多线程学习笔记

--20171029 start

注：学习《Linux多线程服务端编程 使用muduo C++网络库》笔记。

# 线程安全的对象生命期管理

编写线程安全的类不是难事，用同步原语保护内部状态即可。但是对象的生和死不能由对象自身拥有的互斥器mutex来保护。如何避免对象析构时可能存在的race condition（竞态条件）是C++多线程编程面临的基本问题，可以借助Boost库的shared\_ptr和weak\_ptr完美解决。这也是实现线程安全的Observer模式的必备技术。

## 当析构函数遇到多线程

C++中当一个对象能被多个线程同时看到时，那么对象的销毁时机就会变得模糊不清，可能出现多种竞态条件（race condition）：

* 在即将析构一个对象时，从何而知此刻是否有别的线程正在执行该对象的成员函数？
* 如何保证在执行成员函数期间，对象不会在另一个线程被析构？
* 在调用某个对象的成员函数之前，如何得知这个对象还活着？它的析构函数会不会碰巧执行到一半？

### 线程安全的定义

一个线程安全的class应当满足以下三个条件：

* 多个线程同时访问时，其表现出正确的行为。
* 无论操作系统如何调度这些线程，无论这些线程的执行顺序如何交织（interleaving）。
* 调用端代码无需额外的同步或其他协调动作。

C++标准库里的大多数class都不是线程安全的，包括string、vector、map等，因为这些class通常需要在外部加锁才能供多个线程同时访问。

## 对象的创建很简单

对象构造要做到线程安全，唯一的要求是在构造期间不要泄露this指针，即：

* 不要在构造函数中注册任何回调。
* 也不要在构造函数中把this传给跨线程的对象。
* 即便在构造函数的最后一行也不行。

这是因为在构造函数执行期间对象还没有完成初始化，如果this被泄露给了其他对象（其自身创建的子对象除外），那么别的线程有可能访问这个半成品对象，这会造成难以预料的后果。

可以通过二段式构造--即构造函数+initialize()来避免在构造函数中传递this。这虽然不符合C++教条，但是多线程下别无选择。另外，这样构造函数不必主动抛异常，调用方靠initialize()的返回值来判断对象是否构造成功，这能简化错误处理。

## 销毁太难

作为class数据成员的MutexLock只能用于同步本class的其他数据成员的读和写，它不能保护安全地析构。因为MutexLock成员的生命期最多与对象一样长，而析构动作可说是发生在对象身故之后。另外，对于基类对象，那么调用到基类析构函数的时候，派生类对象的那部分已经析构了，那么基类对象拥有的MutexLock不能保护整个析构过程。再说，析构过程本来也不需要保护，因为只有别的线程都访问不到这个对象的时候，析构才是安全的，否则会有竞态条件的发生。

例如，下面的代码试图用互斥锁来保护析构函数：

|  |  |
| --- | --- |
| Foo::~Foo()  {  MutexLockGuard lock(mutex\_);  // free internal state (1)  } | void Foo::update()  {  MutexLockGuard lock(mutex\_); (2)  // make use of interanl state  } |

此时，有A、B两个线程都能看到Foo对象x，线程A即将销毁x，线程B正准备调用x->update()。

1. 线程A执行到了析构函数的(1)处，已经持有了互斥锁，即将继续往下执行；
2. 线程B通过if(x)的检测，阻塞在(2)处。

接下来发生什么无从知晓。因为析构函数会把mutex\_销毁，那么(2)有可能永远阻塞下去，有可能进入“临界区”，然后core dump，或者发生其他更糟糕的事情。

一个函数如果要锁住相同类型的多个对象，为了保证始终按相同的顺序加锁，我们可以比较mutex对象的地址，始终先加锁地址较小的mutex。

例如swap()函数：

void swap(Counter& a, Counter& b)

{

MutexLockGuard aLock(a.mutex\_);

MutexLockGuard bLock(b.mutex\_);

int64\_t value = a.value\_;

a.value\_ = b.value\_;

b.value\_ = value;

