附录A 对C++11特性的简要介绍

新的C++标准，不仅带来了对并发的支持，也将其他语言的一些特性带入标准库中。本附录中，会给出对这些新特性进行简要介绍(这些特性用在线程库中)。除了thread_local(详见A.8部分)以外，就没有与并发直接相关的内容了，但对于多线程代码来说，它们都是很重要。我已只列出有必要的部分(例如，右值引用)，这样能够使代码更容易理解。由于对新特性不熟，对用到某些特性的代码理解起来会有一些困难。没关系，当对这些特性渐渐熟知后，就能很容易的理解代码。由于C++11的应用越来越广泛，这些特性在代码中的使用也将会变越来越普遍。

话不多说，让我们从线程库中的右值引用开始，来熟悉对象之间所有权(线程，锁等等)的转移。

A.1 右值引用

如果从事过C++编程，就会对引用比较熟悉，引用允许为已经存在的对象创建一个新的名字。对新引用所做的访问和修改操作，都会影响它的原型。

例如：

```
int var=42;
int& ref=var;  // 创建一个var的引用
ref=99;
assert(var==99);  // 原型的值被改变了，因为引用被赋值了
```

目前为止，我们用过的所有引用都是左值引用(C++11之前)——对左值的引用。lvalue指的是可以放在赋值表达式左边的事物——在栈上或堆上分配的命名对象，或者其他对象成员——有明确的内存地址。rvalue指的是可以出现在赋值表达式右侧的对象——例如，文字常量和临时变量。因此，左值引用只能被绑定在左值上，而不是右值。

不能这样写：

```
int& i=42;  // 编译失败
```

例如，因为42是一个右值。你可能通常使用下面的方式讲一个右值绑定到一个const左值引用上：

```
int const& i = 42;
```

这算是钻了标准的一个空子吧。不过，这种情况我们之前也介绍过，我们通过对左值的const引用创建临时性对象，作为参数传递给函数。其允许隐式转换，所以你可这样写：

```
void print(std::string const& s);
print("hello");  //创建了临时std::string对象
```

C++11标准介绍了*右值引用*(rvalue reference)，这种方式只能绑定右值，不能绑定左值，其通过两个 `&&` 来进行声明：

```cpp
int&& i=42;
int j=42;
int&& k=j;  // 编译失败
```

因此，可以使用函数重载的方式来确定：函数有左值或右值为参数的时候，看是否能被同名且对应参数为左值或右值引用的函数所重载。其基础就是C++11新添语义——*移动语义*(move semantics)。

A.1.1 移动语义

右值通常都是临时的，可以随意修改。如果知道函数的某个参数是一个右值，就可以将其看作为一个临时存储，不影响程序的正确性。比起拷贝右值参数的内容，不如移动其内容。动态数组比较大时，能节省很多内存，提供更多的优化空间。试想，一个函数以 `std::vector<int>` 作为一个参数，就需要将其拷贝进来，而不对原始的数据做任何操作。C++03/98的办法是，将这个参数作为一个左值的const引用传入，然后做内部拷贝：

```cpp
void process_copy(std::vector<int> const& vec_)
{
  std::vector<int> vec(vec_);
  vec.push_back(42);
}
```

这就允许函数能以左值或右值的形式进行传递，不过任何情况下都是通过拷贝来完成的。如果使用右值引用版本的函数来重载这个函数，就能避免在传入右值的时，进行内部拷贝，从而可以对原始值进行任意的修改：

```cpp
void process_copy(std::vector<int> && vec)
{
  vec.push_back(42);
}
```

如果这个问题存在于类的构造函数中，内部右值会在新的实例中使用。可以参考一下代码中的例子(默认构造函数会分配很大一块内存，在析构函数中释放)。

代码A.1 使用移动构造函数的类

```cpp
class X
{
private:
```

```
  int* data;

public:
  X():
    data(new int[1000000])
  {}

  ~X()
  {
    delete [] data;
  }

  X(const X& other):  // 1
   data(new int[1000000])
  {
    std::copy(other.data,other.data+1000000,data);
  }

  X(X&& other):  // 2
    data(other.data)
  {
    other.data=nullptr;
  }
};
```

一般情况下，拷贝构造函数①都是这么定义：分配一块新内存，然后将数据拷贝进去。不过，现在有了一个新的构造函数，可以接受右值引用来获取老数据②，就是移动构造函数。在这个例子中，只是将指针拷贝到数据中，将other以空指针的形式留在了新实例中。使用右值里创建变量，就能避免了空间和时间上的多余消耗。

X类(代码A.1)中的移动构造函数，仅作为一次优化。在其他例子中，有些类型的构造函数只支持移动构造函数，而不支持拷贝构造函数。例如，智能指针 `std::unique_ptr<>` 的非空实例中，只允许这个指针指向其对象，所以拷贝函数在这里就不能用了(如果使用拷贝函数，就会有两个 `std::unique_ptr<>` 指向该对象，不满足 `std::unique_ptr<>` 定义)。不过，移动构造函数允许对指针的所有权进行传递，并且允许 `std::unique_ptr<>` 像一个带有返回值的函数一样使用——指针的转移是通过移动，而非拷贝。

如果你已经知道，某个变量在之后就不会在用到了，这时候可以选择显式的移动，你可以使用 `static_cast<X&&>` 将对应变量转换为右值，或者通过调用 `std::move()` 函数来做这件事：

```
X x1;
X x2=std::move(x1);
X x3=static_cast<X&&>(x2);
```

想要将参数值不通过拷贝，转化为本地变量或成员变量时，就可以使用这个办法。虽然右值引用参数绑定了右值，不过在函数内部，会当做左值来进行处理：

```
void do_stuff(X&& x_)
{
  X a(x_);  // 拷贝
  X b(std::move(x_));  // 移动
}
do_stuff(X());  // ok，右值绑定到右值引用上
X x;
do_stuff(x);  // 错误，左值不能绑定到右值引用上
```

移动语义在线程库中用的比较广泛，无拷贝操作对数据进行转移可以作为一种优化方式，避免对将要被销毁的变量进行额外的拷贝。在2.2节中看到，在线程中使用 `std::move()` 转移 `std::unique_ptr<>` 得到一个新实例。在2.3节中，了解了 `std:thread` 实例可以使用移动语义来转移线程的所有权。

`std::thread` 、 `std::unique_lock<>` 、 `std::future<>` 、 `std::promise<>` 和 `std::packaged_task<>` 都不能拷贝，不过这些类都有移动构造函数，能让相关资源在实例中进行传递，并且支持用一个函数将其进行返回。 `std::string` 和 `std::vector<>` 也可以拷贝，不过它们也有移动构造函数和移动赋值操作符，就是为了避免拷贝拷贝大量数据。

C++标准库不会将一个对象显式的转移到另一个对象中，除非是将其销毁或赋值的时候(拷贝和移动的操作很相似)。不过，实践中移动能保证类中的所有状态保持不变，表现良好。一个 `std::thread` 实例可以作为移动源，转移到新(以默认构造方式) `std::thread` 实例中。 `std::string` 可以通过移动原始数据进行构造，并且保留原始数据的状态，不过不能保证的是原始数据中该状态是否正确(根据字符串长度或字符数量决定)。

A.1.2 右值引用和函数模板

使用右值引用作为函数模板的参数时，与之前的用法有些不同：如果函数模板参数以右值引用作为一个模板参数，当对应位置提供左值的时候，模板会自动将其类型认定为左值引用。当提供右值的时候，会当做普通数据使用。可能有些口语化，来看几个例子吧。

考虑一下下面的函数模板：

```
template<typename T>
void foo(T&& t)
{}
```

随后传入一个右值，T的类型将被推导为：

```
foo(42);  // foo<int>(42)
foo(3.14159);  // foo<double><3.14159>
foo(std::string());  // foo<std::string>(std::string())
```

不过，向foo传入左值的时候，T会被推导为一个左值引用：

```
int i = 42;
foo(i);  // foo<int&>(i)
```

因为函数参数声明为 `T&&` ，所以就是引用的引用，可以视为是原始的引用类型。那么foo<int&>()就相当于：

```
foo<int&>(); // void foo<int&>(int& t);
```

这就允许一个函数模板可以即接受左值，又可以接受右值参数。这种方式已在 `std::thread` 的构造函数所使用(2.1节和2.2节)，所以能够将可调用对象移动到内部存储，而非当参数是右值的时候再进行拷贝。

A.2 删除函数

有时让类去做拷贝是没有意义的。`std::mutex`就是一个例子——拷贝一个互斥量，意义何在？`std::unique_lock<>`是另一个例子——一个实例只能拥有一个锁。如果要复制，拷贝的那个实例也能获取相同的锁，这样`std::unique_lock<>`就没有存在的意义了。实例中转移所有权(A.1.2节)是有意义的，其并不是使用的拷贝。

为了避免拷贝操作，会将拷贝构造函数和拷贝赋值操作符声明为私有成员，并且不进行实现。如果对实例进行拷贝，将会引起编译错误。如果有其他成员函数或友元函数想要拷贝一个实例，将会引起链接错误(因为缺少实现)：

```
class no_copies
{
public:
  no_copies(){}
private:
  no_copies(no_copies const&);  // 无实现
  no_copies& operator=(no_copies const&);  // 无实现
};

no_copies a;
no_copies b(a);  // 编译错误
```

C++11中，委员会意识到这种情况。因此，委员会提供了更多的通用机制：可以通过添加`= delete`将一个函数声明为删除函数。

no_copise类就可以写为：

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

这样的描述要比之前的代码更加清晰。也允许编译器提供更多的错误信息描述，当成员函数想要执行拷贝操作的时候，可将链接错误转移到编译时。

拷贝构造和拷贝赋值操作删除后，需要显式写一个移动构造函数和移动赋值操作符，与 `std::thread` 和 `std::unique_lock<>` 一样，类是只移动的。

下面代码中的例子，就展示了一个只移动的类。

代码A.2 只移动类

```
class move_only
{
  std::unique_ptr<my_class> data;
public:
  move_only(const move_only&) = delete;
  move_only(move_only&& other):
    data(std::move(other.data))
  {}
  move_only& operator=(const move_only&) = delete;
  move_only& operator=(move_only&& other)
  {
    data=std::move(other.data);
    return *this;
  }
};

move_only m1;
move_only m2(m1);  // 错误，拷贝构造声明为“已删除”
move_only m3(std::move(m1));  // OK，找到移动构造函数
```

只移动对象可以作为函数的参数进行传递，并且从函数中返回，不过当想要移动左值，通常需要显式的使用 `std::move()` 或 `static_cast<T&&>` 。

可以为任意函数添加 `= delete` 说明符，添加后就说明这些函数不能使用。当然，还可以用于很多的地方。删除函数可以以正常的方式参与重载解析，并且如果使用，就会引起编译错误，这种方式可以用来删除特定的重载。比如，当函数以short作为参数，为了避免扩展为int类型，可以写出重载函数(以int为参数)的声明，然后添加删除说明符：

```
void foo(short);
void foo(int) = delete;
```

现在，任何向foo函数传递int类型参数都会产生一个编译错误，不过调用者可以显式的将其他类型转化为short：

```
foo(42);  // 错误，int重载声明已经删除
foo((short)42);  // OK
```

A.3 默认函数

删除函数的函数可以不进行实现，默认函数就则不同：编译器会创建函数实现，通常都是“默认”实现。当然，这些函数可以直接使用(它们都会自动生成)：默认构造函数，析构函数，拷贝构造函数，移动构造函数，拷贝赋值操作符和移动赋值操作符。

为什么要这样做呢？这里列出一些原因：

- 改变函数的可访问性——编译器生成的默认函数通常都是声明为public(如果想让其为protected或private成员，必须自己实现)。将其声明为默认，可以让编译器来帮助你实现函数和改变访问级别。
- 作为文档——编译器生成版本已经足够使用，那么显式声明就利于其他人阅读这段代码，会让代码结构看起来很清晰。
- 没有单独实现的时候，编译器自动生成函数——通常默认构造函数来做这件事，如果用户没有定义构造函数，编译器将会生成一个。当需要自定一个拷贝构造函数时(假设)，如果将其声明为默认，也可以获得编译器为你实现的拷贝构造函数。
- 编译器生成虚析构函数。
- 声明一个特殊版本的拷贝构造函数。比如：参数类型是非const引用，而不是const引用。
- 利用编译生成函数的特殊性质(如果提供了对应的函数，将不会自动生成对应函数——会在后面具体讲解)。

就像删除函数是在函数后面添加`= delete`一样，默认函数需要在函数后面添加`= default`，例如：

```
class Y
{
private:
  Y() = default;  // 改变访问级别
public:
  Y(Y&) = default;  // 以非const引用作为参数
  T& operator=(const Y&) = default; // 作为文档的形式，声明为默认函数
protected:
  virtual ~Y() = default;  // 改变访问级别，以及添加虚函数标签
};
```

编译器生成函数都有独特的特性，这是用户定义版本所不具备的。最大的区别就是编译器生成的函数都很简单。

列出了几点重要的特性：

- 对象具有简单的拷贝构造函数，拷贝赋值操作符和析构函数，都能通过memcpy或memmove进行拷贝。
- 字面类型用于constexpr函数(可见A.4节)，必须有简单的构造，拷贝构造和析构函数。
- 类的默认构造、拷贝、拷贝赋值操作符和析构函数，也可以用在一个已有构造和析构函数(用户定义)的联合体内。
- 类的简单拷贝赋值操作符可以使用 `std::atomic<>` 类型模板(见5.2.6节)，为某种类型的值提供原子操作。

仅添加 `=default` 不会让函数变得简单——如果类还支持其他相关标准的函数，那这个函数就是简单的——不过，用户显式的实现就不会让这些函数变简单。

第二个区别，编译器生成函数和用户提供的函数等价，也就是类中无用户提供的构造函数可以看作为一个aggregate，并且可以通过聚合初始化函数进行初始化：

```
struct aggregate
{
  aggregate() = default;
  aggregate(aggregate const&) = default;
  int a;
  double b;
};
aggregate x={42,3.141};
```

例子中，x.a被42初始化，x.b被3.141初始化。

第三个区别，编译器生成的函数只适用于构造函数；换句话说，只适用于符合某些标准的默认构造函数。

```
struct X
{
  int a;
};
```

如果创建了一个X的实例(未初始化)，其中int(a)将会被默认初始化。如果对象有静态存储过程，那么a将会被初始化为0；另外，当a没赋值的时候，其不定值可能会触发未定义行为：

```
X x1;  // x1.a的值不明确
```

另外，当使用显示调用构造函数的方式对X进行初始化，a就会被初始化为0：

```
X x2 = X();  // x2.a == 0
```

这种奇怪的属性会扩展到基础类和成员函数中。当类的默认构造函数是由编译器提供，并且一些数据成员和基类都是有编译器提供默认构造函数时，还有基类的数据成员和该类中的数据成员都是内置类型的时候，其值要不就是不确定的，要不就是初始化为0(与默认构造函数是否能被显式调用有关)。

虽然这条规则令人困惑，并且容易造成错误，不过也很有用。当你编写构造函数的时候，就不会用到这个特性。数据成员，通常都可以被初始化(指定了一个值或调用了显式构造函数)，或不会被初始化(因为不需要)：

```
X::X():a(){}  // a == 0
X::X():a(42){}  // a == 42
X::X(){}  // 1
```

第三个例子中①，省略了对a的初始化，X中a就是一个未被初始化的非静态实例，初始化的X实例都会有静态存储过程。

通常的情况下，如果写了其他构造函数，编译器就不会生成默认构造函数。所以，想要自己写一个的时候，就意味着你放弃了这种奇怪的初始化特性。不过，将构造函数显式声明成默认，就能强制编译器为你生成一个默认构造函数，并且刚才说的那种特性会保留：

```
X::X() = default;  // 应用默认初始化规则
```

这种特性用于原子变量(见5.2节)，默认构造函数显式为默认。初始值通常都没有定义，除非具有(a)一个静态存储的过程(静态初始化为0)，(b)显式调用默认构造函数，将成员初始化为0，(c)指定一个特殊的值。注意，这种情况下的原子变量，为允许静态初始化过程，构造函数会通过一个声明为constexpr(见A.4节)的值为原子变量进行初始化。

A.4 常量表达式函数

整型字面值，例如42，就是常量表达式。所以，简单的数学表达式，例如，23x2-4。可以使用其来初始化const整型变量，然后将const整型变量作为新表达的一部分：

```
const int i=23;
const int two_i=i*2;
const int four=4;
const int forty_two=two_i-four;
```

使用常量表达式创建变量也可用在其他常量表达式中，有些事只能用常量表达式去做：

- 指定数组长度：

```
int bounds=99;
int array[bounds];  // 错误，bounds不是一个常量表达式
const int bounds2=99;
int array2[bounds2];  // 正确，bounds2是一个常量表达式
```

- 指定非类型模板参数的值：

```
template<unsigned size>
struct test
{};
test<bounds> ia;  // 错误，bounds不是一个常量表达式
test<bounds2> ia2;  // 正确，bounds2是一个常量表达式
```

- 对类中static const整型成员变量进行初始化：

```
class X
{
  static const int the_answer=forty_two;
};
```

- 对内置类型进行初始化或可用于静态初始化集合：

```
struct my_aggregate
{
  int a;
  int b;
};
static my_aggregate ma1={forty_two,123};  // 静态初始化
int dummy=257;
static my_aggregate ma2={dummy,dummy};  // 动态初始化
```

- 静态初始化可以避免初始化顺序和条件变量的问题。

这些都不是新添加的——可以在1998版本的C++标准中找到对应上面实例的条款。不过，C++11标准中常量表达式进行了扩展，并添加了新的关键字——`constexpr`。C++14和C++17对constexpr做了进一步的扩展：其完整性超出了附录的介绍范围。

`constexpr`会对功能进行修改，当参数和函数返回类型符合要求，并且实现很简单，那么这样的函数就能够被声明为`constexpr`，这样函数可以当做常数表达式来使用：

```
constexpr int square(int x)
{
  return x*x;
}
int array[square(5)];
```

这个例子中，array有25个元素，因为square函数的声明为`constexpr`。当然，这种方式可以当做常数表达式来使用，不意味着什么情况下都是能够自动转换为常数表达式：

```
int dummy=4;
int array[square(dummy)];  // 错误，dummy不是常数表达式
```

dummy不是常数表达式，所以square(dummy)也不是——就是一个普通函数调用——所以其不能用来指定array的长度。

A.4.1 常量表达式和自定义类型

目前为止的例子都是以内置int型展开的。不过，在新C++标准库中，对于满足字面类型要求的任何类型，都可以用常量表达式来表示。

要想划分到字面类型中，需要满足一下几点：

- 一般的拷贝构造函数。
- 一般的析构函数。
- 所有成员变量都是非静态的，且基类需要是一般类型。
- 必须具有一个一般的默认构造函数，或一个constexpr构造函数。

后面会了解一下constexpr构造函数。

现在，先将注意力集中在默认构造函数上，就像下面代码中的CX类一样。

代码A.3 (一般)默认构造函数的类

```
class CX
{
private:
  int a;
  int b;
public:
  CX() = default;  // 1
  CX(int a_, int b_):  // 2
    a(a_),b(b_)
  {}
  int get_a() const
  {
    return a;
  }
  int get_b() const
  {
    return b;
  }
  int foo() const
  {
    return a+b;
  }
};
```

注意，这里显式的声明了默认构造函数①(见A.3节)，为了保存用户定义的构造函数②。因此，这种类型符合字面类型的要求，可以将其用在常量表达式中。

可以提供一个constexpr函数来创建一个实例，例如：

```
constexpr CX create_cx()
{
  return CX();
}
```

也可以创建一个简单的constexpr函数来拷贝参数：

```
constexpr CX clone(CX val)
{
  return val;
}
```

不过，constexpr函数只有其他constexpr函数可以进行调用。在C++14中，这个限制被解除，并且只要不修改任何非局部范围的对象，就可以在constexpr函数中完成所有的操作。在C++11中，CX类中声明成员函数和构造函数为constexpr：

```
class CX
{
private:
  int a;
  int b;
public:
  CX() = default;
  constexpr CX(int a_, int b_):
    a(a_),b(b_)
  {}
  constexpr int get_a() const  // 1
  {
    return a;
  }
  constexpr int get_b()  // 2
  {
    return b;
  }
  constexpr int foo()
  {
    return a+b;
  }
};
```

C++11中，const对于get_a()①来说就是多余的，因为在使用constexpr时就为const了，所以const描述符在这里会被忽略②。C++14中，这会发生变化(由于constexpr 函数的扩展能力)，因此get_b()无需再为隐式的const。

这就允许更多复杂的constexpr函数存在：

```
constexpr CX make_cx(int a)
{
  return CX(a,1);
}
constexpr CX half_double(CX old)
{
  return CX(old.get_a()/2,old.get_b()*2);
}
constexpr int foo_squared(CX val)
{
  return square(val.foo());
}
int array[foo_squared(half_double(make_cx(10)))];  // 49个元素
```

函数都很有趣，如果想要计算数组的长度或一个整型常量，就需要使用这种方式。最大的好处是常量表达式和constexpr函数会涉及到用户定义类型的对象，可以使用这些函数对这些对象进行初始化。因为常量表达式的初始化过程是静态初始化，所以能避免条件竞争和初始化顺序的问题：

```
CX si=half_double(CX(42,19));  // 静态初始化
```

当构造函数被声明为constexpr，且构造函数参数是常量表达式时，初始化过程就是常数初始化(可能作为静态初始化的一部分)。随着并发的发展，C++11标准中有一个重要的改变：允许用户定义构造函数进行静态初始化，就可以在初始化的时候避免条件竞争，因为静态过程能保证初始化过程在代码运行前进行。

特别是关于`std::mutex`(见3.2.1节)或`std::atomic<>`(见5.2.6节)，当想要使用一个全局实例来同步其他变量的访问时，同步访问就能避免条件竞争的发生。构造函数中，互斥量不可能产生条件竞争，因此对于`std::mutex`的默认构造函数应该被声明为constexpr，为了保证互斥量初始化过程是一个静态初始化过程的一部分。

