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 專案的編譯時間是開發流程中的主要瓶頸，而現有快取方案在處理頻繁的分支切換時效率有限。本研究提出一個基於 Btrfs 快照（snapshot）的快取框架，期望透過檔案系統層級的操作，實現近乎即時的快取還原，並降低儲存空間佔用。研究將分析此方法的技術可行性，設計一個概念性的快取架構，並規劃效能驗證藍圖。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 目錄切換至一個已知狀態，從而保存 Rust 的增量編譯資料庫。相較之下，sccache 等編譯器層級的工具無法對整個建置目錄進行原子性快照還原。

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，嚴重影響開發迭代效率。為解決此問題，Rust 編譯器內建了增量編譯機制 [**?**]，而社群也開發了如 sccache [**?**] 等外部快取工具。然而，這些方案在處理頻繁的程式碼分支切換時仍存在瓶頸。特別是當開發者執行 git checkout 切換分支時，原始碼檔案的時間戳會改變，常導致 Rust 原生的增量編譯快取大量失效，引發不必要的重新編譯。

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）。Docker 已在生產環境中運用 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 [**?**]，這為本研究提供了技術上的參考。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，特別是其增量編譯資料庫對於實現快速重新建置至關重要 [**?**]。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：

1. 如何利用 Btrfs 的快照機制，設計一個高效能、低儲存佔用的 Rust 編譯快取框架？
2. 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？
3. 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不同層次。第一層是 Rust 編譯器原生的增量編譯機制，它採用精密的演算法來追蹤相依性，並將中間結果快取於 target/incremental 目錄中 [**?**]。第二層是 sccache 等編譯器包裝器工具，它透過攔截編譯指令，對編譯單元的輸入進行雜湊運算來產生快取鍵，並支援本地或遠端儲存 [**?**]。

這兩種快取機制運作於不同的抽象層級：Rust 增量編譯最為精細，理解語言內部語義與相依圖；sccache 運作於編譯器調用層級，將 rustc 視為黑盒子。關鍵差異在於，當開發者切換 Git 分支時，第一層增量快取會因時間戳改變而失效，第二層 sccache 雖可快速提供編譯產物，但無法保存增量快取內部狀態。本研究探討一個「第零層」的快取機制，以檔案系統快照保存整個 target 目錄的狀態，從而完整保護第一層快取在跨分支切換時的完整性。

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。(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，作為當前活躍的 target 目錄。(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。(4) 情境切換：當執行 git checkout 時，透過腳本將活躍的 target 目錄原子性地切換回對應分支的快照。

概念驗證計畫：為驗證此方法，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。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。效能評估將基於以下指標：(1) 主要指標：重新建置時間。測量在兩個具備顯著差異的分支之間執行 git checkout 後，重新編譯專案所需的時間。預期此方法相較於傳統快取方案，在時間上能有顯著縮短。(2) 次要指標：記錄快照建立與還原操作的耗時、多次建置循環後的磁碟空間使用率，以及快照管理腳本本身的效能開銷。

本研究所提出的方法，其預期效益在於縮短開發情境切換所需的時間、確保快取狀態的一致性，並減少本地開發的 I/O 開銷。然而，此方法亦存在待評估的挑戰，包括對特定檔案系統（Btrfs）的依賴性、快照管理腳本的實作複雜度、長期使用下可能的儲存空間佔用，以及 Btrfs 在特定 I/O 負載下的效能表現 [**?**]。本研究的概念驗證結果，將作為評估此方法實用性的依據。本研究的適用範圍目前限於採用 Btrfs 檔案系統的 Linux 開發環境。

References

[1] Docker Documentation. *BTRFS storage driver*. [Online]. Available: https://docs.docker.com/engine/storage/drivers/btrfs-driver/

[2] The Rust Programming Language Blog. (2016). *Incremental Compilation*. [Online]. Available: https://blog.rust-lang.org/2016/09/08/incremental.html

[3] Rust Compiler Development Guide. *Incremental compilation in detail*. [Online]. Available: https://rustc-dev-guide.rust-lang.org/queries/incremental-compilation-in-detail.html

[4] Mozilla. *mozilla/sccache*. [Online]. GitHub. Available: https://github.com/mozilla/sccache

[5] Phoronix. (2024). *Bcachefs, Btrfs, EXT4, F2FS & XFS File-System Performance On Linux 6.15*. [Online]. Available: https://www.phoronix.com/review/linux-615-filesystems