C++ (/tags/#C++) 基础编程 (/tags/#%E5%9F%BA%E7%A1%80%E7%BC%96%E7%A8%8B)

多线程编程 (/tags/#%E5%A4%9A%E7%BA%BF%E7%A8%8B%E7%BC%96%E7%A8%8B)

C++ 并发编程笔记(三)

C++ 并发编程笔记(三)

Posted by 敬方 on July 6, 2019

> *2019-07-11 20:09:48*

第6章 基于锁的并发数据结构设计

6.1 并发设计的意义

同过合理设计互斥量，让多个线程可以并发的访问这个数据，线程可以对这个数据结构做相同或者不同的操作。

序列化(serialzation):线程轮流访问被保护的数据。这其实是对数据进行串行的访问,而非并发。

一般进行并发数据结构设计的思路都是：减少保护区域,减少序列化操作,就能提升并发访问的潜力。

6.1.1 数据结构并发设计的指导与建议(指南)

数据结构线程安全条件:

- 确保无线程能够看到,数据结构的“不变量”破坏时的状态。
- 小心那些会引起条件竞争的接口,提供完整操作的函数,而非操作步骤。
- 注意数据结构的行为是否会产生异常,从而确保“不变量”的状态稳定。
- 将死锁的概率降到最低。使用数据结构时,需要限制锁的范围,且避免嵌套锁的存在。

需要考虑的问题:

- 锁的范围中的操作,是否允许在锁外执行?
- 数据结构中不同的区域是否能被不同的互斥量所保护?
- 所有操作都需要同级互斥量保护吗?
- 能否对数据结构进行简单的修改,以增加并发访问的概率,且不影响操作语义?

6.2 基于锁的并发数据结构

基于锁的并发数据结构设计,核心在于 **保证程序安全的情况下，保证线程持有锁的时间最短。**

线程安全的 `stack` 和 `queue` 示例，第三、四章中，在此不做过多叙述

6.2.3 线程安全队列——使用细粒度锁和条件变量

首先先看一个单线程的队列:

```
template<typename T>
class queue
{
private:
    struct node
    {
        T data;
        std::unique_ptr<node> next;

        node(T data_);
        data_(std::move(data_))
        {}
    };
    //头部指针

    std::unique_ptr<node> head;
    //队尾部指针

    node* tail;
public:
    queue(){}
    queue(const queue& other)=delete;
    queue& operator=(const queue& other)=delete;
    std::shared_ptr<T> try_pop()
    {
        if(!head){
            return std::shared_ptr<T>();
        }
        std::shared_ptr<T> const res(std::make_shared<T>(std::move(head->data)));
        std::unique_ptr<node> const old_head=std::move(head);
        //将head指针指向下一个

        head=std::move(old_head->next);
        return res;
    }
//尾部插入

    void push(T new_value)
```

```
    {
        std::unique_ptr<node> p(new node(std::move(new_value)));
        node* const new_tail=p.get();

        if(tail)
        {
            tail->next=std::move(p);
        }else{
            head=std::move(p);
        }
        tail=new_tail;
    }
}
```

这里可以看到，单线程情况下，基本使用良好，但是对于多线程而言，在push和pop中没有对头尾指针添加保护锁，同时，为了防止在队列只有一个元素时的时候，head==tail；所以push和try_pop间接访问了这个头尾指针，因此需要对tail添加保护锁。 不过这里使用更简便的方法，减少锁的使用：预分配一个空节点，永远指向队列尾部，这样避免了头尾指针能够被间接访问。但是使用了一个间接层次的指针数据作为虚拟节点。

更改完成时候，在push操作中只用考虑尾部指针tail在pop函数中虽然可以访问tail但是tail只在最初阶段进行比较，更多需要考虑head。同时，添加虚拟节点意味着pop和push不能同时对同一个节点进行操作。

最终，除了操作的元素外需要上锁外，push只对tail上锁，try_pop,先对head上锁，一旦被改变之后就不再加锁。

最终结果：

线程安全队列–细粒度锁版

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

    };
    //头部节点

    std::unique_ptr<node> head;
    //尾部节点
    node* tail;
    //头部互斥保护

    std::mutex head_mutex;
    //尾部信号量

    std::mutex tail_mutex;
public:
    threadsafe_queue():head(new node),tail(head.get){}
    ~threadsafe_queue();
    threadsafe_queue(const threadsafe_queue& other)=delete;
    threadsafe_queue& operator=(const threadsafe_queue& other)=delete;


    node* get_tail()
    {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }
    std::unique_ptr<node> pop_head()
    {
        std::lock_guard<std::mutex> head_lock(head_mutex);
        //使用git_tail 保护尾部指针一次
```

```
        if(head.get()==get_tail()){
            return nullptr;
        }
        std::unique_ptr<node> old_head=std::move(head);
        head=std::move(old_head->next);
        return old_head;
    }
    void push(T new_value)
    {
        std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        const* const new_tail=p.get();
        //tail 加锁

        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data=new_value;
        tail->next=std::move(p);
        tail=new_tail;
    }
    std::shared_ptr<T> try_pop()
    {
        std::unique_ptr<node> old_head=pop_head();
        return old_head?old_head->data:std::shared_ptr<T>();
    }

