關於本書

本書是新 C++ 標準中併發和多執行緒功能的深入指引，內容從基本的 `std::thread`、`std::mutex` 和 `std::async` 等用法到複雜的原子操作和記憶體模型。

本書架構

前四章介紹函式庫提供的各種功能及使用方法。

第 5 章涵蓋了記憶體模型和原子操作的底層細節，包括如何用原子操作對其他程式碼施加排序約束，這章是本書介紹性部分的最後一章。

第 6 章和第 7 章從高層主題開始，並舉例說明如何使用基本功能來構建更複雜的資料結構——如第 6 章的基於上鎖的資料結構，第 7 章的無鎖資料結構。

第 8 章繼續討論更高層次的主題，提供設計多執行緒程式的指引，涵蓋了會影響性能的議題以及各種平行運算法實作的範例。

第 9 章介紹了執行緒管理——執行緒池、工作佇列和中斷操作等。

第 10 章介紹 C++17 對新平行性的支援，它以許多標準函式庫運算法額外多載的形式出現。

第 11 章包括了測試和除錯——錯誤的型態、找出錯誤位置的技術以及如何對錯誤進行測試等。

附錄包括了在新標準中所引入的一些與多執行緒相關的新功能簡要說明，第 4 章所提到的訊息傳遞函式庫實作細節，以及對 C++17 執行緒庫的完整參考。

誰應該讀這本書

如果你正在用 C++ 撰寫多執行緒程式，那你應該閱讀本書；如果你使用的是 C++ 標準函式庫中新的多執行緒功能，那這本書將是你不可或缺的指引；

如果你使用的是其他執行緒庫，那麼後面幾章的指引和技術對你應該仍然有用。

儘管不熟悉新語言的功能，但假設你已經具有良好的 C++ 工作知識——附錄 A 中涵蓋了這些知識；雖然所提供的這些知識或經驗可能很有用，但也不一定需要事先具備撰寫多執行緒程式的知識或經驗。

如何使用這本書

如果你以前從未撰寫過多執行緒程式，建議你從頭到尾依序閱讀本書，但也許可以先跳過第 5 章更詳細的內容。不過第 7 章在很大程度上依賴於第 5 章的內容，因此如果你先跳過第 5 章，那應該將第 7 章留到讀完它之後再讀。

如果你以前沒有使用過新的 C++11 語言功能，那值得在開始閱讀之前先略讀一下附錄 A，以確保你可以快速掌握本書的範例。不過，新語言功能的用法會在書中突顯表示，如果你遇到以前從未見過的內容，可以隨時翻到附錄中查閱。

如果你有在其他環境中撰寫多執行緒程式的豐富經驗，那麼開始的前幾章可能仍然值得一讀，這樣你可以了解到已知的功能是如何映射到 C++ 新的標準上。如果你要使用原子變數進行任何低階工作，那麼第 5 章是必讀的。為了確保你熟悉多執行緒 C++ 中異常安全之類的知識，溫習一下第 8 章是值得的。如果你有特定的工作，索引和目錄應該可以幫助你快速找到相關的章節。

一旦你掌握了 C++ 執行緒庫的用法，附錄 D 應該還是會很有用處，例如用於查找每個類別和函式呼叫的確切細節等。你也可能會希望經常回到主要章節，以刷新你對特定結構的記憶或是查看範例程式。

程式碼編排慣例及下載

程式列表或文本中所有原始程式碼都採用固定寬度的字體，將它們與普通本文分開。許多程式列表中含有註釋，以突顯重要的概念。在某些情況下，程式列表後面的註釋也會鏈結有項目編號。

本書中所有範例的原始程式碼都可以從出版商的網站下載，網址為 www.manning.com/books/c-plus-plus-concurrency-in-action-second-edition；你也可以從 github 下載，網址為 https://github.com/anthonywilliams/ccia_code_samples。

軟體需求

要在不更改下使用本書的程式碼，你需要使用支援範例中所用 C++17 語言功能的最新版 C++ 編譯器（請參閱附錄 A），並且還需要 C++ 標準執行緒庫的複製。

在編寫本書時，最新版本的 g++、clang++ 和 Microsoft Visual Studio 等都隨附了 C++17 標準執行緒庫的實作。它們也支援附錄中大部分的語言功能，而不支援的那些功能也會很快的納入支援行列。

我的公司 Just Software Solutions Ltd，對一些較舊版本編譯器的 C++11 標準執行緒庫的完整實作，以及 clang、gcc 和 Microsoft Visual Studio 較新版本 Concurrency TS 的實作都有販售[1]。本書中的範例都已經用這些實作測試過。

Boost Thread Library[2] 提供了一個基於 C++11 標準執行緒庫建議的 API，並且可以移植到許多平台上。只要明智地將 `std::` 替換成 `boost::` 並使用適當的 `#include` 指令，就可以修改本書中大多數範例使能與 Boost 執行緒庫搭配使用。但有一些功能 Boost 執行緒庫不支援（如 `std::async`），或具有不同的名稱（如 `boost::unique_future`）。

