Red Hat Enterprise Linux 6

效能微調指南

優化 Red Hat Enterprise Linux 6 的子系統處理能力

版 4.0

Last Updated: 2017-10-12

Red Hat Enterprise Linux 6 效能微調指南

優化 Red Hat Enterprise Linux 6 的子系統處理能力
版 4.0

Red Hat 主題專員

編輯者

Don Domingo

Laura Bailey

法律聲明

Copyright © 2011 Red Hat, Inc. and others.

This document is licensed by Red Hat under the Creative Commons Attribution-ShareAlike 3.0 Unported License. If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

摘要

《效能微調指南》詳述了如何優化 Red Hat Enterprise Linux 6 系統，同時也記載 Red Hat Enterprise Linux 6 中與效能相關的升級項目。 儘管本指南中的程序皆已經過測試與確認，建議您將計劃的配置套用在生產環境之前，先在測試環境中詳細測試所有配置。您也應備份您的所有資料，以及進行微調前的配置。

內容目錄

章 1. 總覽 4
1.1. 讀者 4
1.2. 水平擴充能力 5
1.2.1. 平行運算 5
1.3. 分散式系統 5
1.3.1. 通訊 6
1.3.2. 儲存環境 7
1.3.3. 聚集網路 8

章 2. RED HAT ENTERPRISE LINUX 6 效能提升 10
2.1. 支援 64 位元 10
2.2. 票證盤旋鎖 10
2.3. 動態清單結構 11
2.4. 無計時 KERNEL 11
2.5. 控制群組 11
2.6. 改進儲存與檔案系統 12

章 3. 監控和分析系統效能 15
3.1. PROC 檔案系統 15
3.2. GNOME 與 KDE 系統監控程式 15
3.3. 內建的命令列監控工具 16
3.4. TUNED 與 KTUNE 17
3.5. 應用程式設定檔工具 18
3.5.1. SystemTap 18
3.5.2. OProfile 18
3.5.3. Valgrind 19
3.5.4. Perf 19
3.6. RED HAT ENTERPRISE MRG 20

章 4. CPU 21
拓樸 21
執行續 21
插斷 21
4.1. CPU 拓樸 21
4.1.1. CPU 與 NUMA 拓樸 21
4.1.2. 微調 CPU 效能 22
4.1.2.1. 使用 taskset 設定 CPU 關聯 24
4.1.2.2. 使用 numactl 控制 NUMA 政策 24
4.1.3. numastat 26
4.1.4. NUMA 關聯管理 daemon（numad） 27
4.1.4.1. numad 的好處 27
4.1.4.2. 操作模式 28
4.1.4.2.1. 使用 numad 作為服務 28
4.1.4.2.2. 以執行檔方式執行 numad 28
4.2. CPU 排程 29
4.2.1. 即時排程政策 29
4.2.2. 正常的排程政策 30
4.2.3. 選擇政策 30
4.3. 微調插斷與 IRQ 30
4.4. RHEL 6 中，NUMA 的加強功能 31
4.4.1. 優化空機系統與可擴充能力 32
4.4.1.1. 感知拓樸的加強功能 32

4.4.1.2. 多處理器同步的加強功能 32
4.4.2. 優化虛擬功能 32

章 5. 記憶體 34
5.1. 巨型轉譯後備緩衝區 34
5.2. 巨型分頁與通透式巨型分頁 34
5.3. 使用 VALGRIND 側寫記憶體使用量 35
5.3.1. 使用 memcheck 側寫記憶體使用量 35
5.3.2. 以 Cachegrind 側寫快取使用量 36
5.3.3. 使用 Massif 側寫堆積與堆疊空間 37
5.4. 微調處理能力 38
5.5. 微調虛擬記憶體 40

章 6. 輸入/輸出 43
6.1. 功能 43
6.2. 分析 43
6.3. 工具 44
6.4. 配置 47
6.4.1. 完全公平佇列（CFQ） 48
6.4.2. 期限 I/O 排程器 49
6.4.3. Noop 50

章 7. 檔案系統 52
7.1. 檔案系統微調考量 52
7.1.1. 格式化選項 52
7.1.2. 掛載選項 52
7.1.3. 檔案系統維護 53
7.1.4. 應用程式考量 54
7.2. 檔案系統效能的設定檔 54
7.3. 檔案系統 54
7.3.1. ext4 檔案系統 55
7.3.2. XFS 檔案系統 55
7.3.2.1. XFS 的基本微調 56
7.3.2.2. XFS 進階微調 56
7.4. 叢集 58
7.4.1. 全域檔案系統 2 58

章 8. 網路 60
8.1. 網路效能提升 60
8.1.1. 接收封包操控（Receive Packet Steering，RPS） 60
8.1.2. 接收流量操控（Receive Flow Steering，RFS） 60
8.1.3. TCP 精簡串流的 getsockopt 支援 60
8.1.4. 支援通透式代理 61
8.2. 優化網路設定 61
8.2.1. Socket 接收緩衝區大小 62
8.3. 封包接收總覽 63
8.3.1. CPU/快取關聯 63
8.4. 解決常見的佇列/FRAME 遺失問題 63
8.4.1. NIC 硬體緩衝區 64
8.4.2. socket 佇列 64
8.5. MULTICAST 考量 65

附錄 A. 修訂記錄 66

章 1. 總覽

《*效能微調指南*》乃配置與優化 Red Hat Enterprise Linux（Red Hat 企業版 Linux，以下簡稱「RHEL」）的全方位參考文件。儘管本發行版亦包含關於 RHEL 5 效能功能上的資訊，在此提供的皆為 RHEL 6 專屬的指示。

本書分成了幾個詳述 RHEL 特定子系統的章節。《*效能微調指南*》專注於各個子系統的三個重點：

功能

各個子系統的章節皆詳述了 RHEL 6 專屬（或以不同方式實作於 RHEL 6 中）的效能功能。這些章節亦討論關於 RHEL 6 的更新，這些更新已大幅改善了 RHEL 5 的特定子系統。

分析

本書亦列舉了各個子系統的效能指標。這些指標的典型數值將以特定服務的角度來詳述，以協助您理解這些數值在真實世界的生產系統上，會帶來什麼樣的影響。

此外，《*效能微調指南*》亦顯示了擷取子系統效能資料的各種方式（也就是分析）。請注意，在此提到的分析工具另外詳述於其它文件中。

配置

本書中最重要的資訊也許是如何調整 RHEL 6 特定子系統的相關指示。《*效能微調指南*》解釋了如何為特定服務優化 RHEL 6 的子系統。

請注意，微調一個子系統的效能，有時可能會對其它子系統的效能有著負面影響。RHEL 6 的預設配置對於「*大部份*」「*中度*」負載的服務來說已處於最佳狀態。

列舉在《*效能微調指南*》中的程序已經過 Red Hat 工程人員（於測試與生產環境中）的詳細測試。然而，Red Hat 建議您將計劃的配置套用在生產環境之前，先在測試環境中詳細測試所有的配置。在微調您的系統之前，您也應備份所有資料和配置資訊。

1.1. 讀者

本書適合兩種類型的讀者：

系統分析師 / 企業分析師

本書以高階的方式列舉和解釋了與 RHEL 6 效能有關之功能，並提供了足夠的資訊，以詳述子系統在（就預設值和經過優化後）進行特定工作時的效能。用來詳述 RHEL 6 效能功能的詳細程度，足以協助潛在客戶與銷售工程人員理解此平台是否適用於提供高資源需求的服務。

《*效能微調指南*》亦儘可能擴充性各項功能提供了更詳細的文件連結。如此程度的詳細資訊，足以讓讀者理解這些效能功能，以構成高階的策略方案，建置和優化 RHEL 6。這能讓讀者「*同時*」開發、並評估基礎結構的計劃。

此文件的精細程度專注於「功能」上，適合擁有高度 Linux 子系統和企業級網路知識的讀者。

系統管理員

列舉在本書中的相關程序適合持有 RHCE [1] 證照的系統管理員（或具備相等經驗及技術，也就是擁有 3-5 年建置和管理 Linux 的經驗者）閱讀。《*效能微調指南*》儘可能的提供了有關於各項配置所帶來的影響之詳細資訊；即代表我們也詳述了所有可能會遇上的效能利弊情況。

效能微調所需的技術本質，不在於知道如何分析和微調子系統，而在於負責進行效能微調的系統管理員，必須知道如何「*根據特定用途*」，平衡並優化一部 RHEL 6 系統。這代表管理員「*也必須*」知道在嘗試一項配置，以提升某個子系統的效能時，所造成的效能利弊是否能被接受。

1.2. 水平擴充能力

Red Hat 致力於改進 RHEL 6 的效能，並將焦點放在「*水平擴充能力*」（scalability）上。評量效能加速的功能主要根基於，各種工作負載是如何影響平台的效能：也就是說，從單一網頁伺服器到伺服器叢集架構，都是改進的目標。

專注在擴充能力上，可讓 RHEL 在各種負載、目的上，均保有優勢。同時，這表示客戶的企業成長、工作負載增加時，重新配置伺服器環境不會是問題（就成本與人力而言），也更直接。

Red Hat 在「*水平擴充能力*」（horizontal scalability）與「*垂直擴充能力*」（vertical scalability）上皆大有改善；然而，水平擴充能力更常見。水平擴充的精神，是使用多台「*標準電腦*」（standard computer）分散工作負荷，以提升效能及可靠性。

在典型的伺服器叢集中，這些標準電腦多半是 1U 的機台型伺服器與刀鋒伺服器。雖說每台標準電腦可能是簡單的雙 CPU 插槽電腦，但有些伺服器叢集使用的是擁有更多 CPU 插槽的大型系統。一些企業級的網路會混合大型與小型系統；在這種情況下，大型系統是高效能的伺服器組（例如資料庫伺服器組），而小型系統是專職的應用程式伺服器（例如網頁或郵件伺服器）。

這種擴充能力簡化了 IT 架構的成長：中型企業可能只需要兩台機台伺服器，就能符合需求。當企業雇用更多人、擴展業務、增加銷售量等等，IT 需求的量與複雜度都會增加。水平擴充能力能讓 IT 簡化建置其它電腦的過程，只要使用（大致上）相同的配置即可。

簡單來說，水平擴充能力加了萃取層，簡化了管理硬體之事。將 RHEL 平台發展為能水平擴充的系統後，增加 IT 服務的容量與效能就跟加入新的、容易配置的電腦一樣容易。

1.2.1. 平行運算

使用者可以得益於 RHEL 的水平擴充能力，不僅僅是因為這擴充能力簡化了管理系統硬體的工作，還因為它適合現代先進硬體的發展哲學與趨勢。

想想以下情況：大部分複雜的企業應用程式在同一個時間，都有著成千上萬個任務執行，任務間又有著不同的協同方式。早期的電腦都只有單一核心的處理器來執行這些任務，而現代處理器大多擁有多核心。因此，現代的電腦會將多核心放在同一個 CPU 插槽上，讓單插槽的桌上型電腦或筆記型電腦，都擁有多處理器系統。

從 2010 年開始，標準的 Intel 與 AMD 處理器都有著 2 到 16 組 CPU 核心。這些處理器在機台或刀鋒伺服器上非常流行，可以包含到多達 40 組核心。這些低成本、高效能的系統能將大型系統的能力與特性引入企業中。

要達成最佳效能並善用系統，每個核心都應該處於忙碌狀態中。這表示要善用 32 核心的刀鋒伺服器，就應該有 32 個任務同時分頭進行。如果刀鋒伺服器的機台含有 10 台 32 核心的伺服器，那麼整組伺服器就可以同步處理 320 個任務。如果這些任務都屬於同一個工作，那麼就需要協同運作。

RHEL 的設計能妥善利用這些硬體趨勢，並確保企業能因此而得益。〈節 1.3, “分散式系統”〉列出了這些技術，詳述 RHEL 的水平擴充能力。

1.3. 分散式系統

要完全實現水平擴充能力，RHEL 使用了許多「*分散式運算*」（distributed computing）的元件。組成分散式運算的科技共分成以下三層：

通訊

水平擴充能力需要同步處理許多任務。因此，這些任務必須有「*程序間的通訊能力*」（interprocess communication）來協同運作。更進一步來說，擁有水平擴充能力的平台應該能與其它系統共享任務。

儲存裝置

單靠本機儲存裝置並不足以解決水平擴充能力的需要。這需要一些分散式或共享式的儲存裝置，有著一層萃取層讓單一儲存卷冊的容量，可以透過任意增加儲存裝置而加大空間。

管理

分散式運算的最重要任務是「*管理*」。管理層會協調所有軟硬體元件，有效地管理通訊、儲存以及共享資源。

以下章節將詳細討論這每一種技術。

1.3.1. 通訊

通訊層能確保資料傳輸，它包含了兩個部分：

- 硬體
- 軟體

讓多台系統通訊的最簡單（也最快速）的方式，是透過「*共享記憶體*」（shared memory）通訊。這繼承了大家熟悉的記憶體讀、寫運作；跟傳統的記憶體讀寫運作比起來，共享記憶體擁有高頻寬、低延遲、以及低負載的好處。

乙太網路

電腦間最常見的通訊方式是透過乙太網路。時至今日，系統的預設網路為「*GbE*」（十億位元乙太網路，Gigabit Ethernet），且大部分伺服器都有二到四個 GbE 連接埠。GbE 的頻寬大、延遲低，是現代大部分分散式系統的基礎。就算系統擁有更快的網路硬體，使用 GbE 作為管理介面依舊是非常常見的作法。

10GbE

高階伺服器、甚至是中階伺服器，正在快速採用「*10GbE*」（百億位元乙太網路）技術。10GbE 的速度是 GbE 的十倍。它最大的好處是用於現代的多核心處理器時，可讓通訊與運算達到完美的平衡。您可以比較使用 GbE 的單核心系統、以及使用八核心的 10GbE 系統。後者在維護系統整體效能與避免通訊瓶頸上，特別能彰顯其價值。

可惜的是，10GbE 依舊很昂貴。10GbE 的網路卡價格已經降下來，但其間的連接（特別是光纖）依舊很貴，而 10GbE 的網路交換器更是極端昂貴。我們可以預期隨著時間推演，價格會愈來愈低，但目前 10GbE 依舊廣泛用於機房的骨幹、以及對效能要求嚴苛的應用程式上。

Infiniband

Infiniband 的速度甚至比 10GbE 更快。除了 TCP/IP 與 UDP 網路透過乙太網路連接以外，Infiniband 也支援共享記憶體的通訊。這能讓 Infiniband 透過「*RDMA*」（遠端直接記憶體存取，remote direct memory access）與系統一起運作。

使用 RDMA 能讓 Infiniband 將資料直接在系統之間移動，而不需要使用 TCP/IP 或 socket 來連線。也因此能降低延遲時間，這對一些應用程式來說至關重要。

Infiniband 常見於「*HPTC*」（高效能科技運算，High Performance Technical Computing）的應用程式上，這種應用程式需要高頻寬、低延遲、以及低負載的環境。許多超級電腦上的應用程式都得益於此，跟

花錢在更快的處理器、更多記憶體比起來，投資 Infiniband 是改善效能的最佳方式。

RoCCE

「*RoCCE*」（使用乙太網路的 RDMA，RDMA over Ethernet）將 Infiniband 類型的通訊（包括 RDMA）在 10GbE 架構上實作出來。隨著 10GbE 產品愈來愈普遍、價格愈來愈低，可以預期許多系統與應用程式都會更廣泛地運用 RDMA 與 RoCCE。

Red Hat 支援以上這些通訊方法，並用在 RHEL 6 中。

1.3.2. 儲存環境

使用分散式運算，搭配多種共享儲存裝置的環境。這有兩種含意：

- 多台系統將資料儲存在單一位置上
- 單一儲存單元（例如卷冊）是由多個儲存裝置所組成

最為人所熟知的儲存範例，是系統上掛載的本機磁碟。這對所有應用程式都位於一台主機上，或是少少幾台主機上的 IT 運作環境來說，極為恰當。然而，因為 IT 架構已擴展為數十甚至成百上千台，管理眾多本機磁碟變成困難而複雜的工作。

分散式儲存裝置新增了一個階層，讓企業成長時依舊能輕易、自動地管理儲存裝置。讓多台系統共享幾個儲存裝置，可以降低所需管理的裝置數目。

將多個儲存裝置的儲存能力集合成一個卷冊，能同時幫助使用者與管理者。這種分散式儲存環境為儲存集區提供了萃取層，管理者可以輕易地增加硬體。一些分散式儲存技術也提供了更多益處，例如容錯能力與多路徑等等。

NFS

NFS（網路檔案系統，Network File System）能讓多台伺服器或使用者，透過 TCP 或 UDP 掛載、使用同樣的遠端儲存裝置。NFS 廣泛用於儲存多種應用程式共享的資料。對於儲存大量資料來說，NFS 也非常方便。

SAN

SAN（儲存區域網路，Storage Area Networks）使用光纖通道、或 iSCSI 通訊協定，提供遠端存取儲存裝置的能力。光纖通道架構（例如光纖主匯流排介面卡、交換器、以及儲存陣列）結合了高效能、高頻寬、以及大量的儲存空間。SAN 能將儲存裝置與處理方式隔開，讓設計系統時更有彈性。

SAN 的其它主要好處在於提供硬體管理任務的管理環境。這些任務包括：

- 控制對儲存裝置的存取
- 管理大量資料
- 佈建系統
- 備份、複製資料
- 建立快照
- 支援系統備援
- 確保資料的整合性
- 遷移資料

GFS2

Red Hat 的「*GFS2*」（Global File System 2，全球檔案系統 2）檔案系統提供了多種特殊功能。GFS2 的基本功能是提供單一檔案系統，包括同步讀寫、在叢集間的多個成員間共享等。這表示叢集的每個成員都會看到 GFS2 檔案系統「上面」的同樣資料。

GFS2 允許所有系統同步存取「磁碟」。要維持資料的完整性，GFS2 使用了「*DLM*」（分散式鎖定管理員，Distributed Lock Manager）機制，讓同一時間只有一台系統可以寫入特定位置。

GFS2 特別適合容錯的應用程式，這種應用程式需要儲存裝置的高可用性。

欲取得更多有關於各項 GFS2 效能微調的詳細資訊，請參閱《*全域檔案系統 2*》指南。欲知儲存裝置的一般資訊，請參閱《*儲存管理指南*》。這兩本指南都可以在以下網址找到：https://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/?locale=zh-TW。

1.3.3. 聚集網路

網路上的通訊多半透過乙太網路完成，搭配專職的光纖通道 SAN 環境存取儲存裝置。使用專職的網路或序列連線來進行系統管理工作，甚至使用「*heartbeat*」[2]，都是很常見的作法。因此，單一伺服器多半會位於多重網路上。

鑑於每台伺服器上的多重連線都很昂貴、巨大、且複雜，難以管理，將所有連線整合為一的呼聲四起，「*FCoE*」（乙太網路光纖通道，Fibre Channel over Ethernet）與「*iSCSI*」（網際網路 SCSI，Internet SCSI）便應運而生。

FCoE

有了 FCoE，標準的光纖通道指令與資料封包便可以透過單一的「*CNA*」（聚集網路卡，converged network card），在 10GbE 實體架構上傳輸。標準的 TCP/IP 乙太網路交通與光纖通道儲存的操作，都可以透過同樣連線來傳輸。透過一張實體網路卡（以及一條網路線），FCoE 就可以讓多個邏輯網路與儲存裝置連線。

FCoE 有以下好處：

降低連線數量

FCoE 會降低連線至伺服器的網路數量達百分之五十。使用者還是可以因為效能與可用性考量，選擇多重連線；然而，單一連線即可提供儲存與網路連線，這在機台伺服器與刀鋒伺服器上特別有用，因為這些伺服器的元件空間有限。

降低成本

降低連線數量也意味著降低網路線、交換器、以及其它網路配備的數量。乙太網路的沿革與規模經濟有關，乙太網路裝置大量導入市場，讓網路成本大幅降低：這在 100 Mb 與 1 Gb 的乙太網路裝置上，都曾發生過。

同樣地，10 GbE 也會隨著更多企業導入、採用而變得更便宜。同樣，CNA 硬體已經整合為單一晶片，為大眾所採用也會增加市場上的量，假以時日價格便會大幅降低。

iSCSI

iSCSI 是另一種聚集網路的通訊協定；它是 FCoE 的替代方案。就跟光纖通道一樣，iSCSI 提供了區塊等級的網路儲存裝置。然而，iSCSI 並不提供完整的管理環境。iSCSI 優於 FCoE 的主要好處，是 iSCSI 有著光纖通道的能力與彈性，價格卻較低廉。

[1] Red Hat Certified Engineer。欲取得更多資訊，請參閱〈http://www.redhat.com/training/certifications/rhce/〉。

[2] 「*Heartbeat*」（心跳）是在系統間交換資訊，以確保每台系統都運作無誤的方法。如果系統「停止心跳」，即可假設系統已經失效或關機，此時就該讓另一台系統接手工作。

章 2. RED HAT ENTERPRISE LINUX 6 效能提升

2.1. 支援 64 位元

RHEL 6 支援 64 位元的處理器；理論上這些處理器可使用達 16 *EB*（10^18 位元組）的記憶體。由 RHEL 6 的發行日起，此作業系統已經過測試並認證支援達 8TB 的實體記憶體。

RHEL 6 所支援的記憶體數量，將隨著作業系統的更新持續增加，Red Hat 也將持續提供且改善更多功能，以支援使用更大的記憶體區塊。此類型的改善（由 RHEL 6 發行日起）諸如：

- 巨型分頁和通透式巨型分頁
- 改善非對稱式記憶體存取

這些改善的詳情列在以下章節裡。

巨型分頁和通透式巨型分頁

實作於 RHEL 6 中的「*巨型分頁*」（huge page）能讓系統有效率地管理記憶體負載和使用量。巨型分頁會動態使用 2 MB 的分頁（標準原為 4 KB 的分頁）大小，並讓應用程式能夠有效利用數 GB，甚至是數 TB 的記憶體。

巨型分頁無法輕易建立、管理和使用。為了解決此問題，RHEL 6 亦包含了使用「*通透式巨型分頁*」（THP，transparent huge page）功能。THP 能自動管理巨型分頁使用上的許多複雜項目。

欲取得更多有關於巨型分頁與 THP 上的相關資訊，請參閱〈節 5.2, "巨型分頁與通透式巨型分頁"〉。

擴充性 NUMA 所進行的改善

現在許多新的系統皆支援「*NUMA*」（Non-Uniform Memory Access，非對稱式記憶體存取）。NUMA 可簡化設計、建立較大系統上的硬體之過程；然而，這同時也在應用程式開發上，添加了一層複雜性。比方說，NUMA 會實作本機與遠端的記憶體，而存取遠端記憶體的所需時間，可能會比存取本機記憶體的所需時間長上數倍。這項功能（以及其它功能）將會為效能帶來許多不同影響，它會影響作業系統，以及想要建置的應用程式及系統配置。

基於數項能在 NUMA 系統上協助管理使用者與應用程式的新功能，RHEL 6 現在已能透過更加優化的方式善用 NUMA。這些功能包含了「CPU affinity」（與 CPU 建立關聯）、「CPU pinning」（cpuset；釘選到 CPU）、numactl 以及控制群組（cgroup），這能讓一項程序（透過與 CPU 建立關聯）或是應用程式（透過釘選到 CPU）綁定至一個或是一組 CPU。

欲取得更多有關於 RHEL 6 中的 NUMA 支援相關資訊，請參閱〈節 4.1.1, "CPU 與 NUMA 拓樸"〉。

2.2. 票證盤旋鎖

票證盤旋鎖（ticket spinlock）是任何系統中的重要機制，確保程序不會修改另一組程序所使用的記憶體。在記憶體中，資料改變要是不受控制，會導致資料損毀、系統當機。要避免這些情況發生，作業系統會允許一組程序鎖定部分記憶體，進行操作，然後將此記憶體解鎖，或稱「釋放」（free）。

