D.5 mutex头文件

<mutex> 头文件提供互斥工具：互斥类型，锁类型和函数，还有确保操作只执行一次的机制。

头文件内容

```
namespace std
{
  class mutex;
  class recursive_mutex;
  class timed_mutex;
  class recursive_timed_mutex;

  struct adopt_lock_t;
  struct defer_lock_t;
  struct try_to_lock_t;

  constexpr adopt_lock_t adopt_lock{};
  constexpr defer_lock_t defer_lock{};
  constexpr try_to_lock_t try_to_lock{};

  template<typename LockableType>
  class lock_guard;

  template<typename LockableType>
  class unique_lock;

  template<typename LockableType1,typename... LockableType2>
  void lock(LockableType1& m1,LockableType2& m2...);

  template<typename LockableType1,typename... LockableType2>
  int try_lock(LockableType1& m1,LockableType2& m2...);

  struct once_flag;

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

D.5.1 std::mutex类

`std::mutex` 类型为线程提供基本的互斥和同步工具，这些工具可以用来保护共享数据。互斥量可以用来保护数据，互斥量上锁必须要调用lok()或try_lock()。当有一个线程获取已经获取了锁，那么其他线程想要在获取锁的时候，会在尝试或取锁的时候失败(调用try_lock())或阻塞(调用lock())，具体酌情而定。当线程完成对共享数据的访问，之后就必须调用unlock()对锁进行释放，并且允许其他线程来访问这个共享数据。

`std::mutex` 符合Lockable的需求。

类型定义

```
class mutex
{
public:
  mutex(mutex const&)=delete;
  mutex& operator=(mutex const&)=delete;

  constexpr mutex() noexcept;
  ~mutex();

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

std::mutex 默认构造函数

构造一个 `std::mutex` 对象。

声明

```
  constexpr mutex() noexcept;
```

效果
构造一个 `std::mutex` 实例。

后置条件
新构造的 `std::mutex` 对象是未锁的。

抛出
无

std::mutex 析构函数

销毁一个 `std::mutex` 对象。

声明

```
~mutex();
```

先决条件
*this必须是未锁的。

效果
销毁*this。

抛出
无

std::mutex::lock 成员函数

为当前线程获取 `std::mutex` 上的锁。

声明

```
void lock();
```

先决条件
*this上必须没有持有一个锁。

效果
阻塞当前线程，知道*this获取锁。

后置条件
*this被调用线程锁住。

抛出
当有错误产生，抛出 `std::system_error` 类型异常。

std::mutex::try_lock 成员函数

尝试为当前线程获取 `std::mutex` 上的锁。

声明

```
bool try_lock();
```

先决条件
*this上必须没有持有一个锁。

效果
尝试为当前线程*this获取上的锁，失败时当前线程不会被阻塞。

返回
当调用线程获取锁时，返回true。

后置条件
当*this被调用线程锁住，则返回true。

抛出 无

NOTE 该函数在获取锁时，可能失败(并返回false)，即使没有其他线程持有*this上的锁。

std::mutex::unlock 成员函数

释放当前线程获取的 `std::mutex` 锁。

声明

```
void unlock();
```

先决条件
*this上必须持有一个锁。

效果 释放当前线程获取到 `*this` 上的锁。任意等待获取 `*this` 上的线程，会在该函数调用后解除阻塞。

后置条件
调用线程不在拥有*this上的锁。

抛出
无

D.5.2 std::recursive_mutex类

`std::recursive_mutex` 类型为线程提供基本的互斥和同步工具，可以用来保护共享数据。互斥量可以用来保护数据，互斥量上锁必须要调用lok()或try_lock()。当有一个线程获取已经获取了锁，那么其他线程想要在获取锁的时候，会在尝试或取锁的时候失败(调用try_lock())或阻塞(调用lock())，具体酌情而定。当线程完成对共享数据的访问，之后就必须调用unlock()对锁进行释放，并且允许其他线程来访问这个共享数据。

这个互斥量是可递归的，所以一个线程获取 `std::recursive_mutex` 后可以在之后继续使用lock()或try_lock()来增加锁的计数。只有当线程调用unlock释放锁，其他线程才可能用lock()或try_lock()获取锁。