1 The `just::thread` implementation of the C++ Standard Thread Library, http://www.stdthread.co.uk.

2 Boost C++ 函式庫收藏，http://www.boost.org。

1 您好，C++ 的併發處理世界！

本章涵蓋以下內容

- 併發和多執行緒是什麼
- 為什麼你想要在應用程式中使用併發和多執行緒
- C++ 支援併發的歷程
- 一個簡單多執行緒 C++ 程式的樣子

對於 C++ 使用者來說，這是令人振奮的時刻。在 1998 年「C++ 標準」發佈 13 年後，C++ 標準委員會對此語言及其支援的函式庫進行了重大改革。新的 C++ 標準（稱為 C++11 或 C++0x）於 2011 年發佈，並伴隨一系列的改變，讓 C++ 用起來更容易和有效率。委員會還承諾後續將採「串列式發佈模式」，即每三年發佈一次新的 C++ 標準。到目前為止，已經經歷了二次發佈：2014 年的 C++14 標準與 2017 年的 C++17 標準，以及描述這些標準擴充的一些技術規範。

C++11 標準中最重要的新特色之一是支援多執行緒程式，這是 C++ 標準首次承認該語言中能使用多執行緒應用程式，並在函式庫中提供了撰寫多執行緒應用程式的元件，使得不必再依賴操作特定平台的擴充就可撰寫 C++ 多執行緒程式，更讓你能夠撰寫有正確動作的可攜式多執行緒程式。同時，程式設計者也越來越期待能用一般的併發功能，特別是設計多執行緒程式，來提高程式性能。C++14 和 C++17 標準就是建立在這基礎上，進一步支援用 C++ 撰寫多執行緒程式，技術規範也提供了進一步支援。有關併發擴充的技術規範與平行處理也已經納入 C++17 中。

本書主要討論在 C++ 撰寫多執行緒的併發程式，及促成此事可行的 C++ 語言特色和函式庫功能。我將先解釋併發和多執行緒是什麼意思，以及為什麼你會想在程式中使用併發。在快速探討為什麼你**未**在程式中使用併發後，將先概述 C++ 對併發的支援，然後用一個簡單的 C++ 併發實作範例來結束這一章。有多執行緒應用程式開發經驗的讀者可能希望先略過前面這些內容。在後續章節中，將提供更廣泛的範例，並更深入的探討函式庫功能，本書也將深入參考所有 C++ 標準函式庫中能用於多執行緒和併發的功能。

那麼，我所說的**併發**和**多執行緒**是什麼意思呢？

1.1　什麼是併發？

併發最簡單和基本的意思是，兩個或以上獨立的動作大約在同一時間發生。在人生過程中遇到併發是很自然的事，像是我們可以同時走路和說話，或用兩隻手做不同的動作，我們每個人都各自獨立生活——你看球賽的時候我去游泳等等。

1.1.1　在電腦系統中的併發

當我們在電腦領域討論併發時，指的是一個系統同時執行多個獨立工作，而不是循序或一個接一個執行。併發不是新的現象，多工作業系統讓單一台桌上型電腦透過工作切換方式，在同一時間執行多個應用程式早已司空見慣，而擁有多個處理器的高階伺服器，則更能處理真正的併發。與過去不同的是，能夠真正同時執行多個工作的電腦逐漸普及，而不是只給人一種可以這樣做的假象。

以往大多數桌上型電腦都只有一個處理器，只有一個處理單元或核心，現在還是有很多這樣的電腦。這種電腦一次只能執行一項工作，但它每秒可以在工作間切換多次。透過執行一個工作一小段時間，然後換執行另一個工作一小段時間，透過這樣互相切換的方式，這些工作看起來就像是同時在執行，這稱為**工作切換**，這種系統也稱為併發。因為工作切換速度很快，你無法判斷當處理器切換到另一個工作時，目前的工作可能會在哪一點暫停。工作切換為使用者和程式本身提供了併發的假象。因為實際上只有併發的假象，因此在單處理器工作切換環境中執行時，程式產生的行為可能會與在真正併發環境中執行時有明顯差異，特別是關於記憶體模式的錯誤假設（在第 5 章會討論到）可能不會在這樣的環境中出現，這部分會在第 10 章進行更深入的討論。

多年來，多處理器的電腦一直被用於伺服器和執行高性能的計算工作，而使用單晶片上有多個核心的處理器（多核心處理器）的電腦，也漸漸普及到一般電腦上。無論是多處理器或單處理器多個核心（或兩者兼有）的電腦，都能夠真正平行的執行多個工作，這稱為**硬體併發**。

