区域锁（Scoped Locking）习惯用法

目的

区域锁习惯用法确保当控制进入一个区域时，一个锁将自动被获取，当控制离开这个区域时，这个锁将被自动释放。

别名

卫兵（guard），同步块

例子

具有代表性的商业WEB服务器都维护一个"点击计数"的组件，用于记录在一个时间范围内每一个URL被用户访问的次数。为了减少延迟，点击计数组件能够被每个WEB服务器缓存在常驻内存表中。此外，为了增加吞吐量，WEB服务器进程通常是多线程的。因此对点击计数组件的公共方法的访问必须被串行化，用于防止多线程并发更新点击组件造成的内部状态表的损坏。 一种串行化访问点击计数组件的方法是在每一个公共方法中清晰的进行加锁和解锁的操作。

class Hit\_Counter

{

public:

*// Increment the hit count for a URL pathname.*

int increment (const char \*pathname)

{

*// Acquire lock to enter critical section.*

lock\_.acquire ();

Table\_Entry \*entry = find\_or\_create (pathname);

if (entry == 0)

{

*// Something's gone wrong, so bail out.*

lock\_.release ();

return -1;

}

else

{

*// Increment the hit count for this pathname.*

entry->increment\_hit\_count ();

*// Release lock to leave critical section.*

lock\_.release ();

}

*// ...*

}

*// Other public methods omitted.*

private:

*// Find the table entry that maintains the hit count*

*// associated with , creating the entry if*

*// it doesn't exist.*

Table\_Entry \*find\_or\_create (const char \*pathname);

*// Serialize access to the critical section.*

Thread\_Mutex lock\_;

}

虽然上面的代码对于并发的Hit\_Counter组件来说可以正常运作，但是这个实现是冗余和难于开发和维护的。比如, 在所有从increment方法返回的路径上，程序维护人员可能会忘记释放lock\_。而且，因为上面的代码不是异常安全的，所以如果后续的版本抛出一个异常或是调用能够抛出异常的辅助方法都将导致lock\_无法被释放。 如果lock\_没有被释放，当后续线程试图取获取lock\_而被无限的阻塞，将导致整个WEB服务进程被挂起。

语境

一个并发的应用实现被多个线程并发操作的共享资源。

问题

多线程的应用或组件如果实现清晰的加锁/解锁操作将不能达到下面的效果：

Robust locking：当控制进入和离开临界区时，总是应该分别的进行加锁和解锁的操作。如果加锁和解锁的操作被清晰的实现，就很难保证在所有通过临界区的路径上能够正确的释放锁。

一个程序维护人员可能修改increment方法，增加新的错误检测条件，如下所示：

*// ...*

else if (entry->increment\_hit\_count () == -1)

return -1;

同样的，find\_or\_create方法可能被修改成如果发生错误将抛出一个异常。不幸的是，这两处改动都导致increment方法没有释放lock\_就返回。而且，如果这些错误发生的几率很小，这段代码中的问题可能不会在测试过程中被发现。

解决之道

定义一个卫兵（guard）类，当控制进入一个区域时，它的构造函数将自动获取锁，当控制离开这个区域时，它的析构函数将自动释放锁。在一个方法中实例化一个guard类实例用于获取/释放锁，同时定义了临界区的阻塞范围。

实现

范围锁习惯用法的实现是非常直接的 ― 在一个方法或是阻塞区域内定义一个自动获取和释放特定类型锁的guard类。Guard类的构造函数将保存一个指向锁的指针，在进入临界区之间使用这个指针获取锁。当离开临界区时，这个类的析构函数使用这个锁指针来释放锁。使用指向锁的指针是因为C++的包装门面模式定义的锁不允许拷贝和赋值操作。

下面展示了针对Thread Mutex:这种特定类型的锁实现的guard类设计：

class Thread\_Mutex\_Guard

{

public:

*// Store a pointer to the lock and acquire the lock.*

Thread\_Mutex\_Guard(Thread\_Mutex &lock)

: lock\_ (lock)

{

result\_ = lock\_.acquire ();

}

*// Release the lock when the guard goes*

*// out of scope.*

~Thread\_Mutex\_Guard()

{

*// Only release the lock if it was acquired.*

if (result\_ != -1)

lock\_.release ();

