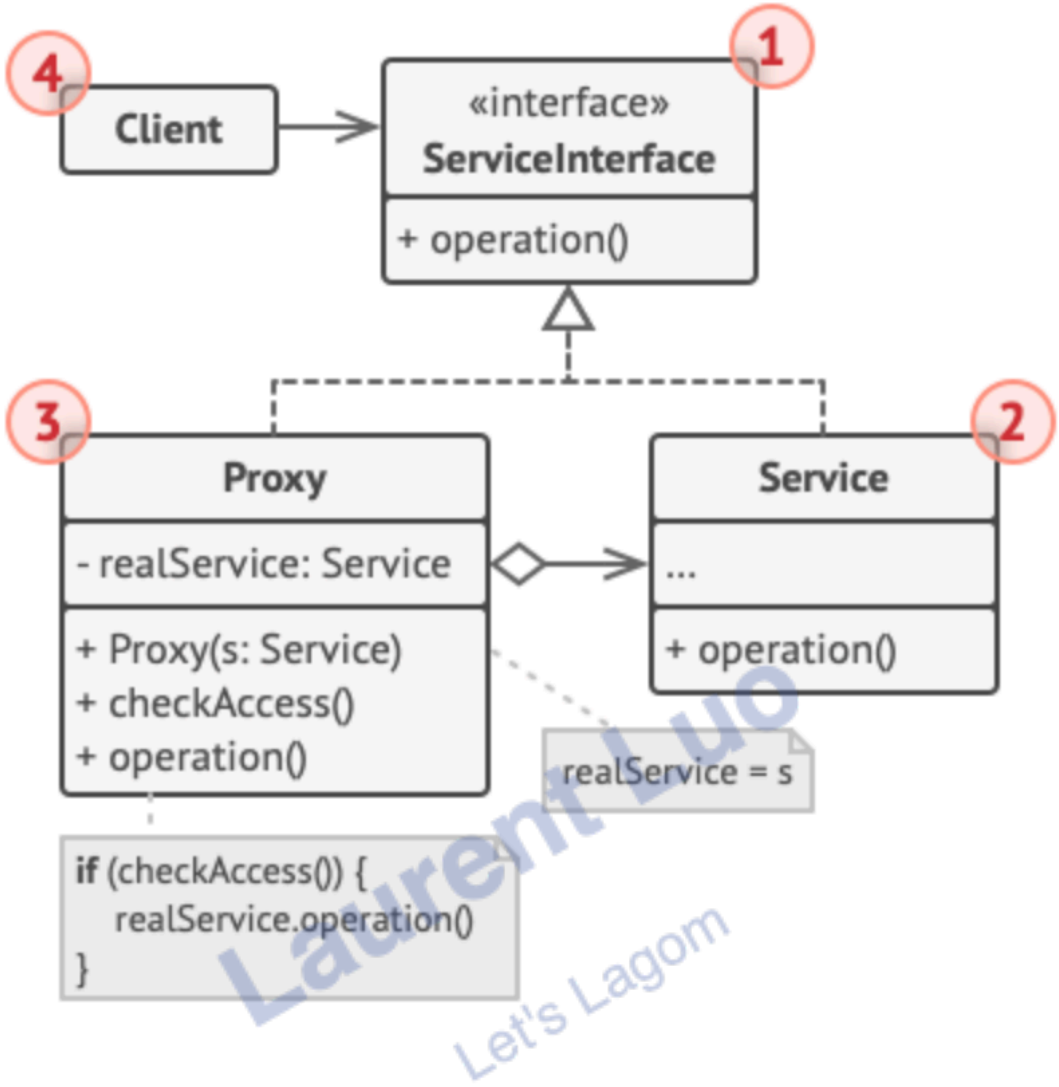
1）模式定义：

结构型模式，提供访问对象的代理。代理提供请求的间接性转发，实现对原对象的访问权限的控制。允许在请求提交给对象前后进行额外处理。

模式来源：

控制对于某个偶尔调用资源消耗较大的复杂对象的访问权限

2）模式结构（主要成员）



2.1）服务接口（Service Interface）声明了服务接口。代理遵循服务接口

2.2）服务（Service）实现服务接口业务逻辑。

2.3）代理（Proxy）包含指向服务对象的引用成员变量。

- 接收到请求后，将请求传递转发给服务对象。
- 通常代理会对服务对象的生命周期进行管理

2.4）客户端：通过同一接口与服务或代理进行交互

3）适用场景

3.1）延迟初始化，对于偶尔使用的重量级服务对象。一直保持运行此类对象会消耗系统资源。

3.2）控制访问，保证特定客户端请求特定服务对象

3.3）本地执行远程服务（远程代理）

3.4）记录日志请求，保存对于服务对象的请求记录

- 缓存请求结果，对缓存生命周期进行管理

3.5）管理复杂服务对象的生命周期，在没有客户端使用时候，销毁对象

4）优缺点

- 优点
  - 对于客户端来说，代理即服务
  - 代理可以对服务对象的生命周期进行管理，及时销毁释放资源
  - 即使对象不存在，代理也可以正常工作
  - 开闭原则：通过创建新代理满足新需求，避免对服务进行修改
- 缺点
  - 创建代理类导致代码复杂性提升
  - 服务响应存在延迟

