1.在C++11中创建新线程

　　在每个c++应用程序中，都有一个默认的主线程，即main函数，在c++11中，我们可以通过创建std::thread类的对象来创建其他线程，每个std :: thread对象都可以与一个线程相关联，只需包含头文件< thread>。可以使用std :: thread对象附加一个回调，当这个新线程启动时，它将被执行。 这些回调可以为函数指针、函数对象、Lambda函数。

　　线程对象可通过std::thread thObj(< CALLBACK>)来创建，新线程将在创建新对象后立即开始，并且将与已启动的线程并行执行传递的回调。此外，任何线程可以通过在该线程的对象上调用join()函数来等待另一个线程退出。

使用函数指针创建线程：

//main.cpp

#include <iostream>

#include <thread>

void thread\_function() {

for (int i = 0; i < 5; i++)

std::cout << "thread function excuting" << std::endl;

}

int main() {

std::thread threadObj(thread\_function);

for (int i = 0; i < 5; i++)

std::cout << "Display from MainThread" << std::endl;

threadObj.join();

std::cout << "Exit of Main function" << std::endl;

return 0;

}1234567891011121314151617

　　使用函数对象创建线程：

#include <iostream>

#include <thread>

class DisplayThread {

public:

void operator ()() {

for (int i = 0; i < 100; i++)

std::cout << "Display Thread Excecuting" << std::endl;

}

};

int main() {

std::thread threadObj((DisplayThread()));

for (int i = 0; i < 100; i++)

std::cout << "Display From Main Thread " << std::endl;

std::cout << "Waiting For Thread to complete" << std::endl;

threadObj.join();

std::cout << "Exiting from Main Thread" << std::endl;

return 0;

}123456789101112131415161718192021

CmakeLists.txt

cmake\_minimum\_required(VERSION 3.10)

project(Thread\_test)

set(CMAKE\_CXX\_STANDARD 11)

find\_package(Threads REQUIRED)

add\_executable(Thread\_test main.cpp)

target\_link\_libraries(Thread\_test ${CMAKE\_THREAD\_LIBS\_INIT})123456

　　每个std::thread对象都有一个相关联的id，std::thread::get\_id() —-成员函数中给出对应线程对象的id;

std::this\_thread::get\_id()—-给出当前线程的id，如果std::thread对象没有关联的线程，get\_id()将返回默认构造的std::thread::id对象：“not any thread”，std::thread::id也可以表示id。

2.joining和detaching 线程

　　启动了线程，你需要明确是要等待线程结束(加入式)，还是让其自主运行(分离式)，一个是通过调用std::thread对象上调用join()函数等待这个线程执行完毕:

std::thread threadObj(funcPtr);

threadObj.join();12

例如，主线程启动10个线程，启动完毕后，main函数等待他们执行完毕，join完所有线程后，main函数继续执行：

#include <iostream>

#include <thread>

#include <algorithm>

class WorkerThread

{

public:

void operator()(){

std::cout<<"Worker Thread "<<std::this\_thread::get\_id()<<"is Excecuting"<<std::endl;

}

};

int main(){

std::vector<std::thread> threadList;

for(int i = 0; i < 10; i++){

threadList.push\_back(std::thread(WorkerThread()));

}

// Now wait for all the worker thread to finish i.e.

// Call join() function on each of the std::thread object

std::cout<<"Wait for all the worker thread to finish"<<std::endl;

std::for\_each(threadList.begin(), threadList.end(), std::mem\_fn(&std::thread::join));

std::cout<<"Exiting from Main Thread"<<std::endl;

return 0;

} 12345678910111213141516171819202122232425

　　另一个是detach可以将线程与线程对象分离,让线程作为后台线程执行,当前线程也不会阻塞了.但是detach之后就无法在和线程发生联系了.如果线程执行函数使用了临时变量可能会出现问题,线程调用了detach在后台运行,临时变量可能已经销毁,那么线程会访问已经被销毁的变量，需要在std::thread对象中调用std::detach()函数:

std::thread threadObj(funcPtr)

threadObj.detach();12

　　调用detach()后，std::thread对象不再与实际执行线程相关联，在线程句柄上调用detach() 和 join()要小心.

