



# 写给大家看的设计模式

c++11 design-pattern c++ pezy 8月17日发布

原文收录在[我的博客](#), 欢迎光临.

本文是针对 <https://github.com/kamranahmed> 的翻译与笔记, 会结合部分个人理解. 若您发现有明显理解有误的地方, 及疏漏之处, 麻烦留言指正, 在下不胜感激.

标题的解读: 设计模式与建构号称软工双雄, 在软件工程领域可谓智慧的结晶, 尤其是设计模式, 由于其高度抽象与最佳实践的特性, 导致初学者以及编程经验不足者, 读此如读天书. 所谓"给人读的", 就是将设计模式请下神坛, 用更容易理解的角度来介绍其精髓. 本人大学时期曾读过一本<大话设计模式>, 就走的通俗易懂之路, 然而, 通俗不能有失准确, 易懂不能理解偏差. 差之毫厘, 谬以千里. 闻者足戒.

Explain them in the simplest way possible. --- 作者的话

## 初窥门径

软工的江湖, 有一个原则贯穿始终, 有如剑道: DRY(don't repeat yourself). 无数先哲们, 想尽各种办法来解决这个终极问题. 所谓设计模式, 就是其中最著名的一个解决方案, 其作者有四位, 号称"东邪, 西毒, ---". 而这种办法, 早已不是一招一式, 不是什么特定的类, 库, 代码, 你没法 include, import 一下就坐享其成. 这些方法被称之为 guidelines, 如果直译的话, 就是指导方针. 听起来比较虚一点, 但它们的确是针对具体问题的.

这里引入 Wikipedia 的描述:

In software engineering, a software design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations.

这可能说的更加精确一些, 但意思就是上述那意思.

## 小心

- 设计模式**不是银弹**(废话, 根本没有银弹)
- 不要教条, 不要中二, 也不要强迫症**. 如果陷入以上三种状态, 请牢记: 设计模式是用来解决问题的, 而不是用来找茬的.
- 因地制宜**, 就是天使, 否则, 则是魔鬼.

原作者使用 PHP7 作为示例代码, 而恰好本人完全不会宇宙最好语言, 只好用老土的 C++ 来阐述.

## 设计模式的类型

- [创建型](#)
- [结构型](#)
- [行为型](#)

## 创建型设计模式

简言之:

创建型模式, 是针对如何**创建对象**的解决方案

Wikipedia:

In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic forms of object creation would usually be provided by the design. Creational design patterns solve this problem by

- [简单工厂](#)
- [工厂方法](#)
- [抽象工厂](#)
- [生成器](#)
- [原型](#)
- [单例](#)

## 🏠 简单工厂

真实案例:

造房子时需要一个门. 你是穿上木匠服开始在你家门口锯木头, 搞得一团糟; 还是从工厂里生产一个.

简言之:

简单工厂为用户提供了一个实例, 而隐藏了具体的实例化逻辑.

Wikipedia:

In object-oriented programming (OOP), a factory is an object for creating other objects – formally a factory is a function or method that returns objects of a varying prototype or class from some method call, which is assumed to be "new".

示例代码:

```
class WoodenDoor : public IDoor {
public:
    WoodenDoor(float width, float height): m_width(width), m_height(height){}
    float GetWidth() override { return m_width; }
    float GetHeight() override { return m_height; }

protected:
    float m_width;
    float m_height;
};

class DoorFactory {
public:
    static IDoor* MakeDoor(float width, float height)
    {
        return new WoodenDoor(width, height);
    }
};

int main()
{
    IDoor* door = DoorFactory::MakeDoor(100, 200);
    std::cout << "Width: " << door->GetWidth() << std::endl;
    std::cout << "Height: " << door->GetHeight() << std::endl;
}
```

使用时机:

当你创建一个对象, 并非简单拷贝赋值, 牵扯很多其他逻辑时, 就应该把它放到一个专门的工厂中, 而不是每次都重复. 这个体现在 C++ 中, 主要就是将 new 语句的逻辑抽象到一个单例, 或者如上述例子一样, 扔到一个统一的静态函数中去.

本质:

其实就是"抽象"在创建对象时的一个具体体现.

## 🏭 工厂方法

真实案例:

如果你主管招聘, 你肯定无法做到什么职位都由你一个人来面试. 根据具体的工作性质, 你需要选择并委托不同的人来按步骤进行面试.

Wikipedia:

In class-based programming, the factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor.

示例代码:

```
void takeInterview() {
    IInterviewer* interviewer = this->makeInterviewer();
    interviewer->askQuestions();
}

protected:
    virtual IInterviewer* makeInterviewer() = 0;
};

template <typename Interviewer>
class OtherManager : public HiringManager {
protected:
    IInterviewer* makeInterviewer() override {
        return new Interviewer();
    }
};

int main()
{
    HiringManager* devManager = new OtherManager<Developer>();
    devManager->takeInterview();

    HiringManager* marketingManager = new OtherManager<CommunityExecutive>();
    marketingManager->takeInterview();
}
```

使用时机:

当一个类中有一些通用的处理,但要等到运行时决定用哪个子类的实现.换句话说,当客户并不知道他需要哪一个子类时.

解释:

- 客户并不知道他需要哪一个子类? 他分明指定了委托人: 开发部主管和市场部主管!

要注意, 这里我们抽象的对象是面试这个过程, 作为用户, 它只知道 Interview 这个接口, 而不清楚这个 Interview 的派生关系. 也就是说作为招聘主管(用户), 他只是找来了开发主管, 市场主管. 然后他说: "面试一下". 他并不知道会面试些什么内容, 什么形式. 这些就是被封装的部分. 这就是上述"不知道需要哪一个子类"的具体解释.

- 这和简单工厂有啥区别?

平心而论, 这篇文章里, 简单工厂的例子并不恰当, 而简单工厂和工厂方法居然用了两个实例, 真是扰乱视听, 让人糊涂. 但即便如此, 还是可以看出, 两者最大的区别: 抽象的维度. 简单工厂的抽象, 是一维的, 它抽象的仅仅是所创建"类型"的接口; 而工厂方法的抽象, 是二维的, 它不仅抽象了所创建"类型"的接口, 而且抽象了"方法"的接口.

