c++单例模式

> 定义：单例模式是创建型设计模式，指的是在系统的生命周期中只能产生一个实例(对象)，确保该类的唯一性。
>
> 一般遇到的写进程池类、日志类、内存池（用来缓存数据的结构，在一处写多出读或者多处写多处读）的话都会用到单例模式

实现方法：全局只有一个实例也就意味着不能用new调用构造函数来创建对象，因此构造函数必须是虚有的。但是由于不能new出对象，所以类的内部必须提供一个函数来获取对象，而且由于不能外部构造对象，因此这个函数不能是通过对象调出来，换句话说这个函数应该是属于对象的，很自然我们就想到了用static。由于静态成员函数属于整个类，在类实例化对象之前就已经分配了空间，而类的非静态成员函数必须在类实例化后才能有内存空间。

单例模式的要点总结：

1. 全局只有一个实例，用static特性实现，构造函数设为私有
2. 通过公有接口获得实例
3. 线程安全
4. 禁止拷贝和赋值

单例模式可以**分为懒汉式和饿汉式**，两者之间的区别在于创建实例的时间不同：懒汉式指系统运行中，实例并不存在，只有当需要使用该实例时，才会去创建并使用实例(这种方式要考虑线程安全)。饿汉式指系统一运行，就初始化创建实例，当需要时，直接调用即可。（本身就线程安全，没有多线程的问题）

懒汉式

- 普通懒汉式会让线程不安全
  因为不加锁的话当线程并发时会产生多个实例，导致线程不安全

```cpp
///  普通懒汉式实现 -- 线程不安全 //
#include <iostream> // std::cout
#include <mutex>    // std::mutex
#include <pthread.h> // pthread_create

class SingleInstance
{
public:
    // 获取单例对象
    static SingleInstance *GetInstance();
    // 释放单例，进程退出时调用
    static void deleteInstance();
        // 打印单例地址
    void Print();
private:
        // 将其构造和析构成为私有的，禁止外部构造和析构
    SingleInstance();
    ~SingleInstance();
    // 将其拷贝构造和赋值构造成为私有函数，禁止外部拷贝和赋值
    SingleInstance(const SingleInstance &signal);
    const SingleInstance &operator=(const SingleInstance &signal);
private:
    // 唯一单例对象指针
    static SingleInstance *m_SingleInstance;
};

//初始化静态成员变量
SingleInstance *SingleInstance::m_SingleInstance = NULL;

SingleInstance* SingleInstance::GetInstance()
{
        if (m_SingleInstance == NULL)
        {
                m_SingleInstance = new (std::nothrow) SingleInstance;  // 没有加锁是线程
        }
    return m_SingleInstance;
}

void SingleInstance::deleteInstance()
```

```
{
    if (m_SingleInstance)
    {
        delete m_SingleInstance;
        m_SingleInstance = NULL;
    }
}

void SingleInstance::Print()
{
        std::cout << "我的实例内存地址是:" << this << std::endl;
}

SingleInstance::SingleInstance()
{
    std::cout << "构造函数" << std::endl;
}

SingleInstance::~SingleInstance()
{
    std::cout << "析构函数" << std::endl;
}
///  普通懒汉式实现 -- 线程不安全  //
```

- 线程安全、内存安全的懒汉式
  上述代码出现的问题：
  i. GetInstance()可能会引发竞态条件，第一个线程在if中判断`m_instance_ptr`是空的，于是开始实例化单例;同时第2个线程也尝试获取单例，这个时候判断`m_instance_ptr`还是空的，于是也开始实例化单例;这样就会实例化出两个对象,这就是线程安全问题的由来
  解决办法：①加锁。②局部变量实例
  ii. 类中只负责new出对象，却没有负责delete对象，因此只有构造函数被调用，析构函数却没有被调用;因此会导致内存泄漏。
  解决办法：使用共享指针

> c++11标准中有一个特性：如果当变量在初始化的时候，并发同时进入声明语句，并发线程将会阻塞等待初始化结束。这样保证了并发线程在获取静态局部变量的时候一定是初始化过的，所以具有线程安全性。因此这种懒汉式是最推

荐的，因为：

i. 通过局部静态变量的特性保证了线程安全 (C++11, GCC > 4.3, VS2015支持该特性);
ii. 不需要使用共享指针和锁
iii. get_instance()函数要返回引用而尽量不要返回指针，

```
///  内部静态变量的懒汉实现  //
class Singleton
{
public:
    ~Singleton(){
        std::cout<<"destructor called!"<<std::endl;
    }
    //或者放到private中
    Singleton(const Singleton&)=delete;
    Singleton& operator=(const Singleton&)=delete;
    static Singleton& get_instance(){
        //关键点!
        static Singleton instance;
        return instance;
    }
    //不推荐，返回指针的方式
    /*static Singleton* get_instance(){
        static Singleton instance;
        return &instance;
        }*/
private:
    Singleton(){
        std::cout<<"constructor called!"<<std::endl;
    }
};
```