`std::recursive_mutex` 符合Lockable的需求。

类型定义

```
class recursive_mutex
{
public:
  recursive_mutex(recursive_mutex const&)=delete;
  recursive_mutex& operator=(recursive_mutex const&)=delete;

  recursive_mutex() noexcept;
 ~recursive_mutex();

  void lock();
  void unlock();
  bool try_lock() noexcept;
};
```

std::recursive_mutex 默认构造函数

构造一个 `std::recursive_mutex` 对象。

声明

```
recursive_mutex() noexcept;
```

效果
构造一个 `std::recursive_mutex` 实例。

后置条件
新构造的 `std::recursive_mutex` 对象是未锁的。

抛出
当无法创建一个新的 `std::recursive_mutex` 时，抛出 `std::system_error` 异常。

std::recursive_mutex 析构函数

销毁一个 `std::recursive_mutex` 对象。

声明

```
~recursive_mutex();
```

先决条件
*this必须是未锁的。

效果
销毁*this。

抛出
无

std::recursive_mutex::lock 成员函数

为当前线程获取 `std::recursive_mutex` 上的锁。

声明

```
void lock();
```

效果

阻塞线程，直到获取*this上的锁。

先决条件

调用线程锁住*this上的锁。当调用已经持有一个*this的锁时，锁的计数会增加1。

抛出

当有错误产生，将抛出 `std::system_error` 异常。

std::recursive_mutex::try_lock 成员函数

尝试为当前线程获取 `std::recursive_mutex` 上的锁。

声明

```
bool try_lock() noexcept;
```

效果

尝试为当前线程*this获取上的锁，失败时当前线程不会被阻塞。

返回

当调用线程获取锁时，返回true；否则，返回false。

后置条件

当*this被调用线程锁住，则返回true。

抛出 无

NOTE 该函数在获取锁时，当函数返回true时， `*this` 上对锁的计数会加一。如果当前线程还未获取 `*this` 上的锁，那么该函数在获取锁时，可能失败(并返回false)，即使没有其他线程持有 `*this` 上的锁。

std::recursive_mutex::unlock 成员函数

释放当前线程获取的 `std::recursive_mutex` 锁。

声明

```
void unlock();
```

先决条件
*this上必须持有一个锁。

效果 释放当前线程获取到 `*this` 上的锁。如果这是 `*this` 在当前线程上最后一个锁，那么任意等待获取 `*this` 上的线程，会在该函数调用后解除其中一个线程的阻塞。

后置条件
`*this` 上锁的计数会在该函数调用后减一。

抛出
无

D.5.3 std::timed_mutex类

`std::timed_mutex` 类型在 `std::mutex` 基本互斥和同步工具的基础上，让锁支持超时。互斥量可以用来保护数据，互斥量上锁必须要调用lok(),try_lock_for(),或try_lock_until()。当有一个线程获取已经获取了锁，那么其他线程想要在获取锁的时候，会在尝试或取锁的时候失败(调用try_lock())或阻塞(调用lock())，或直到想要获取锁可以获取，亦或想要获取的锁超时(调用try_lock_for()或try_lock_until())。在线程调用unlock()对锁进行释放，其他线程才能获取这个锁被获取(不管是调用的哪个函数)。

`std::timed_mutex` 符合TimedLockable的需求。

类型定义

```
class timed_mutex
{
public:
  timed_mutex(timed_mutex const&)=delete;
  timed_mutex& operator=(timed_mutex const&)=delete;

  timed_mutex();
```

```
    ~timed_mutex();

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

    template<typename Rep,typename Period>
    bool try_lock_for(
        std::chrono::duration<Rep,Period> const& relative_time);

    template<typename Clock,typename Duration>
    bool try_lock_until(
        std::chrono::time_point<Clock,Duration> const& absolute_time);
};
```

std::timed_mutex 默认构造函数

构造一个 `std::timed_mutex` 对象。

声明

```
timed_mutex();
```

效果

构造一个 `std::timed_mutex` 实例。

后置条件

新构造一个未上锁的 `std::timed_mutex` 对象。

抛出

当无法创建出新的 `std::timed_mutex` 实例时，抛出 `std::system_error` 类型异常。

std::timed_mutex 析构函数

销毁一个 `std::timed_mutex` 对象。

声明

```
~timed_mutex();
```

先决条件
*this必须没有上锁。

效果
销毁*this。

抛出
无

std::timed_mutex::lock 成员函数

为当前线程获取 `std::timed_mutex` 上的锁。

声明

```
void lock();
```

先决条件
调用线程不能已经持有*this上的锁。

效果
阻塞当前线程，直到获取到*this上的锁。

后置条件
*this被调用线程锁住。

抛出
当有错误产生，抛出 `std::system_error` 类型异常。

std::timed_mutex::try_lock 成员函数

尝试获取为当前线程获取 `std::timed_mutex` 上的锁。

声明

```
bool try_lock();
```

先决条件
调用线程不能已经持有*this上的锁。

效果
尝试获取*this上的锁，当获取失败时，不阻塞调用线程。

返回
当锁被调用线程获取，返回true；反之，返回false。

后置条件
当函数返回为true，*this则被当前线程锁住。

抛出
无

NOTE 即使没有线程已获取*this上的锁，函数还是有可能获取不到锁(并返回false)。

std::timed_mutex::try_lock_for 成员函数

尝试获取为当前线程获取 `std::timed_mutex` 上的锁。

声明

```
template<typename Rep,typename Period>
bool try_lock_for(
    std::chrono::duration<Rep,Period> const& relative_time);
```

