

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

```
1  #define ATOMIC_BOOL_LOCK_FREE 参见详述
2  #define ATOMIC_CHAR_LOCK_FREE 参见详述
3  #define ATOMIC_SHORT_LOCK_FREE 参见详述
4  #define ATOMIC_INT_LOCK_FREE 参见详述
5  #define ATOMIC_LONG_LOCK_FREE 参见详述
6  #define ATOMIC_LLONG_LOCK_FREE 参见详述
7  #define ATOMIC_CHAR16_T_LOCK_FREE 参见详述
8  #define ATOMIC_CHAR32_T_LOCK_FREE 参见详述
9  #define ATOMIC_WCHAR_T_LOCK_FREE 参见详述
10 #define ATOMIC_POINTER_LOCK_FREE 参见详述
11
12 #define ATOMIC_VAR_INIT(value) 参见详述
13
14 namespace std
15 {
16     enum memory_order;
17
18     struct atomic_flag;
19     参见类型定义详述 atomic_bool;
20     参见类型定义详述 atomic_char;
21     参见类型定义详述 atomic_char16_t;
22     参见类型定义详述 atomic_char32_t;
23     参见类型定义详述 atomic_schar;
24     参见类型定义详述 atomic_uchar;
25     参见类型定义详述 atomic_short;
26     参见类型定义详述 atomic_ushort;
27     参见类型定义详述 atomic_int;
28     参见类型定义详述 atomic_uint;
29     参见类型定义详述 atomic_long;
30     参见类型定义详述 atomic_ulong;
31     参见类型定义详述 atomic_llong;
```

```
32  参见类型定义详述 atomic_ullong;
33  参见类型定义详述 atomic_wchar_t;
34
35  参见类型定义详述 atomic_int_least8_t;
36  参见类型定义详述 atomic_uint_least8_t;
37  参见类型定义详述 atomic_int_least16_t;
38  参见类型定义详述 atomic_uint_least16_t;
39  参见类型定义详述 atomic_int_least32_t;
40  参见类型定义详述 atomic_uint_least32_t;
41  参见类型定义详述 atomic_int_least64_t;
42  参见类型定义详述 atomic_uint_least64_t;
43  参见类型定义详述 atomic_int_fast8_t;
44  参见类型定义详述 atomic_uint_fast8_t;
45  参见类型定义详述 atomic_int_fast16_t;
46  参见类型定义详述 atomic_uint_fast16_t;
47  参见类型定义详述 atomic_int_fast32_t;
48  参见类型定义详述 atomic_uint_fast32_t;
49  参见类型定义详述 atomic_int_fast64_t;
50  参见类型定义详述 atomic_uint_fast64_t;
51  参见类型定义详述 atomic_int8_t;
52  参见类型定义详述 atomic_uint8_t;
53  参见类型定义详述 atomic_int16_t;
54  参见类型定义详述 atomic_uint16_t;
55  参见类型定义详述 atomic_int32_t;
56  参见类型定义详述 atomic_uint32_t;
57  参见类型定义详述 atomic_int64_t;
58  参见类型定义详述 atomic_uint64_t;
59  参见类型定义详述 atomic_intptr_t;
60  参见类型定义详述 atomic_uintptr_t;
61  参见类型定义详述 atomic_size_t;
62  参见类型定义详述 atomic_ssize_t;
63  参见类型定义详述 atomic_ptrdiff_t;
64  参见类型定义详述 atomic_intmax_t;
65  参见类型定义详述 atomic_uintmax_t;
66
67  template<typename T>
68  struct atomic;
69
70  extern "C" void atomic_thread_fence(memory_order order);
71  extern "C" void atomic_signal_fence(memory_order order);
72
73  template<typename T>
74  T kill_dependency(T);
75 }
```

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::atomic<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::atomic<>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

```
1 #define ATOMIC_BOOL_LOCK_FREE 参见详述
2 #define ATOMIC_CHAR_LOCK_FREE 参见详述
3 #define ATOMIC_SHORT_LOCK_FREE 参见详述
4 #define ATOMIC_INT_LOCK_FREE 参见详述
5 #define ATOMIC_LONG_LOCK_FREE 参见详述
6 #define ATOMIC_LLONG_LOCK_FREE 参见详述
7 #define ATOMIC_CHAR16_T_LOCK_FREE 参见详述
8 #define ATOMIC_CHAR32_T_LOCK_FREE 参见详述
9 #define ATOMIC_WCHAR_T_LOCK_FREE 参见详述
10 #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<int>` 和 `std::atomic<unsigned>` 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:

```
1 std::atomic<int> i = ATOMIC_VAR_INIT(42);
```

```
2  std::string s;  
3  std::atomic<std::string*> 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

```
1  typedef enum memory_order  
2  {  
3      memory_order_relaxed,memory_order_consume,  
4      memory_order_acquire,memory_order_release,  
5      memory_order_acq_rel,memory_order_seq_cst  
6  } 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