具体到例子里, 简单工厂实例中, 客户就是要一个门, 而不关心创建过程, 最后实际创造的是一个木门. 这个颇为讽刺, 如果客户要的是个铁门呢? 那就事与愿违了. 所以在这个例子里, 也是存在二维抽象的. 一是"门"这个类型的抽象, 二是"造门"这个方法的抽象. 简单工厂只做到了前者, 而没有给出后者的解决方案, 这才造成了客户可能吃了哑巴亏. 如果我们按照工厂方法的思路, 将门工厂造门这件事进行细分, 木门交给木门工厂, 铁门交给铁门工厂. 这就和工厂方法里的例子别无二致了. 客户需要先指定委托对象, 而不关心具体怎么造门:

```
DoorFactory* woodenDoorFactory = new WoodenDoorFactory();
woodenDoorFactory->MakeDoor(100, 200);

DoorFactory* ironDoorFactory = new IronDoorFactory();
ironDoorFactory->MakeDoor(100, 200);
```

这就成了工厂方法了. 所谓二维: "DoorFactory" 和 "MakeDoor", 前者是无论委托给谁, 反正它是一个门工厂; 后者是管它造什么门, 反正它会给我造出一个 100\*200 的门来. 客户不关心细节, 只关心结果. 这就使抽象的本质了. 第二个例子也是一样, "Interview" 和 "HiringManager" 是二维抽象, 客户要的是完成面试过程, 要的就是一个面试官. 至于找了谁来充当面试官, 进行的是什么样的面试. 最终都是看不到的. 最后抽象成的结果就是: "面试官完成了面试"这么件事.

真实案例:

接着简单工厂里的案例. 根据实际需要, 您可以从木门商店获得木门, 从铁门商店获得铁门, 从相关商店获得 PVC 门. 除此之外, 你还需要不同专业的人, 来帮你装门, 木门需要木匠, 铁门需要焊工. 可以看到, 门之间存在着一定的对应依赖关系. 木门-木匠, 铁门-焊工, 等等.

简言之:

工厂们的工厂, 一家管理各自独立却又互相依赖的一批工厂, 而不关心各自的细节.

Wikipedia:

The abstract factory pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes

示例代码:

```
        return new Door();
    }
    IDoorFittingExpert* MakeFittingExpert() override {
        return new DoorFittingExpert();
    }
};

int main()
{
    IDoorFactory* woodenFactory = new DoorFactory<WoodenDoor, Carpenter>();
    {
        IDoor* door = woodenFactory->MakeDoor();
        IDoorFittingExpert* expert = woodenFactory->MakeFittingExpert();
        door->GetDescription();
        expert->GetDescription();
    }

    IDoorFactory* ironFactory = new DoorFactory<IronDoor, Welder>();
    {
        IDoor* door = ironFactory->MakeDoor();
        IDoorFittingExpert* expert = ironFactory->MakeFittingExpert();
        door->GetDescription();
        expert->GetDescription();
    }
}
```

使用时机:

当遇到不那么简单的创建逻辑, 伴随着与之相关的依赖关系时.

本质:

依然可以用维度来理解抽象工厂. 抽象工厂比工厂方法又多了一维. 我们再把三个工厂理一遍: 简单工厂, 是针对一种"类型"的抽象; 工厂方法, 是针对一种"类型", 以及一种"创建方法"的抽象; 抽象工厂, 是针对一组"类型"与"创建方法"的抽象, 组内每一套类型与创建方法一一对应. 用造门这个例子来说: 简单工厂, 是封装了"造门"的操作, 输出的是一种门; 工厂方法, 是封装了"多种造门"的操作, 并委托"多家工厂", 输出的是"各种门". 抽象工厂, 是封装了"多种造门"的操作, "提供多种专业人员"的操作, 并委托给"多家工厂", 输出的是"各种门", 以及"各种专业人员", 且"门"与"专业人员"一一对应.

例子中, 抽象工厂提供了两套"类型 - 创建操作"(分别是"门 - 造门", "专业人员 - 提供专业人员"), 其实这个个数是无限的. 你可以提供 n 套这样的对应关系. 然后委托给相关的工厂. 这就是"工厂们的工厂"的具体含义.

## 生成器

真实案例:

假设你在哈迪斯(美国连锁快餐集团), 正想下单. 如果你说, 要一个"大哈迪", 他们很快就能交给你, 而不多问一句. 这是简单工厂的例子. 但, 当创建逻辑涉及更多步骤时, 譬如, 你在 Subway 买汉堡, 那么你可能需要做出更多选择, 想要哪种面包? 哪种酱汁? 哪种奶酪? 这种情况下, 就需要用到生成器模式了.

简言之:

允许你创建不同风格的对象, 同时避免构造器污染: 当对象有好几种口味的时候尤其有用. 或者是创建对象的过程涉及很多步骤时.

Wikipedia:

The builder pattern is an object creation software design pattern with the intentions of finding a solution to the telescoping constructor anti-pattern.

简单说下"the telescoping constructor anti-pattern"(可伸缩构造器的反模式)是什么. 你总会看到下面这种构造函数:

```
Burger(int size, bool cheese = true, bool peperoni = true, bool tomato = false, bool lettuce = true);
```

你应该已经察觉了, 构造函数的参数数量可能会迅速失控, 而且其参数安排会越来越难理解. 将来想要增加更多选项, 这个列表会一直增长下去. 这就被称为"the telescoping constructor anti-pattern"(可伸缩构造器的反模式).

示例代码:

```
int size_ = 7;
bool cheese_ = false;
bool peperoni_ = false;
bool lettuce_ = false;
bool tomato_ = false;
};

class Burger::BurgerBuilder {
public:
    BurgerBuilder(int size) { burger_ = new Burger(size); }
    BurgerBuilder& AddCheese() { burger_->cheese_ = true; return *this; }
    BurgerBuilder& AddPeperoni() { burger_->peperoni_ = true; return *this; }
    BurgerBuilder& AddLettuce() { burger_->lettuce_ = true; return *this; }
    BurgerBuilder& AddTomato() { burger_->tomato_ = true; return *this; }
    Burger* Build() { return burger_; }
private:
    Burger* burger_;
};

int main()
{
    Burger* burger = Burger::BurgerBuilder(14).AddPeperoni().AddLettuce().AddTomato().Build();
    burger->showFlavors();
}
```

上述代码与原文代码略有不同, 但表达的主旨, 以及最终用法完全一致. (额外添加了输出函数, 方便运行查看)

使用时机:

当对象拥有好几种口味, 且需要避免构造器伸缩时使用. 与工厂模式运用场景不同之处在于: 当创建过程仅仅一步到位, 使用工厂模式. 如果需要分步进行, 则考虑使用生成器模式.

本质:

生成器模式的本质, 就是将构造函数中的参数列表**方法化**. 长长的参数列表, 无论是面向对象还是函数式编程, 都是大忌. 该模式主要就是为了解决该问题. 函数式编程中对该问题的解决方式是: [柯里化](#), 其本质与生成器模式是一样的.

## 原型

真实案例:

还记得多利吗? 这个山羊是克隆的! 先不谈细节, 关键点都集中在于克隆上.

简言之:

通过克隆, 基于已有对象来创建对象.

Wikipedia:

The prototype pattern is a creational design pattern in software development. It is used when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects.

