第 1 章

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（synchronization primitives）保护内部状态即可。但是对象的生与死不能由对象自身拥有的 mutex（互斥器）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 库中的 `shared_ptr` 和 `weak_ptr` [1] 完美解决。这也是实现线程安全的 Observer 模式的必备技术。

本章源自 2009 年 12 月我在上海祝成科技举办的 C++ 技术大会的一场演讲《当析构函数遇到多线程》，读者应具有 C++ 多线程编程经验，熟悉互斥器、竞态条件等概念，了解智能指针，知道 Observer 设计模式。

1.1 当析构函数遇到多线程

与其他面向对象语言不同，C++ 要求程序员自己管理对象的生命期，这在多线程环境下显得尤为困难。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？

解决这些 race condition 是 C++ 多线程编程面临的基本问题。本文试图以 `shared_ptr` 一劳永逸地解决这些问题，减轻 C++ 多线程编程的精神负担。

[1] 这两个 class 也是 TR1 的一部分，位于 `std::tr1` 命名空间；在 C++11 中，它们是标准库的一部分。

1.1.1 线程安全的定义

依据 [JCP]，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）。
- 调用端代码无须额外的同步或其他协调动作。

依据这个定义，C++ 标准库里的大多数 class 都不是线程安全的，包括 `std::string`、`std::vector`、`std::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。

1.1.2 MutexLock 与 MutexLockGuard

为了便于后文讨论，先约定两个工具类。我相信每个写 C++ 多线程程序的人都实现过或使用过类似功能的类，代码见 §2.4。

`MutexLock` 封装临界区（critical section），这是一个简单的资源类，用 RAII 手法 [CCS，条款 13] 封装互斥器的创建与销毁。临界区在 Windows 上是 struct `CRITICAL_SECTION`，是可重入的；在 Linux 下是 `pthread_mutex_t`，默认是不可重入的 [2]。`MutexLock` 一般是别的 class 的数据成员。

`MutexLockGuard` 封装临界区的进入和退出，即加锁和解锁。`MutexLockGuard` 一般是个栈上对象，它的作用域刚好等于临界区域。

这两个 class 都不允许拷贝构造和赋值，它们的使用原则见 §2.1。

1.1.3 一个线程安全的 Counter 示例

编写单个的线程安全的 class 不算太难，只需用同步原语保护其内部状态。例如下面这个简单的计数器类 `Counter`：

```
1  // A thread-safe counter
2  class Counter : boost::noncopyable
3  {
4    // copy-ctor and assignment should be private by default for a class.
5   public:
6    Counter() : value_(0) {}
```

[2] 可重入与不可重入的讨论见 §2.1.1。

```
7     int64_t value() const;
8     int64_t getAndIncrease();
9
10   private:
11    int64_t value_;
12    mutable MutexLock mutex_;
13  };
14
15  int64_t Counter::value() const
16  {
17    MutexLockGuard lock(mutex_);  // lock 的析构会晚于返回对象的构造，
18    return value_;                // 因此有效地保护了这个共享数据。
19  }
20
21  int64_t Counter::getAndIncrease()
22  {
23    MutexLockGuard lock(mutex_);
24    int64_t ret = value_++;
25    return ret;
26  }
27  // In a real world, atomic operations are preferred.
28  // 当然在实际项目中，这个 class 用原子操作更合理，这里用锁仅仅为了举例。
```

这个 class 很直白，一看就明白，也容易验证它是线程安全的。每个 Counter 对象有自己的 mutex_，因此不同对象之间不构成锁争用（lock contention）。即两个线程有可能同时执行 $\mathcal{L}$24，前提是它们访问的不是同一个 Counter 对象。注意到其 mutex_ 成员是 mutable 的，意味着 const 成员函数如 Counter::value() 也能直接使用 non-const 的 mutex_。思考：如果 mutex_ 是 static，是否影响正确性和/或性能？

尽管这个 Counter 本身毫无疑问是线程安全的，但如果 Counter 是动态创建的并通过指针来访问，前面提到的对象销毁的 race condition 仍然存在。

1.2 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。

之所以这样规定，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（escape）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。

```
// 不要这么做（Don't do this.）
class Foo : public Observer  // Observer 的定义见第 10 页
{
 public:
  Foo(Observable* s)
  {
    s->register_(this);  // 错误，非线程安全
  }

  virtual void update();
};
```

对象构造的正确方法：

```
// 要这么做（Do this.）
class Foo : public Observer
{
 public:
  Foo();
  virtual void update();

  // 另外定义一个函数，在构造之后执行回调函数的注册工作
  void observe(Observable* s)
  {
    s->register_(this);
  }
};

Foo* pFoo = new Foo;
Observable* s = getSubject();
pFoo->observe(s);  // 二段式构造，或者直接写 s->register_(pFoo);
```

这也说明，二段式构造——即构造函数 +initialize()——有时会是好办法，这虽然不符合 C++ 教条，但是多线程下别无选择。另外，既然允许二段式构造，那么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。

即使构造函数的最后一行也不要泄露 this，因为 Foo 有可能是个基类，基类先于派生类构造，执行完 Foo::Foo() 的最后一行代码还会继续执行派生类的构造函数，这时 most-derived class 的对象还处于构造中，仍然不安全。

相对来说，对象的构造做到线程安全还是比较容易的，毕竟曝光少，回头率为零。而析构的线程安全就不那么简单，这也是本章关注的焦点。

1.3 销毁太难

对象析构，这在单线程里不构成问题，最多需要注意避免空悬指针和野指针[3]。而在多线程程序中，存在了太多的竞态条件。对一般成员函数而言，做到线程安全的办法是让它们顺次执行，而不要并发执行（关键是不要同时读写共享状态），也就是让每个成员函数的临界区不重叠。这是显而易见的，不过有一个隐含条件或许不是每个人都能立刻想到：成员函数用来保护临界区的互斥器本身必须是有效的。而析构函数破坏了这一假设，它会把 mutex 成员变量销毁掉。悲剧啊！

1.3.1 mutex 不是办法

mutex 只能保证函数一个接一个地执行，考虑下面的代码，它试图用互斥锁来保护析构函数：（注意代码中的 (1) 和 (2) 两处标记。）

```
Foo::~Foo()
{
  MutexLockGuard lock(mutex_);
  // free internal state  (1)
}
```

```
void Foo::update()
{
  MutexLockGuard lock(mutex_);  // (2)
  // make use of internal 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，而线程 B 正准备调用 x->update()。

