设计模式

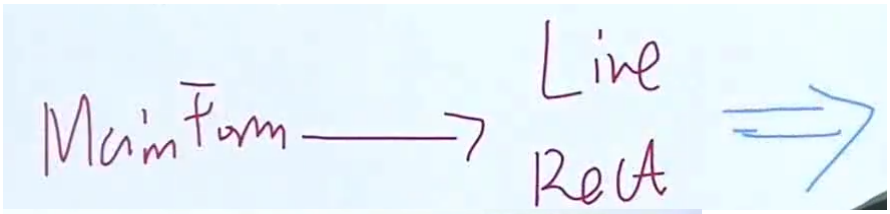
简介

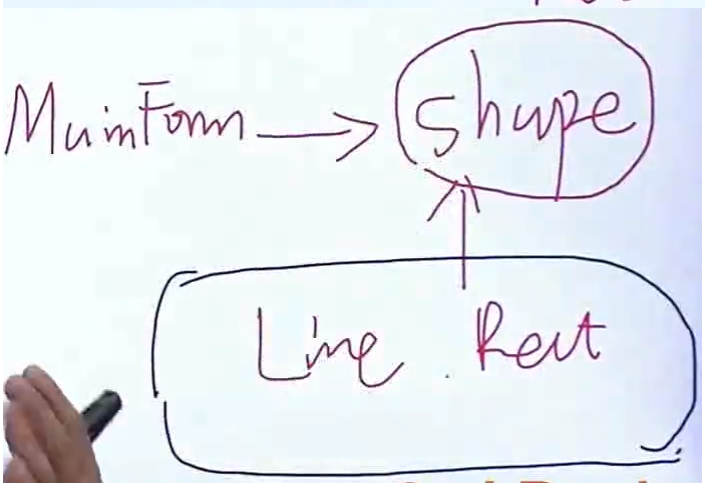
分解与抽象，两大法宝。
目的只有一个，尽可能地复用，从而应对各种变化。

原则

依赖倒置原则（DIP）

- **高层模块(稳定)**不应该依赖于**低层模块(变化)**，二者都应该依赖于**抽象(稳定)** 。
- **抽象(稳定)**不应该依赖于**实现细节(变化)** ，实现细节应该依赖于**抽象(稳定)**。





开放封闭原则（OCP）

- 对扩展开放，对更改封闭
- 类模块应该是可扩展的，但是不可修改

单一职责原则（SRP）

- 一个类应该仅有一个引起它变化的原因
- 变化的方向隐含着类的责任

Liskov替换原则（LSP）

- 子类必须能够替换它们的基类（IS-A）
- 继承表达类型抽象

接口隔离原则（ISP）

- 不应该强迫客户程序依赖它们不用的方法
- 接口应该小而完备

优先使用对象组合，而不是类继承

封装变化点

使用封装来创建对象之间的分界层，让设计者可以在分界层的一侧进行修改，而不会对另一侧产生不良的影响，从而实现层次间的松耦合。
一侧变化，一侧稳定。

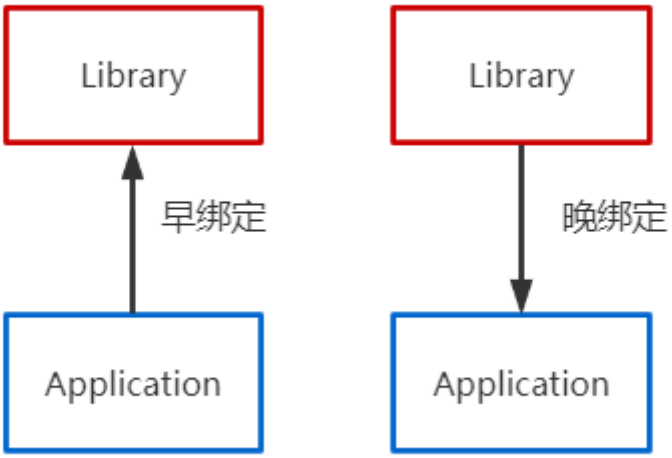
针对接口编程，而不是针对实现编程

模板方法（Template Method）

动机

在软件构建过程中，对于某一项任务，它常常有稳定的整体操作结构，但各个子步骤却有很多改变的需求，或者由于固有的原因（比如框架与应用之间的关系）而无法和任务的整体结构同时实现。
如何在确定稳定操作结构的前提下，来灵活应对各个子步骤的变化或者晚期实现需求?

早绑定与晚绑定



Library在前，Application在后，前后依赖关系。

定义

定义一个操作中的算法的骨架(稳定)，而将一些步骤延迟(变化)到子类中。Template Method使得子类可以不改变(复用)一个算法的结构即可重定义(override 重写)该算法的某些特定步骤。

总结

Template Method模式是一种非常基础性的设计模式，在面向对象系统中有着大量的应用。它用最简洁的机制（虚函数的多态性）为很多应用程序框架提供了灵活的扩展点，是代码复用方面的基本实现结构。
除了可以灵活应对子步骤的变化外，“不要调用我，让我来调用你”的反向控制结构是Template Method的典型应用。
在具体实现方面，被Template Method调用的虚方法可以具有实现，也可以没有任何实现（抽象方法、纯虚方法），但一般推荐将它们设置为protected方法。

单例模式

饿汉模式

```
class Singleton {
private:
    static Singleton *instance;
private:
    Singleton() {}
    ~Singleton() {}
    Singleton(const Singleton &other) {}
    Singleton& operator=(const Singleton& other) {}
public:
    static Singleton* getInstance() {
        return instance;
    }
};
Singleton* Singleton::instance = new Singleton;
```

