C++完美实现 Singleton 模式

Singleton 模式是常用的设计模式之一，但是要实现一个真正实用的设计模式却也不是件容易的事情。

1. 标准的实现

```
class Singleton
{
public:
    static Singleton * Instance()
    {
        if( 0== _instance)
        {
            _instance = new Singleton;
        }
        return _instance;
    }
protected:
    Singleton(void)
    {
    }
    virtual ~Singleton(void)
    {
    }
    static Singleton* _instance;
};
```

这是教科书上使用的方法。看起来没有什么问题，其实包含很多的问题。下面我们一个一个的解决。

2. 自动垃圾回收

上面的程序必须记住在程序结束的时候，释放内存。为了让它自动的释放内存，我们引入 auto_ptr 改变它。

```
#include <memory>
#include <iostream>
using namespace std;
class Singleton
{
public:
    static Singleton * Instance()
    {
        if( 0== _instance.get())
        {
            _instance.reset( new Singleton);
        }
        return _instance.get();
```

```
    }
protected:
    Singleton(void)
    {
        cout <<"Create Singleton"<<endl;
    }
    virtual ~Singleton(void)
    {
        cout << "Destroy Singleton"<<endl;
    }
    friend class auto_ptr<Singleton>;
    static auto_ptr<Singleton> _instance;
};
    //Singleton.cpp
    auto_ptr<Singleton> Singleton::_instance;
```

3. 增加模板

在我的一个工程中，有多个的 Singleton 类，对 Singleton 类，我都要实现上面这一切，这让我觉得烦死了。于是我想到了模板来完成这些重复的工作。现在我们要添加本文中最吸引人单件实现：

```
/*********************************************************************
*****
  (c) 2003-2005 C2217 Studio
  Module:   Singleton.h
  Author:   Yangjun D.
  Created:   9/3/2005   23:17
  Purpose:   Implement singleton pattern
  History:
*********************************************************************
*****/
#pragma once

#include <memory>
using namespace std;
using namespace C2217::Win32;

namespace C2217
{
namespace Pattern
{
template <class T>
class Singleton
{
public:
    static inline T* instance();
```

```
private:
    Singleton(void){}
    ~Singleton(void){}
    Singleton(const Singleton&){}
    Singleton & operator= (const Singleton &){}

    static auto_ptr<T> _instance;
};

template <class T>
auto_ptr<T> Singleton<T>::_instance;

template <class T>
 inline T* Singleton<T>::instance()
{
    if( 0== _instance.get())
    {
        _instance.reset ( new T);
    }

    return _instance.get();
}

//Class that will implement the singleton mode,
//must use the macro in it's delare file
#define DECLARE_SINGLETON_CLASS( type ) \
    friend class auto_ptr< type >;\
    friend class Singleton< type >;
}
}
```

4. 线程安全

上面的程序可以适应单线程的程序。但是如果把它用到多线程的程序就会发生问题。主要的问题在于同时执行 `_instance.reset ( new T);` 就会同时产生两个新的对象，然后马上释放一个，这跟 Singleton 模式的本意不符。所以，你需要更加安全的版本：

```
/*****************************************************************
*****
  (c) 2003-2005 C2217 Studio
  Module:    Singleton.h
  Author:    Yangjun D.
  Created:   9/3/2005   23:17
  Purpose:   Implement singleton pattern
  History:
```

```
*****************************************************************
*****/
#pragma once

#include <memory>
using namespace std;
#include "Interlocked.h"
using namespace C2217::Win32;

namespace C2217
{
namespace Pattern
{
template <class T>
class Singleton
{
public:
	static inline T* instance();

private:
	Singleton(void){}
	~Singleton(void){}
	Singleton(const Singleton&){}
	Singleton & operator= (const Singleton &){}

	static auto_ptr<T> _instance;
	static CResGuard _rs;
};

template <class T>
auto_ptr<T> Singleton<T>::_instance;

template <class T>
CResGuard Singleton<T>::_rs;

template <class T>
 inline T* Singleton<T>::instance()
{
	if( 0 == _instance.get() )
	{
		CResGuard::CGuard gd(_rs);
		if( 0== _instance.get())
		{
			_instance.reset ( new T);
```

```
        }
    }
    return _instance.get();
}

//Class that will implement the singleton mode,
//must use the macro in it's delare file
#define DECLARE_SINGLETON_CLASS( type ) \
    friend class auto_ptr< type >;\
    friend class Singleton< type >;
}
}
```