圖 1.1 顯示一台電腦要執行兩個工作的理想化場景，其中每個工作被分成 10 個相等的小區塊。在雙核心機器（有兩個處理核心）上，每個工作都有自己的執行核心；而在運行工作切換的單核心機器上，每個工作區塊會彼此相互交織在一起，但區塊間會有少許空隙（如圖 1.1 中以比雙核心機器區塊間分隔線粗一點的灰色線段分隔區塊），每次要從一個工作換到另一個工作時，系統必須執行**語意切換**，但這需要一點切換時間。為了執行語意切換，作業系統必須儲存 CPU 狀態和目前執行工作的指令指標位置，算出要切換到哪個工作，並重新載入被切換到工作的 CPU 狀態，最後 CPU 還需要將新工作的指令和資料從記憶體載入到快取中，這會對 CPU 執行指令產生阻礙，造成更多延遲。

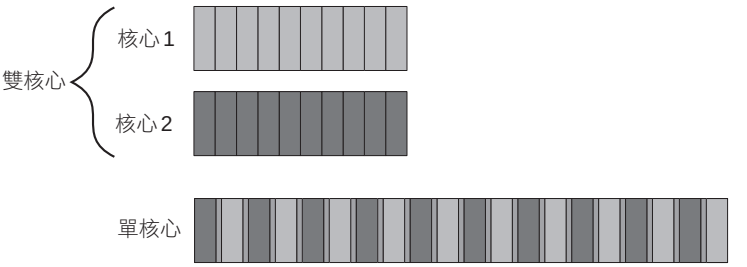

圖 1.1　併發的兩種方式：雙核心機器的平行執行與單核心機器上的工作切換

雖然在多處理器或多核心系統的硬體，很明顯的可以使用併發，但某些處理器也可以在單核心上執行多個執行緒，其中需要考慮的主要因素是**硬體執行緒**的數量，這是硬體可以真正同時執行多少個獨立工作的衡量指標。即使系統可以真正實現硬體併發功能，也很可能會遇到比硬體可以平行執行還要多的工作要進行，而在這情況下仍然需要用到工作切換。例如，在一般桌上型電腦上，即使電腦外表上看起來是處於閒置狀態，仍可能有數百個工作在後台執行，同時還可以執行文字處理器、編譯器、編輯器和網頁瀏覽器（或任何其他應用程式）。圖 1.2 顯示將工作分割成接近相同大小區塊的理想化場景下，雙核心機器上四個工作間的切換。實際上，許多問題會使工作分割不均勻及執行時序不規則，有些在探討影響併發程式性能的因素問題，會在第 8 章中說明。

圖 1.2　在雙核心上四個工作間的切換

無論程式是在單核心處理器或是在多核心處理器的機器上執行，本書所涵蓋的所有技術、功能和類別都可以使用，而且不會受到是透過工作切換或真正硬體的併發達成併發目的的影響。但要如何在程式中使用併發會受到硬體可以使用併發的數量決定。這議題也會在第 8 章中談到利用 C++ 語言設計併發程式所涉及的問題時再討論。

1.1.2　併發的方法

試想一下，有二位程式開發人員在一個軟體專案上合作。如果他們處於不同的辦公室，且各有自己的參考手冊，那他們可以不受互相干擾的專注於自己的工作；但要彼此討論的話就比較麻煩，必須使用電話、電子郵件，或者起身走到對方辦公室才能進行討論，而不是只要轉個身就能互相交談。除此之外，還要多負擔兩個辦公室的管理和購買多份參考手冊等額外費用。

如果將程式開發人員安排在同一間辦公室，他們就可以很方便地自由交談，討論程式設計的細節，也可以很容易地在紙上或白板上繪製流程圖幫助討論或說明設計理念；而且只有一間辦公室需要管理和提供一組資源就足夠了。不好的一面是，他們可能會發現很難集中注意力，且在共享資源上也可能出現問題（「參考手冊現在在哪裡？」）。

這兩種組織開發人員的方式說明了實現併發的兩種基本方法，其中每個開發人員代表一個執行緒，而每間辦公室則表示一個處理程序。第一種方法是有多個單執行緒的處理流程，如同兩位開發人員各有自己的辦公室；第二種方法是在單個處理流程中有多個執行緒，就像讓兩位開發人員處在同一間辦公室。你可以任意的組合這些情況並具有多個處理程序，其中有一些是多執行緒，而有些是單執行緒，但原則是一樣的。這就是在程式中實現兩種併發方法的概要說明。

多個處理程序的併發

在應用程式中使用併發的第一種方法，是將程式分割成多個、獨立的單執行緒程序，且同時執行這些程序，就像你可以同時執行網頁瀏覽器和文字處理器一樣。然後，這些獨立的程序可以經由一般程序之間的通訊頻道（訊號、Socket 介面、檔案、管線等）相互傳遞訊息，如圖 1.3 所示。但其中有一個缺點，因為作業系統通常會在處理程序之間提供很多保護，來避免一個程序意外修改到屬於另一個程序的資料，所以程序間的這種通訊一般在設定上比較複雜或通訊也較為緩慢。另一個缺點是，執行多個處理程序存有一個固有的代價，即啟動處理程序需要花些時間，作業系統也需要貢獻一些內部資源來管理處理程序。