使用锁、共享指针实现的懒汉式单例模式

◦ 基于 shared_ptr, 用了C++比较倡导的 RAII思想，用对象管理资源，当 shared_ptr 析构的时候，new 出来的对象也会被 delete掉。以此避免内存泄漏。

    - 加了锁，使用互斥量来达到线程安全。这里使用了两个 if判断语句的技术称为**双检锁**；好处是，只有判断指针为空的时候才加锁，避免每次调用 get_instance的方法都加锁，锁的开销毕竟还是有点大的。

不足之处在于： 使用智能指针会要求用户也得使用智能指针，非必要不应该提出这种约束; 使用锁也有开销; 同时代码量也增多了，实现上我们希望越简单越好。

```cpp
#include <iostream>
#include <memory> // shared_ptr
#include <mutex>  // mutex

// version 2:
// with problems below fixed:
// 1. thread is safe now
// 2. memory doesn't leak

class Singleton {
public:
    typedef std::shared_ptr<Singleton> Ptr;
    ~Singleton() {
        std::cout << "destructor called!" << std::endl;
    }
    Singleton(Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;
    static Ptr get_instance() {

        // "double checked lock"
        if (m_instance_ptr == nullptr) {
            std::lock_guard<std::mutex> lk(m_mutex);
            if (m_instance_ptr == nullptr) {
                m_instance_ptr = std::shared_ptr<Singleton>(new Singleton);
            }
        }
        return m_instance_ptr;
    }


private:
    Singleton() {
        std::cout << "constructor called!" << std::endl;
    }
    static Ptr m_instance_ptr;
    static std::mutex m_mutex;
};

// initialization static variables out of class
```

```
Singleton::Ptr Singleton::m_instance_ptr = nullptr;
std::mutex Singleton::m_mutex;
```

饿汉式

```
// 饿汉实现 /
class Singleton
{

public:
    // 获取单实例
    static Singleton* GetInstance();
    // 释放单实例，进程退出时调用
    static void deleteInstance();
    // 打印实例地址
    void Print();

private:
    // 将其构造和析构成为私有的，禁止外部构造和析构
    Singleton();
    ~Singleton();

    // 将其拷贝构造和赋值构造成为私有函数，禁止外部拷贝和赋值
    Singleton(const Singleton &signal);
    const Singleton &operator=(const Singleton &signal);

private:
    // 唯一单实例对象指针
    static Singleton *g_pSingleton;
};

// 代码一运行就初始化创建实例 ，本身就线程安全
Singleton* Singleton::g_pSingleton = new (std::nothrow) Singleton;

Singleton* Singleton::GetInstance()
{
    return g_pSingleton;
}

void Singleton::deleteInstance()
{
    if (g_pSingleton)
    {
```

```
        delete g_pSingleton;
        g_pSingleton = NULL;
    }
}

void Singleton::Print()
{
    std::cout << "我的实例内存地址是:" << this << std::endl;
}

Singleton::Singleton()
{
    std::cout << "构造函数" << std::endl;
}

Singleton::~Singleton()
{
    std::cout << "析构函数" << std::endl;
}
// 饿汉实现 /
```

面试题

- 懒汉模式和恶汉模式的实现（判空！！！加锁！！！），并且要能说明原因（为什么判空两次？）
- 构造函数的设计（为什么私有？除了私有还可以怎么实现（进阶）？）
- 对外接口的设计（为什么这么设计？）
- 单例对象的设计（为什么是static？如何初始化？如何销毁？（进阶））
- 对于C++编码者，需尤其注意C++11以后的单例模式的实现（为什么这么简化？怎么保证的（进阶））