5）与其他模式的关系

5.1）适配器模式、装饰器模式的区别

- 代理模式为对象提供相同接口
- 适配器为被封装对象提供不同接口
- 装饰模式则是为对象提供加强的接口

5.2）与外观模式的区别：外观模式与代理模式都缓存了复杂实体，对其初始化

- 代理与服务对象遵循同一接口
- 外观模式是对子系统进行管理，外部暴露接口可能与其管理的子系统实例集合的通用接口不同

5.3）与装饰器模式的区别：装饰模式与代理模式的结构相似，都是基于组合原则，将部分工作委派给另一个对象。

- 装饰的生成由客户端控制
- 代理自行管理服务对象的生命周期

三、行为模式

13-责任链模式

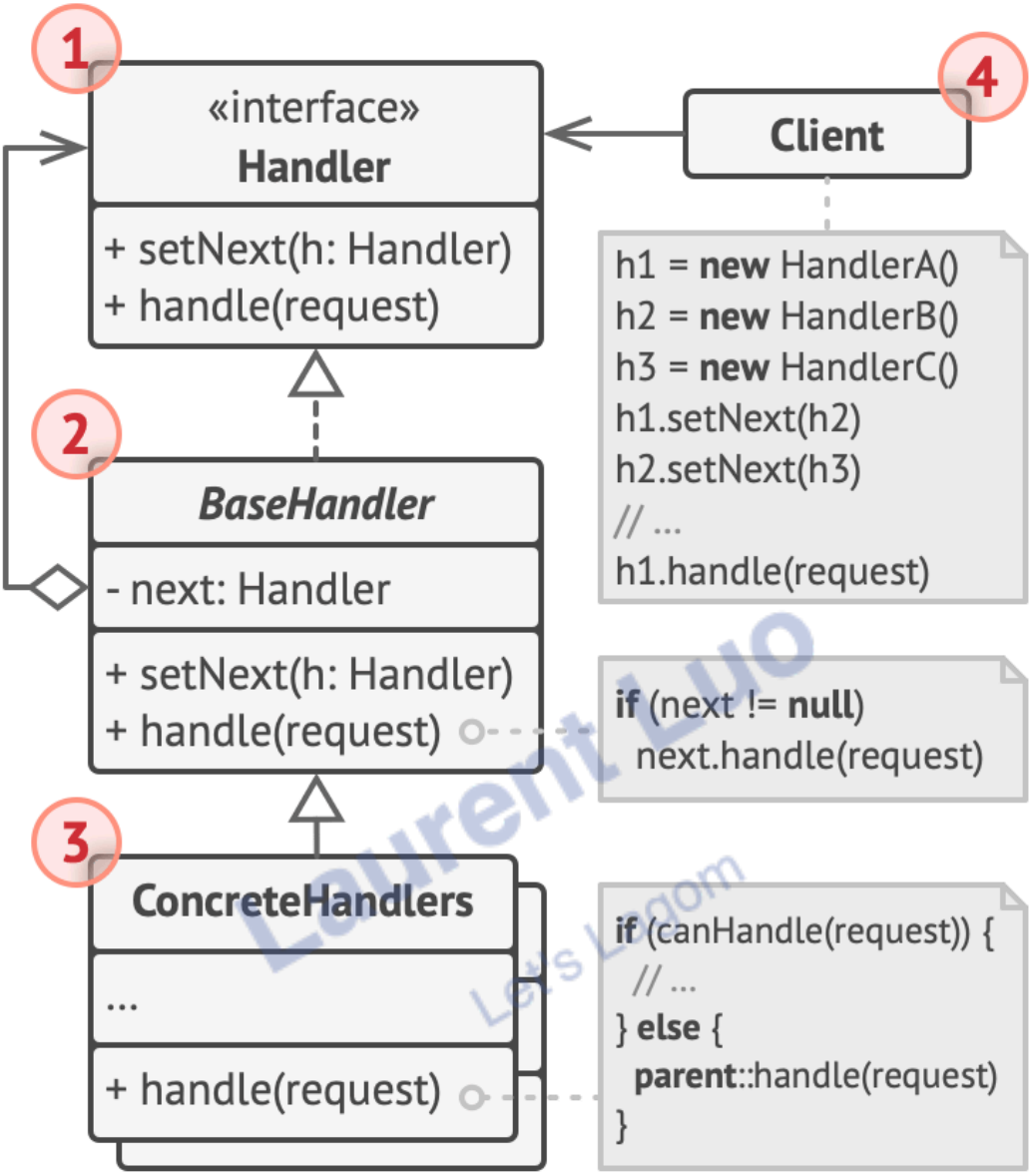
1）模式定义：

行为设计模式，将请求沿着处理者链进行发送（代理/组合树）。每个处理者都可以处理请求或将此请求传递给后继处理者处理。解耦客户请求和处理客户请求的操作对象。

模式来源

不同请求的响应权限不同（用户身份认证责任链）（对象树）

2）模式结构（主要成员）



2.1）处理者（Handler）声明具体处理者的通用接口

- 请求处理方法
- 后继处理者设置

2.2）基础处理者（Base Handler）可选类，保管所有处理者共用的方法（声明后继处理者设置方法）

2.3）具体处理者（Concrete Handler）处理请求的实际代码。

- 判断是否能对请求进行处理
- 判断是否需要传递给后继者

2.4）客户端：发送请求给任意一个处理者

3）适用场景

3.1）当需要使用不同方式处理不同种类请求，并且请求类型和顺序未知

3.2）当必须按顺序执行多个处理者时（模型接口串型调用）

3.3）如果所需处理者及其顺序必须在运行时改变（动态插入、移除处理者，改变责任链顺序）

4）优缺点

- 优点
  - 控制请求处理顺序
  - 单一职责原则：对发起操作与执行操作的类进行解耦
  - 开闭原则：通过新增处理者实现新增需求
- 缺点
  - 部分请求可能为能未被处理

5）与其他模式的关系

5.1）责任链模式、命令模式、中介者模式、观察者模式 实现了处理请求发送者和接受者之间不同连接的方法

- 责任链模式：按顺序动态传递请求
- 命令模式：将命令封装成独立对象，建立发送者和请求者之间建立单向连接
- 中介者：清除发送者和请求者之间的直接连接，利用中介转发通信
- 观察者：允许接收者动态订阅或取消接受请求

5.2）搭配组合模式，叶组件接收到请求后，可以将请求沿着全体父组件的链一直传递至对象树的底部

5.3）责任链管理者通过命令模式实现?

- 将请求封装成独立对象，用于职责链处理。

5.4）责任链与装饰模式区别（定义包装对象的装饰顺序）

- 责任链和装饰模式依赖递归传递请求
- 责任链的管理者可以独立执行一切操作，或停止传递请求。
- 装饰无法中断请求传递

14-命令模式

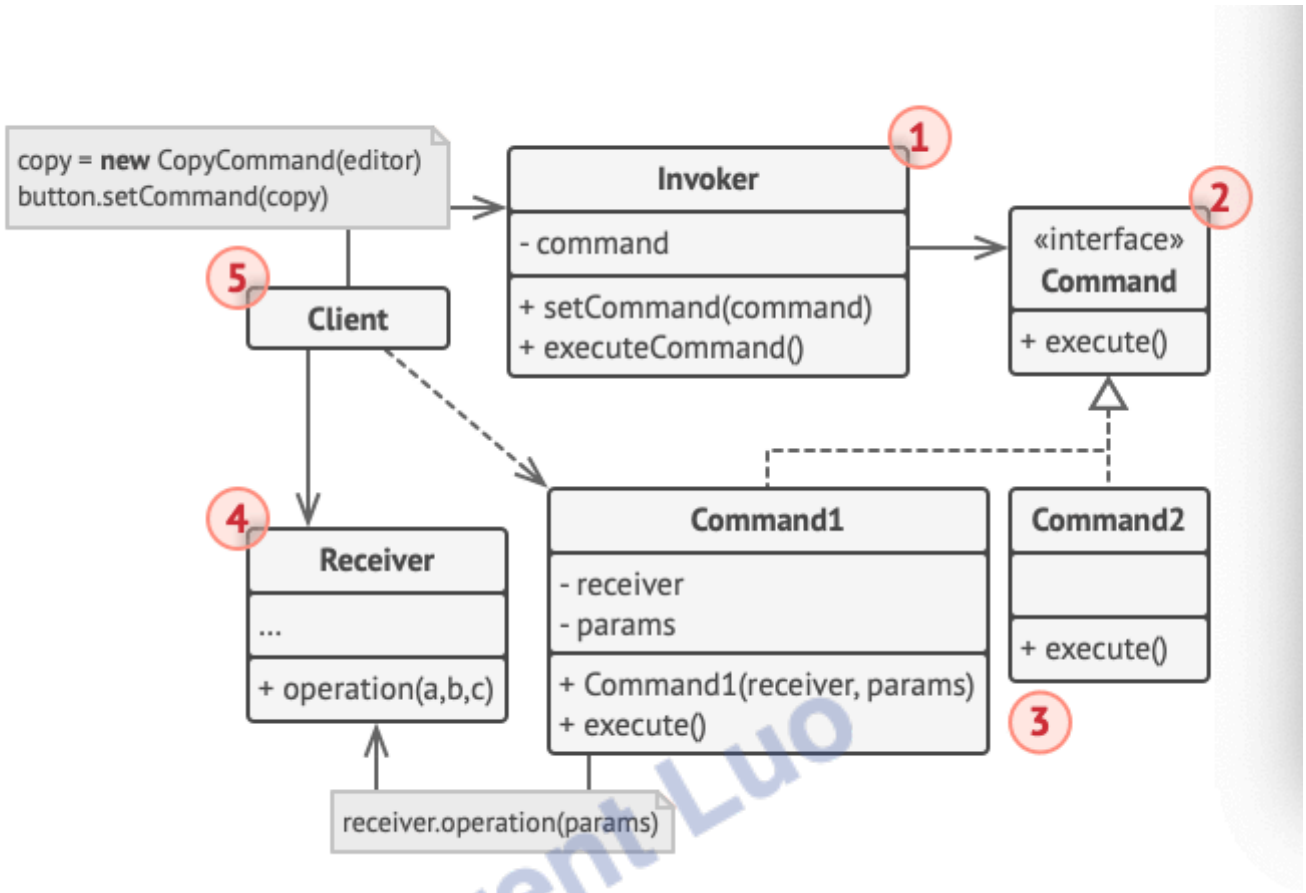
1）模式定义：

行为设计模式，将请求转换为包含请求相关所有信息的独立对象。根据不同请求将方法通过转换实现参数化、延迟请求执行、可撤销等操作。

模式来源

- 包含行为的参数化UI元素的回调函数。还被用于对任务进行排序和记录历史操作。
- 继承造成大量子类，基类的修改代价大
- 传统业务分层：将软件拆分成GU层I、业务逻辑层。但导致GUI访问业务逻辑层

2）模式结构（主要成员）



2.1）发送者（Sender）负责对请求进行初始化，包含一个成员变量存储对命令对象的引用。发送者触发命令（兵法发送请求），同时发送者不负责创建命令对象，通常通过构造函数从客户端获取预生成的命令