先决条件
调用线程不能已经持有*this上的锁。

效果
在指定的relative_time时间内，尝试获取*this上的锁。当relative_time.count()为0或负数，将会立即返回，就像调用try_lock()一样。否则，将会阻塞，直到获取锁或超过给定的relative_time的时间。

返回
当锁被调用线程获取，返回true；反之，返回false。

后置条件
当函数返回为true，*this则被当前线程锁住。

抛出
无

NOTE 即使没有线程已获取*this上的锁，函数还是有可能获取不到锁(并返回false)。线程阻塞的时长可能会长于给定的时间。逝去的时间可能是由一个稳定时钟所决定。

std::timed_mutex::try_lock_until 成员函数

尝试获取为当前线程获取 `std::timed_mutex` 上的锁。

声明

```
template<typename Clock,typename Duration>
bool try_lock_until(
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

先决条件
调用线程不能已经持有*this上的锁。

效果
在指定的absolute_time时间内，尝试获取*this上的锁。当 `absolute_time<=Clock::now()` 时，将会立即返回，就像调用try_lock()一样。否则，将会阻塞，直到获取锁或Clock::now()返回的时间等于或超过给定的absolute_time的时间。

返回
当锁被调用线程获取，返回true；反之，返回false。

后置条件
当函数返回为true，*this则被当前线程锁住。

抛出
无

NOTE 即使没有线程已获取*this上的锁，函数还是有可能获取不到锁(并返回false)。这里不保证调用函数要阻塞多久，只有在函数返回false后，在Clock::now()返回的时间大于或等于absolute_time时，线程才会接触阻塞。

std::timed_mutex::unlock 成员函数

将当前线程持有 `std::timed_mutex` 对象上的锁进行释放。

声明

```
void unlock();
```

先决条件

调用线程已经持有*this上的锁。

效果

当前线程释放 `*this` 上的锁。任一阻塞等待获取 `*this` 上的线程，将被解除阻塞。

后置条件

*this未被调用线程上锁。

抛出

无

D.5.4 std::recursive_timed_mutex类

`std::recursive_timed_mutex` 类型在 `std::recursive_mutex` 提供的互斥和同步工具的基础上，让锁支持超时。互斥量可以用来保护数据，互斥量上锁必须要调用lok(),try_lock_for(),或try_lock_until()。当有一个线程获取已经获取了锁，那么其他线程想要在获取锁的时候，会在尝试或取锁的时候失败(调用try_lock())或阻塞(调用lock())，或直到想要获取锁可以获取，亦或想要获取的锁超时(调用try_lock_for()或try_lock_until())。在线程调用unlock()对锁进行释放，其他线程才能获取这个锁被获取(不管是调用的哪个函数)。

该互斥量是可递归的，所以获取 `std::recursive_timed_mutex` 锁的线程，可以多次的对该实例上的锁获取。所有的锁将会在调用相关unlock()操作后，可由其他线程获取该实例上的锁。

`std::recursive_timed_mutex` 符合TimedLockable的需求。

类型定义

```
1 class recursive_timed_mutex
2 {
3 public:
```

```
    recursive_timed_mutex(recursive_timed_mutex const&)=delete;
    recursive_timed_mutex& operator=(recursive_timed_mutex const&)=delete;

    recursive_timed_mutex();
    ~recursive_timed_mutex();

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

    template<typename Rep,typename Period>
    bool try_lock_for(
        std::chrono::duration<Rep,Period> const& relative_time);

    template<typename Clock,typename Duration>
    bool try_lock_until(
        std::chrono::time_point<Clock,Duration> const& absolute_time);
};
```

std::recursive_timed_mutex 默认构造函数

构造一个 `std::recursive_timed_mutex` 对象。

声明

```
recursive_timed_mutex();
```

效果

构造一个 `std::recursive_timed_mutex` 实例。

后置条件

新构造的 `std::recursive_timed_mutex` 实例是没有上锁的。

抛出

当无法创建一个 `std::recursive_timed_mutex` 实例时，抛出 `std::system_error` 类异常。

std::recursive_timed_mutex 析构函数

析构一个 `std::recursive_timed_mutex` 对象。

声明

```
~recursive_timed_mutex();
```

先决条件
*this不能上锁。

效果
销毁*this。

抛出
无

std::recursive_timed_mutex::lock 成员函数

为当前线程获取 `std::recursive_timed_mutex` 对象上的锁。

声明

```
void lock();
```

先决条件
*this上的锁不能被线程调用。

效果
阻塞当前线程，直到获取*this上的锁。

后置条件
`*this` 被调用线程锁住。当调用线程已经获取 `*this` 上的锁，那么锁的计数会再增加1。

抛出
当错误出现时，抛出 `std::system_error` 类型异常。

std::recursive_timed_mutex::try_lock 成员函数

尝试为当前线程获取 `std::recursive_timed_mutex` 对象上的锁。

声明