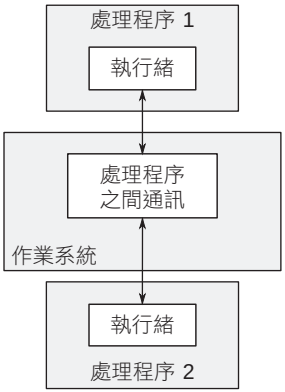

圖 1.3　在同時執行的二個程序之間的通訊

這也並非全是缺點：增加額外保護的作業系統一般會在處理程序和更高階的通訊機制間提供，這表示可以更容易地用處理程序撰寫安全的併發程式，而不必完全仰賴執行緒。事實上，像是 Erlang（www.erlang.org/）程式語言提供的環境，就是使用處理程序作為併發基本的組成區塊並有很大的成效。

使用獨立處理過程實現併發還有額外的好處 —— 可以藉由網路連接的不同機器執行獨立的處理程序。雖然這會增加通訊代價，但在仔細的設計下，這對增加平行可行性和改善性能而言，是非常值得嘗試的做法。

多個執行緒的併發

併發的另一種方法是在單一處理程序中執行多個執行緒。執行緒很像是輕量級的處理程序，因為每個執行緒獨立的執行，而且在不同的指令順序下執行。但是，處理程序中的所有執行緒都使用相同的位址空間，所有執行緒可以直接存取大部分資料，其中全域變數仍然保持是全域的，指向物件或資料的指標或參照也可以在執行緒之間傳遞。雖然這可能要在處理程序之間共用記憶體，而這點因為相同資料的記憶體位址在不同處理程序中不一定一樣，所以設定會比較複雜而且很難管理。圖 1.4 顯示在一個處理程序中，二個執行緒經由共用記憶體進行通訊。

因為作業系統的簿記操作會比較少，共用位址空間和執行緒之間對資料缺少保護，這二項因素會使得用多個執行緒的代價比用多個處理程序要小。但共用記憶體的彈性也是有代價的：如果資料會被多個執行緒存取，程式開發者就必須確保每個執行緒存取時看到的資料是一致的。這議題一直存在於執行緒間共享資料上，而用來避免這問題的工具和指引，涵蓋了本書所有內容，特別是在第 3、4、5 和第 8 章中。只要在寫程式時小心一些，這些問題也並不是不可避免的，但確實意味著對執行緒間的通訊，必須要投入相當大的周詳考慮。

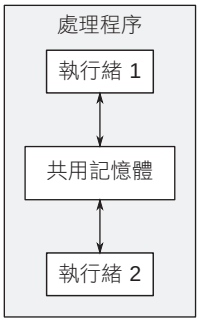

圖 1.4　單處理程序中二個同時執行的執行緒之間的通訊

與多個單執行緒處理程序相比，單一處理程序多個執行緒之間的啟動和通訊所需付出的代價較低，這意味著儘管存有共用記憶體的潛在問題，但這是包括 C++ 在內的主流程式語言對併發較為偏愛的方法。此外，C++ 標準也不會對處理程序之間的通訊提供任何內在支援，所以使用多個處理程序的程式必須依賴特定平台的應用程式介面完成。因此，本書將只探討用多執行緒實現併發功能，並在後續對併發的參考都假設是透過用多執行緒完成的。

在多執行緒程式中另一個常出現的名詞是「**平行**」，下一節將討論這二者間的差異。

1.1.3　併發與平行

在多執行緒程式中，併發和平行在含義上有很大的重疊程度。實際上，對許多人而言二者指的是同樣的事情，主要差異出現在細節、重點和意圖上，這兩個名詞都是以可用的硬體同時執行多項工作，但平行更注重性能方面。當提到**平行**主要關切的是以可用的硬體提高對大量資料處理時的性能，而提到**併發**則主要關切的是關注點的分離或反應。但這種二分法並不能將二者切割得那麼清楚，二者在意義上仍有相當程度的重疊，但至少可以協助說明二者間的區別。本書將會提供這二種多工的範例。

在澄清了併發和平行之後，現在繼續探討為什麼要在程式中使用併發。

1.2 為什麼要使用併發？

在程式中使用併發有兩個主要的原因：關注點分離和效率。實際上，我甚至會說這幾乎是使用併發的唯一原因：只要你夠努力地追根究底，就可以發現任何其他原因都可以歸結到這二者之一，甚至兩者兼有（好吧，除了「因為我想」之類的原因以外）。

1.2.1 為了分離關注點而使用併發

在寫軟體時將問題的關注點分開總是一個好的想法：透過將相關的程式組合在一起，不相關的程式分開，可以使程式更容易理解和測試，而且比較不容易出現錯誤。縱使這些不同區域的操作要同時進行，也可以用併發將不同的功能區域分開；在沒有明確的使用併發下，就只能撰寫工作切換框架的程式，或是在操作過程中主動呼叫不相關程式的區域。

