一、 设计模式

1. 设计模式介绍

> 设计模式（Design pattern）代表了最佳的实践，通常被有经验的面向对象的软件开发人员所采用。设计模式是软件开发人员在软件开发过程中面临的一般问题的解决方案。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的。
>
> 使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性、程序的重用性。

- 创建型设计模式

> 与对象相关（5个）： 单例模式， 工厂方法模式， 抽象工厂模式， 建造者模式， 原型模式 。

- 结构型设计模式

> 从结构上解决模块之间的耦合问题 （7个）：
> 适配器模式， 装饰者模式， 代理模式， 外观模式， 桥接模式， 组合模式， 享元模式 。

- 行为型设计模式

> 处理类或者对象如何交互，如何分配职责（11个）：
> 策略模式， 模板方法模式， 观察者模式， 迭代器模式， 责任链模式， 命令模式， 备忘录模式， 状态模式， 访问者模式， 中介者模式， 解释器模式 。

2. 单例模式

> 所谓单例,就是整个程序有且仅有一个实例。该类负责创建自己的对象,同时确保只有一个对象被创建
>
> 单例模式的作用主要是为了避免创建多个实例，目的是为了产生全局唯一的一个实例。此处的全局表示的是进程内部，多个进程肯定有自己多个实例，因为操作系统按照进程来划分内存。
>
> 单例又分为饿汉式和懒汉式。饿汉式则是一上来则创建对象，而懒汉式是只有使用到的时候才会创建对象。

1. 懒汉式

> 懒汉式的意思是： 只有到来获取对象的时候才急忙的去创建对象

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

class stu {
public:
    static stu * getInstance() {
        if (instance == nullptr)
            instance = new stu();
        return instance;
    }

private:
    stu() {};
    static stu * instance;
};

//静态成员需要在外面初始化。这里不创建对象。
stu * stu::instance = nullptr;

int main() {

    stu *s1 =   stu::getInstance();
    stu *s2 =   stu::getInstance();

    //同一个实例
    cout <<( s1 == s2 ) << endl;

    return 0;
}
```

2. 饿汉式

> 饿汉式的写法有很多种，此处使用堆创建的方式演示。 饿汉式属于线程安全。

```
#include <iostream>

using namespace std;

class stu {
public:
    static stu * getInstance() {
        return instance;
    }

    ~stu(){
        cout << "执行析构函数" << endl;
    }
private :
    stu() {
        cout << "执行构造函数" << endl;
    };
    static stu * instance ;
};

//一上来即创建对象
stu *stu::instance  = new stu();

int main() {

    stu *s1 =   stu::getInstance();
    stu *s2 =   stu::getInstance();

    //同一个实例
    cout <<( s1 == s2 ) << endl;

    return 0;
}
```

3. 多线程问题

> 在多个线程同时并发获取对象的情况下，还是有可能出现创建多实例的情况。

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

class stu {
public:
    static stu * getInstance() {
        //每次获取实例都要检查锁，效率很低。
        m.lock();
        if (instance == nullptr){
            instance = new stu();
        }
        m.unlock();
        return instance;
    }

private:
    mutex m;
    stu() {};
    static stu * instance;
};

//静态成员需要在外面初始化。
stu * stu::instance = nullptr;
```

4. 双锁检查

> 对获取实例的函数进行优化，执行双锁检查。上面的代码，每次获取实例都需要上锁，其实没有必要。
>
> 双锁检查 全称是 双重检查锁定模式， 简称叫做 DCLP 。

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

class stu {
public:
    static stu * getInstance() {
        //这里先做一层检查，主要是为了第二次来获取实例考虑
        if (instance == nullptr){
            //每次获取实例都要检查锁，效率很低。
            m.lock();
            if (instance == nullptr){
                instance = new stu();
            }
            m.unlock();
        }
        return instance;
    }

private:
    mutex m;
    stu() {};
    static stu * instance;
};

//静态成员需要在外面初始化。
stu * stu::instance = nullptr;
```

5. 指令乱序问题考虑

> 严格来说双锁检查仍然有可能出现指令乱序的后果。例如：下面这行代码

