1. Thread 

（译注：本中文文档对应于Boost 1.35.0的英文文档。原英文文档版权归Anthony Williams所有，本中文文档版权归Tu Yongce <yongce@126.com>所有。）

1.1. 概述

Boost.Thread使我们能够用可移植的C++代码来使用有共享数据的多个执行线程。它提供了类和函数来管理线程本身，以及在线程间同步数据或者为每一个线程提供单独的数据拷贝。

Boost.Thread库最初由William E. Kempf设计和实现。本版本进行了大量重写以紧密跟进C++标准委员会发布的建议，特别是N2497、N2320、N2184、N2139和N2094。

1.2. 自Boost 1.34以来的变化

自Boost 1.34发行版以来，几乎Boost.Thread的每一行代码都发生了变化（译注：Boost.Thread最初发行版出现于Boost 1.25.0）。然而，大多数接口变化被扩展了，因此新代码很大程度上向后兼容于旧有代码。新的特性和breaking变化描述如下。

新特性

- boost::thread实例和各种锁类型现在是可移动的。
- 线程可以在中断点被中断。
- 条件变量现在能够用于任何实现了Lockable概型的类型，通过使用boost::condition_variable_any（boost::condition是boost::condition_variable_any的类型别名，以向后兼容）。boost::condition_variable被提供作为优化，并只与boost::unique_lock<boost::mutex>（boost::mutex::scoped_lock）一块工作。
- 线程ID从boost::thread中分离出来，因此一个线程可以获得它自己的ID（使用boost::this_thread::get_id()），并且这些ID可以用作关联容器的键值，因为它们有比较操作符的全部集合。
- 超时设定现在使用Boost.Date_Time库实现，通过类型别名boost::system_time提供的绝对超时设定，并支持许多情形下的相对超时。boost::xtime被支持以向后兼容。
- 锁被实现为公共可访问的模板boost::lock_guard、boost::unique_lock、boost::shared_lock和boost::upgrade_lock，它们在互斥类型基础上模板化。Lockable概型被扩展包含公共可用的

lock()和unlock()成员函数，它们被锁类型使用。

breaking变化

下面的列表应该覆盖了所有打破向后兼容的公共接口变化。

- boost::try_mutex已经被移除，其功能包含在boost::mutex中。boost::try_mutex作为一个类型别名，但不再是一个单独的类。
- boost::recursive_try_mutex已经被移除，其功能包含在boost::recursive_mutex中。boost::recursive_try_mutex作为一个类型别名，但不再是一个单独的类。
- boost::detail::thread::lock_ops已经被移除。依赖于lock_ops实现细节的代码将再工作，因为这已经被移除，事实上不再需要，因为互斥类型现在已经提供了公共的lock()和unlock()成员函数。
- 拥有第二个类型为bool的参数的scoped_lock构造函数不再提供。在先前的版本中，

```
boost::mutex::scoped_lock some_lock(some_mutex, false);
```

能够被用于创建一个与一个互斥对象关联的锁对象，但是在构造时不会锁定。现在通过采用boost::defer_lock_type作为第二个参数的构造函数来代替该功能：

```
boost::mutex::scoped_lock some_lock(some_mutex, boost::defer_lock);
```

- boost::read_write_mutex已经被boost::shared_mutex代替。

1.3. 线程管理

1.3.1. 概要

boost::thread类负责启动和管理线程。每个boost::thread对象代表一个单独的执行线程或者Not-a-Thread，至多一个boost::thread对象代表一个给定的执行线程：boost::thread类型的对象是不可复制的。

然而，boost::thread类型的对象是可移动的（movable），因此它们能够被存储在move-aware容器中，也可由函数返回。这允许线程创建细节被封装在一个函数中。例如：

```
boost::thread make_thread();
void f()
{
    boost::thread some_thread = make_thread();
    some_thread.join();
}
```

启动线程

通过传递一个可调用类型的对象给构造函数来启动一个新的线程，传递的对象必须能够进行无参调用。该对象随后被复制到内部存储中，并在新创建的执行线程中调用。如果该对象禁止（或者不能）复制，那么可以使用boost::ref来传递函数对象的引用。在这种情况下，Boost.Thread的用户必须确保被引用的对象比新创建的执行线程的生命期更长。例如：

```
struct callable
{
    void operator() ();
};

boost::thread copies_are_safe()
{
    callable x;
    return boost::thread(x);
} // x被销毁，但是新创建的线程有一份拷贝，因此没有问题

boost::thread oops()
{
    callable x;
    return boost::thread(boost::ref(x));
} // x被销毁，但新创建的线程仍然拥有一个对它的引用，将导致未定义行为
```

如果你想使用一个需要参数的函数或者可调用对象来构造一个boost::thread的实例，这可以通过boost::bind来实现。例如：

```
void find_the_question(int the_answer);
boost::thread deep_thought_2(boost::bind(find_the_question, 42));
```

汇合与分离（Joining & Detaching）

当代表一个执行线程的boost::thread对象被销毁时，线程变为分离的（detached）。一旦一个线程是分离的，它将继续执行直到该函数或者可调用对象的调用完成或者程序终止。线程也可以通过显式地对boost::thread对象调用detach()成员函数来分离。这种情况下，boost::thread对象不再代表一个已分离线程，而代表Not-a-Thread。

为了等待一个执行线程结束，必须使用boost::thread对象的join()或者timed_join()成员函数。join()将阻塞调用线程，直到boost::thread对象代表的线程结束。如果boost::thread对象代表的执行线程已经结束，或者boost::thread对象代表Not-a-Thread，那么join()立即返回。timed_join()是类似的，除了一点：在指定时间内被等待线程仍然没有结束，timed_join()调用也将返回。

中断

可以通过调用对应的boost::thread对象的interrupt()成员函数来中断一个正在运行的线程。在中断启用下，当一个被中断的线程接下来执行某一指定中断点时（或者如果它当前正被阻塞，并且正在执行一个中断点），那么该被中断线程将抛出一个boost::thread_interrupted异常。如果未被捕获，将引起被中断线程的执行终止。和其它任何异常一样，栈将被展开（unwind），对象析构函数将被调用。

如果线程想避免被中断，它可以创建一个boost::this_thread::disable_interruption的实例。该类对

象在构造后对创建它的线程禁用中断，并在析构后恢复中断状态到禁用之前的状态。例如：

```
void f()
{
    // 这里中断被启用
    {
        boost::this_thread::disable_interruption di;
        // 中断被禁用
        {
            boost::this_thread::disable_interruption di2;
            // 中断仍被禁用
        } // di2被销毁，中断状态被恢复
        // 中断仍被禁用
    } // di被销毁，中断状态被恢复
    // 现在中断被启用
}
```

boost::this_thread::disable_interruption实例的影响可以通过构造一个boost::this_thread::restore_interruption实例来临时撤消，并传入该boost::this_thread::disable_interruption对象。这将恢复中断状态到该boost::this_thread::disable_interruption对象被构造之前的状态。当该boost::this_thread::restore_interruption对象被销毁时，再次禁用中断。例如：

```
void g()
{
    // 这里中断被启用
    {
        boost::this_thread::disable_interruption di;
        // 中断被禁用
        {
            boost::this_thread::restore_interruption ri(di);
            // 现在启用中断
        } // ri被销毁，再次禁用中断
    } // di被销毁，中断状态被恢复
    // 现在中断被启用
}
```

在任何时刻，可以通过调用boost::this_thread::interruption_enabled()来查询当前线程的中断状态。

预定义中断点

下面的函数是中断点，如果当前线程的中断被启用并请求中断当前线程，那么这些函数将抛出boost::thread_interrupted异常：

```
boost::thread::join()
boost::thread::timed_join()
boost::condition_variable::wait()
boost::condition_variable::timed_wait()
boost::condition_variable_any::wait()
```

```
boost::condition_variable_any::timed_wait()
boost::thread::sleep()
boost::this_thread::sleep()
boost::this_thread::interruption_point()
```

线程ID

类boost::thread::id的对象可以用于标识线程。每一个正在运行的执行线程都有一个唯一的ID，可以通过相应的boost::thread对象调用get_id()成员函数获得，或者通过在线程内调用boost::this_thread::get_id()获得。类boost::thread::id的对象可以复制，可以用作关联容器的键值：提供了所有的比较操作符。也可以使用流插入操作符把线程ID写到输出流中，虽然其输出格式未指定。