考慮一個有使用者介面的密集處理應用程式，例如桌上型電腦的 DVD 播放器應用程式，此應用程式基本上有兩組主要功能，它不只需要從光碟片中讀取資料、將影像和聲音資料解碼，並即時將解碼後資料傳送到圖形和聲音處理硬體，以便 DVD 能無間斷地播放；另外它還必須能接收使用者的輸入，例如使用者按下「暫停」或「返回功能表」鈕，甚至「退片」鈕等。在單執行緒情況下，將 DVD 播放程式碼與使用者介面程式碼匯合再一起，因此程式在播放過程中必須定期檢查使用者是否有輸入。如果藉由使用多執行緒來分開這些關注的點，使用者介面程式碼和 DVD 播放程式碼就不再需要如此緊密地交織在一起，可以用一個執行緒處理使用者介面，而另一個處理 DVD 播放。它們之間當然也會產生如使用者按下「暫停」鈕時的交互作用，但現在這些交互作用會與手頭上的工作直接相關。

當使用者發出要求時，這要求會傳遞給負責處理使用者介面的執行緒，此執行緒通常可以立即對使用者要求做出回應，即使回應是顯示繁忙的游標圖示或「請等待」的訊息，都會產生一種立即反應的感覺。類似地，不同的獨立執行緒通常用來執行必須在後台連續運作的工作，例如桌面搜尋應用程式對檔案系統變化的監控，以這種方式使用執行緒，因為執行緒之間的交互作用可以限制在可清楚識別的點上，而不必在不同工作的處理邏輯間穿插，因此一般會使得每個執行緒中的處理邏輯更為簡單。

在這情形下，因為執行緒的劃分是基於設計概念，而不是企圖增加可以使用的數量，所以執行緒的數量和可以使用的 CPU 數目無關。

1.2.2　為性能使用併發：工作與資料的平行處理

多處理器系統已經存在數十年了，但它們之前大多只用於超級電腦、大型主機和大型伺服器系統中。目前晶片製造商逐漸走向在單晶片上設計 2、4、16 個等多個核心或多個處理器，而不只是提升單核心的性能。因此，多核心桌上型電腦或多核心嵌入式設備，現在也越來越普遍。這些電腦計算能力的提升並非來自執行單個工作的速度變快，而是來自於能平行地執行多個工作。過去程式開發者可以輕鬆坐著，不必費心改善他們的程式，程式自然會在新一代的處理器上跑得更快，但現在，正如 Herb Sutter 所說「免費午餐已經結束了」[1]，如果軟體要獲得更強的計算能力，就必須將程式設計為能同時執行多個工作。因此，那些到現在還忽視併發的程式設計者，勢必需要盡快設法將併發添加到他們的工具箱中。

要提高性能，有兩種使用併發的方法，第一種也是最明顯的是將單一工作分成幾個部分，然後平行地執行各個部分，以減少整體執行時間，這稱為**工作平行**。雖然聽起來很簡單，但因為各部分間可能存有許多互相依存關係，所以它可能會是一個相當複雜的程序，這種是針對處理程序分割，即一個執行緒執行演算法的一部分，另一個執行緒執行不同的部分。分割也可以對資料進行，這情況下每個執行緒執行相同的作用，但使用資料的不同部分，這種方法稱為**資料平行**。

容易受到這種平行性影響的演算法通常被稱為**令人尷尬的平行**演算法，儘管這意味著你可能會因為程式能如此容易平行而感到尷尬，但這仍是一件好事情；另外，這樣的演算法也被稱為**自然的平行**和**合宜地併發**。令人尷尬的平行演算法擁有良好的可擴充性，當可用的硬體執行緒數量增加，演算法中的平行性也可以隨之相對增加。這演算法完美的體現了「人多好辦事」這句諺語。對於演算法中不是令人尷尬的平行的部分，也許可以將演算法分割成固定的平行工作數（因此不能擴充）。在執行緒之間分割工作的技術將在第 8 章和第 10 章討論。

為提高性能使用併發的第二種方法是利用可用的平行性來解決更大的問題；可以的話一次處理 2、10、或 20 個檔案，而不是一次只處理一個。雖然這是**資料平行性**的應用，但藉由同時在多組資料上執行相同的操作，則會有不同的注意焦點。雖然仍需要花同樣的時間處理一大塊資料，但現在可以用相同的時間量處理更多資料。很明顯的，這種方式也有它的限制，也不是在所有

1　"The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," Herb Sutter, Dr. Dobb's Journal, 30(3), March 2005. http://www.gotw.ca/publications/concurrency-ddj.htm.