```
bool try_lock() noexcept;
```

效果
尝试获取*this上的锁，当获取失败时，直接不阻塞线程。

返回
当调用线程获取了锁，返回true，否则返回false。

后置条件
当函数返回true， `*this` 会被调用线程锁住。

抛出
无

NOTE 该函数在获取锁时，当函数返回true时， `*this` 上对锁的计数会加一。如果当前线程还未获取 `*this` 上的锁，那么该函数在获取锁时，可能失败(并返回false)，即使没有其他线程持有 `*this` 上的锁。

std::recursive_timed_mutex::try_lock_for 成员函数

尝试为当前线程获取 `std::recursive_timed_mutex` 对象上的锁。

声明

```
template<typename Rep,typename Period>
bool try_lock_for(
    std::chrono::duration<Rep,Period> const& relative_time);
```

效果
在指定时间relative_time内，尝试为调用线程获取*this上的锁。当relative_time.count()为0或负数时，将会立即返回，就像调用try_lock()一样。否则，调用会阻塞，直到获取相应的锁，或超出了relative_time时限时，调用线程解除阻塞。

返回
当调用线程获取了锁，返回true，否则返回false。

后置条件
当函数返回true， `*this` 会被调用线程锁住。

抛出
无

NOTE 该函数在获取锁时，当函数返回true时， `*this` 上对锁的计数会加一。如果当前线程还未获取 `*this` 上的锁，那么该函数在获取锁时，可能失败(并返回false)，即使没有其他线程持有 `*this` 上的锁。等待时间可能要比指定的时间长很多。逝去的时间可能由一个稳定时钟来计算。

std::recursive_timed_mutex::try_lock_until 成员函数

尝试为当前线程获取 `std::recursive_timed_mutex` 对象上的锁。

声明

```
template<typename Clock,typename Duration>
bool try_lock_until(
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

效果
在指定时间absolute_time内，尝试为调用线程获取*this上的锁。当absolute_time<=Clock::now()时，将会立即返回，就像调用try_lock()一样。否则，调用会阻塞，直到获取相应的锁，或Clock::now()返回的时间大于或等于absolute_time时，调用线程解除阻塞。

返回
当调用线程获取了锁，返回true，否则返回false。

后置条件
当函数返回true， `*this` 会被调用线程锁住。

抛出
无

NOTE 该函数在获取锁时，当函数返回true时， `*this` 上对锁的计数会加一。如果当前线程还未获取 `*this` 上的锁，那么该函数在获取锁时，可能失败(并返回false)，即使没有其他线程持有 `*this` 上的锁。这里阻塞的时间并不确定，只有当函数返回false，然后Clock::now()返回的时间大于或等于absolute_time时，调用线程将会解除阻塞。

std::recursive_timed_mutex::unlock 成员函数

释放当前线程获取到的 `std::recursive_timed_mutex` 上的锁。

声明

```
  void unlock();
```

效果

当前线程释放 `*this` 上的锁。当 `*this` 上最后一个锁被释放后，任何等待获取 `*this` 上的锁将会解除阻塞，不过只能解除其中一个线程的阻塞。

后置条件

调用线程*this上锁的计数减一。

抛出

无

D.5.5 std::lock_guard类型模板

`std::lock_guard` 类型模板为基础锁包装所有权。所要上锁的互斥量类型，由模板参数Mutex来决定，并且必须符合Lockable的需求。指定的互斥量在构造函数中上锁，在析构函数中解锁。这就为互斥量锁部分代码提供了一个简单的方式；当程序运行完成时，阻塞解除，互斥量解锁(无论是执行到最后，还是通过控制流语句break或return，亦或是抛出异常)。

`std::lock_guard` 是不可MoveConstructible(移动构造), CopyConstructible(拷贝构造)和CopyAssignable(拷贝赋值)。

类型定义

```
1  template <class Mutex>
2  class lock_guard
3  {
4  public:
5    typedef Mutex mutex_type;
6
7    explicit lock_guard(mutex_type& m);
8    lock_guard(mutex_type& m, adopt_lock_t);
9    ~lock_guard();
10
11   lock_guard(lock_guard const& ) = delete;
12   lock_guard& operator=(lock_guard const& ) = delete;
```

```
13 };
```

