7.2 无锁数据结构的例子

为了演示一些在设计无锁数据结构中所使用到的技术，我们将看到一些无锁实现的简单数据结构。这里不仅要在每个例子中描述一个有用的数据结构实现，还将使用这些例子的某些特别之处来阐述对于无锁数据结构的设计。

如之前所提到的，无锁结构依赖与原子操作和内存序及相关保证，以确保多线程以正确的顺序访问数据结构。最初，所有原子操作默认使用的是memory_order_seq_cst内存序；因为简单，所以使用(所有memory_order_seq_cst都遵循一种顺序)。不过，在后面的例子中，我们将会降低内存序的要求，使用memory_order_acquire, memory_order_release, 甚至memory_order_relaxed。虽然这个例子中没有直接的使用锁，但需要注意的是对 `std::atomic_flag` 的使用。一些平台上的无锁结构实现(实际上在C++的标准库的实现中)，使用了内部锁(详见第5章)。另一些平台上，基于锁的简单数据结构可能会更适合;当然，还有很多平台不能一一说明；在选择一种实现前，需要明确需求，并且配置各种选项以满足要求。

那么，回到数据结构上来吧，最简单的数据结构——栈。

7.2.1 写一个无锁的线程安全栈

栈的要求很简单：查询顺序是添加顺序的逆序——先入后出(LIFO)。所以，要确保一个值安全的添加入栈就十分重要，因为很可能在添加后，马上被其他线程索引，同时确保只有一个线程能索引到给定值也是很重要。最简单的栈就是链表，head指针指向第一个节点(可能是下一个被索引到的节点)，并且每个节点依次指向下一个节点。

在这样的情况下，添加一个节点相对来说很简单：

1. 创建一个新节点。
2. 将当新节点的next指针指向当前的head节点。
3. 让head节点指向新节点。

在单线程的上下文中，这种方式没有问题，不过当多线程对栈进行修改时，这几步就不够用了。至关重要的是，当有两个线程同时添加节点的时候，在第2步和第3步的时候会产生条件竞争：一个线程可能在修改head的值时，另一个线程正在执行第2步，并且在第3步中对head进行更新。这就会使之前那个线程的工作被丢弃，亦或是造成更加糟糕的后果。在了解如何解决这个条件竞争

之前，还要注意一个很重要的事：当head更新，并指向了新节点时，另一个线程就能读取到这个节点了。因此，在head设置为指向新节点前，让新节点完全准备就绪就变得很重要了；因为，在这之后就不能对节点进行修改了。

OK，那如何应对讨厌的条件竞争呢？答案就是：在第3步的时候使用一个原子“比较/交换”操作，来保证当步骤2对head进行读取时，不会对head进行修改。当有修改时，可以循环“比较/交换”操作。下面的代码就展示了，不用锁来实现线程安全的push()函数。

清单7.2 不用锁实现push()

```
template<typename T>
class lock_free_stack
{
private:
  struct node
  {
    T data;
    node* next;

    node(T const& data_):  // 1
     data(data_)
    {}
  };

  std::atomic<node*> head;
public:
  void push(T const& data)
  {
    node* const new_node=new node(data); // 2
    new_node->next=head.load();  // 3
    while(!head.compare_exchange_weak(new_node->next,new_node));  // 4
  }
};
```

上面代码近乎能匹配之前所说的三个步骤：创建一个新节点②，设置新节点的next指针指向当前head③，并设置head指针指向新节点④。node结构用其自身的构造函数来进行数据填充①，必须保证节点在构造完成后随时能被弹出。之后需要使用compare_exchange_weak()来保证在被存储到new_node->next的head指针和之前的一样③。代码的亮点是使用“比较/交换”操作：当其返回false时，因为比较失败(例如，head被其他线程锁修改)，会使用head中的内容更新new_node->next(第一个参数)的内容。循环中不需要每次都重新加载head指针，因为编译器会帮你完成这件事。同样，因为循环可能直接就失败了，所以这里使用compare_exchange_weak要好于使用compare_exchange_strong(详见第5章)。

所以，这里暂时不需要pop()操作，可以先快速检查一下push()的实现是否有违指导意见。这里唯一一个能抛出异常的地方就构造新node的时候①，不过其会自行处理，且链表中的内容没有被修改，所以这里是安全的。因为在构建数据的时候，是将其作为node的一部分作为存储的，并且使用compare_exchange_weak()来更新head指针，所以这里没有恶性的条件竞争。“比较/交换”成功时，节点已经准备就绪，且随时可以提取。因为这里没有锁，所以就不存在死锁的情况，这里的push()函数实现的很成功。

那么，你现在已经有往栈中添加数据的方法了，现在需要删除数据的方法。其步骤如下，也很简单：

1. 读取当前head指针的值。
2. 读取head->next。
3. 设置head到head->next。
4. 通过索引node，返回data数据。
5. 删除索引节点。

但在多线程环境下，就不像看起来那么简单了。当有两个线程要从栈中移除数据，两个线程可能在步骤1中读取到同一个head(值相同)。当其中一个线程处理到步骤5，而另一个线程还在处理步骤2时，这个还在处理步骤2的线程将会解引用一个悬空指针。这只是写无锁代码所遇到的最大问题之一，所以现在只能跳过步骤5，让节点泄露。

另一个问题就是：当两个线程读取到同一个head值，他们将返回同一个节点。这就违反了栈结构的意图，所以你需要避免这样的问题产生。你可以像在push()函数中解决条件竞争那样来解决这个问题：使用“比较/交换”操作更新head。当“比较/交换”操作失败时，不是一个新节点已被推入，就是其他线程已经弹出了想要弹出的节点。无论是那种情况，都得返回步骤1(“比较/交换”操作将会重新读取head)。

当“比较/交换”成功，就可以确定当前线程是弹出给定节点的唯一线程，之后就可以放心的执行步骤4了。这里先看一下pop()的雏形：

```
template<typename T>
class lock_free_stack
{
public:
  void pop(T& result)
  {
    node* old_head=head.load();
    while(!head.compare_exchange_weak(old_head,old_head->next));
    result=old_head->data;
  }
};
```

虽然这段代码很优雅，但这里还有两个节点泄露的问题。首先，这段代码在空链表的时候不工作：当head指针式一个空指针时，当要访问next指针时，将引起未定义行为。这很容易通过对nullptr的检查进行修复(在while循环中)，要不对空栈抛出一个异常，要不返回一个bool值来表明成功与否。

第二个问题就是异常安全问题。当在第3章中介绍栈结构时，了解了在返回值的时候会出现异常安全问题：当有异常被抛出时，复制的值将丢失。在这种情况下，传入引用是一种可以接受的解决方案；因为这样就能保证，当有异常抛出时，栈上的数据不会丢失。不幸的是，不能这样做；只能在单一线程对值进行返回的时候，才进行拷贝，以确保拷贝操作的安全性，这就意味着在拷贝结束后这个节点就被删除了。因此，通过引用获取返回值的方式就没有任何优势：直接返回也是可以的。若想要安全的返回值，你必须使用第3章中的其他方法：返回指向数据值的(智能)指针。

当返回的是智能指针时，就能返回nullptr以表明没有值可返回，但是要求在堆上对智能指针进行内存分配。将分配过程做为pop()的一部分时(也没有更好的选择了)，堆分配时可能会抛出一个异常。与此相反，可以在push()操作中对内存进行分配——无论怎样，都得对node进行内存分配。返回一个 `std::shared_ptr<>` 不会抛出异常，所以在pop()中进行分配就是安全的。将上面的观点放在一起，就能看到如下的代码。

清单7.3 带有节点泄露的无锁栈

```
template<typename T>
class lock_free_stack
{
private:
  struct node
  {
    std::shared_ptr<T> data;  // 1 指针获取数据
    node* next;

    node(T const& data_):
      data(std::make_shared<T>(data_))  // 2 让std::shared_ptr指向新分配出来的T
    {}
  };

  std::atomic<node*> head;
public:
  void push(T const& data)
  {
    node* const new_node=new node(data);
    new_node->next=head.load();
    while(!head.compare_exchange_weak(new_node->next,new_node));
  }
```

```
  std::shared_ptr<T> pop()
  {
    node* old_head=head.load();
    while(old_head && // 3 在解引用前检查old_head是否为空指针
      !head.compare_exchange_weak(old_head,old_head->next));
    return old_head ? old_head->data : std::shared_ptr<T>();  // 4
  }
};
```

智能指针指向当前数据①，这里必须在堆上为数据分配内存(在node结构体中)②。而后，在compare_exchage_weak()循环中③，需要在old_head指针前，检查指针是否为空。最终，如果存在相关节点，那么将会返回相关节点的值；当不存在时，将返回一个空指针④。注意，结构是无锁的，但并不是无等待的，因为在push()和pop()函数中都有while循环，当compare_exchange_weak()总是失败的时候，循环将会无限循环下去。

7.2.2 停止内存泄露：使用无锁数据结构管理内存

第一次了解pop()时，为了避免条件竞争(当有线程删除一个节点的同时，其他线程还持有指向该节点的指针，并且要解引用)选择了带有内存泄露的节点。但是，不论什么样的 c++ 程序，存在内存泄露都不可接受。所以，现在来解决这个问题！

基本问题在于，当要释放一个节点时，需要确认其他线程没有持有这个节点。当只有一个线程调用pop()，就可以放心的进行释放。当节点添加入栈后，push()就不会与节点有任何的关系了，所以只有调用pop()函数的线程与已加入节点有关，并且能够安全的将节点删除。