```
extern Foo* x; // visible by all threads
```

```
// thread A
delete x;
x = NULL;  // helpless
```

```
// thread B
if (x) {
  x->update();
}
```

尽管线程 A 在销毁对象之后把指针置为了 NULL，尽管线程 B 在调用 x 的成员函数之前检查了指针 x 的值，但还是无法避免一种 race condition：

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 通过了 if (x) 检测，阻塞在 (2) 处。

接下来会发生什么，只有天晓得。因为析构函数会把 mutex_ 销毁，那么 (2) 处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情况。

这个例子至少说明 delete 对象之后把指针置为 NULL 根本没用，如果一个程序要靠这个来防止二次释放，说明代码逻辑出了问题。

[3] 空悬指针（dangling pointer）指向已经销毁的对象或已经回收的地址，野指针（wild pointer）指的是未经初始化的指针（http://en.wikipedia.org/wiki/Dangling_pointer）。

1.3.2 作为数据成员的 mutex 不能保护析构

前面的例子说明，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（或者身亡之时）。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时，析构才是安全的，否则会有 §1.1 谈到的竞态条件发生。

另外如果要同时读写一个 class 的两个对象，有潜在的死锁可能。比方说有 swap() 这个函数：

```
void swap(Counter& a, Counter& b)
{
  MutexLockGuard aLock(a.mutex_);  // potential dead lock
  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。operator=() 也是类似的道理。

```
Counter& Counter::operator=(const Counter& rhs)
{
  if (this == &rhs)
    return *this;

  MutexLockGuard myLock(mutex_);   // potential dead lock
  MutexLockGuard itsLock(rhs.mutex_);
  value_ = rhs.value_;  // 改成 value_ = rhs.value() 会死锁
  return *this;
}
```

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。

1.4 线程安全的 Observer 有多难

一个动态创建的对象是否还活着，光看指针是看不出来的（引用也一样看不出来）。指针就是指向了一块内存，这块内存上的对象如果已经销毁，那么就根本不能

访问 [CCS，条款 99]（就像 `free(3)` 之后的地址不能访问一样），既然不能访问又如何知道对象的状态呢？换句话说，判断一个指针是不是合法指针没有高效的办法，这是 C/C++ 指针问题的根源[4]。（万一原址又创建了一个新的对象呢？再万一这个新的对象的类型异于老的对象呢？）

在面向对象程序设计中，对象的关系主要有三种：composition、aggregation、association。composition（组合/复合）关系在多线程里不会遇到什么麻烦，因为对象 x 的生命期由其唯一的拥有者 owner 控制，owner 析构的时候会把 x 也析构掉。从形式上看，x 是 owner 的直接数据成员，或者 `scoped_ptr` 成员，抑或 owner 持有的容器的元素。

后两种关系在 C++ 里比较难办，处理不好就会造成内存泄漏或重复释放。association（关联/联系）是一种很宽泛的关系，它表示一个对象 a 用到了另一个对象 b，调用了后者的成员函数。从代码形式上看，a 持有 b 的指针（或引用），但是 b 的生命期不由 a 单独控制。aggregation（聚合）关系从形式上看与 association 相同，除了 a 和 b 有逻辑上的整体与部分关系。如果 b 是动态创建的并在整个程序结束前有可能被释放，那么就会出现 §1.1 谈到的竞态条件。

那么似乎一个简单的解决办法是：只创建不销毁。程序使用一个对象池来暂存用过的对象，下次申请新对象时，如果对象池里有存货，就重复利用现有的对象，否则就新建一个。对象用完了，不是直接释放掉，而是放回池子里。这个办法当然有其自身的很多缺点，但至少能避免访问失效对象的情况发生。

这种山寨办法的问题有：

- 对象池的线程安全，如何安全地、完整地把对象放回池子里，防止出现“部分放回”的竞态？（线程 A 认为对象 x 已经放回了，线程 B 认为对象 x 还活着。）
- 全局共享数据引发的 lock contention，这个集中化的对象池会不会把多线程并发的操作串行化？
- 如果共享对象的类型不止一种，那么是重复实现对象池还是使用类模板？
- 会不会造成内存泄漏与分片？因为对象池占用的内存只增不减，而且多个对象池不能共享内存（想想为何）。

回到正题上来，如果对象 x 注册了任何非静态成员函数回调，那么必然在某处持有了指向 x 的指针，这就暴露在了 race condition 之下。

[4] 在 Java 中，一个 reference 只要不为 null，它一定指向有效的对象。

一个典型的场景是 Observer 模式（代码见 recipes/thread/test/Observer.cc）。

```
class Observer // : boost::noncopyable
{
 public:
  virtual ~Observer();
  virtual void update() = 0;
  // ...
};

class Observable // : boost::noncopyable
{
 public:
  void register_(Observer* x);
  void unregister(Observer* x);

  void notifyObservers() {
    for (Observer* x : observers_) {  // 这行是 C++11
      x->update(); // (3)
    }
  }
 private:
  std::vector<Observer*> observers_;
};
```

当 Observable 通知每一个 Observer 时 (*L*17)，它从何得知 Observer 对象 x 还活着？要不试试在 Observer 的析构函数里调用 unregister() 来解注册？恐难奏效。

```
class Observer
{
  // 同前
  void observe(Observable* s) {
    s->register_(this);
    subject_ = s;
  }

  virtual ~Observer() {
    subject_->unregister(this);
  }

  Observable* subject_;
};
```

我们试着让 Observer 的析构函数去调用 unregister(this)，这里有两个 race conditions。其一：*L*32 如何得知 subject_ 还活着？其二：就算 subject_ 指向某个永久存在的对象，那么还是险象环生：

1. 线程 A 执行到 *L*32 之前，还没有来得及 unregister 本对象。
2. 线程 B 执行到 *L*17，x 正好指向是 *L*32 正在析构的对象。

这时悲剧又发生了，既然 x 所指的 Observer 对象正在析构，调用它的任何非静态成员函数都是不安全的，何况是虚函数[5]。更糟糕的是，Observer 是个基类，执行到 $\mathcal{L}32$ 时，派生类对象已经析构掉了，这时候整个对象处于将死未死的状态，core dump 恐怕是最幸运的结果。

这些 race condition 似乎可以通过加锁来解决，但在哪儿加锁，谁持有这些互斥锁，又似乎不是那么显而易见的。要是有什么活着的对象能帮帮我们就好了，它提供一个 isAlive() 之类的程序函数，告诉我们那个对象还在不在。可惜指针和引用都不是对象，它们是内建类型。

1.5 原始指针有何不妥

指向对象的原始指针（raw pointer）是坏的，尤其当暴露给别的线程时。Observable 应当保存的不是原始的 Observer*，而是别的什么东西，能分辨 Observer 对象是否存活。类似地，如果 Observer 要在析构函数里解注册（这虽然不能解决前面提到的 race condition，但是在析构函数里打扫战场还是应该的），那么 subject_ 的类型也不能是原始的 Observable*。

有经验的 C++ 程序员或许会想到用智能指针。没错，这是正道，但也没那么简单，有些关窍需要注意。这两处直接使用 shared_ptr 是不行的，会形成循环引用，直接造成资源泄漏。别着急，后文会一一讲到。

