Parallel Computing (V) C++11 Threading

Cheng-Hung Lin

std::thread 常用的成員函式

- get_id(): 取得目前的執行緒的 id, 回傳一個為 std::thread::id 的類型。
- joinable(): 檢查是否可join。
- join(): 等待執行緒完成。
- detach(): 與該執行緒分離, 一旦該執行緒執行完後它所分配的資源會被釋放。
- native_handle(): 取得平台原生的native handle (例如Win32的Handle, unix的pthread_t)。

- 其他相關的常用函式有,
- sleep_for(): 停止目前執行緒一段指定的時間。
- yield(): 暫時放棄CPU一段時間, 讓給其它執行緒。

Example 1

- c++ 最簡單的 std::thread 範例如下所示，呼叫 thread 建構子時會立即同時地開始執行這個新建立的執行緒，之後 main() 的主執行緒也會繼續執行，基本上這就是一個基本的建立執行緒的功能了。

```
#include <iostream>
#include <thread>

void myfunc() {
    std::cout << "myfunc\n";
    // do something ...
}

int main() {
    std::thread t1(myfunc);
    t1.join();
    return 0;
}
```

```
$ g++ thread1.cpp -o thread1 -lpthread
$ ./thread1
myfunc
```

Example 2. 建立新 thread 來執行一個函式，且帶入有/無參數

- 以下例子為建立新 c++ thread 來執行一個函式，其中 t1 是呼叫無參數的 foo() 函式，而 t2 執行緒是呼叫 bar() 有參數的函式。

```
// g++ std-thread1.cpp -o a.out -std=c++11 -pthread
#include <iostream>
#include <thread>

void foo() {
    std::cout << "foo\n";
}

void bar(int x) {
    std::cout << x << "bar\n";
}
```

```
int main() {
    std::thread t1(foo); // 建立一個新執行緒且執行 foo 函式
    std::thread t2(bar, 0); // 建立一個新執行緒且執行 bar 函式
    std::cout << "main, foo and bar now execute concurrently...\n"; // synchronize threads
    std::cout << "sleep 1s\n";
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "join t1\n";
    t1.join(); // 等待 t1 執行緒結束
    std::cout << "join t2\n";
    t2.join(); // 等待 t2 執行緒結束
    std::cout << "foo and bar completed.\n";
    return 0;
}
```

```
$ ./thread2
main, foo and bar now execute concurrently...
0bar
foo
sleep 1s
join t1
join t2
foo and bar completed.
$ ./thread2
foo
main, foo and bar now execute concurrently...
sleep 1s
0bar
join t1
join t2
foo and bar completed.
```

Example 3:join 等待 thread 執行結束

- 在 main 主執行緒建立 t1 執行緒後，主執行緒便繼續往下執行
- 如果主執行緒需要等 t1 執行完畢後才能繼續執行的話就需要使用 join，即等待 t1 執行緒執行完 foo 後，主執行緒才能繼續執行，否則主執行緒會一直卡(blocking)在 join 這一行。

```
#include <iostream>
#include <thread>
#include <chrono>

void foo() {
        std::this_thread::sleep_for(std::chrono::milliseconds(200));
        std::cout<<"foo"<<std::endl;
}

int main() {
    std::thread t1(foo);
    std::cout<<"main 1"<<std::endl;
    t1.join();
    std::cout<<"main 2"<<std::endl;
    return 0;
}
```

Example 4: detach 不等待 thread 執行結束

- 如果主執行緒不想等或是可以不用等待 t1 執行緒的話，就可以使用 detach 來讓 t1 執行緒分離，接著主執行緒就可以繼續執行，t1執行緒也在繼續執行。

- 在整個程式結束前最好養成好習慣確保所有子執行緒都已執行完畢。

- 因為在 linux 系統如果主執行緒執行結束還有子執行緒在執行的話會跳出個錯誤訊息。

```
#include <iostream>
#include <thread>
#include <chrono>

void foo() {
        std::this_thread::sleep_for(std::chrono::milliseconds(200));
        std::cout<<"foo"<<std::endl;
}

int main() {
    std::thread t1(foo);
    std::cout<<"main 1"<<std::endl;
    t1.detach();
    std::cout<<"main 2"<<std::endl;
    return 0;
}
```

用陣列建立多個 thread

```
#include <iostream>
#include <thread>

void foo(int n) {
    std::cout << "foo() " << n << "\n";
}

int main() {
    std::thread threads[4];

    for (int i = 0; i < 4; i++) {
        threads[i] = std::thread(foo, i);
    }

    for (int i = 0; i < 4; i++) {
        threads[i].join();
    }

    std::cout << "main() exit.\n";

    return 0;
}
```

用 vector 建立多個 thread

```
#include <iostream>
#include <thread>
#include <vector>

void foo(int n) {
    std::cout << "foo() " << n << std::endl;
}

int main() {
    std::vector<std::thread> threads;

    for (int i = 0; i < 4; i++) {
        threads.push_back(std::thread(foo, i));
    }

    for (int i = 0; i < 4; i++) {
        threads[i].join();
    }

    std::cout << "main() exit.\n";

    return 0;
}
```

