程序设计实习

郭炜　微博 http://weibo.com/guoweiofpku
http://blog.sina.com.cn/u/3266490431

多线程程序设计

多线程

- 让DEV C++ 支持thread:

1) 下载最新 MinGW64并安装到 Dev C++的文件夹下。文件夹结构如下：

```
Dev-cpp
        MinGW64
                bin
                i686-w64-mingw32
                include
                lib
                ....
```

2) 在MinGW64文件夹下找到 libwinpthread-1.dll，将其和编译出来的exe文件放在同一个文件夹下面

3) 运行 exe 文件

多线程概念

```
#include <iostream>
#include <thread>
using namespace std;
struct MyThread {
	void operator () (int i) {
		while(true)
			cout << i << " IN MYTHREAD\n";
	}
};
void my_thread(int x)
{
	while(x)
		cout << "in my_thread\n";
}
```

多线程概念

```
int main()
{

        MyThread x;   // 对x 的要求：可复制
        thread th(x,200);    // 创建线程并执行
        thread th1(my_thread, 100); //100是参数
        while(true)
                cout << "in main\n";
        return 0;
}
```

输出：
in my_thread
in my_thread
200 IN MYTHREAD
200 IN MYTHREAD
in main
in my_thread
200 IN MYTHREAD
200 IN MYTHREAD
in main
in main
in my_thread
200 IN MYTHREAD
……

多线程概念

- thread::join 等待线程结束

```
int main()
{

        MyThread x;   // 对x 的要求：可复制
        thread th(x,200);    // 创建线程并执行
        thread th1(my_thread, 100);
       th.join(); //主线程等待th结束后才会继续
        while(true)
               cout << "in main\n";
        return 0;
}
```

输出：
in my_thread
in my_thread
200 IN MYTHREAD
200 IN MYTHREAD
in my_thread
200 IN MYTHREAD
200 IN MYTHREAD
in my_thread
200 IN MYTHREAD
……

多线程概念

- detach 解除对线程的控制

```
void func()
{
        while(1);
}
int main()
{
    std::thread t(func);
    t.detach();       //有此行程序才可正常结束。此后不可 t.join()。
    cout << t.joinable() << endl; // 0
    return 0;
}
```

主线程结束时整个程序就结束，哪怕子线程还没结束。thread对象如果没有join或detach，则其析构时导致异常。

线程参数传递

```
void func(int i, double d, const std::string& s)
{
    while(true)
       cout << i << ", " << d << ", " << s
       << endl;
}
int main()
{
       thread t(func, 1, 12.50, "sample");
       t.join();
       return 0;
}
```

线程异常捕获

不能用以下方法捕获 t1,t2运行时的异常：

```
try
{
    std::thread t1(func);
    std::thread t2(func);

    t1.join();
    t2.join();
}
catch(const std::exception& ex)
{
    std::cout << ex.what() << std::endl;
}
```

线程异常捕获

```
#include <iostream>
#include <thread>
#include <vector>
#include <exception>
using namespace std;
vector<exception_ptr>  g_exceptions;
void func()
{
    try
    {
        throw exception ();
    }
    catch(...)
    {
       g_exceptions.push_back(current_exception());
    }
}
```

线程异常捕获

```
int main()
{
    thread t(func);
    t.join();
    for(auto& e : g_exceptions)     {
        try         {
            if(e != nullptr)
            {
                rethrow_exception(e);
            }
        }
        catch(const exception& e)         {
            cout << e.what() << endl;
        }
    }
    return 0;
}   // std::exception
```

this_thread名字空间

➢thread的一些常用函数：位于std::this_thread命名空间中

get_id: 返回当前线程的id.