std::lock_guard 自动上锁的构造函数

使用互斥量构造一个 `std::lock_guard` 实例。

声明

```
explicit lock_guard(mutex_type& m);
```

效果
通过引用提供的互斥量，构造一个新的 `std::lock_guard` 实例，并调用m.lock()。

抛出
m.lock()抛出的任何异常。

后置条件
*this拥有m上的锁。

std::lock_guard 获取锁的构造函数

使用已提供互斥量上的锁，构造一个 `std::lock_guard` 实例。

声明

```
lock_guard(mutex_type& m,std::adopt_lock_t);
```

先决条件
调用线程必须拥有m上的锁。

效果
调用线程通过引用提供的互斥量，以及获取m上锁的所有权，来构造一个新的 `std::lock_guard` 实例。

抛出
无

后置条件
*this拥有m上的锁。

std::lock_guard 析构函数

销毁一个 `std::lock_guard` 实例，并且解锁相关互斥量。

声明

```
~lock_guard();
```

效果
当*this被创建后，调用m.unlock()。

抛出
无

D.5.6 std::unique_lock类型模板

`std::unique_lock` 类型模板相较 `std::loc_guard` 提供了更通用的所有权包装器。上锁的互斥量可由模板参数Mutex提供，这个类型必须满足BasicLockable的需求。虽然，通常情况下，制定的互斥量会在构造的时候上锁，析构的时候解锁，但是附加的构造函数和成员函数提供灵活的功能。互斥量上锁，意味着对操作同一段代码的线程进行阻塞；当互斥量解锁，就意味着阻塞解除(不论是裕兴到最后，还是使用控制语句break和return，亦或是抛出异常)。
`std::condition_variable` 的邓丹函数是需要 `std::unique_lock<std::mutex>` 实例的，并且所有 `std::unique_lock` 实例都适用于 `std::conditin_variable_any` 等待函数的Lockable参数。

当提供的Mutex类型符合Lockable的需求，那么 `std::unique_lock<Mutex>` 也是符合Lockable的需求。此外，如果提供的Mutex类型符合TimedLockable的需求，那么
`std::unique_lock<Mutex>` 也符合TimedLockable的需求。

`std::unique_lock` 实例是MoveConstructible(移动构造)和MoveAssignable(移动赋值)，但是不能CopyConstructible(拷贝构造)和CopyAssignable(拷贝赋值)。

类型定义

```
template <class Mutex>
class unique_lock
{
public:
  typedef Mutex mutex_type;

  unique_lock() noexcept;
  explicit unique_lock(mutex_type& m);
  unique_lock(mutex_type& m, adopt_lock_t);
  unique_lock(mutex_type& m, defer_lock_t) noexcept;
  unique_lock(mutex_type& m, try_to_lock_t);

  template<typename Clock,typename Duration>
  unique_lock(
      mutex_type& m,
      std::chrono::time_point<Clock,Duration> const& absolute_time);

  template<typename Rep,typename Period>
      unique_lock(
      mutex_type& m,
      std::chrono::duration<Rep,Period> const& relative_time);

  ~unique_lock();

  unique_lock(unique_lock const& ) = delete;
  unique_lock& operator=(unique_lock const& ) = delete;

  unique_lock(unique_lock&& );
  unique_lock& operator=(unique_lock&& );

  void swap(unique_lock& other) noexcept;

  void lock();
  bool try_lock();
  template<typename Rep, typename Period>
  bool try_lock_for(
      std::chrono::duration<Rep,Period> const& relative_time);
  template<typename Clock, typename Duration>
  bool try_lock_until(
      std::chrono::time_point<Clock,Duration> const& absolute_time);
  void unlock();

  explicit operator bool() const noexcept;
  bool owns_lock() const noexcept;
  Mutex* mutex() const noexcept;
```

```
46    Mutex* release() noexcept;
47  };
```

std::unique_lock 默认构造函数

不使用相关互斥量，构造一个 std::unique_lock 实例。

声明

```
  unique_lock() noexcept;
```

效果
构造一个 std::unique_lock 实例，这个新构造的实例没有相关互斥量。

后置条件
this->mutex()==NULL, this->owns_lock()==false.

std::unique_lock 自动上锁的构造函数

使用相关互斥量，构造一个 std::unique_lock 实例。

声明

```
  explicit unique_lock(mutex_type& m);
```

效果
通过提供的互斥量，构造一个 std::unique_lock 实例，且调用m.lock()。

抛出
m.lock()抛出的任何异常。

后置条件
this->owns_lock()==true, this->mutex()==&m.

std::unique_lock 获取锁的构造函数

使用相关互斥量和持有的锁，构造一个 std::unique_lock 实例。

声明

```
unique_lock(mutex_type& m,std::adopt_lock_t);
```

先决条件

调用线程必须持有m上的锁。

效果

通过提供的互斥量和已经拥有m上的锁，构造一个 `std::unique_lock` 实例。

抛出

无

后置条件

this->owns_lock()==true, this->mutex()==&m.

