**[C++中的单例模式](http://blog.csdn.net/hackbuteer1/article/details/7460019)**

标签： [c++](http://www.csdn.net/tag/c%2b%2b)[编译器](http://www.csdn.net/tag/%e7%bc%96%e8%af%91%e5%99%a8)[设计模式](http://www.csdn.net/tag/%e8%ae%be%e8%ae%a1%e6%a8%a1%e5%bc%8f)[class](http://www.csdn.net/tag/class)[delete](http://www.csdn.net/tag/delete)

2012-04-14 10:08 146865人阅读 [评论](http://blog.csdn.net/hackbuteer1/article/details/7460019#comments)(66) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/hackbuteer1/article/details/7460019#report)

本文章已收录于：

版权声明：本文为博主原创文章，未经博主允许不得转载。

        单例模式也称为单件模式、单子模式，可能是使用最广泛的设计模式。其意图是保证一个类仅有一个实例，并提供一个访问它的全局访问点，该实例被所有程序模块共享。有很多地方需要这样的功能模块，如系统的日志输出，GUI应用必须是单鼠标，MODEM的联接需要一条且只需要一条电话线，操作系统只能有一个窗口管理器，一台PC连一个键盘。  
       单例模式有许多种实现方法，在C++中，甚至可以直接用一个全局变量做到这一点，但这样的代码显的很不优雅。 使用全局对象能够保证方便地访问实例，但是不能保证只声明一个对象——也就是说除了一个全局实例外，仍然能创建相同类的本地实例。  
《设计模式》一书中给出了一种很不错的实现，定义一个单例类，使用类的私有静态指针变量指向类的唯一实例，并用一个公有的静态方法获取该实例。  
       单例模式通过类本身来管理其唯一实例，这种特性提供了解决问题的方法。唯一的实例是类的一个普通对象，但设计这个类时，让它只能创建一个实例并提供对此实例的全局访问。唯一实例类Singleton在静态成员函数中隐藏创建实例的操作。习惯上把这个成员函数叫做Instance()，它的返回值是唯一实例的指针。

定义如下：

**[cpp]** [view plain](http://blog.csdn.net/hackbuteer1/article/details/7460019) [copy](http://blog.csdn.net/hackbuteer1/article/details/7460019) [print?](http://blog.csdn.net/hackbuteer1/article/details/7460019)

1. class CSingleton
2. {
3. private:
4. CSingleton()   //构造函数是私有的
5. {
6. }
7. static CSingleton \*m\_pInstance;
8. public:
9. static CSingleton \* GetInstance()
10. {
11. if(m\_pInstance == NULL)  //判断是否第一次调用
12. m\_pInstance = new CSingleton();
13. return m\_pInstance;
14. }
15. };

[http://static.blog.csdn.net/images/save_snippets.png](javascript:;)

class CSingleton

{

private:

CSingleton() //构造函数是私有的

{

}

static CSingleton \*m\_pInstance;

public:

static CSingleton \* GetInstance()

{

if(m\_pInstance == NULL) //判断是否第一次调用

m\_pInstance = new CSingleton();

return m\_pInstance;

}

};

用户访问唯一实例的方法只有GetInstance()成员函数。如果不通过这个函数，任何创建实例的尝试都将失败，因为类的构造函数是私有的。GetInstance()使用懒惰初始化，也就是说它的返回值是当这个函数首次被访问时被创建的。这是一种防弹设计——所有GetInstance()之后的调用都返回相同实例的指针：

CSingleton\* p1 = CSingleton :: GetInstance();  
CSingleton\* p2 = p1->GetInstance();  
CSingleton & ref = \* CSingleton :: GetInstance();  
对GetInstance稍加修改，这个设计模板便可以适用于可变多实例情况，如一个类允许最多五个实例。  
   
单例类CSingleton有以下特征：  
它有一个指向唯一实例的静态指针m\_pInstance，并且是私有的；  
它有一个公有的函数，可以获取这个唯一的实例，并且在需要的时候创建该实例；  
它的构造函数是私有的，这样就不能从别处创建该类的实例。  
大多数时候，这样的实现都不会出现问题。有经验的读者可能会问，m\_pInstance指向的空间什么时候释放呢？更严重的问题是，该实例的析构函数什么时候执行？  
如果在类的析构行为中有必须的操作，比如关闭文件，释放外部资源，那么上面的代码无法实现这个要求。我们需要一种方法，正常的删除该实例。  
可以在程序结束时调用GetInstance()，并对返回的指针掉用delete操作。这样做可以实现功能，但不仅很丑陋，而且容易出错。因为这样的附加代码很容易被忘记，而且也很难保证在delete之后，没有代码再调用GetInstance函数。  
一个妥善的方法是让这个类自己知道在合适的时候把自己删除，或者说把删除自己的操作挂在操作系统中的某个合适的点上，使其在恰当的时候被自动执行。  
我们知道，程序在结束的时候，系统会自动析构所有的全局变量。事实上，系统也会析构所有的类的静态成员变量，就像这些静态成员也是全局变量一样。利用这个特征，我们可以在单例类中定义一个这样的静态成员变量，而它的唯一工作就是在析构函数中删除单例类的实例。如下面的代码中的CGarbo类（Garbo意为垃圾工人）：

