[**C++11的线程库（一）**](http://blog.csdn.net/charles_r_chiu/article/details/46049633)

标签： [c++](http://www.csdn.net/tag/c%2b%2b)[c++11](http://www.csdn.net/tag/c%2b%2b11)[编程](http://www.csdn.net/tag/%e7%bc%96%e7%a8%8b)

2015-05-27 17:24 2436人阅读 [评论](http://blog.csdn.net/charles_r_chiu/article/details/46049633#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/charles_r_chiu/article/details/46049633#report)

http://static.blog.csdn.net/images/category_icon.jpg分类：

C++*（22）* http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：转载请注明出处：http://blog.csdn.net/charles\_r\_chiu

目录[(?)[+]](http://blog.csdn.net/charles_r_chiu/article/details/46049633)

[TOC]   
个人认为C++11最明智的举动之一就是加入了线程库，在多核CPU越来越普及，在多线程编程日益，我们可以说是”疯狂“的时代，一个不支持多线程（我是指本身不支持，在编程过程中需要依赖线程库）的语言，如何能普遍适用？   
在C++11之前在C/C++中使用多线程编程并非鲜见，这样的代码主要是使用POSIX线程Pthread和OpenMP编译器指令两种编程模型来完成程序的线程化。其中POSIX线程是POSIX标准中关于线程的部分，程序员可以通过一些Pthread线程的API来完成线程的创建、数据的共享、同步等功能。Pthread主要用于C语言，在类UNIX系统上，如FreeBSD、NetBSD、OpenBSD、GNU/Linux、Mac OS X，甚至在Windows上也都有实现，不过Windows上Pthread的实现并非”原生“，主要还是包装为Windows的线程库。不过在使用的便利性上，Pthread不如后来者OpenMP。

**原子操作与C++11原子类型**

// 原子类型.cpp : 定义控制台应用程序的入口点。

//

#include "stdafx.h"

#include <iostream>

#include <thread>

#include <atomic>

using namespace std;

atomic\_llong total{ 0 };

void func(int){

for (long long i = 0; i < 100000000LL; ++i){

total += i;

}

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

thread t1(func, 0);

thread t2(func, 0);

t1.join();

t2.join();

cout << total << endl;

system("pause");

return 0;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24

相比于基于C语言以及过程编程的pthread“原子操作API”而言，C++11对于“原子操作”概念的抽象遵从了面向对象的思想——C++11标准定义的都是所谓的“原子类型”。编译器可以保证原子类型在线程间被互斥的访问，这样设计从并行编程的角度看，是由于需要同步的总是数据而不是代码，因此C++11对数据进行了抽象，会有利于产生行为更为良好的并行代码。而进一步地，一些琐碎的概念，比如互斥锁、临界区则可被C++11的抽象所掩盖，因此并行代码的编写也会变得更加简单。我们可以在看到内置类型的原子类型的定义：

atomic\_bool abool; //对应bool

atomic\_char achar; //char

atomic\_schar aschar; //signed char

atomic\_uchar auchar; //unsigned char

atomic\_int aint; //int

atomic\_uint auint; //unsigned int

atomic\_short ashort; //short

atomic\_ushort aushort; //unsigned short

atomic\_long along; //long

atomic\_ulong aulong; //unsigned long

atomic\_llong allong; //long long

atomic\_ullong aullong; //unsigned long long

atomic\_char16\_t achar16\_t; //char16\_t

atomic\_char32\_t achar32\_t; //char32\_t;

atomic\_wchar\_t awchar\_t; //wchar\_t

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15

不过更为普遍的应该是使用atomic类模板。通过该模板，可以定义出任意需要的原子类型：   
std::atomic t;   
如上，声明可一个类型为T的原子类型变量t。编译器会保证产生并行情况下行为良好的代码，以避免线程之间对于数据t的竞争。对于线程而言，原子类型通常属于“资源型”的数据，这意味着多个线程通常只能访问的原子类型的拷贝。因此在C++11中，原子类型只能从其模板参数类型中进行构造，标准不允许原子类型经行拷贝构造、移动构造，以及operator=等，防止以外发生如下面：

atomic<float> af{ 1.2f };