常見的記憶體鎖定方式，是透過「*盤旋鎖*」（spin lock），這能讓程序持續檢查，看看是否有鎖可以使用，若有的話便立即取用。如果系統上有多組程序競逐同樣的鎖，則最先要求鎖的程序會最先拿到。在所有程序對記憶體都擁有同樣存取權限時，這方法堪稱「公允」，且運作良好。

不幸的是，在 NUMA 系統上，並不是所有程序都能平等地存取鎖。在同樣 NUMA 節點上的程序在取得鎖時，會有不公平的劣勢。在遠端 NUMA 節點上的程序，會遭遇拿不到鎖與效能降低的問題。

要解決這個問題，RHEL 採用了「*票證盤旋鎖*」（ticket spinlock）功能。這功能會在鎖上加入保留訂位的佇列機制，讓「*所有*」程序都能依照提出需求的順序取得鎖。這能降低要求鎖的時候之時間問題與不公平的缺點。

因為票證盤旋鎖的負載比一般的盤旋鎖要來得重一些，因此在 NUMA 系統上有更多擴充性優勢，效能更佳。

2.3. 動態清單結構

作業系統需要系統上每個處理器的一組資訊。在 RHEL 5 中，這組資訊會分配到固定大小的記憶體陣列。每個處理器的資訊都透過對此陣列的索引而取得。對於處理器不多的系統來說，這方法快速、簡單、且直接。

然而，隨著系統中的記憶體愈來愈多，這方法的負擔就顯著加重。因為固定大小的記憶體陣列是單一、共享的資源，所以當更多處理器同時存取時，就會變成瓶頸。

為了解決這問題，RHEL 6 使用了「*動態清單結構*」（dynamic list structure）來儲存處理器資訊。這允許處理器資訊所使用的陣列會以動態方式分配：如果系統上只有八個處理器，那麼清單中就只會建立八個項目。如果有 2,048 個處理器，那麼就會建立 2,048 個項目。

動態清單結構也提供了更細緻的鎖定功能。舉例來說，如果處理器 6、72、183、657、931、與 1546 需要同時更新資訊，動態清單結構便可提供更高的同步處理功能。跟小型系統比起來，大型、高效能系統就更有可能會發生這種情形。

2.4. 無計時 KERNEL

在 RHEL 的較早版本中，kernel 使用了以計時器為基礎的機制，這機制會不斷產生系統插斷。每次插斷時，系統會「*輪詢*」（polled）；也就是說，系統會檢查是否有待完成的工作。

根據設定，此系統插斷或「*timer tick*」（計時器的滴答）會每秒發生數百到數十萬次。不管系統的負載為何，這些插斷每秒都會發生。在負載較輕的系統上，會讓程序無法有效地使用睡眠狀態，進而對「*電力消耗*」產生影響。

系統最有效的省電運作模式，是以最快的速度完成工作，然後進入最深沈的睡眠狀態，愈久愈好。要達成此目的，RHEL 6 使用了「*無計時 kernel*」（tickless kernel）。有了這一功能，插斷計時器會從閒置迴圈中移除，將 RHEL 6 完全改為以插斷為驅動的環境。

無計時 kernel 允許系統在閒置期間進入最深沈的睡眠狀態，並在有工作該做時快速回應。

欲知更多資訊，請參閱《*電源管理指南*》，網址為 https://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/?locale=zh-TW。

2.5. 控制群組

RHEL 提供了許多有助於效能微調的選項。儘管是擁有數百個處理器的大型系統，亦可經由微調提供高效能。然而，要為這些系統進行微調，需要有相當程度的專業知識，以及完善定義的工作量。以往，大型系統的費用昂貴且數量較少，因此能給予個別的特殊處理。現在這些系統已是主流，因此需要使用更有效率的工具。

現在，強大的系統已被使用來進行服務整併。原本在四至八個較舊伺服器上執行的工作，已能被置入單一伺服器中執行。如先前在〈節 1.2.1, “平行運算”〉中所討論到，目前許多中階的系統，皆包含了比以往高階系統還要多的核心。

現今許多應用程式皆被設計為能平行處理、使用多執行續或程序，以提升效能。然而，鮮少應用程式能夠有效善用超過八個執行續。也因為如此，應用程式一般需要被安裝在一部 32 個 CPU 的系統上，以最大化其生產力。

請考量此狀況：現今的小型、廉價主流系統的效能，已經能與過去昂貴系統的高效能抗衡。便宜、高效能的機器能讓系統架構師將更多的服務，整併至更少的系統中。

然而，有些資源（例如 I/O 和網路通訊）是共享的，並且其成長速度沒有 CPU 快。因此，一部容納了多項應用程式的系統，在某應用程式耗費了過高的單一資源時，可能會遇上整體效能降低的問題。

為了解決此問題，RHEL 6 現在已支援「*控制群組*」（cgroup）。cgroup 能讓管理員視需求為特定工作分配資源。比方說，這代表能夠為某資料庫應用程式分配四個 CPU 80% 的運算能力、60GB 的記憶體，以及 40% 的磁碟 I/O。在一部相同系統上執行的網站應用程式，則可能被分配到兩個 CPU、2GB 的記憶體，以及 50% 的可用網路頻寬。

如此一來，資料庫與網站應用程式兩者皆能提供較佳的效能，因為系統避免了這兩項應用程式消耗過高的系統資源。此外，cgroup 有許多方面乃「*自行微調*」的，因此能夠讓系統根據工作量上的改變，作出相應的反應。

cgroup 有兩項重要元件：

- 指定給 cgroup 的多項任務
- 分配給這些任務的資源

指定給 cgroup 的任務會在 cgroup「*中*」執行。任何任務所衍生出的子任務也會在相同的 cgroup 中執行。這能讓管理員將整個應用程式視為單一單元來管理。管理員亦可配置下列資源的分配：

- CPUset
- 記憶體
- I/O
- 網路（頻寬）

在 CPUset 中，cgroup 能讓管理員配置 CPU 的數量、與特定 CPU 或節點建立關聯[3]，以及任務所能使用的 CPU 時間。使用 cgroup 來配置 CPUset 對於確保較佳的整體效能來說非常重要，它可避免應用程式消耗過多的資源，同時確保應用程式能取得 CPU 的運作時間。

I/O 頻寬與網路頻寬是以其它資源控制器來管理的。與先前相同，該資源控制器能讓您決定一個 cgroup 中的任務可使用多少頻寬，並同時確保 cgroup 中的任務不會使用過量的資源，或是遇上資源不足的問題。

cgroup 能讓管理員以高階的方式，定義和分配各個應用程式所需（且即將使用）的系統資源。接著系統便會自動管理和平衡各項應用程式，提供良好的效能並優化整體的系統效能。

欲知如何使用控制群組的資訊，請參閱《*資源管理指南*》，網址為 https://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/?locale=zh-TW。

2.6. 改進儲存與檔案系統

RHEL 6 也針對儲存與檔案系統管理，作了多項改進。此版本中最顯著的兩項優點，是支援了 ext4 與 XFS。欲知儲存與檔案系統的效能改進之詳盡內容，請參閱〈章 7, *檔案系統*〉。

ext4

ext4 是 RHEL 6 的預設檔案系統。它是 EXT 檔案系統的第四版，理論上支援的最大檔案系統大小為 1 EB，單一檔案的最大大小為 16 TB。RHEL 6 支援的最大檔案系統為 16 TB，單一檔案大小亦為 16 TB。除了儲存空間更大以外，ext4 也有多項新功能，例如：

- 以扇區為基礎的 metadata
- 延遲分配
- 日誌校驗

欲知 ext4 檔案系統的更多詳情，請參閱〈節 7.3.1, “ext4 檔案系統”〉。

XFS

XFS 是非常穩固、成熟的 64 位元日誌型檔案系統，支援單一主機上非常大型的檔案與檔案系統。這檔案系統最開始由 SGI 公司發展，長時間用在極大型的伺服器與儲存陣列。XFS 的功能包括：

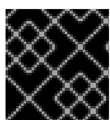
- 延遲分配
- 動態分配 inode
- B-tree 索引，用於未使用空間管理的擴充功能
- 線上磁碟重組與擴充檔案系統
- 複雜的 metadata 事先讀取演算法則

雖然 XFS 支援至 EB（10^18 位元組），但 RHEL 所支援的最大 XFS 檔案系統為 100 TB（10^12 位元組）。欲知更多 XFS 的詳情，請參閱〈節 7.3.2, “XFS 檔案系統”〉。

大型開機啟動磁碟

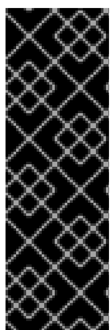
傳統 BIOS 支援的最大磁碟大小為 2.2 TB。RHEL 6 系統可使用新型、支援大於 2.2 TB 磁碟的 BIOS，方法是使用新的磁碟結構，名為「*全域分割表*」（GPT，Global Partition Table）。GPT 僅能用於資料硬碟；不能搭配 BIOS 用於開機磁碟。因此，開機磁碟的最大大小仍為 2.2 TB。BIOS 最早是為 IBM PC 所建立，並不斷大幅精進以符合現代的硬體，而「*統一可延伸韌體介面*」（UEFI，Unified Extensible Firmware Interface）則是設計用來支援新興硬體。

RHEL 6 也支援 UEFI，用來取代 BIOS（仍受支援）。使用 UEFI、執行 RHEL 6 的系統允許使用 GPT 與 2.2 TB 以上的分割區，作為開機與資料分割區。



重要

RHEL 6 並不支援 32 位元 x86 系統的 UEFI。



重要

請注意，UEFI 和 BIOS 兩者的開機配置差異甚大。因此，系統開機時使用的韌體，必須與進行安裝時所使用的韌體相同。您無法在使用了 BIOS 的系統上安裝作業系統，然後移到使用了 UEFI 的系統上啟動。

RHEL 6 支援 UEFI 2.2 規格。RHEL 6 應該能啟動、執行支援 UEFI 2.3 規格以上的硬體，不過這些較新規格所定義的額外功能將無法使用。UEFI 的規格可由 http://www.uefi.org/specs/agreement/ 取得。

[3] 節點一般被定義為插槽（socket）中的一組 CPU 或核心。

章 3. 監控和分析系統效能

本章節詳細介紹了可用來監控和分析系統與應用程式效能的各項工具，並指出各項工具最適用於哪些情況下。各項工具所搜集的資料可顯示影響效能的瓶頸或其它系統問題。

3.1. PROC 檔案系統

proc「檔案系統」包含了代表 Linux kernel 現有狀態的檔案階層之目錄。它允許應用程式與使用者從 kernel 的角度，來檢視系統。

proc 目錄也包含了系統的硬體資訊，以及任何執行中程序的資訊。這些檔案大部分都是唯讀的，但有些檔案（尤其是位於 **/proc/sys** 目錄中的檔案）可以由使用者與應用程式來調整，以改變 kernel 中的配置。

欲知更多檢視、編輯 **proc** 目錄中的檔案之資訊，請參閱《*建置指南*》，網址為：https://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/?locale=zh-TW。

3.2. GNOME 與 KDE 系統監控程式

GNOME 與 KDE 桌面環境都擁有圖形化工具，幫助使用者監控、修改系統行為。

GNOME 系統監控程式

「**GNOME 系統監控程式**」會顯示系統的基本資訊，並讓使用者監控系統程序、資源、或檔案系統的使用量。請在「**終端機**」中執行 **gnome-system-monitor** 指令，或點選「**應用程式**」選單，然後選擇「**系統工具 > 系統監控**」。

「**GNOME 系統監控**」有四個分頁：

系統

顯示電腦軟硬體的基本資訊。

程序

顯示運作中的程序，以及這些程序間的關係，還有每個程序的詳細資訊。它也讓您篩選欲顯示的程序，並對這些程序進行一些動作（啟動、停止、終結、改變優先順序等）。

資源

顯示目前的 CPU 使用時間、記憶體與置換空間的使用量、以及網路使用量。

檔案系統

列出所有已掛載的檔案系統，及每個檔案系統的基本資訊，例如檔案系統類型、掛載點、及記憶體使用量。

欲知「**GNOME 系統監控程式**」的更多資訊，請參閱其「**求助**」選單，或是《*建置指南*》，網址為 https://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/?locale=zh-TW。

KDE 系統監視器

KDE 的「**系統監視器**」能讓使用者監控系統現有負載與執行中的程序，也能讓使用者對程序進行操作。請在「**終端機**」中執行 **ksysguard**，或點選「**Kickoff 應用程式啟動器**」並選取「**應用程式 > 系統 > 系統監視器**」。

KDE 的「**系統監視器**」有兩個分頁：

行程表

顯示執行中程序的清單，預設上按照字母排序。您可以用其它方式排序，包括 CPU 的總使用量、實體或共享記憶體的使用量、擁有者、優先順序等等。您也可以篩選結果、搜尋特定程序、或對程序進行一些動作。

系統負載

顯示 CPU 使用量、記憶體與置換空間的使用量、以及網路使用量的歷史圖表。將滑鼠移到圖表上，就可以看到詳細的分析與圖例。

欲知 KDE「**系統監視器**」的更多資訊，請參閱該應用程式的「**說明**」選單。

3.3. 內建的命令列監控工具

除了圖形化介面的工具以外，RHEL 提供了多種工具，從命令列監控系統。這些命令列工具的好處是可以在 runlevel 5 以外的環境中執行。本節會簡單討論每項工具，並列舉每項工具最適合的環境。

top

top 能以動態、即時的方式，檢視系統上執行中的程序。您會看到多種資訊，包括目前 Linux kernel 所管理的任務。它也有一些管理程序的能力。不管是 **top** 的運作或顯示的資訊，都是可以加以配置的，任何配置的內容在開機後都會持續存在。

預設上，所顯示的程序是以 CPU 使用率的百分比來排序，讓使用者立即得知那些程序耗用最多資源。

欲知 **top** 的使用詳情，請參閱 man page，指令為：**man top**。

ps

ps 工具能取得當下運作中程序的狀態。預設上，這些程序僅限於執行此指令的使用者所執行、且與同樣終端機相關的程序。

ps 所顯示的資料比 **top** 還多，但不是動態的。

欲知 **ps** 的使用詳情，請參閱 man page，指令為：**man ps**。

vmstat

vmstat（虛擬記憶體的統計數據）會列出系統程序、記憶體、分頁、區塊裝置 I/O、插斷、與 CPU 活動的瞬間資料。

雖然這程式不像 **top** 提供動態資料，但您可以指定取樣的間隔，觀察幾乎是即時的系統活動。

欲知 **vmstat** 的使用詳情，請參閱 man page，指令為：**man vmstat**。

sar

sar（系統活動回報程式）會蒐集、回報本日截至目前為止的系統活動資訊。預設的資訊包括本日的 CPU 使用率，每十分鐘採樣一次。

```
12:00:01 AM     CPU     %user     %nice    %system   %iowait    %steal
%idle
12:10:01 AM     all      0.10      0.00       0.15      2.96      0.00
96.79
12:20:01 AM     all      0.09      0.00       0.13      3.16      0.00
96.61
```

```
12:30:01 AM     all      0.09      0.00      0.14      2.11      0.00
97.66
...
```

若不想透過 **top** 等類似工具來建立系統活動的定期報告，這工具非常好用。

欲知 **sar** 的使用詳情，請參閱 man page，指令為：**man sar**。

3.4. TUNED 與 KTUNE

tuned 是一組 daemon，用來監控、蒐集多種系統元件的使用量資料，並視需要使用這些資訊來動態微調系統。它可以在 CPU 與網路有所變動時予以回應，並調整設定以改進啟用中裝置的效能，或降低非使用中裝置的耗用電量。

伴隨而來的 **ktune** 加上 **tuned-adm** 工具提供了數種預先配置好的微調設定檔，能在多種特定使用情況下，加強效能並降低所消耗的電量。請編輯這些設定檔、或建立新的設定檔，為您的環境建立專用的效能解決方案。

tuned-adm 所提供的設定檔包括：

default

預設的省電設定檔。這是最基本的省電設定檔。它僅啟用磁碟與 CPU 的嵌入程式。請注意，這與關閉 **tuned-adm** 不同，因為此時 **tuned** 與 **ktune** 都是停用的。

latency-performance

伺服器設定檔，提供典型的延遲效能微調。它會停用 **tuned** 與 **ktune** 省電機制。**cpuspeed** 模式會變為 **performance**。每個裝置的 I/O elevator 會變為 **deadline**。為求電源管理的服務品質，**cpu_dma_latency** 所需要的值會被註冊為 **0**。

throughput-performance

伺服器的設定檔，用於典型的效能吞吐微調環境。如果系統沒有企業級的儲存裝置，建議使用此設定檔。它與 **latency-performance** 非常類似，僅有以下不同：

- ***kernel.sched_min_granularity_ns***（排程器的最小多工之精細程度）設為 **10** ms；
- ***kernel.sched_wakeup_granularity_ns***（排程器的喚醒功能之精細程度）設為 **15** ms；
- ***vm.dirty_ratio***（虛擬機器「需要變更」(dirty) 的比例）設為 **40%**；同時
- 啟用通透式巨型分頁。

enterprise-storage

建議將此設定檔用於搭配了企業級儲存裝置（包括含備用電池的控制器快取保護，以及磁碟上的快取管理）的企業級伺服器配備。它與 **throughput-performance** 設定檔相似，外加一項設定：檔案系統會以 **barrier=0** 重新掛載。

virtual-guest

建議將此設定檔用於搭配了企業級儲存裝置（包括含備用電池的控制器快取保護，以及磁碟上的快取管理）的企業級伺服器配備。它與 **throughput-performance** 設定檔相似，除了：

- ***readahead*** 的值設為 **4x**，同時

- 非 root/boot 的檔案系統會以 **barrier=0** 重新掛載。

virtual-host

根據 ***enterprise-storage*** 設定檔， ***virtual-host*** 也會降低虛擬記憶體的 swap 動作，並更積極啟用需要變更分頁的回寫（writeback）功能。這設定檔可在 RHEL 6.3 以後找到，建議用在虛擬主機上，包括 KVM 與 RHEV 的主機。

關於 **tuned** 與 **ktune** 的進一步資訊，請參閱 Red Hat Enterprise Linux 6《*電源管理指南*》，網址為 https://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/?locale=zh-TW。

3.5. 應用程式設定檔工具

建立設定檔是蒐集程式執行時的行為與資訊之過程。為應用程式建立設定檔，可以看出程式的哪個部分可以進行優化，以增加程式的整體速度、降低記憶體使用量等等。應用程式設定檔工具能幫忙簡化這過程。

RHEL 6 支援三種設定檔工具：**SystemTap**、**OProfile** 與 **Valgrind**。詳細記載這些工具的內容超出本指南的範圍；然而，本節能為使用者指出更多詳細資訊的連結，以及每個設定檔工具所適用的任務之一覽。

3.5.1. SystemTap

SystemTap 是追蹤與偵測工具，讓使用者仔細監控、分析作業系統的活動（尤其是 kernel 的活動）。這程式所提供的資訊與其它工具（例如 **netstat**、**top**、**ps** 與 **iostat**）類似，但包括了額外的篩選與分析選項。

SystemTap 提供了系統活動與應用程式行為之更深入、更精準的分析資料，讓使用者找出系統與應用程式的瓶頸。

Eclipse 的 Function Callgraph 嵌入程式使用的後端即為 SystemTap，能徹底監控程式的狀態，包括系統呼叫、傳回值、時間、與使用者空間的變數，並提供視覺化的資訊，好讓優化更為容易。

欲知更多關於 SystemTap 的資訊，請參閱《*SystemTap 初學者指南*》，網址為 https://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/?locale=zh-TW。

3.5.2. OProfile

OProfile（**oprofile**）是系統全域的效能監控工具。它使用了處理器的專職效能監控硬體，以擷取關於 kernel 與系統可執行檔的資訊，例如何時參照記憶體、要求 L2 快取的次數、收到硬體插斷的次數。它也可以用來決定處理器的用量，得知哪些應用程式與服務是最常用的。

OProfile 也可透過 Eclipse OProfile 嵌入程式，與 Eclipse 合用。這嵌入程式能讓使用者輕易地找出程式中最耗時間的部份，並以豐富的圖形化方式呈現 OProfile 的所有命令列函數。

然而，請注意 OProfile 的幾項限制：

- 效能監控的樣本可能不夠精準：因為處理器可能不會照順序執行指令，樣本可能是從鄰近指令的紀錄而來，而非來自觸發插斷的指令。
- 因為 OProfile 是系統全域的程式，並將程序重複執行多次，因此可以累積多次執行樣本。這表示您可能需要清除之前執行的樣本資料。
- OProfile 專注於辨明受限於 CPU 的程序之問題，因此並不會辨明休眠中、等待被其它事件鎖定之程序。

欲知更多使用 OProfile 的詳情，請參閱《*建置指南*》，網址為 https://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/?locale=zh-TW；或參閱系統上的 **oprofile** 文件，位於 `/usr/share/doc/oprofile-<version>`。

3.5.3. Valgrind

Valgrind 提供了數種偵測與側寫的工具，以協助改善應用程式的效能與正確性。這些工具可以偵測與記憶體及執行續相關的錯誤，還有過度執行堆積、堆疊、與陣列等，讓使用者輕易地找到、修正程式碼中的錯誤。這些工具也可以側寫快取、堆積、分支預測，以找出增加應用程式速度、將應用程式使用的記憶體最小化之因素。

Valgrind 會在人為的 CPU 環境中執行應用程式，在應用程式執行時評測程式碼，藉此分析應用程式。然後 Valgrind 會針對應用程式執行至使用者指定的檔案子、檔案、或網路插槽，清楚印出每個相關程序的「註解」。這種等級的評測端視使用的 Valgrind 工具及設定而有所不同，但請注意，執行這評測碼可能會花上正常執行時間的 4-50 倍。

Valgrind 可直接用在應用程式上，而不需要重新編譯。然而，因為 Valgrind 使用了偵錯資訊，以找尋程式碼中的問題，因此如果應用程式與所支援的函式庫在編譯時並未啟用偵錯資訊，那麼強烈建議您納入此資訊，重新編譯。

至於 RHEL 6.4，Valgrind 整合了 gdb（GNU Project Debugger）以改進偵錯的效率。

Valgrind 的更多資訊可在《*開發指南*》中找到，網址為 https://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/?locale=zh-TW；或安裝 valgrind 套件後使用 `man valgrind` 指令以讀取 man page。相關的文件可以從以下地方找到：

- `/usr/share/doc/valgrind-<version>/valgrind_manual.pdf`
- `/usr/share/doc/valgrind-<version>/html/index.html`

欲知 Valgrind 如何用於側寫系統記憶體的資訊，請參閱〈節 5.3, "使用 Valgrind 側寫記憶體使用量"〉。

3.5.4. Perf

perf 工具提供了多種有用的效能記數器，讓使用者評量系統上其它指令所帶來的衝擊：

`perf stat`

這指令提供了一般效能事件的全面統計資料，包括所執行的指令與消耗的時脈週期。您可以透過選用的旗標來蒐集事件資訊，而不是預設的評量事件。自 RHEL 6.4 起，使用者可以執行 `perf stat`，根據一或多個控制群組（cgroup）來篩選監控資料。欲知更多詳情，請參閱 man page，指令為：`man perf-stat`。

`perf record`

這指令會將效能資料記錄至檔案中，以方便日後使用 `perf report` 進行分析。欲知更多詳情，請參閱 man page，指令為：`man perf-record`。

`perf report`

這指令會從檔案中讀取效能資料，並加以分析。欲知更多詳情，請參閱 man page，指令為：`man perf-report`。

`perf list`

這指令會列出特定機器上的可用事件。這些事件會根據系統的效能監控之軟硬體配置，而有所不同。欲知更多詳情，請參閱 man page，指令為：`man perf-list`。

