D.2 condition_variable头文件

<condition_variable>头文件提供了条件变量的定义。其作为基本同步机制，允许被阻塞的线程在某些条件达成或超时时，解除阻塞继续执行。

头文件内容

```
namespace std
{
  enum class cv_status { timeout, no_timeout };

  class condition_variable;
  class condition_variable_any;
}
```

D.2.1 std::condition_variable类

`std::condition_variable` 允许阻塞一个线程，直到条件达成。

`std::condition_variable` 实例不支持CopyAssignable(拷贝赋值), CopyConstructible(拷贝构造), MoveAssignable(移动赋值)和 MoveConstructible(移动构造)。

类型定义

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

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

```

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

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

  template <typename Predicate>
  void wait(std::unique_lock<std::mutex>& lock,Predicate pred);

  template <typename Clock, typename Duration>
  cv_status wait_until(
       std::unique_lock<std::mutex>& lock,
       const std::chrono::time_point<Clock, Duration>& absolute_time);

  template <typename Clock, typename Duration, typename Predicate>
  bool wait_until(
       std::unique_lock<std::mutex>& lock,
       const std::chrono::time_point<Clock, Duration>& absolute_time,
       Predicate pred);

  template <typename Rep, typename Period>
  cv_status wait_for(
       std::unique_lock<std::mutex>& lock,
       const std::chrono::duration<Rep, Period>& relative_time);

  template <typename Rep, typename Period, typename Predicate>
  bool wait_for(
       std::unique_lock<std::mutex>& lock,
       const std::chrono::duration<Rep, Period>& relative_time,
       Predicate pred);
};

void notify_all_at_thread_exit(condition_variable&,unique_lock<mutex>);
```

std::condition_variable 默认构造函数

构造一个 `std::condition_variable` 对象。

声明

```
condition_variable();
```

效果

构造一个新的 `std::condition_variable` 实例。

抛出

当条件变量无法够早的时候，将会抛出一个 `std::system_error` 异常。

std::condition_variable 析构函数

销毁一个 `std::condition_variable` 对象。

声明

```
~condition_variable();
```

先决条件

之前没有使用*this总的wait(),wait_for()或wait_until()阻塞过线程。

效果

销毁*this。

抛出

无

std::condition_variable::notify_one 成员函数

唤醒一个等待当前 `std::condition_variable` 实例的线程。

声明

```
void notify_one() noexcept;
```

效果

唤醒一个等待*this的线程。如果没有线程在等待，那么调用没有任何效果。

抛出

当效果没有达成，就会抛出 `std::system_error` 异常。

同步

`std::condition_variable` 实例中的notify_one(),notify_all(),wait(),wait_for()和wait_until()都是序列化函数(串行调用)。调用notify_one()或notify_all()只能唤醒正在等待中的线程。

std::condition_variable::notify_all 成员函数

唤醒所有等待当前 `std::condition_variable` 实例的线程。

声明

```
void notify_all() noexcept;
```

效果

唤醒所有等待*this的线程。如果没有线程在等待，那么调用没有任何效果。

抛出

当效果没有达成，就会抛出 `std::system_error` 异常

同步

`std::condition_variable` 实例中的notify_one(),notify_all(),wait(),wait_for()和wait_until()都是序列化函数(串行调用)。调用notify_one()或notify_all()只能唤醒正在等待中的线程。

std::condition_variable::wait 成员函数

通过 `std::condition_variable` 的notify_one()、notify_all()或伪唤醒结束等待。

等待

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

先决条件

当线程调用wait()即可获得锁的所有权,lock.owns_lock()必须为true。

效果

自动解锁lock对象，对于线程等待线程，当其他线程调用notify_one()或notify_all()时被唤醒，亦或该线程处于伪唤醒状态。在wait()返回前，lock对象将会再次上锁。

抛出

当效果没有达成的时候，将会抛出 `std::system_error` 异常。当lock对象在调用wait()阶段被解锁，那么当wait()退出的时候lock会再次上锁，即使函数是通过异常的方式退出。

NOTE:伪唤醒意味着一个线程调用wait()后，在没有其他线程调用notify_one()或notify_all()时，还处以苏醒状态。因此，建议对wait()进行重载，在可能的情况下使用一个谓词。否则，建议wait()使用循环检查与条件变量相关的谓词。

