BOOST的Singleton模版详解

sailzeng 2013-07-08 19:54 浏览(130) 评论(1) 取消收藏

发表者简介： 曾星(sailzeng)， 专家工程师，天美艺游工作室，服务器后台开发 更多信息 | + 关 注

de36IzLKFUS6rlH+QS+S7WO79e7YSbAOGyaDVsILVVo4vnOfxBAR

BOOST的Singleton模版详解

首先要说明，这个准确说并不是BOOST的singleton实现，而是BOOST的POOL库的singleton实现。BOOST库中其实有若干个singleton模版，这个只是其中一个。但网上大部分介绍的介绍的BOOST的Singleton实现都是这个，所以大家也就默认了。而且这个的确算是比较特殊和有趣的一个实现。

网上比较有名的文章是这篇《2B程序员，普通程序员和文艺程序员的Singleton实现》介绍，我虽然对Singleton模版无爱，但自己的项目组中也有人用这个实现，所以还是研究了一下这个实现，特别网上真正解释清楚这个东东的人并不多（包括原文），所以还是研究了一下。

1 为啥2B实现有问题

为了介绍清楚这个实现，我们还要先解释清楚为啥2B实现有问题，首先说明，2B实现和BOOST的实现都可以解决多线程调用Singleton导致多次初始化的问题。

```
//H文件
template <typename T> class Singleton_2B
{
protected:
    typedef  T  object_type;
    //利用的是类的静态全局变量
    static T instance_;
public:
    static T* instance()
    {
        return &instance_;
    }
};

//因为是类的静态变量，必须有一个通用的声明
template<typename T> typename Singleton_2B<T>::object_type  Singleton_2B<T>::instance_;

//测试的例子代码。
class Object_2B_1
{

    //其实使用友元帮助我们可以让Object_2B的构造函数是protected的，从而真正实现单子的意图
    friend class Singleton_2B<Object_2B_1>;
    //注意下面用protected，大家无法构造实例
protected:
    Object_2B_1();
    ~Object_2B_1(){};
public:
    void do_something();
protected:
    int data_2b_1_;
};

class Object_2B_2
```

```
{
    friend class Singleton_2B<Object_2B_2>;
protected:
    Object_2B_2();
    ~Object_2B_2(){};
public:
    void do_something();
protected:
    int data_2b_2_;
};

//CPP文件
Object_2B_1::Object_2B_1():
    data_2b_1_(1)
{
    printf("Object_2B_1::Object_2B_1() this:[%p] data_2b_1_ [%d].\n",this,data_2b_1_);
    Singleton_2B<Object_2B_2>::instance()->do_something();
};

void Object_2B_1::do_something()
{
    data_2b_1_+= 10000;
    printf("Object_2B_1::do_something() this:[%p] data_2b_1_ [%d].\n",this,data_2b_1_);

}



Object_2B_2::Object_2B_2():
        data_2b_2_(2)
{
    printf("Object_2B_2::Object_2B_2() this:[%p] data_2b_2_ [%d].\n",this,data_2b_2_);
    Singleton_2B<Object_2B_1>::instance()->do_something();
};

void Object_2B_2::do_something()
{
    data_2b_2_+= 10000;
    printf("Object_2B_2::do_something() this:[%p] data_2b_2_ [%d].\n",this,data_2b_2_);
}
```

如代码：Singleton_2B是一个singleton的模板封装，根据这个模版，我们实现了2个单子类，Object_2B_1和Object_2B_2，为了模仿问题，我们在其各自的构造函数里面又都调用了其他一个对象的instance函数。因为我们知道，全局和类static 变量的初始化是编译器自己控制的，我们无法干涉，所以如果假设Object_2B_1::instance_静态成员变量被先构造，他的构造函数调用里面调用Object_2B_2::instance().do_something()函数时，Object_2B_2::instance_可能还没有构造出来。从而导致问题。

但会导致什么问题呢？崩溃？不一定是,（因为静态数据区的空间应该是先分配的），而且结果这个和编译器的实现有关系，

GCC的输出结果如下：

```
//GCC编译器的输出如下：
Object_2B_2::Object_2B_2() this:[0046E1F4] data_2b_2_ [2].
//注意下面,do_something函数被调用了，但是没有崩溃，data_2b_1_默认被初始化为了0
Object_2B_1::do_something() this:[0046E1F0] data_2b_1_ [10000].
//注意下面，do_something函数被调用了后，构造函数才起作用，data_2b_1_又被初始化为了
Object_2B_1::Object_2B_1() this:[0046E1F0] data_2b_1_ [1].Object_2B_2::do_somethin[illegible]s:[0046E1F4] dat
a_2b_2_ [10002].
```

VS2010的DEBUG版本的输出和GCC一致，但有意思的是Realse版本输出结果如下：

```
//VC++2010的release版本输出
Object_2B_2::Object_2B_2() this:[0046E1F4] data_2b_2_ [2].
//注意下面的10001，感觉就像构造函数被偷偷调用过一样。有意思。
Object_2B_1::do_something() this:[0046E1F0] data_2b_1_ [10001].
Object_2B_1::Object_2B_1() this:[0046E1F0] data_2b_1_ [1].Object_2B_2::do_something() this:[0046E1F4] dat
a_2b_2_ [10002].
```