Example 7: vector addition

```
#include <iostream>
#include <thread>
#define N 100

int A[N];
int B[N];
int C[N];
int goldenC[N];

void fun(int i){
        C[i] = A[i] + B[i];
}

int main() {
    int i;
    std::thread threads[N];
    struct timespec t_start, t_end;
    double elapsedTime;
    for(i = 0; i < N; i++) {
        A[i] = rand()%100;
        B[i] = rand()%100;
    }
```

```
// start time
   clock_gettime( CLOCK_REALTIME, &t_start);
   for(i = 0; i < N; i++){
       threads[i] = std::thread(fun, i);
   }

   for(i = 0; i < N; i++){
       threads[i].join();
   }
   // stop time
   clock_gettime( CLOCK_REALTIME, &t_end);

   // compute and print the elapsed time in millisec
   elapsedTime = (t_end.tv_sec - t_start.tv_sec) * 1000.0;
   elapsedTime += (t_end.tv_nsec - t_start.tv_nsec) / 1000000.0;
   printf("Parallel elapsedTime: %lf ms\n", elapsedTime);
   // start time
   clock_gettime( CLOCK_REALTIME, &t_start);
   for(i=0; i< N; i++){
       goldenC[i] = A[i] + B[i];
   }
```

```
// stop time
   clock_gettime( CLOCK_REALTIME, &t_end);

   // compute and print the elapsed time in millisec
   elapsedTime = (t_end.tv_sec - t_start.tv_sec) * 1000.0;
   elapsedTime += (t_end.tv_nsec - t_start.tv_nsec) / 1000000.0;
   printf("Sequential elapsedTime: %lf ms\n", elapsedTime);

   int pass = 1;
   for(i = 0; i < N; i++) {
       if(goldenC[i] != C[i]){
               pass = 0;
       }
   }
   if(pass==1)
       printf("Test pass!\n");
   else
       printf("Test fail!\n");
   return 0;
}
```

```
$ ./thread7
Parallel elapsedTime: 5.818500 ms
Sequential elapsedTime: 0.001343 ms
Test pass!
```

Example 8: Vector addition using 4 threads

```
#include <iostream>
#include <thread>
#define N 1000000
#define NUM_THREADS 4
int A[N];
int B[N];
int C[N];
int goldenC[N];

void fun(int pos){
    int i;
    for(i=pos; i<pos + N/NUM_THREADS; i++)
        C[i] = A[i] + B[i];
}
```

```
int main() {
    int i;
    std::thread threads[NUM_THREADS];
    struct timespec t_start, t_end;
    double elapsedTime;
    for(i = 0; i < N; i++) {
        A[i] = rand()%100;
        B[i] = rand()%100;
    }
    // start time
    clock_gettime( CLOCK_REALTIME, &t_start);
    for(i = 0; i < NUM_THREADS; i++){
        threads[i] = std::thread(fun, i*N/NUM_THREADS);
    }

    for(i = 0; i < NUM_THREADS; i++){
        threads[i].join();
    }
    // stop time
    clock_gettime( CLOCK_REALTIME, &t_end);
```

```
// compute and print the elapsed time in millisec
    elapsedTime = (t_end.tv_sec - t_start.tv_sec) * 1000.0;
    elapsedTime += (t_end.tv_nsec - t_start.tv_nsec) / 1000000.0;
    printf("Parallel elapsedTime: %lf ms\n", elapsedTime);

    // start time
    clock_gettime( CLOCK_REALTIME, &t_start);
    for(i=0; i< N; i++){
        goldenC[i] = A[i] + B[i];
    }
    // stop time
    clock_gettime( CLOCK_REALTIME, &t_end);

    // compute and print the elapsed time in millisec
    elapsedTime = (t_end.tv_sec - t_start.tv_sec) * 1000.0;
elapsedTime += (t_end.tv_nsec - t_start.tv_nsec) / 1000000.0;
    printf("Sequential elapsedTime: %lf ms\n", elapsedTime);
```

```
    int pass = 1;
    for(i = 0; i < N; i++) {
        if(goldenC[i] != C[i]){
                pass = 0;
        }
    }
    if(pass==1)
        printf("Test pass!\n");
    else
        printf("Test fail!\n");
    return 0;
}
```

```
$ ./thread8
Parallel elapsedTime: 1.108929 ms
Sequential elapsedTime: 3.552504 ms
Test pass!
```

多執行緒呼叫同一個函式 (沒有 mutex 鎖)

```
#include <iostream>
#include <thread>

using namespace std;

int g_count = 0;

void print(int n, char c) {
    for (int i = 0; i < n; ++i) {
        std::cout << c;
        g_count++;
    }
    std::cout << '\n';

    std::cout << "count=" << g_count << std::endl;
}
```

```
int main() {
    std::thread t1(print, 10, 'A');
    std::thread t2(print, 5, 'B');
    t1.join();
    t2.join();

    return 0;
}
```

```
$ ./thread9
AAAAAAAAAA
count=10B
BBBB
count=15
```

