

设计模式

设计模式

设计模式的八个原则：

重构关键技法

工厂模式

抽象工厂模式

单例模式

下面三个是组件协作类型的设计模式

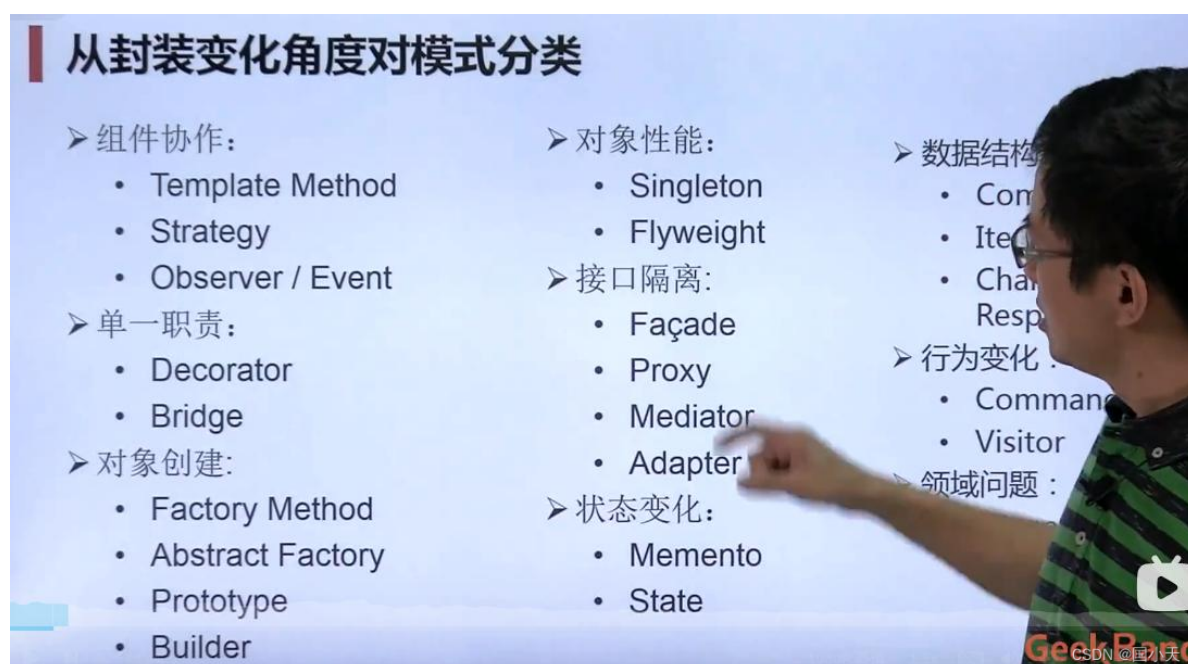
模板方法

策略模式

观察者模式

设计模式的八个原则：

1. 依赖倒置原则：高层次的代码（稳定）不应该依赖低层次的代码（变化）、抽象的代码不应该依赖具体的代码。
2. 开放封闭原则：类模块应该开放扩展的，而其原先的代码尽量封闭不可改变。
3. 单一职责原则：一个类应该仅有一个变化的原因，该变化隐含了它的职责，职责太多时会导致扩展时对代码东拉西扯，造成混乱。
4. 替换原则：子类必须能够替换它的基类（IS-A），继承可以表达类型抽象。
5. 接口隔离原则：接口应该小而完备，不该强迫用户使用多余的方法。
6. 优先使用组合而不是继承：继承通常会让子类 and 父类的耦合度增加、组合的方式只要求组件具备良好定义的接口。
7. 封装变化点：
8. 针对接口编程，而不是针对实现编程。



从封装变化角度对模式分类

- 组件协作：
 - Template Method
 - Strategy
 - Observer / Event
- 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：
 - Command
 - Iterator
 - Chain of Responsibility
- 行为变化：
 - Command
 - Visitor
- 领域问题：