3.将参数传递给线程

　　要将参数传递给线程的可关联对象或函数，只需将参数传递给std::thread构造函数，默认情况下，所有的参数都将复制到新线程的内部存储中。

　　给线程传递参数：

#include <iostream>

#include <string>

#include <thread>

void threadCallback(int x, std::string str) {

std::cout << "Passed Number = " << x << std::endl;

std::cout << "Passed String = " << str << std::endl;

}

int main() {

int x = 10;

std::string str = "Sample String";

std::thread threadObj(threadCallback, x, str);

threadObj.join();

return 0;

}123456789101112131415

　　给线程传递引用：

#include <iostream>

#include <thread>

void threadCallback(int const& x) {

int& y = const\_cast<int&>(x);

y++;

std::cout << "Inside Thread x = " << x << std::endl;

}

int main() {

int x = 9;

std::cout << "In Main Thread : Before Thread Start x = " << x << std::endl;

std::thread threadObj(threadCallback, x);

threadObj.join();

std::cout << "In Main Thread : After Thread Joins x = " << x << std::endl;

return 0;

} 1234567891011121314151617

输出结果为：

In Main Thread : Before Thread Start x = 9

Inside Thread x = 10

In Main Thread : After Thread Joins x = 9

Process finished with exit code 0

　　即使threadCallback接受参数作为引用，但是并没有改变main中x的值，在线程引用外它是不可见的。这是因为线程函数threadCallback中的x是引用复制在新线程的堆栈中的临时值，使用std::ref可进行修改：

#include <iostream>

#include <thread>

void threadCallback(int const& x) {

int& y = const\_cast<int&>(x);

y++;

std::cout << "Inside Thread x = " << x << std::endl;

}

int main() {

int x = 9;

std::cout << "In Main Thread : Before Thread Start x = " << x << std::endl;

std::thread threadObj(threadCallback, std::ref(x));

threadObj.join();

std::cout << "In Main Thread : After Thread Joins x = " << x << std::endl;

return 0;

}1234567891011121314151617

输出结果为：

In Main Thread : Before Thread Start x = 9

Inside Thread x = 10

In Main Thread : After Thread Joins x = 10

Process finished with exit code 0

　　指定一个类的成员函数的指针作为线程函数，将指针传递给成员函数作为回调函数，并将指针指向对象作为第二个参数：

#include <iostream>

#include <thread>

class DummyClass {

public:

DummyClass() { }

DummyClass(const DummyClass& obj) { }

void sampleMemberfunction(int x) {

std::cout << "Inside sampleMemberfunction " << x << std::endl;

}

};

int main() {

DummyClass dummyObj;

int x = 10;

std::thread threadObj(&DummyClass::sampleMemberfunction, &dummyObj, x);

threadObj.join();

return 0;

}1234567891011121314151617181920

4.线程间数据的共享与竞争条件

　　在多线程间的数据共享很简单，但是在程序中的这种数据共享可能会引起问题，其中一种便是竞争条件。当两个或多个线程并行执行一组操作，访问相同的内存位置，此时，它们中的一个或多个线程会修改内存位置中的数据，这可能会导致一些意外的结果，这就是竞争条件。竞争条件通常较难发现并重现，因为它们并不总是出现，只有当两个或多个线程执行操作的相对顺序导致意外结果时，它们才会发生。

　　例如创建5个线程，这些线程共享类Wallet的一个对象，使用addMoney()成员函数并行添加100元。所以，如果最初钱包中的钱是0，那么在所有线程的竞争执行完毕后，钱包中的钱应该是500，但是，由于所有线程同时修改共享数据，在某些情况下，钱包中的钱可能远小于500。

测试如下：

#include <iostream>

#include <thread>

#include <algorithm>

class Wallet {

int mMoney;

public:

Wallet() : mMoney(0) { }

int getMoney() { return mMoney; }

void addMoney(int money) {

for (int i = 0; i < money; i++) {

mMoney++;

}

}

};

int testMultithreadWallet() {

Wallet walletObject;

std::vector<std::thread> threads;

for (int i = 0; i < 5; i++) {

threads.push\_back(std::thread(&Wallet::addMoney, &walletObject, 100));