}

private:

*// Reference to the lock we're managing.*

Thread\_Mutex &lock\_;

*// Records if the lock was acquired successfully.*

int result\_;

};

变化

范围锁习惯用法有如下的变化： 策略化的范围锁。针对每种不同类型的锁定义相应的guard类是冗余的，容易出错的，可能会增加应用和组件的内存占用。因此，范围锁模式的一个普通的变化是使用由参数化类型方法或多态方法实现的策略化的锁模式。

1、 参数化类型。在这个方法中定义一个guard类模板，这个模板在被特定的锁类型参数化后实现自动的加锁和解锁操作。

下面展示了Guard模板类，模板参数为锁类型，比如CRITICAL\_SECTION、Mutex等

template<typename LOCK>

class Guard

{

public:

*// Store a pointer to the lock and acquire the lock.*

Guard (LOCK &lock): lock\_ (lock) {

result\_ = lock\_.acquire ();

}

*// Release the lock when the guard goes out of scope.*

Guard (void)

{

if (result\_ != -1)

lock\_.release ();

}

private:

*// Reference to the lock we're managing.*

LOCK &lock\_;

*// Records if the lock was acquired successfully.*

int result\_;

};

使用这个实现，模板可以被任意类型的锁实例化。

2、 多态。在这个方法中，传递一个多态的锁对象到guard的构造函数，在guard内部定义一个针对这个多态锁对象私有实例用于实现范围锁。在这个实现中，锁类使用桥模式对一个多态的锁层次提供一个统一的对象接口。 下面展示了一个使用多态锁的guard类实现：

class Guard

{

public:

*// Store a pointer to the lock and acquire the lock.*

Guard (Lock &lock): lock\_ (lock)

{

result\_ = lock\_.acquire ();

}

*// Release the lock when the guard goes out of scope.*

Guard (void)

{

if (result\_ != -1)

lock\_.release ();

}

private:

*// Reference to the lock we're managing.*

Lock &lock\_;

*// Records if the lock was acquired successfully.*

int result\_;

};

一般情况下，当锁策略在编译时刻就可以确定，应该使用参数化类型方法。当锁策略直到运行时才可以确定，应该使用多态方法。通常，在模板运行时性能和使用多态带来的潜在的运行时扩展性之间综合考虑。

已知应用

范围锁习惯用法被广泛的使用在ACE面向对象网络编程工具中。 Rogue Wave（一个提供xml schema 到C++数据绑定工具的公司） Threads.h++库中定义了一系列模仿ACE范围锁设计的guard类。

Java语言定义了一种称之为同步块的语言特性，该特性使用Java语言实现了区域锁习惯用法。

结论

使用范围锁带来的两个益处：

1、增加程序的健壮性。通过使用这个习惯用法，当控制进入/离开被C++方法和阻塞区域定义的临界区时，将自动实现加锁/解锁的操作。因此，这个方法通过减少并发编程中的一类常见的错误来增加并发应用程序的健壮性。

2、降低维护的工作量。如果参数化方法或多态方法被用于实现guard/lock类，它将直接增加聚合，减少错误。因为这里只有一个实现，而不时针对每个guard类型都有一个分别的实现。这种集中化的概念避免了版本的爆炸。

在并发应用和组件中使用区域锁习惯用法将带来如下的缺陷：

1、当进行递归使用时，存在潜在的死锁。如果一个使用范围锁的方法递归的调用自己，那么将导致自死锁，除非使用的是可递归的mutex。线程安全的接口模式通过确保只有接口方法可以使用范围锁来避免了这个问题，但是目前的方法实现没有使用线程安全接口模式。

2、受到编程语言规定的语义的限制。因为范围锁习惯用法是基于C++的语言特性，没有必要绑定操作系统规定的系统调用。因此，当线程或进程在一个保护的临界区中被中止，范围锁将不能被释放 例如，下面针对increment函数的修改将阻止区域锁发挥作用：

Thread\_Mutex\_Guard guard (lock\_);

Table\_Entry \*entry = find\_or\_create (pathname);

if (entry == 0)

*// Something's gone wrong, so exit the*

*// thread.*

thread\_exit ();

*// Destructor will not be called so the*

*// will not be released!*