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

工厂模式

Factory Method Abstract Factory Prototype Builder

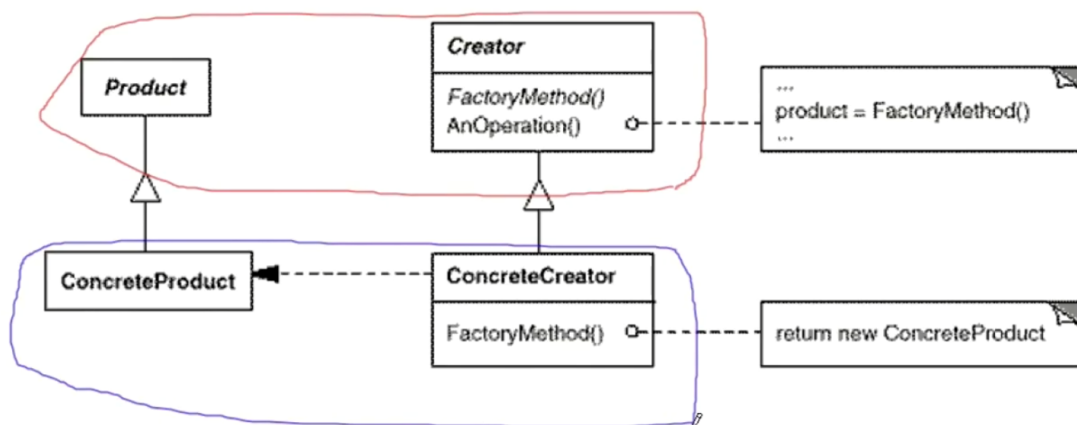
绕开new，避免对象创建过程中所导致的紧耦合，从而支持对象创建的稳定，是接口抽象之后的第一步工作

new过程会依赖具体的类，所以要避开。

通过虚函数实现一个多态的new。

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到子类。（目的是解耦 手段是虚函数）。

结构 (Structure)



红色的是稳定的部分，蓝色的是不稳定的部分。

能够解决单个对象的需求变化，缺点在于要求创建的方法或参数要相同。

```
#include <iostream>
using namespace std;

// 抽象产品类
class Product {
public:
    virtual void operation() = 0;
};

// 具体产品类A
class ConcreteProductA : public Product {
public:
    void operation() override {
        // 具体产品A的操作
        cout<<"A operation"<<endl;
    }
};
```

```

// 具体产品类B
class ConcreteProductB : public Product {
public:
    void operation() override {
        // 具体产品B的操作
        cout<<"B operation"<<endl;
    }
};

// 抽象工厂类
class Factory {
public:
    virtual Product* createProduct() = 0;
};

// 具体工厂类A
class ConcreteFactoryA : public Factory {
public:
    Product* createProduct() override {
        cout<<"A createProduct"<<endl;
        return new ConcreteProductA();
    }
};

// 具体工厂类B
class ConcreteFactoryB : public Factory {
public:
    Product* createProduct() override {
        cout<<"B createProduct"<<endl;
        return new ConcreteProductB();
    }
};

// 客户端代码
//在上述示例中，抽象产品类Product定义了产品的公共接口，
// 具体的产品类ConcreteProductA和ConcreteProductB
// 实现了该接口。抽象工厂类Factory定义了创建产品的接口，
// 具体的工厂类ConcreteFactoryA和ConcreteFactoryB
// 实现了该接口，并分别创建具体的产品。
int main() {
    Factory* factory = new ConcreteFactoryA(); // 或者可以使用工厂方法来创建具体的工厂
    类
    Product* product = factory->createProduct();
    product->operation();

    delete product;
    delete factory;

    return 0;
}

