附录 D　C++11 线程库参考名录

D.1　<chrono>头文件

<chrono>头文件给出了 3 个类，分别表示时间点、时长和时钟，其中时钟类可以作为时间点的信息来源。每种时钟都含有一个 is_steady 静态数据成员，用于表示该时钟是否属于恒稳时钟，即它是否按恒定速率运行（且无法调整自身表示的时间）。std::chrono::steady_clock 是唯一一个能确保速率恒定的时钟类。

头文件内容：

```
namespace std
{
   namespace chrono
   {
       template<typename Rep,typename Period = ratio<1>>
       class duration;
       template<
           typename Clock,
           typename Duration = typename Clock::duration>
       class time_point;
       class system_clock;
       class steady_clock;
       typedef unspecified-clock-type high_resolution_clock;
   }
}
```

D.1.1　std::chrono::duration 类模板

std::chrono::duration 类模板用于表示时长（具有一定的时间跨度）。模板

参数 Rep 指定以哪种型别表示计时单元的数量，而参数 Period 则是另一类模板 std::ratio 的具现化，表示计时单元的大小(单位是分数形式的秒)。据此举例，则 std::chrono::duration<int, std::milli>表示一类时长，以 int 型值计数，计时单元为毫秒；又如，std::chrono::duration <short, std::ratio<1,50>>表示另一类时长，以 short 型值计数，计时单元为 1/50 秒；再比如，std:: chrono:: duration<long long, std::ratio<60,1>>则采用 long long 型值计数，计时单元为分钟。

类定义：

```
template <class Rep, class Period=ratio<1>>
class duration
{
public:
    typedef Rep rep;
    typedef Period period;

    constexprduration() = default;
    ~duration() = default;

    duration(const duration&) = default;
    duration& operator=(const duration&) = default;

    template <class Rep2>
    constexpr explicit duration(const Rep2& r);

    template <class Rep2, class Period2>
    constexprduration(const duration<Rep2, Period2>& d);

    constexpr rep count() const;
    constexpr duration operator+() const;
    constexpr duration operator-() const;
    duration& operator++();
    duration operator++(int);
    duration& operator--();
    duration operator--(int);
    duration& operator+=(const duration& d);
    duration& operator-=(const duration& d);
    duration& operator*=(constrep&rhs);
    duration& operator/=(constrep&rhs);
    duration& operator%=(constrep&rhs);
    duration& operator%=(constduration&rhs);
    static constexpr duration zero();
    static constexpr duration min();
    static constexpr duration max();
};

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator==(
    const duration<Rep1, Period1>& lhs,
```

```
    const duration<Rep2, Period2>& rhs);

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator!=(
    const duration<Rep1, Period1>& lhs,
    const duration<Rep2, Period2>& rhs);

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<(
    const duration<Rep1, Period1>&lhs,
    const duration<Rep2, Period2>&rhs);

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<=(
    const duration<Rep1, Period1>&lhs,
    const duration<Rep2, Period2>&rhs);

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>(
    const duration<Rep1, Period1>&lhs,
    const duration<Rep2, Period2>&rhs);

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>=(
    const duration<Rep1, Period1>&lhs,
    const duration<Rep2, Period2>&rhs);

template <class ToDuration, class Rep, class Period>
constexpr ToDuration duration_cast(const duration<Rep, Period>& d);
```

要求：

Rep 必须是内建的数值型别，或者与数值型别相似的用户自定义类型；Period 必须是类模板 std::ratio 的实例化。

1. std::chrono::duration::rep typedef

这是个 typedef，在时长值中，该型别用于保存计时单元的数量。

声明：

```
typedef Rep rep;
```

在 duration 对象内部，用于表示计时单元数量的值与 rep 型别一致。

2. std::chrono::duration::period typedef

该 typedef 是类模板 std::ratio 的某种实例化，由其设定计时单元的大小，它本身的含义是分数形式的秒，作用是结合计数表示总时长。譬如，假设 period 是 std::ratio<1, 50>，而某 duration 变量调用 count()的结果为 N，那么该时长值就表示 N/50 秒。

声明：

```
typedef Period period;
```

3. std::chrono::duration()默认构造函数

依照默认值构造一个 std::chrono::duration 实例。

声明：

```
constexpr duration() = default;
```

作用：

模板类 duration(模板参数为型别 rep)的内部值发生默认初始化。

4. std::chrono::duration()转换构造函数，转换的来源是某个计数值

根据某个具体的计数值构造一个 std::chrono::duration 实例。

声明：

```
template <class Rep2>
constexpr explicit duration(const Rep2& r);
```

作用：

模板类 duration 的内部值通过 static_cast<rep>(r)初始化。

要求：

仅当以下两个条件同时满足时，这个构造函数才会参与重载解析：一是型别 Rep2 可隐式转换成型别 Rep；二是 Rep 属于浮点型别，或 Rep2 不属于浮点型别。

后置条件：

```
this->count()==static_cast<rep>(r)
```

5. std::chrono::duration()转换构造函数，转换的来源是另一个 std::chrono::duration 值

将另一个 std::chrono::duration 对象按其计数值进行标量化，再据此构造一个 std::chrono::duration 实例。

声明：

```
template <class Rep2, class Period2>
constexpr duration(const duration<Rep2,Period2>& d);
```

作用：

对 duration 对象的内部值进行初始化，所依照的初值取自 duration_cast<duration<Rep, Period>>(d).count()。

要求：

仅当以下两个条件同时满足时，该构造函数才会参与重载解析：一是 Rep 属于浮点型别，或 Rep2 不属于浮点型别；二是分数 Period2 与分数 Period 之间正好是整数

倍乘关系（ratio_divide<Period2, Period>::den == 1）。如果有一个计时单元有较小的 duration 值，我们要将它存储到计时单元较大的 duration 变量中，只要满足上述条件，就能避免发生意外截断（及相应的精度损失）。

后置条件：

```
this->count()==duration_cast<duration<Rep,Period>>(d).count()
```

示例：

```
duration<int,ratio<1,1000>>ms(5);
duration<int,ratio<1,1>> s(ms);
duration<double,ratio<1,1>> s2(ms);
duration<int,ratio<1,1000000>> us(ms);
```

①5 毫秒

②错误：不能将毫秒当作整数秒存储

③正确：s2.count() == 0.005

④正确：us.count() == 5000

6. std::chrono::duration::count()成员函数

获取时长值。

声明：

```
constexpr rep count() const;
```

返回：

返回 duration 对象的内部值，其型别与 rep 一致。

7. std::chrono::duration::operator+ 一元加法运算符

这是个空操作：它仅返回*this 的副本。

声明：

```
constexpr duration operator+() const;
```

返回：

```
*this
```

8. std::chrono::duration::operator- 一元负值运算符

返回 duration 对象，该新对象调用 count()的结果是原对象 this->count()的负值。

声明：

```
constexpr duration operator-() const;
```

返回：

```
duration(-this->count());
```

9. std::chrono::duration::operator++ 前置自增运算符

自增内部计数值。

声明：

```
duration& operator++();
```

作用：

```
++this->internal_count;
```

返回：

```
*this
```

10. std::chrono::duration::operator++ 后置自增运算符

自增内部计数值，并返回自增前的*this。

声明：

```
duration operator++(int);
```

作用：

```
duration temp(*this);
++(*this);
return temp;
```

11. std::chrono::duration::operator-- 前置自减运算符

自减内部计数值。

声明：

```
duration& operator--();
```

作用：

```
--this->internal_count;
```

返回：

```
*this
```

12. std::chrono::duration::operator-- 后置自减运算符

自减内部计数值，并返回自减前的*this。

声明：

```
duration operator--(int);
```

作用：

```
duration temp(*this);
--(*this);
return temp;
```

13. std::chrono::duration::operator+= 复合赋值操作符

令*this对象的内部计数值与另一duration对象的计数值相加，以所得之和为自己赋值。

声明：

```
duration& operator+=(duration const& other);
```

作用：

```
internal_count+=other.count();
```

返回：

```
*this
```

14. std::chrono::duration::operator-= 复合赋值操作符

令*this对象的内部计数值与另一duration对象的计数值相减，以所得之差为自己赋值。

声明：

```
duration& operator-=(duration const& other);
```

作用：

```
internal_count-=other.count();
```

返回：

```
*this
```

15. std::chrono::duration::operator*= 复合赋值操作符

令*this对象的内部计数值与给定的值相乘，以所得之积为自己赋值。

声明：

```
duration& operator*=(rep const& rhs);
```

作用：

```
internal_count*=rhs;
```

返回：

```
*this
```

16. std::chrono::duration::operator/= 复合赋值操作符

令*this对象的内部计数值与给定的值相除，以所得之商为自己赋值。

声明：

```
duration& operator/=(rep const& rhs);
```

作用：

```
internal_count/=rhs;
```

返回：

```
*this
```

17. std::chrono::duration::operator%= 复合赋值操作符

将*this 对象的内部计数值与给定的值相除，以所得之余数为自己赋值。

声明：

```
duration& operator%=(rep const& rhs);
```

作用：

```
internal_count%=rhs;
```

返回：

```
*this
```

18. std::chrono::duration::operator%= 复合赋值操作符

令*this 对象的内部计数值与另一个 duration 对象的计数值相除，以所得之余数为自己赋值。

声明：

```
duration& operator%=(duration const& rhs);
```

作用：

```
internal_count%=rhs.count();
```

返回：

```
*this
```

19. std::chrono::duration::zero()静态成员函数

返回一个表示时长为 0 的 duration 对象。

声明：

```
constexpr duration zero();
```

返回：

```
duration(duration_values<rep>::zero());
```

20. std::chrono::duration::min()静态成员函数

指定某种计时单元，类模板 duration 根据它进行实例化，返回一个实例化的 duration 对象，其值是采用该计时单元所能表示的最短时长。

声明：

```
constexpr duration min();
```

返回：

```
duration(duration_values<rep>::min());
```

21. std::chrono::duration::max()静态成员函数

指定某种计时单元，类模板 duration 根据它进行实例化，返回一个实例化的 duration 对象，其值是采用该计时单元所能表示的最长时长。

声明：

```
constexpr duration max();
```

返回：

```
duration(duration_values<rep>::max());
```

22. std::chrono::duration 等值比较运算符

比较两个 duration 对象是否相等,即便双方的计数值型别和计时单元型别并不吻合，也强行比较。

声明：

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator==(
    const duration<Rep1, Period1>&lhs,
    const duration<Rep2, Period2>&rhs);
```

要求：

lhs 可以隐式转换成 rhs 所属型别，但无法反向转换；或者，rhs 可以隐式转换为 lhs 所属型别，但无法反向转换。如果它们都无法隐式转换成对方的型别，或它们分属 duration 的两种不同的实例化型别，却可以同时互相转换，则比较表达式出错。

作用：

假若 CommonDuration 和 std::common_type< duration< Rep1, Period1>, duration< Rep2, Period2>>::type 完全一致，那么 lhs==rhs，会返回 CommonDuration(lhs).count()== CommonDuration(rhs).count()。

23. std::chrono::duration 相异比较运算符

比较两个 duration 对象是否相异，即便双方的计数值型别和计时单元型别并不吻合，也强行比较。

声明：

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator!=(
    const duration<Rep1, Period1>&lhs,
    const duration<Rep2, Period2>&rhs);
```

要求：

lhs 可以隐式转换成 rhs 所属型别，但无法反向转换；或者，rhs 可以隐式转换为 lhs 所属型别，但无法反向转换。如果它们都无法隐式转换成对方的型别，或它们分属 duration 的两种不同的实例化型别，却可以同时互相转换，则比较表达式出错。

返回：

```
!(lhs==rhs)
```

24. std::chrono::duration 小于比较运算符

比较两个 duration 对象，判定运算符左边的对象是否小于右边的对象，即便双方的计数值型别和计时单元型别并不吻合，也强行比较。

声明：

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<(
    const duration<Rep1, Period1>& lhs,
    const duration<Rep2, Period2>& rhs);
```

要求：

lhs 可以隐式转换成 rhs 所属型别，但无法反向转换；或者，rhs 可隐式转换为 lhs 所属型别，但无法反向转换。如果它们都无法隐式转换成对方的型别，或它们分属 duration 的两种不同的实例化型别，却可以同时互相转换，则比较表达式出错。

作用：

假若 CommonDuration 和 std::common_type<duration<Rep1,Period1>, duration<Rep2, Period2>>::type 完全一致，则 lhs<rhs 会返回 CommonDuration (lhs).count() < Common Duration(rhs).count()。

25. std::chrono::duration 大于比较运算符

比较两个 duration 对象，判定运算符左边的对象是否大于右边的对象，即便双方的计数值型别和计时单元型别并不吻合，也强行比较。

声明：

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>(
    const duration<Rep1, Period1>& lhs,
    const duration<Rep2, Period2>& rhs);
```

要求：

lhs 可以隐式转换成 rhs 所属型别，但无法反向转换；或者，rhs 可隐式转换为 lhs 所属型别，但无法反向转换。如果它们都无法隐式转换成对方的型别，或它们分属 duration 的两种不同的实例化型别，却可以同时互相转换，则比较表达式出错。

返回：

```
rhs<lhs
```

26. std::chrono::duration 小于等于比较运算符

比较两个 duration 对象，判定运算符左边的对象是否小于等于右边的对象，无论双方的计数值型别和计时单元型别是否吻合。

声明：

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<=(
    const duration<Rep1, Period1>& lhs,
    const duration<Rep2, Period2>& rhs);
```

要求：

lhs 可以隐式转换成 rhs 所属型别，但无法反向转换；或者，rhs 可隐式转换为 lhs 所属型别，但无法反向转换。如果它们都无法隐式转换成对方的型别，或它们分属 duration 的两种不同的实例化型别，却可以同时互相转换，则比较表达式出错。

返回：

```
!(rhs<lhs)
```

27. std::chrono::duration 大于等于比较运算符

比较两个 duration 对象，判定运算符左边的对象是否大于等于右边的对象，无论双方的计数值型别和计时单元型别是否吻合。

声明：

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>=(
    const duration<Rep1, Period1>& lhs,
    const duration<Rep2, Period2>& rhs);
```

要求：

lhs 可以隐式转换成 rhs 所属型别，但无法反向转换；或者，rhs 可隐式转换为 lhs 所属型别，但无法反向转换。如果它们都无法隐式转换成对方的型别，或它们分属

duration 的两种不同的实例化型别，却可以同时互相转换，则比较表达式出错。

返回：

```
!(lhs<rhs)
```

28. std::chrono::duration_cast()非成员函数

将一个 std::chrono::duration 对象显式转化为一个 std::chrono::duration 实例。

声明：

```
template <class ToDuration, class Rep, class Period>
constexpr ToDuration duration_cast(const duration<Rep, Period>& d);
```

要求：

ToDuration 必须是类模板 std::chrono::duration 的实例化。

返回：

返回 duration 对象，它由 d 转化而来，其型别通过 ToDuration 指定。各时长对象的计数值具有不同的型别和取值范围，有可能因转化而导致精度损失，编译器会选取令损失程度最小的方式进行操作。

D.1.2 std::chrono::time_point 类模板

类模板 std::chrono::time_point 代表一个时间点，它以某个特定的时钟为衡量标准。它的值是一段时长，表示从该时钟的纪元时刻直至该时间点的跨度。模板参数 Clock 指定了参照的时钟（各个时钟必定分别属于其特有的型别），而参数 Duration 则必须是类模板 std::chrono::duration 的实例化的型别，用于度量纪元时刻到该时间点的时长。模板参数 Clock 本身具备默认的时长型别，而参数 Duration 的型别在默认情况下会与它保持一致。

类定义：

```
template <class Clock,class Duration = typename Clock::duration>
class time_point
{
public:
    typedef Clock clock;
    typedef Duration duration;
    typedef typename duration::rep rep;
    typedef typename duration::period period;

    time_point();
    explicit time_point(const duration& d);

    template <class Duration2>
    time_point(const time_point<clock, Duration2>& t);
```

```
    duration time_since_epoch() const;

    time_point& operator+=(const duration& d);
    time_point& operator-=(const duration& d);

    static constexpr time_point min();
    static const exprtime_point max();
};
```

1. std::chrono::time_point()默认构造函数

构造一个 time_point 对象，表示与自身关联的 Clock 类的纪元时刻，内部时长依照 Duration::zero()的值进行初始化。

声明：

```
time_point();
```

后置条件：

对于按默认方式构造出的 time_point 对象 tp，tp.time_since_epoch() == tp::duration::zero()成立。

2. std::chrono::time_point()时长构造函数

依据指定的时长和关联的 Clock 类构造一个 time_point 对象，代表某时间点，从关联时钟的纪元时刻到该时间点的跨度即为指定时长。

声明：

```
explicit time_point(const duration& d);
```

后置条件：

若 time_point 对象 tp 是依照某时长 d 以 tp(d)方式构造而成的，则 tp.time_since_epoch() == d 成立。

3. std::chrono::time_point()转换构造函数

依据某 time_point 对象构造出新的 time_point 对象，两者都与同一个 Clock 类关联，但所含时长值不同。

声明：

```
template <class Duration2>
time_point(const time_point<clock, Duration2>& t);
```

要求：

Duration2 会隐式转换为 Duration。

作用：

time_point(t.time_since_epoch())会发生隐式转换，根据相同的方式，t.time_since_

epoch()的返回值也会隐式转换成 Duration 类型的对象，得出的 time_point 新对象中含有该返回值。

4. std::chrono::time_point::time_since_epoch()成员函数

返回一个时长值，表示从关联时钟的纪元时刻直到某特定 time_point 对象的跨度。

声明：

```
duration time_since_epoch() const;
```

返回：

返回存放在*this 对象中的时长值。

5. std::chrono::time_point::operator+= 复合赋值操作符

将 time_point 对象中存有的时长值与给定的时长值相加，以所得之和为自己赋值。

声明：

```
time_point& operator+=(const duration& d);
```

作用：

将*this 对象内部的 duration 对象与 d 相加，以所得之和为自己赋值，实际效果相当于`this->internal_duration += d;`。

返回：

返回*this 对象。

6. std::chrono::time_point::operator-= 复合赋值操作符

将 time_point 对象中存有的时长值与给定的时长值相减，以所得之差为自己赋值。

声明：

```
time_point& operator-=(const duration& d);
```

作用：

将*this 对象内部的 duration 对象与 d 相减，以所得之差为自己赋值，实际效果相当于`this->internal_duration-= d;`。

返回：

返回*this 对象。

7. std::chrono::time_point::min()静态成员函数

取得一个 time_point 对象，它是自身所属型别能够表示出的最小值。

声明：

```
static constexpr time_point min();
```

返回：

```
time_point(time_point::duration::min())
```

8. std::chrono::time_point::max()静态成员函数

取得一个 time_point 对象，它是自身所属型别能够表示出的最大值。

声明：

```
static constexpr time_point max();
```

返回：

```
time_point(time_point::duration::max())
```

D.1.3 std::chrono::system_clock 类

std::chrono::system_clock 类提供的功能是从系统级别范围的实时时钟上取得当前实际时间。当前时刻可通过调用 std::chrono::system_clock::now()来获取。凭借 `std::chrono::system_clock::to_time_t()` 和 `std::chrono::system_clock::to_time_point()` 函数，std:: chrono::system_clock::time_point 的实例可以与 time_t 型别的值相互转换。系统时钟不是恒稳时钟，先前调用 std::chrono::system_clock::now()而取得的时间值，有可能晚于后来的调用所获得的值（例如，为了同步外界时钟，操作系统的时钟被手动调整过）。

类定义：

```
class system_clock
{
public:
    typedef unspecified-integral-type rep;
    typedef std::ratio<unspecified,unspecified> period;
    typedef std::chrono::duration<rep,period> duration;
    typedef std::chrono::time_point<system_clock> time_point;
    static const bool is_steady=unspecified;

    static time_point now() noexcept;

    static time_t to_time_t(const time_point& t) noexcept;
    static time_point from_time_t(time_t t) noexcept;
};
```

1. std::chrono::system_clock::rep typedef

该 typedef 属于某种整型，作用是记录时长值中所含的计时单元的数目。

声明：

```
typedef unspecified-integral-type rep;
```

提示：unspecified-integral-type 是指某种整型，可能是 short、unsigned short、int、unsigned int、long、unsigned long，以及 intX_t、int_fastX_t、int_leastX_t、uintX_t、uint_fastX_t、uint_leastX_t 等，当前 C++标准并未规定具体采用哪一种。

2. std::chrono::system_clock::period typedef

该 typedef 是类模板 std::ratio 的某种实例化，表示不相等的两个 duration 值或两个 time_point 值之间的最小数值差别（单位是秒，或秒的某种分数形式）。period 型别指定了时钟的精度，而不是计时单元的大小。

声明：

```
typedef std::ratio<unspecified,unspecified> period;
```

提示：unspecified 是未明确规定的类型。

3. std::chrono::system_clock::duration typedef

类模板 std::chrono::duration 的某种实例化，用于表示两个时间点的差别，它们都取自系统级别范围的实时时钟。

声明：

```
typedef std::chrono::duration<
    std::chrono::system_clock::rep,
    std::chrono::system_clock::period> duration;
```

4. std::chrono::system_clock::time_point typedef

类模板 std::chrono::time_point 的某种实例化，表示取自系统级别范围的实时时钟的时间点。

声明：

```
typedef std::chrono::time_point<std::chrono::system_clock> time_point;
```

5. std::chrono::system_clock::now()静态成员函数

从系统级别范围的实时时钟取得当前实际时间。

声明：

```
time_point now() noexcept;
```

返回：

返回 time_point 对象，表示系统级别范围的实时时钟的当前时间。

抛出：

若发生错误，则抛出 std::system_error 型别的异常。

6. std::chrono::system_clock:to_time_t()静态成员函数

将 time_point 的实例转化为 time_t 值。

声明：

```
time_t to_time_t(time_point const& t) noexcept;
```

返回：

返回一个型别为 time_t 的值，与时间点 t 表示相同的时间，以秒为精度进行舍入或截断。

抛出：

若发生错误，则抛出 std::system_error 型别的异常。

7. std::chrono::system_clock::from_time_t()静态成员函数

将 time_t 型别的值转化为 time_point 实例。

声明：

```
time_point from_time_t(time_t const& t) noexcept;
```

返回：

返回一个 time_point 值，它与 t 表示相同的时间。

抛出：

若发生错误，则抛出 std::system_error 型别的异常。

D.1.4 std::chrono::steady_clock 类

std::chrono::steady_clock 类的功能是作为系统级别范围的恒稳时钟。当前时间可通过调用 std::chrono::steady_clock::now()获取。由 std::chrono::steady_clock::now()获取的时间与当前实际时间没有固定关系。恒稳时钟无法回拨，如果 std::chrono::steady_clock::now()发生两次调用，那么第二次调用返回的时间点必定大于等于第一次调用返回的时间点。时钟以恒定速率计时。

类定义：

```
class steady_clock
{
public:
    typedef unspecified-integral-type rep;
    typedef std::ratio<
        unspecified,unspecified> period;
    typedef std::chrono::duration<rep,period> duration;
    typedef std::chrono::time_point<steady_clock>
        time_point;
    static const bool is_steady=true;

    static time_point now() noexcept;
};
```

1. std::chrono::steady_clock::rep typedef

该 typedef 为整型，用于表示时长值中计时单元的数量。

声明：

```
typedef unspecified-integral-type rep;
```

2. std::chrono::steady_clock::period typedef

该 typedef 是类模板 std::ratio 的某种实例化，表示不相等的两个 duration 值或两个 time_point 值之间的最小数值差别（单位是秒，或秒的某种分数形式）。period 型别指定了时钟的精度，而不是计时单元的大小。

声明：

```
typedef std::ratio<unspecified,unspecified> period;
```

3. std::chrono::steady_clock::duration typedef

类模板 `std::chrono::duration` 的某种实例化，用于表示两个时间点的差别，这两个时间点都取自系统级别范围的恒稳时钟。

声明：

```
typedef std::chrono::duration<
    std::chrono::steady_clock::rep,
    std::chrono::steady_clock::period> duration;
```

4. std::chrono::steady_clock::time_point typedef

类模板 `std::chrono::time_point` 的某种实例化，表示取自系统级别范围的恒稳时钟的时间点。

声明：

```
typedef std::chrono::time_point<std::chrono::steady_clock> time_point;
```

5. std::chrono::steady_clock::now()静态成员函数

从系统级别范围的恒稳时钟取得当前时间。

声明：

```
time_point now() noexcept;
```

返回：

返回一个 time_point 对象，表示系统级别范围的恒稳时钟的当前时间。

抛出：

若发生错误，则抛出 std::system_error 型别的异常。

同步：

如果 std::chrono::steady_clock::now()的某次调用先于另一次发生，那么第一次调用返回的 time_point 值必将小于等于第二次调用返回的值。

D.1.5 std::chrono::high_resolution_clock typedef

std::chrono::high_resolution_clock 类的功能是提供最高精度的系统级别范围的时钟。与其他时钟相同，当前时间通过调用 std::chrono::high_resolution_clock::now()获取。std::chrono::high_resolution_clock 可能是 std::chrono::system_clock 类或 std::chrono::steady_clock 类的 typedef，也可能是另一个单独的类。尽管 std::chrono::high_resolution_clock 的精度是标准库所支持的全部时钟里最高的，但调用 std::chrono::high_resolution_clock::now()仍要花费一定时间。如果我们调用 std::chrono::high_resolution_clock::now()针对简短操作计时，则必须考虑函数调用本身的额外开销。

类定义：

```
class high_resolution_clock
{
public:
    typedef unspecified-integral-type rep;
    typedef std::ratio<
        unspecified,unspecified> period;
    typedef std::chrono::duration<rep,period> duration;
    typedef std::chrono::time_point<
        unspecified> time_point;
    static const bool is_steady=unspecified;

    static time_point now() noexcept;
};
```

D.2 <condition_variable>头文件

<condition_variable>头文件给出了各种条件变量的定义。它们都是基本的同步机制的工具，让线程发生阻塞，等到某种条件成立为止，或一路等待，直到超出限定的时长。

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

线程借助 std::condition_variable 类得以等待某条件而阻塞，条件成立之后才可继续运行。std::condition_variable 的实例不满足可拷贝赋值（copy assignable）、可拷贝构造（copy constructible）、可移动赋值（move assignable）或可移动构造（move constructible）的型别要求[①]。

类定义：

```
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,
        conststd::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,
```

[①] 译者注：这些术语都是针对型别的具名要求，即 named requirement，属于 C++20 的 concept 新特性。

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

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

1. std::condition_variable()默认构造函数

构造一个 std::condition_variable 对象。

声明：

```
condition_variable();
```

作用：

构造一个新的 std::condition_variable 实例。

抛出：

如果条件变量无法构造，将抛出 std::system_error 型别的异常。

2. std::condition_variable()析构函数

销毁 std::condition_variable 对象。

声明：

```
~condition_variable();
```

前置条件：

在*this 对象上，没有线程因调用 `wait()`、`wait_for()`或 `wait_until()`而被阻塞。

作用：

销毁*this 对象。

抛出：

无。

3. std::condition_variable::notify_one()成员函数

如果有线程正在 std::condition_variable 实例上等待，则唤醒其中一个。

声明：

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

作用：

唤醒一个等待*this 的线程。如果没有线程正在等待，调用就不会产生效果。

抛出：

若未能达到上述效果，就会抛出 std::system_error 型别的异常。

同步：

在同一个 std::condition_variable 实例上，如果发生 notify_one()、notify_all()、wait()、wait_for()或 wait_until()的多个调用，它们就会按串行方式逐一执行。调用 notify_one()和 notify_all()只会唤醒已经开始等待的线程。

4. std::condition_variable::notify_all()成员函数

唤醒全部正在等待 std::condition_variable 的某个具体实例的线程。

声明：

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

作用：

唤醒全部正在等待*this 对象的线程。如果没有线程正在等待，调用就不会产生效果。

抛出：

若未能达到上述效果，就会抛出 std::system_error 型别的异常。

同步：

在同一个 std::condition_variable 实例上，如果发生 notify_one()、notify_all()、wait()、wait_for()或 wait_until()的多个调用，它们就会按串行方式逐一执行。调用 notify_one()和 notify_all()只会唤醒已经开始等待的线程。

5. std::condition_variable::wait()成员函数

使线程等待，直到 std::condition_variable 上发生 notify_one()或 notify_all()的调用才被唤醒，或直到伪唤醒发生。

声明：

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

前置条件：

lock.owns_lock()的结果为 true，且发起本函数调用的线程拥有该锁。

作用：

按原子化方式解开给定的 lock 对象，并使线程发生阻塞，直到其他线程调用 notify_one()或 notify_all()而被唤醒，或直到伪唤醒发生。lock 对象在 wait()调用返回前将重新锁住。

抛出：

若未能达到上述效果，就会抛出 std::system_error 型别的异常。lock 对象在 wait()调用期间会被解锁，而在函数退出之际会被重新锁定，即便因抛出异常而退出也同样如此。

注记：

伪唤醒的含义是，一个线程在调用 wait()后无端苏醒，但其他线程却未曾调用过

notify_one()或 notify_all()。wait()有一个接受断言的重载，所以在此建议，只要有可能就优先采用该版本。否则，建议设立一个循环，反复查验与条件变量关联的断言，并将wait()调用放置到循环中。

同步：

在同一个 std::condition_variable 实例上，如果发生 notify_one()、notify_all()、wait()、wait_for()或 wait_until()的多个调用，它们就会按串行方式逐一执行。调用 notify_one()和 notify_all()只会唤醒已经开始等待的线程。

6. std::condition_variable::wait()通过参数接受断言的成员函数重载

使线程等待，直到 std::condition_variable 上发生 notify_one()或 notify_all()的调用才被唤醒，且断言成立。

声明：

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

前置条件：

表达式 pred()必须合法，且其返回值可以转化为布尔值。lock.owns_lock()的结果为true，且发起 wait()调用的线程拥有该锁。

作用：

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

抛出：

抛出所有在 pred()调用中产生的异常，或者，若未能达到上述效果，则抛出 std::system_error 型别的异常。

注记：

由于可能发生伪唤醒，pred()的调用次数无法确定。断言 pred()被调用时，lock 对象所指涉的互斥总会被锁定。而本函数返回的前提是，当且仅当(bool)pred()的求值结果为true。

同步：

在同一个 std::condition_variable 实例上，如果发生 notify_one()、notify_all()、wait()、wait_for()或 wait_until()的多个调用，它们就会按串行方式逐一执行。notify_one()和notify_all()的调用只会唤醒已经开始等待的线程。