另一方面，当栈同时处理多线程对pop()的调用时，就需要知道节点在什么时候被删除。这实际上就需要你写一个节点专用的垃圾收集器。这听起来有些可怖，同时也相当棘手，不过并不是多么糟糕：这里需要检查节点，并且检查哪些节点被pop()访问。不需要对push()中的节点有所担心，因为这些节点推到栈上以后，才能被访问到，而多线程只能通过pop()访问同一节点。

当没有线程调用pop()时，这时可以删除栈上的任意节点。因此，当添加节点到“可删除”列表中时，就能从中提取数据了。而后，当没有线程通过pop()访问节点时，就可以安全的删除这些节点了。那怎么知道没有线程调用pop()了呢？很简单——计数即可。当计数器数值增加时，就是有节点推入；当减少时，就是有节点被删除。这样从“可删除”列表中删除节点就很安全了，直到计数器的值为0为止。当然，这个计数器必须是原子的，这样它才能在多线程的情况下正确的进行计数。下面的清单中，展示了修改后的pop()函数，有些支持功能的实现将在清单7.5中给出。

清单7.4 没有线程通过pop()访问节点时，就对节点进行回收

```
template<typename T>
class lock_free_stack
{
private:
  std::atomic<unsigned> threads_in_pop;  // 1 原子变量
  void try_reclaim(node* old_head);
public:
  std::shared_ptr<T> pop()
  {
    ++threads_in_pop;  // 2 在做事之前，计数值加1
    node* old_head=head.load();
    while(old_head &&
      !head.compare_exchange_weak(old_head,old_head->next));
    std::shared_ptr<T> res;
    if(old_head)
    {
      res.swap(old_head->data);  // 3 回收删除的节点
    }
    try_reclaim(old_head);  // 4 从节点中直接提取数据，而非拷贝指针
    return res;
  }
};
```

threads_in_pop①原子变量用来记录有多少线程试图弹出栈中的元素。当pop()②函数调用的时候，计数器加一；当调用try_reclaim()时，计数器减一，当这个函数被节点调用时，说明这个节点已经被删除④。因为暂时不需要将节点删除，可以通过swap()函数来删除节点上的数据③(而非只是拷贝指针)，当不再需要这些数据的时候，这些数据会自动删除，而不是持续存在着(因为这里还有对未删除节点的引用)。接下来看一下try_reclaim()是如何实现的。

清单7.5 采用引用计数的回收机制

```
template<typename T>
class lock_free_stack
{
private:
  std::atomic<node*> to_be_deleted;

  static void delete_nodes(node* nodes)
  {
    while(nodes)
    {
      node* next=nodes->next;
      delete nodes;
```

```
      nodes=next;
    }
  }
  void try_reclaim(node* old_head)
  {
    if(threads_in_pop==1)  // 1
    {
      node* nodes_to_delete=to_be_deleted.exchange(nullptr);  // 2 声明“可删除”列
      if(!--threads_in_pop)  // 3 是否只有一个线程调用pop()?
      {
        delete_nodes(nodes_to_delete);  // 4
      }
      else if(nodes_to_delete)  // 5
      {
         chain_pending_nodes(nodes_to_delete);  // 6
      }
      delete old_head;  // 7
    }
    else
    {
      chain_pending_node(old_head);  // 8
      --threads_in_pop;
    }
  }
  void chain_pending_nodes(node* nodes)
  {
    node* last=nodes;
    while(node* const next=last->next)  // 9 让next指针指向链表的末尾
    {
      last=next;
    }
    chain_pending_nodes(nodes,last);
  }

  void chain_pending_nodes(node* first,node* last)
  {
    last->next=to_be_deleted;  // 10
    while(!to_be_deleted.compare_exchange_weak(  // 11 用循环来保证last->next的正
      last->next,first));
    }
    void chain_pending_node(node* n)
    {
      chain_pending_nodes(n,n);  // 12
    }
};
```

回收节点时①，threads_in_pop的数值是1，也就是当前线程正在对pop()进行访问，这时就可以安全的将节点进行删除了⑦(将等待节点删除也是安全的)。当数值不是1时，删除任何节点都不安全，所以需要向等待列表中继续添加节点⑧。

假设在某一时刻，threads_in_pop的值为1。那就可以尝试回收等待列表，如果不回收，节点就会继续等待，直到整个栈被销毁。要做到回收，首先要通过一个原子exchange操作声明②删除列表，并将计数器减一③。如果之后计数的值为0，就意味着没有其他线程访问等待节点链表。出现新的等待节点时，不必为其烦恼，因为它们将被安全的回收。而后，可以使用delete_nodes对链表进行迭代，并将其删除④。

当计数值在减后不为0，回收节点就不安全；所以如果存在⑤，就需要将其挂在等待删除链表之后⑥，这种情况会发生在多个线程同时访问数据结构的时候。一些线程在第一次测试threads_in_pop①和对“回收”链表的声明②操作间调用pop()，这可能新填入一个已经被线程访问的节点到链表中。在图7.1中，线程C添加节点Y到to_be_deleted链表中，即使线程B仍将其引用作为old_head，之后会尝试访问其next指针。在线程A删除节点的时候，会造成线程B发生未定义的行为。

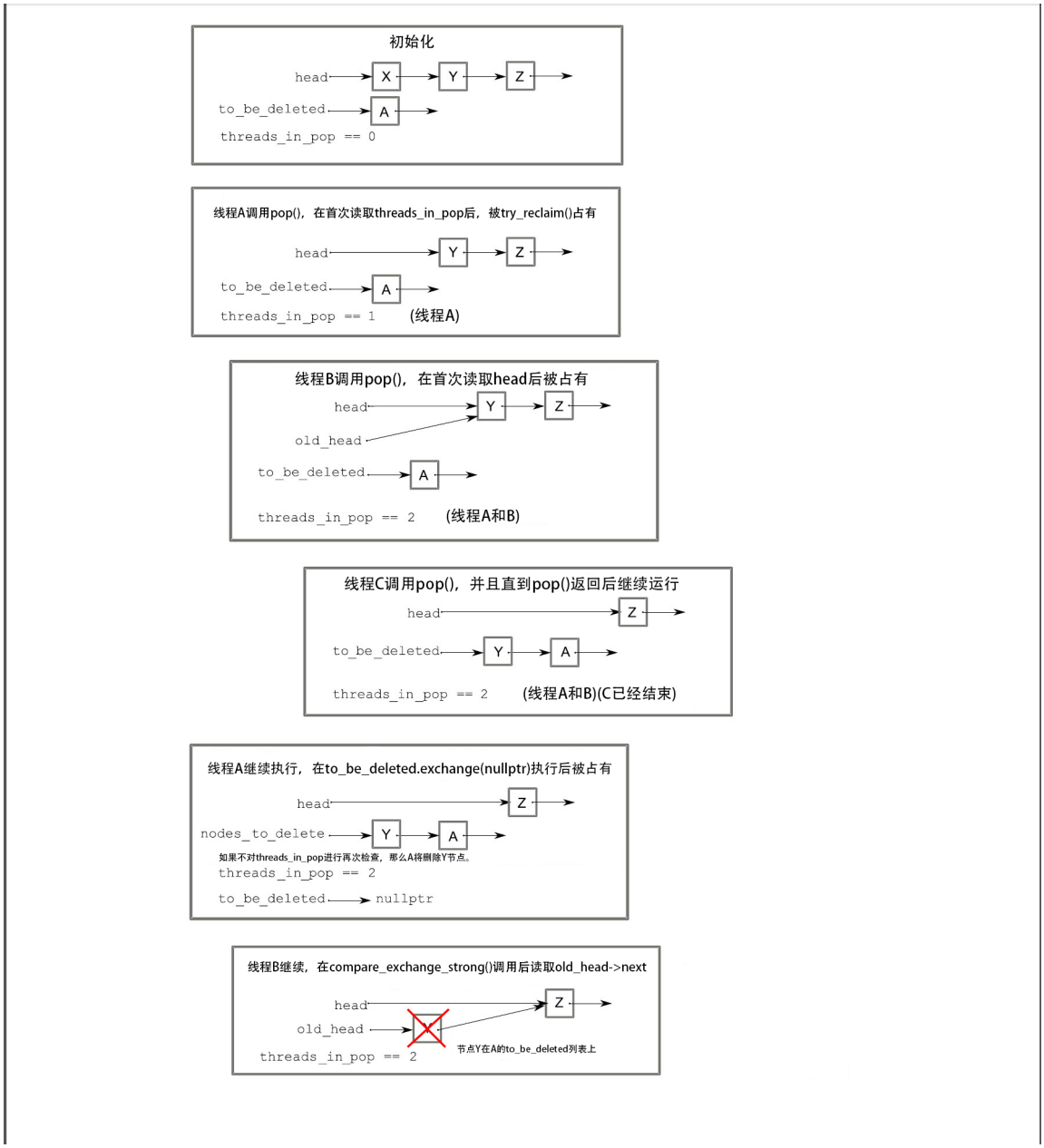

图7.1 三个线程同时调用pop()，说明为什么要在try_reclaim()对声明节点进行删除前，对threads_in_pop进行检查。

为了将等待删除的节点添加入等待删除链表，需要复用节点的next指针将等待删除节点链接在一起。在这种情况下，将已存在的链表链接到删除链表后面，通过遍历的方式找到链表的末尾⑨，将最后一个节点的next指针替换为当前to_be_deleted指针⑩，并且将链表中的第一个节点作为新的to_be_deleted指针进行存储⑪。这里需要在循环中使用compare_exchange_weak来保证，通过其他线程添加进来的节点不会发生内存泄露。这样，在链表发生改变时，更新next指针很方便。添加单个节点是一种特殊情况，因为这需要将这个节点作为第一个节点，同时也是最后一个节点进行添加⑫。