//atomic<float> af1{ af }; //这里无法编译

* 1
* 2

从上面可以看到，af1{ af }的构造方式在C++11中是不允许的，我们可以通过以前的经验轻松知道如何在类的代码中禁止这些行为，事实上，atomic模板类的拷贝构造函数、移动构造函数、operator=等总是默认被删除的。不过从atomic类型的变量来构造其他模板参数类型T的变量则是可以的，比如：

atomic<float> af{ 1.2f };

//atomic<float> af1{ af }; //这里无法编译

//下面的都是正确的

float af2 = af;

vector<float> vfl{ af };

stack<float> sfl;

sfl.push(af);

* 1
* 2
* 3
* 4
* 5
* 6
* 7

这是由于atomic类模板总是定义了从atomic到T的类型转换函数的缘故，在需要的时候，编译器会隐式地完成完成原子类型到其对应的类型的转化。能够实现在线程间保持原子性的原因是编译器能够保证针对原子类型的操作都是原子操作。正如之前所说，原子操作都是平台相关的，因此有必要为常见的原子操作进行抽象，定义统一的结构，并根据编译选项，并根据编译选项（或环境）产生其平台相关的实现。在C++11中，标准将原子操作定义为atomic模板类的成员函数，这囊括了绝大多数典型的操作，如读、写、交换等，当然，对于内置类型而言，主要是通过重载一些全局操作符来完成的，在编译的时候，会产生一条特殊的lock前缀的x86指令，lock能够控制总线及实现x86平台上的原子性。下面是atomic类型及相关的操作：

| **操作** | **atomic\_flag** | **atomic\_bool** | **atomic\_integral-type** | **atomic** | **atomic** | **atomic** | **atomic** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| test\_and\_set | Y |  |  |  |  |  |  |
| clear | Y |  |  |  |  |  |  |
| is\_lock\_free |  | y | y | y | y | y | y |
| load |  | y | y | y | y | y | y |
| store |  | y | y | y | y | y | y |
| exchange |  | y | y | y | y | y | y |
| compare\_exchange\_weak+strong |  | y | y | y | y | y | y |
| fetch\_add,+= |  |  | y |  | y | y |  |
| fetch\_sub,-= |  |  | y |  | y | y |  |
| fetch\_or,|= |  | y |  |  | y |  |  |
| fetch\_and,&= |  |  | y |  |  | y |  |
| fetch\_xor,^= |  |  | y |  |  | y |  |
| ++,– |  |  | y |  | y | y | y |

这里的atomic-integral-type和integraltype指的是前面提到的所有的原子类型的整型，而class-type则是指自定义类型。可以看到，对于大多数的原子类型而言，都可以执行读（load）、写（store）、交换（exchange）、比较并交换（compare\_exchange\_weak/compare\_exchange\_stronge）等操作。通常情况下，这些原子操作已经足够使用了。如下：

atomic<int> a;

a = 1; //a.store(1);

int b = a; //b = a.load();

* 1
* 2
* 3

这里的赋值语句b=a其实就等价b=a.load()。同样，a=1也相当于a.store(1)，由于这些操作都是原子的，所以原来的从操作也是原子的，从而避免了线程间关于a的竞争。由于封装内部的实现和平台相关，所以，一般来说这些接口都封装了平台上最高性能的实现。   
这里首先要注意的是atomic\_flag，这个类型和其他不同，atomic\_flag是无锁的（lock-free），级线程对其访问不需要加锁，因此对于atomic\_flag而言，也就不需要使用load、store等成员函数进行读写，但是可以通过atomic\_flag的成员test\_and\_set以及clear实现一个自旋锁（spinlock）：

// 自旋锁实现.cpp : 定义控制台应用程序的入口点。

//

#include "stdafx.h"

#include <iostream>

#include <atomic>

#include <thread>

#include <Windows.h>

using namespace std;

std::atomic\_flag lock = ATOMIC\_FLAG\_INIT; //声明了全局变量，初始化为值ATOMIC\_FLAG\_INIT，即false状态

void func(int n){

while (lock.test\_and\_set(std::memory\_order\_acquire)) //尝试获得自旋锁

{

cout << "Waiting from thread : " << n << endl; //自旋

}

cout << "Thread " << n << " starts working" << endl;