每一个boost::thread::id的实例要么引向某个线程，要么引向Not-a-Thread。在比较时，引向Not-a-Thread的实例都是相等的，但不等于任何一个引向实际执行线程的实例。boost::thread::id上的比较操作符为每一个不相等的线程ID产生一个全序。

1.3.2. 类 thread

```
class thread
{
public:
    thread();
    ~thread();

    template <class F>
    explicit thread(F f);

    template <class F>
    thread(detail::thread_move_t<F> f);

    // move support
    thread(detail::thread_move_t<thread> x);
    thread& operator=(detail::thread_move_t<thread> x);
    operator detail::thread_move_t<thread>();
    detail::thread_move_t<thread> move();

    void swap(thread& x);

    class id;
    id get_id() const;

    bool joinable() const;
    void join();
    bool timed_join(const system_time& wait_until);

    template<typename TimeDuration>
```

```
    bool timed_join(TimeDuration const& rel_time);

    void detach();

    static unsigned hardware_concurrency();

    typedef platform-specific-type native_handle_type;
    native_handle_type native_handle();

    void interrupt();
    bool interruption_requested() const;

    // backwards compatibility
    bool operator==(const thread& other) const;
    bool operator!=(const thread& other) const;

    static void yield();
    static void sleep(const system_time& xt);
};
```

默认构造函数

```
thread();
```

效果：构造一个boost::thread实例，引向Not-a-Thread。

构造函数

```
template <typename Callable>
thread(Callable func);
```

效果：参数func被线程库复制到内部存储中，然后在一个新创建的执行线程上调用那份拷贝。

后件：*this引向一个新创建的执行线程。

抛出：如果错误发生，抛出boost::thread_resource_error异常。

析构函数

```
~thread();
```

效果：如果*this拥有一个关联的执行线程，调用detach()；销毁*this。

成员函数 joinable()

```
bool joinable() const;
```

返回：如果*this引向一个执行线程，则返回true；否则返回false。

成员函数 join()

```
void join();
```

前件：this->get_id() != boost::this_thread::get_id()

效果：如果*this引向一个执行线程，等待那个执行线程完成。

后件：如果*this引向一个执行线程，那么那个执行线程已经完成。*this不再引向任何执行线程。

抛出：如果当前线程被中断，将抛出boost::thread_interrumpted。

备注：join()是一个预定义的中断点。

成员函数 timed_join()

```
bool timed_join(const system_time &wait_until);

template <typename TimeDuration>
bool timed_join(TimeDuration const &rel_time);
```

前件：this->get_id() != boost::this_thread::get_id()

效果：如果*this引向一个执行线程，那么等待那个执行线程结束，直到到达时刻wait_until或者超出指定的时间段rel_time。如果*this没有引向执行线程，那么立即返回。

返回：如果*this引向一个执行线程，并且该执行线程在超时前结束，那么返回true；否则返回false。

后件：如果*this引向一个执行线程，并且timed_join返回true，那么该执行线程已经结束，并且*this不再引向任何执行线程。如果timed_join返回false，*this不会发生变化。

抛出：如果当前执行线程被中断，那么将抛出boost::thread_interrupted异常。

备注：timed_join是一个预定义中断点。

成员函数 detach()

```
void detach();
```

效果：如果*this引向一个执行线程，那么该执行线程变为分离的，并且不再有关联的boost::thread对象。

后件：*this不再引向任何一个执行线程。

成员函数 get_id()

```
thread::id get_id() const;
```

返回：如果*this引向一个执行线程，那么将返回一个代表那个线程的boost::thread::id实例；否则返回一个默认构造的boost::thread::id。

成员函数 interrupt()

```
void interrupt();
```

效果：如果*this引向一个执行线程，那么在下次遇到任何一个预定义中断点时，且在启用中断的情形下，该线程将被中断，或者如果它当前在中断启用的情形下阻塞于一个预定义中断点的调用（？没明白）。

静态成员函数 hardware_concurrency()

```
unsigned hardware_concurrency();
```

返回：当前系统可用的硬件线程数（例如，CPU数、内核数或者超线程单元数），或者如果该信息不可用时返回0。

operator ==

```
bool operator == (const thread &other) const;
```

返回：get_id() == other.get_id()

operator !=

```
bool operator != (const thread &other) const;
```

返回：get_id() != other.get_id()

静态成员函数 sleep()

```
void sleep(system_time const &abs_time);
```

效果：挂起当前线程，直到指定时刻到达。

抛出：如果当前执行线程被中断，则抛出boost::thread_interrupted。

备注：sleep()是一个预定义中断点。

静态成员函数 yield()

```
void yield();
```

效果：参见boost::this_thread::yield()。

1.3.2.1. 类 thread::id

```
class thread::id
{
public:
    id();

    bool operator==(const id& y) const;
    bool operator!=(const id& y) const;
    bool operator<(const id& y) const;
    bool operator>(const id& y) const;
    bool operator<=(const id& y) const;
    bool operator>=(const id& y) const;

    template<class charT, class traits>
    friend std::basic_ostream<charT, traits>&
    operator<< (std::basic_ostream<charT, traits>& os, const id& x);
};
```

默认构造函数

```
id();
```

效果：构造一个boost::thread::id实例，代表Not-a-Thread。

operator ==

```
bool operator == (const id &y) const;
```

返回：如果*this和y代表同一个执行线程，或者都代表Not-a-Thread，那么将返回true；否则返回false。

operator !=

```
bool operator != (const id &y) const;
```

返回：如果*this和y代表不同的执行线程，或者一个代表执行线程，另一个代表Not-a-Thread，那么返回true；否则返回false。（译注：返回!(*this == y)）

operator <

```
bool operator< (const id &y) const;
```

返回：如果*this != y为true且在实现定义的boost::thread::id值的全序中*this在y之前，则返回true；否则返回false。

备注：一个代表Not-a-Thread的boost::thread::id实例在比较时总是小于一个代表执行线程的实例。

operator >

```
bool operator > (const id &y) const;
```

返回：y < *this

operator <=

```
bool operator <= (const id &y) const;
```

返回：!(y < *this)

operator >=

```
bool operator >= (const id &y) const;
```

返回：!(this < y)

友元 operator <<

```
template <class charT, class traits>
friend std::basic_ostream<chart, traits>&
operator << (std::basic_ostream<chart, traits> &os, const id &x);
```

效果：把boost::thread::id实例x的表示写入到流os中，以致对于boost::thread::id的两个实例a和b，如果a == b，那么a和b的输出表示相同，如果a != b，那么输出表示不同。

返回：os

1.3.3. 名字空间 this_thread

get_id() interruption_point() interruption_requested() interruption_enabled() sleep() yield() at_thread_exit() Class disable_interruption Class restore_interruption

非成员函数 get_id()

```
thread::id get_id();
```

返回：代表当前正在执行的线程的boost::thread::id实例。

抛出：如果发生错误，抛出boost::thread_resource_error异常。

非成员函数 interruption_point()

```
void interruption_point();
```

效果：检查是否当前线程已经被中断。

抛出：如果boost::this_thread::interruption_enable()和boost::this_thread::interruption_requested()都返回true时，抛出boost::thread_interrupted异常。

非成员函数 interruption_requested()

```
bool interruption_requested();
```

返回：如果当前线程已经被请求中断，则返回true；否则返回false。

非成员函数 interruption_enabled ()

```
bool interruption_enabled ();
```

返回：如果当前线程已经启用中断，则返回true；否则返回false。

非成员函数 sleep ()

```
template <typename TimeDuration>
void sleep(TimeDuration const &rel_time);
```

效果：挂起当前线程，直到指定的时间过去。

抛出：如果当前执行线程被中断，那么抛出boost::thread_interrupted异常。

备注：sleep()是一个预定义中断点。

非成员函数 yield ()

```
void yield();
```

效果：放弃当前线程的时间片的剩余部分，允许其它线程运行。

非成员函数模板 at_thread_exit()

```
template <typename Callable>
void at_thread_exit(Callable func);
```

效果：func的一份拷贝被存储在线程存储中，并在当前线程退出时调用该拷贝。

后件：func的一份拷贝被存储以在线程退出时调用。

抛出：如果无法为func的拷贝分配内存，将抛出std::bad_alloc异常；如果线程库内发生任何其他错误，将抛出boost::thread_restore_error异常。