在低负荷的情况下，这种方式没有问题，因为在没有线程访问pop()，有一个合适的静态指针。不过，这只是一个瞬时的状态，也就是为什么在回收前，需要检查threads_in_pop计数为0③的原因；同样也是删除节点⑦前进行对计数器检查的原因。删除节点是一项耗时的工作，并且希望其

他线程能对链表做的修改越小越好。从第一次发现threads_in_pop是1，到尝试删除节点，会用很长的时间，这样就会让线程有机会调用pop()，会让threads_in_pop不为0，阻止节点的删除操作。

在高负荷的情况，不会存在静态；因为，其他线程在初始化之后，都能进入pop()。在这样的情况下，to_ne_deleted链表将会无界的增加，并且会再次泄露。当这里不存在任何静态的情况时，就得为回收节点寻找替代机制。关键是要确定没有线程访问一个给定节点，那么这个节点就能被回收。现在，最简单的替换机制就是使用*风险指针*(hazard pointer)。

7.2.3 检测使用风险指针(不可回收)的节点

“风险指针”这个术语引用于Maged Michael的技术发现[1]。之所以这样叫，是因为删除一个节点可能会让其他引用其的线程处于危险之中。当其他线程持有这个删除的节点的指针，并且解引用进行操作的时候，将会出现未定义行为。这里的基本观点就是，当有线程去访问要被(其他线程)删除的对象时，会先设置对这个对象设置一个风险指针，而后通知其他线程，删除这个指针是一个危险的行为。一旦这个对象不再被需要，那么就可以清除风险指针了。如果了解牛津/剑桥的龙舟比赛，那么这里使用到的机制和龙舟比赛开赛时差不多：每个船上的舵手都举起手来，以表示他们还没有准备好。只要有舵手的手是举着的，那么裁判就不能让比赛开始。当所有舵手的手都放下后，比赛才能开始；在比赛还未开始或感觉自己船队的情况有变时，舵手可以再次举手。

当线程想要删除一个对象，那么它就必须检查系统中其他线程是否持有风险指针。当没有风险指针的时候，那么它就可以安全删除对象。否则，它就必须等待风险指针的消失了。这样，线程就得周期性的检查其想要删除的对象是否能安全删除。

看起来很简单，在 c++ 中应该怎么做呢？

首先，需要一个地点能存储指向访问对象的指针，这个地点就是风险指针。这个地点必须能让所有线程看到，需要其中一些线程可以对数据结构进行访问。如何正确和高效的分配这些线程，的确是一个挑战，所以这个问题可以放在后面解决，而后假设你有一个get_hazard_pointer_for_current_thread()的函数，这个函数可以返回风险指针的引用。当你读取一个指针，并且想要解引用它的时候，你就需要这个函数——在这种情况下head数值源于下面的列表：

```
std::shared_ptr<T> pop()
{
  std::atomic<void*>& hp=get_hazard_pointer_for_current_thread();
  node* old_head=head.load();  // 1
  node* temp;
  do
```

```
  {
    temp=old_head;
    hp.store(old_head);  // 2
    old_head=head.load();
  } while(old_head!=temp); // 3
  // ...
}
```

在while循环中就能保证node不会在读取旧head指针①时，以及在设置风险指针的时被删除了。这种模式下，其他线程不知道有线程对这个给定的节点进行了访问。幸运的是，当旧head节点要被删除时，head本身是要改变的，所以需要对head进行检查，并持续循环，直到head指针中的值与风险指针中的值相同③。使用风险指针，如同依赖对已删除对象的引用。当使用默认的new和delete操作对风险指针进行操作时，会出现未定义行为，所以需要确定实现是否支持这样的操作，或使用自定义分配器来保证这种用法的正确性。

现在已经设置了风险指针，那就可以对pop()进行处理了，基于现在了解到的安全知识，这里不会有其他线程来删除节点。啊哈！这里每一次重新加载old_head时，解引用刚刚读取到的指针时，就需要更新风险指针。当从链表中提取一个节点时，就可以将风险指针清除了。如果没有其他风险指针引用节点，就可以安全的删除节点了；否则，就需要将其添加到链表中，之后再将其删除。下面的代码就是对该方案的完整实现。

清单7.6 使用风险指针实现的pop()

```
std::shared_ptr<T> pop()
{
  std::atomic<void*>& hp=get_hazard_pointer_for_current_thread();
  node* old_head=head.load();
  do
  {
    node* temp;
    do  // 1 直到将风险指针设为head指针
    {
      temp=old_head;
      hp.store(old_head);
      old_head=head.load();
    } while(old_head!=temp);
  }
  while(old_head &&
    !head.compare_exchange_strong(old_head,old_head->next));
  hp.store(nullptr);  // 2 当声明完成，清除风险指针
  std::shared_ptr<T> res;
  if(old_head)
```

```
  {
    res.swap(old_head->data);
    if(outstanding_hazard_pointers_for(old_head))  // 3 在删除之前对风险指针引用的
    {
      reclaim_later(old_head);  // 4
    }
    else
    {
      delete old_head;  // 5
    }
    delete_nodes_with_no_hazards();  // 6
  }
  return res;
}
```

首先，循环内部会对风险指针进行设置，在当“比较/交换”操作失败会重载old_head，再次进行设置①。使用compare_exchange_strong()，是因为需要在循环内部做一些实际的工作：当compare_exchange_weak()伪失败后，风险指针将被重置(没有必要)。这个过程能保证风险指针在解引用(old_head)之前，能被正确的设置。当已声明了一个风险指针，那么就可以将其清除了②。如果想要获取一个节点，就需要检查其他线程上的风险指针，检查是否有其他指针引用该节点③。如果有，就不能删除节点，只能将其放在链表中，之后再进行回收④；如果没有，就能直接将这个节点删除了⑤。最后，如果需要对任意节点进行检查，可以调用reclaim_later()。如果链表上没有任何风险指针引用节点，就可以安全的删除这些节点⑥。当有节点持有风险指针，就只能让下一个调用pop()的线程离开。

当然，这些函数——get_hazard_pointer_for_current_thread(), reclaim_later(), outstanding_hazard_pointers_for(), 和delete_nodes_with_no_hazards()——的实现细节我们还没有看到，先来看看它们是如何工作的。

为线程分配风险指针实例的具体方案：使用get_hazard_pointer_for_current_thread()与程序逻辑的关系并不大(不过会影响效率，接下会看到具体的情况)。可以使用一个简单的结构体：固定长度的“线程ID-指针”数组。get_hazard_pointer_for_curent_thread()就可以通过这个数据来找到第一个释放槽，并将当前线程的ID放入到这个槽中。当线程退出时，槽就再次置空，可以通过默认构造 `std::thread::id()` 将线程ID放入槽中。这个实现就如下所示：

清单7.7 get_hazard_pointer_for_current_thread()函数的简单实现

```
unsigned const max_hazard_pointers=100;
struct hazard_pointer
{
```

```
  std::atomic<std::thread::id> id;
  std::atomic<void*> pointer;
};
hazard_pointer hazard_pointers[max_hazard_pointers];

class hp_owner
{
  hazard_pointer* hp;

public:
  hp_owner(hp_owner const&)=delete;
  hp_owner operator=(hp_owner const&)=delete;
  hp_owner():
    hp(nullptr)
  {
    for(unsigned i=0;i<max_hazard_pointers;++i)
    {
      std::thread::id old_id;
      if(hazard_pointers[i].id.compare_exchange_strong(  // 6 尝试声明风险指针的所
         old_id,std::this_thread::get_id()))
      {
        hp=&hazard_pointers[i];
        break;  // 7
      }
    }
    if(!hp)  // 1
    {
      throw std::runtime_error("No hazard pointers available");
    }
  }

  std::atomic<void*>& get_pointer()
  {
    return hp->pointer;
  }

  ~hp_owner()  // 2
  {
    hp->pointer.store(nullptr);  // 8
    hp->id.store(std::thread::id());  // 9
  }
};

std::atomic<void*>& get_hazard_pointer_for_current_thread()  // 3
{
```

```
49    thread_local static hp_owner hazard;  // 4 每个线程都有自己的风险指针
50    return hazard.get_pointer();  // 5
51  }
```

get_hazard_pointer_for_current_thread()的实现看起来很简单③：一个hp_owner④类型的thread_local(本线程所有)变量，用来存储当前线程的风险指针，可以返回这个变量所持有的指针⑤。之后的工作：第一次有线程调用这个函数时，新hp_owner实例就被创建。这个实例的构造函数⑥，会通过查询“所有者/指针”表，寻找没有所有者的记录。其用compare_exchange_strong()来检查某个记录是否有所有者，并进行析构②。当compare_exchange_strong()失败，其他线程的拥有这个记录，所以可以继续执行下去。当交换成功，当前线程就拥有了这条记录，而后对其进行存储，并停止搜索⑦。当遍历了列表也没有找到物所有权的记录①，就说明有很多线程在使用风险指针，所以这里将抛出一个异常。

