Module05-08
C++ Boost：多线程

C++ Boost - 多线程

- 容器相关
- 字符串和文字处理
- 正则表达式
- 智能指针
- 函数对象相关
- 序列化
- 日期与时间
- → 多线程
- 网络

多线程 - 线程相关概念

- 多线程：
  - 线程相关概念
  - 线程管理
  - 互斥体 (Mutex)
  - 锁 (Lock)
  - 条件变量 (Condition Variable)
  - 一次性初始化操作 (One-time Initialization)
  - 栅栏 (Barriers)
  - 线程局部存储 (TSS)
  - 时间和日期相关

多线程 - 线程相关概念

- 线程相关概念

  线程概念的描述，请参考书籍《 C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns 》

  - 进程与线程 (Chapter 5 section 5.2)
  - 进程和线程的创建策略 (Chapter 5 section 5.3)
  - 用户线程和内核线程 (Chapter 5 section 5.4)
  - 线程调度级别 (Chapter 5 section 5.5)
  - 同步机制 (Chapter 6 section 6.4)

  (该书作者： Douglas C. Schmidt, Stephen D. Huston)

多线程 - 线程管理

- 多线程：
  - 线程相关概念
  - 线程管理
  - 互斥体 (Mutex)
  - 锁 (Lock)
  - 条件变量 (Condition Variable)
  - 一次性初始化操作 (One-time Initialization)
  - 栅栏 (Barriers)
  - 线程局部存储 (TSS)
  - 时间和日期相关

- 创建线程
  - 可以从一个函数对象、函数指针来创建线程

```
struct ThreadFunc {
    void operator()(string s) {
        cout << s << endl;
    }
};

void func() {
    cout << "hello, boost!" << endl;
}

void createThreads() {
    boost::thread th1(ThreadFunc(), "boost threads");
    boost::thread th2(func);

    th1.join();
    th2.join();
}
```

- 创建线程
  - 可以从一个函数对象的引用来创建线程

```
struct ThreadFunc {
    void operator()(string s) {
        cout << s << endl;
    }
};

void boost::thread createThread() {
    ThreadFunc func;
    boost::thread th1(boost::ref(func), "boost threads");

    th1.join();
}
```

多线程 - 线程管理

- 连接和分离 (Joining and detaching)
  - 当代表线程执行体的 boost::thread 对象被销毁的时候，线程执行体处于被分离的状态（ detached ）。线程执行体此时继续执行，直到线程函数或可调用对象执行结束，或者程序终止。 线程执行体也可以通过显式调用 boost::thread 成员函数 detach() 分离（ detach ）， 在这种状况下， boost::thread 对象结束了和线程执行体的关系，不再代表任何线程执行体（ Not-a-Thread ）。
  - 如果要等待一个线程结束，使用 boost::thread 线程对象的成员函数 join() 或 timed_join()。 函数 join() 会阻塞（谁调用谁阻塞）调用线程直到等待的线程对象执行结束。 如果等待的线程对象已经结束，或者等待的线程对象没有关联任何线程执行体（ Not-a-Thread ），函数 join() 会立即返回。 函数 timed_join() 与 join() 类似，只是在等待指定的时间后也会返回。

多线程 - 线程管理

- 线程中断
  - 一个正在执行的线程可以通过调用对应 boost::thread 对象的函数 interrupt() 来中断。 当一个被中断的线程在中断打开时再次执行到某个特定的中断点 interruption points，它会触发一个 boost::thread_interrupted 异常。 如果这个异常没有被捕捉，会导致该线程终止。 如同其它异常一样，栈会被展开，自动变量会被析构。
  - 如果一个线程不希望被中断，可以创建一个 boost::this_thread::disable_interruption 对象来达到这个目的。该对象创建成功后，线程中断关闭，直到该对象被销毁

- 线程中断（示例 1 ）

```
void f1(const int& id) {
    cout << "thread #" << id << ": started" << endl;
    boost::system_time const timeout =
boost::get_system_time()
            + boost::posix_time::seconds(3);
    thread::sleep(timeout);
    cout << "thread #" << id << ": ended" << endl;
}
void f2(const int& id) {
    cout << "thread #" << id << ": started" << endl;
    thread::yield();
    cout << "thread #" << id << ": ended" << endl;
}
void f3(const int& id) {
    cout << "thread #" << id << ": started" << endl;
    boost::this_thread::interruption_point();
    cout << "thread #" << id << ": ended" << endl;
}
```

多线程 - 线程管理

- 线程中断（示例 1 ）（续）

```
int main() {
    thread t1(f1, 1);
    t1.interrupt();
    thread t2(f2, 2);
    thread t3(f3, 3);
    t3.interrupt();

    t1.join(); t2.join(); t3.join();
}
```