```
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

```
1 struct atomic_flag
2 {
3     atomic_flag() noexcept = default;
4     atomic_flag(const atomic_flag&) = delete;
5     atomic_flag& operator=(const atomic_flag&) = delete;
6     atomic_flag& operator=(const atomic_flag&) volatile = delete;
7
8     bool test_and_set(memory_order = memory_order_seq_cst) volatile
```

```
9     noexcept;
10    bool test_and_set(memory_order = memory_order_seq_cst) noexcept;
11    void clear(memory_order = memory_order_seq_cst) volatile noexcept;
12    void clear(memory_order = memory_order_seq_cst) noexcept;
13 };
14
15 bool atomic_flag_test_and_set(volatile atomic_flag*) noexcept;
16 bool atomic_flag_test_and_set(atomic_flag*) noexcept;
17 bool atomic_flag_test_and_set_explicit(
18     volatile atomic_flag*, memory_order) noexcept;
19 bool atomic_flag_test_and_set_explicit(
20     atomic_flag*, memory_order) noexcept;
21 void atomic_flag_clear(volatile atomic_flag*) noexcept;
22 void atomic_flag_clear(atomic_flag*) noexcept;
23 void atomic_flag_clear_explicit(
24     volatile atomic_flag*, memory_order) noexcept;
25 void atomic_flag_clear_explicit(
26     atomic_flag*, memory_order) noexcept;
27
28 #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

```
1 bool atomic_flag_test_and_set(volatile atomic_flag* flag) noexcept;  
2 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

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

```
3 bool atomic_flag_test_and_set_explicit(  
4     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

```
1 bool atomic_flag_test_and_set_explicit(  
2     volatile atomic_flag* flag, memory_order order) noexcept;  
3 bool atomic_flag_test_and_set_explicit(  
4     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

```
1 void clear(memory_order order = memory_order_seq_cst) volatile noexcept;  
2 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

```
1 void atomic_flag_clear(volatile atomic_flag* flag) noexcept;  
2 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

```
1 void atomic_flag_clear_explicit(  
2     volatile atomic_flag* flag, memory_order order) noexcept;  
3 void atomic_flag_clear_explicit(  
4     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)，因为你懂得，因为这样原子操作就无法执行。

类型定义

```
1  template<typename BaseType>
2  struct atomic
3  {
4      atomic() noexcept = default;
5      constexpr atomic(BaseType) noexcept;
6      BaseType operator=(BaseType) volatile noexcept;
7      BaseType operator=(BaseType) noexcept;
8
9      atomic(const atomic&) = delete;
10     atomic& operator=(const atomic&) = delete;
11     atomic& operator=(const atomic&) volatile = delete;
12
13     bool is_lock_free() const volatile noexcept;
14     bool is_lock_free() const noexcept;
15
16     void store(BaseType, memory_order = memory_order_seq_cst)
17         volatile noexcept;
18     void store(BaseType, memory_order = memory_order_seq_cst) noexcept;
```

```
19  BaseType load(memory_order = memory_order_seq_cst)
20      const volatile noexcept;
21  BaseType load(memory_order = memory_order_seq_cst) const noexcept;
22  BaseType exchange(BaseType, memory_order = memory_order_seq_cst)
23      volatile noexcept;
24  BaseType exchange(BaseType, memory_order = memory_order_seq_cst)
25      noexcept;
26
27  bool compare_exchange_strong(
28      BaseType & old_value, BaseType new_value,
29      memory_order order = memory_order_seq_cst) volatile noexcept;
30  bool compare_exchange_strong(
31      BaseType & old_value, BaseType new_value,
32      memory_order order = memory_order_seq_cst) noexcept;
33  bool compare_exchange_strong(
34      BaseType & old_value, BaseType new_value,
35      memory_order success_order,
36      memory_order failure_order) volatile noexcept;
37  bool compare_exchange_strong(
38      BaseType & old_value, BaseType new_value,
39      memory_order success_order,
40      memory_order failure_order) noexcept;
41  bool compare_exchange_weak(
42      BaseType & old_value, BaseType new_value,
43      memory_order order = memory_order_seq_cst)
44      volatile noexcept;
45  bool compare_exchange_weak(
46      BaseType & old_value, BaseType new_value,
47      memory_order order = memory_order_seq_cst) noexcept;
48  bool compare_exchange_weak(
49      BaseType & old_value, BaseType new_value,
50      memory_order success_order,
51      memory_order failure_order) volatile noexcept;
52  bool compare_exchange_weak(
53      BaseType & old_value, BaseType new_value,
54      memory_order success_order,
55      memory_order failure_order) noexcept;
56  operator BaseType () const volatile noexcept;
57  operator BaseType () const noexcept;
58 };
59
60 template<typename BaseType>
61 bool atomic_is_lock_free(volatile const atomic<BaseType>*) noexcept;
62 template<typename BaseType>
63 bool atomic_is_lock_free(const atomic<BaseType>*) noexcept;
```