一旦hp_owner实例被一个给定的线程所创建，那么之后的访问将会很快，因为指针在缓存中，所以表不需要再次遍历。

当线程退出时，hp_owner的实例将会被销毁。析构函数会在 `std::thread::id()` 设置拥有者ID前，将指针重置为nullptr,这样就允许其他线程对这条记录进行复用⑧⑨。

实现get_hazard_pointer_for_current_thread()后，outstanding_hazard_pointer_for()实现就简单了：只需要对风险指针表进行搜索，就可以找到对应记录。

```
bool outstanding_hazard_pointers_for(void* p)
{
  for(unsigned i=0;i<max_hazard_pointers;++i)
  {
    if(hazard_pointers[i].pointer.load()==p)
    {
      return true;
    }
  }
  return false;
}
```

实现都不需要对记录的所有者进行验证：没有所有者的记录会是一个空指针，所以比较代码将总返回false，通过这种方式将代码简化。

reclaim_later()和delete_nodes_with_no_hazards()可以对简单的链表进行操作；reclaim_later()只是将节点添加到列表中，delete_nodes_with_no_hazards()就是搜索整个列表，并将无风险指针的记录进行删除。下面将展示它们的具体实现。

清单7.8 回收函数的简单实现

```
template<typename T>
void do_delete(void* p)
{
  delete static_cast<T*>(p);
}

struct data_to_reclaim
{
  void* data;
  std::function<void(void*)> deleter;
  data_to_reclaim* next;

  template<typename T>
  data_to_reclaim(T* p):  // 1
    data(p),
    deleter(&do_delete<T>),
    next(0)
  {}

  ~data_to_reclaim()
  {
    deleter(data);  // 2
  }
};

std::atomic<data_to_reclaim*> nodes_to_reclaim;

void add_to_reclaim_list(data_to_reclaim* node)  // 3
{
  node->next=nodes_to_reclaim.load();
  while(!nodes_to_reclaim.compare_exchange_weak(node->next,node));
}

template<typename T>
void reclaim_later(T* data)  // 4
{
  add_to_reclaim_list(new data_to_reclaim(data));  // 5
}

void delete_nodes_with_no_hazards()
{
  data_to_reclaim* current=nodes_to_reclaim.exchange(nullptr);  // 6
```

```
43    while(current)
44    {
45      data_to_reclaim* const next=current->next;
46      if(!outstanding_hazard_pointers_for(current->data))  // 7
47      {
48        delete current;  // 8
49      }
50      else
51      {
52        add_to_reclaim_list(current);  // 9
53      }
54      current=next;
55    }
56  }
```

首先，reclaim_later()是一个函数模板④。因为风险指针是一个通用解决方案，所以这里就不能将栈节点的类型写死。使用 `std::atomic<void*>` 对风险指针进行存储。需要对任意类型的指针进行处理，不过不能使用 `void*` 形式，因为当要删除数据项时，delete操作只能对实际类型指针进行操作。data_to_reclaim的构造函数处理的就很优雅：reclaim_later()只是为指针创建一个data_to_reclaim的实例，并且将实例添加到回收链表中⑤。add_to_reclaim_list()③就是使用compare_exchange_weak()循环来访问链表头(就如你之前看到的那样)。

当将节点添加入链表时，data_to_reclaim的析构函数不会被调用；析构函数会在没有风险指针指向节点的时候调用，这也就是delete_nodes_with_no_hazards()的作用。

delete_nodes_with_no_hazards()将已声明的链表节点进行回收，使用的是exchange()函数⑥(这个步骤简单且关键，是为了保证只有一个线程回收这些节点)。这样，其他线程就能自由将节点添加到链表中，或在不影响回收指定节点线程的情况下，对节点进行回收。

只要有节点存在于链表中，就需要检查每个节点，查看节点是否被风险指针所指向⑦。如果没有风险指针，那么就可以安全的将记录删除(并且清除存储的数据)⑧。否则，就只能将这个节点添加到链表的后面，再进行回收⑨。

虽然这个实现很简单，也的确安全的回收了被删除的节点，不过这个过程增加了很多开销。遍历风险指针数组需要检查max_hazard_pointers原子变量，并且每次pop()调用时，都需要再检查一遍。原子操作很耗时——在台式CPU上，100次原子操作要比100次非原子操作慢——所以，这里pop()成为了性能瓶颈。这种方式，不仅需要遍历节点的风险指针链表，还要遍历等待链表上的每一个节点。显然，这种方式很糟糕。当有max_hazard_pointers在链表中，那么就需要检查max_hazard_pointers多个已存储的风险指针。我去！还有更好一点的方法吗？

对风险指针(较好)的回收策略

当然有更好的办法。这里只展示一个风险指针的简单实现，来帮助解释技术问题。首先，要考虑的是内存性能。比起对回收链表上的每个节点进行检查都要调用pop()，除非有超过max_hazard_pointer数量的节点存在于链表之上，要不就不需要尝试回收任何节点。这样就能保证至少有一个节点能够回收，如果只是等待链表中的节点数量达到max_hazard_pointers+1，那比之前的方案也没好到哪里去。当获取了max_hazard_pointers数量的节点时，可以调用pop()对节点进行回收，所以这样也不是很好。不过，当有2max_hazard_pointers个节点在列表中时，就能保证至少有max_hazard_pointers可以被回收，在再次尝试回收任意节点前，至少会对pop()有max_hazard_pointers次调用。这就很不错了。比起检查max_hazard_pointers个节点就调用max_hazard_pointers次pop()(而且还不一定能回收节点)，当检查2max_hazard_pointers个节点时，每max_hazard_pointers次对pop()的调用，就会有max_hazard_pointers个节点能被回收。这就意味着，对两个节点检查调用pop()，其中就有一个节点能被回收。

这个方法有个缺点(有增加内存使用的情况)：就是得对回收链表上的节点进行计数，这就意味着要使用原子变量，并且还有很多线程争相对回收链表进行访问。如果还有多余的内存，可以增加内存的使用来实现更好的回收策略：每个线程中的都拥有其自己的回收链表，作为线程的本地变量。这样就不需要原子变量进行计数了。这样的话，就需要分配max_hazard_pointers x max_hazard_pointers个节点。所有节点被回收完毕前时，有线程退出，那么其本地链表可以像之前一样保存在全局中，并且添加到下一个线程的回收链表中，让下一个线程对这些节点进行回收。

风险指针另一个缺点：与IBM申请的专利所冲突[2]。要让写的软件在一个国家中使用，那么就必须拥有合法的知识产权，所以需要拥有合适的许可证。这对所有无锁的内存回收技术都适用(这是一个活跃的研究领域)，所以很多大公司都会有自己的专利。你可能会问，“为什么用了这么大的篇幅来介绍一个大多数人都没办法的技术呢？”，这公平性的问题。首先，使用这种技术可能不需要买一个许可证。比如，当你使用GPL下的免费软件许可来进行软件开发，那么你的软件将会包含到IBM不主张声明中。其次，也是很重要的，在设计无锁代码的时候，还需要从使用的技术角度进行思考，比如，高消耗的原子操作。

所以，是否有非专利的内存回收技术，且能被大多数人所使用呢？很幸运，的确有。引用计数就是这样一种机制。

7.2.4 检测使用引用计数的节点

回到7.2.2节的问题，“想要删除节点还能被其他读者线程访问，应该怎么办?"。当能安全并精确的识别，节点是否还被引用，以及没有线程访问这些节点的具体时间，以便将对应节点进行删除。风险指针是通过将使用中的节点存放到链表中，解决问题。而引用计数是通过对每个节点上访问的线程数量进行统计，解决问题。

看起来简单粗暴……不，优雅；实际上管理起来却是很困难：首先，你会想到的就是由 `std::shared_ptr<>` 来完成这个任务，其是有内置引用计数的指针。不幸的是，虽然 `std::shared_ptr<>` 上的一些操作是原子的，不过其也不能保证是无锁的。智能指针上的原子操作和对其他原子类型的操作并没有什么不同，但是 `std::shared_ptr<>` 旨在用于有多个上下文的情况下，并且在无锁结构中使用原子操作，无异于对该类增加了很多性能开销。如果平台支持 `std::atomic_is_lock_free(&some_shared_ptr)` 实现返回true，那么所有内存回收问题就都迎刃而解了。使用 `std::shared_ptr<node>` 构成的链表实现，如下所示：

清单7.9 无锁栈——使用无锁 `std::shared_ptr<>` 的实现

```
template<typename T>
class lock_free_stack
{
private:
  struct node
  {
    std::shared_ptr<T> data;
    std::shared_ptr<node> next;
    node(T const& data_):
      data(std::make_shared<T>(data_))
    {}
  };

  std::shared_ptr<node> head;
public:
  void push(T const& data)
  {
    std::shared_ptr<node> const new_node=std::make_shared<node>(data);
    new_node->next=head.load();
    while(!std::atomic_compare_exchange_weak(&head,
        &new_node->next,new_node));
  }
  std::shared_ptr<T> pop()
  {
    std::shared_ptr<node> old_head=std::atomic_load(&head);
    while(old_head && !std::atomic_compare_exchange_weak(&head,
        &old_head,old_head->next));
    return old_head ? old_head->data : std::shared_ptr<T>();
  }
};
```