情況下都有益，但這種方式所增加的處理量的確可以帶來新的可能性——例如，如果影像的不同區域可以平行處理的話，透過這方式就能增加視訊處理的解析度。

1.2.3 不用併發的時機

知道不使用併發的時機與知道何時要用它一樣重要。基本上，不使用併發的唯一理由是當獲得的好處不如付出的代價的時候。在許多情況下併發的程式比較難理解，因此撰寫和維護多執行緒程式需要投入些直接的智力成本，而且程式所增加額外的複雜度也可能導致存在較多的錯誤。除非潛在對性能提升的好處夠大，或關注點可以分開得相當清楚，證明確保程式正確無誤所需的額外開發時間以及維護多執行緒程式相關的額外成本是值得的，否則不要使用併發功能。

也許性能提升得不如預期那麼好，因為作業系統必須分配相關核心的資源和堆疊空間，然後將新執行緒加到排程器中，而這些都需要花時間，所以啟動執行緒會需要一些代價。就算在執行緒上的工作能很快地完成，縮短工作所佔用的時間也可能會因啟動執行緒的代價而相形見絀，甚至可能使程式整體的性能比原單執行緒直接執行還要差。

此外，執行緒是有限資源，如果同時執行太多執行緒，會消耗作業系統過多資源，並使整個系統的運作變慢。不僅如此，因為每個執行緒都需要獨立的堆疊空間，所以使用過多的執行緒也可能耗盡處理程序可用的記憶體或位址空間。這對於具有平面結構的 32 位元處理程序來說，因為有 4GB 可用位址空間的限制，這問題更為嚴重；假設每個執行緒都需要使用 1MB 堆疊（在許多系統是這樣的），則位址空間將被 4,096 個執行緒用完，沒有任何空間留給程式碼、靜態資料或堆積資料。雖然 64 位元或更大的系統不容易受到這種直接位址空間的限制，但它們的資源仍然有限，如果執行過多的執行緒，最終仍會造成問題。雖然可以用執行緒池（請參閱第 9 章）限制執行緒數量，但執行緒池也不是靈丹妙藥，它們也有自己的問題要面對。

如果主 / 從系統的伺服端程式為每個連接啟動一個單獨的執行緒，對少量連接這將運作得很好，但如果相同技術是用在必須處理很多連接的高需求伺服器上，則因為會啟動過多的執行緒而很快的耗盡系統資源。在這情形下，小心謹慎地使用執行緒池也許可以提供最佳性能（請參閱第 9 章）。

最後，運作的執行緒越多，作業系統就必須進行越多的語意切換，而每個語意切換都會佔用可以處理有用工作的時間，所以有時候增加額外的執行緒反而會降低程式整體性能，而不是增加。因此，如果嘗試要讓系統達到最佳性能，就必須在考慮硬體可用的併發性（或沒有）下，調整運作的執行緒數量。

為提升性能而使用併發就像其他的最佳化策略一樣，它也許能大幅提高程式性能，但也可能使程式過於複雜難於理解，也更容易存有錯誤。因此，只有在具可衡量獲益可能性的程式中，才值得在這些關鍵性能部分執行併發。當然，如果性能提升的潛力與設計清晰度或關注點分離相比只是次要的話，那使用多執行緒設計仍然是值得的。

無論是為了性能、關注點分離，或只是因為是「多執行緒星期一」，假設你已經決定要在程式中使用併發，這對 C++ 程式開發者又意味著什麼呢？

1.3　C++ 中的併發和多執行緒

對 C++ 來說，透過多執行緒對併發的標準化支援是一個比較新的內容。只有從 C++11 標準開始，你才能夠在不需要求助於特定平台的擴展下，撰寫多執行緒程式。為了理解 C++ 標準執行緒庫中背後許多決定的基本原理，了解它的歷程很重要。

1.3.1　多執行緒在 C++ 中的歷程

1998 年發佈的 C++ 標準並不承認有執行緒，而各式語言元素的操作效果是依抽象機器順序撰寫，而且也沒正式定義記憶體模式，所以如果沒有將編譯器特定的執行緒功能擴充到 1998 年的 C++ 標準，現在就無法撰寫多執行緒程式。

因為編譯器供應商可以自由地在程式語言中添加一些擴充，而且不同的 C 應用程式介面對多執行緒的使用情況（像是 POSIX C 標準和 Microsoft Windows API 中的擴充），已經使得許多 C++ 編譯器供應商開始透過各式特定平台的擴充來支援多執行緒。這種編譯器的支援通常限制於只能使用與平台相對應的 C 應用程式介面，並確保 C++ 執行階段程式庫（如異常處理機制的程式）能在多個執行緒存在的情況下工作。雖然很少有編譯器供應商提供正式的多執行緒感知記憶體模式，但編譯器和處理器的行為就已經相當好，造成多執行緒 C++ 程式大量的出現。

C++ 程式開發者對只能使用特定平台的 C 應用程式介面處理多執行緒並不滿足，他們期待能有通用的類別庫提供物件導向的多執行緒功能。應用框架（如 MFC）和通用 C++ 函式庫（如 Boost 和 ACE）已經將一組含有基本特定平台應用程式介面的 C++ 類別包裹在一起，並提供多執行緒高階功能以簡化工作。雖然這些類別庫的細節差異很大，特別是在啟動新執行緒方面，但類別的整體型式卻有很多共同處。對許多 C++ 類別庫一個常見很重要的設計，是在上鎖情況下使用慣用的「資源獲取即初始化（RAII）」，以確保在離開相關範圍時互斥鎖會解鎖，這為程式開發者提供了相當大的好處。