```
64 template<typename BaseType>
65 void atomic_init(volatile atomic<BaseType>*, void*) noexcept;
66 template<typename BaseType>
67 void atomic_init(atomic<BaseType>*, void*) noexcept;
68 template<typename BaseType>
69 BaseType atomic_exchange(volatile atomic<BaseType>*, memory_order)
70     noexcept;
71 template<typename BaseType>
72 BaseType atomic_exchange(atomic<BaseType>*, memory_order) noexcept;
73 template<typename BaseType>
74 BaseType atomic_exchange_explicit(
75     volatile atomic<BaseType>*, memory_order) noexcept;
76 template<typename BaseType>
77 BaseType atomic_exchange_explicit(
78     atomic<BaseType>*, memory_order) noexcept;
79 template<typename BaseType>
80 void atomic_store(volatile atomic<BaseType>*, BaseType) noexcept;
81 template<typename BaseType>
82 void atomic_store(atomic<BaseType>*, BaseType) noexcept;
83 template<typename BaseType>
84 void atomic_store_explicit(
85     volatile atomic<BaseType>*, BaseType, memory_order) noexcept;
86 template<typename BaseType>
87 void atomic_store_explicit(
88     atomic<BaseType>*, BaseType, memory_order) noexcept;
89 template<typename BaseType>
90 BaseType atomic_load(volatile const atomic<BaseType>*) noexcept;
91 template<typename BaseType>
92 BaseType atomic_load(const atomic<BaseType>*) noexcept;
93 template<typename BaseType>
94 BaseType atomic_load_explicit(
95     volatile const atomic<BaseType>*, memory_order) noexcept;
96 template<typename BaseType>
97 BaseType atomic_load_explicit(
98     const atomic<BaseType>*, memory_order) noexcept;
99 template<typename BaseType>
100 bool atomic_compare_exchange_strong(
101     volatile atomic<BaseType>*, BaseType * old_value,
102     BaseType new_value) noexcept;
103 template<typename BaseType>
104 bool atomic_compare_exchange_strong(
105     atomic<BaseType>*, BaseType * old_value,
106     BaseType new_value) noexcept;
107 template<typename BaseType>
108 bool atomic_compare_exchange_strong_explicit(
```

```
109 volatile atomic<BaseType>*, BaseType * old_value,  
110 BaseType new_value, memory_order success_order,  
111 memory_order failure_order) noexcept;  
112 template<typename BaseType>  
113 bool atomic_compare_exchange_strong_explicit(  
114     atomic<BaseType>*, BaseType * old_value,  
115     BaseType new_value, memory_order success_order,  
116     memory_order failure_order) noexcept;  
117 template<typename BaseType>  
118 bool atomic_compare_exchange_weak(  
119     volatile atomic<BaseType>*, BaseType * old_value, BaseType new_value)  
120     noexcept;  
121 template<typename BaseType>  
122 bool atomic_compare_exchange_weak(  
123     atomic<BaseType>*, BaseType * old_value, BaseType new_value) noexcept;  
124 template<typename BaseType>  
125 bool atomic_compare_exchange_weak_explicit(  
126     volatile atomic<BaseType>*, BaseType * old_value,  
127     BaseType new_value, memory_order success_order,  
128     memory_order failure_order) noexcept;  
129 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

```
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

```
1 BaseType operator=(BaseType b) volatile noexcept;  
2 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

```
1 bool is_lock_free() const volatile noexcept;  
2 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

```
1  template<typename BaseType>
2  bool atomic_is_lock_free(volatile const atomic<BaseType>* p) noexcept;
3  template<typename BaseType>
4  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

```
1  BaseType load(memory_order order = memory_order_seq_cst)
2      const volatile noexcept;
3  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

```
1 template<typename BaseType>
2 BaseType atomic_load(volatile const atomic<BaseType>* p) noexcept;
3 template<typename BaseType>
4 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

```
1 template<typename BaseType>
2 BaseType atomic_load_explicit(
3     volatile const atomic<BaseType>* p, memory_order order) noexcept;
4 template<typename BaseType>
5 BaseType atomic_load_explicit(
6     const atomic<BaseType>* p, memory_order order) noexcept;
```

效果

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

std::atomic::operator BaseType 转换操作

加载存储在*this中的值。

声明

```
1 operator BaseType() const volatile noexcept;
2 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

```
1 void store(BaseType new_value, memory_order order = memory_order_seq_cst)
2     volatile noexcept;
3 void store(BaseType new_value, memory_order order = memory_order_seq_cst)
4     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

```
1 template<typename BaseType>
2 void atomic_store(volatile atomic<BaseType>* p, BaseType new_value)
3     noexcept;
4 template<typename BaseType>
5 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

```
1 template<typename BaseType>
2 void atomic_store_explicit(
3     volatile atomic<BaseType>* p, BaseType new_value, memory_order order)
4     noexcept;
5 template<typename BaseType>
6 void atomic_store_explicit(
7     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

```
1 BaseType exchange(
2     BaseType new_value,
3     memory_order order = memory_order_seq_cst)
4     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