}

void foo(int n){

cout << " Thread " << n << " is going to start" << endl;

lock.clear();

cout << " Thread " << n << " starting working" << endl;

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

lock.test\_and\_set();

thread t1(func, 1);

thread t2(foo, 2);

t1.join();

Sleep(100);

t2.join();

Sleep(100);

system("pause");

return 0;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33

在上面的程序，我们先是申明了一个全局变量的atomic\_flag变量lock，并且初始化为ATOMIC\_FLAG\_INIT，也就是false的状态。在线程t1中，通过不停地通过lock的成员test\_and\_set()来设置lock为true。这里的test\_and\_set是一种原子操作，用于在一个内存空间原子地写入新植并返回旧值。因此test\_and\_set会返回之前的lock值。由于31行中设置了lock的值为true，所以线程t1会一直自旋在while循环中，知道线程t2将该设置位通过成员函数clear设置为false，此时线程t1自旋结束，开始后面的代码。当然还可以将lock封装为锁操作，比如：

void Lock(atomic\_flag \*lock) { while (lock->test\_and\_set()); }

void Ublock(atomic\_flag \*lock) { lock->clear(); }

* 1
* 2

这样一来，就可以想以前那样互斥的访问临界区了。除此之外，很多时候，了解底层的程序员会考虑使用无锁编程，以最大限度地挖掘并行编程的性能，而C++11的无锁机制为这样的实现提供了高级语言的支持。事实上，C++11中，原子操作还可以包含一个参数：memory\_order，使用该参数能够进一步释放并行的潜在的性能。   
如果只是简单地想在线程间进行数据的同步的话，上面说到的原子类型已经为程序员提供了一些同步保障。不过这样做的线程安全是基于一种假设，即所谓的顺序一致性(sequential consistent)的内存模型(memory model)。如下：

// 内存模型顺序一致性.cpp : 定义控制台应用程序的入口点。

//

#include "stdafx.h"

#include <iostream>

#include <thread>

#include <atomic>

using namespace std;

atomic<int> a{ 0 };

atomic<int> b{ 0 };

void ValueSet(int){

int t = 1;

a = t;

b = 2;

}

void Observer(int){

cout << "(" << a << "," << b << ")" << endl; //可能有多少输出

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

thread t1(ValueSet, 0);

thread t2(Observer, 0);

t1.join();

t2.join();

cout << "Got (" << a << "," << b << ")" << endl; //Got(1,2)

getchar(); //pause

return 0;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27

在上面的代码中，我们创建了两个线程t1和t2，分别执行ValueSet和Observer函数。在ValueSet中，为a和b的值分别是1和2，而在Observer中，只打印出a和b的值。其实由于两个线程的中关于a和b顺序的不确定性，因此，Observer输出的结果可能是多种可能性，如下所示：

在本例中Observer只是试图一窥线程ValueSet的执行状况，不过在 本例中Observer的窥探对于最终结果并不是必须的，因此赋值语句的执行顺序，也就是ValueSet的执行时间顺序并不会对最后的结果有什么影响。如果编译器认定a、b的赋值语句的执行先后顺序对输出结果有任何影响的话，则可以依情况将指令重排列(reorder)以提高性能，如果a、b赋值语句的执行顺序必须是先a后b，则编译器不会执行这样的优化。如果原子操作的先后顺序置之不理，很可能发生错误：

// 线程顺序.cpp : 定义控制台应用程序的入口点。

//

#include "stdafx.h"

#include <iostream>

#include <thread>

#include <atomic>

using namespace std;

atomic<int> a;

atomic<int> b;

void Thread1(int)

{

int t = 1;

a = t;

b = 2;