同步

`std::condition_variable` 实例中的notify_one(),notify_all(),wait(),wait_for()和wait_until()都是序列化函数(串行调用)。调用notify_one()或notify_all()只能唤醒正在等待中的线程。

std::condition_variable::wait 需要一个谓词的成员函数重载

等待 `std::condition_variable` 上的notify_one()或notify_all()被调用，或谓词为true的情况，来唤醒线程。

声明

```
template<typename Predicate>
void wait(std::unique_lock<std::mutex>& lock,Predicate pred);
```

先决条件

pred()谓词必须是合法的，并且需要返回一个值，这个值可以和bool互相转化。当线程调用wait()即可获得锁的所有权,lock.owns_lock()必须为true。

效果

正如

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

抛出

pred中可以抛出任意异常，或者当效果没有达到的时候，抛出 `std::system_error` 异常。

NOTE:潜在的伪唤醒意味着不会指定pred调用的次数。通过lock进行上锁，pred经常会被互斥量引用所调用，并且函数必须返回(只能返回)一个值，在 `(bool)pred()` 评估后，返回true。

同步

`std::condition_variable` 实例中的notify_one(),notify_all(),wait(),wait_for()和wait_until()都是序列化函数(串行调用)。调用notify_one()或notify_all()只能唤醒正在等待中的线程。

std::condition_variable::wait_for 成员函数

`std::condition_variable` 在调用notify_one()、调用notify_all()、超时或线程伪唤醒时，结束等待。

声明

```
template<typename Rep,typename Period>
cv_status wait_for(
    std::unique_lock<std::mutex>& lock,
    std::chrono::duration<Rep,Period> const& relative_time);
```

先决条件

当线程调用wait()即可获得锁的所有权,lock.owns_lock()必须为true。

效果

当其他线程调用notify_one()或notify_all()函数时，或超出了relative_time的时间，亦或是线程被伪唤醒，则将lock对象自动解锁，并将阻塞线程唤醒。当wait_for()调用返回前，lock对象会再次上锁。

返回

线程被notify_one()、notify_all()或伪唤醒唤醒时，会返回 `std::cv_status::no_timeout` ；反之，则返回 `std::cv_status::timeout` 。

抛出

当效果没有达成的时候，会抛出 `std::system_error` 异常。当lock对象在调用wait_for()函数前解锁，那么lock对象会在wait_for()退出前再次上锁，即使函数是以异常的方式退出。

NOTE:伪唤醒意味着，一个线程在调用wait_for()的时候，即使没有其他线程调用notify_one()和notify_all()函数，也处于苏醒状态。因此，这里建议重载wait_for()函数，重载函数可以使用谓词。要不，则建议wait_for()使用循环的方式对与谓词相关的条件变量进行检查。在这样做的时候还需要小心，以确保超时部分依旧有效；wait_until()可能适合更多的情况。这样的话，线程阻塞的时间就要比指定的时间长了。在有这样可能性的地方，流逝的时间是由稳定时钟决定。

同步

`std::condition_variable` 实例中的notify_one(),notify_all(),wait(),wait_for()和wait_until()都是序列化函数(串行调用)。调用notify_one()或notify_all()只能唤醒正在等待中的线程。

std::condition_variable::wait_for 需要一个谓词的成员函数重载

`std::condition_variable` 在调用notify_one()、调用notify_all()、超时或线程伪唤醒时，结束等待。

声明

```
template<typename Rep,typename Period,typename Predicate>
bool wait_for(
    std::unique_lock<std::mutex>& lock,
    std::chrono::duration<Rep,Period> const& relative_time,
    Predicate pred);
```

先决条件

pred()谓词必须是合法的，并且需要返回一个值，这个值可以和bool互相转化。当线程调用wait()即可获得锁的所有权,lock.owns_lock()必须为true。

效果

等价于

```
internal_clock::time_point end=internal_clock::now()+relative_time;
while(!pred())
{
  std::chrono::duration<Rep,Period> remaining_time=
      end-internal_clock::now();
  if(wait_for(lock,remaining_time)==std::cv_status::timeout)
      return pred();
}
return true;
```

返回

当pred()为true，则返回true；当超过relative_time并且pred()返回false时，返回false。

NOTE:潜在的伪唤醒意味着不会指定pred调用的次数。通过lock进行上锁，pred经常会被互斥量引用所调用，并且函数必须返回(只能返回)一个值，在 `(bool)pred()` 评估后返回true，或在指定时