7. std::condition_variable::wait_for()成员函数

使线程等待，直到 std::condition_variable 上发生 notify_one()或 notify_all()的调用才

被唤醒，或直到超出限定的时长，或直到线程发生伪唤醒。

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);
```

前置条件：

lock.owns_lock()的结果为 true，且发起本函数调用的线程拥有该锁。

作用：

按原子化方式解开给定的 lock 对象，并使线程发生阻塞，直到其他线程调用 notify_one()或 notify_all()才被唤醒，或直到超出限定的时长，或直到伪唤醒发生。lock 对象在 wait_for()返回前将重新锁住。

返回：

如果线程被 notify_one()或 notify_all()唤醒，或发生伪唤醒，就返回 std::cv_status::no_timeout，否则返回 std::cv_status::timeout。

抛出：

若未能达到上述效果，就抛出 std::system_error 型别的异常。lock 对象在 wait_for()调用期间会被解锁，而在函数退出之际会被重新锁定，即便因抛出异常而退出也同样如此。

注记：

伪唤醒的含义是，一个线程在调用 wait_for()后无端苏醒，但其他线程却未曾调用过 notify_one()或 notify_all()。wait_for()有一个接受断言的重载，所以在此建议，只要有可能就优先采用该版本。否则，建议设立一个循环，反复查验与条件变量关联的断言，并将 wait_for()调用放置到循环中。如果采用这种方式，则务必小心处理，以确保限时功能依旧有效运作；在许多情况下，wait_until()可能更加适合。线程实际阻塞的时间也许会比原本指定的更长。只要有可能，就应该由恒稳时钟判定已经消耗的等待时间。

同步：

在同一个 std::condition_variable 实例上，如果发生 notify_one()、notify_all()、wait()、wait_for()或 wait_until()的多个调用，它们就会按串行方式逐一执行。调用 notify_one()和 notify_all()只会唤醒已经开始等待的线程。

8. std::condition_variable::wait_for()通过参数接受断言的成员函数重载

使线程等待，直到 std::condition_variable 上发生 notify_one()或 notify_all()的调用才被唤醒，且断言成立，或直到超出限定的时长。

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);
```

前置条件：

表达式 pred()必须合法，且其返回值可以转化为布尔值。lock.owns_lock()的结果为 true，且发起 wait_for()调用的线程拥有该锁。

作用：

```
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：

如果 pred()的最近一次调用结果为 true，则返回 true；如果等待超出了 relative_time 所限定的时长，且 pred()的结果为 false，则返回 false。

注记：

由于可能发生伪唤醒，pred()的调用次数无法确定。断言 pred()被调用时，lock 对象所指涉的互斥总会被锁定。而本函数返回的前提是，当且仅当(bool)pred()的求值结果为 true，或者等待的时长超出了 relative_time 所限定的时长。线程实际阻塞的时间也许会比原本指定的更长。只要有可能，就应该由恒稳时钟判定已消耗的等待时间。

抛出：

抛出所有在 pred()调用中产生的异常，或者，若未能达到上述效果，则抛出 std::system_error 型别的异常。

同步：

在同一个 std::condition_variable 实例上，如果发生 notify_one()、notify_all()、wait()、wait_for()或 wait_until()的多个调用，它们就会按串行方式逐一执行。调用 notify_one()和 notify_all()只会唤醒已经开始等待的线程。

9. std::condition_variable::wait_until()成员函数

使线程等待，直到 std::condition_variable 上发生 notify_one()或 notify_all()的调用而被唤醒，或直到超出限定的时长，或直到伪唤醒发生。

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);
```

前置条件：

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

作用：

按原子化方式解开给定的 lock 对象，并使线程发生阻塞，直到其他线程调用 notify_one()或 notify_all()才被唤醒，或直到 Clock::now()返回的时刻等于/晚于参数 absolute_time 所指定的时刻，或直到伪唤醒发生。lock 对象在 wait_until()的调用返回前将重新锁住。

返回：

线程被 notify_one()或 notify_all()唤醒，或发生伪唤醒时，会返回 std::cv_status::no_timeout；反之，则返回 std::cv_status::timeout。如果线程被 notify_one()或 notify_all()唤醒，或发生伪唤醒，就返回 std::cv_status::no_timeout，否则返回 std::cv_status::timeout。

抛出：

若未能达到上述效果，就抛出 std::system_error 型别的异常。lock 对象在 wait_until()调用期间会被解锁，而在函数退出之际会被重新锁定，即便因抛出异常而退出也同样如此。

注记：

伪唤醒的含义是，一个线程在调用 wait_until()后无端苏醒，但其他线程却未曾调用过 notify_one()或 notify_all()。wait_until()有一个接受断言的重载，所以在此建议，只要有可能就优先采用该版本。否则，建议设立一个循环，反复查验与条件变量关联的断言，并将 wait_until()调用放置到循环中。发起调用的线程的阻塞时间并不确定，只有当 Clock::now()返回的时刻等于/晚于参数 absolute_time 所指定的时刻时，该线程才会结束阻塞。

同步：

在同一个 std::condition_variable 实例上，如果发生 notify_one()、notify_all()、wait()、wait_for()或 wait_until()的多个调用，它们就会按串行方式逐一执行。调用 notify_one()和 notify_all()只会唤醒已经开始等待的线程。

10. std::condition_variable::wait_until()通过参数接受断言的成员函数重载

使线程等待，直到 std::condition_variable 上发生 notify_one()或 notify_all()的调用而被唤醒，且断言成立，或者，直到超出限定的时长。

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);
```

前置条件：

表达式 pred()必须合法，且其返回值可以转化为布尔值。lock.owns_lock()的结果为

true，且发起 wait_until()调用的线程拥有该锁。

作用：

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

返回：

如果 pred()的最近一次调用结果为 true，则返回 true；如果 Clock::now()所返回的时刻等于/晚于参数 absolute_time 所指定的时刻，且 pred()的结果为 false，则返回 false。

注记：

由于可能发生伪唤醒，pred()的调用次数无法确定。断言 pred()被调用时，lock 对象所指涉的互斥总会被锁定。而本函数返回的前提是，当且仅当(bool)pred()的求值结果为 true，或 Clock::now()返回的时刻等于/晚于参数 absolute_time 所指定的时刻。发起调用的线程的阻塞时间并不确定，只有当断言函数返回 false，而 Clock::now()返回的时刻等于/晚于参数 absolute_time 所指定的时刻时，该线程才会结束阻塞。

抛出：

抛出所有在 pred()调用中产生的异常，或者，若未能达到上述效果，则抛出 std::system_error 型别的异常。

同步：

在同一个 std::condition_variable 实例上，如果发生 notify_one()、notify_all()、wait()、wait_for()或 wait_until()的多个调用，它们就会按串行方式逐一执行。调用 notify_one()和 notify_all()只会唤醒已经开始等待的线程。

11. std::notify_all_at_thread_exit()非成员函数

一旦当前线程退出，就唤醒在某 std::condition_variable 上等待的全部线程。

声明：

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

前置条件：

lk.owns_lock()的结果为 true，且发起本函数调用的线程拥有该锁。假设条件变量 cv 上有线程正在等待，则它们曾经调用过 `wait()`、`wait_for()`或 `wait_until()`，因而也在调用时接受了某些锁对象，那么 lk.mutex()的返回值必须恰好是这批锁中的一员。

作用：

lk 所持有的锁的归属权转移给线程的内部存储，在发起本函数调用的线程退出时，

安排 cv 对象被知会。

作用：

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

抛出：

当没有达到上述效果时，抛出 std::system_error 异常。

注记：

在线程结束前，锁 lk 会一直被持有，故务必小心防范死锁。这里建议发起本函数调用的线程及早完成任务并退出，不得执行导致阻塞的操作。用户必须保证等待的线程不会做出错误的假设，当它们被唤醒时，发起调用的线程肯定已经退出，需特别注意发生伪唤醒的情形。要做到这点，可以在等待的线程上查验某个断言，而该断言成立的条件是以互斥保护负责唤醒的线程，且在调用 notify_all_at_thread_exit()完成之前，互斥之上的锁均不会释放。

D.2.2 std::condition_variable_any 类

线程借助 std::condition_variable_any 类以等待某条件而被阻塞，条件成立之后才可继续运行。std::condition_variable 只能配合 std::unique_lock<std::mutex>使用。然而，无论任何类型，只要符合 Lockable 的型别要求*，就可以和 std::condition_variable_any 搭配使用。std::condition_variable_any 的实例不满足 CopyAssignable、CopyConstructible、MoveAssignable 或 MoveConstructible 的型别要求[①]。

类定义：

```
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);
```

① 译者注：这些术语都是针对型别的具名要求，即 named requirements，属于 C++20 的 concept 新特性。

```
    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);
};
```

1. 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();
```

作用：

构造一个新的 std::condition_variable_any 对象。

抛出：

如果条件变量无法构造，将抛出 std::system_error 型别的异常。

2. std::condition_variable_any()析构函数

销毁 std::condition_variable_any 对象。

声明：

```
~condition_variable_any();
```

前置条件：

之前没有使用*this 对象中的 wait()、wait_for()或 wait_until()阻塞过线程。

作用：

销毁*this 对象。

抛出：

无。

3. std::condition_variable_any::notify_one()成员函数

如果有线程正在 std::condition_variable_any 实例上等待，则唤醒其中一个。

声明：

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

作用：

唤醒一个等待*this 实例的线程。如果没有线程正在等待，调用就不会产生效果。

抛出：

若未能达到上述效果，就抛出 std::system_error 型别的异常。

同步：

在同一个 std::condition_variable_any 实例上，如果发生 notify_one()、notify_all()、wait()、wait_for()或 wait_until()的多个调用，它们就会按串行方式逐一执行。调用 notify_one()和 notify_all()只会唤醒已经开始等待的线程。

4. std::condition_variable_any::notify_all()成员函数

唤醒全部正在等待 std::condition_variable_any 的某个具体实例的线程。

声明：

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

作用：

唤醒全部正在等待*this 对象的线程。如果没有线程正在等待，调用就不会产生效果。

抛出：

若未能达到上述效果，就抛出 std::system_error 型别的异常。

同步：

在同一个 std::condition_variable_any 实例上，如果发生 notify_one()、notify_all()、wait()、wait_for()或 wait_until()的多个调用，它们就会按串行方式逐一执行。调用 notify_one()和 notify_all()只会唤醒已经开始等待的线程。

5. std::condition_variable_any::wait()成员函数

使线程等待，直到 `std::condition_variable_any` 上发生 `notify_one()` 或 `notify_all()` 的调用才被唤醒，或直到伪唤醒发生。

声明：

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

前置条件：

模板参数 Lockable 满足 Lockable 的型别要求，且 lock 对象拥有一个锁。

作用：

按原子化方式解开给定的 lock 对象，并使线程发生阻塞，直到其他线程调用 notify_one()或 notify_all()才被唤醒，或直到伪唤醒发生。lock 对象在 wait()调用返回前将被重新锁住。

抛出：

若未能达到上述效果，就抛出 std::system_error 型别的异常。lock 对象在 wait()调用期间会被解锁，而在函数退出之际会被重新锁定，即便因抛出异常而退出也同样如此。

注记：

伪唤醒的含义是，一个线程在调用 wait()后无端苏醒，但其他线程却未曾调用过 notify_one()或 notify_all()。wait()有一个接受断言的重载，所以在此建议，只要有可能就优先采用该版本。否则，建议设立一个循环，反复查验与条件变量关联的断言，并将 wait()调用放置到循环中。

同步：

在同一个 std::condition_variable_any 实例上，如果发生 notify_one()、notify_all()、wait()、wait_for()或 wait_until()的多个调用，它们就会按串行方式逐一执行。调用 notify_one()和 notify_all()只会唤醒已经开始等待的线程。

6. std::condition_variable_any::wait()通过参数接受断言的成员函数重载

使线程等待，直到 std::condition_variable_any 上发生 notify_one()或 notify_all()的调用而被唤醒，且断言成立。

声明：

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

前置条件：

表达式 pred()必须合法，且其返回值可以转化为布尔值。模板参数 Lockable 满足 Lockable 的型别要求，且 lock 对象拥有一个锁。

作用：

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

抛出：

抛出任何在 pred()调用中产生的异常，或者，若未能达到上述效果，则抛出 std::system_error 型别的异常。

注记：

由于可能发生伪唤醒，pred()的调用次数无法确定。断言 pred()被调用时，lock 对象所指涉的互斥总会被锁定。而本函数返回的前提是，当且仅当(bool)pred()的求值结果为 true。

同步：

在同一个 std::condition_variable_any 实例上，如果发生 notify_one()、notify_all()、wait()、wait_for()或 wait_until()的多个调用，它们就会按串行方式逐一执行。notify_one()和 notify_all()的调用只会唤醒已经开始等待的线程。

7. std::condition_variable_any::wait_for()成员函数

使线程等待，直到 std::condition_variable_any 上发生 notify_one()或 notify_all()的调用而被唤醒，或直到超出限定的时长，或直到线程发生伪唤醒。

声明：

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

前置条件：

模板参数 Lockable 满足 Lockable 的型别要求，且 lock 对象拥有一个锁。

作用：

按原子化方式解开给定的 lock 对象，并使线程发生阻塞，直到其他线程调用 notify_one()或 notify_all()才被唤醒，或直到超出限定的时长，或直到伪唤醒发生。lock 对象在 wait_for()返回前将重新锁住。

返回：

如果线程被 notify_one()或 notify_all()唤醒，或发生伪唤醒，就返回 std::cv_status::no_timeout，否则返回 std::cv_status::timeout。

抛出：

若未能达到上述效果，就抛出 std::system_error 型别的异常。lock 对象在 wait_for()调用期间会被解锁，而在函数退出之际会被重新锁定，即便因抛出异常而退出也同样如此。

注记：

伪唤醒的含义是，一个线程在调用 wait_for()后无端苏醒，但其他线程却未曾调用过 notify_one()或 notify_all()。wait_for()有一个接受断言的重载，所以在此建议，只要有可

能就优先采用该版本，否则，建议设立一个循环，反复查验与条件变量关联的断言，并将 wait_for()调用放置到循环中。如果采用这种方式，则务必小心处理，以确保限时功能依旧有效运作。在许多情况下，wait_until()可能更加适合。线程实际阻塞的时间也许会比原本指定的更长。只要有可能，就应该由恒稳时钟判定已消耗的等待时间。

同步：

在同一个 std::condition_variable_any 实例上，如果发生 notify_one()、notify_all()、wait()、wait_for()或 wait_until()的多个调用，它们就会按串行方式逐一执行。调用 notify_one()和 notify_all()只会唤醒已经开始等待的线程。

8. std::condition_variable_any::wait_for()通过参数接受断言的成员函数重载

使线程等待，直到 std::condition_variable_any 上发生 notify_one()或 notify_all()的调用才被唤醒，且断言成立，或直到超出限定的时长。

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);
```

前置条件：

表达式 pred()必须合法，且其返回值可以转化为布尔值。模板参数 Lockable 满足 Lockable 的型别要求，且 lock 对象拥有一个锁。

作用：

```
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：

如果 pred()的最近一次调用结果为 true，则返回 true；如果等待超出了 relative_time 所限定的时长，且 pred()的结果为 false，则返回 false。

注记：

由于可能发生伪唤醒，pred()的调用次数无法确定。断言 pred()被调用时，lock 对象所指涉的互斥总会被锁定。而本函数返回的前提是，当且仅当(bool)pred()的求值结果为 true，或者等待的时长超出了 relative_time 所限定的时长。线程实际阻塞的时间也许会

比原本指定的更长。只要有可能，就应该由恒稳时钟判定已消耗的等待时间。

抛出：

抛出所有在 pred()调用中产生的异常，或者，若未能达到上述效果，则抛出 std::system_error 型别的异常。

同步：

在同一个 std::condition_variable_any 实例上，如果发生 notify_one()、notify_all()、wait()、wait_for()或 wait_until()的多个调用，它们就会按串行方式逐一执行。调用 notify_one()和 notify_all()只会唤醒已经开始等待的线程。

9. std::condition_variable_any::wait_until()成员函数

使线程等待，直到 std::condition_variable_any 上发生 notify_one()或 notify_all()的调用才被唤醒，或直到超出限定的时长，或直到伪唤醒发生。

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);
```

前置条件：

模板参数 Lockable 满足 Lockable 的型别要求，且 lock 对象拥有一个锁。

作用：

按原子化方式解开给定的 lock 对象，并使线程发生阻塞，直到其他线程调用 notify_one()或 notify_all()而被唤醒，或直到 Clock::now()返回的时刻等于/晚于参数 absolute_time 所指定的时刻，或直到伪唤醒发生。lock 对象在 wait_until()的调用返回前将被重新锁住。

返回：

如果线程被 notify_one()或 notify_all()唤醒，或发生伪唤醒，就返回 std::cv_status::no_timeout，否则返回 std::cv_status::timeout。

抛出：

若未能达到上述效果，就抛出 std::system_error 型别的异常。lock 对象在 wait_until()调用期间会被解锁，而在函数退出之际会被重新锁定，即便因抛出异常而退出也同样如此。

注记：

伪唤醒的含义是，一个线程在调用 wait_until()后无端苏醒，但其他线程却未曾调用过 notify_one()或 notify_all()。wait_until()有一个接受断言的重载，所以在此建议，只要有可能就优先采用该版本。否则，建议设立一个循环，反复查验与条件变量关联的断言，并将 wait_until()调用放置到循环中。发起调用的线程的阻塞时间并不确定，

只有当 Clock::now()返回的时刻等于/晚于参数 absolute_time 所指定的时刻，该线程才会结束阻塞。

同步：

在同一个 std::condition_variable_any 实例上，如果发生 notify_one()、notify_all()、wait()、wait_for()或 wait_until()的多个调用，它们就会按串行方式逐一执行。调用 notify_one()和 notify_all()只会唤醒已经开始等待的线程。

10. std::condition_variable_any::wait_until()通过参数接受断言的成员函数重载

使线程等待，直到 std::condition_variable_any 上发生 notify_one()或 notify_all()的调用才被唤醒，且断言成立，或直到超出限定的时长。

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);
```

前置条件：

表达式 pred()必须合法，且其返回值可以转化为布尔值。模板参数 Lockable 满足 Lockable 的型别要求，且 lock 对象拥有一个锁。

作用：

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

返回：

若 pred()的最近一次调用结果为 true，则返回 true；若调用 Clock::now()所返回的时刻等于/晚于参数 absolute_time 所指定的时刻，且 pred()的结果为 false，则返回 false。

注记：

由于可能发生伪唤醒，pred()的调用次数无法确定。断言 pred()被调用时，lock 对象所指涉的互斥总会被锁定。而本函数返回的前提是，当且仅当(bool)pred()的求值结果为 true，或者，Clock::now()返回的时刻等于/晚于参数 absolute_time 所指定的时刻。发起调用的线程的阻塞时间并不确定，只有当断言函数返回 false，而 Clock::now()返回的时刻等于/晚于参数 absolute_time 所指定的时刻时，该线程才会结束阻塞。

抛出：

抛出任何在 pred()调用中产生的异常，或者，若未能达到上述效果，则抛出 std::system_error 型别的异常。

同步：

在同一个 std::condition_variable_any 实例上，如果发生 notify_one()、notify_all()、wait()、wait_for()或 wait_until()的多个调用，它们就会按串行方式逐一执行。调用 notify_one()和 notify_all()只会唤醒已经开始等待的线程。

D.3 <atomic>头文件

<atomic>头文件给出了一组基本原子类型，以及针对这些类型的操作，还有一个类模板，只要用户的自定义类型符合一定的条件，即可借该模板进行原子化。

头文件内容：

```
#define ATOMIC_BOOL_LOCK_FREE 见后文详述
#define ATOMIC_CHAR_LOCK_FREE 见后文详述
#define ATOMIC_SHORT_LOCK_FREE 见后文详述
#define ATOMIC_INT_LOCK_FREE 见后文详述
#define ATOMIC_LONG_LOCK_FREE 见后文详述
#define ATOMIC_LLONG_LOCK_FREE 见后文详述
#define ATOMIC_CHAR16_T_LOCK_FREE 见后文详述
#define ATOMIC_CHAR32_T_LOCK_FREE 见后文详述
#define ATOMIC_WCHAR_T_LOCK_FREE 见后文详述
#define ATOMIC_POINTER_LOCK_FREE 见后文详述

#define ATOMIC_VAR_INIT(value)见后文详述

namespace std
{
    enum memory_order;

    struct atomic_flag;
    typedef  见后文详述  atomic_bool;
    typedef  见后文详述  atomic_char;
    typedef  见后文详述  atomic_char16_t;
    typedef  见后文详述  atomic_char32_t;
    typedef  见后文详述  atomic_schar;
    typedef  见后文详述  atomic_uchar;
    typedef  见后文详述  atomic_short;
    typedef  见后文详述  atomic_ushort;
    typedef  见后文详述  atomic_int;
    typedef  见后文详述  atomic_uint;
    typedef  见后文详述  atomic_long;
    typedef  见后文详述  atomic_ulong;
    typedef  见后文详述  atomic_llong;
    typedef  见后文详述  atomic_ullong;
    typedef  见后文详述  atomic_wchar_t;

    typedef  见后文详述   atomic_int_least8_t;
    typedef  见后文详述   atomic_uint_least8_t;
```

```
    typedef  见后文详述   atomic_int_least16_t;
    typedef  见后文详述   atomic_uint_least16_t;
    typedef  见后文详述   atomic_int_least32_t;
    typedef  见后文详述   atomic_uint_least32_t;
    typedef  见后文详述   atomic_int_least64_t;
    typedef  见后文详述   atomic_uint_least64_t;
    typedef  见后文详述   atomic_int_fast8_t;
    typedef  见后文详述   atomic_uint_fast8_t;
    typedef  见后文详述   atomic_int_fast16_t;
    typedef  见后文详述   atomic_uint_fast16_t;
    typedef  见后文详述   atomic_int_fast32_t;
    typedef  见后文详述   atomic_uint_fast32_t;
    typedef  见后文详述   atomic_int_fast64_t;
    typedef  见后文详述   atomic_uint_fast64_t;
    typedef  见后文详述   atomic_int8_t;
    typedef  见后文详述   atomic_uint8_t;
    typedef  见后文详述   atomic_int16_t;
    typedef  见后文详述   atomic_uint16_t;
    typedef  见后文详述   atomic_int32_t;
    typedef  见后文详述   atomic_uint32_t;
    typedef  见后文详述   atomic_int64_t;
    typedef  见后文详述   atomic_uint64_t;
    typedef  见后文详述  atomic_intptr_t;
    typedef  见后文详述  atomic_uintptr_t;
    typedef  见后文详述  atomic_size_t;
    typedef  见后文详述  atomic_ssize_t;
    typedef  见后文详述  atomic_ptrdiff_t;
    typedef  见后文详述  atomic_intmax_t;
    typedef  见后文详述  atomic_uintmax_t;

    template<typename T>
    struct atomic;

    extern "C" void atomic_thread_fence(memory_order order);
    extern "C" void atomic_signal_fence(memory_order order);

    template<typename T>
    T kill_dependency(T);
}
```

D.3.1 std::atomic_xxx 类定义

表 D.1 提供了各种原子整数型别的 typedef，以兼容未来的新 C 标准[①]。对于 C++17，这些 typedef 必须是对应的 std::atomic <T>特化。对于更早的 C++标准，这些

[①] 译者注：即 C18，官方正式名称是 ISO/IEC9899:2018。

typedef 则是对应特化的基类，两者具备相同的接口。

表 D.1 原子类型的 typedef 和对应的 std::atmoic<>特化版本

std::atomic_itype 原子类型	std::atomic<>相关特化类
std::atomic_char	std::atomic<char>
std::atomic_schar	std::atomic<signed char>
std::atomic_uchar	std::atomic<unsigned char>
std::atomic_short	std::atomic<short>
std::atomic_ushort	std::atomic<unsigned short>
std::atomic_int	std::atomic<int>
std::atomic_uint	std::atomic<unsigned int>
std::atomic_long	std::atomic<long>
std::atomic_ulong	std::atomic<unsigned long>
std::atomic_llong	std::atomic<long long>
std::atomic_ullong	std::atomic<unsigned long long>
std::atomic_wchar_t	std::atomic<wchar_t>
std::atomic_char16_t	std::atomic<char16_t>
std::atomic_char32_t	std::atomic<char32_t>

D.3.2 ATOMIC_xxx_LOCK_FREE 宏

这些宏设定各原子类型和对应的内建型别是否具备无锁特性。

宏定义：

```
#define ATOMIC_BOOL_LOCK_FREE 见后文详述
#define ATOMIC_CHAR_LOCK_FREE 见后文详述
#define ATOMIC_SHORT_LOCK_FREE 见后文详述
#define ATOMIC_INT_LOCK_FREE 见后文详述
#define ATOMIC_LONG_LOCK_FREE 见后文详述
#define ATOMIC_LLONG_LOCK_FREE 见后文详述
#define ATOMIC_CHAR16_T_LOCK_FREE 见后文详述
#define ATOMIC_CHAR32_T_LOCK_FREE 见后文详述
#define ATOMIC_WCHAR_T_LOCK_FREE 见后文详述
#define ATOMIC_POINTER_LOCK_FREE 见后文详述
```

ATOMIC_xxx_LOCK_FREE 的值是 0、1 或 2。

- 0 的含义是，在有符号/无符号的内建型别的对应原子类型上，从来都不存在无锁操作。
- 1 的含义是，这些原子类型中的某些特定实例具备无锁操作，而其他实例则不然。
- 2 表示全都是无锁操作。

例如，如果 ATOMIC_INT_LOCK_FREE 为 2，那么在 std::atomic<int>和 std::atomic<unsigned>的实例上的操作始终无锁。ATOMIC_POINTER_LOCK_FREE 用于描述原子指针 std::atomic<T*>，说明其特化版本上的操作是否无锁。

D.3.3 ATOMIC_VAR_INIT 宏

ATOMIC_VAR_INIT 给出了一种方法，可依照某个具体的值将原子变量初始化。

声明：

```
#define ATOMIC_VAR_INIT(value)  见后文详述
```

这个宏可以展开成一个符号序列，遵从下面表达式的形式，依照预定值初始化一个标准原子类型：

```
std::atomic<type> x = ATOMIC_VAR_INIT(val);
```

指定的值应与原子变量对应的非原子类型相容，例如：

```
std::atomic<int>i = ATOMIC_VAR_INIT(42);
std::string s;
std::atomic<std::string*> p = ATOMIC_VAR_INIT(&s);
```

这种初始化并非原子操作。假设初始化正在进行，而另一个线程访问该变量，初始化遂没有严格地先于访问完成，这将形成数据竞争，因此导致未定义行为。

D.3.4 std::memory_order 枚举型别

std::memory_order 枚举型别用于表示原子操作的内存次序约束。

声明：

```
typedef enum memory_order
{
    memory_order_relaxed,memory_order_consume,
    memory_order_acquire,memory_order_release,
    memory_order_acq_rel,memory_order_seq_cst
}memory_order;
```

带有内存次序标记的操作的行为遵循下列规则（内存次序约束的详细描述请参见第 5 章）。

1. std::memory_order_relaxed

   操作不额外提供任何内存次序约束。

2. std::memory_order_release

   标记了在指定的内存区域的释放操作，若另一获取操作会读取同一内存区域上的值，则它们形成同步关系。

3. std::memory_order_acquire

   标记了在指定的内存区域的获取操作。如果在同一内存区域上的值本来由一个释放

操作写入，则它与本操作形成同步关系。

4. std::memory_order_acq_rel

标记的必须是“读-改-写”操作，其行为在指定的内存区域上同时遵循 std::memory_order_acquire 和 std::memory_order_release 两种内存次序。

5. std::memory_order_seq_cst

标记的操作是单一全局次序的组成部分，该全局次序由先后次序一致的原子操作构成。另外，若标记的是存储操作，则其行为相当于 std::memory_order_release 操作；若标记的是载入操作，则其行为相当于 std::memory_order_acquire 操作；若标记的是“读-改-写”操作，则其行为相当于 std::memory_order_acquire 和 std::memory_order_release 操作的结合。这是所有操作的默认内存次序。

6. std::memory_order_consume

标记了指定内存区域上的消耗操作。C++17 标准已说明不应再使用该内存次序。

D.3.5 std::atomic_thread_fence()函数

std::atomic_thread_fence()函数在代码中插入“内存屏障”或“内存栅栏”，以强制代码中的操作服从内存次序约束。

声明：

```
extern "C" void atomic_thread_fence(std::memory_order order);
```

作用：

安插栅栏以施加内存次序约束。

假设栅栏带有内存次序标记 std::memory_order_release、std::memory_order_acq_rel 或 std::memory_order_seq_cst，并由某一线程负责安插，而该线程在栅栏之后还执行了存储的原子操作，如果另一获取操作从同一内存区域读出该值，那么该栅栏与获取操作形成同步关系。

假设栅栏带有内存次序标记 std::memory_order_acquire、std::memory_order_acq_rel 或 std::memory_order_seq_cst，并由某一线程负责安插，而该线程在栅栏之前曾执行过读出的原子操作，如果另一释放操作向同一内存区域写入了某个值，那么释放操作与该栅栏形成同步关系。

抛出：

无。

D.3.6 std::atomic_signal_fence()函数

std::atomic_signal_fence()函数在代码中插入“内存屏障”或“内存栅栏”，如果一个线程需要执行某信号处理函数，其中的操作便因而与同一个线程上的其他操作服从内存次序约束。

声明：

```
extern "C" void atomic_signal_fence(std::memory_order order);
```

作用：

安插栅栏以施加所需的内存次序约束。这个函数与 std::atomic_thread_fence()等价，只不过内存次序约束针对某个线程和同一线程上的信号处理函数起作用。

抛出：

无。

D.3.7 std::atomic_flag 类

std::atomic_flag 类是一种简单、直接的原子标志。它是 C++11 标准保证具备无锁特性的唯一一种数据型别（尽管在大多数实现中，许多原子类型也具备无锁特性）。std::atomic_flag 的实例或处于设立状态，或处于清零状态。

类定义：

```
struct atomic_flag
{
    atomic_flag() noexcept = default;
    atomic_flag(const atomic_flag&) = delete;
    atomic_flag& operator=(const atomic_flag&) = delete;
    atomic_flag& operator=(const atomic_flag&) volatile = delete;

    bool test_and_set(memory_order = memory_order_seq_cst) volatile
     noexcept;
    bool test_and_set(memory_order = memory_order_seq_cst) noexcept;
    void clear(memory_order = memory_order_seq_cst) volatile noexcept;
    void clear(memory_order = memory_order_seq_cst) noexcept;
};

bool atomic_flag_test_and_set(volatile atomic_flag*) noexcept;
bool atomic_flag_test_and_set(atomic_flag*) noexcept;
bool atomic_flag_test_and_set_explicit(
    volatile atomic_flag*, memory_order) noexcept;
bool atomic_flag_test_and_set_explicit(
    atomic_flag*, memory_order) noexcept;
void atomic_flag_clear(volatile atomic_flag*) noexcept;
void atomic_flag_clear(atomic_flag*) noexcept;
```

```
void atomic_flag_clear_explicit(
    volatile atomic_flag*, memory_order) noexcept;
void atomic_flag_clear_explicit(
    atomic_flag*, memory_order) noexcept;

#define ATOMIC_FLAG_INIT unspecified
```

1. std::atomic_flag()默认构造函数

C++11 没有规定按默认方式构造的 std::atomic_flag 的初值是清零还是设立。如果 std::atomic_flag 对象具有静态生存期，则按静态方式进行初始化。

声明：

```
std::atomic_flag() noexcept = default;
```

作用：

构造一个新的 std::atomic_flag 对象，但其状态并不明确。

抛出：

无。

2. std::atomic_flag 的实例依照 ATOMIC_FLAG_INIT 进行初始化

std::atomic_flag 的实例可以依照 ATOMIC_FLAG_INIT 进行初始化，即它被初始化为清零状态。如果 std::atomic_flag 对象具有静态生存期，则按静态方式进行初始化。