1.3.3.1. 类 this_thread::disable_interruption

```
class disable_interruption
{
public:
    disable_interruption();
    ~disable_interruption();
};
```

类boost::this_thread::disable_interruption对象在被构造后对当前线程禁用中断，并在析构后恢复到先前的中断状态。disable_interruption的实例不允许复制或者移动。

构造函数

```
disable_interruption();
```

效果：存储boost::this_thread::interruption_enabled()的当前状态，并对当前线程禁用中断。

后件：当前线程的boost::this_thread::interruption_enabled()返回false。

析构函数

```
~disable_interruption();
```

前件：必须从*this被构造的线程中调用。

效果：恢复boost::this_thread::interruption_enabled()的当前状态到构造*this之前的状态。

后件：当前线程的boost::this_thread::interruption_enabled()返回在*this构造时存储的值。

1.3.3.2. 类 this_thread::restore_interruption

```
class restore_interruption
{
```

```
public:
    explicit restore_interruption(disable_interruption& disabler);
    ~restore_interruption();
};
```

类boost::this_thread:: restore_interruption实例在被构造后，当前线程的中断状态被恢复到传入参数boost::this_thread::disable_interruption实例存储的中断状态。当实例被销毁时，中断再次被禁用。restore_interruption实例不能够被复制或者移动。

构造函数

```
explicit restore_interruption(disable_interruption& disabler);
```

前件：必须从disabler被构造的线程中调用。

效果：恢复当前线程的boost::this_thread::interruption_enabled()的当前状态为disabler构造之前的状态。

后件：当前线程的boost::this_thread::interruption_enabled()返回存储在disabler中的值。

析构函数

```
~ restore_interruption ();
```

前件：必须从*this被构造的线程中调用。

效果：对当前线程禁用中断。

后件：当前线程的boost::this_thread::interruption_enabled()返回false。

1.3.4. 类 thread_group

```
class thread_group: private noncopyable
{
public:
    thread_group();
    ~thread_group();

    thread* create_thread(const function0<void>& threadfunc);
    void add_thread(thread* thrd);
    void remove_thread(thread* thrd);
    void join_all();
    void interrupt_all();
    int size() const;
};
```

thread_group用于处理一组相关线程。可以使用add_thread和create_thread成员函数把新线程添加到该组中。thread_group是不可复制和移动的。

构造函数

```
thread_group();
```

效果：创建一个空的线程组。

析构函数

```
~thread_group();
```

效果：销毁*this并删除组中的所有boost::thread对象。

成员函数 create_thread

```
thread* create_thread(const function0<void>& threadfunc);
```

效果：创建一个新的boost::thread对象，就像new thread(threadfunc)一样，并把它加入到组中。

后件：this->size()增加1，新线程正在运行。

返回：新创建的boost::thread对象的指针。

成员函数 add_thread

```
void add_thread(thread* thrd);
```

前件：表达式delete thrd是良好的，不会导致未定义行为。

效果：取得thrd所指boost::thread对象的所有权，并把它加入到组中。

后件：this->size()增加1。

成员函数 remove_thread

```
void remove_thread(thread* thrd);
```

效果：如果thrd是组中成员，则从组中移除它且不调用delete。

后件：如果thrd是组中成员，则this->size()减少1。

成员函数 join_all

```
void join_all();
```

效果：为组中每个boost::thread对象调用join()。

后件：组中所有线程都已结束。

备注：由于join()是一个预定义中断点，所以join_all()也是一个中断点。

成员函数 interrupt_all

```
void interrupt_all();
```

效果：为组中每个boost::thread对象调用interrupt()。

成员函数 size

```
int size() const;
```

返回：组中的线程数。

1.4. 同步

1.4.1. 互斥概型

互斥对象有助于从数据竞争中得到保护，允许线程间线程安全的数据同步。线程可以通过调用锁函数来获取互斥对象的所有权，并调用对应的解锁函数来放弃所有权。互斥可是是递归和非递归的，并且可以同时把所有权授予给一个或多个线程。Boost.Thread提供具有排他所有权语义的递归和非递归的互斥，以及一个共享所有权（多读者/单写者）的互斥。

Boost.Thread为可锁对象提供了四个基本概型：Lockable、TimedLockable、SharedLockable和UpgradeLockable。每一个互斥类型实现了这些概型的一个或多个，就像不同锁类型所做的一样。

1.4.1.1. Lockable 概型

```
void lock()
bool try_lock()
void unlock()
```

Lockable概型建模了排他所有权。实现了Lockable概型的类型应该提供上面的三个成员函数。通过lock()或者try_lock()调用获得的锁的所有权必须通过unlock()调用释放。

void lock()

效果：当前线程阻塞，直到当前线程获得*this的所有权。

后件：当前线程拥有*this。

抛出：如果发生错误，将抛出boost::thread_resource_error异常。

bool try_lock()

效果：尝试非阻塞地为当前线程获取所有权。

返回：如果获得所有权，则返回true；否则返回false。

后件：如果调用返回true，当前线程拥有*this。

抛出：如果发生错误，将抛出boost::thread_resource_error异常。

void unlock()

前件：当前线程拥有*this。

效果：释放当前线程的所有权。

后件：当前线程不再拥有*this。

1.4.1.2. TimedLockable 概型

```
bool timed_lock(boost::system_time const& abs_time)
```

```
template <typename DurationType>
bool timed_lock(DurationType const& rel_time)
```

TimedLockable概型提炼了Lockable概型，增添了在试图获取锁时的超时支持。实现了TimedLockable概型的类型应该满足Lockable概型需求，另外还必须提供上面的两个成员函数。通过timed_lock调用获得的锁的所有权必须通过unlock()调用释放。

bool timed_lock(boost::system_time const &abs_time)

效果：尝试为当前线程获取所有权。调用阻塞，直到获得所有权或者指定的时间到达。如果指定的时间已经过去，行为和try_lock()一样。

返回：如果获得所有权，则返回true；否则返回false。

后件：如果返回true，则当前线程拥有*this。

抛出：如果发生错误，将抛出boost::thread_resource_error异常。

template <typename DurationType> bool timed_lock(DurationType const &rel_time)

效果：就像timed_lock(boost::get_system_time() + rel_time)。

1.4.1.3. SharedLockable 概型

```
void lock_shared()
bool try_lock_shared()
bool timed_lock_shared(boost::system_time const& abs_time)
void unlock_shared()
```

SharedLockable概型是TimedLockable概型的提炼，其除了允许排他所有权外，还允许共享所有权。这是标准的多读者/单写者模型：至多一个线程可以拥有互斥所有权，并且如果任何一个线程拥有排他所有权，那么没有其它线程可以拥有共享或者排他所有权。另外，多个线程可以同时拥有共享所有权。

对于要实现SharedLockable概型的类型，除了满足TimedLockable概型外，还必须提供上面的四个成员函数。通过lock_shared()、try_lock_shared()或者timed_lock_shared()调用获得的锁的所有权必须通过unlock_shared()调用释放。

void lock_shared()

效果：当前线程阻塞，直到为当前线程获得共享所有权。

后件：当前线程拥有*this的共享所有权。

抛出：如果发生错误，抛出boost::thread_resource_error异常。

bool try_lock_shared()

效果：尝试非阻塞地为当前线程获取共享所有权。

返回：如果获得共享所有权，则返回true；否则返回false。

后件：如果调用返回true，则当前线程拥有*this的共享所有权。

抛出：如果发生错误，则抛出boost::thread_resource_error异常。

bool timed_lock_shared(boost::system_time const &abs_time)

效果：尝试为当前线程获取共享所有权。调用阻塞，直到获得共享所有权或者指定的时间到达。如果指定的时间已经过去，行为同try_lock_shared()。

返回：如果为当前线程获得了共享所有权，则返回true；否则返回false。

后件：如果调用返回true，则当前线程拥有*this的共享所有权。

抛出：如果发生错误，则抛出boost::thread_resource_error异常。

void unlock_shared()

前件：当前线程拥有*this的共享所有权。

效果：释放当前线程对*this的共享所有权。

后件：当前线程不再拥有*this的共享所有权。

1.4.1.4. UpgradeLockable 概型

void lock_upgrade()
void unlock_upgrade()
void unlock_upgrade_and_lock()
void unlock_upgrade_and_lock_shared()
void unlock_and_lock_upgrade()

