C++ 的多執行序程式開發 Thread：基本使用(二)

上課老師：莊啟宏

執行緒的同步(unique_lock的使用)

- unique_lock 中的 unique 表示獨占所有權。
- unique_lock 獨占的是 mutex 對象，就是對 mutex 鎖的獨占。
- 用法：
  - （1）新建一個 unique_lock 對象 > std::mutex mymutex;
  - （2）給對象傳入一個 std::mutex 對象作為參數 > unique_lock lock(mymutex);

```cpp
#include <iostream>
#include<thread>
#include<unistd.h>
#include<mutex>
using namespace std;
std::mutex mymutex;
void sayHello()
{
    int k=0;
    unique_lock<mutex> lock(mymutex);
    while(k<2)
    {
        k++;
        cout<<endl<<"hello"<<endl;
        sleep(2);
    }
}
void sayWorld()
{
    unique_lock<mutex> lock(mymutex);
    while(1)
    {
         cout<<endl<<"world"<<endl;
         sleep(1);
    }
}
int main()
{
   thread threadHello(&sayHello);
   thread threadWorld(&sayWorld);
   threadHello.join();
   threadWorld.join();
   return 0;
}
```

- 程序執行步驟： 首先同時運行 threadHello 執行緒和 threadWorld執行緒，先進入 threadHello 執行緒的 sayHello()函數，這個時候加了mymutex 鎖，另外一个 threadWorld 執行緒進入後發現 mymutex 鎖沒有釋放，只能等待。
- 當經過兩個循環（每個循環 2 秒後）threadHello執行緒結束，unique_lock lock(mymutex) 的生命周期結束，mymutex 鎖釋放，執行 threadWorld 執行緒，此時開始 sayworld()。

同時鎖定多個 mutex

- 雖然一般狀況下，大多是一次去鎖定一個 mutex 來使用的。但是在實際使用的時候，有的時候還是會需要同時去鎖定多個 mutex，來避免資料不同步的問題。例如，在 cppreference 就有提出一個銀行帳號例子，在這種狀況，就必須要同時鎖定「轉出者」和「轉入者」兩者個 mutex，否則會有問題。

```cpp
#include <mutex>
#include <thread>
#include <chrono>
#include <iostream>
#include <string>

using namespace std;

struct bank_account
{
  explicit bank_account(string name, int money)
  {
    sName = name;
    iMoney = money;
  }

  string sName;
  int iMoney;
  mutex mMutex;
};
```

```
void transfer( bank_account &from, bank_account &to, int amount )
{
  // don't actually take the locks yet
  unique_lock<mutex> lock1( from.mMutex, defer_lock );
  unique_lock<mutex> lock2( to.mMutex, defer_lock );

  // lock both unique_locks without deadlock
  lock( lock1, lock2 );

  from.iMoney -= amount;
  to.iMoney += amount;

  // output log
  cout << "Transfer " << amount << " from "
       << from.sName << " to " << to.sName << endl;
}

int main()
{
  bank_account Account1( "User1", 100 );
  bank_account Account2( "User2", 50 );

  thread t1( [&](){ transfer( Account1, Account2, 10 ); } );
  thread t2( [&](){ transfer( Account2, Account1, 5 ); } );

  t1.join();
  t2.join();
}
```

- 由於在進行轉帳（transfer()）的時候，必須要先鎖定兩個帳號，然後再做數值的修改，所以這邊的範例，是在 transfer() 裡，先透過 unique_lock 來管理兩個帳號的 mutex；不過要注意的是，比較不一樣的地方，是在建立 unique_lock 物件的時候，他還加上了第二個參數 defer_lock ~
- 這個參數的目的，是告訴 unique_lock 雖然要讓他去管理指定的 mutex 物件，但是不要立刻去鎖定他、而是維持沒有鎖定的狀態。而接下來，則是再透過 lock() 這個函式，來同時鎖定 lock1 和 lock2 這兩個 unique_lock 物件。

- 為什麼需要這樣做，而不直接寫成：

  lock_guard<mutex> lock1(from.mMutex);
  lock_guard<mutex> lock2(to.mMutex);

- 因為如果用上面的寫法，直接依序各自鎖定兩個 mutex 的話，有可能會在多執行序的情況下，產生 dead lock 的狀況。
- 舉例來說，如果同時要求進行「由 A 轉帳給 B」（thread 1）和「由 B 轉帳給 A」（thread 2）的動作的話，有可能會產生一個狀況，就是在 thread 1 裡面已經鎖定了 A 的 mutex，但是在試圖鎖定 B 的 mutex 的時候，thread 2 已經鎖定了 B 的 mutex；而同樣地，這時候 thread 2 也需要去鎖定 A 的 mutex，但是他卻已經被 thread 1 鎖定了。

- 如此一來，就會變成 thread 1 鎖著 A 的 mutex 在等 thread 2 把 B 的 mutex 解鎖，而 thread 2 鎖著 B 的 mutex 在等 thread 1 把 A 的 mutex 解鎖的狀況...這樣的情況，基本上是無解的。而透過像上面這樣的方法，使用 lock() 這個函式，就可以一口氣把多個 mutex 物件進行鎖定，並免這樣的狀況發生。

作業一~三

- 請撰寫一個有兩個執行緒的程式, 分別模擬兩個玩猜拳遊戲的人, 每次出拳後顯示輸贏以及目前雙方的輸贏總次數。
- 請撰寫一個等待泡麵的程式, 讓使用者輸入要泡麵的分鐘數, 並且在時間到時提醒使用者可以吃麵。
- 現今有 3 個人共用一本活期存款的賬戶，而於同一個時間分別存入 500元、 1000元、 2000元，又分別的提出200元、300元、500元，請顯示餘額（請開三個執行緒）