}

void Thread2(int){

while (b != 2); //自旋等待

cout << a << endl; //总是期待a的值为1

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

thread t1(Thread1, 0);

thread t2(Thread2, 0);

t1.join();

t2.join();

system("pause");

return 0;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28

在默认情况下，C++11中原子类型的变量在线程中总是保持着顺序执行的特点（非原子类型则没有必要，因为不需要在线程之间进行同步），我们称这样的特性为顺序一致的，即代码在线程中运行的顺序与程序员看到的代码顺序一致。事实上，顺序一致性只是C++11中多种内存模型中的一种，而在C++11中，并不是只支持顺序一致单个内存模型的原子变量，以为顺序一致性往往意味着最低效的同步方式，C++11提供了更高效的原子类型变量同步方式。   
内存模型通常是一个硬件上的概念，表示的是机器指令是以什么样的顺序被处理器执行的。内存模型分为强顺序和弱顺序。多线程总是共享代码的，那么强顺序意味着：对于多个线程而言，其看到的指令执行顺序是一致的。具体而言，对于共享内存的处理器而言，需要看到内存中的数据被改变的顺序与机器指令中的一致，反之，如果线程间看到的内存数据被改变的顺序与机器指令中声明的不一致的话，则是弱顺序。在现实实现中采用强顺序的内存模型的平台，对于任何一个线程而言，其看到的原子操作都是顺序。而采用弱顺序内存模型的平台，比如PowerPC、Alpha、Itanlium、ArmV7，如果想要保证指令执行的顺序，通常要在汇编中加入一条所谓的内存栅栏（memory barrier）指令。为什么会有弱顺序内存模型？简单地说，弱顺序内存模型可以使得处理器进一步发掘指令中的并行性，使得指令执行的性能高效。   
上面都是硬件上的一些可能的内存模型的描述。而C++11中定义的内存模型和顺序一致性跟硬件的内存模型的强顺序、弱顺序之间其实没有什么太强的关系。事实上，在高级语言与机器指令之间还有一层隔离，这层隔离是由编译器来完成的。编译器处于代码优化的考虑，会将指令前后移动，从而获得最佳的机器指令的排列以及产生最佳的运行时性能。究其C++11而言，要保证代码的顺序一致性，就必须同时做到：   
编译器保证原子操作的指令间顺序不变，即保证产生的读写原子类型的变量的机器指令与代码编写者可拿到的是一致的。   
处理器对原子操作的汇编指令的执行顺序不变。这对于x86这样的强顺序的体系结构而言没有任何问题；对于弱顺序的体系结构而言则需要编译器在每次原子操作后加入内存栅栏。   
那么，C++11中，对于原子类型的成员函数（原子操作）总是保证了顺序一致性。这对于x86这些平台只需要禁止编译器对原子类型变量间的重排序优化。而对于PowerPC这样的平台来说，不仅禁止了编译器的优化，还加入了大量的内存栅栏。在C++11中总共定义了7中memory\_order的枚举值：

| **枚举值** | **定义说明** |
| --- | --- |
| memory\_order\_relaxed | 不对执行顺序做任何保证 |
| memory\_order\_acquire | 本线程中，所有后续的读操作必须在本条原子操作完成后执行 |
| memory\_order\_release | 本线程中，所有之前的写操作完成后才能执行本条原子操作 |
| memory\_order\_acq\_rel | 同时包含memory\_order\_acquire和memory\_order\_release标记 |
| memory\_order\_consume | 本线程中，所有后续的有关本原子类型的操作，必须在本条原子操作完成之后执行 |
| memory\_order\_seq\_cst | 全部存取都按顺序执行 |

memory\_order\_seq\_cst表示该原子操作必须是顺序一致的，这是C++11中所有atomic原子操作的默认值，不带memory\_order参数的原子操作就是使用该值。而memory\_order\_relaxed则表示该原子操作是松散的，可以被任意重排序的。通常我们可以把atomic成员函数可使用的memory\_order的值分为以下3种：   
原子存储操作（store）可以使用memory\_order\_relaxed、memory\_order\_release、memory\_order\_seq\_cst。   
原子读取操作（load）可以使用memory\_order\_relaxed、memory\_order\_consume、memory\_order\_acquire、memory\_order\_seq\_cst。   
RMW操作（read-modify-write），即一些需要同时读写的操作，比如之前提过的atomic\_flag类型的test\_and\_set()操作。又比如atomic类模板的atomic\_compare\_exchange()操作等都是同时需要读写的。RMW操作可以使用memory\_order\_relaxed、memory\_order\_consume、memory\_order\_acquire、memory\_order\_release、memory\_order\_acq\_rel、memory\_order\_seq\_cst。   
一些形如“operator =”、“operator +=”的函数，事实上都是memory\_order\_seq\_cst作为memory\_order参数的原子操作的简单封装。这些操作都是采用顺序一致性的内存模型。如果要指定内存顺序的话，应该使用load、atomic\_fetch\_add这样的版本。