}

如果线程A执行swap(a,b)；而同时线程B执行swap(b,a);就有可能死锁。

## 神器shared\_ptr/weak\_ptr

shared\_ptr的几个关键点：

shared\_ptr控制对象的生命期。shared\_ptr是强引用，只要有一个指向x对象的shared\_ptr存在，该x对象就不会析构。当指向x的最后一个shared\_ptr析构或reset()的时候，x保证会被销毁。

weak\_ptr不控制对象的生命期，但是它知道对象是否还活着。如果对象还活着，那么它可以提升为有效的shared\_ptr；如果对象已经死了，那么提升会失败，返回一个空的shared\_ptr。“提升/lock()”行为是线程安全的。

shared\_ptr/weak\_ptr的“计数”在主流平台上是原子操作，没有用锁，性能不俗。

shared\_ptr/weak\_ptr的线程安全级别与string和STL容器一样。

## 系统地避免各种指针错误

C++可能出现的内存问题及解决方案：

* 缓冲区溢出：用vector<char>/string或自己编写的Buffer class来管理缓冲区，自动记住用缓冲区的长度，并通过成员函数而不是裸指针来修改缓冲区。
* 空悬指针/野指针：用shared\_ptr/weak\_ptr。
* 重复释放：用scoped\_ptr，只在对象析构的时候释放一次。
* 内存泄露：用scoped\_ptr，对象析构的时候自动释放内存。
* 不配对的new[]/delete：把new[]统统替换为vector/scoped\_array。
* 内存碎片：？

shared\_ptr、scoped\_ptr和weak\_ptr都是值语意，要么是栈上对象，或是其他对象的直接数据成员，或是标准库容器里的元素。几乎不会有下面的用法：

shared\_ptr<Foo>\* pFoo = new shared\_ptr<Foo>(new Foo);

还要注意，如果这几种智能指针是对象x的数据成员，而它的模板参数T是个incomplete类型，那么x的析构函数不能是默认或内联的，必须在.cpp文件里面显示定义，否则会有编译错或运行错。

## shared\_ptr的线程安全

shared\_ptr本身不是百分百线程安全的。它的引用计数本身是安全而且无锁的，但对象的读写则不是，因为shared\_ptr有两个数据成员，读写操作不能原子化。shared\_ptr的线程安全级别与内建类型、标准库容器、string一样，即：

* 一个shared\_ptr对象实体可被多个线程同时读取；
* 两个shared\_ptr对象实体可以被两个线程同时写入，“析构”算写操作；
* 如果要从多个线程读写同一个shared\_ptr对象，那么需要加锁。

以上是shared\_ptr对象本身的线程安全级别，不是它管理的对象的线程安全级别。

## shared\_ptr技术与陷阱

* 意外延长对象的生命期。如果不小心遗留了一个shared\_ptr的拷贝，那么对象就永世长存了。
* 函数参数。因为要修改引用计数，shared\_ptr的拷贝开销比拷贝原始指针要高，但是需要拷贝的时候并不多。多数情况下它们可以以const reference方式传递，一个线程只需要在最外层函数有一个实体对象，之后都可以用const reference来使用这个shared\_ptr。
* 析构动作在创建时被捕获。这意味着：虚析构不再是必须的；shared\_ptr<void>可以持有任何对象，而且能安全地释放；shared\_ptr对象可以安全地跨越模块边界，比如从DLL里返回，而不会造成从模块A分配的内存在模块B里被释放这种错误；二进制兼容性，即便Foo对象的大小变了，那么旧的客户代码仍然可以使用新的动态库，而无需重新编译。前提是Foo的头文件中不出现访问对象的成员的inline函数，并且Foo对象的由动态库中的Factory构造，返回其shared\_ptr；析构动作可以定制。
* 析构所在的线程。对象的析构是同步的，当最后一个指向x的shared\_ptr离开其作用域的时候，x会同时在同一个线程析构。这个线程不一定是对象诞生的线程。这个特性是把双刃剑：如果对象的析构比较耗时，那么可能会拖慢关键线程的速度（如果最后一个shared\_ptr引发的析构发生在关键线程）；同时，我们可以用一个单独的线程来专门做析构，通过一个BlockingQueue<shared\_ptr<void> >把对象的析构都转移到那个专用线程，从而解放关键线程。
* 现成的RAII handle。每一个明确的资源配置动作（如new）都应该在单一语句中执行，并在该语句中立刻将配置获得的资源交给handle对象（如shared\_ptr），程序一般不出现delete。shared\_ptr是管理共享资源的利器，需要注意避免循环引用，通常的做法是owner持有指向child的shared\_ptr，child持有指向owner的weak\_ptr。