在一些情况下，使用 `std::shared_ptr<>` 实现的结构并非无锁，这就需要手动管理引用计数。

一种方式是对每个节点使用两个引用计数：内部计数和外部计数。两个值的总和就是对这个节点的引用数。外部计数记录有多少指针指向节点，即在指针每次进行读取的时候，外部计数加一。当线程结束对节点的访问时，内部计数减一。指针在读取时，外部计数加一；在读取结束时，内部计数减一。

当不需要“外部计数-指针”对时(该节点就不能被多线程所访问了)，在外部计数减一和在被弃用的时候，内部计数将会增加。当内部计数等于0，那么就没有指针对该节点进行引用，就可以将该节点安全的删除。使用原子操作来更新共享数据也很重要。现在，就让我们来看一下使用这种技术实现的无锁栈，只有确定节点能安全删除的情况下，才能进行节点回收。

下面程序清单中就展示了内部数据结构，以及对push()简单优雅的实现。

清单7.10 使用分离引用计数的方式推送一个节点到无锁栈中

```
template<typename T>
class lock_free_stack
{
private:
  struct node;

  struct counted_node_ptr  // 1
  {
    int external_count;
    node* ptr;
  };

  struct node
  {
    std::shared_ptr<T> data;
    std::atomic<int> internal_count;  // 2
    counted_node_ptr next;  // 3

    node(T const& data_):
      data(std::make_shared<T>(data_)),
      internal_count(0)
    {}
  };

  std::atomic<counted_node_ptr> head;  // 4

public:
  ~lock_free_stack()
  {
```

```
    while(pop());
  }

  void push(T const& data)  // 5
  {
    counted_node_ptr new_node;
    new_node.ptr=new node(data);
    new_node.external_count=1;
    new_node.ptr->next=head.load();
    while(!head.compare_exchange_weak(new_node.ptr->next,new_node));
  }
};
```

外部计数包含在counted_node_ptr的指针中①，且这个结构体会被node中的next指针③和内部计数②用到。counted_node_ptr是一个简单的结构体，所以可以使用特化 `std::atomic<>` 模板来对链表的头指针进行声明④。

且counted_node_ptr体积够小，能够让 `std::atomic<counted_node_ptr>` 无锁。在一些平台上支持双字比较和交换操作，可以直接对结构体进行操作。当你的平台不支持这样的操作时，最好使用 `std::shared_ptr<>` 变量(如清单7.9那样)；当类型的体积过大，超出了平台支持指令，那么原子 `std::atomic<>` 将使用锁来保证其操作的原子性(从而会让你的“无锁”算法“基于锁”来完成)。另外，如果想要限制计数器的大小，需要已知平台上指针所占的空间(比如，地址空间只剩下48位，而一个指针就要占64位)，可以将计数存在一个指针空间内，不过为了适应平台，也可以存在一个机器字当中。这样的技巧需要对特定系统有足够的了解，当然已经超出本书讨论的范围。

push()相对简单⑤，可以构造一个counted_node_ptr实例，去引用新分配出来的(带有相关数据的)node，并且将node中的next指针设置为当前head。之后使用compare_exchange_weak()对head的值进行设置，就像之前代码清单中所示。因为internal_count刚被设置，所以其值为0，并且external_count是1。因为这是一个新节点，那么这个节点只有一个外部引用(head指针)。

通常，pop()都有一个从繁到简的过程，实现代码如下。

清单7.11 使用分离引用计数从无锁栈中弹出一个节点

```
template<typename T>
class lock_free_stack
{
private:
  void increase_head_count(counted_node_ptr& old_counter)
  {
    counted_node_ptr new_counter;
```

```
8
9       do
10      {
11        new_counter=old_counter;
12        ++new_counter.external_count;
13      }
14      while(!head.compare_exchange_strong(old_counter,new_counter));  // 1
15
16      old_counter.external_count=new_counter.external_count;
17    }
18  public:
19    std::shared_ptr<T> pop()
20    {
21      counted_node_ptr old_head=head.load();
22      for(;;)
23      {
24        increase_head_count(old_head);
25        node* const ptr=old_head.ptr;  // 2
26        if(!ptr)
27        {
28          return std::shared_ptr<T>();
29        }
30        if(head.compare_exchange_strong(old_head,ptr->next))  // 3
31        {
32          std::shared_ptr<T> res;
33          res.swap(ptr->data);  // 4
34
35          int const count_increase=old_head.external_count-2;  // 5
36
37          if(ptr->internal_count.fetch_add(count_increase)==  // 6
38             -count_increase)
39          {
40            delete ptr;
41          }
42
43          return res;  // 7
44        }
45        else if(ptr->internal_count.fetch_sub(1)==1)
46        {
47          delete ptr;  // 8
48        }
49      }
50    }
51  };
```

当加载head的值之后，就必须将外部引用加一，是为了表明这个节点正在引用，并且保证在解引用时的安全性。在引用计数增加前解引用指针，那么就会有线程能够访问这个节点，从而当前引用指针就成为了一个悬空指针。这就是将引用计数分离的主要原因：通过增加外部引用计数，保证指针在访问期间的合法性。在compare_exchange_strong()的循环中①完成增加，通过比较和设置整个结构体来保证指针不会在同一时间内被其他线程修改。

当计数增加，就能安全的解引用ptr，并读取head指针的值，就能访问指向的节点②。如果指针是空指针，那么将会访问到链表的最后。当指针不为空时，就能尝试对head调用compare_exchange_strong()来删除这个节点③。

当compare_exchange_strong()成功时，就拥有对应节点的所有权，并且可以和data进行交换④，然后返回。这样数据就不会持续保存，因为其他线程也会对栈进行访问，所以会有其他指针指向这个节点。而后，可以使用原子操作fetch_add⑥，将外部计数加到内部计数中去。如果现在引用计数为0，那么之前的值(fetch_add返回的值)，在相加之前肯定是一个负数，这种情况下就可以将节点删除。这里需要注意的是，相加的值要比外部引用计数少2⑤;当节点已经从链表中删除，就要减少一次计数，并且这个线程无法再次访问指定节点，所以还要再减一。无论节点是否被删除，都能完成操作，所以可以将获取的数据进行返回⑦。

当“比较/交换”③失败，就说明其他线程在之前把对应节点删除了，或者其他线程添加了一个新的节点到栈中。无论是哪种原因，需要通过“比较/交换”的调用，对具有新值的head重新进行操作。不过，首先需要减少节点(要删除的节点)上的引用计数。这个线程将再也没有办法访问这个节点了。如果当前线程是最后一个持有引用(因为其他线程已经将这个节点从栈上删除了)的线程，那么内部引用计数将会为1，所以减一的操作将会让计数器为0。这样，你就能在循环⑧进行之前将对应节点删除了。

目前，使用默认 `std::memory_order_seq_cst` 内存序来规定原子操作的执行顺序。在大多数系统中，这种操作方式都很耗时，且同步操作的开销要高于内存序。现在，就可以考虑对数据结构的逻辑进行修改，对数据结构的部分放宽内存序要求；就没有必要在栈上增加过度的开销了。现在让我们来检查一下栈的操作，并且扪心自问，这里能对一些操作使用更加宽松的内存序么？如果使用了，能确保同级安全吗？

7.2.5 应用于无锁栈上的内存模型

在修改内存序之前，需要检查一下操作之间的依赖关系。而后，再去确定适合这种需求关系的最小内存序。为了保证这种方式能够工作，需要在从线程的视角进行观察。其中最简单的视角就是，向栈中推入一个数据项，之后让其他线程从栈中弹出这个数据。

即使在简单的例子中，都需要三个重要的数据参与。1、counted_node_ptr转移的数据head。2、head引用的node。3、节点所指向的数据项。

做push()的线程，会先构造数据项和节点，再设置head。做pop()的线程，会先加载head的值，再做在循环中对head做“比较/交换”操作，并增加引用计数，再读取对应的node节点，获取next的指向的值，现在就可以看到一组需求关系。next的值是普通的非原子对象，所以为了保证读取安全，这里必须确定存储(推送线程)和加载(弹出线程)的先行关系。因为唯一的原子操作就是push()函数中的compare_exchange_weak()，这里需要释放操作来获取两个线程间的先行关系，这里compare_exchange_weak()必须是 `std::memory_order_release` 或更严格的内存序。当compare_exchange_weak()调用失败，什么都不会改变，并且可以持续循环下去，所以使用 `std::memory_order_relaxed` 就足够了。

```
void push(T const& data)
{
  counted_node_ptr new_node;
  new_node.ptr=new node(data);
  new_node.external_count=1;
  new_node.ptr->next=head.load(std::memory_order_relaxed)
  while(!head.compare_exchange_weak(new_node.ptr->next,new_node,
    std::memory_order_release,std::memory_order_relaxed));
}
```

那pop()的实现呢？为了确定先行关系，必须在访问next值之前使用 `std::memory_order_acquire` 或更严格内存序的操作。因为，在increase_head_count()中使用compare_exchange_strong()就获取next指针指向的旧值，所以想要其获取成功就需要确定内存序。如同调用push()那样，当交换失败，循环会继续，所以在失败的时候使用松散的内存序：

```
void increase_head_count(counted_node_ptr& old_counter)
{
  counted_node_ptr new_counter;

  do
  {
    new_counter=old_counter;
    ++new_counter.external_count;
  }
  while(!head.compare_exchange_strong(old_counter,new_counter,
        std::memory_order_acquire,std::memory_order_relaxed));

  old_counter.external_count=new_counter.external_count;
}
```