**[cpp]** [view plain](http://blog.csdn.net/hackbuteer1/article/details/7460019) [copy](http://blog.csdn.net/hackbuteer1/article/details/7460019) [print?](http://blog.csdn.net/hackbuteer1/article/details/7460019)

1. class CSingleton
2. {
3. private:
4. CSingleton()
5. {
6. }
7. static CSingleton \*m\_pInstance;
8. class CGarbo   //它的唯一工作就是在析构函数中删除CSingleton的实例
9. {
10. public:
11. ~CGarbo()
12. {
13. if(CSingleton::m\_pInstance)
14. delete CSingleton::m\_pInstance;
15. }
16. };
17. static CGarbo Garbo;  //定义一个静态成员变量，程序结束时，系统会自动调用它的析构函数
18. public:
19. static CSingleton \* GetInstance()
20. {
21. if(m\_pInstance == NULL)  //判断是否第一次调用
22. m\_pInstance = new CSingleton();
23. return m\_pInstance;
24. }
25. };

[http://static.blog.csdn.net/images/save_snippets.png](javascript:;)

class CSingleton

{

private:

CSingleton()

{

}

static CSingleton \*m\_pInstance;

class CGarbo //它的唯一工作就是在析构函数中删除CSingleton的实例

{

public:

~CGarbo()

{

if(CSingleton::m\_pInstance)

delete CSingleton::m\_pInstance;

}

};

static CGarbo Garbo; //定义一个静态成员变量，程序结束时，系统会自动调用它的析构函数

public:

static CSingleton \* GetInstance()

{

if(m\_pInstance == NULL) //判断是否第一次调用

m\_pInstance = new CSingleton();

return m\_pInstance;

}

};

类CGarbo被定义为CSingleton的私有内嵌类，以防该类被在其他地方滥用。  
程序运行结束时，系统会调用CSingleton的静态成员Garbo的析构函数，该析构函数会删除单例的唯一实例。  
使用这种方法释放单例对象有以下特征：  
在单例类内部定义专有的嵌套类；  
在单例类内定义私有的专门用于释放的静态成员；  
利用程序在结束时析构全局变量的特性，选择最终的释放时机；  
使用单例的代码不需要任何操作，不必关心对象的释放。  
  
进一步的讨论

但是添加一个类的静态对象，总是让人不太满意，所以有人用如下方法来重新实现单例和解决它相应的问题，代码如下：

**[cpp]** [view plain](http://blog.csdn.net/hackbuteer1/article/details/7460019) [copy](http://blog.csdn.net/hackbuteer1/article/details/7460019) [print?](http://blog.csdn.net/hackbuteer1/article/details/7460019)

1. class CSingleton
2. {
3. private:
4. CSingleton()   //构造函数是私有的
5. {
6. }
7. public:
8. static CSingleton & GetInstance()
9. {
10. static CSingleton instance;   //局部静态变量
11. return instance;
12. }
13. };

[http://static.blog.csdn.net/images/save_snippets.png](javascript:;)

class CSingleton

{

private:

CSingleton() //构造函数是私有的

{

}

public:

static CSingleton & GetInstance()

{

static CSingleton instance; //局部静态变量

return instance;

}

};

使用局部静态变量，非常强大的方法，完全实现了单例的特性，而且代码量更少，也不用担心单例销毁的问题。  
但使用此种方法也会出现问题，当如下方法使用单例时问题来了，  
Singleton singleton = Singleton :: GetInstance();  
这么做就出现了一个类拷贝的问题，这就违背了单例的特性。产生这个问题原因在于：编译器会为类生成一个默认的构造函数，来支持类的拷贝。

最后没有办法，我们要禁止类拷贝和类赋值，禁止程序员用这种方式来使用单例，当时领导的意思是GetInstance()函数返回一个指针而不是返回一个引用，函数的代码改为如下：

**[cpp]** [view plain](http://blog.csdn.net/hackbuteer1/article/details/7460019) [copy](http://blog.csdn.net/hackbuteer1/article/details/7460019) [print?](http://blog.csdn.net/hackbuteer1/article/details/7460019)

