设计模式

“每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的解决方案的核心。这样，你就能一次又一次地使用该方案而不必做重复劳动”。
当客户改变需求，可以很快改变

- 理解松耦合设计思想
- 掌握面向对象设计原则
- 掌握重构技法改善设计
- 掌握GOF 核心设计模式

设计模式目标：可复用
具体方法：面向对象

复用性：真正的复用性不是代码复用，是**编译单位**的复用性，二进制的复用性，在原来的代码后面加几行不是复用之前的代码，已经破坏了代码的结构，也可能会引起bug。
程序员要想到以后代码会扩展的情况。

如何解决 复杂性 ？
分解
人们面对复杂性有一个常见的做法：即 分而治之 ，将大问题分解为多个小问题，将复杂问题分解为多个简单问题。
抽象
更高层次来讲，人们处理复杂性有一个通用的技术，即抽象。由于不能掌握全部的复杂对象，我们选择忽视它的非本质细节，而去处理泛化和理想化了的对象模型。

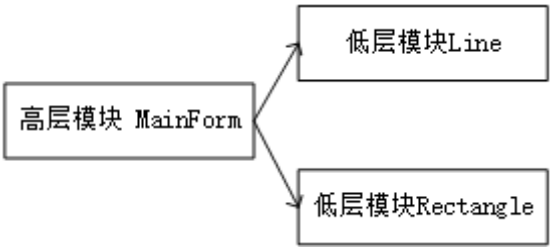
重新认识面向对象

- 理解隔离变化
  - 从宏观层面来看，面向对象的构建方式更能适应软件的变化，能将变化所带来的影响减为最小
- 各司其职
  - 从微观层面来看，面向对象的方式更强调各个类的“责任”
  - 由于需求变化导致的新增类型不应该影响原来类型的实现——是所谓各负其责
- 对象是什么？
  - 从语言实现层面来看，对象封装了代码和数据。
  - 从规格层面讲，对象是一系列可被使用的公共接口。
  - 从概念层面讲，对象是某种拥有责任的抽象。

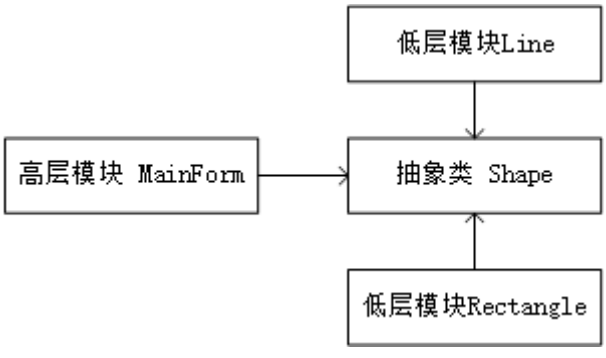
多态可以使得接口保持一致，新增不同的设计方法。

面向对象设计原则

- 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)。



修改低层会影响高层



抽象不变

- 开放封闭原则（OCP）(开闭原则)
  - 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)
  - 一个类已更改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),里氏代换原则,is a的另外一种表达方式
  - 子类必须能够替换他们的基类(IS-A)，(设计子类不应该不能使用父类功能)
  - 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他们不用的方法
  - 接口应该小而完备
- 优先使用对象组合，和不是类继承
  - 类继承通常为"白箱复用"，对象组合通常为“黑箱复用”。
  - 继承在某种程度上破坏了封装性(父类给子类暴露的较多)，子类父类耦合度高
  - 而对象组合则只要求被组合的对象具有良好定义的接口，耦合度低。 (有点像我在调制识别时写的helper类，这个类和识别类不是继承关系，写成对象组合，即一个类里面定义了另一个类)
- 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进行修改，而不会对另一侧产生不良的影响，从而是实现层次间的松耦合。
    封装的作用不仅仅是封装代码和数据，更高层次的理解是封装变化和稳定，一侧变化，一侧稳定。(参考依赖倒置原则的两个图，感觉项目中通过一个大的信号生成类去隔离各种调制信号生成方式类就是这种思想)
- 针对接口编程，而不是针对实现编程(依赖倒置原则的另一个角度。往往违背其中一个原则，另一个原则也会违背)
  - 不讲变量声明为某个特定的具体类，而是声明为某个接口。
  - 客户程序无需获知对象的具体类型，只需要知道对象所具有的接口。
  - 减少系统中各部分的依赖关系，从而实现 “高内聚，低耦合”