`perf top`

這指令的效用與 **top** 類似。它會即時產生、顯示效能記數器的側寫資料。欲知更多詳情，請參閱 man page，指令為：**man perf-top**。

欲知更多關於 **perf** 的資訊，請參閱 Red Hat Enterprise Linux《*開發指南*》，網址為 https://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/?locale=zh-TW。

3.6. RED HAT ENTERPRISE MRG

Red Hat Enterprise MRG 的 Realtime（即時）元件包括 **Tuna**：能讓使用者調整系統的可微調參數、並檢視這些變更的工具。一開始發展這工具是為了與 Realtime 元件一起使用，但也可以用來調整標準的 Red Hat Enterprise Linux 系統。

有了 Tuna，使用者可以調整或停用不必要的系統活動，包括：

- 與電源管理、錯誤偵測、與系統管理插斷的 BIOS 參數；
- 網路設定，例如插斷聯合（interrupt coalescing）與 TCP 的使用；
- 在日誌型檔案系統中的日誌活動；
- 系統日誌；
- 插斷與使用者程序是否由一個或一群 CPU 所處理；
- 是否使用 swap 空間；以及
- 如何處理記憶體不足的例外情況。

欲知使用 Tuna 介面微調 Red Hat Enterprise MRG 的概念性詳盡資訊，請參閱《*Realtime 微調指南．一般系統微調*》一章。欲知 Tuna 介面的詳細指示，請參閱《*Tuna 使用者指南*》。這兩本指南都可以在以下網址找到：https://access.redhat.com/site/documentation/Red_Hat_Enterprise_MRG/?locale=zh-TW。

章 4. CPU

CPU 這個名詞代表「*中央處理器*」（central processing unit），對於大部份系統來說，此名詞容易造成誤解，因為「*中央*」一般意味著「*單一*」，而目前大部份的系統皆裝載了超過一顆中央處理器或核心。實體上來講，CPU 會封裝為單一單位，並連接在主機板的「*插槽*」（socket）中。主機板上的各個插槽皆擁有各別的連接用處：它們可連接其它 CPU 插槽、記憶體控制器、插斷控制器，以及其它周邊裝置。對於作業系統來說，插槽代表 CPU 與相關資源的邏輯分群。大部份時候，我們將會以此概念為中心來討論 CPU 微調。

RHEL 會保留大量有關於系統 CPU 事件的數據；這些數據對於微調並改善 CPU 效能來說相當有幫助。〈節 4.1.2, “微調 CPU 效能”〉中討論了一些較有幫助的數據、如何找到並分析這些數據以進行效能微調。

拓樸

在較舊的系統上，所裝載的 CPU 也較少，這種情況下可使用「*SMP*」（對稱式多處理器，Symmetric Multi-Processor）架構。這代表系統中的各個 CPU 皆會以相似（或對稱）的方式存取可用的記憶體。近年來，插槽與 CPU 的對應數量已大幅增長，要讓處理器能對稱式存取系統中所有的記憶體，將會耗費極高的成本。目前大部份擁有高 CPU 數量的系統，皆採用了一種名為「*NUMA*」（非對稱式記憶體存取，Non-Uniform Memory Access）的架構來取代 SMP。

AMD 處理器已透過了其 *Hyper Transport*（HT）互聯技術採用了此類型的架構一段時間了，而 Intel 則在進行其 *Quick Path Interconnect*（QPI）設計時，開始實作了 NUMA。因為您為應用程式分配資源時，需要考量系統的「*拓樸*」，因此 NUMA 與 SMP 需個別進行不同的微調。

執行續

在 Linux 作業系統中，執行動作的單位為「*執行續*」（thread）。執行續擁有註冊文本、堆疊，以及它們在 CPU 上執行的一項可執行程式碼區段。作業系統（OS）將負責在可用的 CPU 上排程這些執行續。

作業系統會透過在可用的核心上，為執行續進行負載平衡，以優化 CPU 效能。因為作業系統主要僅運用 CPU 資源，擴充性應用程式效能，它可能並未作出最佳的效能。將某個應用程式執行續移至另一插槽上的 CPU 中，效能並不一定會優於等待目前的 CPU，因為進行插槽之間的記憶體存取作業可能會花上更多時間。對於高效能的應用程式來說，應用程式設計者最好事先決定執行續應放置在哪。〈節 4.2, “CPU 排程”〉討論了如何優化分配 CPU 與記憶體，並以最佳的方式執行應用程式執行續。

插斷

「*插斷*」（interrupt，在 Linux 中亦稱為 IRQ）是一項比較不明顯（然而依然重要），卻可影響應用程式效能的系統事件。這些事件由作業系統處理，並由周邊設備使用來發出「資料到達」，或是「作業完成」的訊號，例如網路寫入或是計時器事件。

正在執行應用程式碼的 OS 或 CPU 處理插斷的方式，不會影響應用程式的運作。然而，它可能會影響應用程式的效能。本章節亦同時討論如何避免插斷對應用程式效能所帶來的一些負面影響。

4.1. CPU 拓樸

4.1.1. CPU 與 NUMA 拓樸

最早的電腦處理器為「*單一處理器*」（uniprocessor），表示電腦只有一個 CPU。當時能平行處理許多程序的假象，是作業系統讓單一處理器快速切換，執行多個執行續。設計師發現將執行指令的時脈調快，只能將系統效能加快到一定程度（以現有科技來說，建立穩定的時脈波形是限制）。要讓系統的整體效能加快，設計師決定在系統上增加 CPU，允許雙處理器平行處理。增加處理器的趨勢至今方興未艾。

早期的大部分多處理器系統中，每個 CPU 連至每個記憶體位置（通常是平行匯流排）的邏輯路徑都是一樣的。這讓系統中的每個 CPU 都以同樣時間存取任何記憶體位置。這類架構稱為 SMP（Symmetric Multi-Processor，對稱多處理架構）系統。在 CPU 數量不多的情形下，SMP 得以運作無礙，可一旦 CPU 增加到一定數量（8 或 16）時，要平行存取記憶體會使用太多系統資源，導致周邊配備缺乏空間。

要在系統中使用更多 CPU，結合了兩種概念：

1. 序列匯流排
2. NUMA 拓樸

序列匯流排是單線的通訊路徑，時脈非常高，將需要傳輸的資料封包化高速傳出。硬體設計師開始採用序列匯流排作為 CPU 之間、以及 CPU 與記憶體控制晶片及其它周邊之間的高速連線。這表示與其得在「*每個*」CPU 到記憶體子系統之間，鋪設 32 或 64 條路徑，不如只需「 *一條*」路徑，徹底降低主機板上的所需空間。

同時，硬體設計師透過降低晶片大小，在同樣空間中塞入更多電晶體。設計師不再將一個一個 CPU 直接放在主機板上，而是整合到同一個晶片上，成為單一的多核心處理器。然後，與其讓每個處理器封裝以同樣方式存取記憶體，設計師開始使用 NUMA（非一致性記憶體存取，Non-Uniform Memory Access）策略，讓每個封裝/插槽的組合擁有一或多個專用的記憶體區域，以供高速存取。每個插槽之間也有較慢速的連線，以存取其它插槽的記憶體。

舉一個簡單的 NUMA 範例，假設我們的主機板上有兩個 CPU 插槽，每個插槽上都裝有四核心的處理器。這表示這台系統上總共有八顆 CPU；每個插槽四顆。每個插槽也連上各自 4 GB 的記憶體插槽，亦即系統總共有 8 GB 的記憶體。在此範例中，編號 0-3 的 CPU 位於編號 0 的插槽，4-7 的 CPU 位於插槽 1。此範例中的插槽也相當於 NUMA 的節點。

CPU 0 要存取記憶體 0 可能需要三個時脈週期：第一個週期向記憶體控制晶片出示位址，第二個週期設定對記憶體位置的存取，第三個週期讀取或寫入該位置。然而，因為 CPU 4 位於不同的插槽上，需要透過兩組記憶體控制晶片（先是插槽 1 的本地記憶體控制晶片，然後是插槽 0 的遠端記憶體控制晶片）來存取，因此需要四個週期才能存取同樣區塊的記憶體。如果系統競逐同樣的記憶體位置，（亦即超過一個 CPU 同時存取同樣位置的記憶體），記憶體控制晶片需要加以仲裁、將存取過程序列化，這樣一來存取記憶體的時間就會加長。增加快取的一致性（確保本地 CPU 的快取包含同樣記憶體位置的同樣資料）會讓這過程更加複雜。

Intel (Xeon) 與 AMD (Opteron) 的最新高階處理器都有 NUMA 拓樸。AMD 處理器之間的連線稱為 HT（HyperTransport），而 Intel 的稱為 QPI（QuickPath Interconnect）。兩者的分別在於連接 CPU、記憶體或周邊裝置的實體連線，但事實上兩者都像開關一樣，讓一個裝置與另一個裝置能以通透的方式連接。在此情況下，「通透」指得是使用交互連線時，不需要透過特別的 API 程式；而不是「不耗費任何成本」的意思。

因為系統架構的差異性極大，因此要歸類存取非本地記憶體的效能耗損並不實際。我們可以說，每個交互連結中的每個「*中繼點*」（hop）都會讓效能降低些許，因此參照一個離現有 CPU 兩個交互連線遠的記憶體位置時，會增加至少「*2N + 記憶體週期時間*」的存取時間，其中 N 是每個中繼點所耗去的時間。

考慮到此效能耗損，對效能極其敏感的應用程式應避免在 NUMA 拓樸系統上，定期存取遠端記憶體。這應用程式應該設定為留在特定的節點上，並從該節點上分配記憶體。

要這樣做的話，應用程式必須知道幾件事：

1. 系統的「*拓樸*」為何？
2. 目前應用程式在何處執行？
3. 最近的記憶體插槽在哪裡？

4.1.2. 微調 CPU 效能

讀取此節能讓使用者了解如何微調，以取得更佳的 CPU 效能，此節也簡介了數種工具以達成此目標。

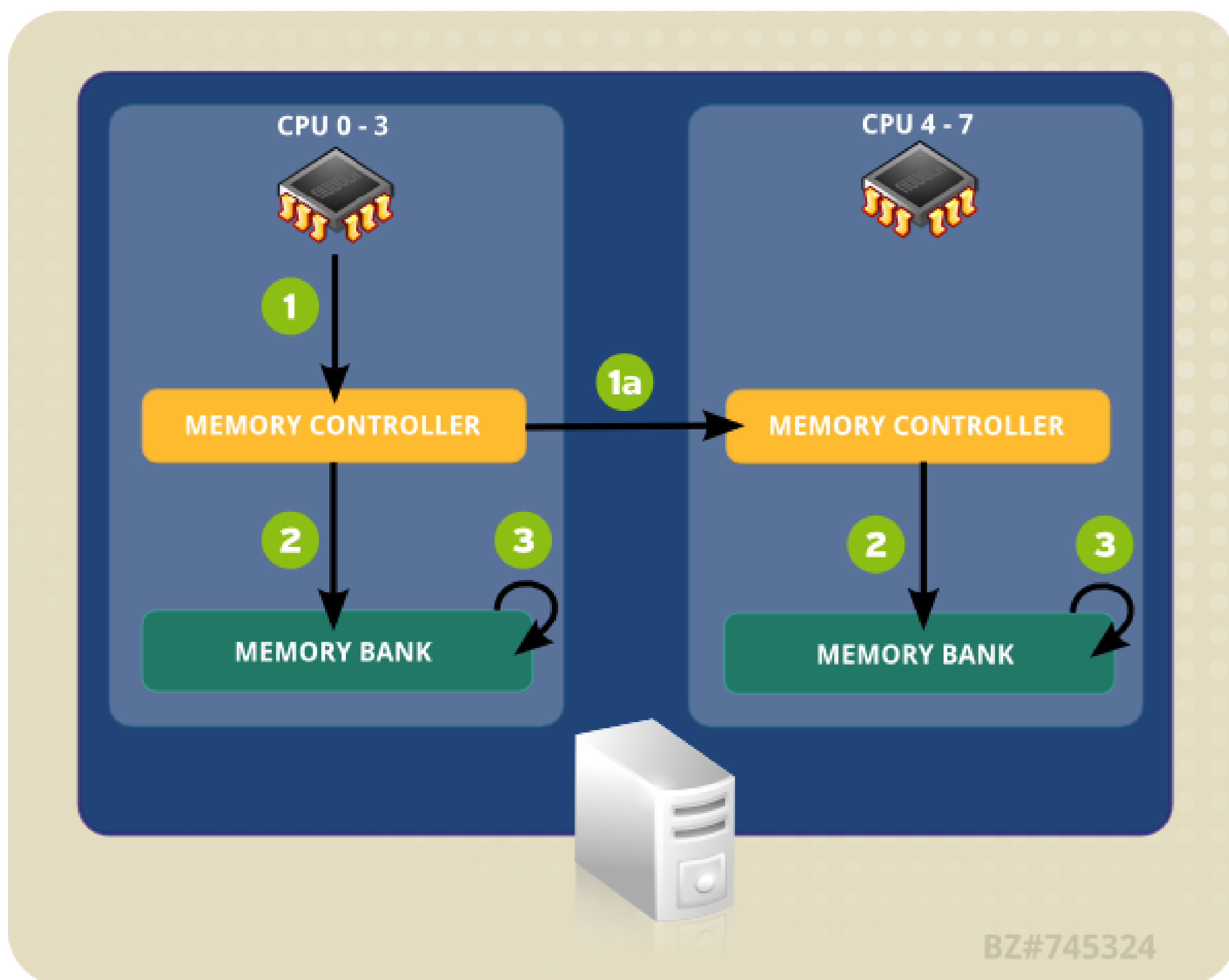
NUMA 一開始是用來連接單一處理器與多記憶體插槽。自從 CPU 製造商精心打造處理器，並減小晶片大小後，單一 CPU 就可包含多核心。這些 CPU 核心聚集一處，每個都可以用同樣時間存取本機記憶體插槽，核心之間也可以共享快取；然而每次在核心、記憶體、與快取之間「切換」時，都會消耗些許效能。

圖形 4.1, “NUMA 拓樸的本地與遠端記憶體存取” 中的範例系統包含了兩個 NUMA 節點。每個節點都有四個 CPU、一組記憶體插槽、以及一組記憶體控制晶片。節點上的任何 CPU 都可直接存取節點上的記憶體插槽。如節點 1 的箭頭所示，步驟如下：

1. 一顆處理器（0-3 的任何一顆）將記憶體位址傳給本地的記憶體控制器。

2. 記憶體控制器設定對記憶體位址的存取。

3. CPU 對那組記憶體位址進行讀或寫。



圖形 4.1. NUMA 拓樸的本地與遠端記憶體存取

然而，如果一個節點上的某個 CPU 需要存取屬於另一個 NUMA 節點的記憶體插槽，需要採取的路徑就會迂迴些：

1. 一顆處理器（0-3 的任何一顆）將記憶體位址傳給遠端的記憶體控制器。

   1. 針對遠端記憶體位址的 CPU 需求會傳到遠端的記憶體控制器上，看起來就像從該節點的本機存取節點。

2. 遠端記憶體控制器設定對遠端記憶體位址的存取。

3. CPU 對那組遠端記憶體位址進行讀或寫。

每個動作都需要透過多個記憶體控制器，因此遠端存取的時間會是本機的兩倍有餘。因此，多核心系統的主要效能考量，是確保資訊能透過最短或最快的路徑，以最有效率的方式傳輸。

要配置應用程式以取得最佳 CPU 效能，您需要知道：

- 系統拓樸（系統元件是如何相連的）、
- 應用程式執行時所使用的核心、以及
- 最近的記憶體插槽之位置。

RHEL 6 附有多種工具，幫助使用者尋找資訊，並根據這些資訊微調系統。以下章節會對 CPU 效能微調的工具作簡介。

4.1.2.1. 使用 taskset 設定 CPU 關聯

taskset 會透過程序 ID，擷取、設定執行中程序的 CPU 關聯（CPU affinity）。這程式可以用來使用特定的 CPU 關聯來啟動程序，綁定一組特定程序與一個（或多個）特定 CPU。然而，**taskset** 並不能保證分配本地記憶體。如果您需要分配本地記憶體所帶來的額外效能，我們建議您使用 **numactl** 而非 **taskset**；詳情請見〈節 4.1.2.2, "使用 **numactl** 控制 NUMA 政策"〉。

CPU 關聯是以位元遮罩的方式呈現。最低位元相當於第一個邏輯 CPU，最高位元相當於最後一個邏輯 CPU。通常這些遮罩是以 16 位元方式呈現，因此 **0x00000001** 表示編號為 0 的處理器，而 **0x00000003** 表示編號為 0 與 1 的處理器。

要為執行中的程序設定 CPU 關聯，請執行以下指令，並以您想要綁定的一或多顆處理器取代 *mask*、將您想要改變關聯之程序 ID 取代 *pid*。

```
# taskset -p mask pid
```

要使用給定的關聯啟動程序，請執行以下指令，並以您希望程序想要綁定的一或多顆處理器取代 *mask*、並以您想要執行的程式、選項、與引數取代 *program*。

```
# taskset mask -- program
```

您可使用位元遮罩來指定處理器，也可以使用 **-c** 選項加上以逗號分隔的個別處理器清單（或處理器範圍），例如：

```
# taskset -c 0,5,7-9 -- myprogram
```

關於 **taskset** 的進一步資訊，請參閱 man page，指令為：**man taskset**。

4.1.2.2. 使用 numactl 控制 NUMA 政策

numactl 會使用特定的排程或記憶體取代政策，執行程序。所選擇的政策會套用在該程序與其子程序上。**numactl** 也可以為共享記憶體區段或檔案，設定一致性的政策；並為程序設定 CPU 關聯與記憶體關聯。它會使用 **/sys** 檔案系統來決定系統拓樸。

/sys 檔案系統包含了 CPU、記憶體、與周邊裝置是如何透過 NUMA 相互連結的資訊。特別是 **/sys/devices/system/cpu** 目錄包含了系統 CPU 如何相互連結的資訊。**/sys/devices/system/node** 目錄包含了系統上的 NUMA 節點的資訊、以及這些節點之間的相對距離。

在 NUMA 系統上，處理器與記憶體插槽的距離愈長，處理器存取記憶體插槽的速度就愈慢。因此，對效能敏感的應用程式應該配置好，從最近的可用記憶體插槽存取記憶體。

對效能敏感的應用程式也應該配置好，在多個核心上執行，特別是會用到多執行續的應用程式更該這麼配置。因為第一級快取多半較小，如果多執行續在同樣的 CPU 核心上執行，每個執行續都可能會將前一個執行續所存取的快取資料逐出。當作業系統試圖在這些執行續之間進行多工作業、同時執行續持續逐出其他執行續的快取資料時，取代快取段（cache line）的執行時間就會大幅增加。這問題稱為「*快取置換*」（cache thrashing）。因此，建議綁定多執行續應用程式至節點上，而非單一 CPU 核心上，因為這允許執行續在多層級（第一、第二、與最後快取）上共享快取段，並降低填滿快取的運作需求。然而，如果所有執行續都存取同樣的快取資料，綁定應用程式至單一 CPU 核心有其效能上的優勢。

numactl 能讓您綁定應用程式至特定 CPU 核心或 NUMA 節點，並分配記憶體使其與 CPU 核心或多核心與應用程式產生關聯。numactl 的有用選項有：

--show

顯示現有程序的 NUMA 政策設定。這參數並不需要其他參數，用法類似：**numactl --show**。

--hardware

顯示系統上可用節點的清單。

--membind

僅能從特定節點分配記憶體。當使用此選項時，如果這些節點上的記憶體不足，分配運作就會失敗。此參數的用法為 **numactl --membind=*nodes program***，其中 *nodes* 是您想要分配記憶體的節點清單，*program* 是向節點索求記憶體之程式。節點的編號可以是用逗號分隔的清單、一段範圍、或是以上二者的結合。**numactl** 的更多資訊可以在 man page 中找到：**man numactl**。

--cpunodebind

只在屬於特定一或多個節點上的 CPU 執行一個指令（與其子程序）。此參數的用法是 **numactl --cpunodebind=*nodes program***，其中 *nodes* 是該與特定程式（*program*）綁定的 CPU 之節點清單。節點的編號可以是用逗號分隔的清單、一段範圍、或是以上二者的結合。**numactl** 的更多資訊可以在 man page 中找到：**man numactl**。

--physcpubind

僅在特定處理器上執行指令（及其子程序）。此參數的用法是 **numactl --physcpubind=*cpu program***，其中 *cpu* 是以逗號隔開的實體 CPU 編號，如 **/proc/cpuinfo** 檔案中的處理器欄位所列，而 *program* 是僅在這些 CPU 上執行的程式。CPU 也可以透過相對於現有 **cpuset** 來指定。詳情請參閱 **numactl** 的 man page，指令為：**man numactl.**

--localalloc

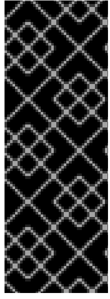
表示記憶體應該總是分配至現有節點上。

--preferred

可能的話，記憶體會分配至指定的節點。如果記憶體無法分配至指定的節點上，就會改分配到其它節點。這選項只接受單一節點編號，例如：**numactl --preferred=*node***。詳情請參閱 numactl 的 man page，指令為：**man numactl**。

libnuma 函式庫包含在 numactl 套件中，提供了簡單的程式介面給 NUMA 政策使用，並由 kernel 所支援。這函式庫提供了比 **numactl** 工具程式更多、更精細的微調功能。欲知更多詳情，請參閱 numa 的 man page，指令為：**man numa(3)**。

4.1.3. numastat



重要

之前，**numastat** 工具是由 Andi Kleen 所寫的 Perl script。現在 RHEL 6.4 使用的版本已經大幅重新撰寫。

為了與之前的版本相容，預設指令（`numastat`，沒有任何選項或參數）對於相容性採取嚴格的措施；請注意使用此指令並加上選項或參數後，會顯著改變螢幕輸出的內容及其格式。

numastat 顯示了每個 NUMA 節點上，程序與作業系統對記憶體存取的統計資料（例如能否正確分配的數據）。預設上，執行 `numastat` 會顯示每個節點上，多少記憶體分頁是由以下事件類別所佔據。

`numa_miss` 與 `numa_foreign` 值愈低，表示 CPU 的效能愈好。

numastat 的更新版本也能顯示程序的記憶體是否橫跨系統，或是透過 **numactl** 集中在特定的節點上。

交叉參照 **numastat** 的結果（透過每個 CPU 的 **top** 結果）來驗證程序的執行續都在分配記憶體的同樣節點上執行。

預設的追蹤類別

numa_hit
: 成功分配至此節點的數目。

numa_miss
: 試圖分配到其它節點，但因為其它節點的記憶體不足而分配至此節點的數目。每個 `numa_miss` 事件都有相對應至另一個節點的 `numa_foreign` 事件。

numa_foreign
: 一開始試圖分配至此節點，但最後卻分配至其它節點的數目。每個 `numa_foreign` 事件都有相對應至另一個節點的 `numa_miss` 事件。

interleave_hit
: 用交錯政策成功分配至此節點的數目。

local_node
: 此節點上，一組程序成功分配記憶體至此節點的次數。

other_node
: 此節點上，其它節點上的一組程序分配記憶體至此節點的次數。

使用以下任何選項，都會將記憶體的顯示單位改為 MB（四捨五入至小數點後二位），並改變其它特定的 **numastat** 行為，如下所述。

-c
: 水平濃縮資訊表。這在擁有大量 NUMA 節點的系統上非常有用，但欄寬與欄間距難以預測。使用此選項時，記憶體數量會四捨五入至最近的 MB 數。

`-m`

以每個節點為基礎，顯示系統全域的記憶體使用資訊，與 **/proc/meminfo** 中的資訊類似。

`-n`

顯示與原始 **numastat** 指令所列出的相同資訊（numa_hit、numa_miss、numa_foreign、interleave_hit、local_node、以及 other_node），加上更新的格式，單位為 MB。