声明：

```
#define ATOMIC_FLAG_INIT unspecified
```

使用方式：

```
std::atomic_flag flag=ATOMIC_FLAG_INIT;
```

作用：

构造一个新的 std::atomic_flag 对象，并初始化为清零状态。

抛出：

无。

3. std::atomic_flag::test_and_set()成员函数

检查标志是否处于成立状态，同时将它设置为成立，这两项操作以原子化方式一并执行。

声明：

```
bool test_and_set(memory_order order = memory_order_seq_cst) volatile noexcept;
bool test_and_set(memory_order order = memory_order_seq_cst) noexcept;
```

作用：

以原子化方式设置标志成立。

返回：

如果标志已经处于成立状态，就返回 `true`；否则标志处于清零状态，返回 `false`。

抛出：

无。

注记：

这是针对`*this`对象所在的内存区域的“读-改-写”操作。

4. std::atomic_flag_test_and_set()非成员函数

检查标志是否正处于成立状态，同时将它设置为成立，这两项操作以原子化方式一并执行。

声明：

```
bool atomic_flag_test_and_set(volatile atomic_flag* flag) noexcept;
bool atomic_flag_test_and_set(atomic_flag* flag) noexcept;
```

作用：

```
return flag->test_and_set();
```

5. std::atomic_flag_test_and_set_explicit()非成员函数

检查标志是否正处于成立状态，同时将它设置为成立，这两项操作以原子化方式一并执行。

声明：

```
bool atomic_flag_test_and_set_explicit(
    volatile atomic_flag* flag, memory_order order) noexcept;
bool atomic_flag_test_and_set_explicit(
    atomic_flag* flag, memory_order order) noexcept;
```

作用：

```
return flag->test_and_set(order);
```

6. std::atomic_flag::clear()成员函数

以原子化方式把标志清零。

声明：

```
void clear(memory_order order = memory_order_seq_cst) volatile noexcept;
void clear(memory_order order = memory_order_seq_cst) noexcept;
```

前置条件：

所提供的参数 order 只能是 std::memory_order_relaxed、std::memory_order_release 或 std::memory_order_seq_cst。

作用：

以原子化方式把标志清零。

抛出：

无。

注记：

对于内存区域上的*this，这个操作属于“写”操作(存储操作)。

7. std::atomic_flag_clear()非成员函数

以原子化方式把标志清零。

声明：

```
void atomic_flag_clear(volatile atomic_flag* flag) noexcept;
void atomic_flag_clear(atomic_flag* flag) noexcept;
```

作用：

```
flag->clear();
```

8. std::atomic_flag_clear_explicit()非成员函数

以原子化方式把标志清零。

声明：

```
void atomic_flag_clear_explicit(
    volatile atomic_flag* flag, memory_order order) noexcept;
void atomic_flag_clear_explicit(
    atomic_flag* flag, memory_order order) noexcept;
```

作用：

```
return flag->clear(order);
```

D.3.8 std::atomic 类模板

类模板 std::atomic 用于包装其他型别，从而为其提供原子操作，接受包装的型别要满足下列条件。模板参数 BaseType 必须满足以下条件。

- 具有平实默认构造函数。
- 具有平实复制赋值操作符。

- 具有平实析构函数。

- 能够以逐位对比的方式判定是否等值（bitwise-equality comparable）。

根据上述要求，std::atomic<some-simple-struct>（包装了某个简单的结构体）和 std::atomic<some-built-in-type>（包装了内建型别）没有问题，但 std::atomic<std::string>则不然。除去泛化模板所具有的功能，内建整型和指针的特化还提供了更多操作，如 x++。std::atomic 的实例不满足 CopyConstructible 或 CopyAssignable 的型别要求[①]，原因是这两个操作无法凭单一原子操作完成。

类定义：

```
template<typename BaseType>
struct atomic
{
    using value_type = T;
    static constexpr bool is_always_lock_free = implementation-defined ;
    atomic() noexcept = default;
    constexpr atomic(BaseType) noexcept;
    BaseType operator=(BaseType) volatile noexcept;
    BaseType operator=(BaseType) noexcept;

    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;

    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;
    void store(BaseType,memory_order = memory_order_seq_cst)
        volatile noexcept;
    void store(BaseType,memory_order = memory_order_seq_cst) noexcept;
    BaseType load(memory_order = memory_order_seq_cst)
        const volatile noexcept;
    BaseType load(memory_order = memory_order_seq_cst) const noexcept;
    BaseType exchange(BaseType,memory_order = memory_order_seq_cst)
        volatile noexcept;
    BaseType exchange(BaseType,memory_order = memory_order_seq_cst)
        noexcept;

    bool compare_exchange_strong(
        BaseType&  old_value, BaseType new_value,
        memory_order order = memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_strong(
        BaseType&  old_value, BaseType new_value,
        memory_order order = memory_order_seq_cst) noexcept;
    bool compare_exchange_strong(
        BaseType&  old_value, BaseType new_value,
        memory_order success_order,
```

① 译者注：这两个术语都是针对型别的具名要求，即 named requirements，属于 C++的 concept 新特性。

```
        memory_order failure_order) volatile noexcept;
    bool compare_exchange_strong(
        BaseType&  old_value, BaseType new_value,
        memory_order success_order,
        memory_order failure_order) noexcept;
    bool compare_exchange_weak(
        BaseType&  old_value, BaseType new_value,
        memory_order order = memory_order_seq_cst)
        volatile noexcept;
    bool compare_exchange_weak(
        BaseType&  old_value, BaseType new_value,
        memory_order order = memory_order_seq_cst) noexcept;
    bool compare_exchange_weak(
        BaseType&  old_value, BaseType new_value,
        memory_order success_order,
        memory_order failure_order) volatile noexcept;
    bool compare_exchange_weak(
        BaseType&  old_value, BaseType new_value,
        memory_order success_order,
        memory_order failure_order) noexcept;

    operator BaseType () const volatile noexcept;
    operator BaseType () const noexcept;
};

template<typename BaseType>
bool atomic_is_lock_free(volatile const atomic<BaseType>*) noexcept;
template<typename BaseType>
bool atomic_is_lock_free(const atomic<BaseType>*) noexcept;
template<typename BaseType>
void atomic_init(volatile atomic<BaseType>*, void*) noexcept;
template<typename BaseType>
void atomic_init(atomic<BaseType>*, void*) noexcept;
template<typename BaseType>
BaseType atomic_exchange(volatile atomic<BaseType>*, memory_order)
    noexcept;
template<typename BaseType>
BaseType atomic_exchange(atomic<BaseType>*, memory_order) noexcept;
template<typename BaseType>
BaseType atomic_exchange_explicit(
    volatile atomic<BaseType>*, memory_order) noexcept;
template<typename BaseType>
BaseType atomic_exchange_explicit(
    atomic<BaseType>*, memory_order) noexcept;
template<typename BaseType>
void atomic_store(volatile atomic<BaseType>*, BaseType) noexcept;
template<typename BaseType>
void atomic_store(atomic<BaseType>*, BaseType) noexcept;
template<typename BaseType>
void atomic_store_explicit(
    volatile atomic<BaseType>*, BaseType, memory_order) noexcept;
template<typename BaseType>
void atomic_store_explicit(
    atomic<BaseType>*, BaseType, memory_order) noexcept;
```

```
template<typename BaseType>
BaseType atomic_load(volatile const atomic<BaseType>*) noexcept;
template<typename BaseType>
BaseType atomic_load(const atomic<BaseType>*) noexcept;
template<typename BaseType>
BaseType atomic_load_explicit(
    volatile const atomic<BaseType>*, memory_order) noexcept;
template<typename BaseType>
BaseType atomic_load_explicit(
    const atomic<BaseType>*, memory_order) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_strong(
    volatile atomic<BaseType>*,BaseType * old_value,
    BaseType new_value) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_strong(
    atomic<BaseType>*,BaseType * old_value,
    BaseType new_value) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_strong_explicit(
    volatile atomic<BaseType>*,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_strong_explicit(
    atomic<BaseType>*,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_weak(
    volatile atomic<BaseType>*,BaseType * old_value,BaseType new_value)
    noexcept;
template<typename BaseType>
bool atomic_compare_exchange_weak(
    atomic<BaseType>*,BaseType * old_value,BaseType new_value) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_weak_explicit(
    volatile atomic<BaseType>*,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_weak_explicit(
    atomic<BaseType>*,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
```

注记：

虽然上面的非成员函数以模板形式定义，但它们有可能只是对应的成员函数的一组重载版本[①]，我们不应该使用其显式设定模板参数的特化版本。

① 译者注：这些模板形式的非成员函数的存在是为了兼容 C 语言，详见 5.2.7 小节。

1. std::atomic()构造函数

依照默认初值构造一个 std::atomic 对象。

声明：

```
atomic() noexcept;
```

作用：

依照默认初值构造一个新的 std::atomic 对象。对于具有静态生存期的对象，将进行静态初始化。

注记：

如果 std::atomic 的实例具有非静态生存期，并且由默认构造函数进行初始化，就无法保证其初值符合预期。

抛出：

无。

2. std::atomic_init()非成员函数

按非原子化的方式将给定的值存储到 std::atomic<BaseType>的目标实例中。

声明：

```
template<typename BaseType>
void atomic_init(atomic<BaseType> volatile* p, BaseType v) noexcept;
template<typename BaseType>
void atomic_init(atomic<BaseType>* p, BaseType v) noexcept;
```

作用：

按非原子化的方式将给定 v 值存储到*p 对象。假设一个 atomic<BaseType>实例没有进行默认初始化，或在构造完成后进行了其他操作[①]，如果再对它发起 atomic_init()的调用，就会导致未定义行为。

抛出：

无。

注记：

由于这是非原子化存储操作，如果另一个线程在指针 p 的目标对象上同时进行并发访问（即使是原子操作），将引发数据竞争。

3. std::atomic()转换构造函数

根据给出的 BaseType 值构造一个 std::atomic 对象。

声明：

① 译者注：有可能发生一种特殊情况，即该函数针对同一个 p 指针发生两次调用，无论是否并发，都会导致未定义行为。

```
constexpr atomic(BaseType b) noexcept;
```

作用：

根据 b 值构造一个 std::atomic 对象。对于具有静态生存期的对象，将进行静态初始化。

抛出：

无。

4. std::atomic 转换赋值操作符

将新值存储到*this 对象中。

声明：

```
BaseType operator=(BaseType b) volatile noexcept;
BaseType operator=(BaseType b) noexcept;
```

作用：

```
return this->store(b);
```

5. std::atomic::is_lock_free()成员函数

判定*this 对象上的操作是否无锁。

声明：

```
bool is_lock_free() const volatile noexcept;
bool is_lock_free() const noexcept;
```

返回：

如果*this 对象上的操作均是无锁操作，就返回 true，否则返回 false。

抛出：

无。

6. std::atomic_is_lock_free()非成员函数

判定目标对象上的操作是否无锁。

声明：

```
template<typename BaseType>
bool atomic_is_lock_free(volatile const atomic<BaseType>* p) noexcept;
template<typename BaseType>
bool atomic_is_lock_free(const atomic<BaseType>* p) noexcept;
```

作用：

```
return p->is_lock_free();
```

7. std::atomic::is_always_lock_free()静态成员函数

判定所属型别的全部对象是否总会具备无锁操作。

声明：

```
static constexpr bool is_always_lock_free() = implementation-defined;
```

提示：implementation-defined 意为线程库实现自行选定修饰符。

返回：

如果所属型别的全部对象肯定具备无锁操作，就返回 true，否则返回 false。

8. std::atomic::load()成员函数

以原子化方式载入 atomic<BaseType>实例的当前值。

声明：

```
BaseType load(memory_order order = memory_order_seq_cst)
    const volatile noexcept;
BaseType load(memory_order order = memory_order_seq_cst) const noexcept;
```

前置条件：

所提供的参数 order 只能是 std::memory_order_relaxed、std::memory_order_acquire、std::memory_order_consume 或 std::memory_order_seq_cst 之一。

作用：

以原子化方式载入*this 对象所存有的值。

返回：

返回*this 对象在本函数调用发生时所存有的值。

抛出：

无。

注记：

这是针对*this 对象所在的内存区域的原子化载入操作。

9. std::atomic_load()非成员函数

以原子化方式载入 std::atomic 实例的当前值。

声明：

```
template<typename BaseType>
BaseType atomic_load(volatile const atomic<BaseType>* p) noexcept;
template<typename BaseType>
BaseType atomic_load(const atomic<BaseType>* p) noexcept;
```

作用：

```
return p->load();
```

10. std::atomic_load_explicit()非成员函数

以原子化方式载入 std::atomic 实例的当前值。

声明：

```
template<typename BaseType>
BaseType atomic_load_explicit(
    volatile const atomic<BaseType>* p, memory_order order) noexcept;
template<typename BaseType>
BaseType atomic_load_explicit(
    const atomic<BaseType>* p, memory_order order) noexcept;
```

作用：

```
return p->load(order);
```

11. std::atomic::operator BastType()转换操作符

载入*this 对象所存储的值。

声明：

```
operator BaseType() const volatile noexcept;
operator BaseType() const noexcept;
```

作用：

```
return this->load();
```

12. std::atomic::store()成员函数

以原子化方式将新值存储到 atomic<BaseType>的实例中。

声明：

```
void store(BaseType new_value,memory_order order = memory_order_seq_cst)
    volatile noexcept;
void store(BaseType new_value,memory_order order = memory_order_seq_cst)
    noexcept;
```

前置条件：

所提供的参数 order 只能是 std::memory_order_relaxed、std::memory_order_release 或 std::memory_order_seq_cst 之一。

作用：

以原子化方式将新值 new_value 存储到*this 对象中。

抛出：

无。

注记：

这是针对*this 对象所在的内存区域的原子化存储操作。

13. std::atomic_store()非成员函数

以原子化方式将新值存储到 atomic<BaseType>的实例中。

声明：

```
template<typename BaseType>
void atomic_store(volatile atomic<BaseType>* p, BaseType new_value)
    noexcept;
template<typename BaseType>
void atomic_store(atomic<BaseType>* p, BaseType new_value) noexcept;
```

作用：

```
p->store(new_value);
```

14. std::atomic_explicit()非成员函数

以原子化方式存储一个新值到 atomic<BaseType>实例中。

声明：

```
template<typename BaseType>
void atomic_store_explicit(
    volatile atomic<BaseType>* p, BaseType new_value, memory_order order)
    noexcept;
template<typename BaseType>
void atomic_store_explicit(
    atomic<BaseType>* p, BaseType new_value, memory_order order) noexcept;
```

作用：

```
p->store(new_value,order);
```

15. std::atomic::exchange()成员函数

读取旧值，同时存储新值，这两项操作以原子化方式一并执行。

声明：

```
BaseType exchange(
    BaseType new_value,
    memory_order order = memory_order_seq_cst)
    volatile noexcept;
```

作用：

获取*this 对象现存的旧值，同时向*this 对象存储新值，这两项操作以原子化方式一并执行。

返回：

返回*this 对象在存储操作发生前一刻的值。

抛出：

无。

注记：

这是针对*this 对象所在的内存区域的原子化“读-改-写”操作。

16. std::atomic_exchange()非成员函数

读取 atomic<BaseType>实例的旧值，同时向该实例存储新值，这两项操作以原子化方式一并执行。

声明：

```
template<typename BaseType>
BaseType atomic_exchange(volatile atomic<BaseType>* p, BaseType new_value)
    noexcept;
template<typename BaseType>
BaseType atomic_exchange(atomic<BaseType>* p, BaseType new_value) noexcept;
```

作用：

```
return p->exchange(new_value);
```

17. std::atomic_exchange_explicit()非成员函数

读取 atomic<BaseType>实例的旧值，同时向该实例存储新值，这两项操作以原子化方式一并执行。

声明：

```
template<typename BaseType>
BaseType atomic_exchange_explicit(
    volatile atomic<BaseType>* p, BaseType new_value, memory_order order)
    noexcept;
template<typename BaseType>
BaseType atomic_exchange_explicit(
    atomic<BaseType>* p, BaseType new_value, memory_order order) noexcept;
```

作用：

```
return p->exchange(new_value,order);
```

18. std::atomic::compare_exchange_strong()成员函数

对比存有的旧值和期望值，若相等则存储新值，否则就将期望值更新为所读取的值，这些操作以原子化方式一并执行。

声明：

```
bool compare_exchange_strong(
    BaseType& expected,BaseType new_value,
    memory_order order = std::memory_order_seq_cst) volatile noexcept;
bool compare_exchange_strong(
    BaseType& expected,BaseType new_value,
    memory_order order = std::memory_order_seq_cst) noexcept;
bool compare_exchange_strong(
    BaseType& expected,BaseType new_value,
    memory_order success_order,memory_order failure_order)
```

```
    volatile noexcept;
bool compare_exchange_strong(
    BaseType& expected,BaseType new_value,
    memory_order success_order,memory_order failure_order) noexcept;
```

前置条件：

参数 failure_order 的取值不得为 std::memory_order_release 或 std::memory_order_ acq_rel。

作用：

逐位对比参数 expected 的值和*this 对象中存储的值，如果相等就将 new_value 的值存储到*this 对象中；否则，把 expected 更新为所读取的值，这些操作以原子化方式一并执行。

返回：

若*this 对象中存有的值等于 expected 的值，就返回 true，否则返回 false。

抛出：

无。

注记 1：

本函数有两个重载接收 3 个参数，另两个重载接收 4 个参数，而在它们的参数之间，如果 success_order==order 和 failure_order==order 同时成立，那么 3 个参数的重载与 4 个参数的重载分别对应等价。不过，若 order 取值 std::memory_order_acq_rel，而 failure_order 则应取值 std::memory_order_acquire，才可以保持等价；又若 order 取值 std::memory_order_release，而 failure_order 则应取值 std::memory_order_relaxed，才可以保持等价。

注记 2：

如果本函数调用的返回值是 true，那么实际执行的是针对*this 对象所在的内存区域的原子化“读-改-写”操作，且服从内存次序 success_order；否则，实际执行的是针对*this 对象所在的内存区域的原子化载入操作，且服从内存次序 failure_order。

19. std::atomic_compare_exchange_strong()非成员函数

若存有的值等于期望值，则将新值存储到实例中，否则，就将期望值更新为读取的值，这些操作以原子化方式一并执行。

声明：

```
template<typename BaseType>
bool atomic_compare_exchange_strong(
    volatile atomic<BaseType>* p, BaseType * old_value,BaseType new_value)
    noexcept;
template<typename BaseType>
bool atomic_compare_exchange_strong(
```

```
    atomic<BaseType>* p,BaseType * old_value, BaseType new_value) noexcept;
```

作用：

```
return p->compare_exchange_strong(*old_value, new_value);
```

20. std::atomic_compare_exchange_strong_explicit()非成员函数

对比存有的旧值和期望值，若相等则存储新值，否则就将期望值更新为所读取的值，这些操作以原子化方式一并执行。

声明：

```
template<typename BaseType>
bool atomic_compare_exchange_strong_explicit(
    volatile atomic<BaseType>* p,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_strong_explicit(
    atomic<BaseType>* p,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
```

作用：

```
return p->compare_exchange_strong(
    *old_value,new_value,success_order,failure_order) noexcept;
```

21. std::atomic::compare_exchange_weak()成员函数

对比存有的旧值和期望值，如果两者相等，且对比操作和更新操作能以原子化方式一并执行，就存储新值；如果两者互异，或对比操作和更新操作无法以原子化方式一并执行，就将期望值更新为所读取的值。

声明：

```
bool compare_exchange_weak(
    BaseType& expected,BaseType new_value,
    memory_order order = std::memory_order_seq_cst) volatile noexcept;
bool compare_exchange_weak(
    BaseType& expected,BaseType new_value,
    memory_order order = std::memory_order_seq_cst) noexcept;
bool compare_exchange_weak(
    BaseType& expected,BaseType new_value,
    memory_order success_order,memory_order failure_order)
    volatile noexcept;
bool compare_exchange_weak(
    BaseType& expected,BaseType new_value,
    memory_order success_order,memory_order failure_order) noexcept;
```

前置条件：

参数failure_order不得取值std::memory_order_release或std::memory_

order_acq_rel。

作用：

逐位对比参数 expected 的值和*this 对象中存储的值，如果两者相等，且对比操作和更新操作能以原子化方式一并执行，就将 new_value 的值存储到*this 对象中；如果两者互异，或对比操作和更新操作无法以原子化方式一并执行，就把 expected 更新为所读取的值。

返回：

若*this 对象中存有的值等于 expected 的值，且 new_value 成功存储到*this 对象中，就返回 true；否则返回 false。

抛出：

无。

注记 1：

本函数有两个重载接收 3 个参数，另两个重载接收 4 个参数，而在它们的参数之间，如果 success_order==order 和 failure_order==order 同时成立，那么 3 个参数的重载与 4 个参数的重载分别对应等价。不过，若 order 取值 std::memory_order_acq_rel，而 failure_order 则应取值 std::memory_order_acquire，才可以保持等价；又若 order 取值 std::memory_order_release，而 failure_order 则应取值 std::memory_order_relaxed，才可以保持等价。

注记 2：

如果本函数调用的返回值是 true，那么实际执行的是针对*this 对象所在的内存区域的原子化“读-改-写”操作，且服从内存次序 success_order；否则，实际执行的是针对*this 对象所在的内存区域的原子化载入操作，且服从内存次序 failure_order。

22. std::atomic_compare_exchange_weak()非成员函数

对比存有的旧值和期望值，如果两者相等，且对比操作和更新操作能以原子化方式一并执行，就存储新值。如果两者不相等，或更新未进行，那期望值会更新为新值。

声明：

```
template<typename BaseType>
bool atomic_compare_exchange_weak(
    volatile atomic<BaseType>* p,BaseType * old_value,BaseType new_value)
    noexcept;
template<typename BaseType>
bool atomic_compare_exchange_weak(
    atomic<BaseType>* p,BaseType * old_value,BaseType new_value) noexcept;
```

作用：

```
return p->compare_exchange_weak(*old_value,new_value);
```

23. std::atomic_compare_exchange_weak_explicit()非成员函数

对比新值和期望值，如果两者相等，那么存储新值并且以原子化方式进行更新操作；如果两者互异，或对比操作和更新操作无法以原子化方式一并执行，就将期望值更新为所读取的值。

声明：

```
template<typename BaseType>
bool atomic_compare_exchange_weak_explicit(
    volatile atomic<BaseType>* p,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_weak_explicit(
    atomic<BaseType>* p,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
```

作用：

```
return p->compare_exchange_weak(
    *old_value,new_value,success_order,failure_order);
```

D.3.9 类模板 std::atomic 的特化

标准库给出了类模板 `std::atomic` 针对整型和指针型别的特化。除了泛化模板所具备的操作，标准库针对整型特化的模板另外提供了原子化相加、原子化相减和原子化位运算。除了泛化模板所具备的操作，标准库针对指针型别特化的模板增添了算术形式的原子化指针运算功能。

标准库针对各种整型给出了下列特化：

```
std::atomic<bool>
std::atomic<char>
std::atomic<signed char>
std::atomic<unsigned char>
std::atomic<short>
std::atomic<unsigned short>
std::atomic<int>
std::atomic<unsigned>
std::atomic<long>
std::atomic<unsigned long>
std::atomic<long long>
std::atomic<unsigned long long>
std::atomic<wchar_t>
std::atomic<char16_t>
std::atomic<char32_t>
```

以及 `std::atomic<T*>`，其中 `T` 可以是任意型别。

D.3.10 std::atomic<integral-type>特化

std::atomic<*integral-type*>是为每一个基础整型提供的 std::atomic 类模板，其中提供了一套完整的整型操作。下面的特化模板也适用于 std::atomic 类模板：

```
std::atomic<char>
std::atomic<signed char>
std::atomic<unsigned char>
std::atomic<short>
std::atomic<unsigned short>
std::atomic<int>
std::atomic<unsigned>
std::atomic<long>
std::atomic<unsigned long>
std::atomic<long long>
std::atomic<unsigned long long>
std::atomic<wchar_t>
std::atomic<char16_t>
std::atomic<char32_t>
```

这些特化的实例不满足 CopyConstructible 或 CopyAssignable 的型别要求[①]，原因是这两个操作无法凭单一原子操作完成。

类定义：

```
template<>
struct atomic<integral-type>
{
    atomic() noexcept = default;
    constexpr atomic(integral-type) noexcept;
    bool operator=(integral-type) volatile noexcept;

    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;

    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;

    void store(integral-type,memory_order = memory_order_seq_cst)
        volatile noexcept;
    void store(integral-type,memory_order = memory_order_seq_cst) noexcept;
    integral-type load(memory_order = memory_order_seq_cst)
        const volatile noexcept;
    integral-type load(memory_order = memory_order_seq_cst) const noexcept;
    integral-type exchange(
        integral-type,memory_order = memory_order_seq_cst)
```

① 译者注：这两个术语都是针对型别的具名要求，即 named requirements，属于 C++20 的 concept 新特性。

```
    volatile noexcept;
integral-type exchange(
    integral-type,memory_order = memory_order_seq_cst) noexcept;

bool compare_exchange_strong(
    integral-type & old_value,integral-type new_value,
     memory_order order = memory_order_seq_cst) volatile noexcept;
bool compare_exchange_strong(
    integral-type & old_value,integral-type new_value,
     memory_order order = memory_order_seq_cst) noexcept;
bool compare_exchange_strong(
    integral-type & old_value,integral-type new_value,
     memory_order success_order,memory_order failure_order)
    volatile noexcept;
bool compare_exchange_strong(
    integral-type & old_value,integral-type new_value,
     memory_order success_order,memory_order failure_order) noexcept;
bool compare_exchange_weak(
    integral-type & old_value,integral-type new_value,
     memory_order order = memory_order_seq_cst) volatile noexcept;
bool compare_exchange_weak(
    integral-type & old_value,integral-type new_value,
     memory_order order = memory_order_seq_cst) noexcept;
bool compare_exchange_weak(
    integral-type & old_value,integral-type new_value,
     memory_order success_order,memory_order failure_order)
    volatile noexcept;
bool compare_exchange_weak(
    integral-type & old_value,integral-type new_value,
     memory_order success_order,memory_order failure_order) noexcept;

operator integral-type() const volatile noexcept;
operator integral-type() const noexcept;

integral-type fetch_add(
    integral-type,memory_order = memory_order_seq_cst)
    volatile noexcept;
integral-type fetch_add(
    integral-type,memory_order = memory_order_seq_cst) noexcept;
integral-type fetch_sub(
    integral-type,memory_order = memory_order_seq_cst)
    volatile noexcept;
integral-type fetch_sub(
    integral-type,memory_order = memory_order_seq_cst) noexcept;
integral-type fetch_and(
    integral-type,memory_order = memory_order_seq_cst)
    volatile noexcept;
integral-type fetch_and(
    integral-type,memory_order = memory_order_seq_cst) noexcept;
integral-type fetch_or(
    integral-type,memory_order = memory_order_seq_cst)
    volatile noexcept;
integral-type fetch_or(
    integral-type,memory_order = memory_order_seq_cst) noexcept;
```

```
    integral-type fetch_xor(
        integral-type,memory_order = memory_order_seq_cst)
        volatile noexcept;
    integral-type fetch_xor(
        integral-type,memory_order = memory_order_seq_cst) noexcept;

    integral-type operator++() volatile noexcept;
    integral-type operator++() noexcept;
    integral-type operator++(int) volatile noexcept;
    integral-type operator++(int) noexcept;
    integral-type operator--() volatile noexcept;
    integral-type operator--() noexcept;
    integral-type operator--(int) volatile noexcept;
    integral-type operator--(int) noexcept;

    integral-type operator+=(integral-type) volatile noexcept;
    integral-type operator+=(integral-type) noexcept;
    integral-type operator-=(integral-type) volatile noexcept;
    integral-type operator-=(integral-type) noexcept;
    integral-type operator&=(integral-type) volatile noexcept;
    integral-type operator&=(integral-type) noexcept;
    integral-type operator|=(integral-type) volatile noexcept;
    integral-type operator|=(integral-type) noexcept;
    integral-type operator^=(integral-type) volatile noexcept;
    integral-type operator^=(integral-type) noexcept;
};

bool atomic_is_lock_free(volatile const atomic<integral-type>*) noexcept;
bool atomic_is_lock_free(const atomic<integral-type>*) noexcept;
void atomic_init(volatile atomic<integral-type>*,integral-type) noexcept;
void atomic_init(atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_exchange(
    volatile atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_exchange(
    atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_exchange_explicit(
    volatile atomic<integral-type>*,integral-type, memory_order) noexcept;
gral-typeinte atomic_exchange_explicit(
    atomic<integral-type>*,integral-type, memory_order) noexcept;
void atomic_store(volatile atomic<integral-type>*,integral-type) noexcept;
void atomic_store(atomic<integral-type>*,integral-type) noexcept;
void atomic_store_explicit(
    volatile atomic<integral-type>*,integral-type, memory_order) noexcept;
void atomic_store_explicit(
    atomic<integral-type>*,integral-type, memory_order) noexcept;
integral-type atomic_load(volatile const atomic<integral-type>*) noexcept;
integral-type atomic_load(const atomic<integral-type>*) noexcept;
integral-type atomic_load_explicit(
    volatile const atomic<integral-type>*,memory_order) noexcept;
integral-type atomic_load_explicit(
    const atomic<integral-type>*,memory_order) noexcept;
bool atomic_compare_exchange_strong(
    volatile atomic<integral-type>*,
    integral-type* old_value,integral-type new_value) noexcept;
```

```
bool atomic_compare_exchange_strong(
    atomic<integral-type>*,
    integral-type* old_value,integral-type new_value) noexcept;
bool atomic_compare_exchange_strong_explicit(
    volatile atomic<integral-type>*,
    integral-type* old_value,integral-type new_value,
    memory_order success_order,memory_order failure_order) noexcept;
bool atomic_compare_exchange_strong_explicit(
    atomic<integral-type>*,
    integral-type* old_value,integral-type new_value,
    memory_order success_order,memory_order failure_order) noexcept;
bool atomic_compare_exchange_weak(
    volatile atomic<integral-type>*,
    integral-type* old_value,integral-type new_value) noexcept;
bool atomic_compare_exchange_weak(
    atomic<integral-type>*,
    integral-type* old_value,integral-type new_value) noexcept;
bool atomic_compare_exchange_weak_explicit(
    volatile atomic<integral-type>*,
    integral-type* old_value,integral-type new_value,
    memory_order success_order,memory_order failure_order) noexcept;
bool atomic_compare_exchange_weak_explicit(
    atomic<integral-type>*,
    integral-type* old_value,integral-type new_value,
    memory_order success_order,memory_order failure_order) noexcept;

integral-type atomic_fetch_add(
    volatile atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_fetch_add(
    atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_fetch_add_explicit(
    volatile atomic<integral-type>*,integral-type, memory_order) noexcept;
integral-type atomic_fetch_add_explicit(
    atomic<integral-type>*,integral-type, memory_order) noexcept;
integral-type atomic_fetch_sub(
    volatile atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_fetch_sub(
    atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_fetch_sub_explicit(
    volatile atomic<integral-type>*,integral-type, memory_order) noexcept;
integral-type atomic_fetch_sub_explicit(
    atomic<integral-type>*,integral-type, memory_order) noexcept;
integral-type atomic_fetch_and(
    volatile atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_fetch_and(
    atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_fetch_and_explicit(
    volatile atomic<integral-type>*,integral-type, memory_order) noexcept;
integral-type atomic_fetch_and_explicit(
    atomic<integral-type>*,integral-type, memory_order) noexcept;
integral-type atomic_fetch_or(
    volatile atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_fetch_or(
    atomic<integral-type>*,integral-type) noexcept;
```

```
integral-type atomic_fetch_or_explicit(
    volatile atomic<integral-type>*,integral-type, memory_order) noexcept;
integral-type atomic_fetch_or_explicit(
    atomic<integral-type>*,integral-type, memory_order) noexcept;
integral-type atomic_fetch_xor(
    volatile atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_fetch_xor(
    atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_fetch_xor_explicit(
    volatile atomic<integral-type>*,integral-type, memory_order) noexcept;
integral-type atomic_fetch_xor_explicit(
    atomic<integral-type>*,integral-type, memory_order) noexcept;
```