- 线程中断（示例2）

```
void print10() {
    using namespace boost::posix_time;

    boost::this_thread::disable_interruption di;
    cout << boost::this_thread::interruption_enabled() << endl;
    cout << "thread #" << boost::this_thread::get_id() << ":";
    for (int i = 1; i < 11; ++i)
        cout << i << ' ';
    cout << endl;
    // 虽然sleep是中断点，但此处不会被打断
    boost::this_thread::sleep(seconds(1));
    cout << "print10() --> 1\n";

    boost::this_thread::restore_interruption ri(di);
    cout << boost::this_thread::interruption_enabled() << endl;
    // 此处可以被打断
    boost::this_thread::sleep(seconds(1));
    cout << "print10() --> 2\n";
}
```

多线程 - 线程管理

- 预定义中断点 (Interruption Points)
  - 下列函数是中断点（ interruption points ），会在被调用线程对象中断开启的情况下抛出 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()`

多线程 - 线程管理

- 线程 ID
  - boost::thread::id 对象可用于标识线程。 每个执行期的线程都有一个唯一的标识，可通过成员函数 get_id() 获得 ，或者在线程函数内部通过函数 boost::this_thread::get_id() 获得 。 boost::thread::id 对象支持拷贝，并提供完整的比较操作符，可以用于关联式容器的键值 。 尽管没有指定输出格式，线程标识 (Thread IDs) 还是可以以标准操作输出到流。
  - boost::thread::id 对象可能标识线程，也可能没有标识线程（ Not-a-Thread ）。没有标识线程的对象（ Not-a-Thread ）在比较时都是相同的，但是和标识了线程的 boost::thread::id 对象都不一样。 boost::thread::id 的比较操作是一致的。

多线程 - 线程管理

- class thread 接口

```
#include <boost/thread/thread.hpp>
class thread {
public:
    thread();
    ~thread();
    template<class F>
    explicit thread(F f);

    template <class F,class A1,class A2,...>
    thread(F f,A1 a1,A2 a2,...);
    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 thread 接口（续）

```
    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);
};
void swap(thread& lhs, thread& rhs);
```

多线程 - 线程管理

- Namespace this_thread

```
namespace boost {
namespace this_thread {
thread::id get_id();
void interruption_point();
bool interruption_requested();
bool interruption_enabled();
void sleep();
void yield();
class disable_interruption;
class restore_interruption;
template<typename Callable>
void at_thread_exit(Callable func);
}
}
```

多线程 - 线程管理

- class thread_group
  - thread_group 提供一组类似线程对象的集合。新的线程对象可以通过函数 add_thread 和 create_thread 加入到这个集合。thread_group 对象不支持拷贝和转移语义。

多线程 - 线程管理

- class thread_group 接口

```
#include <boost/thread/thread.hpp>
class thread_group: private noncopyable {
public:
    thread_group();
    ~thread_group();
    template<typename F>
    thread* create_thread(F threadfunc);
    void add_thread(thread* thrd);
    void remove_thread(thread* thrd);
    void join_all();
    void interrupt_all();
    int size() const;
};
```

- class thread_group 示例

```
void f1() {
    cout << "f1()" << endl;
}

void f2() {
    cout << "f2()" << endl;
}

int main() {
    boost::thread_group grp;
    for (int i = 0; i < 3; ++i)
        grp.create_thread(f1);
    grp.add_thread(new boost::thread(f2));