2.2）命令（Commnad）接口通常仅声明一个执行命令的方法

2.3）具体命令（Concrete Command）实现各种类型的请求，具体命令自身并不完成工作，而是委派给一个业务逻辑对象

2.4）接收者（Receiver）包含部分业务逻辑，任何对象都可以作为接受者。接受者负责处理相应请求。

2.5）客户端（Client）创建并配置具体命令对象，客户端将包括接受者实体在内的所有请求参数床底给命令的构造参数，生成的命令实例就可以与一个或多个发送者相关联了。

3）适用场景

3.1）需要通过操作来参数化对象

- 将特定的方法调用转化为独立对象

3.2）（用户请求序列）可以将操作放入队列中、操作的执行、远程执行操作

3.3）实现操作的撤销和回滚，使用命令模式/备忘录模式

4）优缺点

- 优点
  - 维护单一职责：解耦触发和执行操作的类
  - 维护开闭原则：创建新命令，不需要修改现有客户端代码
  - 实现撤销和恢复功能
  - 实现操作的延迟执行
  - 实现命令之间的组合
- 缺点
  - 增加了代码的复杂度

5）与其他模式的关系

5.1）责任链模式、命令模式、中介模式、观察者模式用于处理请求发送者和接受者之间的不同连接方法

- 责任链按照顺序将请求动态传递给一系列潜在的接受者
- 命令模式在发送者和请求者之间建立单向连接
- 中介者要求发送者和请求者之间的通信通过中介对象
- 观察者允许接受者动态订阅或取消接受请求

5.2）搭配责任链模式

- 责任链管理者可以使用命令模式实现，利用命令模式包装责任链，命令模式负责指派合适的职责链处理业务

5.3）配合备忘录模式，实现撤销。

- 命令用于对目标对象执行不同的操作，备忘录保存一条命令执行前该对象的状态

5.4）命令模式和策略模式区别：

- 相同点：通过某些行为参数化对象
- 命令模式：将任何操作转换为对象，操作的参数作为对象成员变量。将操作放入队列、保存历史命令或向远程服务发送命令
- 策略模式在同一个上下文类切换不同的中间处理逻辑

5.5）搭配原型模式

- 使用原型模式构建命令实例

5.6）访问者模式区别：

- 访问者模式是命令模式的加强版本。访问者模式的对象可对不同类的多种对象执行操作。

15-迭代器模式（PS 对类方法重载实现类元素的迭代）

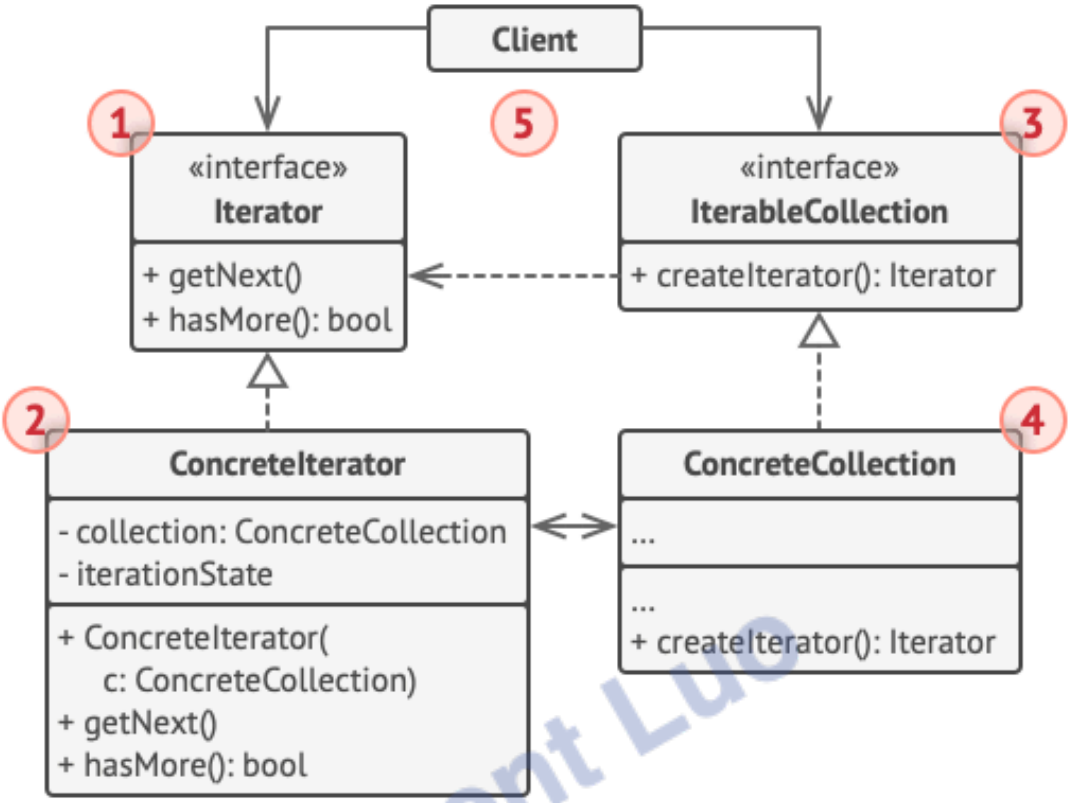
1）模式定义：

行为设计模式：在不暴露底层表现形式的情况下遍历集合中所有元素，将访问集合元素的遍历行为抽取为单独的迭代器对象

模式来源

集合使用不同数据结构存储元素，并提供某种访问元素的方式

2）模式结构（主要成员）



2.1）迭代器（Iterator）接口声明了遍历集合所需的操作

2.2）具体迭代器（Concrete Iterators）实现遍历集合的特定算法，迭代器对象需要跟踪当前遍历进度

2.3）集合（Collection）接口声明一个或多个方法获取集合兼容的迭代器

2.4）具体集合（Concrete Collection）在客户端请求迭代器时返回特定具体迭代器实体

2.5）客户端（）通过集合和迭代器接口与两者交互，无需与具体类进行耦合

3）适用场景

3.1）当集合的数据结构较为复杂时，希望对客户端隐藏复杂性时，使用迭代器模式

- 迭代器封装了与复杂数据结构的交互细节

3.2）减少程序中重复的遍历代码

3.3）当集合的数据结构无法预知时

4）优缺点

- 优点
  - 单一职责：将遍历算法抽象出单独的类
  - 开闭原则：通过新建迭代器类实现新的迭代算法
  - 可以实现并行遍历，因为每个迭代器对象都包含其自身的遍历状态
- 缺点