如果泛化模板所提供的操作与上述函数同名（见附录 D.3.8 小节），那么它们具有相同的语义。

1. std::atomic<integral-type>::fetch_add()成员函数

载入存有的值，并与给定的 i 值相加，以所得之和替换旧值，这些操作以原子化方式一并执行。

声明：

```
integral-type fetch_add(
    integral-type i,memory_order order = memory_order_seq_cst)
    volatile noexcept;
integral-type fetch_add(
    integral-type i,memory_order order = memory_order_seq_cst) noexcept;
```

作用：

获取*this 对象中存有的值，并与给定的 i 值相加，将所得之和存储到*this 对象中，这些操作以原子化方式一并执行。

返回：

返回*this 对象在存储动作发生前一刻的值。

抛出：

无。

注记：

这是针对*this 对象所在的内存区域的原子化“读-改-写”操作。

2. std::atomic_fetch_add()非成员函数

读取 atomic<*integral-type*>实例的值，并与给定的 i 值相加，以所得之和替换实例中的旧值，这些操作以原子化方式一并执行。

声明：

```
integral-type atomic_fetch_add(
    volatile atomic<integral-type>* p, integral-type i) noexcept;
```

```
integral-type atomic_fetch_add(
    atomic<integral-type>* p, integral-type i) noexcept;
```

作用：

```
return p->fetch_add(i);
```

3. std::atomic_fetch_add_explicit()非成员函数

读取 atomic<*integral-type*>实例的值，并与给定的 i 值相加，以所得之和替换实例中的旧值，这些操作以原子化方式一并执行。

声明：

```
integral-type atomic_fetch_add_explicit(
    volatile atomic<integral-type>* p, integral-type i,
    memory_order order) noexcept;
integral-type atomic_fetch_add_explicit(
    atomic<integral-type>* p, integral-type i, memory_order order)
    noexcept;
```

作用：

```
return p->fetch_add(i,order);
```

4. std::atomic<integral-type>::fetch_sub()成员函数

载入存有的值，并与给定的 i 值相减，以所得之差替换旧值，这些操作以原子化方式一并执行。

声明：

```
integral-type fetch_sub(
    integral-type i,memory_order order = memory_order_seq_cst)
    volatile noexcept;
integral-type fetch_sub(
    integral-type i,memory_order order = memory_order_seq_cst) noexcept;
```

作用：

获取*this 对象中存有的值，并与给定的 i 值相减，将所得之差存储到*this 对象中，这些操作以原子化方式一并执行。

返回：

返回*this 对象在存储动作发生前一刻的值。

抛出：

无。

注记：

这是针对*this 对象所在的内存区域的原子化“读-改-写”操作。

5. std::atomic_fetch_sub()非成员函数

读取 atomic<*integral-type*>实例的值，并与给定的 i 值相减，以所得之差替换实例中的旧值，这些操作以原子化方式一并执行。

声明：

```
integral-type atomic_fetch_sub(
    volatile atomic<integral-type>* p, integral-type i) noexcept;
integral-type atomic_fetch_sub(
    atomic<integral-type>* p, integral-type i) noexcept;
```

作用：

```
return p->fetch_sub(i);
```

6. std::atomic_fetch_sub_explicit()非成员函数

读取 atomic<*integral-type*>实例的值，并与给定的 i 值相减，以所得之差替换实例中的旧值，这些操作以原子化方式一并执行。

声明：

```
integral-type atomic_fetch_sub_explicit(
    volatile atomic<integral-type>* p, integral-type i,
    memory_order order) noexcept;
integral-type atomic_fetch_sub_explicit(
    atomic<integral-type>* p, integral-type i, memory_order order)
    noexcept;
```

作用：

```
return p->fetch_sub(i,order);
```

7. std::atomic<integral-type>::fetch_and()成员函数

载入存有的值，并与给定的 i 值进行按位与运算，以所得结果替换旧值，这些操作以原子化方式一并执行。

声明：

```
integral-type fetch_and(
    integral-type i,memory_order order = memory_order_seq_cst)
    volatile noexcept;
integral-type fetch_and(
    integral-type i,memory_order order = memory_order_seq_cst) noexcept;
```

作用：

获取*this 对象中存有的值，并与给定的 i 值进行按位与运算，将所得结果存储到*this 对象中，这些操作以原子化方式一并执行。

返回：

返回*this 对象在存储动作发生前一刻的值。

抛出：

无。

注记：

这是针对*this 对象所在的内存区域的原子化“读-改-写”操作。

8. std::atomic_fetch_and()非成员函数

读取 atomic<*integral-type*>实例的值，并与给定的 i 值进行按位与运算，将所得结果替换实例中的旧值，这些操作以原子化方式一并执行。

声明：

```
integral-type atomic_fetch_and(
    volatile atomic<integral-type>* p, integral-type i) noexcept;
integral-type atomic_fetch_and(
    atomic<integral-type>* p, integral-type i) noexcept;
```

作用：

```
return p->fetch_and(i);
```

9. std::atomic_fetch_and_explicit()非成员函数

读取 atomic<*integral-type*>实例的值，并与给定的 i 值进行按位与运算，将所得结果替换实例中的旧值，这些操作以原子化方式一并执行。

声明：

```
integral-type atomic_fetch_and_explicit(
    volatile atomic<integral-type>* p, integral-type i,
    memory_order order) noexcept;
integral-type atomic_fetch_and_explicit(
    atomic<integral-type>* p, integral-type i, memory_order order)
    noexcept;
```

作用：

```
return p->fetch_and(i,order);
```

10. std::atomic<integral-type>::fetch_or()成员函数

载入存有的值，并与给定的 i 值进行按位或运算，以所得结果替换旧值，这些操作以原子化方式一并执行。

声明：

```
integral-type fetch_or(
    integral-type i,memory_order order = memory_order_seq_cst)
    volatile noexcept;
integral-type fetch_or(
```

```
    integral-type i,memory_order order = memory_order_seq_cst) noexcept;
```

作用：

获取*this 对象中存有的值，并与给定的 i 值进行按位或运算，将所得结果存储到*this 对象中，这些操作以原子化方式一并执行。

返回：

返回*this 对象在存储动作发生前一刻的值。

抛出：

无。

注记：

这是针对*this 对象所在的内存区域的原子化“读-改-写”操作。

11. std::atomic_fetch_or()非成员函数

读取 atomic<*integral-type*>实例的值，并与给定的 i 值进行按位或运算，以所得结果替换实例中的旧值，这些操作以原子化方式一并执行。

声明：

```
integral-type atomic_fetch_or(
    volatile atomic<integral-type>* p, integral-type i) noexcept;
integral-type atomic_fetch_or(
    atomic<integral-type>* p, integral-type i) noexcept;
```

作用：

```
return p->fetch_or(i);
```

12. std::atomic_fetch_or_explicit()非成员函数

读取 atomic<*integral-type*>实例的值，并与给定的 i 值进行按位或运算，以所得结果替换实例中的旧值，这些操作以原子化方式一并执行。

声明：

```
integral-type atomic_fetch_or_explicit(
    volatile atomic<integral-type>* p, integral-type i,
    memory_order order) noexcept;
integral-type atomic_fetch_or_explicit(
    atomic<integral-type>* p, integral-type i, memory_order order)
    noexcept;
```

作用：

```
return p->fetch_or(i,order);
```

13. std::atomic<integral-type>::fetch_xor()成员函数

载入存有的值，并与给定的 i 值进行按位异或运算，以所得结果替换旧值，这些操

作以原子化方式一并执行。

声明：

```
integral-type fetch_xor(
    integral-type i,memory_order order = memory_order_seq_cst)
    volatile noexcept;
integral-type fetch_xor(
    integral-type i,memory_order order = memory_order_seq_cst) noexcept;
```

作用：

获取*this 对象中存有的值，并与给定的 i 值进行按位异或运算，将所得结果存储到*this 对象中，这些操作以原子化方式一并执行。

返回：

返回*this 对象之前存储的值。

抛出：

无。

注记：

这是针对*this 对象所在的内存区域的原子化“读-改-写”操作。

14. std::atomic_fetch_xor()非成员函数

读取 atomic<*integral-type*>实例的值，并与给定的 i 值进行按位异或运算，以所得结果替换实例中的旧值，这些操作以原子化方式一并执行。

声明：

```
integral-type atomic_fetch_xor(
    volatile atomic<integral-type>* p, integral-type i) noexcept;
integral-type atomic_fetch_xor(
    atomic<integral-type>* p, integral-type i) noexcept;
```

作用：

```
return p->fetch_xor(i);
```

15. std::atomic_fetch_xor_explicit()非成员函数

读取 atomic<*integral-type*>实例的值，并与给定的 i 值进行按位异或运算，以所得结果替换实例中的旧值，这些操作以原子化方式一并执行。

声明：

```
integral-type atomic_fetch_xor_explicit(
    volatile atomic<integral-type>* p, integral-type i,
    memory_order order) noexcept;
integral-type atomic_fetch_xor_explicit(
    atomic<integral-type>* p, integral-type i, memory_order order)
    noexcept;
```

作用：

```
return p->fetch_xor(i,order);
```

16. std::atomic<integral-type>::operator++ 前缀自增运算符

令*this 对象中存有的值自增，并返回新值，全部操作以原子化方式一并执行。

声明：

```
integral-type operator++() volatile noexcept;
integral-type operator++() noexcept;
```

作用：

```
return this->fetch_add(1) + 1;
```

17. std::atomic<integral-type>::operator++ 后缀自增运算符

令*this 对象中存有的值自增，并返回旧值，全部操作以原子化方式一并执行。

声明：

```
integral-type operator++(int) volatile noexcept;
integral-type operator++(int) noexcept;
```

作用：

```
return this->fetch_add(1);
```

18. std::atomic<integral-type>::operator-- 前缀自减运算符

令*this 对象中存有的值自减，并返回新值，全部操作以原子化方式一并执行。

声明：

```
integral-type operator--() volatile noexcept;
integral-type operator--() noexcept;
```

作用：

```
return this->fetch_sub(1) - 1;
```

19. std::atomic<integral-type>::operator-- 后缀自减运算符

令*this 对象中存有的值自减，并返回旧值，全部操作以原子化方式一并执行。

声明：

```
integral-type operator--(int) volatile noexcept;
integral-type operator--(int) noexcept;
```

作用：

```
return this->fetch_sub(1);
```

20. std::atomic<integral-type>::operator+= 复合赋值操作符

使*this 对象中存有的值与给定的值相加，将所得之和存储到*this 对象中，并返回新值，全部操作以原子化方式一并执行。

声明：

```
integral-type operator+=(integral-type i) volatile noexcept;
integral-type operator+=(integral-type i) noexcept;
```

作用：

```
return this->fetch_add(i) + i;
```

21. std::atomic<integral-type>::operator-= 复合赋值操作符

使*this 对象中存有的值与给定的值相减，将所得之差存储到*this 对象中，并返回新值，全部操作以原子化方式一并执行。

声明：

```
integral-type operator-=(integral-type i) volatile noexcept;
integral-type operator-=(integral-type i) noexcept;
```

作用：

```
return this->fetch_sub(i,std::memory_order_seq_cst) - i;
```

22. std::atomic<integral-type>::operator&= 复合赋值操作符

使*this 对象中存有的值与给定的值进行按位与运算，将所得结果存储到*this 对象中，并返回新值，全部操作以原子化方式一并执行。

声明：

```
integral-type operator&=(integral-type i) volatile noexcept;
integral-type operator&=(integral-type i) noexcept;
```

作用：

```
return this->fetch_and(i)& i;
```

23. std::atomic<integral-type>::operator|= 复合赋值操作符

使*this 对象中存有的值与给定的值进行按位或运算，将所得结果存储到*this 对象中，并返回新值，全部操作以原子化方式一并执行。

声明：

```
integral-type operator|=(integral-type i) volatile noexcept;
integral-type operator|=(integral-type i) noexcept;
```

作用：

```
return this->fetch_or(i,std::memory_order_seq_cst) | i;
```

24. std::atomic<integral-type>::operator^= 复合赋值操作符

使*this对象中存有的值与给定的值进行按位异或运算，将所得结果存储到*this对象中，并返回新值，全部操作以原子化方式一并执行。

声明：

```
integral-type operator^=(integral-type i) volatile noexcept;
integral-type operator^=(integral-type i) noexcept;
```

作用：

```
return this->fetch_xor(i,std::memory_order_seq_cst) ^ i;
```

D.3.11 std::atomic<T*>偏特化

类模板std::atomic的偏特化std::atomic<T*>针对各种指针给出了对应的原子化数据型别，它具有指针的全套功能。std::atomic的实例不满足CopyConstructible或CopyAssignable的型别要求[①]，原因是这两个操作无法凭单一原子操作完成。

类定义：

```
template<typename T>
struct atomic<T*>
{
    atomic() noexcept = default;
    constexpr atomic(T*) noexcept;
    bool operator=(T*) volatile;
    bool operator=(T*);

    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;

    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;
    void store(T*,memory_order = memory_order_seq_cst) volatile noexcept;
    void store(T*,memory_order = memory_order_seq_cst) noexcept;
    T* load(memory_order = memory_order_seq_cst) const volatile noexcept;
    T* load(memory_order = memory_order_seq_cst) const noexcept;
    T* exchange(T*,memory_order = memory_order_seq_cst) volatile noexcept;
    T* exchange(T*,memory_order = memory_order_seq_cst) noexcept;
```

① 译者注：这两个术语都是针对型别的具名要求，即 named requirements，属于C++的 concept 新特性的其中一部分。

```
    bool compare_exchange_strong(
        T* & old_value, T* new_value,
        memory_order order = memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_strong(
        T* & old_value, T* new_value,
        memory_order order = memory_order_seq_cst) noexcept;
    bool compare_exchange_strong(
        T* & old_value, T* new_value,
        memory_order success_order,memory_order failure_order)
        volatile noexcept;
    bool compare_exchange_strong(
        T* & old_value, T* new_value,
        memory_order success_order,memory_order failure_order) noexcept;
    bool compare_exchange_weak(
        T* & old_value, T* new_value,
        memory_order order = memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_weak(
        T* & old_value, T* new_value,
        memory_order order = memory_order_seq_cst) noexcept;
    bool compare_exchange_weak(
        T* & old_value, T* new_value,
        memory_order success_order,memory_order failure_order)
        volatile noexcept;
    bool compare_exchange_weak(
        T* & old_value, T* new_value,
        memory_order success_order,memory_order failure_order) noexcept;

    operator T*() const volatile noexcept;
    operator T*() const noexcept;

    T* fetch_add(
        ptrdiff_t,memory_order = memory_order_seq_cst) volatile noexcept;
    T* fetch_add(
        ptrdiff_t,memory_order = memory_order_seq_cst) noexcept;
    T* fetch_sub(
        ptrdiff_t,memory_order = memory_order_seq_cst) volatile noexcept;
    T* fetch_sub(
        ptrdiff_t,memory_order = memory_order_seq_cst) noexcept;

    T* operator++() volatile noexcept;
    T* operator++() noexcept;
    T* operator++(int) volatile noexcept;
    T* operator++(int) noexcept;
    T* operator--() volatile noexcept;
    T* operator--() noexcept;
    T* operator--(int) volatile noexcept;
    T* operator--(int) noexcept;

    T* operator+=(ptrdiff_t) volatile noexcept;
    T* operator+=(ptrdiff_t) noexcept;
    T* operator-=(ptrdiff_t) volatile noexcept;
    T* operator-=(ptrdiff_t) noexcept;
};
```

```
bool atomic_is_lock_free(volatile const atomic<T*>*) noexcept;
bool atomic_is_lock_free(const atomic<T*>*) noexcept;
void atomic_init(volatile atomic<T*>*, T*) noexcept;
void atomic_init(atomic<T*>*, T*) noexcept;
T* atomic_exchange(volatile atomic<T*>*, T*) noexcept;
T* atomic_exchange(atomic<T*>*, T*) noexcept;
T* atomic_exchange_explicit(volatile atomic<T*>*, T*, memory_order)
    noexcept;
T* atomic_exchange_explicit(atomic<T*>*, T*, memory_order) noexcept;
void atomic_store(volatile atomic<T*>*, T*) noexcept;
void atomic_store(atomic<T*>*, T*) noexcept;
void atomic_store_explicit(volatile atomic<T*>*, T*, memory_order)
    noexcept;
void atomic_store_explicit(atomic<T*>*, T*, memory_order) noexcept;
T* atomic_load(volatile const atomic<T*>*) noexcept;
T* atomic_load(const atomic<T*>*) noexcept;
T* atomic_load_explicit(volatile const atomic<T*>*, memory_order) noexcept;
T* atomic_load_explicit(const atomic<T*>*, memory_order) noexcept;
bool atomic_compare_exchange_strong(
    volatile atomic<T*>*,T* * old_value,T* new_value) noexcept;
bool atomic_compare_exchange_strong(
    volatile atomic<T*>*,T* * old_value,T* new_value) noexcept;
bool atomic_compare_exchange_strong_explicit(
    atomic<T*>*,T* * old_value,T* new_value,
    memory_order success_order,memory_order failure_order) noexcept;
bool atomic_compare_exchange_strong_explicit(
    atomic<T*>*,T* * old_value,T* new_value,
    memory_order success_order,memory_order failure_order) noexcept;
bool atomic_compare_exchange_weak(
    volatile atomic<T*>*,T* * old_value,T* new_value) noexcept;
bool atomic_compare_exchange_weak(
    atomic<T*>*,T* * old_value,T* new_value) noexcept;
bool atomic_compare_exchange_weak_explicit(
    volatile atomic<T*>*,T* * old_value, T* new_value,
    memory_order success_order,memory_order failure_order) noexcept;
bool atomic_compare_exchange_weak_explicit(
    atomic<T*>*,T* * old_value, T* new_value,
    memory_order success_order,memory_order failure_order) noexcept;

T* atomic_fetch_add(volatile atomic<T*>*, ptrdiff_t) noexcept;
T* atomic_fetch_add(atomic<T*>*, ptrdiff_t) noexcept;
T* atomic_fetch_add_explicit(
    volatile atomic<T*>*, ptrdiff_t, memory_order) noexcept;
T* atomic_fetch_add_explicit(
    atomic<T*>*, ptrdiff_t, memory_order) noexcept;
T* atomic_fetch_sub(volatile atomic<T*>*, ptrdiff_t) noexcept;
T* atomic_fetch_sub(atomic<T*>*, ptrdiff_t) noexcept;
T* atomic_fetch_sub_explicit(
    volatile atomic<T*>*, ptrdiff_t, memory_order) noexcept;
T* atomic_fetch_sub_explicit(
    atomic<T*>*, ptrdiff_t, memory_order) noexcept;
```

如果泛化模板所提供的操作与上述函数同名（见附录 D.3.8 小节），那么它们具有相同的语义。

1. std::atomic<T*>::fetch_add()成员函数

载入 atomic<T*>实例的值，按标准指针运算法则与给定的 i 值相加，以所得之和替换实例中的旧值，并返回旧值，这些操作以原子化方式一并执行。

声明：

```
T* fetch_add(
    ptrdiff_ti,memory_order order = memory_order_seq_cst)
    volatile noexcept;
T* fetch_add(
    ptrdiff_ti,memory_order order = memory_order_seq_cst) noexcept;
```

作用：

获取*this 对象中存有的值，并与给定的 i 值相加，将所得之和存储到*this 对象中，这些操作以原子化方式一并执行。

返回：

返回*this 对象在存储动作发生前一刻的值。

抛出：

无。

注记：

这是针对*this 对象所在的内存区域的原子化“读-改-写”操作。

2. std::atomic_fetch_add()非成员函数

读取 atomic<T*>实例的值，按标准指针运算法则与给定的 i 值相加，以所得之和替换实例中的旧值，这些操作以原子化方式一并执行。

声明：

```
T* atomic_fetch_add(volatile atomic<T*>* p, ptrdiff_ti) noexcept;
T* atomic_fetch_add(atomic<T*>* p, ptrdiff_ti) noexcept;
```

作用：

```
return p->fetch_add(i);
```

3. std::atomic_fetch_add_explicit()非成员函数

读取 atomic<T*>实例的值，按标准指针运算法则与给定的 i 值相加，以所得之和替换实例中的旧值，这些操作以原子化方式一并执行。

声明：

```
T* atomic_fetch_add_explicit(
    volatile atomic<T*>* p, ptrdiff_ti,memory_order order) noexcept;
T* atomic_fetch_add_explicit(
    atomic<T*>* p, ptrdiff_ti, memory_order order) noexcept;
```

作用：

```
return p->fetch_add(i,order);
```

4. std::atomic<T*>::fetch_sub()成员函数

载入 atomic<T*>实例的值，按标准指针运算法则与给定的 i 值相减，以所得之差替换实例中的旧值，并返回旧值，这些操作以原子化方式一并执行。

声明：

```
T* fetch_sub(
    ptrdiff_ti,memory_order order = memory_order_seq_cst)
    volatile noexcept;
T* fetch_sub(
    ptrdiff_ti,memory_order order = memory_order_seq_cst) noexcept;
```

作用：

获取*this 对象中存有的值，并与给定的 i 值相减，将所得之差存储到*this 对象中，这些操作以原子化方式一并执行。

返回：

返回*this 对象在存储动作发生前一刻的值。

抛出：

无。

注记：

这是针对*this 对象所在的内存区域的原子化“读-改-写”操作。

5. std::atomic_fetch_sub()非成员函数

读取 atomic<T*>实例的值，按标准指针运算法则与给定的 i 值相减，以所得之差替换实例中的旧值，这些操作以原子化方式一并执行。

声明：

```
T* atomic_fetch_sub(volatile atomic<T*>* p, ptrdiff_ti) noexcept;
T* atomic_fetch_sub(atomic<T*>* p, ptrdiff_ti) noexcept;
```

作用：

```
return p->fetch_sub(i);
```

6. std::atomic_fetch_sub_explicit()非成员函数

读取 atomic<T*>实例的值，按标准指针运算法则与给定的 i 值相减，将所得之差替换实例中的旧值，这些操作以原子化方式一并执行。

声明：

```
T* atomic_fetch_sub_explicit(
```

```
    volatile atomic<T*>* p, ptrdiff_ti,memory_order order) noexcept;
T* atomic_fetch_sub_explicit(
    atomic<T*>* p, ptrdiff_ti, memory_order order) noexcept;
```

作用：

```
return p->fetch_sub(i,order);
```

7. std::atomic<T*>::operator++ 前缀自增运算符

令*this 对象中存有的值按标准指针运算法则自增，并返回新值，这些操作以原子化方式一并执行。

声明：

```
T* operator++() volatile noexcept;
T* operator++() noexcept;
```

作用：

```
return this->fetch_add(1) + 1;
```

8. std::atomic<T*>::operator++ 后缀自增运算符

令*this 对象中存有的值按标准指针运算法则自增，并返回旧值，这些操作以原子化方式一并执行。

声明。

```
T* operator++(int) volatile noexcept;
T* operator++(int) noexcept;
```

作用：

```
return this->fetch_add(1);
```

9. std::atomic<T*>::operator-- 前缀自减运算符

令*this 对象中存有的值按标准指针运算法则自减，并返回新值，这些操作以原子化方式一并执行。

声明：

```
T* operator--() volatile noexcept;
T* operator--() noexcept;
```

作用：

```
return this->fetch_sub(1) - 1;
```

10. std::atomic<T*>::operator-- 后缀自减运算符

令*this 对象中存有的值按标准指针运算法则自减，并返回旧值，这些操作以原子

化方式一并执行。

声明：

```
T* operator--(int) volatile noexcept;
T* operator--(int) noexcept;
```

作用：

```
return this->fetch_sub(1);
```

11. std::atomic<T*>::operator+= 复合赋值操作符

使*this 对象中存有的值与给定的值按标准指针运算法则相加，将所得之和存储到*this 对象中，并返回新值，全部操作以原子化方式一并执行。

声明：

```
T* operator+=(ptrdiff_ti) volatile noexcept;
T* operator+=(ptrdiff_ti) noexcept;
```

作用：

```
return this->fetch_add(i) + i;
```

12. std::atomic<T*>::operator-= 复合赋值操作符

使*this 对象中存有的值与给定的值按标准指针运算法则相减，将所得之差存储到*this 对象中，并返回新值，全部操作以原子化方式一并执行。

声明：

```
T* operator-=(ptrdiff_ti) volatile noexcept;
T* operator-=(ptrdiff_ti) noexcept;
```

作用：

```
return this->fetch_sub(i) - i;
```

D.4 <future>头文件

<future>头文件提供了不少工具，用于处理另一个线程执行的操作所产生的异步结果。

头文件内容：

```
namespace std
{
    enum class future_status {
        ready, timeout, deferred };

    enum class future_errc
    {
```

```
        broken_promise,
        future_already_retrieved,
        promise_already_satisfied,
        no_state
    };

    class future_error;

    const error_category&future_category();
    error_code make_error_code(future_errc e);
    error_condition make_error_condition(future_errc e);

    template<typename ResultType>
    class future;

    template<typename ResultType>
    class shared_future;

    template<typename ResultType>
    class promise;

    template<typename FunctionSignature>
    class packaged_task; // 未给出定义

    template<typename ResultType,typename ... Args>
    class packaged_task<ResultType (Args...)>;

    enum class launch {
        async, deferred
    };

    template<typename FunctionType,typename ... Args>
    future<result_of<FunctionType(Args...)>::type>
    async(FunctionType&&func,Args&& ... args);

    template<typename FunctionType,typename ... Args>
    future<result_of<FunctionType(Args...)>::type>
    async(std::launch policy,FunctionType&& func, Args&& ... args);
}
```

D.4.1 std::future 类模板

std::future 类模板给出了一种方法，以等待另一线程的异步结果，它配合 `std::promise` 和 `std::packaged_task` 类模板，以及 std::async()函数模板一起使用，后面三者用于提供异步结果。给定任意一个异步结果，它在任意时刻都只准许一个 std::future 实例指涉。

std::future 的实例满足 MoveConstructible 和 MoveAssignable 的型别要求，但不满足

CopyAssignable 或 CopyConstructible[①]的型别要求。

类定义：

```
template<typename ResultType>
class future
{
public:
    future() noexcept;
    future(future&&) noexcept;
    future& operator=(future&&) noexcept;
    ~future();

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

    shared_future<ResultType>share();

    bool valid() const noexcept;

    /*返回值见后文详述*/ get();

    void wait();

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

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

1. std::future()默认构造函数

构造一个 std::future 对象，它尚未关联异步结果。

声明：

```
future() noexcept;
```

作用：

构造一个新的 std::future 实例。

后置条件：

valid()返回 false。

抛出：

无。

① 译者注：这些术语都是针对型别的具名要求，即 named requirements，属于 C++20 的 concept 新特性。

2. std::future()移动构造函数

依照某 std::future 对象构造另一个新实例，并将前者所关联的异步结果移交给后者。

声明：

```
future(future&& other) noexcept;
```

作用：

依照参数 other 构造一个新的 std::future 对象。

后置条件：

如果 other 对象已经与某异步结果关联，那么在构造函数发生调用后，它会转而与新构造的 std::future 对象关联。对象 other 不再关联异步结果。如果在这个构造函数调用前后分别执行 this->valid()和 other.valid()，那么它们的返回值相同。other.valid()返回 false。

抛出：

无。

3. std::future()移动赋值操作符

将异步结果的归属权从一个 std::future 对象转移给另一个。

声明：

```
future(future&& other) noexcept;
```

作用：

在两个 std::future 实例之间转移异步状态的归属权。

后置条件：

如果 other 对象已经与某异步结果关联，那么在构造函数发生调用后，它会转而与*this 对象关联。对象 other 不再关联异步结果。如果*this 对象在本函数调用发生前，已经关联了某异步状态，其归属权就会被释放。如果该状态的最后一份指涉正是来自*this 对象，则连带销毁该状态。如果在这个构造函数调用前后分别执行 this->valid()和 other.valid()，那么它们的返回值相同。other.valid()返回 false。

抛出：

无。

4. std::future()析构函数

销毁 std::future 对象。

声明：

```
~future();
```

作用：

销毁*this 对象。如果*this 对象关联了异步结果，且这个实例是该结果的最后一份指涉，则连带销毁该异步结果。

抛出：

无。

5. std::future::share()成员函数

构造一个新的 std::shared_future 实例，并将*this 对象关联的异步结果的归属权转移给新的 std::shared_future 实例。

声明：

```
shared_future<ResultType> share();
```

作用：

```
shared_future<ResultType>(std::move(*this));
```

后置条件：

如果*this 对象与某异步结果关联，那么在 share()调用发生后，它会转而与*this 对象关联。`this->valid()`返回 false。

抛出：

无。

6. std::future::valid()成员函数

判定 std::future 实例是否与某异步结果关联。

声明：

```
bool valid() const noexcept;
```

返回：

如果*this 对象关联了异步结果就返回 true，否则返回 false。

抛出：

无。

7. std::future::wait()成员函数

如果*this 对象关联的状态已经含有一个延后执行的函数，则立即调用该函数。否则，等待 std::future 实例所关联的异步结果进入就绪状态。

声明：