`-p` ***pattern***

根據指定的樣式（pattern），顯示每個節點的記憶體資訊。如果 *pattern* 的值包括數字，**numastat** 會假設那是數字型態的程序 ID。反之，**numastat** 會從程序的命令列搜尋特定的樣式。

-p 選項之後的引數會被視為用來篩選的樣式。額外的樣式會加大篩選條件，而非縮小。

`-s`

根據 **total** 欄位，降冪顯示資料，讓最消耗記憶體的程序列在上面。

或者，您可以指定 *node*（節點），那麼表格就會根據 *node* 欄位來排序。使用此選項時， *node* 的值必須緊跟在 **-s** 選項之後，如以下所示：

```
numastat -s2
```

不要在此選項及其值之間，加入任何空白字元。

`-v`

顯示更詳盡的資料。亦即：多程序的程序資訊會以詳盡的方式列出。

`-V`

顯示 **numastat** 的版本資訊。

`-z`

忽略所顯示資訊中，任何欄、列的零。請注意，一些接近零而被四捨五入為零的值還是會被列出。

4.1.4. NUMA 關聯管理 daemon（numad）

numad 是自動的 NUMA 關聯（affinity）管理 daemon，監控系統中的 NUMA 拓樸與資源使用量，以動態改進 NUMA 資源的分配與管理，進而改善系統效能。

根據系統負載，**numad** 最高能將效能提昇 50%。要達成這目標，**numad** 會定期從 **/proc** 檔案系統存取資訊，以節點為單位監控可用的系統資源。接下來 daemon 會試著將重要程序放到擁有足夠記憶體與 CPU 資源的 NUMA 節點上，好將 NUMA 效能最大化。程序管理的現有門檻為單一 CPU 至少 50%，以及至少 300 MB 的記憶體。**numad** 會試圖維持資源的利用等級，並在需要於 NUMA 節點間移動程序時讓整個分配重新達到平衡。

numad 也提供了預先放置（pre-placement）的建議服務，透過多種工作管理系統，協助程序一開始綁定 CPU 與記憶體資源。不管 **numad** 是否在系統上以 daemon 的形態執行，預先放置的建議服務都是可用的。詳情請參閱 man page 中的 **-w** 選項，以了解預先配置的建議，指令為：**man numad**。

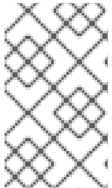
4.1.4.1. numad 的好處

numad 主要能加惠擁有長時間執行的程序之系統，這種程序會消耗大量資源，特別是包含在總系統資源中的子集合裡的程序。

numad 也許也能加惠消耗多 NUMA 節點的珍貴資源之應用程式。然而， **numad** 能提供的好處，會隨著系統上所消耗資源的比例逐漸增高而降低。

一支程式只執行幾分鐘，或不消耗太多資源時，**numad** 就不太可能會改善效能。擁有持續、不可預期的記憶體存取模式之系統，例如位於記憶體中的大型資料庫，也不太可能得益於 **numad**。

4.1.4.2. 操作模式



注意

大量合併 kernel 記憶體的統計資料後，這些資料也許會相互牴觸。因此，當 KSM daemon 合併大量記憶體時，**numad** 會感到混淆。然而，如果您的系統擁有大量閒置記憶體，關閉並停用 KSM daemon 會達到更高的效能。

numad 可以以兩種方式執行：

- 以服務方式執行
- 以可執行檔方式執行

4.1.4.2.1. 使用 numad 作為服務

numad 服務執行時，會根據系統的負載，動態調整系統。

要啟動服務，請執行：

```
# service numad start
```

要讓重新開機之後此服務依舊會執行，請執行：

```
# chkconfig numad on
```

4.1.4.2.2. 以執行檔方式執行 numad

要以執行檔方式執行 **numad**，請執行：

```
# numad
```

numad 會一直執行，直到停止為止。執行時的活動都會記錄在 **/var/log/numad.log** 檔案裡。

要限制 **numad** 管理特定程序，請以下列選項啟動：

```
# numad -S 0 -p pid
```

-p pid

將特定的 *pid* 明確加入清單中。此處指定的程序只有在達到 **numad** 程序的顯著門檻時，才會受到管理。

-S mode

-S 參數會指定程序掃描類型。設定為 **0** 會限制 **numad** 的管理方式為明確納入程序。

要停止 **numad**，請執行：

```
# numad -i 0
```

停止 **numad** 並不會移除用來改善 NUMA 關聯之改變。如果系統大幅使用這些改變，再次執行 **numad** 會在新條件下，調整關聯以改善效能。

欲知 **numad** 選項的進一步資訊，請參閱 **numad** 的 man page，指令為：**man numad**。

4.2. CPU 排程

「*排程器*」（scheduler）負責讓系統中的 CPU 保持忙碌。Linux 排程器有多種「*排程政策*」（scheduling policy），決定一組執行續何時會在某個特定 CPU 核心上執行多久時間。

排程政策共有兩大類：

1. 即時政策
   - SCHED_FIFO
   - SCHED_RR
2. 正常政策
   - SCHED_OTHER
   - SCHED_BATCH
   - SCHED_IDLE

4.2.1. 即時排程政策

即時執行續會先被排程，而正常的執行續會在所有即時執行續被排程後，才會被排程。

「*即時*」（realtime）政策用於極需時間的任務，必須一次完成，不被插斷。

SCHED_FIFO

這政策也稱為「*靜態優先順序排程*」（static priority scheduling），因為此政策為每個執行續定義了固定的優先順序（介於 1 到 99）。這個排程器會以優先順序掃描 SCHED_FIFO 執行續清單，並排程擁有最高執行順序的執行續以執行之。這個執行續會一直執行，直到被阻絕、退出、或被擁有更高優先順序的執行續所取代為止。

即使是最低權限的即時執行續，也比非即時的執行續擁有更高的權限；如果只有一個即時執行續，那麼 **SCHED_FIFO** 的優先值就無關緊要。

SCHED_RR

這是源自 **SCHED_FIFO** 政策的一種輪詢（round-robin）政策。**SCHED_RR** 執行續也會被賦予固定的權限值，介於 1 到 99。然而，擁有同樣權限值的執行續會以一固定時間量，又稱「時間配量」（time slice），用輪詢方式執行。**sched_rr_get_interval(2)** 系統呼叫會傳回這時間值，但這時間長度不能由使用者決定。如果您需要多執行續以同樣的優先順序執行時，此政策就非常有用。

欲知更多即時排程政策的已定義語法之資訊，請參閱〈*IEEE 1003.1 POSIX 標準*〉一節，此節位於「系統政策」–「即時」之下，網址為 http://pubs.opengroup.org/onlinepubs/009695399/functions/xsh_chap02_08.html。

定義執行續優先順序的最好方式，是從賦予低優先順序開始，然後只有在確定優先順序不足時，再予以增加。即時執行續並不像普通執行續一樣，會分配時間配量；**SCHED_FIFO** 執行續會一直執行，直到被阻絕、退出、或被擁有更高優先順序的執行續所取代為止。因此我們不建議將優先順序設為 **99**，因為這會讓該執行續的優先順序與轉移或監控執行續的一樣高。如果您的執行續進入運算上的迴圈，那麼後二者就會被阻絕、無法執行。在這情況下，單一處理器就會漸漸被鎖住。

在 Linux kernel 中，**SCHED_FIFO** 政策包括了限定最大頻寬的機制。這會保護即時應用程式師，避免即時應用任務獨占 CPU。這機制可以透過以下 **/proc** 檔案系統的參數來調整：

/proc/sys/kernel/sched_rt_period_us

定義多長時間該被視為 100% 的 CPU 頻寬，單位為 ms（「us」相當於「µs」）。預設值為 1000000µs，也就是 1 秒。

/proc/sys/kernel/sched_rt_runtime_us

定義要給執行中的即時執行續之時間，單位為 ms（「us」相當於「µs」）。預設值為 950000µs，也就是 0.95 秒。

4.2.2. 正常的排程政策

正常的排程政策有三種：**SCHED_OTHER**、**SCHED_BATCH**、以及 **SCHED_IDLE**。然而，**SCHED_BATCH** 與 **SCHED_IDLE** 政策都是給非常低優先順序的工作使用，因此本指南不會著墨太多。

SCHED_OTHER 或 SCHED_NORMAL

預設的排程政策。此政策使用了 CFS（完全公平排程器，Completely Fair Scheduler）好為使用此政策的所有執行續提供完全公平的存取時間。CFS 會建立動態的優先順序清單，部分根據每個執行續的 ***niceness*** 值。（此參數的詳細資料請參閱《*建置指南*》與 **/proc** 檔案系統。）這能給使用者一些非直接的控制等級，以控制程序的優先順序；但動態優先順序清單可以由 CFS 直接改變。

4.2.3. 選擇政策

為應用程式的執行續選擇正確的排程政策，並不是件直接了當的工作。一般來說，即時政策應該給重要、或非常需要時間的任務使用，以快速排程、同時不需要額外時間來執行。一般來說，正常政策的資料總傳輸量會比即時政策更佳，因為正常政策能讓執行續執行時更有效率（也就是說，不需要常常為了先佔式多工而重新排程。）

如果您管理大量執行續，又關心資料的總處理能力（每秒的網路封包數、寫入磁碟數等等），那麼請使用 **SCHED_OTHER**，讓系統管理 CPU 的使用率。

如果您關心的是事件的回應時間（或延遲時間），那麼請使用 **SCHED_FIFO**。如果執行續的數目不多，那麼請考慮隔離單一實體 CPU，並將執行續移到該 CPU 的核心上，不讓其它執行續在這些 CPU 核心上競逐時間。

4.3. 微調插斷與 IRQ

插斷請求（IRQ）是在硬體等級發送的服務請求。插斷可以由專職的硬體線、或橫跨硬體匯流排，以資訊封包（一種訊息插斷：MSI，Message Signaled Interrupt）的方式發出。

啟用插斷時，收到 IRQ 會提示一組開關發出插斷文本。kernel 插斷會發送程式碼，擷取 IRQ 的號碼及其相關的 ISR（插斷服務處理程式，Interrupt Service Routine），並依序呼叫每個 ISR。ISR 會確定插斷，忽略來自同樣 IRQ 的重複插斷，然後將延遲後的處理程式放入佇列中，以完成處理插斷，並避免 ISR 忽略未來的插斷。

/proc/interrupts 檔案會根據每個 CPU 及每個 I/O 裝置，列出插斷的數目。它會顯示 IRQ 編號、每個 CPU 核心處理的插斷數目、以及以逗號隔開的驅動程式清單（這驅動程式清單已註冊以收到該插斷）。（詳情請參閱 pro(5) 的 man page，指令為：**man 5 proc**。）

IRQ 有一組相關的「關聯」屬性：***smp_affinity***，這會定義允許為該 IRQ 執行 ISR 的 CPU 核心。這屬性可以透過指定插斷關聯與應用程式的執行續關聯，至一或多個指定的 CPU 核心，進而改善應用程式效能。這允許在特定的插斷與應用程式執行續之間，共享快取線。

特定 IRQ 編號的插斷關聯值儲存在相關的 **/proc/irq/*IRQ_NUMBER*/smp_affinity** 檔案裡，root 使用者可以加以檢視、修改。儲存在這檔案中的值是十進位的位元遮罩，表示系統上的所有 CPU 核心。

作為範例，要為一台有著四 CPU 核心的伺服器之乙太網路驅動程式設定插斷關聯，請先找出與該乙太網路驅動程式相關的 IRQ 編號：

```
# grep eth0 /proc/interrupts
32:   0     140      45       850264      PCI-MSI-edge      eth0
```

使用 IRQ 編號，找出正確的 ***smp_affinity*** 檔案：

```
# cat /proc/irq/32/smp_affinity
f
```

smp_affinity 的預設值為 **f**，表示 IRQ 可以接受系統上任何 CPU 的服務。設定這個值為 **1**（如下所示）表示只有 CPU 0 可以服務此插斷：

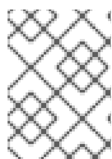
```
# echo 1 >/proc/irq/32/smp_affinity
# cat /proc/irq/32/smp_affinity
1
```

逗號可以用來加在多個 ***smp_affinity*** 的值之間，以分開 32 位元的群組。這在擁有超過 32 核心的系統上是必要的。例如以下範例就顯示，一台 64 核心系統上的所有核心都為 IRQ 40 提供服務：

```
# cat /proc/irq/40/smp_affinity
ffffffff,ffffffff
```

要只用 64 核心系統的前 32 個核心服務 IRQ 40，請使用以下指令：

```
# echo 0xffffffff,00000000 > /proc/irq/40/smp_affinity
# cat /proc/irq/40/smp_affinity
ffffffff,00000000
```



注意

在支援「*interrupt steering*」（操縱插斷）的系統上，修改 IRQ 的 ***smp_affinity*** 會設定硬體，讓使用特定 CPU 來服務插斷的決定會在硬體等級上決定，不需要 kernel 干預。

4.4. RHEL 6 中，NUMA 的加強功能

RHEL 6 包括了多種加強功能，以善用現今高度可擴充系統的完整潛力。本節簡介了 RHEL 6 中，NUMA 最重要的效能加強功能。

4.4.1. 優化空機系統與可擴充能力

4.4.1.1. 感知拓樸的加強功能

以下加強功能能讓 RHEL 偵測低階硬體與架構的詳細資料，改善系統自動優化的能力。

增強偵測拓樸的功能

這允許作業系統偵測開機時，低階硬體的詳細資料（例如邏輯 CPU、hyperthread、核心、插槽、NUMA 節點、以及節點間的存取時間），以及對系統處理作優化。

完全公平排程器

這種新的排程模式能確保所有程序皆平等地共享執行時間。將這排程器與偵測拓樸功能相結合，能讓程序納入同樣插槽上的 CPU 之排程中，避免遠端存取記憶體而效能不彰，並儘可能的保留快取內容。

`malloc`

現在 **`malloc`** 已經優化，確保分配給一個程序的記憶體區域，會在實體位置上儘可能的接近程序所執行的 CPU 核心。這能加快存取記憶體的速度。

分配 skbuff I/O 緩衝區

跟 **`malloc`** 類似，這功能也已優化，以使用在實體位置上靠近 CPU 處理 I/O 運作（例如插斷）時的記憶體。

裝置插斷關聯

裝置插斷關聯（device interrupt affinity）是裝置驅動程式記錄的資訊，哪個 CPU 處理哪個插斷的資訊可以用來限制同樣插槽上的 CPU 之插斷處理，保留快取的關聯性，限制插槽間的大量通訊。

4.4.1.2. 多處理器同步的加強功能

要在多處理器之間協同任務，需要頻繁、耗時的操作，以確保同步執行的程序不會破壞資料的完整性。RHEL 包括了以下加強功能，以改進這一部分的效能：

RCU 鎖

一般來說，90% 的鎖定都是為了達成唯讀的目的。RCU（Read-Copy-Update，讀取-複製-更新）鎖定機制，能移除存取非修改資料時，取得專一存取鎖定的需求。現在這鎖定模式可用在分配分頁快取記憶體時：現在鎖定模式僅用在分配或取消分配運作上。

針對單一 CPU 與單一插槽的演算法則

許多演算法則都已經更新，好在同樣插槽上的 CPU 進行鎖定協同作業時，允許更精細的鎖定。多種全域盤旋鎖（spinlock）已經被針對單一插槽的鎖定方式所取代，同時新的記憶體分配程式之區域、以及相關記憶體分頁清單，能讓記憶體分配邏輯在進行分配或取消分配運作時，更有效率地橫跨記憶體對應的資料結構之子集合。

4.4.2. 優化虛擬功能

因為 KVM 善用了 kernel 的功能，因此以 KVM 為基礎的虛擬化客座端會馬上得益於所有空機優化所帶來的好處。RHEL 也包括了多種加強功能，讓虛擬化客座端的效能可以直逼空機系統的效能。這些加強功能

專注在存取儲存裝置與網路時的 I/O 路徑，讓密集的系統負載（例如資料庫與檔案伺服功能）也能使用虛擬化建置環境。NUMA 對虛擬系統的加強功能包括：

釘選到 CPU

釘選到 CPU（CPU pinning）功能能將虛擬客座端固定在特定插槽上執行，以優化使用本地快取，同時移除插槽間通訊與存取遠端記憶體的繁瑣需求。

通透式巨型分頁

啟用了通透式巨型分頁之後，系統會對大量、連續的記憶體，自動進行感知 NUMA 的記憶體分配需求，降低競逐鎖定與 TLB（翻譯邊旁緩衝區，translation lookaside buffer）記憶體管理的運作次數，讓虛擬客座端的效能增加至多 20%。

以 kernel 為基礎的 I/O 實作

現在 kernel 中已實作了虛擬客座端的 I/O 子系統，大幅降低節點間通訊、與存取記憶體的時間，因其避免了大量的文本交換、以及同步與通訊的負載。

章 5. 記憶體

請參閱此章節，以取得有關於 RHEL 中的可用記憶體管理功能的相關資訊，並了解如何使用這些管理功能，來優化您系統上的記憶體使用率。

5.1. 巨型轉譯後備緩衝區

在記憶體管理中，有一部分是將實體記憶體的位址轉譯為虛擬記憶體位址。將實體位址對應至虛擬位置的對應關係之資料結構稱為「分頁表」。因為從分頁表讀取每個位址的對應關係會很花時間、也極耗資源，因此有份快取會儲存最常用的位址。這快取就稱為「轉譯後備緩衝區」（簡稱 TLB，Translation Lookaside Buffer）。

然而，TLB 能儲存的位址對應有限。如果要求的位址對應不在 TLB 裡面，那麼還是得讀取分頁表，以取得實體與虛擬位址的對應關係；這情況稱為「找不到 TLB」（TLB miss）。跟不太需要記憶體的應用程式比起來，需要大量記憶體的應用程式更受此情況所影響，這與這兩種應用程式對記憶體需求、及用來快取位址對應的分頁大小之相互關係有關。因為每次找不到 TLB 都牽涉到讀取分頁表，因此儘量避免這情況發生，就成為當務之急。

巨型 TLB（Huge TLB）能管理大區塊的記憶體，一次將更多位址對應放進快取中。這會降低找不到 TLB 的情況發生，進而改善需要大量記憶體的應用程式之效能。

關於配置巨型 TLB 的資訊，可以在 kernel 的文件中找到：**/usr/share/doc/kernel-doc-*version*/Documentation/vm/hugetlbpage.txt**。

5.2. 巨型分頁與通透式巨型分頁

記憶體是成區管理的，每個區塊稱為「*分頁*」（page）。每個分頁的大小是 4,096 位元組。1 MB 的記憶體等於 256 個分頁；1 GB 的記憶體等於 256,000 個分頁，餘此類推。CPU 擁有內建的「*記憶體管理單元*」（memory management unit），每個分頁都透過「*分頁表條目*」（page table entry）來參照得知。

要啟用系統以管理大量記憶體，有兩種方式：

- 在硬體的記憶體管理單元裡，增加分頁表的條目
- 增加分頁大小

第一種方法代價不菲，因為現代處理器中的硬體記憶體管理單元僅支援上百或上千個分頁表條目。除此之外，與成千上萬個分頁（數百萬位元組的記憶體）運作良好的硬體與記憶體管理的演算法則，並不一定與上百萬個分頁（甚至十億以上）的運作良好。這會導致效能上的問題：當應用程式需要使用超過記憶體管理單元所支援的記憶體分頁時，系統就會回到較慢、以軟體為基礎的記憶體管理模式，這會讓整台系統更慢。

RHEL 6 透過「*巨型分頁*」（huge page），採用第二種方法。

簡單來說，巨型分頁是大小為 2 MB 與 1 GB 的記憶體區塊。2 MB 分頁的分頁表適用於管理以 GB 為單位的記憶體大小；而 1 GB 分頁的分頁表適用於管理以 TB 為單位的記憶體大小。

巨型分頁必須在啟用時就指定。手動管理巨型分頁並不容易，同時常需要大量變更程式碼，讓使用上更有效率。因此，RHEL 6 採用了「*通透式巨型分頁*」（THP，transparent huge page）。THP 是將建立、管理、使用巨型分頁等工作自動化的萃取層。

THP 會隱藏使用巨型分頁時的複雜事，不讓系統管理者與程式設計師接觸到。因為 THP 的目標是改進效能，因此設計師（不管是來自社群還是 Red Hat）都在多種系統、配置、應用程式、與負載下，測試過 THP 並加以優化。這讓 THP 的預設值能改善大多數系統配置下的效能。

請注意 THP 目前只能對應匿名的記憶體區域，例如堆積與堆疊區域。

5.3. 使用 VALGRIND 側寫記憶體使用量

Valgrind 是一種為使用者空間的執行檔提供評測工具的架構。它有多種工具，可以用來側寫、分析程式效能。本節描述的工具能幫助使用者偵測記憶體錯誤，例如使用了未初始化的記憶體、不當分配、或取消分配記憶體，並加以分析。所有工具都包含在 valgrind 套件中，可以透過以下指令執行：

```
valgrind --tool=toolname program
```

請以您想要使用的工具名稱取代 *toolname*（以記憶體側寫工具來說，是 **memcheck**、**massif**、或 **cachegrind**），並以您想以 Valgrind 側寫的檔案名稱取代 *program*。請注意，使用 Valgrind 時，會讓程式執行速度比平常更慢。

Valgrind 的功能簡介位於〈節 3.5.3, "Valgrind"〉。欲知更多詳情，包括 Eclipse 中的可用嵌入程式之資訊，都位於《*開發指南*》中，網址為 https://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/?locale=zh-TW。相關文件可以在安裝了 valgrind 之後，透過 **man valgrind** 指令來檢視，或在以下位置找到：

- **/usr/share/doc/valgrind-*version*/valgrind_manual.pdf**、以及
- **/usr/share/doc/valgrind-*version*/html/index.html**。

5.3.1. 使用 memcheck 側寫記憶體使用量

memcheck 是預設的 Valgrind 工具，可透過 **valgrind *program*** 指令直接執行，不需要外加 **--tool=memcheck**。它會偵測、回報多種難以偵測、診斷的記憶體錯誤，例如不該發生的記憶體存取、使用未定義或未初始化的值、不正確的釋放（free）堆疊記憶體、重疊的指標、以及記憶體洩漏等。使用 memcheck 後，程式的執行時間會比平常慢 10 到 30 倍。

memcheck 會根據所偵測到的問題類型，傳回特定的錯誤。這些錯誤的詳細資料都列在 Valgrind 的文件中：**/usr/share/doc/valgrind-*version*/valgrind_manual.pdf**。

請注意，memcheck 只會回報這些錯誤 – 它無法避免這些錯誤發生。如果您的程式存取記憶體時會造成離散上的問題，那麼即使執行了 memcheck，這些問題還是會發生。然而，memcheck 會在錯誤發生之前，就把錯誤訊息記錄下來。

memcheck 提供了數種命令列選項，專注在檢查過程上。部份選項為：

--leak-check

啟用這選項時，memcheck 會在用戶端程式完成執行時，搜尋記憶體漏洞。預設值為 **summary**（摘要），會列出漏洞的數目。其它選項包括 **yes**（是）與 **full**（完整），兩者都會列出每個漏洞的詳細資料；而選項 **no**（否）會停用檢查記憶體漏洞的功能。

--undef-value-errors

啟用時（設定為 **yes**），memcheck 會在使用未定義的值時，回報錯誤。停用時（設定為 **no**），memcheck 不會回報未定義值的錯誤。預設上這功能是啟用的。停用這功能會稍稍加快 memcheck 的執行速度。

--ignore-ranges

允許使用者指定一或多組範圍，讓 memcheck 檢查位址的可用性時，予以忽略。多組範圍可用逗號隔開，例如 **--ignore-ranges=0xPP-0xQQ,0xRR-0xSS**。

