


小肚哥

博客园 首页 新随笔 联系 管理 订阅 

随笔- 2 文章- 0 评论- 1

C++ 常用设计模式（学习笔记）

1、工厂模式：简单工厂模式、工厂方法模式、抽象工厂模式

1)、简单工厂模式：主要特点是需要在工厂类中做判断，从而创造相应的产品，当增加新产品时，需要修改工厂类。



复制代码

```
typedef enum
{
    T80 = 1,
    T99
}TankType;

class Tank
{
public:
    virtual void message() = 0;
};

class Tank80:public Tank
{
public:
    void message()
    {
        cout << "Tank80" << endl;
    }
};

class Tank99:public Tank
{
public:
    void message()
    {
        cout << "Tank99" << endl;
    }
};

class TankFactory
{
public:
    Tank* createTank(TankType type)
    {
        switch(type)
        {
            case 1:
                return new Tank80();
            case 2:
                return new Tank99();
            default:
                return NULL;
        }
    }
};
```



复制代码

2)、工厂方法模式：是指定义一个创建对象的接口，让子类决定实例化哪一个类，Factory Method使一个类的实例化延迟到其子类。

主要解决：主要解决接口选择的问题。

何时使用：我们明确地计划不同条件下创建不同实例时。

如何解决：让其子类实现工厂接口，返回的也是一个抽象的产品。

关键代码：创建过程在其子类执行。

缺点：每增加一种产品，就需要增加一个对象工厂。相比简单工厂模式，工厂方法模式需要定义更多的类。

 复制代码

```
class Tank
{
public:
    virtual void message() = 0;
};

class Tank80:public Tank
{
public:
    void message()
    {
        cout << "Tank80" << endl;
    }
};

class Tank99:public Tank
{
public:
    void message()
    {
        cout << "Tank99" << endl;
    }
};

class TankFactory
{
public:
    virtual Tank* createTank() = 0;
};

class Tank80Factory:public TankFactory
{
public:
    Tank* createTank()
    {
        return new Tank80();
    }
};

class Tank99Factory:public TankFactory
{
public:
    Tank* createTank()
    {
        return new Tank99();
    }
};
```

 复制代码

3)、抽象工厂模式：提供一个创建一系列相关或相互依赖的对象接口，而无需指定它们的具体类。

主要解决：主要解决接口选择的问题。

何时使用：系统的产品有多于一个的产品族，而系统只消费其中某一族的产品。

如何解决：在一个产品族里面，定义多个产品。

关键代码：在一个工厂里聚合多个同类产品。

缺点：产品族扩展非常困难，要增加一个系列的某一产品，既要在抽象的 Creator 里加代码，又要在具体的里面加代码。

 复制代码

```
class Tank
{
public:
    virtual void message() = 0;
};

class Tank80:public Tank
{
public:
    void message()
    {
        cout << "Tank80" << endl;
    }
};
```

```

class Tank99:public Tank
{
public:
    void message()
    {
        cout << "Tank99" << endl;
    }
};

class Plain
{
public:
    virtual void message() = 0;
};

class Plain80: public Plain
{
public:
    void message()
    {
        cout << "Plain80" << endl;
    }
};

class Plain99: public Plain
{
public:
    void message()
    {
        cout << "Plain99" << endl;
    }
};

class Factory
{
public:
    virtual Tank* createTank() = 0;
    virtual Plain* createPlain() = 0;
};

class Factory80:public Factory
{
public:
    Tank* createTank()
    {
        return new Tank80();
    }
    Plain* createPlain()
    {
        return new Plain80();
    }
};

class Factory99:public Factory
{
public:
    Tank* createTank()
    {
        return new Tank99();
    }
    Plain* createPlain()
    {
        return new Plain99();
    }
};

```

 复制代码

2、策略模式:是指定义一系列的算法,把它们一个个封装起来,并且使它们可以互相替换。使得算法可以独立于使用它的客户而变化,也就是说这些算法所完成的功能是一样的,对外接口是一样的,只是各自现实上存在差异。

主要解决:在有多种算法相似的情况下,使用 if...else 所带来的复杂和难以维护。

何时使用:一个系统有许多许多类,而区分它们的只是他们直接的行为。

如何解决:将这些算法封装成一个一个的类,任意地替换。

关键代码：实现同一个接口。

缺点：1、策略类会增多。2、所有策略类都需要对外暴露。

 复制代码

//传统策略模式实现

```
class Hurt
{
public:
    virtual void redBuff() = 0;
};

class AdcHurt:public Hurt
{
public:
    void redBuff()
    {
        cout << "Adc hurt" << endl;
    }
};

class ApcHurt:public Hurt
{
public:
    void redBuff()
    {
        cout << "Apc hurt" << endl;
    }
};

//方法1: 传入一个指针参数
class Soldier
{
public:
    Soldier(Hurt* hurt):m_hurt(hurt)
    {
    }
    ~Soldier()
    {
    }
    void beInjured()
    {
        m_hurt->redBuff();
    }
private:
    Hurt* m_hurt;
};

//方法2: 传入一个参数标签
typedef enum
{
    adc,
    apc
}HurtType;

class Master
{
public:
    Master(HurtType type)
    {
        switch(type)
        {
            case adc:
                m_hurt = new AdcHurt;
                break;
            case apc:
                m_hurt = new ApcHurt;
                break;
            default:
                m_hurt = NULL;
                break;
        }
    }
    ~Master()
    {
    }
    void beInjured()
```

```

    {
        if(m_hurt != NULL)
        {
            m_hurt->redBuff();
        }
        else
        {
            cout << "Not hurt" << endl;
        }
    }
};

private:
    Hurt* m_hurt;
};

//方法3: 使用模板类
template <typename T>
class Tank
{
public:
    void beInjured()
    {
        m_hurt.redBuff();
    }
private:
    T m_hurt;
};
//END

//使用函数指针实现策略模式
void adcHurt(int num)
{
    cout << "adc hurt:" << num << endl;
}

void apcHurt(int num)
{
    cout << "apc hurt:" << num << endl;
}

//普通函数指针
class Aid
{
public:
    typedef void (*HurtFun)(int);

    Aid(HurtFun fun):m_fun(fun)
    {
    }
    void beInjured(int num)
    {
        m_fun(num);
    }
private:
    HurtFun m_fun;
};

//使用std::function , 头文件: #include<functional>
class Bowman
{
public:
    typedef std::function<void(int)> HurtFunc;

    Bowman(HurtFunc fun):m_fun(fun)
    {
    }
    void beInjured(int num)
    {
        m_fun(num);
    }
private:
    HurtFunc m_fun;
};
//END