```
void wait();
```

前置条件：

`this->valid()`会返回 true。

作用：

如果关联的状态已经含有一个延后执行的函数，则立即调用该函数，并将返回值存储到异步结果中，如果有异常抛出，则将异常存储到异步结果中。否则，使调用本函数

的线程阻塞，直到*this 对象关联的异步结果准备就绪。

抛出：

无。

8. std::future::wait_for()成员函数

等待 std::future 实例所关联的异步结果进入就绪状态，或一直等待，直到超出限定的时长。

声明：

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

前置条件：

`this->valid()`会返回 true。

作用：

如果*this 对象关联的异步结果已经含有一个延后执行的函数，该函数由 std::async()的调用安排执行，但尚未开始，则马上返回且不引发阻塞。否则引发阻塞，直到与*this 对象关联的异步结果准备就绪，或一直等待，直到超出 relative_time 限定的时长。

返回：

如果*this 对象关联的异步结果已经含有一个延后执行的函数，该函数由 std::async()的调用安排执行，但尚未开始，就返回 `std::future_status::deferred`；如果与*this 对象关联的异步结果已经就绪，就返回 `std::future_status::ready`；如果等待时长超出了 relative_time 所限定的时长，就返回 `std::future_status::timeout`。

注记：

线程阻塞的时长可能超过指定的时长。只要有可能，就应当由恒稳时钟判定已消耗的等待时间。

抛出：

无。

9. std::future::wait_until()成员函数

等待 std::future 实例所关联的异步结果进入就绪状态，或一直等待，直到超出限定的时长。

声明：

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

前置条件：

this->valid()会返回 true。

作用：

如果*this 对象关联的异步结果已经含有一个延后执行的函数，该函数由 std::async()的调用安排执行，但尚未开始，则马上返回且不引发阻塞。否则引发阻塞，直到与*this 对象关联的异步结果准备就绪，或直到 Clock::now()返回的时刻晚于/等于参数 absolute_time 所指定的时刻。

返回：

如果*this 对象关联的异步结果已经含有一个延后执行的函数，该函数由 std::async()的调用安排执行，但尚未开始，就返回 `std::future_status::deferred`；如果与*this 对象关联的异步结果已经就绪，就返回 `std::future_status::ready`；如果 Clock::now()返回的时刻晚于/等于参数 absolute_time 所指定的时刻，就返回 `std::future_status::timeout`。

注记：

线程阻塞的实际时间并不确定，只有函数返回 std::future_status::timeout，且 Clock::now()返回的时刻晚于/等于参数 absolute_time 所指定的时刻，线程才会脱离阻塞。

抛出：

无。

10. std::future::get()成员函数

如果关联的状态已经含有一个延后执行的函数，该函数由 std::async()的调用安排执行，则立即调用该函数并返回结果。否则等待，直到 std::future 实例关联的异步结果准备就绪，并将返回值存储到异步结果中。如果有异常抛出，则将异常存储到异步结果中。

声明：

```
void future<void>::get();
R& future<R&>::get();
R future<R>::get();
```

前置条件：

`this->valid()`会返回 `true`。

作用：

如果关联的状态已经含有一个延后执行的函数，就立即调用该函数并返回结果。如果有异常抛出，则向外传播。否则引发阻塞，直到与*this 对象关联的异步结果准备就绪。如果异步结果中有异常，就重新抛出该异常，否则返回异步结果中所存储的值。

返回：

如果关联状态含有一个延后执行的函数，则返回该函数调用的结果。如果参数

ResultType 为 void，则该调用正常返回。如果参数 ResultType 是某个型别 R 的引用，即 R&，就返回关联状态所存储的引用，否则返回所存储的值。

抛出：

如果延后执行的函数抛出异常，则本函数重新抛出该异常，或将它存储到异步结果中。

后置条件：

```
this->valid()==false
```

D.4.2 std::shared_future 类模板

std::shared_future 类模板给出了一种方法以等待其他线程的异步结果，它配合 std::promise 和 std::packaged_task 类模板，以及 std::async()函数模板一起使用，后面三者用于提供异步结果。多个 std::shared_future 实例可以共同指涉一个异步结果。

std::shared_future 的实例满足 MoveConstructible 和 MoveAssignable[①]的型别要求。

我们也可以把某个 std::future 实例作为来源，按移动方式构造另一个 std::shared_future 实例，两者具有相同的模板参数 ResultType。

给定一个 std::shared_future 实例，如果其上同时发生多个访问，它们之间不会自动同步。因此，如果多个线程访问同一个 std::shared_future 实例，却未采取外部同步措施，这不是安全行为。但是，对 std::shared_future 所关联的状态的多个访问会彼此同步，因而，如果 std::shared_future 的多个实例都与同一个状态关联，那么多个线程同时独立访问这些实例是安全行为。

类定义：

```
template<typename ResultType>
class shared_future
{
public:
    shared_future() noexcept;
    shared_future(future<ResultType>&&) noexcept;

    shared_future(shared_future&&) noexcept;
    shared_future(shared_future const&);
    shared_future& operator=(shared_future const&);
    shared_future& operator=(shared_future&&) noexcept;
    ~shared_future();

    bool valid() const noexcept;

    /*返回值见后文详述*/ get() const;
```

① 译者注：这两个术语都是针对型别的具名要求，即 named requirements，属于 C++20 的 concept 新特性。

```
    void wait() const;

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

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

1. std::shared_future()默认构造函数

构造一个 std::shared_future 对象，它尚未关联异步结果。

声明：

```
shared_future() noexcept;
```

作用：

构造一个新的 std::shared_future 实例。

后置条件：

对于新构建的实例，valid()的结果是 false。

抛出：

无。

2. std::shared_future()移动构造函数

依照某 std::shared_future 对象构造一个新对象，并将前者所关联的异步结果移交给后者。

声明：

```
shared_future(shared_future&& other) noexcept;
```

作用：

构造一个新的 std::shared_future 实例。

后置条件：

如果 other 对象已经与某异步结果关联，那么在构造函数发生调用后，它会转而与新构建的 std::shared_future 对象关联。对象 other 不再关联异步结果。

抛出：

无。

3. std::shared_future()移动构造函数，它以 std::future 作为来源

依照某 std::future 对象构造一个新的 `std::shared_future` 对象，并将前者所关联的异步结果移交给后者。

声明：

```
shared_future(std::future<ResultType>&& other) noexcept;
```

作用：

构造一个新的 std::shared_future 对象。

后置条件：

如果 other 对象已经与某异步结果关联，那么在构造函数发生调用后，它会转而与新构建的 std::shared_future 对象关联。对象 other 不再关联异步结果。

抛出：

无。

4. std::shared_future()拷贝构造函数

依照某 std::shared_future 对象构造另一个新对象，如果源对象已经关联了异步结果，则它会被源对象和副本共同指涉。

声明：

```
shared_future(shared_future const& other);
```

作用：

构造一个新的 std::shared_future 实例。

后置条件：

如果 other 对象已经与某异步结果关联，那么在构造函数发生调用后，它会同时关联 other 对象和新构建的 std::shared_future 对象。

抛出：

无。

5. std::shared_future()析构函数

销毁 std::shared_future 对象。

声明：

```
~shared_future();
```

作用：

销毁*this 对象。如果*this 对象关联了异步结果，而这个 std::shared_future 实例正是该结果的最后一份指涉，且再没有 `std::promise` 和 `std::packaged_task` 与这个结果关联，那么连带销毁此异步结果。

抛出：

无。

6. std::shared_future::valid()成员函数

判定 std::shared_future 实例是否关联了某异步结果。

声明：

```
bool valid() const noexcept;
```

返回：

如果*this 具有关联的异步结果，就返回 true，否则返回 false。

抛出：

无。

7. std::shared_future::wait()成员函数

如果*this 对象关联的状态已经含有一个延后执行的函数，则立即调用该函数。否则，等待 std::shared_future 实例所关联的异步结果进入就绪状态。

声明：

```
void wait() const;
```

前置条件：

this->valid()会返回 true。

作用：

如果多个 std::shared_future 实例共享相同的关联状态，那么，在多个线程上对这些实例调用 get()和 wait()，这些调用会按串行方式逐一执行。如果关联状态含有一个延后执行的函数，那么 get()或 wait()的第一次调用会立即运行该函数，并将其返回值存储到异步结果中。如果有异常抛出，则将异常存储到异步结果中。否则引发阻塞，直到与*this 对象关联的异步结果准备就绪。

抛出：

无。

8. std::shared_future::wait_for()成员函数

等待 std::shared_future 实例所关联的异步结果进入就绪状态，或一直等待，直到超出限定的时长。

声明：

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

前置条件：

this->valid()会返回 true。

作用：

如果*this 对象关联的异步结果已经含有一个延后执行的函数，该函数由 std::async()的调用安排执行，但尚未开始，就马上返回且不引发阻塞。否则引发阻塞，直到与*this 对象关联的异步结果准备就绪，或一直等待，直到超出 relative_time 限定的时长。

返回：

如果*this 对象关联的异步结果已经含有一个延后执行的函数，该函数由 std::async()的调用安排执行，但尚未开始，就返回 `std::future_status::deferred`；如果与*this 对象关联的异步结果已经就绪，就返回 `std::future_status::ready`；如果等待时长超出了 relative_time 所限定的时长，就返回 `std::future_status::timeout`。

注记：

线程阻塞的时长可能超过指定的时长。只要有可能，就应当由恒稳时钟判定已消耗的等待时间。

抛出：

无。

9. std::shared_future::wait_until()成员函数

等待 std::shared_future 实例所关联的异步结果进入就绪状态，或一直等待，直到超出限定的时长。

声明：

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

前置条件：

this->valid()会返回 true。

作用：

如果*this 对象关联的异步结果已经含有一个延后执行的函数，该函数由 std::async()的调用安排执行，但尚未开始，就马上返回且不引发阻塞。否则引发阻塞，直到与*this 对象关联的异步结果准备就绪，或直到 Clock::now()返回的时刻晚于/等于参数 absolute_time 所指定的时刻。

返回：

如果*this 对象关联的异步结果已经含有一个延后执行的函数，该函数由 std::async()的调用安排执行，但尚未开始，就返回 `std::future_status::deferred`；如果与*this 对象关联的异步结果已经就绪，就返回 `std::future_status::ready`；如果 Clock::now()返回的时刻晚于/等于参数 absolute_time 的值，就返回 `std::future_`

status::timeout。

注记：

线程阻塞的实际时间并不确定，只有函数返回 std::future_status::timeout，且 Clock::now()返回的时刻晚于/等于 absolute_time 参数的值，线程才会脱离阻塞。

抛出：

无。

10. std::shared_future::get()成员函数

如果关联的状态已经含有一个延后执行的函数，该函数由 std::async()的调用安排执行，就立即调用该函数并返回结果。否则等待，直到 std::shared_future 实例关联的异步结果准备就绪，并将返回值存储到异步结果中。如果有异常抛出，则将异常存储到异步结果中。

声明：

```
void shared_future<void>::get() const;
R& shared_future<R&>::get() const;
R const& shared_future<R>::get() const;
```

前置条件：

this->valid()会返回 true。

作用：

如果多个 std::shared_future 实例共享相同的关联状态，那么，在多个线程上对这些实例调用 get()和 wait()，这些调用会按串行方式逐一执行。如果关联状态含有一个延后执行的函数，那么 get()或 wait()的第一次调用会立即运行该函数，并将其返回值存储到异步结果中。如果有异常抛出，则将异常存储到异步结果中。本函数调用会引发阻塞，直到与*this 对象关联的异步结果准备就绪。如果异步结果中有异常，就重新抛出该异常，否则返回异步结果中所存储的值。

返回：

如果参数 ResultType 为 void，则该调用正常返回。如果参数 ResultType 是某个型别 R 的引用，即 R&，就返回关联状态所存储的引用，否则返回所存储的值的 const 引用。

抛出：

如果异步结果有任何异常，则重新抛出该异常。

D.4.3 std::packaged_task 类模板

std::packaged_task 类模板的功能是包装函数或其他可调用对象。一旦通过 std:: packaged_task 实例调用该函数（或可调用对象），执行结果便会存储到异步结果中，以便交由 std::future 的实例获取。

std::packaged_task 的实例满足 MoveConstructible 和 MoveAssignable 的型别要求，但不满足 CopyAssignable 或 CopyConstructible[①]的型别要求。

类定义：

```
template<typename FunctionType>
class packaged_task; // 未定义

template<typename ResultType,typename... ArgTypes>
class packaged_task<ResultType(ArgTypes...)>
{
public:

    packaged_task() noexcept;
    packaged_task(packaged_task&&) noexcept;
    ~packaged_task();

    packaged_task& operator=(packaged_task&&) noexcept;

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

    void swap(packaged_task&) noexcept;

    template<typename Callable>
    explicit packaged_task(Callable&& func);

    template<typename Callable,typename Allocator>
    packaged_task(std::allocator_arg_t, const Allocator&, Callable&&);

    bool valid() const noexcept;
    std::future<ResultType>get_future();
    void operator()(ArgTypes...);
    void make_ready_at_thread_exit(ArgTypes...);
    void reset();
};
```

1. std::packaged_task()默认构造函数

构造一个 `std::packaged_task` 对象。

声明：

```
packaged_task() noexcept;
```

作用：

构造一个 `std::packaged_task` 对象，它尚未关联任务或异步结果。

抛出：

无。

① 译者注：这些术语都是针对型别的具名要求，即 named requirements，属于 C++20 的 concept 新特性。

2. std::packaged_task()构造函数，依据可调用对象进行构造

以可调用对象为蓝本的 std::packaged_task 构造函数，构造一个 std::packaged_task 对象，它关联了一个任务及其异步结果。

声明：

```
template<typename Callable>
packaged_task(Callable&& func);
```

前置条件：

表达式 The expression func(args...)需合法，其中 args...的各个元素 args-i 应为 ArgTypes...中的对应型别 ArgTypes-i。（译者注：“-i”表示第 i 个元素）返回值应能够转化为型别 ResultType。

作用：

构造一个 std::packaged_task 实例，它与型别为 ResultType 的异步结果关联，但结果尚未就绪。该实例还与一个 Callable 型别的任务关联，任务是参数 func 的副本。

抛出：

如果构造函数无法为异步结果分配内存,则抛出型别为 std::bad_alloc 的异常。如果 Callable 副本或其移动构造函数抛出异常，则本构造函数重新抛出该异常。

3. 采用内存配置器的 std::packaged_task()构造函数

采用内存配置器的 std::packaged_task 构造函数构造一个 std::packaged_task 对象，其中含有关联的任务及其异步结果，它们的内存由给定的配置器分配。

声明：

```
template<typename Allocator,typename Callable>
packaged_task(
    std::allocator_arg_t, Allocator const& alloc,Callable&& func);
```

前置条件：

表达式 The expression func(args...)需合法，其中 args...的各个元素 args-i 应为 ArgTypes...中的对应型别 ArgTypes-i（提示：“-i”表示第 i 个元素）。返回值应能够转化为型别 ResultType。

作用：

构造一个 std::packaged_task 实例，它与型别为 ResultType 的异步结果关联，但结果尚未就绪。该实例还与一个 Callable 型别的任务关联，任务是参数 func 的副本。关联的异步结果和任务所需的内存通过配置器或其副本分配。

抛出：

配置器试图为异步结果和任务分配内存，如果这一过程抛出了异常，则本构造函数重新抛出该异常。如果 Callable 副本或其移动构造函数抛出异常，则本构造函数重新抛出该异常。

4. std::packaged_task()移动构造函数

依照某 std::packaged_task 对象构造一个新对象，并将前者所关联的异步结果和任务移交给后者。

声明：

```
packaged_task(packaged_task&& other) noexcept;
```

作用：

构造一个 std::packaged_task 实例。

后置条件：

如果 other 对象已经与某异步结果和任务关联，那么在构造函数发生调用后，它们会转而与新构建的 std::packaged_task 对象关联。

对象 other 不再关联异步结果。

抛出：

无。

5. std::packaged_task()移动赋值操作符

将关联的异步结果的归属权从一个 std::packaged_task 对象转移给另一个。

声明：

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

作用：

将 other 对象关联的异步结果和任务的归属权转移给*this 对象，原有的异步结果全被丢弃，相当于 std::packaged_task(other).swap(*this)。

后置条件：

如果 other 对象已经与某异步结果和任务关联，那么在执行移动赋值操作之后，它们会转而与*this 对象关联。对象 other 不再关联异步结果。

返回：

*this。

抛出：

无。

6. std::packaged_task::swap()成员函数

在两个 std::packaged_task 对象间交换它们关联的异步结果。

声明：

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

作用：

在 other 对象和*this 对象间交换它们关联的异步结果。

后置条件：

如果 other 对象已经与某异步结果和任务关联，那么在执行 swap()之后，它们会转而与*this 对象关联。如果*this 对象已经与某异步结果和任务关联，那么在执行 swap()之后，它们会转而与 other 对象关联。

抛出：

无。

7. std::packaged_task()析构函数

销毁 std::packaged_task 对象。

声明：

```
~packaged_task();
```

作用：

销毁*this 对象。如果*this 对象具有关联的异步结果，且该结果中并未存有任务或异常，那么异步结果进入就绪状态，并成为一个 std::future_error 型别的异常，其值是错误代码 std::future_errc::broken_promise。

抛出：

无。

8. std::packaged_task::get_future()成员函数

获取一个 std::future 实例，通过它即能得到与*this 对象关联的异步结果。

声明：

```
std::future<ResultType> get_future();
```

前置条件：

*this 对象具有关联的异步结果。

返回：

返回一个 std::future 实例，通过它即能取得与*this 对象关联的异步结果。

抛出：

如果之前已经通过调用 get_future()获取过 std::future 以进一步得到异步结果，就抛出 std::future_error 型别的异常，其值是错误代码 std::future_errc::future_already_ retrieved。

9. std::packaged_task::reset()成员函数

令 std::packaged_task 实例与新的异步结果关联，该结果由实例所存储的同一个任务产生。

声明：

```
void reset();
```

前置条件：

*this 对象具有关联的异步任务。

作用：

相当于*this=packaged_task(std::move(f))，其中变量 f 是*this 对象存储的关联任务。

抛出：

如果无法为新的异步结果分配内存，则抛出型别为 std::bad_alloc 的异常。

10. std::packaged_task::valid()成员函数

判定*this 对象是否具有关联任务和异步结果。

声明：

```
bool valid() const noexcept;
```

返回：

如果*this 对象具有关联任务和异步结果，就返回 true，否则返回 false。

抛出：

无。

11. std::packaged_task::operator()函数调用操作符

调用 std::packaged_task 实例的关联任务，并将其返回值或抛出的异常存储到关联的异步结果中。

声明：

```
void operator()(ArgTypes... args);
```

前置条件：

*this 对象具有关联任务。

作用：

调用关联的任务 func，相当于 INVOKE(func,args...)。如果调用正常返回，则将其返回值存储到*this 对象关联的异步结果中。如果调用因发生异常而退出，则将该异常存储到*this 对象关联的异步结果中。

后置条件：

*this 对象关联的异步结果已经就绪，其中存储了某个值或异常。本函数调用会令这些阻塞自动清除。

抛出：

如果异步结果已经存有值或异常，就抛出 std::future_error 型别的异常，其值是错误代码 std::future_errc::promise_already_satisfied。

同步：

如果成功执行函数调用操作符，它会与 std::future<ResultType>::get()或 std::shared_future<ResultType>::get()的调用同步，后面两个调用会获取异步结果中存储的值或异常。

12. std::packaged_task::make_ready_at_thread_exit()成员函数

调用 std::packaged_task 实例的关联任务，并将其返回值或抛出的异常存储到关联的异步结果中，但在线程退出之际才使该结果进入就绪状态。

声明：

```
void make_ready_at_thread_exit(ArgTypes... args);
```

前置条件：

*this 对象具有关联任务。

作用：

调用关联的任务 func，相当于 INVOKE(func,args...)。如果调用正常返回，则将其返回值存储到*this 对象关联的异步结果中。如果调用因发生异常而退出，则将该异常存储到*this 对象关联的异步结果中。做出适当调整，当前线程一旦退出，便使关联的异步状态进入就绪状态。

后置条件：

*this 对象关联的异步结果将存储某个值或异常，但在当前线程退出之际，该结果才会进入就绪状态。如果有线程因等待异步结果而发生阻塞，那么它们会在当前线程退出之际脱离阻塞。

抛出：

如果异步结果已经存有值或异常，就抛出 std::future_error 型别的异常，其值是错误代码 std::future_errc::promise_already_satisfied。

如果*this 对象并不具有关联的异步状态，就抛出 std::future_error 型别的异常，其值是错误代码 std::future_errc::no_state。

同步：

如果某线程成功调用 make_ready_at_thread_exit()，该线程的完结动作与 std::future <ResultType>::get()或 std::shared_future<ResultType>::get()的调用同步，后面两个调用会获取异步结果中存储的值或异常。

D.4.4 std::promise 类模板

std::promise 类模板给出了一种等待异步结果的方法，该结果可以在另一线程上通过 std::future 的实例获取。

模板参数 ResultType 即为异步结果中存储的值的型别。

调用 std::promise 的特定实例的成员函数 get_future()，即可得到一个 std::future 对象，它关联了该实例的异步结果。成员函数 set_value()将异步结果设成某个值，型别为 ResultType，或者，成员函数 set_exception()将异步结果设为异常。

std::promise 的实例满足 MoveConstructible 和 MoveAssignable 的型别要求，但不满足 CopyAssignable 或 CopyConstructible 的型别要求[①]。

类定义：

```
template<typename ResultType>
class promise
{
public:
    promise();
    promise(promise&&) noexcept;
    ~promise();
    promise& operator=(promise&&) noexcept;

    template<typename Allocator>
    promise(std::allocator_arg_t, Allocator const&);

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

    void swap(promise& ) noexcept;

    std::future<ResultType>get_future();

    void set_value(/*参数见后文详述*/);
    void set_exception(std::exception_ptr p);
};
```

1. std::promise()默认构造函数

构造一个 std::promise 对象。

声明：

```
promise();
```

① 译者注：这些术语都是针对型别的具名要求，即 named requirements，属于 C++20 的 concept 新特性。

作用：

构造一个 std::promise 对象，它关联了一个尚未就绪的异步结果，该结果的型别是 ResultType。

抛出：

如果构造函数无法为异步结果分配内存，则抛出型别为 std::bad_alloc 的异常。

2. 采用内存配置器的 std::promise()构造函数

构造一个 std::packaged_task 对象，所关联的异步结果由给定的配置器分配。

声明：

```
template<typename Allocator>
promise(std::allocator_arg_t, Allocator const& alloc);
```

作用：

构造一个 std::promise 对象，它关联了一个尚未就绪的异步结果，该结果的型别是 ResultType。这个对象关联的异步结果由给定的配置器分配。

抛出：

配置器试图为异步结果分配内存，如果这一过程抛出了异常，则本构造函数重新抛出该异常。

3. std::promise()移动构造函数

依照某 std::promise 对象构造一个新对象，并将前者所关联的异步结果移交给后者。

声明：

```
promise(promise&& other) noexcept;
```

作用：

构造一个新的 std::promise 对象。

后置条件：

如果 other 对象已经与某异步结果关联，那么在构造函数发生调用后，它会转而与新构建的 std::promise 对象关联。对象 other 不再关联异步结果。

抛出：

无。

4. std::promise()移动赋值操作符

将关联的异步结果的归属权从一个 std::promise 对象转移给另一个。

声明：

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

作用：

将 other 对象关联的异步结果的归属权转移给*this 对象，如果*this 对象已经具有关联的异步结果，则该结果会进入就绪状态，并存储一个 std::future_error 型别的异常，其值是错误代码 std::future_errc::broken_promise。

后置条件：

如果 other 对象已经与某异步结果关联，那么在执行移动赋值操作之后，它会转而与新构建的 std:promise 对象关联（即*this 对象）。对象 other 不再关联异步结果。

返回：

*this。

抛出：

无。

5. std::promise::swap()成员函数

在两个 std::promise 对象之间交换关联的异步结果的归属权。

声明：

```
void swap(promise& other);
```

作用：

在 other 对象和*this 对象间交换它们关联的异步结果。

后置条件：

如果 other 对象已经与某异步结果关联，那么在执行 swap()之后，它会转而与*this 对象关联。

如果*this 对象已经与某异步结果关联，那么在执行 swap()之后，它会转而与 other 对象关联。

抛出：

无。

6. std::promise()析构函数

销毁 std::promise 对象。

声明：

```
~promise();
```

作用：

销毁*this 对象。如果*this 对象具有关联的异步结果，且该结果中并未存有任务或异常，那么异步结果进入就绪状态，并成为一个 std::future_error 型别的异常，其值是错误代码 std::future_errc::broken_promise。

抛出：

无。

7. std::promise::get_future()成员函数

获取一个 std::future 实例，通过它即能得到与*this 对象关联的异步结果。

声明：

```
std::future<ResultType> get_future();
```

前置条件：

*this 对象具有关联的异步结果。

返回：

返回一个 std::future 实例，通过它即能取得与*this 对象关联的异步结果。

抛出：

如果之前已经通过调用 get_future()获取过 std::future 以进一步得到异步结果，就抛出 std::future_error 型别的异常，其值是错误代码 std::future_errc::future_already_retrieved。

8. std::promise::set_value()成员函数

将一个值存储到*this 对象关联的异步结果中。

声明：

```
void promise<void>::set_value();
void promise<R&>::set_value(R& r);
void promise<R>::set_value(R const& r);
void promise<R>::set_value(R&& r);
```

前置条件：

*this 对象具有关联的异步结果。

作用：

如果型别 ResultType 不是 void，则将 r 值存储到*this 对象关联的异步结果中。

后置条件：

*this 对象关联的异步结果进入就绪状态，其中存有一个值。本函数调用会令这些阻塞自动清除。

抛出：

如果异步结果已经存有值或异常，就抛出 std::future_error 型别的异常，其值是错误代码 std::future_errc::promise_already_satisfied。如果参数 r 的拷贝构造函数或移动构造函数抛出异常，则本函数重新抛出该异常。

同步：

若同一个 std::promise 实例上发生多个 set_value()、set_value_at_

thread_exit()、set_exception()和set_exception_at_thread_exit()的并发调用，则它们会按串行方式逐一执行。如果成功调用set_value()，它会与std::future<ResultType>::get()或std::shared_future<ResultType>::get()的调用构成先行关系，后面两个调用会获取异步结果中存储的值。

9. std::promise::set_value_at_thread_exit()成员函数

将一个值存储到*this对象关联的异步结果中，但在当前线程退出之际，该结果才会进入就绪状态。

声明：

```
void promise<void>::set_value_at_thread_exit();
void promise<R&>::set_value_at_thread_exit(R& r);
void promise<R>::set_value_at_thread_exit(R const& r);
void promise<R>::set_value_at_thread_exit(R&& r);
```

前置条件：

*this对象具有关联的异步结果。

作用：

如果型别ResultType不是void，则将r值存储到*this对象关联的异步结果中。标记异步结果，使之具备一妥善存好的值。做出适当调整，当前线程一旦退出，便使关联的异步结果进入就绪状态。

后置条件：

*this对象关联的异步结果将存储某个值，但在当前线程退出之际，该结果才会进入就绪状态。如果有线程因等待异步结果而发生阻塞，那么它们会在当前线程退出之际脱离阻塞。

抛出：

如果异步结果已经存有值或异常，就抛出std::future_error型别的异常，其值是错误代码std::future_errc::promise_already_satisfied。如果参数r的拷贝构造函数或移动构造函数抛出异常，则本函数重新抛出该异常。

同步：

若同一个std::promise实例上发生多个set_value()、set_value_at_thread_exit()、set_exception()和set_exception_at_thread_exit()的并发调用，则它们会按串行方式逐一执行。如果某线程成功调用set_value_at_thread_exit()，则该线程的完结动作与std::future<ResultType>::get()或std::shared_future<ResultType>::get()的调用构成先行关系，后面两个调用会获取异步结果中存储的值。

10. std::promise::set_exception()成员函数

将一个异常存储到*this 对象关联的异步结果中。

声明：

```
void set_exception(std::exception_ptr e);
```

前置条件：

*this 对象具有关联的异步结果，且(bool)e 的值是 true。

作用：

将异常 e 存储到*this 对象关联的异步结果中。

后置条件：

*this 对象关联的异步结果进入就绪状态，其中存有一个异常。如果有线程因等待异步结果而发生阻塞，则它们会在当前线程退出之际脱离阻塞。

抛出：

如果异步结果已经存有值或异常，就抛出 std::future_error 型别的异常，其值是错误代码 std::future_errc::promise_already_satisfied。

同步：

若同一个 std::promise 实例上发生多个 set_value()和 set_exception()的并发调用，则它们会按串行方式逐一执行。如果成功调用 set_exception()，则它会与 std::future<ResultType>::get()或 std::shared_future<ResultType>::get()的调用构成先行关系，后面两个调用会获取异步结果中存储的值。

11. std::promise::set_exception_at_thread_exit()成员函数

将一个异常存储到*this 对象关联的异步结果中，但在当前线程退出之际，该结果才会进入就绪状态。

声明：

```
void set_exception_at_thread_exit(std::exception_ptr e);
```

前置条件：

*this 对象具有关联的异步结果，且(bool)e 的值是 true。

作用：

将异常 e 存储到*this 对象关联的异步结果中。做出适当调整，当前线程一旦退出，便使关联的异步结果进入就绪状态。

后置条件：

*this 对象关联的异步结果将存储某个异常，但在当前线程退出之际，该结果才会进入就绪状态。如果有线程因等待异步结果而发生阻塞，那么它们会在当前线程退出之际脱离阻塞。

抛出：

如果异步结果已经存有值或异常，就抛出 std::future_error 型别的异常，其值是错误代码 std::future_errc::promise_already_satisfied。

同步：

若同一个std::promise实例上发生多个set_value()、set_value_at_thread_exit()、set_exception()和 set_exception_at_thread_exit()的并发调用，则它们会按串行方式逐一执行。如果某线程成功调用 set_exception_at_thread_exit()，则该线程的完结动作与 std::future<ResultType>::get()或 std::shared_future<ResultType>::get()的调用构成先行关系，后面两个调用会获取异步结果中存储的异常。

D.4.5 std::async()函数模板

std::async()是一种运行自含的异步任务的简单函数，以充分利用可供运行的并发硬件资源。std::async()的调用会返回一个含有任务结果的 std::future 对象。根据启动策略，任务或者在自己专属的线程上异步运行，又或者，future 对象的成员函数 wait()或 get()由某线程负责调用，任务也由该线程同步运行。

声明：

```
enum class launch
{
    async,deferred
};

template<typename Callable,typename ... Args>
future<result_of<Callable(Args...)>::type>
async(Callable&&func,Args&& ... args);

template<typename Callable,typename ... Args>
future<result_of<Callable(Args...)>::type>
async(launch policy,Callable&&func,Args&& ... args);
```

前置条件：

对于给出的参数值 func 和 args，表达式 INVOKE(func,args)合法。模板参数 Callable 和 Args 中的每个元素都满足 MoveConstructible 型别的要求。

作用：

在线程内部存储空间构造参数 func 和 args...的副本（后文分别称为 fff 和 xyz...）。

如果启动策略是 std::launch::async，则在自己专属的线程上运行 INVOKE(fff, xyz...)。上述所返回的 std::future 对象在该线程完成之际即准备就绪，并且将持有返回值，或持有任务函数调用抛出的异常。

返回的 std::future 对象指涉某异步状态，而如果有多个 future 关联该状态（译者注：包含 std::future 和 std::shared_future），对于最后关联该状态的 future，在它准备就绪之前，其析构函数会一直阻塞。如果启动策略是 std::launch::deferred，则 fff 和 xyz...将以延后函数的调用形式存储在返回的 std::future 对象中。

返回的 std::future 对象指涉某异步状态，如果有多个 future 共享该关联状态[①]，而它们中的一个最先在某线程上调用成员函数 wait()或 get()，这将令同一个线程同步执行 INVOKE (fff,xyz...)。在该 future 上调用 get()，即可获得执行 INVOKE(fff, xyz...)所返回的值或所抛出的异常。

如果启动策略取值 std::launch::async | std::launch::deferred，或者策略参数被忽略，则代码行为相当于指定了 std::launch::async 策略或 std::launch::deferred 策略。线程库实现将因应逐次调用而选择行为方式，以便充分利用可供运行的并发硬件资源，但不至于发生线程过饱和。

std::async()调用在任何情况下都立即返回。

同步：

std::async()的调用将返回一个 std::future 对象，它指涉某异步状态，如果有 std::future 或 std::shared_future 的实例共同关联该状态，那么任务函数的调用完成以后，实例上的 wait()、get()、wait_for()或 wait_until()调用才会随之成功返回。在采用 std::launch::async 策略的情形中，负责调用任务函数的线程先行完结后，那些调用才会随之成功返回。