- 简单集合的交互并不需要使用迭代器模式
- 对于某些特殊集合，使用迭代器模式可能比直接遍历的效率更低

5）与其他模式的关系

5.1）搭配组合模式：

- 使用迭代器模式遍历组合树的所有节点

5.2）搭配工厂方法模式：

- 迭代器工厂让子类集合返回不同类型的迭代器

5.3）同时使用访问者模式和迭代器来遍历复杂数据结构

016-中介模式

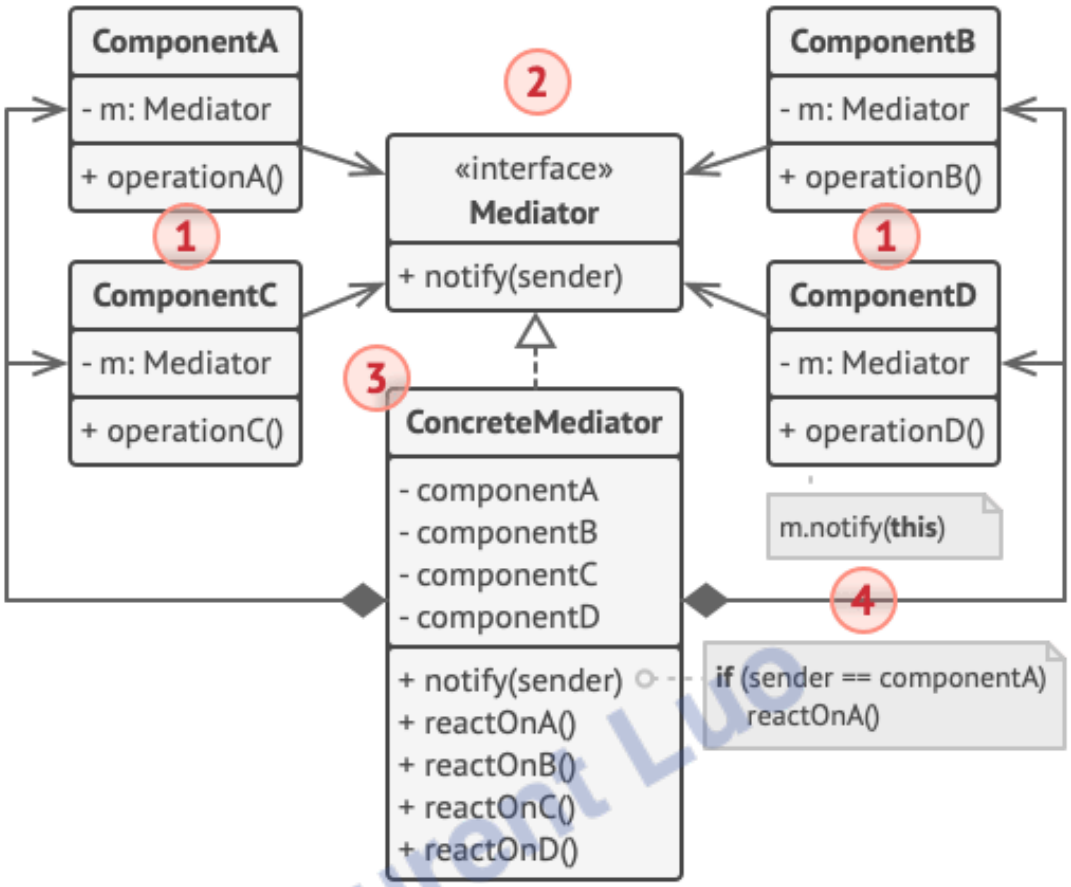
1）模式定义：

行为设计模式，维护迪拉米特法则，减少对象之间无序的依赖关系。限制对象之间的直接交互，只能通过中介对象实现通信。

模式来源

GUI组件之间的点对点的通信，导致组件之间的关系复杂，容易形成多对多的关系网络。

2）模式结构（主要成员）



2.1）组件（Component）包含业务逻辑的类，每一个组件都有一个指向中介者的引用。该引用被声明为中介者接口类型。但组件并不清楚中介者实际所属的类。

2.2）中介者（Mediator）接口声明了与组件的交流方法（通知方法）。组件将任意上下文（包括自己的对象）作为该方法的参数。解除请求接收组件和发送组件之间的耦合。

2.3）具体中介者（Concrete Mediator） 封装了多种组件间的关系。具体中介者通常保存所有组件的引用并对其管理（控制生命周期）

- 组件之间的通信，通过通知中介者，中介者收到通知后，确定发送者，由发送者及参数确定接受者。
- 因此对于组件来说，中介对象是黑箱。发送者不知道请求具体由谁响应。

3）适用场景

维持迪拉米特法则：

- 非作用不通信，若联系三方转
- 当一些对象和其他对象紧密耦合难以对其修改时，使用中介模式
- 当组件因过于依赖其他组件而无法在不同应用中复用时候（适配器、中介者模式）
- 为了在不同情境下复用一些基本行为，导致创建大量组件子类时（结构型模式），配合中介者模式

3.1）慎用：出现多对多的交互复杂对象群时，优先划分对象群各个对象的职责边界是否合理。

3.2）当对象群内部职责划分清晰，需要以复杂方式进行通信的场合

4）优缺点

- 优点
  - 单一职责原则：中介者承担组件交流抽取到同一位置，便于理解与维护
  - 开闭原则：无需修改实际组件就能增加新的中介者
  - 减轻应用中多个组件间的耦合
  - 提升组件的复用性
- 缺点
  - 中介与组件形成一对多的关系，成为上帝对象

5）与其他模式的关系

5.1）责任链、命令模式、中介模式、观察者模式用于处理请求发送者和接受者之间的不同连接方式：

- 责任链按顺序动态传递请求给接收者
- 命令模式将请求包装成为独立对象，建立发送者和请求者之间的单向连接
- 中介者清除发送者和请求者之间的直接联系
- 观察者允许接受者动态订阅或取消接收请求

5.2）外观模式和中介者职责类似：在紧密耦合的类中组织起合作

- 外观模式管理子系统集合，子系统之间可以直接交流，但子系统并不知道外观对象的存在
- 中介者将系统组件的沟通行为中间化，组件只知道中介对象。

5.3）中介者和观察者之间的区别

- 中介者的目的是消除系统组件之间的依赖
- 观察者是在对象之间建立动态单向连接
- 中介者模式实现方式依赖于观察者，当中介者承担发布者角色时，其他组件作为订阅者。因此在行为模式上是类似的。
- 如果所有组件都成为发布者，他们之间可以建立动态链接，就实现了去中心化，成为了分布式观察者模式