欲知完整的選項清單，請參閱 **/usr/share/doc/valgrind-*version*/valgrind_manual.pdf**。

5.3.2. 以 Cachegrind 側寫快取使用量

Cachegrind 會模擬程式與電腦的快取階層、以及分支預測程式（選用）之互動。它會追蹤模擬後的第一階指令快取與資料快取的使用量，以偵測不良程式碼與此階快取之互動；它也會追蹤與最後一階（不管是第二或第三階）快取的使用量，以追蹤對主記憶體的存取。也因此，執行 Cachegrind 的程式之速度會比正常方式慢上 20 到 100 倍。

要執行 Cachegrind，請執行以下指令，並以您想要透過 Cachegrind 側寫的程式取代 *program*：

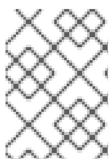
```
# valgrind --tool=cachegrind program
```

Cachegrind 可以為整個程式、以及程式中的每個函數，蒐集以下統計資料：

- 第一階指令快取的讀取（或已執行的指令）次數與讀取不到之次數，以及最後一階快取讀取不到指令的次數；
- 資料快取的讀取次數（或記憶體的讀取次數）、讀不到的次數、以及讀不到最後一階快取資料的次數；
- 資料快取的寫入次數（或記憶體的寫入次數）、寫不進的次數、以及寫不進最後一階快取資料的次數；
- 已執行與預測失敗的有條件分支之次數；以及
- 已執行與預測失敗的非直接分支之次數。

Cachegrind 會將這些統計資料的摘要資訊印至主控台，並寫入更多詳細的側寫資訊至檔案中（預設檔案為 **cachegrind.out.*pid***，其中 *pid* 是您執行 Cachegrind 的程式之程序 ID）。這檔案可以由相隨的 **cg_annotate** 工具進一步處理，如：

```
# cg_annotate cachegrind.out.pid
```



注意

cg_annotate 可以處理的每一行超過 120 個字元，端視路徑的長度而定。要讓這些資料更清楚易讀，我們建議您在執行這指令之前，將終端機視窗拉到可容納這長度的大小。

您也可以比較 Cachegrind 建立的側寫檔，透過圖表程式比較之前與之後的效能。您可以使用 **cg_diff** 指令，用第一個側寫檔取代 *first*，用之後的側寫檔取代 *second*：

```
# cg_diff first second
```

這指令會產生結合的檔案，請利用 **cg_annotate** 來檢視。

Cachegrind 支援多種專用於此輸出檔的選項。其中一些選項為：

--I1

指定第一階指令快取的大小、相關性、以及行的大小，並以逗號隔開：**--I1=*size*,*associativity*,*line size***。

--D1

指定第一階資料快取的大小、相關性、以及行的大小，並以逗號隔開：**--D1=*size*,*associativity*,*line size***。

--LL

指定最後一階指令快取的大小、相關性、以及行的大小，並以逗號隔開：**--LL=*size*,*associativity*,*line size***。

--cache-sim

啟用或停用蒐集存取快取成功的次數、與存取快取失敗的次數之功能。預設值為 **yes**（啟用）。

請注意，停用上述選項及 **--branch-sim** 會讓 Cachegrind 不蒐集任何資訊。

--branch-sim

啟用或停用蒐集分支指令與預測失敗次數之功能。預設上設為 **no**（停用），因為這功能會降低 Cachegrind 約 25% 的速度。

請注意，停用上述選項及 **--cache-sim** 會讓 Cachegrind 不蒐集任何資訊。

欲知完整的選項清單，請參閱 **/usr/share/doc/valgrind-*version*/valgrind_manual.pdf**。

5.3.3. 使用 Massif 側寫堆積與堆疊空間

Massif 會取得特定程式所使用的堆積空間；包括使用空間與其它用來記錄、調整所分配到的額外空間。這可以幫助您降低程式所使用的記憶體量，進而增加程式速度、降低程式耗盡電腦的置換空間的可能性。Massif 也提供了詳細資料，讓您得知程式的哪一部分取用了堆積記憶體。執行 Massif 的程式之執行速度會比正常情況慢約 20 倍。

要側寫程式的堆積使用量，請指定 **massif** 作為 Valgrind 所要使用的工具：

```
# valgrind --tool=massif program
```

Massif 所蒐集的側寫資料會被寫入至檔案中，預設檔案為 **massif.out.*pid***，其中 *pid* 是所指定的 *program*（程式）之程序 ID。

這側寫資料可透過 **ms_print** 指令來製作圖表，如：

```
# ms_print massif.out.pid
```

這產生的圖表會顯示程式執行期間的記憶體使用量，以及程式中在多個地方負責分配的點，包括分配記憶體極峰值的點。

Massif 提供了多種命令列選項，將結果輸出。可用選項包括：

--heap

指定是否要對堆積進行側寫。預設值為 **yes**（是）。把這選項設定為 **no**（否）將不會對堆積進行側寫。

--heap-admin

指定側寫堆積時，管理所需的區塊位元組數目。預設值為每個區塊 **8** 個位元。

--stacks
: 指定是否要對堆疊進行側寫。預設值為 **no**（停用）。要啟用對堆疊的側寫，請將此選項設為 **yes**（啟用），但請注意，這會大幅降低 Massif 的速度。也請注意 Massif 會假定主堆疊的起始大小為零，方便控制程式側寫時，堆疊部分的大小。

--time-unit
: 指定側寫時所用的時間單位。此選項可用的三種值為：所執行的指令（**i**），亦為預設值，適用於大部分情況；即時（**ms**，單位為 ms），可以用在某些情況下；以及分配 / 取消分配給堆積與 / 或堆疊的位元組（**B**），用於執行時間非常短的程式、以及測試環境，因為這選項非常容易在不同機器上執行。此選項在將 Massif 的結果用 **ms_print** 圖形化時，非常有用。

欲知完整的選項清單，請參閱 **/usr/share/doc/valgrind-*version*/valgrind_manual.pdf**。

5.4. 微調處理能力

請閱讀此節以了解記憶體、kernel、與檔案系統處理能力的概觀，以上三者的相關參數，以及調整這些參數的優劣。

要在微調時暫時設定這些值，請將要設定的值利用 echo 指令傳送到 proc 檔案系統中的正確檔案。例如要暫時設定 ***overcommit_memory*** 為 **1**，請執行：

```
# echo 1 > /proc/sys/vm/overcommit_memory
```

請注意，proc 檔案系統中用來設定參數的路徑，會因為系統而有所不同。

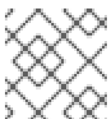
要永久設定這些值，請使用 **sysctl** 指令。欲知更多詳情，請參閱《*建置指南*》，網址為 https://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/?locale=zh-TW。

與處理能力相關的記憶體微調參數

以下所有參數都位於 proc 檔案系統中的 **/proc/sys/vm/** 裡。

overcommit_memory
: 定義接受或拒絕大型記憶體需求的狀況。此參數有三種可用值：

- **0** – 預設設定。kernel 會進行探索式的記憶體過度寫入處理，方法是預測記憶體的可用量、並讓公然違規的需求失敗。不幸的是，因為記憶體是以探索式、而非精準的演算法則來分配，因此這設定有時會過度使用系統上的可用記憶體。
- **1** – kernel 不進行記憶體過度寫入處理。在此設定下，過度使用記憶體的機會會增加，但對於頻繁存取記憶體的任務來說，效能也會增加。
- **2** – kernel 拒絕相等或大於總可用置換空間與實體記憶體比例（於 ***overcommit_ratio*** 指定）的記憶體需求。如果您想要降低記憶體過度寫入的風險，這是最佳設定。



注意

這設定建議用於置換空間大於實體記憶體的系統上。

overcommit_ratio

指定 ***overcommit_memory*** 設為 **2** 的時候，實體記憶體的比例。預設值為 **50**。

max_map_count

定義一組程序可使用的記憶體對應區域之最大值。在大部分情況下，預設值 **65530** 就很適合。如果應用程式需要對應多於此數目的檔案，請增加這個值。

nr_hugepages

定義 kernel 中配置的巨型分頁數。預設值為 `0`。只有在系統上擁有連續且足夠的實體記憶體時，才可以分配（或取消分配）巨型分頁。以此參數保留的分頁不能用做其它用途。欲知更多資訊，請參閱安裝於系統中的文件：**/usr/share/doc/kernel-doc-*kernel_version*/Documentation/vm/hugetlbpage.txt**。

與處理能力相關的 kernel 微調參數

以下所有參數都位於 proc 檔案系統中的 **/proc/sys/kernel/** 裡。

msgmax

定義訊息佇列中，任何單一訊息的最大允許大小，單位為位元組。這個值不能超過佇列的大小（***msgmnb***）。預設值為 **65536**。

msgmnb

定義單一訊息佇列的最大大小，單位為位元組。預設值為 **65536** 位元組。

msgmni

定義訊息佇列辨識子的最大數量（也因此是佇列的最大數量）。64 位元電腦的預設值為 **1985**；32 位元電腦的預設值為 **1736**。

shmall

定義系統上一次可以使用的共享記憶體總數，單位為位元組。64 位元電腦的預設值為 **4294967296**；32 位元電腦的預設值為 **268435456**。

shmmax

定義允許用於 kernel 的最大共享記憶體區段，單位為位元組。64 位元電腦的預設值為 **68719476736**；32 位元電腦的預設值為 **4294967295**。然而請注意，kernel 能支援的數量遠超過這個值。

shmmni

定義系統全域的共享記憶體區域之最大數量。64 與 32 位元的預設值均為 **4096**。

threads-max

定義系統全域中，kernel 一次可使用的執行續（任務）總數。預設值與 kernel 的 ***max_threads*** 值相同。使用的公式為：

```
max_threads = mempages / (8 * THREAD_SIZE / PAGE_SIZE )
```

threads-max 的最小值為 **20**。

與處理能力相關的檔案系統微調參數

以下所有參數都位於 proc 檔案系統中的 **/proc/sys/fs/** 裡。

aio-max-nr

定義位於所有非同步 I/O 情況的事件之最大允許數。預設值為 **65536**。請注意，改變這個值並不會預先分配任何 kernel 資料結構或改變其大小。

file-max

列出 kernel 分配的檔案處理之最大數量。預設值與 kernel 中的 ***files_stat.max_files*** 值相符，亦即設定為 **(mempages * (PAGE_SIZE / 1024)) / 10** 或 **NR_FILE**（RHEL 的值為 8192），兩者取其大者。如果可用的檔案處理不夠而發生錯誤，加大這個值可以解決問題。

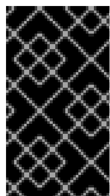
OOM Killer 的微調參數

OOM（記憶體不足，Out of Memory）指得是一種運算狀態，其中所有可用記憶體，包括置換空間，都已經分配出去。預設上，這種情況會導致系統當機並停止運作。然而，將 ***/proc/sys/vm/panic_on_oom*** 參數設為 **0**，會告訴 kernel 在 OOM 發生時，呼叫 **oom_killer** 函式。通常 **oom_killer** 會終結有問題的程序，系統便可正常運作。

以下參數可以針對單一程序來設定，讓使用者對於被 **oom_killer** 函式刪除的程序，進行更深的控制。它位於 proc 檔案系統的 **/proc/*pid*/** 之下，其中 *pid* 是程序 ID。

oom_adj

定義從 **-16** 到 **15** 之間的值，決定程序的 ***oom_score***。***oom_score*** 愈高，程序就愈有可能被 **oom_killer** 刪除。設定 ***oom_adj*** 的值為 **-17** 會取消 **oom_killer** 對於該程序之功能。



重要

任何調整過的程序之子程序，都會繼承父程序的 ***oom_score***。舉例來說，如果一組 **sshd** 程序不會被 **oom_killer** 函式所終結，那麼所有由該組 SSH session 所啟動的程序也不會被終結。發生 OOM 時，這會影響 **oom_killer** 函式挽救系統的能力。

5.5. 微調虛擬記憶體

通常使用虛擬記憶體的是程序、檔案系統快取、以及 kernel。要妥善使用虛擬記憶體，得依賴多種因素，而這些因素可以由以下參數調整：

swappiness

其值為 0 到 100，控制系統 swap（記憶體置換）的程度。值愈高表示效能優先，系統會積極地將非作用中的程序移出實體記憶體。值愈低則儘可能避免 swap，降低回應的延遲時間。預設值為 **60**。

min_free_kbytes

系統上保留的最低 swap 空間，單位為 KB。用來計算每個低記憶體區域的浮水印值，然後依照大小比例，指定保留記憶體分頁的值。

警告

設定此參數時請小心，過高或過低的值都會造成傷害。

將 ***min_free_kbytes*** 設定得太低，會讓系統無法取回記憶體。這會導致系統無法回應，並透過 OOM-killing 機制，終結多個程序。

然而，將這個值設得太高（系統總記憶體的 5-10%）會馬上導致系統記憶體不足。Linux 的設計，是使用所有可用記憶體來快取檔案系統的資料。將 ***min_free_kbytes*** 值設定得太高，會讓系統花去太多時間取回記憶體。

dirty_ratio

百分比數值。當需要變更的資料累積到系統總記憶體的此百分比時，就開始將需要變更資料寫出（透過 **pdflush**）。預設值為 **20**。

dirty_background_ratio

百分比數值。當需要變更的資料累積到系統總記憶體的此百分比時，就開始在背景將需要變更資料寫出（透過 **pdflush**）。預設值為 **10**。

drop_caches

將這個值設定為 **1**、**2**、或 **3** 會讓 kernel 放棄多種分頁快取與 slab 快取的組合。

1

系統會無效判定，並釋放所有分頁快取記憶體。

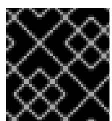
2

系統會釋放所有沒用到的 slab 快取記憶體。

3

系統會釋放所有分頁快取與 slab 快取記憶體。

這是非破壞性的操作。因為需要變更的物件不能被釋放，因此建議您在設定此參數之前，先執行 **sync**。



重要

不建議在正式投產的環境中，使用 ***drop_caches*** 來釋放記憶體。

要在微調時暫時設定這些值，請透過 echo 指令將想用的值傳至 proc 檔案系統中。例如要暫時設定 ***swappiness*** 為 **50**，請執行：

```
# echo 50 > /proc/sys/vm/swappiness
```

要永久設定這個值，請使用 **sysctl** 指令。欲知更多詳情，請參閱《*建置指南*》，網址為 https://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/?locale=zh-TW。

章 6. 輸入/輸出

6.1. 功能

RHEL 6 包含了數個 I/O 堆疊上的效能改善：

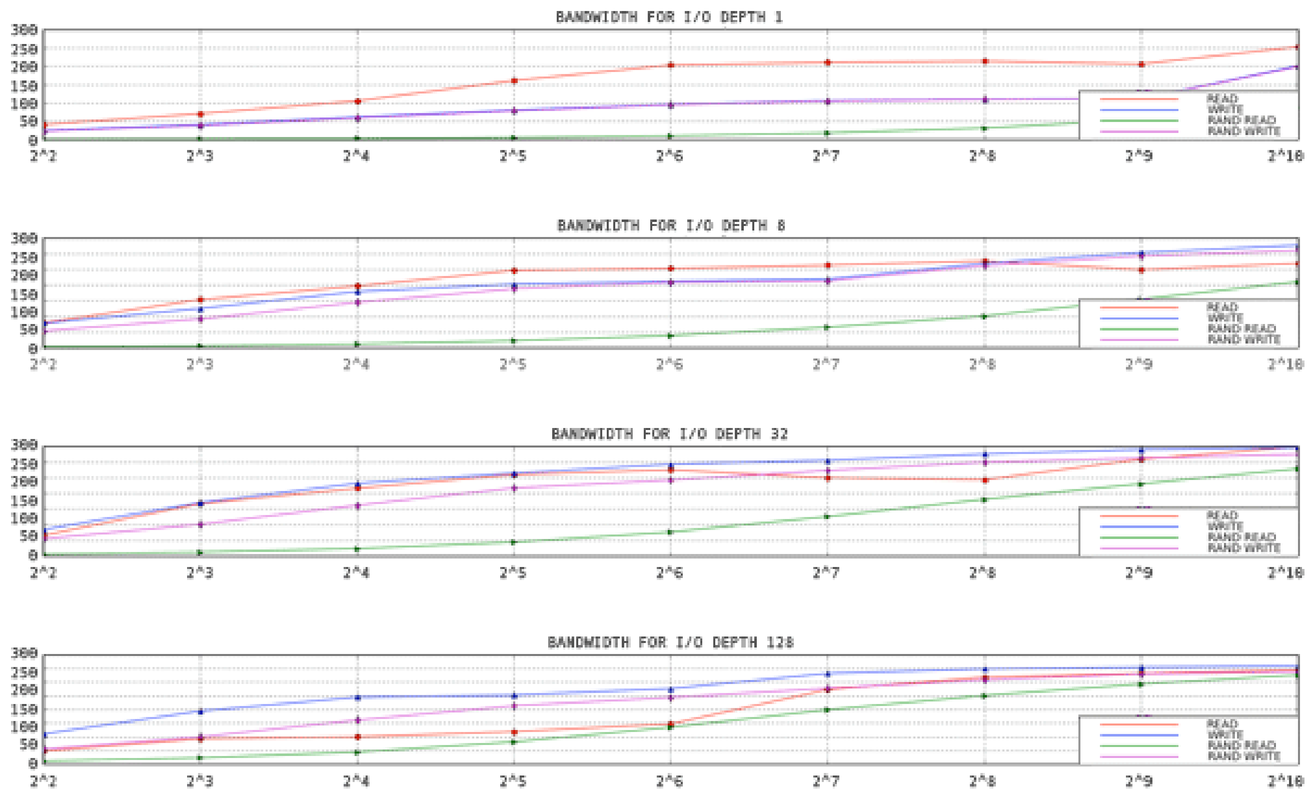
- 現在 RHEL 6 可以自動偵測固態硬碟（SSD），並且 I/O 排程器的效能也已經過微調，以有效善用這些裝置每秒所能執行的每秒高 I/O（IOPS）。
- 支援捨棄的功能已附加至 kernel 中，以向基礎儲存裝置回報未使用的區塊範圍。這有助於 SSD 的磨損平均（wear-leveling）演算法。它亦可藉由詳細監控實際的儲存空間使用量，來協助支援邏輯區塊佈建（一種儲存裝置的虛擬位址空間）的儲存裝置。
- 檔案系統屏障實作已在 RHEL 6.1 中大幅修改，以提升效能。
- **`pdflush`** 已被 per-backing-device 排清執行續取代，這大幅改善了擁有大量 LUN 配置的系統之擴充性。

6.2. 分析

欲成功微調儲存堆疊效能，您必須理解資料如何通過系統，並且熟悉儲存結構的基礎，以及它在各種工作環境下的效能如何。同時您亦必須對於欲微調的項目擁有詳細的理解。

每當您建置新的系統時，建議您由下而上分析儲存裝置。請從原生的 LUN 或磁碟開始，並使用直接 I/O（跳過 kernel 分頁快取的 I/O）來評估其效能。這是您所能進行的最基本測試，並且也會是您用來測量堆疊中的 I/O 效能的一項標準。請由基本的工作量產生程式開始（例如 **aio-stress**），它會產生循序且隨機的讀取和寫入，並利用各種 I/O 大小和佇列深度。

以下是源自於一系列 **aio-stress** 的圖表，每個項目皆會進行四個步驟：循序寫入、循序讀取、隨機寫入和隨機讀取。在此範例中，該工具已配置好以各種記錄大小（x 軸）和佇列深度（一個圖表一個）執行。佇列深度代表一段特定時間內，正在進行的 I/O 作業的總數量。



Y 軸顯示了頻寬/秒，單位為 MB。X 軸則顯示了 I/O 大小，單位為 KB。

圖形 6.1. aio-stress 對於 1 個執行續和 1 個檔案的輸出

請注意傳輸量的線由左下方角落升至右上方。也請注意無論記錄大小為何，您可藉由增加進行中的 I/O 之數量，以從儲存裝置取得更多傳輸量。

透過對存裝置執行這些基本工作，您將能大致得知儲存裝置在有工作負載時的效能如何。請保留這些測試所產生的資料，以便在進行更加複雜的工作量時，拿來進行比較。

若您會使用裝置對應程式或是 md，請新增此階層並重複您的測試。若效能明顯降低，請確認這是否屬於預期的情況，以及是否有方法能夠解釋此情況。比方說，若將 checksum raid 階層加入堆疊中，效能降低是可預期的。非預期的效能驟減可能是配置不當的 I/O 作業所造成的。就預設值，RHEL 會以優化的方式配置分割區和裝置對應程式的 metadata。然而，並非所有類型的儲存裝置皆會回報其最佳配置，因此可能需要進行手動微調。

增加了裝置對應程式或 md 層之後，請在區塊裝置上新增一個檔案系統，並針對這檔案系統（依然使用直接 I/O）進行測試。請再次拿測試結果與先前的測試進行比較，並確認您了解這之間的不同代表什麼。一般來說，直接的寫入 I/O 在預先分配的檔案上效能較佳，因此請確認您在測試效能之前，預先分配檔案。

有用的綜合工作量產生程式包含了：

- **aio-stress**
- **iozone**
- **fio**

6.3. 工具

可協助診斷 I/O 子系統效能問題的工具有數個，**vmstat** 提供了系統效能的粗略總覽。下列欄位與 I/O 最有關聯：***si***（swap in）、***so***（swap out）、***bi***（block in）、***bo***（block out），以及 ***wa***（I/O wait time）。當您的 swap 空間和您的資料分割區位於相同裝置上的時候，***si*** 和 ***so*** 有助於您使用它們來作為

整體記憶體壓力的指標。***si*** 和 ***bi*** 為讀取作業，而 ***so*** 和 ***bo*** 則為寫入作業，單位皆為 KB。***wa*** 為閒置時間；它會顯示等待 I/O 完成時，有哪些執行佇列被阻擋。

以 **vmstat** 分析您的系統能讓您檢視 I/O 子系統是否需要為任何的效能問題負責。 ***free***、***buff*** 和 ***cache*** 欄位也相當重要。***cache*** 的值會隨著 ***bo*** 的值增加，並且 ***cache*** 減少和 ***free*** 增加代表系統正在執行分頁快取的回寫和無效判定。

請注意，**vmstat** 所回報的 I/O 數量為所有裝置的所有 I/O 彙總。當您判斷 I/O 子系統中可能有效能問題時，您可透過 **iostat** 以詳細分析問題所在，這將會以裝置來細分 I/O 回報。您亦可截取其它詳細資訊，例如平均請求大小、每秒的讀取和寫入數量，以及正在進行的 I/O 合併數量。

要辨識您的儲存裝置效能時，透過使用平均請求大小和平均佇列大小（**avgqu-sz**），您可藉由產生的圖表來估算您儲存裝置的效能，請注意，若平均請求大小為 4KB，而平均佇列大小為 1，則傳輸量不太可能具有極佳的效能。

若效能數據與您所預期的效能不符，您可透過 **blktrace** 進行更加精細的分析。**blktrace** 工具程式套件能詳細提供花在 I/O 子系統中的時間有多少。**blktrace** 的輸出會是一組二進位的追蹤檔案，此檔案能藉由類似 **blkparse** 的工具程式進行後續的處理。

blkparse 為 **blktrace** 的姊妹工具程式。它會從 trace 讀取原生輸出，並產生一個速記的文字版本。

以下為 **blktrace** 的輸出範例：

```
8,64   3        1     0.000000000  4162  Q  RM 73992 + 8 [fs_mark]
8,64   3        0     0.000012707     0  m   N cfq4162S / alloced
8,64   3        2     0.000013433  4162  G  RM 73992 + 8 [fs_mark]
8,64   3        3     0.000015813  4162  P   N [fs_mark]
8,64   3        4     0.000017347  4162  I   R 73992 + 8 [fs_mark]
8,64   3        0     0.000018632     0  m   N cfq4162S / insert_request
8,64   3        0     0.000019655     0  m   N cfq4162S / add_to_rr
8,64   3        0     0.000021945     0  m   N cfq4162S / idle=0
8,64   3        5     0.000023460  4162  U   N [fs_mark] 1
8,64   3        0     0.000025761     0  m   N cfq workload slice:300
8,64   3        0     0.000027137     0  m   N cfq4162S / set_active
wl_prio:0 wl_type:2
8,64   3        0     0.000028588     0  m   N cfq4162S / fifo=(null)
8,64   3        0     0.000029468     0  m   N cfq4162S / dispatch_insert
8,64   3        0     0.000031359     0  m   N cfq4162S / dispatched a
request
8,64   3        0     0.000032306     0  m   N cfq4162S / activate rq,
drv=1
8,64   3        6     0.000032735  4162  D   R 73992 + 8 [fs_mark]
8,64   1        1     0.004276637     0  C   R 73992 + 8 [0]
```