UpgradeLockable概型是SharedLockable概型的提炼，其除了允许共享所有权和排他所有权外，还允许提升所有权。这是对由SharedLockable概型提出的多读者/单写者模型的扩展：一个线程可以拥有提升所有权，同时其他线程可以拥有共享所有权。拥有提升所有权的线程随时可以把所有权提升为互斥所有权。如果没有其他线程拥有共享所有权，那么该提升立即完成，线程进而拥有排他所有权，其必须通过unlock()调用释放，就像是通过lock()调用获得的。

如果拥有提升所有权的线程试图提升，同时其他线程拥有共享所有权，那么该尝试将失败，线程阻塞直到排他所有权可以被获得。

所有权除了可以提升外，还可以降级： UpgradeLockable概型实现的排他所有权可以降级为提升所有权或者共享所有权，提升所有权可以降级为普通的共享所有权。

对于要实现UpgradeLockable概型的类型，除了满足SharedLockable概型的需求外，还必须提供上面的五个成员函数。

通过lock_upgrade()调用获得的锁的所有权必须通过unlock_upgrade()调用释放。如果所有权类型通过任一unlock_xxx_and_lock_yyy()函数调用发生了改变，那么所有权必须通过对应于新所有权的解锁函数调用来释放。

void lock_upgrade()

效果：当前线程阻塞，直到为当前线程获得提升所有权。

后件：当前线程拥有*this的提升所有权。

抛出：如果发生错误，则抛出boost::thread_resource_error异常。

void unlock_upgrade()

前件：当前线程拥有*this的提升所有权。

效果：释放当前线程对*this的提升所有权。

后件：当前线程不再拥有*this的提升所有权。

void unlock_upgrade_and_lock()

前件：当前线程拥有*this的提升所有权。

效果：原子地释放当前线程对*this的提升所有权，并获得*this的排他所有权。如果任何其他线程拥有共享所有权，调用阻塞直到排他所有权可以获得。

后件：当前线程拥有*this的排他所有权。

void unlock_upgrade_and_lock_shared()

前件：当前线程拥有*this的提升所有权。

效果：原子地释放当前线程对*this的提升所有权，并获得*this的共享所有权（不会阻塞）。

后件：当前线程拥有*this的共享所有权。

void unlock_and_lock_upgrade()

前件：当前线程拥有*this的排他所有权。

效果：原子地释放当前线程对*this的排他所有权，并获得*this的提升所有权（不会阻塞）。

后件：当前线程拥有*this的提升所有权。

1.4.2. 锁类型

类模板 lock_guard
类模板 unique_lock
类模板 shared_lock
类模板 upgrade_lock
类模板 upgrade_to_unique_lock

1.4.2.1. 类模板 lock_guard

```
template<typename Lockable>
class lock_guard
{
public:
    explicit lock_guard(Lockable& m);
    lock_guard(Lockable& m, boost::adopt_lock_t);
```

```
    ~lock_guard();
};
```

boost::lock_guard非常简单：在构造时，它获取作为模板参数的Lockable概型实现的所有权。在析构时，释放所有权。这里提供了简单的RAII风格的对Lockable对象的加锁，有助于异常安全的加锁和解锁。另外，lock_guard(Lockable &m, boost::adopt_lock_t)构造函数允许boost::lock_guard对象取得当前线程已经拥有的锁的所有权。

lock_guard(Lockable &m)

效果：存储m的引用，并调用m.lock()。

抛出：m.lock()调用中抛出的任何异常。

lock_guard(Lockable &m, boost::adopt_lock_t)

前件：当前线程拥有m上的锁，等价于通过m.lock()调用获得的锁。

效果：存储m的引用，取得m的锁状态的所有权。

~lock_guard()

效果：对传给构造函数的Lockable对象调用m.unlock()。

1.4.2.2. 类模板 unique_lock

```
template<typename Lockable>
class unique_lock
{
public:
    explicit unique_lock(Lockable& m_);
    unique_lock(Lockable& m_,adopt_lock_t);
    unique_lock(Lockable& m_,defer_lock_t);
    unique_lock(Lockable& m_,try_to_lock_t);
    unique_lock(Lockable& m_,system_time const& target_time);

    ~unique_lock();

    unique_lock(detail::thread_move_t<unique_lock<Lockable> > other);
    unique_lock(detail::thread_move_t<upgrade_lock<Lockable> > other);

    operator detail::thread_move_t<unique_lock<Lockable> >();
    detail::thread_move_t<unique_lock<Lockable> > move();
    unique_lock& operator=(detail::thread_move_t<unique_lock<Lockable> > other);
    unique_lock& operator=(detail::thread_move_t<upgrade_lock<Lockable> > other);

    void swap(unique_lock& other);
    void swap(detail::thread_move_t<unique_lock<Lockable> > other);
```

```
    void lock();
    bool try_lock();

    template<typename TimeDuration>
    bool timed_lock(TimeDuration const& relative_time);
    bool timed_lock(::boost::system_time const& absolute_time);

    void unlock();

    bool owns_lock() const;
    operator unspecified-bool-type() const;
    bool operator!() const;

    Lockable* mutex() const;
    Lockable* release();
};
```

boost::unique_lock比boost::lock_guard复杂得多：它不仅提供了RAII风格的加锁，还允许延迟获取锁，直到lock()成员函数被显式调用，或者以非阻塞方式尝试获取锁，或者使用超时机制。因此，仅当锁对象已经锁住了Lockable对象，或者采用了一个Lockable对象的锁，才在析构函数中调用unlock()。

如果提供的Lockable类型本身建模了TimedLockable概型，则boost::unique_lock的特化也建模了TimedLockable概型（例如boost::unique_lock<boost::timed_mutex>），或者其他Lockable概型（例如boost::unique_lock<boost::mutex>）。

如果mutext()返回指向Lockable对象m的指针且owns_lock()返回true，那么我们说boost::unique_lock实例拥有Lockable对象m的锁状态。如果拥有一个Lockable对象的锁状态的对象被销毁，那么析构函数将调用mutext()->unlock()。

boost::unique_lock的成员函数不是线程安全的。特别地，boost::unique_lock有意于对一个特定线程的Lockabel对象的所有权建模，释放锁状态所有权的成员函数（包括析构函数）必须在获取锁状态所有权的同一线程中调用。

unique_lock(Lockable &m)

效果：存储m的引用，并调用m.lock()。

后件：owns_lock()返回true，mutext()返回&m。

抛出：m.lock()调用中抛出的任何异常。

unique_lock(Lockable &m, boost::adopt_lock_t)

前件：当前线程拥有m上的排他锁。

效果：存储m的引用，并获得m的锁状态的所有权。

后件：owns_lock()返回true，mutex()返回&m。

unique_lock(Lockable &m, boost::defer_lock_t)

效果：存储m的引用。

后件：owns_lock()返回false，mutex()返回&m。

unique_lock(Lockable &m, boost::try_to_lock_t)

效果：存储m的引用；调用m.try_lock()，如果调用返回true，则获得锁状态的所有权。

后件：mutex()返回&m。如果try_lock()调用返回true，那么owns_lock()返回true，否则owns_lock()返回false。

unique_lock(Lockable &m, boost::system_time const &abs_time)

效果：存储m的引用；调用m.timed_lock(abs_time)，并且如果调用返回true，则获得锁状态的所有权。

后件：mutex()返回&m。如果timed_lock()调用返回true，那么owns_lockI()返回true，否则owns_lock()返回false。

抛出：m.timed_lock(abs_time)调用中的任何异常。

~unique_lock()

效果：如果owns_lock()返回true，则调用mutex()->unlock()。

bool owns_lock() const

返回：如果*this拥有与*this关联的Lockable对象的锁，那么返回true。

Lockable* mutex() const

返回：与*this关联的Lockable对象指针，如果没有此对象则返回NULL。

operator unspecified-bool-type() const

返回：如果owns_lock()返回true，则返回一个在布尔上下文中计算结果为true的值；否则返回一个在布尔上下文中计算结果为false的值。

bool operator! () const

返回：!owns_lock()

Lockable* release()

效果：移除*this和Lockable对象之间的关联，不会影响Lockable对象的锁状态。如果之前owns_lock()返回true，那么由调用代码负责确保该Lockable对象被正确地解锁。