```



3、适配器模式：将一个类的接口转换成客户希望的另一个接口，使得原本由于接口不兼容而不能一起工作的哪些类可以一起工作。

主要解决：主要解决在软件系统中，常常要将一些“现存的对象”放到新的环境中，而新环境要求的接口是现对象不能满足的。

何时使用：1、系统需要使用现有的类，而此类的接口不符合系统的需要。2、想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作，这些源类不一定有一致的接口。3、通过接口转换，将一个类插入另一个类系中。（比如老虎和飞禽，现在多了一个飞虎，在不增加实体的需求下，增加一个适配器，在里面包容一个虎对象，实现飞的接口。）

如何解决：继承或依赖（推荐）。

关键代码：适配器继承或依赖已有的对象，实现想要的目标接口。

缺点：1、过多地使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是 A 接口，其实内部被适配成了 B 接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。

 复制代码

//使用复合，对象模式

```
class Deque //双端队列，被适配类
{
public:
    void push_back(int x)
    {
        cout << "Deque push_back:" << x << endl;
    }
    void push_front(int x)
    {
        cout << "Deque push_front:" << x << endl;
    }
    void pop_back()
    {
        cout << "Deque pop_back" << endl;
    }
    void pop_front()
    {
        cout << "Deque pop_front" << endl;
    }
};

class Sequence //顺序类，目标类
{
public:
    virtual void push(int x) = 0;
    virtual void pop() = 0;
};

class Stack:public Sequence //栈，适配类
{
public:
    void push(int x)
    {
        m_deque.push_back(x);
    }
    void pop()
    {
        m_deque.pop_back();
    }
private:
    Deque m_deque;
};

class Queue:public Sequence //队列，适配类
{
public:
    void push(int x)
    {
        m_deque.push_back(x);
    }
    void pop()
    {
        m_deque.pop_front();
    }
private:
    Deque m_deque;
};
```

```
};
//END
```

 复制代码

 复制代码

//使用继承, 类模式

class Deque //双端队列, 被适配类

```
{
public:
    void push_back(int x)
    {
        cout << "Deque push_back:" << x << endl;
    }
    void push_front(int x)
    {
        cout << "Deque push_front:" << x << endl;
    }
    void pop_back()
    {
        cout << "Deque pop_back" << endl;
    }
    void pop_front()
    {
        cout << "Deque pop_front" << endl;
    }
};
```

class Sequence //顺序类, 目标类

```
{
public:
    virtual void push(int x) = 0;
    virtual void pop() = 0;
};
```

class Stack:public Sequence, private Deque //栈, 适配类

```
{
public:
    void push(int x)
    {
        push_back(x);
    }
    void pop()
    {
        pop_back();
    }
};
```

class Queue:public Sequence, private Deque //队列, 适配类

```
{
public:
    void push(int x)
    {
        push_back(x);
    }
    void pop()
    {
        pop_front();
    }
};
```