抛出：

如果所需的内部存储空间无法分配，则抛出 std::bad_alloc 异常；如果未能达到上述效果，则抛出 std::future_error 型别的异常；如果在构造可调用对象 fff 或参数 xyz 的过程中抛出异常，那么 std::async()函数就会重新抛出该异常。

D.5 <mutex>头文件

<mutex>头文件提供了一些工具，用于确保资源访问的互相排斥：几种互斥型别和锁型别，以及多个函数。它们可以实现一种方法，确保某项操作恰好执行一次。

头文件内容：

```
namespace std
{
    class mutex;
```

[①] 译者注：包含 std::future 和 std::shared_future。

```
    class recursive_mutex;
    class timed_mutex;
    class recursive_timed_mutex;
    class shared_mutex;
    class shared_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 LockableType>
    class shared_lock;

    template<typename ... LockableTypes>
    class scoped_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 类为多线程提供了基本的互斥和同步功能，用途是保护共享数据。这种互斥必须先由 lock()或 try_lock()锁住，之后访问目标数据才会受到保护。互斥锁每次只能由一个线程持有，因此，如果另一个线程试图给互斥上锁，则会因调用 try_lock()而失败，或因调用 lock()而阻塞。一旦线程完成了共享数据的访问，就必须调用 unlock()释放锁，以便其他线程获得锁。

std::mutex 满足 Lockable 的型别要求。

类定义：

```
class mutex
```

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

    constexpr mutex() noexcept;
    ~mutex();

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

1. std::mutex()默认构造函数

构造一个 std::mutex 对象。

声明：

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

作用：

构造一个 std::mutex 实例。

后置条件：

新构造的 std::mutex 实例处于未锁定的初始状态。

抛出：

无。

2. std::mutex()析构函数

销毁 std::mutex 对象。

声明：

```
~mutex();
```

前置条件：

*this 对象必须未被锁定。

作用：

销毁 *this 对象。

抛出：

无。

3. std::mutex::lock()成员函数

令当前线程在 std::mutex 对象上获取锁。

声明：

```
void lock();
```

前置条件：

发起函数调用的线程必须还未在*this 对象上持锁。

作用：

阻塞当前线程，直到取得*this 对象上的锁。

后置条件：

*this 对象由发起函数调用的线程锁住。

抛出：

如果发生任何错误，则抛出型别为 std::system_error 的异常。

4. std::mutex::try_lock()成员函数

令当前线程试图在 std::mutex 对象上获取锁。

声明：

```
bool try_lock();
```

前置条件：

发起函数调用的线程必须还未在*this 对象上持锁。

作用：

令发起函数调用的线程试图以非阻塞方式在*this 对象上获取锁。

返回：

如果发起函数调用的线程获得了锁就返回 true，否则返回 false。

后置条件：

如果函数返回 true，则*this 对象被发起函数调用的线程锁定。

抛出：

无。

注记：

即便其他线程都没有在*this 对象上持锁，本函数依然可能无法获取锁（并返回 false）。

5. std::mutex::unlock()成员函数

令当前线程释放目标 std::mutex 对象持有的锁。

声明：

```
void unlock();
```

前置条件：

发起函数调用的线程必须已经在*this 对象上持锁。

作用：

释放当前线程在*this 对象上持有的锁。如果有线程因为等待锁住*this 对象而阻塞，则让它们当中的一个脱离阻塞。

后置条件：

发起函数调用的线程不再锁住*this 对象。

抛出：

无。

D.5.2 std::recursive_mutex 类

std::recursive_mutex 类（递归互斥）为多线程提供了基础的互斥和同步功能，用途是保护共享数据。递归互斥必须先由 lock()或 try_lock()锁住，目标数据的访问之后才会受到保护。互斥锁每次只能由一个线程持有，因此，如果另一个线程试图给 recursive_mutex 对象上锁，则会因调用 try_lock()而失败，或因调用 lock()而阻塞。一旦线程完成了对共享数据的访问，就必须调用 unlock()释放锁，以便其他线程获得锁。

因为 std::recursive_mutex 具有递归特性，所以即便一个线程已经在该类的某个实例上持锁，它还能再次调用 lock()或 try_lock()，令锁的计数值增加。针对 lock()或 try_lock()的每次成功调用，该获得锁的线程都需要为之逐一执行 unlock()，在全部执行完成以前，此递归互斥实例无法被另一个线程锁住。

std::recursive_mutex 满足 Lockable 的型别要求。

类定义：

```
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;
};
```

1. std::recursive_mutex()默认构造函数

构造一个 std::recursive_mutex 对象。

声明：

```
recursive_mutex() noexcept;
```

作用：

构造一个 std::recursive_mutex 对象。

后置条件：

新构造的 std::recursive_mutex 实例处于未锁定的初始状态。

抛出：

如果无法构造出新的 std::recursive_mutex 实例，则抛出型别为 std::system_error 的异常。

2. std::recursive_mutex()析构函数

销毁 std::recursive_mutex 对象。

声明：

```
~recursive_mutex();
```

前置条件：

*this 对象必须未被锁定。

作用：

销毁*this 对象。

抛出：

无。

3. std::recursive_mutex::lock()成员函数

令当前线程在 std::recursive_mutex 对象上获取锁。

声明：

```
void lock();
```

作用：

阻塞当前线程，直到取得*this 对象上的锁。

后置条件：

*this 对象由发起函数调用的线程锁住。如果该线程在*this 对象上已经持锁，则令这个锁的计数值加 1。

抛出：

如果发生任何错误，则抛出型别为 std::system_error 的异常。

4. std::recursive_mutex::try_lock()成员函数

令当前线程试图在 std::recursive_mutex 对象上获取锁。

声明：

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

作用：

令发起函数调用的线程试图以非阻塞方式在*this 对象上获取锁。

返回：

如果发起函数调用的线程获得了锁就返回 true，否则返回 false。

后置条件：

如果函数返回值为 true，则表明发起函数调用的线程在*this 对象上获得了一个新的锁。

抛出：

无。

注记：

如果发起函数调用的线程已经在*this 对象上持锁，则函数返回 true，且这个锁的计数值加 1。如果当前线程尚未在*this 对象上持锁，即便其他线程也未在*this 对象上持锁，本函数依然有可能无法获得锁（并返回 false）。

5. std::recursive_mutex::unlock()成员函数

在目标 std::recursive_mutex 对象上，当前线程释放它所持锁中的一个。

声明：

```
void unlock();
```

前置条件：

发起函数调用的线程必须已经在*this 对象上持锁。

作用：

释放当前线程在*this 对象上所持锁中的一个。如果这是当前线程在*this 对象上持有的最后一个锁，而有线程因为等待锁住*this 对象而阻塞，则让它们其中的一个脱离阻塞。

后置条件：

发起函数调用的线程在*this 对象上所持锁的计数值减 1。

抛出：

无。

D.5.3 std::timed_mutex 类

std::mutex 类提供了基本的互斥和同步功能，而 std::timed_mutex 类（限时互斥）以此为基础，为其上的锁提供了计时功能。这种互斥必须先由 lock()、try_lock()、try_lock_for()或 try_lock_until()锁住，之后访问目标数据才

会受到保护。假设另一个线程已经在该互斥上持锁，如果再以 try_lock()试图为之加锁就会失败；lock()将引发阻塞，直到获得锁为止。若采用 try_lock_for()或 try_lock_until()，将引发阻塞直至获得锁，或者直到超出时限。一旦获得了锁（无论通过哪个函数获得），该锁必须先由 unlock()的调用释放，另一个线程才可以在该互斥上获取锁。

std::timed_mutex 满足 TimedLockable 的型别要求（可限时加锁）。

类定义：

```
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);
};
```

1. std::timed_mutex()默认构造函数

构造一个 std::timed_mutex 对象。

声明：

```
timed_mutex();
```

作用：

构造一个 std::timed_mutex 实例。

后置条件：

新构造的 std::timed_mutex 实例处于未锁定的初始状态。

抛出：

如果无法创建出新的 std::timed_mutex 实例，则抛出型别为 std::system_error 的异常。

2. std::timed_mutex()析构函数

销毁 std::timed_mutex 对象。

声明：

```
~timed_mutex();
```

前置条件：

*this 对象必须未被锁定。

作用：

销毁*this 对象。

抛出：

无。

3. std::timed_mutex::lock()成员函数

令当前线程在 std::timed_mutex 对象上获取锁。

声明：

```
void lock();
```

前置条件：

发起函数调用的线程必须还未在*this 对象上持锁。

作用：

阻塞当前线程，直到取得*this 对象上的锁。

后置条件：

*this 对象由发起函数调用的线程锁住。

抛出：

如果发生任何错误，则抛出型别为 std::system_error 的异常。

4. std::timed_mutex::try_lock()成员函数

令当前线程试图在 std::timed_mutex 对象上获取锁。

声明：

```
bool try_lock();
```

前置条件：

发起函数调用的线程必须还未在*this 对象上持锁。

作用：

令发起函数调用的线程试图以非阻塞方式在*this 对象上获取锁。

返回：

如果发起函数调用的线程获取了锁就返回 true，否则返回 false。

后置条件：

如果函数返回 true，则*this 对象被发起函数调用的线程锁定。

抛出：

无。

注记：

即便其他线程都没有在*this 对象上持锁，本函数依然可能无法获取锁（并返回 false）。

5. std::timed_mutex::try_lock_for()成员函数

令当前线程试图在 std::timed_mutex 对象上获取锁。

声明：

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

前置条件：

发起函数调用的线程必须还未在*this 对象上持锁。

作用：

在参数 relative_time 限定的时长内，发起函数调用的线程试图从*this 对象获取锁。如果 relative_time.count()的结果小于等于零，此函数调用会马上返回，相当于直接调用 try_lock()。

否则此函数调用发生阻塞，直到获得锁，或直到超出 relative_time 限定的时长。

返回：

如果发起函数调用的线程获得了锁就返回 true，否则返回 false。

后置条件：

如果函数返回 true，则*this 对象被发起函数调用的线程锁定。

抛出：

无。

注记：

即便其他线程都没有在*this 对象上持锁，本函数依然可能无法获取锁（并返回 false）。线程实际阻塞的时间也许会比原本指定的更长。只要有可能，就应该由恒稳时钟判定已经消耗的等待时间。

6. std::timed_mutex::try_lock_until()成员函数

令当前线程试图在 std::timed_mutex 对象上获取锁。

声明：

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

前置条件：

发起函数调用的线程必须还未在*this 对象上持锁。

作用：

在参数 absolute_time 指定的时刻之前，发起函数调用的线程试图从*this 对象获取锁。在进入函数调用的那一刻，如果 absolute_time<=Clock::now()成立，此函数调用就会马上返回，相当于直接调用 try_lock()。否则，此函数调用发生阻塞，直到获得锁，或直到 Clock::now()返回的时刻等于/晚于参数 absolute_time 所指定的时刻。

返回：

如果发起函数调用的线程获得了锁就返回 true，否则返回 false。

后置条件：

如果函数返回 true，则*this 对象被发起函数调用的线程锁定。

抛出：

无。

注记：

即便其他线程都没有在*this 对象上持锁，本函数依然可能无法获取锁（并返回 false）。发起调用的线程的阻塞时间并不确定，只有当此函数返回 false，且 Clock::now()返回的时刻等于/晚于参数 absolute_time 所指定的时刻，该线程才会结束阻塞。

7. std::timed_mutex::unlock()成员函数

当前线程释放在 std::timed_mutex 对象上持有的锁。

声明：

```
void unlock();
```

前置条件：

发起函数调用的线程必须已经在*this 对象上持锁。

作用：

释放当前线程在*this 对象上持有的锁。如果有线程因为等待锁住*this 对象而阻塞，则让它们当中的一个脱离阻塞。

后置条件：

发起函数调用的线程不再锁住*this 对象。

抛出：

无。

D.5.4 std::recursive_timed_mutex 类

std::recursive_mutex 类提供了互斥和同步功能，而 std::recursive_timed_mutex（限时递归互斥）以此为基础，为其上的锁提供了计时功能。限时递归互斥必须先由 lock()、try_lock()、try_lock_for()或 try_lock_until()锁住，之后访问目标数据才会受到保护。假设另一个线程已经在限时递归互斥上持锁，再通过 try_lock()试图为之加锁就会失败，而 lock()将引发阻塞，直到获得锁为止。若采用 try_lock_for()或 try_lock_until()将引发阻塞，直至获得锁，或者直到超出时限。一旦获得了锁（无论通过哪个函数取得），该锁必须先由 unlock()的调用释放，另一个线程才可以在限时递归互斥上获取锁。

因为 std::recursive_timed_mutex 具有递归特性，所以，即便一个线程已经在该类的某个实例上持锁，它还能再次调用 lock()或 try_lock()令锁的计数值增加。这些锁必须全都先由 unlock()的调用逐一释放，另一个线程才可以在该实例上获取锁。

std::recursive_timed_mutex 满足 TimedLockable 的型别（可限时加锁）要求。

类定义：

```
class recursive_timed_mutex
{
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);
};
```

1. 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();
```

作用：

构造一个 std::recursive_timed_mutex 对象。

后置条件：

新构造的 std::recursive_timed_mutex 对象处于未锁定的初始状态。

抛出：

如果无法创建出新的 std::recursive_timed_mutex 实例，则抛出型别为 std::system_error 的异常。

2. std::recursive_timed_mutex()析构函数

销毁 std::recursive_timed_mutex 对象。

声明：

```
~recursive_timed_mutex();
```

前置条件：

*this 对象必须没有被锁定。

作用：

销毁*this 对象。

抛出：

无。

3. std::recursive_timed_mutex::lock()成员函数

令当前线程在 std::recursive_timed_mutex 对象上获取锁。

声明：

```
void lock();
```

前置条件：

发起函数调用的线程必须还未在*this 对象上持锁。

作用：

阻塞当前线程，直到取得*this 对象上的锁。

后置条件：

*this 对象由发起函数调用的线程锁住。如果该线程在*this 对象上已经持锁，则令该锁的计数值加 1。

抛出：

如果发生任何错误，则抛出型别为 std::system_error 的异常。

4. std::recursive_timed_mutex::try_lock()成员函数

令当前线程试图在 std::recursive_timed_mutex 对象上获取锁。

声明：

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

作用：

令发起函数调用的线程试图以非阻塞方式在*this 对象上获取锁。

返回：

如果发起函数调用的线程获得了锁就返回 true，否则返回 false。

后置条件：

如果函数返回 true，则*this 对象被发起函数调用的线程锁定。

抛出：

无。

注记：

如果发起函数调用的线程已经在*this 对象上持锁，则函数返回 true，且该锁的计数值加 1。如果当前线程尚未在*this 对象上持锁，即便其他线程也未在*this 对象上持锁，本函数依然有可能无法获得锁（并返回 false）。

5. std::recursive_timed_mutex::try_lock_for()成员函数

令当前线程试图在 std::recursive_timed_mutex 对象上获取锁。

声明：

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

作用：

在参数 relative_time 限定的时长内，发起函数调用的线程试图从*this 对象获取锁。如果 relative_time.count()的结果小于等于零，此函数调用会马上返回，相当于直接调用 try_lock()。否则此函数调用发生阻塞，直到获得锁，或直到超出 relative_time 限定的时长。

返回：

如果发起函数调用的线程获得了锁就返回 true，否则返回 false。

后置条件：

如果函数返回 true，则*this 对象被发起函数调用的线程锁定。

抛出：

无。

注记：

如果发起函数调用的线程已经在*this 对象上持锁，则函数返回 true，且该锁的计数值加 1。如果当前线程尚未在*this 对象上持锁，即便其他线程也未在*this 对象上持锁，本函数依然有可能无法获得锁（并返回 false）。线程实际阻塞的时间也许会比原本指定的更长。只要有可能，就应该由恒稳时钟判定已经消耗的等待时间。

6. std::recursive_timed_mutex::try_lock_until()成员函数

令当前线程试图在 std::recursive_timed_mutex 对象上获取锁。

声明：

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

作用：

在参数 absolute_time 指定的时刻之前，发起函数调用的线程试图从*this 对象获取锁。在进入函数调用的那一刻，如果 absolute_time<=Clock::now()成立，此函数调用就会马上返回，相当于直接调用 try_lock()。否则此函数调用发生阻塞，直到获得锁，或直到 Clock::now()返回的时刻等于/晚于参数 absolute_time 所指定的时刻。

返回：

如果发起函数调用的线程获得了锁就返回 true，否则返回 false。

后置条件：

如果函数返回 true，则*this 对象被发起函数调用的线程锁定。

抛出：

无。

注记：

如果发起函数调用的线程已经在*this 对象上持锁，则函数返回 true，且该锁的计数值加 1。如果当前线程尚未在*this 对象上持锁，即便其他线程也未在*this 对象上持锁，本函数依然有可能无法获得锁（并返回 false）。发起调用的线程的阻塞时间并不确定，只有当此函数返回 false，而 Clock::now()返回的时刻等于/晚于参数 absolute_time 所指定的时刻，该线程才会结束阻塞。

7. std::recursive_timed_mutex::unlock()成员函数

当前线程释放在 std::recursive_timed_mutex 对象上持有的锁。

声明：

```
void unlock();
```

前置条件：

发起函数调用的线程必须已经在*this 对象上持锁。

作用：

释放当前线程在*this 对象上所持锁中的一个。如果这是当前线程在*this 对象上持有的最后一个锁，而有线程因为等待锁住*this 对象而阻塞，则让它们当中的一个结束阻塞。

后置条件：

发起函数调用的线程在*this 对象上所持锁的计数值减 1。

抛出：

无。

D.5.5 std::shared_mutex 类

std::shared_mutex 类为多线程提供了一种互斥和同步功能，用于保护频繁读取但少有改动的共享数据。这种互斥准许一个线程持有一个排他锁（exclusive lock），也准许一个或多个线程共同持有共享锁（shared lock）。如果要改动互斥所保护的数据，该互斥必须先由 lock()或 try_lock()加以排他锁。排他锁每次只能由一个线程持有，因此，如果另一个线程试图给互斥上锁，则会因调用 try_lock()而失败，或因调用 lock()而阻塞。一旦线程完成了共享数据的修改，就必须调用 unlock()释放锁，以便其他线程获得锁。仅需读取受保护数据的线程通过调用 lock_shared()或 try_lock_shared()获取共享锁。多个线程可同时持有同一个共享锁，所以即便一个线程已经在某互斥上持有共享锁，另一个线程依然能在该互斥上获取共享锁。如果一个线程尝试获取排他锁，则要等待。一旦获得共享锁的线程访问完了受保护数据，该线程必须调用 unlock_shared()释放共享锁。

std::shared_mutex 满足 Lockable 的型别要求。

类定义：

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

    shared_mutex() noexcept;
    ~shared_mutex();

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

    void lock_shared();
    void unlock_shared();
    bool try_lock_shared();
```

```
};
```

1. std::shared_mutex()默认构造函数

构造一个 std::shared_mutex 对象。

声明：

```
shared_mutex() noexcept;
```

作用：

构造一个 std::shared_mutex 实例。

后置条件：

新构造的 std::shared_mutex 对象处于未锁定的初始状态。

抛出：

无。

2. std::shared_mutex()析构函数

销毁 std::shared_mutex 对象。

声明：

```
~shared_mutex();
```

前置条件：

*this 对象必须未被锁定。

作用：

销毁*this 对象。

抛出：

无。

3. std::shared_mutex::lock()成员函数

令当前线程在 std::shared_mutex 对象上获取排他锁。

声明：

```
void lock();
```

前置条件：

发起函数调用的线程必须还未在*this 对象上持锁。

作用：

阻塞当前线程，直到在*this 对象上取得排他锁。

后置条件：

发起函数调用的线程以排他锁将*this 对象锁住。

抛出：

如果发生任何错误，则抛出型别为 std::system_error 的异常。

4. std::shared_mutex::try_lock()成员函数

令当前线程试图在 std::shared_mutex 对象上获取排他锁。

声明：

```
bool try_lock();
```

前置条件：

发起函数调用的线程必须还未在*this 对象上持锁。

作用：

发起函数调用的线程试图以非阻塞方式在*this 对象上获取排他锁。

返回：

如果发起函数调用的线程获得了锁就返回 true，否则返回 false。

后置条件：

如果函数返回 true，则发起函数调用的线程以排他锁将*this 对象锁住。

抛出：

无。

注记：

即便其他线程都没有在*this 对象上持锁，本函数依然可能无法获取锁（并返回 false）。

5. std::shared_mutex::unlock()成员函数

令当前线程释放在 std::shared_mutex 对象上持有的排他锁。

声明：

```
void unlock();
```

前置条件：

发起函数调用的线程必须已经在*this 对象上持有排他锁。

作用：

释放当前线程在*this 对象上持有的排他锁。如果有线程因为等待锁住*this 对象而阻塞，那么，让其中一个等待排他锁的线程结束阻塞，或者让一定数量的等待共享锁的线程结束阻塞。

后置条件：

发起函数调用的线程不再锁住*this 对象。

抛出：

无。

6. std::shared_mutex::lock_shared()成员函数

令当前线程在 `std::shared_mutex` 对象上获取共享锁。

声明：

```
void lock_shared();
```

前置条件：
发起函数调用的线程必须还未在*this 对象上持锁。
作用：
阻塞当前线程，直到在*this 对象上取得共享锁。
后置条件：
发起函数调用的线程以共享锁将*this 对象锁住。
抛出：
如果发生任何错误，则抛出型别为 std::system_error 的异常。

7. std::shared_mutex::try_lock_shared()成员函数

令当前线程试图在 std::shared_mutex 对象上获取共享锁。
声明：

```
bool try_lock_shared();
```

前置条件：
发起函数调用的线程必须还未在*this 对象上持锁。
作用：
令发起函数调用的线程以非阻塞方式试图在*this 对象上获取共享锁。
返回：
如果发起函数调用的线程获得了锁就返回 true，否则返回 false。
后置条件：
如果函数返回 true，则发起函数调用的线程以共享锁将*this 对象锁住。
抛出：
无。
注记：
即便其他线程都没有在*this 对象上持锁，本函数依然可能无法获取锁（并返回 false）。

8. std::shared_mutex::unlock_shared()成员函数

当前线程释放在 std::shared_mutex 对象上持有的共享锁。
声明：

```
void unlock_shared();
```

前置条件：
发起函数调用的线程必须已经在*this 对象上持有共享锁。

作用：

释放当前线程在*this 对象上持有的共享锁。如果该共享锁是*this 对象上最后一个锁，且有线程因为等待锁住*this 对象而阻塞，那么，让其中一个等待排他锁的线程结束阻塞，或者让一定数量的等待共享锁的线程结束阻塞。

后置条件：

发起函数调用的线程不再锁住*this 对象。

抛出：

无。

D.5.6 std::shared_timed_mutex 类

std::shared_timed_mutex 类为多线程提供了一种互斥和同步功能，用于保护频繁读取但少有改动的共享数据。这种互斥准许一个线程持有一个排他锁，也准许一个或多个线程共同持有共享锁。如果要改动互斥所保护的数据，该互斥必须先由 lock()或 try_lock()加以排他锁。排他锁每次只能由一个线程持有，因此，如果另一个线程试图给互斥上锁，则会因调用 try_lock()而失败，或因调用 lock()而阻塞。一旦线程完成了共享数据的修改，就必须调用 unlock()释放锁，以便其他线程获得锁。仅需读取受保护数据的线程通过调用 lock_shared()或 try_lock_shared()获取共享锁。多个线程可同时持有同一个共享锁，所以即便一个线程已经在某互斥上持有共享锁，另一个线程依然能在该互斥上获取共享锁。如果一个线程尝试获取排他锁，则要等待。一旦获得共享锁的线程访问完了受保护数据，该线程必须调用 unlock_shared()释放共享锁。

std::shared_timed_mutex 满足 Lockable 的型别要求。

类定义：

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

    shared_timed_mutex() noexcept;
    ~shared_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);

    void lock_shared();
    void unlock_shared();
    bool try_lock_shared();

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

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

1. std::shared_timed_mutex()默认构造函数

构造一个 std::shared_timed_mutex 对象。

声明：

```
shared_timed_mutex() noexcept;
```

作用：

构造一个 std::shared_timed_mutex 实例。

后置条件：

新构造的 std::shared_timed_mutex 实例处于未锁定的初始状态。

抛出：

无。

2. std::shared_timed_mutex()析构函数

销毁 std::shared_timed_mutex 对象。

声明：

```
~shared_timed_mutex();
```

前置条件：

*this 对象必须未被锁定。

作用：

销毁*this 对象。

抛出：

无。

3. std::shared_timed_mutex::lock()成员函数

令当前线程在 std::shared_timed_mutex 对象上获取排他锁。

声明：

```
void lock();
```

前置条件：

发起函数调用的线程必须还未在*this 对象上持锁。

作用：

阻塞当前线程，直到在*this 对象上取得排他锁。

后置条件：

发起函数调用的线程以排他锁将*this 对象锁住。

抛出：

如果发生任何错误，则抛出型别为 std::system_error 的异常。

4. std::shared_timed_mutex::try_lock()成员函数

令当前线程试图在 std::shared_timed_mutex 对象上获取排他锁。

声明：

```
bool try_lock();
```

前置条件：

发起函数调用的线程必须还未在*this 对象上持锁。

作用：

发起函数调用的线程试图以非阻塞方式在*this 对象上获取排他锁。

返回：

如果发起函数调用的线程获得了锁就返回 true，否则返回 false。

后置条件：

如果函数返回 true，则发起函数调用的线程以排他锁将*this 对象锁定。

抛出：

无。

注记：

即便其他线程都没有在*this 对象上持锁，本函数依然可能无法获取锁（并返回 false）。

5. std::shared_timed_mutex::try_lock_for()成员函数

令当前线程试图在 std::shared_timed_mutex 对象上获取排他锁。

声明：

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

前置条件：

发起函数调用的线程必须还未在*this 对象上持锁。

作用：

在参数 relative_time 限定的时长内，发起函数调用的线程试图从*this 对象获取排他锁。如果 relative_time.count()的结果小于等于零，此函数调用会马上返回，相当于直接调用 try_lock()。否则此函数调用发生阻塞，直到获得锁，或直到超出 relative_ time 限定的时长。

返回：

如果发起函数调用的线程获得了锁就返回 true，否则返回 false。

后置条件：

如果函数返回 true，则*this 对象被发起函数调用的线程锁定。

抛出：

无。

注记：

即便其他线程都没有在*this 对象上持锁，本函数依然可能无法获取锁（并返回 false）。线程实际阻塞的时间也许会比原本指定的更长。只要有可能，就应该由恒稳时钟判定已经消耗的等待时间。

6. std::shared_timed_mutex::try_lock_until()成员函数

令当前线程试图在 std::shared_timed_mutex 对象上获取排他锁。

声明：

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

前置条件：

发起函数调用的线程必须还未在*this 对象上持锁。

作用：

在参数 absolute_time 指定的时刻之前，发起函数调用的线程试图从*this 对象获取排他锁。在进入函数调用的那一刻，如果 absolute_time<=Clock::now()成立，此函数调用就会马上返回，相当于直接调用 try_lock()。否则此函数调用发生阻塞，直到获得锁，或直到 Clock::now()返回的时刻等于/晚于参数 absolute_time 所指定的时刻。

返回：

如果发起函数调用的线程获得了锁就返回 true，否则返回 false。

后置条件：

如果函数返回 true，则*this 对象被发起函数调用的线程锁定。

抛出：

无。

注记：

即便其他线程都没有在*this 对象上持锁，本函数依然可能无法获取锁（并返回 false）。发起调用的线程的阻塞时间并不确定，只有当此函数返回 false，而 Clock::now() 返回的时刻等于/晚于参数 absolute_time 所指定的时刻，该线程才会结束阻塞。

7. std::shared_timed_mutex::unlock()成员函数

释放当前线程在 std::shared_timed_mutex 对象上持有的排他锁。

声明：

```
void unlock();
```

前置条件：

发起函数调用的线程必须已经在*this 对象上持有排他锁。

作用：

释放当前线程在*this 对象上持有的排他锁。如果有线程因为等待锁住*this 对象而阻塞，那么，让其中一个等待排他锁的线程结束阻塞，或者让一定数量的等待共享锁的线程结束阻塞。

后置条件：

发起函数调用的线程不再锁住*this 对象。

抛出：

无。

8. std::shared_timed_mutex::lock_shared()成员函数

令当前线程在 std::shared_timed_mutex 对象上获取共享锁。

声明：

```
void lock_shared();
```

前置条件：

发起函数调用的线程必须还未在*this 对象上持锁。

作用：

阻塞当前线程，直到在*this 对象上取得共享锁。

后置条件：

发起函数调用的线程以共享锁将*this 对象锁住。

抛出：

如果发生任何错误，则抛出型别为 std::system_error 的异常。

9. std::shared_timed_mutex::try_lock_shared()成员函数

令当前线程试图在 std::shared_timed_mutex 对象上获取共享锁。

声明：

```
bool try_lock_shared();
```

前置条件：

发起函数调用的线程必须还未在*this 对象上持锁。

作用：

发起函数调用的线程以非阻塞方式试图在*this 对象上获取共享锁。

返回：

如果发起函数调用的线程获得了锁就返回 true，否则返回 false。

后置条件：

如果函数返回 true，则发起函数调用的线程以共享锁将*this 对象锁住。

抛出：

无。

注记：

即便其他线程都没有在*this 对象上持锁，本函数依然可能无法获取锁（并返回 false）。

10. std::shared_timed_mutex::try_lock_shared_for()成员函数

令当前线程试图在 std::shared_timed_mutex 对象上获取共享锁。

声明：

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

前置条件：

发起函数调用的线程必须还未在*this 对象上持锁。

作用：

在参数 relative_time 限定的时长内，发起函数调用的线程试图在*this 对象上获取共享锁。如果 relative_time.count()的结果小于等于零，此函数调用会马上返回，相当于直接调用 try_lock()。否则，此函数调用发生阻塞，直到获得锁，或直到超出 relative_time 限定的时长。

返回：

如果发起函数调用的线程获得了锁就返回 true，否则返回 false。

后置条件：

如果函数返回 true，则*this 对象被发起函数调用的线程锁定。

抛出：

无。

注记：

即便其他线程未在*this 对象上持锁，本函数依然有可能无法获得锁（并返回 false）。线程实际阻塞的时间也许会比原本指定的更长。只要有可能，就应该由恒稳时钟判定已经消耗的等待时间。

11. std::shared_timed_mutex::try_lock_until()成员函数

令当前线程试图在 std::shared_timed_mutex 对象上获取共享锁。

声明：

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

前置条件：

发起函数调用的线程必须还未在*this 对象上持锁。

作用：

在参数 relative_time 限定的时长内，发起函数调用的线程试图从*this 对象获取共享锁。在进入函数调用的那一刻，如果 absolute_time<=Clock::now()成立，此函数调用就会马上返回，相当于直接调用 try_lock()。否则，此函数调用发生阻塞，直到获得锁，或直到 Clock::now()返回的时刻等于/晚于参数 absolute_time 所指定的时刻。

返回：

如果发起函数调用的线程获得了锁就返回 true，否则返回 false。

后置条件：

如果函数返回 true，则*this 对象被发起函数调用的线程锁定。

抛出：

无。

注记：

即便其他线程未在*this 对象上持锁，本函数依然有可能无法获得锁（并返回 false）。发起调用的线程的阻塞时间并不确定，只有当此函数返回 false，而 Clock::now()返回的时刻等于/晚于参数 absolute_time 所指定的时刻，该线程才会结束阻塞。

12. std::shared_timed_mutex::unlock_shared()成员函数

当前线程释放在 std::shared_timed_mutex 对象上持有的共享锁。

声明：