A.4.2 常量表达式对象

目前，已经了解了constexpr在函数上的应用。constexpr也可以用在对象上，主要是用来做判断的。验证对象是否是使用常量表达式，constexpr构造函数或组合常量表达式进行初始化。

且这个对象需要声明为const：

```
constexpr int i=45;  // ok
constexpr std::string s(“hello”);  // 错误，std::string不是字面类型

int foo();
constexpr int j=foo();  // 错误，foo()没有声明为constexpr
```

A.4.3 常量表达式函数的要求

将一个函数声明为constexpr，也是有几点要求，当不满足这些要求，constexpr声明将会报编译错误。C++11中，对constexpr函数的要求如下：

- 所有参数都必须是字面类型。
- 返回类型必须是字面类型。
- 函数体内必须有一个return。
- return的表达式需要满足常量表达式的要求。
- 构造返回值/表达式的任何构造函数或转换操作，都需要是constexpr。

看起来很简单，要在内联函数中使用到常量表达式，返回的还是个常量表达式，还不能对任何东西进行改动。

constexpr函数就是无害的纯函数。

C++14中，要求减少了。尽管保留了无副作用纯功能概念，但允许包含更多：

- 允许使用多个return语句。
- 可以修改在函数中创建的对象。
- 允许使用循环、条件和switch语句。

constexpr类成员函数，需要追加几点要求：

- constexpr成员函数不能是虚函数。
- 对应类必须有字面类的成员。

constexpr构造函数的规则也有些不同：

- 构造函数体必须为空。
- 每一个基类必须可初始化。
- 每个非静态数据成员都需要初始化。
- 初始化列表的任何表达式，必须是常量表达式。
- 构造函数可选择要进行初始化的数据成员，并且基类必须有constexpr构造函数。

- 任何用于构建数据成员的构造函数和转换操作，以及和初始化表达式相关的基类必须为constexpr。

这些条件同样适用于成员函数，除非函数没有返回值，也就没有return语句。另外，构造函数对初始化列表中的所有基类和数据成员进行初始化，一般的拷贝构造函数会隐式的声明为constexpr。

A.4.4 常量表达式和模板

将constexpr应用于函数模板，或一个类模板的成员函数。根据参数，如果模板的返回类型不是字面类，编译器会忽略其常量表达式的声明。当模板参数类型合适，且为一般inline函数，就可以将类型写成constexpr类型的函数模板。

```
template<typename T>
constexpr T sum(T a,T b)
{
  return a+b;
}
constexpr int i=sum(3,42);  // ok, sum<int>是constexpr
std::string s=
  sum(std::string("hello"),
      std::string(" world"));  // 也行，不过sum<std::string>就不是constexpr了
```

函数需要满足所有constexpr函数所需的条件。不能用多个constexpr来声明一个函数，因为其是一个模板，所以也会带来一些编译错误。

A.5 Lambda函数

Lambda函数在C++11中的加入很是令人兴奋，因为Lambda函数能够大大简化代码复杂度(语法糖：利于理解具体的功能)，避免实现调用对象。C++11的Lambda函数语法允许在需要使用的时候进行定义。能为等待函数，例如`std::condition_variable`(如同4.1.1节中的例子)提供很好谓词函数，其语义可以用来快速的表示可访问的变量，而非使用类中函数来对成员变量进行捕获。

Lambda表达式就一个自给自足的函数，不需要传入函数仅依赖全局变量和函数，甚至都可以不用返回一个值。这样的Lambda表达式的一系列语义都需要封闭在括号中，还要以方括号作为前缀：

```
[]{  // Lambda表达式以[]开始
  do_stuff();
  do_more_stuff();
}();  // 表达式结束，可以直接调用
```

例子中，Lambda表达式通过后面的括号调用，不过这种方式不常用。一方面，如果想要直接调用，可以在写完对应的语句后就对函数进行调用。对于函数模板，传递一个参数进去时很常见的事情，甚至可以将可调用对象作为其参数传入。可调用对象通常也需要一些参数，或返回一个值，亦或两者都有。如果想给Lambda函数传递参数，可以参考下面的Lambda函数，其使用起来就像是一个普通函数。例如，下面代码是将vector中的元素使用`std::cout`进行打印：

```
std::vector<int> data=make_data();
std::for_each(data.begin(),data.end(),[](int i){std::cout<<i<<"\n";});
```

返回值也是很简单的，当Lambda函数体包括一个return语句，返回值的类型就作为Lambda表达式的返回类型。例如，使用一个简单的Lambda函数来等待`std::condition_variable`(见4.1.1节)中的标志被设置。

代码A.4 Lambda函数推导返回类型

```
std::condition_variable cond;
bool data_ready;
std::mutex m;
void wait_for_data()
{
  std::unique_lock<std::mutex> lk(m);
  cond.wait(lk,[]{return data_ready;});  // 1
}
```

Lambda的返回值传递给cond.wait()①，函数就能推断出data_ready的类型是bool。当条件变量从等待中苏醒后，上锁阶段会调用Lambda函数，并且当data_ready为true时，返回到wait()中。

当Lambda函数体中有多个return语句，就需要显式的指定返回类型。只有一个返回语句的时候，也可以这样做，不过这样可能会让你的Lambda函数体看起来更复杂。返回类型可以使用跟在参数列表后面的箭头(->)进行设置。如果Lambda函数没有任何参数，还需要包含(空)的参数列表，这样做是为了能显式的对返回类型进行指定。对条件变量的预测可以写成下面这种方式：

```
cond.wait(lk,[]()->bool{return data_ready;});
```

还可以对Lambda函数进行扩展，比如：加上log信息的打印，或做更加复杂的操作：

```
cond.wait(lk,[]()->bool{
  if(data_ready)
  {
    std::cout<<”Data ready”<<std::endl;
    return true;
  }
  else
  {
    std::cout<<”Data not ready, resuming wait”<<std::endl;
    return false;
  }
});
```

虽然简单的Lambda函数很强大，能简化代码，不过其真正的强大的地方在于对本地变量的捕获。

A.5.1 引用本地变量的Lambda函数

Lambda函数使用空的`[]`(Lambda introducer)就不能引用当前范围内的本地变量；其只能使用全局变量，或将其他值以参数的形式进行传递。当想要访问一个本地变量，需要对其进行捕获。最简单的方式就是将范围内的所有本地变量都进行捕获，使用`[=]`就可以完成这样的功能。函数被创建的时候，就能对本地变量的副本进行访问了。

实践一下：

```
std::function<int(int)> make_offseter(int offset)
{
  return [=](int j){return offset+j;};
}
```

当调用make_offseter时，就会通过`std::function<>`函数包装返回一个新的Lambda函数体。

这个带有返回的函数添加了对参数的偏移功能。例如：

```
int main()
{
  std::function<int(int)> offset_42=make_offseter(42);
  std::function<int(int)> offset_123=make_offseter(123);
  std::cout<<offset_42(12)<<”,“<<offset_123(12)<<std::endl;
  std::cout<<offset_42(12)<<”,“<<offset_123(12)<<std::endl;
}
```

屏幕上将打印出54,135两次，因为第一次从make_offseter中返回，都是对参数加42。第二次调用后，make_offseter会对参数加上123。所以，会打印两次相同的值。

这种本地变量捕获的方式相当安全，所有的东西都进行了拷贝，所以可以通过Lambda函数对表达式的值进行返回，并且可在原始函数之外的地方对其进行调用。这也不是唯一的选择，也可以选择通过引用的方式捕获本地变量。在本地变量被销毁的时候，Lambda函数会出现未定义的行为。

下面的例子，就介绍一下怎么使用`[&]`对所有本地变量进行引用：

```
int main()
{
  int offset=42;  // 1
  std::function<int(int)> offset_a=[&](int j){return offset+j;};  // 2
  offset=123;  // 3
  std::function<int(int)> offset_b=[&](int j){return offset+j;};  // 4
  std::cout<<offset_a(12)<<",”<<offset_b(12)<<std::endl;  // 5
  offset=99;  // 6
  std::cout<<offset_a(12)<<",”<<offset_b(12)<<std::endl;  // 7
}
```

之前的例子中，使用`[=]`来对要偏移的变量进行拷贝，offset_a函数就是个使用`[&]`捕获offset的引用的例子②。所以，offset初始化成42也没什么关系①；offset_a(12)的例子通常会依赖与当前offset的值。在③上，offset的值会变为123，offset_b④函数将会使用到这个值，同样第二个函数也是使用引用的方式。

现在，第一行打印信息⑤，offset为123，所以输出为135,135。不过，第二行打印信息⑦就有所不同，offset变成99⑥，所以输出为111,111。offset_a和offset_b都对当前值进行了加12的操作。

这些选项不会让你感觉到特别困惑，你可以选择以引用或拷贝的方式对变量进行捕获，并且还可以通过调整中括号中的表达式，来对特定的变量进行显式捕获。如果想要拷贝所有变量，可以使用`[=]`，通过参考中括号中的符号，对变量进行捕获。下面的例子将会打印出1239，因为i是拷贝进Lambda函数中的，而j和k是通过引用的方式进行捕获的：

```
int main()
{
  int i=1234,j=5678,k=9;
  std::function<int()> f=[=,&j,&k]{return i+j+k;};
  i=1;
  j=2;
  k=3;
  std::cout<<f()<<std::endl;
}
```

或者，也可以通过默认引用方式对一些变量做引用，而对一些特别的变量进行拷贝。这种情况下，就要使用`[&]`与拷贝符号相结合的方式对列表中的变量进行拷贝捕获。下面的例子将打印出5688，因为i通过引用捕获，但j和k通过拷贝捕获：

```
int main()
{
  int i=1234,j=5678,k=9;
  std::function<int()> f=[&,j,k]{return i+j+k;};
  i=1;
  j=2;
  k=3;
  std::cout<<f()<<std::endl;
}
```

如果只想捕获某些变量，可以忽略=或&，仅使用变量名进行捕获就行。加上&前缀，是将对应变量以引用的方式进行捕获，而非拷贝的方式。下面的例子将打印出5682，因为i和k是通过引用的范式获取的，而j是通过拷贝的方式：

```
int main()
{
  int i=1234,j=5678,k=9;
  std::function<int()> f=[&i,j,&k]{return i+j+k;};
  i=1;
  j=2;
  k=3;
  std::cout<<f()<<std::endl;
}
```

最后一种方式为了确保预期的变量能捕获。当在捕获列表中引用任何不存在的变量都会引起编译错误。当选择这种方式，就要小心类成员的访问方式，确定类中是否包含一个Lambda函数的成员变量。类成员变量不能直接捕获，如果想通过Lambda方式访问类中的成员，需要在捕获列表中添加this指针。下面的例子中，Lambda捕获this后，就能访问到some_data类中的成员：

```
struct X
{
  int some_data;
  void foo(std::vector<int>& vec)
  {
    std::for_each(vec.begin(),vec.end(),
         [this](int& i){i+=some_data;});
  }
};
```

并发的上下文中，Lambda是很有用的，其可以作为谓词放在`std::condition_variable::wait()`(见4.1.1节)和`std::packaged_task<>`(见4.2.1节)中，或是用在线程池中，对小任务进行打包。也可以线程函数的方式`std::thread`的构造函数(见2.1.1)，以及作为一个并行算法实现，在parallel_for_each()(见8.5.1节)中使用。

C++14后，Lambda表达式可以是真正通用Lamdba了，参数类型被声明为auto而不是指定类型。这种情况下，函数调用运算也是一个模板。当调用Lambda时，参数的类型可从提供的参数中推导出来，例如：

```
auto f=[](auto x){ std::cout<<”x=”<<x<<std::endl;};
f(42); // x is of type int; outputs “x=42”
f(“hello”); // x is of type const char*; outputs “x=hello”
```

C++14还添加了广义捕获的概念，因此可以捕获表达式的结果，而不是对局部变量的直接拷贝或引用。最常见的方法是通过移动只移动的类型来捕获类型，而不是通过引用来捕获，例如：

```
std::future<int> spawn_async_task(){
  std::promise<int> p;
  auto f=p.get_future();
  std::thread t([p=std::move(p)](){ p.set_value(find_the_answer());});
  t.detach();
  return f;
}
```

这里，promise通过p=std::move(p)捕获移到Lambda中，因此可以安全地分离线程，从而不用担心对局部变量的悬空引用。构建Lambda之后，p处于转移过来的状态，这就是为什么需要提前获得future的原因。

A.6 变参模板

变参模板：就是可以使用不定数量的参数进行特化的模板。就像你接触到的变参函数一样，printf就接受可变参数。现在，就可以给你的模板指定不定数量的参数了。变参模板在整个`C++`线程库中都有使用，例如：`std::thread`的构造函数就是一个变参类模板。从使用者的角度看，仅知道模板可以接受无限个参数就够了，不过当要写一个模板或对其工作原理很感兴趣就需要了解一些细节。

和变参函数一样，变参部分可以在参数列表章使用省略号`...`代表，变参模板需要在参数列表中使用省略号：

```
template<typename ... ParameterPack>
class my_template
{};
```

即使主模板不是变参模板，模板进行部分特化的类中，也可以使用可变参数模板。例如，`std::packaged_task<>`(见4.2.1节)的主模板就是一个简单的模板，这个简单的模板只有一个参数：

```
template<typename FunctionType>
class packaged_task;
```

不过，并不是所有地方都这样定义。对于部分特化模板来说像是一个“占位符”：

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

部分特化的类就包含实际定义的类。在第4章，可以写一个`std::packaged_task<int(std::string,double)>`来声明一个以`std::string`和double作为参数的任务，当执行这个任务后结果会由`std::future<int>`进行保存。

声明展示了两个变参模板的附加特性。第一个比较简单：普通模板参数(例如ReturnType)和可变模板参数(Args)可以同时声明。第二个特性，展示了`Args...`特化类的模板参数列表中如何使用，为了展示实例化模板中的Args的组成类型。实际上，因为是部分特化，所以其作为一种模式进行匹配。在列表中出现的类型(被Args捕获)都会进行实例化。参数包(parameter pack)调用可变参数Args，并且使用`Args...`作为包的扩展。

和可变参函数一样，变参部分可能什么都没有，也可能有很多类型项。例如，`std::packaged_task<my_class()>`中ReturnType参数就是my_class，并且Args参数包是空的，不过`std::packaged_task<void(int,double,my_class&,std::string*)>`中，ReturnType为void，并且Args列表中的类型就有：int, double, my_class&和std::string*。

A.6.1 扩展参数包

变参模板主要依赖扩展功能，因为不能限制有更多的类型添加到模板参数中。首先，列表中的参数类型使用到的时候，可以使用包扩展，比如：需要给其他模板提供类型参数。

```
template<typename ... Params>
struct dummy
{
  std::tuple<Params...> data;
};
```

成员变量data是一个`std::tuple<>`实例，包含所有指定类型，所以dummy<int, double, char>的成员变量就为`std::tuple<int, double, char>`。可以将包扩展和普通类型相结合：

```
template<typename ... Params>
struct dummy2
{
  std::tuple<std::string,Params...> data;
};
```

这次，元组中添加了额外的(第一个)成员类型`std::string`。其优雅之处在于，可以通过包扩展的方式创建一种模式，这种模式会在之后将每个元素拷贝到扩展之中，可以使用`...`来表示扩展模式的结束。例如，创建使用参数包来创建元组中所有的元素，不如在元组中创建指针，或使用`std::unique_ptr<>`指针，指向对应元素：

```
template<typename ... Params>
struct dummy3
{
  std::tuple<Params* ...> pointers;
  std::tuple<std::unique_ptr<Params> ...> unique_pointers;
};
```

类型表达式会比较复杂，提供的参数包是在类型表达式中产生，并且表达式中使用 `...` 作为扩展。当参数包已经扩展，包中的每一项都会代替对应的类型表达式，在结果列表中产生相应的数据项。因此，当参数包Params包含int，int，char类型，那么 `std::tuple<std::pair<std::unique_ptr<Params>,double> ... >` 将扩展为 `std::tuple<std::pair<std::unique_ptr<int>,double>`，`std::pair<std::unique_ptr<int>,double>`，`std::pair<std::unique_ptr<char>,double> >`。如果包扩展当做模板参数列表使用时，模板就不需要变长的参数了。如果不需要了，参数包就要对模板参数的要求进行准确的匹配：

```
template<typename ... Types>
struct dummy4
{
  std::pair<Types...> data;
};
dummy4<int,char> a;  // 1 ok，为std::pair<int, char>
dummy4<int> b;  // 2 错误，无第二个类型
dummy4<int,int,int> c;  // 3 错误，类型太多
```

可以使用包扩展的方式，对函数的参数进行声明：

```
template<typename ... Args>
void foo(Args ... args);
```

这将会创建一个新参数包args，其是一组函数参数，而非一组类型，并且这里 `...` 也能像之前一样进行扩展。例如，可以在 `std::thread` 的构造函数中使用，使用右值引用的方式获取函数所有的参数(见A.1节)：

```
template<typename CallableType,typename ... Args>
thread::thread(CallableType&& func,Args&& ... args);
```

函数参数包也可以用来调用其他函数，将制定包扩展成参数列表，匹配调用的函数。如同类型扩展一样，也可以使用某种模式对参数列表进行扩展。例如，使用 `std::forward()` 以右值引用的方式来保存提供给函数的参数：

```
template<typename ... ArgTypes>
void bar(ArgTypes&& ... args)
{
  foo(std::forward<ArgTypes>(args)...);
}
```

注意一下这个例子，包扩展包括对类型包ArgTypes和函数参数包args的扩展，并且省略了其余的表达式。当这样调用bar函数：

```
int i;
bar(i,3.141,std::string("hello "));
```

将会扩展为

```
template<>
void bar<int&,double,std::string>(
         int& args_1,
         double&& args_2,
         std::string&& args_3)
{
  foo(std::forward<int&>(args_1),
      std::forward<double>(args_2),
      std::forward<std::string>(args_3));
}
```

这样就将第一个参数以左值引用的形式，正确的传递给了foo函数，其他两个函数都是以右值引用的方式传入的。

最后一件事，参数包中使用 `sizeof...` 操作可以获取类型参数类型的大小， `sizeof...(p)` 就是p参数包中所包含元素的个数。不管是类型参数包或函数参数包，结果都是一样的。这可能是唯一一次在使用参数包的时候，没有加省略号；这里的省略号是作为 `sizeof...` 操作的一部分，所以不算是用到省略号。

下面的函数会返回参数的数量：

```
template<typename ... Args>
unsigned count_args(Args ... args)
{
  return sizeof... (Args);
}
```

就像普通的sizeof操作一样， `sizeof...` 的结果为常量表达式，所以其可以用来指定定义数组长度，等等。

A.8 线程本地变量

线程本地变量允许程序中的每个线程都有一个独立的实例拷贝，可以使用`thread_local`关键字来对这样的变量进行声明。命名空间内的变量，静态成员变量，以及本地变量都可以声明成线程本地变量，为了在线程运行前对这些数据进行存储操作：

```
thread_local int x;  // 命名空间内的线程本地变量

class X
{
  static thread_local std::string s;  // 线程本地的静态成员变量
};
static thread_local std::string X::s;  // 这里需要添加X::s

void foo()
{
  thread_local std::vector<int> v;  // 一般线程本地变量
}
```

由命名空间或静态数据成员构成的线程本地变量，需要在线程单元对其进行使用**前**进行构建。有些实现中，会将对线程本地变量的初始化过程放在线程中去做，还有一些可能会在其他时间点做初始化。在一些有依赖的组合中，会根据具体情况来进行决定。将没有构造好的线程本地变量传递给线程单元使用，不能保证它们会在线程中进行构造。这样就可以动态加载带有线程本地变量的模块——变量首先需要在一个给定的线程中进行构造，之后其他线程就可以通过动态加载模块对线程本地变量进行引用。

函数中声明的线程本地变量，需要使用一个给定线程进行初始化(通过第一波控制流将这些声明传递给指定线程)。如果指定线程调用没有调用函数，那么这个函数中声明的线程本地变量就不会构造。本地静态变量也是同样的情况，除非其单独的应用于每一个线程。

静态变量与线程本地变量会共享一些属性——它们可以做进一步的初始化(比如，动态初始化)。如果在构造线程本地变量时抛出异常，`srd::terminate()`就会将程序终止。

析构函数会在构造线程本地变量的那个线程返回时调用，析构顺序是构造的逆序。当初始化顺序没有指定时，确定析构函数和这些变量是否有相互依存关系就尤为重要了。当线程本地变量的析构函数抛出异常时，`std::terminate()`会被调用，将程序终止。

当线程调用 `std::exit()` 或从main()函数返回(等价于调用 `std::exit()` 作为main()的“返回值”)时，线程本地变量也会为了这个线程进行销毁。应用退出时还有线程在运行，对于这些线程来说，线程本地变量的析构函数就没有被调用。

虽然，线程本地变量在不同线程上有不同的地址，不过还是可以获取指向这些变量的一般指针。指针会在线程中，通过获取地址的方式，引用相应的对象。当引用被销毁的对象时，会出现未定义行为，所以在向其他线程传递线程本地变量指针时，就需要保证指向对象所在的线程结束后，不对相应的指针进行解引用。

A.9 模板类参数的推导

C++17扩展了模板参数自动推断类型的思想：如果声明模板化的对象，在很多情况下，模板参数的类型可以从对象初始化器中推导出来。

具体来说，如果使用类模板来声明对象，而不指定模板参数列表，则根据函数模板的常规类型推导规则，使用类模板中指定的构造函数从对象的初始值设定项中推导模板参数。

例如，互斥量模板类型以std::lock_guard作为一个模板参数。构造函数还接受单个参数，该参数是对该类型的引用。如果将对象声明为std::lock_guard类型，则可以从提供的互斥体类型推断类型参数：

```
std::mutex m;
std::lock_guard guard(m); // 推断出std::lock_guard<std::mutex>
```

同样适用于std::scoped_lock，虽然它有多个模板参数，但也能从多个互斥参数推导得出：

```
std::mutex m1;
std::shared_mutex m2;
std::scoped_lock guard(m1,m2);
// 推断出std::scoped_lock<std::mutex,std::shared_mutex>
```

