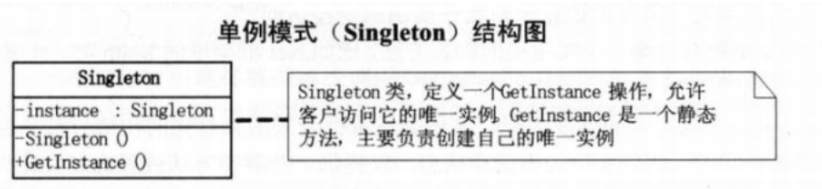
动机:在软件系统中,经常有这样一些特殊的类,必须保证它们在系统中只存在一个实例,才能确保它们的逻辑正确性. 以及良好的效率. 如何绕过常规的构造函数,提供一种机制来保证一个类只有一个实例? 这应该是类设计者的责任,而不是使用者的责任.

定义:保证一个类仅有一个实例,并提供一个实例的全局访问点. --设计模式GOF.



在软件系统中，经常有这样一些特殊的类，必须保证他们在系统中只存在一个实例，才能确保它们的逻辑正确性、以及良好的效率。所以得考虑如何绕过常规的构造器（不允许使用者new出一个对象），提供一种机制来保证一个类只有一个实例。

```
#include<iostream>
using namespace std;
class Singleton{
private:
    Singleton(){
        cout << "Create Singleton." << endl;
    }
    Singleton(Singleton&){}
    static Singleton* m_pSingleton;
public:
    virtual ~Singleton(){}
    static Singleton* GetInstance(){
        if(NULL ==  m_pSingleton){
            m_pSingleton = new Singleton();
        }
        return m_pSingleton;
    }
};
Singleton* Singleton::m_pSingleton = NULL;
/*存在问题:
1. 线程安全的问题,当多线程获取单例时有可能引发竞态条件: 第一个线程在if中判断 m_pSingleton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_pSingleton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:加锁
1. 内存泄漏. 注意到类中只负责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;
*/
/*懒汉式(Lazy-Initialization)的方法是直到使用时才实例化对象,
也就说直到调用GetInstance() 方法的时候才 new 一个单例的对象。
好处是如果被调用就不会占用内存。
*/
int main(){
    Singleton* instance1 = Singleton::GetInstance();
    Singleton* instance2 = Singleton::GetInstance();
    return 0;
}
```

```
#include<iostream>
```

```
using namespace std;
/*
最推荐的懒汉式单例(magic static )--局部静态变量,
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。
C++静态变量的生存期是从声明到程序结束，这也是一种懒汉式。
这是最推荐的一种单例实现方式:
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);
不需要使用共享指针，代码简洁;
注意在使用的时候需要声明单例的引用 Single& 才能获取对象。
*/
class Singleton{
public:
    ~Singleton(){
        cout << "单例对象销毁" << endl;
    }
    static Singleton& GetInstance(){
        /*饿汉式的特点是一开始就加载了，如果说懒汉式是“时间换空间”,
        那么饿汉式就是“空间换时间”，因为一开始就创建了实例,
        所以每次用到的之后直接返回就好了。饿汉模式是线程安全的。*/
        static Singleton instance;
        return instance;
    }
private:
    Singleton(){
        cout << "单例对象创建" << endl;
    }
    Singleton(const Singleton&){}
};
int main(){
    Singleton& instance1 = Singleton::GetInstance();
    Singleton& instance2 = Singleton::GetInstance();
    return 0;
}
```

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

template<typename T>
class Singleton{
public:
    static T& GetInstance(){
        static T m_stiVlue;
        return m_stiVlue;
    }
private:
    Singleton(){}
    ~Singleton(){}
    Singleton(Singleton&){}
};
class A{
public:
    A(){
```

```
        a = "create AAAAAAA";
    }
    void fun(){
        cout << "A.a = " << a << endl;
    }
private:
    string a;
};
class B{
public:
    B(){
        b = "create BBBBBBB";
    }
    void fun(){
        cout << "B.b = " << b << endl;
    }
private:
    string b;
};
int main(){
    Singleton<A>::GetInstance().fun();
    Singleton<B>::GetInstance().fun();
    return 0;
}
```