```
void unlock_shared();
```

前置条件：

发起函数调用的线程必须已经在*this 对象上持有共享锁。

作用：

释放当前线程在*this 对象上持有的共享锁。如果该共享锁是*this 对象上最后一个锁，且有线程因为等待锁住*this 对象而阻塞，那么，让其中一个等待排他锁的线程结束阻塞，或者让一定数量的等待共享锁的线程结束阻塞。

后置条件：

发起函数调用的线程不再锁住*this 对象。

抛出：

无。

D.5.7 std::lock_guard 类模板

std::lock_guard 类模板为锁的归属权提供基本的包装。受到锁定的互斥由模板参数 Mutex 指定型别，它必须满足 Lockable 型别的要求。所指定的互斥在构造函数中锁定，并在析构函数中解锁。这就提供了一种简易方法，借互斥锁定代码块，而在控制流程离开代码块之际，解锁该互斥，无论是自然运行至末尾，还是因流程控制语句而转移（如 break 或 return），又或者是因抛出异常而脱离。

std::lock_guard 的实例不满足 MoveConstructible、CopyConstructible 或 CopyAssignable 的型别要求[①]。

类定义：

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

    explicit lock_guard(mutex_type& m);
    lock_guard(mutex_type& m, adopt_lock_t);
    ~lock_guard();

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

① 译者注：这些术语都是针对型别的具名要求，即 named requirements，属于 C++20 的 concept 新特性。

1. std::lock_guard()构造函数，此版本对互斥加锁

构造一个 std::lock_guard 实例，它锁住给定的互斥。

声明：

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

作用：

构造一个 std::lock_guard 实例，它指涉给定的互斥 m，调用 m.lock()。

抛出：

如果 m.lock()抛出任何异常，则本构造函数重新抛出该异常。

后置条件：

*this 对象在互斥 m 上持有一个锁。

2. std::lock_guard()构造函数，此版本接管参数互斥上的锁

构造一个 std::lock_guard 实例，原本在所给定的互斥上的锁将由它持有。

声明：

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

前置条件：

发起函数调用的线程必须已在互斥 m 上持锁。

作用：

构造一个 std::lock_guard 实例，它指涉给定的互斥 m，其上的锁原本由发起函数调用的线程持有，这个锁的归属权由新构造的实例接管。

抛出：

无。

后置条件：

*this 对象持有互斥 m 上的锁，该锁原本由发起函数调用的线程持有。

3. std::lock_guard()析构函数

销毁 std::lock_guard 实例，解锁相关的互斥。

声明：

```
~lock_guard();
```

作用：

为所指涉的互斥 m 调用 m.unlock()，该互斥是在构造*this 对象时通过参数提供的。

抛出：

无。

D.5.8 std::scoped_lock 类模板

std::scoped_lock 类模板同时为多个互斥上的锁的归属权提供基本包装。受到锁定的互斥由模板参数包 Mutexes 指定型别，其中每一个都必须满足 Lockable 的型别要求。所指定的各个互斥在构造函数中锁定，并在析构函数中解锁。这就提供了一种简易方法，借一组互斥锁定代码块，而在控制流程离开代码块之际，解锁该互斥，无论是自然运行至末尾，还是因流程控制语句而转移（如 break 或 return），又或者是因抛出异常而脱离。

std::scoped_lock 的实例不满足 MoveConstructible、CopyConstructible 或 CopyAssignable 的型别要求[①]。

类定义：

```
template <class ... Mutexes>
class scoped_lock
{
public:

    explicit scoped_lock(Mutexes& ... m);
    scoped_lock(Mutexes& ... m, adopt_lock_t);
    ~scoped_lock();

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

1. std::scoped_lock()构造函数，此版本对互斥加锁

构造一个 std::scoped_lock 实例，它锁住给定的各个互斥。

声明：

```
explicit scoped_lock(Mutexes& ... m);
```

作用：

构造一个 std::lock_guard 实例，它指涉给定的各个互斥。以组合形式对每个互斥调用 m.lock()、m.try_lock() 和 m.unlock()。为了避免死锁，所采用的算法与 std::lock() 的非成员函数相同。

抛出：

如果 m.lock() 或 m.try_lock() 抛出任何异常，则本构造函数重新抛出该异常。

后置条件：

*this 对象在每个给定的互斥上都持有一个锁。

① 译者注：这些术语都是针对型别的具名要求，即 named requirements，属于 C++的 concept 新特性。

2. std::scoped_lock()构造函数，此版本接管参数互斥上的锁

构造一个 std::scoped_lock 实例，原本在给定的各个互斥上的锁将由它持有，这些互斥必须已经由发起函数调用的线程锁住。

声明：

```
scoped_lock(Mutexes& ... m,std::adopt_lock_t);
```

前置条件：

发起函数调用的线程必须已在全体互斥 m 上持锁。

作用：

构造一个 std::scoped_lock 实例，它指涉给定的各个互斥 m。其上的各个锁原本由发起函数调用的线程持有，这些锁的归属权由新构造的实例接管。

抛出：

无。

后置条件：

*this 对象持有各个互斥 m 上的锁，这些锁原本由发起函数调用的线程持有。

3. std::scoped_lock()析构函数

销毁 std::scoped_lock 实例，解锁各个相关的互斥。

声明：

```
~scoped_lock();
```

作用：

为所指涉的每个互斥 m 调用 m.unlock()，每一个互斥都是在构造*this 对象时通过参数提供的。

抛出：

无。

D.5.9 std::unique_lock 类模板

std::unique_lock 类模板用于包装锁的归属权，它比 std::lock_guard 更加通用。受到锁定的互斥由模板参数 Mutex 指定型别，它必须满足 BasicLockable 的型别要求（基本的可锁型别）。一般地，所指定的互斥在构造函数中锁定，并在析构函数中解锁。不过，这个类还提供了额外的构造函数和成员函数，从而准许按其他方式加锁和解锁。这就提供了一种借互斥锁定代码块的方法，而在控制流程离开代码块之际，解锁该互斥，无论是自然运行至代码块末尾，还是因流程控制语句而退出（如 break 或 return），又或是因抛出异常而脱离。std::condition_variable 的 wait()

函数需要一个 std::unique_lock<std::mutex>实例，并且 std::unique_lock 的全部实例化版本均适合充当 std::condition_variable_any 所含的 wait()函数的 Lockable 参数。只要给出的 Mutex 型别满足 Lockable 的型别要求，std::unique_lock<Mutex>就满足 Lockable 的型别要求。

此外，如果给出的 Mutex 型别满足 TimedLockable 的型别要求，那么 std::unique_lock<Mutex>就会满足 TimedLockable 的型别要求。

std::unique_lock 的实例满足 MoveConstructible 和 MoveAssignable 的型别要求，但不满足 CopyConstructible 或 CopyAssignable 的型别要求[①]。

类定义：

```
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);
```

① 译者注：这些术语都是针对型别的具名要求，即 named requirements，属于 C++20 的 concept 新特性。

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

1. std::unique_lock()默认构造函数

构造一个 std::unique_lock 实例，它尚未关联任何互斥。

声明：

```
unique_lock() noexcept;
```

作用：

构造一个 std::unique_lock 实例，它并没有关联的互斥。

后置条件：

this->mutex()==NULL 和 this->owns_lock()==false 均成立。

2. 对互斥加锁的 std::unique_lock()构造函数

构造一个 std::unique_lock 实例，它锁住给定的互斥。

声明：

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

作用：

构造一个 std::unique_lock 实例，它指涉给定的互斥 m，并为其调用 m.lock()。

抛出：

如果 m.lock()抛出任何异常，则本构造函数重新抛出该异常。

后置条件：

this->owns_lock()==true 和 this->mutex()==&m 均成立。

3. 接管互斥锁的 std::unique_lock()构造函数

构造一个 std::unique_lock 实例，原本所给定的互斥上的锁将由它持有。

声明：

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

前置条件：

发起函数调用的线程必须已在互斥 m 上持锁。

作用：

构造一个 std::unique_lock 实例，它指涉给定的互斥 m。其上的锁原本由发起函数调用的线程持有，这个锁的归属权由新构造的实例接管。

抛出：

无。

后置条件：

this->owns_lock()==true 和 this->mutex()==&m 均成立。

4. 延后加锁的 std::unique_lock()构造函数

构造一个 std::unique_lock 实例，它尚未持有所给定的互斥上的锁。

声明：

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

作用：

构造一个 std::unique_lock 实例，它指涉给定的互斥。

抛出：

无。

后置条件：

this->owns_lock()==false 和 this->mutex()==&m 均成立。

5. 试图立即加锁的 std::unique_lock()构造函数

构造一个 std::unique_lock 实例，它与给定的互斥关联，并试图从该互斥取得锁。

声明：

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

前置条件：

用于实例化 std::unique_lock 的 Mutex 型别必须满足 Lockable 的型别要求。

作用：

构造一个 std::unique_lock 实例，它指涉给定的互斥 m，并为其调用 m.try_lock()。

抛出：

无。

后置条件：

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

6. std::unique_lock()构造函数，此版本试图在限定时长内加锁

构造一个 std::unique_lock 实例，它与给定的互斥关联，并试图从该互斥取得锁。

声明：

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

前置条件：

用于实例化 std::unique_lock 的 Mutex 型别必须满足 TimedLockable 的型别要求。

作用：

构造一个 std::unique_lock 实例，它指涉给定的互斥 m，并为其调用 m.try_lock_for(relative_time)。

抛出：

无。

后置条件：

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

7. std::unique_lock()构造函数，此版本试图在限定的时间点来临之前加锁

构造一个 std::unique_lock 实例，它与给定的互斥关联，并试图从该互斥取得锁。

声明：

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

前置条件：

用于实例化 std::unique_lock 的 Mutex 型别必须满足 TimedLockable 的型别要求。

作用：

构造一个 std::unique_lock 实例，它指涉给定的互斥 m，并为其调用 m.try_lock_until(absolute_time)。

抛出：

无。

后置条件：

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

8. std::unique_lock()移动构造函数

将某std::unique_lock对象上的锁的归属权转移给新创建的std::unique_lock对象。

声明：

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

作用：

构造一个std::unique_lock实例。如果对象other本来持有互斥上的锁，那么在本构造函数调用之后，该锁归属新构造的std::unique_lock对象所有。

后置条件：

other.mutex()和other.owns_lock在x调用构造函数前的旧值分别等于x.mutex()和x.owns_lock()的新结果。other.mutex()==NULL和other.owns_lock()==false都成立。

抛出：

无。

注记：

std::unique_lock不满足CopyConstructible的型别要求，故它不含拷贝构造函数，仅含移动构造函数。

9. std::unique_lock()移动赋值操作符

将某std::unique_lock对象上的锁的归属权转移给另一个std::unique_lock对象。

声明：

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

作用：

在调用前，如果this->owns_lock()的返回值是true，则调用this->unlock()。如果对象other在赋值操作之前持有一个锁，则该锁现在改由*this对象持有。

后置条件：

this->mutex()的结果等于赋值操作前的other.mutex()的值，并且this->owns_lock()的结果等于赋值操作前的other.owns_lock()的值。other.mutex()==

NULL 和 other.owns_ lock()==false 都成立。

抛出：

无。

注记：

std::unique_lock 不满足 CopyAssignable 的型别要求，故它不含拷贝构造函数，仅含移动构造函数。

10. std::unique_lock()析构函数

销毁 std::unique_lock 实例，并解锁该实例所持有的相关互斥。

声明：

```
~unique_lock();
```

作用：

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

抛出：

无。

11. std::unique_lock::swap()成员函数

在两个 std::unique_lock 对象间交换它们所关联互斥的归属权。

声明：

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

作用：

如果 other 对象在本函数调用前持有一个锁，则该锁改由*this 对象持有。

如果*this 对象在本函数调用前持有一个锁，则该锁改由 other 对象持有。

后置条件：

this->mutex()的结果等于本函数调用前的 other.mutex()的值。

other.mutex()的结果等于本函数调用前的 this->mutex()的值。

this->owns_lock()的结果等于本函数调用前的 other.owns_lock()的值。

other.owns_lock()的结果等于本函数调用前的 this->owns_lock()的值。

抛出：

无。

12. 针对 std::unique_lock 的非成员函数 swap()

在两个 std::unique_lock 对象间交换它们所关联互斥的归属权。

声明：

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

作用：

```
lhs.swap(rhs)
```

抛出：

无。

13. std::unique_lock::lock()成员函数

在*this 对象所关联的互斥上获取锁。

声明：

```
void lock();
```

前置条件：

this->mutex()!=NULL 和 this->owns_lock()==false 均成立。

作用：

调用 this->mutex()->lock()。

抛出：

如果 this->mutex()->lock()抛出任何异常，则本函数重新抛出该异常。如果 this-> mutex()==NULL 成立，则抛出型别为 std::system_error 的异常，其错误代码为 std::errc:: operation_not_permitted。在进入函数调用之际，如果 this->owns_lock()==true 成立，则抛出型别为 std::system_error 的异常，其错误代码为 std::errc::resource_deadlock_would_occur。

后置条件：

this->owns_lock()==true 成立。

14. std::unique_lock::try_lock()成员函数

试图在*this 对象所关联的互斥上获取锁。

声明：

```
bool try_lock();
```

前置条件：

用于实例化 std::unique_lock 的 Mutex 型别必须满足 Lockable 的型别要求。this->mutex()!=NULL 和 this->owns_lock()==false 都成立。

作用：

调用 this->mutex()->try_lock()。

返回：

如果 this->mutex()->try_lock()的调用返回 true，则本函数返回 true，否则本函数返回 false。

抛出：

如果 this->mutex()->try_lock()抛出任何异常，则本函数重新抛出该异常。如果 this-> mutex()==NULL 成立，则抛出型别为 std::system_error 的异常，其错误代码为 std::errc::operation_not_permitted。在进入函数调用之际，如果 this->owns_lock()==true 成立，则抛出型别为 std::system_error 的异常，其错误代码为 std::errc::resource_deadlock_ would_occur。

后置条件：

如果本函数返回 true，则 this->owns_lock()==true 成立，否则 this->owns_lock()== false 成立。

15. std::unique_lock::unlock()成员函数

释放*this 对象所关联的互斥上的锁。

声明：

```
void unlock();
```

前置条件：

this->mutex()!=NULL 和 this->owns_lock()==true 均成立。

作用：

调用 this->mutex()->unlock()。

抛出：

如果 this->mutex()->unlock()抛出任何异常，则本函数重新抛出该异常。在进入函数调用之际，如果 this->owns_lock()==false 成立，则抛出型别为 std::system_error 的异常，其错误代码为 std::errc::operation_not_permitted。

后置条件：

this->owns_lock()==false 成立。

16. std::unique_lock::try_lock_for()成员函数

试图在指定的时长内在*this 对象所关联的互斥上获取锁。

声明：

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

前置条件：

用于实例化 std::unique_lock 的 Mutex 型别必须满足 TimedLockable 的型别要求。this->mutex()!=NULL 和 this->owns_lock()==false 都成立。

作用：

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

返回：

如果 this->mutex()->try_lock_for()的调用返回 true，则本函数返回 true，否则本函数返回 false。

抛出：

如果 this->mutex()->try_lock_for()抛出任何异常，则本函数重新抛出该异常。如果 this->mutex()==NULL 成立，则抛出型别为 std::system_error 的异常，其错误代码为 std::errc::operation_not_permitted。在进入函数调用之际，如果 this->owns_lock()==true 成立，则抛出型别为 std::system_error 的异常，其错误代码为 std::errc::resource_deadlock_would_occur。

后置条件：

如果本函数返回 true，则 this->owns_lock()==true 成立，否则 this->owns_lock()==false 成立。

17. std::unique_lock::try_lock_until()成员函数

试图在指定的时限前在*this 对象所关联的互斥上获取锁。

声明：

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

前置条件：

用于实例化 std::unique_lock 的 Mutex 型别必须满足 TimedLockable 的型别要求。this->mutex()!=NULL 和 this->owns_lock()==false 都成立。

作用：

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

返回：

如果 this->mutex()->try_lock_until()的调用返回 true，则本函数返回 true，否则本函数返回 false。

抛出：

如果 this->mutex()->try_lock_until()抛出任何异常，则本函数重新抛出该异常。如果 this->mutex()==NULL 成立，则抛出型别为 std::system_error 的异常，其错误代码为 std::errc::operation_not_permitted。在进入函数调用之际，如果 this->mutex()==NULL 成立，则抛出型别为 std::system_error 的异常，其错误代码为 std::errc::resource_ deadlock_would_occur。

后置条件：

如果本函数返回 true，则 this->owns_lock()==true 成立，否则 this->owns_lock()==false 成立。

18. std::unique_lock::operator bool()成员函数

判定*this 对象是否在关联的互斥上持有锁。

声明：

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

返回：

this->owns_lock()的结果。

抛出：

无。

注记：

这是个显式转换运算符，所以，只有当其运行语境将其返回结果视作布尔值时，才会被隐式调用。如果运行语境将其结果视作整型 0/1 值，隐式调用就不会发生。

19. std::unique_lock::owns_lock()成员函数

判定*this 对象是否在关联的互斥上持有锁。

声明：

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

返回：

如果*this 对象在关联的互斥上持有锁就返回 true，否则返回 false。

抛出：

无。

20. std::unique_lock::mutex()成员函数

如果*this 对象正关联着某个互斥，则返回指向该互斥的指针。

声明：

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

返回：

返回一个指针，目标是*this 对象所关联的互斥，否则返回 NULL。

抛出：

无。

21. std::unique_lock::release()成员函数

如果*this 对象关联了互斥，就返回该互斥，并解除其关联。

声明：

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

作用：

断开*this 对象和互斥之间的关联，如果后者已经加锁，则该锁将继续被持有。

返回：

如果*this 对象在本函数调用前关联了互斥，就返回一个指向该互斥的指针，否则返回 NULL。

后置条件：

this->mutex()==NULL 和 this->owns_lock()==false 均成立。

抛出：

无。

注记：

在调用本函数之前，如果 this->owns_lock()的返回值是 true，则本函数的调用者肩负解锁关联互斥的责任。

D.5.10 std::shared_lock 类模板

类模板 std::shared_lock 与 std::unique_lock 等价，只不过它获取的是共享锁而非排他锁。受到锁定的互斥由模板参数 Mutex 指定型别，它必须满足 SharedLockable 的型别要求（可加共享锁）*。一般地，所指定的互斥在构造函数中锁定，并在析构函数中解锁。不过，这个类还提供了额外的构造函数和成员函数，从而准许按其他方式加锁和解锁。这就提供了一种借互斥锁定代码块的方法，而在控制流程离开代码块之际，解锁该互斥，无论是自然运行至代码块末尾，还是因流程控制语句而退出（如 break 或 return），又或是因抛出异常而脱离。std::shared_lock 的全部实例化版本均适合充当 std::condition_variable_any 所含的 wait()函数的 Lockable 参数。

每一份 std::shared_lock<Mutex>的实例化版本都满足 Lockable 的型别要求。此外，如果给出的 Mutex 型别满足 TimedLockable 的型别要求，那么 std::shared_lock<Mutex>亦满足 SharedTimedLockable 的型别要求（可限时加共享锁）*。

std::shared_lock 的实例满足 MoveConstructible 和 MoveAssignable 的型别要求，但不满足 CopyConstructible 或 CopyAssignable 的型别要求[①]。

① 译者注：这些术语都是针对型别的具名要求，即 named requirements，属于 C++20 的 concept 新特性。

类定义：

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

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

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

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

    ~shared_lock();

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

    shared_lock(shared_lock&& );
    shared_lock& operator=(shared_lock&& );

    void swap(shared_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;
    Mutex* release() noexcept;
};
```

1. std::shared_lock()默认构造函数

构造一个 `std::shared_lock` 实例，它尚未关联任何互斥。

声明：

```
shared_lock() noexcept;
```

作用：

构造一个 std::shared_lock 实例，它没有关联的互斥。

后置条件：

this->mutex()==NULL 和 this->owns_lock()==false 均成立。

2. std::shared_lock()构造函数

构造一个 std::shared_lock 实例，它在给定的互斥上取得共享锁。

声明：

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

作用：

构造一个 std::shared_lock 实例，它指涉给定的互斥 m，调用 m.lock_shared()。

抛出：

如果 m.lock_shared()抛出任何异常，则本构造函数重新抛出该异常。

后置条件：

this->owns_lock()==true 和 this->mutex()==&m 均成立。

3. 接管互斥锁的 std::shared_lock()构造函数

构造一个 std::shared_lock 实例，原本所给定的互斥上的锁将由它持有。

声明：

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

前置条件：

发起函数调用的线程必须已在互斥 m 上持有一个共享锁。

作用：

构造一个 std::shared_lock 实例，它指涉给定的互斥 m，其上的共享锁原本由发起函数调用的线程持有，该锁的归属权由新构造的实例接管。

抛出：

无。

后置条件：

this->owns_lock()==true 和 this->mutex()==&m 均成立。

4. 延后加锁的 std::shared_lock()构造函数

构造一个 std::shared_lock 实例，它尚未持有所给定的互斥上的锁。

声明：

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

作用：

构造一个 std::shared_lock 实例，它指涉给定的互斥。

抛出：

无。

后置条件：

this->owns_lock()==false 和 this->mutex()==&m 均成立。

5. 试图立即加锁的 std::shared_lock()构造函数

构造一个 std::shared_lock 实例，它与给定的互斥关联，并试图从该互斥取得一个共享锁。

声明：

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

前置条件：

用于实例化 std::shared_lock 的 Mutex 型别必须满足 Lockable 的型别要求。

作用：

构造一个 std::shared_lock 实例，它指涉给定的互斥 m，并为其调用 try_lock_shared()。

抛出：

无。

后置条件：

this->owns_lock()返回调用 m.try_lock_shared()的结果，且 this->mutex()==&m 成立。

6. std::shared_lock()构造函数，此版本试图在限定时长内加锁

依据给出的互斥构造一个 std::shared_lock 实例，并试图在该互斥上获取锁。

声明：

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

前置条件：

用于实例化 std::unique_lock 的 Mutex 型别必须满足 SharedTimedLockable 的型别要求。

作用：

构造一个 std::shared_lock 实例，它指涉给定的互斥 m，并为其调用 m.try_lock_shared_for(relative_time)。

抛出：

无。

后置条件：

this->owns_lock() 返回调用 m.try_lock_shared_for() 的结果，且 this-> mutex()==&m 成立。

7. std::shared_lock()构造函数，此版本试图在限定的时间点来临之前加锁

构造一个 std::shared_lock 实例，它与给定的互斥关联，并试图从该互斥取得共享锁。

声明：

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

前置条件：

用于实例化 std::shared_lock 的 Mutex 型别必须满足 SharedTimedLockable 的型别要求。

作用：

构造一个 std::shared_lock 实例，它指涉给定的互斥 m，并为其调用 m.try_lock_shared_ until(absolute_time)。

抛出：

无。

后置条件：

this->owns_lock() 返回调用 m.try_lock_shared_until() 的结果，且 this->mutex()==&m 成立。

8. std::shared_lock()移动构造函数

将某 std::shared_lock 对象上的共享锁的归属权移交给新创建的 std::shared_lock 对象。

声明：

```
shared_lock(shared_lock&& other) noexcept;
```

作用：

构造一个 std::shared_lock 实例。如果对象 other 本来持有互斥上的共享锁，那么在本构造函数调用之后，该锁归属新构造的 std::shared_lock 对象所有。

后置条件：

对于新构造的 std::shared_lock 对象 x，other.mutex()和 other.owns_lock 在 x 调用构造函数前的旧值分别等于 x.mutex()和 x.owns_lock()的新结果。other.mutex()==NULL 和 other.owns_lock()==false 均成立。

抛出：

无。

注记：

std::shared_lock 不满足 CopyConstructible 的型别要求*，故它不含拷贝构造函数，仅含移动构造函数。

9. std::shared_lock 移动赋值操作符

将某 std::shared_lock 对象上的共享锁的归属权转移给另一个 std::shared_lock 对象。

声明：

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

作用：

在本函数调用前，如果 this->owns_lock()的返回值是 true，则调用 this->unlock()。如果对象 other 在赋值操作之前拥有一个共享锁，则该锁现在改由*this 对象持有。

后置条件：

this->mutex()的结果等于赋值操作前的 other.mutex()的值，并且 this->owns_lock()的结果等于赋值操作前的 other.owns_lock()的值。other.mutex()==NULL 和 other.owns_lock()==false 均成立。

抛出：

无。

注记：

std::shared_lock 不满足 CopyAssignable 的型别要求，故它不含拷贝构造函数，仅含移动构造函数。

10. std::shared_lock()析构函数

销毁 std::shared_lock 实例，并解锁该实例所持有的相关互斥。

声明：

```
~shared_lock();
```

作用：

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

抛出：

无。

11. std::shared_lock::swap()成员函数

在两个 std::shared_lock 对象间交换它们所关联互斥的归属权。

声明：

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

作用：

如果 other 对象在本函数调用前持有一个锁，则该锁改由*this 对象持有。

如果*this 对象在本函数调用前持有一个锁，则该锁改由 other 对象持有。

后置条件：

this->mutex()的结果等于本函数调用前的 other.mutex()的值。

other.mutex()的结果等于本函数调用前的 this->mutex()的值。

this->owns_lock()的结果等于本函数调用前的 other.owns_lock()的值。

other.owns_lock()的结果等于本函数调用前的 this->owns_lock()的值。

抛出：

无。

12. 针对 std::shared_lock 的非成员函数 swap()

在两个 std::shared_lock 对象间交换它们所关联互斥的归属权。

声明：

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

作用：

```
lhs.swap(rhs)
```

抛出：

无。

13. std::shared_lock::lock()成员函数

在*this 对象所关联的互斥上获取一个共享锁。

声明：

```
void lock();
```

前置条件：

this->mutex()!=NULL 和 this->owns_lock()==false 均成立。

作用：

调用 this->mutex()->lock_shared()。

抛出：

如果 this->mutex()->lock_shared()抛出任何异常，则本函数重新抛出该异常。如果 this->mutex()==NULL 成立，则抛出型别为 std::system_error 的异常，其错误代码为 std::errc::operation_not_permitted。在进入函数调用之际，如果 this->owns_lock()==true 成立，则抛出型别为 std::system_error 的异常，其错误代码为 std::errc::resource_deadlock_would_occur。

后置条件：

this->owns_lock()==true 成立。

14. std::shared_lock::try_lock()成员函数

试图在*this 对象所关联的互斥上获取一个共享锁。

声明：

```
bool try_lock();
```

前置条件：

用于实例化 std::shared_lock 的 Mutex 型别必须满足 Lockable 的型别要求。this->mutex()!=NULL 和 this->owns_lock()==false 都成立。

作用：

调用 this->mutex()->try_lock_shared()。

返回：

如果 this->mutex()->try_lock_shared()的调用返回 true，则本函数返回 true，否则本函数返回 false。

抛出：

如果 this->mutex()->try_lock_shared()抛出任何异常，则本函数重新抛出该异常。如果 this->mutex()==NULL 成立，则抛出型别为 std::system_error 的异常，其错误代码为 std::errc::operation_not_permitted。在进入函数调用之际，如果 this->owns_lock()==true 成立，则抛出型别为 std::system_error 的异常，其错误代码为 std::errc::resource_deadlock_would_occur。

后置条件：

如果本函数返回 true，则 this->owns_lock()==true 成立，否则 this->owns_lock()==false 成立。

15. std::shared_lock::unlock()成员函数

释放*this 对象所关联的互斥上的一个共享锁。

声明：

```
void unlock();
```

前置条件：

this->mutex()!=NULL 和 this->owns_lock()==true 均成立。

作用：

调用 this->mutex()->unlock_shared()。

抛出：

如果 this->mutex()->unlock_shared()抛出任何异常，则本函数重新抛出该异常。在进入函数调用之际，如果 this->owns_lock()==false 成立，则抛出型别为 std::system_error 的异常，其错误代码为 std::errc::operation_not_permitted。

后置条件：