## 对象池

# 线程同步概要

线程同步的四项原则，按重要性排列：

* 首要原则是尽量最低限度的共享对象，减少需要同步的场合。一个对象能不暴露给别的线程就不要暴露；如果要暴露，优先考虑immutable对象；实在不行才暴露可以修改的对象，并用同步措施来充分保护它。
* 其次是使用高级的并发编程构件，如TaskQueue、Producer-Consumer Queue、CountDownLatch等等。
* 最后不得已必须使用底层同步原语时，只用非递归的互斥器和条件变量，慎用读写锁，不要用信号量。
* 除了使用atomic整数之外，不自己编写lock-free代码，也不要用“内核级”同步原语。不凭空猜测“哪种做法性能会更好”，比如spin lock vs mutex。

## 互斥器（mutex）

互斥器保护了临界区，任何一个时刻最多只有一个线程在此mutex划出的临界区内活动。单独使用mutex时，我们主要为了保护共享数据。

使用互斥器的基本原则：

* 用RAII手法封装互斥器的创建、销毁、加锁、解锁这四个操作。
* 只用非递归的mutex（即不可重入的mutex）。
* 不手工调用lock()和unlock()函数，一切交给栈上的Guard对象的构造和析构函数负责。Guard对象的生命期正好等于临界区。这样我们保证始终在同一个函数同一个scope里对某个mutex加锁和解锁。避免在foo()里加锁，然后跑到bar()里解锁；也避免了在不同的语句分支中分别加锁、解锁。这种做法被称为Scoped Locking。
* 在每次构造Guard对象的时候，思考一路上（调用栈上）已经持有的锁，防止因加锁顺序不同而导致死锁。由于Guard对象是栈上对象，看函数调用栈就能分析用锁的情况。

次要原则有：

不使用跨进程的mutex，进程间通信只用TCP Socket。

加锁、解锁在同一个线程，线程a不能去unlock线程b已经锁住的mutex（RAII自动保证）。

别忘了解锁（RAII自动保证）。

不重复加锁（RAII自动保证）。

必要的时候可以考虑用PTHREAD\_MUTEX\_ERRORCHECK来排错。

### 只使用非递归的mutex

mutex分为递归（recursive）和非递归（non-recursive）两种，另外的名字是重入（reentrant）和非可重入（non-reentrant）。这两种mutex作为线程间的同步工具时没有区别，它们的唯一区别在于：同一个线程可以对recursive mutex加锁，但不能重复对non-recursive mutex加锁。

在同一个线程里多次对non-recursive mutex加锁会立刻导致死锁，这是它的优势，能帮助我们思考代码对锁的期求，并且及早（在编码阶段）发现问题。

### 死锁

死锁会让程序行为失常，其他一些锁使用不当则会影响性能，例如潘爱民老师写的《Lock Convoys Explained》详细解释了一种性能衰退的现象。http://blog.csdn.net/panaimin/article/details/5981766。

编写高性能多线程程序至少需要知道false sharing和CPU cache效应，可以看几篇入门文章：

<http://www.nwcpp.org/Downloads/2007/Machine_Architecture_-_NWCPP.pdf>

<http://www.aristeia.com/TalkNotes/ACCU2011_CPUCaches.pdf>

<http://igoro.com/archive/gallery-of-processor-cache-effects/>

<http://simplygenius.net/Article/FalseSharing>

## 条件变量（condition variable）

互斥器是加锁原语，用来排他性的访问共享数据，它不是等待原语。

如果需要等待某个条件成立，应该使用条件变量。条件变量是一个或多个线程等待某个布尔表达式为真，即等待别的线程“唤醒”它。

条件变量只有一个正确使用的方式，几乎不可能出错。对于wait端：

* 必须与mutex一起使用，该布尔表达式的读写需受此mutex保护。
* 在mutex已上锁的时候才能调用wait()。
* 把判断布尔条件和wait()放到while循环中。

对于signal/broadcast端：

不一定要在mutex已上锁的情况下调用signal（理论上）。

在signal之前一般要修改布尔表达式。

修改布尔表达式通常要用mutex保护（至少用作full memory barrier）。

注意区分signal和broadcast：broadcast通常用于表明状态变化，signal通常用于表示资源可用。