// 内存模型\_memory\_order.cpp : 定义控制台应用程序的入口点。

//

#include "stdafx.h"

#include <iostream>

#include <thread>

#include <atomic>

using namespace std;

atomic<int> a;

atomic<int> b;

int Thread1(int)

{

int t = 1;

a.store(t, memory\_order\_relaxed);

b.store(2, memory\_order\_relaxed);

return 0;

}

int Thread2(int){

while (b.load(memory\_order\_relaxed) != 2); //自旋等待

cout << a.load(memory\_order\_relaxed) << endl;

return 0;

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

thread t1(Thread1, 0);

thread t2(Thread2, 0);

t1.join();

t2.join();

system("pause");

return 0;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30

上面的代码采用了memory\_order\_relaxed最为memory\_order参数。那么排除顺序一致和松散两种方式，如何保证程序既快又对的运行呢？仔细分析上面的程序，发现只需要保证a.store先于b.store发生，b.load先于a.load发生就行。下面我们使用其他的memory\_order实现这种原子操作：

// 内存模型\_memory\_order\_1.cpp : 定义控制台应用程序的入口点。

//

#include "stdafx.h"

#include <iostream>

#include <atomic>

#include <thread>

using namespace std;

atomic<int> a;

atomic<int> b;

int Thread1(int)

{

int t = 1;

a.store(t, memory\_order\_relaxed); //不对执行顺序做任何保证

b.store(2, memory\_order\_release); //本原子操作前多有的写原子操作必须完成

return 0;

}

int Thread2(int){

while (b.load(memory\_order\_acquire) != 2); //本原子操作必须完成才能执行之后的所有读原子操作

cout << a.load(memory\_order\_relaxed) << endl; //不对执行顺序做任何保证

return 0;

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

thread t1(Thread1, 0);

thread t2(Thread2, 0);

t1.join();

t2.join();

system("pause");

return 0;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30

上面的代码中我们只是确定了代码执行的部分顺序。

***线程局部存储***

线程局部存储(TLS,thread local storage)是一个已经有的概念。所谓的局部存储变量就是拥有线程生命期及线程可见性的变量。C++11只是对TLS做了语法上的统一，而对其实现并没有做任何性能上的规定：   
thread\_local int rage=1;   
从这里<https://msdn.microsoft.com/en-us/library/hh567368.aspx#featurelist>，可以看到vs并不完全支持。从cppreference(<http://en.cppreference.com/w/cpp/language/storage_duration>) 上面可以看到一些简单的例子：

#include <iostream>

#include <string>

#include <thread>

#include <mutex>

thread\_local unsigned int rage = 1;

std::mutex cout\_mutex;

void increase\_rage(const std::string& thread\_name)

{

++rage;

std::lock\_guard<std::mutex> lock(cout\_mutex);

std::cout << "Rage counter for " << thread\_name << ": " << rage << '\n';

}

int main()

{

std::thread a(increase\_rage, "a"), b(increase\_rage, "b");

{

std::lock\_guard<std::mutex> lock(cout\_mutex);

std::cout << "Rage counter for main: " << rage << '\n';

}

a.join();

b.join();

return 0;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29

可以得到输出：

Rage counter for main: 1

Rage counter for a: 2

Rage counter for b: 2

* 1
* 2
* 3

上面的代码中使用了thread\_local的变量限定，使得变量rage只在每个线程中初始化，并且生命周期与属于线程相同，并且在线程中的操作并不会影响到其他线程中的值，我们可以认为这个变量是有了“归属”。

**thread的成员变量**

为了更好的使用thread，我们在下面讨论下其成员变量：

// thread C++11 header

#pragma once

#ifndef \_THREAD\_

#define \_THREAD\_

#ifndef RC\_INVOKED

#ifdef \_M\_CEE

#error <thread> is not supported when compiling with /clr or /clr:pure.