间relative_time内完成。线程阻塞的时间就要比指定的时间长了。在有这样可能性的地方，流逝的时间是由稳定时钟决定。

抛出

当效果没有达成时，会抛出 `std::system_error` 异常或者由pred抛出任意异常。

同步

`std::condition_variable` 实例中的notify_one(),notify_all(),wait(),wait_for()和wait_until()都是序列化函数(串行调用)。调用notify_one()或notify_all()只能唤醒正在等待中的线程。

std::condition_variable::wait_until 成员函数

`std::condition_variable` 在调用notify_one()、调用notify_all()、指定时间内达成条件或线程伪唤醒时，结束等待。

声明

```
template<typename Clock,typename Duration>
cv_status wait_until(
    std::unique_lock<std::mutex>& lock,
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

先决条件

当线程调用wait()即可获得锁的所有权,lock.owns_lock()必须为true。

效果

当其他线程调用notify_one()或notify_all()函数，或Clock::now()返回一个大于或等于absolute_time的时间，亦或线程伪唤醒，lock都将自动解锁，并且唤醒阻塞的线程。在wait_until()返回之前lock对象会再次上锁。

返回

线程被notify_one()、notify_all()或伪唤醒唤醒时，会返回 `std::cv_status::no_timeout` ；反之，则返回 `std::cv_status::timeout` 。

抛出

当效果没有达成的时候，会抛出 `std::system_error` 异常。当lock对象在调用wait_for()函数前解锁，那么lock对象会在wait_for()退出前再次上锁，即使函数是以异常的方式退出。

NOTE:伪唤醒意味着一个线程调用wait()后，在没有其他线程调用notify_one()或notify_all()时，还处以苏醒状态。因此，这里建议重载wait_until()函数，重载函数可以使用谓词。要不，则建议wait_until()使用循环的方式对与谓词相关的条件变量进行检查。这里不保证线程会被阻塞多长时间，只有当函数返回false后(Clock::now()的返回值大于或等于absolute_time)，线程才能解除阻塞。

同步

`std::condition_variable` 实例中的notify_one(),notify_all(),wait(),wait_for()和wait_until()都是序列化函数(串行调用)。调用notify_one()或notify_all()只能唤醒正在等待中的线程。

std::condition_variable::wait_until 需要一个谓词的成员函数重载

`std::condition_variable` 在调用notify_one()、调用notify_all()、谓词返回true或指定时间内达到条件，结束等待。

声明

```
template<typename Clock,typename Duration,typename Predicate>
bool wait_until(
    std::unique_lock<std::mutex>& lock,
    std::chrono::time_point<Clock,Duration> const& absolute_time,
    Predicate pred);
```

先决条件

pred()必须是合法的，并且其返回值能转换为bool值。当线程调用wait()即可获得锁的所有权,lock.owns_lock()必须为true。

效果

等价于

```
while(!pred())
{
  if(wait_until(lock,absolute_time)==std::cv_status::timeout)
    return pred();
}
return true;
```

返回

当调用pred()返回true时，返回true；当Clock::now()的时间大于或等于指定的时间

absolute_time，并且pred()返回false时，返回false。

NOTE:潜在的伪唤醒意味着不会指定pred调用的次数。通过lock进行上锁，pred经常会被互斥量引用所调用，并且函数必须返回(只能返回)一个值，在 `(bool)pred()` 评估后返回true，或Clock::now()返回的时间大于或等于absolute_time。这里不保证调用线程将被阻塞的时长，只有当函数返回false后(Clock::now()返回一个等于或大于absolute_time的值)，线程接触阻塞。

抛出

当效果没有达成时，会抛出 `std::system_error` 异常或者由pred抛出任意异常。

同步

`std::condition_variable` 实例中的notify_one(),notify_all(),wait(),wait_for()和wait_until()都是序列化函数(串行调用)。调用notify_one()或notify_all()只能唤醒正在等待中的线程。

std::notify_all_at_thread_exit 非成员函数

当当前调用函数的线程退出时，等待 `std::condition_variable` 的所有线程将会被唤醒。

声明

```
void notify_all_at_thread_exit(
  condition_variable& cv,unique_lock<mutex> lk);
```

先决条件

当线程调用wait()即可获得锁的所有权,lk.owns_lock()必须为true。lk.mutex()需要返回的值要与并发等待线程相关的任意cv中锁对象提供的wait(),wait_for()或wait_until()相同。

效果

将lk的所有权转移到内部存储中，并且当有线程退出时，安排被提醒的cv类。这里的提醒等价于