#### 示例代码:

```
#include <string>

class Sheep {
public:
    Sheep(const std::string& name, const std::string& category = "Mountain Sheep") : name_(name), category_(category) {}
    void SetName(const std::string& name) { name_ = name; }
    void ShowInfo() { std::cout << name_ << " : " << category_ << std::endl; }
private:
    std::string name_;
    std::string category_;
};

int main()
{
    Sheep jolly("Jolly");
    jolly.ShowInfo();

    Sheep dolly(jolly); // copy constructor
    dolly.SetName("Dolly");
    dolly.ShowInfo();

    Sheep doolly = jolly; // copy assignment
    doolly.SetName("Doolly");
    doolly.ShowInfo();
}
```

上述代码, 利用的是 **Sheep** 默认的拷贝构造和拷贝赋值函数, 当然也可重写这两个函数实现自定义操作.

#### 使用时机:

当需要的对象, 与已存在的对象非常相似, 或当创建过程比克隆一下更费时的时候.

#### 本质:

原型模型, 基本已经嵌入在各种语言实现里了. 其核心就是 Copy. 其实这个策略不局限于代码架构, 当你重装完操作系统, 并安装了必备软件, 一般会打包一份 ghost, 下次给别人重装的时候, 直接一键 ghost 即可, 如有特殊需要, 可以装好后再调整. 这个过程实际上也是原型模式. 再比如, 我们开发时总会先搭建环境(脚手架), 现在可以用 docker 来搭建了, 那么你用别人的 docker 包来搭建时, 其实也是原型模式. 这样理解, 就比较通俗了.

## 🕒 单例

#### 真实案例:

所谓一山不容二虎, 一国不可两君. 遇到大事, 总应该由同一位老大来处理. 那么这位老大就是单例.

#### 简言之:

确保一个特定类的一个对象, 只能创建一次.

#### Wikipedia:

In software engineering, the singleton pattern is a software design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system.

实际上, 单利模式常被认为是反模式, 要避免过度使用. 它本质上类似全局变量, 可能会造成耦合度过高的问题, 也可能给调试带来困难. 故而一定要谨慎使用.

#### 示例代码:

创造单例, 要确保构造函数私有化, 拷贝构造, 拷贝赋值应该禁用. 创建一个静态变量来存此单例.

```
#include <iostream>
#include <string>
#include <cassert>

class President {
public:
    static President& GetInstance() {
```

```

    }

    President(const President&) = delete;
    President& operator=(const President&) = delete;

private:
    President() {}
};

int main()
{
    const President& president1 = President::GetInstance();
    const President& president2 = President::GetInstance();

    assert(&president1 == &president2); // same address, point to same object.
}

```

## 结构型设计模式

简言之:

结构型模式重点关注对象组合, 换句话说, 实体如何互相调用. 还有另一个解释: 对"如何构建一个软件组件?"问题的回答.

Wikipedia:

In software engineering, structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities.

- [适配器](#)
- [桥接](#)
- [组成](#)
- [装饰](#)
- [外观](#)
- [享元](#)
- [代理](#)

### 适配器

真实案例:

三个例子: 1) 你从相机存储卡传照片给电脑, 需要与此兼容的适配器, 来保证连接. 2) 电源适配器, 三脚插头转两脚插头. 3) 翻译, 看好莱坞大片, 将英文字幕转为中文.

简言之:

适配器模式, 包装一个对象, 让本不兼容其他类的该对象变得兼容.

Wikipedia:

In software engineering, the adapter pattern is a software design pattern that allows the interface of an existing class to be used as another interface. It is often used to make existing classes work with others without modifying their source code.

示例代码:

```

#include <iostream>

class ILion {
public:
    virtual void Roar() {
        std::cout << "I am a Lion" << std::endl;
    }
};

```

```
        lion.Roar();
    }
};

class WildDog
{
public:
    void Bark() {
        std::cout << "I am a wild dog." << std::endl;
    }
};

// now we added a new class `WildDog` the hunter can hunt it also
```

本质:

软工领域, 很多问题都可以用在**中间加一层**的方式来解决. 适配器模式, 是这种策略的典型应用之一.

## 🌉 桥接



问答



头条



专栏



讲堂



更多

真实案例:

如果你有一个网站, 有着很多不同种类的页面. 而此刻你有一个功能是允许用户来改变主题样式. 该怎么做? 为每个不同页面都创建多个副本? 还是创建单独的主题, 并根据用户偏好加载它们? 桥接模式允许你实现第二种方案.

一图胜千言:



简言之:

桥接模式, 优先考虑组合而非继承. 将实现细节从层次结构中, 剥离并独立成另一套层次结构.

Wikipedia:

The bridge pattern is a design pattern used in software engineering that is meant to "decouple an abstraction from its implementation so that the two can vary independently"

示例代码:

```
#include <iostream>
#include <string>

class ITheme {
public:
    virtual std::string GetColor() = 0;
};

class DarkTheme : public ITheme {
public:
    std::string GetColor() override { return "Dark Black"; }
};

class LightTheme : public ITheme {
public:
    std::string GetColor() override { return "Off white"; }
};

class AquaTheme : public ITheme {
public:
    std::string GetColor() override { return "Light blue"; }
};

class IWebPage {
public:
```

本质:



"Theme"就是剥离出来的抽象概念. 玻璃之前, 页面一大堆, 但是有规律: 很多页面, 只是样式不同, 但内容相同. 怎么做? DRY 原则告诉我们, 相同的东西保留一份, 将不同的东西抽象出来. 这也与 "高内聚, 低耦合" 这个最终目的一脉相承. 抽出 Theme 之后, 剩下的 Page 部分, 内聚就比较高了, 也可以称之为: 更加纯粹了, 只负责页面内容.

还要注意到适配器和桥接模式的相同点, 它们都存在一个强耦合的**关联**(UML)关系. 譬如 `WildDogAdapter` 就必然包含一个 `WildDog` (**组合**(UML)关系). `IWebPage` 就必然包含一个 `ITheme` (也是**组合**(UML)关系). 而从这个角度来看, 你发现桥接里, 这种关系发生在接口层面, 而适配器, 只是简单的发生在两个类层面. 这就好比简单工厂与工厂方法的关系. 是维度的上升, 而多出的这个维度, 就是桥接模式中"层次"的体现.

## 🧩 组成

真实案例:

每个公司都是由员工组成的. 每个员工, 有相同点: 如都有薪水, 都需要负责, 都可能上级, 也都可能有下属.

简言之:

组合模式让客户以统一的方式对待各个独立的对象.

Wikipedia:

In software engineering, the composite pattern is a partitioning design pattern. The composite pattern describes that a group of objects is to be treated in the same way as a single instance of an object. The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies. Implementing the composite pattern lets clients treat individual objects and compositions uniformly.

