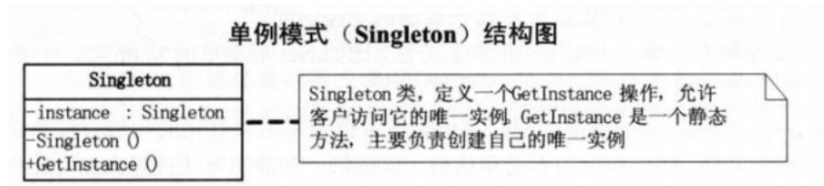


**动机:**在软件系统中,经常有这样一些特殊的类,必须保证它们在系统中只存在一个实例,才能确保它们的逻辑正确性. 以及良好的效率. 如何绕过常规的构造函数,提供一种机制来保证一个类只有一个实例? 这应该是类设计者的责任,而不是使用者的责任.

**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。

```
1 #include<iostream>
2 using namespace std;
3 class Singleton{
4 private:
5     Singleton(){
6         cout << "Create Singleton." << endl;
7     }
8     Singleton(Singleton&){}
9     static Singleton* m_pSingleton;
10 public:
11     virtual ~Singleton(){}
12     static Singleton* GetInstance(){
13         if(NULL == m_pSingleton){
14             m_pSingleton = new Singleton();
15         }
16         return m_pSingleton;
17     }
18 };
19 Singleton* Singleton::m_pSingleton = NULL;
20 /*存在问题:
21 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,
22 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;
23 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
24 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,
25 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;
26 */
27 /*懒汉式(Lazy-Initialization)的方法是直到使用时才实例化对象,
28 也就是说直到调用GetInstance() 方法的时候才 new 一个单例的对象。
29 好处是如果被调用就不会占用内存。
30 */
31 int main(){
32     Singleton* instance1 = Singleton::GetInstance();
33     Singleton* instance2 = Singleton::GetInstance();
34     return 0;
35 }
```

```
1 #include<iostream>
```

```

2 using namespace std;
3 /*
4  最推荐的懒汉式单例(magic static )--局部静态变量，
5  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。
6  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。
7  这是最推荐的一种单例实现方式：
8  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);
9  不需要使用共享指针，代码简洁；
10  注意在使用的时候需要声明单例的引用 Singleton& 才能获取对象。
11 */
12 class Singleton{
13 public:
14     ~Singleton(){
15         cout << "单例对象销毁" << endl;
16     }
17     static Singleton& GetInstance(){
18         /*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”，
19         那么饿汉式就是“空间换时间”，因为一开始就创建了实例，
20         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。*/
21         static Singleton instance;
22         return instance;
23     }
24 private:
25     Singleton(){
26         cout << "单例对象创建" << endl;
27     }
28     Singleton(const Singleton&){}
29 };
30 int main(){
31     Singleton& instance1 = Singleton::GetInstance();
32     Singleton& instance2 = Singleton::GetInstance();
33     return 0;
34 }

```

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 template<typename T>
6 class Singleton{
7 public:
8     static T& GetInstance(){
9         static T m_stiVlue;
10        return m_stiVlue;
11    }
12 private:
13     Singleton(){}
14     ~Singleton(){}
15     Singleton(Singleton&){}
16 };
17 class A{
18 public:
19     A(){

```

```

20     a = "create AAAAAA";
21 }
22 void fun(){
23     cout << "A.a = " << a << endl;
24 }
25 private:
26     string a;
27 };
28 class B{
29 public:
30     B(){
31         b = "createBBBBB";
32     }
33     void fun(){
34         cout << "B.b = " << b << endl;
35     }
36 private:
37     string b;
38 };
39 int main(){
40     Singleton<A>::GetInstance().fun();
41     Singleton<B>::GetInstance().fun();
42     return 0;
43 }

```

```

192:DesignPattnsStudy weishichun$ ls Singleton
Singleton_1.cpp      Singleton_2.cpp      Singleton_Template_3.cpp  Singleton单例模式.pdf
192:DesignPattnsStudy weishichun$ g++ -o Single1.out Singleton_1.cpp
192:DesignPattnsStudy weishichun$ ./Single1.out
Create Singleton.
192:DesignPattnsStudy weishichun$ g++ -o Single2.out Singleton_2.cpp
192:DesignPattnsStudy weishichun$ ./Single2.out
单例对象创建
单例对象销毁
192:DesignPattnsStudy weishichun$ g++ -o Single3.out Singleton_Template_3.cpp
192:DesignPattnsStudy weishichun$ ./Single3.out
A.a = create AAAAAA
B.b = createBBBBB
192:DesignPattnsStudy weishichun$ █

```