返回：与*this关联的Lockable对象的指针，如果没有此对象则返回NULL。

后件：*this不再与任何Lockable对象关联。mutex()返回NULL，owns_lock()返回false。

1.4.2.3. 类模板 shared_lock

```
template<typename Lockable>
class shared_lock
```

```
{
public:
    explicit shared_lock(Lockable& m_);
    shared_lock(Lockable& m_,adopt_lock_t);
    shared_lock(Lockable& m_,defer_lock_t);
    shared_lock(Lockable& m_,try_to_lock_t);
    shared_lock(Lockable& m_,system_time const& target_time);
    shared_lock(detail::thread_move_t<shared_lock<Lockable> > other);
    shared_lock(detail::thread_move_t<unique_lock<Lockable> > other);
    shared_lock(detail::thread_move_t<upgrade_lock<Lockable> > other);

    ~shared_lock();

    operator detail::thread_move_t<shared_lock<Lockable> >();
    detail::thread_move_t<shared_lock<Lockable> > move();

    shared_lock& operator=(detail::thread_move_t<shared_lock<Lockable> > other);
    shared_lock& operator=(detail::thread_move_t<unique_lock<Lockable> > other);
    shared_lock& operator=(detail::thread_move_t<upgrade_lock<Lockable> > other);
    void swap(shared_lock& other);

    void lock();
    bool try_lock();
    bool timed_lock(boost::system_time const& target_time);
    void unlock();

    operator unspecified-bool-type() const;
    bool operator!() const;
    bool owns_lock() const;
};
```

跟boost::unique_lock一样，boost::shared_lock建模了Lockable概型，但是boost::shared_lock实例上的加锁不是获取提供的Locakble对象的唯一所有权，而是获取共享所有权。

跟boost::unique_lock一样，它不仅提供了RAII风格的加锁，还允许延迟获取锁，直到lock()成员函数被显示调用，或者以非阻塞方式尝试获取锁，或者使用超时机制。因此，仅当锁对象已经对Lockable对象加锁，或者采用了一个Lockable对象上的锁，才需要在析构函数中调用unlock()。

如果mutex()返回Lockable对象m的指针且owns_lock()返回true，那么我们说boost::shared_lock实例拥有m的锁状态。如果一个拥有Lockable对象的锁状态的对象被销毁，那么析构函数将调用mutex()->unlock_shared()。

boost::shared_lock的成员函数不是线程安全的。特别地，boost::shared_lock有意于对特定线程的Lockable对象的共享所有权建模，释放锁状态所有权的成员函数（包括析构函数）必须在获取锁状态所有权的同一线程中调用。

shared_lock(Lockable &m)

效果：存储m的引用，调用m.lock_shared()。

后件：owns_lock()返回true，mutex()返回&m。

抛出：m.lock_shared()调用中的任何异常。

shared_lock(Lockable &m, boost::adopt_lock_t)

前件：当前线程拥有m上的排他锁（译注：为什么不是共享锁？）。

效果：存储m的引用，获得m的锁状态的所有权。

后件：owns_lock()返回true，mutex()返回&m。

shared_lock(Lockable &m, boost::defer_lock_t)

效果：存储m的引用。

后件：owns_lock()返回false，mutex()返回&m。

shared_lock(Lockable &m, boost::try_to_lock_t)

效果：存储m的引用。调用m.try_lock_shared()，如果调用返回true，则获得锁状态的所有权。

后件：mutex()返回&m。如果try_lock_shared()调用返回true，那么owns_lock()返回true，否则owns_lock()返回false。

shared_lock(Lockable &m, boost::system_time const &abs_time)

效果：存储m的引用。调用m.timed_lock_shared(abs_time)，如果调用返回true，则获得锁状态的所有权。（译注：原文是timed_lock）

后件：mutex()返回&m。如果timed_lock_shared()调用返回true，那么owns_lock()返回true，否则owns_lock()返回false。

抛出：m.timed_lock_shared(abs_time)调用中抛出的任何异常。（译注：原文是timed_lock）

~shared_lock()

效果：如果owns_lock()返回true，则调用mutex()->unlock_shared()。

bool owns_lock() const

返回：如果*this拥有与之关联的Lockable对象上的锁，则返回true；否则返回false。

Lockable* mutex() const

返回：与*this关联的Lockable对象的指针，如果无此对象则返回NULL。

operator unspecified-bool-type() const

返回：如果owns_lock()返回true，那么返回一个在布尔上下文中计算结果为true的值；否则返回一个在布尔上下文中计算结果为false的值。

bool operator !() const

返回：!owns_lock()。

Lockable* release()

效果：*this和Lockable对象之间的关联被移除，不会影响Lockable对象的锁状态。如果之前owns_lock()返回true，那么调用代码负责确保该Lockable对象被正确地解锁。

返回：与*this关联的Lockable对象的指针，如果无此对象则返回NULL。

后件：*this不再与任何Lockable对象关联。mutex()返回NULL，owns_lock()返回false。

1.4.2.4. 类模板 upgrade_lock

```
template<typename Lockable>
class upgrade_lock
{
public:
    explicit upgrade_lock(Lockable& m_);

    upgrade_lock(detail::thread_move_t<upgrade_lock<Lockable> > other);
    upgrade_lock(detail::thread_move_t<unique_lock<Lockable> > other);

    ~upgrade_lock();

    operator detail::thread_move_t<upgrade_lock<Lockable> >();
    detail::thread_move_t<upgrade_lock<Lockable> > move();

    upgrade_lock& operator=(detail::thread_move_t<upgrade_lock<Lockable> > other);
    upgrade_lock& operator=(detail::thread_move_t<unique_lock<Lockable> > other);

    void swap(upgrade_lock& other);

    void lock();
    void unlock();

    operator unspecified-bool-type() const;
    bool operator!() const;
    bool owns_lock() const;
};
```

跟boost::unique_lock一样，boost::upgrade_lock建模了Lockable概型，但boost::upgrade_lock实例上的加锁不是获取Lockable对象的唯一所有权，而是获取提升所有权。

跟boost::unique_lock一样，它不仅提供了RAII风格的加锁，还允许推迟获取锁，直到显式调用lock()成员函数，或者尝试以非阻塞方式获取锁，或者使用超时机制。因此，仅当锁对象已经对Lockable对象加锁或者采用了Lockable对象上的锁，才在析构函数中调用unlock()。

如果mutex()返回Lockable对象m的指针且owns_lock()返回true，那么我们说boost::upgrade_lock实例拥有m的锁状态。如果一个拥有Lockable对象的锁状态的对象被销毁，那么其析构函数将调用

mutex()->unlock_upgrade()。

boost::upgrade_lock的成员函数不是线程安全的。特别地，boost::upgrade_lock有意于建模一个特定的线程的Lockable对象的提升所有权，释放锁状态所有权的成员函数（包括析构函数）必须由获得锁状态所有权的同一线程所调用。

1.4.2.5. 类模板 upgrade_to_unique_lock

```
template <class Lockable>
class upgrade_to_unique_lock
{
public:
    explicit upgrade_to_unique_lock(upgrade_lock<Lockable>& m_);

    ~upgrade_to_unique_lock();

    upgrade_to_unique_lock(detail::thread_move_t<upgrade_to_unique_lock<Lockable> >
other);
    upgrade_to_unique_lock&
operator=(detail::thread_move_t<upgrade_to_unique_lock<Lockable> > other);
    void swap(upgrade_to_unique_lock& other);

    operator unspecified-bool-type() const;
    bool operator!() const;
    bool owns_lock() const;
};
```

boost::upgrade_to_unique_lock允许boost::upgrade_lock到排他所有权的临时提升。当利用boost::upgrade_lock实例的引用构造时，如果那个实例拥有某个Lockable对象上的提升所有权，那么所有权被提升为排他所有权。当boost::upgrade_to_unique_lock实例被销毁时，Lockable的所有权被降级为原来的提升所有权。

1.4.3. 互斥类型

```
类 mutex
类型别名 try_mutex
类 timed_mutex
类 recursive_mutex
类型别名 recursive_try_mutex
类 recursive_timed_mutex
类 shared_mutex
```