```
1 template<typename BaseType>
2 BaseType atomic_exchange(volatile atomic<BaseType>* p, BaseType new_value)
3     noexcept;
4 template<typename BaseType>
5 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

```
1 template<typename BaseType>
2 BaseType atomic_exchange_explicit(
3     volatile atomic<BaseType>* p, BaseType new_value, memory_order order)
4     noexcept;
5 template<typename BaseType>
6 BaseType atomic_exchange_explicit(
7     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

```
1 bool compare_exchange_strong(  
2     BaseType& expected, BaseType new_value,  
3     memory_order order = std::memory_order_seq_cst) volatile noexcept;  
4 bool compare_exchange_strong(  
5     BaseType& expected, BaseType new_value,  
6     memory_order order = std::memory_order_seq_cst) noexcept;  
7 bool compare_exchange_strong(  
8     BaseType& expected, BaseType new_value,  
9     memory_order success_order, memory_order failure_order)  
10    volatile noexcept;  
11 bool compare_exchange_strong(  
12     BaseType& expected, BaseType new_value,  
13     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*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==order和failure_order==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

```
1  template<typename BaseType>
2  bool atomic_compare_exchange_strong(
3      volatile atomic<BaseType>* p, BaseType * old_value, BaseType new_value)
4      noexcept;
5  template<typename BaseType>
6  bool atomic_compare_exchange_strong(
7      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

```
1  template<typename BaseType>
2  bool atomic_compare_exchange_strong_explicit(
3      volatile atomic<BaseType>* p, BaseType * old_value,
4      BaseType new_value, memory_order success_order,
5      memory_order failure_order) noexcept;
6  template<typename BaseType>
7  bool atomic_compare_exchange_strong_explicit(
8      atomic<BaseType>* p, BaseType * old_value,
9      BaseType new_value, memory_order success_order,
10     memory_order failure_order) noexcept;
```

效果

```
1  return p->compare_exchange_strong(
```



```
2      *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

```
1  bool compare_exchange_weak(  
2      BaseType& expected,BaseType new_value,  
3      memory_order order = std::memory_order_seq_cst) volatile noexcept;  
4  bool compare_exchange_weak(  
5      BaseType& expected,BaseType new_value,  
6      memory_order order = std::memory_order_seq_cst) noexcept;  
7  bool compare_exchange_weak(  
8      BaseType& expected,BaseType new_value,  
9      memory_order success_order,memory_order failure_order)  
10     volatile noexcept;  
11 bool compare_exchange_weak(  
12     BaseType& expected,BaseType new_value,  
13     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*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==order和failure_order==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

```
1  template<typename BaseType>
2  bool atomic_compare_exchange_weak(
3      volatile atomic<BaseType>* p, BaseType * old_value, BaseType new_value)
4      noexcept;
5  template<typename BaseType>
6  bool atomic_compare_exchange_weak(
7      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

```
1  template<typename BaseType>
2  bool atomic_compare_exchange_weak_explicit(
3      volatile atomic<BaseType>* p, BaseType * old_value,
4      BaseType new_value, memory_order success_order,
5      memory_order failure_order) noexcept;
6  template<typename BaseType>
7  bool atomic_compare_exchange_weak_explicit(
8      atomic<BaseType>* p, BaseType * old_value,
9      BaseType new_value, memory_order success_order,
10     memory_order failure_order) noexcept;
```

效果

```
1 return p->compare_exchange_weak(  
2     *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：

```
1 std::atomic<bool>  
2 std::atomic<char>  
3 std::atomic<signed char>  
4 std::atomic<unsigned char>  
5 std::atomic<short>  
6 std::atomic<unsigned short>  
7 std::atomic<int>  
8 std::atomic<unsigned>  
9 std::atomic<long>  
10 std::atomic<unsigned long>  
11 std::atomic<long long>  
12 std::atomic<unsigned long long>  
13 std::atomic<wchar_t>  
14 std::atomic<char16_t>  
15 std::atomic<char32_t>;
```

`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：