空悬指针

有两个指针 p1 和 p2，指向堆上的同一个对象 Object，p1 和 p2 位于不同的线程中（图 1-1 的左图）。假设线程 A 通过 p1 指针将对象销毁了（尽管把 p1 置为了 NULL），那 p2 就成了空悬指针（图 1-1 的右图）。这是一种典型的 C/C++ 内存错误。

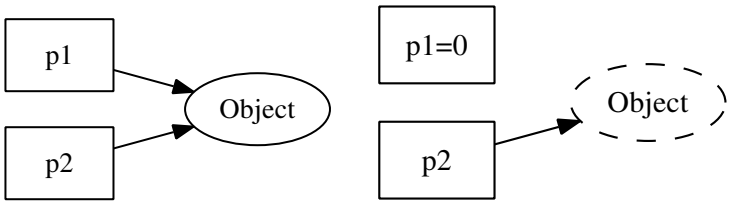

图 1-1

要想安全地销毁对象，最好在别人（线程）都看不到的情况下，偷偷地做。（这正是垃圾回收的原理，所有人都用不到的东西一定是垃圾。）

[5] C++ 标准对在构造函数和析构函数中调用虚函数的行为有明确规定，但是没有考虑并发调用的情况。

一个“解决办法”

一个解决空悬指针的办法是，引入一层间接性，让 p1 和 p2 所指的对象永久有效。比如图 1-2 中的 proxy 对象，这个对象，持有一个指向 `Object` 的指针。（从 C 语言的角度，p1 和 p2 都是二级指针。）

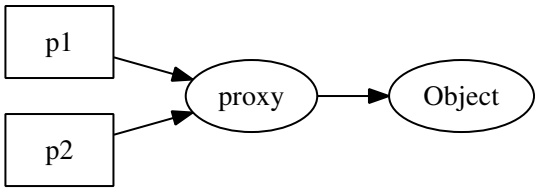

图 1-2

当销毁 `Object` 之后，proxy 对象继续存在，其值变为 0（见图 1-3）。而 p2 也没有变成空悬指针，它可以通过查看 proxy 的内容来判断 Object 是否还活着。

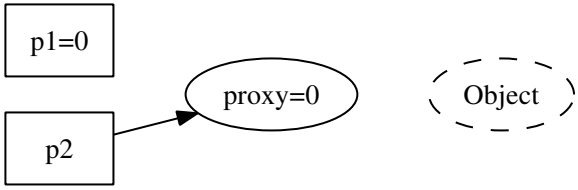

图 1-3

要线程安全地释放 `Object` 也不是那么容易，race condition 依旧存在。比如 p2 看第一眼的时候 proxy 不是零，正准备去调用 `Object` 的成员函数，期间对象已经被 p1 给销毁了。

问题在于，何时释放 proxy 指针呢？

一个更好的解决办法

为了安全地释放 proxy，我们可以引入引用计数（reference counting），再把 p1 和 p2 都从指针变成对象 sp1 和 sp2。proxy 现在有两个成员，指针和计数器。

1. 一开始，有两个引用，计数值为 2（见图 1-4）。

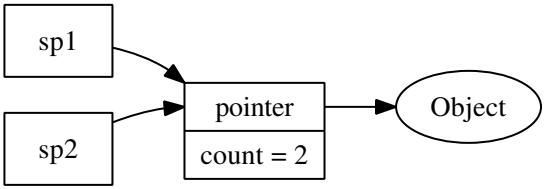

图 1-4

2. sp1 析构了，引用计数的值减为 1（见图 1-5）。

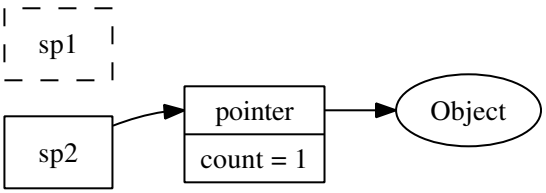

图 1-5

3. sp2 也析构了，引用计数降为 0，可以安全地销毁 proxy 和 `Object` 了（见图 1-6）。

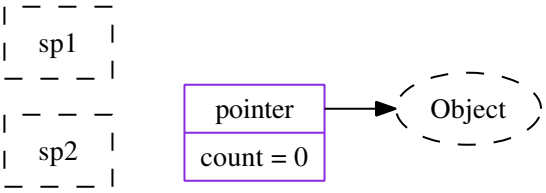

图 1-6

慢着！这不正是引用计数型智能指针吗？

一个万能的解决方案

引入另外一层间接性（another layer of indirection）[6]，用对象来管理共享资源（如果把 `Object` 看作资源的话），亦即 handle/body 惯用技法（idiom）。当然，编写线程安全、高效的引用计数 handle 的难度非凡，作为一名谦卑的程序员[7]，用现成的库就行。万幸，C++ 的 TR1 标准库里提供了一对“神兵利器”，可助我们完美解决这个头疼的问题。

1.6 神器 shared_ptr/weak_ptr

`shared_ptr` 是引用计数型智能指针，在 Boost 和 `std::tr1` 里均提供，也被纳入 C++11 标准库，现代主流的 C++ 编译器都能很好地支持。`shared_ptr<T>` 是一个类模板（class template），它只有一个类型参数，使用起来很方便。引用计数是自动化

[6] http://en.wikipedia.org/wiki/Abstraction_layer

[7] 参见 Edsger W. Dijkstra 的著名演讲《The Humble Programmer》（http://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html）。

资源管理的常用手法，当引用计数降为 0 时，对象（资源）即被销毁。weak_ptr 也是一个引用计数型智能指针，但是它不增加对象的引用次数，即弱（weak）引用。

shared_ptr 的基本用法和语意请参考手册或教程，本书从略。谈几个关键点。

- shared_ptr 控制对象的生命期。shared_ptr 是强引用（想象成用铁丝绑住堆上的对象），只要有一个指向 x 对象的 shared_ptr 存在，该 x 对象就不会析构。当指向对象 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（想象成用棉线轻轻拴住堆上的对象）。如果对象还活着，那么它可以提升（promote）为有效的 shared_ptr；如果对象已经死了，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 std::string 和 STL 容器一样，后面还会讲。

孟岩在《垃圾收集机制批判》[8] 中一针见血地点出智能指针的优势：“C++ 利用智能指针达成的效果是：一旦某对象不再被引用，系统刻不容缓，立刻回收内存。这通常发生在关键任务完成后的清理（clean up）时期，不会影响关键任务的实时性，同时，内存里所有的对象都是有用的，绝对没有垃圾空占内存。”

1.7 插曲：系统地避免各种指针错误

我同意孟岩说的[9] “大部分用 C 写的上规模的软件都存在一些内存方面的错误，需要花费大量的精力和时间把产品稳定下来。”举例来说，就像 Nginx 这样成熟且广泛使用的 C 语言产品都会不时暴露出低级的内存错误[10]。