std::unique_lock 递延锁的构造函数

使用相关互斥量和非持有的锁，构造一个 `std::unique_lock` 实例。

声明

```
unique_lock(mutex_type& m,std::defer_lock_t) noexcept;
```

效果

构造的 `std::unique_lock` 实例引用了提供的互斥量。

抛出

无

后置条件

this->owns_lock()==false, this->mutex()==&m.

std::unique_lock 尝试获取锁的构造函数

使用提供的互斥量，并尝试从互斥量上获取锁，从而构造一个 `std::unique_lock` 实例。

声明

```
unique_lock(mutex_type& m,std::try_to_lock_t);
```

先决条件

使 `std::unique_lock` 实例化的Mutex类型，必须符合Loackable的需求。

效果

构造的 `std::unique_lock` 实例引用了提供的互斥量，且调用m.try_lock()。

抛出

无

后置条件

this->owns_lock()将返回m.try_lock()的结果，且this->mutex()==&m。

std::unique_lock 在给定时长内尝试获取锁的构造函数

使用提供的互斥量，并尝试从互斥量上获取锁，从而构造一个 `std::unique_lock` 实例。

声明

```
template<typename Rep,typename Period>
unique_lock(
    mutex_type& m,
    std::chrono::duration<Rep,Period> const& relative_time);
```

先决条件

使 `std::unique_lock` 实例化的Mutex类型，必须符合TimedLockable的需求。

效果

构造的 `std::unique_lock` 实例引用了提供的互斥量，且调用m.try_lock_for(relative_time)。

抛出

无

后置条件

this->owns_lock()将返回m.try_lock_for()的结果，且this->mutex()==&m。

std::unique_lock 在给定时间点内尝试获取锁的构造函数

使用提供的互斥量，并尝试从互斥量上获取锁，从而构造一个 `std::unique_lock` 实例。

声明

```
template<typename Clock,typename Duration>
unique_lock(
    mutex_type& m,
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

先决条件

使 `std::unique_lock` 实例化的Mutex类型，必须符合TimedLockable的需求。

效果

构造的 `std::unique_lock` 实例引用了提供的互斥量，且调用m.try_lock_until(absolute_time)。

抛出

无

后置条件

this->owns_lock()将返回m.try_lock_until()的结果，且this->mutex()==&m。

std::unique_lock 移动构造函数

将一个已经构造 `std::unique_lock` 实例的所有权，转移到新的 `std::unique_lock` 实例上去。

声明

```
unique_lock(unique_lock&& other) noexcept;
```

先决条件

使 `std::unique_lock` 实例化的Mutex类型，必须符合TimedLockable的需求。

效果

构造的 `std::unique_lock` 实例。当other在函数调用的时候拥有互斥量上的锁，那么该锁的所有权将被转移到新构建的 `std::unique_lock` 对象当中去。

后置条件

对于新构建的 `std::unique_lock` 对象x，x.mutex等价与在构造函数调用前的other.mutex()，并

且x.owns_lock()等价于函数调用前的other.owns_lock()。在调用函数后，other.mutex()==NULL，other.owns_lock()=false。

抛出
无

NOTE `std::unique_lock` 对象是不可CopyConstructible(拷贝构造)，所以这里没有拷贝构造函数，只有移动构造函数。

std::unique_lock 移动赋值操作

将一个已经构造 `std::unique_lock` 实例的所有权，转移到新的 `std::unique_lock` 实例上去。

声明

```
unique_lock& operator=(unique_lock&& other) noexcept;
```

效果
当this->owns_lock()返回true时，调用this->unlock()。如果other拥有mutex上的锁，那么这个所将归*this所有。

后置条件
this->mutex()等于在为进行赋值前的other.mutex()，并且this->owns_lock()的值与进行赋值操作前的other.owns_lock()相等。other.mutex()==NULL, other.owns_lock()==false。

抛出
无

NOTE `std::unique_lock` 对象是不可CopyAssignable(拷贝赋值)，所以这里没有拷贝赋值函数，只有移动赋值函数。

std::unique_lock 析构函数

销毁一个 `std::unique_lock` 实例，如果该实例拥有锁，那么会将相关互斥量进行解锁。

声明

```
~unique_lock();
```

效果

当this->owns_lock()返回true时，调用this->mutex()->unlock()。

抛出

无

std::unique_lock::swap 成员函数

交换 `std::unique_lock` 实例中相关的所有权。

声明

```
void swap(unique_lock& other) noexcept;
```

效果

如果other在调用该函数前拥有互斥量上的锁，那么这个锁将归 `*this` 所有。如果 `*this` 在调用哎函数前拥有互斥量上的锁，那么这个锁将归other所有。

抛出

无

std::unique_lock 上非成员函数swap

交换 `std::unique_lock` 实例中相关的所有权。

声明

```
void swap(unique_lock& lhs,unique_lock& rhs) noexcept;
```

效果

lhs.swap(rhs)

抛出

无

std::unique_lock::lock 成员函数

获取与*this相关互斥量上的锁。

声明