};
```

在此基础之上添加，可上锁和等待的线程安全队列；但是由于wait_and_pop等操作会降低程序的性能。

```
template<typename T>
class threadsafe_queue
{
private:
    struct  node
    {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };
    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex  tail_mutex;
    node* tail;
    //环境信号变量

    std::condition_variable data_cond;
public:
    threadsafe_queue():head(new node),tail(head.get()){}
    threadsafe_queue(const threadsafe_queue& other)=delete;
    threadsafe_queue& operator=(const threadsafe_queue& other)=delete;
    node* get_tail()
    {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }
    std::unique_ptr<node> pop_head()
    {
        std::unique_ptr<node> old_head=std::move(head);
        head=std::move(old_head->next);
        return old_head;
    }
    //数据等待线程锁

    std::unique_ptr<std::mutex> wait_for_data()
    {
        std::unique_lock<std::mutex> head_lock(head_mutex);
        //等待环境唤醒

        data_cond.wait(head_lock,[&]{return head.get()!=get_tail();});
```

```
        //将锁的实例，返回给调用者

        return std::move(head_lock);
    }
    std::unique_ptr<node> wait_pop_head()
    {
        //添加数据等待线程锁

        std::unique_lock<std::mutex> head_lock(wait_for_data());

        return pop_head();
    }
    std::unique_ptr<node> wait_pop_head(T& value)
    {
        std::unique_lock<std::mutex> head_lock(wait_for_data());
        //获取头部数据

        value=std::move(*head->data);

        return pop_head();
    }

    void wait_and_pop(T& value)
    {
        std::unique_ptr<node> const old_head=wait_pop_head(value);
    }
    //试着拿出头部

    std::unique_ptr<node> try_pop_head()
    {
        std::lock_guard<std::mutex> head_lock(head_mutex);
        if(head.get()==get_tail())
        {
            return std::unique_ptr<node>();
        }
        return pop_head();
    }
    std::unique_ptr<node> try_pop_head(T& value)
    {
```

```
        std::lock_guard<std::mutex> head_lock(head_mutex);
        if(head.get()==get_tail())
        {
            return std::unique_ptr<node>();
        }
        value=std::move(*head->data);
        return pop_head();
    }

    std::shared_ptr<T>  try_pop();
    bool try_pop(T& value);
    std::shared_ptr<T> wait_and_pop();
    void wait_and_pop(T& value);
    void push(T  new_value);
    bool empty();
};

//推入新节点

template<T>
void threadsafe_queue<T>::push(T new_data)
{
    std::shared_ptr new_data(std::make_shared<T>(std::move(new_value)));
    std::unique_ptr<node> p(new node);
    {
        //尾部加锁

        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data=new_data;
        node* new_tail=p.get();
        tail->next=std::move(p);
        tail=new_tail;
    }
    //发射环境信号

    data_cond.notify_one();
}

//线程安全队列
```

```
template<T>
std::shared_ptr<T> threadsafe_queue<T>::wait_and_pop()
{
    std::unique_ptr<node> const old_head=wait_pop_head();
    return old_head->data;
}
template<T>
void threadsafe_queue<T>::wait_and_pop(T& value)
{
    //传递值，然后返回取出的头部

    std::unique_ptr<node> const old_head=wait_pop_head(value);
}
template<T>
std::shared_ptr threadsafe_queue<T>::try_pop()
{
    std::unique_ptr<node> old_head=try_pop_head();
    return old_head?old_head->data:std::shared_ptr<T>();
}

template<T>
bool threadsafe_queue<T>::try_pop(T& value)
{
    std::unique_ptr<node> old_head=try_pop_head(value);
    return old_head;
}

template<T>
bool empty()
{
    std::lock_guard<std::mutex> head_lock(head_mutex);
    return (head.get()==get_tail());
}
```

6.3 基于锁设计更加复杂的数据结构

这里主要以定义一个简单的线程安全查询表和链表为例，进行工作

6.3.1 一个线程安全的查询表

首先明确查询表的基本操作有：

- 添加一队“键值-数据”
- 修改指定键值所对应的数据
- 删除一组值
- 通过给定键值，获取对应数据

`std::map` 椎间盘美好常见的关联容器和比较

- 二叉树；比如：红黑树：并不会提高对高并发的访问，每一个都要访问根节点，根节点需要时常上锁
- 有序数组：是最坏的选择，无法提前感知那个有序
- 哈希表：结合桶，对每个桶进行互斥加锁，提高并发性能。

```
//定义模板：关键字、值、hash映射

template<typename Key,typename Value,typename Hash=std::hash<Key> >



class threadsafe_lookup_table
{
private:
    //定义桶的基本类型

    class bucket_type
    {
    private:
        //设置键值对基本类型

        typedef std::pair<Key,Value>    bucket_value;
        //设置键值队列表

        typedef std::list<bucket_value> bucket_data;
        //定义列表迭代器

        typedef typename bucket_data::iterator bucket_iterator;
        //定义桶中的数据列表

        bucket_data data;
        //桶的互斥信号变量

        mutable boost::shared_mutex mutex;
        //通过关键字查找迭代器

        bucket_iterator find_entry_for(Key const& key) const
        {
            return  std::find_if(data.begin(),
                data.end(),
                [&](bucket_value const& item){return item.first==key;}
                );
        }
    public:
```

```
        //通过引入的方式，查找数据

        Value value_for(Key const& key,Value const& default_value) const
        {
            boost::shared_lock<boost::shared_mutex> lock(mutex);
            bucket_iterator const found_entry=find_entry_for(key);
            //返回查找的关键值

            return (found_entry==data.end())?default_value:found_entry->second;
        }
        //更新键值对

        void add_or_update_mapping(Key const& key,Value const& value)
        {
            std::unique_lock<boost::shared_mutex> lock(mutex);
            bucket_iterator const found_entry=find_entry_for(key);
            if(found_entry==data.end())
            {
                data.push_back(bucket_value(key,value));
            }else{
                found_entry->second=value;
            }
        }
        //移除关键字

        void remove_mapping(Key const& key)
        {
            std::unique_lock<boost::shared_mutex> lock(mutex);
            bucket_iterator const found_entry=find_entry_for(key);
            if(found_entry!=data.end())
            {
               data.erase(found_entry);
            }
        }
    };
    //end define bucket_type