内存方面的问题在 C++ 里很容易解决，我第一次也是最后一次见到别人的代码里有内存泄漏是在 2004 年实习那会儿，我自己写的 C++ 程序从来没有出现过内存方面的问题。

[8] http://blog.csdn.net/myan/article/details/1906

[9]《Java 替代 C 语言的可能性》（http://blog.csdn.net/myan/article/details/1482614）。

[10] http://trac.nginx.org/nginx/ticket/{134,135,162}

C++ 里可能出现的内存问题大致有这么几个方面：

1. 缓冲区溢出（buffer overrun）。
2. 空悬指针/野指针。
3. 重复释放（double delete）。
4. 内存泄漏（memory leak）。
5. 不配对的 new[]/delete。
6. 内存碎片（memory fragmentation）。

正确使用智能指针能很轻易地解决前面 5 个问题，解决第 6 个问题需要别的思路，我会在 §9.2.1 和 §A.1.8 探讨。

1. 缓冲区溢出：用 `std::vector<char>/std::string` 或自己编写 `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。
2. 空悬指针/野指针：用 `shared_ptr/weak_ptr`，这正是本章的主题。
3. 重复释放：用 `scoped_ptr`，只在对象析构的时候释放一次。
4. 内存泄漏：用 `scoped_ptr`，对象析构的时候自动释放内存。
5. 不配对的 new[]/delete：把 new[] 统统替换为 `std::vector/scoped_array`。

正确使用上面提到的这几种智能指针并不难，其难度大概比学习使用 `std::vector/std::list` 这些标准库组件还要小，与 `std::string` 差不多，只要花一周的时间去适应它，就能信手拈来。我认为，在现代的 C++ 程序中一般不会出现 delete 语句，资源（包括复杂对象本身）都是通过对象（智能指针或容器）来管理的，不需要程序员还为此操心。

在这几种错误里边，内存泄漏相对危害性较小，因为它只是借了东西不归还，程序功能在一段时间内还算正常。其他如缓冲区溢出或重复释放等致命错误可能会造成安全性（security 和 data safety）方面的严重后果。

需要注意一点：`scoped_ptr/shared_ptr/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。几乎不会有下面这种用法：

```
shared_ptr<Foo>* pFoo = new shared_ptr<Foo>(new Foo);  // WRONG semantic
```

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的或内联的，必须在 .cpp 文件里边显式定义，否则会有编译错或运行错（原因见 §10.3.2）。

1.8 应用到 Observer 上

既然通过 weak_ptr 能探查对象的生死，那么 Observer 模式的竞态条件就很容易解决，只要让 Observable 保存 weak_ptr<Observer> 即可：

recipes/thread/test/Observer_safe.cc

```
39  class Observable  // not 100% thread safe!
40  {
41   public:
42    void register_(weak_ptr<Observer> x);  // 参数类型可用 const weak_ptr<Observer>&
43    // void unregister(weak_ptr<Observer> x);  // 不需要它
44    void notifyObservers();
45
46   private:
47    mutable MutexLock mutex_;
48    std::vector<weak_ptr<Observer> > observers_;
49    typedef std::vector<weak_ptr<Observer> >::iterator Iterator;
50  };
51
52  void Observable::notifyObservers()
53  {
54    MutexLockGuard lock(mutex_);
55    Iterator it = observers_.begin();  // Iterator 的定义见第 49 行
56    while (it != observers_.end())
57    {
58      shared_ptr<Observer> obj(it->lock());  // 尝试提升，这一步是线程安全的
59      if (obj)
60      {
61        // 提升成功，现在引用计数值至少为 2 （想想为什么？）
62        obj->update();  // 没有竞态条件，因为 obj 在栈上，对象不可能在本作用域内销毁
63        ++it;
64      }
65      else
66      {
67        // 对象已经销毁，从容器中拿掉 weak_ptr
68        it = observers_.erase(it);
69      }
70    }
71  }
```

recipes/thread/test/Observer_safe.cc

就这么简单。前文代码 (3) 处（p. 10 的 £17）的竞态条件已经弥补了。思考：如果把 £48 改为 vector<shared_ptr<Observer> > observers_;，会有什么后果？

解决了吗

把 Observer* 替换为 weak_ptr<Observer> 部分解决了 Observer 模式的线程安全，但还有以下几个疑点。这些问题留到本章 §1.14 中去探讨，每个都是能解决的。

侵入性 强制要求 Observer 必须以 shared_ptr 来管理。

不是完全线程安全 Observer 的析构函数会调用 subject_->unregister(this)，万一 subject_ 已经不复存在了呢？为了解决它，又要求 Observable 本身是用 shared_ptr 管理的，并且 subject_ 多半是个 weak_ptr<Observable>。

锁争用（lock contention） 即 Observable 的三个成员函数都用了互斥器来同步，这会造成 register_() 和 unregister() 等待 notifyObservers()，而后者的执行时间是无上限的，因为它同步回调了用户提供的 update() 函数。我们希望 register_() 和 unregister() 的执行时间不会超过某个固定的上限，以免殃及无辜群众。

死锁 万一 £62 的 update() 虚函数中调用了 (un)register 呢？如果 mutex_ 是不可重入的，那么会死锁；如果 mutex_ 是可重入的，程序会面临迭代器失效（core dump 是最好的结果），因为 vector observers_ 在遍历期间被意外地修改了。这个问题乍看起来似乎没有解决办法，除非在文档里做要求。（一种办法是：用可重入的 mutex_，把容器换为 std::list，并把 ++it 往前挪一行。）

我个人倾向于使用不可重入的 mutex，例如 Pthreads 默认提供的那个，因为“要求 mutex 可重入”本身往往意味着设计上出了问题（§2.1.1）。Java 的 intrinsic lock 是可重入的，因为要允许 synchronized 方法相互调用（派生类调用基类的同名 synchronized 方法），我觉得这也是无奈之举。

1.9 再论 shared_ptr 的线程安全

虽然我们借 shared_ptr 来实现线程安全的对象释放，但是 shared_ptr 本身不是 100% 线程安全的。它的引用计数本身是安全且无锁的，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。根据文档 [11]，shared_ptr 的线程安全级别和内建类型、标准库容器、std::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。

请注意，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。

[11] http://www.boost.org/doc/libs/release/libs/smart_ptr/shared_ptr.htm#ThreadSafety

要在多个线程中同时访问同一个 shared_ptr，正确的做法是用 mutex 保护：

```
MutexLock mutex; // No need for ReaderWriterLock
shared_ptr<Foo> globalPtr;

// 我们的任务是把 globalPtr 安全地传给 doit()
void doit(const shared_ptr<Foo>& pFoo);
```

globalPtr 能被多个线程看到，那么它的读写需要加锁。注意我们不必用读写锁，而只用最简单的互斥锁，这是为了性能考虑。因为临界区非常小，用互斥锁也不会阻塞并发读。