```
instance = new stu();
```
它背后的意思是这样的：

> 第一步：为stu对象分配一片内存
> 第二步：构造一个stu对象，存入已分配的内存区
> 第三步：将instance指向这片内存区

每一个步骤可以看成是一个指令，计算机在执行指令的时候，有可能顺序是 `1 - 3 - 2` 的顺序，简称指令乱序。刚好执行到3步骤的时候，线程被挂起。后来的线程发现instance 不为空，则直接返回。但是此时instance指向的空间并没有创建出来stu对象。做一个极端的假设，如果第一个线程此时一直被挂起，后来的线程此时已经去访问类中成员了，那么将会发生可怕的后果。

6. 如何解决指令乱序

> 一种说法是使用 `volatile` 来解决指令乱序的问题，但是仍然有可能存在问题。另一种说法是使用 c++11之后出现的atomic 来实现原子操作。 还有一种更为简单的方法是，在多线程场景下，直接使用饿汉式即可。无需进行判断。

3. 观察者模式

> 观察者模式是行为设计模式之一。当您对对象的状态感兴趣并希望在有任何更改时收到通知时，观察者设计模式非常有用。在观察者模式中，监视另一个对象状态的对象称为Observer，正在被监视的对象称为Subject。
>
> 观察者模式最重要的作用就是 **解耦**， 将观察者与被观察者解耦，使得他们之间的依赖性更小。

- **Subject**：抽象的目标对象
- **ConcreteSubject:** 具体目标对象 (也就是被观察的目标)
- **Observer**：是抽象的“观察”角色，它定义了一个**更新接口**，使得在被观察者状态发生改变时通知自己。这仅仅是一个抽象类（接口而已）
- **ConcreteObserver**：具体的观察者。 作为上面**Observer** 的具体实现。也就是真正去观察的角色

1. 简单入门

> 只要写观察者和被观察者两个角色即可。

- 被观察者里面需要有两个函数：
  注册观察者的函数（知道是被谁观察，以便未来通知他们）和通知观察者的函数 （在这通知观察者）
- 观察者里面也需要有两个函数：
  注册被观察者（知道要观察者，以便未来注销观察）和接收通知的函数（用于接收被观察者的通知）

```cpp
#include <iostream>
#include <thread>
#include <fstream>
#include <sstream>

using namespace std;


class observer;
class subject{

public:
    observer * ob;

    //应该需要知道它被那些对象观察，所以它里面一定包含有这些对象
    void addObserver(observer * ob){
        this->ob = ob;
    }


    //当这个被观察的对象的状态出现了某种变化，则需要通知那些观察它的对象。
    //所以它里面应该还有一个通知的函数，
    void notify();

};

//观察者，去执行观察动作的执行者，
class observer{

public:
    subject * sj;

    //应该要知道它观察着哪些对象，所以这里面应该有一个函数，用于设置观察谁？
    void addsubject(subject * sj){
        this->sj = sj;
    }

    //当被观察者的对象状态发生改变，它会通知这个观察者，所以这个观察者一定要有一个函数
    //用来接收这个通知
```

```
    void update(){
        cout << "观察者收到通知了。" <<endl;
    }

};


void subject::notify() {
    ob->update();
}

int main() {

    //创建观察者对象
    observer ob;

    //创建被观察者对象：目标对象
    subject sj;

    //绑定观察者和被观察者，相互绑定
    ob.addsubject(&sj);
    sj.addObserver(&ob);


    //如果遍历的次数能被5整除，则通知观察者。
    int number ;
    while(1){
        number ++;
        cout << "number = " <<number <<endl;
        if(number % 5 == 0){
            sj.notify();
        }
        this_thread::sleep_for(chrono::seconds(1));
    }
    return 0;
}
```

2. 进阶入门

> 此小节相比上一个小节的改变则是增加了多个观察者，并且在某些观察者对象销毁的时候，从目标对象（被观察者）里面移除该对象。

```cpp
#include <iostream>
#include <thread>
#include <fstream>
#include <sstream>
#include <list>
#include <thread>

using namespace std;


class observer;
class subject{

public:
    //如果担心观察者重复存储，所以此处可以使用set集合
    list<observer *> ob_list;

    //应该需要知道它被那些对象观察，所以它里面一定包含有这些对象
    void addObserver(observer * ob){
       // this->ob = ob;
        ob_list.push_front(ob);
    }

    void removeObserver(observer * ob){
        ob_list.remove(ob);
    }


    //当这个被观察的对象的状态出现了某种变化，则需要通知那些观察它的对象。
    //所以它里面应该还有一个通知的函数，
    void notify();

};

//观察者，去执行观察动作的执行者，
class observer{

public:
    subject * sj;
```

```
    string name;
    observer(string name ) : name(name){}

    //应该要知道它观察着哪些对象，所以这里面应该有一个函数，用于设置观察谁?
    void addsubject(subject * sj){
        this->sj = sj;
    }

    //当被观察者的对象状态发生改变，它会通知这个观察者，所以这个观察者一定要有一个函数
    //用来接收这个通知
    void update(){
        cout << name << ":  收到通知了。" <<endl;
    }

    ~observer(){
        cout << name <<" 了执行析构函数" <<endl;
        //让被观察者忘了这个对象，以后不要通知它了
        sj->removeObserver(this);
        sj = nullptr;