```
void lock();
```

先决条件
this->mutex()!=NULL, this->owns_lock()==false.

效果
调用this->mutex()->lock()。

抛出
抛出任何this->mutex()->lock()所抛出的异常。当this->mutex()==NULL，抛出 `std::sytem_error` 类型异常，错误码为 `std::errc::operation_not_permitted` 。当this->owns_lock()==true时，抛出 `std::system_error` ，错误码为 `std::errc::resource_deadlock_would_occur` 。

后置条件
this->owns_lock()==true。

std::unique_lock::try_lock 成员函数

尝试获取与*this相关互斥量上的锁。

声明

```
bool try_lock();
```

先决条件
`std::unique_lock` 实例化说是用的Mutex类型，必须满足Lockable需求。this->mutex()!=NULL, this->owns_lock()==false。

效果
调用this->mutex()->try_lock()。

抛出
抛出任何this->mutex()->try_lock()所抛出的异常。当this->mutex()==NULL，抛出 `std::sytem_error` 类型异常，错误码为 `std::errc::operation_not_permitted` 。当this-

>owns_lock()==true时，抛出 `std::system_error` ，错误码为 `std::errc::resource_deadlock_would_occur` 。

后置条件

当函数返回true时，this->ows_lock()==true，否则this->owns_lock()==false。

std::unique_lock::unlock 成员函数

释放与*this相关互斥量上的锁。

声明

```
void unlock();
```

先决条件

this->mutex()!=NULL, this->owns_lock()==true。

抛出

抛出任何this->mutex()->unlock()所抛出的异常。当this->owns_lock()==false时，抛出 `std::system_error` ，错误码为 `std::errc::operation_not_permitted` 。

后置条件

this->owns_lock()==false。

std::unique_lock::try_lock_for 成员函数

在指定时间内尝试获取与*this相关互斥量上的锁。

声明

```
template<typename Rep, typename Period>
bool try_lock_for(
    std::chrono::duration<Rep,Period> const& relative_time);
```

先决条件

`std::unique_lock` 实例化说是用的Mutex类型，必须满足TimedLockable需求。this->mutex()!=NULL, this->owns_lock()==false。

效果

调用this->mutex()->try_lock_for(relative_time)。

返回

当this->mutex()->try_lock_for()返回true，返回true，否则返回false。

抛出

抛出任何this->mutex()->try_lock_for()所抛出的异常。当this->mutex()==NULL，抛出 `std::sytem_error` 类型异常，错误码为 `std::errc::operation_not_permitted` 。当this->owns_lock()==true时，抛出 `std::system_error` ，错误码为 `std::errc::resource_deadlock_would_occur` 。

后置条件

当函数返回true时，this->ows_lock()==true，否则this->owns_lock()==false。

std::unique_lock::try_lock_until 成员函数

在指定时间点尝试获取与*this相关互斥量上的锁。

声明

```
template<typename Clock, typename Duration>
bool try_lock_until(
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

先决条件

`std::unique_lock` 实例化说是用的Mutex类型，必须满足TimedLockable需求。this->mutex()!=NULL, this->owns_lock()==false。

效果

调用this->mutex()->try_lock_until(absolute_time)。

返回

当this->mutex()->try_lock_for()返回true，返回true，否则返回false。

抛出

抛出任何this->mutex()->try_lock_for()所抛出的异常。当this->mutex()==NULL，抛出 `std::sytem_error` 类型异常，错误码为 `std::errc::operation_not_permitted` 。当this-

>owns_lock()==true时，抛出 `std::system_error` ，错误码为 `std::errc::resource_deadlock_would_occur` 。

后置条件
当函数返回true时，this->ows_lock()==true，否则this->owns_lock()==false。

std::unique_lock::operator bool成员函数

检查*this是否拥有一个互斥量上的锁。

声明

```
explicit operator bool() const noexcept;
```

返回
this->owns_lock()

抛出
无

NOTE 这是一个explicit转换操作，所以当这样的操作在上下文中只能被隐式的调用，所返回的结果需要被当做一个布尔量进行使用，而非仅仅作为整型数0或1。

std::unique_lock::owns_lock 成员函数

检查*this是否拥有一个互斥量上的锁。

声明

```
bool owns_lock() const noexcept;
```

返回
当*this持有一个互斥量的锁，返回true；否则，返回false。

抛出
无

std::unique_lock::mutex 成员函数

当*this具有相关互斥量时，返回这个互斥量

声明

```
mutex_type* mutex() const noexcept;
```

返回
当*this有相关互斥量，则返回该互斥量；否则，返回NULL。

抛出
无

std::unique_lock::release 成员函数

当*this具有相关互斥量时，返回这个互斥量，并将这个互斥量进行释放。

声明

```
mutex_type* release() noexcept;
```

效果
将*this与相关的互斥量之间的关系解除，同时解除所有持有锁的所有权。

返回
返回与*this相关的互斥量指针，如果没有相关的互斥量，则返回NULL。

后置条件
this->mutex()==NULL, this->owns_lock()==false。

抛出
无

NOTE 如果this->owns_lock()在调用该函数前返回true，那么调用者则有责任里解除互斥量上的锁。

D.5.7 std::lock函数模板

`std::lock` 函数模板提供同时锁住多个互斥量的功能，且不会有因改变锁的一致性而导致的死锁。

声明

```
template<typename LockableType1,typename... LockableType2>
void lock(LockableType1& m1,LockableType2& m2...);
```