    //定义查询的基本桶向量容器
```

```
    std::vector<std::unique_ptr<bucket_type> > buckets;
    //hash映射表

    Hash hasher;
    //根据关键字查找桶

    bucket_type& get_bucket(Key const& key) const
    {
        std::size_t const bucket_index=hasher(key)%buckets.size();
        return *buckets[bucket_index];
    }
//公共的类接口

public:
    //定义关键字类型

    typedef Key key_type;
    //定义映射的值

    typedef Value mapped_type;
    //定义hash函数

    typedef Hash hash_type;
    //基本的构造函数

    threadsafe_lookup_table(unsigned num_buckets=19,
                            Hash const& hasher_=Hash()):
                            buckets(num_buckets),
                            hasher(hasher_)
    {
        for(unsigned i=0;i<num_buckets;++i)
        {
            buckets[i].reset(new bucket_type);
        }
    }
    threadsafe_lookup_table(threadsafe_lookup_table const& other)=delete;
    threadsafe_lookup_table& operator=(threadsafe_lookup_table const& other)=delete;
    //根据关键字查找值
```

```
    Value value_for(Key const& key,
                    Value const& default_value=Value()) const
    {
        return get_bucket(key).value_for(key,default_value);
    }

    void add_or_update_mapping(Key const& key,Value const& value)
    {
        get_bucket(key).add_or_update_mapping(key,value);
    }
    void remove_mapping(Key const& key)
    {
        get_bucket(key).remove_mapping(key);
    }
};
```

6.3.2 编写一个使用锁的线程安全链表

链表的基本功能:

链表的基本操作

- 向列表添加一个元素
- 当某个条件满足时,就从链表中删除某个元素
- 当某个条件满足时,从链表中查找某个元素
- 当某个条件满足时,更新链表中的某个元素
- 将当前容器中链表中的每个元素,复制到另一个容器中

线程安全的迭代器

```
//定义模板类


template<typename   T>
class threadsafe_list
{
    //链表数据节点

    struct node
    {
        std::mutex m;
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
        //构造函数

        node():next(){}
        //数值构造函数

        node(T const& value):data(std::make_shared<T>(value)){}
    };
    //定义头部节点

    node head;
public:
        threadsafe_list(){}
        ~threadsafe_list()
        {
            remove_if([](node const&){return true;});
        }
        threadsafe_list(threadsafe_list const&  other)=delete;
        threadsafe_list& operator=(threadsafe_list const& other)=delete;
        //从头部插入

        void push_front(T const& value)
        {
            //创建新节点
            std::unique_ptr<node> new_node(new node(value));
            //头部节点加锁
```

```
        std::lock_guard<std::mutex> lk(head.m);
        new_node->next=std::move(head.next);
        head.next=std::move(new_node);
    }
    //定义迭代函数

    template<typename Function>
    void for_each(Function f)
    {
        node* current=&head;
        std::unique_lock<std::mutex> lk(head.m);
        //便利链表

        while(node* const next=current->next.get())
        {
            //保护下一个节点数据

            std::unique_lock<std::mutex> next_lk(next->m);
            //上一个节点解锁

            lk.unlock();
            //执行函数

            f(*next->data);
            //更改当前指针

            current=next;
            //移动对象

            lk=std::move(next_lk);
        }
    }
    //查找一个条件的元素
    //定义查找关键函数模板

    template<typename Predicate>
    std::shared_ptr<T> find_first_if(Predicate p)
    {
        node* current=&head;
```

```
        std::unique_lock<std::mutex> lk(head.m);
        while(node* const   next=current->next.get())
        {

            std::unique_lock<std::mutex> next_lk(next->m);
            lk.unlock();
            if(p(*next->data))
            {
                return next->data;
            }
            current=next;
            lk=std::move(next_lk);
        }
        return std::shared_ptr<T>();
    }
    //按照条件删除元素

    template<typename Predicate>
    void remove_if(Predicate p)
    {
        node* current=&head;
        std::unique_lock<std::mutex> lk(head.m);
        while(node* const next=current->next.get())
        {
            std::unique_lock<std::mutex> next_lk(next->m);
            //是否符合查找条件

            if(p(*next->data))
            {
                std::unique_ptr<node> old_next=std::move(current->next);
                current->next=std::move(next->next);
                next_lk.unlock();
            }else{
                //解锁下一个

                lk.unlock();
                //移动当前指针

                current=next;
```

```
                    //移动下一个锁

                    lk=std::move(next_lk);
                }
            }
        }
};
```

第7章 无锁并发数据结构设计

7.1 定义和意义

使用互斥量、条件变量,以及“期望”来同步阻塞数据的算法和数据结构。

无锁数据结构：作为无锁结构,就意味着线程可以并发的访问这个数据结构。但是一般，这样的线程不能做相同的操作，并且在无锁算法中的循环会让一些线程处于”饥饿”状态。 **无等待数据结构**：首先,是无锁数据结构;并且,每个线程都能在有限的步数内完成操作,暂且不管其他线程是如何工作的。

活锁：活锁的产生是,两个线程同时尝试修改数据结构,但每个线程所做的修改操作都会让另一个线程重启,所以两个线程就会陷入循环,多次的尝试完成自己的操作。

这就是“无锁-无等待”代码的缺点:虽然提高了并发访问的能力,减少了单个线程的等待时间,但是其可能会将整体性能拉低。

7.2 无锁数据结构的例子

一个简单的线程安全栈结构

```
template <typename T>
class lock_free_stack
{
    struct node
    {
        //获取指针数据

        std::shared_ptr<T> data;
        node* next;
        node(T const& data_):data(std::make_shared<T>(data_)){}