当compare_exchange_strong()调用成功，那么ptr中的值就被存到old_counter中。存储操作是push()中的一个释放操作，并且compare_exchange_strong()操作是一个获取操作，现在存储同步于加载，并且能够获取先行关系。因此，在push()中存储ptr的值，要先行于在pop()中对ptr->next的访问。现在的操作就安全了。

要注意的是，内存序对head.load()的初始化并不妨碍分析，所以现在就可以使用 `std::memory_order_relaxed` 。

接下来，compare_exchange_strong()将old_head.ptr->next设置为head。是否需要做什么来保证操作线程中的数据完整性呢？当交换成功，你就能访问ptr->data，所以这里需要保证在push()线程中已经对ptr->data进行了存储(在加载之前)。在increase_head_count()中的获取操作，能保证与push()线程中的存储和“比较/交换”同步。这里的先行关系是：在push()线程中存储数据，先行于存储head指针；调用increase_head_count()先行于对ptr->data的加载。即使，pop()中的“比较/交换”操作使用 `std::memory_order_relaxed` ，这些操作还是能正常运行。唯一不同的地方就是，调用swap()让ptr->data有所变化，且没有其他线程可以对同一节点进行操作(这就是“比较/交换”操作的作用)。

当compare_exchange_strong()失败，那么新值就不会去更新old_head，继续循环。这里，已确定在increase_head_count()中使用 `std::memory_order_acquire` 内存序的可行性，所以这里使用 `std::memory_order_relaxed` 也可以。

其他线程呢？是否需要设置一些更为严格的内存序来保证其他线程的安全呢？回答是“不用”。因为，head只会因“比较/交换”操作有所改变；对于“读-改-写”操作来说，push()中的“比较/交换”操作是构成释放序列的一部分。因此，即使有很多线程在同一时间对head进行修改，push()中的compare_exchange_weak()与increase_head_count()(读取已存储的值)中的compare_exchange_strong()也是同步的。

剩余操作就可以用来处理fetch_add()操作(用来改变引用计数的操作)，因为已知其他线程不可能对该节点的数据进行修改，所以从节点中返回数据的线程可以继续执行。不过，当线程获取其他线程修改后的值时，就代表操作失败(swap()是用来提取数据项的引用)。那么，为了避免数据竞争，需要保证swap()先行于delete操作。一种简单的解决办法，在“成功返回”分支中对fetch_add()使用 `std::memory_order_release` 内存序，在“再次循环”分支中对fetch_add()使用 `std::memory_order_qcquire` 内存序。不过，这就有点矫枉过正：只有一个线程做delete操作(将引用计数设置为0的线程)，所以只有这个线程需要获取操作。幸运的是，因为fetch_add()是一个“读-改-写”操作，是释放序列的一部分，所以可以使用一个额外的load()做获取。当“再次循环”分支将引用计数减为0时，fetch_add()可以重载引用计数，这里使用 `std::memory_order_acquire` 为了保持需求的同步关系；并且，fetch_add()本身可以使用 `std::memory_order_relaxed` 。使用新pop()的栈实现如下。

清单7.12 基于引用计数和松散原子操作的无锁栈

```
template<typename T>
class lock_free_stack
{
private:
  struct node;
  struct counted_node_ptr
  {
    int external_count;
    node* ptr;
  };

  struct node
  {
    std::shared_ptr<T> data;
    std::atomic<int> internal_count;
    counted_node_ptr next;

    node(T const& data_):
      data(std::make_shared<T>(data_)),
      internal_count(0)
    {}
  };

  std::atomic<counted_node_ptr> head;

  void increase_head_count(counted_node_ptr& old_counter)
  {
    counted_node_ptr new_counter;

    do
    {
      new_counter=old_counter;
      ++new_counter.external_count;
    }
    while(!head.compare_exchange_strong(old_counter,new_counter,
                                        std::memory_order_acquire,
                                        std::memory_order_relaxed));
    old_counter.external_count=new_counter.external_count;
  }
public:
  ~lock_free_stack()
  {
    while(pop());
  }

```

```
46  void push(T const& data)
47  {
48    counted_node_ptr new_node;
49    new_node.ptr=new node(data);
50    new_node.external_count=1;
51    new_node.ptr->next=head.load(std::memory_order_relaxed)
52    while(!head.compare_exchange_weak(new_node.ptr->next,new_node,
53                                      std::memory_order_release,
54                                      std::memory_order_relaxed));
55  }
56  std::shared_ptr<T> pop()
57  {
58    counted_node_ptr old_head=
59       head.load(std::memory_order_relaxed);
60    for(;;)
61    {
62      increase_head_count(old_head);
63      node* const ptr=old_head.ptr;
64      if(!ptr)
65      {
66        return std::shared_ptr<T>();
67      }
68      if(head.compare_exchange_strong(old_head,ptr->next,
69                                      std::memory_order_relaxed))
70      {
71        std::shared_ptr<T> res;
72        res.swap(ptr->data);
73
74        int const count_increase=old_head.external_count-2;
75
76        if(ptr->internal_count.fetch_add(count_increase,
77              std::memory_order_release)==-count_increase)
78        {
79          delete ptr;
80        }
81
82        return res;
83      }
84      else if(ptr->internal_count.fetch_add(-1,
85                   std::memory_order_relaxed)==1)
86      {
87        ptr->internal_count.load(std::memory_order_acquire);
88        delete ptr;
89      }
90    }
```

```
  }
};
```

这是一种锻炼，不过锻炼要告一段落了，我们已经获得比之前好很多的栈实现。在深思熟虑后，通过使用更多的松散操作，在不影响并发性的同时提高性能。实现中的pop()有37行，而功能等同于清单6.1中的那7行的基于锁的栈实现，和清单7.2中无内存管理的无锁栈实现。对于接下来要设计的无锁队列，将看到类似的情况：无锁结构的复杂性，主要在于内存的管理。

7.2.6 写一个无锁的线程安全队列

队列的提供的挑战与栈的有些不同，因为push()和pop()在队列中，操作的不是同一个地方。因此，同步的需求就不一样了。需要保证对一端的修改是正确的，且对另一端是可见的。不过，在清单6.6中队列有一个try_pop()成员函数，其作用和清单7.2中简单的无锁栈的pop()功能差不多，那么就可以合理的假设无锁代码都很相似。这是为什么呢？

如果将清单6.6中的代码作为基础，就需要两个node指针：head和tail。可以让多线程对它们进行访问，所以这两个节点最好是原子的，这样就不用考虑互斥问题了。让我们对清单6.6中的代码做一些修改，并且看一下应该从哪里开始设计。先来看一下下面的代码。

清单7.13 单生产者/单消费者模型下的无锁队列

```
template<typename T>
class lock_free_queue
{
private:
  struct node
  {
    std::shared_ptr<T> data;
    node* next;

    node():
       next(nullptr)
    {}
  };

  std::atomic<node*> head;
  std::atomic<node*> tail;

  node* pop_head()
  {
```

```
    node* const old_head=head.load();
    if(old_head==tail.load())  // 1
    {
      return nullptr;
    }
    head.store(old_head->next);
    return old_head;
  }
public:
  lock_free_queue():
      head(new node),tail(head.load())
  {}

  lock_free_queue(const lock_free_queue& other)=delete;
  lock_free_queue& operator=(const lock_free_queue& other)=delete;

  ~lock_free_queue()
  {
    while(node* const old_head=head.load())
    {
      head.store(old_head->next);
      delete old_head;
    }
  }
  std::shared_ptr<T> pop()
  {
    node* old_head=pop_head();
    if(!old_head)
    {
      return std::shared_ptr<T>();
    }

    std::shared_ptr<T> const res(old_head->data);  // 2
    delete old_head;
    return res;
  }

  void push(T new_value)
  {
    std::shared_ptr<T> new_data(std::make_shared<T>(new_value));
    node* p=new node;  // 3
    node* const old_tail=tail.load();  // 4
    old_tail->data.swap(new_data);  // 5
    old_tail->next=p;  // 6
    tail.store(p);  // 7
```

```
65     }
66 };
```

一眼望去，这个实现并没什么不好，当只有一个线程调用一次push()，且只有一个线程调用pop()。在这种情况下，队列完美工作。push()和pop()之间的先行关系就很重要了，这直接关系到获取到的data。对tail的存储⑦同步于对tail的加载①；存储之前节点的data指针⑤先行于存储tail；并且，加载tail先行于加载data指针②，所以对data的存储要先行于加载，一切都没问题。因此，这是一个完美的*单生产者，单消费者*(SPSC, single-producer, single-consume)队列。

问题在于当多线程对push()或pop()并发调用。先看一下push()：如果有两个线程并发调用push()，那么它们会新分配两个节点作为虚拟节点③，也会读取到相同的tail值④，因此也会同时修改同一个节点，同时设置data和next指针⑤⑥。明显的数据竞争！

pop_head()函数也有类似的问题。当有两个线程并发的调用这个函数时，这两个线程就会读取到同一个head中同样的值，并且会同时通过next指针去复写旧值。两个线程现在都能索引到同一个节点——真是一场灾难！这里，不仅要保证只有一个pop()线程可以访问给定项，还要保证其他线程在读取head指针时，可以安全的访问节点中的next。这就和无锁栈中pop()的问题一样了，那么就有很多解决方案可以在这里使用。

pop()的问题解决了，那么push()呢？问题在于为了获取push()和pop()间的先行关系，就需要在为虚拟节点设置数据项前，更新tail指针。这就意味着，并发访问push()时，因为每个线程所读取到的是同一个tail指针，所以线程会为同一个数据项进行竞争。