#endif /\* \_M\_CEE \*/

#include <exception>

#include <iosfwd>

#include <functional>

#include <chrono>

#include <concrt.h>

#include <thr/xthread>

#include <thr/xtimec.h>

#pragma pack(push,\_CRT\_PACKING)

#pragma warning(push,3)

#pragma push\_macro("new")

#undef new

#pragma warning(disable: 4521 4522 4800)

#define \_\_STDCPP\_THR\_\_ 1 /\* nonzero if threads supported \*/

\_STD\_BEGIN

class thread

{ // class for observing and managing threads

public:

class id;

typedef void \*native\_handle\_type;

thread() \_NOEXCEPT

{ // construct with no thread

\_Thr\_set\_null(\_Thr);

}

template<class \_Fn,

class... \_Args>

explicit thread(\_Fn&& \_Fx, \_Args&&... \_Ax)

{ // construct with \_Fx(\_Ax...)

\_Launch(&\_Thr,

\_STD bind(\_Decay\_copy(\_STD forward<\_Fn>(\_Fx)),

\_Decay\_copy(\_STD forward<\_Args>(\_Ax))...));

}

~thread() \_NOEXCEPT

{ // clean up

if (joinable())

\_XSTD terminate();

}

thread(thread&& \_Other) \_NOEXCEPT

: \_Thr(\_Other.\_Thr)

{ // move from \_Other

\_Thr\_set\_null(\_Other.\_Thr);

}

thread& operator=(thread&& \_Other) \_NOEXCEPT

{ // move from \_Other

return (\_Move\_thread(\_Other));

}

thread(const thread&) = delete;

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

void swap(thread& \_Other) \_NOEXCEPT

{ // swap with \_Other

\_STD swap(\_Thr, \_Other.\_Thr);

}

bool joinable() const \_NOEXCEPT

{ // return true if this thread can be joined

return (!\_Thr\_is\_null(\_Thr));

}

void join();

void detach()

{ // detach thread

if (!joinable())

\_Throw\_Cpp\_error(\_INVALID\_ARGUMENT);

\_Thrd\_detachX(\_Thr);

\_Thr\_set\_null(\_Thr);

}

id get\_id() const \_NOEXCEPT;

static unsigned int hardware\_concurrency() \_NOEXCEPT

{ // return number of hardware thread contexts

return (::Concurrency::details::\_GetConcurrency());

}

native\_handle\_type native\_handle()

{ // return Win32 HANDLE as void \*

return (\_Thr.\_Hnd);

}

private:

thread& \_Move\_thread(thread& \_Other)

{ // move from \_Other

if (joinable())

\_XSTD terminate();

\_Thr = \_Other.\_Thr;

\_Thr\_set\_null(\_Other.\_Thr);

return (\*this);

}

\_Thrd\_t \_Thr;

};

namespace this\_thread {

thread::id get\_id() \_NOEXCEPT;

inline void yield() \_NOEXCEPT

{ // give up balance of time slice

if (::Concurrency::details::\_CurrentScheduler::\_Id() != -1)

{ // yield, then quit

::Concurrency::details::\_Context::\_Yield();

return;

}

\_Thrd\_yield();

}

inline void sleep\_until(const stdext::threads::xtime \*\_Abs\_time)

{ // sleep until \_Abs\_time

if (::Concurrency::details::\_CurrentScheduler::\_Id() != -1)

{

stdext::threads::xtime \_Now;

stdext::threads::xtime\_get(&\_Now, stdext::threads::TIME\_UTC);

::Concurrency::wait(\_Xtime\_diff\_to\_millis2(\_Abs\_time, &\_Now));

return;

}

\_Thrd\_sleep(\_Abs\_time);

}

template<class \_Clock,

class \_Duration> inline

void sleep\_until(

const chrono::time\_point<\_Clock, \_Duration>& \_Abs\_time)

{ // sleep until time point

stdext::threads::xtime \_Tgt;

\_Tgt.sec = chrono::duration\_cast<chrono::seconds>(

\_Abs\_time.time\_since\_epoch()).count();

\_Tgt.nsec = (long)chrono::duration\_cast<chrono::nanoseconds>(

\_Abs\_time.time\_since\_epoch() - chrono::seconds(\_Tgt.sec)).count();

sleep\_until(&\_Tgt);

}

template<class \_Rep,

class \_Period> inline

void sleep\_for(const chrono::duration<\_Rep, \_Period>& \_Rel\_time)