2 BOOST的实现如何规避问题

接着我们就来看看BOOST的模版是使用什么技巧，即保证多线程下不重复初始化，又让相互之间的调用更加安全。

```
template <typename T>
class Singleton_WY
{
private:
    struct object_creator
    {
        object_creator()
        {
            Singleton_WY<T>::instance();
        }
        inline void do_nothing() const {}
    };
    //利用类的静态对象object_creator的构造初始化,在进入main之前已经调用了instance
    //从而避免了多次初始化的问题
    static object_creator create_object_;
public:
    static T *instance()
    {
        static T obj;
        //do_nothing 是必要的，do_nothing的作用有点意思，
        //如果不加create_object_.do_nothing();这句话，在main函数前面
        //create_object_的构造函数都不会被调用，instance当然也不会被调用，
        //我的估计是模版的延迟实现的特效导致，如果没有这句话，编译器也不会实现
       //Singleton_WY<T>::object_creator,所以就会导致这个问题
        create_object_.do_nothing();
        return &obj;
    }
};
//因为create_object_是类的静态变量，必须有一个通用的声明
template <typename T>  typename Singleton_WY<T>::object_creator Singleton_WY<T>::create_object_;

//测试的例子
class Object_WY_1
{
    //其实使用友元帮助我们可以让Object_2B的构造函数是protected的，从而真正实现单子的意图
    friend class Singleton_WY<Object_WY_1>;
    //注意下面用protected，大家无法构造实例
protected:
    Object_WY_1();
    ~Object_WY_1(){};
public:
    void do_something();
protected:
    int data_wy_1_;
};
```

```
class Object_WY_2
{
    friend class Singleton_WY<Object_WY_2>;
protected:
    Object_WY_2();
    ~Object_WY_2(){};
public:
    void do_something();
protected:
    int data_wy_2_;
};

//CPP代码
Object_WY_1::Object_WY_1():
    data_wy_1_(1)
{
    printf("Object_WY_1::Object_WY_1() this:[%p] data_2b_1_ [%d].\n",this,data_wy_1_);
    Singleton_WY<Object_WY_2>::instance()->do_something();
};

void Object_WY_1::do_something()
{
    data_wy_1_+= 10000;
    printf("Object_2B_1::do_something() this:[%p] data_2b_1_ [%d].\n",this,data_wy_1_);

}



Object_WY_2::Object_WY_2():
    data_wy_2_(2)
{
    printf("Object_WY_2::Object_WY_2() this:[%p] data_2b_2_ [%d].\n",this,data_wy_2_);
    Singleton_WY<Object_WY_1>::instance()->do_something();
};

void Object_WY_2::do_something()
{
    data_wy_2_+= 10000;
    printf("Object_WY_2::do_something() this:[%p] data_2b_2_ [%d].\n",this,data_wy_2_);
}
```

调用输出的结果如下，大家可以发现调用顺序正确了Object_WY_2的构造函数内部调用：Singleton_WY<Object_WY_1>::instance()函数的时候，Object_WY_1的单子实例就被创建出来了。

```
Object_WY_2::Object_WY_2() this:[00ECA138] data_2b_2_ [2].
Object_WY_1::Object_WY_1() this:[00ECA140] data_2b_1_ [1].
Object_WY_2::do_something() this:[00ECA138] data_2b_2_ [10002].
Object_2B_1::do_something() this:[00ECA140] data_2b_1_ [10001].
```

首先BOOST的这个实现的Singleton的数据分成两个部分，一个是内部类的object_creator的静态成员creator_object_,一个是instance函数内部的静态变量static T obj;如果外部的有人调用了instance()函数，静态变量obj就会被构造出来，而静态成员creator_object_会在main函数前面构造，他的构造函数内部也调用instance()，这样就会保证静态变量一定会在main函数前面初始化出来。

到此为止，这部分还都能正常理解，但instance()函数中的这句就是有点诡异技巧的了。

```
create_object_.do_nothing();
```

其实这句话如果单独分析，并没有明确的作用，因为如果类的静态成员creator_object_的构造就应该让单子对象被初始化。但一旦你注释掉这句话，你会发现create_object_的构造函数都不会被调用。在main函数之前，什么事情都没有发生（VC++2010和GCC都一样），BOOST的代码注释只说是确保create_object_的构造被调用，但也没有明确原因。

我估计这还是和模版的编译有潜在的关系，模版都是Lazy Evaluation。所以如果编译器没有编译过create_object_.do_no

thing();编译器就会漏掉create_object_的对象一切实现，也就完全不会编译Singleton_WY<T>::object_creator和Singleton_WY<T>:: create_object_代码，所以就会导致这个问题。使用dumpbin 分析去掉前后的obj文件，大约可以证明这点。所以create_object_.do_nothing();这行代码必须要有。

3 个人感觉

也许是因为我本身对Singleton的模版就不感冒，我对文艺青年的这个Singleton也没有太大胃口，
一方面是技巧性过强，我不才，do_nothing()那句话的问题我研究了半天。
二是由于他将所有的instance初始化放在了main函数前面，好处是避免了多线程多次初始化的麻烦，但也限制了初始化的多样性。一些太强的逻辑关系的情况下这招并不好。
三是这种依靠类static变量的方式，无法按需启动，回收。
四是性能，每次do_nothine也是无谓的消耗呀。
为了一个很简单的风险（多次初始化），引入一个技巧性很强的又有各种限制的东东。是否有点画蛇添足。YY。

告别2012，迎接2013.

de36IzLKFUS6rlH+QS+S7WO79e7YSbAOGyaDVsILVVo4vn0fxBAR

[375/409/91 ms]