示例代码:

```
#include <iostream>
#include <string>
#include <vector>

class Employee {
public:
    Employee(const std::string& name, float salary): name_(name), salary_(salary) {}
    virtual std::string GetName() { return name_; }
    virtual float GetSalary() { return salary_; }

protected:
    float salary_;
    std::string name_;
};

class Developer : public Employee {
public:
    Developer(const std::string& name, float salary) : Employee(name, salary) {}
};

class Designer : public Employee {
public:
    Designer(const std::string& name, float salary) : Employee(name, salary) {}
};
```

本质:

组合模式在我看来甚至称不上什么模式, 其核心就是**多态**特性的体现. 另一个核心在于, 容器存储的是接口类型, 利用多态, 可以迭代处理通用操作.

## 🎨 装饰

真实案例:

假设你经营一家提供多种服务的汽车服务店. 你如何来计算收费账单? 通常会选择一项服务的同时, 动态更新服务的总价. 这里, 每一种服务都是装饰器.

简言之:

装饰模式将对象包装在装饰类对象中, 从而在运行时动态改变该对象的行为.

In object-oriented programming, the decorator pattern is a design pattern that allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class. The decorator pattern is often useful for adhering to the Single Responsibility Principle, as it allows functionality to be divided between classes with unique areas of concern.

#### 示例代码:

```
#include <iostream>
#include <string>

class ICoffee {
public:
    virtual float GetCost() = 0;
    virtual std::string GetDescription() = 0;
};

class SimpleCoffee : public ICoffee {
public:
    float GetCost() override { return 10; }
    std::string GetDescription() override { return "Simple coffee"; }
};

class CoffeePlus : public ICoffee {
public:
    CoffeePlus(ICoffee& coffee): coffee_(coffee) {}
    virtual float GetCost() = 0;
    virtual std::string GetDescription() = 0;
protected:
    ICoffee& coffee_;
};

class MilkCoffee : public CoffeePlus {
```

#### 本质:

装饰模式的形态很有意思,像是静态语言的动态化.其实实现上与组合模式类似,但关键之处在于,其**依赖的对象是其父类接口**.顾名思义,装饰模式,装饰的是对象自身,且支持重复装饰.譬如上述实例中,完全可以加两遍牛奶.但最终要保证接口的一致性,就像你的房子无论装饰成什么样子,它依然只是你的房子.

## 外观

#### 真实案例:

如何开机?你会说"按一下电源键".你会有这样的反应,是因为计算机提供了一个超级简单的接口,而隐藏了一系列复杂的开机操作所致.这个简单接口,对于复杂操作来说,就是外观.

#### 简言之:

外观模式为复杂的子系统提供了一个简单接口.

#### Wikipedia:

A facade is an object that provides a simplified interface to a larger body of code, such as a class library.

#### 示例代码:

```
#include <iostream>

class Computer {
public:
    void GetElectricShock() { std::cout << "Ouch!" << std::endl; }
    void MakeSound() { std::cout << "Beep beep!" << std::endl; }
    void ShowLoadingScreen() { std::cout << "Loading..." << std::endl; }
    void Bam() { std::cout << "Ready to be used!" << std::endl; }
    void CloseEverything() { std::cout << "Bup bup bup bzzzz!" << std::endl; }
    void Sooth() { std::cout << "Zzzzz" << std::endl; }
    void PullCurrent() { std::cout << "Haaah!" << std::endl; }
};
```

```
ComputerFacade(Computer& computer): computer_(computer) {}
void TurnOn() {
    computer_.GetElectricShock();
    computer_.MakeSound();
    computer_.ShowLoadingScreen();
    computer_.Bam();
}
void TurnOff() {
    computer_.CloseEverything();
}
```

本质:

同样很难称之为模式的模式。用的依然是“多加一层”的思想, 通过封装的方式来实现。多的这一层, 就是所谓的“外观”了。

## 享元

真实案例:

你有没有在摊位边喝现泡茶的经验? 他们经常除了你要求的这一杯外, 还额外沏更多的茶, 留给其他的潜在客户。以此来节省资源, 包括热气, 火候等。享元模式就是针对这一特点的: 共享。

简言之:

通常以最小的存储用量或计算成本为代价, 共享给尽可能多的相似对象。

Wikipedia:

In computer programming, flyweight is a software design pattern. A flyweight is an object that minimizes memory use by sharing as much data as possible with other similar objects; it is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory.

示例代码:

```
#include <iostream>
#include <string>
#include <unordered_map>

// Anything that will be cached is flyweight.
// Types of tea here will be flyweights.
class KarakTea {};

class TeaMaker {
public:
    KarakTea* Make(const std::string& preference) {
        if (availableTea_.find(preference) == availableTea_.end())
            availableTea_.insert({preference, new KarakTea()});

        return availableTea_.at(preference);
    }

private:
    std::unordered_map<std::string, KarakTea*> availableTea_;
};

class TeaShop {
public:
    TeaShop(TeaMaker& teaMaker): teaMaker_(teaMaker) {}
    void TakeOrder(const std::string& teaType, int table) {
```

本质:

享元模式的本质就是最基本的缓存思想, 无论是计算机体系结构中的 cache 还是操作系统中的 page table, 都是这种思想的体现。在程序设计中, 实现这一思想, 最常用的数据结构就是哈希表。如例子中所示。其最简单的用法描述就是: key 存在了, 直接取走; 不存在, 创建一个。以此节省重复的创建与冗余的空间。

## 代理

真实案例:

你应该用过门禁卡开门吧? 其实有很多种方式来开门, 如用门禁卡, 或是输入安全密码等等。门的主要功能本来只是“开”, 而现在门禁系统就像是加之于门的代理, 使之拥有

简言之:

使用代理模式, 一个类会表现出另一个类的功能.

Wikipedia:

A proxy, in its most general form, is a class functioning as an interface to something else. A proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes. Use of the proxy can simply be forwarding to the real object, or can provide additional logic. In the proxy extra functionality can be provided, for example caching when operations on the real object are resource intensive, or checking preconditions before operations on the real object are invoked.

示例代码:

```
#include <iostream>
#include <string>

class IDoor {
public:
    virtual void Open() = 0;
    virtual void Close() = 0;
};

class LabDoor : public IDoor {
public:
    void Open() override { std::cout << "Opening lab door" << std::endl; }
    void Close() override { std::cout << "Closing the lab door" << std::endl; }
};

class Security {
public:
    Security(IDoor& door): door_(door) {}
    bool Authenticate(const std::string& password) { return password == "$ecret"; }
    void Open(const std::string& password) {
        if (Authenticate(password)) door_.Open();
        else std::cout << "Big no! It ain't possible." << std::endl;
    }
    void Close() { door_.Close(); }
};
```

另一个例子是一些数据映射的实现. 例如, 我最近为 MongoDB 做了一个 ODM (对象数据映射), 使用的就是这个模式: 我用魔术方法 `_call()` 给 MongoDB 的类加了一个代理. 所有方法通过代理, 实际都会调用 MongoDB 类的方法, 并返回其调用结果. 但我增加了两个特例: `find` 和 `findOne` 方法查询数据时, 会被映射到相应的对象, 并返回该对象. 而不再是返回 `Cursor` 了.

