面向对象程序设计基础

目录

在大一的程设课上，我们系统学习了C++的语法，掌握了一些编写小型程序的技能。实际上，要想写出一个可读性好、可复用、鲁棒性强的程序，掌握一些基本的设计原则是十分必要的。

本讲的内容并不针对具体的某一语言，而且相比之前的一些内容，本讲的知识更需要在长期的实践中“内化”；与此同时，与软件工程相关的理论博大精深，本讲仅仅挑选一些代表性的原则，只能带领大家入门，想要了解更多还需要仔细阅读文末提供的书单~

KISS

KISS代表着“Keep It Simple and Stupid”。KISS原则指出，简单性应该是软件开发的主要目标，应该避免不必要的复杂性。

不过，如何界定“简单”？KISS原则指出，为了保证代码的**灵活性**和**可扩展性**，我们可能不得不增加代码的复杂度。但除此之外，在这种问题固有复杂性的基础之上增加自制的复杂性，是十分不明智的做法——程序并非程序员炫技的场所，而应该是一件简约的艺术品。

一言以概之：如无必要，勿增实体。

Loose Coupling⭐

Loose Coupling，即松耦合原则。这一原则指出：模块与模块之间的耦合（即相互关联的程度）应该越小越好，或者说，它们应该尽可能少地感知到对方的存在。

举一个例子吧（本例选自 *Clean C++* 一书）：

考虑你有一台电灯，和一个用于控制电灯的开关：

```
class Lamp
{
public:
    void on()
    {

    }

    void off()
    {

    }
}
```

```
class Switch
{
public:
    Switch(Lamp& lamp): lamp(lamp) {}

    void toggle()
    {
        if (state)
        {
            state = false;
            lamp.off();
        }

        else
        {
            state = true;
            lamp.on();
        }
    }
}
```

在这样的设计方法下，开关可以工作，但可能会带来一个问题： `Switch` 类中包含了 `Lamp` 类的引用， `Switch` 类与 `Lamp` 类之间存在着强耦合关系——`Switch` 类可以感知到 `Lamp` 类的存在。

这种写法不仅不符合常理，而且不便于维护和扩展：试想，如果我们想要用开关控制电扇、充电器等其它电器该怎么办？难道我们需要分别设计 `SwitchForLamp` 、 `SwitchForFan` 、 `SwitchForCharger` 类吗？

如何解决这类耦合问题？一个方法是：将两个类之间相关联的部分抽象成一个接口（interface），第二个类此时不需要包含第一个类的实例或引用，而只需要对接口负责，从而降低耦合度，提高程序的可扩展性。

以上程序可以改写如下（在C++中，接口可以使用虚基类实现）：

```
#include <iostream>

class Switchable
{
public:
    virtual void on() = 0;
    virtual void off() = 0;
};

class Switch
{
public:
    Switch(Switchable& switchable) : Switchable(switchable) {}
    void toggle()
    {
        if (state)
        {
            state = false;
            switchable.off();
        }
        else
        {
            state = true;
            switchable.on();
        }
    }
```

```
private:
    Switchable& switchable;
    bool state {false};
};

class Lamp: public Switchable
{
public:
    void on() override
    {
        std::cout << "Lamp is on!" << std::endl;
    }

    void off() override
    {
        std::cout << "Lamp is off!" << std::endl;
    }
};

class Fan: public Switchable
{
public:
    void on() override
    {
        std::cout << "Fan is on!" << std::endl;
    }

    void off() override
    {
        std::cout << "Fan is off!" << std::endl;
    }
};

int main()
{
    Lamp lamp;
    Switch switch1(lamp);
    switch1.toggle();
    switch1.toggle();

    Fan fan;
    Switch switch2(fan);
    switch2.toggle();
    switch2.toggle();
}
```

在以上更改中，开关与其它电器耦合的部分被抽象为一个接口 `Switchable`，开关只需要对这一接口进行操作，避免了开关与具体电器类的耦合。

SOLID ⭐

SOLID是以下五大面向对象设计原则的缩写：

- 单一功能原则（**S**ingle Responsibility Principle，SRP）
- 开闭原则（**O**pen Closed Principle，OCP）
- 里氏替换原则（**L**iskov Substitution Principle，LSP）
- 接口隔离原则（**I**nterface Segregation Principle，ISP）

