第二章

邊車模式

第一個單節點模式是邊車模式（Sidecar）[譯註 1]。邊車模式是由兩個容器組成的單節點模式。第一個是**應用程式容器**（*Application Container*），包含應用程式的核心邏輯，沒有這個容器，應用程式將不存在。除應用程式容器外，還有一個**邊車容器**（*Sidecar Container*）。邊車的作用是加強和改進應用程式容器，通常沒有應用程式容器的商業邏輯（知識）。用一個最簡單的形式來說，原本的應用程式容器可能很難改進功能，使用邊車容器可以用來增加功能。邊車模式容器透過原子性**容器群組**（*Container Group*），在同一台計算機上共同調度，例如 Kubernetes 中的 pod API 物件。除了安排在同一台機器上，應用程式容器和邊車容器共享了一些資源，包含部分的檔案系統、主機名稱、網路、還有其他名稱空間（Namespace）。圖 2-1 描述了邊車模式的樣子：

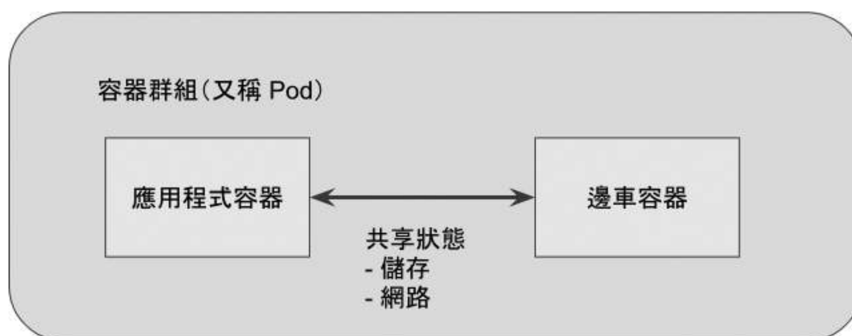

圖 2-1　一般邊車模式

譯註 1　Sidecar 可以翻成側車或邊車，這模式也叫做搭檔、伴侶、跟班模式。

碁峰 www.gotop.com.tw

邊車模式範例：讓遺留服務支援 HTTPS

假設有個遺留 Web 服務[譯註 2]。當初建置時，內部網路安全並非公司優先考量的重點，因此，應用程式僅透過未加密的 HTTP 來處理請求，而不是 HTTPS。由於最近的資安事件，公司要求所有網站需要走 HTTPS。

因為這個應用程式的原始碼，是用舊的建置系統建置的，但那個系統已經無法運作，所以要求使用 HTTPS 的需求，加遽了這個團隊的痛苦。將這樣的 HTTP 應用程式容器化並不難，只要能讓這個舊服務在使用舊版 Linux 套件的容器運行即可。不過，要增加 HTTPS 到這個應用程式顯然是有難度的。當建議使用邊車模式可以更容易解決問題的同時，開發團隊正在抉擇是否將遺留的系統復活，同時把應用程式原始碼移植到新版本作業系統。

邊車模式在這種情況下的應用是很直覺的。遺留 Web 服務只配置服務本地端（127.0.0.1），表示只有共享本地網路的服務，才能夠存取這個服務。通常，這不是實際的選擇，因為這表示沒有人可以存取這個 Web 服務。在這種舊的容器使用邊車模式，我們會增加 nginx 邊車容器。

這個 nginx 容器與遺留 Web 應用程式跑在同一個網路的命名空間，因此，它可以存取遺留 Web 應用程式。這時候，nginx 服務可以中斷外部 HTTPS 的流量，然後將流量轉導到遺留 Web 應用程式（見圖 2-2）。因為未加密傳輸流量僅透過容器群組內部的迴路（loopback），所以這樣的資料安全性是滿足網路安全團隊的要求。同樣的，透過使用邊車模式，團隊已經可以對應用程式進行現代化的改造，而且不需要重新弄清楚如何建立應用 HTTPS 應用服務。

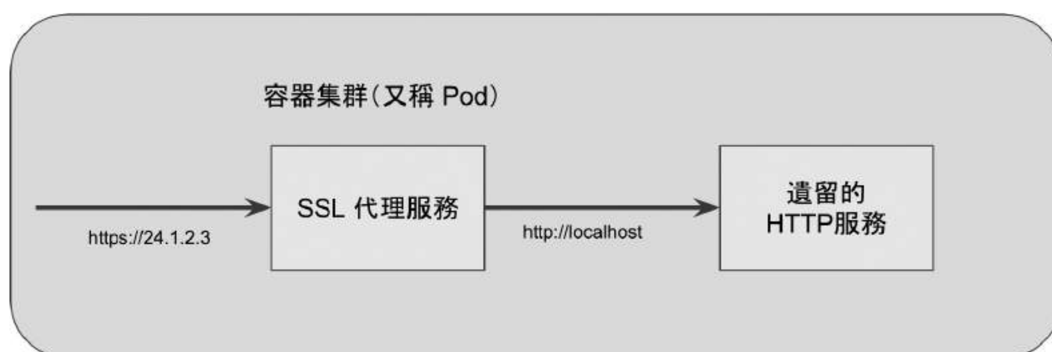

圖 2-2　HTTPS 邊車模式

[譯註 2] Legacy 中文為遺留、遺產之意，也可以用老舊、古老、過往相關詞代替，口語習慣使用英文，文字則用「遺留」或保留不翻。

碁峰 www.gotop.com.tw

第三章

大使模式

上一章介紹了邊車模式，這個模式增加一個容器到既有的容器，增加其功能。本章介紹大使模式（Ambassador Pattern），它代替應用程式容器與其他世界的服務做互動。與其他單節點模式一樣，這兩個容器緊密連接在單一台機器，它們在這台機器裡緊密且共存。圖 3-1 顯示了這種模式的典型樣式：

