多執行緒 — C++ Thread

李謦伊 Jul 25 · 13 min read

之前有提到為了提高 CPU 的使用率，可以採用多執行緒的方式，以及介紹多執行緒的相關概念，不太清楚觀念的可以去看我那篇文章~~

多執行緒

為了提高CPU的使用率，將某些需要耗時較多的任務或是大量I/O操作 (I/O處理速度很慢)，採用多執行緒可以適當地提高程式的執行效率。

medium.com

本文將要來介紹如何利用 C++建立一個多執行緒以及應用~~

C++ 11之後有了std::thread函式庫，需要引入的標頭檔: <thread>

先來介紹 Thread 的 member function 有哪些吧~

- 用來查看當前執行緒的id

```
thread::get_id()
```

- 檢查此執行緒是否還和主執行緒連接 (已經完成join、detach 的執行緒都是false)

```
thread::joinable()
```

- 將執行緒與主執行緒的連接切斷，並且此執行緒會繼續獨立執行下去，直到執行結束時釋放分配的資源

```
thread::detach()
```

- 交換兩個執行緒物件

```
thread::swap()
```

```
==========================================================================
```

```
std::this_thread 命名空間(Namespace)
```

- 用來查看當前執行緒的id

```
this_thread::get_id()
```

```
- 暫時中斷此執行緒，OS會調用其他執行緒執行

this_thread::yield()

- 設定一個時間，讓此執行緒在指定的時刻後再繼續執行

this_thread::sleep_until()

- 暫時中斷此執行緒，等待指定的一段時間後才會被執行

this_thread::sleep_for()
```

接著來示範一些簡單的執行緒操作~~

- 建立執行緒、等待指定執行緒結束

使用 thread <執行緒名稱>(<function>) 建立一個執行緒，若要傳入參數，可以在 function 後加入第二個參數

使用 join() 將主執行緒暫停，等待指定的執行緒結束，主執行緒才會結束

```
#include <iostream>
#include <thread>
using namespace std;
```

```
void first_thread_job()
{
 cout << "This is the first thread "<< endl;
}


// 傳入string x
void second_thread_job(string x)
{
 cout << "This is the second thread " << x << endl;
}


int main()
{
 // 建立執行緒
 thread first_thread(first_thread_job);
 thread second_thread(second_thread_job, "abc");


 // 將主執行緒暫停，等待指定的執行緒結束
 first_thread.join();
 second_thread.join();


return 0;
}


// ====== output ======
// This is the first thread
// This is the second thread abc
```

- 接下來要使用到同步機制 (Synchronized)有以下幾種：互斥量 (Mutex), 訊號量 (Semaphore), 條件變數 (Condition Variable), 原子變數 (Atomic), 隊列 (Queue), 事件 (Event)

- Queue

Thread 無法回傳值，所以要使用 queue.push() 將要傳回的值存入 queue，再用 queue.pop() 取出

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


using namespace std;
queue<int> q1;
queue<int>::size_type q1_size;


void first_thread_job(int x)
{
 // 將元素放入 queue
 q1.push(x);
 cout << "This is the first thread " << x << endl;
}


int main()
{
 thread first_thread(first_thread_job, 2);
 first_thread.join();


 // 傳回隊列的第一個元素，並沒有將此元素剔除隊列
 int a = q1.front();
 cout << "The first element is " << a << endl;
 q1_size = q1.size();
 cout << "The queue1 length is " << q1_size << endl;
```

```
 // 彈出對列的第一個元素
 q1.pop();
 q1_size = q1.size();
 cout << "The queue2 length is " << q1_size << endl;


return 0;
}


// ====== output ======
// This is the first thread 2
// The first element is 2
// The queue1 length is 1
// The queue2 length is 0
```

- Lock

當同時有幾個 Thread 要用到同一個資料時，為了不發生 Race Condition 的現象，需要使用 lock() 以及 unlock() 來將其鎖定住，不讓其他Thread 執行，C++需要引入標頭檔: <mutex>

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


using namespace std;


// 定義 lock
mutex mu;
```

```
void first_thread_job()
{
 mu.lock();
 cout << "This is the first thread "<< endl;
 mu.unlock();
}


int main()
{
 thread first_thread(first_thread_job);
 first_thread.join();


 return 0;
}
```

- 除了 Lock之外，mutex 還提供了 lock_guard 及 unique_lock
- lock_guard

採用 RAII 方法來對 mutex 對象進行自動加鎖、解鎖的動作，可以保證執行緒的安全

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


using namespace std;
mutex g_mutex;
```

```
void first_thread_job()
{
 lock_guard<mutex> lock(g_mutex);
 cout << "This is the first thread " << endl;
}


int main()
{
 thread first_thread( first_thread_job);
 first_thread.join();


 return 0;
}
```

- unique_lock

獨占所有權的方式對 mutex 對象進行自動加鎖、解鎖的動作，具有 lock_guard 的功能，而且更靈活，支持移動賦值的動作，還支持同時鎖定多個 mutex，但所花費的時間與記憶體更多，因此如果是 lock_guard 可以處理的執行緒，會盡量使用lock_guard

unique_lock 寫法跟 lock_guard 類似

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

```
using namespace std;
mutex u_mutex;


void first_thread_job()
{
 unique_lock<mutex> lock(u_mutex);
 cout << "This is the first thread " << endl;
}


int main()
{
 thread first_thread( first_thread_job);
 first_thread.join();


 return 0;
}
```

- Semaphore

mutex 的擴充版，可以允許多個執行緒同時執行，以下是 Semaphore 的基本程式碼

```
- 初始化 Semaphore

sem_init(sem_t *sem, int pshared, unsigned int value);