对于那些可能推导错误类型的模板，模板作者可以编写显式的推导指南，以确保导出正确的类型。不过，这些都超出了本书的范畴。

A.10 本章总结

本附录仅是摘录了部分C++11标准的新特性，因为这些特性和线程库之间有着良好的互动。其他的新特性，包括：静态断言(static_assert)，强类型枚举(enum class)，委托构造函数，Unicode码支持，模板别名，以及统一的初始化序列。对于新功能的详细描述已经超出了本书的范围，需要另外一本书来进行详细介绍。C++14和C++17中也有很多的变化，但这些都超出了本书的范围。对标准改动的最好的概述可能就是由Bjarne Stroustrup编写的《C++11FAQ》[1], 其他C++的参考书籍也会在未来对C++11标准进行覆盖。

希望这里的简短介绍，能让你了解这些新功能和线程库之间的关系，并且在写多线程代码的时候能用到这些新功能。虽然，本附录只为新特性提供了足够简单的例子，并非新功能的一份完整的参考或教程。如果想在你的代码中大量使用这些新功能，我建议去找相关权威的参考书或教程，了解更加详细的情况。

---

[1] http://www.research.att.com/~bs/C++0xFAQ.html

附录B 并发库的简单比较

虽然，C++11才开始正式支持并发，不过高级编程语言都支持并发和多线程已经不是什么新鲜事了。例如，Java在第一个发布版本中就支持多线程编程，在某些平台上也提供符合POSIX C标准的多线程接口，还有Erlang支持消息的同步传递(有点类似于MPI)。当然还有使用C++类的库，比如Boost，其将底层多线程接口进行包装，适用于任何给定的平台(不论是使用POSIX C的接口，或其他接口)，其对支持的平台会提供可移植接口。

这些库或者编程语言，已经写了很多多线程应用，并且在使用这些库写多线程代码的经验，可以借鉴到C++中，本附录就对Java，POSIX C，使用Boost线程库的C++，以及C++11中的多线程工具进行简单的比较，当然也会交叉引用本书的相关章节。

<table>
<tr><th>特性</th><th>启动线程</th><th>互斥量</th><th>监视/等待谓词</th><th>原子操作和并发感知内存模型</th><th>线程安全容器</th><th>Futures(期望)</th><th>线程池</th><th>线程中断</th></tr>
<tr><td>章节引用</td><td>第2章</td><td>第3章</td><td>第4章</td><td>第5章</td><td>第6章和第7章</td><td>第4章</td><td>第9章</td><td>第9章</td></tr>
<tr><td rowspan="3">C++11</td><td rowspan="3">std::thread和其成员函数</td><td>std::mutex类和其成员函数</td><td>std::condition_variable</td><td>std::atomic_xxx类型</td><td rowspan="3">N/A</td><td>std::future<></td><td rowspan="3">N/A</td><td rowspan="3">N/A</td></tr>
<tr><td>std::lock_guard<>模板</td><td rowspan="2">std::condition_variable_any类和其成员函数</td><td>std::atomic<>类模板</td><td>std::shared_future<></td></tr>
<tr><td>std::unique_lock<>模板</td><td>std::atomic_thread_fence()函数</td><td>std::atomic_future<>类模板</td></tr>
</table>

Boost线程库	boost::thread类和成员函数	boost::mutex类和其成员函数	boost::condition_variable类和其成员函数	N/A	N/A	boost::unique_future<>类模板	N/A	boost::thread类的interrupt()成员函数
		boost::lock_guard<>类模板	boost::condition_variable_any类和其成员函数			boost::shared_future<>类模板		
		boost::unique_lock<>类模板						
POSIX C	pthread_t类型相关的API函数	pthread_mutex_t类型相关的API函数	pthread_cond_t类型相关的API函数	N/A	N/A	N/A	N/A	pthread_cancel()
	pthread_create()	pthread_mutex_lock()	pthread_cond_wait()					
	pthread_detach()	pthread_mutex_unlock()	pthread_cond_timed_wait()					
	pthread_join()	等等	等等					

Java	java.lang.thread类	synchronized块	java.lang.Object类的wait()和notify()函数，用在内部synchronized块中	java.util.concurrent.atomic包中的volatile类型变量	java.util.concurrent包中的容器	与java.util.concurrent.future接口相关的类	java.util.concurrent.ThreadPoolExecutor类	java.lang.Thread类的interrupt()函数

消息传递框架与完整的ATM示例

ATM：自动取款机。

回到第4章，我举了一个使用消息传递框架在线程间发送信息的例子。这里就会使用这个实现来完成ATM功能。下面完整代码就是功能的实现，包括消息传递框架。

代码C.1实现了一个消息队列，可以将消息以指针(指向基类)的方式存储在列表中，指定消息类型会由基类派生模板进行处理。推送包装类的构造实例，以及存储指向这个实例的指针，弹出实例的时候，将会返回指向其的指针。因为message_base类没有任何成员函数，在访问存储消息之前，弹出线程就需要将指针转为 `wrapped_message<T>` 指针。

代码C.1 简单的消息队列

```
#include <mutex>
#include <condition_variable>
#include <queue>
#include <memory>

namespace messaging
{
  struct message_base  // 队列项的基础类
  {
    virtual ~message_base()
    {}
  };

  template<typename Msg>
  struct wrapped_message:  // 每个消息类型都需要特化
    message_base
  {
    Msg contents;

    explicit wrapped_message(Msg const& contents_):
      contents(contents_)
    {}
  };

  class queue  // 我们的队列
  {
    std::mutex m;
    std::condition_variable c;
```

```
    std::queue<std::shared_ptr<message_base> > q;  // 实际存储指向message_base类指针的队列
  public:
    template<typename T>
    void push(T const& msg)
    {
      std::lock_guard<std::mutex> lk(m);
      q.push(std::make_shared<wrapped_message<T> >(msg));  // 包装已传递的信息，存储指针
      c.notify_all();
    }

    std::shared_ptr<message_base> wait_and_pop()
    {
      std::unique_lock<std::mutex> lk(m);
      c.wait(lk,[&]{return !q.empty();});  // 当队列为空时阻塞
      auto res=q.front();
      q.pop();
      return res;
    }
  };
}
```

发送通过sender类(见代码C.2)实例处理过的消息。只能对已推送到队列中的消息进行包装。对sender实例的拷贝，只是拷贝了指向队列的指针，而非队列本身。

代码C.2 sender类

```
namespace messaging
{
  class sender
  {
    queue*q;  // sender是一个队列指针的包装类
  public:
    sender():  // sender无队列(默认构造函数)
      q(nullptr)
    {}

    explicit sender(queue*q_):  // 从指向队列的指针进行构造
      q(q_)
    {}

    template<typename Message>
    void send(Message const& msg)
    {
      if(q)
```

```
      {
        q->push(msg);  // 将发送信息推送给队列
      }
    }
  };
}
```

接收信息部分有些麻烦。不仅要等待队列中的消息，还要检查消息类型是否与所等待的消息类型匹配，并调用处理函数进行处理。那么就从receiver类的实现开始吧。

代码C.3 receiver类

```
namespace messaging
{
  class receiver
  {
    queue q;  // 接受者拥有对应队列
  public:
    operator sender()  // 允许将类中队列隐式转化为一个sender队列
    {
      return sender(&q);
    }
    dispatcher wait()  // 等待对队列进行调度
    {
      return dispatcher(&q);
    }
  };
}
```

sender只是引用一个消息队列，而receiver是拥有一个队列。可以使用隐式转换的方式获取sender引用的类。难点在于wait()中的调度，这里创建了一个dispatcher对象引用receiver中的队列，dispatcher类实现会在下一个清单中看到。如你所见，任务是在析构函数中完成的。在这个例子中，所要做的工作是对消息进行等待，以及对其进行调度。

代码C.4 dispatcher类

```
namespace messaging
{
  class close_queue  // 用于关闭队列的消息
  {};

  class dispatcher
  {
    queue* q;
```

```
    bool chained;

    dispatcher(dispatcher const&)=delete;  // dispatcher实例不能被拷贝
    dispatcher& operator=(dispatcher const&)=delete;

    template<
      typename Dispatcher,
      typename Msg,
      typename Func>  // 允许TemplateDispatcher实例访问内部成员
    friend class TemplateDispatcher;

    void wait_and_dispatch()
    {
      for(;;)  // 1 循环，等待调度消息
      {
        auto msg=q->wait_and_pop();
        dispatch(msg);
      }
    }

    bool dispatch(  // 2 dispatch()会检查close_queue消息，然后抛出
      std::shared_ptr<message_base> const& msg)
    {
      if(dynamic_cast<wrapped_message<close_queue>*>(msg.get()))
      {
        throw close_queue();
      }
      return false;
    }
  public:
    dispatcher(dispatcher&& other):  // dispatcher实例可以移动
      q(other.q),chained(other.chained)
    {
      other.chained=true;  // 源不能等待消息
    }

    explicit dispatcher(queue* q_):
      q(q_),chained(false)
    {}

    template<typename Message,typename Func>
    TemplateDispatcher<dispatcher,Message,Func>
    handle(Func&& f)  // 3 使用TemplateDispatcher处理指定类型的消息
    {
      return TemplateDispatcher<dispatcher,Message,Func>(
        q,this,std::forward<Func>(f));
    }
```

```
56
57      ~dispatcher() noexcept(false)  // 4 析构函数可能会抛出异常
58      {
59        if(!chained)
60        {
61          wait_and_dispatch();
62        }
63      }
64    };
65  }
```

从wait()返回的dispatcher实例将马上被销毁，因为是临时变量，也如前文提到的，析构函数在这里做真正的工作。析构函数调用wait_and_dispatch()函数，这个函数中有一个循环①，等待消息的传入(这样才能进行弹出操作)，然后将消息传递给dispatch()函数。dispatch()函数本身②很简单，会检查小时是否是一个close_queue消息，当是close_queue消息时抛出一个异常。如果不是，函数将会返回false来表明消息没有被处理。因为会抛出close_queue异常，所以析构函数会标示为 `noexcept(false)` 。在没有任何标识的情况下，析构函数都为 `noexcept(true)` ④型，这表示没有任何异常抛出，并且close_queue异常将会使程序终止。

虽然不会经常的去调用wait()函数，但在大多数时间里，都希望对一条消息进行处理。这时就需要handle()成员函数③的加入。这个函数是一个模板，并且消息类型不可推断，所以需要指定需要处理的消息类型，并且传入函数(或可调用对象)进行处理，并将队列传入当前dispatcher对象的handle()函数。在测试析构函数中的chained值前要等待消息，不仅是避免“移动”类型的对象对消息进行等待，而且允许将等待状态转移到新的TemplateDispatcher实例中。

代码C.5 TemplateDispatcher类模板

```
1   namespace messaging
2   {
3     template<typename PreviousDispatcher,typename Msg,typename Func>
4     class TemplateDispatcher
5     {
6       queue* q;
7       PreviousDispatcher* prev;
8       Func f;
9       bool chained;
10
11      TemplateDispatcher(TemplateDispatcher const&)=delete;
12      TemplateDispatcher& operator=(TemplateDispatcher const&)=delete;
13
```

```
14      template<typename Dispatcher,typename OtherMsg,typename OtherFunc>
15      friend class TemplateDispatcher;  // 所有特化的TemplateDispatcher类型实例都是友元类
16
17      void wait_and_dispatch()
18      {
19        for(;;)
20        {
21          auto msg=q->wait_and_pop();
22          if(dispatch(msg))  // 1 如果消息处理过后，会跳出循环
23            break;
24        }
25      }
26
27      bool dispatch(std::shared_ptr<message_base> const& msg)
28      {
29        if(wrapped_message<Msg>* wrapper=
30           dynamic_cast<wrapped_message<Msg>*>(msg.get()))  // 2 检查消息类型，并且调用函
    数
31        {
32          f(wrapper->contents);
33          return true;
34        }
35        else
36        {
37          return prev->dispatch(msg);  // 3 链接到之前的调度器上
38        }
39      }
40    public:
41      TemplateDispatcher(TemplateDispatcher&& other):
42          q(other.q),prev(other.prev),f(std::move(other.f)),
43          chained(other.chained)
44      {
45        other.chained=true;
46      }
47      TemplateDispatcher(queue* q_,PreviousDispatcher* prev_,Func&& f_):
48          q(q_),prev(prev_),f(std::forward<Func>(f_)),chained(false)
49      {
50        prev_->chained=true;
51      }
52
53      template<typename OtherMsg,typename OtherFunc>
54      TemplateDispatcher<TemplateDispatcher,OtherMsg,OtherFunc>
55      handle(OtherFunc&& of)  // 4 可以链接其他处理器
56      {
57        return TemplateDispatcher<
58            TemplateDispatcher,OtherMsg,OtherFunc>(
59            q,this,std::forward<OtherFunc>(of));
```

```
    }

    ~TemplateDispatcher() noexcept(false)  // 5 这个析构函数也是noexcept(false)的
    {
      if(!chained)
      {
        wait_and_dispatch();
      }
    }
  };
}
```

`TemplateDispatcher<>`类模板仿照了dispatcher类，二者几乎相同。特别是析构函数上，都是调用wait_and_dispatch()来等待处理消息。

处理消息的过程中，如果不抛出异常，就需要检查一下在循环中①消息是否已经得到了处理。当成功的处理了一条消息，处理过程就可以停止了，这样就可以等待下一组消息的传入了。当获取了一个和指定类型匹配的消息，使用函数调用的方式②就要好于抛出异常(处理函数也可能会抛出异常)。如果消息类型不匹配，就可以链接前一个调度器③。在第一个实例中，dispatcher实例确实作为一个调度器，当在handle()④函数中进行链接后，就允许处理多种类型的消息。在链接了之前的`TemplateDispatcher<>`实例后，当消息类型和当前的调度器类型不匹配的时候，调度链会依次的向前寻找类型匹配的调度器。因为任何调度器都可能抛出异常(包括dispatcher中对close_queue消息进行处理的默认处理器)，析构函数在这里会再次被声明为`noexcept(false)`⑤。

这种简单的架构允许你想队列推送任何类型的消息，并且调度器有选择的与接收端的消息进行匹配。同样，也允许为了推送消息，将消息队列的引用进行传递的同时，保持接收端的私有性。

为了完成第4章的例子，消息的组成将在清单C.6中给出，各种状态机将在代码C.7,C.8和C.9中给出。最后，驱动代码将在C.10给出。

代码C.6 ATM消息

```
struct withdraw
{
  std::string account;
  unsigned amount;
  mutable messaging::sender atm_queue;

  withdraw(std::string const& account_,
           unsigned amount_,
```

```
            messaging::sender atm_queue_):
    account(account_),amount(amount_),
    atm_queue(atm_queue_)
  {}
};

struct withdraw_ok
{};

struct withdraw_denied
{};

struct cancel_withdrawal
{
  std::string account;
  unsigned amount;
  cancel_withdrawal(std::string const& account_,
                    unsigned amount_):
    account(account_),amount(amount_)
  {}
};

struct withdrawal_processed
{
  std::string account;
  unsigned amount;
  withdrawal_processed(std::string const& account_,
                       unsigned amount_):
    account(account_),amount(amount_)
  {}
};

struct card_inserted
{
  std::string account;
  explicit card_inserted(std::string const& account_):
    account(account_)
  {}
};

struct digit_pressed
{
  char digit;
  explicit digit_pressed(char digit_):
    digit(digit_)
  {}
};
```

```
56
57  struct clear_last_pressed
58  {};
59
60  struct eject_card
61  {};
62
63  struct withdraw_pressed
64  {
65    unsigned amount;
66    explicit withdraw_pressed(unsigned amount_):
67      amount(amount_)
68    {}
69  };
70
71  struct cancel_pressed
72  {};
73
74  struct issue_money
75  {
76    unsigned amount;
77    issue_money(unsigned amount_):
78      amount(amount_)
79    {}
80  };
81
82  struct verify_pin
83  {
84    std::string account;
85    std::string pin;
86    mutable messaging::sender atm_queue;
87
88    verify_pin(std::string const& account_,std::string const& pin_,
89               messaging::sender atm_queue_):
90      account(account_),pin(pin_),atm_queue(atm_queue_)
91    {}
92  };
93
94  struct pin_verified
95  {};
96
97  struct pin_incorrect
98  {};
99
100 struct display_enter_pin
101 {};
102
```

```
struct display_enter_card
{};

struct display_insufficient_funds
{};

struct display_withdrawal_cancelled
{};

struct display_pin_incorrect_message
{};

struct display_withdrawal_options
{};

struct get_balance
{
  std::string account;
  mutable messaging::sender atm_queue;

  get_balance(std::string const& account_,messaging::sender atm_queue_):
    account(account_),atm_queue(atm_queue_)
  {}
};

struct balance
{
  unsigned amount;
  explicit balance(unsigned amount_):
    amount(amount_)
  {}
};

struct display_balance
{
  unsigned amount;
  explicit display_balance(unsigned amount_):
    amount(amount_)
  {}
};

struct balance_pressed
{};
```

代码C.7 ATM状态机

```
class atm
{
  messaging::receiver incoming;
  messaging::sender bank;
  messaging::sender interface_hardware;

  void (atm::*state)();

  std::string account;
  unsigned withdrawal_amount;
  std::string pin;

  void process_withdrawal()
  {
    incoming.wait()
      .handle<withdraw_ok>(
       [&](withdraw_ok const& msg)
       {
         interface_hardware.send(
           issue_money(withdrawal_amount));

         bank.send(
           withdrawal_processed(account,withdrawal_amount));

         state=&atm::done_processing;
       })
      .handle<withdraw_denied>(
       [&](withdraw_denied const& msg)
       {
         interface_hardware.send(display_insufficient_funds());

         state=&atm::done_processing;
       })
      .handle<cancel_pressed>(
       [&](cancel_pressed const& msg)
       {
         bank.send(
           cancel_withdrawal(account,withdrawal_amount));

         interface_hardware.send(
           display_withdrawal_cancelled());

         state=&atm::done_processing;
       });
   }

```

```
47    void process_balance()
48    {
49      incoming.wait()
50        .handle<balance>(
51         [&](balance const& msg)
52         {
53           interface_hardware.send(display_balance(msg.amount));
54
55           state=&atm::wait_for_action;
56         })
57        .handle<cancel_pressed>(
58         [&](cancel_pressed const& msg)
59         {
60           state=&atm::done_processing;
61         });
62    }
63
64    void wait_for_action()
65    {
66      interface_hardware.send(display_withdrawal_options());
67
68      incoming.wait()
69        .handle<withdraw_pressed>(
70         [&](withdraw_pressed const& msg)
71         {
72           withdrawal_amount=msg.amount;
73           bank.send(withdraw(account,msg.amount,incoming));
74           state=&atm::process_withdrawal;
75         })
76        .handle<balance_pressed>(
77         [&](balance_pressed const& msg)
78         {
79           bank.send(get_balance(account,incoming));
80           state=&atm::process_balance;
81         })
82        .handle<cancel_pressed>(
83         [&](cancel_pressed const& msg)
84         {
85           state=&atm::done_processing;
86         });
87    }
88
89    void verifying_pin()
90    {
91      incoming.wait()
92        .handle<pin_verified>(
93         [&](pin_verified const& msg)
```

```
        {
          state=&atm::wait_for_action;
        })
       .handle<pin_incorrect>(
        [&](pin_incorrect const& msg)
        {
          interface_hardware.send(
          display_pin_incorrect_message());
          state=&atm::done_processing;
        })
       .handle<cancel_pressed>(
        [&](cancel_pressed const& msg)
        {
          state=&atm::done_processing;
        });
    }

    void getting_pin()
    {
      incoming.wait()
        .handle<digit_pressed>(
        [&](digit_pressed const& msg)
        {
          unsigned const pin_length=4;
          pin+=msg.digit;

          if(pin.length()==pin_length)
          {
            bank.send(verify_pin(account,pin,incoming));
            state=&atm::verifying_pin;
          }
        })
       .handle<clear_last_pressed>(
        [&](clear_last_pressed const& msg)
        {
          if(!pin.empty())
          {
            pin.pop_back();
          }
        })
       .handle<cancel_pressed>(
        [&](cancel_pressed const& msg)
        {
          state=&atm::done_processing;
        });
    }

```

```
  void waiting_for_card()
  {
    interface_hardware.send(display_enter_card());

    incoming.wait()
      .handle<card_inserted>(
       [&](card_inserted const& msg)
       {
         account=msg.account;
         pin="";
         interface_hardware.send(display_enter_pin());
         state=&atm::getting_pin;
       });
  }

  void done_processing()
  {
    interface_hardware.send(eject_card());
    state=&atm::waiting_for_card;
  }

  atm(atm const&)=delete;
  atm& operator=(atm const&)=delete;
public:
  atm(messaging::sender bank_,
  messaging::sender interface_hardware_):
  bank(bank_),interface_hardware(interface_hardware_)
  {}

  void done()
  {
    get_sender().send(messaging::close_queue());
  }

  void run()
  {
    state=&atm::waiting_for_card;
    try
    {
      for(;;)
      {
        (this->*state)();
      }
    }
    catch(messaging::close_queue const&)
    {
    }
```

```
  }

  messaging::sender get_sender()
  {
    return incoming;
  }
};
```

代码C.8 银行状态机

```
class bank_machine
{
  messaging::receiver incoming;
  unsigned balance;
public:
  bank_machine():

  balance(199)
  {}

  void done()
  {
    get_sender().send(messaging::close_queue());
  }

  void run()
  {
    try
    {
      for(;;)
      {
        incoming.wait()
          .handle<verify_pin>(
           [&](verify_pin const& msg)
           {
             if(msg.pin=="1937")
             {
               msg.atm_queue.send(pin_verified());
             }
             else
             {
               msg.atm_queue.send(pin_incorrect());
             }
           })
          .handle<withdraw>(
           [&](withdraw const& msg)
```

```
            {
              if(balance>=msg.amount)
              {
                msg.atm_queue.send(withdraw_ok());
                balance-=msg.amount;
              }
              else
              {
                msg.atm_queue.send(withdraw_denied());
              }
            })
           .handle<get_balance>(
            [&](get_balance const& msg)
            {
              msg.atm_queue.send(::balance(balance));
            })
           .handle<withdrawal_processed>(
            [&](withdrawal_processed const& msg)
            {
            })
           .handle<cancel_withdrawal>(
            [&](cancel_withdrawal const& msg)
            {
            });
      }
    }
    catch(messaging::close_queue const&)
    {
    }
  }

  messaging::sender get_sender()
  {
  return incoming;
  }
};
```