    cout << grp.size() << endl;
    grp.join_all();
}
```

多线程 - 同步机制

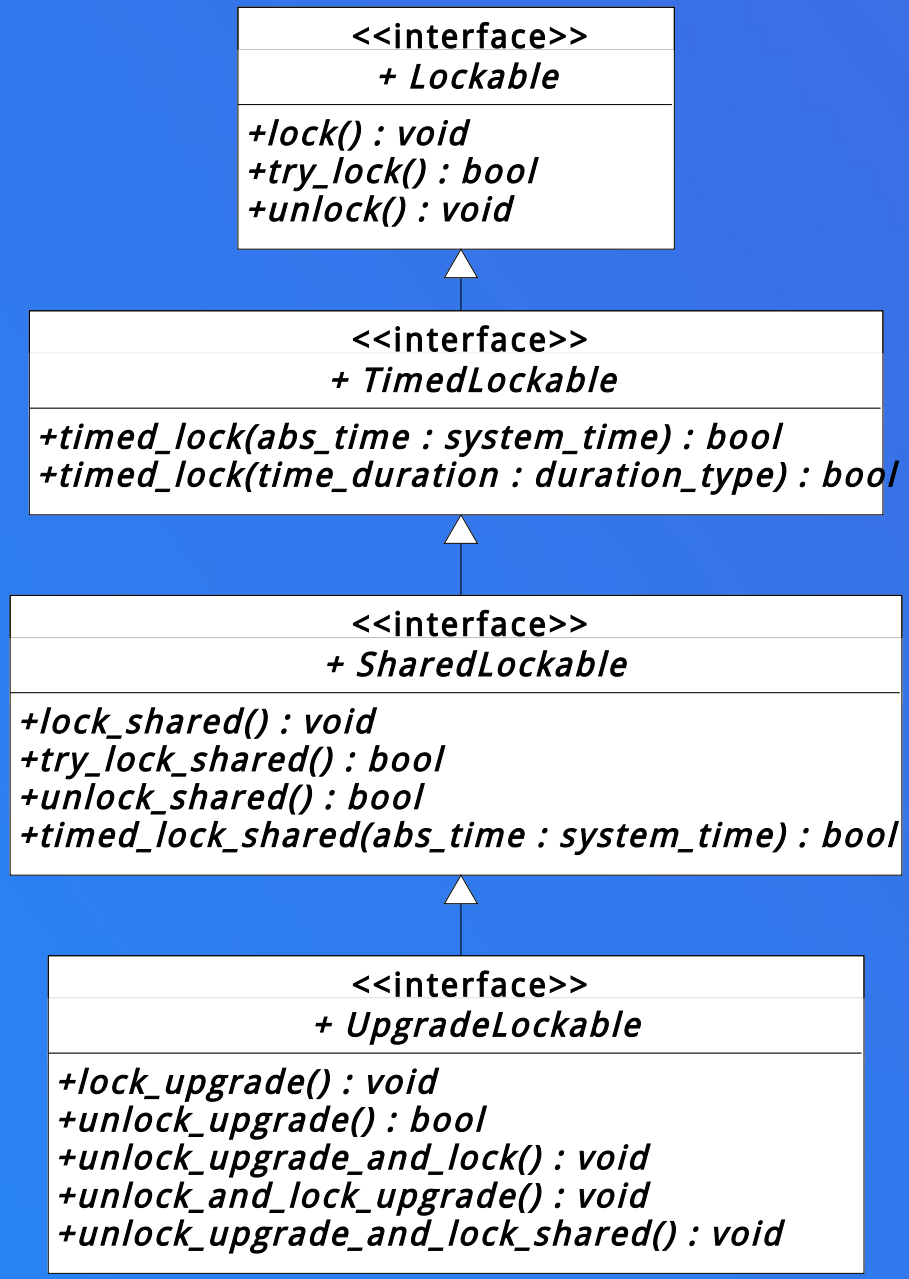
- 同步机制
  - boost thread 的同步机制，主要有以下几个基础设施支撑：
    - 互斥体 (Mutex: Mutual Exclusion)
    - 锁 (Lock)
    - 条件变量 (Condition Variable)
    - 栅栏 (Barriers)

多线程 - 互斥体

- 多线程：
  - 线程相关概念
  - 线程管理
  - 互斥体 (Mutex)
  - 锁 (Lock)
  - 条件变量 (Condition Variable)
  - 一次性初始化操作 (One-time Initialization)
  - 栅栏 (Barriers)
  - 线程局部存储 (TSS)
  - 时间和日期相关

多线程 - 互斥体

- Mutex Concepts
  - Lockable
  - TimedLockable
  - SharedLockable
  - UpgradeLockable



多线程 - 互斥体

- 关于 Mutex
  - 互斥体： Mutex(Mutual Exclusion)
  - boost thread 同步机制支持以下几种类型的 Mutex：
    - mutex
    - timed_mutex
    - recursive_mutex
    - recursive_timed_mutex
    - shared_mutex

多线程 - 互斥体

- mutex
  - boost::mutex 实现 Lockable concept ， 提供一个独占式的互斥体。对于一个实例最多允许一个线程拥有其锁定 。支持函数 lock(), try_lock() 和 unlock() 并发调用。

多线程 - 互斥体

- mutex 接口

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

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

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

    typedef unique_lock<mutex> scoped_lock;
    typedef unspecified-type scoped_try_lock;
};
```

多线程 - 互斥体

- timed_mutex
  - boost::timed_mutex 实现 TimedLockable concept ，提供一个独占式的互斥体。对于一个实例最多允许一个线程拥有其锁定 。支持函数 lock(), try_lock() ， timed_lock() 和 unlock() 并发调用。

- timed_mutex 接口

```
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 platform-specific-type native_handle_type;
    native_handle_type native_handle();
    typedef unique_lock<timed_mutex> scoped_timed_lock;
    typedef unspecified-type scoped_try_lock;
    typedef scoped_timed_lock scoped_lock;
};
```

多线程 - 互斥体

- recursive_mutex
  - boost::recursive_mutex 实现了 Lockable concept ，提供一个递归式的互斥体 。对于一个实例最多允许一个线程拥有其锁定 。支持函数 lock(), try_lock() 和 unlock() 并发调用。 如果一个线程已经锁定一个 boost::recursive_mutex 实例， 那么这个线程可以多次通过 lock() 或 try_lock() 锁定这个实例，针对每一次成功的锁定动作，需要调用 unlock() 来解除锁定。