{ // sleep for duration

stdext::threads::xtime \_Tgt = \_To\_xtime(\_Rel\_time);

sleep\_until(&\_Tgt);

}

} // namespace this\_thread

class thread::id

{ // thread id

public:

id() \_NOEXCEPT

{ // id for no thread

\_Thr\_set\_null(\_Thr);

}

template<class \_Ch,

class \_Tr>

basic\_ostream<\_Ch, \_Tr>& \_To\_text(

basic\_ostream<\_Ch, \_Tr>& \_Str)

{ // insert representation into stream

return (\_Str << \_Thr\_val(\_Thr));

}

size\_t hash() const

{ // hash bits to size\_t value by pseudorandomizing transform

return (\_STD hash<size\_t>()((size\_t)\_Thr\_val(\_Thr)));

}

private:

id(const thread& \_Thrd)

: \_Thr(\_Thrd.\_Thr)

{ // construct from thread object

}

id(\_Thrd\_t \_Thrd)

: \_Thr(\_Thrd)

{ // construct from thread identifier

}

\_Thrd\_t \_Thr;

friend thread::id thread::get\_id() const \_NOEXCEPT;

friend thread::id this\_thread::get\_id() \_NOEXCEPT;

friend bool operator==(thread::id \_Left, thread::id \_Right) \_NOEXCEPT;

friend bool operator<(thread::id \_Left, thread::id \_Right) \_NOEXCEPT;

};

inline void thread::join()

{ // join thread

if (!joinable())

\_Throw\_Cpp\_error(\_INVALID\_ARGUMENT);

if (\_Thr\_is\_null(\_Thr))

\_Throw\_Cpp\_error(\_INVALID\_ARGUMENT);

if (get\_id() == \_STD this\_thread::get\_id())

\_Throw\_Cpp\_error(\_RESOURCE\_DEADLOCK\_WOULD\_OCCUR);

if (\_Thrd\_join(\_Thr, 0) != \_Thrd\_success)

\_Throw\_Cpp\_error(\_NO\_SUCH\_PROCESS);

\_Thr\_set\_null(\_Thr);

}

inline thread::id thread::get\_id() const \_NOEXCEPT

{ // return id for this thread

return (id(\*this));

}

inline thread::id this\_thread::get\_id() \_NOEXCEPT

{ // return id for current thread

return (\_Thrd\_current());

}

inline void swap(thread& \_Left, thread& \_Right) \_NOEXCEPT

{ // swap \_Left with \_Right

\_Left.swap(\_Right);

}

inline bool operator==(thread::id \_Left, thread::id \_Right) \_NOEXCEPT

{ // return true if \_Left and \_Right identify the same thread

return (\_Thrd\_equal(\_Left.\_Thr, \_Right.\_Thr));

}

inline bool operator!=(thread::id \_Left, thread::id \_Right) \_NOEXCEPT

{ // return true if \_Left and \_Right do not identify the same thread

return (!(\_Left == \_Right));

}

inline bool operator<(thread::id \_Left, thread::id \_Right) \_NOEXCEPT

{ // return true if \_Left precedes \_Right

return (\_Thrd\_lt(\_Left.\_Thr, \_Right.\_Thr));

}

inline bool operator<=(thread::id \_Left, thread::id \_Right) \_NOEXCEPT

{ // return true if \_Left precedes or equals \_Right

return (!(\_Right < \_Left));

}

inline bool operator>(thread::id \_Left, thread::id \_Right) \_NOEXCEPT

{ // return true if \_Left follows \_Right

return (\_Right < \_Left);

}

inline bool operator>=(thread::id \_Left, thread::id \_Right) \_NOEXCEPT

{ // return true if \_Left follows or equals \_Right

return (!(\_Left < \_Right));

}

template<class \_Ch,

class \_Tr>

basic\_ostream<\_Ch, \_Tr>& operator<<(

basic\_ostream<\_Ch, \_Tr>& \_Str,

thread::id \_Id)

{ // insert id into stream

return (\_Id.\_To\_text(\_Str));

}

// TEMPLATE STRUCT SPECIALIZATION hash

template<>

struct hash<thread::id>

: public unary\_function<thread::id, size\_t>

{ // hash functor for thread::id

typedef thread::id \_Kty;

size\_t operator()(const \_Kty& \_Keyval) const

{ // hash \_Keyval to size\_t value by pseudorandomizing transform

return (\_Keyval.hash());

}

};

\_STD\_END

#pragma pop\_macro("new")

#pragma warning(pop)

#pragma pack(pop)

#endif /\* RC\_INVOKED \*/

#endif /\* \_THREAD\_ \*/

/\*

\* Copyright (c) 1992-2012 by P.J. Plauger. ALL RIGHTS RESERVED.