yield: 放弃当前时间片，让调度器先运行其他可用的线程。

sleep_for:阻塞当前线程，时间不少于其参数指定的时间。

sleep_util:在参数指定的时间到达之前，使当前线程一直处于阻塞状态。

线程转移

➢thread对象不可复制，可移动，可作为函数返回值，可放入容器

```
#include <iostream>
#include <thread>
using namespace std;
int main()
{
    thread t([]() { cout << "ok" << endl; });
    thread t2 = move(t); //线程不可复制，可移动
    thread t3;
    t3 = move(t2);       //线程不可赋值，可移动
    t3.join();           //转移后 t 和 t2 不可 join
    cout << "main thread\n";
    return 0;
}
```

线程转移

➢线程可以作为函数返回值

```
std::thread f() {
    return std::thread( [](->void { cout << "f1" << endl; });
}
std::thread g() {
    std::thread t( [] (int x)->int { cout << x << endl;
                                     return 0;
                                     },100);
    return t;
}
int main()
{
        thread t1 = g();
        thread t2 = f();
        t1.join();
        t2.join();
        return 0;
}
```

线程同步问题

```
#include <iostream>
#include <thread>
using namespace std;
int balance;
void withDraw(int x)
{
        if( balance >= x)
                balance -= x;
}
int main()
{
   balance = 100;
   thread t1(withDraw,50);
   thread t2(withDraw,90);
   t1.join();
   t2.join();
   cout << "balance remain:" << balance << endl;
   return 0;
}
```

线程同步问题

```
#include <iostream>
#include <thread>
using namespace std;
int balance;
void withDraw(int x)
{
        if( balance >= x)
                balance -= x;
}
int main()
{
   balance = 100;
   thread t1(withDraw,50);
   thread t2(withDraw,90);
   t1.join();
   t2.join();
   cout << "balance remain:" << balance << endl;
   return 0;
}
```

输出：

50

或

10

或

-40 (不同步)

线程同步问题

```
#include <iostream>
#include <thread>
using namespace std;
int balance;
void withDraw(int x)  {
        if( balance >= x) {
                for(int i = 0;i < 10000; ++i); //增大不同步概率
                balance -= x;
        }
}
int main() {
   balance = 100;
   thread t1(withDraw,50);
   thread t2(withDraw,90);
   t1.join();
   t2.join();
   cout << "balance remain:" << balance << endl;
   return 0;
}
```

输出：

50

或

10

或

-40

线程同步问题

```
int balance;
void withDraw(int x,string name) {
    if( balance >= x) {
            balance -= x;
            cout << name << " withdraw " << x << " done" << endl;
    }
    else cout << name << " failed to withdraw" << endl;
}
int main() {
   balance = 100;
   thread t1(withDraw,50,"Mike");
   thread t2(withDraw,90,"Jane");
   t1.join();
   t2.join();
   cout << "balance remain:" << balance << endl;
   return 0;
}
```

线程同步问题

编译为t2.exe，输出结果：

```
C:\tmp>t2
Mike withdraw 50 done
Jane failed to withdraw
balance remain:50

C:\tmp>t2
MikeJane failed to withdraw withdraw
50 done
balance remain:50

C:\tmp>t2
Mike withdraw Jane failed to withdraw50 done

balance remain:50
```

用互斥量mutex加锁避免同步问题

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;
mutex myLock ;
int balance;
void withDraw(int x,string name)
{
        myLock.lock(); //若myLock已经被锁住，则操作系统将线程挂起等待
        if( balance >= x) {
                for(int i = 0;i < 10000; ++i);
                balance -= x;
                cout << name << " withdraw " << x << " done" << endl;
        }
        else
                cout << name << " failed to withdraw" << endl;
        myLock.unlock(); //若有线程在等待myLock,则操作系统将其唤醒
}
```

用互斥量mutex加锁避免同步问题

编译为t2.exe，输出结果：