CresGuard 类主要的功能是线程访问同步,代码如下：

```
/*****************************************************************
***************
Module:  Interlocked.h
Notices: Copyright (c) 2000 Jeffrey Richter
*****************************************************************
**************/


#pragma once
//////////////////////////////////////////////////////////////////////////
/////////////

// Instances of this class will be accessed by multiple threads. So,
// all members of this class (except the constructor and destructor)
// must be thread-safe.
class CResGuard {
public:
  CResGuard()  { m_lGrdCnt = 0; InitializeCriticalSection(&m_cs); }
  ~CResGuard() { DeleteCriticalSection(&m_cs); }

  // IsGuarded is used for debugging
  BOOL IsGuarded() const { return(m_lGrdCnt > 0); }

public:
  class CGuard {
  public:
    CGuard(CResGuard& rg) : m_rg(rg) { m_rg.Guard(); };
    ~CGuard() { m_rg.Unguard(); }

  private:
    CResGuard& m_rg;
```

```
  };

private:
  void Guard()   { EnterCriticalSection(&m_cs); m_lGrdCnt++; }
  void Unguard() { m_lGrdCnt--; LeaveCriticalSection(&m_cs); }

  // Guard/Unguard can only be accessed by the nested CGuard class.
  friend class CResGuard::CGuard;

private:
  CRITICAL_SECTION m_cs;
  long m_lGrdCnt;   // # of EnterCriticalSection calls
};


//////////////////////////////////////////////////////////////////////////////
/////////////
```

5. 实用方法

比如你有一个需要实现单件模式的类，就应该这样实现：

```
#pragma once
#include "singleton.h"
using namespace C2217::Pattern;

class ServiceManger
{
public:
    void Run()
    {
    }
private:
    ServiceManger(void)
    {
    }
    virtual ~ServiceManger(void)
    {
    }
    DECLARE_SINGLETON_CLASS(ServiceManger);
};

typedef Singleton<ServiceManger> SSManger;
```

在使用的时候很简单，跟一般的 Singleton 实现的方法没有什么不同。

```
int _tmain(int argc, _TCHAR* argv[])
{
```

```
        SSManger::instance()->Run();
    }
```

一个简单的 Singleton 模式的实现，可以看到 C++语言背后隐藏的丰富的语意，我希望有人能实现一个更好的 Singleton 让大家学习。我从一开始实现 Singleton 类的过程，其实就是我学习 C++的过程，越是深入越觉得 C++了不起。

mexo

似乎可以这样：

```
template <class T>
class Singleton
{
public:
static inline T& instance()
{
static T _instance;
return _instance;
}

private:
Singleton(void);
~Singleton(void);
Singleton(const Singleton<T>&);
Singleton<T>& operator= (const Singleton<T> &);
};
```

mexo 提出的方案的确不错，但是好象也并不完美（我不是指多线程解决方案），因为他把模板类的构造函数放在私有段里了，如果放在 protected 段里 就好得多，因为你的类可以从模板类继承，这样就不再需要你的那个 typedef Singleton<ServiceManger> SSManger;定义了。示例如下：

```
template <class T>
class Singleton {
public:
static T& instance() {
static T _instance;
return _instance;
}
protected:
Singleton(void) {}
virtual ~Singleton(void) {}
```

```
Singleton(const Singleton<T>&); //不实现
Singleton<T>& operator= (const Singleton<T> &); //不实现
};
```

下面是一个需要做为单例的类，只需从 Singleton 继承即可

```
class Test : public Singleton<Test> {
public:
void foo();
private:
Test();
~Test();
friend class Singleton<Test>;
};
```

这样，别人在使用的时候，只需要写

```
Test::instance().foo();
```

而再也不需要写：

```
Singleton<Test>::instance().foo();
```

或者

```
typedef Singleton<Test> STest;
STest::instance().foo();
```