1.4.3.1. 类mutex

```
class mutex: boost::noncopyable
```

```
{
public:
    mutex();
    ~mutex();

    void lock();
    bool try_lock();
    void unlock();

    typedef unique_lock<mutex> scoped_lock;
    typedef scoped_lock scoped_try_lock;
};
```

boost::mutex实现了Lockable概型，提供一个排他所有权的互斥类型。在任意时刻，至多一个线程能够拥有给定的boost::mutex实例上的锁。对lock()、try_lock()和unlock()的多个并发调用是允许的。

1.4.3.2. 类型别名 try_mutex

```
typedef mutex try_mutex;
```

boost::try_mutex是boost::mutex的类型别名，以兼容boost的早期发行版本。

1.4.3.3. 类 timed_mutex

```
class timed_mutex: boost::noncopyable
{
public:
    timed_mutex();
    ~timed_mutex();

    void lock();
    void unlock();
    bool try_lock();
    bool timed_lock(system_time const & abs_time);

    template<typename TimeDuration>
    bool timed_lock(TimeDuration const & relative_time);

    typedef unique_lock<timed_mutex> scoped_timed_lock;
    typedef scoped_timed_lock scoped_try_lock;
    typedef scoped_timed_lock scoped_lock;
};
```

boost::timed_mutex实现了TimedLockable概型，提供一个排他所有权的互斥类型。在任意时刻，至多一个线程能够拥有给定的boost::timed_mutex实例上的锁。对lock()、try_lock()、timed_lock()和unlock()的多个并发调用是允许的。

1.4.3.4. 类 recursive_mutex

```
class recursive_mutex: boost::noncopyable
{
public:
    recursive_mutex();
    ~recursive_mutex();

    void lock();
    bool try_lock();
    void unlock();

    typedef unique_lock<recursive_mutex> scoped_lock;
    typedef scoped_lock scoped_try_lock;
};
```

boost::recursive_mutex实现了Lockable概型，提供一个排他所有权的递归互斥类型。在任意时刻，至多一个线程能够拥有给定的boost::recursive_mutex实例上的锁。对lock()、try_lock()和unlock()的多个并发调用是允许的。一个已经拥有给定的boost::recursive_mutex实例上的排他所有权的线程可以调用lock()或try_lock()来获取该互斥对象的其它层级的所有权。在其它线程能够获得所有权之前，必须为线程所获得的所有权的每一层级调用一次unlock()。

1.4.3.5. 类型别名 recursive_try_mutex

```
typedef recursive_mutex recursive_try_mutex;
```

boost::recursive_try_mutex是boost::recursive_mutex的类型别名，以兼容boost的早期发行版本。

1.4.3.6. 类 recursive_timed_mutex

```
class recursive_timed_mutex: boost::noncopyable
{
public:
    recursive_timed_mutex();
    ~recursive_timed_mutex();

    void lock();
    bool try_lock();
    void unlock();

    bool timed_lock(system_time const & abs_time);

    template<typename TimeDuration>
    bool timed_lock(TimeDuration const & relative_time);
```

```
    typedef unique_lock<recursive_timed_mutex> scoped_lock;
    typedef scoped_lock scoped_try_lock;
    typedef scoped_lock scoped_timed_lock;
};
```

boost::recursive_timed_mutex实现了TimedLockable概型，提供一个排他所有权的递归互斥类型。在任意时刻，至多一个线程能够拥有给定的boost::recursive_timed_mutex实例上的锁。对lock()、try_lock()、timed_lock()和unlock()的多个并发调用是允许的。一个已经拥有给定的boost::recursive_timed_mutex实例上的排他所有权的线程可以调用lock()、timed_lock()或者try_lock()来获取该互斥对象的其它层级的所有权。在其它线程能够获得所有权之前，必须为线程所获得的所有权的每一层级调用一次unlock()。

1.4.3.7. 类 shared_mutex

```
class shared_mutex
{
public:
    shared_mutex();
    ~shared_mutex();

    void lock_shared();
    bool try_lock_shared();
    bool timed_lock_shared(system_time const& timeout);
    void unlock_shared();

    void lock();
    bool try_lock();
    bool timed_lock(system_time const& timeout);
    void unlock();

    void lock_upgrade();
    void unlock_upgrade();

    void unlock_upgrade_and_lock();
    void unlock_and_lock_upgrade();
    void unlock_and_lock_shared();
    void unlock_upgrade_and_lock_shared();
};
```

类boost::shared_mutex提供了一个多读者/单写者实现的互斥类型，实现了UpgradeLockable概型。对lock()、try_lock()、timed_lock()、lock_shared()、try_lock_shared()和timed_lock_shared()的多个并发调用是允许的。

1.4.4. 条件变量

类 condition_variable 类 condition_variable_any 类型别名 condition

1.4.4.1. 概要

类condition_variable和condition_variable_any提供了一个线程等待来自其它线程在特定条件为真时的通知的机制。一般用法是，一个线程锁定一个互斥对象，然后在一个condition_variable或者condition_variable_any的实例上调用wait。当线程从等待状态中被唤醒时，它检查特定的条件是否为真，如果为真则继续执行。如果条件不为真，则线程再次调用wait进入等待状态。在最简单的情形中，该条件是一个布尔变量。例如：

```
boost::condition_variable cond;
boost::mutex mut;
bool data_ready;

void process_data();

void wait_for_data_to_process()
{
    boost::unique_lock<boost::mutex> lock(mut);
    while(!data_ready)
    {
        cond.wait(lock);
    }
    process_data();
}
```

注意，上面例子中lock被传递给wait：wait将自动地添加该线程到等待该条件变量的线程集中，并解锁该互斥对象。当线程被唤醒时，互斥对象将在wait调用返回前被再次锁定。这允许其它线程获取该互斥对象以更新共享数据，并确保与该条件关联的数据被正确地同步。

同时，另一个线程设置条件为真，然后对该条件变量调用notify_one或者notify_all来唤醒一个或所有等待线程。例如：

```
void retrieve_data();
void prepare_data();

void prepare_data_for_processing()
{
    retrieve_data();
    prepare_data();
    {
        boost::lock_guard<boost::mutex> lock(mut);
```

```
            data_ready=true;
        }
        cond.notify_one();
}
```

注意，在上面例子中，在共享数据被更新之前同一个互斥对象被锁定，但是在调用notify_one时不需要锁定该互斥对象。

这个例子使用了condition_variable类型的对象，但使用condition_variable_any同样也能很好地工作：condition_variable_any更加通用，可以与任意类型的锁或者互斥对象一起工作，而condition_variable要求传递给wait的锁是boost::unique_lock<boost::mutex>的实例。这可以使condition_variable基于对互斥类型的了解在某些方面做些优化；典型地，condition_variable_any比condition_variable拥有一个更加复杂的实现。

1.4.4.2. 类 condition_variable

```
class boost::condition_variable
{
public:
    condition_variable();
    ~condition_variable();

    void wait(boost::unique_lock<boost::mutex>& lock);

    template<typename predicate_type>
    void wait(boost::unique_lock<boost::mutex>& lock,predicate_type predicate);

    bool timed_wait(boost::unique_lock<boost::mutex>& lock,boost::system_time const&
abs_time);

    template<typename duration_type>
    bool timed_wait(boost::unique_lock<boost::mutex>& lock,duration_type const&
rel_time);

    template<typename predicate_type>
    bool timed_wait(boost::unique_lock<boost::mutex>& lock,boost::system_time const&
abs_time,predicate_type predicate);

    template<typename duration_type,typename predicate_type>
    bool timed_wait(boost::unique_lock<boost::mutex>& lock,duration_type const&
rel_time,predicate_type predicate);

// backwards compatibility

    bool timed_wait(boost::unique_lock<boost::mutex>& lock,boost::xtime const&
```

```
abs_time);

    template<typename predicate_type>
    bool timed_wait(boost::unique_lock<boost::mutex>& lock,boost::xtime const&
abs_time,predicate_type predicate);
};
```

condition_varialbe()

效果：构造一个类condition_variable的对象。

抛出：如果发生错误，抛出boost::thread_resource_error错误。

~condition_variable()

前件：已经通过调用notify_one或者notify_all通知了所有等待于*this上的线程（虽然相应的wait或者timed_wait调用不一定已经返回）。

效果：销毁该对象。

void notify_one()

效果：如果有任何线程当前因调用wait或者timed_wait等待*this而阻塞，那么对其中一个线程解除阻塞。

void notify_all()