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

public:
    lock_free_stack();
    ~lock_free_stack();
    //push函数
    void push(T const& data)
    {
        node* const new_node=new node(data);
        //加载数据

        new_node->next=head.load();
        //用原子操作替换节点

        while(!head.compare_exchange_weak(new_node->next,new_node));
    }
    std::shared_ptr<T> pop(T& result)
    {
        node* old_head=head.load();
        //使用原子操作替换节点

        while(old_head&&!head.compare_exchange_weak(old_head,old_head->next));
        //返回指针值

        return old_head?old_head->data:std::shared_ptr<T>();
    }
```

```
};
```

7.2.2 停止内存泄露：使用无锁数据结构管理内存

可以添加原子变量让栈变为线程安全的栈,同时添加引用计数，帮助

```
template<typename T>
class lock_free_stack
{
private:
    //原子变量

    std::atomic<unsigned> threads_in_pop;
    void try_reclaim(node* old_head);
public:
    std::shared_ptr<T> pop()
    {
        //在做事之前,计数值加1

        ++threads_in_pop;
        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)
        {
            //回收删除的节点

            res.swap(old_head->data);
        }
        //从节点中直接提取数据,而非拷贝指针

        try_reclaim(old_head);
        return  res;
        }

}
```

采用引用计数的回收机制

```
template<typename   T>
class lock_free_stack
{
private:
    //即将被删除的数

    std::atomic<node*>  to_be_deleted;
    static void delete_nodes(node*  nodes)
    {
        while(nodes)
        {
            node* next=nodes->next;
            delete nodes;
            nodes=next;
        }

    }
    //删除头部节点

    void try_reclaim(node* old_head)
    {
        //是否为第一次删除

        if(threads_in_pop==1)
        {
            //声明“可删除”列表

            node* nodes_to_delete=to_be_deleted.exchange(nullptr);
            //是否只有一个线程调用pop

            if(!--threads_in_pop)
            {
                delete_nodes(nodes_to_delete);
            }else if(nodes_to_delete)
            {
                chain_pending_nodes(nodes_to_delete)
            }
            //删除节点
```

```
        delete old_head;

    }else{
        chain_pending_nodes(old_head);
        --threads_in_pop;
    }
}
void chain_pending_nodes(node* nodes)
{
    node* last=nodes;
    //让next指针指向链表的末尾

    while(node* const next=last->next) {
        last=next;
    }
    chain_pending_nodes(nodes,last);
}
void chain_pending_nodes(node* first,node* last)
{
    //last标记为即将删除

    last->next=to_be_deleted;
    //用循环来保证last->next的正确性

    while(!to_be_deleted.compare_exchange_weak(last->next,first));
}
void chain_pending_node(node* n)
{
    chain_pending_nodes(n,n);
}
```

7.2.3 检测使用风险指针(不可回收)的节点

风险指针 :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。

利用风险指针实现pop操作

```
std::shared_ptr<T> pop()
{
    //获取风险指针

    std::atomic<void*>& hp=get_hazard_pointer_for_current_thread();
    node* old_head=head.load();
    //比较交换操作失败，则重置操作1

    do
    {
        node* temp;
        // 1 直到将风险指针设为head指针

        do
        {
            temp=old_head;
            hp.store(old_head);
            old_head=head.load();
        }while(old_head!=temp);
    //检查head==old_head?head=old_head->next:head=old_head;

    }while(old_head&&!head.compare_exchange_strong(old_head,old_head->next));
    // 2 当声明完成,清除风险指针

    hp.store(nullptr);
    std::shared_ptr<T> res;
    if(old_head)
    {
        res.swap(old_head->data);
        // 3 在删除之前对风险指针引用的节点进行检查

        if(outstanding_hazard_pointers_for(old_head))
        {
            //将其放在链表中，之后进行回收

            reclaim_later(old_head);
        }else{
            delete  old_head;
```

```
        }
        //检查并删除风险节点

        delete_nodes_with_no_hazards();
    }
        return  res;
}
```

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;
};
//异常节点数组

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;
            //检查old_id是否含有hazard_pointers中的异常指针

            if(hazard_pointers[i].id.compare_exchange_strong(old_id.std::thread::get_id()))
            {
                //如果含有则hp指向该异常指针，

                hp=&hazard_pointers[i];
                break;
            }
        }
        //如果不含有风险指针就抛出异常

        if(!p)
        {
            throw std::runtime_error("No hazard pointer available");
        }
```

```
    }
    std::atomic<void *>&get_pointer
    {
        return hp->pointer;
    }
    ~hp_owner()
    {
        hp->pointer.store(nullptr);
        hp->id.store(std::thread::id());
    }

};
//获取当前的风险指针

std::atomic<void *>& get_hazard_pointer_for_current_thread()
{
    //每个线程都有自己的风险指针

    thread_local static hp_owner hazard;
    //获取指针数目

    return hazard.get_pointer();
}
//搜索风险表，查找对应记录

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;
}


//风险指针的回收函数
```

```
template<typename T>
void do_delete(void* p)
{
    delete static_cast<T*>(p);
}
//删除缓冲队列

struct data_to_reclaim
{
    void* data;
    std::Function<void(void* )> deleter;
    data_to_reclaim* next;
    template<typename T>
    //删除缓冲链中元素

    data_to_reclaim(T* p):data(p),deleter(&do_delete<T>),next(0){}
    ~data_to_reclaim()
    {
        deleter(data);
    }
};
//定义释放节点

std::atomic<data_to_reclaim*> nodes_to_reclaim;
//头插法将数据插入

void add_to_reclaim_list(data_to_reclaim* node)
{
    //指针指向下一个

    node->next=nodes_to_reclaim.load();
    //将数据节点与head相交换，因此最终插入到头结点之后

    while(!nodes_to_reclaim.compare_exchange_weak(node->next,node));
}
//创建相关实例，将数据添加到待删除队列

template<typename T>
void reclaim_later(T* data)
```

```
{
    add_to_reclaim_list(new data_to_reclaim(data));
}

//删除相关指针，将已经声明的链表节点进行回收

void delete_nodes_with_no_hazards()
{
    data_to_reclaim* current=nodes_to_reclaim.exchange(nullptr);
    //当节点不为空的时候

    while(current)
    {
        data_to_reclaim* const next=current->next;
        //判断节点是否属于风险指针

        if(!outstanding_hazard_pointers_for(current->data))
        {
            //没有指针就安全删除

            delete current;

        }else{
            //是风险指针就把节点添加到链表的后面，再统一删除

            add_to_reclaim_list(current);
        }
        current=next;

    }
}
```