代码C.9 用户状态机

```
class interface_machine
{
  messaging::receiver incoming;
public:
  void done()
  {
    get_sender().send(messaging::close_queue());
```

```
  }

  void run()
  {
    try
    {
      for(;;)
      {
        incoming.wait()
          .handle<issue_money>(
           [&](issue_money const& msg)
           {
             {
               std::lock_guard<std::mutex> lk(iom);
               std::cout<<"Issuing "
                 <<msg.amount<<std::endl;
             }
           })
          .handle<display_insufficient_funds>(
           [&](display_insufficient_funds const& msg)
           {
             {
               std::lock_guard<std::mutex> lk(iom);
               std::cout<<"Insufficient funds"<<std::endl;
             }
           })
          .handle<display_enter_pin>(
           [&](display_enter_pin const& msg)
           {
             {
               std::lock_guard<std::mutex> lk(iom);
               std::cout<<"Please enter your PIN (0-9)"<<std::endl;
             }
           })
          .handle<display_enter_card>(
           [&](display_enter_card const& msg)
           {
             {
               std::lock_guard<std::mutex> lk(iom);
               std::cout<<"Please enter your card (I)"
                 <<std::endl;
             }
           })
          .handle<display_balance>(
           [&](display_balance const& msg)
           {
             {
```

```
                std::lock_guard<std::mutex> lk(iom);
                std::cout
                  <<"The balance of your account is "
                  <<msg.amount<<std::endl;
              }
            })
          .handle<display_withdrawal_options>(
            [&](display_withdrawal_options const& msg)
            {
              {
                std::lock_guard<std::mutex> lk(iom);
                std::cout<<"Withdraw 50? (w)"<<std::endl;
                std::cout<<"Display Balance? (b)"
                  <<std::endl;
                std::cout<<"Cancel? (c)"<<std::endl;
              }
            })
          .handle<display_withdrawal_cancelled>(
            [&](display_withdrawal_cancelled const& msg)
            {
              {
                std::lock_guard<std::mutex> lk(iom);
                std::cout<<"Withdrawal cancelled"
                  <<std::endl;
              }
            })
          .handle<display_pin_incorrect_message>(
            [&](display_pin_incorrect_message const& msg)
            {
              {
                std::lock_guard<std::mutex> lk(iom);
                std::cout<<"PIN incorrect"<<std::endl;
              }
            })
          .handle<eject_card>(
            [&](eject_card const& msg)
            {
              {
                std::lock_guard<std::mutex> lk(iom);
                std::cout<<"Ejecting card"<<std::endl;
              }
            });
      }
    }
    catch(messaging::close_queue&)
    {
    }
```

```
  }

  messaging::sender get_sender()
  {
    return incoming;
  }
};
```

代码C.10 驱动代码

```
int main()
{
  bank_machine bank;
  interface_machine interface_hardware;

  atm machine(bank.get_sender(),interface_hardware.get_sender());

  std::thread bank_thread(&bank_machine::run,&bank);
  std::thread if_thread(&interface_machine::run,&interface_hardware);
  std::thread atm_thread(&atm::run,&machine);

  messaging::sender atmqueue(machine.get_sender());

  bool quit_pressed=false;

  while(!quit_pressed)
  {
    char c=getchar();
    switch(c)
    {
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
      atmqueue.send(digit_pressed(c));
      break;
    case 'b':
      atmqueue.send(balance_pressed());
      break;
    case 'w':
```

```
      atmqueue.send(withdraw_pressed(50));
      break;
    case 'c':
      atmqueue.send(cancel_pressed());
      break;
    case 'q':
      quit_pressed=true;
      break;
    case 'i':
      atmqueue.send(card_inserted("acc1234"));
      break;
    }
  }

  bank.done();
  machine.done();
  interface_hardware.done();

  atm_thread.join();
  bank_thread.join();
  if_thread.join();
}
```

附录D C++线程库参考

D.1 <chrono>头文件

<chrono>头文件作为 `time_point` 的提供者，具有代表时间点的类，duration类和时钟类。每个时钟都有一个 `is_steady` 静态数据成员，这个成员用来表示该时钟是否是一个*稳定的*时钟(以匀速计时的时钟，且不可调节)。 `std::chrono::steady_clock` 是唯一个能保证稳定的时钟类。

头文件正文

```
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类型模板

`std::chrono::duration` 类模板可以用来表示时间。模板参数 `Rep` 和 `Period` 是用来存储持续时间的数据类型， `std::ratio` 实例代表了时间的长度(几分之一秒)，其表示了在两次“时钟滴答”后的时间(时钟周期)。因此， `std::chrono::duration<int, std::milli>` 即为，时间以毫秒数的形式存储到int类型中，而 `std::chrono::duration<short, std::ratio<1,50>>` 则是记录1/50秒的个数，并将个数存入short类型的变量中，还有 `std::chrono::duration <long long , std::ratio<60,1>>` 则是将分钟数存储到long long类型的变量中。

类的定义

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

  constexpr duration() = 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>
  constexpr duration(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*=(const rep& rhs);
  duration& operator/=(const rep& rhs);

  duration& operator%=(const rep& rhs);
  duration& operator%=(const duration& 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` 必须是内置数值类型，或是自定义的类数值类型。

`Period` 必须是 `std::ratio<>` 实例。

std::chrono::duration::Rep 类型

用来记录 `dration` 中时钟周期的数量。

声明

```
typedef Rep rep;
```

`rep` 类型用来记录 `duration` 对象内部的表示。

std::chrono::duration::Period 类型

这个类型必须是一个 `std::ratio` 特化实例，用来表示在继续时间中，1s所要记录的次数。例如，当 `period` 是 `std::ratio<1, 50>`， `duration` 变量的count()就会在N秒钟返回50N。

声明

```
typedef Period period;
```

std::chrono::duration 默认构造函数

使用默认值构造 `std::chrono::duration` 实例

声明

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

效果

`duration` 内部值(例如 `rep` 类型的值)都已初始化。

std::chrono::duration 需要计数值的转换构造函数

通过给定的数值来构造 `std::chrono::duration` 实例。

声明

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

效果

`duration` 对象的内部值会使用 `static_cast<rep>(r)` 进行初始化。

结果

当Rep2隐式转换为Rep，Rep是浮点类型或Rep2不是浮点类型，这个构造函数才能使用。

后验条件

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

std::chrono::duration 需要另一个std::chrono::duration值的转化构造函数

通过另一个 `std::chrono::duration` 类实例中的计数值来构造一个 `std::chrono::duration` 类实例。

声明

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

结果

duration对象的内部值通过 `duration_cast<duration<Rep,Period>>(d).count()` 初始化。

要求

当Rep是一个浮点类或Rep2不是浮点类，且Period2是Period数的倍数(比如，ratio_divide<Period2,Period>::den==1)时，才能调用该重载。当一个较小的数据转换为一个较大的数据时，使用该构造函数就能避免数位截断和精度损失。

后验条件

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

例子

```
duration<int, ratio<1, 1000>> ms(5);  // 5毫秒
duration<int, ratio<1, 1>> s(ms);  // 错误：不能将ms当做s进行存储
duration<double, ratio<1,1>> s2(ms);  // 合法：s2.count() == 0.005
duration<int, ration<1, 1000000>> us<ms>;  // 合法:us.count() == 5000
```

std::chrono::duration::count 成员函数

查询持续时长。

声明

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

返回

返回duration的内部值，其值类型和rep一样。

std::chrono::duration::operator+ 加法操作符

这是一个空操作：只会返回*this的副本。

声明

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

返回

`*this`

std::chrono::duration::operator- 减法操作符

返回将内部值只为负数的*this副本。

声明

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

返回

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

std::chrono::duration::operator++ 前置自加操作符

增加内部计数值。

声明

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

结果

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

返回

`*this`

std::chrono::duration::operator++ 后置自加操作符

自加内部计数值，并且返回还没有增加前的*this。

声明

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

结果

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

std::chrono::duration::operator-- 前置自减操作符

自减内部计数值

声明

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

结果

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

返回

`*this`

std::chrono::duration::operator-- 前置自减操作符

自减内部计数值，并且返回还没有减少前的*this。

声明

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

结果

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

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

将其他duration对象中的内部值增加到现有duration对象当中。

声明

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

结果

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

返回

`*this`

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

现有duration对象减去其他duration对象中的内部值。

声明

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

结果

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

返回

`*this`

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

内部值乘以一个给定的值。

声明

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

结果

```
internal_count*=rhs;
```

返回

`*this`

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

内部值除以一个给定的值。

声明

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

结果

```
internal_count/=rhs;
```

返回

`*this`

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

内部值对一个给定的值求余。

声明

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

结果

```
internal_count%=rhs;
```

返回

`*this`

std::chrono::duration::operator%= 复合赋值操作符(重载)

内部值对另一个duration类的内部值求余。

声明

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

结果

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

返回

`*this`

std::chrono::duration::zero 静态成员函数

返回一个内部值为0的duration对象。

声明

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

返回

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

std::chrono::duration::min 静态成员函数

返回duration类实例化后能表示的最小值。

声明

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

返回

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

std::chrono::duration::max 静态成员函数

返回duration类实例化后能表示的最大值。

声明

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

返回

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

std::chrono::duration 等于比较操作符

比较两个duration对象是否相等。

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` 和 `rhs` 两种类型可以互相进行隐式转换。当两种类型无法进行隐式转换，或是可以互相转换的两个不同类型的duration类，则表达式不合理。

结果

当 `CommonDuration` 和 `std::common_type< duration< Rep1, Period1>, duration< Rep2, Period2>>::type` 同类，那么 `lhs==rhs` 就会返回 `CommonDuration(lhs).count()==CommonDuration(rhs).count()` 。

std::chrono::duration 不等于比较操作符

比较两个duration对象是否不相等。

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` 和 `rhs` 两种类型可以互相进行隐式转换。当两种类型无法进行隐式转换，或是可以互相转换的两个不同类型的duration类，则表达式不合理。

返回

`!(lhs==rhs)`

std::chrono::duration 小于比较操作符

比较两个duration对象是否小于。

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` 和 `rhs` 两种类型可以互相进行隐式转换。当两种类型无法进行隐式转换，或是可以互相转换的两个不同类型的duration类，则表达式不合理。

结果

当 `CommonDuration` 和 `std::common_type< duration< Rep1, Period1>, duration< Rep2, Period2>>::type` 同类，那么 `lhs<rhs` 就会返回 `CommonDuration(lhs).count()<CommonDuration(rhs).count()` 。

std::chrono::duration 大于比较操作符

比较两个duration对象是否大于。

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` 和 `rhs` 两种类型可以互相进行隐式转换。当两种类型无法进行隐式转换，或是可以互相转换的两个不同类型的duration类，则表达式不合理。

返回

`rhs<lhs`

std::chrono::duration 小于等于比较操作符

比较两个duration对象是否小于等于。

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` 和 `rhs` 两种类型可以互相进行隐式转换。当两种类型无法进行隐式转换，或是可以互相转换的两个不同类型的duration类，则表达式不合理。

返回

`!(rhs<lhs)`

std::chrono::duration 大于等于比较操作符

比较两个duration对象是否大于等于。

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` 和 `rhs` 两种类型可以互相进行隐式转换。当两种类型无法进行隐式转换，或是可以互相转换的两个不同类型的duration类，则表达式不合理。

返回

`!(lhs<rhs)`

std::chrono::duration_cast 非成员函数

显示将一个 `std::chrono::duration` 对象转化为另一个 `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 `std::chrono::duration` 的实例。

返回

duration类d转换为指定类型ToDuration。这种方式可以在不同尺寸和表示类型的转换中尽可能减少精度损失。

D.1.2 std::chrono::time_point类型模板

`std::chrono::time_point` 类型模板通过(特别的)时钟来表示某个时间点。这个时钟代表的是从epoch(1970-01-01 00:00:00 UTC，作为UNIX系列系统的特定时间戳)到现在的时间。模板参数Clock代表使用的使用(不同的使用必定有自己独特的类型)，而Duration模板参数使用来测量从epoch到现在的时间，并且这个参数的类型必须是 `std::chrono::duration` 类型。Duration默认存储Clock上的测量值。

类型定义

```
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 constexpr time_point max();
};
```

std::chrono::time_point 默认构造函数

构造time_point代表着，使用相关的Clock，记录从epoch到现在的时间；其内部计时使用Duration::zero()进行初始化。

声明

```
time_point();
```

后验条件

对于使用默认构造函数构造出的time_point对象tp，`tp.time_since_epoch() == tp::duration::zero()`。

std::chrono::time_point 需要时间长度的构造函数

构造time_point代表着，使用相关的Clock，记录从epoch到现在的时间。

声明

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

后验条件

当有一个time_point对象tp，是通过duration d构造出来的(tp(d))，那么`tp.time_since_epoch() == d`。

std::chrono::time_point 转换构造函数

构造time_point代表着，使用相关的Clock，记录从epoch到现在的时间。

声明

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

要求

Duration2必须呢个隐式转换为Duration。

效果

当`time_point(t.time_since_epoch())`存在，从t.time_since_epoch()中获取的返回值，可以隐式转换成Duration类型的对象，并且这个值可以存储在一个新的time_point对象中。

(扩展阅读：**as-if准则**)

std::chrono::time_point::time_since_epoch 成员函数

返回当前time_point从epoch到现在的具体时长。

声明

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

返回

duration的值存储在*this中。

std::chrono::time_point::operator+= 复合赋值函数

将指定的duration的值与原存储在指定的time_point对象中的duration相加，并将加后值存储在*this对象中。

声明

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

效果

将d的值和duration对象的值相加，存储在*this中，就如同this->internal_duration += d;

返回

`*this`

std::chrono::time_point::operator-= 复合赋值函数

将指定的duration的值与原存储在指定的time_point对象中的duration相减，并将加后值存储在*this对象中。

声明

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

效果

将d的值和duration对象的值相减，存储在*this中，就如同this->internal_duration -= d;

返回

`*this`

std::chrono::time_point::min 静态成员函数

获取time_point对象可能表示的最小值。

声明

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

返回

```
time_point(time_point::duration::min()) (see 11.1.1.15)
```

std::chrono::time_point::max 静态成员函数

获取time_point对象可能表示的最大值。

声明

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

返回

```
time_point(time_point::duration::max()) (see 11.1.1.16)
```

##D.1.3 std::chrono::system_clock类

`std::chrono::system_clock` 类提供给了从系统实时时钟上获取当前时间功能。可以调用 `std::chrono::system_clock::now()` 来获取当前的时间。 `std::chrono::system_clock::time_point` 也可以通过 `std::chrono::system_clock::to_time_t()` 和 `std::chrono::system_clock::to_time_point()` 函数返回值转换成time_t类型。系统时钟不稳定，所以 `std::chrono::system_clock::now()` 获取到的时间可能会早于之前的一次调用(比如，时钟被手动调整过或与外部时钟进行了同步)。

###类型定义

```
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;
};
```

std::chrono::system_clock::rep 类型定义

将时间周期数记录在一个duration值中

声明

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

std::chrono::system_clock::period 类型定义

类型为 `std::ratio` 类型模板，通过在两个不同的duration或time_point间特化最小秒数(或将1秒分为好几份)。period指定了时钟的精度，而非时钟频率。

声明

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

std::chrono::system_clock::duration 类型定义

类型为 `std::ratio` 类型模板，通过系统实时时钟获取两个时间点之间的时长。

声明

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

std::chrono::system_clock::time_point 类型定义

类型为 `std::ratio` 类型模板，通过系统实时时钟获取当前时间点的时间。

声明

```
typedef std::chrono::time_point&lt;std::chrono::system_clock&gt; time_point;
```

std::chrono::system_clock::now 静态成员函数

从系统实时时钟上获取当前的外部设备显示的时间。

声明

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

返回

time_point类型变量来代表当前系统实时时钟的时间。

抛出

当错误发生，`std::system_error`异常将会抛出。

std::chrono::system_clock:to_time_t 静态成员函数

将time_point类型值转化为time_t。

声明

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

返回

通过对t进行舍入或截断精度，将其转化为一个time_t类型的值。

抛出

当错误发生，`std::system_error`异常将会抛出。

std::chrono::system_clock::from_time_t 静态成员函数

声明

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

返回

time_point中的值与t中的值一样。

抛出

当错误发生，`std::system_error`异常将会抛出。

D.1.4 std::chrono::steady_clock类

`std::chrono::steady_clock`能访问系统稳定时钟。可以通过调用`std::chrono::steady_clock::now()`获取当前的时间。设备上显示的时间，与使用`std::chrono::steady_clock::now()`获取的时间没有固定的关系。稳定时钟是无法回调的，所以在`std::chrono::steady_clock::now()`两次调用后，第二次调用获取的时间必定等于或大于第一次获得的时间。

类型定义

```
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;
};
```

std::chrono::steady_clock::rep 类型定义

定义一个整型，用来保存duration的值。

声明

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

std::chrono::steady_clock::period 类型定义

类型为 `std::ratio` 类型模板，通过在两个不同的duration或time_point间特化最小秒数(或将1秒分为好几份)。period指定了时钟的精度，而非时钟频率。

声明

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

std::chrono::steady_clock::duration 类型定义

类型为 `std::ratio` 类型模板，通过系统实时时钟获取两个时间点之间的时长。

声明

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

std::chrono::steady_clock::time_point 类型定义

`std::chrono::time_point` 类型实例，可以存储从系统稳定时钟返回的时间点。

声明

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

std::chrono::steady_clock::now 静态成员函数

从系统稳定时钟获取当前时间。

声明

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

返回

time_point表示当前系统稳定时钟的时间。

抛出

当遇到错误，会抛出 `std::system_error` 异常。

同步

当先行调用过一次 `std::chrono::steady_clock::now()` ，那么下一次time_point获取的值，一定大于等于第一次获取的值。

D.1.5 std::chrono::high_resolution_clock类定义

`td::chrono::high_resolution_clock` 类能访问系统高精度时钟。和所有其他时钟一样，通过调用 `std::chrono::high_resolution_clock::now()` 来获取当前时间。 `std::chrono::high_resolution_clock` 可能是 `std::chrono::system_clock` 类或 `std::chrono::steady_clock` 类的别名，也可能就是独立的一个类。

通过 `std::chrono::high_resolution_clock` 具有所有标准库支持时钟中最高的精度，这就意味着使用
`std::chrono::high_resolution_clock::now()` 要花掉一些时间。所以，当你再调用 `std::chrono::high_resolution_clock::now()` 的时候，需要注意函数本身的时间开销。

类型定义

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

头文件内容

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

  class condition_variable;
  class condition_variable_any;
}
```

D.2.1 std::condition_variable类

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

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

类型定义

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

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

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

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

```

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

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

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

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

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

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

std::condition_variable 默认构造函数

构造一个 `std::condition_variable` 对象。

声明

```
condition_variable();
```

效果

构造一个新的 `std::condition_variable` 实例。

抛出

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

std::condition_variable 析构函数

销毁一个 `std::condition_variable` 对象。

声明

```
~condition_variable();
```

先决条件

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

效果

销毁*this。

抛出

无

std::condition_variable::notify_one 成员函数

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

声明

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

效果

唤醒一个等待*this的线程。如果没有线程在等待，那么调用没有任何效果。

抛出

当效果没有达成，就会抛出 `std::system_error` 异常。

同步

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

std::condition_variable::notify_all 成员函数

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

声明

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

效果

唤醒所有等待*this的线程。如果没有线程在等待，那么调用没有任何效果。

抛出

当效果没有达成，就会抛出 `std::system_error` 异常

同步

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

std::condition_variable::wait 成员函数

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

等待

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

先决条件

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

效果

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

抛出

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

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

同步

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

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

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

声明

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

先决条件

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

效果

正如

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

抛出

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

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

同步

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

std::condition_variable::wait_for 成员函数

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

声明

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

先决条件

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

效果

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

返回

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

抛出

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

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

同步

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

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

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

声明

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

先决条件

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

效果

等价于

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

返回

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

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

抛出

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

同步

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

std::condition_variable::wait_until 成员函数

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

声明

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

先决条件

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

效果

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

返回

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

抛出

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

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

同步

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

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

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

声明

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

先决条件

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

效果

等价于

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

返回

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

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

抛出

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

同步

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

std::notify_all_at_thread_exit 非成员函数

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

声明

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

先决条件

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

效果

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

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

抛出

当效果没有达成时，抛出 `std::system_error` 异常。

NOTE:在线程退出前，掌握着锁的所有权，所以这里要避免死锁发生。这里建议调用该函数的线程应该尽快退出，并且在该线程可以执行一些阻塞的操作。用户必须保证等地线程不会错误的将唤醒线程当做已退出的线程，特别是伪唤醒。可以通过等待线程上的谓词测试来实现这一功能，在互斥量保护的情况下，只有谓词返回true时线程才能被唤醒，并且在调用notify_all_at_thread_exit(std::condition_variable_any类中函数)前是不会释放锁。

D.2.2 std::condition_variable_any类

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

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

类型定义

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

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

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

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

```
17
18    template <typename Lockable, typename Predicate>
19    void wait(Lockable& lock, Predicate pred);
20
21    template <typename Lockable, typename Clock,typename Duration>
22    std::cv_status wait_until(
23        Lockable& lock,
24        const std::chrono::time_point<Clock, Duration>& absolute_time);
25
26    template <
27        typename Lockable, typename Clock,
28        typename Duration, typename Predicate>
29    bool wait_until(
30        Lockable& lock,
31        const std::chrono::time_point<Clock, Duration>& absolute_time,
32        Predicate pred);
33
34    template <typename Lockable, typename Rep, typename Period>
35    std::cv_status wait_for(
36        Lockable& lock,
37        const std::chrono::duration<Rep, Period>& relative_time);
38
39    template <
40        typename Lockable, typename Rep,
41        typename Period, typename Predicate>
42    bool wait_for(
43        Lockable& lock,
44        const std::chrono::duration<Rep, Period>& relative_time,
45        Predicate pred);
46  };
```

std::condition_variable_any 默认构造函数

构造一个 `std::condition_variable_any` 对象。

声明