//END

 复制代码

4、单例模式：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

主要解决：一个全局使用的类频繁地创建与销毁。

何时使用：想控制实例数目，节省系统资源的时候。

如何解决：判断系统是否已存在单例，如果有则返回，没有则创建。

关键代码：构造函数是私有的。

单例大约有兩種实现方法：懒汉与饿汉。

懒汉：顾名思义，不到万不得已就不会去实例化类，也就是说在第一次用到类实例的时候才会去实例化，所以上边的经典方法被归为懒汉实现；

饿汉：饿了肯定要饥不择食。所以在单例类定义的时候就进行实例化。

特点与选择：

由于要进行线程同步，所以在访问量比较大，或者可能访问的线程比较多时，采用饿汉实现，可以实现更好的性

能。这是以空间换时间。

在访问量较小时，采用懒汉实现。这是以时间换空间。

 复制代码

```
//懒汉式一般实现：非线性安全，getInstance返回的实例指针需要delete
class Singleton
{
public:
    static Singleton* getInstance();
    ~Singleton() {}

private:
    static Singleton* m_pSingleton;
    Singleton() {}
    Singleton(const Singleton& obj) = delete; //明确拒绝
    Singleton& operator=(const Singleton& obj) = delete; //明确拒绝
};

Singleton* Singleton::m_pSingleton = NULL;

Singleton* Singleton::getInstance()
{
    if(m_pSingleton == NULL)
    {
        m_pSingleton = new Singleton;
    }
    return m_pSingleton;
}
//END

//懒汉式：加lock，线程安全
std::mutex mt;

class Singleton
{
public:
    static Singleton* getInstance();
private:
    Singleton() {}
    Singleton(const Singleton&) = delete; //明确拒绝
    Singleton& operator=(const Singleton&) = delete; //明确拒绝

    static Singleton* m_pSingleton;
};

Singleton* Singleton::m_pSingleton = NULL;

Singleton* Singleton::getInstance()
{
    if(m_pSingleton == NULL)
    {
        mt.lock();
        m_pSingleton = new Singleton();
        mt.unlock();
    }
    return m_pSingleton;
}
//END

//返回一个reference指向local static对象
//多线程可能存在不确定性：任何一种non-const static对象，不论它是local或non-local，在多线程环境下“等待某事发生”都会有麻烦。解决的方法：在程序的单线程启动阶段手工调用所有reference-returning函数。
class Singleton
{
public:
    static Singleton& getInstance();
private:
    Singleton() {}
    Singleton(const Singleton&) = delete; //明确拒绝
    Singleton& operator=(const Singleton&) = delete; //明确拒绝
};

Singleton& Singleton::getInstance()
{
    static Singleton singleton;
```



```

    return singleton;
}
//END

//饿汉式：线程安全，注意delete
class Singleton
{
public:
    static Singleton* getInstance();
private:
    Singleton() {}
    Singleton(const Singleton&) = delete; //明确拒绝
    Singleton& operator=(const Singleton&) = delete; //明确拒绝

    static Singleton* m_pSingleton;
};

Singleton* Singleton::m_pSingleton = new Singleton();

Singleton* Singleton::getInstance()
{
    return m_pSingleton;
}
//END

```

 复制代码

5、原型模式：用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。


主要解决：在运行期建立和删除对象。

何时使用：1). 当我们的对象类型不是开始就能确定的，而这个类型是在运行期确定的话，那么我们通过这个类型的对象克隆出一个新的对象比较容易一些；2). 有的时候，我们需要一个对象在某个状态下的副本，此时，我们使用原型模式是最好的选择；例如：一个对象，经过一段处理之后，其内部的状态发生了变化；这个时候，我们需要一个这个状态的副本，如果直接new一个新的对象的话，但是它的状态是不对的，此时，可以使用原型模式，将原来的对象拷贝一个出来，这个对象就和之前的对象是完全一致的了；3). 当我们处理一些比较简单的对象时，并且对象之间的区别很小，可能就几个属性不同而已，那么就可以使用原型模式来完成，省去了创建对象时的麻烦了；4). 有的时候，创建对象时，构造函数的参数很多，而自己又不完全的知道每个参数的意义，就可以使用原型模式来创建一个新的对象，不必去理会创建的过程。

->适当的时候考虑一下原型模式，能减少对应的工作量，减少程序的复杂度，提高效率

如何解决：利用已有的一个原型对象，快速地生成和原型对象一样的实例。

关键代码：拷贝，return new className(*this);

 复制代码

```

class Clone
{
public:
    Clone()
    {
    }
    virtual ~Clone()
    {
    }
    virtual Clone* clone() = 0;
    virtual void show() = 0;
};

class Sheep:public Clone
{
public:
    Sheep(int id, string name):Clone(),m_id(id),m_name(name)
    {
        cout << "Sheep() id add:" << m_id << endl;
        cout << "Sheep() name add:" << m_name << endl;
    }
    ~Sheep()
    {
    }

    Sheep(const Sheep& obj)
    {
        this->m_id = obj.m_id;
        this->m_name = obj.m_name;
        cout << "Sheep(const Sheep& obj) id add:" << m_id << endl;
    }
}

```

```

        cout << "Sheep(const Sheep& obj) name add:" << &m_name << endl;
    }

    Clone* clone()
    {
        return new Sheep(*this);
    }
    void show()
    {
        cout << "id : " << m_id << endl;
        cout << "name:" << m_name.data() << endl;
    }
private:
    int m_id;
    string m_name;
};

int main()
{
    Clone* s1 = new Sheep(1, "abs");
    s1->show();
    Clone* s2 = s1->clone();
    s2->show();
    delete s1;
    delete s2;
    return 0;
}

```

 复制代码

6、模板模式：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

主要解决：多个子类有相同的方法，并且逻辑相同，细节有差异。

如何解决：对重要，复杂的算法，将核心算法设计为模板方法，周边细节由子类实现，重构时，经常使用的方法，将相同的代码抽象到父类，通过钩子函数约束行为。

关键代码：在抽象类实现通用接口，细节变化在子类实现。

缺点：每一个不同的实现都需要一个子类来实现，导致类的个数增加，使得系统更加庞大。

 复制代码

```

class Computer
{
public:
    void product()
    {
        installCpu();
        installRam();
        installGraphicsCard();
    }

protected:
    virtual void installCpu() = 0;
    virtual void installRam() = 0;
    virtual void installGraphicsCard() = 0;
};

class ComputerA:public Computer
{
protected:
    void installCpu() override
    {
        cout << "ComputerA install Inter Core i5" << endl;
    }

    void installRam() override
    {
        cout << "ComputerA install 2G Ram" << endl;
    }

    void installGraphicsCard() override
    {
        cout << "ComputerA install Gtx940 GraphicsCard" << endl;
    }
};

```

```

class ComputerB:public Computer
{
protected:
    void installCpu() override
    {
        cout << "ComputerB install Inter Core i7" << endl;
    }

    void installRam() override
    {
        cout << "ComputerB install 4G Ram" << endl;
    }

    void installGraphicsCard() override
    {
        cout << "ComputerB install Gtx960 GraphicsCard" << endl;
    }
};