懒汉模式

```
class Singleton {
private:
    static Singleton* instance;
private:
    Singleton() {}
    ~Singleton() {}
    Singleton(const Singleton& other) {}
    Singleton& operator=(const Singleton& other) {}
public:
    static Singleton* getInstance() {
        if (instance == nullptr)
            instance = new Singleton();
        return instance;
    }
};
```

懒汉-线程安全

```
class Singleton {
private:
    static Singleton* instance;
private:
    Singleton() {}
```

```
    ~Singleton() {}
    Singleton(const Singleton& other) {}
    Singleton& operator=(const Singleton& other) {}
public:
    static Singleton* getInstance() {
        //如果锁前不检查，也正确，但锁的代价过高，因为有些只是想读取一下而不是创建，但也会因为锁的原因阻塞住
        if (instance == nullptr) {
            Lock lock;
            //锁后如果不检查，那根本就不正确，这个锁加的就没有意义
            if (instance == nullptr)
                instance = new Singleton();
        }
        return instance;
    }
};
```

上面这个双检查锁的方案看似合理，但在2000年左右被专家发现会存在内存读写reorder不安全的问题。
reorder：指令虽然是顺序执行的，但是到了汇编层次后，其指令可能与我们的假设不一样。如`instance = new Singleton();`这句，会做三件事：分配内存；调用构造器；将结果返回给`instance`，但是这三步是我们假想出来的。真正在指令层，这三步可能会reorder，比如先分配内存，再把内存地址给到instance，再调用构造器。这样的话，就乱套了。

C++11版本之后的跨平台实现（volatile）

```
std::atomic<Singleton*> Singleton::m_instance;
std::mutex Singleton::m_mutex;
Singleton* Singleton::getInstance() {
    Singleton* tmp = m_instance.load(std::memory_order_relaxed);
    std::atomic_thread_fence(std::memory_order_acquire);//获取内存fence
    if (tmp == nullptr) {
        std::lock_guard<std::mutex> lock(m_mutex);
        tmp = m_instance.load(std::memory_order_relaxed);
        if (tmp == nullptr) {
            tmp = new Singleton;
            std::atomic_thread_fence(std::memory_order_release);//释放内存fence
            m_instance.store(tmp, std::memory_order_relaxed);
        }
    }
    return tmp;
}
```

策略模式（Strategy）

定义

定义一系列算法，把它们一个个封装起来，并且使它们可互相替换（变化）。该模式使得算法可独立于使用它的客户程序(稳定)而变化（扩展，子类化）。

例子

多个国家计算税率，计算方式不同

总结

- Strategy及其子类为组件提供了一系列可重用的算法，从而可以使得类型在**运行时**方便地根据需要在各个算法之间进行切换
- Strategy模式提供了用条件判断语句以外的另一种选择，消除条件判断语句，就是在解耦合。含有许多条件判断语句的代码通常都需
  要Strategy模式。当出现很多的if...else语句时，考虑能否用策略模式，前提是这些if...else是**可能变化的**。比如像一周有七天这样的if else选择，就不需要策略模式。

观察者模式（Observer）

动机

在软件构建过程中，我们需要为某些对象建立一种"通知依赖关系" ——一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知。如果这样的依赖关系过于紧密，将使软件不能很好地抵御变化。

使用面向对象技术，可以将这种依赖关系弱化，并形成一种稳定的依赖关系。从而实现软件体系结构的松耦合。

核心是抽象的通知依赖关系。

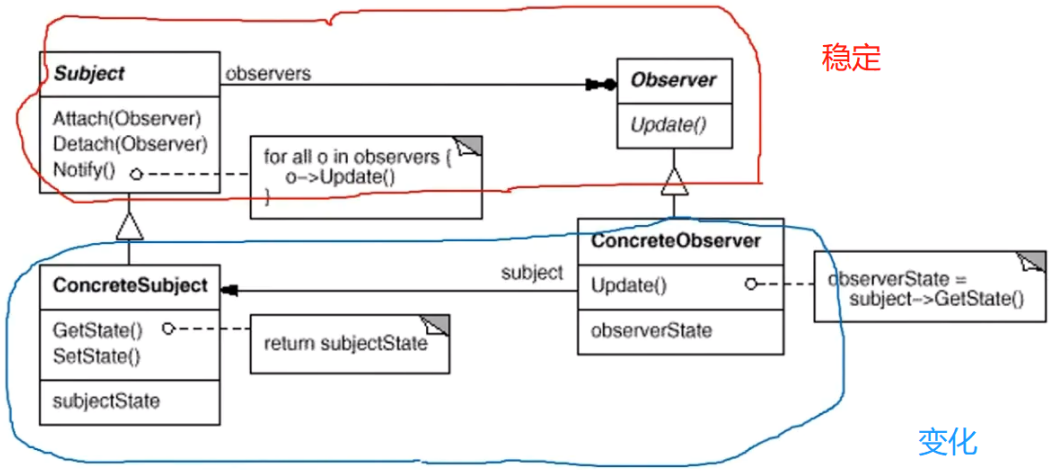
定义

定义对象间的一种一对多（变化）的依赖关系，以便当一个对象(Subject)的状态发生改变时，所有依赖于它的对象都得到通知并自动更新。

例子

进度条，通知多个。

结构（Structure）



总结

使用面向对象的抽象，Observer模式使得我们可以**独立地**改变目标与观察者，从而使二者之间的依赖关系达致**松耦合。**
目标发送通知时，无需指定观察者，通知（可以携带通知信息作为参数）会自动传播。【发布到抽象通知机制
观察者自己决定是否需要订阅通知，目标对象对此一无所知。
Observer模式是基于事件的UI框架中非常常用的设计模式，也是MVC模式的一个重要组成部分。