多线程下的push()

第一个选择是在两个真实节点中添加一个虚拟节点。这种方法，需要当前tail节点更新next指针，这样让节点看起来像一个原子变量。当一个线程成功将next指针指向一个新节点，就说明其成功的添加了一个指针；否则，就不得不再次读取tail，并重新对指针进行添加。这里就需要对pop()进行简单的修改，为了消除持有空指针的节点再次进行循环。这个方法的缺点：每次pop()函数的调用，通常都要删除两个节点，每次添加一个节点，都需要分配双份内存。

第二个选择是让data指针原子化，并通过“比较/交换”操作对其进行设置。如果“比较/交换”成功，就说明你能获取tail指针，并能够安全的对其next指针进行设置，也就是更新tail。因为有其他线程对数据进行了存储，所以会导致“比较/交换”操作的失败，这时就要重新读取tail，重新循环。当原子操作对于 `std::shared_ptr<>` 是无锁的，那么就可以轻松一下了。如果不是，你就需要一个替代方案了；一种可能是让pop()函数返回一个 `std::unique_ptr<>` (毕竟，这个指针指针只能引用指定对象)，并且将数据作为一个普通指针存储在队列中的方案。这就需要队列支持存储 `std::atomic<T*>` 类型，对于compare_exchange_strong()的调用就很有必要了。当使用的是类似于清单7.11中的引用计数模式，来解决多线程对pop()和push()的访问。

清单7.14 push()的第一次修订(不正确的)

```
void push(T new_value)
{
  std::unique_ptr<T> new_data(new T(new_value));
  counted_node_ptr new_next;
  new_next.ptr=new node;
  new_next.external_count=1;
  for(;;)
  {
    node* const old_tail=tail.load();  // 1
    T* old_data=nullptr;
    if(old_tail->data.compare_exchange_strong(
      old_data,new_data.get()))  // 2
    {
      old_tail->next=new_next;
      tail.store(new_next.ptr);  // 3
      new_data.release();
      break;
    }
  }
}
```

使用引用计数方案可以避免竞争，不过竞争不只在push()中。可以再看一下7.14中的修订版push()，与栈中模式相同：加载一个原子指针①，并且对该指针解引用②。同时，另一个线程可以对指针进行更新③，最终回收该节点(在pop()中)。当节点回收后，再对指针进行解引用，就对导致未定义行为。啊哈！这里有个诱人的方案，就是给tail也添加计数器，就像给head做的那样，不过队列中的节点的next指针中都已经拥有了一个外部计数。在同一个节点上有两个外部计数，为了避免过早的删除节点，这就是对之前引用计数方案的修改。通过对node结构中外部计数器数量的统计，解决这个问题。当外部计数器销毁时，统计值减一(将对应的外部计数添加到内部)。当内部计数是0，且没有外部计数器时，对应节点就可以被安全删除了。这个技术，是我查阅Joe Seigh的*原子指针+*项目[5]的时候看到的。下面push()的实现就使用的就是这种方案。

清单7.15 使用带有引用计数tail，实现的无锁队列中的push()

```
template<typename T>
class lock_free_queue
{
private:
  struct node;
  struct counted_node_ptr
  {
```

```
    int external_count;
    node* ptr;
  };

  std::atomic<counted_node_ptr> head;
  std::atomic<counted_node_ptr> tail;  // 1

  struct node_counter
  {
    unsigned internal_count:30;
    unsigned external_counters:2;  // 2
  };

  struct node
  {
    std::atomic<T*> data;
    std::atomic<node_counter> count;  // 3
    counted_node_ptr next;

    node()
    {
      node_counter new_count;
      new_count.internal_count=0;
      new_count.external_counters=2;  // 4
      count.store(new_count);

      next.ptr=nullptr;
      next.external_count=0;
     }
  };
public:
  void push(T new_value)
  {
    std::unique_ptr<T> new_data(new T(new_value));
    counted_node_ptr new_next;
    new_next.ptr=new node;
    new_next.external_count=1;
    counted_node_ptr old_tail=tail.load();

    for(;;)
    {
      increase_external_count(tail,old_tail);  // 5

      T* old_data=nullptr;
      if(old_tail.ptr->data.compare_exchange_strong(  // 6
```

```
            old_data,new_data.get()))
      {
        old_tail.ptr->next=new_next;
        old_tail=tail.exchange(new_next);
        free_external_counter(old_tail);  // 7
        new_data.release();
        break;
      }
      old_tail.ptr->release_ref();
    }
  }
};
```

清单7.15中，tail和head一样都是atomic类型①，并且node结构体中用count成员变量替换了之前的internal_count③。count成员变量包括了internal_count和外部external_counters成员②。注意，这里你需要2bit的external_counters，因为最多就有两个计数器。因为使用了位域，所以就将internal_count指定为30bit的值，就能保证计数器的总体大小是32bit。内部计数值就有充足的空间来保证这个结构体能放在一个机器字中(包括32位和64位平台)。重要的是，为的就是避免条件竞争，将结构体作为一个单独的实体来更新。让结构体的大小保持在一个机器字内，对其的操作就如同原子操作一样，还可以在多个平台上使用。

node初始化时，internal_count设置为0，external_counter设置为2④，因为当新节点加入队列中时，都会被tail和上一个节点的next指针所指向。push()与清单7.14中的实现很相似，除了为了对tail中的值进行解引用，需要调用节点data成员变量的compare_exchange_strong()成员函数⑥保证值的正确性；在这之前还要调用increase_external_count()增加计数器的计数⑤，而后在对尾部的旧值调用free_external_counter()⑦。

push()处理完毕，再来看一下pop()。下面的实现，是将清单7.11中的引用计数pop()与7.13中队列弹出pop()混合的版本。

清单7.16 使用尾部引用计数，将节点从无锁队列中弹出

```
template<typename T>
class lock_free_queue
{
private:
  struct node
  {
    void release_ref();
  };
public:
  std::unique_ptr<T> pop()
```

```
  {
    counted_node_ptr old_head=head.load(std::memory_order_relaxed);  // 1
    for(;;)
    {
      increase_external_count(head,old_head);  // 2
      node* const ptr=old_head.ptr;
      if(ptr==tail.load().ptr)
      {
        ptr->release_ref();  // 3
        return std::unique_ptr<T>();
      }
      if(head.compare_exchange_strong(old_head,ptr->next))  // 4
      {
        T* const res=ptr->data.exchange(nullptr);
        free_external_counter(old_head);  // 5
        return std::unique_ptr<T>(res);
      }
      ptr->release_ref();  // 6
    }
  }
};
```

在进入循环，并将加载值的外部计数增加②之前，需要加载old_head值作为启动①。当head与tail节点相同的时候，就能对引用进行释放③，因为这时队列中已经没有数据，所以返回的是空指针。如果队列中还有数据，可以尝试使用compare_exchange_strong()来做声明④。与7.11中的栈一样，将外部计数和指针做为一个整体进行比较的；当外部计数或指针有所变化时，需要将引用释放后，再次进行循环⑥。当交换成功时，已声明的数据就归你所有，那么为已弹出节点释放外部计数后⑤，就能把对应的指针返回给调用函数了。当两个外部引用计数都被释放，且内部计数降为0时，节点就可以被删除。对应的引用计数函数将会在7.17,7.18和7.19中展示。

清单7.17 在无锁队列中释放一个节点引用

```
template<typename T>
class lock_free_queue
{
private:
  struct node
  {
    void release_ref()
    {
      node_counter old_counter=
        count.load(std::memory_order_relaxed);
      node_counter new_counter;
```

```
      do
      {
        new_counter=old_counter;
        --new_counter.internal_count;  // 1
      }
      while(!count.compare_exchange_strong(  // 2
            old_counter,new_counter,
            std::memory_order_acquire,std::memory_order_relaxed));
      if(!new_counter.internal_count &&
         !new_counter.external_counters)
      {
        delete this;  // 3
      }
    }
  };
};
```

node::release_ref()的实现，只是对7.11中lock_free_stack::pop()进行小幅度的修改得到。不过，7.11中的代码仅是处理单个外部计数的情况，所以想要修改internal_count①，只需要使用fetch_sub就能让count结构体自动更新。因此，需要一个“比较/交换”循环②。降低internal_count时，在内外部计数都为0时，就代表这是最后一次引用，之后就可以将这个节点删除③。

清单7.18 从无锁队列中获取一个节点的引用

```
template<typename T>
class lock_free_queue
{
private:
  static void increase_external_count(
    std::atomic<counted_node_ptr>& counter,
    counted_node_ptr& old_counter)
  {
    counted_node_ptr new_counter;
    do
    {
      new_counter=old_counter;
      ++new_counter.external_count;
    }
    while(!counter.compare_exchange_strong(
      old_counter,new_counter,
      std::memory_order_acquire,std::memory_order_relaxed));

    old_counter.external_count=new_counter.external_count;
```

```
  }
};
```

清单7.18展示的是另一方面。这次，并不是对引用的释放，会得到一个新引用，并增加外部计数的值。increase_external_count()和7.12中的increase_head_count()很相似，不同的是increase_external_count()这里作为静态成员函数，通过将外部计数器作为第一个参数传入函数，对其进行更新，而非只操作一个固定的计数器。