```



7、建造者模式：将复杂对象的构建和其表示分离，使得同样的构建过程可以创建不同的表示。

主要解决：一个复杂对象的创建工作，由各个部分的子对象用一定的算法构成；由于需求变化，这个复杂对象的各个部分经常面临变化，但将它们组合在一起的算法却相对稳定。

如何解决：将变与不变分开

关键代码：建造者：创建和提供实例，Director：管理建造出来的实例的依赖关系。。

缺点：1、产品必须有共同点，范围有限制。 2、如内部变化复杂，会有很多的建造类。



```

typedef enum
{
    type1,
    type2
}ProductType;

class Product    //产品
{
public:
    void setNum(int num);
    void setColor(string color);
    void setType(ProductType type);

    void showProduct();
private:
    int m_num;
    string m_color;
    ProductType m_type;
};

void Product::setNum(int num)
{
    m_num = num;
}

void Product::setColor(string color)
{
    m_color = color;
}

void Product::setType(ProductType type)
{
    m_type = type;
}

void Product::showProduct()
{
    cout << "Product: " << endl;
    cout << "    num : " << m_num << endl;
    cout << "    color: " << m_color.data() << endl;
    cout << "    type : " << m_type << endl;
}

```

```
//建造者父类, 定义接口
class Builder
{
public:
    Builder(){}
    virtual ~Builder(){}
    virtual void buildNum(int num) = 0;
    virtual void buildColor(string color) = 0;
    virtual void buildType(ProductType type) = 0;
    virtual void createProduct() = 0;
    virtual Product* getProduct() = 0;
    virtual void show() = 0;
};

//建造者A
class BuilderA:public Builder
{
public:
    BuilderA(){}
    ~BuilderA(){}
    void buildNum(int num) override;
    void buildColor(string color) override;
    void buildType(ProductType type) override;
    void createProduct() override;
    Product* getProduct() override;
    void show() override;
private:
    Product* m_product;
};

void BuilderA::buildNum(int num)
{
    cout << "BuilderA build Num: " << num << endl;
    m_product->setNum(num);
}

void BuilderA::buildColor(string color)
{
    cout << "BuilderA build color: " << color.data() << endl;
    m_product->setColor(color);
}

void BuilderA::buildType(ProductType type)
{
    cout << "BuilderA build type: " << type << endl;
    m_product->setType(type);
}

void BuilderA::createProduct()
{
    cout << "BuilderA CreateProduct: " << endl;
    m_product = new Product();
}

Product* BuilderA::getProduct()
{
    return m_product;
}

void BuilderA::show()
{
    m_product->showProduct();
}

//建造者B
class BuilderB:public Builder
{
public:
    BuilderB(){}
    ~BuilderB(){}
    void buildNum(int num) override;
    void buildColor(string color) override;
    void buildType(ProductType type) override;
    void createProduct() override;
    Product* getProduct() override;
    void show() override;
private:
    Product* m_product;
```

```
};

void BuilderB::buildNum(int num)
{
    cout << "BuilderB build Num: " << num << endl;
    m_product->setNum(num);
}

void BuilderB::buildColor(string color)
{
    cout << "BuilderB build color: " << color.data() << endl;
    m_product->setColor(color);
}

void BuilderB::buildType(ProductType type)
{
    cout << "BuilderB build type: " << type << endl;
    m_product->setType(type);
}

void BuilderB::createProduct()
{
    cout << "BuilderB CreateProduct: " << endl;
    m_product = new Product();
}

Product* BuilderB::getProduct()
{
    return m_product;
}

void BuilderB::show()
{
    m_product->showProduct();
}

//管理类, 负责安排构造的具体过程
class Director
{
public:
    Director(Builder* builder):m_builder(builder)
    {
    }

    void construct(int num, string color, ProductType type)
    {
        m_builder->createProduct();
        m_builder->buildNum(num);
        m_builder->buildColor(color);
        m_builder->buildType(type);
    }

private:
    Builder* m_builder;
};
```

 复制代码

8、外观模式：为子系统的一组接口定义一个一致的界面，外观模式提供了一个高层接口，这个接口使得这一子系统更加容易被使用；对于复杂的系统，系统为客户提供一个简单的接口，把复杂的实现过程封装起来，客户不需要了解系统内部的细节。

主要解决：客户不需要了解系统内部复杂的细节，只需要一个接口；系统入口。

如何解决：客户不直接与系统耦合，而是通过外观类与系统耦合。

关键代码：客户与系统之间加一个外观层，外观层处理系统的调用关系、依赖关系等。

缺点：需要修改时不易继承、不易修改。

 复制代码

```
class Cpu
{
public:
    void productCpu()
    {
        cout << "Product Cpu" << endl;
    }
};
```

```

class Ram
{
public:
    void productRam()
    {
        cout << "Product Ram" << endl;
    }
};

class Graphics
{
public:
    void productGraphics()
    {
        cout << "Product Graphics" << endl;
    }
};

class Computer
{
public:
    void productComputer()
    {
        Cpu cpu;
        cpu.productCpu();
        Ram ram;
        ram.productRam();
        Graphics graphics;
        graphics.productGraphics();
    }
};

int main()
{
    //客户直接调用computer生产函数，无需关心具体部件的生产过程。也可直接单独生产部件
    Computer computer;
    computer.productComputer();

    Cpu cpu;
    cpu.productCpu();

    return 0;
}