```
1  condition_variable_any();
```

效果

构造一个新的 `std::condition_variable_any` 实例。

抛出
当条件变量构造成功，将抛出 `std::system_error` 异常。

std::condition_variable_any 析构函数

销毁 `std::condition_variable_any` 对象。

声明

```
~condition_variable_any();
```

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

效果
销毁*this。

抛出
无

std::condition_variable_any::notify_one 成员函数

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

声明

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

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

抛出
当效果没有达成，就会抛出std::system_error异常。

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

std::condition_variable_any::notify_all 成员函数

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

声明

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

效果

唤醒所有等待*this的线程。如果没有线程在等待，那么调用没有任何效果

抛出

当效果没有达成，就会抛出std::system_error异常。

同步

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

std::condition_variable_any::wait 成员函数

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

声明

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

先决条件

Lockable类型需要能够上锁，lock对象拥有一个锁。

效果

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

抛出

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

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

同步

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

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

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

声明

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

先决条件

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

效果

正如

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

抛出

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

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

同步

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

std::condition_variable_any::wait_for 成员函数

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

声明

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

先决条件

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

效果

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

返回

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

抛出

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

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

同步

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

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

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

声明

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

先决条件

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

效果

正如

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

返回

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

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

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

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

std::condition_variable_any::wait_until 成员函数

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

声明

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

先决条件
Lockable类型需要能够上锁，lock对象拥有一个锁。

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

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

抛出

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

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

同步

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

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

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

声明

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

先决条件

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

效果

等价于

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

返回

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

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

抛出

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

同步

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

D.3 <atomic>头文件

<atomic>头文件提供一组基础的原子类型，和提供对这些基本类型的操作，以及一个原子模板函数，用来接收用户定义的类型，以适用于某些标准。

###头文件内容

```
#define ATOMIC_BOOL_LOCK_FREE 参见详述
#define ATOMIC_CHAR_LOCK_FREE 参见详述
#define ATOMIC_SHORT_LOCK_FREE 参见详述
#define ATOMIC_INT_LOCK_FREE 参见详述
#define ATOMIC_LONG_LOCK_FREE 参见详述
#define ATOMIC_LLONG_LOCK_FREE 参见详述
#define ATOMIC_CHAR16_T_LOCK_FREE 参见详述
#define ATOMIC_CHAR32_T_LOCK_FREE 参见详述
#define ATOMIC_WCHAR_T_LOCK_FREE 参见详述
#define ATOMIC_POINTER_LOCK_FREE 参见详述

#define ATOMIC_VAR_INIT(value) 参见详述

namespace std
{
  enum memory_order;

  struct atomic_flag;
  参见类型定义详述 atomic_bool;
  参见类型定义详述 atomic_char;
  参见类型定义详述 atomic_char16_t;
  参见类型定义详述 atomic_char32_t;
  参见类型定义详述 atomic_schar;
  参见类型定义详述 atomic_uchar;
  参见类型定义详述 atomic_short;
  参见类型定义详述 atomic_ushort;
  参见类型定义详述 atomic_int;
  参见类型定义详述 atomic_uint;
  参见类型定义详述 atomic_long;
  参见类型定义详述 atomic_ulong;
  参见类型定义详述 atomic_llong;
  参见类型定义详述 atomic_ullong;
  参见类型定义详述 atomic_wchar_t;

  参见类型定义详述 atomic_int_least8_t;
  参见类型定义详述 atomic_uint_least8_t;
```

```
  参见类型定义详述 atomic_int_least16_t;
  参见类型定义详述 atomic_uint_least16_t;
  参见类型定义详述 atomic_int_least32_t;
  参见类型定义详述 atomic_uint_least32_t;
  参见类型定义详述 atomic_int_least64_t;
  参见类型定义详述 atomic_uint_least64_t;
  参见类型定义详述 atomic_int_fast8_t;
  参见类型定义详述 atomic_uint_fast8_t;
  参见类型定义详述 atomic_int_fast16_t;
  参见类型定义详述 atomic_uint_fast16_t;
  参见类型定义详述 atomic_int_fast32_t;
  参见类型定义详述 atomic_uint_fast32_t;
  参见类型定义详述 atomic_int_fast64_t;
  参见类型定义详述 atomic_uint_fast64_t;
  参见类型定义详述 atomic_int8_t;
  参见类型定义详述 atomic_uint8_t;
  参见类型定义详述 atomic_int16_t;
  参见类型定义详述 atomic_uint16_t;
  参见类型定义详述 atomic_int32_t;
  参见类型定义详述 atomic_uint32_t;
  参见类型定义详述 atomic_int64_t;
  参见类型定义详述 atomic_uint64_t;
  参见类型定义详述 atomic_intptr_t;
  参见类型定义详述 atomic_uintptr_t;
  参见类型定义详述 atomic_size_t;
  参见类型定义详述 atomic_ssize_t;
  参见类型定义详述 atomic_ptrdiff_t;
  参见类型定义详述 atomic_intmax_t;
  参见类型定义详述 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);
}
```

std::atomic_xxx类型定义

为了兼容新的C标准(C11)，C++支持定义原子整型类型。这些类型都与 `std::atimic<T>;` 特化类相对应，或是用同一接口特化的一个基本类型。

Table D.1 原子类型定义和与之相关的std::atmoic<>特化模板

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

(译者注：该表与第5章中的表5.1几乎一致)

D.3.2 ATOMIC_xxx_LOCK_FREE宏

这里的宏指定了原子类型与其内置类型是否是无锁的。

宏定义

```
#define ATOMIC_BOOL_LOCK_FREE 参见详述
#define ATOMIC_CHAR_LOCK_FREE参见详述
#define ATOMIC_SHORT_LOCK_FREE 参见详述
#define ATOMIC_INT_LOCK_FREE 参见详述
#define ATOMIC_LONG_LOCK_FREE 参见详述
#define ATOMIC_LLONG_LOCK_FREE 参见详述
#define ATOMIC_CHAR16_T_LOCK_FREE 参见详述
#define ATOMIC_CHAR32_T_LOCK_FREE 参见详述
#define ATOMIC_WCHAR_T_LOCK_FREE 参见详述
#define ATOMIC_POINTER_LOCK_FREE 参见详述
```

`ATOMIC_xxx_LOCK_FREE` 的值无非就是0，1，2。0意味着，在对有无符号的相关原子类型操作是有锁的；1意味着，操作只对一些特定的类型上锁，而对没有指定的类型不上锁；2意味着，所有操作都是无锁的。例如，当 `ATOMIC_INT_LOCK_FREE` 是2的时候，在 `std::atomic&lt;int&gt;` 和 `std::atomic&lt;unsigned&gt;` 上的操作始终无锁。

宏 `ATOMIC_POINTER_LOCK_FREE` 描述了，对于特化的原子类型指针 `std::atomic<T*>` 操作的无锁特性。

D.3.3 ATOMIC_VAR_INIT宏

`ATOMIC_VAR_INIT` 宏可以通过一个特定的值来初始化一个原子变量。

声明

`#define ATOMIC_VAR_INIT(value)参见详述`

宏可以扩展成一系列符号，这个宏可以通过一个给定值，初始化一个标准原子类型，表达式如下所示：

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

给定值可以兼容与原子变量相关的非原子变量，例如：

```
std::atomic&amp;lt;int&gt; i = ATOMIC_VAR_INIT(42);
std::string s;
std::atomic&amp;lt;std::string*&gt; p = ATOMIC_VAR_INIT(&s);
```

这样初始化的变量是非原子的，并且在变量初始化之后，其他线程可以随意的访问该变量，这样可以避免条件竞争和未定义行为的发生。

D.3.4 std::memory_order枚举类型

`std::memory_order` 枚举类型用来表明原子操作的约束顺序。

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;
```

通过标记各种内存序变量来标记操作的顺序(详见第5章，在该章节中有对书序约束更加详尽的介绍)

std::memory_order_relaxed

操作不受任何额外的限制。

std::memory_order_release

对于指定位置上的内存可进行释放操作。因此，与获取操作读取同一内存位置所存储的值。

std::memory_order_acquire

操作可以获取指定内存位置上的值。当需要存储的值通过释放操作写入时，是与存储操同步的。

std::memory_order_acq_rel

操作必须是“读-改-写”操作，并且其行为需要在 `std::memory_order_acquire` 和 `std::memory_order_release` 序指定的内存位置上进行操作。

std::memory_order_seq_cst

操作在全局序上都会受到约束。还有，当为存储操作时，其行为好比`std::memory_order_release`操作；当为加载操作时，其行为好比`std::memory_order_acquire`操作；并且，当其是一个“读-改-写”操作时，其行为和`std::memory_order_acquire`和`std::memory_order_release`类似。对于所有顺序来说，该顺序为默认序。

std::memory_order_consume

对于指定位置的内存进行消耗操作(consume operation)。

(译者注：与memory_order_acquire类似)

##D.3.5 std::atomic_thread_fence函数

`std::atomic_thread_fence()`会在代码中插入“内存栅栏”，强制两个操作保持内存约束顺序。

声明

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

效果

插入栅栏的目的是为了保证内存序的约束性。

栅栏使用`std::memory_order_release`，`std::memory_order_acq_rel`，或`std::memory_order_seq_cst`内存序，会同步与一些内存位置上的获取操作进行同步，如果这些获取操作要获取一个已存储的值(通过原子操作进行的存储)，就会通过栅栏进行同步。

释放操作可对`std::memory_order_acquire`，`std::memory_order_acq_rel`，或`std::memory_order_seq_cst`进行栅栏同步，；当释放操作存储的值，在一个原子操作之前读取，那么就会通过栅栏进行同步。

抛出

无

D.3.6 std::atomic_signal_fence函数

`std::atomic_signal_fence()` 会在代码中插入“内存栅栏”，强制两个操作保持内存约束顺序，并且在对应线程上执行信号处理操作。

声明

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

效果

根据需要的内存约束序插入一个栅栏。除非约束序应用于“操作和信号处理函数在同一线程”的情况下，否则，这个操作等价于 `std::atomic_thread_fence(order)` 操作。

抛出

无

D.3.7 std::atomic_flag类

`std::atomic_flag` 类算是原子标识的骨架。在C++11标准下，只有这个数据类型可以保证是无锁的(当然，更多的原子类型在未来的实现中将采取无锁实现)。

对于一个 `std::atomic_flag` 来说，其状态不是set，就是clear。

类型定义

```
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
```

std::atomic_flag 默认构造函数

这里未指定默认构造出来的 `std::atomic_flag` 实例是clear状态，还是set状态。因为对象存储过程是静态的，所以初始化必须是静态的。

声明

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

效果

构造一个新 `std::atomic_flag` 对象，不过未指明状态。(薛定谔的猫？)

抛出

无

std::atomic_flag 使用ATOMIC_FLAG_INIT进行初始化

`std::atomic_flag` 实例可以使用 `ATOMIC_FLAG_INIT` 宏进行创建，这样构造出来的实例状态为clear。因为对象存储过程是静态的，所以初始化必须是静态的。

声明

```
#define ATOMIC_FLAG_INIT unspecified
```

用法

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

效果

构造一个新 `std::atomic_flag` 对象，状态为clear。

抛出

无

NOTE:

对于内存位置上的*this，这个操作属于“读-改-写”操作。

std::atomic_flag::test_and_set 成员函数

自动设置实例状态标识，并且检查实例的状态标识是否已经设置。

声明

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

效果

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

std::atomic_flag_test_and_set 非成员函数

自动设置原子变量的状态标识，并且检查原子变量的状态标识是否已经设置。

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;
```

效果

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

std::atomic_flag_test_and_set_explicit 非成员函数

自动设置原子变量的状态标识，并且检查原子变量的状态标识是否已经设置。

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;
```

效果

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

std::atomic_flag::clear 成员函数

自动清除原子变量的状态标识。

声明

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

先决条件

支持 `std::memory_order_relaxed`，`std::memory_order_release` 和 `std::memory_order_seq_cst` 中任意一个。

效果

自动清除变量状态标识。

抛出

无

NOTE:对于内存位置上的*this，这个操作属于“写”操作(存储操作)。

std::atomic_flag_clear 非成员函数

自动清除原子变量的状态标识。

声明

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

效果

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

std::atomic_flag_clear_explicit 非成员函数

自动清除原子变量的状态标识。

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;
```

效果

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

##D.3.8 std::atomic类型模板

`std::atomic` 提供了对任意类型的原子操作的包装，以满足下面的需求。

模板参数BaseType必须满足下面的条件。

- 具有简单的默认构造函数
- 具有简单的拷贝赋值操作
- 具有简单的析构函数
- 可以进行位比较

这就意味着 `std::atomic<some-simple-struct>` 会和使用 `std::atomic<some-built-in-type>` 一样简单；不过对于 `std::atomic<std::string>` 就不同了。

除了主模板，对于内置整型和指针的特化，模板也支持类似x++这样的操作。

`std::atomic` 实例是不支持 `CopyConstructible` (拷贝构造)和 `CopyAssignable` (拷贝赋值)，原因你懂得，因为这样原子操作就无法执行。

类型定义

```
template<typename BaseType>
struct atomic
{
  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,
```

```
      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>
```

```
130 bool atomic_compare_exchange_weak_explicit(
131   atomic<BaseType>*,BaseType * old_value,
132   BaseType new_value, memory_order success_order,
133   memory_order failure_order) noexcept;
```

NOTE:虽然非成员函数通过模板的方式指定，不过他们只作为从在函数提供，并且对于这些函数，不能显示的指定模板的参数。

std::atomic 构造函数

使用默认初始值，构造一个 `std::atomic` 实例。

声明

```
1 atomic() noexcept;
```

效果

使用默认初始值，构造一个新 `std::atomic` 实例。因对象是静态存储的，所以初始化过程也是静态的。

NOTE:当 `std::atomic` 实例以非静态方式初始化的，那么其值就是不可估计的。

抛出

无

std::atomic_init 非成员函数

`std::atomic<BaseType>` 实例提供的值，可非原子的进行存储。

声明

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

效果

将值v以非原子存储的方式，存储在*p中。调用 `atomic<BaseType>` 实例中的atomic_init()，这里需要实例不是默认构造出来的，或者在构造出来的时候被执行了某些操作，否则将会引发未定义行为。

NOTE:因为存储是非原子的，对对象指针p任意的并发访问(即使是原子操作)都会引发数据竞争。

抛出

无

std::atomic 转换构造函数

使用提供的BaseType值去构造一个 `std::atomic` 实例。

声明

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

效果

通过b值构造一个新的 `std::atomic` 对象。因对象是静态存储的，所以初始化过程也是静态的。

抛出

无

std::atomic 转换赋值操作

在*this存储一个新值。

声明

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

效果

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

std::atomic::is_lock_free 成员函数

确定对于*this是否是无锁操作。

声明

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

返回
当操作是无锁操作，那么就返回true，否则返回false。

抛出
无

std::atomic_is_lock_free 非成员函数

确定对于*this是否是无锁操作。

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;
```

效果

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

std::atomic::load 成员函数

原子的加载 `std::atomic` 实例当前的值

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;
```

先决条件

支持 `std::memory_order_relaxed` 、 `std::memory_order_acquire` 、 `std::memory_order_consume` 或 `std::memory_order_seq_cst` 内存序。

效果

原子的加载已存储到*this上的值。

返回

返回存储在*this上的值。

抛出

无

NOTE:是对于*this内存地址原子加载的操作。

std::atomic_load 非成员函数

原子的加载 `std::atomic` 实例当前的值。

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;
```

效果

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

std::atomic_load_explicit 非成员函数

原子的加载 `std::atomic` 实例当前的值。

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;
```

效果

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

std::atomic::operator BastType转换操作

加载存储在*this中的值。

声明

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

效果

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

std::atomic::store 成员函数

以原子操作的方式存储一个新值到 `atomic<BaseType>` 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;
```

先决条件

支持 `std::memory_order_relaxed` 、 `std::memory_order_release` 或 `std::memory_order_seq_cst` 内存序。

效果
将new_value原子的存储到*this中。

抛出
无

NOTE:是对于*this内存地址原子加载的操作。

std::atomic_store 非成员函数

以原子操作的方式存储一个新值到 `atomic<BaseType>` 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;
```

效果

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

std::atomic_explicit 非成员函数

以原子操作的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;
```

效果

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

std::atomic::exchange 成员函数

原子的存储一个新值，并读取旧值。

声明

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

效果

原子的将new_value存储在*this中，并且取出*this中已经存储的值。

返回

返回*this之前的值。

抛出

无

NOTE:这是对*this内存地址的原子“读-改-写”操作。

std::atomic_exchange 非成员函数

原子的存储一个新值到 `atomic<BaseType>` 实例中，并且读取旧值。

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;
```

效果

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

std::atomic_exchange_explicit 非成员函数

原子的存储一个新值到 `atomic<BaseType>` 实例中，并且读取旧值。

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;
```

效果

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

std::atomic::compare_exchange_strong 成员函数

当期望值和新值一样时，将新值存储到实例中。如果不相等，那么就实用新值更新期望值。

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;
```

先决条件

failure_order不能是 `std::memory_order_release` 或 `std::memory_order_acq_rel` 内存序。

效果

将存储在*this中的expected值与new_value值进行逐位对比，当相等时间new_value存储在*this中；否则，更新expected的值。

返回

当new_value的值与*this中已经存在的值相同，就返回true；否则，返回false。

抛出

无

NOTE:在success_order和failure_order的情况下，三个参数的重载函数与四个参数的重载函数等价。除非，order是`std::memory_order_acq_rel`时，failure_order是`std::memory_order_acquire`，且当order是`std::memory_order_release`时，failure_order是`std::memory_order_relaxed`。

NOTE:当返回true和success_order内存序时，是对*this内存地址的原子“读-改-写”操作；反之，这是对*this内存地址的原子加载操作(failure_order)。

std::atomic_compare_exchange_strong 非成员函数

当期望值和新值一样时，将新值存储到实例中。如果不相等，那么就实用新值更新期望值。

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;
```

效果

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

std::atomic_compare_exchange_strong_explicit 非成员函数

当期望值和新值一样时，将新值存储到实例中。如果不相等，那么就实用新值更新期望值。

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;
```

效果

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

std::atomic::compare_exchange_weak 成员函数

原子的比较新值和期望值，如果相等，那么存储新值并且进行原子化更新。当两值不相等，或更新未进行，那期望值会更新为新值。

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;
```

先决条件

failure_order不能是 `std::memory_order_release` 或 `std::memory_order_acq_rel` 内存序。

效果

将存储在*this中的expected值与new_value值进行逐位对比，当相等时间new_value存储在*this中；否则，更新expected的值。

返回

当new_value的值与*this中已经存在的值相同，就返回true；否则，返回false。

抛出

无

NOTE:在success_order和failure_order的情况下，三个参数的重载函数与四个参数的重载函数等价。除非，order是 `std::memory_order_acq_rel` 时，failure_order是 `std::memory_order_acquire` ，且当order是 `std::memory_order_release` 时，failure_order是 `std::memory_order_relaxed` 。

NOTE:当返回true和success_order内存序时，是对*this内存地址的原子"读-改-写"操作；反之，这是对*this内存地址的原子加载操作(failure_order)。

std::atomic_compare_exchange_weak 非成员函数

原子的比较新值和期望值，如果相等，那么存储新值并且进行原子化更新。当两值不相等，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;
```

效果

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

std::atomic_compare_exchange_weak_explicit 非成员函数

原子的比较新值和期望值，如果相等，那么存储新值并且进行原子化更新。当两值不相等，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_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;
```

效果

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

D.3.9 std::atomic模板类型的特化

`std::atomic` 类模板的特化类型有整型和指针类型。对于整型来说，特化模板提供原子加减，以及位域操作(主模板未提供)。对于指针类型来说，特化模板提供原子指针的运算(主模板未提供)。

特化模板提供如下整型：

```
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&gt;
```

`std::atomic<T*>` 原子指针，可以使用以上的类型作为T。

D.3.10 特化std::atomic < integral-type >

`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>
```

因为原子操作只能执行其中一个，所以特化模板的实例不可 `CopyConstructible` (拷贝构造)和 `CopyAssignable` (拷贝赋值)。

类型定义

```
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)
      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;
```

```
56
57    integral-type fetch_add(
58        integral-type,memory_order = memory_order_seq_cst)
59        volatile noexcept;
60    integral-type fetch_add(
61        integral-type,memory_order = memory_order_seq_cst) noexcept;
62    integral-type fetch_sub(
63        integral-type,memory_order = memory_order_seq_cst)
64        volatile noexcept;
65    integral-type fetch_sub(
66        integral-type,memory_order = memory_order_seq_cst) noexcept;
67    integral-type fetch_and(
68        integral-type,memory_order = memory_order_seq_cst)
69        volatile noexcept;
70    integral-type fetch_and(
71        integral-type,memory_order = memory_order_seq_cst) noexcept;
72    integral-type fetch_or(
73        integral-type,memory_order = memory_order_seq_cst)
74        volatile noexcept;
75    integral-type fetch_or(
76        integral-type,memory_order = memory_order_seq_cst) noexcept;
77    integral-type fetch_xor(
78        integral-type,memory_order = memory_order_seq_cst)
79        volatile noexcept;
80    integral-type fetch_xor(
81        integral-type,memory_order = memory_order_seq_cst) noexcept;
82
83    integral-type operator++() volatile noexcept;
84    integral-type operator++() noexcept;
85    integral-type operator++(int) volatile noexcept;
86    integral-type operator++(int) noexcept;
87    integral-type operator--() volatile noexcept;
88    integral-type operator--() noexcept;
89    integral-type operator--(int) volatile noexcept;
90    integral-type operator--(int) noexcept;
91    integral-type operator+=(integral-type) volatile noexcept;
92    integral-type operator+=(integral-type) noexcept;
93    integral-type operator-=(integral-type) volatile noexcept;
94    integral-type operator-=(integral-type) noexcept;
95    integral-type operator&=(integral-type) volatile noexcept;
96    integral-type operator&=(integral-type) noexcept;
97    integral-type operator|=(integral-type) volatile noexcept;
98    integral-type operator|=(integral-type) noexcept;
99    integral-type operator^=(integral-type) volatile noexcept;
100   integral-type operator^=(integral-type) noexcept;
101 };
102
```