多執行緒呼叫同一個函式 (有 mutex 鎖)

```
// g++ std-mutex.cpp -o a.out -std=c++11 -pthread
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;
std::mutex g_mutex;
int g_count = 0;
void print(int n, char c) {
    // critical section (exclusive access to std::cout signaled by locking mtx):
    g_mutex.lock();
    for (int i = 0; i < n; ++i) {
        std::cout << c;
        g_count++;
    }
    std::cout << '\n';

    std::cout << "count=" << g_count << std::endl;
    g_mutex.unlock();
}
```

```
int main() {
    std::thread t1(print, 10, 'A');
    std::thread t2(print, 5, 'B');
    t1.join();
    t2.join();
    return 0;
}
```

```
$ ./thread10
AAAAAAAAAA
count=10
BBBBB
count=15
```

condition_variable

- 使用 std::condition_variable 的 wait 會把目前的執行緒 thread 停下來並且等候事件通知，而在另外一個執行緒裡我們可以使用 std::condition_variable 的 notify_one 或 notify_all 去發送通知那些正在等待的事件
- 需要引入的標頭檔:<condition_variable>

- 以下為 condition_variable 常用的成員函式與說明，
  - wait:阻塞當前執行緒直到條件變量被喚醒
  - notify_one:通知一個正在等待的執行緒
  - notify_all:通知所有正在等待的執行緒

- 使用 std::condition_variable 的 wait 必須要搭配 std::unique_lock<std::mutex> 一起使用。

用 notify_one 通知一個正在 wait 的執行緒

- 先開一個新的執行緒 worker_thread 然後使用 std::condition_variable 的 wait 事件的通知
- 此時 worker_thread 會阻塞(block)直到事件通知才會被喚醒,
  之後 main 主程式延遲個 5 ms 在使用 std::condition_variable 的 notify_one 發送,
  之後 worker_thread 收到 來自主執行緒的事件通知就離開 wait 繼續往下 cout 完就結束該執行緒,

```
#include <iostream>
#include <string>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex m;
std::condition_variable cond_var;

void worker_thread()
{
    std::unique_lock<std::mutex> lock(m);
    std::cout << "worker_thread() wait\n";
    cond_var.wait(lock);

    // after the wait, we own the lock.
    std::cout << "worker_thread() is processing data\n";
}
```

```
int main(){
    std::thread worker(worker_thread);

    std::this_thread::sleep_for(std::chrono::milliseconds(5));
    std::cout << "main() notify_one\n";
    cond_var.notify_one();

    worker.join();
    std::cout << "main() end\n";
    return 0;
}
```

```
$ ./thread11
worker_thread() wait
main() notify_one
worker_thread() is processing data
main() end
```

用 notify_all 通知全部多個 wait 等待的執行緒

- 以下範例主要目的是建立5個執行緒並等待通知,
- 之後主程式執行go函式裡的cond_var.notify_all()去通知所有正在等待的執行緒, 也就是剛剛建立的5個執行緒,
- 這5個執行緒分別收到通知後從wait函式離開, 之後檢查ready變數為true就離開迴圈,
- 接著印出thread id然後結束該執行緒。

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex m;
std::condition_variable cond_var;
bool ready = false;

void print_id(int id) {
    std::unique_lock<std::mutex> lock(m);
    while (!ready) {
        cond_var.wait(lock);
    }
    std::cout << "thread " << id << '\n';
}

void go() {
    std::unique_lock<std::mutex> lock(m);
    ready = true;
    cond_var.notify_all();
}
```

```
int main()
{
    std::thread threads[5];
    // spawn 5 threads:
    for (int i=0; i<5; ++i)
        threads[i] = std::thread(print_id,i);

    std::cout << "5 threads ready to race...\n";
    go();

    for (auto& th : threads)
        th.join();

    return 0;
}
```

```
$ ./thread12
5 threads ready to race...
thread 1
thread 0
thread 2
thread 3
thread 4
$ ./thread12
5 threads ready to race...
thread 1
thread 2
thread 4
thread 0
thread 3
$ ./thread12
5 threads ready to race...
thread 2
thread 3
thread 4
thread 0
thread 1
```

- 這個範例多使用了一個額外的 ready 變數來輔助判斷，也間接介紹了 cond_var.wait的另一種用法，

- 使用一個 while 迴圈來不斷的檢查 ready 變數，條件不成立的話就 cond_var.wait繼續等待，
- 等到下次cond_var.wait被喚醒又會再度檢查這個 ready 值，一直迴圈檢查下去，
- 這技巧在某些情形下可以避免假喚醒這個問題，
- 簡單說就是「cond_var.wait被喚醒後還要多判斷一個 bool 變數，一定要條件成立才會結束等待，否則繼續等待」。