- 依赖反转原则（**D**ependency Inversion Principle，DIP）。

单一功能原则

单一功能原则指出，每个软件单元（类、函数等），应该只有一个单一的、定义明确的责任。

如何界定单一责任？一个比较普适的定义是，改变该软件单元只能有一个原因。如果有多个原因，那么该单元就应该拆分。

开闭原则

开闭原则指出，软件单元（类、函数等）应该对于扩展是开放的，但是对于修改是封闭的。

具体来讲，如果我们需要给一个软件添加新的功能，我们通常不建议修改源码，而更加建议通过**继承**的方式。

里氏替换原则 ⭐

里氏原则指出，派生类（子类）对象可以在程序中代替其基类（超类）对象。

换句话说，一个软件实体如果使用的是一个父类，那么也一定适用于其子类——把一个软件里面的父类都替换为它的子类，程序的行为是不会发生变化的。

利用这一原则，我们可以判断类与类之间的继承关系是否合适。

举个例子，假设我们拥有一个矩形类：

```
class Rectangle
{
public:
    Rectangle(int width, int height) : width(width), height(height) {}

    void setWidth(int width)
    {
        this->width = width;
    }

    void setHeight(int height)
    {
        this->height = height;
    }

    void setEdges(int width, int height)
    {
        this->width = width;
        this->height = height;
    }

private:
    int width;
    int height;
};
```

我们想要再新建立一个正方形类。根据初中几何知识：正方形是一种特殊的矩形——因此一种直观的想法是：让正方形类去继承矩形类：

```
class Square: public Rectangle
{
    // ...
};
```

但如果站在里氏替换原则的角度来看，这一设计是不科学的！比如我们考虑以下操作：

```
Rectangle rectangle;
rectangle.setHeight(20);
rectangle.setEdges(10, 5);
```

根据里氏替换原则，派生类对象（Square）一定可以替换基类对象（Rectangle），假如我们进行这一替换：

```
Square square;
square.setHeight(20);
square.setEdges(10, 5);
```

这时就出现了问题：

- 第一个操作会产生歧义：该操作是只改变正方形的宽（这样会违背正方形的定义），还是同时改变正方形的长和宽（这样违背函数的字面意思）。
- 第二个操作则会直接违背正方形的定义。

可以看到，派生类对象在此处替换基类对象会产生很多问题，这一继承是不科学的！

接口隔离原则⭐

接口隔离原则指出，程序员在设计接口时应当将臃肿庞大的接口拆分成更小的和更具体的接口，让接口中只包含客户感兴趣的方法——使用多个专门的接口比使用单一的总接口要好。

换句话讲，接口约束了类的行为，是一种减轻代码耦合程度的好方法。但如果一个接口太过宽泛，可能会带来一些不必要的麻烦。举例说明：

我们想要定义一个“鸟”接口：

```
class Bird
{
public:
    virtual void eat() = 0;
    virtual void breathe() = 0;
    virtual void fly() = 0;
};
```

在此基础上实现一个鸽子类，现在一切看上去都正常：

```
class Pigeon: public Bird
{
public:
    virtual void eat() override
    {
        // ...
    }

    virtual void breathe() override
    {
        // ...
    }

    virtual void fly() override
    {
        // ...
    }
};
```

我们再实现一个企鹅类：

```
class Penguin: public Bird
{
```

```
public:
    virtual void eat() override
    {
        // ...
    }

    virtual void breathe() override
    {
        // ...
    }

    virtual void fly() override
    {
        // ???
    }
};
```

问题发生了。我们在一开始设计“鸟”这一接口时，想当然地以为所有地鸟类都会飞，却忽略了企鹅不会飞这一特例。

为了避免这样的情况发生，我们需要小心地将接口拆分：

```
class Lifeform
{
public:
    virtual void eat() = 0;
    virtual void breathe() = 0;
};

class Flyable
{
public:
    virtual void fly() = 0;
};

class Pigeon: public Lifeform, public Flyable
{
public:
    void eat() override
    {
        // ...
    }

    void breathe() override
    {
        // ...
    }

    void fly() override
    {
        // ...
    }
};

class Penguin: public Lifeform
{
public:
    void eat() override
    {
```

```
        // ...
    }

    void breathe() override
    {
        // ...
    }
};
```