```
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;
integral-type 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;
```

这些操作在主模板中也有提供(见D.3.8)。

std::atomic<integral-type>::fetch_add 成员函数

原子的加载一个值，然后使用与提供i相加的结果，替换掉原值。

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;
```

效果
原子的查询*this中的值，将old-value+i的和存回*this。

返回
返回*this之前存储的值。

抛出
无

NOTE:对于*this的内存地址来说，这是一个“读-改-写”操作。

std::atomic_fetch_add 非成员函数

从 `atomic<integral-type>` 实例中原子的读取一个值，并且将其与给定i值相加，替换原值。

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;
```

效果

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

std::atomic_fetch_add_explicit 非成员函数

从 `atomic<integral-type>` 实例中原子的读取一个值，并且将其与给定i值相加，替换原值。

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;
```

效果

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

std::atomic<integral-type>::fetch_sub 成员函数

原子的加载一个值，然后使用与提供i相减的结果，替换掉原值。

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;
```

效果

原子的查询*this中的值，将old-value-i的和存回*this。

返回

返回*this之前存储的值。

抛出

无

NOTE:对于*this的内存地址来说，这是一个“读-改-写”操作。

std::atomic_fetch_sub 非成员函数

从 `atomic<integral-type>` 实例中原子的读取一个值，并且将其与给定i值相减，替换原值。

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;
```

效果

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

std::atomic_fetch_sub_explicit 非成员函数

从 `atomic<integral-type>` 实例中原子的读取一个值，并且将其与给定i值相减，替换原值。

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;
```

效果

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

std::atomic<integral-type>::fetch_and 成员函数

从 `atomic<integral-type>` 实例中原子的读取一个值，并且将其与给定i值进行位与操作后，替换原值。

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;
```

效果

原子的查询*this中的值，将old-value&i的和存回*this。

返回

返回*this之前存储的值。

抛出

无

NOTE:对于*this的内存地址来说，这是一个“读-改-写”操作。

std::atomic_fetch_and 非成员函数

从 `atomic<integral-type>` 实例中原子的读取一个值，并且将其与给定i值进行位与操作后，替换原值。

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;
```

效果

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

std::atomic_fetch_and_explicit 非成员函数

从 `atomic<integral-type>` 实例中原子的读取一个值，并且将其与给定i值进行位与操作后，替换原值。

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;
```

效果

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

std::atomic<integral-type>::fetch_or 成员函数

从 `atomic<integral-type>` 实例中原子的读取一个值，并且将其与给定i值进行位或操作后，替换原值。

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;
```

效果

原子的查询*this中的值，将old-value|i的和存回*this。

返回

返回*this之前存储的值。

抛出

无

NOTE:对于*this的内存地址来说，这是一个“读-改-写”操作。

std::atomic_fetch_or 非成员函数

从 `atomic<integral-type>` 实例中原子的读取一个值，并且将其与给定i值进行位或操作后，替换原值。

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;
```

效果

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

std::atomic_fetch_or_explicit 非成员函数

从 `atomic<integral-type>` 实例中原子的读取一个值，并且将其与给定i值进行位或操作后，替换原值。

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;
```

效果

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

std::atomic<integral-type>::fetch_xor 成员函数

从 `atomic<integral-type>` 实例中原子的读取一个值，并且将其与给定i值进行位亦或操作后，替换原值。

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;
```

效果

原子的查询*this中的值，将old-value^i的和存回*this。

返回

返回*this之前存储的值。

抛出

无

NOTE:对于*this的内存地址来说，这是一个“读-改-写”操作。

std::atomic_fetch_xor 非成员函数

从 `atomic<integral-type>` 实例中原子的读取一个值，并且将其与给定i值进行位异或操作后，替换原值。

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;
```

效果

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

std::atomic_fetch_xor_explicit 非成员函数

从 `atomic<integral-type>` 实例中原子的读取一个值，并且将其与给定i值进行位异或操作后，替换原值。

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;
```

效果

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

std::atomic<integral-type>::operator++ 前置递增操作

原子的将*this中存储的值加1，并返回新值。

声明

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

效果

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

std::atomic<integral-type>::operator++ 后置递增操作

原子的将*this中存储的值加1，并返回旧值。

声明

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

效果

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

std::atomic<integral-type>::operator-- 前置递减操作

原子的将*this中存储的值减1，并返回新值。

声明

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

效果

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

std::atomic<integral-type>::operator-- 后置递减操作

原子的将*this中存储的值减1，并返回旧值。

声明

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

效果

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

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

原子的将给定值与*this中的值相加，并返回新值。

声明

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

效果

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

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

原子的将给定值与*this中的值相减，并返回新值。

声明

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

效果

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

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

原子的将给定值与*this中的值相与，并返回新值。

声明

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

效果

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

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

原子的将给定值与*this中的值相或，并返回新值。

声明

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

效果

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

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

原子的将给定值与*this中的值相亦或，并返回新值。

声明

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

效果

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

std::atomic<T*> 局部特化

`std::atomic<T*>` 为 `std::atomic` 特化了指针类型原子变量，并提供了一系列相关操作。

`std::atomic<T*>` 是CopyConstructible(拷贝构造)和CopyAssignable(拷贝赋值)的，因为操作是原子的，在同一时间只能执行一个操作。

类型定义

```
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;

  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;
```

在主模板中也提供了一些相同的操作(可见11.3.8节)。

std::atomic<T*>::fetch_add 成员函数

原子的加载一个值，然后使用与提供i相加(使用标准指针运算规则)的结果，替换掉原值。

声明

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

效果

原子的查询*this中的值，将old-value+i的和存回*this。

返回

返回*this之前存储的值。

抛出

无

NOTE:对于*this的内存地址来说，这是一个“读-改-写”操作。

std::atomic_fetch_add 非成员函数

从 `atomic<T*>` 实例中原子的读取一个值，并且将其与给定i值进行位相加操作(使用标准指针运算规则)后，替换原值。

声明

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

效果

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

std::atomic_fetch_add_explicit 非成员函数

从 `atomic<T*>` 实例中原子的读取一个值，并且将其与给定i值进行位相加操作(使用标准指针运算规则)后，替换原值。

声明

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

效果

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

std::atomic<T*>::fetch_sub 成员函数

原子的加载一个值，然后使用与提供i相减(使用标准指针运算规则)的结果，替换掉原值。

声明

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

效果

原子的查询*this中的值，将old-value-i的和存回*this。

返回

返回*this之前存储的值。

抛出

无

NOTE:对于*this的内存地址来说，这是一个“读-改-写”操作。

std::atomic_fetch_sub 非成员函数

从 `atomic<T*>` 实例中原子的读取一个值，并且将其与给定i值进行位相减操作(使用标准指针运算规则)后，替换原值。

声明

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

效果

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

std::atomic_fetch_sub_explicit 非成员函数

从 `atomic<T*>` 实例中原子的读取一个值，并且将其与给定i值进行位相减操作(使用标准指针运算规则)后，替换原值。

声明

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

效果

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

std::atomic<T*>::operator++ 前置递增操作

原子的将*this中存储的值加1(使用标准指针运算规则)，并返回新值。

声明

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

效果

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

std::atomic<T*>::operator++ 后置递增操作

原子的将*this中存储的值加1(使用标准指针运算规则)，并返回旧值。

声明

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

效果

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

std::atomic<T*>::operator-- 前置递减操作

原子的将*this中存储的值减1(使用标准指针运算规则)，并返回新值。

声明

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

效果

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

std::atomic<T*>::operator-- 后置递减操作

原子的将*this中存储的值减1(使用标准指针运算规则)，并返回旧值。

声明

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

效果

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

std::atomic<T*>::operator+= 复合赋值操作

原子的将*this中存储的值与给定值相加(使用标准指针运算规则)，并返回新值。

声明

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

效果

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

std::atomic<T*>::operator-= 复合赋值操作

原子的将*this中存储的值与给定值相减(使用标准指针运算规则)，并返回新值。

声明

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

效果

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

D.4 <future>头文件

`<future>`头文件提供处理异步结果(在其他线程上执行额结果)的工具。

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; // no definition provided

  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类型模板

`std::future`类型模板是为了等待其他线程上的异步结果。其和`std::promise`，`std::packaged_task`类型模板，还有`std::async`函数模板，都是为异步结果准备的工具。只有`std::future`实例可以在任意时间引用异步结果。

`std::future`实例是MoveConstructible(移动构造)和MoveAssignable(移动赋值)，不过不能CopyConstructible(拷贝构造)和CopyAssignable(拷贝赋值)。

类型声明

```
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;

  see description 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);
};
```

std::future 默认构造函数

不使用异步结果构造一个 `std::future` 对象。

声明

```
future() noexcept;
```

效果

构造一个新的 `std::future` 实例。

后置条件

valid()返回false。

抛出

无

std::future 移动构造函数

使用另外一个对象，构造一个 `std::future` 对象，将相关异步结果的所有权转移给新 `std::future` 对象。

声明

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

效果

使用已有对象构造一个新的 `std::future` 对象。

后置条件
已有对象中的异步结果，将于新的对象相关联。然后，解除已有对象和异步之间的关系。`this->valid()` 返回的结果与之前已有对象 `other.valid()` 返回的结果相同。在调用该构造函数后， `other.valid()` 将返回false。

抛出
无

std::future 移动赋值操作

将已有 `std::future` 对象中异步结果的所有权，转移到另一对象当中。

声明

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

效果
在两个 `std::future` 实例中转移异步结果的状态。

后置条件
当执行完赋值操作后， `*this.other` 就与异步结果没有关系了。异步状态(如果有的话)在释放后与 `*this` 相关，并且在最后一次引用后，销毁该状态。 `this->valid()` 返回的结果与之前已有对象 `other.valid()` 返回的结果相同。在调用该构造函数后， `other.valid()` 将返回false。

抛出
无

std::future 析构函数

销毁一个 `std::future` 对象。

声明

```
~future();
```

效果
销毁 `*this` 。如果这是最后一次引用与 `*this` 相关的异步结果，之后就会将该异步结果销毁。

抛出
无

std::future::share 成员函数

构造一个新 `std::shared_future` 实例，并且将 `*this` 异步结果的所有权转移到新的 `std::shared_future` 实例中。

声明

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

效果
如同 shared_future(std::move(*this))。

后置条件
当调用share()成员函数，与 `*this` 相关的异步结果将与新构造的 `std::shared_future` 实例相关。 `this->valid()` 将返回false。

抛出
无

std::future::valid 成员函数

检查 `std::future` 实例是否与一个异步结果相关联。

声明

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

返回
当与异步结果相关时，返回true，否则返回false。

抛出
无

std::future::wait 成员函数

如果与 `*this` 相关的状态包含延迟函数，将调用该函数。否则，会等待 `std::future` 实例中的异步结果准备就绪。

声明

```
void wait();
```

先决条件
`this->valid()` 将会返回true。

效果
当相关状态包含延迟函数，调用延迟函数，并保存返回的结果，或将抛出的异常保存成为异步结果。否则，会阻塞到 `*this` 准备就绪。

抛出
无

std::future::wait_for 成员函数

等待 `std::future` 实例上相关异步结果准备就绪，或超过某个给定的时间。

声明

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

先决条件
`this->valid()` 将会返回true。

效果

如果与 `*this` 相关的异步结果包含一个 `std::async` 调用的延迟函数(还未执行)，那么就不阻塞立即返回。否则将阻塞实例，直到与 `*this` 相关异步结果准备就绪，或超过给定的relative_time时长。

返回

当与 `*this` 相关的异步结果包含一个 `std::async` 调用的延迟函数(还未执行)，返回 `std::future_status::deferred` ；当与 `*this` 相关的异步结果准备就绪，返回 `std::future_status::ready` ；当给定时间超过relative_time时，返回 `std::future_status::timeout` 。

NOTE:线程阻塞的时间可能超多给定的时长。时长尽可能由一个稳定的时钟决定。

抛出

无

std::future::wait_until 成员函数

等待 `std::future` 实例上相关异步结果准备就绪，或超过某个给定的时间。

声明

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

先决条件

this->valid()将返回true。

效果

如果与 `*this` 相关的异步结果包含一个 `std::async` 调用的延迟函数(还未执行)，那么就不阻塞立即返回。否则将阻塞实例，直到与 `*this` 相关异步结果准备就绪，或 `Clock::now()` 返回的时间大于等于absolute_time。

返回

当与 `*this` 相关的异步结果包含一个 `std::async` 调用的延迟函数(还未执行)，返回 `std::future_status::deferred` ；当与 `*this` 相关的异步结果准备就绪，返回 `std::future_status::ready` ； `Clock::now()` 返回的时间大于等于absolute_time，返回 `std::future_status::timeou`

`t` 。

NOTE:这里不保证调用线程会被阻塞多久，只有函数返回 `std::future_status::timeout` ，然后 `Clock::now()` 返回的时间大于等于absolute_time的时候，线程才会解除阻塞。

抛出
无

std::future::get 成员函数

当相关状态包含一个 `std::async` 调用的延迟函数，调用该延迟函数，并返回结果；否则，等待与 `std::future` 实例相关的异步结果准备就绪，之后返回存储的值或异常。

声明

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

先决条件
this->valid()将返回true。

效果
如果*this相关状态包含一个延期函数，那么调用这个函数并返回结果，或将抛出的异常进行传播。

否则，线程就要被阻塞，直到与*this相关的异步结果就绪。当结果存储了一个异常，那么就就会将存储异常抛出。否则，将会返回存储值。

返回
当相关状态包含一个延期函数，那么这个延期函数的结果将被返回。否则，当ResultType为void时，就会按照常规调用返回。如果ResultType是R&(R类型的引用)，存储的引用值将会被返回。否则，存储的值将会返回。

抛出
异常由延期函数，或存储在异步结果中的异常(如果有的话)抛出。

后置条件

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

D.4.2 std::shared_future类型模板

`std::shared_future`类型模板是为了等待其他线程上的异步结果。其和`std::promise`，`std::packaged_task`类型模板，还有`std::async`函数模板，都是为异步结果准备的工具。多个`std::shared_future`实例可以引用同一个异步结果。

`std::shared_future`实例是CopyConstructible(拷贝构造)和CopyAssignable(拷贝赋值)。你也可以同ResultType的`std::future`类型对象，移动构造一个`std::shared_future`类型对象。

访问给定`std::shared_future`实例是非同步的。因此，当有多个线程访问同一个`std::shared_future`实例，且无任何外围同步操作时，这样的访问是不安全的。不过访问关联状态时是同步的，所以多个线程访问多个独立的`std::shared_future`实例，且没有外围同步操作的时候，是安全的。

类型定义

```
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;

  see description get() const;

  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;
};
```

std::shared_future 默认构造函数

不使用关联异步结果，构造一个 `std::shared_future` 对象。

声明

```
shared_future() noexcept;
```

效果

构造一个新的 `std::shared_future` 实例。

后置条件

当新实例构建完成后，调用valid()将返回false。

抛出

无

std::shared_future 移动构造函数

以一个已创建 `std::shared_future` 对象为准，构造 `std::shared_future` 实例，并将使用 `std::shared_future` 对象关联的异步结果的所有权转移到新的实例中。

声明

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

效果

构造一个新 `std::shared_future` 实例。

后置条件

将other对象中关联异步结果的所有权转移到新对象中，这样other对象就没有与之相关联的异步结果了。

抛出
无

std::shared_future 移动对应std::future对象的构造函数

以一个已创建 `std::future` 对象为准，构造 `std::shared_future` 实例，并将使用 `std::shared_future` 对象关联的异步结果的所有权转移到新的实例中。

声明

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

效果
构造一个 `std::shared_future` 对象。

后置条件
将other对象中关联异步结果的所有权转移到新对象中，这样other对象就没有与之相关联的异步结果了。

抛出
无

std::shared_future 拷贝构造函数

以一个已创建 `std::future` 对象为准，构造 `std::shared_future` 实例，并将使用 `std::shared_future` 对象关联的异步结果(如果有的话)拷贝到新创建对象当中，两个对象共享该异步结果。

声明

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

效果
构造一个 `std::shared_future` 对象。

后置条件
将other对象中关联异步结果拷贝到新对象中，与other共享关联的异步结果。

抛出
无

std::shared_future 析构函数

销毁一个 `std::shared_future` 对象。

声明

```
~shared_future();
```

效果
将 `*this` 销毁。如果 `*this` 关联的异步结果与 `std::promise` 或 `std::packaged_task` 不再有关联，那么该函数将会切断 `std::shared_future` 实例与异步结果的联系，并销毁异步结果。

抛出
无

std::shared_future::valid 成员函数

检查 `std::shared_future` 实例是否与一个异步结果相关联。

声明

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

返回
当与异步结果相关时，返回true，否则返回false。

抛出
无

std::shared_future::wait 成员函数

当*this关联状态包含一个延期函数，那么调用这个函数。否则，等待直到与 `std::shared_future` 实例相关的异步结果就绪为止。

声明

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

先决条件

this->valid()将返回true。

效果

当有多个线程调用 `std::shared_future` 实例上的get()和wait()时，实例会序列化的共享同一关联状态。如果关联状态包括一个延期函数，那么第一个调用get()或wait()时就会调用延期函数，并且存储返回值，或将抛出异常以异步结果的方式保存下来。

抛出

无

std::shared_future::wait_for 成员函数

等待 `std::shared_future` 实例上相关异步结果准备就绪，或超过某个给定的时间。

声明

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

先决条件

`this->valid()` 将会返回true。

效果

如果与 `*this` 相关的异步结果包含一个 `std::async` 调用的延期函数(还未执行)，那么就不阻塞立即返回。否则将阻塞实例，直到与 `*this` 相关异步结果准备就绪，或超过给定的relative_time时长。

返回

当与 `*this` 相关的异步结果包含一个 `std::async` 调用的延迟函数(还未执行)，返回 `std::future_status::deferred` ；当与 `*this` 相关的异步结果准备就绪，返回 `std::future_status::ready` ；当给定时间超过relative_time时，返回 `std::future_status::timeout` 。

NOTE:线程阻塞的时间可能超多给定的时长。时长尽可能由一个稳定的时钟决定。

抛出

无

std::shared_future::wait_until 成员函数

等待 `std::future` 实例上相关异步结果准备就绪，或超过某个给定的时间。

声明

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

先决条件

this->valid()将返回true。

效果

如果与 `*this` 相关的异步结果包含一个 `std::async` 调用的延迟函数(还未执行)，那么就不阻塞立即返回。否则将阻塞实例，直到与 `*this` 相关异步结果准备就绪，或 `Clock::now()` 返回的时间大于等于absolute_time。

返回

当与 `*this` 相关的异步结果包含一个 `std::async` 调用的延迟函数(还未执行)，返回 `std::future_status::deferred` ；当与 `*this` 相关的异步结果准备就绪，返回 `std::future_status::ready` ； `Clock::now()` 返回的时间大于等于absolute_time，返回 `std::future_status::timeout` 。

NOTE:这里不保证调用线程会被阻塞多久，只有函数返回 `std::future_status::timeout` ，然后 `Clock::now()` 返回的时间大于等于absolute_time的时候，线程才会解除阻塞。

抛出
无

std::shared_future::get 成员函数

当相关状态包含一个 `std::async` 调用的延迟函数，调用该延迟函数，并返回结果；否则，等待与 `std::shared_future` 实例相关的异步结果准备就绪，之后返回存储的值或异常。

声明

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

先决条件
this->valid()将返回true。

效果
当有多个线程调用 `std::shared_future` 实例上的get()和wait()时，实例会序列化的共享同一关联状态。如果关联状态包括一个延期函数，那么第一个调用get()或wait()时就会调用延期函数，并且存储返回值，或将抛出异常以异步结果的方式保存下来。

阻塞会知道*this关联的异步结果就绪后解除。当异步结果存储了一个一行，那么就会抛出这个异常。否则，返回存储的值。

返回
当ResultType为void时，就会按照常规调用返回。如果ResultType是R&(R类型的引用)，存储的引用值将会被返回。否则，返回存储值的const引用。

抛出
抛出存储的异常(如果有的话)。

D.4.3 std::packaged_task类型模板

`std::packaged_task` 类型模板可打包一个函数或其他可调用对象，所以当函数通过 `std::packaged_task` 实例被调用时，结果将会作为异步结果。这个结果可以通过检索 `std::future` 实例来查找。

`std::packaged_task` 实例是可以MoveConstructible(移动构造)和MoveAssignable(移动赋值),不过不能CopyConstructible(拷贝构造)和CopyAssignable(拷贝赋值)。

类型定义

```
template<typename FunctionType>
class packaged_task; // undefined

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();
};
```

std::packaged_task 默认构造函数

构造一个 `std::packaged_task` 对象。

声明

```
packaged_task() noexcept;
```

效果
不使用关联任务或异步结果来构造一个 `std::packaged_task` 对象。

抛出
无

std::packaged_task 通过可调用对象构造

使用关联任务和异步结果，构造一个 `std::packaged_task` 对象。

声明

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

先决条件
表达式 `func(args...)` 必须是合法的，并且在 `args...` 中的args-i参数，必须是 `ArgTypes...` 中ArgTypes-i类型的一个值。且返回值必须可转换为ResultType。

效果
使用ResultType类型的关联异步结果，构造一个 `std::packaged_task` 对象，异步结果是未就绪的，并且Callable类型相关的任务是对func的一个拷贝。

抛出
当构造函数无法为异步结果分配出内存时，会抛出 `std::bad_alloc` 类型的异常。其他异常会在使用Callable类型的拷贝或移动构造过程中抛出。

std::packaged_task 通过有分配器的可调用对象构造

使用关联任务和异步结果，构造一个 `std::packaged_task` 对象。使用以提供的分配器为关联任务和异步结果分配内存。

声明

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

先决条件

表达式 `func(args...)` 必须是合法的，并且在 `args...` 中的args-i参数，必须是 `ArgTypes...` 中ArgTypes-i类型的一个值。且返回值必须可转换为ResultType。