清单7.19 无锁队列中释放节点外部计数器

```
template<typename T>
class lock_free_queue
{
private:
  static void free_external_counter(counted_node_ptr &old_node_ptr)
  {
    node* const ptr=old_node_ptr.ptr;
    int const count_increase=old_node_ptr.external_count-2;

    node_counter old_counter=
      ptr->count.load(std::memory_order_relaxed);
    node_counter new_counter;
    do
    {
      new_counter=old_counter;
      --new_counter.external_counters;  // 1
      new_counter.internal_count+=count_increase;  // 2
    }
    while(!ptr->count.compare_exchange_strong(  // 3
           old_counter,new_counter,
           std::memory_order_acquire,std::memory_order_relaxed));

    if(!new_counter.internal_count &&
       !new_counter.external_counters)
    {
      delete ptr;  // 4
    }
  }
};
```

与increase_external_count()对应的是free_external_counter()。这里的代码和7.11中的lock_free_stack::pop()类似，不过做了一些修改用来处理external_counters计数。使用单个

compare_exchange_strong()对计数结构体中的两个计数器进行更新③，就像之前release_ref()降低internal_count一样。和7.11中一样，internal_count会进行更新②，并且external_counters将会减一①。当内外计数值都为0，就没有更多的节点可以被引用，所以节点就可以安全的删除④。这个操作需要作为独立的操作来完成(因此需要“比较/交换”循环)，来避免条件竞争。如果将两个计数器分开来更新，在两个线程的情况下，可能都会认为自己最后一个引用者，从而将节点删除，最后导致未定义行为。

虽然现在的队列工作正常，且无竞争，但是还是有一个性能问题。当一个线程对old_tail.ptr->data成功的完成compare_exchange_strong()(7.15中的⑥)，就可以执行push()操作；并且，能确定没有其他线程在同时执行push()操作。这里，让其他线程看到有新值的加入，要比只看到空指针的好，因此在compare_exchange_strong()调用失败的时候，线程就会继续循环。这就是忙等待，这种方式会消耗CPU的运算周期，且什么事情都没做。因此，忙等待这就是一个锁。push()的首次调用，是要在其他线程完成后，将阻塞去除后才能完成，所以这里的实现只是*半无锁*(no longer lock-free)结构。不仅如此，还有当线程被阻塞的时候，操作系统会给不同的线程以不同优先级，用于获取互斥锁。在当前情况下，不可能出现不同优先级的情况，所以阻塞线程将会浪费CPU的运算周期，直到第一个线程完成其操作。处理的技巧出自于“无锁技巧包”：等待线程可以帮助push()线程完成操作。

无锁队列中的线程间互助

为了恢复代码无锁的属性，就需要让等待线程，在push()线程没什么进展时，做一些事情，就是帮进展缓慢的线程完成其工作。

在这种情况下，可以知道线程应该去做什么：尾节点的next指针需要指向一个新的虚拟节点，且tail指针之后也要更新。因为虚拟节点都是一样的，所以是谁创建的都不重要。当将next指针放入一个原子节点中时，就可以使用compare_exchange_strong()来设置next指针。当next指针已经被设置，就可以使用compare_exchange_weak()循环对tail进行设置，能保证next指针始终引用的是同一个原始节点。如果引用的不是同一个原始节点，那么其他部分就已经更新，可以停止尝试再次循环。这个需求只需要对pop()进行微小的改动，其目的就是为了加载next指针；这个实现将在下面展示。

清单7.20 修改pop()用来帮助push()完成工作

```
template<typename T>
class lock_free_queue
{
private:
  struct node
  {
    std::atomic<T*> data;
    std::atomic<node_counter> count;
```

```
    std::atomic<counted_node_ptr> next;  // 1
  };
public:
  std::unique_ptr<T> pop()
  {
    counted_node_ptr old_head=head.load(std::memory_order_relaxed);
    for(;;)
    {
      increase_external_count(head,old_head);
      node* const ptr=old_head.ptr;
      if(ptr==tail.load().ptr)
      {
        return std::unique_ptr<T>();
      }
      counted_node_ptr next=ptr->next.load();  // 2
      if(head.compare_exchange_strong(old_head,next))
      {
        T* const res=ptr->data.exchange(nullptr);
        free_external_counter(old_head);
        return std::unique_ptr<T>(res);
      }
      ptr->release_ref();
    }
  }
};
```

如之前所说，改变很简单：next指针线程就是原子的①，所以load②也是原子的。在这个例子中，可以使用默认memory_order_seq_cst内存序，所以这里可以忽略对load()的显式调用，并且依赖于加载对象隐式转换成counted_node_ptr，不过这里的显式调用就可以用来提醒：哪里需要显式添加内存序。

以下代码对push()有更多的展示。

清单7.21 无锁队列中简单的帮助性push()的实现

```
template<typename T>
class lock_free_queue
{
private:
  void set_new_tail(counted_node_ptr &old_tail,  // 1
                    counted_node_ptr const &new_tail)
  {
    node* const current_tail_ptr=old_tail.ptr;
```

```
    while(!tail.compare_exchange_weak(old_tail,new_tail) &&  // 2
          old_tail.ptr==current_tail_ptr);
    if(old_tail.ptr==current_tail_ptr)  // 3
      free_external_counter(old_tail);  // 4
    else
      current_tail_ptr->release_ref();  // 5
  }
public:
  void push(T new_value)
  {
    std::unique_ptr<T> new_data(new T(new_value));
    counted_node_ptr new_next;
    new_next.ptr=new node;
    new_next.external_count=1;
    counted_node_ptr old_tail=tail.load();

    for(;;)
    {
      increase_external_count(tail,old_tail);

      T* old_data=nullptr;
      if(old_tail.ptr->data.compare_exchange_strong(  // 6
         old_data,new_data.get()))
      {
        counted_node_ptr old_next={0};
        if(!old_tail.ptr->next.compare_exchange_strong(  // 7
           old_next,new_next))
        {
          delete new_next.ptr;  // 8
          new_next=old_next;  // 9
        }
        set_new_tail(old_tail, new_next);
        new_data.release();
        break;
      }
      else  // 10
      {
        counted_node_ptr old_next={0};
        if(old_tail.ptr->next.compare_exchange_strong(  // 11
           old_next,new_next))
        {
          old_next=new_next;  // 12
          new_next.ptr=new node;  // 13
        }
        set_new_tail(old_tail, old_next);  // 14
```

```
54          }
55        }
56      }
57  };
```

与清单7.15中的原始push()相似，不过还是有些不同。当对data进行设置⑥，就需要对另一线程帮忙的情况进行处理，在else分支就是具体的帮助⑩。

对节点中的data指针进行设置⑥时，新版push()对next指针的更新使用的是compare_exchange_strong()⑦(这里使用compare_exchange_strong()来避免循环),当交换失败，就能知道另有线程对next指针进行设置，所以就可以删除一开始分配的那个新节点⑧。还需要获取next指向的值——其他线程对tail指针设置的值。

对tail指针的更新，实际在set_new_tail()中完成①。这里使用一个compare_exchange_weak()循环②来更新tail，如果其他线程尝试push()一个节点时，external_count部分将会改变。不过，当其他线程成功的修改了tail指针时，就不能对其值进行替换；否则，队列中的循环将会结束，这是一个相当糟糕的主意。因此，当“比较/交换”操作失败的时候，就需要保证ptr加载值要与tail指向的值相同。当新旧ptr相同时，循环退出③，这就代表对tail的设置已经完成，所以需要释放旧外部计数器④。当ptr值不一样时，那么另一线程可能已经将计数器释放了，所以这里只需要对该线程持有的单次引用进行释放即可⑤。

当线程调用push()时，未能在循环阶段对data指针进行设置，那么这个线程可以帮助成功的线程完成更新。首先，会尝试更新next指针，让其指向该线程分配出来的新节点⑪。当指针更新成功，就可以将这个新节点作为新的tail节点⑫，且需要分配另一个新节点，用来管理队列中新推送的数据项⑬。在再进入循环之前，可以通过调用set_new_tail来设置tail节点⑭。

读者可能已经意识到，比起大量的new和delete操作，这样的代码更加短小精悍，因为新节点实在push()中被分配，而在pop()中被销毁。因此，内存分配器的效率也需要考虑到；一个糟糕的分配器可能会让无锁容器的扩展特性消失的一干二净。选择和实现高效的分配器，已经超出了本书的范围，不过需要牢记的是：测试以及衡量分配器效率最好的办法，就是对使用前和使用后进行比较。为优化内存分配，包括每个线程有自己的分配器，以及使用回收列表对节点进行回收，而非将这些节点返回给分配器。

例子已经足够多了；那么，让我们从这些例子中提取出一些指导建议吧。

[1] “Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes,” Maged M.Michael, *in PODC '02: Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing* (2002), ISBN 1-58113-485-1.

[2] Maged M. Michael, U.S. Patent and Trademark Office application number 20040107227, “Method for efficient implementation of dynamic lock-free data structures with safe memory reclamation.”

[3] GNU General Public License http://www.gnu.org/licenses/gpl.html.

[4] IBM Statement of Non-Assertion of Named Patents Against OSS, http://www.ibm.com/ibm/licensing/patents/pledgedpatents.pdf.

[5] Atomic Ptr Plus Project, http://atomic-ptr-plus.sourceforge.net/.