为了拷贝 globalPtr，需要在读取它的时候加锁，即：

```
void read()
{
  shared_ptr<Foo> localPtr;
  {
    MutexLockGuard lock(mutex);
    localPtr = globalPtr;  // read globalPtr
  }
  // use localPtr since here, 读写 localPtr 也无须加锁
  doit(localPtr);
}
```

写入的时候也要加锁：

```
void write()
{
  shared_ptr<Foo> newPtr(new Foo);  // 注意，对象的创建在临界区之外
  {
    MutexLockGuard lock(mutex);
    globalPtr = newPtr;  // write to globalPtr
  }
  // use newPtr since here, 读写 newPtr 无须加锁
  doit(newPtr);
}
```

注意到上面的 read() 和 write() 在临界区之外都没有再访问 globalPtr，而是用了一个指向同一 Foo 对象的栈上 shared_ptr local copy。下面会谈到，只要有这样的 local copy 存在，shared_ptr 作为函数参数传递时不必复制，用 reference to const 作为参数类型即可。另外注意到上面的 new Foo 是在临界区之外执行的，这种写法通常比在临界区内写 globalPtr.reset(new Foo) 要好，因为缩短了临界区长度。如果要销毁对象，我们固然可以在临界区内执行 globalPtr.reset()，但是这样往往会让对象析构发生在临界区以内，增加了临界区的长度。一种改进办法是像上面一样定义一个 localPtr，用它在临界区内与 globalPtr 交换（swap()），这样能保证把对象的销毁推迟到临界区之外。练习：在 write() 函数中，globalPtr = newPtr; 这一句有可能会在临界区内销毁原来 globalPtr 指向的 Foo 对象，设法将销毁行为移出临界区。

1.10 shared_ptr 技术与陷阱

意外延长对象的生命期 shared_ptr 是强引用（“铁丝”绑的），只要有一个指向 x 对象的 shared_ptr 存在，该对象就不会析构。而 shared_ptr 又是允许拷贝构造和赋值的（否则引用计数就无意义了），如果不小心遗留了一个拷贝，那么对象就永世长存了。例如前面提到如果把 p. 16 中 £48 observers_ 的类型改为 vector<shared_ptr<Observer> >，那么除非手动调用 unregister()，否则 Observer 对象永远不会析构。即便它的析构函数会调用 unregister()，但是不去 unregister() 就不会调用 Observer 的析构函数，这变成了鸡与蛋的问题。这也是 Java 内存泄漏的常见原因。

另外一个出错的可能是 boost::bind，因为 boost::bind 会把实参拷贝一份，如果参数是个 shared_ptr，那么对象的生命期就不会短于 boost::function 对象：

```
class Foo
{
  void doit();
};

shared_ptr<Foo> pFoo(new Foo);
boost::function<void()> func = boost::bind(&Foo::doit, pFoo);  // long life foo
```

这里 func 对象持有了 shared_ptr<Foo> 的一份拷贝，有可能会在不经意间延长倒数第二行创建的 Foo 对象的生命期。

函数参数 因为要修改引用计数（而且拷贝的时候通常要加锁），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。例如有几个函数都要用到 Foo 对象：

```
void save(const shared_ptr<Foo>& pFoo);     // pass by const reference
void validateAccount(const Foo& foo);

bool validate(const shared_ptr<Foo>& pFoo)  // pass by const reference
{
  validateAccount(*pFoo);
  // ...
}
```

那么在通常情况下，我们可以传常引用（pass by const reference）：

```
void onMessage(const string& msg)
{
  shared_ptr<Foo> pFoo(new Foo(msg));  // 只要在最外层持有一个实体，安全不成问题
  if (validate(pFoo)) {  // 没有拷贝 pFoo
    save(pFoo);          // 没有拷贝 pFoo
  }
}
```

遵照这个规则，基本上不会遇到反复拷贝 shared_ptr 导致的性能问题。另外由于 pFoo 是栈上对象，不可能被别的线程看到，那么读取始终是线程安全的。

析构动作在创建时被捕获 这是一个非常有用的特性，这意味着：

- 虚析构不再是必需的。
- 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，而无须重新编译。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。

最后这个特性的实现比较巧妙，因为 shared_ptr<T> 只有一个模板参数，而“析构行为”可以是函数指针、仿函数（functor）或者其他什么东西。这是泛型编程和面向对象编程的一次完美结合。有兴趣的读者可以参考 Scott Meyers 的文章 [12]。这个技术在后面的对象池中还会用到。

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 我认为 RAII （资源获取即初始化）是 C++ 语言区别于其他所有编程语言的最重要的特性，一个不懂 RAII 的 C++ 程序员不是一个合格的 C++ 程序员。初学 C++ 的教条是“new 和 delete 要配对，new 了之后要记着 delete”；如果使用 RAII [CCS，条款 13]，要改成“每一个明确的资源配置动作（例如 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），程序中一般不出现 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。

[12] http://www.artima.com/cppsource/top_cpp_aha_moments.html

1.11 对象池

假设有 Stock 类，代表一只股票的价格。每一只股票有一个唯一的字符串标识，比如 Google 的 key 是 "NASDAQ:GOOG"，IBM 是 "NYSE:IBM"。Stock 对象是个主动对象，它能不断获取新价格。为了节省系统资源，同一个程序里边每一只出现的股票只有一个 Stock 对象，如果多处用到同一只股票，那么 Stock 对象应该被共享。如果某一只股票没有再在任何地方用到，其对应的 Stock 对象应该析构，以释放资源，这隐含了“引用计数”。

为了达到上述要求，我们可以设计一个对象池 StockFactory [13]。它的接口很简单，根据 key 返回 Stock 对象。我们已经知道，在多线程程序中，既然对象可能被销毁，那么返回 shared_ptr 是合理的。自然地，我们写出如下代码（可惜是错的）。

```
// version 1: questionable code
class StockFactory : boost::noncopyable
{
 public:
  shared_ptr<Stock> get(const string& key);

 private:
  mutable MutexLock mutex_;
  std::map<string, shared_ptr<Stock> > stocks_;
};
```

get() 的逻辑很简单，如果在 stocks_ 里找到了 key，就返回 stocks_[key]；否则新建一个 Stock，并存入 stocks_[key]。