效果

使用ResultType类型的关联异步结果，构造一个 `std::packaged_task` 对象，异步结果是未就绪的，并且Callable类型相关的任务是对func的一个拷贝。异步结果和任务的内存通过内存分配器alloc进行分配，或进行拷贝。

抛出

当构造函数无法为异步结果分配出内存时，会抛出 `std::bad_alloc` 类型的异常。其他异常会在使用Callable类型的拷贝或移动构造过程中抛出。

std::packaged_task 移动构造函数

通过一个 `std::packaged_task` 对象构建另一个，将与已存在的 `std::packaged_task` 相关的异步结果和任务的所有权转移到新构建的对象当中。

声明

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

效果

构建一个新的 `std::packaged_task` 实例。

后置条件

通过other构建新的 `std::packaged_task` 对象。在新对象构建完成后，other与其之前相关联的异步结果就没有任何关系了。

抛出

无

std::packaged_task 移动赋值操作

将一个 `std::packaged_task` 对象相关的异步结果的所有权转移到另外一个。

声明

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

效果

将other相关异步结果和任务的所有权转移到 `*this` 中，并且切断异步结果和任务与other对象的关联，如同 `std::packaged_task(other).swap(*this)` 。

后置条件

与other相关的异步结果与任务移动转移，使*this.other无关联的异步结果。

返回

```
*this
```

抛出

无

std::packaged_task::swap 成员函数

将两个 `std::packaged_task` 对象所关联的异步结果的所有权进行交换。

声明

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

效果

将other和*this关联的异步结果与任务进行交换。

后置条件

将与other关联的异步结果和任务，通过调用swap的方式，与*this相交换。

抛出

无

std::packaged_task 析构函数

销毁一个 `std::packaged_task` 对象。

声明

```
~packaged_task();
```

效果

将 `*this` 销毁。如果 `*this` 有关联的异步结果，并且结果不是一个已存储的任务或异常，那么异步结果状态将会变为就绪，伴随就绪的是一个 `std::future_error` 异常和错误码 `std::future_errc::broken_promise` 。

抛出

无

std::packaged_task::get_future 成员函数

在*this相关异步结果中，检索一个 `std::future` 实例。

声明

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

先决条件

*this具有关联异步结果。

返回

一个与*this关联异构结果相关的一个 `std::future` 实例。

抛出

如果一个 `std::future` 已经通过get_future()获取了异步结果，在抛出 `std::future_error` 异常时，错误码是 `std::future_errc::future_already_retrieved`

std::packaged_task::reset 成员函数

将一个 `std::packaged_task` 对实例与一个新的异步结果相关联。

声明

```
void reset();
```

先决条件
*this具有关联的异步任务。

效果
如同 `*this=packaged_task(std::move(f))`，f是*this中已存储的关联任务。

抛出
如果内存不足以分配给新的异构结果，那么将会抛出 `std::bad_alloc` 类异常。

std::packaged_task::valid 成员函数

检查*this中是都具有关联任务和异步结果。

声明

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

返回
当*this具有相关任务和异步结构，返回true；否则，返回false。

抛出
无

std::packaged_task::operator() 函数调用操作

调用一个 `std::packaged_task` 实例中的相关任务，并且存储返回值，或将异常存储到异常结果当中。

声明

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

先决条件

*this具有相关任务。

效果

像 `INVOKE(func,args...)` 那要调用相关的函数func。如果返回征程，那么将会存储到*this相关的异步结果中。当返回结果是一个异常，将这个异常存储到*this相关的异步结果中。

后置条件

*this相关联的异步结果状态为就绪，并且存储了一个值或异常。所有阻塞线程，在等待到异步结果的时候被解除阻塞。

抛出

当异步结果已经存储了一个值或异常，那么将抛出一个 `std::future_error` 异常，错误码为 `std::future_errc::promise_already_satisfied` 。

同步

`std::future<ResultType>::get()` 或 `std::shared_future<ResultType>::get()` 的成功调用，代表同步操作的成功，函数将会检索异步结果中的值或异常。

std::packaged_task::make_ready_at_thread_exit 成员函数

调用一个 `std::packaged_task` 实例中的相关任务，并且存储返回值，或将异常存储到异常结果当中，直到线程退出时，将相关异步结果的状态置为就绪。

声明

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

先决条件

*this具有相关任务。

效果

像 `INVOKE(func,args...)` 那要调用相关的函数func。如果返回征程，那么将会存储到 `*this` 相关的异步结果中。当返回结果是一个异常，将这个异常存储到 `*this` 相关的异步结果中。当当前线程退出的时候，可调配相关异步状态为就绪。

后置条件

*this的异步结果中已经存储了一个值或一个异常，不过在当前线程退出的时候，这个结果都是非就绪的。当当前线程退出时，阻塞等待异步结果的线程将会被解除阻塞。

抛出

当异步结果已经存储了一个值或异常，那么将抛出一个 `std::future_error` 异常，错误码为 `std::future_errc::promise_already_satisfied` 。当无关联异步状态时，抛出 `std::future_error` 异常，错误码为 `std::future_errc::no_state` 。

同步

`std::future<ResultType>::get()` 或 `std::shared_future<ResultType>::get()` 在线程上的成功调用，代表同步操作的成功，函数将会检索异步结果中的值或异常。

D.4.4 std::promise类型模板

`std::promise` 类型模板提供设置异步结果的方法，这样其他线程就可以通过 `std::future` 实例来索引该结果。

ResultType模板参数，该类型可以存储异步结果。

`std::promise` 实例中的异步结果与某个 `srd::future` 实例相关联，并且可以通过调用get_future()成员函数来获取这个 `srd::future` 实例。ResultType类型的异步结果，可以通过set_value()成员函数对存储值进行设置，或者使用set_exception()将对应异常设置进异步结果中。

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

类型定义

```
template<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(see description);
  void set_exception(std::exception_ptr p);
};
```

std::promise 默认构造函数

构造一个 `std::promise` 对象。

声明

```
promise();
```

效果

使用ResultType类型的相关异步结果来构造 `std::promise` 实例，不过异步结果并未就绪。

抛出

当没有足够内存为异步结果进行分配，那么将抛出 `std::bad_alloc` 型异常。

std::promise 带分配器的构造函数

构造一个 `std::promise` 对象，使用提供的分配器来为相关异步结果分配内存。

声明

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

效果

使用ResultType类型的相关异步结果来构造 `std::promise` 实例，不过异步结果并未就绪。异步结果的内存由alloc分配器来分配。

抛出
当分配器为异步结果分配内存时，如有抛出异常，就为该函数抛出的异常。

std::promise 移动构造函数

通过另一个已存在对象，构造一个 `std::promise` 对象。将已存在对象中的相关异步结果的所有权转移到新创建的 `std::promise` 对象当中。

声明

```
promise(promise&& other) noexcept;
```

效果
构造一个 `std::promise` 实例。

后置条件
当使用other来构造一个新的实例，那么other中相关异构结果的所有权将转移到新创建的对象上。之后，other将无关联异步结果。

抛出
无

std::promise 移动赋值操作符

在两个 `std::promise` 实例中转移异步结果的所有权。

声明

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

效果
在other和 `*this` 之间进行异步结果所有权的转移。当 `*this` 已经有关联的异步结果，那么该异步结果的状态将会为就绪态，且伴随一个 `std::future_error` 类型异常，错误码为 `std::future_errc::broken_promise` 。

后置条件
将other中关联的异步结果转移到*this当中。other中将无关联异步结果。

返回

```
*this
```

抛出
无

std::promise::swap 成员函数

将两个 `std::promise` 实例中的关联异步结果进行交换。

声明

```
void swap(promise& other);
```

效果
交换other和*this当中的关联异步结果。

后置条件
当swap使用other时，other中的异步结果就会与*this中关联异步结果相交换。二者返回来亦然。

抛出
无

std::promise 析构函数

销毁 `std::promise` 对象。

声明

```
~promise();
```

效果
销毁 `*this` 。当 `*this` 具有关联的异步结果，并且结果中没有存储值或异常，那么结果将会置为就绪，伴随一个 `std::future_error` 异常，错误码为 `std::future_errc::broken_promise` 。

抛出
无

std::promise::get_future 成员函数

通过*this关联的异步结果，检索出所要的 `std::future` 实例。

声明

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

先决条件
*this具有关联异步结果。

返回
与*this关联异步结果关联的 `std::future` 实例。

抛出
当 `std::future` 已经通过get_future()获取过了，将会抛出一个 `std::future_error` 类型异常，伴随的错误码为 `std::future_errc::future_already_retrieved` 。

std::promise::set_value 成员函数

存储一个值到与*this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);
```

先决条件
*this具有关联异步结果。

效果
当ResultType不是void型，就存储r到*this相关的异步结果当中。

后置条件

*this相关的异步结果的状态为就绪，且将值存入。任意等待异步结果的阻塞线程将解除阻塞。

抛出

当异步结果已经存有一个值或一个异常，那么将抛出`std::future_error`型异常，伴随错误码为`std::future_errc::promise_already_satisfied`。r的拷贝构造或移动构造抛出的异常，即为本函数抛出的异常。

同步

并发调用set_value()和set_exception()的线程将被序列化。要想成功的调用set_exception()，需要在之前调用`std::future<Result-Type>::get()`或`std::shared_future<ResultType>::get()`，这两个函数将会查找已存储的异常。

std::promise::set_value_at_thread_exit 成员函数

存储一个值到与*this关联的异步结果中，到线程退出时，异步结果的状态会被设置为就绪。

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);
```

先决条件

*this具有关联异步结果。

效果

当ResultType不是void型，就存储r到*this相关的异步结果当中。标记异步结果为“已存储值”。当前线程退出时，会安排相关异步结果的状态为就绪。

后置条件

将值存入*this相关的异步结果，且直到当前线程退出时，异步结果状态被置为就绪。任何等待异步结果的阻塞线程将解除阻塞。

抛出

当异步结果已经存有一个值或一个异常，那么将抛出`std::future_error`型异常，伴随错误码为`std::future_errc::promise_already_satisfied`。r的拷贝构造或移动构造抛出的异常，即为本函数抛出的异常。

同步

并发调用set_value(), set_value_at_thread_exit(), set_exception()和set_exception_at_thread_exit()的线程将被序列化。要想成功的调用set_exception()，需要在之前调用`std::future<Result-Type>::get()`或`std::shared_future<ResultType>::get()`，这两个函数将会查找已存储的异常。

std::promise::set_exception 成员函数

存储一个异常到与*this关联的异步结果中。

声明

```
1 void set_exception(std::exception_ptr e);
```

先决条件

*this具有关联异步结果。(bool)e为true。

效果

将e存储到*this相关的异步结果中。

后置条件

在存储异常后，*this相关的异步结果的状态将置为继续。任何等待异步结果的阻塞线程将解除阻塞。

抛出

当异步结果已经存有一个值或一个异常，那么将抛出`std::future_error`型异常，伴随错误码为`std::future_errc::promise_already_satisfied`。

同步

并发调用set_value()和set_exception()的线程将被序列化。要想成功的调用set_exception()，需要在之前调用`std::future<Result-Type>::get()`或`std::shared_future<ResultType>::get()`，这两个函数将会查找已存储的异常。

std::promise::set_exception_at_thread_exit 成员函数

存储一个异常到与*this关联的异步结果中，知道当前线程退出，异步结果被置为就绪。

声明

```
void set_exception_at_thread_exit(std::exception_ptr e);
```

先决条件

*this具有关联异步结果。(bool)e为true。

效果

将e存储到*this相关的异步结果中。标记异步结果为“已存储值”。当前线程退出时，会安排相关异步结果的状态为就绪。

后置条件

将值存入*this相关的异步结果，且直到当前线程退出时，异步结果状态被置为就绪。任何等待异步结果的阻塞线程将解除阻塞。

抛出

当异步结果已经存有一个值或一个异常，那么将抛出 `std::future_error` 型异常，伴随错误码为 `std::future_errc::promise_already_satisfied` 。

同步

并发调用set_value(), set_value_at_thread_exit(), set_exception()和set_exception_at_thread_exit()的线程将被序列化。要想成功的调用set_exception()，需要在之前调用 `std::future<Result-Type>::get()` 或 `std::shared_future<ResultType>::get()` ，这两个函数将会查找已存储的异常。

D.4.5 std::async函数模板

`std::async` 能够简单的使用可用的硬件并行来运行自身包含的异步任务。当调用 `std::async` 返回一个包含任务结果的 `std::future` 对象。根据投放策略，任务在其所在线程上是异步运行的，当有线程调用了这个future对象的wait()和get()成员函数，则该任务会同步运行。

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);
```

先决条件

表达式 `INVOKE(func,args)` 能都为func提供合法的值和args。Callable和Args的所有成员都可MoveConstructible(可移动构造)。

效果

在内部存储中拷贝构造 `func` 和 `arg...` (分别使用fff和xyz...进行表示)。

当policy是 `std::launch::async` ,运行 `INVOKE(fff,xyz...)` 在所在线程上。当这个线程完成时，返回的 `std::future` 状态将会为就绪态，并且之后会返回对应的值或异常(由调用函数抛出)。析构函数会等待返回的 `std::future` 相关异步状态为就绪时，才解除阻塞。

当policy是 `std::launch::deferred` ，fff和xyx...都会作为延期函数调用，存储在返回的 `std::future` 。首次调用future的wait()或get()成员函数，将会共享相关状态，之后执行的 `INVOKE(fff,xyz...)` 与调用wait()或get()函数的线程同步执行。

执行 `INVOKE(fff,xyz...)` 后，在调用 `std::future` 的成员函数get()时，就会有值返回或有异常抛出。

当policy是 `std::launch::async | std::launch::deferred` 或是policy参数被省略，其行为如同已指定的 `std::launch::async` 或 `std::launch::deferred` 。具体实现将会通过逐渐递增的方式(call-by-call basis)最大化利用可用的硬件并行，并避免超限分配的问题。

在所有的情况下， `std::async` 调用都会直接返回。

同步

完成函数调用的先行条件是，需要通过调用 `std::future` 和 `std::shared_future` 实例的wait(),get(),wait_for()或wait_until()，返回的对象与 `std::async` 返回的 `std::future` 对象关联的状态相同才算成功。就 `std::launch::async` 这个policy来说，在完成线程上的函数前，也需要先行对上面的函数调用后，成功的返回才行。

抛出

当内部存储无法分配所需的空间，将抛出 `std::bad_alloc` 类型异常；否则，当效果没有达到，或任何异常在构造fff和xyz...发生时，抛出 `std::future_error` 异常。

D.5 <mutex>头文件

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

头文件内容

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

  struct adopt_lock_t;
  struct defer_lock_t;
  struct try_to_lock_t;

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

  template<typename LockableType>
  class lock_guard;

  template<typename LockableType>
  class unique_lock;

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

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

  struct once_flag;

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

D.5.1 std::mutex类

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

`std::mutex` 符合Lockable的需求。

类型定义

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

  constexpr mutex() noexcept;
  ~mutex();

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

std::mutex 默认构造函数

构造一个 `std::mutex` 对象。

声明

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

效果

构造一个 `std::mutex` 实例。

后置条件

新构造的 `std::mutex` 对象是未锁的。

抛出

无

std::mutex 析构函数

销毁一个 `std::mutex` 对象。

声明

```
~mutex();
```

先决条件
*this必须是未锁的。

效果
销毁*this。

抛出
无

std::mutex::lock 成员函数

为当前线程获取 `std::mutex` 上的锁。

声明

```
void lock();
```

先决条件
*this上必须没有持有一个锁。

效果
阻塞当前线程，知道*this获取锁。

后置条件
*this被调用线程锁住。

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

std::mutex::try_lock 成员函数

尝试为当前线程获取 `std::mutex` 上的锁。

声明

```
bool try_lock();
```

先决条件
*this上必须没有持有一个锁。

效果
尝试为当前线程*this获取上的锁，失败时当前线程不会被阻塞。

返回
当调用线程获取锁时，返回true。

后置条件
当*this被调用线程锁住，则返回true。

抛出
无

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

std::mutex::unlock 成员函数

释放当前线程获取的 `std::mutex` 锁。

声明

```
void unlock();
```

先决条件
*this上必须持有一个锁。

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

后置条件

调用线程不在拥有*this上的锁。

抛出

无

D.5.2 std::recursive_mutex类

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

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

`std::recursive_mutex`符合Lockable的需求。

类型定义

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

  recursive_mutex() noexcept;
 ~recursive_mutex();

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

std::recursive_mutex 默认构造函数

构造一个 `std::recursive_mutex` 对象。

声明

```
recursive_mutex() noexcept;
```

效果
构造一个 `std::recursive_mutex` 实例。

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

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

std::recursive_mutex 析构函数

销毁一个 `std::recursive_mutex` 对象。

声明

```
~recursive_mutex();
```

先决条件
*this必须是未锁的。

效果
销毁*this。

抛出
无

std::recursive_mutex::lock 成员函数

为当前线程获取 `std::recursive_mutex` 上的锁。

声明

```
void lock();
```

效果
阻塞线程，直到获取*this上的锁。

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

抛出
当有错误产生，将抛出 `std::system_error` 异常。

std::recursive_mutex::try_lock 成员函数

尝试为当前线程获取 `std::recursive_mutex` 上的锁。

声明

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

效果
尝试为当前线程*this获取上的锁，失败时当前线程不会被阻塞。

返回
当调用线程获取锁时，返回true；否则，返回false。

后置条件
当*this被调用线程锁住，则返回true。

抛出
无

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

std::recursive_mutex::unlock 成员函数

释放当前线程获取的`std::recursive_mutex`锁。

声明

```
1 void unlock();
```

先决条件
*this上必须持有一个锁。

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

后置条件
`*this`上锁的计数会在该函数调用后减一。

抛出
无

D.5.3 std::timed_mutex类

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

`std::timed_mutex`符合TimedLockable的需求。

类型定义

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

  timed_mutex();
  ~timed_mutex();

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

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

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

std::timed_mutex 默认构造函数

构造一个 `std::timed_mutex` 对象。

声明

```
timed_mutex();
```

效果

构造一个 `std::timed_mutex` 实例。

后置条件

新构造一个未上锁的 `std::timed_mutex` 对象。

抛出

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

std::timed_mutex 析构函数

销毁一个 `std::timed_mutex` 对象。

声明

```
~timed_mutex();
```

先决条件
*this必须没有上锁。

效果
销毁*this。

抛出
无

std::timed_mutex::lock 成员函数

为当前线程获取 `std::timed_mutex` 上的锁。

声明

```
void lock();
```

先决条件
调用线程不能已经持有*this上的锁。

效果
阻塞当前线程，直到获取到*this上的锁。

后置条件
*this被调用线程锁住。

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

std::timed_mutex::try_lock 成员函数

尝试获取为当前线程获取 `std::timed_mutex` 上的锁。

声明

```
bool try_lock();
```

先决条件
调用线程不能已经持有*this上的锁。

效果
尝试获取*this上的锁，当获取失败时，不阻塞调用线程。

返回
当锁被调用线程获取，返回true；反之，返回false。

后置条件
当函数返回为true，*this则被当前线程锁住。

抛出
无

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

std::timed_mutex::try_lock_for 成员函数

尝试获取为当前线程获取 `std::timed_mutex` 上的锁。

声明

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

先决条件
调用线程不能已经持有*this上的锁。

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

返回
当锁被调用线程获取，返回true；反之，返回false。

后置条件
当函数返回为true，*this则被当前线程锁住。

抛出
无

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

std::timed_mutex::try_lock_until 成员函数

尝试获取为当前线程获取 `std::timed_mutex` 上的锁。

声明

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

先决条件
调用线程不能已经持有*this上的锁。

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

返回
当锁被调用线程获取，返回true；反之，返回false。

后置条件
当函数返回为true，*this则被当前线程锁住。

抛出
无

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

std::timed_mutex::unlock 成员函数

将当前线程持有 `std::timed_mutex` 对象上的锁进行释放。

声明

```
1 void unlock();
```

先决条件
调用线程已经持有*this上的锁。

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

后置条件
*this未被调用线程上锁。

抛出
无

D.5.4 std::recursive_timed_mutex类

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

其他线程才能获取这个锁被获取(不管是调用的哪个函数)。

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

`std::recursive_timed_mutex` 符合TimedLockable的需求。

类型定义

```
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);
};
```

std::recursive_timed_mutex 默认构造函数

构造一个 `std::recursive_timed_mutex` 对象。

声明

```
recursive_timed_mutex();
```

效果

构造一个 `std::recursive_timed_mutex` 实例。

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

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

std::recursive_timed_mutex 析构函数

析构一个 `std::recursive_timed_mutex` 对象。

声明

```
~recursive_timed_mutex();
```

先决条件
*this不能上锁。

效果
销毁*this。

抛出
无

std::recursive_timed_mutex::lock 成员函数

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

声明

```
void lock();
```

先决条件
*this上的锁不能被线程调用。

效果
阻塞当前线程，直到获取*this上的锁。

后置条件

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

抛出

当错误出现时，抛出 `std::system_error` 类型异常。

std::recursive_timed_mutex::try_lock 成员函数

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

声明

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

效果

尝试获取*this上的锁，当获取失败时，直接不阻塞线程。

返回

当调用线程获取了锁，返回true，否则返回false。

后置条件

当函数返回true， `*this` 会被调用线程锁住。

抛出

无

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

std::recursive_timed_mutex::try_lock_for 成员函数

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

声明

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

效果

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

返回

当调用线程获取了锁，返回true，否则返回false。

后置条件

当函数返回true，`*this`会被调用线程锁住。

抛出

无

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

std::recursive_timed_mutex::try_lock_until 成员函数

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

声明

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

效果

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

返回
当调用线程获取了锁，返回true，否则返回false。

后置条件
当函数返回true，`*this`会被调用线程锁住。

抛出
无

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

std::recursive_timed_mutex::unlock 成员函数

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

声明