在許多情況下，現有的 C++ 編譯器結合了特定平台的應用程式介面及與平台無關的類別庫（如 Boost 和 ACE），以支援多執行緒，提供撰寫 C++ 多執行緒程式堅實的基礎，因此可以寫出數百萬行 C++ 指令的多執行緒應用程式。但是，缺乏標準的支援意味著在某些情況下會因為缺少執行緒感知記憶體模式而造成問題，尤其是對於那些想要用處理器硬體的知識來獲得更高性能的人，或是那些撰寫編譯器行為會因平台而異的跨平台程式的人。

1.3.2 C++11 對併發的支援

所有這些都隨著 C++11 標準的發佈而改變，C++ 標準函式庫不僅有一個執行緒感知記憶模型，而且已經擴大到包含了用來管理執行緒的類別（請參閱第 2 章）、保護共享資料（請參閱第 3 章）、執行緒之間的同步操作（請參閱第 4 章）和低階原子操作（請參閱第 5 章）等。

C++11 執行緒庫主要以前面提到的 C++ 類別庫之前所累積的經驗為基礎，特別是將 Boost 執行緒庫當成新函式庫基礎的主要模型，新函式庫與 Boost 執行緒庫中許多類別有相同的名稱和結構。隨著標準的演進，這也一直是一種雙向的交流；為了能與 C++ 標準匹配，Boost 執行緒庫本身也在許多方面做了改變，因此對從 Boost 轉移過來的使用者應該會感到很熟悉。

如本章一開始提到的，併發支援是 C++11 標準的改變之一，對 C++ 功能做了很多增強，使程式開發者的工作更方便。雖然這些大多超出本書範圍，但其中一些改變對執行緒庫及使用方式會有直接影響。附錄 A 提供了這些特色的扼要介紹。

1.3.3　在 C++14 和 C++17 中對併發和平行更多的支援

C++14 對併發和平行增加的唯一具體支援，是為了保護共享資料的新型互斥鎖（請參閱第 3 章），而 C++17 則增加了比較多，包括為初學者提供一整套平行演算法（請參閱第 10 章）。這兩個標準都強化了標準庫的核心語言和其他部分，這些強化更簡化了多執行緒程式的開發。

如前所述，併發有一個技術規範，它描述了 C++ 標準提供的函式和類別擴充，特別是針對執行緒間的同步操作（請參閱第 4 章）。

直接在 C++ 中支援原子操作，使程式開發者不必再靠特定平台的組合語言，就能用已經定義過的語義撰寫高效能程式。這對於那些想要寫有效率、可攜式程式的人來說，真是一個恩賜；編譯器不只會照料平台的細節，還可以在考慮操作的語義下產生優化器，更好地優化整個程式。

1.3.4　C++ 執行緒庫的效率

高效率計算功能的開發人員對 C++ 語言新內含低階功能的類別（像那些在標準 C++ 執行緒庫中的），經常會浮現出對效率的關切。如果是致力於效能上，與直接使用基本低階功能相比，用高階功能實作的代價就很重要，這代價可視為是**抽象懲罰**。

C++ 標準委員會在設計 C++ 標準函式庫通體上及標準 C++ 執行緒庫細節上時，就已經意識到這一點，因此設計目標之一是，它所提供的相同功能，如果直接使用低階應用程式介面，所獲得的好處將會很少或根本沒有。因此，該函式庫被設計成可以在大多數主要平台上有效率的實作（抽象懲罰較低）。

C++ 標準委員會的另一個目標是，對那些為最終性能而希望在緊挨底層工作的人，確保 C++ 也提供了足夠的低階功能。為達成這目標，伴隨新的記憶體模式還帶來一個可以直接控制個別位元 / 位元組、執行緒間同步化、和任何改變可見化的全方位的原子操作函式庫，許多以前開發人員只能使用特定平台的組合語言的部分，現在透過這些原子類型和對應的操作都可以使用，使用新標準類型和操作的程式也更可攜和容易維護。

C++ 標準函式庫也提供高階抽象性和功能，使開發多執行緒程式更容易且不容易出錯。因為需要執行額外的程式，有時使用這些功能對性能會有影響，但這種對性能影響的代價並不一定表示有較高的抽象懲罰。一般來說這代價

不會超過自行開發相同功能所產生的代價，而且無論如何編譯器都會在內部連結許多額外的程式。

在某些情形下，高階功能提供超出特定使用所需的額外功能性，大多數時候這並不是問題：不使用就不需付費；但有極少數的情形，這種未使用的功能會影響其他程式的性能。如果重視的是性能而且影響太大時，那最好從低階自己動手製作所需的功能。在絕大多數情形下，這導致的額外複雜性和出錯機率會遠超過性能改善帶來的好處。即使由剖析已經證明瓶頸出在 C++ 標準函式庫的功能，但這也可能是來自差的程式設計，而不是函式庫實作不好。例如，如果太多執行緒在爭奪互斥鎖，這對性能影響將很顯著，與其嘗試從互斥鎖操作中找出稍許空閒時間，不如重組程式架構以減少在互斥鎖上的爭奪會更有益，至於如何設計減少爭奪的程式請參閱第 8 章內容。