先决条件
提供的可锁对象LockableType1, LockableType2...，需要满足Lockable的需求。

效果
使用未指定顺序调用lock(),try_lock()获取每个可锁对象(m1, m2...)上的锁，还有unlock()成员来避免这个类型陷入死锁。

后置条件
当前线程拥有提供的所有可锁对象上的锁。

抛出
任何lock(), try_lock()和unlock()抛出的异常。

NOTE 如果一个异常由 `std::lock` 所传播开来，当可锁对象上有锁被lock()或try_lock()获取，那么unlock()会使用在这些可锁对象上。

D.5.8 std::try_lock函数模板

`std::try_lock` 函数模板允许尝试获取一组可锁对象上的锁，所以要不全部获取，要不一个都不获取。

声明

```
template<typename LockableType1,typename... LockableType2>
int try_lock(LockableType1& m1,LockableType2& m2...);
```

先决条件
提供的可锁对象LockableType1, LockableType2...，需要满足Lockable的需求。

效果

使用try_lock()尝试从提供的可锁对象m1,m2...上逐个获取锁。当锁在之前获取过，但被当前线程使用unlock()对相关可锁对象进行了释放后，try_lock()会返回false或抛出一个异常。

返回

当所有锁都已获取(每个互斥量调用try_lock()返回true)，则返回-1，否则返回以0为基数的数字，其值为调用try_lock()返回false的个数。

后置条件

当函数返回-1，当前线程获取从每个可锁对象上都获取一个锁。否则，通过该调用获取的任何锁都将被释放。

抛出

try_lock()抛出的任何异常。

NOTE 如果一个异常由 `std::try_lock` 所传播开来，则通过try_lock()获取锁对象，将会调用unlock()解除对锁的持有。

D.5.9 std::once_flag类

`std::once_flag` 和 `std::call_once` 一起使用，为了保证某特定函数只执行一次(即使有多个线程在并发的调用该函数)。

`std::once_flag` 实例是不能CopyConstructible(拷贝构造)，CopyAssignable(拷贝赋值)，MoveConstructible(移动构造)，以及MoveAssignable(移动赋值)。

类型定义

```
struct once_flag
{
  constexpr once_flag() noexcept;

  once_flag(once_flag const& ) = delete;
  once_flag& operator=(once_flag const& ) = delete;
};
```

std::once_flag 默认构造函数

`std::once_flag` 默认构造函数创建了一个新的 `std::once_flag` 实例(并包含一个状态，这个状态表示相关函数没有被调用)。

声明

```
constexpr once_flag() noexcept;
```

效果

`std::once_flag` 默认构造函数创建了一个新的 `std::once_flag` 实例(并包含一个状态，这个状态表示相关函数没有被调用)。因为这是一个constexpr构造函数，在构造的静态初始部分，实例是静态存储的，这样就避免了条件竞争和初始化顺序的问题。

D.5.10 std::call_once函数模板

`std::call_once` 和 `std::once_flag` 一起使用，为了保证某特定函数只执行一次(即使有多个线程在并发的调用该函数)。

声明

```
template<typename Callable,typename... Args>
void call_once(std::once_flag& flag,Callable func,Args args...);
```

先决条件

表达式 `INVOKE(func,args)` 提供的func和args必须是合法的。Callable和每个Args的成员都是可MoveConstructible(移动构造)。

效果

在同一个 `std::once_flag` 对象上调用 `std::call_once` 是串行的。如果之前没有在同一个 `std::once_flag` 对象上调用过 `std::call_once` ，参数func(或副本)被调用，就像INVOKE(func, args),并且只有可调用的func不抛出任何异常时，调用 `std::call_once` 才是有效的。当有异常抛出，异常会被调用函数进行传播。如果之前在 `std::once_flag` 上的 `std::call_once` 是有效的，那么再次调用 `std::call_once` 将不会在调用func。

同步

在 `std::once_flag` 上完成对 `std::call_once` 的调用的先决条件是，后续所有对 `std::call_once` 调用都在同一 `std::once_flag` 对象。

抛出

当效果没有达到，或任何异常由调用func而传播，则抛出 `std::system_error` 。