```
this->owns_lock()==false.
```

16. std::shared_lock::try_lock_for()成员函数

试图在指定的时限内在*this 对象所关联的互斥上获取一个共享锁。

声明：

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

前置条件：

用于实例化 std::shared_lock 的 Mutex 型别必须满足 SharedTimedLockable 的型别要求。this->mutex()!=NULL 和 this->owns_lock()==false 都成立。

作用：

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

返回：

如果 this->mutex()->try_lock_shared_for()的调用返回 true，则本函数返回 true，否则本函数返回 false。

抛出：

如果 this->mutex()->try_lock_shared_for()抛出任何异常，则本函数重新抛出该异常。如果 this->mutex()==NULL 成立，则抛出型别为 std::system_error 的异常，其错误代码为 std::errc::operation_not_permitted。在进入

函数调用之际，如果this->owns_lock()==true成立，则抛出型别为std::system_error的异常，其错误代码为std::errc::resource_deadlock_would_occur。

后置条件：

如果本函数返回true，则this->owns_lock()==true成立，否则this->owns_lock()==false成立。

17. std::shared_lock::try_lock_until()成员函数

试图在指定的时限内在*this对象所关联的互斥上获取一个共享锁。

声明：

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

前置条件：

用于实例化std::shared_lock的Mutex型别必须满足SharedTimedLockable的型别要求。this->mutex()!=NULL和this->owns_lock()==false都成立。

作用：

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

返回：

如果this->mutex()->try_lock_shared_until()的调用返回true，则本函数返回true，否则本函数返回false。

抛出：

如果this->mutex()->try_lock_shared_until()抛出任何异常，则本函数重新抛出该异常。如果this->mutex()==NULL成立，则抛出型别为std::system_error的异常，其错误代码为std::errc::operation_not_permitted。在进入函数调用之际，如果this->owns_lock()==true成立，则抛出型别为std::system_error的异常，其错误代码为std::errc::resource_deadlock_would_occur。

后置条件：

如果本函数返回true，则this->owns_lock()==true成立，否则this->owns_lock()==false成立。

18. std::shared_lock::operator bool()成员函数

判定*this对象是否在关联的互斥上持有一个共享锁。

声明：

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

返回：

this->owns_lock()。

抛出：

无。

注记：

这是个显式转换运算符，所以，只有当运行语境将其返回结果视作布尔值时，才会被隐式调用，但如果运行语境将其结果视作整型 0/1 值，隐式调用就不会发生。

19. std::shared_lock::owns_lock()成员函数

判定*this 对象是否在关联的互斥上持有一个共享锁。

声明：

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

返回：

如果*this 对象在关联的互斥上持有一个共享锁，就返回 true，否则返回 false。

抛出：

无。

20. std::shared_lock::mutex()成员函数

如果*this 对象正关联着某个互斥，则返回指向该互斥的指针。

声明：

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

返回：

返回一个指针，目标是*this 对象所关联的互斥，否则返回 NULL。

抛出：

无。

21. std::shared_lock::release()成员函数

如果*this 对象关联了互斥，就返回该互斥，并解除其关联。

声明：

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

作用：

断开*this 对象和互斥之间的关联，如果后者已经加锁，则该锁将继续被持有。

返回：

如果*this 对象在本函数调用前关联了互斥，就返回一个指向该互斥的指针，否则返回 NULL。

后置条件：

this->mutex()==NULL 和 this->owns_lock()==false 均成立。

抛出：

无。

注记：

在调用本函数之前，如果 this->owns_lock()的返回值是 true，则本函数的调用者肩负解锁关联互斥的责任。

D.5.11 std::lock()函数模板

std::lock()函数模板提供的加锁方法可同时锁定多个互斥，因此不必再冒险逐一加锁，从而避免因加锁次序不一致而导致死锁。

声明：

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

前置条件：

给定的可锁对象所属的型别（即 LockableType1、LockableType2 等）都应符合 Lockable 的型别要求。

作用：

为给定的每个可锁对象调用其成员函数 lock()、try_lock()和 unlock()，在避免死锁的前提下为它们分别获得一个锁，其中的加锁次序并不明确。

后置条件：

当前线程在每个给出的可锁对象上都分别持有一个锁。

抛出：

如果传入对象的 lock()、try_lock()或 unlock()调用抛出任何异常，则本函数重新抛出该异常。

注记：

假设出现了异常，而传入的对象中的某些已通过自身的 lock()或 try_lock()获得了锁，那么，其中的每个持锁对象都要先行调用 unlock()，全部调用完成后，std::lock 的函数调用才向外传播异常。

D.5.12 std::try_lock()函数模板

std::try_lock()函数模板准许我们试着一次锁定多个可锁对象，使得它们全部被锁住，或一个都没有被锁住。

声明：

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

前置条件：

给定的可锁对象所属的型别（即 LockableType1、LockableType2 等）都应符合 Lockable 的型别要求。

作用：

逐一为每个给定的可锁对象调用其自身的 try_lock() 函数，试图分别为它们获取一个锁。如果此过程中的某个 try_lock() 调用返回 false 或抛出异常，那么已经获取锁的对象会相应地调用 unlock() 以释放其锁。

返回：

如果全部对象都获得了锁（全体 try_lock() 的调用都返回 true），就返回-1；否则返回一个索引值（从 0 开始），标示第几个 try_lock() 调用返回了 false。

后置条件：

如果本函数返回-1，则表明当前线程在每个给定的可锁对象上都持有一个锁。否则，本函数调用中一度取得的任意的锁都会被释放。

抛出：

如果任何一个 try_lock() 的调用抛出任何异常，本函数会重新抛出该异常。

注记：

假设出现了异常，而传入的对象中的某些已通过自身的 try_lock() 获得了锁，那么，其中的每个持锁对象都要先行调用 unlock()，全部调用完成后，std::try_lock 的函数调用才向外传播异常。

D.5.13 std::once_flag 类

std::once_flag 的实例需要配合 std::call_once() 函数使用，以确保具体某个函数正好被调用一次，即便有多个线程并发调用。

std::once_flag 的实例不满足 CopyConstructible、CopyAssignable、MoveConstructible 和 MoveAssignable 的型别要求[①]。

类定义：

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

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

① 译者注：这些术语都是针对型别的具名要求，即 named requirements，属于 C++20 的 concept 新特性。

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

std::once_flag()默认构造函数

std::once_flag()默认构造函数创建一个 std::once_flag 实例，其状态表明相关的函数尚未发生调用。

声明：

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

作用：

构造一个 std::once_flag 实例，其状态表明相关的函数尚未发生调用。由于本函数是 constexpr 构造函数，因此具有静态生存期的实例在静态初始化阶段即进行初始化，从而避免条件竞争或初始化次序问题。

D.5.14 std::call_once()函数模板

std::call_once()函数模板需要配合 std::call_once_flag 实例使用，以确保具体某个函数正好被调用一次，即便有多个线程并发调用。

声明：

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

前置条件：

对于给定的参数值 func 和 args，表达式 INVOKE(func,args)应合法有效。模板参数 Callable 和 Args 的每个成员都符合 MoveConstructible 的型别要求。

作用：

针对同一个 std::once_flag 对象发起的多个 std::call_once()调用会按串行方式逐一执行。针对同一个 std::once_flag 对象，如果 std::call_once()函数未曾发起过有效调用，则参数 func（或 func 的副本）代表的可调用对象会被调用，相当于执行 INVOKE(func,args)，当且仅当 func 的调用正常返回，没有抛出异常，这个 std::call_once()才是有效调用。如果可调用对象抛出了异常，则该异常会传播到本函数的调用者。针对同一个 std::once_flag 对象，如果 std::call_once()函数曾经发起过有效调用，则 std::call_once()函数会直接返回，而不调用 func。

同步：

针对同一个 std::once_flag 对象，如果多个线程同时发起了 std::call_once()函数调用，则其中只有一个是有效调用，它会先行结束，其他的 std::call_once()调用才会发生。

抛出：

如果上述效果无法达到，或者 func 所表示的可调用对象在调用过程中向外传播了任何异常，则本函数抛出 std::system_error 型别的异常。

D.6 <ratio>头文件

<ratio>头文件提供了编译期对分数运算的支持。

头文件内容：

```
namespace std
{
    template<intmax_t N,intmax_t D=1>
    class ratio;

    // 分数算术运算
    template <class R1, class R2>
    using ratio_add = /*见后文详述*/;

    template <class R1, class R2>
    using ratio_subtract = /*见后文详述*/;

    template <class R1, class R2>
    using ratio_multiply = /*见后文详述*/;

    template <class R1, class R2>
    using ratio_divide = /*见后文详述*/;

    // 分数比较运算
    template <class R1, class R2>
    struct ratio_equal;

    template <class R1, class R2>
    struct ratio_not_equal;

    template <class R1, class R2>
    struct ratio_less;

    template <class R1, class R2>
    struct ratio_less_equal;

    template <class R1, class R2>
    struct ratio_greater;

    template <class R1, class R2>
    struct ratio_greater_equal;

    typedef ratio<1, 1000000000000000000> atto;
    typedef ratio<1, 1000000000000000> femto;
    typedef ratio<1, 1000000000000> pico;
    typedef ratio<1, 1000000000> nano;
    typedef ratio<1, 1000000> micro;
```

```
    typedef ratio<1, 1000> milli;
    typedef ratio<1, 100> centi;
    typedef ratio<1, 10> deci;
    typedef ratio<10, 1> deca;
    typedef ratio<100, 1> hecto;
    typedef ratio<1000, 1> kilo;
    typedef ratio<1000000, 1> mega;
    typedef ratio<1000000000, 1> giga;
    typedef ratio<1000000000000, 1> tera;
    typedef ratio<1000000000000000, 1> peta;
    typedef ratio<1000000000000000000, 1> exa;
}
```

D.6.1 std::ratio 类模板

std::ratio 类模板提供了一种机制，使我们可以在编译期进行实数值算术运算，如二分之一（std::ratio<1,2>）、三分之二（std::ratio<2,3>）、四十三分之十五(std::ratio<15,43>)等。在C++标准库中,对类模板 std::chrono::duration 进行实例化时，这些值用于指定其计时单元大小。

类定义：

```
template <intmax_t N, intmax_t D = 1>
class ratio
{
public:
    typedef ratio<num, den> type;
    static constexpr intmax_t num= /*见后文详述*/;
    static constexpr intmax_t den= /*见后文详述*/;
};
```

要求：

D 值不得为 0。

描述：

num 和 den 是分数 N/D 的分子和分母,其中 N/D 是最简分式。den 值永远是正数。如果 N 和 D 的正负号相同，则 num 为正数，否则 num 为负数。

示例：

```
ratio<4,6>::num == 2
ratio<4,6>::den == 3
ratio<4,-6>::num == -2
ratio<4,-6>::den == 3
```

D.6.2 std::ratio_add 模板别名

std::ratio_add 模板别名提供了一种机制，采用分数的运算法则，在编译期将

两个 std::ratio 值相加。

定义：

```
template <class R1, class R2>
using ratio_add = std::ratio<see below>;
```

前置条件：

R1 和 R2 均必须为类模板 std::ratio 的实例化。

作用：

将 ratio_add<R1,R2>定义成一个别名，代表 std::ratio 的某个实例化，表示两个分数值 R1 和 R2 的和，前提是所求之和没有溢出。如果运算结果发生溢出，则表明程序存在问题。在结果没有溢出的前提下，std::ratio_add<R1,R2>中的 den 值与 num 值和 std::ratio<R1::num * R2::den + R2::num * R1::den, R1::den * R2::den>中的值相同。

示例：

```
std::ratio_add<std::ratio<1,3>, std::ratio<2,5>>::num == 11
std::ratio_add<std::ratio<1,3>, std::ratio<2,5>>::den == 15

std::ratio_add<std::ratio<1,3>, std::ratio<7,6>>::num == 3
std::ratio_add<std::ratio<1,3>, std::ratio<7,6>>::den == 2
```

D.6.3 std::ratio_subtract 模板别名

std::ratio_subtract 模板别名提供了一种机制，采用分数的运算法则，在编译期将两个 std::ratio 值相减。

定义：

```
template <class R1, class R2>
using ratio_subtract = std::ratio<see below>;
```

前置条件：

R1 和 R2 均必须为类模板 std::ratio 的实例化。

作用：

将 ratio_subtract<R1,R2>定义成一个别名，代表 std::ratio 的某个实例化，表示两个分数值 R1 和 R2 的差，前提是所求之差没有溢出。如果运算结果发生溢出，则表明程序存在问题。在结果没有溢出的前提下，std::ratio_subtract <R1,R2>中的 den 值与 num 值和 std::ratio<R1::num * R2::den - R2::num * R1::den, R1::den * R2::den>中的值相同。

示例：

```
std::ratio_subtract<std::ratio<1,3>, std::ratio<1,5>>::num == 2
std::ratio_subtract<std::ratio<1,3>, std::ratio<1,5>>::den == 15
```

```
std::ratio_subtract<std::ratio<1,3>, std::ratio<7,6>>::num == -5
std::ratio_subtract<std::ratio<1,3>, std::ratio<7,6>>::den == 6
```

D.6.4 std::ratio_multiply 模板别名

std::ratio_multiply 模板别名提供了一种机制，采用分数的运算法则，在编译期将两个 std::ratio 值相乘。

定义：

```
template <class R1, class R2>
using ratio_multiply = std::ratio<see below>;
```

前置条件：

R1 和 R2 均必须为类模板 std::ratio 的实例化。

作用：

将 ratio_multiply<R1,R2>定义成一个别名，代表 std::ratio 的某个实例化，表示两个分数值 R1 和 R2 的积，前提是所求乘积没有溢出。如果运算结果发生溢出，则表明程序存在问题。在结果没有溢出的前提下，std::ratio_subtract<R1,R2>中的 den 值与 num 值和 std::ratio<R1::num * R2::num, R1::den * R2::den>中的值相同。

示例：

```
std::ratio_multiply<std::ratio<1,3>, std::ratio<2,5>>::num == 2
std::ratio_multiply<std::ratio<1,3>, std::ratio<2,5>>::den == 15

std::ratio_multiply<std::ratio<1,3>, std::ratio<15,7>>::num == 5
std::ratio_multiply<std::ratio<1,3>, std::ratio<15,7>>::den == 7
```

D.6.5 std::ratio_divide 模板别名

std::ratio_divide 模板别名提供了一种机制，采用分数的运算法则，在编译期将两个 std::ratio 值相除。

定义：

```
template <class R1, class R2>
using ratio_divide = std::ratio</*见下*/>;
```

前置条件：

R1 和 R2 均必须为类模板 std::ratio 的实例化。

作用：

将 ratio_divide<R1,R2>定义成一个别名，代表 std::ratio 的某个实例化，

表示两个分数值 R1 和 R2 的商，前提是所求之商没有溢出。如果运算结果发生溢出，则表明程序存在问题。在结果没有溢出的前提下，std::ratio_add<R1,R2>中的 den 值与 num 值和 std::ratio<R1::num * R2::den, R1::den * R2::num>中的值相同。

示例：

```
std::ratio_divide<std::ratio<1,3>, std::ratio<2,5>>::num == 5
std::ratio_divide<std::ratio<1,3>, std::ratio<2,5>>::den == 6

std::ratio_divide<std::ratio<1,3>, std::ratio<15,7>>::num == 7
std::ratio_divide<std::ratio<1,3>, std::ratio<15,7>>::den == 45
```

D.6.6 std::ratio_equal 类模板

std::ratio_equal 类模板提供了一种机制，采用分数的运算法则，在编译期对比两个 std::ratio 值是否相等。

类定义：

```
template <class R1, class R2>
class ratio_equal:
    public std::integral_constant<
        bool,(R1::num == R2::num)& & (R1::den == R2::den)>
{};
```

请注意，第二个模板参数是个布尔值，而非类型。

前置条件：

R1 和 R2 均必须为类模板 std::ratio 的实例化。

示例：

```
std::ratio_equal<std::ratio<1,3>, std::ratio<2,6>>::value == true
std::ratio_equal<std::ratio<1,3>, std::ratio<1,6>>::value == false
std::ratio_equal<std::ratio<1,3>, std::ratio<2,3>>::value == false
std::ratio_equal<std::ratio<1,3>, std::ratio<1,3>>::value == true
```

D.6.7 std::ratio_not_equal 类模板

std::ratio_not_equal 类模板提供了一种机制，采用分数的运算法则，在编译期对比两个 std::ratio 值是否互异。

类定义：

```
template <class R1, class R2>
class ratio_not_equal:
    public std::integral_constant<bool,!ratio_equal<R1,R2>::value>
{};
```

前置条件：

R1 和 R2 均必须为类模板 std::ratio 的实例化。

示例：

```
std::ratio_not_equal<std::ratio<1,3>, std::ratio<2,6>>::value == false
std::ratio_not_equal<std::ratio<1,3>, std::ratio<1,6>>::value == true
std::ratio_not_equal<std::ratio<1,3>, std::ratio<2,3>>::value == true
std::ratio_not_equal<std::ratio<1,3>, std::ratio<1,3>>::value == false
```

D.6.8 std::ratio_less 类模板

std::ratio_less 类模板提供了一种机制，采用分数的运算法则，在编译期比较两个 std::ratio 值的大小。

类定义：

```
template <class R1, class R2>
class ratio_less:
    public std::integral_constant<bool,/*见译者注*/ >
{};
```

译者注：第二个模板参数是一个布尔值。

前置条件：

R1 和 R2 均必须为类模板 std::ratio 的实例化。

作用：

只要有可能，模板实现应该采用避免溢出的方法来计算结果。如果发生溢出，则表明程序存在问题。

示例：

```
std::ratio_less<std::ratio<1,3>, std::ratio<2,6>>::value == false
std::ratio_less<std::ratio<1,6>, std::ratio<1,3>>::value == true
std::ratio_less<
    std::ratio<999999999,1000000000>,
    std::ratio<1000000001,1000000000>>::value == true
std::ratio_less<
    std::ratio<1000000001,1000000000>,
    std::ratio<999999999,1000000000>>::value == false
```

D.6.9 std::ratio_greater 类模板

std::ratio_greater 类模板提供了一种机制，采用分数的运算法则，在编译期比较两个 std::ratio 值的大小。

类定义：

```
template <class R1, class R2>
class ratio_greater:
    public std::integral_constant<bool,ratio_less<R2,R1>::value>
{};
```

译者注：请注意第二个模板参数，它是个布尔值，由 ratio_less<R1,R2>::value 决定。

前置条件：

R1 和 R2 均必须为类模板 std::ratio 的实例化。

D.6.10 std::ratio_less_equal 类模板

std::ratio_less_equal 类模板提供了一种机制，采用分数的运算法则，在编译期比较两个 std::ratio 值的大小。

类定义：

```
template <class R1, class R2>
class ratio_less_equal:
    public std::integral_constant<bool,!ratio_less<R2,R1>::value>
{};
```

译者注：请注意第二个模板参数，它是个布尔值，由!ratio_less<R1,R2>::value 决定。

前置条件：

R1 和 R2 均必须为类模板 std::ratio 的实例化。

D.6.11 std::ratio_greater_equal 类模板

std::ratio_greater_equal 类模板提供了一种机制，采用分数的运算法则，在编译期比较两个 std::ratio 值的大小。

类定义：

```
template <class R1, class R2>
class ratio_greater_equal:
    public std::integral_constant<bool, !ratio_less<R1,R2>::value>
{};
```

译者注：请注意第二个模板参数，它是个布尔值，由!ratio_less<R1,R2>::value 决定。

前置条件：

R1 和 R2 均必须为类模板 std::ratio 的实例化。

D.7 <thread>头文件

<thread>头文件给出了管控线程和分辨不同线程的工具，还提供了一些函数，可以让当前线程休眠。

头文件：

```
namespace std
{
    class thread;

    namespace this_thread
    {
        thread::id get_id() noexcept;

        void yield() noexcept;

        template<typename Rep,typename Period>
        void sleep_for(
            std::chrono::duration<Rep,Period> sleep_duration);

        template<typename Clock,typename Duration>
        void sleep_until(
            std::chrono::time_point<Clock,Duration> wake_time);
    }
}
```

D.7.1 std::thread 类

std::thread 类的实例用于管控一个执行线程（thread of execution）。它提供了一种方法，以启动新的执行线程，然后等待该执行线程结束运行。它还提供了一种分辨不同线程的方法，让其他函数也得以管控执行线程。

类定义：

```
class thread
{
public:
    // 型别
    class id;
    typedef implementation-defined native_handle_type; //可选的 typedef

    // 构造函数和析构函数
    thread() noexcept;

    ~thread();

    template<typename Callable,typename Args...>
    explicit thread(Callable&&func, Args&&... args);

    // 拷贝构造函数和移动构造函数
    thread(thread const& other) = delete;
    thread(thread&& other) noexcept;

    thread& operator=(thread const& other) = delete;
    thread& operator=(thread&& other) noexcept;
```

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

    void join();
    void detach();
    bool joinable() const noexcept;

    id get_id() const noexcept;

    native_handle_type native_handle();

    static unsigned hardware_concurrency() noexcept;
};

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

1. std::thread::id 类

一个 std::thread::id 实例标识出一个特定的执行线程。

类定义：

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

bool operator==(thread::id x, thread::id y) noexcept;
bool operator!=(thread::id x, thread::id y) noexcept;
bool operator<(thread::id x, thread::id y) noexcept;
bool operator<=(thread::id x, thread::id y) noexcept;
bool operator>(thread::id x, thread::id y) noexcept;
bool operator>=(thread::id x, thread::id y) noexcept;

template<typename charT, typename traits>
basic_ostream<charT, traits>&
operator<< (basic_ostream<charT, traits>&& out, thread::id id);
```

注记：

如果一个 std::thread::id 实例的值用于标识某特定的执行线程，则它不等于按默认方式构造的 std::thread::id 实例的值，也不等于代表其他任何执行线程的值。针对某个特定线程，它具有的 std::thread::id 实例的值并不可预知，如果同一个程序运行多次，该特定线程具有的 std::thread::id 实例的值也可能不同。std::thread::id 满足 CopyAssignable 和 CopyConstructible 的型别要求，故我们可以随意复制和赋值。

译者注：这两个术语都是针对型别的具名要求，即 named requirements，属于 C++的 concept 新特性的其中一部分。

2. std::thread::id()默认构造函数

构造一个 std::thread::id 对象，它尚未代表任何执行线程。

声明：

```
id() noexcept;
```

作用：

构造一个 std::thread::id 实例，它的值表示“没有线程”，这是唯一确定的值。

抛出：

无。

注记：

全部按默认方式构造的 std::thread::id 实例都存有相同的值。

3. std::thread::id 等值比较运算符

对比两个 std::thread::id 实例，判定它们是否代表相同的线程。

声明：

```
bool operator==(std::thread::id lhs, std::thread::id rhs) noexcept;
```

返回：

如果参数 lhs 和 rhs 的值表示同一个执行线程，或者两者的值都表示“没有线程”，就返回 true。如果参数 lhs 和 rhs 的值分别表示不同的执行线程，或者其中一个的值表示某执行线程，而另一个的值表示“没有线程”，则返回 false。

抛出：

无。

4. std::thread::id 相异比较运算符

对比两个 std::thread::id 实例，判定它们是否代表不同的线程。

声明：

```
bool operator!=(std::thread::id lhs,std::thread::id rhs) noexcept;
```

返回：

```
!(lhs==rhs)
```

抛出：

无。

5. std::thread::id 小于比较运算符

全体线程的 ID 值构成一个序列，利用本运算符对比两个 std::thread::id 实例，

判定其中一个是否在序列中位于另一个前方。

声明：

```
bool operator<(std::thread::id lhs, std::thread::id rhs) noexcept;
```

返回：

在全体线程的 ID 值构成的序列中，如果 lhs 的值排在 rhs 前方，就返回 true。如果 lhs!=rhs，那么 lhs<rhs 与 rhs<lhs 中有且仅有一个成立，其中一个成立，另一个的结果就是 false。如果 lhs==rhs，那么 lhs<rhs 与 rhs<lhs 的结果皆为 false。

抛出：

无。

注记：

按默认方式构造的 std::thread::id 实例持有的值表示“没有线程”，该值唯一确定，它小于表示执行线程的其他任何 std::thread::id 实例的值。如果两个 std::thread::id 实例的值相等，则它们彼此都不小于对方。任意一组两两不等的 std::thread::id 实例的值会构成一个总序列，在程序的一次运行过程中始终保持不变。如果同一个程序运行多次，该序列可能有异。

6. std::thread::id 小于等于比较运算符

对比两个 std::thread::id 实例，判定两者是否相等，或者一个是否在线程 ID 总序列中位于另一个前方。

声明：

```
bool operator<=(std::thread::id lhs, std::thread::id rhs) noexcept;
```

返回：

```
!(rhs<lhs)
```

抛出：

无。

7. std::thread::id 大于比较运算符

对比两个 std::thread::id 实例，判定一个是否在线程 ID 总序列中位于另一个后方。

声明：

```
bool operator>(std::thread::id lhs, std::thread::id rhs) noexcept;
```

返回：

```
rhs<lhs
```

抛出：

无。

8. std::thread::id 大于等于比较运算符

对比两个 std::thread::id 实例，判定它们是否相等，或者一个是否在线程 ID 总序列中位于另一个后方。

声明：

```
bool operator>=(std::thread::id lhs, std::thread::id rhs) noexcept;
```

返回：

```
!(lhs<rhs)
```

抛出：

无。

9. std::thread::id 流插入运算符

将某个 std::thread::id 的值以字符串的表示形式插入指定的流。

声明：

```
template<typename charT, typename traits>
basic_ostream<charT, traits>&
operator<< (basic_ostream<charT, traits>&& out, thread::id id);
```

作用：

将某个 std::thread::id 的值以字符串的表示形式插入指定的流。

返回：

```
out
```

抛出：

无。

注记：

字符串的表示形式并不明确。相等的 std::thread::id 实例具有相同的表现形式，不相等的 std::thread::id 实例的表现形式相异。

10. std::thread::native_handle_type typedef

native_handle_type 是一个型别的 typedef，与平台相关的 API 配合使用。

声明：

```
typedef implementation-defined native_handle_type;
```

译者注：implementation-defined 意为由线程库实现的、自行定义的某种

型别。

注记：

这个 typedef 是可选项。如果出现，程序库的实现需给出一个型别，适合与平台相关的原生 API 一起使用。

11. std::thread::native_handle()成员函数

返回一个型别为 native_handle_type 的值，它表示与*this 对象关联的执行线程。

声明：

```
native_handle_type native_handle();
```

注记：

这个函数是可选项。如果出现，所返回的值应该适合与平台相关的原生 API 一起使用。

12. std::thread()默认构造函数

构造一个 std::thread 对象，但它没有关联的执行线程。

声明：

```
thread() noexcept;
```

作用：

构造一个 std::thread 实例，但它没有关联的执行线程。

后置条件：

对于新构造的 std::thread 对象 x，x.get_id()==id()成立[①]。

抛出：

无。

13. std::thread()构造函数

构造一个 std::thread 对象，它关联一个新的执行线程。

声明：

```
template<typename Callable,typename Args...>
explicit thread(Callable&& func, Args&&... args);
```

前置条件：

func 和 args 的每个元素均必须满足 MoveConstructible 的型别要求。

作用：

① 译者注：右边的 id()表示按默认构造方式构造一个 std::thread::id 对象，其值表示“没有线程”。

构造一个 std::thread 对象，并使之关联一个新的执行线程。将参数 func 和 args 中的每个元素复制或移动到线程内部存储空间，该存储空间的生存期与新的执行线程一致。在新的执行线程上运行 INVOKE(copy-of-func,copy-of-args)，其中 copy-of-func 即 func 的副本，copy-of-args 即 args 的副本）。

后置条件：

对于新构造的 std::thread 对象 x，x.get_id()!=id()成立。

抛出：

如果新线程无法启动，就抛出型别为 std::system_error 的异常。如果因复制或移动参数 func 和 args 到线程内部存储空间而导致异常，则本构造函数重新抛出该异常。

同步：

此构造函数将先行调用，新的执行线程会随之运行给出的函数。

14. std::thread()移动构造函数

将执行线程的归属权从某个 std::thread 对象转移到新创建的 std::thread 对象上。

声明：

```
thread(thread&& other) noexcept;
```

作用：

构建一个 std::thread 实例。如果对象 other 已经与某执行线程关联，那么在构造函数调用以后，该执行线程就会与新的 std::thread 对象关联。否则，新创建的 std::thread 对象不具有任何关联的执行线程。

后置条件：

对于新构造的 std::thread 对象 x，x.get_id()等于在 x 的构造函数调用前的 other.get_id()的值。且 other.get_id()==id()成立。

抛出：

无。

注记：

std::thread 对象不满足 CopyConstructible 的型别要求，故它们没有拷贝构造函数，只有移动构造函数。

15. std::thread()析构函数

销毁 std::thread 对象。

声明：

```
~thread();
```

作用：

销毁*this 对象。如果*this 对象关联了执行线程（this->joinable()会返回 true），就调用 std::terminate()终止程序。

抛出：

无。

16. std::thread 移动赋值操作符

从一个 std::thread 对象向另一个对象转移执行线程的归属权。

声明：

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

作用：

如果在赋值操作之前，this->joinable()返回 true，就调用 std::terminate()终止程序。如果 other 对象已经关联了执行线程，那么在赋值操作之后，该执行线程与*this 对象关联，否则*this 对象不具有关联的执行线程。

后置条件：

this->get_id()等于赋值操作前的 other.get_id()的值，且 other.get_id()==id()成立。

抛出：

无。

注记：

std::thread 对象不满足 CopyConstructible 的型别要求，故它们没有拷贝构造函数，只有移动构造函数。

17. std::thread::swap()成员函数

在两个 std::thread 之间交换它们所关联的执行线程的归属权。

声明：

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

作用：

如果 other 对象已经关联了执行线程，那么在赋值操作之后，该执行线程与*this 对象关联，否则*this 对象不具有关联的执行线程。如果*this 对象已经关联了执行线程，那么在赋值操作之后，该执行线程与 other 对象关联，否则 other 对象不具有关联的执行线程。

后置条件：

当前 this->get_id()的值等于函数调用前的 other.get_id()的值，当前 other.get_id()的值等于函数调用前的 this->get_id()的值。

抛出：

无。

18. 针对 std::threads 的非成员函数 swap()

在两个 std::thread 之间交换它们所关联的执行线程的归属权。

声明：

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

作用：

```
lhs.swap(rhs)
```

抛出：

无。

19. std::thread::joinable()成员函数

检查*this 对象是否具有关联的执行线程。

声明：

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

返回：

如果*this 对象具有关联的执行线程就返回 true，否则返回 false。

抛出：

无。

20. std::thread::join()成员函数

等待*this 对象所关联的执行线程完成运行。

声明：

```
void join();
```

前置条件：

this->joinable()会返回 true。

作用：

阻塞当前线程，直到*this 对象关联的执行线程完成运行为止。

后置条件：

this->get_id()==id()成立。本函数调用发生后，*this 对象原本关联的执行线程结束运行。

同步：

*this 对象所关联的执行线程将先行结束，join()调用会随之返回。

抛出：

如果上述效果无法达到，或 this->joinable()返回 false，就抛出 std::

system_error 型别的异常。

21. std::thread::detach()成员函数

分离*this 对象所关联的执行线程，以让其自行完结。

声明：

```
void detach();
```

前置条件：

this->joinable()的返回值是 true。

作用：

分离*this 对象所关联的执行线程。

后置条件：

this->get_id()==id()且 this->joinable()==false 成立。

*this 对象所关联的执行线程被分离，该线程不再与 std::thread 对象关联。

抛出：

如果上述效果无法达到，或 this->joinable()返回 false，就抛出 std::system_error 型别的异常。

22. std::thread::get_id()成员函数

返回一个型别为 native_handle_type 的值，它标识与*this 对象关联的执行线程。

声明：

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

返回：

如果*this 对象关联了执行线程，则返回标识该线程的 std::thread::id 实例，否则返回按默认方式构造的 std::thread::id 对象。

抛出：

无。

23. std::thread::hardware_concurrency()静态成员函数

返回一个参考值，表示当前硬件可以并发运行的线程数目。

声明：

```
unsigned hardware_concurrency() noexcept;
```

返回：

返回当前硬件可以并发运行的线程数目（这个数目可能是系统中处理器的数目）。

如果无法获取这项信息，或者无法清楚界定，则本函数返回 0。

抛出：

无。

D.7.2 std::this_thread 名字空间

std::this_thread 名字空间中的函数在发起调用的线程上运作。

1. std::this_thread::get_id()非成员函数

返回一个型别为 std::thread::id 的值，用于标识当前的执行线程。

声明：

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

返回：

返回一个 std::thread::id 实例，它的值标识了当前线程。

抛出：

无。

2. std::this_thread::yield()非成员函数

告知线程库，如果某线程调用了本函数，那么该线程在调用的那一刻并不需要运行。本函数通常用于计算量密集的循环中，避免过度消耗 CPU 时间。

声明：

```
void yield() noexcept;
```

作用：

让线程库有机会进行调度，替换当前线程。

抛出：

无。

3. std::this_thread::sleep_for()非成员函数

在指定的时长内暂停当前线程的执行。

声明：

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

作用：

阻塞当前线程，一直等待，直到超出 relative_time 所限定的时长。

注记：

线程实际阻塞的时间也许会比原本指定的更长。只要有可能，就应该由恒稳时钟判定已消耗的等待时间。

抛出：

无。

4. std::this_thread::sleep_until()非成员函数

阻塞当前线程，一直等待，直到超出限定的时长。

声明。

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

作用：

阻塞当前线程，直至在预设的时钟上，超出由参数 absolute_time 所限定的时长。

注记：

发起调用的线程的阻塞时间并不确定，只有当 Clock::now()的时刻等于/晚于 absolute_time 所指定的时刻，线程才会结束阻塞。

抛出：

无。