```
lk.unlock();
cv.notify_all();
```

抛出

当效果没有达成时，抛出 `std::system_error` 异常。

NOTE:在线程退出前，掌握着锁的所有权，所以这里要避免死锁发生。这里建议调用该函数的线程应该尽快退出，并且在该线程可以执行一些阻塞的操作。用户必须保证等地线程不会错误的将唤醒线程当做已退出的线程，特别是伪唤醒。可以通过等待线程上的谓词测试来实现这一功能，在

互斥量保护的情况下，只有谓词返回true时线程才能被唤醒，并且在调用notify_all_at_thread_exit(std::condition_variable_any类中函数)前是不会释放锁。

D.2.2 std::condition_variable_any类

`std::condition_variable_any` 类允许线程等待某一条件为true的时候继续运行。不过 `std::condition_variable` 只能和 `std::unique_lock<std::mutex>` 一起使用，`std::condition_variable_any` 可以和任意可上锁(Lockable)类型一起使用。

`std::condition_variable_any` 实例不能进行拷贝赋值(CopyAssignable)、拷贝构造(CopyConstructible)、移动赋值(MoveAssignable)或移动构造(MoveConstructible)。

类型定义

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

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

  void notify_one() noexcept;
  void notify_all() noexcept;

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

  template <typename Lockable, typename Predicate>
  void wait(Lockable& lock, Predicate pred);

  template <typename Lockable, typename Clock,typename Duration>
  std::cv_status wait_until(
      Lockable& lock,
      const std::chrono::time_point<Clock, Duration>& absolute_time);

  template <
```

```
        typename Lockable, typename Clock,
        typename Duration, typename Predicate>
    bool wait_until(
        Lockable& lock,
        const std::chrono::time_point<Clock, Duration>& absolute_time,
        Predicate pred);

    template <typename Lockable, typename Rep, typename Period>
    std::cv_status wait_for(
        Lockable& lock,
        const std::chrono::duration<Rep, Period>& relative_time);

    template <
        typename Lockable, typename Rep,
        typename Period, typename Predicate>
    bool wait_for(
        Lockable& lock,
        const std::chrono::duration<Rep, Period>& relative_time,
        Predicate pred);
};
```

std::condition_variable_any 默认构造函数

构造一个 std::condition_variable_any 对象。

声明

```
condition_variable_any();
```

效果

构造一个新的 std::condition_variable_any 实例。

抛出

当条件变量构造成功，将抛出 std::system_error 异常。

std::condition_variable_any 析构函数

销毁 std::condition_variable_any 对象。

声明

```
~condition_variable_any();
```

先决条件
之前没有使用*this总的wait(),wait_for()或wait_until()阻塞过线程。

效果
销毁*this。

抛出
无

std::condition_variable_any::notify_one 成员函数

`std::condition_variable_any` 唤醒一个等待该条件变量的线程。

声明

```
void notify_all() noexcept;
```

效果
唤醒一个等待*this的线程。如果没有线程在等待，那么调用没有任何效果

抛出
当效果没有达成，就会抛出std::system_error异常。

同步
`std::condition_variable` 实例中的notify_one(),notify_all(),wait(),wait_for()和wait_until()都是序列化函数(串行调用)。调用notify_one()或notify_all()只能唤醒正在等待中的线程。

std::condition_variable_any::notify_all 成员函数

唤醒所有等待当前 `std::condition_variable_any` 实例的线程。

声明

```
void notify_all() noexcept;
```

效果

唤醒所有等待*this的线程。如果没有线程在等待，那么调用没有任何效果

抛出

当效果没有达成，就会抛出std::system_error异常。

同步

`std::condition_variable` 实例中的notify_one(),notify_all(),wait(),wait_for()和wait_until()都是序列化函数(串行调用)。调用notify_one()或notify_all()只能唤醒正在等待中的线程。

std::condition_variable_any::wait 成员函数

通过 `std::condition_variable_any` 的notify_one()、notify_all()或伪唤醒结束等待。

声明

```
template<typename Lockable>
void wait(Lockable& lock);
```

先决条件

Lockable类型需要能够上锁，lock对象拥有一个锁。

效果

自动解锁lock对象，对于线程等待线程，当其他线程调用notify_one()或notify_all()时被唤醒，亦或该线程处于伪唤醒状态。在wait()返回前，lock对象将会再次上锁。