违背接口原则

```
vector<Line> lineVector;
vector<Rect> rectVector;
```

面向接口的设计

```
vector<Shape*> shapeVector;//放抽象接口
...
shapeVector[i]->Draw(e.Graphics);//多态调用，各负其值
```

面向接口设计

产业强盛的标志，接口设计是一种专门的工作，可以分工协作，才能实现复用性。

GOF-23 模式分类

- 从目的来看：
  - 创建型（Creational）模式：将对象的部分创建工作延迟到子类或者其他对象，从而应对需求变化为对象创建时具体类型实现引来的冲击。
  - 结构型（Structural）模式：通过类继承或者对象组合获得更灵活的结构，从而应对需求变化为对象的结构带来的冲击。
  - 行为型（Behavioral）模式：通过类继承或者对象组合来划分类与对象间的职责，从而应对需求变化为多个交互的对象带来的冲击。
- 从范围来看：
  - 类模式处理类与子类的静态关系。
  - 对象模式处理对象间的动态关系。

2

从封装变化角度对模式分类

- 组件协作：
  - Template Method
  - Observer / Event
  - Strategy
- 单一职责：
  - Decorator
  - Bridge
- 对象创建：
  - Factory Method
  - Abstract Factory
  - Prototype
  - Builder
- 对象性能：
  - Singleton
  - Flyweight
- 接口隔离：
  - Façade
  - Proxy
  - Mediator
  - Adapter
- 状态变化：
  - Memento
  - State
- 数据结构：
  - Composite
  - Iterator
  - Chain of Resposibility
- 行为变化：
  - Command
  - Visitor
- 领域问题：
  - Interpreter

3

重构获得模式 Refactoring to Patterns

- 面向对象设计模式是“好的面向对象设计”，所谓“好的面向对象设计”指是那些可以满足 “应对变化，提高复用”的设计 。
- 现代软件设计的特征是“需求的频繁变化”。设计模式的要点是“寻找变化点，然后在变化点处应用设计模式，从而来更好地应对需求的变化”.“什么时候、什么地点应用设计模式”比“理解设计模式结构本身”更为重要。
- 设计模式的应用不宜先入为主，一上来就使用设计模式是对设计模式的最大误用。没有一步到位的设计模式。敏捷软件开发实践提倡的“Refactoring to Patterns”是目前普遍公认的最好的使用设计模式的方法。

4

推荐图书

重构关键技法

- 静态->动态
- 早绑定->晚绑定
- 继承->组合
- 编译时依赖->运行时依赖
- 紧耦合->松耦合

设计模式就是在稳定和和变化中找到隔离点。

https://blog.csdn.net/huashuolin001/article/details/108258802

https://www.jianshu.com/p/69eef7651667
https://www.runoob.com/design-pattern/design-pattern-intro.html

