C++並行程式設計

**目錄**

[C++程式設計：遞迴 1](#_Toc204799122)

[1.1 範例：費波那契數列（Fibonacci） 1](#_Toc204799123)

[1.2 範例：最大公因數（GCD） 1](#_Toc204799124)

[1.3 範例：河內塔（Tower of Hanoi） 2](#_Toc204799125)

# 何謂並行

## 作業系統的程序（Process）並行：

作業系統會將不同的工作任務（tasks）分配給不同的程序。程序之間是獨立的，彼此擁有自己的記憶體空間，因此程序之間的通訊需要額外的機制（例如：pipe, socket, 檔案）。

* 程序切換（context switch）的開銷比較大，因為作業系統需要儲存和還原整個程序的狀態。
* 適合用在需要高安全性或分散式系統的應用，例如：網頁伺服器、資料庫。

## 為何並行

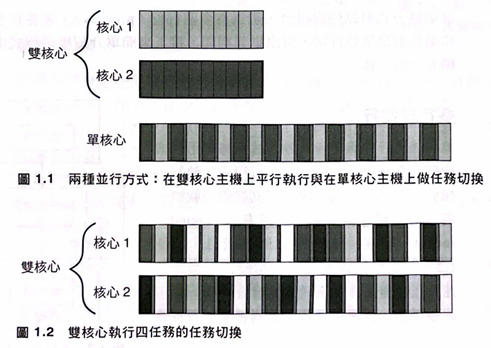
在應用程式中使用並行主要有兩個原因：「分離注意力焦點」（separation of concerns） 和 「提升效能」（performance）。絕大多數採用並行的原因都可以歸結到這兩點，甚至是兩點同時成立。

## 多執行緒（Multithreading）並行

這是一種在單一程序（process）內執行多個執行緒（threads）的方式。

* 共享記憶體：所有執行緒共享相同的記憶體空間（address space），因此它們之間可以更快速、方便地共享資料。
* 輕量級：執行緒的創建和切換開銷比程序小得多。
* 通訊快速：因為共享記憶體，執行緒之間的通訊非常有效率。

適合用在需要頻繁共享資料、追求高效能的應用。C++標準函式庫主要支援的是這種多執行緒並行方式。



## 將Hello, World修改為多執行緒版本

傳統的「Hello World」

#include <iostream>

using namespace std;

int main(){

    cout << "Hello World\n";

}

改為多執行續版本

#include <iostream>

#include <thread>

using namespace std;

void hello(){

    cout << "Hello Concurrent World\n";

}

int main(){

    cout << "Hello World\n";

    thread t(hello); // 這會啟動一個新的執行緒，這個新執行緒會立即開始執行hello函式中的程式碼

    t.join(); // 讓main執行緒等待t所代表的新執行緒執行結束。

}

# 管理thrread

## 啟動執行緒

thread物件代表了一個正在執行或可以執行的執行緒

## 等待執行緒結束 (Joining)

在main或其他執行緒結束前，通常需要確保它所啟動的所有子執行緒都已經完成任務。如果不等待，父執行緒可能會提前結束，導致子執行緒還沒執行完就被終止，造成不可預期的後果。

* thread.join()函式會阻塞目前的執行緒（例如main執行緒），直到t所代表的子執行緒執行結束為止。
* thread.joinable()：會返回一個布林值，指示該std::thread物件是否與一個可執行的執行緒相關聯。

## 等待例外狀況 (Detaching)

有時，我們不希望等待新執行緒完成。新執行緒可以獨立於其父執行緒在背景執行，當父執行緒結束時，子執行緒也不會受到影響。這就像一個精靈程序（daemon process）。

Thread.detach()會將t所代表的執行緒從std::thread物件中分離。此後，這個執行緒會在背景獨立運行，並由C++執行緒執行時期函式庫來接管其資源。當它完成任務時，其資源會被自動回收。旦呼叫了detach()，該std::thread物件就不再與執行緒相關聯，joinable()會返回false。

## 背景執行緒

在thread上呼叫detach()所產生的執行緒，就是背景執行緒（background thread）。

## 傳遞引數到執行緒函式

啟動執行緒時，經常需要傳遞參數給執行緒要執行的函式。傳遞的方式和呼叫一般函式一樣，直接在 std::thread 建構式中，在函式名後面加上參數。

#include <iostream>

#include <thread>

#include <string>

using namespace std;

void f(int i, const std::string& s) {

    cout << "Thread received: " << i << " and " << s << endl;