7.2.4 检测使用引用计数的节点

通过增加外部引用计数,保证指针在访问期间的合法性。

分离计数方式的无锁栈

```
template<typename T>
class lock_free_stack
{
private:
    struct node;
    //指向下一个指针的节点

    struct counted_node_ptr
    {
        //外部引用计数

        int external_count;
        node* ptr;
    };
    struct node
    {
        std::shared_ptr<T> data;
        //节点的内部引用计数

        std::atomic<int> internal_count;
        //下一个指针节点

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

        }
    };
    //头部节点，它只有引用指针和计数，数据直接是node

    std::atomic<counted_node_ptr> head;
    //增加头部的引用计数

    void increase_head_count(counted_node_ptr& old_counter)
    {
        //创建新的计数指针

        counted_node_ptr new_counter;
        do{
```

```
            //new_counter指向新的指针

            new_counter=old_counter;
            //增加外部引用计数

            ++new_counter.external_count;
            //循环直到，old_counter指向头部，head指向new_counter;

        }while(!head.compare_exchange_strong(old_counter,new_counter));
        //修改指针的外部引用次数，每被引用一次，计数+1

        old_counter.external_count=new_counter.external_count;

    }
public:
    ~lock_free_stack()
    {
        while(pop());
    }
    //添加函数

    void push(T const& data)
    {
        //新的下一个指针

        counted_node_ptr new_node;
        new_node.ptr=new node(data);
        new_node.external_count=1;
        //新节点的下一个节点指向,old head

        new_node.ptr->next=head.load();
        //便利指针，直到new_node.ptr的下一个指针是head;即现在最前面的指针是new_node,将head指针指向new_node;

        while(!head.compare_exchange_weak(new_node.ptr->next,new_node));
    }
    //pop弹出函数

    std::shared_ptr<T> pop()
    {
```

```
        counted_node_ptr old_head=head.load();
        for(;;)
        {
            increase_head_count(old_head);
            //获取头部数据指针

            node* const ptr=old_head.ptr;
            //如果是一个空指针

            if(!ptr)
            {
                return std::shared_ptr<T>();
            }
            //将head指针后移

            if(head.compare_exchange_strong(old_head,ptr->next))
            {
                //返回指针数据

                std::shared_ptr<T> res;

                res.swap(ptr->data);
                //取出节点后，头部节点的引用计数-2

                int const count_increase=old_head.external_count-2;
                //如果现在的内部引用计数为0

                if(ptr->internal_count.fetch_add(count_increase)==-count_increase)
                {
                    //直接删除指针

                    delete ptr;
                }
                return res;
                //如果指针的内部引用计数为2

            }else if(ptr->internal_count.fetch_sub(1)==1){
                //删除指针
```

```
                delete ptr;
            }
        }
    }

};
```

7.2.5 应用于无锁栈上的内存模型

对于不同的多线程相互数据，在修改内存之前，需要检查一下操作之间的依赖关系。然后再去确定适合这种需求关系的最小内存。

对于push操作，接受数据之后，先构造节点，再插入队列–设置head,因此push中的唯一原子操作就是 `compare_exchange_weak()` 函数，对于同push()操作没有必要考虑，它需要和pop()之中的 `head.compare_exchange_strong` 有严格的内存顺序。

对于pop()操作，必须在访问 `next` 值之前使用 `std::memory_order_acquire` 或者更加严格的内存操作顺序，保证 `next` 指针指向的内容不被改变。因为在 `increase_head_count()` 中使用 `compare_exchange_strong()` 就获取 `next` 指针指向的旧值。因此在交换成功的时候必须使用严格内存序，但是当交换失败时，因为只涉及到内部操作，因此可以使用松散内存序。然后循环直到交换成功。

基于引用计数和松散原子操作的无锁线程

```
class lock_free_stack
{
private:
    struct node;
    struct counted_node_ptr
    {
        int external_count;
        node*   ptr;
    };
    struct  node
    {
        node(T  const&  data_):
        data(std::make_shared<T>(data_)),
                        internal_count(0)
                        {}
        std::shared_ptr<T>  data;
        std::atomic<int>    internal_count;
        counted_node_ptr    next;
    };
    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());
    }
    void push(T const& data)
    {
        counted_node_ptr new_node;
        new_node.ptr=new node(data);
        new_node.external_count=1;
        //加载head数据

        new_node.ptr->next=head.load(std::memory_order_relaxed)
        while(!head.compare_exchange_weak(
            new_node.ptr->next,
            new_node,
            //这里必须要和increase_head_count的compare_exchange_strong中成功时有序

            std::memory_order_release,
            std::memory_order_relaxed
            ));
    }
    std::shared_ptr<T> pop()
    {
        //加载头部指针，因为这里的载入没有强制的竞争行为，所以可以是relaxed的

        counted_node_ptr old_head=head.load(std::memory_order_relaxed);
        for(;;)
        {
            increase_head_count(old_head);
            node* const ptr=old_head.ptr;
            if(!ptr)
            {
                return std::shared_ptr<T>();
            }
            if(head.compare_exchange_strong(old_head,ptr->next,std::memory_order_relaxed))
            {
                std::shared_ptr<T> res;
                res.swap(ptr->data);
                int const count_increase=old_head.external_count-2;
                if(ptr->internal_count.fetch_add(count_increase,std::memory_order_release)==-count_increase)
                {
```

```
                    delete ptr;
                }
                return res;
            }else if(
                ptr->internal_count.fetch_add(-1,std::memory_order_relaxed)==1
                )
            {
                ptr->internal_count.load(std::memory_order_acquire);
                delete ptr;
            }
        }

    }

};
```

