國立臺灣師範大學資訊工程學系

114 資訊專題研究（一）期中書面報告

運用 **Btrfs** 寫入時複製機制加速 **Rust** 建置快取之研究

A Study on Accelerating Rust Build Caching with Btrfs Copy-on-Write Mechanism

指導教授紀博文教授

學生鍾詠傑撰

中華民國 114 年 11 月

摘要

本研究旨在探討運用 Btrfs 檔案系統的寫入時複製（Copy-on-Write, CoW）特性，以加速 Rust 語言的編譯建置快取流程。Rust 專案的編譯時間過長是開發流程中的主要瓶頸，而現有的快取方案如 sccache 在本地端操作上仍有儲存與 I/O 效率的限制。本研究提出一個基於 Btrfs 原生功能（如 reflink 與快照）的快取框架，期望能透過檔案系統層級的優化，實現近乎即時的快取複製與還原，並大幅降低儲存空間佔用。研究將分析此方法的技術可行性，設計一個概念性的多層次快取架構，並規劃效能驗證藍圖，以評估其相較於傳統檔案系統及現有工具的優勢。此框架的核心概念是透過 Btrfs 快照，以原子性的方式將整個 target 目錄替換為一個預先存在的、已知的良好狀態，從而完美地保存 Rust 自身高度優化的增量編譯資料庫，這是像 sccache 這類編譯器包裝器快取工具無法實現的。

研究動機與研究問題

Rust 語言以其安全性與高效能獲得廣泛應用，但其複雜的編譯過程常導致開發者需耗費大量時間等待專案建置，尤其在大型專案中，此問題嚴重影響開發迭代效率。為解決此問題，社群開發了如 sccache 等外部快取工具，但這些工具多依賴網路共享或傳統檔案複製，不僅可能引入網路延遲，在本地端操作時也因大量 I/O 與儲存冗餘而有效能瓶頸。

Btrfs 檔案系統提供了原生的寫入時複製（CoW）功能，包含 reflink（檔案層級的 CoW 連結）與快照（子卷層級的即時備份）。當檔案系統中的資料被修改時，Btrfs 並不會直接覆寫原始的資料區塊，而是將修改後的內容寫入一個新的區塊，然後更新檔案系統的中繼資料以指向這個新位置。快照是一種特殊的子磁區，它在建立之初與其來源子磁區共享所有資料區塊，因此建立快照幾乎是一個瞬時完成的中繼資料操作。Docker 已在生產環境中成功運用 Btrfs 驅動程式，將映像檔分層對應到 Btrfs 子磁區，並將容器的可寫狀態對應到 Btrfs 快照，這為本研究提供了強而有力的技術先例。

我們觀察到，Rust 編譯過程產生的 target 目錄結構龐大且內容相似度高，特別是其增量編譯系統會將編譯過程模型化為一個由查詢組成的有向無環圖，並將這個相依性圖以及查詢結果持久化到 target/incremental 目錄內。這個複雜的資料庫對於實現快速的重新建置至關重要，但當開發者切換 Git 分支時，這些精細的增量快取往往會失效。這正符合 Btrfs CoW 機制的應用場景。

基於以上觀察，本研究希望回答以下核心研究問題：

（一）如何利用 Btrfs 的 reflink 與快照機制，設計一個高效能、低儲存佔用的 Rust 編譯快取框架？

（二）相較於在 ext4 等傳統檔案系統上使用標準檔案複製，Btrfs 的 CoW 操作在快取建置與還原速度上能帶來多大的效能提升？

（三）此框架應如何與 Rust 原生的增量編譯及 sccache 等外部工具整合，以形成一個更全面的多層次快取體系？

初步文獻探討

現有的 Rust 編譯快取方案主要分為三個層次。第一層是 Rust 編譯器原生的增量編譯機制，它採用一種「紅綠」(red-green) 演算法來追蹤相依性，並將中間結果快取於 target/incremental 目錄中。這是最精細的層級，能理解 Rust 語言的內部語義。第二層是 sccache 等編譯器包裝器工具，它透過攔截編譯指令，對編譯單元的所有輸入進行雜湊指紋運算來產生快取鍵，並支援本地磁碟或遠端儲存後端（如 S3、Redis）。sccache 運作於編譯器調用層級，將 rustc 視為黑盒子，對 Rust 內部的相依圖一無所知。

然而，這些方案在處理頻繁的情境切換（如 Git 分支切換）時仍有限制。當執行 git

checkout 切換分支時，原始碼檔案的時間戳會改變，導致增量編譯快取大量失效。sccache 雖可從遠端獲取編譯產物，但需要處理成千上萬個小檔案的網路 I/O，且無法保存增量編譯的內部狀態。