效果：如果有任何线程当前因调用wait或者timed_wait等待*this而阻塞，那么对所有这些线程解除阻塞。

void wait(boost::unique_lock<boost::mutex> &lock)

前件：当前线程锁定了lock，并且，要么当前没有其它线程等待*this，要么当前所有等待*this的线程在调用wait或者timed_wait时用的锁对象的成员函数mutex()会返回和本次wait调用的lock->mutex()相同的值。

效果：自动调用lock.unlock()并阻塞当前线程。在被this->notify_one()和this->notify_all()调用通知时，或者spuriously，线程将解除阻塞。当线程被解除阻塞时（无论出于什么原因），在wait()调用返回前会通过调用lock.lock()获得锁。如果函数异常退出，同样会调用lock.lock()获得锁。

后件：当前线程锁定了lock。

抛出：如果发生错误，抛出boost::thread_resource_error异常。如果wait被与当前执行线程关联的boost::thread对象上的interrupt()调用中断，抛出boost::thread_interrupted异常。

template <typename predicate_type>
void wait(boost::unique_lock<boost::mutex> &lock, predicate_type pred)

效果：等价于

```
while (!pred())
{
    wait(lock);
```

```
}
```

bool timed_wait(boost::unique_lock<boost::mutex&> &lock, boost::system_time const &abs_time)

前件：当前线程锁定了lock，并且，要么当前没有其它线程等待*this，要么当前所有等待*this的线程在调用wait或者timed_wait时用的锁对象的成员函数mutex()会返回和本次wait调用的lock->mutex()相同的值。

效果：自动调用lock.unlock()，并阻塞当前线程。在被this->notify_one()和this->notify_all()调用通知时，或者当由boost::get_system_time()报告的时间等于或者晚于指定的abs_time时，或者spuriously，线程将解除阻塞。当线程被解除阻塞时（无论出于什么原因），在wait()调用返回前会通过调用lock.lock()获得锁。如果函数异常退出，同样会调用lock.lock()获得锁。

返回：如果调用因达到指定的时间abs_time而超时返回，则返回false；否则返回true。

后件：当前线程锁定lock。

抛出：如果发生错误，抛出boost::thread_resource_error异常。如果wait被与当前执行线程关联的boost::thread对象上的interrupt()调用中断，抛出boost::thread_interrupted异常。

template <typename duration_type>
bool timed_wait(boost::unique_lock<boost::mutex> &lock, duration_type const &rel_time)

前件：当前线程锁定了lock，并且，要么当前没有其它线程等待*this，要么当前所有等待*this的线程在调用wait或者timed_wait时用的锁对象的成员函数mutex()会返回和本次wait调用的lock->mutex()相同的值。

效果：自动调用lock.unlock()，并阻塞当前线程。在被this->notify_one()和this->notify_all()调用通知时，或者在由rel_time参数指定的时间段已经过去时，或者spuriously，线程将解除阻塞。当线程被解除阻塞时（无论出于什么原因），在wait()调用返回前会通过调用lock.lock()获得锁。如果函数异常退出，同样会调用lock.lock()获得锁。

返回：如果调用因rel_time参数指定的时间段已经过去而返回，则返回false；否则返回true。

后件：当前线程锁定lock。

抛出：如果发生错误，抛出boost::thread_resource_error异常。如果wait被与当前执行线程关联的boost::thread对象上的interrupt()调用中断，抛出boost::thread_interrupted异常。

> 注意：timed_wait的时间段重载版本很难正确地使用。在大多数情形下，应该首选断言版本。

template <typename predicate_type> bool
timed_wait(boost::unique_lock<boost::mutex> &lock, boost::system_time const &abs_time,
predicate_type pred)

效果：等价于

```
while (!pred())
{
    if (!timed_wait(lock, abs_time))
```

```
        {
            return pred();
        }
    }
    return true;
```

1.4.4.3. 类 condition_variable_any

```
class boost::condition_variable_any
{
public:
    condition_variable_any();
    ~condition_variable_any();

    template<typename lock_type>
    void wait(lock_type& lock);

    template<typename lock_type,typename predicate_type>
    void wait(lock_type& lock,predicate_type predicate);

    template<typename lock_type>
    bool timed_wait(lock_type& lock,boost::system_time const& abs_time);

    template<typename lock_type,typename duration_type>
    bool timed_wait(lock_type& lock,duration_type const& rel_time);

    template<typename lock_type,typename predicate_type>
    bool timed_wait(lock_type& lock,boost::system_time const& abs_time,predicate_type
predicate);

    template<typename lock_type,typename duration_type,typename predicate_type>
    bool timed_wait(lock_type& lock,duration_type const& rel_time,predicate_type
predicate);

// backwards compatibility

    template<typename lock_type>
    bool timed_wait(lock_type>& lock,boost::xtime const& abs_time);

    template<typename lock_type,typename predicate_type>
    bool timed_wait(lock_type& lock,boost::xtime const& abs_time,predicate_type
predicate);
};
```

condition_variable_any()

效果：构造一个类condition_variable_any的对象。

抛出：如果发生错误，抛出boost::thread_resource_error异常。

~condition_variable_any()

前件：已经通过调用notify_one或者notify_all通知了所有等待于*this上的线程（虽然相应的wait或者timed_wait调用不一定已经返回）。

效果：销毁对象。

void notify_one()

效果：如果有任何线程当前因调用wait或者timed_wait等待*this而阻塞，那么对其中一个线程解除阻塞。

void notify_all()

效果：如果有任何线程当前因调用wait或者timed_wait等待*this而阻塞，那么对所有这些线程解除阻塞。

template <typename lock_type> void wait(lock_type &lock)

效果：自动调用lock.unlock()并阻塞当前线程。在被this->notify_one()和this->notify_all()调用通知时，或者spuriously，线程将解除阻塞。当线程被解除阻塞时（无论出于什么原因），在wait()调用返回前会通过调用lock.lock()获得锁。如果函数异常退出，同样会调用lock.lock()获得锁。（译注：这里没有condition_variable中对应的前件：调用前需要锁定，但我觉得应该是有的。下同）

后件：当前线程锁定了lock。

抛出：如果发生错误，抛出boost::thread_resource_error异常。如果wait被与当前执行线程关联的boost::thread对象上的interrupt()调用中断，抛出boost::thread_interrupted异常。

template <typename lock_type, typename predicate_type>
void wait(block_type &lock, predicate_type pred)

效果：等价于

```
while (!pred())
{
    wait(lock);
}
```

template <typename lock_type>
bool timed_wait(lock_type &lock, boost::system_time const &abs_time)

效果：自动调用lock.unlock()，并阻塞当前线程。在被this->notify_one()和this->notify_all()调用通知时，或者当由boost::get_system_time()报告的时间等于或者晚于指定的abs_time时，或者spuriously，线程将解除阻塞。当线程被解除阻塞时（无论出于什么原因），在wait()调用返回前会通过调用lock.lock()获得锁。如果函数异常退出，同样会调用lock.lock()获得锁。

返回：如果调用因达到指定的时间abs_time而超时返回，则返回false；否则返回true。

后件：当前线程锁定lock。

抛出：如果发生错误，抛出boost::thread_resource_error异常。如果wait被与当前执行线程关联的boost::thread对象上的interrupt()调用中断，抛出boost::thread_interrupted异常。

template <typename lock_type, typename duration_type>
bool timed_wait(lock_type &lock, duration_type const &rel_time)

效果：自动调用lock.unlock()，并阻塞当前线程。在被this->notify_one()和this->notify_all()调用通知时，或者在由rel_time参数指定的时间段已经过去时，或者spuriously，线程将解除阻塞。当线程被解除阻塞时（无论出于什么原因），在wait()调用返回前会通过调用lock.lock()获得锁。如果函数异常退出，同样会调用lock.lock()获得锁。

返回：如果调用因rel_time参数指定的时间段已经过去而返回，则返回false；否则返回true。

后件：当前线程锁定lock。

抛出：如果发生错误，抛出boost::thread_resource_error异常。如果wait被与当前执行线程关联的boost::thread对象上的interrupt()调用中断，抛出boost::thread_interrupted异常。

> 注意：timed_wait的时间段重载版本很难正确地使用。在大多数情形下，应该首选断言版本。

template <typename lock_type, typename predicate_type> bool
timed_wait(lock_type &lock, boost::system_time const &abs_time, predicate_type pred)