}

for (int i = 0; i < 5; i++) {

threads.at(i).join();

}

return walletObject.getMoney();

}

int main() {

int val = 0;

for (int k = 0; k < 100; k++) {

if ((val=testMultithreadWallet()) != 500) {

std::cout << "Error at count = " << k << " Money in Wallet = " << val << std::endl;

}

}

return 0;

}12345678910111213141516171819202122232425262728293031323334353637383940

每个线程并行地增加相同的成员变量“mMoney”，看似是一条线，但是这个“nMoney++”实际上被转换为3条机器命令：

·在Register中加载”mMoney”变量

·增加register的值

·用register的值更新“mMoney”变量

在这种情况下，一个增量将被忽略，因为不是增加mMoney变量，而是增加不同的寄存器，“mMoney”变量的值被覆盖。

5.使用mutex处理竞争条件

　　为了处理多线程环境中的竞争条件，我们需要mutex互斥锁，在修改或读取共享数据前，需要对数据加锁，修改完成后，对数据进行解锁。在c++11的线程库中，mutexes在< mutexe >头文件中，表示互斥体的类是std::mutex。

　　就上面的问题进行处理，Wallet类提供了在Wallet中增加money的方法，并且在不同的线程中使用相同的Wallet对象，所以我们需要对Wallet的addMoney()方法加锁。在增加Wallet中的money前加锁，并且在离开该函数前解锁，看代码：Wallet类内部维护money，并提供函数addMoney()，这个成员函数首先获取一个锁，然后给wallet对象的money增加指定的数额，最后释放锁。

#include <iostream>

#include <thread>

#include <vector>

#include <mutex>

class Wallet {

int mMoney;

std::mutex mutex;

public:

Wallet() : mMoney(0) { }

int getMoney() { return mMoney;}

void addMoney(int money) {

mutex.lock();

for (int i = 0; i < money; i++) {

mMoney++;

}

mutex.unlock();

}

};

int testMultithreadWallet() {

Wallet walletObject;

std::vector<std::thread> threads;

for (int i = 0; i < 5; ++i) {

threads.push\_back(std::thread(&Wallet::addMoney, &walletObject, 1000));

}

for (int i = 0; i < threads.size(); i++) {

threads.at(i).join();

}

return walletObject.getMoney();

}

int main() {

int val = 0;

for (int k = 0; k < 1000; k++) {

if ((val = testMultithreadWallet()) != 5000) {

std::cout << "Error at count= " << k << " money in wallet" << val << std::endl;

}

}

return 0;

}1234567891011121314151617181920212223242526272829303132333435363738394041424344

这种情况保证了钱包里的钱不会出现少于5000的情况，因为addMoney()中的互斥锁确保了只有在一个线程修改完成money后，另一个线程才能对其进行修改，但是，如果我们忘记在函数结束后对锁进行释放会怎么样？这种情况下，一个线程将退出而不释放锁，其他线程将保持等待，为了避免这种情况，我们应当使用std::lock\_guard，这是一个template class，它为mutex实现RALL，它将mutex包裹在其对象内，并将附加的mutex锁定在其构造函数中，当其析构函数被调用时，它将释放互斥体。

class Wallet {

int mMoney;

std::mutex mutex;

public:

Wallet() : mMoney(0) { }

int getMoney() { return mMoney;}

void addMoney(int money) {

std::lock\_guard<std::mutex> lockGuard(mutex);

for (int i = 0; i < mMoney; ++i) {

//如果在此处发生异常，lockGuadr的析构函数将会因为堆栈展开而被调用

mMoney++;

//一旦函数退出，那么lockGuard对象的析构函数将被调用，在析构函数中mutex会被释放

}

}

};

123456789101112131415161718

6.条件变量

　　条件变量是一种用于在2个线程之间进行信令的事件，一个线程可以等待它得到信号，其他的线程可以给它发信号。在c++11中，条件变量需要头文件< condition\_variable>，同时，条件变量还需要一个mutex锁。