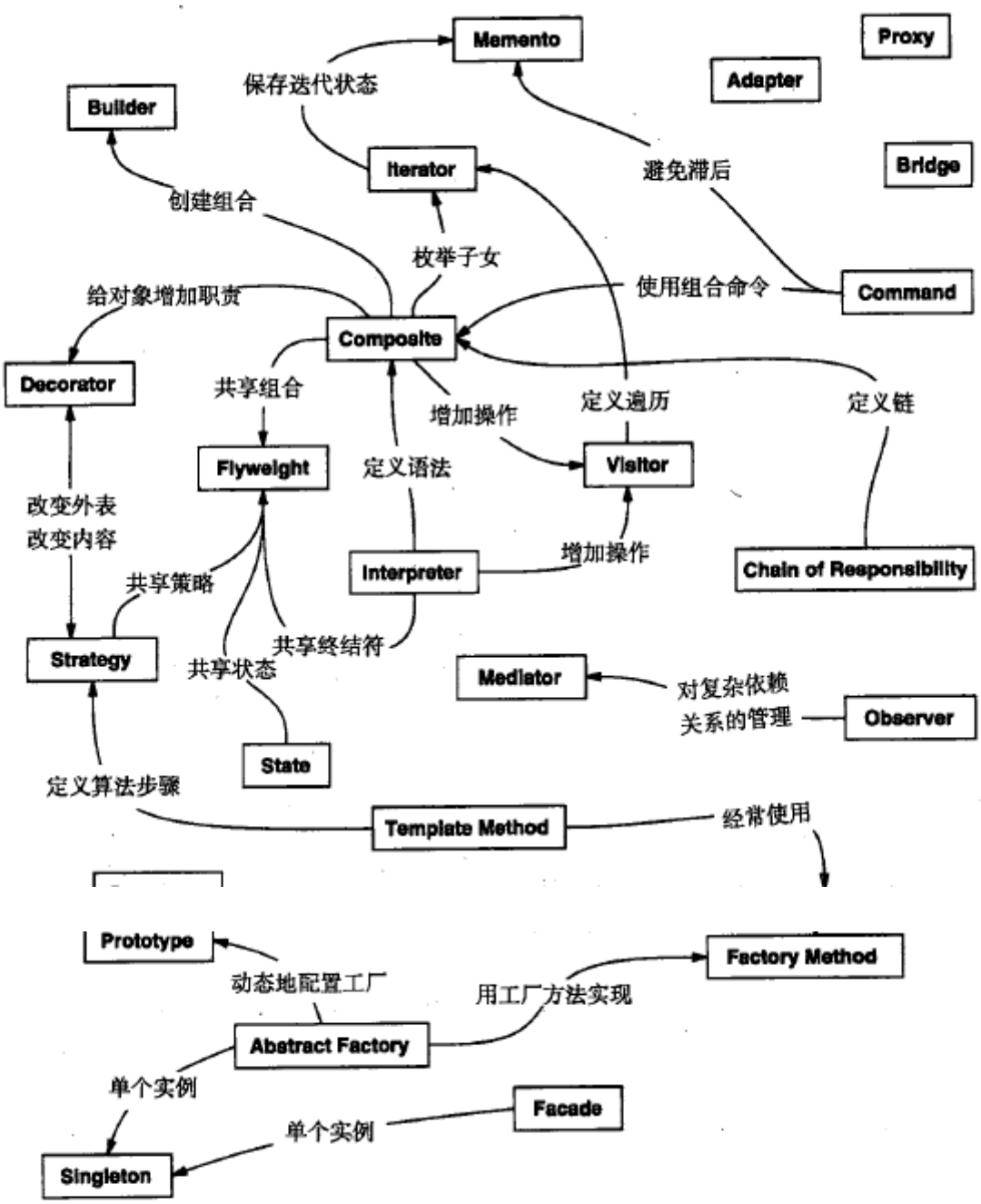


图1-1 设计模式之间的关系

单例模式-创建型模式

https://blog.csdn.net/sinat_21107433/article/details/102649056

如window任务管理器，回收站，多线程线程池，文件系统，一个数字滤波器只有一个AD转换器，一个会计系统专用于一个公司

- 一个类只有一个实例，该类能自行创建实例

- 类的构造函数(注意析构函数不影响)设为私有的，外部类就无法调用该构造函数，也就无法生成多个实例。

写法：

- 懒汉式：
  - 线程不安全
  - 加锁(代价过高)
  - 双检测锁(reorder:编译器指令重排)
  - C++11双检测锁（m_mutex）
  - C++11局部静态变量(Effictive C++的作者提出的)
- 饿汉式
  - 线程安全