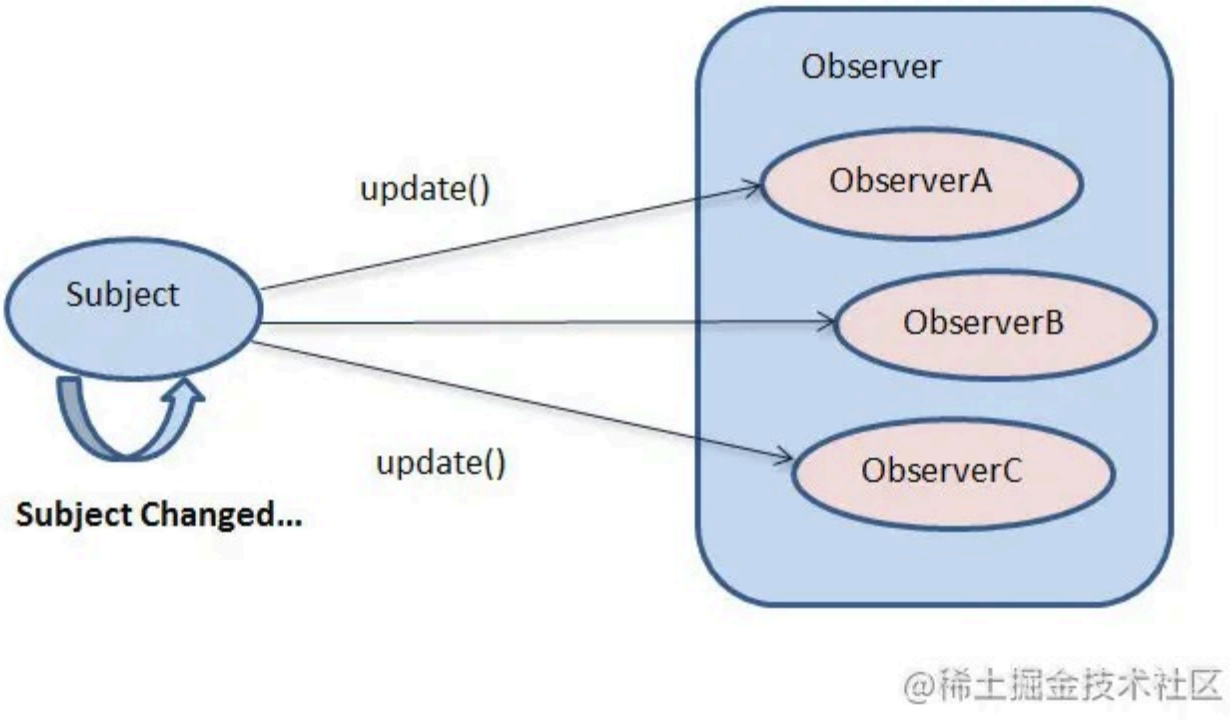
Observe模式

定义

又叫做观察者模式，被观察者叫做subjec，观察者叫做observer

观察者模式(Observer Pattern)：定义对象间一种一对多的依赖关系，使得当每一个对象改变状态，则所有依赖于它的对象都会得到通知并自动更新。

观察者模式所做的工作其实就是在解耦，让耦合的双方都依赖于抽象而不是具体，从而使得各自的变化都不会影响另一边的变化。当一个对象的改变需要改变其他对象的时候，而且它不知道具体有多少对象有待改变的时候，应该考虑使用观察者模式。一旦观察目标的状态发生改变，所有的观察者都将得到通知。具体来说就是被观察者需要用一个容器比如vector存放所有观察者对象，以便状态发生变化时给观察着发通知。观察者内部需要实例化被观察者对象的实例（需要前向声明）



观察者模式中主要角色--2个接口,2个类

1. 抽象主题（Subject）角色(接口)：主题角色将所有对观察者对象的引用保存在一个

集合中，每个主题可以有任意多个观察者。 抽象主题提供了增加和删除观察者对象的接口。

2. 抽象观察者（Observer）角色(接口)：为所有的具体观察者定义一个接口，在观察的主题发生改变时更新自己。
3. 具体主题（ConcreteSubject）角色(1个)：存储相关状态到具体观察者对象，当具体主题的内部状态改变时，给所有登记过的观察者发出通知。具体主题角色通常用一个具体子类实现。
4. 具体观察者（ConcretedObserver）角色(多个)：存储一个具体主题对象，存储相关状态，实现抽象观察者角色所要求的更新接口，以使得其自身状态和主题的状态保持一致。

```
//观察者
interface Observer {
    public void update();
}
//被观察者
abstract class Subject {
    private Vector<Observer> obs = new Vector();

    public void addObserver(Observer obs){
        this.obs.add(obs);
    }
    public void delObserver(Observer obs){
        this.obs.remove(obs);
    }
    protected void notifyObserver(){
        for(Observer o: obs){
            o.update();
        }
    }
    public abstract void doSomething();
}
//具体被观察者
class ConcreteObserver1 implements Observer {
    public void update() {
        System.out.println("观察者1收到信息，并进行处理");
    }
}
class ConcreteObserver2 implements Observer {
    public void update() {
        System.out.println("观察者2收到信息，并进行处理");
    }
}
//客户端
public class Client {
    public static void main(String[] args){
        Subject sub = new ConcreteSubject();
        sub.addObserver(new ConcreteObserver1()); //添加观察者1
        sub.addObserver(new ConcreteObserver2()); //添加观察者2
        sub.doSomething();
```

```
    }
}
//输出
被观察者事件发生改变
观察者1收到信息，并进行处理
观察者2收到信息，并进行处理
可以看到当被观察者发生改变过后，观察者都收到了通知
```