}

int main() {

    int i = 3;

    string s = "hello";

    // 啟動一個新執行緒，傳遞 i 和 s 作為參數

    thread t(f, i, s);

    // 重要觀念：std::thread 的建構式會對參數進行\*\*複製\*\*

    // 即使函式 f 的第二個參數是參考 `const std::string&`，

    // s 仍然會被複製一份，然後新執行緒會使用這份複製來執行。

    // 這可以避免主執行緒在子執行緒讀取前，就修改或釋放 s。

    t.join();

    return 0;

}

如果真的需要傳遞參數的參考，讓子執行緒能夠修改父執行緒的變數，就必須使用 std::ref

#include <iostream>

#include <thread>

#include <string>

using namespace std;

void update\_string(string& s) {

    s += " updated by thread.";

}

int main() {

    string my\_string = "original string";

    cout << "Original string: " << my\_string << endl;

    // 啟動新執行緒，並使用 std::ref 傳遞 my\_string 的參考

    thread t(update\_string, ref(my\_string));

    t.join();

    cout << "String after thread: " << my\_string << endl;

    return 0;

}

// Original string: original string

// String after thread: original string updated by thread.

## 轉移執行緒所有權

std::thread 物件是不可複製的（non-copyable），但它是可移動的（moveable）。這代表一個執行緒的所有權可以從一個 std::thread 物件轉移到另一個，而不用重新啟動新的執行緒。

#include <iostream>

#include <thread>

using namespace std;

void task() {

    cout << "Task is running." << endl;

}

int main() {

    thread t1(task); // t1 擁有一個執行緒的所有權

    thread t2;  // 宣告一個新的 std::thread 物件，但不立刻啟動

    // 將 t1 的執行緒所有權轉移給 t2

    t2 = move(t1);

    // 此時，t1 不再擁有任何執行緒，是不可加入的 (not joinable)

    // t2 現在擁有原本由 t1 啟動的執行緒所有權

    cout<<t1.joinable()<<endl;

    cout<<t2.joinable()<<endl;

    t2.join(); // 我們現在要用 t2 來等待執行緒結束

    return 0;

}

## 決定執行緒數

這是一個實用的函式，可以返回硬體支援的並行執行緒數量，通常是CPU核心數。這是決定要啟動多少個執行緒的一個重要參考。

std::thread::hardware\_concurrency()：返回CPU核心數，可以幫助你避免創建過多執行緒，造成過度的上下文切換開銷。

#include <iostream>

#include <thread>

#include <vector>

using namespace std;

void worker\_task(int id) {

    cout << "Worker thread " << id << " is running." << endl;

}

int main() {

    // 取得硬體支援的並行執行緒數量

    unsigned int num\_threads = std::thread::hardware\_concurrency(); // 16 threads

    // 如果無法取得，給一個預設值，例如 4

    if (num\_threads == 0) {

        num\_threads = 4;

    }

    cout << "Hardware concurrency: " << num\_threads << " threads." << endl;

    vector<thread> threads;

    for (unsigned int i = 0; i < num\_threads; ++i) {

        // 啟動與核心數相同數量的執行緒

        threads.push\_back(thread(worker\_task, i));

    }

    // 等待所有執行緒結束

    for (auto& t : threads) {

        t.join();

    }

    return 0;

}

## 識別執行緒

每個執行緒都有一個唯一的ID，可以用來識別它。

#include <iostream>

#include <thread>

#include <vector>

using namespace std;

void worker\_task() {

    thread::id my\_id = this\_thread::get\_id();

    cout << "Hello from thread with ID: " << my\_id <<endl;

}

int main() {

    cout << "Main thread ID: " << this\_thread::get\_id() << endl;

    thread t1(worker\_task);

    thread t2(worker\_task);

    // 取得 std::thread 物件所代表的執行緒 ID

    cout << "t1 thread ID: " << t1.get\_id() << endl;

    cout << "t2 thread ID: " << t2.get\_id() << endl;

    t1.join();

    t2.join();

    return 0;

}