017-备忘录模式

1）模式定义：

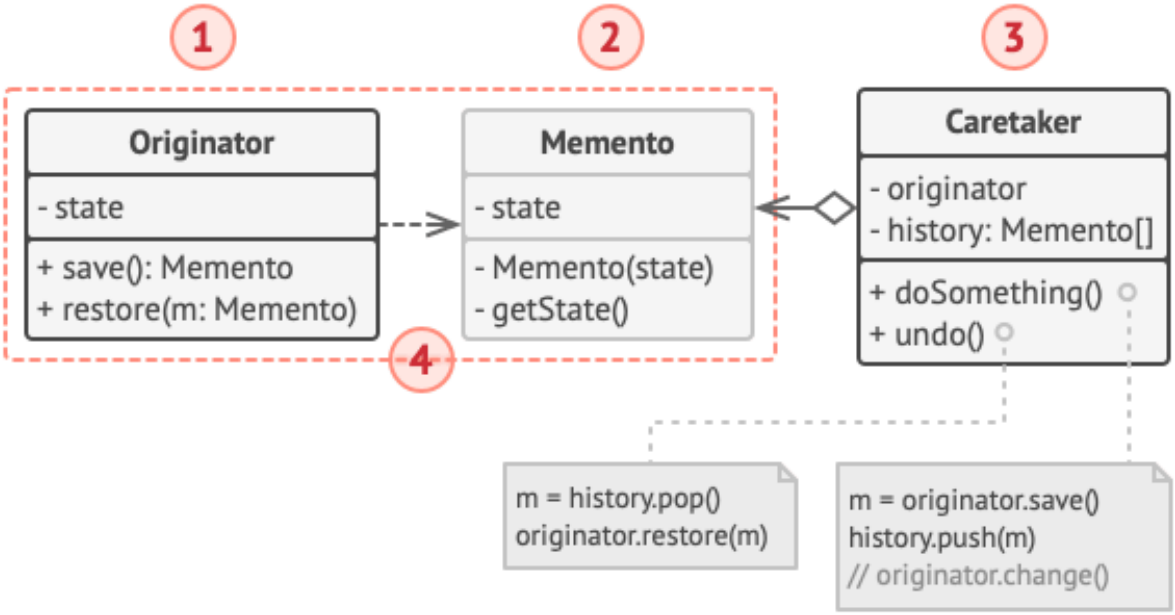
行为设计模式，允许在不暴露对象实现细节的情况下保存和恢复对象之前的状态。

模式来源

用户执行撤销操作，程序从历史记录获取最近的快照，用于恢复所有对象的状态。此时需要考虑对象的状态中是否存在私有属性，以及封装破损的情况。

2）模式结构（主要成员）

- 基于嵌套的实现（C++，JAVA）



备忘录类被嵌套在原发器中，这样原发器就可以访问备忘录成员变量和方法（即使声明为私有），那么对于负责人来说，只能在栈中保存备忘录，不能修改状态。

2.1）原发器（Originator）：类可以生成自身状态的快照（由实例自己保存自己状态），必要时通过快照恢复自身状态

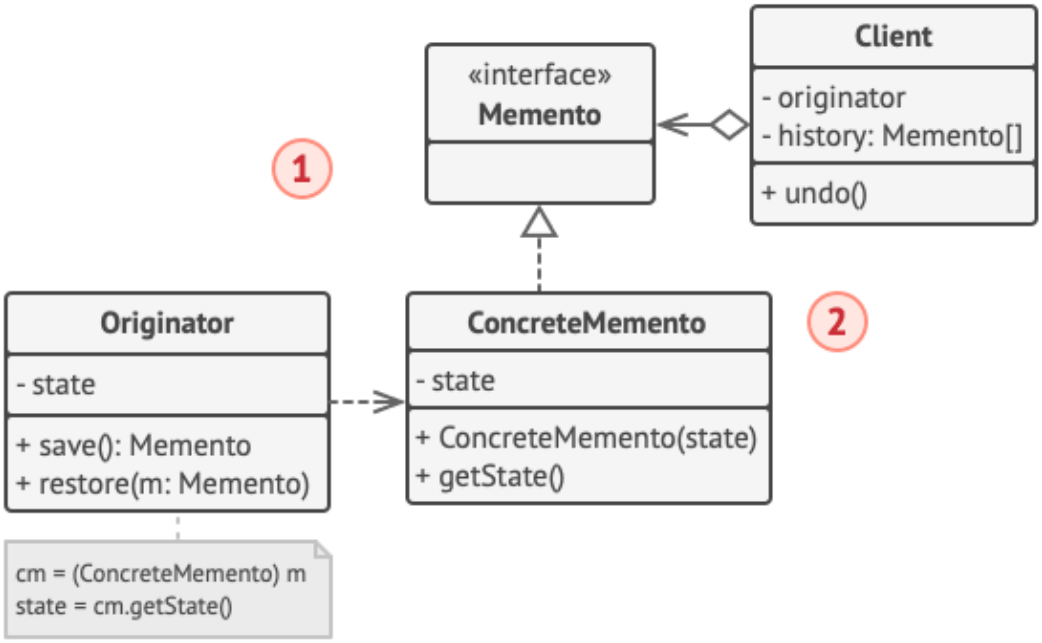
2.2）备忘录（Memento）原发器状态快照的值对象（value object），将备忘录设为不可变的，通过构造函数一次性传递数据

2.3）负责人（Caretaker）负责发起生成快照的指令与发起让原发器恢复状态的指令

- 负责人通过保存备忘录栈来记录原发器的历史状态。当原发器需要回溯时，负责人将栈中最近历史备忘录传递给原发器的恢复方法

——

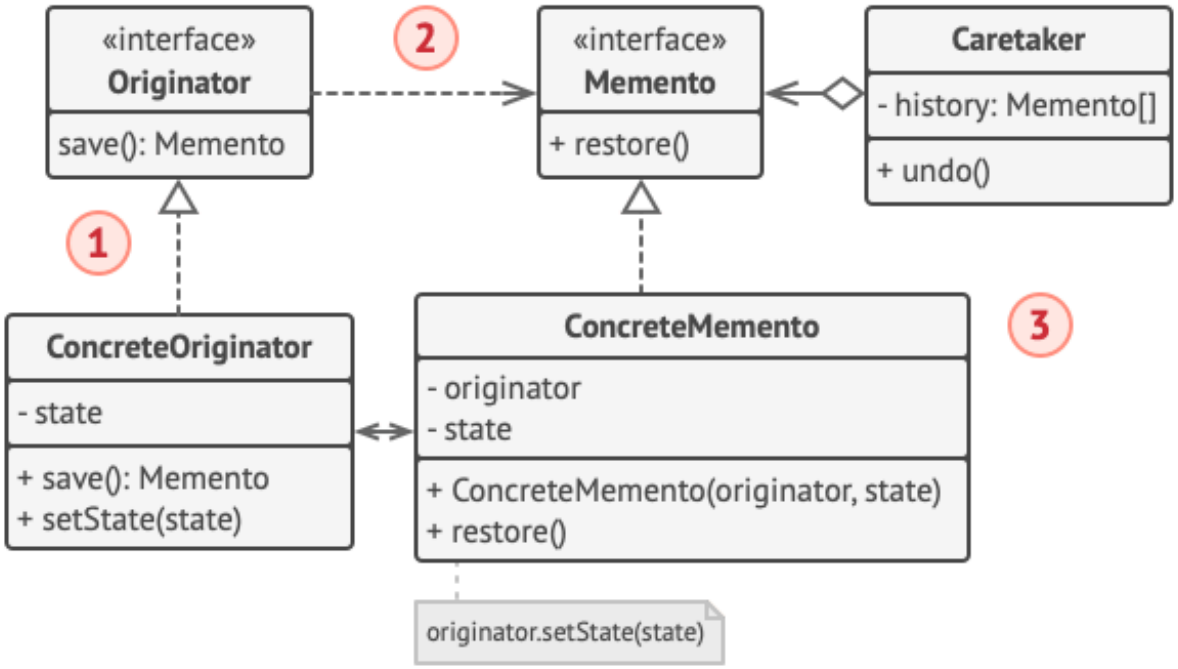
- 基于中间接口的实现（PHP）



2.4）在没有嵌套类的情况下，可以规定负责人通过声明中间接口与备忘录互动

2.5）原发器直接与备忘录对象进行交互，访问备忘录类中声明的成员变量和方法。（缺点：需要将备忘录所有成员变量声明为公有）

- 封装更加严格的实现（非嵌套）｜大话设计模式的实现方式



2.6）单独抽象原发器与备忘录类。备忘录定义状态恢复方法。

- 原发器类：定义自身状态，实现创建备忘录方法，实现状态转换方法

2.7）Memento备忘录类

- 类实例化操作由原发器实现（建立一对一关系）

2.7）Caretaker负责人类，只负责存储/调用相应的备忘录。

- 没有修改备忘录内部状态的权限。

2.9）原发器将实例自身与状态作为参数传递给备忘录类的构造函数。

- 状态的恢复方法由原发器实现(setstate())。

3）适用场景

3.1）功能负责，需要维护或记录历史信息的类，通过备忘录模式创建对象快照恢复之前状态。

3.2）保护数据安全性，避免封装破损的情况（对象成员私有、保护变量被外部对象直接访问情况）

3.3）配合命令模式，利用备忘录模式存储可撤销的操作状态

4）优缺点

- 优点
  - 不破坏对象封装情况下创建对象快照
  - 通过负责人维护原发器的历史状态，将原发器状态管理与原发器对象业务处理逻辑隔离
- 缺点
  - 如果客户端频繁创建快照，导致内存消耗
  - 对象生命周期不可控：负责人类由于管理备忘录，其生命周期依赖原发器的生命周期。
  - 绝大部分动态编程语言不能确保备忘录中的状态不被修改（Python）