在極少數情況下，C++ 標準函式庫沒有提供所需的性能或行為，這時就必須使用特定平台的功能。

1.3.5 特定平台的功能

雖然 C++ 執行緒庫對多執行緒和併發提供了幾乎全方位的功能，但任何平台都會有超出它所提供的特定平台功能。為了在不放棄使用標準 C++ 執行緒庫的優點下，能很方便地存取這些功能，C++ 執行緒庫中的類型可能會提供 `native_handle()` 成員函式，以直接使用特定平台的 API 進行基礎實作。從性質看，用 `native_handle()` 進行的任何操作都需完全依靠平台，已超出了本書範圍（以及標準 C++ 函式庫）。

在考慮使用特定平台的功能之前，應該先了解標準函式庫所提供的內容，因此讓我們以一個範例作為開始。

1.4 開始吧！

你已經擁有一個又好又閃耀的 C++11/C++14/C++17 編譯器，接下來呢？C++ 多執行緒程式長什麼樣子？它看起來就像其他 C++ 程式，通常由變數、類別和函式混合組成；唯一真正的區別是，某些函式可能是併發執行，因此如第 3 章所述，必須確保對併發存取共享資料是安全的。為了能同時執行一些函式，必須用某些特定的函式和物件來管理不同的執行緒。

1.4.1　你好，併發世界

現在從一個經典的範例開始：一個顯示「Hello World」的程式，下面顯示了在單執行緒執行的「Hello World」簡單程式，作為移到多執行緒時的基準：

```
#include <iostream>
int main()
{
    std::cout<<"Hello World\n";
}
```

這程式做的只是將「Hello World」寫到標準輸出串流，將它和下面程式列表中，以啟動另一個執行緒來顯示訊息的簡單「你好，併發世界」程式比較。

程式列表 1.1　簡單的「你好，併發世界」程式

```
#include <iostream>
#include <thread>
void hello()
{
    std::cout<<"Hello Concurrent World\n";
}
int main()
{
    std::thread t(hello);
    t.join();
}
```

第一個差別是額外的 `#include <thread>`，這在標準 C++ 函式庫對多執行緒支援聲明中是新的標頭，目的是宣告載入在 `<thread>` 中用來管理執行緒的函式和類別，而用於保護共享資料的函式和類別則在其他標頭中宣告。

其次，用於輸出訊息的程式已經移到另一個獨立函式，因為每個執行緒都必須有個**初始函式**，作為新執行緒開始執行的位置。對於程式原來的執行緒，初始函式為 `main()`，但其他執行緒會在 `std::thread` 物件的建構函式中指定，目前範例中 `std::thread` 物件 `t` 以 `hello()` 函式做為初始函式。

再下一個區別為，不是將訊息直接傳送到標準輸出或從 `main()` 呼叫 `hello()`，程式會啟動一個新的執行緒來做這件事，因此會有二個執行緒——由 `main()` 啟動的原執行緒和 `hello()` 啟動的新執行緒。

新執行緒啟動後，原來的執行緒會繼續執行，如果它不需要等待新執行緒完成，它會快樂地繼續執行到 `main()` 結束，並結束程式，這可能會在新執

行緒有機會執行之前發生，這就是為什麼要呼叫 `join()` 函式的原因，如第 2 章中描述的，這樣會叫原執行緒（在 `main()` 中）等待與 `std::thread` 物件相關聯的執行緒，也就是 `t`。

如果只是將訊息傳送到標準輸出而已，卻做了那麼多努力，如第 1.2.3 節所述，對這麼簡單的工作，通常並不值得使用這些多執行緒的努力，特別是如果原來的執行緒在此期間沒什麼事要做。本書稍後，將透過一些範例呈現使用多執行緒的明顯好處。

本章小結

本章討論了併發和多執行緒是什麼意思，以及為什麼會（或不會）選擇在程式中使用它，另外也討論了多執行緒在 C++ 的歷程，從 1998 年標準完全不支援開始，到各種特定平台的擴充，再到 C++11 標準對多執行緒的適當支援，繼續到 C++14 和 C++17 標準及併發技術規範。因為晶片製造商不再提高單個核心的執行速度，而大多選擇改以多核心的形式增強處理能力，允許能同時執行更多的工作，這些即時的支援讓程式開發者更能夠善用較新的 CPU 所提供更大的硬體併發性。。

在第 1.4 節範例中，也展示了簡單的使用 C++ 標準函式庫的類別和函式；在 C++ 中，使用多執行緒本身並不複雜，複雜性出自於設計程式讓它有預期的行為。

在第 1.4 節的範例之後，是該做一些更實質性事情的時候了；在第 2 章，將探討一些用於管理執行緒的類別和函式。