```
void unlock();
```

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

后置条件
调用线程*this上锁的计数减一。

抛出
无

D.5.5 std::lock_guard类型模板

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

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

类型定义

```
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;
};
```

std::lock_guard 自动上锁的构造函数

使用互斥量构造一个`std::lock_guard`实例。

声明

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

效果

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

抛出

m.lock()抛出的任何异常。

后置条件

*this拥有m上的锁。

std::lock_guard 获取锁的构造函数

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

声明

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

先决条件
调用线程必须拥有m上的锁。

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

抛出
无

后置条件
*this拥有m上的锁。

std::lock_guard 析构函数

销毁一个 `std::lock_guard` 实例，并且解锁相关互斥量。

声明

```
~lock_guard();
```

效果
当*this被创建后，调用m.unlock()。

抛出
无

D.5.6 std::unique_lock类型模板

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

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

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

类型定义

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

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

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

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

  ~unique_lock();

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

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

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

  void swap(unique_lock& other) noexcept;

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

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

std::unique_lock 默认构造函数

不使用相关互斥量，构造一个 `std::unique_lock` 实例。

声明

```
unique_lock() noexcept;
```

效果

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

后置条件

this->mutex()NULL, this->owns_lock()false.

std::unique_lock 自动上锁的构造函数

使用相关互斥量，构造一个 `std::unique_lock` 实例。

声明

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

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

抛出
m.lock()抛出的任何异常。

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

std::unique_lock 获取锁的构造函数

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

声明

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

先决条件
调用线程必须持有m上的锁。

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

抛出
无

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

std::unique_lock 递延锁的构造函数

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

声明

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

效果
构造的 `std::unique_lock` 实例引用了提供的互斥量。

抛出
无

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

std::unique_lock 尝试获取锁的构造函数

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

声明

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

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

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

抛出
无

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

std::unique_lock 在给定时长内尝试获取锁的构造函数

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

声明

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

先决条件

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

效果

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

抛出

无

后置条件

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

std::unique_lock 在给定时间点内尝试获取锁的构造函数

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

声明

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

先决条件

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

效果

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

抛出

无

后置条件

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

std::unique_lock 移动构造函数

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

声明

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

先决条件

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

效果

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

后置条件

对于新构建的 `std::unique_lock` 对象x，x.mutex等价与在构造函数调用前的other.mutex()，并且x.owns_lock()等价于函数调用前的other.owns_lock()。在调用函数后，other.mutex()==NULL，other.owns_lock()=false。

抛出

无

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

std::unique_lock 移动赋值操作

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

声明

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

效果

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

后置条件

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

抛出

无

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

std::unique_lock 析构函数

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

声明

```
~unique_lock();
```

效果

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

抛出

无

std::unique_lock::swap 成员函数

交换`std::unique_lock`实例中相关的所有权。

声明

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

效果

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

抛出

无

std::unique_lock 上非成员函数swap

交换`std::unique_lock`实例中相关的所有权。

声明

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

效果

lhs.swap(rhs)

抛出

无

std::unique_lock::lock 成员函数

获取与*this相关互斥量上的锁。

声明

```
1 void lock();
```

先决条件

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

效果

调用this->mutex()->lock()。

抛出

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

后置条件

this->owns_lock()==true。

std::unique_lock::try_lock 成员函数

尝试获取与*this相关互斥量上的锁。

声明

```
bool try_lock();
```

先决条件

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

效果

调用this->mutex()->try_lock()。

抛出

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

后置条件

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

std::unique_lock::unlock 成员函数

释放与*this相关互斥量上的锁。

声明

```
void unlock();
```

先决条件

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

抛出

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

后置条件

this->owns_lock()==false。

std::unique_lock::try_lock_for 成员函数

在指定时间内尝试获取与*this相关互斥量上的锁。

声明

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

先决条件

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

效果

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

返回

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

抛出

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

后置条件

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

std::unique_lock::try_lock_until 成员函数

在指定时间点尝试获取与*this相关互斥量上的锁。

声明

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

先决条件

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

效果

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

返回

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

抛出

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

后置条件

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

std::unique_lock::operator bool成员函数

检查*this是否拥有一个互斥量上的锁。

声明

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

返回
this->owns_lock()

抛出
无

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

std::unique_lock::owns_lock 成员函数

检查*this是否拥有一个互斥量上的锁。

声明

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

返回
当*this持有一个互斥量的锁，返回true；否则，返回false。

抛出
无

std::unique_lock::mutex 成员函数

当*this具有相关互斥量时，返回这个互斥量

声明

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

返回
当*this有相关互斥量，则返回该互斥量；否则，返回NULL。

抛出
无

std::unique_lock::release 成员函数

当*this具有相关互斥量时，返回这个互斥量，并将这个互斥量进行释放。

声明

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

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

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

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

抛出
无

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

D.5.7 std::lock函数模板

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

声明

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

先决条件

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

效果

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

后置条件

当前线程拥有提供的所有可锁对象上的锁。

抛出

任何lock(), try_lock()和unlock()抛出的异常。

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

D.5.8 std::try_lock函数模板

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

声明

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

先决条件

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

效果

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

返回

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

后置条件

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

抛出

try_lock()抛出的任何异常。

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

D.5.9 std::once_flag类

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

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

类型定义

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

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

std::once_flag 默认构造函数

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

声明

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

效果

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

D.5.10 std::call_once函数模板

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

声明

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

先决条件

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

效果

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

同步

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

抛出

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

D.6 <ratio>头文件

`<ratio>` 头文件提供在编译时进行的计算。

头文件内容

```
namespace std
{
  template<intmax_t N,intmax_t D=1>
  class ratio;

  // ratio arithmetic
  template <class R1, class R2>
  using ratio_add = see description;

  template <class R1, class R2>
  using ratio_subtract = see description;

  template <class R1, class R2>
  using ratio_multiply = see description;

  template <class R1, class R2>
  using ratio_divide = see description;

  // ratio comparison
  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类型模板

`std::ratio`类型模板提供了一种对在编译时进行计算的机制，通过调用合理的数，例如：半(`std::ratio<1,2>`),2/3(std::ratio<2, 3>)或15/43(std::ratio<15, 43>)。其使用在C++标准库内部，用于初始化`std::chrono::duration`类型模板。

类型定义

```
template <intmax_t N, intmax_t D = 1>
class ratio
{
public:
  typedef ratio<num, den> type;
  static constexpr intmax_t num= see below;
  static constexpr intmax_t den= see below;
};
```

要求

D不能为0。

描述

num和den分别为分子和分母，构造分数N/D。den总是正数。当N和D的符号相同，那么num为正数；否则num为负数。

例子

```
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` 模板别名提供了两个 `std::ratio` 在编译时相加的机制(使用有理计算)。

定义

```
template <class R1, class R2>
using ratio_add = std::ratio<see below>;
```

先决条件

R1和R2必须使用 `std::ratio` 进行初始化。

效果

ratio_add<R1, R2>被定义为一个别名，如果两数可以计算，且无溢出，该类型可以表示两个 `std::ratio` 对象R1和R2的和。如果计算出来的结果溢出了，那么程序里面就有问题了。在算术溢出的情况下， `std::ratio_add<R1, R2>` 应该应该与 `std::ratio<R1::num * R2::den + R2::num * R1::den, R1::den * R2::den>` 相同。

例子

```
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` 模板别名提供两个 `std::ratio` 数在编译时进行相减(使用有理计算)。

定义

```
template <class R1, class R2>
using ratio_subtract = std::ratio<see below>;
```

先决条件

R1和R2必须使用 `std::ratio` 进行初始化。

效果

ratio_add<R1, R2>被定义为一个别名，如果两数可以计算，且无溢出，该类型可以表示两个 `std::ratio` 对象R1和R2的和。如果计算出来的结果溢出了，那么程序里面就有问题了。在算术溢出的情况下， `std::ratio_subtract<R1, R2>` 应该应该与 `std::ratio<R1::num * R2::den - R2::num * R1::den, R1::den * R2::den>` 相同。

例子

```
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` 模板别名提供两个 `std::ratio` 数在编译时进行相乘(使用有理计算)。

定义

```
template <class R1, class R2>
using ratio_multiply = std::ratio<see below>;
```

先决条件

R1和R2必须使用 `std::ratio` 进行初始化。

效果

ratio_add<R1, R2>被定义为一个别名，如果两数可以计算，且无溢出，该类型可以表示两个 `std::ratio` 对象R1和R2的和。如果计算出来的结果溢出了，那么程序里面就有问题了。在算术溢出的情况下， `std::ratio_multiply<R1, R2>` 应该应该与 `std::ratio<R1::num * R2::num, R1::den * R2::den>` 相同。

例子

```
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` 模板别名提供两个 `std::ratio` 数在编译时进行相除(使用有理计算)。

定义

```
template <class R1, class R2>
using ratio_multiply = std::ratio<see below>;
```

先决条件

R1和R2必须使用 `std::ratio` 进行初始化。

效果

ratio_add<R1, R2>被定义为一个别名，如果两数可以计算，且无溢出，该类型可以表示两个 `std::ratio` 对象R1和R2的和。如果计算出来的结果溢出了，那么程序里面就有问题了。在算术溢出的情况下， `std::ratio_multiply<R1, R2>` 应该应该与 `std::ratio<R1::num * R2::num * R2::den, R1::den * R2::den>` 相同。

例子

```
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类型模板

`std::ratio_equal` 类型模板提供在编译时比较两个 `std::ratio` 数(使用有理计算)。

类型定义

```
template <class R1, class R2>
class ratio_equal:
  public std::integral_constant<
    bool,(R1::num == R2::num) && (R1::den == R2::den)>
{};
```

先决条件

R1和R2必须使用 `std::ratio` 进行初始化。

例子

```
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类型模板

`std::ratio_not_equal` 类型模板提供在编译时比较两个 `std::ratio` 数(使用有理计算)。

类型定义

```
template <class R1, class R2>
class ratio_not_equal:
  public std::integral_constant<bool,!ratio_equal<R1,R2>::value>
{};
```

先决条件

R1和R2必须使用 `std::ratio` 进行初始化。

例子

```
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类型模板

`std::ratio_less` 类型模板提供在编译时比较两个 `std::ratio` 数(使用有理计算)。

类型定义

```
template <class R1, class R2>
class ratio_less:
  public std::integral_constant<bool,see below>
{};
```

先决条件

R1和R2必须使用 `std::ratio` 进行初始化。

效果

std::ratio_less<R1,R2>可通过 `std::integral_constant<bool, value >` 导出，这里value为 `(R1::num*R2::den) < (R2::num*R1::den)` 。如果有可能，需要实现使用一种机制来避免计算结果已出。当溢出发生，那么程序中就肯定有错误。

例子

```
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类型模板

`std::ratio_greater` 类型模板提供在编译时比较两个 `std::ratio` 数(使用有理计算)。

类型定义

```
template <class R1, class R2>
class ratio_greater:
  public std::integral_constant<bool,ratio_less<R2,R1>::value>
{};
```

先决条件

R1和R2必须使用 `std::ratio` 进行初始化。

D.6.10 std::ratio_less_equal类型模板

`std::ratio_less_equal` 类型模板提供在编译时比较两个 `std::ratio` 数(使用有理计算)。

类型定义

```
template <class R1, class R2>
class ratio_less_equal:
  public std::integral_constant<bool,!ratio_less<R2,R1>::value>
{};
```

先决条件

R1和R2必须使用 `std::ratio` 进行初始化。

D.6.11 std::ratio_greater_equal类型模板

`std::ratio_greater_equal` 类型模板提供在编译时比较两个 `std::ratio` 数(使用有理计算)。

类型定义

```
template <class R1, class R2>
class ratio_greater_equal:
  public std::integral_constant<bool,!ratio_less<R1,R2>::value>
{};
```

先决条件

R1和R2必须使用 `std::ratio` 进行初始化。

D.7 <thread>头文件

`<thread>`头文件提供了管理和辨别线程的工具，并且提供函数，可让当前线程休眠。

头文件内容

```
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`用来管理线程的执行。其提供让新的线程执行或执行，也提供对线程的识别，以及提供其他函数用于管理线程的执行。

```
class thread
{
public:
  // Types
  class id;
  typedef implementation-defined native_handle_type; // optional

  // Construction and Destruction
  thread() noexcept;
  ~thread();
```

```
  template<typename Callable,typename Args...>
  explicit thread(Callable&& func,Args&&... args);

  // Copying and Moving
  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);
```

std::thread::id 类

可以通过 `std::thread::id` 实例对执行线程进行识别。

类型定义

```
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>&
```

```
16 operator<< (basic_ostream<charT, traits>&& out, thread::id id);
```

Notes

`std::thread::id` 的值可以识别不同的执行，每个 `std::thread::id` 默认构造出来的值都不一样，不同值代表不同的执行线程。

`std::thread::id` 的值是不可预测的，在同一程序中的不同线程的id也不同。

`std::thread::id` 是可以CopyConstructible(拷贝构造)和CopyAssignable(拷贝赋值)，所以对于 `std::thread::id` 的拷贝和赋值是没有限制的。

std::thread::id 默认构造函数

构造一个 `std::thread::id` 对象，其不能表示任何执行线程。

声明

```
1 id() noexcept;
```

效果

构造一个 `std::thread::id` 实例，不能表示任何一个线程值。

抛出

无

NOTE 所有默认构造的 `std::thread::id` 实例存储的同一个值。

std::thread::id 相等比较操作

比较两个 `std::thread::id` 的值，看是两个执行线程是否相等。

声明

```
1 bool operator==(std::thread::id lhs,std::thread::id rhs) noexcept;
```

返回

当lhs和rhs表示同一个执行线程或两者不代表没有任何线程，则返回true。当lsh和rhs表示不同执行线程或其中一个代表一个执行线程，另一个不代表任何线程，则返回false。

抛出
无

std::thread::id 不相等比较操作

比较两个 `std::thread::id` 的值，看是两个执行线程是否相等。

声明

```
bool operator! =(std::thread::id lhs,std::thread::id rhs) noexcept;
```

返回
`!(lhs==rhs)`

抛出
无

std::thread::id 小于比较操作

比较两个 `std::thread::id` 的值，看是两个执行线程哪个先执行。

声明

```
bool operator<(std::thread::id lhs,std::thread::id rhs) noexcept;
```

返回
当lhs比rhs的线程ID靠前，则返回true。当lhs!=rhs，且 `lhs<rhs` 或 `rhs<lhs` 返回true，其他情况则返回false。当lhs==rhs，在 `lhs<rhs` 和 `rhs<lhs` 时返回false。

抛出
无

NOTE 当默认构造的 `std::thread::id` 实例，在不代表任何线程的时候，其值小于任何一个代表执行线程的实例。当两个实例相等，那么两个对象代表两个执行线程。任何一组不同的 `std::thread::id` 的值，是由同一序列构造，这与程序执行的顺序相同。同一个可执行程序可能有不同的执行顺序。

std::thread::id 小于等于比较操作

比较两个 `std::thread::id` 的值，看是两个执行线程的ID值是否相等，或其中一个先行。

声明

```
bool operator<(std::thread::id lhs,std::thread::id rhs) noexcept;
```

返回

`!(rhs<lhs)`

抛出

无

std::thread::id 大于比较操作

比较两个 `std::thread::id` 的值，看是两个执行线程的是后行的。

声明

```
bool operator>(std::thread::id lhs,std::thread::id rhs) noexcept;
```

返回

`rhs<lhs`

抛出

无

std::thread::id 大于等于比较操作

比较两个 `std::thread::id` 的值，看是两个执行线程的ID值是否相等，或其中一个后行。

声明

```
bool operator>=(std::thread::id lhs,std::thread::id rhs) noexcept;
```

返回

`!(lhs<rhs)`

抛出
无

std::thread::id 插入流操作

将 `std::thread::id` 的值通过给指定流写入字符串。

声明

```
template<typename charT, typename traits>
basic_ostream<charT, traits>&
operator<< (basic_ostream<charT, traits>&& out, thread::id id);
```

效果
将 `std::thread::id` 的值通过给指定流插入字符串。

返回
无

NOTE 字符串的格式并未给定。`std::thread::id` 实例具有相同的表达式时，是相同的；当实例表达式不同，则代表不同的线程。

std::thread::native_handler 成员函数

`native_handle_type` 是由另一类型定义而来，这个类型会随着指定平台的API而变化。

声明

```
typedef implementation-defined native_handle_type;
```

NOTE 这个类型定义是可选的。如果提供，实现将使用原生平台指定的API，并提供合适的类型作为实现。

std::thread 默认构造函数

返回一个 `native_handle_type` 类型的值，这个值可以可以表示*this相关的执行线程。

声明

```
native_handle_type native_handle();
```

NOTE 这个函数是可选的。如果提供，会使用原生平台指定的API，并返回合适的值。

std::thread 构造函数

构造一个无相关线程的 `std::thread` 对象。

声明

```
thread() noexcept;
```

效果
构造一个无相关线程的 `std::thread` 实例。

后置条件
对于一个新构造的 `std::thread` 对象x，x.get_id() == id()。

抛出
无

std::thread 移动构造函数

将已存在 `std::thread` 对象的所有权，转移到新创建的对象中。

声明

```
thread(thread&& other) noexcept;
```

效果

构造一个 `std::thread` 实例。与other相关的执行线程的所有权，将转移到新创建的 `std::thread` 对象上。否则，新创建的 `std::thread` 对象将无任何相关执行线程。

后置条件

对于一个新构建的 `std::thread` 对象x来说，x.get_id()等价于未转移所有权时的other.get_id()。get_id()==id()。

抛出

无

NOTE `std::thread` 对象是不可CopyConstructible(拷贝构造)，所以该类没有拷贝构造函数，只有移动构造函数。

std::thread 析构函数

销毁 `std::thread` 对象。

声明

```
~thread();
```

效果

销毁 `*this` 。当 `*this` 与执行线程相关(this->joinable()将返回true)，调用 `std::terminate()` 来终止程序。

抛出

无

std::thread 移动赋值操作

将一个 `std::thread` 的所有权，转移到另一个 `std::thread` 对象上。

声明

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

效果

在调用该函数前，this->joinable返回true，则调用 `std::terminate()` 来终止程序。当other在执行赋值前，具有相关的执行线程，那么执行线程现在就与 `*this` 相关联。否则， `*this` 无相关执行线程。

后置条件

this->get_id()的值等于调用该函数前的other.get_id()。oter.get_id()==id()。

抛出

无

NOTE `std::thread` 对象是不可CopyAssignable(拷贝赋值)，所以该类没有拷贝赋值函数，只有移动赋值函数。

std::thread::swap 成员函数

将两个 `std::thread` 对象的所有权进行交换。

声明

```
1 void swap(thread& other) noexcept;
```

效果

当other在执行赋值前，具有相关的执行线程，那么执行线程现在就与 `*this` 相关联。否则，`*this` 无相关执行线程。对于 `*this` 也是一样。

后置条件

this->get_id()的值等于调用该函数前的other.get_id()。other.get_id()的值等于没有调用函数前this->get_id()的值。

抛出

无

std::thread的非成员函数swap

将两个 `std::thread` 对象的所有权进行交换。

声明

```
1 void swap(thread& lhs,thread& rhs) noexcept;
```

效果
lhs.swap(rhs)

抛出
无

std::thread::joinable 成员函数

查询*this是否具有相关执行线程。

声明

```
1 bool joinable() const noexcept;
```

返回
如果*this具有相关执行线程，则返回true；否则，返回false。

抛出
无

std::thread::join 成员函数

等待*this相关的执行线程结束。

声明

```
1 void join();
```

先决条件
this->joinable()返回true。

效果
阻塞当前线程，直到与*this相关的执行线程执行结束。

后置条件
this->get_id()==id()。与*this先关的执行线程将在该函数调用后结束。

同步
想要在*this上成功的调用该函数，则需要依赖有joinable()的返回。

抛出
当效果没有达到或this->joinable()返回false，则抛出 `std::system_error` 异常。

std::thread::detach 成员函数

将*this上的相关线程进行分离。

声明

```
1 void detach();
```

先决条件
this->joinable()返回true。

效果
将*this上的相关线程进行分离。

后置条件
this->get_id()id(), this->joinable()false

与*this相关的执行线程在调用该函数后就会分离，并且不在会与当前 `std::thread` 对象再相关。

抛出
当效果没有达到或this->joinable()返回false，则抛出 `std::system_error` 异常。

std::thread::get_id 成员函数

返回 `std::thread::id` 的值来表示*this上相关执行线程。

声明

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

返回

当*this具有相关执行线程，将返回 `std::thread::id` 作为识别当前函数的依据。否则，返回默认构造的 `std::thread::id` 。

抛出

无

std::thread::hardware_concurrency 静态成员函数

返回硬件上可以并发线程的数量。

声明

```
unsigned hardware_concurrency() noexcept;
```

返回

硬件上可以并发线程的数量。这个值可能是系统处理器的数量。当信息不用或只有定义，则该函数返回0。

抛出

无

D.7.2 this_thread命名空间

这里介绍一下 `std::this_thread` 命名空间内提供的函数操作。

this_thread::get_id 非成员函数

返回 `std::thread::id` 用来识别当前执行线程。

声明

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

返回
可通过 `std:thread::id` 来识别当前线程。

抛出
无

this_thread::yield 非成员函数

该函数用于通知库，调用线程不需要立即运行。一般使用小循环来避免消耗过多CPU时间。

声明

```
void yield() noexcept;
```

效果
使用标准库的实现来安排线程的一些事情。

抛出
无

this_thread::sleep_for 非成员函数

在指定的指定时长内，暂停执行当前线程。

声明

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

效果
在超出relative_time的时长内，阻塞当前线程。

NOTE 线程可能阻塞的时间要长于指定时长。如果可能，逝去的时间由将会由一个稳定时钟决定。

抛出
无

this_thread::sleep_until 非成员函数

暂停指定当前线程，直到到了指定的时间点。

声明

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

效果
在到达absolute_time的时间点前，阻塞当前线程，这个时间点由指定的Clock决定。

NOTE 这里不保证会阻塞多长时间，只有Clock::now()返回的时间等于或大于absolute_time时，阻塞的线程才能被解除阻塞。

抛出
无