细心的读者或许已经发现这里有一个问题，Stock 对象永远不会被销毁，因为 map 里存的是 shared_ptr，始终有“铁丝”绑着。那么或许应该仿照前面 Observable 那样存一个 weak_ptr？比如

```
// // version 2: 数据成员修改为 std::map<string, weak_ptr<Stock> > stocks_;
shared_ptr<Stock> StockFactory::get(const string& key)
{
  shared_ptr<Stock> pStock;
  MutexLockGuard lock(mutex_);
  weak_ptr<Stock>& wkStock = stocks_[key];  // 如果 key 不存在，会默认构造一个
  pStock = wkStock.lock();  // 尝试把“棉线”提升为“铁丝”
  if (!pStock) {
    pStock.reset(new Stock(key));
    wkStock = pStock;  // 这里更新了 stocks_[key]，注意 wkStock 是个引用
  }
  return pStock;
}
```

[13] recipes/thread/test/Factory.cc 包含这里提到的各个版本。

这么做固然 Stock 对象是销毁了，但是程序却出现了轻微的内存泄漏，为什么？

因为 stocks_ 的大小只增不减，stocks_.size() 是曾经存活过的 Stock 对象的总数，即便活的 Stock 对象数目降为 0。或许有人认为这不算泄漏，因为内存并不是彻底遗失不能访问了，而是被某个标准库容器占用了。我认为这也算内存泄漏，毕竟是“战场”没有打扫干净。

其实，考虑到世界上的股票数目是有限的，这个内存不会一直泄漏下去，大不了把每只股票的对象都创建一遍，估计泄漏的内存也只有几兆字节。如果这是一个其他类型的对象池，对象的 key 的集合不是封闭的，内存就会一直泄漏下去。

解决的办法是，利用 shared_ptr 的定制析构功能。shared_ptr 的构造函数可以有一个额外的模板类型参数，传入一个函数指针或仿函数 d，在析构对象时执行 d(ptr)，其中 ptr 是 shared_ptr 保存的对象指针。shared_ptr 这么设计并不是多余的，因为反正要在创建对象时捕获释放动作，始终需要一个 bridge。

```
template<class Y, class D> shared_ptr::shared_ptr(Y* p, D d);
template<class Y, class D> void shared_ptr::reset(Y* p, D d);
// 注意 Y 的类型可能与 T 不同，这是合法的，只要 Y* 能隐式转换为 T*。
```

那么我们可以利用这一点，在析构 Stock 对象的同时清理 stocks_。

```
// version 3
class StockFactory : boost::noncopyable
{
  // 在 get() 中，将 pStock.reset(new Stock(key)); 改为：
  // pStock.reset(new Stock(key),
  //              boost::bind(&StockFactory::deleteStock, this, _1));  // ***

 private:
  void deleteStock(Stock* stock)
  {
    if (stock) {
      MutexLockGuard lock(mutex_);
      stocks_.erase(stock->key());
    }
    delete stock;  // sorry, I lied
  }
  // assuming StockFactory lives longer than all Stock's ...
  // ...
```

这里我们向 pStock.reset() 传递了第二个参数，一个 boost::function，让它在析构 Stock* p 时调用本 StockFactory 对象的 deleteStock 成员函数。

警惕的读者可能已经发现问题，那就是我们把一个原始的 StockFactory this 指针保存在了 boost::function 里（*** 处），这会有线程安全问题。如果这个 StockFactory 先于 Stock 对象析构，那么会 core dump。正如 Observer 在析构函数里去调用 Observable::unregister()，而那时 Observable 对象可能已经不存在了。

当然这也是能解决的，要用到 §1.11.2 介绍的弱回调技术。

1.11.1 enable_shared_from_this

StockFactory::get() 把原始指针 this 保存到了 boost::function 中（*** 处），如果 StockFactory 的生命期比 Stock 短，那么 Stock 析构时去回调 StockFactory::deleteStock 就会 core dump。似乎我们应该祭出惯用的 shared_ptr 大法来解决对象生命期问题，但是 StockFactory::get() 本身是个成员函数，如何获得一个指向当前对象的 shared_ptr<StockFactory> 对象呢？

有办法，用 enable_shared_from_this。这是一个以其派生类为模板类型实参的基类模板 [14]，继承它，this 指针就能变身为 shared_ptr。

```
class StockFactory : public boost::enable_shared_from_this<StockFactory>,
                     boost::noncopyable
{ /* ... */ };
```

为了使用 shared_from_this()，StockFactory 不能是 stack object，必须是 heap object 且由 shared_ptr 管理其生命期，即：

```
shared_ptr<StockFactory> stockFactory(new StockFactory);
```

万事俱备，可以让 this 摇身一变，化为 shared_ptr<StockFactory> 了。

```
// version 4
shared_ptr<Stock> StockFactory::get(const string& key)
{
  // change
  // pStock.reset(new Stock(key),
  //              boost::bind(&StockFactory::deleteStock, this, _1));
  // to
  pStock.reset(new Stock(key),
               boost::bind(&StockFactory::deleteStock,
                           shared_from_this(),
                           _1));
  // ...
```

这样一来，boost::function 里保存了一份 shared_ptr<StockFactory>，可以保证调用 StockFactory::deleteStock 的时候那个 StockFactory 对象还活着。

注意一点，shared_from_this() 不能在构造函数里调用，因为在构造 StockFactory 的时候，它还没有被交给 shared_ptr 接管。

最后一个问题，StockFactory 的生命期似乎被意外延长了。

[14] http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

1.11.2 弱回调

把 shared_ptr 绑（boost::bind）到 boost:function 里，那么回调的时候 StockFactory 对象始终存在，是安全的。这同时也延长了对象的生命期，使之不短于绑得的 boost:function 对象。

有时候我们需要“如果对象还活着，就调用它的成员函数，否则忽略之”的语意，就像 Observable::notifyObservers() 那样，我称之为“弱回调”。这也是可以实现的，利用 weak_ptr，我们可以把 weak_ptr 绑到 boost::function 里，这样对象的生命期就不会被延长。然后在回调的时候先尝试提升为 shared_ptr，如果提升成功，说明接受回调的对象还健在，那么就执行回调；如果提升失败，就不必劳神了。

使用这一技术的完整 StockFactory 代码如下：

```
class StockFactory : public boost::enable_shared_from_this<StockFactory>,
                     boost::noncopyable
{
 public:
  shared_ptr<Stock> get(const string& key)
  {
    shared_ptr<Stock> pStock;
    MutexLockGuard lock(mutex_);
    weak_ptr<Stock>& wkStock = stocks_[key];  // 注意 wkStock 是引用
    pStock = wkStock.lock();
    if (!pStock)
    {
      pStock.reset(new Stock(key),
                   boost::bind(&StockFactory::weakDeleteCallback,
                               boost::weak_ptr<StockFactory>(shared_from_this()),
                               _1));
      // 上面必须强制把 shared_from_this() 转型为 weak_ptr, 才不会延长生命期,
      // 因为 boost::bind 拷贝的是实参类型, 不是形参类型
      wkStock = pStock;
    }
    return pStock;
  }

 private:
  static void weakDeleteCallback(const boost::weak_ptr<StockFactory>& wkFactory,
                                 Stock* stock)
  {
    shared_ptr<StockFactory> factory(wkFactory.lock());  // 尝试提升
    if (factory)   // 如果 factory 还在, 那就清理 stocks_
    {
      factory->removeStock(stock);
    }
    delete stock;  // sorry, I lied
  }
```

```
  void removeStock(Stock* stock)
  {
    if (stock)
    {
      MutexLockGuard lock(mutex_);
      stocks_.erase(stock->key());
    }
  }

 private:
  mutable MutexLock mutex_;
  std::map<string, weak_ptr<Stock> > stocks_;
};
```

