D.7 thread头文件

<thread> 头文件提供了管理和辨别线程的工具，并且提供函数，可让当前线程休眠。

头文件内容

```
namespace std
{
  class thread;

  namespace this_thread
  {
    thread::id get_id() noexcept;

    void yield() noexcept;

    template<typename Rep,typename Period>
    void sleep_for(
        std::chrono::duration<Rep,Period> sleep_duration);

    template<typename Clock,typename Duration>
    void sleep_until(
        std::chrono::time_point<Clock,Duration> wake_time);
  }
}
```

D.7.1 std::thread类

std::thread 用来管理线程的执行。其提供让新的线程执行或执行，也提供对线程的识别，以及提供其他函数用于管理线程的执行。

```
class thread
{
public:
  // Types
```

```
  class id;
  typedef implementation-defined native_handle_type; // optional

  // Construction and Destruction
  thread() noexcept;
  ~thread();

  template<typename Callable,typename Args...>
  explicit thread(Callable&& func,Args&&... args);

  // Copying and Moving
  thread(thread const& other) = delete;
  thread(thread&& other) noexcept;

  thread& operator=(thread const& other) = delete;
  thread& operator=(thread&& other) noexcept;

  void swap(thread& other) noexcept;

  void join();
  void detach();
  bool joinable() const noexcept;

  id get_id() const noexcept;
  native_handle_type native_handle();
  static unsigned hardware_concurrency() noexcept;
};

void swap(thread& lhs,thread& rhs);
```

std::thread::id 类

可以通过 `std::thread::id` 实例对执行线程进行识别。

类型定义

```
class thread::id
{
public:
  id() noexcept;
};

bool operator==(thread::id x, thread::id y) noexcept;
```

```
bool operator!=(thread::id x, thread::id y) noexcept;
bool operator<(thread::id x, thread::id y) noexcept;
bool operator<=(thread::id x, thread::id y) noexcept;
bool operator>(thread::id x, thread::id y) noexcept;
bool operator>=(thread::id x, thread::id y) noexcept;

template<typename charT, typename traits>
basic_ostream<charT, traits>&
operator<< (basic_ostream<charT, traits>&& out, thread::id id);
```

Notes

`std::thread::id` 的值可以识别不同的执行，每个 `std::thread::id` 默认构造出来的值都不一样，不同值代表不同的执行线程。

`std::thread::id` 的值是不可预测的，在同一程序中的不同线程的id也不同。

`std::thread::id` 是可以CopyConstructible(拷贝构造)和CopyAssignable(拷贝赋值)，所以对于 `std::thread::id` 的拷贝和赋值是没有限制的。

std::thread::id 默认构造函数

构造一个 `std::thread::id` 对象，其不能表示任何执行线程。

声明

```
id() noexcept;
```

效果

构造一个 `std::thread::id` 实例，不能表示任何一个线程值。

抛出

无

NOTE 所有默认构造的 `std::thread::id` 实例存储的同一个值。

std::thread::id 相等比较操作

比较两个 `std::thread::id` 的值，看是两个执行线程是否相等。

声明

```
bool operator==(std::thread::id lhs,std::thread::id rhs) noexcept;
```

返回
当lhs和rhs表示同一个执行线程或两者不代表没有任何线程，则返回true。当lsh和rhs表示不同执行线程或其中一个代表一个执行线程，另一个不代表任何线程，则返回false。

抛出
无

std::thread::id 不相等比较操作

比较两个 `std::thread::id` 的值，看是两个执行线程是否相等。

声明

```
bool operator!=(std::thread::id lhs,std::thread::id rhs) noexcept;
```

返回
`!(lhs==rhs)`

抛出
无

std::thread::id 小于比较操作

比较两个 `std::thread::id` 的值，看是两个执行线程哪个先执行。

声明

```
bool operator<(std::thread::id lhs,std::thread::id rhs) noexcept;
```

返回
当lhs比rhs的线程ID靠前，则返回true。当lhs!=rhs，且 `lhs<rhs` 或 `rhs<lhs` 返回true，其他情况则返回false。当lhs==rhs，在 `lhs<rhs` 和 `rhs<lhs` 时返回false。