7.2.6 写一个无锁的线程安全队列

先看一个简单的生产者/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())
        {
            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()
    {
        //获取头节点
```

```
        node* old_head=pop_head();
        if(!old_head)
        {
            return  std::shared_ptr<T>();
        }
        //获取头部数据

        std::shared_ptr<T> const res(old_head->data);
        delete old_head;
        //返回头部数据

        return res;
    }
    void push(T new_value)
    {
        std::shared_ptr<T>  new_data(std::make_shared<T>(new_value));
        node* p=new node;
        //获取尾部节点

        node* const old_tail=tail.load();
        //将尾部数据和新指针交换

        old_tail->data.swap(new_data);
        //更换尾部数据
        old_tail->next=p;
        //尾指针存储p

        tail.store(p);
    }
};
```

对于线程安全的队列而言，需要注意的地方是在尾部节点插入的地方和头部节点删除的地方，可以借鉴内外部的引用计数的方法，在删除和添加操作中，使用原子操作，避免线程之间的相互竞争。

线程安全队列的完全代码,这里全部写出来，注意看代码的注释

```
template<typename T>
class lock_free_queue
{
private:
    struct node;
    //节点之间的链接类

    struct counted_node_ptr
    {
        int external_count;
        node* ptr;
    };
    //头指针

    std::atomic<counted_node_ptr> head;
    //尾部指针

    std::atomic<counted_node_ptr> tail;
    //引用计数器

    struct node_counter
    {
        //内部引用计数,大小为30bit

        unsigned internal_count:30;
        //外部引用计数,大小为2bit 即0-3

        unsigned external_counters：2;
    };
    //定义元素节点

    struct node
    {
        //基本构造函数
        node()
        {
            node_counter new_count;
            new_count.internal_count=0;
            //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
```

```
            new_count.external_counts=2;
            //存储新值

            count.store(new_count);
            //下一个节点指针为空指针

            next.ptr=nullptr;
            //下一个指针的外部引用计数为0

            next.external_count=0;
        }
        //释放一个节点引用

        void release_ref()
        {
            //获取计数器指针

            node_counter old_counter=count.load(std::memory_order_relaxed);
            node_counter new_counter;
            do
            {
                //将旧计数器，存入新的临时变量中

                new_counter=old_counter;
                //外部引用计数--

                --new_counter.internal_count;
                //当count与old_count相同时结束循环

            }while(!count.compare_exchange_strong(
                old_counter,
                new_counter,
                std::memory_order_acquire,
                std::memory_order_relaxed
                ));
            //当内外部引用都为空的时候，删除指针

            if(!new_counter.internal_count&&
```

```
            !new_counter.external_counters)
        {
            delete this;
        }

    }

    std::atomic<T*> data;
    //节点计数器，记录内外部引用次数

    std::atomic<node_counter> count;
    //链接关系类，next指针

    counted_node_ptr next;
};

//增加一个外部节点的引用

static void increase_external_count(
    std::atomic<counted_node_ptr>& counter,
    counted_node_ptr& old_counter
    )
{
    //临时记录变量

    counted_node_ptr new_counter;
    do
    {
        //暂存旧计数器

        new_counter=old_counter;
        //增加外部引用

        ++new_counter.external_count;
        //当counter和old_counter指向相同时，跳出循环

    }while(!counter.compare_exchange_strong(
        old_counter,
        new_counter,
```

```
            std::memory_order_acquire,
            std::memory_order_relaxed
            ));
        //计算结构存入old_counter中

        old_counter.external_count=new_counter.external_count;
    }
    //删除外部节点的引用

    static void free_external_counter(counted_node_ptr& old_node_ptr)
    {
        //获取旧指针的临时变量

        node* const ptr=old_node_ptr.ptr;
        //和添加时相反，减少两个外部引用

        int const count_increase=old_node_ptr.external_count-2;
        //获取计数器

        node_counter old_counter=ptr->count.load(std::memory_order_relaxed);
        //创建新计数器

        node_counter new_counter;

        do
        {
            new_counter=old_counter;
            //外部计数器--

            --new_counter.external_counters;
            //拷贝引用数目

            new_counter.internal_count+=count_increase;
        }while(!ptr->count.compare_exchange_strong(
            old_counter,
            new_counter,
            std::memory_order_acquire,
            std::memory_order_relaxed
            ));
```

```
        if(!new_counter.internal_count&&
            !new_counter.external_counters)
        {
            delete ptr;
        }
    }
public:
    lock_free_queue();
    ~lock_free_queue();
    //添加新元素函数

    void push(T new_value)
    {
        //创建智能指针

        std::unique_ptr<T> new_data(new T(new_value));
        //下一个指向链接

        counted_node_ptr new_next;
        new_next.ptr=new node;
        new_next.external_count=1;
        //暂存旧的尾部指针

        counted_node_ptr old_tail=tail.load();
        for(;;)
        {
            //增加现有指针和尾部指针的外部引用计数

            increase_external_count(tail,old_tail);
            T* old_data=nullptr;
            //将尾部指针的数据更换为新数据，将tail指针指向新尾部

            if(old_tail.ptr->data.compare_exchange_strong(
                old_data,
                new_data.get()
                ))
                //当old_data=old_tail.ptr->data时成立