```

 复制代码

9、组合模式：将对象组合成树形结构以表示“部分-整体”的层次结构，组合模式使得用户对单个对象和组合对象的使用具有一致性。

主要解决：它在我们树型结构的问题中，模糊了简单元素和复杂元素的概念，客户程序可以像处理简单元素一样处理复杂元素，从而使得客户程序与复杂元素的内部结构解耦。

如何解决：树枝和树叶实现统一接口，树枝内部组合该接口。

关键代码：树枝内部组合该接口，并且含有内部属性list，里面放Component。

 复制代码

```

class Company
{
public:
    Company(string name):m_name(name)
    {}
    virtual ~Company(){}
    virtual void add(Component* company) = 0;
    virtual void remove(string name) = 0;
    virtual void display(int depth) = 0;
    string getName()
    {
        return m_name;
    }

protected:
    string m_name;
};

//具体的公司
class ConcreteCompany:public Company //树枝

```

```

{
public:
    ConcreteCompany(string name):Company(name)
    {}
    ~ConcreteCompany()
    {
        cout << "~ConcreteCompany()" << endl;
    }
    void add(Company* company) override;
    void remove(string name) override;
    void display(int depth) override;

private:
    list<shared_ptr<Company>> m_listCompany;
};

void ConcreteCompany::add(Company* company)
{
    shared_ptr<Company> temp(company);
    m_listCompany.push_back(temp);
}

void ConcreteCompany::remove(string name)
{
    list<shared_ptr<Company>>::iterator iter = m_listCompany.begin();
    for(; iter != m_listCompany.end(); iter++)
    {
        shared_ptr<Company> temp(*iter);
        string strName = temp.get()->getName();
        if(name == strName)
        {
            m_listCompany.erase(iter);
        }
    }
}

void ConcreteCompany::display(int depth)
{
    for(int i = 0; i < depth; i++)
    {
        cout << "-";
    }
    cout << m_name.data() << endl;
    list<shared_ptr<Company>>::iterator iter = m_listCompany.begin();
    for(; iter != m_listCompany.end(); iter++)
    {
        shared_ptr<Company> temp(*iter);
        temp.get()->display(depth + 2);
    }
}

//公司下的部门
class FinanceDept:public Company    //树叶
{
public:
    FinanceDept(string name):Company(name)
    {}
    ~FinanceDept()
    {
        cout << "~FinanceDept()" << endl;
    }
    void add(Company* company) override;
    void remove(string name) override;
    void display(int depth) override;
};

void FinanceDept::add(Company* company)
{
    cout << "FinanceDept add failed" << endl;
}

void FinanceDept::remove(string name)
{
    cout << "FinanceDept remove failed" << endl;
}

```

```

void FinanceDept::display(int depth)
{
    for(int i = 0; i < depth; i++)
    {
        cout << "-";
    }
    cout << m_name.data() << endl;
}

//公司下的部门
class HRDept:public Company //树叶
{
public:
    HRDept(string name):Company(name)
    {}
    ~HRDept()
    {
        cout << "~HRDept()" << endl;
    }
    void add(Company* company) override;
    void remove(string name) override;
    void display(int depth) override;
};

void HRDept::add(Company* company)
{
    cout << "HRDept add failed" << endl;
}

void HRDept::remove(string name)
{
    cout << "HRDept remove failed" << endl;
}

void HRDept::display(int depth)
{
    for(int i = 0; i < depth; i++)
    {
        cout << "-";
    }
    cout << m_name.data() << endl;
}

int main(int argc, char *argv[])
{
    Company* root = new ConcreteCompany("zong");
    Company* f1 = new FinanceDept("F1");
    Company* h1 = new HRDept("H1");
    root->add(f1);
    root->add(h1);
    Company* c1 = new ConcreteCompany("fen1");
    Company* f2 = new FinanceDept("F2");
    Company* h2 = new HRDept("H2");
    c1->add(f2);
    c1->add(h2);
    root->add(c1);
    root->display(0);
    delete root;

    return 0;
}

```

 复制代码

10、代理模式：为其它对象提供一种代理以控制对这个对象的访问。

主要解决：在直接访问对象时带来的问题，比如：要访问的对象在远程服务器上。在面向对象系统中，有些对象由于某些原因，直接访问会给使用者或系统带来很多麻烦，可以在访问此对象时加上一个对此对象的访问层。

如何解决：增加中间代理层。

关键代码：实现与被代理类组合。

 复制代码

```

class Gril
{
public:

```



```

    Gril(string name = "gril"):m_string(name){}
    string getName()
    {
        return m_string;
    }
private:
    string m_string;
};

class Profession
{
public:
    virtual ~Profession(){}
    virtual void profess() = 0;
};

class YoungMan:public Profession
{
public:
    YoungMan(Gril gril):m_gril(gril){}
    void profess()
    {
        cout << "Young man love " << m_gril.getName().data() << endl;
    }
private:
    Gril m_gril;
};

class ManProxy:public Profession
{
public:
    ManProxy(Gril gril):m_man(new YoungMan(gril)){}
    void profess()
    {
        cout << "I am Proxy" << endl;
        m_man->profess();
    }
private:
    YoungMan* m_man;
};

int main(int argc, char *argv[])
{
    Gril gril("hei");
    Profession* proxy = new ManProxy(gril);
    proxy->profess();
    delete proxy;
    return 0;
}

```

 复制代码

11、享元模式：运用共享技术有效地支持大量细粒度的对象。

主要解决：在有大量对象时，把其中共同的部分抽象出来，如果有相同的业务请求，直接返回内存中已有的对象，避免重新创建。

如何解决：用唯一标识码判断，如果内存中有，则返回这个唯一标识码所标识的对象。

关键代码：将内部状态作为标识，进行共享。

 复制代码

```

//以Money的类别作为内部标识，面值作为外部状态。
enum MoneyCategory    //类别，内在标识，作为标识码
{
    Coin,
    bankNote
};

enum FaceValue        //面值，外部标识，需要存储的对象
{
    ValueOne = 1,
    ValueTwo
};

class Money            //抽象父类
{