```

抽象工厂模式

数据访问层需要创建一系列的对象，需要创建reader，writer等等不同的对象。比如调用不同的数据库的时候，需要绑定不同的数据库类型，这里就需要绕开new。

引入一个抽象工厂类和多个具体工厂类，使得客户端代码与具体工厂类解耦。每个具体工厂类负责创建一组相关的产品，产品之间有依赖关系。

最本质的区别就是：抽象工厂创建的是一组相关的对象。

单例模式

Singleton Flyweight

抽象来实现面向对象。

动机

- 特殊的类，必须保证在系统中只存在一个实例，才能保证逻辑正确性以及良好的效率
- 如何绕过常规的构造器，提供一种机制来保证一个类只有一个实例

```
class Singleton{
    // 这里要明确写在private中
private:
    Singleton();
    Singleton(const Singleton & other);
public:
    // 实例构造器可以是protected 方便子类创建
    static Singleton* getInstance();
    static Singleton* m_instance;
}
// 静态变量时堆对象
Singleton* Singleton::m_instance=nullptr;

// 线程非安全版本
Singleton* Singleton::getInstance(){
    // 多线程情况下多次执行
    if(m_instance == nullptr){
        m_instance = new Singleton();
    }
    return m_instance;
}

// 线程安全版本 但锁的代价过高
// 如果已经创建了 再访问就浪费 读的情况下就不用加锁
Singleton* Singleton::getInstance(){
    // 多线程情况下加锁 等待执行
    Lock lock;
    if(m_instance == nullptr){
        m_instance = new Singleton();
    }
    return m_instance;
}

// 双检查锁 但由于内存读写reorder不安全
Singleton* Singleton::getInstance(){
    // 避免读取时候加锁
    if(m_instance == nullptr){
        Lock lock;
        // 这里的判断就是担心在35之后36行之前 有多个线程进来，锁前锁后都要检查一遍
        if(m_instance == nullptr){
```

```

        m_instance = new Singleton();
    }
}
return m_instance;
}

```

// reorder: 代码会按照指令序列执行，但是到了指令层的时候，指令执行不一定是这样

```

m_instance = new Singleton();

```

先分配内存，再调用构造器，最后赋值返回

但是实际上可能是先分配内存，直接赋值返回，然后再调用构造器。

threadA进去构造，但是还没有调用构造器，此时threadB进来判断为nullptr，立马返回，但其实还没有构造，还不能用了。

// 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;
}

```

C++11的单例模式

懒汉模式:

```

class A{
    A() {}
    A(const A& a)=delete;
    A& operator=(const A& a)=delete;

    // 懒汉模式的声明 声明的时候就已经初始化
    static A& GetInstance()
    {
        static A a ;
        return a;
    }
}

```

// 饿汉模式的声明 调用这个函数的时候才进行初始化 但是会有重入的问题 这里使用call_once进行初始化

```

static A& GetInstance()
{
    static A* ptr = nullptr;
    if(!ptr)
    {
        ptr = new A;
    }
}

```