```
C:\tmp>t2
Mike withdraw 50 done
Jane failed to withdraw
balance remain:50

C:\tmp>t2
Jane withdraw 90 done
Mike failed to withdraw
balance remain:10
```

注意： metux是不可复制的

几种互斥量

std::mutex

std::recursive_mutex

std::timed_mutex

std::recursive_timed_mutex

std::timed_mutex， **std::recursive_timed_mutex**提供 **try_lock_for/ try_lock_until**方法，允许你等待一个lock，直到超时，或者达到定义的时间。

try_lock

```
mutex mtx;
void func1()
{
        for(int i = 0;i < 1000000; ++i);
        mtx.lock();
        cout << "locked by func1" << endl;
        mtx.unlock();
}
void func2()
{
        if( mtx.try_lock()) { //能加锁就加锁，加不了就继续，不挂起
                cout << "locked by func2" << endl;
                for(int i = 0;i < 100000; ++i);
                mtx.unlock();
        }
        else
                cout << "not locked by func2" <<endl;
}
```

try_lock

```
int main()
{
        thread th1(func1);
        thread th2(func2);
        th1.join();
        th2.join();
}
```

可能输出：

```
C:\tmp>t2
locked by func1not locked by func2

C:\tmp>t2
locked by func2
locked by func1

C:\tmp>t2
locked by func1
not locked by func2
```

线程休眠

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>
using namespace std;
mutex g_lock;

void func()
{
    g_lock.lock();

    cout << "entered thread " << this_thread::get_id() << endl;
    this_thread::sleep_for(chrono::seconds(rand() % 10));
    cout << "leaving thread " << this_thread::get_id() << endl;

    g_lock.unlock();
}
```

线程休眠

```
int main()
{
    srand((unsigned int)time(0));

    thread t1(func);
    thread t2(func);
    thread t3(func);

    t1.join();
    t2.join();
    t3.join();

    return 0;
}
```

```
C:\tmp>t2
entered thread 3
leaving thread 3
entered thread 2
leaving thread 2
entered thread 4
leaving thread 4
```

线程安全的容器

```
template <typename T>
class Vector  {
    mutex _lock;
    vector<T> _elements;
public:
    void add(T element)      {
        _lock.lock();
        _elements.push_back(element);
        _lock.unlock();
    }
    void dump()     {
        _lock.lock();
        for(auto e : _elements)
            cout << e << endl;
        _lock.unlock();
    }
};
```

线程安全的容器

```
void func(Vector<int>& cont,int start,int end)
{
        for(int i = start;i < end; ++i)
                cont.add(i);
}
int main()
{
    Vector<int> cont;
    thread t1(func, ref(cont),1,5000); //线程参数传引用必须用 ref
    thread t2(func, ref(cont),20,2500);
    thread t3(func, ref(cont),30,35000);
    t1.join();
    t2.join();
    t3.join();
    cont.dump();
    return 0;
}   //不加锁则有时运行会产生崩溃错误
```

recursive_mutex

➢同一线程lock两次mutex导致死锁

```
template <typename T>
class Vector {
    mutex   _lock;      vector<T> _elements;
public:
    void add(T element)      {
        _lock.lock();
        _elements.push_back(element);
        _lock.unlock();
    }
    void addArray(int num, int * a)     { //此函数被调用会导致线程死锁
        for (int i = 0; i < num; i++)          {
            _lock.lock();
            add(a[i]);
            _lock.unlock();
        }
    }
};
```

recursive_mutex

➢同一线程lock两次mutex导致死锁，用recursive_mutex替代mutex

```
template <typename T>
class Vector {
    recursive_mutex   _lock;      vector<T> _elements;
public:
    void add(T element)      {
        _lock.lock();
        _elements.push_back(element);
        _lock.unlock();
    }
    void addArray(int num, int * a)     {
        for (int i = 0; i < num; i++)         {
            _lock.lock();
            add(a[i]);
            _lock.unlock();
        }
    }
};
```

recursive_mutex 被同一线程lock的最大次数没有标准。如果达到最大次数，会抛出std::system_error异常

用lock_guard管理mutex

- lock_guard 在构造时申请mutex并加锁，析构时自动解锁其管理的mutex
- lock_guard 不可复制，不可移动

```
template <typename T>
class Vector {
    recursive_mutex   _lock;
    vector<T> _elements;
public:
    void add(T element)      {
        lock_guard<recursive_mutex> locker(_lock);
        _elements.push_back(element);
}
    void addArray(int num, int * a)     {
        for (int i = 0; i < num; i++)         {
            lock_guard<recursive_mutex> locker(_lock);
            add(a[i]);
        }
    }
};
```