如上文所示，所有的鸟类都需要呼吸和进食，我们可以大胆地将其封装为 `Lifeform` 接口，而并非所有鸟类都会飞，所以需要将其单独提取出来作为 `Flyable` 接口。在实现不同的鸟类时，我们将这些接口进行筛选组合即可。

依赖倒转原则⭐

依赖倒转原则指出，在实际的开发场景中，类与类之间的依赖关系是十分复杂，在设计依赖关系时，高层模块不应该依赖低层模块，二者都应该依赖其抽象。

什么意思呢？考虑以下实例，一个用户在某在线网络平台上拥有一个账户，而这个账户又存储着该用户的信息。由此，两者不可避免地产生了下列的循环依赖关系——你中有我，我中有你：

```
class Account;

class Customer
{
public:
    // ...
    void setAccount(Account *account)
    {
        customerAccount = account;
    }
    // ...
private:
    Account *customerAccount;
};

class Account
{
public:
    void setOwner(Customer *customer)
    {
        owner = customer;
    }

private:
    Customer *owner;
};

int main()
{
    Account* account = new Account { };
    Customer* customer = new Customer { };
    account->setOwner(customer);
    customer->setAccount(account);
}
```

这会导致很严重的问题：首先代码的可读性由于循环依赖下降，而且两者的生命周期不相互独立——如果 `Account` 对象的生命周期先于 `Customer` 对象结束， `Customer` 对象中将会产生一个空指针，调用 `Customer` 对象中的成员函数可能会导致程序崩溃。

而依赖倒转原则为解决此类问题提供了一套流程：

1. 不允许两个类中的其中一个直接访问另一个类，要想进行这种访问操作，需要通过接口。
2. 实现这个接口。

在本例中，我们不再使得 `Account` 类中包含有 `Customer` 类的指针，所有 `Account` 类需要访问 `Customer` 类的行为，都被定义进一个叫做 `Owner` 的接口中，而后， `Customer` 类需要实现这个接口：

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

class Owner
{
public:
    virtual std::string getName() = 0;
};

class Account;

class Customer : public Owner
{
public:
    void setAccount(Account* account)
    {
        customerAccount = account;
    }
    virtual std::string getName()  override
    {
        // return the Customer's name here...
    }
    // ...
private:
    Account* customerAccount;
    // ...
};

class Account
{
public:
    void setOwner(Owner* owner)
    {
        this->owner = owner;
    }
    //...
private:
    Owner* owner;
};
```

经过修改之后， `Account` 类将不依赖于 `Customer` 类。

设计模式 ⭐

C++、C#、Python等语言为实现继承、多态等面向对象特性提供了丰富的语法。那么在具体的软件工程中，又该如何使用这些特性呢？这就是设计模式。设计模式是上述SOLID原则在软件工程中的具体体现。

设计模式共计分为3大类22小类：

- **创建型模式**提供创建对象的机制， 增加已有代码的灵活性和可复用性。
- **结构型模式**介绍如何将对象和类组装成较大的结构， 并同时保持结构的灵活和高效。
- **行为模式**负责对象间的高效沟通和职责委派。

不同的设计模式之间有着相似的理念和重叠之处。合理利用设计模式可以让代码更加规范、更容易维护，但盲目使用设计模式也不是明智之举。

本讲将介绍一个难度较大，而且应用较为广泛的设计模式——**桥接模式**（属于结构型模式）。

桥接模式的定义如下：桥接模式是将类**抽象部分**与**实现部分**分离，使它们都可以独立地变化。

什么是**抽象部分**？什么是**实现部分**？让我们先考虑以下场景：一家奶茶店售卖不同种类的奶茶，奶茶既有不同的容量，也有不同的口味。如果我们只需要改变奶茶的容量，可以做出如下设计：

```
class IMilkTea // 通用接口
{
    virtual void order() = 0;
};

class MilkTeaSmallCup: public IMilkTea
{
    void order() override
    {
        std::cout << "order info:" << std::endl;
        std::cout << "size: small cup" << std::endl;
    }
};

class MilkTeaMediumCup: public IMilkTea
{
    void order() override
    {
        std::cout << "order info:" << std::endl;
        std::cout << "size: medium cup" << std::endl;
    }
};

class MilkTeaLargeCup: public IMilkTea
{
    void order() override
    {
        std::cout << "order info:" << std::endl;
        std::cout << "size: large cup" << std::endl;
    }
};
```

当类的变化只有一个维度时，继承的思路是比较直接而简单的。但当我们将“口味”也加入继承体系中，也就是当类的变化有两个维度时，沿用上面的思路将会使得类的数量急剧增长：