            {
```

```
                //更新尾指针指向

                old_tail.ptr->next=new_next;
                //将旧指针移动到old_tail

                old_tail=tail.exchange(new_next);
                //释放外部计数指针

                free_external_counter(old_tail);
                //释放指针所有权

                new_data.release();
                //跳出循环

                break;

            }
            //释放指针所有权

            old_tail.ptr->release_ref();
        }
    }
    //出队列相关函数

    std::unique_ptr<T> pop()
    {
        //加载头节点

        counted_node_ptr old_head=head.load(std::memory_order_relaxed);
        for(;;)
        {
            //增加外部计数器

            increase_external_count(head,old_head);
            //获取临时头节点中的node指针

            node* const ptr=old_head.ptr;
            //首尾节点指向一处，即队列为空
```

```
            if(ptr==tail.load().ptr)
            {
                ptr->release_ref();
                return std::unique_ptr<T>();
            }
            //将head指针，指向old_head旧指针指向的节点

            if(head.compare_exchange_strong(old_head,ptr->next))
            {
                //获取左值

                T* const res=ptr->data.exchange(nullptr);
                //释放外部引用计数

                free_external_counter(old_head);
                //返回获取的指针

                return std::unique_ptr<T>(res);
            }
            //释放旧节点

            ptr->release_ref();
        }
    }
};
```

无锁队列中的线程间互助

通过在node节点中设置next指针可以在pop()函数中通过对 `next` 指针的读取方便快速的使用compare_exchange_strong,进行头指针移动；对于push的实现稍微复杂一点

```
template<typename   T>
class lock_free_queue
{
private:
    struct node
    {
        std::atomic<T*> data;
        std::atomic<node_counter> count;
        //下一个指针

        std::atomic<counted_node_ptr> next;
    };

    void set_new_tail(
    counted_node_ptr &old_tail,
    counted_node_ptr const &new_tail
    )
    {
        //获取旧尾指针

        node* const current_tail_ptr=old_tail.ptr;
        while(!tail.compare_exchange_weak(old_tail,new_tail)&&old_tail.ptr==current_tail_ptr);
        //当前尾部指针与旧指针相同

        if(old_tail.ptr==current_tail_ptr)
            //释放外部计数

            free_external_counter(old_tail);
        else
            //否则释放当前指针

            current_tail_ptr->release_ref();
    }
public:
    void push(T new_value)
    {
        //新数据

        std::unique_ptr<T> new_data(new T(new_value));
```

```
        //新节点

        counted_node_ptr new_next;
        new_next.ptr=new node;
        //外部引用设置为1

        new_next.external_count=1;
        counted_node_ptr old_tail=tail.load();
        for(;;)
        {
            //增加外部引用

            increase_external_count(tail,old_tail);
            //获取旧数据

            T* old_data=nullptr;
            //旧尾指针数据与old_data相同

            if(old_tail.ptr->data.compare_exchange_strong(
                                    old_data,
                                    new_data.get()
                                    ))
            {
                //初始化next指针，准备交换数据
                counted_node_ptr old_next={0};
                //当尾指针指向新节点时

                if(!old_tail.ptr->next.compare_exchange_strong(
                                            old_next,
                                            new_next)
                    )
                {

                    //删除新节点

                    delete new_next.ptr;
                    //新next指针指向原指针指向

                    new_next=old_next;
```

```
                }
                set_new_tail(old_tail,new_next);
                new_data.release();
                break;
            }else{
                //初始化新的尾指针
                counted_node_ptr old_next={0};
                //如果旧尾指针next指向为old_next,将next指针指向新next