5）与其他模式的关系

5.1）配合命令模式实现撤销与对象回溯。

- 命令用于执行不同操作
- 备忘录用来保存一条命令执行前的对象状态

5.2）配合迭代器模式，获取当前迭代器状态，在需要时回滚

5.3）原型模式可以看作备忘录模式的简化版本

- 原型模式是将对象当前状态进行复制，所处理的状态较简单，不需要链接外部状态或资源。

018-观察者模式

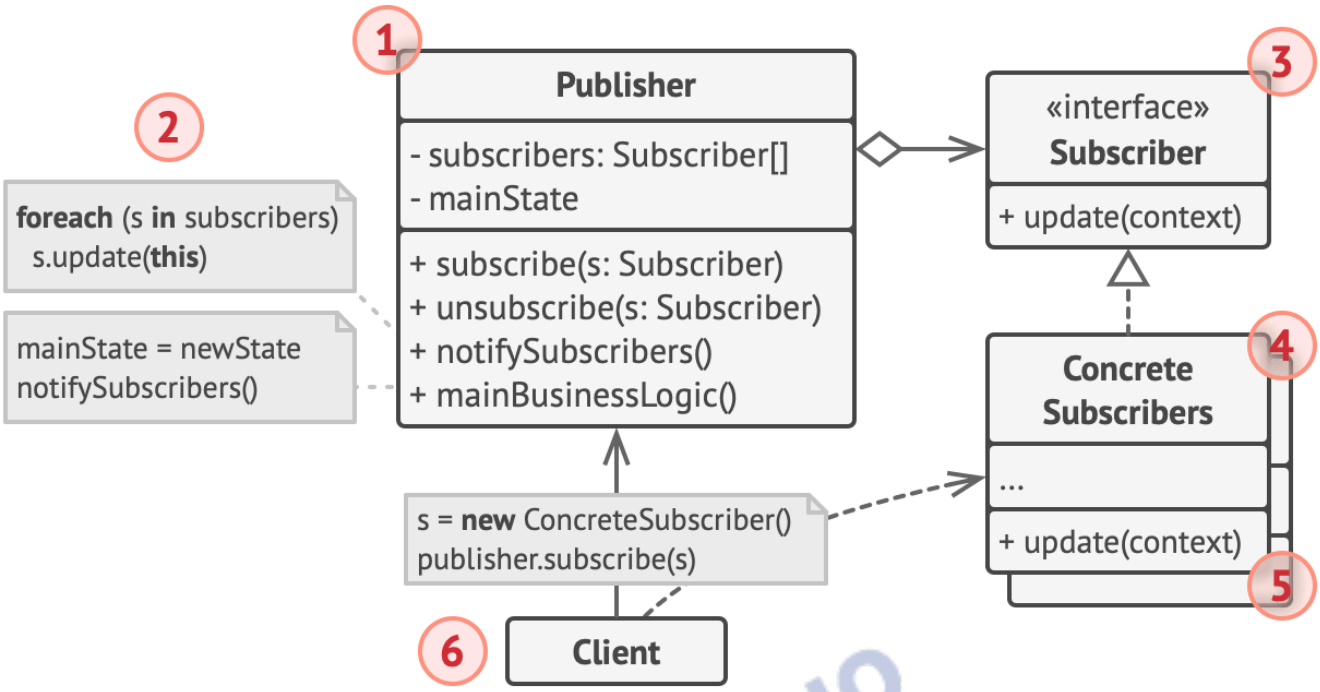
1）模式定义：

行为设计模式，定义一种订阅机制，在对象事件发生时通过多个“观察”该对象的其他对象。

模式来源

类之间的行为收到某类对象的状态变化影响。对于此类关系

2）模式结构（主要成员）



2.1）发布者（Publisher）：

- 与订阅者属于一对多的关系。向其他对象发送值得关注的事件。
- 发布者维护用于存储订阅者对象引用的列表。
- 当新事件发生时，发送者遍历订阅列表并调用每个订阅者对象的通知方法。（订阅方法由订阅者基类声明）

2.2）订阅者基类（Subscriber）接口声明了公开的通知接口。

2.3）具体订阅者（Concrete Subsctibers）实现通知接口，并执行一些操作来回应发布者的通知。

- 对于通知的响应可能需要部分上下文信息，因此发布者可以将自身作为参数传入。

2.4）客户端（client）创建发布者和订阅者对象，为订阅者注册

3）适用场景

3.1）当一个对象的状态是动态变化时，且此对象的状态改变需要改变其他对象（parser模块将自己作为参数传入，根据不同的message type 接口工厂绑定不同的服务接口，不同的服务接口调用不同的数据格式化工厂，调用不同的格式化方法。）

3.2）当应用中的一些对象必须观察其他对象，可使用该模式（并行服务接口调用的提前结束机制）

4）优缺点

- 优点
  - 开闭原则：无需改变发布者代码就可以引入新的订阅者
  - 可以在运行引入发布者类
- 缺点
  - 订阅者的通知顺序是随机的

- 抽象订阅者还是依赖于抽象通知者，因此需要维护一个通知者能够调用的公开接口。（继承机制）

5）与其他模式的关系

5.1）责任链、命令、中介者和观察者模式都是用于处理请求发送者和接受者之间的不同连接方式

- 责任链，按顺序将请求动态传递给一系列接受者，直到接受者对请求进行处理
- 命令模式为将请求封装成独立对象，建立发送者和请求者之间的单向连接
- 中介者解除发送者和请求者之间的直接连接
- 观察者允许接受者动态订阅或取消接受请求

5.2）中介者和观察者的区别

- 中介者消除系统组件之间的相互依赖，并将依赖转移到一个中介对象上，形成一对多的联系。
- 观察者是在对象之间建立动态的单向连接。
- 某种实现方式上，中介者依赖于观察者模式。
- 当所有的组件都变成了发布者，发布者之间还可以建立动态链接，此时程序完成了去中心化

5.3）通过事件委托解决观察者模式的不足

- python 通过鸭子类型，解除继承机制，实现不同类实例声明同一个接口。（只需要保证方法的参数列表和返回值类型一致即可）
- 发布者维护一个事件委托，由事件委托管理不同的订阅者。因此不同订阅者不需要继承同一个父类。

019-状态模式

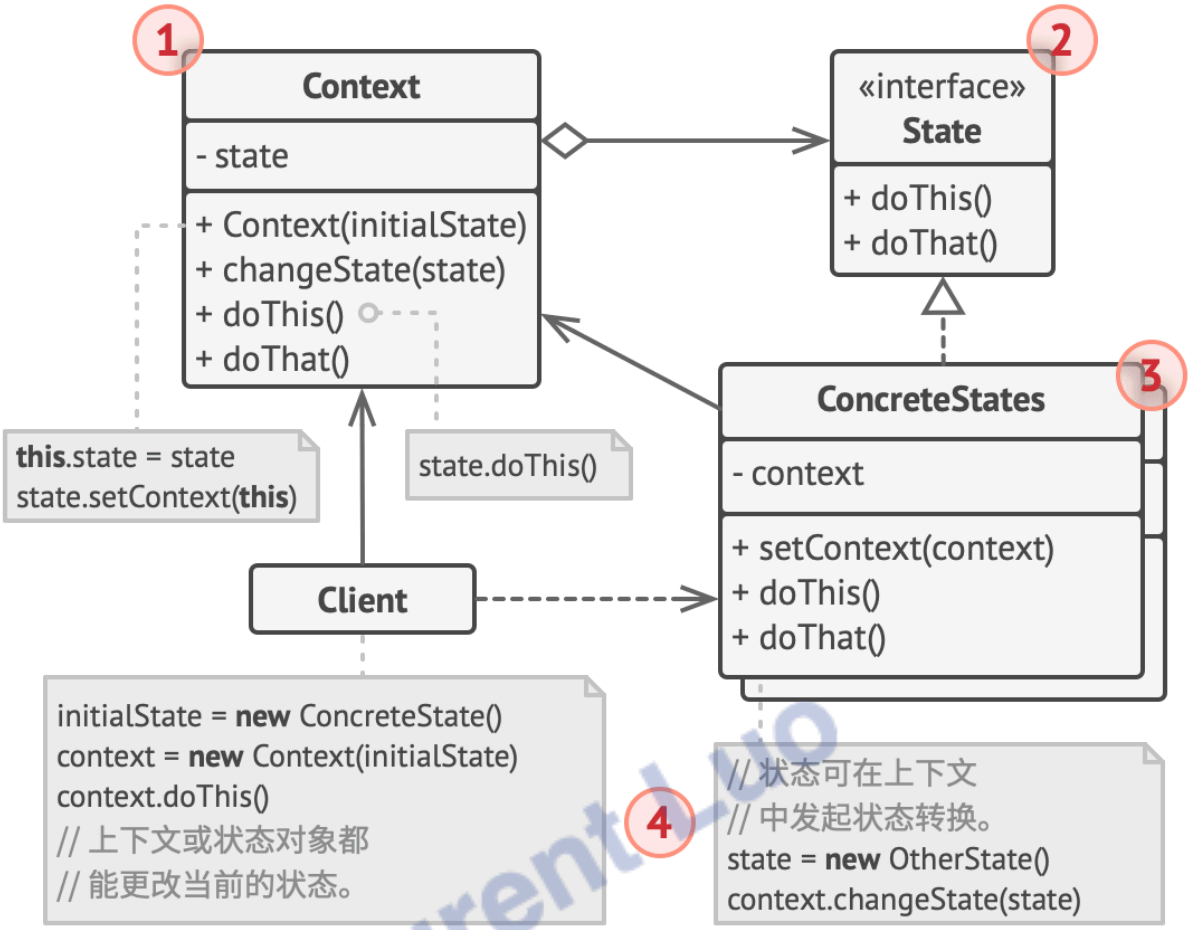
1）模式定义：

行为设计模式，对象的行为取决于其状态，状态模式在对象的内部状态变化时，改变其行为。状态模式替换条件判断实现的有限状态机，使其看上去就像改变了自身所属的类一样。

模式来源：

程序在任意时刻会处于几种有限的状态中，在任何一个特定状态中，程序的行为都不相同。（Word 文本编辑模式、网页模式、大纲模式）

2）模式结构（主要成员）



2.1）上下文（Context）：保存了对于一个具体状态对象的引用，并会将所有与该状态相关的工作委派给他们。

- 上下文通过抽象状态接口与具体状态对象交互
- 提供一个设置器用于切换自身状态对象

2.2）状态（state）接口声明特定于状态的方法。这些方法能被所有具体状态理解。

2.3）具体状态（Concrete State）实现特定于状态的方法，由上下文实例调用。

- 状态切换信号：来自作为参数输入的外部变量
- 具体状态间的切换操作可以由：上下文 or 具体状态实现。
- 可以提供一个封装有部分通用行为的中间抽象类

3）适用场景

3.1）对象基于自身不同状态进行不同行为，尽管状态有限，但是对象自身状态变换频繁。

- 将所有特定于状态的代码抽取到一组独立的类。

3.2）某个类需要根据成员变量的当前值改变自身行为，导致使用大量的条件判断语句时。

3.3）基于条件的状态机转换中存在许多重复代码，不同状态具有相似模板代码，使用状态类层次结构，将公用代码抽取到抽象基类中。

4）优缺点

- 优点
  - 单一职责原则：将特定状态相关的代码放在单独类中
  - 开闭原则：无需修改已有状态和上下文就能引入新状态
  - 消除有限状态机的条件语句
- 缺点
  - 评估状态数量和变换频率判断是否使用状态模式（简单工厂模式）