多线程 - 互斥体

- recursive_mutex 接口

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

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

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

    typedef unique_lock<recursive_mutex> scoped_lock;
    typedef unspecified-type scoped_try_lock;
};
```

- recursive_mutex 示例

```
boost::recursive_mutex rm; // 如果是非递归互斥体，下面的用法直接导致死锁

void f1() {
    boost::lock_guard<boost::recursive_mutex> lock(rm);
    cout << "f1() with recursive_mutex" << endl;
}

void f2() {
    boost::lock_guard<boost::recursive_mutex> lock(rm);
    cout << "f2() with recursive_mutex" << endl;

    f1();
}

int main() {
    boost::thread t(f2);
    t.join();
}
```

多线程 - 互斥体

- recursive_timed_mutex
  - boost::recursive_timed_mutex 实现了 TimedLockable concept ，提供一个递归式的互斥体。对于一个实例最多允许一个线程拥有其锁定 。支持函数 lock(), try_lock() ，timed_lock() 和 unlock() 并发调用。如果一个线程已经锁定一个 boost::recursive_timed_mutex 实例，那么这个线程可以多次通过 lock() ， timed_lock() 或 try_lock() 锁定这个实例，针对每一次成功的锁定动作，需要调用 unlock() 来解除锁定。

- recursive_timed_mutex 接口

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

    typedef unique_lock<recursive_timed_mutex> scoped_lock;
    typedef unspecified-type scoped_try_lock;
    typedef scoped_lock scoped_timed_lock;
};
```

多线程 - 互斥体

- shared_mutex
  - boost::shared_mutex 提供了一个 multiple-reader / single-writer 互斥体，实现了 UpgradeLockable concept ， 允许并发调用函数 lock(), try_lock(), timed_lock(), lock_shared(), try_lock_shared() 和 timed_lock_shared() 。

- shared_mutex 接口

```
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();
};
```

多线程 - 互斥体

- mutex 使用方式

```
boost::mutex m;
int k = 0;

void f() {
    m.lock();
    try {
        ++k;
        // do other things ...
        // release mutex
        m.unlock();
    } catch (...) {
        m.unlock();
    }
}
```

多线程 - 锁

- 多线程：
  - 线程相关概念
  - 线程管理
  - 互斥体 (Mutex)
  - 锁 (Lock)
  - 条件变量 (Condition Variable)
  - 一次性初始化操作 (One-time Initialization)
  - 栅栏 (Barriers)
  - 线程局部存储 (TSS)
  - 时间和日期相关

多线程 - 锁

- 锁 (Lock)
  - 从本质上讲，锁 (lock) 是管理 Lackable 对象 (特别是 Mutex) 生命周期的一种手段，为 Lockable 对象提供了一个 RAII 风格的外观，方便实现异常安全的锁定和解锁。
  - 目前版本支持的锁类型：
    - lock_guard
    - unique_lock
    - shared_lock
    - upgrade_lock
    - upgrade_to_unique_lock

- 比较 Lock 和 Mutex 的锁定方式

```
boost::mutex m;
int k = 0;
void func1() { // 直接操纵mutex对象
    m.lock();
    try {
        ++k;
        // do some things
        m.unlock(); // 注意解锁
    } catch (...) {
        m.unlock(); // 注意解锁
    }
}
void func2() { // 使用lock管理mutex对象，RAII风格
    boost::mutex::scoped_lock lock(m); // alternate:
//  boost::lock_guard<boost::mutex> lock(m);
    ++k;
    // do some things
} // 到这里lock对象销毁，解除对m的锁定
```

- lock_guard
  - 该锁类型非常简单：构造函数传入一个可锁定对象，构造函数取得可锁定对象的所有权。 析构时释放所有权。 这样为 Lockable 对象提供了一个 RAII 风格的外观，方便实现异常安全的锁定和解锁。
  - 接口

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

    ~lock_guard();
};
```

多线程 - 锁

- unique_lock
  - 该锁比 lock_guard 复杂：不仅提供 RAII 风格的外观，也允许延迟获得锁定，直到 lock() 函数显式调用，或者支持非阻塞方式获得锁，或者是支持超时锁。 所以，在析构函数中，仅在 Lackable 对象被该锁锁定的情况下或是该锁接管了一个 Lockable 对象的情况下，才调用 unlock() 函数。

- unique_lock 接口

```
template<typename Lockable>
class unique_lock {
public:
    unique_lock();
    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);
```

- unique_lock 接口（续 1 ）

```
    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();
```

- unique_lock 接口（续 2 ）

```
    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();
};
```

- unique_lock 和 lock_guard 使用方式