用lock_guard管理mutex

- 将mutex声明为mutable

```
class Vector {
        mutable recursive_mutex   _lock;
        vector<T> _elements;
public:

    void dump() const
    {
        lock_guard<recursive_mutex> locker(_lock);
        //_lock 非mutable则编译不过
        for(auto e : _elements)
            cout << e << endl;
    }
};
```

用unique_lock管理mutex

- 其管理的 mutex可以是加锁的，也可以是未加锁的。不一定要在构造函数中上锁，可以用成员函数lock上锁

- unique_lock 析构函数自动解锁，也有成员函数unlock用于解锁

- unique_lock 不可复制但可移动，因此可以作为函数的返回值。因此其托管的mutex可以转移到 unique_lock 对象生存期以外由别的 unique_lock继续托管

```
mutex mtx;
unique_lock<mutex> get_lock(){
    unique_lock<mutex> lck(mtx);
    cout<<"preparing data..."<<endl;
    return lck; //移动构造
}
int main(){
    unique_lock<mutex> lck(get_lock());
    cout<<"processing data..."<<endl;
    return 0;    }
```

用unique_lock管理mutex

➢unique_guard的成员函数

lock locks the associated mutex

try_lock tries to lock the associated mutex, returns if the mutex is not available

try_lock_for attempts to lock the associated TimedLockable mutex, returns if the mutex has been unavailable for the specified time duration

try_lock_until tries to lock the associated TimedLockable mutex, returns if the mutex has been unavailable until specified time point has been reached

unlock unlocks the associated mutex

http://www.tuicool.com/articles/6F7zUf

死锁

- 多个线程申请多个mutex可能导致死锁

```
#include <mutex>
#include <thread>
#include <chrono>
#include <iostream>
#include <string>
using namespace std;
struct Account
{
  Account(string n, int b):name(n),balance(b)
  {
    name = n;
    balance = b;
  }
  string name;
  int balance;
  mutex mtx;
};
```

死锁

```
void transfer( Account &from, Account &to, int amount )
{
  if ( & from == & to )
        return;
  from.mtx.lock();
  this_thread::sleep_for(std::chrono::seconds(1));
  to.mtx.lock();
  from.balance -= amount;
  to.balance += amount;
  cout << "Transfer " << amount << " from "<< from.name << " to "
       << to.name << endl;
  from.mtx.unlock();
  to.mtx.unlock();
}
```

死锁

```
int main()
{
  Account Account1( "Mike",100  );
  Account Account2( "Jack", 50 );
  thread t1( [&](){ transfer( Account1, Account2, 10 ); } );
  thread t2( [&](){ transfer( Account2, Account1, 5 ); } );
  t1.join();
  t2.join();
  return 0;
}
```

很容易导致死锁！

Mike 拿到一把锁，然后Jack拿到另一把锁，于是双方都在等待对方不会释放的锁，导致死锁

用同时等待多个资源的办法避免死锁

```
void transfer( Account &from, Account &to, int amount )
{
  if ( & from == & to )
        return;
  lock( from.mtx,to.mtx ); //必须两个锁都能锁住才会继续，否则blocked
  this_thread::sleep_for(std::chrono::seconds(1));
  from.balance -= amount;
  to.balance += amount;
  from.mtx.unlock();
  to.mtx.unlock();
  cout << "Transfer " << amount << " from "<< from.name << " to "
       << to.name << endl;
}
```