```
1  std::atomic<char>
2  std::atomic<signed char>
3  std::atomic<unsigned char>
4  std::atomic<short>
5  std::atomic<unsigned short>
6  std::atomic<int>
7  std::atomic<unsigned>
8  std::atomic<long>
9  std::atomic<unsigned long>
10 std::atomic<long long>
11 std::atomic<unsigned long long>
12 std::atomic<wchar_t>
13 std::atomic<char16_t>
14 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

```
1  template<>
2  struct atomic<integral-type>
3  {
4      atomic() noexcept = default;
5      constexpr atomic(integral-type) noexcept;
6      bool operator=(integral-type) volatile noexcept;
7
8      atomic(const atomic&) = delete;
9      atomic& operator=(const atomic&) = delete;
10     atomic& operator=(const atomic&) volatile = delete;
11
12     bool is_lock_free() const volatile noexcept;
13     bool is_lock_free() const noexcept;
14
15     void store(integral-type, memory_order = memory_order_seq_cst)
16         volatile noexcept;
17     void store(integral-type, memory_order = memory_order_seq_cst) noexcept;
18     integral-type load(memory_order = memory_order_seq_cst)
19         const volatile noexcept;
20     integral-type load(memory_order = memory_order_seq_cst) const noexcept;
21     integral-type exchange(
22         integral-type, memory_order = memory_order_seq_cst)
23         volatile noexcept;
```

```
24  integral-type exchange(  
25      integral-type, memory_order = memory_order_seq_cst) noexcept;  
26  
27  bool compare_exchange_strong(  
28      integral-type & old_value, integral-type new_value,  
29      memory_order order = memory_order_seq_cst) volatile noexcept;  
30  bool compare_exchange_strong(  
31      integral-type & old_value, integral-type new_value,  
32      memory_order order = memory_order_seq_cst) noexcept;  
33  bool compare_exchange_strong(  
34      integral-type & old_value, integral-type new_value,  
35      memory_order success_order, memory_order failure_order)  
36      volatile noexcept;  
37  bool compare_exchange_strong(  
38      integral-type & old_value, integral-type new_value,  
39      memory_order success_order, memory_order failure_order) noexcept;  
40  bool compare_exchange_weak(  
41      integral-type & old_value, integral-type new_value,  
42      memory_order order = memory_order_seq_cst) volatile noexcept;  
43  bool compare_exchange_weak(  
44      integral-type & old_value, integral-type new_value,  
45      memory_order order = memory_order_seq_cst) noexcept;  
46  bool compare_exchange_weak(  
47      integral-type & old_value, integral-type new_value,  
48      memory_order success_order, memory_order failure_order)  
49      volatile noexcept;  
50  bool compare_exchange_weak(  
51      integral-type & old_value, integral-type new_value,  
52      memory_order success_order, memory_order failure_order) noexcept;  
53  
54  operator integral-type() const volatile noexcept;  
55  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
103 bool atomic_is_lock_free(volatile const atomic<integral-type>*) noexcept;
104 bool atomic_is_lock_free(const atomic<integral-type>*) noexcept;
105 void atomic_init(volatile atomic<integral-type>*, integral-type) noexcept;
106 void atomic_init(atomic<integral-type>*, integral-type) noexcept;
107 integral-type atomic_exchange(
108     volatile atomic<integral-type>*, integral-type) noexcept;
109 integral-type atomic_exchange(
110     atomic<integral-type>*, integral-type) noexcept;
111 integral-type atomic_exchange_explicit(
112     volatile atomic<integral-type>*, integral-type, memory_order) noexcept;
113 integral-type atomic_exchange_explicit(
```

```
114     atomic<integral-type>*,integral-type, memory_order) noexcept;
115 void atomic_store(volatile atomic<integral-type>*,integral-type) noexcept;
116 void atomic_store(atomic<integral-type>*,integral-type) noexcept;
117 void atomic_store_explicit(
118     volatile atomic<integral-type>*,integral-type, memory_order) noexcept;
119 void atomic_store_explicit(
120     atomic<integral-type>*,integral-type, memory_order) noexcept;
121 integral-type atomic_load(volatile const atomic<integral-type>*) noexcept;
122 integral-type atomic_load(const atomic<integral-type>*) noexcept;
123 integral-type atomic_load_explicit(
124     volatile const atomic<integral-type>*,memory_order) noexcept;
125 integral-type atomic_load_explicit(
126     const atomic<integral-type>*,memory_order) noexcept;
127 bool atomic_compare_exchange_strong(
128     volatile atomic<integral-type>*,
129     integral-type * old_value,integral-type new_value) noexcept;
130 bool atomic_compare_exchange_strong(
131     atomic<integral-type>*,
132     integral-type * old_value,integral-type new_value) noexcept;
133 bool atomic_compare_exchange_strong_explicit(
134     volatile atomic<integral-type>*,
135     integral-type * old_value,integral-type new_value,
136     memory_order success_order,memory_order failure_order) noexcept;
137 bool atomic_compare_exchange_strong_explicit(
138     atomic<integral-type>*,
139     integral-type * old_value,integral-type new_value,
140     memory_order success_order,memory_order failure_order) noexcept;
141 bool atomic_compare_exchange_weak(
142     volatile atomic<integral-type>*,
143     integral-type * old_value,integral-type new_value) noexcept;
144 bool atomic_compare_exchange_weak(
145     atomic<integral-type>*,
146     integral-type * old_value,integral-type new_value) noexcept;
147 bool atomic_compare_exchange_weak_explicit(
148     volatile atomic<integral-type>*,
149     integral-type * old_value,integral-type new_value,
150     memory_order success_order,memory_order failure_order) noexcept;
151 bool atomic_compare_exchange_weak_explicit(
152     atomic<integral-type>*,
153     integral-type * old_value,integral-type new_value,
154     memory_order success_order,memory_order failure_order) noexcept;
155
156 integral-type atomic_fetch_add(
157     volatile atomic<integral-type>*,integral-type) noexcept;
158 integral-type atomic_fetch_add(
```

```
159     atomic<integral-type>*,integral-type) noexcept;
160 integral-type atomic_fetch_add_explicit(
161     volatile atomic<integral-type>*,integral-type, memory_order) noexcept;
162 integral-type atomic_fetch_add_explicit(
163     atomic<integral-type>*,integral-type, memory_order) noexcept;
164 integral-type atomic_fetch_sub(
165     volatile atomic<integral-type>*,integral-type) noexcept;
166 integral-type atomic_fetch_sub(
167     atomic<integral-type>*,integral-type) noexcept;
168 integral-type atomic_fetch_sub_explicit(
169     volatile atomic<integral-type>*,integral-type, memory_order) noexcept;
170 integral-type atomic_fetch_sub_explicit(
171     atomic<integral-type>*,integral-type, memory_order) noexcept;
172 integral-type atomic_fetch_and(
173     volatile atomic<integral-type>*,integral-type) noexcept;
174 integral-type atomic_fetch_and(
175     atomic<integral-type>*,integral-type) noexcept;
176 integral-type atomic_fetch_and_explicit(
177     volatile atomic<integral-type>*,integral-type, memory_order) noexcept;
178 integral-type atomic_fetch_and_explicit(
179     atomic<integral-type>*,integral-type, memory_order) noexcept;
180 integral-type atomic_fetch_or(
181     volatile atomic<integral-type>*,integral-type) noexcept;
182 integral-type atomic_fetch_or(
183     atomic<integral-type>*,integral-type) noexcept;
184 integral-type atomic_fetch_or_explicit(
185     volatile atomic<integral-type>*,integral-type, memory_order) noexcept;
186 integral-type atomic_fetch_or_explicit(
187     atomic<integral-type>*,integral-type, memory_order) noexcept;
188 integral-type atomic_fetch_xor(
189     volatile atomic<integral-type>*,integral-type) noexcept;
190 integral-type atomic_fetch_xor(
191     atomic<integral-type>*,integral-type) noexcept;
192 integral-type atomic_fetch_xor_explicit(
193     volatile atomic<integral-type>*,integral-type, memory_order) noexcept;
194 integral-type atomic_fetch_xor_explicit(
195     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