抛出

当效果没有达成的时候，将会抛出 `std::system_error` 异常。当lock对象在调用wait()阶段被解锁，那么当wait()退出的时候lock会再次上锁，即使函数是通过异常的方式退出。

NOTE:伪唤醒意味着一个线程调用wait()后，在没有其他线程调用notify_one()或notify_all()时，还处以苏醒状态。因此，建议对wait()进行重载，在可能的情况下使用一个谓词。否则，建议wait()使用循环检查与条件变量相关的谓词。

同步

std::condition_variable_any实例中的notify_one(),notify_all(),wait(),wait_for()和wait_until()都是序列化函数(串行调用)。调用notify_one()或notify_all()只能唤醒正在等待中的线程。

std::condition_variable_any::wait 需要一个谓词的成员函数重载

等待 `std::condition_variable_any` 上的notify_one()或notify_all()被调用，或谓词为true的情况，来唤醒线程。

声明

```
template<typename Lockable,typename Predicate>
void wait(Lockable& lock,Predicate pred);
```

先决条件

pred()谓词必须是合法的，并且需要返回一个值，这个值可以和bool互相转化。当线程调用wait()即可获得锁的所有权,lock.owns_lock()必须为true。

效果

正如

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

抛出

pred中可以抛出任意异常，或者当效果没有达到的时候，抛出 `std::system_error` 异常。

NOTE:潜在的伪唤醒意味着不会指定pred调用的次数。通过lock进行上锁，pred经常会被互斥量引用所调用，并且函数必须返回(只能返回)一个值，在 `(bool)pred()` 评估后，返回true。

同步

`std::condition_variable_any` 实例中的notify_one(),notify_all(),wait(),wait_for()和wait_until()都是序列化函数(串行调用)。调用notify_one()或notify_all()只能唤醒正在等待中的线程。

std::condition_variable_any::wait_for 成员函数

`std::condition_variable_any` 在调用notify_one()、调用notify_all()、超时或线程伪唤醒时，结束等待。

声明

```
template<typename Lockable,typename Rep,typename Period>
std::cv_status wait_for(
```

```
    Lockable& lock,
    std::chrono::duration<Rep,Period> const& relative_time);
```

先决条件

当线程调用wait()即可获得锁的所有权,lock.owns_lock()必须为true。

效果

当其他线程调用notify_one()或notify_all()函数时，或超出了relative_time的时间，亦或是线程被伪唤醒，则将lock对象自动解锁，并将阻塞线程唤醒。当wait_for()调用返回前，lock对象会再次上锁。

返回

线程被notify_one()、notify_all()或伪唤醒唤醒时，会返回 `std::cv_status::no_timeout` ；反之，则返回std::cv_status::timeout。

抛出

当效果没有达成的时候，会抛出 `std::system_error` 异常。当lock对象在调用wait_for()函数前解锁，那么lock对象会在wait_for()退出前再次上锁，即使函数是以异常的方式退出。

NOTE:伪唤醒意味着，一个线程在调用wait_for()的时候，即使没有其他线程调用notify_one()和notify_all()函数，也处于苏醒状态。因此，这里建议重载wait_for()函数，重载函数可以使用谓词。要不，则建议wait_for()使用循环的方式对与谓词相关的条件变量进行检查。在这样做的时候还需要小心，以确保超时部分依旧有效；wait_until()可能适合更多的情况。这样的话，线程阻塞的时间就要比指定的时间长了。在有这样可能性的地方，流逝的时间是由稳定时钟决定。

同步

`std::condition_variable_any` 实例中的notify_one(),notify_all(),wait(),wait_for()和wait_until()都是序列化函数(串行调用)。调用notify_one()或notify_all()只能唤醒正在等待中的线程。

std::condition_variable_any::wait_for 需要一个谓词的成员函数重载

`std::condition_variable_any` 在调用notify_one()、调用notify_all()、超时或线程伪唤醒时，结束等待。

声明

```
template<typename Lockable,typename Rep,
    typename Period, typename Predicate>
bool wait_for(
    Lockable& lock,
```

```
    std::chrono::duration<Rep,Period> const& relative_time,
    Predicate pred);
```

先决条件

pred()谓词必须是合法的，并且需要返回一个值，这个值可以和bool互相转化。当线程调用wait()即可获得锁的所有权,lock.owns_lock()必须为true。