\* Consult your license regarding permissions and restrictions.

V6.00:0009 \*/

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83
* 84
* 85
* 86
* 87
* 88
* 89
* 90
* 91
* 92
* 93
* 94
* 95
* 96
* 97
* 98
* 99
* 100
* 101
* 102
* 103
* 104
* 105
* 106
* 107
* 108
* 109
* 110
* 111
* 112
* 113
* 114
* 115
* 116
* 117
* 118
* 119
* 120
* 121
* 122
* 123
* 124
* 125
* 126
* 127
* 128
* 129
* 130
* 131
* 132
* 133
* 134
* 135
* 136
* 137
* 138
* 139
* 140
* 141
* 142
* 143
* 144
* 145
* 146
* 147
* 148
* 149
* 150
* 151
* 152
* 153
* 154
* 155
* 156
* 157
* 158
* 159
* 160
* 161
* 162
* 163
* 164
* 165
* 166
* 167
* 168
* 169
* 170
* 171
* 172
* 173
* 174
* 175
* 176
* 177
* 178
* 179
* 180
* 181
* 182
* 183
* 184
* 185
* 186
* 187
* 188
* 189
* 190
* 191
* 192
* 193
* 194
* 195
* 196
* 197
* 198
* 199
* 200
* 201
* 202
* 203
* 204
* 205
* 206
* 207
* 208
* 209
* 210
* 211
* 212
* 213
* 214
* 215
* 216
* 217
* 218
* 219
* 220
* 221
* 222
* 223
* 224
* 225
* 226
* 227
* 228
* 229
* 230
* 231
* 232
* 233
* 234
* 235
* 236
* 237
* 238
* 239
* 240
* 241
* 242
* 243

上面是从vs2013中摘取的关于thread的部分源码。虽然不是完全支持整个C++11，但是毕竟好用一些。   
首先是成员类型（Member types）：

| **Member type** | **Definition** |
| --- | --- |
| native\_handle\_type | implementation-defined |

成员类   
id ： 用来描述线程的id，成员级别是public。   
成员函数

| **Member function** | **成员级别** | **作用** |
| --- | --- | --- |
| (constructor)构造函数 | public | 构建一个thread对象 |
| 析构函数 | public | 析构一个使用过或者被分离的thread对象 |
| operator= | public | 移动thread对象 |
| joinable | public | 检查这个线程是否是可以运行的，例如潜在的并行运行的上下文 |
| get\_id | public | 返回线程的id |
| native\_handle | public | 返回底层实现线程的处理 |
| hardware\_concurrency | public static | 返回实际支持的并发线程的数量 |
| join | public | 等待一个线程完成并退出 |
| detach | public | 线程执行独立的线程处理 |
| swap | public | 交换两个线程对象 |

非成员函数   
std::swap(std::thread)：专业的std::swap算法。   
简单的用法如下：

#include <iostream>

#include <thread>

#include <mutex>

#include <atomic>

using namespace std;

void func1(int a){

cout << "this is function 1" << endl;

cout << " the input of function 1 is: " << a << endl;

}

void func2(int a){

cout << "this is function 2" << endl;

cout << " the input of function 2 is: " << a << endl;

}

int main()

{

cout << thread::hardware\_concurrency() << endl; //返回实际支持的并发线程的数量

thread t1(func1, 0);

thread t2(func2, 10);

cout << t1.get\_id() << endl;

cout << t2.get\_id() << endl;

/\*t1.join();

t2.join();\*/

system("pause");

return 0;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25

当然这么写程序是有问题的，多线程编程中，如果想要程序能够按照自己的设计的发展，就必须要使用互斥、锁等操作，使得整个程序的运行脉络更加清晰。下面分析mutex、atomic类等的成员以及使用。