本质:

依然体现了加一层的思想, 这与上面遇到的适配器模式和外观模式都很类似. 这里我们先比较一下三者的差别: 适配器的目的很明确, 是为了适应已有接口, 出发点是兼容; 外观模式的目的是简化繁琐的接口, 出发点是封装; 而代理模式的目的是增加更多功能, 出发点也是兼容. 但代理模式的兼容, 与适配器有很大差别, 适配器是真的从接口上兼容, 继承同样的接口类, 实现父类的虚方法; 代理模式则不然, 它的兼容, 更像是一种伪装, 只是接口的名称保持一致, 但实际上并无多大关联. 譬如例子里, 以前你用门, 有 `Open` 和 `Close` 方法, 现在换成安全门了, 你依然习惯性的想用这两种方法. 然而安全门只是门的代理, 所以它的这两种同名方法, 其实是伪造给你看的, 与之前的方法并无接口上的兼容性.

代理模式广泛使用在 API 设计中, 其核心是为了兼容用户习惯.

## 行为型设计模式

简言之:

关注对象间的责任分配. 它们与结构模式最大的不同在于: 它们不仅仅指定结构, 还概述了结构之间消息传递/通讯的模式. 换句话说, 它们回答了"软件组件们的行为是如何运转的"这个问题.

Wikipadia:

In software engineering, behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

- [迭代器](#)
- [中介者](#)
- [备忘录](#)
- [观察者](#)
- [访问者](#)
- [策略](#)
- [状态](#)
- [模板方法](#)

## 责任链

真实案例:

假设, 您的账户里有三种付款方式可供选择(A, B 和 C). 但额度各不一样, A 有 100\$, B 有 300\$, C 有 1000\$. 支付优先级顺序是从 A 到 C. 当您尝试购买价格为 210\$ 的东西时, 用责任链来处理, 会先去看 A 方式可否搞定, 如果搞不定, 就再去用 B, 如果依然搞不定, 再去用 C. 直到找到合适的方式. 这里的 A, B 和 C 构成的链条, 以及这样的现象就是责任链.

简言之:

它有助于建立一条对象链. 请求会从一端开始, 依次访问对象, 直到找到合适的处理程序.

Wikipedia:

In object-oriented design, the chain-of-responsibility pattern is a design pattern consisting of a source of command objects and a series of processing objects. Each processing object contains logic that defines the types of command objects that it can handle; the rest are passed to the next processing object in the chain.

示例代码:

```
#include <iostream>
#include <exception>
#include <string>

class Account {
public:
    Account(float balance) : balance_(balance) {}
    virtual std::string GetClassName() { return "Account"; }
    void SetNext(Account* const account) { successor_ = account; }
    bool CanPay(float amount) { return balance_ >= amount; }
    void Pay(float amountToPay) {
        if (CanPay(amountToPay)) {
            std::cout << "Paid " << amountToPay << " using " << GetClassName() << std::endl;
        } else if (successor_) {
            std::cout << "Cannot pay using " << GetClassName() << ". Proceeding..." << std::endl;
            successor_>Pay(amountToPay);
        } else {
            throw "None of the accounts have enough balance.";
        }
    }
protected:
    Account* successor_ = nullptr;
    float balance_;
};
```

本质:

责任链的本质其实是对象的单链表实现 + 对单链表的迭代. 上述例子中, 我们的迭代其实是接近递归的形式. 如果我们将整个责任链以统一的借口( Account )存储在 list 或 vector 中, 然后用 for 循环来迭代访问. 也同样可称为责任链. 所以不要被名词吓唬住, 始终回归到最基本的思想, 以及最基本的数据结构与逻辑手段. 这样才可以灵活应用.

## 命令

真实案例:

一个典型的例子是, 在饭馆点菜. 你(客户)要求服务员(调用者)上一些食物(命令). 然后服务员将命令简短的传达给厨师(接收者). 厨师拥有做饭的必要知识与技能. 另一个例

简言之:

允许你将操作封装在对象中. 这种模式背后的核心思想是分离客户与接收者.

Wikipadia:

In object-oriented programming, the command pattern is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time. This information includes the method name, the object that owns the method and values for the method parameters.

示例代码:

```
#include <iostream>

class Bulb {
public:
    void TurnOn() { std::cout << "Bulb has been lit" << std::endl; }
    void TurnOff() { std::cout << "Darkness!" << std::endl; }
};

class ICommand {
public:
    virtual void Execute() = 0;
    virtual void Undo() = 0;
    virtual void Redo() = 0;
};

class TurnOn : public ICommand {
public:
    TurnOn(Bulb& bulb): bulb_(bulb) {}
    void Execute() override { bulb_.TurnOn(); }
    void Undo() override { bulb_.TurnOff(); }
    void Redo() override { Execute(); }
private:
    Bulb& bulb_;
};
```

命令模式通常被用来实现交易基础系统, 当你执行一系列命令的同时维系一个历史记录. 如果最终命令执行成功就罢了, 若不成功, 将通过历史回溯, 来撤销这一系列命令.(这更像是一个原子命令的执行过程, 请见[ACID](#))

本质:

命令模式本质上, 是对消息协议的一种抽象, 而用的手法是回调. 就像举出的例子里, 提出要求的是客户, 执行要求的是厨师, 而连接两者的是服务员, 但真正连接者是消息的接口, 这个就是命令. 同样的, 电视机遥控器也是命令. 这个命令一定具备"简单", "高粒度"等特点. 通常在运用的时候, 会加上对这个命令的历史记录与回溯. 也就是示例代码中的那三个基本接口: `execute`, `undo`, `redo`. 值得注意的是, `execute` 一定是 ACID 的, 通常会包含一系列命令的集合. 一旦有一个命令执行失败, 那么会整体回滚.

## ☞ 迭代器

真实案例:

老式的收音机是迭代器的一个好例子, 用户可以从任意频道开始, 通过点击"下一个"或"上一个"按钮, 来浏览响应的频道. 也可以用 MP3 播放器和电视机来举例, 你同样可以通过"向前"和"向后"按钮来连续切换频道. 换句话说, 它们都提供了一个接口来遍历各个频道, 歌曲或电台.

简言之:

它呈现了一种访问对象元素, 却不暴露底层方法的方式.