    }

};


void subject::notify() {
    for(observer *ob : ob_list){
        ob->update();
    }
}


//使用子线程的方式来模拟观察者2号的销毁
void observer02(subject *sj){
    observer ob2("观察者2号");
    ob2.addsubject(sj);
    sj->addObserver(&ob2);

    for(int i = 0 ; i < 10 ; i++){
```

```
        cout << "--------->休眠-==" << i <<endl;
        this_thread::sleep_for(chrono::seconds(1));
    }
}

int main() {

    //创建观察者对象
    observer ob("观察者1号");

    //创建被观察者对象：目标对象
    subject sj;

    //绑定观察者和被观察者，相互绑定
    ob.addsubject(&sj);
    sj.addObserver(&ob);

    //创建观察者2号，放置到一个线程里面。
    thread t(observer02 , &sj);


    //如果遍历的次数能被5整除，则通知观察者。
    int number ;
    while(1){
        number ++;
        cout << "number = " <<number <<endl;
        if(number % 5 == 0){
            sj.notify();
        }
        this_thread::sleep_for(chrono::milliseconds(500));
    }
    return 0;
}
```

3. 强化练习

> 一般来说，观察者模式并不是只有两个类： subject 和 observer 。 由于观察者模式的通用性，以及面向对象的流行，通常会把他们两个做成抽象类，然后在配合各自的子类来实现观察者模式。

> 比如：自习课的时候，班里除了班长认真学习之外，有的同学玩游戏，有的同学看NBA，有的同学聊微信，此时大家想了一个法子，让班长看到班主任来的时候，通知他们，别被班主任抓到。

- 抽象目标

> 这是抽象类，是所有具体对象的父类，也就是表示了所有的被观察者都具备注册观察者和通知观察者的功能

```
class Observer; //需要前置声明

//抽象目标
class Subject{
public:

    virtual void addObserver(Observer*) = 0;
    virtual void notify(string action) = 0;
    virtual void removeObserver(Observer*) = 0 ;
};
```

- 抽象观察者

> 抽象类，所有观察者的父类，表示所有的观察者都可以接收到通知和取消观察者的注册

```
//抽象观察者
class Observer{
public:

    Observer()= default;
    virtual ~Observer()= default;
    virtual void addSubject(Subject *sub)=0;
    virtual void update(string action) = 0;
};
```

- 目标对象

> 也就是真正发通知的被观察者。此处指的是班长。

```
//具体的被观察对象 :: 班长
class ClassMonitor : public Subject{

    list<Observer *> ob_list;

public:
    //添加观察者
    void addObserver(Observer* ob){
        ob_list.push_back(ob);
    }

    //移除观察者
    void removeObserver(Observer* ob){
        ob_list.remove(ob);
    }

    //通知观察者
    void notify(string action)  override {
        for(Observer *ob :ob_list){
            ob->update(action);
        }
    }

};
```

- 观察者对象

> 这里指的是接收通知的观察者，即： 看NBA 和玩游戏的这些同学

```
//具体的观察者
class NBAStudent : public Observer{
public:
    Subject *sub;

    void addSubject(Subject *sub){
        this ->sub = sub;
    }

    void update(string action){
        cout << "收到信息：" << action << " , 停止看NBA" <<endl;
    }

    ~NBAStudent(){
        sub->removeObserver(this);
    }
};

//具体的观察者
class GameStudent : public Observer{
public:
    Subject *sub;

    void addSubject(Subject *sub){
        this ->sub = sub;
    }

    void update(string action){
        cout << "收到信息：" << action << " , 停止玩游戏" <<endl;
    }

    ~GameStudent(){
        sub->removeObserver(this);
    }
};
```

- main入口测试

```cpp
#include <iostream>
#include <list>
#include <thread>

using namespace std;

int main() {


    //1. 创建目标对象
    Subject *sj = new ClassMonitor();

    //2. 创建观察者
    Observer *game = new GameStudent();
    Observer *nba = new NBAStudent();

    //3. 两者相互关联
    sj->addObserver(game);
    sj->addObserver(nba);


    game->addSubject(sj);
    nba->addSubject(sj);


    //4. 模拟事件

    int number = 0 ;
    while(1){
        number ++ ;

        cout <<"i ==" << number <<endl;
        if(number % 30 == 0 ){
            sj->notify("校长来啦! ");
        }else if(number %20 == 0){
            sj->notify("年级主任来了");
        }else if(number % 10 == 0){
            sj->notify("班主任来了");
        }
```

```
        this_thread::sleep_for(chrono::milliseconds(500));
    }
    return 0;
}
```