```

```

public:
    Money(MoneyCategory cate):m_mCate(cate){}
    virtual ~Money(){ cout << "~Money() " << endl; }
    virtual void save() = 0;
private:
    MoneyCategory m_mCate;
};

class MoneyCoin:public Money    //具体子类1
{
public:
    MoneyCoin(MoneyCategory cate):Money(cate){}
    ~MoneyCoin(){ cout << "~MoneyCoin()" << endl; }
    void save()
    {
        cout << "Save Coin" << endl;
    }
};

class MoneyNote:public Money    //具体子类2
{
public:
    MoneyNote(MoneyCategory cate):Money(cate){}
    ~MoneyNote(){ cout << "~MoneyNote()" << endl; }
    void save()
    {
        cout << "Save BankNote" << endl;
    }
};

class Bank
{
public:
    Bank():m_coin(nullptr),m_note(nullptr),m_count(0){}
    ~Bank()
    {
        if(m_coin != nullptr)
        {
            delete m_coin;
            m_coin = nullptr;
        }
        if(m_note != nullptr)
        {
            delete m_note;
            m_note = nullptr;
        }
    }
    void saveMoney(MoneyCategory cate, FaceValue value)
    {
        switch(cate)    //以类别作为标识码
        {
            case Coin:
            {
                if(m_coin == nullptr)    //内存中存在标识码所标识的对象，则直接调用，不再创建
                {
                    m_coin = new MoneyCoin(Coin);
                }
                m_coin->save();
                m_vector.push_back(value);
                break;
            }
            case bankNote:
            {
                if(m_note == nullptr)
                {
                    m_note = new MoneyNote(bankNote);
                }
                m_note->save();
                m_vector.push_back(value);
                break;
            }
            default:
                break;
        }
    }

    int sumSave()

```

```

    {
        auto iter = m_vector.begin();
        for(; iter != m_vector.end(); iter++)
        {
            m_count += *iter;
        }
        return m_count;
    }

private:
    vector<FaceValue> m_vector;
    Money* m_coin;
    Money* m_note;
    int m_count;
};

int main()
{
    Bank b1;
    b1.saveMoney(Coin, ValueOne);
    b1.saveMoney(Coin, ValueTwo);
    b1.saveMoney(Coin, ValueTwo);
    b1.saveMoney(bankNote, ValueOne);
    b1.saveMoney(bankNote, ValueTwo);
    cout << b1.sumSave() << endl;

    return 0;
}

```

 复制代码

12、桥接模式：将抽象部分与实现部分分离，使它们都可以独立变换。

主要解决：在有很多中可能会变化的情况下，用继承会造成类爆炸问题，不易扩展。

如何解决：把不同的分类分离出来，使它们独立变化，减少它们之间的耦合。

关键代码：将现实独立出来，抽象类依赖现实类。

 复制代码

```

//将各种App、各种手机全部独立分开，使其自由组合桥接
class App
{
public:
    virtual ~App(){ cout << "~App()" << endl; }
    virtual void run() = 0;
};

class GameApp:public App
{
public:
    void run()
    {
        cout << "GameApp Running" << endl;
    }
};

class TranslateApp:public App
{
public:
    void run()
    {
        cout << "TranslateApp Running" << endl;
    }
};

class MobilePhone
{
public:
    virtual ~MobilePhone(){ cout << "~MobilePhone()" << endl; }
    virtual void appRun(App* app) = 0; //实现App与手机的桥接
};

class XiaoMi:public MobilePhone
{
public:
    void appRun(App* app)

```

```

    {
        cout << "XiaoMi: ";
        app->run();
    }
};

class HuaWei:public MobilePhone
{
public:
    void appRun(App* app)
    {
        cout << "HuaWei: ";
        app->run();
    }
};

int main()
{
    App* gameApp = new GameApp;
    App* translateApp = new TranslateApp;
    MobilePhone* mi = new XiaoMi;
    MobilePhone* hua = new HuaWei;
    mi->appRun(gameApp);
    mi->appRun(translateApp);
    hua->appRun(gameApp);
    hua->appRun(translateApp);
    delete hua;
    delete mi;
    delete gameApp;
    delete translateApp;

    return 0;
}

```

 复制代码

13、装饰模式：动态地给一个对象添加一些额外的功能，就新增加功能来说，装饰器模式比生产子类更加灵活。

主要解决：通常我们为了扩展一个类经常使用继承的方式，由于继承为类引入静态特征，并且随着扩展功能的增多，子类会很膨胀。

如何解决：将具体的功能划分，同时继承装饰者类。

关键代码：装饰类复合和继承组件类，具体的扩展类重写父类的方法。

 复制代码

```

class Dumplings    //抽象类  饺子
{
public:
    virtual ~Dumplings() {}
    virtual void showDressing() = 0;
};

class MeatDumplings:public Dumplings    //现实类  肉馅饺子
{
public:
    ~MeatDumplings() { cout << "~MeatDumplings()" << endl; }
    void showDressing()
    {
        cout << "Add Meat" << endl;
    }
};

class DecoratorDumpling:public Dumplings    //装饰类
{
public:
    DecoratorDumpling(Dumplings* d):m_dumpling(d) {}
    virtual ~DecoratorDumpling() { cout << "~DecoratorDumpling()" << endl; }
    void showDressing()
    {
        m_dumpling->showDressing();
    }
private:
    Dumplings* m_dumpling;
};

class SaltDecorator:public DecoratorDumpling    // 装饰类  加盐