                if(old_tail.ptr->next.compare_exchange_strong(
                                        old_next,new_next))
                {
                        old_next=new_next;
                        new_next.ptr=new node;
                }
                    set_new_tail(old_tail,old_next);
            }
        }
    }
};
```

7.3 对于设计无锁数据结构的指导建议

参考链接： c++11 内存模型解读 (https://blog.csdn.net/weixin_36145588/article/details/78873917)

表 6-3 C++11 中的 memory_order 枚举值

枚举值	定义规则
memory_order_relaxed	不对执行顺序做任何保证
memory_order_acquire	本线程中，所有后续的读操作必须在本条原子操作完成后执行
memory_order_release	本线程中，所有之前的写操作完成后才能执行本条原子操作
memory_order_acq_rel	同时包含 memory_order_acquire 和 memory_order_release 标记
memory_order_consume	本线程中，所有后续的有关本原子类型的操作，必须在本条原子操作完成之后执行
memory_order_seq_cst	全部存取都按顺序执行

通常情况下我们把atomic成员函数可使用memory_order值分为以下3组:

- 原子存储操作(store)可使用:memory_order_relaxed、memory_order_release、memory_order_seq_cst
- 原子读取操作(load)可使用:memory_order_relaxed、memory_order_consume、memory_order_acquire、memory_order_seq_cst
- RMW操作(read-modify-write)即同时读写的操作,如atomic_flag.test_and_set()操作,atomic.atomic_compare_exchange()等都是需要同时读写的。可使用:memory_order_relaxed、memory_order_consume、memory_order_acquire、memory_order_release、memory_order_acq_rel、memory_order_seq_cst

根据memory_order使用情况,我们可以将其为 3 类:

- 顺序一致性模型:std::memory_order_seq_cst;最稳定，代价最高；原子操作默认的模型,在C++11中的原子类型的变量在线程中总是保持着顺序执行的特性。
- Acquire-Release 模型:std::memory_order_consume, std::memory_order_acquire, std::memory_order_release, std::memory_order_acq_rel；若线程A中的一个原子store带memory_order_release标签，而线程B中来自同一变量的原子load带memory_order_acquire标签，从线程A的视角发生先于原子store的所有内存写入(non-atomic and relaxed atomic)，在线程B中成为可见副作用，一旦线程B中的原子加载完成，则保证线程B能观察到线程A写入内存的所有内容。

注意：同步仅建立在release和acquire同一原子对象的线程之间，其他线程可能看到与被同步线程的一者或两者相异的内存访问顺序。

- Relax 模型:std::memory_order_relaxed；最不稳定，代价最低

7.3.1 使用 `std::memory_order_seq_cst` 的原型

std::memory_order_seq_cst比起其他内存序要简单的多，因为所有操作都将其作为总序。本章的所有例子,都是从std::memory_order_seq_cst开始,只有当基本操作正常工作的时候,才放宽内存序的选择。

7.3.2 对无锁内存的回收策略

当有其他线程对节点进行访问的时候,节点无法被任一线程删除;为避免过多的内存使用,还是希望这个节点在能删除的时候尽快删除。本章中介绍了三种技术来保证内存可以被安全的回收:

- 等待无线程对数据结构进行访问时,删除所有等待删除的对象。
- 使用风险指针来标识正在被线程访问的对象。
- 对对象进行引用计数,当没有线程对对象进行引用时,将其删除。

7.3.3 指导建议:小心ABA问题

在“基于比较/交换”的算法中要格外小心“ABA问题”。其流程是:

1. 线程1读取原子变量x,并且发现其值是A。
2. 线程1对这个值进行一些操作,比如,解引用(当其是一个指针的时候),或做查询,或其他操作。
3. 操作系统将线程1挂起。
4. 其他线程对x执行一些操作,并且将其值改为B。
5. 另一个线程对A相关的数据进行修改(线程1持有),让其不再合法。可能会在释放指针指向的内存时,代码产生剧烈的反应(大问题);或者只是修改了相关值而已(小问题)。
6. 再来一个线程将x的值改回为A。如果A是一个指针,那么其可能指向一个新的对象,只是与旧对象共享同一个地址而已。
7. 线程1继续运行,并且对x执行“比较/交换”操作,将A进行对比。这里,“比较/交换”成功 (因为其值还是A),不过这是一个错误的A(the wrong A value)。从第2步中读取的数据不再合法,但是线程1无法言明这个问题,并且之后的操作将会损坏数据结构。

解决方案：解决这个问题的一般方法是,让变量x中包含一个ABA计数器。“比较/交换”会对加入计数器的x进行操作。每次的值都不一样,计数随之增长,所以在x还是原值的前提下,即使有线程对x进行修改,“比较/交换”还是会失败。

7.3.4 指导建议:识别忙等待循环和帮助其他线程

在最终队列的例子中,已经见识到线程在执行push操作时,必须等待另一个push操作流程的完成。等待线程就会被孤立,将会陷入到忙等待循环中,当线程尝试失败的时候,会继续循环,这样就会浪费CPU的计算周期。当忙等待循环结束时,就像一个阻塞操作解除,和使用互斥锁的行为一样。通过对算法的修改,当之前的线程还没有完成操作前,让等待线程执行未完成的步骤,就能让忙等待的线程不再被阻塞(**减小锁的粒度**)。在队列例中,需要将一个数据成员转换为一个原子变量,而不是使用非原子变量和使用“比较/交换”操作来做这件事;要是在更加复杂的数据结构中,这将需要更加多的变化来满足需求。

PREVIOUS
C++ 并发编程笔记(二)
(/2019/07/06/CPLUSPLUS_CONCURRENCY_IN_ACTION_02/)

NEXT
C++ 并发编程笔记(四)
(/2019/07/06/CPLUSPLUS_CONCURRENCY_IN_ACTION_04/)

FEATURED TAGS (/tags/)

C++ (/tags/#C++) 基础编程 (/tags/#%E5%9F%BA%E7%A1%80%E7%BC%96%E7%A8%8B) C/C++ (/tags/#C/C++)
后台开发 (/tags/#%E5%90%8E%E5%8F%B0%E5%BC%80%E5%8F%91) C (/tags/#C) 网络编程 (/tags/#%E7%BD%91%E7%BB%9C%E7%BC%96%E7%A8%8B)
STL源码解析 (/tags/#STL%E6%BA%90%E7%A0%81%E8%A7%A3%E6%9E%90) Linux (/tags/#Linux)
操作系统 (/tags/#%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F) 程序设计 (/tags/#%E7%A8%8B%E5%BA%8F%E8%AE%BE%E8%AE%A1)
优化 (/tags/#%E4%BC%98%E5%8C%96) UML (/tags/#UML) UNIX (/tags/#UNIX) 学习笔记 (/tags/#%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0)
面试 (/tags/#%E9%9D%A2%E8%AF%95) Java (/tags/#Java) 读书笔记 (/tags/#%E8%AF%BB%E4%B9%A6%E7%AC%94%E8%AE%B0) go (/tags/#go)
阅读笔记 (/tags/#%E9%98%85%E8%AF%BB%E7%AC%94%E8%AE%B0)

FRIENDS

WY (http://zhengwuyang.com) 简书·JF (http://www.jianshu.com/u/e71990ada2fd) Apple (https://apple.com)
Apple Developer (https://developer.apple.com/)

(https://www.facebook.com/wangpengcheng) (https://github.com/wangpengcheng)

Theme on GitHub (https://github.com/wangpengcheng/wangpengcheng.github.io.git) | Star 12