[**(搬运工)Boost 不同Mutex的大体说明**](http://www.cppblog.com/ming81/archive/2012/07/18/184028.html)

Posted on 2012-07-18 10:58 [点点滴滴](http://www.cppblog.com/ming81/) 阅读(4878) [评论(0)](http://www.cppblog.com/ming81/archive/2012/07/18/184028.html#Post)  [编辑](http://www.cppblog.com/ming81/admin/EditPosts.aspx?postid=184028) [收藏](http://www.cppblog.com/ming81/AddToFavorite.aspx?id=184028) [引用](http://www.cppblog.com/ming81/services/trackbacks/184028.aspx) 所属分类: [02 编程语言](http://www.cppblog.com/ming81/category/15281.html) http://www.cppblog.com/ming81/aggbug/184028.html?webview=1

写过多线程程序的人都知道，不能让多个线程同时访问共享的资源是至关重要的。假如一个线程试图改变共享数据的值，而另外一个线程试图去读取该共享数据的值，结果将是未定义的。为了阻止这样的事情发生，需要用到一些非凡的原始数据类型和操作。其中最重的一个就是总所周知的mutex(“mutual exclusion”的缩写。译注：相互排斥的意思，经常被翻译为“互斥体”)。mutex在同一时间只能答应一个线程访问共享资源。当一个线程需要访问共享资源时，它必须先“锁住”mutex，假如任何其他线程已经锁住了mutex，那么本操作将会一直被阻塞，直到锁住了mutex的线程解锁，这就保证了共享资源，在同一时间，只有一个线程可以访问。

mutex的概念有几个变种。Boost.Threads支持两大类型的mutex：简单mutex和递归mutex。一个简单的mutex只能被锁住一次，假如同一线程试图两次锁定mutex，将会产生死锁。对于递归mutex，一个线程可以多次锁定一个mutex，但必须以同样的次数对mutex进行解锁，否则其他线程将无法锁定该mutex。

在上述两大类mutex的基础上，一个线程如何锁定一个mutex也有些不同变化。一个线程有3种可能方法来锁定mutex：

1. 等待并试图对mutex加锁，直到没有其他线程锁定mutex；

2. 试图对mutex加锁，并立即返回，假如其他线程锁定了mutex；

3. 等待并试图对mutex加锁，直到没有其他线程锁定mutex或者直到规定的时间已过。

看起来最好的mutex类型是递归的mutex了，因为上述3种加锁的方式它都支持。不过，不同的加锁方式有不同的消耗，因此对于特定的应用，Boost.Threads答应你挑选最有效率的mutex。为此，Boost.Threads提供了6中类型的mutex，效率由高到低排列：boost::mutex，boost::try\_mutex，boost::timed\_mutex，boost::recursive\_mutex，boost::recursive\_try\_mutex和boost::recursive\_timed\_mutex。

假如一个线程锁定一个mutex后，而没有解锁，就会发生死锁，这也是最为常见的错误了，为此，Boost.Threads专门进行了设计，可不直接对mutex加锁或者解锁操作，以使这种错误不可能发生(或至少很难发生)。取而代之地，mutex类定义了内嵌的typedef来实现RAII(Resource Acquisition In Initialization，译注：在初始化时资源获得)[4]用以对一个mutex进行加锁或者解锁，这就是所谓的Scoped Lock模式。要构建一个这种类型的锁，需要传送一个mutex引用，构造函数将锁定mutex，析构函数将解锁mutex。C++语言规范确保了析构函数总是会被调用，所以即使有异常抛出，mutex也会被正确地解锁。

这种模式确保了mutex的正确使用。不过必须清楚，尽管Scoped Lock模式保证了mutex被正确解锁，但它不能保证在有异常抛出的时候，所有共享资源任然处于有效的状态，所以，就像进行单线程编程一样，必须确保异常不会让程序处于不一致的状态。同时，锁对象不能传送给另外一个线程，因为他们所维护的状态不会受到此种用法的保护。

列表2举例说明了boost::mutex类的一个简单的用法。其中两个线程被创建，每个循环10次，将id和当前循环计数输出到std::cout，main线程等待着两个线程结束。std::cout对象是一个共享资源，所以每个线程均使用全局mutex，以确保在同一时刻，只有一个线程输出到它。

#include <boost/thread/thread.hpp>

#include <boost/thread/mutex.hpp>

#include <iostream>

boost::mutex io\_mutex;

struct count

{

count(int id) : id(id) { }

void operator()()

{

for (int i = 0; i < 10; ++i)

{

boost::mutex::scoped\_lock lock(io\_mutex);

std::cout << id << ": " << i << std::endl;

}

}

int id;

};

int main(int argc, char\* argv[])

{

boost::thread thrd1(count(1));

boost::thread thrd2(count(2));

thrd1.join();

thrd2.join();

return 0;

}

列表2

也许你已经注重到在列表2的代码中，需要手工写一个函数对象，才能向线程传送数据。尽管代码很简单，但每次都要写这样的代码也会让人有单调沉闷之感。有另外一种更轻易的解决办法，Functional库可以让你通过将需要传入的数据绑定到另外一个函数对象的方式，来创建一个新的函数对象。列表3展现了Boost.Bind库如何不写函数对象，而简化列表2中的代码。

// This program is identical to listing2.cpp except that it uses

// Boost.Bind to simplify the creation of a thread that takes data.

#include <boost/thread/thread.hpp>

#include <boost/thread/mutex.hpp>

#include <boost/bind.hpp>

#include <iostream>

boost::mutex io\_mutex;

void count(int id)

{

for (int i = 0; i < 10; ++i)

{

boost::mutex::scoped\_lock lock(io\_mutex);

std::cout << id << ": " << i << std::endl;

}

}

int main(int argc, char\* argv[])

{

boost::thread thrd1(boost::bind(&count, 1)); // 有无&符号均可

boost::thread thrd2(boost::bind(&count, 2)); // 有无&符号均可

thrd1.join();

thrd2.join();

return 0;

}

列表3