Btrfs 在處理大型檔案或目錄的複製與版本控制上具備獨特優勢。根據相關研究，Btrfs 快照是原子性的元數據操作，能在單一瞬間捕捉整個目錄樹的狀態，且初始不佔用額外物理空間。Docker 使用 Btrfs 驅動程式時，每個映像檔層被儲存為一個唯讀的 Btrfs 子磁區，當容器被建立時，Docker 會為該映像檔最終層建立一個可讀寫的快照，這個快照便成為容器的可寫層。當容器修改檔案時，Btrfs 的 CoW 機制被觸發，相關的資料區塊被複製到容器的快照中，而原始映像檔層則保持不變。

本研究提出將這種模式應用於 Rust 編譯快取，將「特定分支/提交的快取建置狀態」對應到「target 目錄的 Btrfs 快照」。這構成了一個第三層次的快取機制，運作於檔案系統區塊層級，能完美保存最精細層級的增量編譯快取，填補現有方案的空白。

初步研究方法

本研究提出一個名為「快照交換模型」（Snapshot Swap Model）的概念性快取框架。其核心設計是將 Rust 專案的 target 編譯輸出目錄存放於一個獨立的 Btrfs 子卷中，並透過快照管理其不同的建置狀態。

工作流程設計：專案的原始碼與 target 目錄必須位於同一個 Btrfs 檔案系統上。（1）基準快照建立：在主分支上成功完成一次全新建置後，為 target 子磁區建立一個唯讀快照，例如 `btrfs subvolume snapshot -r target target.main.clean`。（2）開發開始：當開發者切換到功能分支時，從最相關的基準快照建立一個新的可讀寫快照，例如 `btrfs subvolume snapshot target.main.clean target.feature-branch`。（3）增量建置：所有後續的 `cargo build` 指令都在這個可讀寫的快照中進行，完美地利用 Rust 原生的增量編譯功能。（4）情境切換：當執行 `git checkout main` 時，建置協調腳本將卸載或重新命名當前的 target.feature-branch，並將 target.main.clean 快照（或其可讀寫複本）掛載為活躍的 target。這個操作是近乎瞬時的，下一次在 main 分支上執行的編譯將會是一個「空建置」，耗時極短。

多層次快取整合：此框架形成一個互補的多層次快取體系。第 0 層為 Btrfs 快照模型，提供對整個建置目錄的完美狀態保真度，以及近乎零成本的還原時間。第 1 層為位於 target/incremental 內的 Rust 原生增量快取，第 0 層 Btrfs 快取的主要效益正在於跨情境切換時能夠完整地保護第 1 層快取的完整性。第 2 層為 sccache，對於從相依性套件產生新的編譯產物，或在不共享 Btrfs 磁區的專案間共享產物，sccache 仍然具有價值。

概念驗證計畫：為驗證此方法的可行性，我們將設計一個概念驗證腳本，在一個中等規模的 Rust 專案上進行基準測試。環境設定包括：準備一台裝有近期 Linux 核心的虛擬機，建立專用區塊裝置並使用 `mkfs.btrfs` 格式化，為專案建立頂層子磁區，並在其內為 target 目錄建立專用子磁區。協調腳本將使用 Bash 或 Python 開發，包含快照建立、啟用、修剪等函數，並可由 git hooks 觸發或手動執行。

測試將比較在 Btrfs 與 ext4 兩種檔案系統上，執行「清除、切換分支、重新編譯」等典型開發場景所需的時間。關鍵評估指標包含：（1）主要指標為在兩個差異顯著的分支之間執行 git checkout 後的重新建置時間，成功標準為相較於溫快取的 sccache 減少至少 50%，相較於冷快取減少至少 80%。（2）次要指標包括快照建立與還原時間、經過 20 次以上循環後的磁碟空間消耗，以及快照管理操作本身所花費的時間。測試將使用 hyperfine 等工具進行嚴謹的基準測試，並使用 `btrfs filesystem usage` 監控磁碟空間使用情況。

此研究呈現了一個高風險、高回報的機會。潛在利益包括近乎瞬時的情境切換、完美的快取保真度，以及減少網路 I/O。潛在風險則包括檔案系統鎖定、協調的複雜性、儲存開銷，以及 Btrfs 在某些 I/O 模式下可能表現不佳的效能風險。概念驗證的結果將成為決定是否進行更廣泛、生產級別實作的唯一依據。