```
boost::mutex m;
int k = 0;
void increment() {
    boost::unique_lock<boost::mutex> lock(m); // alternate:
    // boost::mutex::scoped_lock lock(m);
    for (int i = 0; i < 5; ++i)
        k += i;
    cout << "increment(): k == " << k << endl;
}
void decrement() {
    boost::lock_guard<boost::mutex> lock(m);
    for (int i = 0; i < 5; ++i)
        k -= i;
    cout << "decrement(): k == " << k << endl;
}
int main() {
    boost::thread t1(increment);
    boost::thread t2(decrement);
    t1.join(); t2.join();
}
```

- unique_lock 示例 2

```
boost::mutex m;
vector<string> strings;

void updateStrings() {
    boost::unique_lock<boost::mutex> lock(m);
    if (strings.empty()) {
        // loadStrings() 操作不涉及数据竞态，所以不需锁定
        lock.unlock();
        vector<string> localStrings;
        loadStrings(localStrings);

        // 更改共享数据前，重新获取锁
        lock.lock();
        copy(localStrings.begin(), localStrings.end(),
                back_inserter<string> (strings));
    }
}
```

多线程 - 锁

- shared_lock
  - 和 unique_lock 一样，shared_lock 对 Lockable concept 建模，除支持 Lockable 对象外，还支持获取共享权的获取。
  - 和 boost::unique_lock 一样，不仅提供 RAII 风格的外观，它也允许延迟获得锁定，直到 lock() 函数显式调用，或者支持非阻塞方式获得锁定，或者是支持超时锁定。 所以，在析构函数中，仅在 Lackable 对象被该锁锁定的情况下或是该锁接管了一个 Lockable 对象的情况下，才调用 unlock() 函数。
  - 如果函数 mutex() 返回指向的 m 指针并且 owns_lock() 返回 true，该 boost::shared_lock 实例拥有一个可锁定对象的锁定状态。如果该种实例被销毁，析构函数会调用函数 mutex()->unlock()。

- shared_lock 接口

```
template<typename Lockable>
class shared_lock {
public:
    shared_lock();
    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 接口

```
    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;
};
```

- upgrade_lock
  - 和 unique_lock 一样， boost::upgrade_lock 对 Lockable 建模，但是不仅限于可锁定对提供的独占锁定，还支持可升级锁定。
  - 和 unique_lock 一样，不仅提供 RAII 风格的外观，它也允许延迟获得锁定，直到 lock() 函数显式调用，或者支持非阻塞方式获得锁定，或者是支持超时锁定。 所以，在析构函数中，仅在 Lackable 对象被该锁锁定的情况下或是该锁接管了一个 Lockable 对象的情况下，才调用 unlock() 函数。
  - 如果函数 mutex() 返回指向的 m 指针并且 owns_lock() 返回 true ，该 upgrade_lock 实例拥有一个 Lockable 对象的锁定状态 。如果该种实例被销毁，析构函数会调用函数 mutex()->unlock() 。

- upgrade_lock 接口

```
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 接口

```
    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;
};
```

- upgrade_lock 示例

```
boost::shared_mutex m;
int k = 1;
void f(int id) {
    boost::upgrade_lock<boost::shared_mutex> lock(m);
    cout << "thread #" << id << ": " << k << endl;
    if (k < 6) {
        boost::unique_lock<boost::shared_mutex>
lock2(boost::move(lock)); // alternate:
//      boost::upgrade_to_unique_lock<boost::shared_mutex>
lock2(lock);
        k += 3;
    }
}
int main() {
    boost::thread t1(f, 1);
    boost::thread t2(f, 2);
    boost::thread t3(f, 3);

    t1.join(); t2.join(); t3.join();
}
```

多线程 - 锁

- upgrade_to_unique_lock
  - upgrade_to_unique_lock 允许临时从可升级锁定升级到独占锁定，如果传递 upgrade_lock 对象的引用给构造函数，该对象将升级到独占所有权，当这个对象销毁时， Lockable 对象恢复为可升级锁定。

- upgrade_to_unique_lock 接口

```
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;
};
```

多线程 - 锁

- lock 自由函数
  - 锁定参数提供的 Lockable 对象，并避免死锁。 该函数在多个线程并发调用锁定同一组互斥体 (Mutex)（或其他 Lockable 对象）是安全的，并且不用指定锁定顺序，也不用担心死锁。 如果函数在锁定对象时抛出异常，那么在函数退出前，该函数此次调用已锁定的对象也会被释放。

- lock 函数的接口

```
template<typename Lockable1, typename Lockable2>
void lock(Lockable1& l1, Lockable2& l2);

template<typename Lockable1, typename Lockable2, typename
Lockable3>
void lock(Lockable1& l1, Lockable2& l2, Lockable3& l3);

template<typename Lockable1, typename Lockable2, typename
Lockable3, typename Lockable4>
void lock(Lockable1& l1, Lockable2& l2, Lockable3& l3,
Lockable4& l4);

template<typename Lockable1, typename Lockable2, typename
Lockable3, typename Lockable4, typename Lockable5>
void lock(Lockable1& l1, Lockable2& l2, Lockable3& l3,
Lockable4& l4, Lockable5& l5);
```

- 使用 lock 函数避免死锁