抛出
无

NOTE 当默认构造的 `std::thread::id` 实例，在不代表任何线程的时候，其值小于任何一个代表执行线程的实例。当两个实例相等，那么两个对象代表两个执行线程。任何一组不同的 `std::thread::id` 的值，是由同一序列构造，这与程序执行的顺序相同。同一个可执行程序可能有不同的执行顺序。

std::thread::id 小于等于比较操作

比较两个 `std::thread::id` 的值，看是两个执行线程的ID值是否相等，或其中一个先行。

声明

```
bool operator<(std::thread::id lhs,std::thread::id rhs) noexcept;
```

返回

`!(rhs<lhs)`

抛出

无

std::thread::id 大于比较操作

比较两个 `std::thread::id` 的值，看是两个执行线程的是后行的。

声明

```
bool operator>(std::thread::id lhs,std::thread::id rhs) noexcept;
```

返回

`rhs<lhs`

抛出

无

std::thread::id 大于等于比较操作

比较两个 `std::thread::id` 的值，看是两个执行线程的ID值是否相等，或其中一个后行。

声明

```
bool operator>=(std::thread::id lhs,std::thread::id rhs) noexcept;
```

返回

`!(lhs<rhs)`

抛出

无

std::thread::id 插入流操作

将 `std::thread::id` 的值通过给指定流写入字符串。

声明

```
template<typename charT, typename traits>
basic_ostream<charT, traits>&
operator<< (basic_ostream<charT, traits>&& out, thread::id id);
```

效果

将 `std::thread::id` 的值通过给指定流插入字符串。

返回

无

NOTE 字符串的格式并未给定。 `std::thread::id` 实例具有相同的表达式时，是相同的；当实例表达式不同，则代表不同的线程。

std::thread::native_handler 成员函数

`native_handle_type` 是由另一类型定义而来，这个类型会随着指定平台的API而变化。

声明

```
typedef implementation-defined native_handle_type;
```

NOTE 这个类型定义是可选的。如果提供，实现将使用原生平台指定的API，并提供合适的类型作为实现。

std::thread 默认构造函数

返回一个 `native_handle_type` 类型的值，这个值可以可以表示*this相关的执行线程。

声明

```
native_handle_type native_handle();
```

NOTE 这个函数是可选的。如果提供，会使用原生平台指定的API，并返回合适的值。

std::thread 构造函数

构造一个无相关线程的 `std::thread` 对象。

声明

```
thread() noexcept;
```

效果
构造一个无相关线程的 `std::thread` 实例。

后置条件
对于一个新构造的 `std::thread` 对象x，x.get_id() == id()。

抛出
无

std::thread 移动构造函数

将已存在 `std::thread` 对象的所有权，转移到新创建的对象中。

声明

```
thread(thread&& other) noexcept;
```

效果
构造一个 `std::thread` 实例。与other相关的执行线程的所有权，将转移到新创建的

`std::thread` 对象上。否则，新创建的 `std::thread` 对象将无任何相关执行线程。

后置条件
对于一个新构建的 `std::thread` 对象x来说，x.get_id()等价于未转移所有权时的other.get_id()。get_id()==id()。

抛出
无

NOTE `std::thread` 对象是不可CopyConstructible(拷贝构造)，所以该类没有拷贝构造函数，只有移动构造函数。

std::thread 析构函数

销毁 `std::thread` 对象。

声明

```
~thread();
```

效果
销毁 `*this` 。当 `*this` 与执行线程相关(this->joinable()将返回true)，调用 `std::terminate()` 来终止程序。