```
1  integral-type fetch_add(  
2      integral-type i, memory_order order = memory_order_seq_cst)  
3      volatile noexcept;  
4  integral-type fetch_add(  
5      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

```
1  integral-type atomic_fetch_add(  
2      volatile atomic<integral-type>* p, integral-type i) noexcept;  
3  integral-type atomic_fetch_add(  
4      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

```
1  integral-type atomic_fetch_add_explicit(  
2      volatile atomic<integral-type>* p, integral-type i,  
3      memory_order order) noexcept;  
4  integral-type atomic_fetch_add_explicit(  
5      atomic<integral-type>* p, integral-type i, memory_order order)  
6      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

```
1  integral-type fetch_sub(  
2      integral-type i, memory_order order = memory_order_seq_cst)  
3      volatile noexcept;  
4  integral-type fetch_sub(  
5      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

```
1 integral-type atomic_fetch_sub(  
2     volatile atomic<integral-type>* p, integral-type i) noexcept;  
3 integral-type atomic_fetch_sub(  
4     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

```
1 integral-type atomic_fetch_sub_explicit(  
2     volatile atomic<integral-type>* p, integral-type i,  
3     memory_order order) noexcept;  
4 integral-type atomic_fetch_sub_explicit(  
5     atomic<integral-type>* p, integral-type i, memory_order order)  
6     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

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

```
4 integral-type fetch_and(  
5     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

```
1 integral-type atomic_fetch_and(  
2     volatile atomic<integral-type>* p, integral-type i) noexcept;  
3 integral-type atomic_fetch_and(  
4     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

```
1 integral-type atomic_fetch_and_explicit(  
2     integral-type i, memory_order order = memory_order_seq_cst) noexcept;
```

```
2     volatile atomic<integral-type>* p, integral-type i,  
3     memory_order order) noexcept;  
4     integral-type atomic_fetch_and_explicit(  
5         atomic<integral-type>* p, integral-type i, memory_order order)  
6         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

```
1     integral-type fetch_or(  
2         integral-type i, memory_order order = memory_order_seq_cst)  
3         volatile noexcept;  
4     integral-type fetch_or(  
5         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*和*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

```
1 integral-type atomic_fetch_or(  
2     volatile atomic<integral-type>* p, integral-type i) noexcept;  
3 integral-type atomic_fetch_or(  
4     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

```
1 integral-type atomic_fetch_or_explicit(  
2     volatile atomic<integral-type>* p, integral-type i,  
3     memory_order order) noexcept;  
4 integral-type atomic_fetch_or_explicit(  
5     atomic<integral-type>* p, integral-type i, memory_order order)  
6     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