Wikipadia:

In object-oriented programming, the iterator pattern is a design pattern in which an iterator is used to traverse a container and access the container's elements. The iterator pattern decouples algorithms from containers; in some cases, algorithms are necessarily container-specific and thus cannot be decoupled.

示例代码:

```
#include <algorithm>

class RadioStation {
    friend bool operator==(const RadioStation& lhs, const RadioStation& rhs) { return lhs.frequency_ == rhs.frequency_; }
public:
    RadioStation(float frequency): frequency_(frequency) {}
    float GetFrequency() const { return frequency_; }
private:
    float frequency_;
};

class StationList {
    using Iter = std::vector<RadioStation>::iterator;
public:
    void AddStation(const RadioStation& station) { stations_.push_back(station); }
    void RemoveStation(const RadioStation& toRemove) {
        auto found = std::find(stations_.begin(), stations_.end(), toRemove);
        if (found != stations_.end()) stations_.erase(found);
    }
    Iter begin() { return stations_.begin(); }
    Iter end() { return stations_.end(); }
private:
    std::vector<RadioStation> stations_;
};
```

本质:

同样称不上模式的模式. 这里的迭代器与 C++ 中迭代器的概念完全相同. 我觉得是否将语言中本有的容器包装成对象, 要适当取舍. 切莫为了设计而设计. 就示例代码而言, `StationList` 对象完全多此一举, 裸用容器就可以解决. 在实际应用中, 除非为了接口上的统一, 而使用一个代理( `StationList` 就是一个代理类), 否则完全不用过度设计. 迭代器的本质, 就是迭代. 这是程序语言最基础的一环, 所谓迭代器模式, 仅仅是这一环在面向对象中的体现.

## 🧐 中介者

真实案例:

典型案例是, 你通过手机给人打电话, 你和对方的通讯并非直接送达的, 而是需要通过中间的网络运营商. 这个案例中, 网络运营商就是中介者.

简言之:

中介者模式增加了一个第三方对象(中介者)来控制两个对象(同事)间的交互. 有助于对彼此通信的解耦, 毕竟他们并不需要关心对方的实现细节.

Wikipedia:

In software engineering, the mediator pattern defines an object that encapsulates how a set of objects interact. This pattern is considered to be a behavioral pattern due to the way it can alter the program's running behavior.

示例代码:

```
#include <iostream>
#include <string>
#include <ctime>
#include <iomanip>

class User;

class IChatRoomMediator {
public:
    virtual void ShowMessage(const User& user, const std::string& message) = 0;
};

class ChatRoom : public IChatRoomMediator {
public:
    void ShowMessage(const User& user, const std::string& message) override;
};

class User {
public:
    User(const std::string& name, IChatRoomMediator& chatMediator): name_(name), chatMediator_(chatMediator) {}
    const std::string& GetName() const { return name_; }
    void Send(const std::string& message) { chatMediator_.ShowMessage(*this, message); }
private:
    std::string name_;
```

本质:

中介者的本质, 也是对**消息协议的一种抽象**, 并同样借用了**回调**的手段. 听起来和**命令**模式很类似, 其实也的确很类似. 区别仅在于业务场景的适应上: 命令模式适用于**多种**消息协议, 并将每一种都封装成对象, 是一种平铺式的抽象方式, 你通过多种命令, 来实现多种通讯; 中介者模式呢, 则固定为**某一种**消息协议, 同事之间要遵循相同的协议, 才可以做到通讯. 而这一种, 可以通过继承来具化, 所以是一种纵深式的抽象, 你和对方通讯, 可以是电话, 邮件, 微信等等, 但他们扮演的都是中介者这个角色.

📖 备忘录

真实案例:

以计算器(原发器)为例, 当你做了一组运算后, 最后一次计算过程会保存在内存(备忘录)中. 所以你随时可以通过某个按钮(负责人)来恢复该操作.

简言之:

备忘录模式会抓取并储存对象的当前状态, 之后可便捷的恢复出来.

Wikipedia:

The **memento pattern** is a software design pattern that provides the ability to restore an object to its previous state (undo via rollback).

当你需要快照-恢复这样类似功能时, 该模式将会非常有用.

示例代码:

```
#include <iostream>
#include <string>
#include <memory>

class EditorMemento {
public:
    EditorMemento(const std::string& content): content_(content) {}
    const std::string& GetContent() const { return content_; }
private:
    std::string content_;
};

class Editor {
    using MementoType = std::shared_ptr<EditorMemento>;
public:
    void Type(const std::string& words) { content_ += " " + words; }
    const std::string& GetContent() const { return content_; }
    MementoType Save() { return std::make_shared<EditorMemento>(content_); }
    void Restore(MementoType memento) { content_ = memento->GetContent(); }
private:
    std::string content_;
};

int main()
{
}
```

本质:

备忘录模式, 说白了就是对**缓存**这个行为的对象化. 我们通常想缓存一个状态, 会把这个状态存到容器中(如哈希表, map等), 并用某个 key (如时间戳) 来标定. 但当你要实现, 是一个颇具规模, 多种状态缓存, 以及各种"存-取"穿插的时候, 将状态抽象成一个对象, 就显得十分必要了. 磨刀不误砍柴工, 在特定的场景下, 该模式不是没事找事, 而是真的会简化业务逻辑.

☺ 观察者

真实案例:

一个好例子: 求职者订阅了某职位发布网站, 当有何时的职位出现时, 他们会收到通知.

简言之:

当有对象的状态发生改变时, 所有依赖于该对象的状态的其它对象, 都会收到通知并随即改变状态.



Wikipedia:

The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

示例代码:

```
#include <iostream>
#include <string>
#include <vector>
#include <functional>

class JobPost {
public:
    JobPost(const std::string& title): title_(title) {}
    const std::string& GetTitle() const { return title_; }
private:
    std::string title_;
};

class IObserver {
public:
    virtual void OnJobPosted(const JobPost& job) = 0;
};

class JobSeeker : public IObserver {
public:
    JobSeeker(const std::string& name): name_(name) {}
    void OnJobPosted(const JobPost &job) override {
        std::cout << "Hi " << name_ << "! New job posted: " << job.GetTitle() << std::endl;
    }
private:
    std::string name_;
};
```

本质:

又被称为发布-订阅模式. 但无论叫什么, 其实本质都是**注入+回调**. 订阅是注入的时机, 发布是回调的时机. 观察是注入的时机, 通知是回调的时机. 在实践中, 通常会维护一个订阅列表, 有点类似邮件列表. 发布通知时, 会迭代每一个注入对象, 并执行回调.

## 访问者

真实案例:

假设有人要去迪拜, 他只需要一种方式(例如签证)进入迪拜. 到了之后, 就可以去访问迪拜的任何地方, 而用不着额外申请许可, 或是做一些法律事宜. 只需要让他知道一个地方, 他就可以去访问了. 访问者模式可以做到这一点, 它帮助你添加地点, 以便你无需额外工作就可以尽可能的访问更多地方.