线程不安全的写法
懒汉式:

```
class Singleton
{
private:
        static Singleton* m_instance;//这里是类指针
        Singleton() {};//不能delete，否则getInstance调用不了
        Singleton(const Singleton& other) = delete;//不希望有拷贝构造
public:
        static Singleton* getInstance();//获得实例
};
Singleton* Singleton::m_instance = nullptr;//开头不用写static
Singleton* Singleton::getInstance()//这里开头也不用写static
{
        if (m_instance == nullptr)//懒汉：使用时创建
        {
                m_instance = new Singleton();
        }
        return m_instance;
}
```

线程安全的写法(代价过高)

```
Singleton* Singleton::getInstance() {
    Lock lock;//
    if (m_instance == nullptr) {
        m_instance = new Singleton();
    }
    return m_instance;
}
```

双检查锁（错误的写法）

```
////双检查锁，但由于内存读写reorder不安全，出问题概率很高，不能这么用
//假想顺序
//step1 :分配内存
//step2：构造
//step3：返回指针
//编译器reorder：CPU指令级(有可能，编译优化)：
//step1 : 分配内存
//step2：返回指针
//step3：构造
Singleton* Singleton::getInstance() {

    if(m_instance==nullptr){//第一个判空为了判断是第一次创建
        Lock lock;//两个线程只有一个能先锁，不能都锁上
        if (m_instance == nullptr) {//第二个判空是为了当两个线程同时判空后，一个先锁，让一个先new出
            m_instance = new Singleton();//可能在某个线程new的时候未分配内存就返回指针，导致其他线
        }
    }
    return m_instance;
}
```

C++11双检测锁,内存栅栏

```
//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;
}
```

局部静态变量(最好的写法)

https://blog.csdn.net/wang_anna/article/details/103761565

```
//类内
static Singleton* getInstance()
{
    static Singleton m_instance;//是个对象
    return &m_instance;//返回地址
}
```

饿汉式，线程安全

该模式的特点是类一旦加载就创建一个单例，保证在调用 getInstance 方法之前单例已经存在了。

不管是不是用都会初始化，浪费内存

```
//1.在类中new
//2.在类外new
Singleton* Singleton::m_instance = new Singleton();
```

要点总结

- Singleton模式中的实例构造器可以设置为protected以允许子类派生。
- Singleton模式一般不要支持拷贝构造函数和Clone接口，因为这有可能导致多个对象实例，与Singleton模式的初衷违背。
- 如何实现多线程环境下安全的Singleton？注意对双检查锁的正确实现。

模板模式(template Pattern)-类行为型模式

https://blog.csdn.net/sinat_21107433/article/details/102994585

就是在基类中把算法的框架搭好，在子类中去实现。框架是不变的部分，将可变的行为留给子类。变的部分写成虚函数，子类重写实现虚函数。

非模板模式

```
//程序库开发人员
class Library{
public:
    void Step1(){
                //...
        }
    void Step3(){
                //...
    }
    void Step5(){
                //...
    }
};
```

```
//应用程序开发人员
class Application{
public:
    bool Step2(){
                //...
    }
    void Step4(){
                //...
    }
};

int main(){
        Library lib();
        Application app();
        lib.Step1();
        if (app.Step2()){
                lib.Step3();
        }
        for (int i = 0; i < 4; i++){
                app.Step4();
        }
        lib.Step5();
}
```