两个简单的测试：

```
void testLongLifeFactory()
{
  shared_ptr<StockFactory> factory(new StockFactory);
  {
    shared_ptr<Stock> stock = factory->get("NYSE:IBM");
    shared_ptr<Stock> stock2 = factory->get("NYSE:IBM");
    assert(stock == stock2);
    // stock destructs here
  }
  // factory destructs here
}

void testShortLifeFactory()
{
  shared_ptr<Stock> stock;
  {
    shared_ptr<StockFactory> factory(new StockFactory);
    stock = factory->get("NYSE:IBM");
    shared_ptr<Stock> stock2 = factory->get("NYSE:IBM");
    assert(stock == stock2);
    // factory destructs here
  }
  // stock destructs here
}
```

这下完美了，无论 `Stock` 和 `StockFactory` 谁先挂掉都不会影响程序的正确运行。这里我们借助 `shared_ptr` 和 `weak_ptr` 完美地解决了两个对象相互引用的问题。

当然，通常 Factory 对象是个 singleton，在程序正常运行期间不会销毁，这里只是为了展示弱回调技术 [15]，这个技术在事件通知中非常有用。

本节的 `StockFactory` 只有针对单个 `Stock` 对象的操作，如果程序需要遍历整个 `stocks_`，稍不注意就会造成死锁或数据损坏（§2.1），请参考 §2.8 的解决办法。

[15] 通用的弱回调封装见 recipes/thread/WeakCallback.h，用到了 C++11 的 variadic template 和 rvalue reference。

1.12 替代方案

除了使用 shared_ptr/weak_ptr，要想在 C++ 里做到线程安全的对象回调与析构，可能的办法有以下一些。

1. 用一个全局的 façade 来代理 Foo 类型对象访问，所有的 Foo 对象回调和析构都通过这个 façade 来做，也就是把指针替换为 objId/handle，每次要调用对象的成员函数的时候先 check-out，用完之后再 check-in [16]。这样理论上能避免 race condition，但是代价很大。因为要想把这个 façade 做成线程安全的，那么必然要用互斥锁。这样一来，从两个线程访问两个不同的 Foo 对象也会用到同一个锁，让本来能够并行执行的函数变成了串行执行，没能发挥多核的优势。当然，可以像 Java 的 ConcurrentHashMap 那样用多个 buckets，每个 bucket 分别加锁，以降低 contention。
2. §1.4 提到的“只创建不销毁”手法，实属无奈之举。
3. 自己编写引用计数的智能指针 [17]。本质上是重新发明轮子，把 shared_ptr 实现一遍。正确实现线程安全的引用计数智能指针不是一件容易的事情，而高效的实现就更加困难。既然 shared_ptr 已经提供了完整的解决方案，那么似乎没有理由抗拒它。
4. 将来在 C++11 里有 unique_ptr，能避免引用计数的开销，或许能在某些场合替换 shared_ptr。

其他语言怎么办

有垃圾回收就好办。Google 的 Go 语言教程明确指出，没有垃圾回收的并发编程是困难的（Concurrency is hard without garbage collection）。但是由于指针算术的存在，在 C/C++ 里实现全自动垃圾回收更加困难。而那些天生具备垃圾回收的语言在并发编程方面具有明显的优势，Java 是目前支持并发编程最好的主流语言，它的 util.concurrent 库和内存模型是 C++11 效仿的对象。

1.13 心得与小结

学习多线程程序设计远远不是看看教程了解 API 怎么用那么简单，这最多“主要是为了读懂别人的代码，如果自己要写这类代码，必须专门花时间严肃、认真、系

[16] 这是 Jeff Grossman 在《A technique for safe deletion with object locking》一文中提出的办法 [Gr00]。

[17] 见 http://blog.csdn.net/solstice/article/details/5238671#comments 后面的评论。

统地学习，严禁半桶水上阵”（孟岩）[18]。一般的多线程教程上都会提到要让加锁的区域足够小，这没错，问题是如何找出这样的区域并加锁，本章 §1.9 举的安全读写 shared_ptr 可算是一个例子。

据我所知，目前 C++ 没有特别好的多线程领域专著，但 C 语言有，Java 语言也有。《Java Concurrency in Practice》[JCP] 是我读过的写得最好的书，内容足够新，可读性和可操作性俱佳。C++ 程序员反过来要向 Java 学习，多少有些讽刺。除了编程书，操作系统教材也是必读的，至少要完整地学习一本经典教材的相关章节，可从《操作系统设计与实现》、《现代操作系统》、《操作系统概念》任选一本，了解各种同步原语、临界区、竞态条件、死锁、典型的 IPC 问题等等，防止闭门造车。

分析可能出现的 race condition 不仅是多线程编程的基本功，也是设计分布式系统的基本功，需要反复历练，形成一定的思考范式，并积累一些经验教训，才能少犯错误。这是一个快速发展的领域，要不断吸收新知识，才不会落伍。单 CPU 时代的多线程编程经验到了多 CPU 时代不一定有效，因为多 CPU 能做到真正的并行执行，每个 CPU 看到的事件发生顺序不一定完全相同。正如狭义相对论所说的每个观察者都有自己的时钟，在不违反因果律的前提下，可能发生十分违反直觉的事情。

尽管本章通篇在讲如何安全地使用（包括析构）跨线程的对象，但我建议尽量减少使用跨线程的对象，我赞同水木网友 ilovecpp 说的：“用流水线，生产者消费者，任务队列这些有规律的机制，最低限度地共享数据。这是我所知最好的多线程编程的建议了。”

不用跨线程的对象，自然不会遇到本章描述的各种险态。如果迫不得已要用，希望本章内容能对你有帮助。

小结

- 原始指针暴露给多个线程往往会造成 race condition 或额外的簿记负担。
- 统一用 shared_ptr/scoped_ptr 来管理对象的生命期，在多线程中尤其重要。