简言之:

访问者模式允许你为对象们增加更多的操作, 却不必修改它们.

Wikipedia:

In object-oriented programming and software engineering, the visitor design pattern is a way of separating an algorithm from an object structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures. It is one way to follow the open/closed principle.

示例代码:

```
#include <iostream>

class AnimalOperation;

// visitee
class Animal {
public:
    virtual void Accept(AnimalOperation& operation) = 0;
};
```

```

class Dolphin;

// visitor
class AnimalOperation {
public:
    virtual void visitMonkey(Monkey& monkey) = 0;
    virtual void visitLion(Lion& lion) = 0;
    virtual void visitDolphin(Dolphin& dolphin) = 0;
};

class Monkey : public Animal {
public:
    void shout() { std::cout << "Oh oo oo oo!" << std::endl; }
};

```

本质:

其实本质依然是**注入-回调**模式. 签证是一种注入, 允许你回调 `visit`; 告诉地点是一种注入, 允许去具体地点回调 `visit`. 从示例代码看, 也能看出两层注入回调的意思: 首先是针对**接口**的注入回调, 通过 `Accept` 注入动物行为, 然后回调各个 `visit` 方法, 在回调的同时, 又将自身注入(此刻已经是针对**具体**对象了), 然后再回调具体的动物行为方法.

为什么要这样绕来绕去, 来来回回的呢? 那是因为访问者是一种**维护**模式. 试想, 既有代码已经存在 `Animal` 和其三个派生类了, 以及各自嚎叫的方法. 现在我想用一个**统一的接口, 去迭代的调用这些方法**(假想三个对象都在一个 `vector` 中, 你迭代的时候无法用各自不同的接口). 那么就需要访问者上场了. 首先在 `Animal` 类中增加接口 `Accept`, 留出注入的口子. 然后派生类重写该接口, 并借此将自身注入. 最后将这些方法抽象成一个接口类, 并增加相应的 `visit` 方法. 派生该接口类, 将具体的方法——绑定(就是在绑定回调). 然后我们一旦调用 `Accept`, 各自嚎叫的方法就会自然被回调到了.

综上, 访问者模式, 是一种对**调用**的抽象, 依靠**回调**来实现.

## 💡 策略

真实案例:

以排序算法为例, 最初我们采用冒泡排序, 但随着数据数量的增长, 冒泡排序越来越慢. 为了解决这个问题, 我们改用快速排序. 但虽然对于大型数据集来说效果好了起来, 但对于比较小的数据集而言, 却相当慢. 为了处理这样一个两难, 我们采取了一个策略: 对于小型数据集, 采用冒泡排序; 对于较大一些的, 采用快速排序.

简言之:

策略模式允许你根据实际情况切换算法或策略.

Wikipedia:

In computer programming, the strategy pattern (also known as the policy pattern) is a behavioural software design pattern that enables an algorithm's behavior to be selected at runtime.

示例代码:

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <memory>

class ISortStrategy {
public:
    virtual void Sort(std::vector<int>& vec) = 0;
};

class BubbleSortStrategy : public ISortStrategy {
public:
    void Sort(std::vector<int>& vec) override {
        std::cout << "Sorting using bubble sort" << std::endl;
        _BubbleSort(vec);
    }
private:
    void _BubbleSort(std::vector<int>& vec) {
        using size = std::vector<int>::size_type;
        for (size i = 0; i != vec.size(); ++i)
            for (size j = 0; j != vec.size()-1; ++j)
                if (vec[j] > vec[j+1])
                    std::swap(vec[j], vec[j+1]);
    }
};

```

策略模式的本质依然是**注入+多态**, 将接口注入, 调用相应方法(算法或策略)时, 再根据多态的特性来选择具体实现.

## ☞ 状态

真实案例:

假设你正在使用“画图”程序, 选择了画笔工具来进行绘制. 画刷会根据你选择的颜色而改变其行为: 譬如你选择了红色, 它便会用红色来绘制; 如果选择了蓝色, 它将成为蓝色.

简言之:

它让你在状态改变的同时, 也改变类的行为

Wikipedia:

The state pattern is a behavioral software design pattern that implements a state machine in an object-oriented way. With the state pattern, a state machine is implemented by implementing each individual state as a derived class of the state pattern interface, and implementing state transitions by invoking methods defined by the pattern's superclass. The state pattern can be interpreted as a strategy pattern which is able to switch the current strategy through invocations of methods defined in the pattern's interface.

示例代码:

以一个文字处理程序为例, 那些敲上去的字, 你可以改变它们的状态. 例如, 你选择了加粗, 后续的字都会是粗的, 同样的, 选择了斜体, 后续都会是斜体.

```
#include <iostream>
#include <string>
#include <algorithm>
#include <memory>

class IWritingState {
public:
    virtual void Write(std::string words) = 0;
};

class UpperCase : public IWritingState {
    void Write(std::string words) override {
        std::transform(words.begin(), words.end(), words.begin(), ::toupper);
        std::cout << words << std::endl;
    }
};

class LowerCase : public IWritingState {
    void Write(std::string words) override {
        std::transform(words.begin(), words.end(), words.begin(), ::tolower);
        std::cout << words << std::endl;
    }
};

class Default : public IWritingState {
```

本质:

状态模式的本质依然是**注入+多态**, 这和**策略**如出一辙. 实际上在实践中, 这两模式几乎一样. 只有一些微小的差别:

1. 状态模式通常会缓存当前状态, 你可以通过 `get` 方法取得状态, 但策略模式通常不提供 `get` 方法.
2. 状态模式会提供 `set` 方法替换状态, 但策略模式通常不提供 `set` 方法 (虽然可以用 assign constructor 起到同样效果)
3. 策略对象通常作为参数传递给当前对象, 而状态通常由当前对象所创建.
4. 策略是针对特定方法的, 而状态却是针对整个对象的.

详见<https://stackoverflow.com/que...>

总体来讲, 状态更像是一组策略的集合. 改变对象状态, 会让对象的各种方法都有改变. 而策略往往只是针对某特定算法的.

## ☞ 模板方法

假设我们正在造房子, 其步骤看起来可能如下所示:

1. 建造地基
2. 砌墙
3. 建造屋顶
4. 隔出楼层

这些步骤的顺序是固定的, 即在砌墙之前不能建造屋顶, 但每个步骤都可以修改完善, 譬如砌墙也可以由木头或聚酯, 石头来替代.

简言之:

模板方法定义了如何执行某种算法的框架, 但将具体实现延迟到子类.

Wikipedia:

In software engineering, the template method pattern is a behavioral design pattern that defines the program skeleton of an algorithm in an operation, deferring some steps to subclasses. It lets one redefine certain steps of an algorithm without changing the algorithm's structure.

示例代码:

假设我们有一个构建工具来帮助我们测试, 检查, 构建, 生成构建报告(即代码覆盖报告, 检查结果报告等), 并将我们的应用部署到测试服务器上.