抛出
无

std::thread 移动赋值操作

将一个 `std::thread` 的所有权，转移到另一个 `std::thread` 对象上。

声明

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

效果
在调用该函数前，this->joinable返回true，则调用 `std::terminate()` 来终止程序。当other在执

行赋值前，具有相关的执行线程，那么执行线程现在就与 `*this` 相关联。否则， `*this` 无相关执行线程。

后置条件
this->get_id()的值等于调用该函数前的other.get_id()。oter.get_id()==id()。

抛出
无

NOTE `std::thread` 对象是不可CopyAssignable(拷贝赋值)，所以该类没有拷贝赋值函数，只有移动赋值函数。

std::thread::swap 成员函数

将两个 `std::thread` 对象的所有权进行交换。

声明

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

效果
当other在执行赋值前，具有相关的执行线程，那么执行线程现在就与 `*this` 相关联。否则， `*this` 无相关执行线程。对于 `*this` 也是一样。

后置条件
this->get_id()的值等于调用该函数前的other.get_id()。other.get_id()的值等于没有调用函数前this->get_id()的值。

抛出
无

std::thread的非成员函数swap

将两个 `std::thread` 对象的所有权进行交换。

声明

```
void swap(thread& lhs,thread& rhs) noexcept;
```

效果
lhs.swap(rhs)

抛出
无

std::thread::joinable 成员函数

查询*this是否具有相关执行线程。

声明

```
bool joinable() const noexcept;
```

返回
如果*this具有相关执行线程，则返回true；否则，返回false。

抛出
无

std::thread::join 成员函数

等待*this相关的执行线程结束。

声明

```
void join();
```

先决条件
this->joinable()返回true。

效果
阻塞当前线程，直到与*this相关的执行线程执行结束。

后置条件
this->get_id()==id()。与*this先关的执行线程将在该函数调用后结束。

同步
想要在*this上成功的调用该函数，则需要依赖有joinable()的返回。

抛出
当效果没有达到或this->joinable()返回false，则抛出 `std::system_error` 异常。

std::thread::detach 成员函数

将*this上的相关线程进行分离。

声明

```
void detach();
```

先决条件
this->joinable()返回true。

效果
将*this上的相关线程进行分离。

后置条件
this->get_id()==id(), this->joinable()==false

与*this相关的执行线程在调用该函数后就会分离，并且不在会与当前 `std::thread` 对象再相关。

抛出
当效果没有达到或this->joinable()返回false，则抛出 `std::system_error` 异常。

std::thread::get_id 成员函数

返回 `std::thread::id` 的值来表示*this上相关执行线程。

声明

```
thread::id get_id() const noexcept;
```

返回
当*this具有相关执行线程，将返回 `std::thread::id` 作为识别当前函数的依据。否则，返回默认构造的 `std::thread::id` 。

抛出
无

std::thread::hardware_concurrency 静态成员函数

返回硬件上可以并发线程的数量。

声明

```
unsigned hardware_concurrency() noexcept;
```

返回
硬件上可以并发线程的数量。这个值可能是系统处理器的数量。当信息不用或只有定义，则该函数返回0。

抛出
无

D.7.2 this_thread命名空间

这里介绍一下 `std::this_thread` 命名空间内提供的函数操作。

this_thread::get_id 非成员函数

返回 `std::thread::id` 用来识别当前执行线程。

声明

```
thread::id get_id() noexcept;
```

返回
可通过 `std:thread::id` 来识别当前线程。

抛出
无

this_thread::yield 非成员函数

该函数用于通知库，调用线程不需要立即运行。一般使用小循环来避免消耗过多CPU时间。

声明

```
void yield() noexcept;
```

效果
使用标准库的实现来安排线程的一些事情。

抛出
无

this_thread::sleep_for 非成员函数

在指定的指定时长内，暂停执行当前线程。

声明

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

效果
在超出relative_time的时长内，阻塞当前线程。

NOTE 线程可能阻塞的时间要长于指定时长。如果可能，逝去的时间由将会由一个稳定时钟决定。

抛出
无

this_thread::sleep_until 非成员函数

暂停指定当前线程，直到到了指定的时间点。

声明

```
template<typename Clock,typename Duration>
```

```
void sleep_until(
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

效果

在到达absolute_time的时间点前，阻塞当前线程，这个时间点由指定的Clock决定。

NOTE 这里不保证会阻塞多长时间，只有Clock::now()返回的时间等于或大于absolute_time时，阻塞的线程才能被解除阻塞。

抛出

无