用同时等待多个资源的办法避免死锁

```
void transfer( Account &from, Account &to, int amount )
{
  if ( & from == & to )
        return;
  lock( from.mtx,to.mtx ); //必须两个锁都能锁住才会继续，否则blocked
  this_thread::sleep_for(std::chrono::seconds(1));
  from.balance -= amount;
  to.balance += amount;
  from.mtx.unlock();
  to.mtx.unlock();
  cout << "Transfer " << amount << " from "<< from.name << " to "
       << to.name << endl;
}
```

还可以用将共享资源编号，每个线程都按序号从小到大申请资源的办法避免死锁

用同时等待多个资源的办法避免死锁

➢用lock_guard的实现：

```
void transfer( Account &from, Account &to, int amount )
{
  if ( & from == & to )
        return;
  lock( from.mtx, to.mtx );
  lock_guard<mutex> lock1( from.mtx, adopt_lock );
//adopt_lock表示from.mtx是已经上锁的
  lock_guard<mutex> lock2( to.mtx, adopt_lock );
  this_thread::sleep_for(std::chrono::seconds(1));
  from.balance -= amount;
  to.balance += amount;
  cout << "Transfer " << amount << " from "<< from.name << " to " <<
to.name << endl;
}
```

用同时等待多个资源的办法避免死锁

➢用unique_lock的实现：

```
void transfer( Account &from, Account &to, int amount )
{
  if ( & from == & to )
        return;
  unique_lock<mutex> lock1( from.mtx, defer_lock );
        //defer_lock表示构造时不加锁
  unique_lock<mutex> lock2( to.mtx, defer_lock );
  lock( lock1,lock2);//必须两个锁都能锁住才会继续，否则blocked
  this_thread::sleep_for(std::chrono::seconds(1));
  from.balance -= amount;
  to.balance += amount;
  cout << "Transfer " << amount << " from "<< from.name << " to "
  << to.name << endl;
}
```

条件变量 condition_variable

- 可以有一个或多个线程在等待一个条件变量，
  直到别的线程通过这个条件变量发送消息唤醒等待该变量的一个或全部线程

```
#include <chrono>
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
using namespace std;
mutex mtx;
condition_variable condv;
bool notified = false;
void func1(int i)
{
        unique_lock<mutex> lk(mtx); //条件变量要和mutex一起用
        while(!notified)  {  //必须用while判断，避免“假唤醒”
                cout << "thread " << i <<  " waiting...." << endl;
                condv.wait(lk);//wait立即导致被阻塞，解锁lk。返回时重新加锁
        }
        cout << "thread " << i << " after notified" << endl;
}
```

条件变量 condition_variable

```
int main()
{
    thread threads[3];
    for(int i = 0;i < 3; ++i)
        threads[i] = thread(func1,i); //thread可移动
    this_thread::sleep_for(chrono::seconds(3));
    {
        unique_lock<mutex> lk(mtx);
        notified = true;
        cout << "set true" << endl;
        condv.notify_all(); //唤醒所有等待condv的线程
        //要放到花括号里，确保 lk析构解锁，否则 无法唤醒 waiting的thread
        // condv.notify_one() 则只随机唤醒一个线程
    }
    for (auto & t:threads)
        t.join();
    return 0;
}
```

```
thread 0 waiting....
thread 1 waiting....
thread 2 waiting....
set true
thread 2 after notified
thread 1 after notified
thread 0 after notified
```