优点

- 观察者模式在被观察者和观察者之间建立一个抽象的耦合。被观察者角色所知道的只是一个具体观察者列表，每一个具体观察者都符合一个抽象观察者的接口。被观察者并不认识任何一个具体观察者，它只知道它们都有一个共同的接口。
- 观察者模式支持广播通讯。被观察者会向所有的登记过的观察者发出通知

缺点

- 当观察者对象很多的时候，通知的发布会产生很多时间，影响程序的效率。一个被观察者，多个观察者时，开发代码和调试会比较复杂，java中消息的通知是默认顺序执行的，若其中一个观察者卡壳，会影响到此观察者后面的观察者执行，影响整体的执行， 多级触发时的效率更让人担忧。
- 虽然观察者模式可以随时使观察者知道所观察的对象发生了变化，但是观察者模式没有相应的机制使观察者知道所观察的对象是怎么发生变化的。
- 如果在被观察者之间有循环依赖的话，被观察者会触发它们之间进行循环调用，导致系统崩溃。在使用观察者模式是要特别注意这一点。

工厂模式

MVC模式

MVC 模式的目的是实现一种动态的程序设计，简化后续对程序的修改和扩展，并且使程序某一部分的重复利用成为可能。除此之外，MVC 模式通过对复杂度的简化，使程序的结构更加直

观。软件系统在分离了自身的基本部分的同时，也赋予了各个基本部分应有的功能。专业人员可以通过自身的专长进行相关的分组：

软件系统分为三个基本部分：模型（Model）、视图（View）和控制器（Controller）。

- 模型（Model）：程序员编写程序应有的功能（实现算法等）、数据库专家进行数据管理和数据库设计（可以实现具体的功能）；
- 控制器（Controller）：负责转发请求，对请求进行处理；
- 视图（View）：界面设计人员进行图形界面设计。

MVC模式的优点

低耦合

通过将视图层和业务层分离，允许更改视图层代码而不必重新编译模型和控制器代码，同样，一个应用的业务流程或者业务规则的改变，只需要改动 MVC 的模型层（及控制器）即可。因为模型与控制器和视图相分离，所以很容易改变应用程序的数据层和业务规则。

模型层是自包含的，并且与控制器和视图层相分离，所以很容易改变应用程序的数据层和业务规则。如果想把数据库从 MySQL 移植到 Oracle，或者改变基于 RDBMS 的数据源到 LDAP，只需改变模型层即可。一旦正确的实现了模型层，不管数据来自数据库或是 LDAP 服务器，视图层都将会正确的显示它们。由于运用 MVC 的应用程序的三个部件是相互独立，改变其中一个部件并不会影响其它两个，所以依据这种设计思想能构造出良好的松耦合的构件。

重用性高

随着技术的不断进步，当前需要使用越来越多的方式来访问应用程序了。MVC 模式允许使用各种不同样式的视图来访问同一个服务端的代码，这得益于多个视图（如 WEB（HTTP）浏览器或者无线浏览器（WAP））能共享一个模型。

比如，用户可以通过电脑或通过手机来订购某样产品，虽然订购的方式不一样，但处理订购产品的方式（流程）是一样的。由于模型返回的数据没有进行格式化，所以同样的构件能被不同的界面（视图）使用。例如，很多数据可能用 HTML 来表示，但是也有可能用 WAP 来表示，而这些表示的变化所需要的是仅仅是改变视图层的实现方式，而控制层和模型层无需做任何改变。

由于已经将数据和业务规则从表示层分开，所以可以最大化的进行代码重用了。另外，模型层也有状态管理和数据持久性处理的功能，所以，基于会话的购物车和电子商务过程，也能被 Flash 网站或者无线联网的应用程序所重用。

生命周期成本低

MVC 模式使开发和维护用户接口的技术含量降低。

部署快

使用 MVC 模式进行软件开发，使得软件开发时间得到相当大的缩减，它使后台程序员集中精力于业务逻辑，界面（前端）程序员集中精力于表现形式上。

可维护性高

分离视图层和业务逻辑层使得 WEB 应用更易于维护和修改。

有利软件工程化管理

由于不同的组件（层）各司其职，每一层不同的应用会具有某些相同的特征，这样就有利于通过工程化、工具化的方式管理程序代码。控制器同时还提供了一个好处，就是可以使用控制器来联接不同的模型和视图，来实现用户的需求，这样控制器可以为构造应用程序提供强有力的手段。给定一些**可重用的**模型和视图，控制器可以根据用户的需求选择模型进行处理，然后选择视图将处理结果显示给用户。