　　条件变量是如何运行的：

　　·线程1调用等待条件变量，内部获取mutex互斥锁并检查是否满足条件；

　　·如果没有，则释放锁，并等待条件变量得到发出的信号(线程被阻塞)，条件变量的wait()函数以原子方式提供这两个操作；

　　·另一个线程，如线程2，当满足条件时，向条件变量发信号；

　　·一旦线程1正等待其恢复的条件变量发出信号，线程1便获取互斥锁，并检查与条件变量相关关联的条件是否满足，或者是否是一个上级调用，如果多个线程正在等待，那么notify\_one将只解锁一个线程；

　　·如果是一个上级调用，那么它再次调用wait()函数。

　　条件变量的主要成员函数：

Wait()

它使得当前线程阻塞，直到条件变量得到信号或发生虚假唤醒；

它原子性地释放附加的mutex，阻塞当前线程，并将其添加到等待当前条件变量对象的线程列表中，当某线程在同样的条件变量上调用notify\_one() 或者 notify\_all()，线程将被解除阻塞；

这种行为也可能是虚假的，因此，解除阻塞后，需要再次检查条件；

一个回调函数会传给该函数，调用它来检查其是否是虚假调用，还是确实满足了真实条件；

当线程解除阻塞后，wait()函数获取mutex锁，并检查条件是否满足，如果条件不满足，则再次原子性地释放附加的mutex，阻塞当前线程，并将其添加到等待当前条件变量对象的线程列表中。

notify\_one()

如果所有线程都在等待相同的条件变量对象，那么notify\_one会取消阻塞其中一个等待线程。

notify\_all()

如果所有线程都在等待相同的条件变量对象，那么notify\_all会取消阻塞所有的等待线程。

#include <iostream>

#include <thread>

#include <functional>

#include <mutex>

#include <condition\_variable>

using namespace std::placeholders;

class Application {

std::mutex m\_mutex;

std::condition\_variable m\_condVar;

bool m\_bDataLoaded;

public:

Application() {

m\_bDataLoaded = false;

}

void loadData() {

//使该线程sleep 1秒

std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));

std::cout << "Loading Data from XML" << std::endl;

//锁定数据

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

//flag设为true，表明数据已加载

m\_bDataLoaded = true;

//通知条件变量

m\_condVar.notify\_one();

}

bool isDataLoaded() {

return m\_bDataLoaded;

}

void mainTask() {

std::cout << "Do some handshaking" << std::endl;

//获取锁

std::unique\_lock<std::mutex> mlock(m\_mutex);

//开始等待条件变量得到信号

//wait()将在内部释放锁，并使线程阻塞

//一旦条件变量发出信号，则恢复线程并再次获取锁

//然后检测条件是否满足，如果条件满足，则继续，否则再次进入wait

m\_condVar.wait(mlock, std::bind(&Application::isDataLoaded, this));

std::cout << "Do Processing On loaded Data" << std::endl;

}

};

int main() {

Application app;

std::thread thread\_1(&Application::mainTask, &app);

std::thread thread\_2(&Application::loadData, &app);

thread\_2.join();

thread\_1.join();

return 0;

}

C++线程同步的四种方式

临界区：

CRITICAL\_SECTION Critical; //定义临界区句柄

InitializeCriticalSection(&Critical); //初始化临界区对象

EnterCriticalSection(&Critical);

LeaveCriticalSection(&Critical);

事件：

HANDLE hEvent; //定义事件句柄

hEvent = CreateEvent(NULL, FALSE, TRUE, "event");

WaitForSingleObject(hEvent, INFINITE); //等待对象为有信号状态

SetEvent(hEvent);

信号量：

HANDLE hSemaphore; //定义信号量句柄

hSemaphore = CreateSemaphore(NULL, 1, 100, "sema");

WaitForSingleObject(hSemaphore, INFINITE); //等待信号量为有信号状态

ReleaseSemaphore(hSemaphore, 1, &count);

互斥量：

HANDLE hMutex; //定义互斥对象句柄

hMutex = CreateMutex(NULL, false, "mutex"); //创建互斥对象

WaitForSingleObject(hMutex, INFINITE);

ReleaseMutex(hMutex);