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<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：

```
std::atomic&lt;int> i = ATOMIC_VAR_INIT(42);
```

```
std::string s;
std::atomic&lt;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

```
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

```
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&lt;some-simple-struct&gt;` 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>
bool atomic_compare_exchange_weak_explicit(
  atomic<BaseType>*,BaseType * old_value,
  BaseType new_value, memory_order success_order,
  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

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

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==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

```
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==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

```
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;

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

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

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

bool atomic_is_lock_free(volatile const atomic<integral-type>*) noexcept;
bool atomic_is_lock_free(const atomic<integral-type>*) noexcept;
void atomic_init(volatile atomic<integral-type>*,integral-type) noexcept;
void atomic_init(atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_exchange(
    volatile atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_exchange(
    atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_exchange_explicit(
    volatile atomic<integral-type>*,integral-type, memory_order) noexcept;
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

```
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

```
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;
```

```
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,
```

```
  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;
```

```
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

```
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;
```