```
1 integral-type fetch_xor(  
2     integral-type i) noexcept;
```

```
2     integral-type i, memory_order order = memory_order_seq_cst)
3     volatile noexcept;
4     integral-type fetch_xor(
5     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

```
1     integral-type atomic_fetch_xor_explicit(
2         volatile atomic<integral-type>* p, integral-type i,
3         memory_order order) noexcept;
4     integral-type atomic_fetch_xor_explicit(
5         atomic<integral-type>* p, integral-type i, memory_order order)
6         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

```
1  integral-type atomic_fetch_xor_explicit(  
2      volatile atomic<integral-type>* p, integral-type i,  
3      memory_order order) noexcept;  
4  integral-type atomic_fetch_xor_explicit(  
5      atomic<integral-type>* p, integral-type i, memory_order order)  
6      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

```
1  integral-type operator++() volatile noexcept;  
2  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

```
1  integral-type operator++() volatile noexcept;  
2  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

```
1 integral-type operator--() volatile noexcept;  
2 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

```
1 integral-type operator--() volatile noexcept;  
2 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

```
1 integral-type operator+=(integral-type i) volatile noexcept;  
2 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

```
1 integral-type operator-=(integral-type i) volatile noexcept;  
2 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

```
1 integral-type operator&=(integral-type i) volatile noexcept;  
2 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

```
1 integral-type operator|=(integral-type i) volatile noexcept;  
2 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

```
1 integral-type operator^=(integral-type i) volatile noexcept;  
2 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

```
1 template<typename T>  
2 struct atomic<T*>  
3 {  
4     atomic() noexcept = default;  
5     constexpr atomic(T*) noexcept;  
6     bool operator=(T*) volatile;  
7     bool operator=(T*);  
8  
9     atomic(const atomic&) = delete;
```

```
10  atomic& operator=(const atomic&) = delete;
11  atomic& operator=(const atomic&) volatile = delete;
12
13  bool is_lock_free() const volatile noexcept;
14  bool is_lock_free() const noexcept;
15  void store(T*,memory_order = memory_order_seq_cst) volatile noexcept;
16  void store(T*,memory_order = memory_order_seq_cst) noexcept;
17  T* load(memory_order = memory_order_seq_cst) const volatile noexcept;
18  T* load(memory_order = memory_order_seq_cst) const noexcept;
19  T* exchange(T*,memory_order = memory_order_seq_cst) volatile noexcept;
20  T* exchange(T*,memory_order = memory_order_seq_cst) noexcept;
21
22  bool compare_exchange_strong(
23      T* & old_value, T* new_value,
24      memory_order order = memory_order_seq_cst) volatile noexcept;
25  bool compare_exchange_strong(
26      T* & old_value, T* new_value,
27      memory_order order = memory_order_seq_cst) noexcept;
28  bool compare_exchange_strong(
29      T* & old_value, T* new_value,
30      memory_order success_order,memory_order failure_order)
31      volatile noexcept;
32  bool compare_exchange_strong(
33      T* & old_value, T* new_value,
34      memory_order success_order,memory_order failure_order) noexcept;
35  bool compare_exchange_weak(
36      T* & old_value, T* new_value,
37      memory_order order = memory_order_seq_cst) volatile noexcept;
38  bool compare_exchange_weak(
39      T* & old_value, T* new_value,
40      memory_order order = memory_order_seq_cst) noexcept;
41  bool compare_exchange_weak(
42      T* & old_value, T* new_value,
43      memory_order success_order,memory_order failure_order)
44      volatile noexcept;
45  bool compare_exchange_weak(
46      T* & old_value, T* new_value,
47      memory_order success_order,memory_order failure_order) noexcept;
48
49  operator T*() const volatile noexcept;
50  operator T*() const noexcept;
51
52  T* fetch_add(
53      ptrdiff_t,memory_order = memory_order_seq_cst) volatile noexcept;
54  T* fetch_add(
```