效果：等价于

```
while (!pred())
{
    if (!timed_wait(lock, abs_time))
    {
        return pred();
    }
}
return true;
```

1.4.4.4. 类型别名 condition

```
typedef condition_variable_any condition;
```

boost::condition是boost::condition_variable_any的类型别名，以兼容boost的早期发行版本。

1.4.5. 一次初始化

```
类型别名 once_flag
非成员函数 call_once
```

boost::call_once提供一种机制来确保初始化例程刚好执行一次，不会有数据竞争或者死锁。

1.4.5.1. 类型别名 once_flag

```
typedef platform-specific-type once_flag;
#define BOOST_ONCE_INIT platform-specific-initializer
```

boost::once_flag类型的对象应该使用BOOST_ONCE_INIT初始化。例如：

```
boost::once_flag f = BOOST_ONCE_INIT;
```

1.4.5.2. 非成员函数 call_once

```
template <typename Callable>
void call_once(once_flag &flag, Callable func);
```

需求：Callable是可复制构造的。复制func应该没有副作用，并且调用该拷贝的效果应该等同于调用原始对象。

效果：对同一个once_flag对象上的call_once调用被串行化。如果之前没有该once_flag对象上的有效call_once调用，参数func（或者它的拷贝）被调用，就像调用func(args)，并且仅当func(args)没有异常返回时call_once调用才是有效的。如果抛出异常，异常被传播到调用者。如果之前有该once_flag对象上的有效call_once调用，那么call_once不会调用func。

同步：对一个once_flag对象的有效call_once调用的完成会同步所有作用于同一once_flag对象上的后继call_once调用。

抛出：当不能达到效果或者func中传播出任何异常时，抛出boost::thread_resource_error异常。

```
void call_once(void (*func)(), once_flag &flag);
```

上面的重载版本是为了向后兼容。调用call_once(func, flag)的效果等同于call_once(flag, func)。

1.4.6. 屏障（barrier）

```
类 barrier
```

屏障（barrier）是一个简单的概念。也被广泛称为汇合点（rendezvous），它是多个线程间的同步点。屏障被设定用于特定数量的线程（n），当线程达到屏障时它们必须等待，直到所有n个线程都已经达到。一旦第n个线程达到屏障，那么所有等待线程可以继续处理，屏障被消除。

1.4.6.1. 类 barrier

```
#include <boost/thread/barrier.hpp>

class barrier
{
public:
    barrier(unsigned int count);
    ~barrier();
```

```
    bool wait();
};
```

boost::barrier的实例是不可复制或者移动的。

barrier(unsigned int count)

效果：为count个线程构造一个屏障。

抛出：如果发生错误，抛出boost::thread_resource_error异常。

~barrier()

前件：没有线程等待*this。

效果：销毁*this。

bool wait()

效果：阻塞，直到count个线程在*this上调用了wait。当第count个线程调用wait时，所有等待线程被解除阻塞，屏障被清除。

返回：仅有一个等待线程返回true，其它返回false。（译注：没有指明谁返回true，该返回值有何意义？）

抛出：如果发生错误，抛出boost::thread_resource_error异常。

1.5. 线程局部存储

类 thread_specific_ptr

1.5.1. 概要

线程局部存储（TLS，Thread Local Storage）允许多线程应用程序的每一个线程都拥有一份指定数据项的单独实例。在单线程应用程序中可能会使用静态或者全局数据，在多线程应用程序中这会导致争用、死锁或者数据毁坏。一个例子是C中的errno变量，其被用于存储与标准C库函数相关的错误代码。常见实践是支持多线程应用程序的编译器为每个线程提供errno的单独实例（POSIX有此要求），以避免不同的线程在读和更新该值时发生竞争。

尽管编译器通常以声明语法的扩展形式来提供这种支持（例如，静态或者名字空间作用域变量声明上的__declspec(thread)或者__thread修饰），但这种支持是不可移植的，并且经常仅限于某些情形，例如仅仅支持POD类型。

可移植的线程局部存储 boost::thread_specific_ptr

boost::thread_specific_ptr为线程局部存储提供了一种可移植的机制，能够工作于所有被Boost.Thread所支持的编译器上。boost::thread_specific_ptr的每一个实例代表一个对象（例如errno）的指针，每个线程一定有一个不同的值。当前线程的该指针值可以使用get()成员函数获得，或者通过使用*和->操作符。每个线程中该指针值初始值为NULL，但可以使用reset()成员函数为当前线程设置该值。

如果当前线程的指针值被reset()改变，那么前面的值会通过调用清理例程销毁。另外，可以通过调用release()成员函数重置存储的值为NULL，并返回先前的值，以允许应用程序负责销毁该对象。

线程退出时清理

当一个线程退出时，与每个boost::thread_specific_ptr实例关联的对象被销毁。默认情况下，由指针p指向的对象通过调用delete p销毁，但可以通过为构造函数提供一个清理例程来重载特定的boost::thread_specific_ptr实例的行为。在这种情形下，通过调用func(p)来销毁对象，其中func是提供给构造函数的清理例程。这些清理例程的调用顺序是未指定的。如果清理例程设置了与已经被清理了的boost::thread_specific_ptr实例关联的值，那么该值被添加到清理列表中（译注：这可能会引起死循环？）。当不再有未完成的有值的boost::thread_specific_ptr实例时，清理结束。

1.5.2. 类 thread_specific_ptr

```
template <typename T>
class thread_specific_ptr
{
public:
    thread_specific_ptr();
    explicit thread_specific_ptr(void (*cleanup_function)(T*));
    ~thread_specific_ptr();

    T* get() const;
    T* operator->() const;
    T& operator*() const;

    T* release();
    void reset(T* new_value=0);
};
```

thread_specific_ptr

需求：delete this->get()是良好形式的（well-formed）。

效果：构造一个thread_specific_ptr对象来存储特定于（specific to）每个线程的类型T的对象指针。当reset()被调用或者线程退出时，默认的基于delete的清理函数将被用来销毁任何线程局部对象。

抛出：如果发生错误，抛出boost::thread_resource_error异常。

explicit thread_specific_ptr(void (*cleanup_function)(T*))

需求：cleanup_function(this->get())不会抛出任何异常。

效果：构造一个thread_specific_ptr对象来存储特定于每个线程的类型T的对象指针。当reset()被调用或者线程退出时，提供的cleanup_function将被用来销毁任何线程局部对象。

抛出：如果发生错误，抛出boost::thread_resource_error异常。

~thread_specific_ptr()

效果：调用this->reset()来清理与当前线程相关的值，并销毁*this。

> **注意**：需要小心确保boost::thread_specific_ptr实例被销毁后任何仍在运行的线程不会调用该实例的任何成员函数。

T* get() const

返回：与当前线程关联的指针。

> **注意**：每个线程的与boost::thread_specific_ptr实例关联的初始值为NULL。

T* operator ->() const

返回：this->get()

T& operator * () const

需求：this->get()不为NULL。

返回：*(this->get())

void reset(T* new_value = 0)

效果：如果this->get() != new_value且this->get()不是NULL，那么调用delete this->get()或者cleanup_function(this->get())。存储new_value为与当前线程关联的指针。

后件：this->get() == new_value

抛出：如果发生错误，抛出boost::thread_resource_error异常。

T* release()

效果：返回this->get()并存储NULL为与当前线程关联的指针，不会调用清理函数。

后件：this->get() == 0

1.6. 感谢

Boost.Thread的最初实现由William kempf完成，并有许多其他人的贡献。新版最初尝试按照William Kempf的设计用新代码重写Boost.Thread，以便能够在Boost软件许可证下发布。然而，由于C++标准委员会正在积极地讨论为C++标准化一个线程库，本库也就不断演化以跟进这些建议，同时尽可能地保持向后兼容。

特别感谢Roland Schwarz，他为原始的Boost.Thread库贡献了大量的时间和代码，也积极地参与到重写中。把平台相关的实现划分到单独的目录的方案是由Roland设计的，他的投入为提高当前实现的质量做出了巨大贡献。

也要感谢Peter Dimov、Howard Hinnant、Alexander Terekhov、Chris Thomasson和其他人对代码实现细节的评论。

1.7. 翻译术语表

concept	概型（没有翻译成“概念”，自己造一个）
precondition	前件
postcondition	后件
effect	效果
exclusive lock	排他锁
barrier	屏障（点位符，实在没合适的）