条件变量 condition_variable

➢**condition_variable的等待成员函数**

wait(mtx,pred)　　**pred为函数名或函数对象**

可用来避免假唤醒。

当 **pred()** 条件为**false** 时才会阻塞当前线程，并且在收到通知后只有当 **pred()** 为 **true** 时才会解除阻塞。

wait_for(mtx,time)　　只会阻塞一段时间

wait_until (mtx,absolute_time)　　只会阻塞到某个绝对时间点

用mutex和条件变量实现信号量

➢资源有n个，每个线程每次只需要使用1个（不需要同时占有多个），如何同步？

```
class Semaphore {
public:
  Semaphore(long cnt = 0): count(cnt) { }
  void Signal() {//释放资源
    unique_lock<mutex> lock(mtx);
    ++count;
    cv.notify_one();
  }
  void Wait() {//申请资源
    unique_lock<mutex> lock(mtx);
    cv.wait(lock, [=] { return count > 0; });
    --count;
  }
private:
  mutex mtx;     condition_variable cv;
  long count    ;  //资源总数
};
```

用mutex和条件变量实现信号量

```
Semaphore g_semaphore(2);
mutex g_io_mutex;

void Worker() {
  g_semaphore.Wait();
  thread::id thread_id = this_thread::get_id();
  {
    lock_guard<mutex> lock(g_io_mutex);
    std::cout << "Thread " << thread_id
              << ": wait succeeded" << std::endl;
  }
  std::this_thread::sleep_for(chrono::seconds(3));
  g_semaphore.Signal();
}
```

用mutex和条件变量实现信号量

```
int main() {
  thread threads[3];
  for (int i = 0; i < 3; ++i) {
       threads[i] = thread(Worker);

  }
  for( auto & t :threads)
       t.join();
  return 0;
}
```

原子操作

```
#include <iostream>
#include <thread>
using namespace std;
int val = 0;
void inc (int n) {
    for (int i = 0; i < n; ++i)
       ++ val;
}
int main (int argc, char* argv []) {
    thread t2 (inc,2000000);
    thread t1 (inc,1000000);

    t1.join ();
    t2.join ();
    cout << val << endl;
    return 0;
}
```

原子操作

```
#include <iostream>
#include <thread>
using namespace std; int val = 0;
void inc (int n) {
    for (int i = 0; i < n; ++i)
       ++ val;
}
int main (int argc, char* argv []) {
    thread t2 (inc,2000000);
    thread t1 (inc,1000000);

    t1.join ();
    t2.join ();
    cout << val << endl;
    return 0;
}
```

结果不一定是300000!

原子操作

```
#include <iostream>
#include <thread>
#include <atomic>
using namespace std;
atomic_int val = { 0 }; //整型数据的原子，对其访问自动加锁解锁，比用mutex快数倍！
//atomic<int> val = {0};  也可以
void inc (int n) {
    for (int i = 0; i < n; ++i)
        ++ val;
}
int main (int argc, char* argv []) {
    thread t2 (inc,2000000);
    thread t1 (inc,1000000);

    t1.join ();
    t2.join ();
    cout << val << endl;
    return 0;
}
```

结果一定是300000!

线程局部存储

- 每个线程都可以有自己的局部变量

```
int j = 0;
mutex m;
void foo() {
  m.lock();
  j++;
  cout << j << endl;
  m.unlock();
}
int main(){
        thread t1(foo);
        thread t2(foo);
        t1.join();
        t2.join();
        cout << j << endl;
        return 0;
}
```

输出：

1

2

2

线程局部存储

- 每个线程都可以有自己的局部变量

```
thread_local int j = 0;
mutex m;
void foo() {
  m.lock();
  j++;
  cout << j << endl;
  m.unlock();
}
int main(){
        thread t1(foo);
        thread t2(foo);
        t1.join();
        t2.join();
        cout << j << endl;
        return 0;
}
```

输出：

1

1

0

多线程

- 参考：

https://www.codeproject.com/articles/598695/cplusplus11-threads-locks-and-condition-variables

Qt事件

```
QEvent
        QDropEvent
                QDragMoveEvent
                        QDragEnterEvent
        QCloseEvent
        QPaintEvent
        QInputEvent
                QContextMenuEvent
                QKeyEvent
                QMouseEvent
                QWheelEvent
        QResizeEvent
        QTimerEvent
```