```
#include <iostream>

class Builder {
public:
    void Build() {
        Test();
        Lint();
        Assemble();
        Deploy();
    }
protected:
    virtual void Test() = 0;
    virtual void Lint() = 0;
    virtual void Assemble() = 0;
    virtual void Deploy() = 0;
};

class AndroidBuilder : public Builder {
    void Test() override { std::cout << "Running android tests" << std::endl; }
    void Lint() override { std::cout << "Linting the android code" << std::endl; }
    void Assemble() override { std::cout << "Assembling the android build" << std::endl; }
    void Deploy() override { std::cout << "Deploying android build to server" << std::endl; }
};


class IOSBuilder : public Builder {
```

本质:

模板方法基本就是**多态**的集中体现. 只不过将所有多态方法集中到一个公共接口中. 不过模板方法的核心是, 通过这个统一接口, 确定各个具体接口方法的顺序, 以确立调用结构. 子类各自实现具体细节, 但行为, 结构依旧保持一致. 这就好比"C++ 标准"规定了语言的行为, 各家编译器去各自实现. 而最终, 只要你的代码遵循 C++ 标准, 原则上应该可以在各种编译器上得到一致的结果.

## 最后的话

这篇笔记, 或称之为翻译, 断断续续写完, 竟然花了半个月的时间(业余时间). 但我对于网络上热门资源的一次精读试验. 毕竟有太多的资源, 被 star 或收藏以后就被忘之脑后, 并将积灰多年.

突然想重读设计模式, 源于我想重构, 对于原作者所使用的[工厂方法实现](#)中, 有一段 switch 语句, 看着甚是碍眼. 试想未来我试验 OpenCV 算法, 会不断加入更多的算法, 每一次都需要在多个地方增加不同的代码, 想想就头疼, 自己维护都嫌麻烦. 这才想到去了解一下通过 C++ 语言, 可以怎样更优雅的实现工厂方法.

在了解过程中, 又对自己是否真正理解了几个工厂产生了怀疑, 于是找到了这篇 star 过万的热文.

实际上, 这篇文章介绍的 23 种设计模式, 与[四人帮那本书](#)完全对应. 唯一不同在于行为型设计模式那里, 顺序不太一致. 🤔 所以你也可以把这篇文章视为"精简版"的<设计模式 -

但在精读的过程中, 还是发现了不少问题的, 尤其是针对其示例代码, 其实有一些例子举得并不恰当(上面笔记有提及). 而且明显有强关联的一些模式之间, 却各举各的例子(如三个工厂). 对于稍微复杂的模式, 如(观察者, 访问者), 写的又过于肤浅, 例子浮于表面. 感觉重点不够分明, 适合推广普及, 却不耐咀嚼. 建议配合四人帮那本书一起参阅. (不得不说, 那本书里面的例子, 质量相当高)

最后划一下重点:

说起来 23 种设计模式, 很吓人的样子, 其实应该重点掌握的只有几个而已:

- 创建型模式中, 重点掌握**三个工厂, 与单例**. 简单工厂实际上算不上模式(书里都没单独列出), 关键就是工厂方法, 抽象工厂只不过是工厂的工厂而已. 那么工厂方法其实抽象的是 **new** 的过程, 那么关键就是 **new** 的两边都应该是抽象接口, 封装掉具体实现. 单例模式, 要先了解其弊端, 但那也是它最核心的特点: 全局性. 这玩意会在实际开发中大量使用, 但理想情况下应该是避免的. 剩下俩: 原型就是 copy, 生成器就是"拆迁构造器".
- 结构型模式中, 重点掌握**桥接**, 这个模式是重构神器. 如果领导说, 这个类太复杂了, 把它"抽一抽", "单拎出来". 那么十有八九是运用桥接模式. 结构型其实可以按以下基本特性分个类:
  - **加一层**: 适配器, 外观, 代理
  - **减肥**: 桥接
  - 常见编程技巧: 组合(就是基于接口迭代), 享元(哈希表), 装饰(不断实例化修改父类成员)
- 行为型模式中, 重点掌握**观察者和访问者**. 行为型模式都有一个共同的特点, 就是非常讲究**注入**和**回调**这两个手段. 尤其以观察者与访问者, 用的登峰造极. 前者是所有消息系统的灵魂, 重点掌握消息列表, 与通知的时机. 后者是著名的**IoC(控制反转)**与**DI(依赖注入)**的灵魂. 这也是维护老系统必备的技巧. 它的重点是在**不破坏已有接口**的前提下, 如何增加新的特性. 剩下的模式我们也大致分个类:
  - 基本数据结构在面向对象的体现: 责任链(对象链表轮询), 命令(ACID), 迭代器(C++ 中的 iterator), 备忘录(缓存状态), 状态(状态机)
  - DI: 中介者(注入中介), 观察者(注入订阅者), 访问者(注入访问接口), 策略(注入算法对象)
  - 常见编程技巧: 模板方法(固定接口, 子类实现)

Tips: 不同大类的模式, 也许名字听着类似, 但实质却完全不同. 如中介者, 与适配器, 代理, 感觉就很类似. 但中介者是被**注入**后起到作用的, 后两者却是在**结构上**增加了一层.

划出的几个重点也是面试常常考察的: 工厂, 单例, 桥接, 观察者, 访问者. 其余要么过于简单, 没啥好考察的; 要么已经和语言特性密不可分, 问起来容易被绕过去. 但这五者, 却很具备考察性, 要是能数清楚, 设计模式就算掌握了.

所有示例代码请见: <https://github.com/pezy/Design-Patterns>

如有疑问, 或发现错误, 请留言.

8月17日发布 ...

...

赞 | 13

收藏 | 42

你可能感兴趣的文章

- 一起学设计模式 - 外观模式 1 收藏, 47 浏览
- 一起学设计模式 - 组合模式 21 浏览
- 一起学设计模式 - 工厂模式 96 浏览

评论

默认排序 | 时间排序

文明社会，理性评论

发布评论

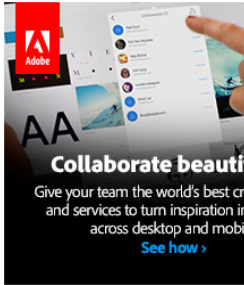
讲堂推荐

更多

JS

Promise 的 N 种用法

讲座



pezy

2.9k 声望

关注作者

发布于专栏

C/C++ 的奇技淫巧(雕虫小技)

一个 C++ 程序员

67 人关注

关注专栏