```

```

{
public:
    SaltDecorator(Dumplings* d):DecoratorDumpling(d){}
    ~SaltDecorator(){ cout << "~SaltDecorator()" << endl; }
    void showDressing()
    {
        DecoratorDumpling::showDressing(); //注意点
        addDressing();
    }

private:
    void addDressing()
    {
        cout << "Add Salt" << endl;
    }
};

class OilDecorator:public DecoratorDumpling //装饰类 加油
{
public:
    OilDecorator(Dumplings* d):DecoratorDumpling(d){}
    ~OilDecorator(){ cout << "~OilDecorator()" << endl; }
    void showDressing()
    {
        DecoratorDumpling::showDressing(); //注意点
        addDressing();
    }

private:
    void addDressing()
    {
        cout << "Add Oil" << endl;
    }
};

class CabbageDecorator:public DecoratorDumpling //装饰类 加蔬菜
{
public:
    CabbageDecorator(Dumplings* d):DecoratorDumpling(d){}
    ~CabbageDecorator(){ cout << "~CabbageDecorator()" << endl; }
    void showDressing()
    {
        DecoratorDumpling::showDressing(); //注意点
        addDressing();
    }

private:
    void addDressing()
    {
        cout << "Add Cabbage" << endl;
    }
};

int main()
{
    Dumplings* d = new MeatDumplings; //原始的肉饺子
    Dumplings* d1 = new SaltDecorator(d); //加盐后的饺子
    Dumplings* d2 = new OilDecorator(d1); //加油后的饺子
    Dumplings* d3 = new CabbageDecorator(d2); //加蔬菜后的饺子
    d3->showDressing();
    delete d;
    delete d1;
    delete d2;
    delete d3;
    return 0;
}

```

 复制代码

14、备忘录模式：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可以将该对象恢复到原来保存的状态。

如何解决：通过一个备忘录类专门存储对象状态。

关键代码：备忘录类、客户类、备忘录管理类；客户类不与备忘录类耦合，而是与备忘录管理类耦合。



```

typedef struct    //需要保存的信息
{
    int grade;
    string arm;
    string corps;
}GameValue;

class Memento    //备忘录类
{
public:
    Memento(){}
    Memento(GameValue value):m_gameValue(value){}
    GameValue getValue()
    {
        return m_gameValue;
    }
private:
    GameValue m_gameValue;
};

class Game    //客户类 游戏
{
public:
    Game(GameValue value):m_gameValue(value)
    {}
    void addGrade()    //等级增加
    {
        m_gameValue.grade++;
    }
    void replaceArm(string arm)    //更换武器
    {
        m_gameValue.arm = arm;
    }
    void replaceCorps(string corps)    //更换工会
    {
        m_gameValue.corps = corps;
    }
    Memento saveValue()    //保存当前信息
    {
        Memento memento(m_gameValue);
        return memento;
    }
    void load(Memento memento) //载入信息
    {
        m_gameValue = memento.getValue();
    }
    void showValue()
    {
        cout << "Grade: " << m_gameValue.grade << endl;
        cout << "Arm : " << m_gameValue.arm.data() << endl;
        cout << "Corps: " << m_gameValue.corps.data() << endl;
    }
private:
    GameValue m_gameValue;
};

class Caretake //备忘录管理类
{
public:
    void save(Memento memento)    //保存信息
    {
        m_memento = memento;
    }
    Memento load()    //读已保存的信息
    {
        return m_memento;
    }
private:
    Memento m_memento;
};

int main()
{
    GameValue v1 = {0, "Ak", "3K"};
    Game game(v1);    //初始值

```

```

game.addGrade();
game.showValue();
cout << "-----" << endl;
Caretake care;
care.save(game.saveValue()); //保存当前值
game.addGrade(); //修改当前值
game.replaceArm("M16");
game.replaceCorps("123");
game.showValue();
cout << "-----" << endl;
game.load(care.load()); //恢复初始值
game.showValue();
return 0;
}

```

 复制代码

15. 中介者模式：用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显示地相互引用，从而使其耦合松散，而且可以独立地改变它们之前的交互。

主要解决：对象与对象之前存在大量的关联关系，这样势必会造成系统变得复杂，若一个对象改变，我们常常需要跟踪与之关联的对象，并做出相应的处理。

如何解决：将网状结构分离为星型结构。

关键代码：将相关对象的通信封装到一个类中单独处理。

 复制代码

```

class Mediator;

class Person //抽象同事类
{
public:
    virtual ~Person() {}
    virtual void setMediator(Mediator* mediator)
    {
        m_mediator = mediator;
    }
    virtual void sendMessage(const string& message) = 0;
    virtual void getMessage(const string& message) = 0;
protected:
    Mediator* m_mediator;
};

class Mediator //抽象中介类
{
public:
    virtual ~Mediator() {}
    virtual void setBuyer(Person* buyer) = 0;
    virtual void setSeller(Person* seller) = 0;
    virtual void send(const string& message, Person* person) = 0;
};

class Buyer:public Person //买家类
{
public:
    void sendMessage(const string& message)
    {
        m_mediator->send(message, this);
    }
    void getMessage(const string& message)
    {
        cout << "Buyer Get: " << message.data() << endl;
    }
};

class Seller:public Person //卖家类
{
public:
    void sendMessage(const string& message)
    {
        m_mediator->send(message, this);
    }
    void getMessage(const string& message)
    {
        cout << "Seller Get: " << message.data() << endl;
    }
};