```
}  
    return *ptr;  
}  
}
```

下面三个是组件协作类型的设计模式

框架和应用之间的关系

模板方法

类模式和对象方式：类强调继承，对象强调组合。

早绑定：晚写的代码调用早写的代码

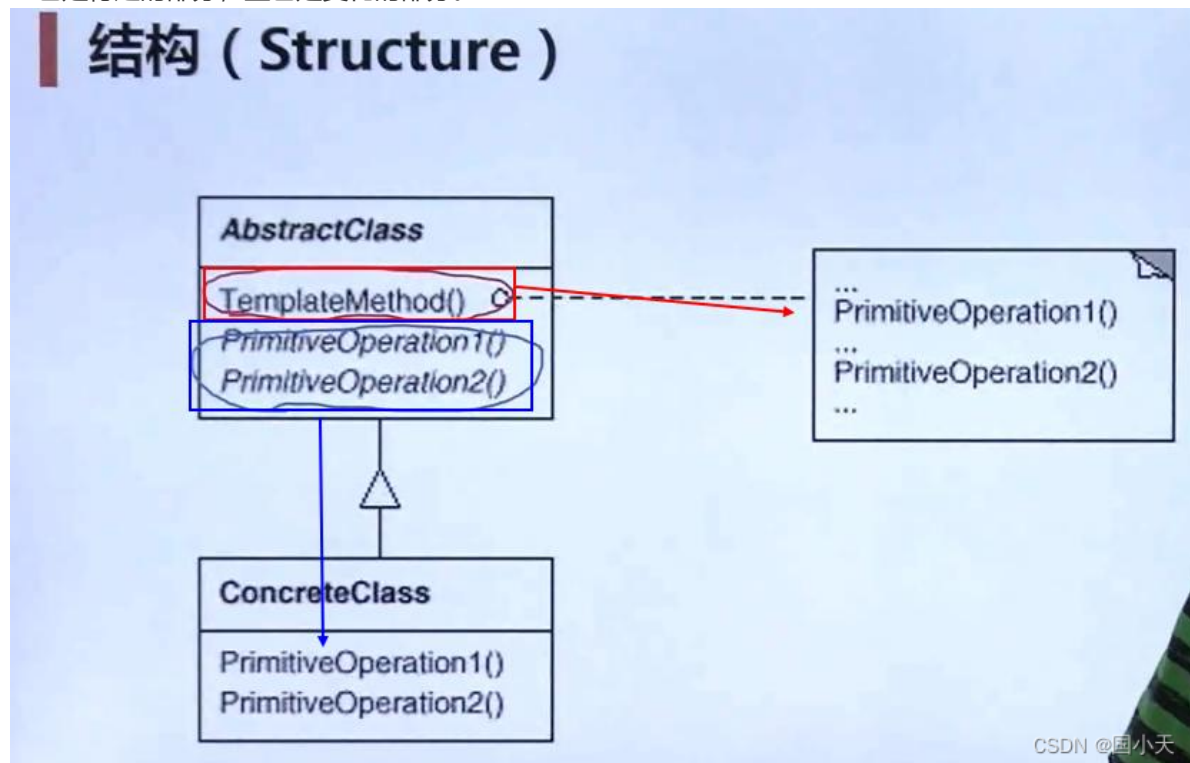
晚绑定：早写的代码调用晚写的代码，延迟到子类，支持子类的变化。稳定中有变化。

使用虚函数实现晚绑定，或者函数指针。

让库函数开发人员调用应用开发人员的代码。

有一个稳定的点，寻找出合适的隔离点。

红色是稳定的部分，蓝色是变化的部分。



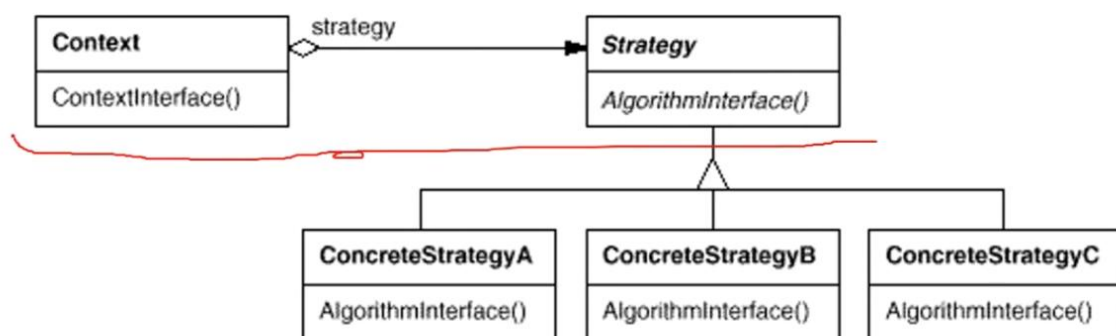
策略模式

扩充的时候能够不修改源代码，创建新的类去计算，剩下的不变。比如税的计算方法，增加不同的国家对于会扩充税费的计算方法，但是只需要继承之前的策略实现新的策略就能够实现扩展。

枚举类型增加的时候需要修改代码

策略是可变的，但是基类的strategy和context是不变的，能够实现扩展且保持源代码稳定。

结构 (Structure)



CSDN @国小天

注意：如果有 if-else 或者 switch-case 的情况就很有可能使用策略模式。从使用角度将，一些不使用的判断就会耗费资源。占据缓存内存等，对性能问题有顺带的好处。

观察者模式

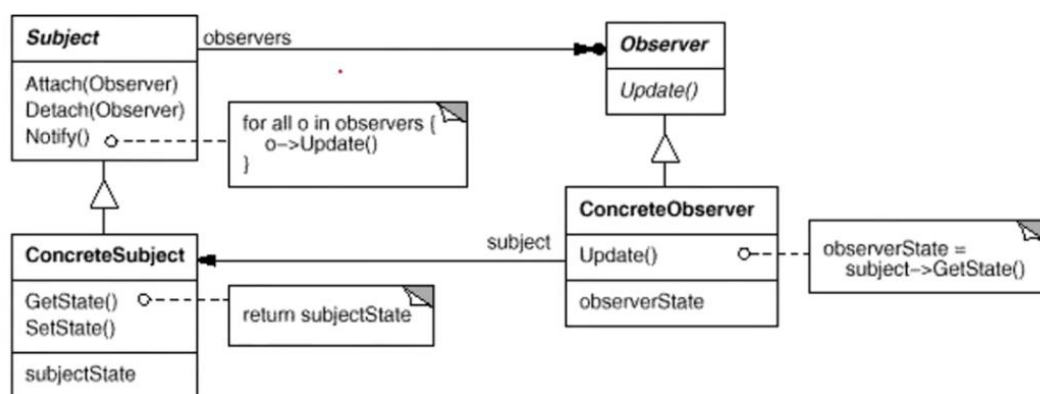
一对多的依赖关系，一个变化有多个更新。

`addObserver(xxx)`

随意的添加观察者的数量，目标发送通知时无需指定观察者，通过通知机制自动执行。

`OnProgress()` 就是通知，`addObserver(xxx)` 就是订阅，可以 `add` 任意个观察者，目标无需考虑观察者，目标对象对此一无所知。