```
class MilkTeaSmallCupFairyGrass: public IMilkTea
{
    void order() override
    {
        std::cout << "order info:" << std::endl;
        std::cout << "size: small cup" << std::endl;
        std::cout << "flavor: fairy grass" << std::endl;
    }
};

class MilkTeaSmallCupPearl: public IMilkTea
{
    void order() override
```

```
    {
        std::cout << "order info:" << std::endl;
        std::cout << "size: small cup" << std::endl;
        std::cout << "flavor: pearl" << std::endl;
    }
};

// class MilkTeaMediumCupPearl, class MilkTeaLargeCupFairyGrass, ...
```

问题的根源在于，我们试图在两个独立的维度（“容量”和“口味”）上扩展奶茶类。这时候，桥接模式就派上了用场：我们将容量视为**抽象部分**，将口味视为**实现部分**，并将两者桥接。

> “抽象部分”和“实现部分”所承担的角色：
>
> - 抽象部分：抽象化给出的定义，只提供高层控制逻辑，依赖于完成底层实际工作的实现对象。抽象部分保存一个对实现化对象的引用（指针）。
> - 实现部分：给出实现化角色的通用接口，抽象部分仅能通过在这里声明的方法与实现对象交互。

例如在本例中，可以做如下修改：

```
// 实现化部分
class IMilkTeaFlavorBase
{
public:
    virtual void GetFlavor() = 0;
};

class MilkTeaPearl: public IMilkTeaFlavorBase
{
public:
    void GetFlavor() override
    {
        std::cout << "flavor: pearl" << std::endl;
    }
};

class MilkTeaFairyGrass: public IMilkTeaFlavorBase
{
public:
    void GetFlavor() override
    {
        std::cout << "flavor: fairy grass" << std::endl;
    }
};

// 抽象化部分
class IMilkTeaSizeBase
{
public:
    virtual void SetFlavor(std::shared_ptr<IMilkTeaFlavorBase> flavorBase)
    {
        this->flavorBase = flavorBase;
    }
    virtual void Order() = 0;
protected:
    std::shared_ptr<IMilkTeaFlavorBase> flavorBase;
};

class MilkTeaSmall: public IMilkTeaSizeBase
```

```
{
public:
    void Order() override
    {
        std::cout << "size: small" << std::endl;
        flavorBase->GetFlavor();
    }
};

class MilkTeaMedium: public IMilkTeaSizeBase
{
public:
    void Order() override
    {
        std::cout << "size: medium" << std::endl;
        flavorBase->GetFlavor();
    }
};

class MilkTeaLarge: public IMilkTeaSizeBase
{
public:
    void Order() override
    {
        std::cout << "size: large" << std::endl;
        flavorBase->GetFlavor();
    }
};

// 使用方法
int main()
{
    // 大杯烧仙草
    std::shared_ptr<MilkTeaFairyGrass> milkTeaFairyGrass =
std::make_shared<MilkTeaFairyGrass>();
    std::shared_ptr<MilkTeaLarge> milkTeaLargeWithFairyGrass = std::make_shared<MilkTeaLarge>
();
    milkTeaLargeWithFairyGrass->SetFlavor(milkTeaFairyGrass);
    milkTeaLargeWithFairyGrass->Order();
}
```

可以在上述示例中看到：抽象部分各类中，都含有一个实现部分的指针。如果需要访问实现部分的方法，可以通过该指针进行访问。这样，我们就通过桥接的方式分离了两个不同的维度，使得类的可扩展性更好。

由于篇幅所限，我们在此处不能对设计模式进行一一介绍，感兴趣的同学可以参考文末给出的阅读清单进行学习。

参考文献和荐读清单

Refactoring.Guru 该网站详细介绍了各设计模式的特点，并提供了不同编程语言的实例。

Clean C++ 这本书的侧重点不在介绍C++语法，而侧重于使用C++语言介绍如何写出可读性强、符合面向对象规范的程序，强推！