5）与其他模式的关系

5.1）桥接模式、状态模式、策略模式的接口相似。都基于组合模式，将工作委派给其他对象。

- 桥接模式：工作执行对象为具体实现类
- 状态模式：工作由具体状态类执行
- 策略模式：工作由具体策略类执行

5.2）状态模式可以看做策略模式的拓展。

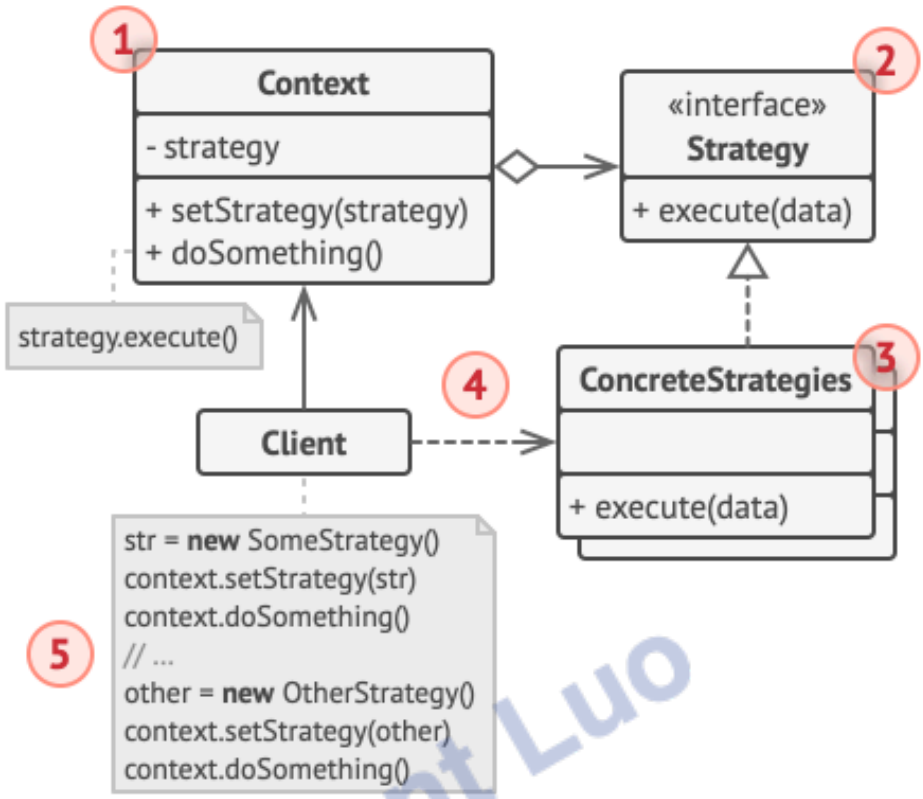
- 相同点：都通过将部分工作委派给其他对象改变其在不同情境下的行为。
- 策略模式：对象之间相互独立
- 状态模式：状态之间没有独立性限制，状态的切换操作可以通过上下文/状态

020-策略模式

1）模式定义：

行为设计模式，定义一系列算法，将每种算法分别放入独立的类中，使算法的对象能够相互替换。

2）模式结构（主要成员）



2.1）上下文（Context）：维护指向具体策略的引用。

- 实例化上下文类时候，传入具体策略类对象的索引。
- 提供设置器替换策略

2.2）策略（Strategy）接口是所有具体策略的通用接口，声明了上下文用于执行策略的方法

2.3）具体策略（Concrete Strategy）实现了上下文所用算法不同变体

- 当上下文需要运行算法时，调用绑定的策略的执行方法。

2.4）客户端（Clinet）创建一个特定策略对象将其传递给上下文

3）适用场景

3.1）使用不同算法变体，并希望在运行时切换算法。

3.2）行为之间相似但不同，将不同行为抽取到一个独立类层次结构

3.3）算法与上下文逻辑耦合性弱，那么利用策略模式将类的业务逻辑和其算法实现隔离。

3.4）替换通过条件判断实现算法变体的切换的堆砌代码

4）优缺点

- 优点
  - 实现对同一个需求不同解决方法调用的解耦。实现运行时切换对象内的算法
  - 隔离算法实现与算法的使用，避免条件语句的堆砌
  - 使用组合模式替代继承
  - 简化单元测试，基于具体算法的公开接口测试，提升模块健壮性，
  - 开闭原则：无需修改上下文就能引入新策略
- 缺点
  - 算法切换类别和频率较低，那么引入策略模式会增加代码复杂度
  - 客户端控制策略选择
  - python通过函数类型功能通过函数方法实现相同功能，无需借助额外的类和接口

5）与其他模式的关系

5.1）桥接模式（分离类的抽象和实现）、状态模式、策略模式的接口相似，基于组合模式将工作委派给其他对象

5.2）命令模式与策略模式的区别

- 命令模式将操作封装成对象，操作的参数成为对象的成员变量，实现操作的延迟执行，将操作放入队列。
- 策略用于描述某件事的不同方式，在同一个上下文中切换不同算法的变体（一个问题的不同解决策略）

5.3）与装饰模式的区别：

- 装饰器模式实现对通用接口进行加强
- 策略模式则替换上下文中接口调用的具体对象

5.4）与模版方法模式的区别：

- 模版方法模式通过扩展子类中的部分内容来改变部分算法。（静态）
- 策略基于组合机制，通过相应行为提供不同策略改变对象部分行为（动态）

5.5）与状态模式的关系：

- 相同点：都是基于组合机制，将工作委派给其他对象改变其在不同情境下的行为
- 不同点：策略使得这些对象之间相互完全独立/具体状态类之间是可以切换和依赖的

021-模版方法模式

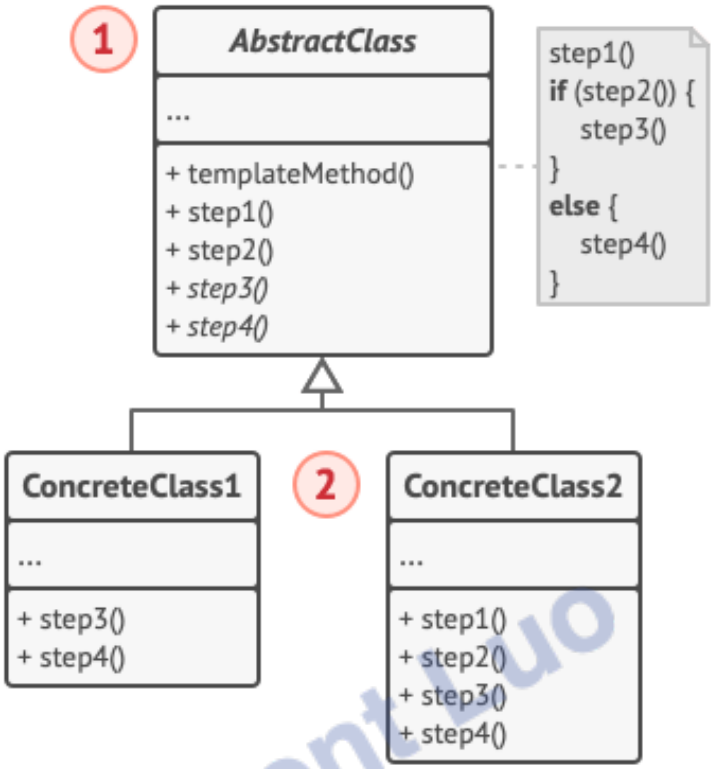
1）模式定义：

行为设计模式，在超类中定义了一个算法框架，允许子类在不改变结构的情况下，重写算法的特定步骤

模式来源

数据挖掘中包含许多重复代码，数据处理和分析的代码一摸一样，在保持算法结构完整的情况下，对代码进行简化

2）模式结构（主要成员）



- 模板方法的模板设计配合工厂模式、原型模式
- 模板方法实现上：搭配策略模式、享元模式
- 模板方法的选择调用上：搭配命令模式

2.1）抽象类（AbstractClass）声明作为算法步骤的方法。以及顺序调用他们的实际模板方法

- 抽象类提供三种类型的接口：
  - 默认操作步骤接口（子类不可重写）
  - 自定操作接口（抽象方法，用于具体类实现）
  - 可选操作接口（具体类根据业务需求决定是否进行重写）
- 声明算法步骤及调用流程