[18] 孟岩《快速掌握一个语言最常用的 50%》博客，这篇博客（http://blog.csdn.net/myan/article/details/3144661）的其他文字也很有趣味：“粗粗看看语法，就撸起袖子开干，边查 Google 边学习”这种路子也有问题，在对于这种语言的脾气秉性还没有了解的情况下大刀阔斧地拼凑代码，写出来的东西肯定不入流。说穿新鞋走老路，新瓶装旧酒，那都是小问题，真正严重的是这样的程序员可以在短时间内堆积大量充满缺陷的垃圾代码。由于通常开发阶段的测试完备程度有限，这些垃圾代码往往能通过这个阶段，从而潜伏下来，在后期成为整个项目的“毒瘤”，反反复复让后来的维护者陷入西西弗斯困境。……其实真正写程序不怕完全不会，最怕一知半解地去攒解决方案。因为你完全不会，就自然会去认真查书学习，如果学习能力好的话，写出来的代码质量不会差。而一知半解，自己动手“土法炼钢”，那搞出来的基本上都是“废铜烂铁”。

- shared_ptr 是值语意，当心意外延长对象的生命期。例如 boost::bind 和容器都可能拷贝 shared_ptr。
- weak_ptr 是 shared_ptr 的好搭档，可以用作弱回调、对象池等。
- 认真阅读一遍 boost::shared_ptr 的文档，能学到很多东西：
  http://www.boost.org/doc/libs/release/libs/smart_ptr/shared_ptr.htm
- 保持开放心态，留意更好的解决办法，比如 C++11 引入的 unique_ptr。忘掉已被废弃的 auto_ptr。

shared_ptr 是 TR1 的一部分，即 C++ 标准库的一部分，值得花一点时间去学习掌握 [19]，对编写现代的 C++ 程序有莫大的帮助。我个人的经验是，一周左右就能基本掌握各种用法与常见陷阱，比学 STL 还快。网络上有一些对 shared_ptr 的批评，那可以算作故意误用的例子，就好比故意访问失效的迭代器来证明 std::vector 不安全一样。

正确使用标准库（含 shared_ptr）作为自动化的内存/资源管理器，解放大脑，从此告别内存错误。

1.14 Observer 之谬

本章 §1.8 把 shared_ptr/weak_ptr 应用到 Observer 模式中，部分解决了其线程安全问题。我用 Observer 举例，因为这是一个广为人知的设计模式，但是它有本质的问题。

Observer 模式的本质问题在于其面向对象的设计。换句话说，我认为正是面向对象（OO）本身造成了 Observer 的缺点。Observer 是基类，这带来了非常强的耦合，强度仅次于友元（friend）。这种耦合不仅限制了成员函数的名字、参数、返回值，还限制了成员函数所属的类型（必须是 Observer 的派生类）。

Observer class 是基类，这意味着如果 Foo 想要观察两个类型的事件（比如时钟和温度），需要使用多继承。这还不是最糟糕的，如果要重复观察同一类型的事件（比如 1 秒一次的心跳和 30 秒一次的自检），就要用到一些伎俩来 work around，因为不能从一个 Base class 继承两次。

[19] 孟岩在《垃圾收集机制批判》中说：在 C++ 中，new 出来的对象没有 delete，这就导致了 memory leak。但是 C++ 早就有了克服这一问题的办法——smart pointer。通过使用标准库里设计精致的各种 STL 容器，还有例如 Boost 库（差不多是个准标准库了）中的 4 个 smart pointers，C++ 程序员只要花上一个星期的时间学习最新的资料，就可以拍着胸脯说：“我写的程序没有 memory leak!”。

现在的语言一般可以绕过 Observer 模式的限制，比如 Java 可以用匿名内部类，Java 8 用 Closure，C# 用 delegate，C++ 用 `boost::function/ boost::bind` [20] 。

在 C++ 里为了替换 Observer，可以用 Signal/Slots，我指的不是 QT 那种靠语言扩展的实现，而是完全靠标准库实现的 thread safe、race condition free、thread contention free 的 Signal/Slots，并且不强制要求 `shared_ptr` 来管理对象，也就是说完全解决了 §1.8 列出的 Observer 遗留问题。这会用到 §2.8 介绍的“借 `shared_ptr` 实现 copy-on-write”技术。

在 C++11 中，借助 variadic template，实现最简单（trivial）的一对多回调可谓不费吹灰之力，代码如下。

recipes/thread/SignalSlotTrivial.h

```
template<typename Signature>
class SignalTrivial;

// NOT thread safe !!!
template <typename RET, typename... ARGS>
class SignalTrivial<RET(ARGS...)>
{
 public:
  typedef std::function<void (ARGS...)> Functor;

  void connect(Functor&& func)
  {
    functors_.push_back(std::forward<Functor>(func));
  }

  void call(ARGS&&... args)
  {
    for (const Functor& f: functors_)
    {
      f(args...);
    }
  }

 private:
  std::vector<Functor> functors_;
};
```

recipes/thread/SignalSlotTrivial.h

我们不难把以上基本实现扩展为线程安全的 Signal/Slots，并且在 `Slot` 析构时自动 unregister。有兴趣的读者可仔细阅读完整实现的代码（recipes/thread/SignalSlot.h）。

[20] 见 §11.5“以 `boost::function` 和 `boost::bind` 取代虚函数”，还有孟岩的《function/bind 的救赎（上）》（http://blog.csdn.net/myan/article/details/5928531）。

结语

《C++ 沉思录》（*Ruminations on C++* 中文版）的附录是王曦和孟岩对作者夫妇二人的采访，在被问到“请给我们三个你们认为最重要的建议”时，Koenig 和 Moo 的第一个建议是“避免使用指针”。我 2003 年读到这段时，理解不深，觉得固然使用指针容易造成内存方面的问题，但是完全不用也是做不到的，毕竟 C++ 的多态要通过指针或引用来起效。6 年之后重新拾起来，发现大师的观点何其深刻，不免掩卷长叹。

这本书详细地介绍了 handle/body idiom，这是编写大型 C++ 程序的必备技术，也是实现物理隔离的“法宝”，值得细读。

目前来看，用 `shared_ptr` 来管理资源在国内 C++ 界似乎并不是一种主流做法，很多人排斥智能指针，视其为“洪水猛兽”（这或许受了 `auto_ptr` 的垃圾设计的影响）。据我所知，很多 C++ 项目还是手动管理内存和资源，因此我觉得有必要把我认为好的做法分享出来，让更多的人尝试并采纳。我觉得 `shared_ptr` 对于编写线程安全的 C++ 程序是至关重要的，不然就得“土法炼钢”，自己“重新发明轮子[21]”。这让我想起了 2001 年前后 STL 刚刚传入国内，大家也是很犹豫，觉得它性能不高，使用不便，还不如自己造的容器类。10 年过去了，现在 STL 已经是主流，大家也适应了迭代器、容器、算法、适配器、仿函数这些“新”名词、“新”技术，开始在项目中普遍使用（至少用 `vector` 代替数组嘛）。我希望，几年之后人们回头看本章内容，觉得“怎么讲的都是常识”，那我的写作目的也就达到了。

[21] http://en.wikipedia.org/wiki/Reinventing_the_wheel