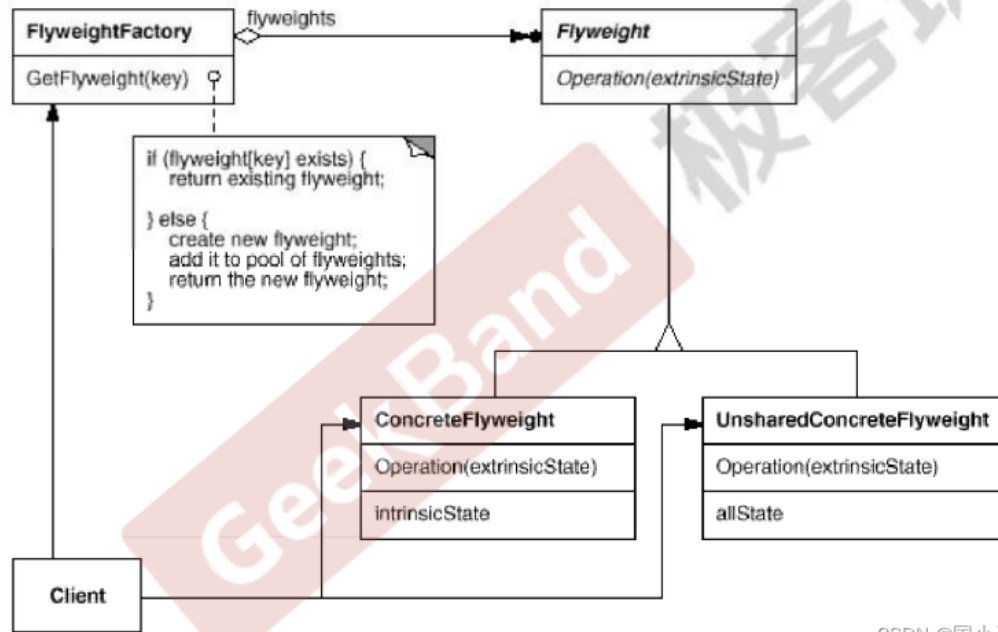
结构 (Structure)



CSDN @国小天

对象复用：[享元模式](#) Flyweight：对于大量存在的细粒度对象，会产生很大的内存消耗。维护一个对象池，当想要的对象在池中时，就不再创建，如果没有就新建并加入，有点像线程池。

结构 (Structure)



CSDN @国小天

```
class FlyweightFactory
{
public:
    ~FlyweightFactory()
    {
        for ( auto it = flies.begin(); it != flies.end(); it++ )
        {
            delete it->second;
        }
        flies.clear();
    }

    Flyweight *getFlyweight( const int key )
    {
        if ( flies.find( key ) != flies.end() )
        {
            return flies[ key ];
        }
        Flyweight *fly = new ConcreteFlyweight( key );
        flies.insert( std::pair<int, Flyweight*>( key, fly ) );
        return fly;
    }
    // ...

private:
    std::map<int, Flyweight*> flies;
    // ...
};
```