模板模式

```
//程序库开发人员
class Library{
public:
    //稳定 template method
    void Run(){
        Step1();
        if (Step2()) { //支持变化 ==> 虚函数的多态调用
            Step3();
        }
        for (int i = 0; i < 4; i++){
            Step4(); //支持变化 ==> 虚函数的多态调用
        }
        Step5();
    }
        virtual ~Library(){ }

protected:
    void Step1() { //稳定
        //.....
    }
    void Step3() {//稳定
        //.....
    }
    void Step5() { //稳定
                //.....
    }
    virtual bool Step2() = 0;//变化
    virtual void Step4() =0; //变化
};


//应用程序开发人员
class Application : public Library {
protected:
        virtual bool Step2(){
                //... 子类重写实现
    }

    virtual void Step4() {
                //... 子类重写实现
    }
};
int main()
{
    Library* pLib=new Application();
    pLib->Run();
    delete pLib;
}
```

模板方法模式是基于类的继承的一种设计模式，使用非常频繁，被广泛应用于框架设计。

优点：

- 在基类中定义算法的框架，并声明一些流程方法，由具体派生类实现细节，派生类中的实现并不会影响基类定义的算法的框架流程；
- 公共行为在基类中提供实现，有利于代码复用；
- 派生类可以覆盖基类的方法，重新实现某些方法，具有灵活性；
  可以很方便的扩展和更换派生类而不影响基类和其他派生类，符合开闭原则和单一职责原则。

缺点：

- 模板方法模式要为每一个不同的基本方法提供一个派生类，如果基类中基本方法很多，那系统中会定义很多个派生类，导致类的个数很多，系统更加庞大。

工厂模式(Factory Method)对象创建型模式

别名：虚构造器

- “对象创建”模式
  通过“对象创建” 模式绕开new，来避免对象创建（new）过程中所导致的紧耦合（依赖具体类），从而支持对象创建的稳定。它是接口抽象之后的第一步工作。

定义一个用于创建对象的接口，让子类决定实例化哪一个类，使一个类的实例化延迟(目的：解耦，手段：虚函数 `virtual`)到子类.

一般适用于不确定new后面的类是什么类型，或者希望由子类指定所创建的对象的时候。

优点： 1、一个调用者想创建一个对象，只要知道其名称就可以了。 2、扩展性高，如果想增加一个产品，只要扩展一个工厂类就可以。 3、屏蔽产品的具体实现，调用者只关心产品的接口。

缺点：每次增加一个产品时，都需要增加一个具体类和对象实现工厂，使得系统中类的个数成倍增加，在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖。这并不是什么好事

```
//抽象类
class ISplitter{
public:
    virtual void split()=0;
    virtual ~ISplitter(){}
};


//工厂基类
class SplitterFactory{
public:
    virtual ISplitter* CreateSplitter()=0;//返回new出来的对象指针
    //virtual ISplitter1* CreateSplitter()=0;如果有多个ISplitter并且相互有联系，可以都在这里写成虚i
    //virtual ISplitter2* CreateSplitter()=0;
    virtual ~SplitterFactory(){}
};
```

```
//具体类
class BinarySplitter : public ISplitter{

};

class TxtSplitter: public ISplitter{

};

class PictureSplitter: public ISplitter{

};

class VideoSplitter: public ISplitter{

};

//具体工厂
class BinarySplitterFactory: public SplitterFactory{
public:
    virtual ISplitter* CreateSplitter(){
        return new BinarySplitter();
    }
};

class TxtSplitterFactory: public SplitterFactory{
public:
    virtual ISplitter* CreateSplitter(){
        return new TxtSplitter();
    }
};

class PictureSplitterFactory: public SplitterFactory{
public:
    virtual ISplitter* CreateSplitter(){
        return new PictureSplitter();
    }
};

class VideoSplitterFactory: public SplitterFactory{
public:
    virtual ISplitter* CreateSplitter(){
        return new VideoSplitter();
    }
};
```

```
class MainForm : public Form
{
    SplitterFactory*  factory;//工厂
public:

    MainForm(SplitterFactory*  factory){//SplitterFactory是基类，传进来子类
        this->factory=factory;
    }

    void Button1_Click(){
        ISplitter * splitter=
            factory->CreateSplitter(); //多态new  ，返回一个ISplitter的子类，new隐藏在CreateSplitt
        splitter->split();
    }
};
```