```
// Code from http://www.justsoftwaresolutions.co.uk/threading/
// with some changes.

class Account {
    boost::mutex m;
    double balance;
public:
    Account() :
        balance() {
    }
    Account(const double& bal) :
        balance(bal) {
    }
    double getBalance() const {
        return balance;
    }
    friend void transfer(Account& from, Account& to, double
amount);
};
```

- 使用 lock 函数避免死锁（续 1 ）

```
int main() {
    Account a1(1200.00);
    Account a2(300.00);

    boost::thread t1(transfer,
                     boost::ref(a1), boost::ref(a2), 134.85);
    boost::thread t2(transfer,
                     boost::ref(a2), boost::ref(a1), 100.30);
    t1.join();
    t2.join();

    cout << "Balance of a1: " << a1.getBalance() << endl;
    cout << "Balance of a2: " << a2.getBalance() << endl;
}
```

- 使用 lock 函数避免死锁（续 2 ）

```
// version 1: 可能造成死锁
void transfer(Account& from, Account& to, double amount) {
    boost::lock_guard<boost::mutex> lockFrom(from.m);
    boost::lock_guard<boost::mutex> lockTo(to.m);
    from.balance -= amount;
    to.balance += amount;
}
```

```
// version 2: OK （使用lock() 和 lock_guard）
void transfer(Account& from, Account& to, double amount) {
    boost::lock(from.m, to.m);
    boost::lock_guard<boost::mutex> lockFrom(from.m,
boost::adopt_lock);
    boost::lock_guard<boost::mutex> lockTo(to.m,
boost::adopt_lock);
    from.balance -= amount;
    to.balance += amount;
}
```

- 使用 lock 函数避免死锁（续 3 ）

```
// version 3: OK （使用lock() 和unique_lock）
void transfer(Account& from, Account& to, double amount) {
    boost::lock(from.m, to.m);
    boost::unique_lock<boost::mutex> lockFrom(from.m,
boost::adopt_lock);
    boost::unique_lock<boost::mutex> lockTo(to.m,
boost::adopt_lock);

    from.balance -= amount;
    to.balance += amount;
}
```

多线程 - 条件变量

- 多线程：
  - 线程相关概念
  - 线程管理
  - 互斥体 (Mutex)
  - 锁 (Lock)
  - 条件变量 (Condition Variable)
  - 一次性初始化操作 (One-time Initialization)
  - 栅栏 (Barriers)
  - 线程局部存储 (TSS)
  - 时间和日期相关

多线程 - 条件变量

- 关于条件变量 (Condition Variable)
  - condition_variable 和 condition_variable_any 提供一种机制，一个线程可以等待另外一个线程内某个事件发生的通知。 通常的应用模式是一个线程锁定一个互斥体，然后通过函数 wait 等待一个 condition_variable 或 condition_variable_any 实例，当线程从 wait 函数激活时， 检查特定的条件是否满足，如果满足，线程继续执行；如果不满足，线程继续等待。

- 关于条件变量：一个小示例

```
boost::condition_variable cond; // 关联多个线程的条件变量
boost::mutex m; // 保护共享资源 k 的互斥体
int k = 0; // 共享资源

void f1() {
    boost::unique_lock<boost::mutex> lock(m);
    while (k < 5) {
        cout << "k < 5, waiting ..." << endl;
        cond.wait(lock);  // #1
    }
    cout << "now k >= 5, printing ..." << endl;
}
void f2() {
    {
        boost::unique_lock<boost::mutex> lock(m);
        cout << "k will be changed ..." << endl;
        k += 5;
    }
    cond.notify_all(); // #2 不需lock
}
```

- 关于条件变量：一个小示例（续）

```
int main() {
    // 如果f2()中是cond.notify_one(), 结果？
    boost::thread t1(f1);
    boost::thread t2(f1);
    boost::thread t3(f2);

    t1.join();
    t2.join();
    t3.join();
}
```

多线程 - 条件变量

- 关于条件变量：一个小示例（续 2 ）
  - #1 ： lock 对象会传递给 wait 函数， wait 函数会自动将线程添加到等待条件变量（ k >= 5 ）的线程集合中，并且解锁传递进来的互斥体。当线程被唤醒，函数退出前互斥体会再次被锁定。这样允许其他线程获取互斥体来更新共享的数据，确保条件变量关联的数据被正确同步。
  - #2 ： 另外的线程（执行 f2() 的线程）将这个条件置为 true ， 然后对条件变量调用函数 notify_one 或 notify_all 来唤醒等待该条件变量的一个线程或多个线程 。
    lock 锁定的范围不需包含 cond.notify_all() 函数的调用。

多线程 - 条件变量

- 条件变量 condition_variable 和 condition_variable_any
  - boost thread 提供 2 种类型的条件变量：
    - condition_variable
    - condition_variable_any
  - condition_variable 的 wait() 操作只支持 unique_lock 类型的锁
  - 而 condition_variable_any 的 wait() 操作支持各种类别的 Lock（ any 指的是任意类型的 Mutex ）
  - 除此之外，二者的操作方式和效果一致

- 条件变量 condition_variable 接口

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

    void notify_one();
    void notify_all();
    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);
```