如您所見，輸出過於密集並且難讀。您可得知哪項程序負責發出 I/O 至您的裝置，雖然有幫助，不過 **blkparse** 亦可在摘要中，提供格式易讀的額外資訊。**blkparse** 的摘要資訊列印在其輸出中的最後部分：

```
Total (sde):
Reads Queued:           19,       76KiB  Writes Queued:     142,183,
568,732KiB
Read Dispatches:        19,       76KiB  Write Dispatches:   25,440,
568,732KiB
Reads Requeued:          0               Writes Requeued:       125
Reads Completed:        19,       76KiB  Writes Completed:   25,315,
568,732KiB
```

```
Read Merges:              0,        0KiB  Write Merges:      116,868,
467,472KiB
IO unplugs:         20,087                Timer unplugs:          0
```

摘要顯示了平均的 I/O 速率、合併活動，並比較讀取與寫入的工作量。然而大部份情況下，**blkparse** 的輸出太過冗長，而沒有什麼作用。值得慶幸的是，您可使用數項工具來協助您視覺化處理這些資料。

btt 提供了 I/O 在不同 I/O 堆疊部分中，所花費的時間上的分析。這些部分包含了：

- Q – 區塊 I/O 已被排入佇列
- G – 取得請求

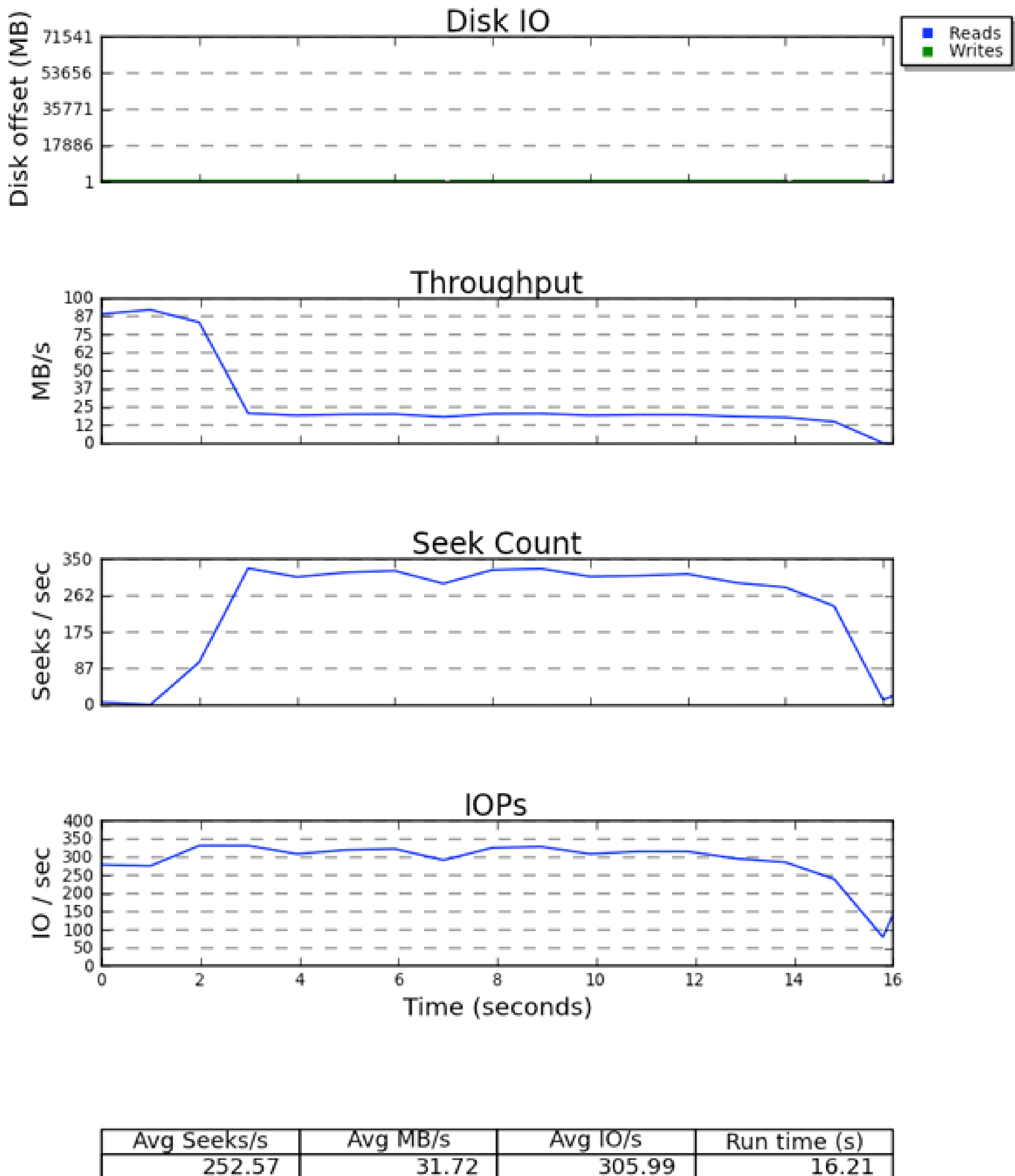
  新排入佇列的 I/O 不是會與既有請求合併的 I/O，因此已分配新的區塊層請求。
- M – 區塊 I/O 已與一項既有的請求合併。
- I – 請求已插入裝置的佇列中。
- D – 請求已傳送給裝置。
- C – 請求已由驅動程式完成。
- P – 區塊裝置佇列已插入，以讓請求能夠聚集。
- U – 裝置佇列未插入，這能讓已聚集的請求傳送給裝置。

btt 將花在各個這些部分中的時間，以及花在切換上的時間細分，例如：

- Q2Q – 請求傳送至區塊層所花費的時間
- Q2G – 從區塊 I/O 排入佇列開始到它被分配到一項請求之間所花費的時間
- G2I – 從請求被分配到它被插入裝置佇列之間所花費的時間
- Q2M – 從區塊 I/O 被排入佇列到它與既有請求合併之間所花費的時間
- I2D – 從請求被插入裝置佇列到它被實際傳送給裝置之間所花費的時間
- M2D – 從區塊 I/O 與既有請求合併到該請求被傳送給裝置之間所花費的時間
- D2C – 裝置請求的服務時間
- Q2C – 花費在請求的區塊層中的時間

您能夠從上述表格中推算出許多有關於工作量的相關資訊。比方說，若 Q2Q 比 Q2C 大上許多，這代表應用程式並未快速發出 I/O 請求。因此，任何您所遇上的效能問題，可能根本與 I/O 子系統無關。若 DC2 非常高，則代表裝置花上了許多時間為請求提供服務。這顯示該裝置可能負載過高（可能是因為它是共享的資源），或可能是因為傳送給裝置的工作之效能狀況不佳。若 Q2G 非常高，代表同一時間排入了大量請求。這顯示儲存裝置可能無法跟上 I/O 請求載入的速度。

最後，**seekwatcher** 會取用 **blktrace** 二進位資料，並產生一組資訊，包括邏輯區塊位址（Logical Block Address，LBA）、傳輸量、每秒的搜尋量，以及每秒的 I/O 量（IOPS）。



Avg Seeks/s	Avg MB/s	Avg IO/s	Run time (s)
252.57	31.72	305.99	16.21

圖形 6.2. seekwatcher 的範例輸出

所有圖表的 X 軸均為時間。LBA 圖顯示的讀取與寫入均以不同顏色表示。值得注意的是總處理能力（Throughput）與每秒搜尋時間（Seek Count）這兩者的關係。對於常進行搜尋的儲存裝置來說，這兩者成負相關。如果您發現有個裝置的總處理能力並不如預期，但您已經達到 IOPS 的限制，那麼 IOPS 圖表就很有用。

6.4. 配置

您需要作的第一個決策，是要使用哪一種 I/O 排程器。本節將對幾種主要排程器作簡介，看看何者最是用

於您的負載。

6.4.1. 完全公平佇列（CFQ）

CFQ 會根據啟動 I/O 請求的程序，為 I/O 請求進行公平的排程。排程類型有三種：即時（RT）、盡其所能（BE）以及閒置。您可透過 **ionice** 指令來手動指定排程類型，或是透過 **ioprio_set** 系統呼叫來進行程式式性的指定。就預設值，程序會被指定為「BE」排程類型。接著，「RT」與「BE」排程類型會更進一步被分成八個 I/O 優先順序，0 代表最高，7 則代表最低。排程類型為「RT」的程序，其排程性質一般會比「BE」或是閒置的程序要來得重要，因此所有排程為「RT」的 I/O 請求總是會在「BE」或是閒置的 I/O 請求之前進行。這代表擁有「即時」優先順序的 I/O 請求可能會使「BE」與閒置類型的 I/O 請求資源不足。「BE」排程為預設的排程類型，而「4」為此類型中的預設優先順序。閒置排程類型的程序，僅會在系統中沒有其它等待處理的 I/O 請求之後，才會被處理。因此請切記，只有在程序的 I/O 請求完全不會影響順向進度的情況下，才將該程序的 I/O 排程類型設為閒置。

CFQ 會藉由為各項執行 I/O 請求的程序指定時間配量，以達到公平佇列的效果。程序在被指定的時間配量之間，（預設值）能進行八項請求。排程器會根據紀錄資料，預先檢查應用程式是否在近期內會發出更多 I/O 請求。若預期程序將會發出更多 I/O 請求，則 CFQ 將會處於閒置狀態並等待該 I/O 請求（儘管有來自於其它程序的 I/O 請求正等待發出）。

CFQ 所執行的閒置動作，並不適合不會受到高度搜尋罰則（seek penalty）影響的硬體，例如高速的外部儲存裝置陣列或是固態硬碟。若需要在這類型的儲存裝置上使用 CFQ（比方說若您希望使用 cgroup 的等比例權重 I/O 排程器），您將需要微調部分設定，以改善 CFQ 的效能。請在位於 **/sys/block/*device*/queue/iosched/** 中，相同名稱的檔案裡，設置下列參數：

```
slice_idle = 0
quantum = 64
group_idle = 1
```

當 ***group_idle*** 設為 1 的時候，I/O 依然會有停止的可能性（基於後端儲存裝置因閒置而不忙碌）。然而，這些停止情況在系統中的所有佇列上發生的頻率，會比閒置的情況要來得低。

CFQ 是個非工作守恒（non-work-conserving）的 I/O 排程器，這代表就算有等待處理的請求，它也可處於閒置狀態中（如以上所討論）。將非工作守恒的排程器組合使用，可能會在 I/O 路徑上造成嚴重的遲緩。此類型的組合，就像是在一個基於主機的硬體 RAID 控制器上使用 CFQ。RAID 控制器本身可能會實作自己的非工作守恒排程器，也因此會造成組合中出現兩個層級的延遲。非工作守恒的排程器在擁有較多資料，並能根據這些資料做決定時，效能會較佳。若要組合這類型的演算法，最下面的排程器僅會看見上方的排程器所傳下的請求。因此，較低的層級所看見的 I/O 模式，並不代表實際的工作量。

微調參數

back_seek_max

反向搜尋一般會影響效能，因為與正向搜尋相較之下，它們可能會因為要重新調整位置，而導致更嚴重的延遲。然而，若是工作量不大，CFQ 還是會執行它們。此微調選項能控制 I/O 排程器允許進行反向搜尋的最大距離（單位為 KB）。預設值為 **16** KB。

back_seek_penalty

基於反向搜尋的低效率，各項反向搜尋皆有與其相聯的罰則。此罰則乃一項乘數；比方說，若有個磁頭位於 1024KB。假設佇列中有兩項請求，一個位於 1008KB 而另一個則為 1040KB。這兩項請求與目前的磁頭位置乃是等距離的。然而，在套用了反向搜尋罰則（預設值：2）之後，在磁碟後方位置上的請求，現在其距離便將會與先前的請求接近兩倍。因此，磁頭將會往前移。

fifo_expire_async

此微調選項能控制一項 async（緩衝寫入）請求能等待多久。在經過了這段有效時間（單位為 ms）之後，一項資源不足的 async 請求將會被移至分派清單中。預設值為 **250** ms。

fifo_expire_sync

對於同步（讀取和 O_DIRECT 寫入）請求來說，這與 fifo_expire_async 微調選項相同。預設值為 **125** ms。

group_idle

當設置後，CFQ 將會閒置在控制群組中，最後一個發出 I/O 請求的程序上。當使用等比例權重 I/O cgroup 時，這應設為 **1**，並將 ***slice_idle*** 設為 **0**（一般在高速的儲存裝置上會這麼做）。

group_isolation

若啟用了群組隔離（設為 **1**），它會藉以犧牲傳輸量來換取較強大的隔離機制。一般來講，若群組隔離被停用的話，公平機制則只會套用在循序的工作量上。啟用群組隔離，將能為循序和隨機的工作量同時提供公平機制。預設值為 **0**（停用）。欲取得更多資訊，請參閱 **Documentation/cgroups/blkio-controller.txt**。

low_latency

當啟用了低度延遲時（設為了 **1**），CFQ 會嘗試提供 300 ms 的等待時間最大值。這將會注重公平機制多過於傳輸量。停用低度延遲（設為 **0**）將會忽略目標延遲，並讓系統中的各項程序取得完整的時間配量。預設值將會啟用低度延遲。

quantum

定量（quantum）可控制 CFQ 在一段時間內會傳送至儲存裝置的 I/O 請求數量，基本上就是限制裝置的佇列深度。就預設值，這將會設為 **8**。儲存裝置可能能夠支援極深的佇列深度，不過增加 ***quantum*** 將會負面影響延遲，尤其是當有大量的循序寫入工作時。

slice_async

此微調選項可控制撥給各項發出非同步（緩衝寫入）I/O 請求之程序的時間配量。就預設值，它會被設為 **40** ms。

slice_idle

這指定 CFQ 在等待進一步需求時，應該閒置多久。RHEL 6.1 以前的預設值為 **8** ms。RHEL 6.2 以後的預設值為 **0**。這個 0 改進了外部 RAID 儲存裝置的總處理能力，方法是移除所有佇列與服務樹等級的閒置時間。然而，設定為 0 會降低內部非 RAID 儲存裝置的效能，因為這會增加整體搜尋數目。對於非 RAID 儲存裝置，我們建議使用大於 0 的值來設定 ***slice_idle***。

slice_sync

此微調選項可決定撥給多少時間配量給發出同步（讀取或直接寫入）I/O 請求的程序。預設值為 **100** ms。

6.4.2. 期限 I/O 排程器

期限 I/O（deadline I/O）排程器會試著對需求提供保證的延遲時間。值得注意的是，衡量延遲時間並不是從需求進入 I/O 排程器開始算起（這是重要的特徵，因為應用程式可能會進入睡眠狀態，等待需求描述子變為可用）。預設上，讀取的優先順序會高於寫入，因為應用程式更有可能會因為讀取 I/O 而延遲。

期限 I/O 排程器會批次處理 I/O；批次處理表示一系列、以 LBA 的順序（單向 elevator）增加的讀取或寫

入 I/O。每次批次處理後，I/O 排程器會檢查寫入的需求是否等待過久，決定是不是要開始新的讀取或寫入批次處理。需求的 FIFO 清單只有在每次批次開始時，會被檢查是不是已經過期，而且只檢查該批次的資料方向。因此，如果選擇的是寫入批次處理，同時有已過期的讀取需求，那麼只有寫入需求完成後，才會輪到讀取需求。

微調參數

fifo_batch

這會決定單一批次處理中，發出的讀取與寫入數量。預設值為 **16**。將這個值提高會讓總處理能力變高，但延遲時間也會變長。

front_merges

如果您確定工作負載永遠不會產生「前端合併」（front merge），您可以將這微調選參數設為 **0**。除非您已經測量過這項檢查的工作負荷，否則建議您使用預設設定（**1**）即可。

read_expire

這項微調參數能讓使用者設定單次讀取所接受的服務時間，單位為 ms。預設值為 **500** ms（半秒）。

write_expire

這項微調參數能讓使用者設定單次寫入所接受的服務時間，單位為 ms。預設值為 **5000** ms（五秒）。

writes_starved

此微調參數會控制在處理單一寫入批次處理前，能處理多少寫入批次處理。這個值愈高，能給寫入處理的參照項目就愈多。

6.4.3. Noop

Noop I/O 排程器使用的是 FIFO（先進先出，first-in first-out）演算法則。合併的需求會發生於一般區塊層，卻是簡單的最後猜中位置的快取。如果系統受限於 CPU 速度，而且儲存裝置夠快，那麼此排程器是最適合的 I/O 排程器。

以下是區塊層的可調整選項。

/sys/block/sdX/queue tunables

add_random

在一些情況下，I/O 事件的負載對 **/dev/random** 的亂數集區的影響，是可以測量的。在這種情況下，將這個值設為零較佳。

max_sectors_kb

就預設值，可傳送給磁碟的最大請求大小為 **512** KB。此微調選項可被使用來提升或降低這個值。最小值是以邏輯區塊大小所限制的；最大值則是以 ***max_hw_sectors_kb*** 限制。有些 SSD 在 I/O 大小超過了內部清除區塊大小時，效能會變差。在這種情況下，建議您將 ***max_hw_sectors_kb*** 微調成清除區塊的大小。您可透過使用一個例如 **iozone** 或是 **aio-stress** 的 I/O 產生程式來進行測試，並使用各種不同記錄大小（比方說由 **512** 位元組至 **1** MB）。

nomerges

這個微調選項有助於除錯。大部分工作負載都能得益於合併需求（即使在更快的儲存裝置，例如 SSD 上亦然）。然而在一些情況下，例如想要知道後端儲存裝置的 IOPS 數據時，會想要停用合併功能，而不停用預先讀取或進行隨機讀取。

nr_requests

每個需求佇列都有可分配給每個讀、寫 I/O 的需求描述子的限制。預設上，這個數值是 **128**，表示在同一個時間內，在把程序放入睡眠狀態之前，佇列中可以有 128 組讀取、以及 128 組寫入。放入睡眠狀態的程序是下一個要試著分配需求的程序，並不一定是已分配所有可用需求的程序。

如果應用程式不適合延遲，那麼不妨降低需求佇列的 ***nr_requests*** 值，並限制儲存裝置的指令佇列深度至低的值（甚至可以低到 **1**），這樣一來回寫 I/O 就無法分配所有可用需求描述子，並以寫入 I/O 填滿整個裝置佇列。一旦 ***nr_requests*** 已分配，所有試圖進行 I/O 的其它程序會被放入睡眠狀態，以等待可用需求出現。這會讓事情變得公平，因為接下來需求會以輪詢方式分配（而不是讓單一程序快速地連續消耗所有需求）。請注意，在使用期限排程器或 noop 排程器時，這不會是問題，因為預設的 CFQ 配置會在這種情況下進行保護。

optimal_io_size

在一些情況下，下層的儲存裝置會回報最適合的 I/O 大小。這在軟、硬體 RAID 之間最常見，因為最適 I/O 大小正是磁條的大小。如果回報了這個值，應用程式應該會儘可能發出適用於最適 I/O 大小的多組對應。

read_ahead_kb

當應用程式從檔案或磁碟循序讀取資料時，作業系統會偵測到此一行為。在這種情況下，作業系統會使用智慧型預先讀取的演算法則，藉以從磁碟中讀取比使用者要求更多的資料。這樣一來，當下一個使用者試著讀取資料時，資料已經位於作業系統的分頁快取中。潛在的壞處是，作業系統可能會從磁碟讀取超過所需的資料，佔據太多分頁快取的空間，對記憶體使用量造成壓力。在這種情況下，太多程序都發生錯誤的預先讀取，會對記憶體造成壓力。

對於裝置對應（device mapper）的裝置，將 ***read_ahead_kb*** 增加至較大的值，例如 **8192**，是個好主意。這是因為裝置對應的裝置通常是由多個底層裝置所構成。將這個預設值（**128** KB）乘以對應的裝置數量，是進行微調的適合起點。

rotational

傳統硬碟通常有著運轉的部件（例如轉盤）。然而 SSD 卻沒有。大部分 SSD 都標榜此一特性。然而，如果您發現有個裝置並不標榜此一特性，也許可以將此值手動設為 **0**；當停用此值時，I/O elevator 並不會使用降低搜尋時間的邏輯，因為在非運轉的裝置上進行搜尋，效能幾乎不會降低。

rq_affinity

發出 I/O 的 CPU 與完成 I/O 的 CPU 不一定是同一個。將 ***rq_affinity*** 設為 **1** 會導致 kernel 將完成 I/O 的動作，發至發出 I/O 的 CPU 上。這可以改善 CPU 資料快取的效能。

章 7. 檔案系統

請詳讀本章節，以取得有關於 RHEL 所支援使用的檔案系統，以及如何優化其效能的相關總覽。

7.1. 檔案系統微調考量

所有檔案系統皆有幾項共同的微調考量：系統上所選擇的格式化與掛載選項，以及可改善應用程式效能的動作。

7.1.1. 格式化選項

檔案系統區塊大小

區塊大小可在進行 **mkfs** 時選擇。可用的區塊大小範圍取決於系統本身：最大上限為主機系統的最大分頁大小，而最小限制則取決於所使用的檔案系統。預設的區塊大小適用於大部份的情況。

若您會建立許多比預設區塊大小還要小的檔案，您可設置較小的區塊大小，以儘可能縮小磁碟上所浪費的空間。然而請注意，設置較小的區塊大小可能會使檔案系統的最大大小受限，並且可能會造成額外的 runtime 負荷，特別是比所選區塊大小還要大的檔案。

檔案系統幾何分佈

若您的系統使用等量（striped）的儲存裝置（例如 RAID 5），您可在進行 **mkfs** 時，將資料和 metadata 與基礎儲存裝置幾何對稱，以改善效能。當使用軟體 RAID（LVM 或是 MD）和某些企業硬體儲存裝置時，這項資訊會被查詢並自動設置，不過在許多情況下，管理員必須在命令列上以 **mkfs** 來手動指定這項幾何分佈。

欲取得更多有關於建立和維護這些檔案系統上的相關資訊，請參閱《*儲存裝置管理指南*》。

外部日誌

工作量若包含大量 metadata，即代表日誌檔案系統（例如 ext4 和 XFS）的記錄檔部分會時常更新。若要儘可能減少檔案系統搜尋日誌的時間，您可將日誌放置在專屬的儲存裝置上。然而請注意，若將日誌放置在速度比主要檔案系統還要慢的外部儲存裝置上，使用該外部儲存裝置所得到的結果可能會不甚理想。

警告

請確認外部日誌有足夠的可靠性。失去外部日誌將造成檔案系統損毀。

執行 **mkfs** 時會建立外部日誌，掛載時會指定日誌裝置。欲取得更多資訊，請參閱 **mke2fs(8)**、**mkfs.xfs(8)** 和 **mount(8)** man page。