抽象工厂 -对象创建型

和工厂模式十分相似，解决的问题也差不多。工厂模式是抽象工厂的特例。当工厂基类中的要扩展的虚函数只有一个时，就是工厂模式，多个时，就是抽像工厂，因为这多个虚函数之间有相互依赖性，如果定义多个基类，不好处理。
高内聚，松耦合。

- “对象创建”模式
  通过“对象创建” 模式绕开new，来避免对象创建（new）过程中所导致的紧耦合（依赖具体类），从而支持对象创建的稳定。它是接口抽象之后的第一步工作。
- 动机（Motivation）
  - 在软件系统中，经常面临着 **“一系列相互依赖的对象”** (和工厂模式的区别)的创建工作；同时，由于需求的变化，往往存在更多系列对象的创建工作。
  - 如何应对这种变化？如何绕过常规的对象创建方法(new)，提供一种“封装机制”来避免客户程序和这种“多系列具体对象创建工作”的紧耦合？

GOF定义-提供一个接口，让该接口负责创建一系列“相关或者相互依赖的对象”，无需指定它们具体的类。

要点总结

- 如果没有应对“多系列对象构建”的需求变化，则没有必要使用Abstract Factory模式，这时候使用简单的工厂完全可以。
- “系列对象”指的是在某一特定系列下的对象之间有相互依赖、或作用的关系。不同系列的对象之间不能相互依赖。
- Abstract Factory模式主要在于应对“新系列”的需求变动。其缺点在于难以应对“新对象”的需求变动。

策略模式-组件协作模式

- “组件协作”模式:
  现代软件专业分工之后的第一个结果是“框架与应用程序的划分”，“组件协作”模式通过晚期绑定，来实现框架与应用程序之间的松耦合，是二者之间协作时常用的模式。
- 动机（Motivation）
  - 在软件构建过程中，某些对象使用的算法可能多种多样，经常改变，如果将这些算法都编码到对象中，将会使对象变得异常复杂；而且有时候支持不使用的算法也是一个性能负担(如果使用if else，那么所有的代码都会放到代码段里，大量代码可能造成负担，装载到CPU的高级缓存等等被迫挤到主存或内存里，而写成多态的，就没问题))。
  - 如何在运行时根据需要透明地更改对象的算法？将算法与对象本身解耦，从而避免上述问题？

违背开闭原则

```
enum TaxBase {
        CN_Tax,
        US_Tax,
        DE_Tax,
        FR_Tax       //更改
};

class SalesOrder{
    TaxBase tax;
public:
    double CalculateTax(){
        //...

        if (tax == CN_Tax){
            //CN***********
        }
        else if (tax == US_Tax){
            //US***********
        }
        else if (tax == DE_Tax){
            //DE***********
        }
        else if (tax == FR_Tax){
            //更改
            //...
        }

            //....
    }

};
```

策略模式

```
class TaxStrategy{
public:
    virtual double Calculate(const Context& context)=0;
    virtual ~TaxStrategy(){}
};


class CNTax : public TaxStrategy{
public:
    virtual double Calculate(const Context& context){
        //***********
    }
};

class USTax : public TaxStrategy{
public:
    virtual double Calculate(const Context& context){
        //***********
    }
};

class DETax : public TaxStrategy{
public:
    virtual double Calculate(const Context& context){
        //***********
    }
};



//扩展
//*********************************
class FRTax : public TaxStrategy{
public:
        virtual double Calculate(const Context& context){
                //.........
        }
};


class SalesOrder{
private:
    TaxStrategy* strategy;

public:
    SalesOrder(StrategyFactory* strategyFactory){
        this->strategy = strategyFactory->NewStrategy();
    }
    ~SalesOrder(){
        delete this->strategy;
    }
```

```
    public double CalculateTax(){
        //...
        Context context();

        double val =
            strategy->Calculate(context); //多态调用
        //...
    }

};
```