- 定義 Semaphore 的名稱
```

```
sem_t *sem

- 設定為0 表示僅供目前的 process 及其 Thread 使用。非0 表示此 Semaphore 與其他 process 共用

int pshared

- 設定 Semaphore 計數器

unsigned int value

- 用來阻塞該執行緒，直到 Semaphore 的值大於0，若解除阻塞後，Semaphore 的值會減1，表示可執行的次數減1

sem_wait(sem_t *sem);

- 當有執行緒阻塞在信號上，調用此函數會使其中一個執行緒解除阻塞，此時 Semaphore 的值加1

sem_post(sem_t *sem);

- 刪除 Semaphore

sem_destroy(sem_t *sem);
```

接下來示範使用 Semaphore

```
#include <iostream>
#include <thread>
#include <semaphore.h>


using namespace std;
sem_t binSem;
int a;


void first_thread_job()
{
 sem_wait(&binSem);
 for (int i = 0; i < 3; i++)
 {
 a += 1;
 cout << "This is the first thread " << a << endl;
 }
}


void second_thread_job()
{
 for (int i = 0; i < 3; i++)
 {
 a -= 1;
 cout << "This is the second thread " << a << endl;
 }
 sem_post(&binSem);
}


int main()
{
 int res;
 // Semaphore 初始化
 res = sem_init(&binSem, 0, 0);
```

```
 // 建立執行緒
 thread first_thread(first_thread_job);
 thread second_thread(second_thread_job);


 // 將主執行緒暫停，等待指定的執行緒結束
 first_thread.join();
 second_thread.join();


return 0;
}


// ====== output ======
// first_thread_job 被阻塞，直到second_thread_job 把信號加1，才開始執行
// This is the second thread -1
// This is the second thread -2
// This is the second thread -3
// This is the first thread -2
// This is the first thread -1
// This is the first thread 0
```

- Condition Variable

用於等待的同步機制，能阻塞一個或多個執行緒，condition_variable 提供 wait() 將執行緒停下來等待通知，直到接收到另一個執行緒發出的通知才會被喚醒，提供 notify_one() 或 notify_all() 這兩個函式。

notify_one() 只會通知其中一個正在等待的執行緒，notify_all() 則是會通知所有正在等待的執行緒，需要和 mutex 配合使用。

其中 wait() 可以有兩個參數，第一個參數是 mutex，第二個參數是返回類型為 bool 的變數，當變數回傳 true 時，執行緒才會停止等待，如果回傳 false，執行緒則會等待下一次的通知

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


using namespace std;
condition_variable cond_var;
mutex u_mutex;
bool ready = false;


void first_thread_job()
{
 unique_lock<mutex> lock(u_mutex);

// 使用 wait() 進行等待
 cout << "thread wait" << endl;
 cond_var.wait(lock);


 cout << "This is the first thread " << endl;
}


void second_thread_job()
{
 unique_lock<mutex> lock(u_mutex);

// 使用 wait() 進行等待 傳入第二個參數
 cout << "thread wait" << endl;
```

```
 cond_var.wait(lock, [](){ return ready; });

 cout << "This is the second thread " << endl;
}

int main()
{
 thread first_thread(first_thread_job);
 thread second_thread(second_thread_job);

 cout << "wait 5 second…" << endl;
 this_thread::sleep_for(std::chrono::milliseconds(5));

// 使用 notify_one() 喚醒執行緒
 cout << "thread notify_one" << endl;
 cond_var.notify_one();

 // 回傳 ready 判斷是否要停止等待
 ready = true;

// 使用 notify_one() 喚醒執行緒
 cout << "thread notify_one" << endl;
 cond_var.notify_one();

 first_thread.join();
 second_thread.join();

 return 0;
}
```

除了 wait() 外，還有提供 wait_for() 和 wait_until() 這兩個函式，限制等待的時間。wait_for() 需要給定指定長度的時間，wait_until() 則是需要指定一個時間點，時間的形式要使用 STL chrono

```
cond_var.wait_for(lock, chrono::seconds(5))


cond_var.wait_until(lock,chrono::system_clock::now()+chrono::secon
ds(5))
```

- Atomic

在C++11中引入了原子操作的概念，提供更簡單的機制確保執行緒的安全與存取共享變數的正確性。在任意時刻只有一個執行緒能存取這個資源，有點類似互斥的保護機制，但原子變數的操作比鎖的使用效率更高。

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


using namespace std;


// 定義原子變數
atomic_int atomic_a(0);
```

```
void first_thread_job()
{
  for (int i = 0; i < 3; i++)
  {
    atomic_a += 1;
    cout << "This is the first thread "<< atomic_a << endl;
  }
}


int main()
{
 // 建立執行緒
 thread first_thread(first_thread_job);
 first_thread.join();


 return 0;
}


// ====== output ======
// This is the first thread 1
// This is the first thread 2
// This is the first thread 3
```

這些就是使用 C++ 建立執行緒的方式 ，若想看使用 Python 建立執行緒的話可以看我這篇文~~

多執行緒 — Python Threading

上一篇文有提到為了提高 CPU 的使用率，可以採用多執行緒的方式，本文將要來介紹如何利用 Python 建立一個多執行緒以及應用~~

medium.com

Multithreading Threads