7.1.2. 掛載選項

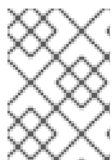
屏障

「寫入屏障」（write barrier）是個用來確保檔案系統 metadata 正確寫入永續性儲存裝置，並正確排序的 kernel 機制；即使停電，揮發性寫入快取的儲存裝置亦能確保資料無誤。啟用了寫入屏障的檔案系統亦可確保任何透過 **fsync()** 傳輸的資料皆能保有永續性（預防停電的狀況發生）。預設上，RHEL 會在所有支援寫入屏障的硬體上啟用屏障。

然而，啟用寫入屏障會使部分應用程式效能大幅降低；特別是大幅使用 **fsync()** 的應用程式，或是建立和刪除許多小型檔案的應用程式。如果您能接受沒有揮發性寫入快取的儲存裝置、或是因為停電而導致檔案不一致和資料遺失等情況，您可藉由使用 ***nobarrier*** 掛載選項來停用屏障。欲取得更多相關資訊，請參閱《*儲存裝置管理指南*》。

存取時間（noatime）

以前讀取檔案時，檔案的存取時間（***atime***）必須更新於 inode metadata 中，這包含了額外的寫入 I/O。若不需要正確的 ***atime*** metadata，請以 ***noatime*** 選項掛載檔案系統，以避免更新這些 metadata。然而在大部份情況下，基於 Red Hat Enterprise Linux 6 kernel 中的預設相對 atime（或是 ***relatime***）特性，***atime*** 不會是個極大的額外負荷。***relatime*** 特性僅會在先前的 ***atime*** 比修改時間（***mtime***）或是狀態更改時間（***ctime***）還要早的情況下更新 ***atime***。



注意

啟用 ***noatime*** 選項也會啟用 ***nodiratime*** 的特性；您無須同時設置 ***noatime*** 和 ***nodiratime***。

增加了對預先讀取的支援

「預先讀取」（read-ahead）會透過事先讀取資料，並將之載入分頁快取中（讓這些資料能預先存在記憶體中，而不用再由硬碟存取），以提升檔案存取的速度。某些工作量（例如那些包含了大量循序 I/O 串流的工作量）能有效受益於高度的預先讀取值。

使用 **tuned** 工具和等量的 LVM 能提升預先讀取的值，不過這對於部分工作量來說，並非永遠足夠。此外，RHEL 不一定總是根據檔案系統設置，適當的預先讀取值。比方說，若某個強大的儲存裝置陣列以單一強大的 LUN 的方式呈現給 RHEL，作業系統並不會把它視為是一個強大的 LUN 陣列，因此預設上也就不會完善使用此儲存裝置潛在的預先讀取效能。

請使用 **blockdev** 指令來檢視並編輯預先讀取值。若要檢視某個區塊裝置目前的預先讀取值，請執行：

```
# blockdev -getra device
```

若要修改該區塊裝置的預先讀取值，請執行下列指令。*N* 代表 512 位元組磁區的數量。

```
# blockdev -setra N device
```

請注意，透過 **blockdev** 指令選擇的值不會保有永續性。我們建議您建立一份 runlevel **init.d** script，以在開機時設置這個值。

7.1.3. 檔案系統維護

清除未使用的區塊

「批次清除」與「線上清除」皆為已掛載之檔案系統，用來清除未使用之區塊所進行的作業。這些作業亦適用於固態硬碟和精簡佈建的儲存裝置。

使用者可透過 **fstrim** 指令來執行「*批次清除作業*」（batch discard operation）。這項指令會清除檔案系統中，所有符合使用者指定條件的未使用區塊。只要檔案系統下的區塊裝置支援實體清除作業，RHEL 6.2 以上版本搭配 XFS 與 ext4 檔案系統就支援這兩項作業類型。若 **/sys/block/*device*/queue/discard_max_bytes** 的值非零的話，即代表實體清除作業受支援。

「*線上清除作業*」（online discard operation）可在進行掛載時透過 **-o discard** 選項（在 **/etc/fstab** 中或是作為 **mount** 指令的一部分）來指定，並在無使用者互動的情況下即時執行。線上清除作業僅會清除狀態由「已使用」切換為「可用」的區塊。線上清除作業在 RHEL 6.2 以上版本的 ext4

檔案系統上受到支援，並且也在 RHEL 6.4 以上的 XFS 檔案系統上受到支援。

Red Hat 建議您使用批次清除作業，除非批次清除對於系統的工作負載來說並不適用，或是您必須透過線上清除作業，以維持效能。

7.1.4. 應用程式考量

預先分派

ext4、XFS 和 GFS2 檔案系統皆支援透過 **fallocate(2)** glibc 呼叫，來進行高效率的空間預先分配功能。在某些情況下，檔案可能會因為寫入模式而過度分散，並間接影響讀取效能，這時空間預先分配就會是一項非常有幫助的技巧。預先分配功能會標記磁碟空間，在不寫入任何資料的情況下，使該空間猶如早已分配給了某個檔案。直到真實的資料寫入某個預先分配的區塊之前，讀取作業所回傳的值將會是零。

7.2. 檔案系統效能的設定檔

tuned-adm 工具能讓使用者輕易地在數個設定檔之間進行切換，這些設定檔主要設計來提升特定使用案例下的效能。對於提升儲存裝置效能來說，特別有幫助的設定檔包含了：

latency-performance

用來微調典型延遲效能的伺服器設定檔。它能停用 **tuned** 和 **ktune** 省電機制。**cpuspeed** 模式會改為 **performance**（效能）。而 I/O elevator 則會更改各項裝置的 **deadline**（期限）。***cpu_dma_latency*** 會註冊為 **0**（這是最低的延遲時間），以儘可能將電源管理服務品質的延遲限制住。

throughput-performance

用來進行傳輸量效能微調的伺服器設定檔。若系統未裝載企業級的儲存裝置，建議使用此設定檔。它與 **latency-performance** 相似，除了：

- ***kernel.sched_min_granularity_ns***（排程器的最小先佔精細程度）已設為 **10** 毫秒、
- ***kernel.sched_wakeup_granularity_ns***（排程器的甦醒精細程度）已設為 **15** 毫秒、
- ***vm.dirty_ratio***（虛擬機器的已變更比例）已設為 40%，並且
- 啟用通透式巨型分頁。

enterprise-storage

建議將此設定檔用於搭配了企業級儲存裝置（包括含備用電池的控制器快取保護，以及磁碟上的快取管理）的企業級伺服器配備。它與 **throughput-performance** 設定檔相似，除了：

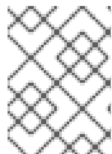
- ***readahead*** 值已設為 **4x**，並且
- 非 root/boot 的檔案系統會被以 **barrier=0** 重新掛載。

欲取得更多有關於 **tuned-adm** 的相關資訊，請參閱 man page（指令為 **man tuned-adm**），或是《*電源管理指南*》，網址為 https://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/?locale=zh-TW。

7.3. 檔案系統

7.3.1. ext4 檔案系統

ext4 檔案系統是 RHEL 5 中，預設的 ext3 檔案系統的可縮放延伸版本。ext4 現在已成為 RHEL 6 的預設檔案系統，並且最大可支援達 16 TB 的檔案系統大小，以及 16 TB 的單一檔案。它同時也移除了 ext3 中，32,000 個子目錄的限制。



注意

若檔案系統大於 16TB，我們建議使用可縮放的大型檔案系統，例如 XFS。欲取得更多相關資訊，請參閱〈節 7.3.2, "XFS 檔案系統"〉。

ext4 檔案系統的預設值，對於大部份工作量來說已足夠，然而若效能分析顯示了該檔案系統的特性會影響效能，您亦可使用以下幾個微調選項：

Inode 表初始化

對於非常大的檔案系統來說，**mkfs.ext4** 程序會花上大量時間，才能初始化該檔案系統中的所有 inode。使用者能藉由 ***-E lazy_itable_init=1*** 選項來遞延這項程序。若使用了此選項，在檔案系統被掛載之後，kernel 程序將會繼續將它初始化。這項初始化發生的頻率可透過 **mount** 指令的 ***-o init_itable=n*** 選項來進行控制，而執行這項背景初始化的所需時間約 1/n。 **n** 的預設值為 **10**。

Auto-fsync 特性

因為並非所有應用程式都能在重新命名、或是截斷和重新寫入一個既有檔案之後，正確進行 **fsync()**，因此預設上 ext4 會在進行了「replace-via-rename」（透過更名來取代）和「replace-via-truncate」（透過截斷來取代）作業後，自動同步檔案。此特性與較舊的 ext3 檔案系統特性一致。然而，**fsync()** 作業可能會很耗時，因此若不需要使用此自動特性的話，請搭配 ***-o noauto_da_alloc*** 選項執行 **mount** 指令，以將它停用。這代表應用程式必須明確地使用 **fsync()**，以確保資料的永續性。

日誌 I/O 優先順序

就預設值，寫入日誌 I/O 的優先順序，會比一般正常 I/O 要來的高。此優先順序能以 **mount** 指令的 ***journal_ioprio=n*** 選項來控制。預設值為 **3**。有效值的範圍為 0 到 7，0 代表優先順序最高的 I/O。

欲取得有關於其它 **mkfs** 和掛載選項上的相關資訊，請參閱 **mkfs.ext4(8)** 和 **mount(8)** man page，以及位於 kernel-doc 套件中的 **Documentation/filesystems/ext4.txt** 檔案。

7.3.2. XFS 檔案系統

XFS 是非常穩固，並擁有高度擴充性的單主機 64 位元日誌檔案系統。XFS 是完全基於磁區的檔案系統，因此它支援非常大的檔案以及檔案系統大小。XFS 檔案系統可容納的檔案數量限制，完全取決於檔案系統中的可用空間。

XFS 支援 metadata 日誌，這可提供更快速的當機復原速度。XFS 檔案系統亦可在被掛載且啟用時，進行磁碟重組和擴大。此外，RHEL 6 亦支援 XFS 特屬的備份和復原工具程式。

XFS 使用基於磁區的資源分配，並且提供了數項分配配置方案（例如延遲分配以及預先分配）。基於磁區的資源分配提供了較為精簡且高效率的追蹤方式，以追蹤檔案系統中已使用的空間，並透過減少磁碟分散以及 metadata 所耗用的空間，來改善大型檔案的效能。延遲分配可將檔案盡量寫入區塊的連續群組中，減少磁碟分散並改善效能。預先分配則是能在應用程式預先知道它需要寫入多少資料的情況下，用來有效避免磁碟分散。

XFS（藉由使用 b-tree 來將所有使用者資料和 metadata 放入索引中）提供了極佳的 I/O 擴充性。因為索引上的所有作業皆會繼承 b-tree 的對數可擴充特性，因此物件計數將會持續增加。XFS 在 **mkfs** 進行時所提供的某些微調選項，會隨著 b-tree 的寬度而改變，並且改變不同子系統的擴充性特質。

7.3.2.1. XFS 的基本微調

一般來講，XFS 的預設格式和掛載選項對於大部份工作量皆適用；除非特定配置變更能改善檔案系統的工作負載，否則 Red Hat 建議您使用預設值。若軟體 RAID 處於使用狀態的話，**mkfs.xfs** 指令會自動為自己配置正確的等量磁條單位與寬度，好與硬體對應。若您使用的是硬體 RAID，您可能將需要進行手動配置。

若檔案系統大小為數 TB，強烈建議使用 **inode64** 掛載選項，除非檔案系統是透過 NFS 匯出，而傳統 32 位元的 NFS 客戶端需要存取檔案系統。

若檔案系統時常需要進行修改，或是處於高載模式下，建議您使用 **logbsize** 掛載選項。預設值為 **MAX**（32 KB，記錄檔等量磁條單位），而最大大小為 256 KB。若檔案系統會大量進行修改，建議使用 256 KB 這個值。

7.3.2.2. XFS 進階微調

在更改 XFS 參數之前，您需要理解為何預設的 XFS 參數會造成效能上的問題。您亦必須熟悉您的應用程式，以及檔案系統針對這些應用程式的作業會有什麼樣的反應。

觀察得到、並透過微調得以修正或改善的效能問題，一般都是檔案系統中的檔案分散、或是資源競用所造成的。有數種用來找出這些問題的方式，並且在某些情況下，若要修正這些問題，與其修改檔案系統配置，不如修改應用程式本身。

若您先前尚未進行過此程序，建議您聯絡當地的 Red Hat 支援工程人員，以進行相關咨詢。

針對大量檔案的情況進行優化

XFS 會加入任意的限制，以限制某個檔案系統所能容納的檔案數量。就一般使用上來講，使用者通常不會達到或超過這項限制。若您一開始便知道預設限制過低，可透過 **mkfs.xfs** 指令來增加 inode 可使用的檔案系統空間百分比。若您在建立了系統後才遇上達到限制值的情況（通常在嘗試建立檔案或目錄時會出現一則 ENOSPC 錯誤，儘管您依然還有可用空間），您可透過 **xfs_growfs** 指令來調整這項限制。

針對單一目錄中的大量檔案進行優化

對於檔案系統來說，其目錄區塊大小永遠會是固定的，並且除非透過 **mkfs** 重新格式化，否則無法變更。最小目錄區塊為檔案系統的區塊大小，預設值為 **MAX**（4 KB，檔案系統區塊大小）。一般來講，您並無任何降低目錄區塊大小的理由。

因為目錄結構基於 b-tree，因此更改區塊大小會影響每項實體 I/O 所能截取或修改的目錄資訊數量。在特定區塊大小的情況下，目錄變得愈大，每項作業所需的 I/O 就會愈多。

然而，當正在使用較大的目錄區塊大小時（和在一個擁有較小目錄區塊大小的檔案系統上的相同作業相比之下），每項修改作業將會使用較多的 CPU 資源。這代表對於較小的目錄大小來說，較大的目錄區塊大小將會造成較低的修改效能。當目錄大小持續增加，直到 I/O 成為了效能限制的因素時，區塊大小較大的目錄的效能便會較佳。

預設配置的 4 KB 檔案系統區塊大小與 4 KB 的目錄區塊大小最適用於含有一至兩百萬個項目，並且每個項目名稱長度為 20 至 40 個位元組的目錄。若您的檔案系統需要更多的項目，較大的目錄區塊大小的效能似乎會較佳 - 16 KB 的區塊大小適用於含有一百至一千萬個目錄項目的系統，而 64 KB 的區塊大小則適用於含有超過一千萬個目錄項目的檔案系統。

若工作負載所使用的隨機目錄查詢多過修改（也就是說，相較之下目錄的讀取多過寫入，或是讀取比寫入更重要），那麼上述遞增區塊大小的閾值量級便要減一。

優化並行處理

和其它檔案系統不同，XFS 可同時執行許多類型的分配與解除分配作業（前提是這些作業必須發生在非共享的物件上）。磁區的分配與解除分配能同時發生，前提是並行的作業必須發生在不同的分配群組中。相同的，inode 的分配與解除分配亦可同時發生，前提是並行的作業會影響不同的分配群組。

當使用擁有高 CPU 數量和多執行續應用程式（進行並行作業）的機器時，分配群組的數量便會顯得相當重要。若只有四個分配群組，那麼持久性、平行的 metadata 作業的縮放比例，便只能受限於這四個 CPU（系統所提供的並行處理限制）。若檔案系統較小，請確認系統所提供的並行處理限制是否支援分配群組的數量。若是較大的檔案系統（幾十 TB 或更大），預設的格式化選項一般會建立足夠的分配群組，以避免造成並行處理上的限制。

應用程式必須注意單點競用，以使用 XFS 檔案系統結構中固有的平行處理原則。您無法並行修改一個目錄，因此會建立和移除大量檔案的應用程式應避免將所有檔案存放在單一目錄中。各個建立的目錄皆會置於不同的分配群組中，因此某些技巧（例如在多個子目錄之間雜湊檔案）和使用單一大型目錄相較之下，能提供擴充性較佳的儲存裝置模式。

優化使用延伸屬性的應用程式

若 inode 中有足夠的可用空間，XFS 可將小的屬性直接存放在 inode 中。若屬性能容納在 inode 中，它便能在無需透過額外 I/O 截取獨立屬性區塊的情況下，截取和修改。內置與非內置屬性的效能相較之下，非內置屬性的效能將會慢上一個量級。

當使用預設的 256 位元組 inode 大小時，可用來存放屬性的空間約有 100 個位元組（這取決於存放在 inode 中的資料磁區指標數量）。預設的 inode 大小其實僅適合存放少量的小型屬性。

在進行 mkfs 時增加 inode 大小可增加存放內置屬性的空間。一個 512 位元組的 inode 大小會將存放屬性的空間增加至約 350 個位元組；一個 2 KB 的 inode 則擁有約 1900 個位元組的可用空間。

然而，可內置的個別屬性皆有大小上的限制 - 屬性名稱與值的最大限制皆為 254 個位元組（也就是說一個名稱長度為 254 個位元組，以及值長度為 254 個位元組的屬性將可被內置）。若超過這些大小限制，屬性便無法內置，儘管有足夠空間足以將所有屬性存放在 inode 中。

為持久性的 metadata 變更進行優化

記錄檔的大小乃判斷持久性 metadata 能進行何種修改程度的主要因素。記錄裝置乃循環性的，因此在尾線能被覆寫之前，記錄檔中的所有變更皆必須寫入磁碟上的真實位置中。這可能需要耗費一段搜尋時間，以將所有需要變更的 metadata 寫回。一般就預設配置，記錄檔的大小會根據檔案系統的總大小設置，因此在大部份情況下您沒有必要微調記錄檔。

一個小型的記錄裝置也會造成非常頻繁的 metadata 回寫 - 記錄檔會持續地推掉其尾線，以釋出空間。因此時常遭到修改的 metadata 將會時常被寫入磁碟，造成作業緩慢。

增加記錄檔大小會增加尾線推送事件之間的時間。這可改善需要變更的 metadata 的彙總，並提供較佳的 metadata 回寫模式，並減少時常修改的 metadata 的回寫次數。然而，較大的記錄檔將需要更多的記憶體，以追蹤記憶體中的所有未處理變更。

若您有一部記憶體不大的機器，大型記錄檔會較不適用，因為此記憶體限制會造成系統無法善用大型記錄檔，便將 metadata 回寫。在這些情況下，小型的記錄檔所帶來的效能一般會優於大型的記錄檔，因為藉由用完記錄檔空間再將 metadata 回寫，會比藉由記憶體回收所驅動的回寫要來得有效率。

請每次都嘗試對應記錄檔以及包含了檔案系統之裝置的基礎等量磁條單位。`mkfs` 就預設值會擴充性 MD 和 DM 裝置進行此動作，不過對於硬體 RAID，您可能需要特別指定。正確進行此設置，可避免記錄檔 I/O 在將變更寫入磁碟中的時候，造成非對應的 I/O 和 read-modify-write（讀取 - 修改 - 寫入）作業。

記錄的作業亦可藉由編輯掛載選項來進行更進一步的改善。增加記憶體內的記錄緩衝區大小（`logbsize`）將會增加變更寫入記錄檔中的速度。預設的記錄緩衝區大小為 MAX（32 KB，記錄等量磁條單位），並且最大大小為 256 KB。一般來講，愈大的值會提供愈高的效能。然而，若 fsync 工作負載較高時，較小的記錄緩衝區的速度，會比對應了大型等量單位的較大緩衝區快上許多。

delaylog 掛載選項也會透過減少更改日誌檔案的次數，來改善持久性的 metadata 變更效能。它會透過在將個別變更寫入日誌之前，先將這些變更彙總至記憶體中：時常修改的 metadata 會週期性地被寫入日誌中，而非僅在每次進行修正時。此選項會增加記憶體使用量，以追蹤需要變更的 metadata，並增加當機時作業遺失的可能性，不過這可改善約一或更多個量級的 metadata 修改速度與擴充性。使用此選項搭配 **fsync**、**fdatasync** 或是 **sync** 來確認資料和 metadata 是否有寫入磁碟中，便可確保資料或是 metadata 的完整性。

7.4. 叢集

叢集儲存裝置會在叢集中的所有伺服器之間，提供一致性的檔案系統映像，這能讓伺服器讀取和寫入單一、共享的檔案系統。這能藉由將像是安裝和更新應用程式這般的任務，限制在單一檔案系統中，以簡化儲存裝置上的管理。一個叢集全域的檔案系統亦可省略應用程式資料的複件，簡化備份和災害復原。

Red Hat 的 High Availability 外掛提供了叢集儲存裝置結合 Red Hat Global File System 2（屬於 Resilient Storage 外掛的一部分）。

7.4.1. 全域檔案系統 2

全域檔案系統 2（GFS2，Global File System 2）是個直接與 Linux kernel 檔案系統結合的原生檔案系統。它能讓叢集中的多部電腦（節點）同時共享相同的儲存裝置。GFS2 檔案系統能自行微調，亦可透過手動微調。此部分詳述了嘗試手動式微調時，所需考量的效能相關注意事項。

RHEL 6.4 改善了 GFS2 中的檔案分散管理。在 RHEL 6.3 以前的版本上，若超過一項程序同時寫入多個檔案，所建立的檔案較容易分散。此分散情況會降低速度，特別是包含大型檔案的工作量。在 RHEL 6.4 上，同時寫入時所會發生的檔案分散情況較少，因此共作量的效能也會較佳。

在 RHEL 中，GFS2 沒有磁碟重組工具，但您可以使用 **filefrag** 工具找出各個檔案，將其複製為暫存檔案，然後重新命名暫存檔案並取代原有檔案，以重組它們。（只要循序寫入，此步驟亦可在 RHEL 6.4 之前的版本中完成。）

因為 GFS2 使用一種全域鎖定機制，且此機制需要叢集中的節點能夠進行通訊，因此您的系統若能避免在這些節點之間發生檔案和目錄的競用，便能達到最佳的效能。可避免競用的方式有：

- 儘可能使用 **fallocate** 來預先分配檔案和目錄，以優化分配程序並避免需要鎖定來源分頁。

- 減少共享於多個節點之間的檔案系統區域，以減少跨節點的快取無效判定，並改善效能。比方說，若有多個節點掛載了相同的檔案系統而存取不同的子目錄，將其中一個子目錄移至另一檔案系統上，可能能夠取得較佳的效能。

- 選擇最佳的資源群組大小與數量。這取決於典型的檔案大小和系統上的可用空間，並影響多個節點是否會嘗試同時使用某個資源群組的機率。資源群組若過多，尋找分配空間時可能會使區塊分配變得緩慢。若資源群組過少，則可能會在解除分配時造成鎖定競用。一般建議測試多項不同配置，以判斷何種配置適用於您的工作量。

然而，競用並非唯一可影響 GFS2 檔案系統效能的問題。其它可改善整體效能的最佳做法包含了：

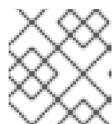
- 根據來自叢集的預期 I/O 模式，以及檔案系統的效能需求選擇您的儲存裝置硬體。

- 儘可能使用固態儲存裝置，以減少搜尋時間。

- 為您的工作量建立適當大小的檔案系統，並確保檔案系統的使用量絕不會超出 80%。較小的檔案系統的備份時間相對之下也會較短，並且進行檔案系統檢查時所需的時間與記憶體也較少，然而若檔案系統對於其工作量來說過小，則容易發生檔案高度分散的情形。

- 當工作量需要使用到大量 metadata，或是當日誌資料使用中的時候，請設置較大的日誌大小。雖然這會使用到較多的記憶體，卻也會提升效能，因為將會有更多可存放資料的日誌空間。

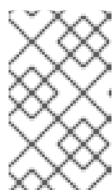
- 確認 GFS2 節點上的時鐘已同步，以避免網路應用程式發生問題。我們建議使用 NTP（網路時間協定）。
- 除非檔案或是目錄的存取時間對於您的應用程式來說相當重要，否則請以 **`noatime`** 和 **`nodiratime`** 掛載選項來掛載檔案系統。