- 条件变量 condition_variable 接口（续）

```
    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);
};
```

- 条件变量 condition_variable_any 接口

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

    void notify_one();
    void notify_all();
    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);
```

- 条件变量 condition_variable_any 接口（续）

```
 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);
};
```

多线程 - 一次性初始化操作

- 多线程：
  - 线程相关概念
  - 线程管理
  - 互斥体 (Mutex)
  - 锁 (Lock)
  - 条件变量 (Condition Variable)
  - 一次性初始化操作 (One-time Initialization)
  - 栅栏 (Barriers)
  - 线程局部存储 (TSS)
  - 时间和日期相关

多线程 - 一次性初始化操作

- 一次性初始化
  - 在多线程环境下，有时候需要保证一些初始化操作只被唯一的调用一次，这时候可以使用 call_once() 函数
  - 接口

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

// boost::once_flag对象由BOOST_ONCE_INIT初始化:
boost::once_flag f=BOOST_ONCE_INIT;

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

多线程 - 一次性初始化操作

- 函数 call_once()
  - 对同一个 once_flag 对象调用 call_once 被序列化。如果在此之前没有有效的 call_once 调用， func() 对象会被调用，如果 func() 成功，那么 call_once 调用被视为有效的，如果 func() 抛出异常，该异常被传递到调用者， call_once 调用被视为无效的；如果对同一个 once_flag 对象有过有效的 call_once 调用， call_once 只是简单返回，并不调用 func 。

多线程 - 一次性初始化操作

- call_once() 示例

```
boost::once_flag once = BOOST_ONCE_INIT; // 注意这个操作不要遗漏了

void func() {
    cout << "Will be called but one time!" << endl;
}

void threadFunc() {
    boost::call_once(&func, once);
}

int main() {
    boost::thread_group threads;
    for (int i = 0; i < 5; ++i)
        threads.create_thread(&threadFunc);
    threads.join_all();
}
```

多线程 - 栅栏

- 多线程：
  - 线程相关概念
  - 线程管理
  - 互斥体 (Mutex)
  - 锁 (Lock)
  - 条件变量 (Condition Variable)
  - 一次性初始化操作 (One-time Initialization)
  - 栅栏 (Barriers)
  - 线程局部存储 (TSS)
  - 时间和日期相关

多线程 - 栅栏

- 关于栅栏 (Barrier)
  - 栅栏也被称为“聚合点 (rendezvous)”，是多个线程间的同步点。
  - barrier 被配置为一定数量的线程使用 (n)，当线程到达 barrier 时，他们必须等待，直到所有线程都到达这个 barrier。当最后一个线程到达这个 barrier，所有的线程才能继续执行，并且这个 barrier 对象被复位。
- 接口

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

- barrier 示例

```
namespace {
// Shared variables for generation barrier test
const int THREADS = 6;
boost::barrier barriers(THREADS / 2);
boost::mutex m;
long global;

void barrierThread(int n) {
    cout << "thread #" << n << " started." << endl;
    for (int i = 0; i < THREADS / 2; ++i) {
        if (barriers.wait()) {
            boost::mutex::scoped_lock lock(m);
            cout << "thread #" << n << " arrived here." <<
endl;
            ++global;
        }
    }
}
} // namespace
```

- barrier 示例 (续)

```
void testBarrier() {
    boost::thread_group g;
    try {
        for (int i = 0; i < THREADS; ++i)
            g.create_thread(boost::bind(barrierThread, i));
        g.join_all();
    } catch (...) {
        g.interrupt_all();
        g.join_all();
        throw ;
    }