效果

正如

```
internal_clock::time_point end=internal_clock::now()+relative_time;
while(!pred())
{
  std::chrono::duration<Rep,Period> remaining_time=
      end-internal_clock::now();
  if(wait_for(lock,remaining_time)==std::cv_status::timeout)
      return pred();
}
return true;
```

返回

当pred()为true，则返回true；当超过relative_time并且pred()返回false时，返回false。

NOTE: 潜在的伪唤醒意味着不会指定pred调用的次数。通过lock进行上锁，pred经常会被互斥量引用所调用，并且函数必须返回(只能返回)一个值，在(bool)pred()评估后返回true，或在指定时间relative_time内完成。线程阻塞的时间就要比指定的时间长了。在有这样可能性的地方，流逝的时间是由稳定时钟决定。

抛出

当效果没有达成时，会抛出 `std::system_error` 异常或者由pred抛出任意异常。

同步

`std::condition_variable_any` 实例中的notify_one(),notify_all(),wait(),wait_for()和wait_until()都是序列化函数(串行调用)。调用notify_one()或notify_all()只能唤醒正在等待中的线程。

std::condition_variable_any::wait_until 成员函数

`std::condition_variable_any` 在调用notify_one()、调用notify_all()、指定时间内达成条件或线程伪唤醒时，结束等待

声明

```
template<typename Lockable,typename Clock,typename Duration>
std::cv_status wait_until(
    Lockable& lock,
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

先决条件

Lockable类型需要能够上锁，lock对象拥有一个锁。

效果

当其他线程调用notify_one()或notify_all()函数，或Clock::now()返回一个大于或等于absolute_time的时间，亦或线程伪唤醒，lock都将自动解锁，并且唤醒阻塞的线程。在wait_until()返回之前lock对象会再次上锁。

返回

线程被notify_one()、notify_all()或伪唤醒唤醒时，会返回std::cv_status::no_timeout；反之，则返回 `std::cv_status::timeout` 。

抛出

当效果没有达成的时候，会抛出 `std::system_error` 异常。当lock对象在调用wait_for()函数前解锁，那么lock对象会在wait_for()退出前再次上锁，即使函数是以异常的方式退出。

NOTE:伪唤醒意味着一个线程调用wait()后，在没有其他线程调用notify_one()或notify_all()时，还处以苏醒状态。因此，这里建议重载wait_until()函数，重载函数可以使用谓词。要不，则建议wait_until()使用循环的方式对与谓词相关的条件变量进行检查。这里不保证线程会被阻塞多长时间，只有当函数返回false后(Clock::now()的返回值大于或等于absolute_time)，线程才能解除阻塞。

同步 `std::condition_variable_any` 实例中的notify_one(),notify_all(),wait(),wait_for()和wait_until()都是序列化函数(串行调用)。调用notify_one()或notify_all()只能唤醒正在等待中的线程。

std::condition_variable_any::wait_unti 需要一个谓词的成员函数重载

`std::condition_variable_any` 在调用notify_one()、调用notify_all()、谓词返回true或指定时间内达到条件，结束等待。

声明

```
template<typename Lockable,typename Clock,
    typename Duration, typename Predicate>
bool wait_until(
    Lockable& lock,
    std::chrono::time_point<Clock,Duration> const& absolute_time,
    Predicate pred);
```

先决条件

pred()必须是合法的，并且其返回值能转换为bool值。当线程调用wait()即可获得锁的所有权,lock.owns_lock()必须为true。

效果

等价于

```
while(!pred())
{
  if(wait_until(lock,absolute_time)==std::cv_status::timeout)
    return pred();
}
return true;
```

返回

当调用pred()返回true时，返回true；当Clock::now()的时间大于或等于指定的时间absolute_time，并且pred()返回false时，返回false。

NOTE：潜在的伪唤醒意味着不会指定pred调用的次数。通过lock进行上锁，pred经常会被互斥量引用所调用，并且函数必须返回(只能返回)一个值，在(bool)pred()评估后返回true，或Clock::now()返回的时间大于或等于absolute_time。这里不保证调用线程将被阻塞的时长，只有当函数返回false后(Clock::now()返回一个等于或大于absolute_time的值)，线程接触阻塞。

抛出

当效果没有达成时，会抛出 `std::system_error` 异常或者由pred抛出任意异常。

同步

`std::condition_variable_any` 实例中的notify_one(),notify_all(),wait(),wait_for()和wait_until()都是序列化函数(串行调用)。调用notify_one()或notify_all()只能唤醒正在等待中的线程。