```

55     ptrdiff_t, memory_order = memory_order_seq_cst) noexcept;
56 T* fetch_sub(
57     ptrdiff_t, memory_order = memory_order_seq_cst) volatile noexcept;
58 T* fetch_sub(
59     ptrdiff_t, memory_order = memory_order_seq_cst) noexcept;
60
61 T* operator++() volatile noexcept;
62 T* operator++() noexcept;
63 T* operator++(int) volatile noexcept;
64 T* operator++(int) noexcept;
65 T* operator--() volatile noexcept;
66 T* operator--() noexcept;
67 T* operator--(int) volatile noexcept;
68 T* operator--(int) noexcept;
69
70 T* operator+=(ptrdiff_t) volatile noexcept;
71 T* operator+=(ptrdiff_t) noexcept;
72 T* operator-=(ptrdiff_t) volatile noexcept;
73 T* operator-=(ptrdiff_t) noexcept;
74 };
75
76 bool atomic_is_lock_free(volatile const atomic<T*>*) noexcept;
77 bool atomic_is_lock_free(const atomic<T*>*) noexcept;
78 void atomic_init(volatile atomic<T*>*, T*) noexcept;
79 void atomic_init(atomic<T*>*, T*) noexcept;
80 T* atomic_exchange(volatile atomic<T*>*, T*) noexcept;
81 T* atomic_exchange(atomic<T*>*, T*) noexcept;
82 T* atomic_exchange_explicit(volatile atomic<T*>*, T*, memory_order)
83     noexcept;
84 T* atomic_exchange_explicit(atomic<T*>*, T*, memory_order) noexcept;
85 void atomic_store(volatile atomic<T*>*, T*) noexcept;
86 void atomic_store(atomic<T*>*, T*) noexcept;
87 void atomic_store_explicit(volatile atomic<T*>*, T*, memory_order)
88     noexcept;
89 void atomic_store_explicit(atomic<T*>*, T*, memory_order) noexcept;
90 T* atomic_load(volatile const atomic<T*>*) noexcept;
91 T* atomic_load(const atomic<T*>*) noexcept;
92 T* atomic_load_explicit(volatile const atomic<T*>*, memory_order) noexcept;
93 T* atomic_load_explicit(const atomic<T*>*, memory_order) noexcept;
94 bool atomic_compare_exchange_strong(
95     volatile atomic<T*>*, T* * old_value, T* new_value) noexcept;
96 bool atomic_compare_exchange_strong(
97     volatile atomic<T*>*, T* * old_value, T* new_value) noexcept;
98 bool atomic_compare_exchange_strong_explicit(
99     atomic<T*>*, T* * old_value, T* new_value,

```

```

100     memory_order success_order, memory_order failure_order) noexcept;
101 bool atomic_compare_exchange_strong_explicit(
102     atomic<T*>*, T* * old_value, T* new_value,
103     memory_order success_order, memory_order failure_order) noexcept;
104 bool atomic_compare_exchange_weak(
105     volatile atomic<T*>*, T* * old_value, T* new_value) noexcept;
106 bool atomic_compare_exchange_weak(
107     atomic<T*>*, T* * old_value, T* new_value) noexcept;
108 bool atomic_compare_exchange_weak_explicit(
109     volatile atomic<T*>*, T* * old_value, T* new_value,
110     memory_order success_order, memory_order failure_order) noexcept;
111 bool atomic_compare_exchange_weak_explicit(
112     atomic<T*>*, T* * old_value, T* new_value,
113     memory_order success_order, memory_order failure_order) noexcept;
114
115 T* atomic_fetch_add(volatile atomic<T*>*, ptrdiff_t) noexcept;
116 T* atomic_fetch_add(atomic<T*>*, ptrdiff_t) noexcept;
117 T* atomic_fetch_add_explicit(
118     volatile atomic<T*>*, ptrdiff_t, memory_order) noexcept;
119 T* atomic_fetch_add_explicit(
120     atomic<T*>*, ptrdiff_t, memory_order) noexcept;
121 T* atomic_fetch_sub(volatile atomic<T*>*, ptrdiff_t) noexcept;
122 T* atomic_fetch_sub(atomic<T*>*, ptrdiff_t) noexcept;
123 T* atomic_fetch_sub_explicit(
124     volatile atomic<T*>*, ptrdiff_t, memory_order) noexcept;
125 T* atomic_fetch_sub_explicit(
126     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

```

1  T* fetch_add(
2      ptrdiff_t i, memory_order order = memory_order_seq_cst)
3      volatile noexcept;
4  T* fetch_add(
5      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

```
1 T* atomic_fetch_add(volatile atomic<T*>* p, ptrdiff_t i) noexcept;
2 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

```
1 T* atomic_fetch_add_explicit(
2     volatile atomic<T*>* p, ptrdiff_t i, memory_order order) noexcept;
3 T* atomic_fetch_add_explicit(
4     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

```
1 T* fetch_sub(  
2     ptrdiff_t i, memory_order order = memory_order_seq_cst)  
3     volatile noexcept;  
4 T* fetch_sub(  
5     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

```
1 T* atomic_fetch_sub(volatile atomic<T*>* p, ptrdiff_t i) noexcept;  
2 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

```
1 T* atomic_fetch_sub_explicit(  
2     volatile atomic<T*>* p, ptrdiff_t i, memory_order order) noexcept;  
3 T* atomic_fetch_sub_explicit(  
4     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

```
1 T* operator++() volatile noexcept;  
2 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

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

```
2 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

```
1 T* operator--() volatile noexcept;  
2 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

```
1 T* operator--() volatile noexcept;  
2 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

```
1 T* operator+=(ptrdiff_t i) volatile noexcept;  
2 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

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

效果

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