注意

Red Hat 強烈建議您搭配 **`noatime`** 選項使用 GFS2。

- 若您需要使用配額，請嘗試減少配額同步交易的頻率，或是使用模糊配額同步，以避免持續的配額檔案更新，造成效能上的問題。



注意

模糊配額計量能讓使用者和群組稍微超用其配額限制。為了避免此問題發生，GFS2 會在某個使用者或群組即將超用其配額限制時，動態式地減少同步的時間。

欲取得更多有關於各項 GFS2 效能微調的詳細資訊，請參閱《*全域檔案系統 2*》指南，網址為 https://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/?locale=zh-TW。

章 8. 網路

RHEL 的網路堆疊已與時俱進，升級了數項自動優化功能。對於大部份工作量來說，自動配置的網路設定能提供優化的效能。

在大部份情況下，網路效能問題實際上是因為硬體功能發生問題，或是設備發生了錯誤。這些問題的討論範圍不包含在本指南中；本章節提到的效能問題和解決方法，僅適用於優化可正常運作的系統。

網路是個專門的子系統，它包含了諸多不同的部分以及敏感的連線功能。這也是為什麼自由軟體社群與 Red Hat 耗費了許多心力，實作了各項自動優化網路效能的方式。正因如此，您可能永遠也無須重新配置網路效能。

8.1. 網路效能提升

RHEL 6.1 提升了下列網路效能：

8.1.1. 接收封包操控（Receive Packet Steering，RPS）

RPS 會啟用一個 NIC **rx** 佇列，使其接收 **softirq** 的工作量散佈在數個 CPU 之間。這可避免網路流量在單一 NIC 硬體佇列上遇上瓶頸。

若要啟用 RPS，請在 **/sys/class/net/*ethX*/queues/*rx-N*/rps_cpus** 中指定目標 CPU 的名稱，請將 ***ethX*** 取代為 NIC 的相應裝置名稱（例如 **eth1**、**eth2**），並將 ***rx-N*** 取代為指定的 NIC 接收佇列。這便能讓檔案中指定的 CPU 處理來自於 ***ethX*** 上，***rx-N*** 佇列中的資料。當指定 CPU 時，請考量佇列的「*快取關聯*」（cache affinity）[4]。

8.1.2. 接收流量操控（Receive Flow Steering，RFS）

RFS 乃 RPS 的延伸，它能讓管理員配置一個雜湊表，此雜湊表會在應用程式接收資料和被網路堆疊查閱時，自動填入。這能判斷哪個應用程式會接收哪些網路資料（根據 source:destination 網路資訊）。

透過使用這項資訊，網路堆疊可排程最適合的 CPU 接收各個封包。若要配置 RFS，請使用下列微調項目：

/proc/sys/net/core/rps_sock_flow_entries

這可控制 kernel 能轉至特定 CPU 的最大 socket/流量數。這是系統全域的共享限制。

/sys/class/net/*ethX*/queues/*rx-N*/rps_flow_cnt

這能控制 kernel 在一張網路卡（***ethX***）上，所能轉至特定接收佇列（***rx-N***）的最大插槽/流量數量。請注意，此微調項目所有各佇列的值的加總，在所有網路卡上皆應與 **/proc/sys/net/core/rps_sock_flow_entries** 的值相等或更低。

和 RPS 不同，RFS 允許接收佇列和應用程式在處理封包流量時，共享相同的 CPU。這在某些情況下可提升效能。然而，此效能改善取決於快取階層、應用程式負載等等。

8.1.3. TCP 精簡串流的 getsockopt 支援

Thin-stream（精簡串流）是個用來描繪某種傳輸協定的名詞，在其中，應用程式會以極低的速率傳送資料，因此此協定的重新傳輸機制不處於飽和狀況下。使用精簡串流協定的應用程式一般會透過 TCP 之類的可靠協定進行傳輸；大部份情況下，此類型的應用程式會提供對於時間非常敏感的服務（比方說股市交易、線上遊戲、控制系統等服務）。

對於時間敏感的服務來說，封包遺失可能會造成極大的影響。為了避免此問題，**getsockopt** 已改善以支援兩項額外選項：

TCP_THIN_DUPACK

此布林值能在進行了一項精簡串流的 dupACK 之後，啟用重新傳輸的動態觸發。

TCP_THIN_LINEAR_TIMEOUTS

此布林值會動態觸發精簡串流的線性逾時。

這兩項選項皆能透過應用程式具體地啟用。欲取得更多有關於這些選項的相關資訊，請參閱 **file:///usr/share/doc/kernel-doc-*version*/Documentation/networking/ip-sysctl.txt**。欲取得更多有關於精簡串流上的相關資訊，請參閱 **file:///usr/share/doc/kernel-doc-*version*/Documentation/networking/tcp-thin.txt**。

8.1.4. 支援通透式代理

Kernel 現在已能處理非本機綁定的 IPv4 TCP 和 UDP socket，以支援通透式代理（TProxy，transparent proxy）。若要啟用此功能，您需要配置相應的 iptables。同時，您也必須正確啟用和配置政策路由。

欲取得更多有關於通透式代理的相關資訊，請參閱 **file:///usr/share/doc/kernel-doc-*version*/Documentation/networking/tproxy.txt**。

8.2. 優化網路設定

效能微調一般是以先佔的方式來進行的。通常，我們會在執行一項應用程式或是建置一部系統之前，調整已知的變數。若這項調整沒有任何效應，我們便嘗試其它變數。此構思的邏輯是：「*就預設值*」來說，系統並非以優化的狀態在運作；也因此，我們「*認為*」我們需要相應地調整系統。在某些情況下，我們可透過導出計算猜測來進行調整。

如先前所提到的，網路堆疊通常會自行優化。此外，若要有效微調網路，您必須擁有詳細的理解，不僅是網路堆疊如何運作，還必須知道特定系統的網路資源需求。錯誤的網路效能配置反而會影響並降低效能。

比方說，請考量 「*bufferfloat 問題*」。增加緩衝區佇列的深度，可能會造成 TCP 連線擁有比連結所允許還要大的壅塞視窗（因為過深的緩衝行為）。然而，這些連線也擁有極大的 RTT 值，因為 frame 花上了太多時間在佇列中。因為無法偵測壅塞，這反而無法帶來優化的效能。

有關於網路效能，「*除非*」發生了明顯的效能問題，否則建議您保留預設設定。此問題包含了 frame 遺失、傳輸量明顯降低等等。儘管如此，最佳的解決方式通常是擴充性問題進行精密檢查而衍生出的解決方法，而非僅是將設定往上微調（增加緩衝區/佇列長度、降低插斷延遲等等）。

若要正確診斷網路效能問題，請使用下列工具：

netstat

會印出網路連線、路由表、介面卡數據、偽裝連線和 multicast 成員的命令列工具程式。它會由 **/proc/net/** 檔案系統截取有關於網路子系統的資訊。這些檔案包含了：

- **/proc/net/dev**（裝置資訊）
- **/proc/net/tcp**（TCP socket 資訊）
- **/proc/net/unix**（Unix 區域 socket 資訊）

欲取得更多有關於 **netstat** 及其位於 **/proc/net/** 的參照檔案的相關資訊，請參閱 **netstat** 的 man page，指令為：**man netstat**。

dropwatch

這是個用來監控 kernel 所捨棄的封包的監控工具程式。欲取得更多資訊，請參閱 **dropwatch** 的 man page，指令為：**man dropwatch**。

ip

這是個用來管理和監控路由、裝置、政策路由以及通道的工具程式。欲取得更多資訊，請參閱 **ip** 的 man page，指令為：**man ip**。

ethtool

這是個用來顯示和更改 NIC 設定的工具程式。欲取得更多資訊，請參閱 **ethtool** 的 man page，指令為：**man ethtool**。

/proc/net/snmp

這是個顯示了 **snmp** 代理程式管理 IP、ICMP、TCP 和 UDP 所需要的 ASCII 資料。它亦可顯示即時的 UDP-lite 數據。

《*SystemTap 初學者指南*》包含了數個範例 script，您可使用它們來設定監控網路效能。網址為 https://access.redhat.com/site/documentation/Red_Hat_Enterprise_Linux/?locale=zh-TW。

在搜集了有關於網路效能問題的資料後，您便應該能夠制訂一套理論 – 以及相應的解決方法。[5]比方說，在 **/proc/net/snmp** 中的 UDP 輸入錯誤若增加的話，即代表當網路堆疊嘗試將新的 frame 排入一個應用程式的 socket 時，一個或更多個 socket 接收佇列已滿。

這代表封包*至少*在一個 socket 佇列上發生了問題，即代表 socket 佇列排出封包的速度太慢，或是封包量對於該 socket 佇列來說太大。若是後者的話，請查看所有網路使用量較大的應用程式使用記錄中，是否有遺失的資料 -- 若要解決此問題，您需要優化或是重新配置該應用程式。

8.2.1. Socket 接收緩衝區大小

Socket 的傳送和接收大小皆是以動態的方式進行調整的，因此您很少需要手動式編輯。若進階分析（例如 SystemTap 網路範例中所使用的分析 **sk_stream_wait_memory.stp**）顯示 socket 佇列的排出率過慢，您可嘗試增加應用程式的 socket 佇列的深度。若要這麼做，請藉由配置下列其中一個值，以增加 socket 所使用的接收緩衝區大小：

rmem_default

這是個用來控制 socket 所使用的接收緩衝區「*預設*」大小的 kernel 參數。若要進行相關配置，請執行下列指令：

```
sysctl -w net.core.rmem_default=N
```

請將 ***N*** 取代為您希望使用的緩衝區大小（位元組）。欲判定此 kernel 參數的值，請參閱 **/proc/sys/net/core/rmem_default**。請注意，**rmem_default** 的值不該超過 **rmem_max**（**/proc/sys/net/core/rmem_max**）；若需要超過這個值，請增加 **rmem_max** 的值。

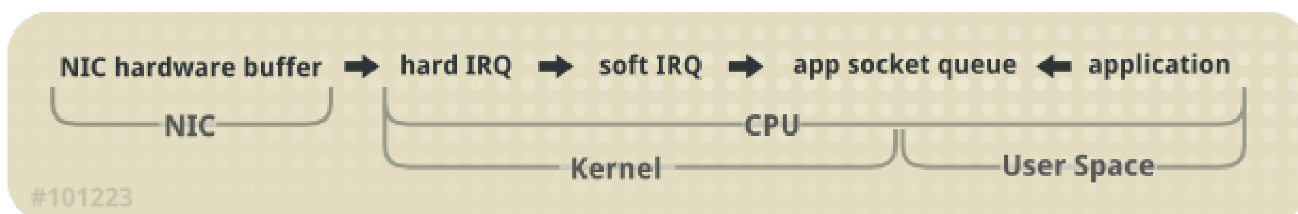
SO_RCVBUF

這是個用來控制 socket 的接收緩衝區「*最大*」大小的 socket 選項（位元組）。欲取得更多有關於 **SO_RCVBUF** 的相關資訊，請用以下指令參閱 man page：**man 7 socket**。

若要配置 `SO_RCVBUF`，請使用 `setsockopt` 工具程式。您可透過 `getsockopt` 截取目前的 `SO_RCVBUF` 值。欲取得更多有關於使用這兩個工具程式的相關資訊，請參閱 `setsockopt` 的 man page，指令為：`man setsockopt`。

8.3. 封包接收總覽

若要有效分析網路瓶頸和效能問題，您必須理解封包接收如何運作。封包接收（packet reception）對於網路效能微調來說相當重要，因為網路的 frame 通常會在接收路徑中遺失。在接收路徑中遺失的 frame，將造成顯著的網路效能影響。



圖形 8.1. 網路接收路徑圖形

Linux kernel 會接收各個 frame，並進行以下四個步驟的程序：

1. *硬體接收*：「*網路介面卡*」（NIC）會接收到線路上的 frame。根據其驅動程式配置，NIC 會將 frame 傳輸至一個內部硬體緩衝記憶體上，或至一個指定的信號緩衝區（ring buffer）上。

2. *Hard IRQ*：NIC 會藉由插斷 CPU 來判斷網路 frame 是否存在。這會使 NIC 驅動程式被通知插斷，並排程「*soft IRQ 作業*」。

3. *Soft IRQ*：此階段會實作實際的 frame 接收程序，並且以 `softirq` 執行。這代表此階段會先佔所有在指定 CPU 上執行的應用程式，但還是允許插入 hard IRQ。

   在此情況下（在與 hard IRQ 相同的 CPU 上執行，藉此減少鎖定額外負荷），kernel 會實際將 frame 由 NIC 硬體緩衝區上移除，並透過網路堆疊來處理它。在此，frame 會被轉送、丟棄或是傳送至一個目標監聽 socket 上。

   當傳送至一個 socket 時，frame 會被附加至擁有此 socket 的應用程式。這項程序會反覆進行，直到 NIC 硬體緩衝區的 frame 耗盡，或是直到達到「*裝置權重*」（`dev_weight`）值為止。欲取得更多有關於裝置權重的相關資訊，請參閱〈節 8.4.1, “NIC 硬體緩衝區”〉。

4. *應用程式接收*：應用程式會透過標準的 POSIX 呼叫（`read`、`recv`、`recvfrom`），來接收 frame，並清除任何所屬 socket 中的佇列。在此情況下，透過網路接收的資料將不再存在網路堆疊上。

8.3.1. CPU/快取關聯

若要維持接收路徑上的高傳輸量，建議您維持 L2 快取的「*速度*」。如先前所描述，網路緩衝區會在與偵測到它們的 IRQ 相同的 CPU 上被接收。這代表緩衝區資料將會存在該接收 CPU 的 L2 快取上。

若要有效善用，請為應用程式設置程序關聯，這些應用程式乃在與 L2 快取共享相同核心的 NIC 上，將會接收到最多資料的應用程式。這將會大幅增加快取命中（cache hit）的機會，並因此改善效能。

8.4. 解決常見的佇列/FRAME 遺失問題

到目前為止，frame 遺失最常見的原因是「*佇列溢出*」（queue overrun）。kernel 會對佇列的長度設下限制，並且在某些情況下，佇列填滿的速度會比它所排出的速度還要快。當此狀況持續太久時，frame 便會開始被捨棄。

如〈圖形 8.1, “網路接收路徑圖形”〉中所述，接收路徑中有兩個主要佇列：NIC 硬體緩衝區以及 socket 佇列。這兩個佇列皆需要經過配置，以確保溢出的問題不會發生。

8.4.1. NIC 硬體緩衝區

NIC 會將其硬體緩衝區填滿 frame；緩衝區接著將會被 NIC 透過插斷所插入的 **softirq** 排出。若要監控此佇列的狀態，請執行以下指令：

```
ethtool -S ethX
```

請將 ***ethX*** 取代為 NIC 的相應裝置名稱。這會顯示 ***ethX*** 中捨棄了多少 frame。通常，因為佇列耗盡了用來存放 frame 的空間時，便會發生捨棄的情況。

用來找出此問題的方法有幾種：

輸入流量

您可藉由減慢輸入流量的速度，以協助避免佇列發生溢位錯誤。您可透過篩選、減少加入的 multicast 群組數量、減少廣播流量等等的動作來避免溢位。

佇列長度

此外，您亦可增加佇列的長度。也就是將特定佇列的緩衝區數量增加到驅動程式所能允許的最大值。若要這麼做，請使用以下指令來編輯 ***ethX*** 的 **rx/tx** 信號參數：

```
ethtool --set-ring ethX
```

附加適當的 **rx** 或是 **tx** 值至先前提到的指令中。欲取得更多相關資訊，請參閱 **man ethtool**。

裝置權重

您亦可增加佇列被排出的速率。若要這麼做，請調整 NIC 的相應「*裝置權重*」。此屬性代表 **softirq** 必須微調 CPU 並將其重新排程前，NIC 所能接收的最大 frame 數量。這是以 **/proc/sys/net/core/dev_weight** 變數來進行控制的。

大部份管理員都會選擇第三個選項。然而請注意，這麼做是有風險的。增加 NIC 在一個循環中可接收的 frame 數量，即代表需要額外的 CPU 週期，在這段時間內，無法在該 CPU 上排程任何應用程式。

8.4.2. socket 佇列

如同 NIC 硬體佇列，socket 佇列會填滿來自於 **softirq** 的網路堆疊。接著應用程式便會排出與它們相應的佇列（透過呼叫 **read**、**recvfrom** 等等）。

若要監控此佇列的狀態，請執行 **netstat** 工具程式；**Recv-Q** 欄位顯示了佇列大小。一般來講，我們會使用相同的管理方式，來處理 socket 佇列中的溢位問題，以及 NIC 硬體緩衝區溢位問題（例如〈節 8.4.1, “NIC 硬體緩衝區”〉）：

輸入流量

第一個選項就是透過配置佇列填滿的速率，以減慢連入流量的速度。若要這麼做，請篩選 frame，或是先佔式地將它們丟棄。您亦可透過降低 NIC 的裝置權重來減慢連入流量的速度[6]。

佇列深度

您亦可透過增加佇列深度，以避免 socket 佇列溢位。若要這麼做，請增加 **rmem_default** kernel 參數或是 **SO_RCVBUF** socket 選項的值。欲取得有關於這兩者的相關資訊，請參閱〈節 8.2, "優化網路設定"〉。

應用程式呼叫頻率

請儘可能優化應用程式使其更頻繁地進行系統呼叫。這包含了修改和重新配置網路應用程式，使其更頻繁地進行 POSIX 系統（例如 **recv**、**read**）。相對的，這能讓應用程式更快速地排出佇列。

對於許多管理員來說，增加佇列深度是較常用的解決方法。這是最簡單的解決方法，不過以長遠的角度來看，不一定永遠適用。隨著網路技術的速度愈來愈快，socket 佇列也會跟著愈快填滿。這代表管理員必須隨著時間的經過，持續重新調整佇列深度。

最佳的解決方式就是增強或是配置應用程式，使其能更快速地由 kernel 中將資料排出（儘管表示您必須將資料排在應用程式空間中）。這能讓資料以更加靈活的方式儲存，因為它能視需求被 swap 出或是傳回。

8.5. MULTICAST 考量

當有多項應用程式在監聽一個 multicast 群組時，就設計上來講處理 multicast frame 的 kernel 程式碼是必要的，以便複製各別 socket 的網路資料。這項複製很費時，並且會在執行 **softirq** 時發生。

因此，在單一 multicast 群組上新增多個監聽程式會直接影響 **softirq** 的執行時間。新增監聽程式至一個 multicast 群組，代表每當該群組接收到一個 frame 時，kernel 就必須建立額外的副本。

當流量低且監聽程式數量較少時，此效應便不會有太大影響。然而，當多個 socket 同時監聽一個高流量的 multicast 群組時，增加的 **softirq** 執行時間，可能會導致於 frame 同時在網路卡上和 socket 佇列中遺失。**softirq** runtime 的增加，亦代表了應用程式在高負載的系統上執行的機會會減少，當監聽高流量 multicast 群組的應用程式數量增加時，multicast frame 遺失的比率也會增加。

您可透過描述於〈節 8.4.2, "socket 佇列"〉或是〈節 8.4.1, "NIC 硬體緩衝區"〉中的方法，以藉由優化您的 socket 佇列和 NIC 硬體緩衝區來解決此 frame 遺失的問題。此外，您亦可優化應用程式的 socket 應用；若要如此，請配置應用程式，以控制單一 socket，並將接收到的網路資料快速地散佈至其它使用者空間程序中。

---

[4] 確認 CPU 和 NIC 之間的快取關聯，代表將它們配置成能夠共享相同的 L2 快取。欲知更多資訊，請參閱〈節 8.3, "封包接收總覽"〉。

[5]〈節 8.3, "封包接收總覽"〉包含了封包流動的總覽，並且應該能幫助您找出網路堆疊中較常出現問題的部分。

[6] 裝置權重可透過 **/proc/sys/net/core/dev_weight** 進行控制。欲取得更多有關於裝置權重與其調整過後之效應的相關資訊，請參閱〈節 8.4.1, "NIC 硬體緩衝區"〉。

附錄 A. 修訂記錄

修訂 4.0-22.3.400	2013-10-31	Rüdiger Landmann

Rebuild with publican 4.0.0

修訂 4.0-22.3	Sat Jun 8 2013	Chester Cheng

Translation & proofreading done. 翻譯、校閱完成。

修訂 4.0-22.2	Tue Jun 4 2013	Terry Chuang

翻譯完成

修訂 4.0-22.1	Thu Apr 18 2013	Chester Cheng

讓翻譯檔案與 XML 來源 4.0-22 同步

修訂 4.0-22	Fri Feb 15 2013	Laura Bailey

Red Hat Enterprise Linux 6.4 出版

修訂 4.0-19	Wed Jan 16 2013	Laura Bailey

小幅修正了內容的一致性（BZ#868404）。

修訂 4.0-18	Tue Nov 27 2012	Laura Bailey

Red Hat Enterprise Linux 6.4 Beta 出版。

修訂 4.0-17	Mon Nov 19 2012	Laura Bailey

新增了 SME 意見以及 numad 的部分（BZ#868404）。

修訂 4.0-16	Thu Nov 08 2012	Laura Bailey

新增了有關於 numad 上的草稿部分（BZ#868404）。

修訂 4.0-15	Wed Oct 17 2012	Laura Bailey

將 SME 意見套用至區塊捨棄的討論部分，並將此部分移至「掛載選項」下（BZ#852990）。

更新了效能設定檔的描述（BZ#858220）。

修訂 4.0-13	Wed Oct 17 2012	Laura Bailey

更新了效能設定檔的描述（BZ#858220）。

修訂 4.0-12	Tue Oct 16 2012	Laura Bailey

改善了文件的瀏覽（BZ#854082）。

修正了 ***file-max*** 的定義（BZ#854094）。

修正了 ***threads-max*** 的定義（BZ#856861）。

修訂 4.0-9	Tue Oct 9 2012	Laura Bailey

已將 FSTRIM 的建議新增至「檔案系統」章節中（BZ#852990）。

已根據客戶意見更新了 ***threads-max*** 參數的描述（BZ#856861）。

更新了有關於 GFS2 片段管理改善上的註釋（BZ#857782）。

修訂 4.0-6	Thu Oct 4 2012	Laura Bailey

新增了有關於 numastat 工具程式上的部分（BZ#853274）。

修訂 4.0-3	Tue Sep 18 2012	Laura Bailey

新增了有關於新 perf 功能的部分（BZ#854082）。

修正了 file-max 參數的描述（BZ#854094）。

修訂 4.0-2	Mon Sep 10 2012	Laura Bailey

已將 BTRFS 部分和基本簡介新增至檔案系統中（BZ#852978）。

記載了有關於 Valgrind 與 GDB 整合上的相關資訊（BZ#853279）。

修訂 3.0-15 **Thursday March 22 2012** **Laura Bailey**

已新增並更新了 tuned-adm 設定檔的描述（BZ#803552）。

修訂 3.0-10 **Friday March 02 2012** **Laura Bailey**

已更新了 threads-max 與 file-max 參數的描述（BZ#752825）。

更新了 slice_idle 參數的預設值（BZ#785054）。

修訂 3.0-8 **Thursday February 02 2012** **Laura Bailey**

已將 numactl 的工作集與 CPU 綁定和記憶體分配的部分重建，並新增至〈節 4.1.2, "微調 CPU 效能"〉（BZ#639784）。

修正了內部連結上的使用（BZ#786099）。

修訂 3.0-5 **Tuesday January 17 2012** **Laura Bailey**

小幅修正了〈節 5.3, "使用 Valgrind 側寫記憶體使用量"〉（BZ#639793）。

修訂 3.0-3 **Wednesday January 11 2012** **Laura Bailey**

確保了內部與外部超連結之間的一致性（BZ#752796）。

新增了〈節 5.3, "使用 Valgrind 側寫記憶體使用量"〉（BZ#639793）。

新增了〈節 4.1.2, "微調 CPU 效能"〉，並更新了〈章 4, *CPU*〉的結構（BZ#639784）。

修訂 1.0-0 **Friday December 02 2011** **Laura Bailey**

發行 Red Hat Enterprise Linux 6.2 GA。