2.2）具体类（Concrete class）重写所有步骤，但不能重写模板方法自身

- 实现自定操作接口
- 根据需求实现可选操作接口

3）适用场景

3.1）整体算法步骤确定，在不改变整体结构的前提下，需要对特定算法步骤进行扩展，

- 将整个算法转换为一系列独立的步骤，便于子类拓展

3.2）当类的算法包含大量相似代码时，使用模板方法抽取出公用确定流程

4）优缺点

- 优点
  - 允许客户端重写一个大型算法中的特定部分
  - 将重复代码提取到超类中（复用父类已有代码）
- 缺点
  - 一旦默认步骤变化，就需要修改所有的类
  - 子类一直默认步骤实现导致违反里氏替换原则（子类可以拓展父类功能，但不能改变父亲原有功能）

5）与其他模式的关系

5.1）搭配工厂方法：工厂方法模式是模板方法模式的一种特殊形式，

- 工厂方法可以作为一个特定的算法步骤（算法步骤的实现还可以通过状态模式、策略模式）

5.2）模板方法与策略模式的区别

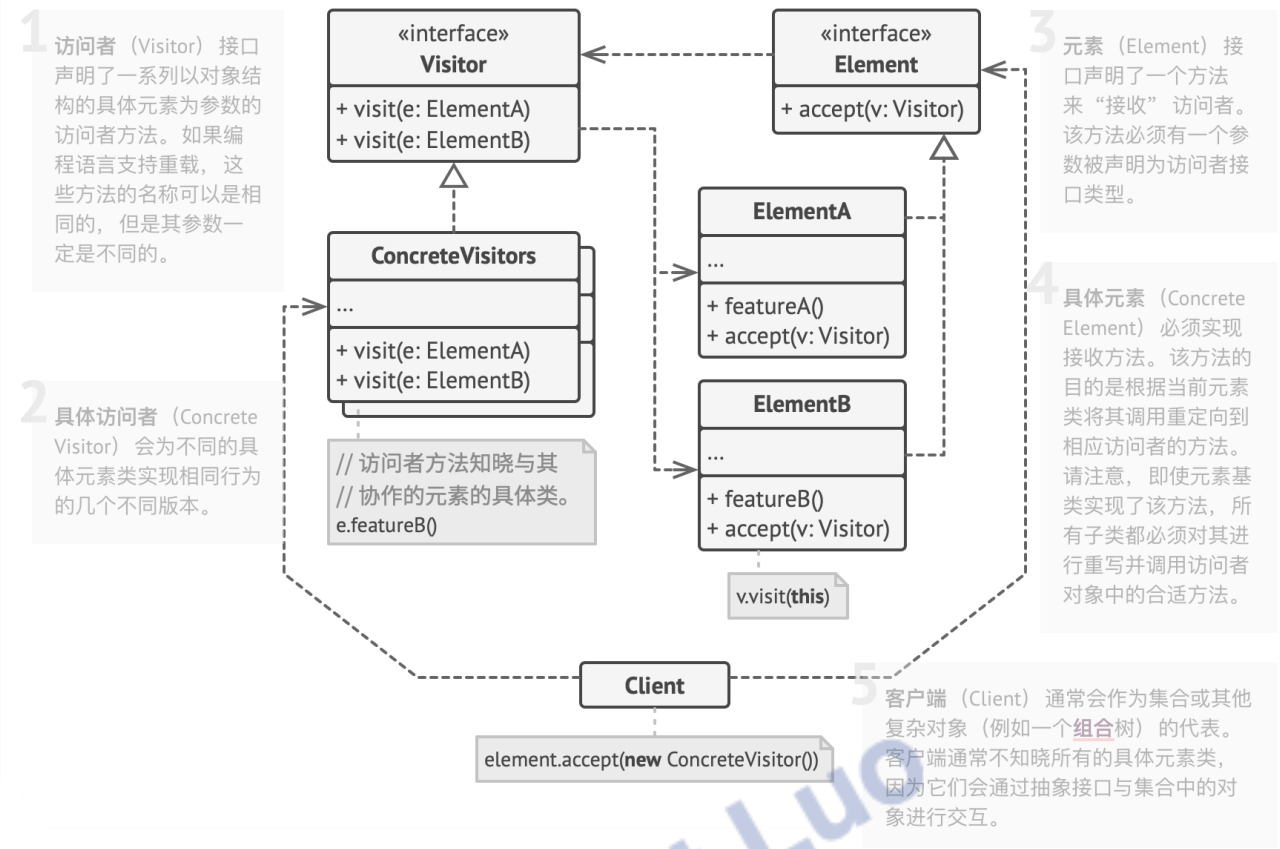
- 模板方法基于继承机制/策略模式基于组合机制
- 策略模式为相同方法提供不同策略改变对象的部分行为，是可动态切换
- 模板方式只在类层次运作，是静态的。

022-访问者模式

1）模式定义：

行为设计模式，将算法与其所作用的对象隔离

2）模式结构（主要成员）



2.1）访问者（VIsitor）：基于具体元素的数量，为每一个元素声明接口

- 要求具体元素数量是可预估且数量稳定（男，女）。

2.2）具体访问者（Concrete Visitor）会为不同的具体元素类实现相同行为的几个不同版本

2.3）元素（Element）声明对象结构序列调用具体元素类的接口，以访问者实例为参数。

2.4）具体元素（Concrete Element）实现接收方法，根据当前元素类将其调用重定向到相应访问者的方法。

2.5）客户端（Client）客户端通过抽象接口和集合中的对象进行交互

3）适用场景

- 使用前提：具体元素类划分清晰，元素数量稳定

3.1）对复杂对象结构中所有元素执行某些操作

3.2）使用访问者模式来清理辅助行为的业务逻辑

- 将非主要行为抽取到一组访问者中。

3.3）当某个行为仅类层次结构中一些类中有意义，在其他类中没有意义时，使用该模式

4）优缺点

- 优点
  - 开闭原则：引入不同类对象上执行的新行为
  - 单一职责原则：将同一行为的不同版本移到同一个类
  - 在与各种对象交互过程中收集信息，遍历树结构的过程记录子节点信息。
- 缺点
  - 基于继承机制，在元素层次结构中添加或移除一个类，需要更新所有的访问者
  - 访问者同某个元素进行交互，访问者可能没有访问元素私有成员变量的权限。

5）与其他模式的关系

5.1）访问者模式是命令模式的加强版，其对象可对不同类的多种对象执行操作

5.2）使用访问者对整个组合模式执行操作

5.3）搭配迭代器模式：遍历复杂数据结构，对其中元素执行所需操作。

99-工厂模式比较

1）简单工厂模式

```
class UserFactory {
    public static function create($type) {
        switch ($type) {
            case 'user': return new User();
            case 'customer': return new Customer();
            case 'admin': return new Admin();
            default:
                throw new Exception('传递的用户类型错误。');
        }
    }
}
```

2）工厂方法模式

其在父类中提供一个创建对象的方法， 允许子类决定实例化对象的类型。

3）抽象工厂模式

它能创建一系列相关或相互依赖的对象， 而无需指定其具体类。

什么是 “系列对象”? 例如有这样一组的对象： `运输工具` + `引擎` + `控制器` 。 它可能会有几个变体：

1. `汽车` + `内燃机` + `方向盘`
2. `飞机` + `喷气式发动机` + `操纵杆`