```

```
};

class HouseMediator:public Mediator //具体的中介类
{
public:
    HouseMediator():m_buyer(nullptr),m_seller(nullptr){}
    void setBuyer(Person* buyer)
    {
        m_buyer = buyer;
    }
    void setSeller(Person *seller)
    {
        m_seller = seller;
    }
    void send(const string& message, Person* person)
    {
        if(person == m_buyer)
        {
            m_seller->getMessage(message);
        }
        if(person == m_seller)
        {
            m_buyer->getMessage(message);
        }
    }
private:
    Person* m_buyer;
    Person* m_seller;
};

int main()
{
    Person* buyer = new Buyer;
    Person* seller = new Seller;
    Mediator* houseMediator = new HouseMediator;
    buyer->setMediator(houseMediator);
    seller->setMediator(houseMediator);
    houseMediator->setBuyer(buyer);
    houseMediator->setSeller(seller);
    buyer->sendMessage("1.5?");
    seller->sendMessage("2!!!");
    return 0;
}
```



16、职责链模式：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之前的耦合关系，将这些对象连成一条链，并沿着这条链传递请求，直到有一个对象处理它为止。

主要解决：职责链上的处理者负责处理请求，客户只需要将请求发送到职责链上即可，无需关心请求的处理细节和请求的传递，所有职责链将请求的发送者和请求的处理者解耦了。

如何解决：职责链链扣类都实现统一的接口。

关键代码：Handler内指明其上级，handleRequest()里判断是否合适，不合适则传递给上级。

```
复制代码
enum RequestLevel
{
    One = 1,
    Two,
    Three
};

class Leader
{
public:
    Leader(Leader* leader):m_leader(leader){}
    virtual ~Leader(){}
    virtual void handleRequest(RequestLevel level) = 0;
protected:
    Leader* m_leader;
};

class Monitor:public Leader //链扣1
{
}
```



```

public:
    Monitor(Leader* leader):Leader(leader){}
    void handleRequest(RequestLevel level)
    {
        if(level < Two)
        {
            cout << "Mointor handle request : " << level << endl;
        }
        else
        {
            m_leader->handleRequest(level);
        }
    }
};

class Captain:public Leader    //链扣2
{
public:
    Captain(Leader* leader):Leader(leader){}
    void handleRequest(RequestLevel level)
    {
        if(level < Three)
        {
            cout << "Captain handle request : " << level << endl;
        }
        else
        {
            m_leader->handleRequest(level);
        }
    }
};

class General:public Leader    //链扣3
{
public:
    General(Leader* leader):Leader(leader){}
    void handleRequest(RequestLevel level)
    {
        cout << "General handle request : " << level << endl;
    }
};

int main()
{
    Leader* general = new General(nullptr);
    Leader* captain = new Captain(general);
    Leader* monitor = new Monitor(captain);
    monitor->handleRequest(Three);
    return 0;
}

```



17、观察者模式：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都要得到通知并自动更新。

主要解决：一个对象更新，其它对象也要更新。

如何解决：目标类通知函数通知所有观察者自动更新。

关键代码：在目标类中增加一个ArrayList来存放观察者们。



//数据模型为目标类，视图为观察者类。当数据模型发生改变时，通知视图类更新

```

class View;

class DataModel    //目标抽象类    数据模型
{
public:
    virtual ~DataModel(){}
    virtual void add(View* view) = 0;
    virtual void remove(View* view) = 0;
    virtual void notify() = 0;    //通知函数
};

class View    //观察者抽象类    视图
{

```

```

public:
    virtual ~View(){ cout << "~View()" << endl; }
    virtual void update() = 0;
};

class IntModel:public DataModel    //具体的目标类, 整数模型
{
public:
    ~IntModel()
    {
        clear();
    }
    void add(View* view)
    {
        auto iter = std::find(m_list.begin(), m_list.end(), view); //判断是否重复添加
        if(iter == m_list.end())
        {
            m_list.push_back(view);
        }
    }
    void remove(View* view)
    {
        auto iter = m_list.begin();
        for(;iter != m_list.end(); iter++)
        {
            if(*iter == view)
            {
                delete *iter;    //释放内存
                m_list.erase(iter); //删除元素
                break;
            }
        }
    }
    void notify() //通知观察者更新
    {
        auto iter = m_list.begin();
        for(; iter != m_list.end(); iter++)
        {
            (*iter)->update();
        }
    }
private:
    void clear()
    {
        if(!m_list.empty())
        {
            auto iter = m_list.begin();
            for(;iter != m_list.end(); iter++) //释放内存
            {
                delete *iter;
            }
        }
    }
private:
    list<View*> m_list;
};

class TreeView:public View //具体的观察者类 视图
{
public:
    TreeView(string name):m_name(name),View(){}
    ~TreeView(){ cout << "~TreeView()" << endl; }
    void update()
    {
        cout << m_name.data() << " : Update" << endl;
    }
private:
    string m_name;
};

int main()
{
    View* v1 = new TreeView("view1");
    View* v2 = new TreeView("view2");
    View* v3 = new TreeView("view3");
    View* v4 = new TreeView("view4");
    DataModel* model = new IntModel;

```

```
model->add(v1);  
model->add(v2);  
model->add(v3);  
model->add(v2);  
model->add(v4);  
model->notify();  
cout << "-----" << endl;  
model->remove(v2);  
model->notify();  
delete model;  
return 0;  
}
```

 复制代码

posted @ 2018-02-26 15:42 [小肚哥](#) [阅读\(...\)](#) [评论\(...\)](#) [编辑](#) [收藏](#)

努力加载评论中...

[刷新评论](#) [刷新页面](#) [返回顶部](#)

Copyright ©2019 小肚哥