   cout << global << endl;  // 6
}
```

多线程 - 线程局部存储

- 多线程：
  - 线程相关概念
  - 线程管理
  - 互斥体 (Mutex)
  - 锁 (Lock)
  - 条件变量 (Condition Variable)
  - 一次性初始化操作 (One-time Initialization)
  - 栅栏 (Barriers)
  - 线程局部存储 (TSS)
  - 时间和日期相关

多线程 - 线程局部存储

- 关于线程局部存储 (Thread Specific Storage)
  - 线程本地化存储（ TSS 或称 TLS-Thread Local Storage ）允许多个线程对象拥有特定数据的独立拷贝．单线程程序通常使用的静态数据和全局数据在多线程程序中会引发访问竞争，死锁，或者数据破坏 ．一个例子就是 C 标准库中的 errno 变量，该变量放 C 标准库中函数的错误代码．对于支持多线程的编译器来说，为每个线程提供一个独立的 errno 是常见的做法，以此避免多个线程竞争对它的访问和更新 (这也是 POSIX 标准要求的).
  - 尽管编译器通常提供一些形式的语法扩展来方便标记线程局部存储 (比如用于 static 或名字空间内变量声明前的 __declspec(thread) 或 __thread 标记)，但是这些支持是不可移植的，也仅限于某些用途，比如只支持 POD 类型 .

多线程 - 线程局部存储

- thread_specific_ptr
  - boost::thread_specific_ptr 提供了一个线程本地化存储机制的可移植实现，在支持 Boost.Thread 的所有平台上适用．每个 boost::thread_specific_ptr 示例的指针指向一个对象 (如 errno) 这些对象要求在不同的线程间有不同的值．当前线程对应的对象的值可以通过成员函数 get() 获取，或是通过 * 和 -> 操作．这些对象指针在每个线程中的初始值为 NULL，但可以通过成员函数 reset() 改变当前线程对应的值．
  - 如果通过 reset() 将 boost::thread_specific_ptr 的指向改变，那么在此之前的值会被清理，另外，存储的值可以通过成员函数 release() 置为 NULL，该函数同时返回这个值，这样应用程序可以有机会销毁这个值关联的对象．

多线程 - 线程局部存储

- 线程退出时清理
  - 当一个线程退出，boost::thread_specific_ptr 实例 所关联的对象会被销毁．通常销毁的动作通过对被指向的对象调用 delete 完成，这个行为也可以通过向 boost::thread_specific_ptr 对象构造时传递一个清理函数 func() 来重载．此时，所指向对象通过调用 func(p) 来销毁 ．这些清理函数调用顺序不是确定的 ．如果一个清理函数将 对象关联的值设置为一个被清理过的值，这个值会被添加到清理列表中．当所有 boost::thread_specific_ptr 对象被置为空时，清理过程结束．

多线程 - 线程局部存储

- thread_specific_ptr 接口

```
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);
};
```

多线程 - 线程局部存储

- thread_specific_ptr 示例

```
boost::thread_specific_ptr<int> value;
void increment() {
    int* p = value.get();
    ++*p;
}
void thread_proc() {
    value.reset(new int(0)); //initialize the thread's storage
    for (int i = 0; i < 10; ++i) {
        increment();
        int* p = value.get();
        assert(*p == i+1);
    }
}
int main() {
    boost::thread_group threads;
    for (int i = 0; i < 5; ++i)
        threads.create_thread(&thread_proc);
    threads.join_all();
}
```

多线程 - 时间和日期相关

- 多线程：
  - 线程相关概念
  - 线程管理
  - 互斥体 (Mutex)
  - 锁 (Lock)
  - 条件变量 (Condition Variable)
  - 一次性初始化操作 (One-time Initialization)
  - 栅栏 (Barriers)
  - 线程局部存储 (TSS)
  - 时间和日期相关

多线程 - 时间和日期相关

- boost thread 和 boost date_time
  - boost thread 中很多函数需要使用 date_time 相关的操作，如：
    - boost::this_thread::sleep()
    - timed_join()
    - timed_wait()
    - timed_lock()
    - …

多线程 - 时间和日期相关

- 绝对时间和相对时间参数

```
// 需要绝对时间
boost::system_time const timeout = boost::get_system_time()
        + boost::posix_time::milliseconds(500);
bool done;
boost::mutex m;
boost::condition_variable cond;
boost::unique_lock<boost::mutex> lk(m);
while (!done) {
    if (!cond.timed_wait(lk, timeout)) { // 需要绝对时间
        throw "timed out";
    }
}

// 需要相对时间(时间间隔)
boost::this_thread::sleep(boost::posix_time::milliseconds(25));
boost::mutex m;
if (m.timed_lock(boost::posix_time::nanoseconds(100))) {
    //  ...
}
```

多线程 - 时间和日期相关

- typedef system_time 和 get_system_time()

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

typedef boost::posix_time::ptime system_time;

system_time get_system_time(); // 返回系统当前时间
```

多线程 - More stuffs...

- Boost.Thread 封装（或模拟）当前主要操作系统中多线程机制，提供了统一的编程接口，简化并发编程的工作。
- 由于在 Linux 下， boost.Thread 的底层实现依赖与 pthread 线程库，所以，如果想对多线程编程有进一步的了解，可以参考以下资料，以进一步的深入：
  - Pthreads Programming (Book) By:
    Bradford Nichols, Dick Buttlar and Jacqueline Proulx Farrell
  - POSIX Threads Programming (Tutorial)
    https://computing.llnl.gov/tutorials/pthreads/