1. class CSingleton
2. {
3. private:
4. CSingleton()   //构造函数是私有的
5. {
6. }
7. public:
8. static CSingleton \* GetInstance()
9. {
10. static CSingleton instance;   //局部静态变量
11. return &instance;
12. }
13. };

[http://static.blog.csdn.net/images/save_snippets.png](javascript:;)

class CSingleton

{

private:

CSingleton() //构造函数是私有的

{

}

public:

static CSingleton \* GetInstance()

{

static CSingleton instance; //局部静态变量

return &instance;

}

};

但我总觉的不好，为什么不让编译器不这么干呢。这时我才想起可以显示的声明类拷贝的构造函数，和重载 = 操作符，新的单例类如下：

**[cpp]** [view plain](http://blog.csdn.net/hackbuteer1/article/details/7460019) [copy](http://blog.csdn.net/hackbuteer1/article/details/7460019) [print?](http://blog.csdn.net/hackbuteer1/article/details/7460019)

1. class CSingleton
2. {
3. private:
4. CSingleton()   //构造函数是私有的
5. {
6. }
7. CSingleton(const CSingleton &);
8. CSingleton & operator = (const CSingleton &);
9. public:
10. static CSingleton & GetInstance()
11. {
12. static CSingleton instance;   //局部静态变量
13. return instance;
14. }
15. };

[http://static.blog.csdn.net/images/save_snippets.png](javascript:;)

class CSingleton

{

private:

CSingleton() //构造函数是私有的

{

}

CSingleton(const CSingleton &);

CSingleton & operator = (const CSingleton &);

public:

static CSingleton & GetInstance()

{

static CSingleton instance; //局部静态变量

return instance;

}

};

关于Singleton(const Singleton);和 Singleton & operate = (const Singleton&);函数，需要声明成私有的，并且只声明不实现。这样，如果用上面的方式来使用单例时，不管是在友元类中还是其他的，编译器都是报错。  
不知道这样的单例类是否还会有问题，但在程序中这样子使用已经基本没有问题了。  
  
考虑到线程安全、异常安全，可以做以下扩展

**[cpp]** [view plain](http://blog.csdn.net/hackbuteer1/article/details/7460019) [copy](http://blog.csdn.net/hackbuteer1/article/details/7460019) [print?](http://blog.csdn.net/hackbuteer1/article/details/7460019)

1. class Lock
2. {
3. private:
4. CCriticalSection m\_cs;
5. public:
6. Lock(CCriticalSection  cs) : m\_cs(cs)
7. {
8. m\_cs.Lock();
9. }
10. ~Lock()
11. {
12. m\_cs.Unlock();
13. }
14. };
16. class Singleton
17. {
18. private:
19. Singleton();
20. Singleton(const Singleton &);
21. Singleton& operator = (const Singleton &);
23. public:
24. static Singleton \*Instantialize();
25. static Singleton \*pInstance;
26. static CCriticalSection cs;
27. };
29. Singleton\* Singleton::pInstance = 0;
31. Singleton\* Singleton::Instantialize()
32. {
33. if(pInstance == NULL)
34. {   //double check
35. Lock lock(cs);           //用lock实现线程安全，用资源管理类，实现异常安全
36. //使用资源管理类，在抛出异常的时候，资源管理类对象会被析构，析构总是发生的无论是因为异常抛出还是语句块结束。
37. if(pInstance == NULL)
38. {
39. pInstance = new Singleton();
40. }
41. }
42. return pInstance;
43. }  [http://static.blog.csdn.net/images/save_snippets.png](javascript:;)class Lock{

private:

CCriticalSection m\_cs;

public:

Lock(CCriticalSection cs) : m\_cs(cs)

{

m\_cs.Lock();

}

~Lock()

{

m\_cs.Unlock();

}

};

class Singleton

{

private:

Singleton();

Singleton(const Singleton &);

Singleton& operator = (const Singleton &);

public:

static Singleton \*Instantialize();

static Singleton \*pInstance;

static CCriticalSection cs;

};

Singleton\* Singleton::pInstance = 0;

Singleton\* Singleton::Instantialize()

{

if(pInstance == NULL)

{ //double check

Lock lock(cs); //用lock实现线程安全，用资源管理类，实现异常安全

//使用资源管理类，在抛出异常的时候，资源管理类对象会被析构，析构总是发生的无论是因为异常抛出还是语句块结束。

if(pInstance == NULL)

{

pInstance = new Singleton();

}

}

return pInstance;

}

之所以在Instantialize函数里面对pInstance 是否为空做了两次判断，因为该方法调用一次就产生了对象，pInstance == NULL 大部分情况下都为false，如果按照原来的方法，每次获取实例都需要加锁，效率太低。而改进的方法只需要在第一次 调用的时候加锁，可大大提高效率。