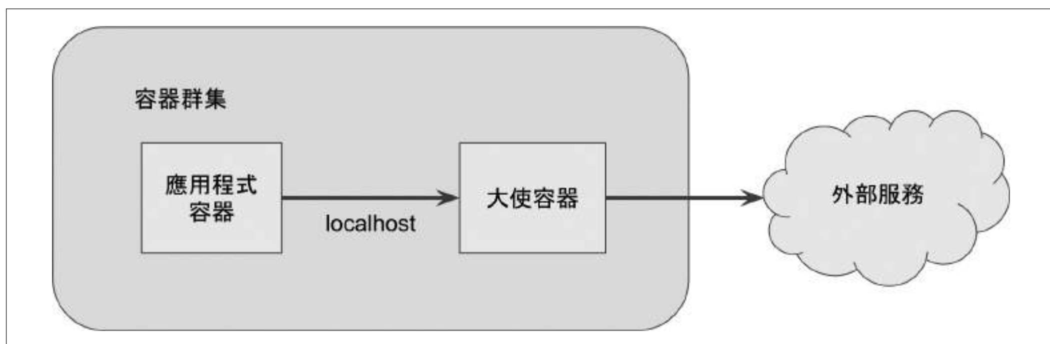

圖 3-1　一般大使模式

大使模式的價值是雙重的。首先，如同其他單節點模式一樣，建置模組化、可重複使用的容器也具有內在價值。關注點的分離使容器更容易建立和維護。同樣，大使容器可以重複使用許多不同的應用程式容器。這種重用加快了應用程式的開發，因為容器的程式碼可以在很多地方重複使用。此外，實作更為一致，品質更好，因為它是一次性建置，可用於許多不同的環境。

本章其餘部分提供了大量使用大使模式來實現一系列實際應用程式的範例。

碁峰 www.gotop.com.tw

用大使模式分片一個服務

有時候，想儲存在儲存層的資料，對單一台機器來講變得很大。在這種情況下，需要分割、拆分儲存層。分片（Sharding）將儲存層分成多個不重複的部分，每個部分都由一台獨立的機器管理。本章重點在介紹一種單一節點模式，這個節點用來與存放在世界上某個地方的分片服務（Sharded Service）進行通訊。不討論如何把分片服務放入既有的服務。分片和多節點分片服務設計模式將在第六章中詳細討論。圖 3-2 顯示了分片服務的示意圖。

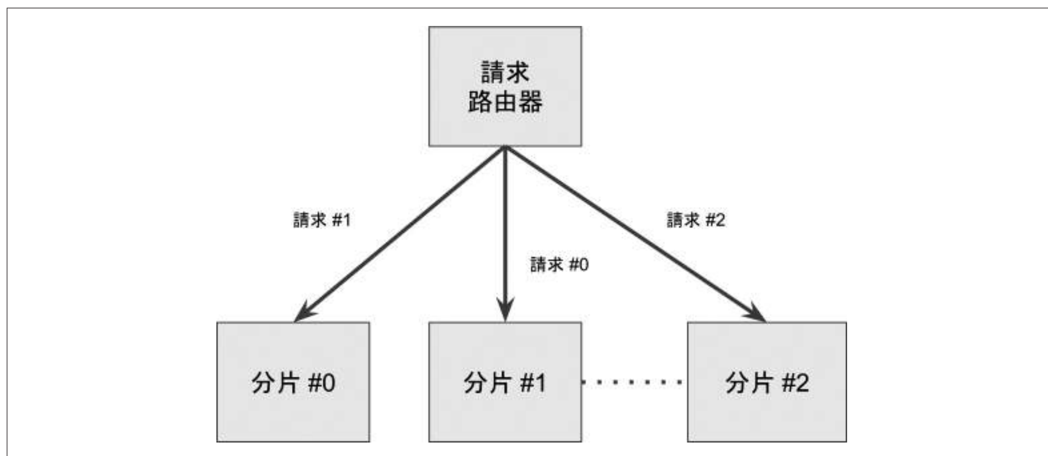

圖 3-2　一般的分片服務

當部署分片服務時，出現的一個問題是：分片服務如何整合前端（Frontend）、中介程式（Middleware）所儲存的資料。很顯然，需要邏輯將特定的請求路由到特定的分片，但通常很難將這種分片客戶端改裝為現有的原始程式碼，這些程式碼需要連接到單個儲存後端。此外，分片服務會讓開發環境（通常只有單一儲存分片）和正式環境[譯註 1]（通常存在多個儲存分片）之間共享配置變得困難。

一種方法是將所有的分片邏輯建立到分片服務本身中。在這種方法中，分片服務還具有無狀態負載平衡器（Stateless Load Balancer），可將流量引導至對應的分片。實際上，這個負載平衡器是一個分散式大使即服務（Distributed Ambassador as a Service）。這使得客戶端大使變得不必要了，因為分片服務的部署更加複雜。另一種方法是在客戶端整

[譯註 1] Production Environment 直譯是生產環境，在軟體開發與其對應的是測試環境，所以本書譯為正式環境。

碁峰 www.gotop.com.tw

第四章

適配器

在前面的章節中，我們看到了邊車模式（Sidecar）如何擴展和擴充現有的應用程式容器。我們還了解了大使（Ambassadors）如何改變和管理應用程式容器與外部世界的通訊方式。本章介紹最後的單節點模式：**適配器**（*Adapter*）模式。在適配器模式中，**適配器容器**（*Adapter Container*）用於修改**應用程式容器**的介面，以使其符合所有應用程式所需的某些預定義介面。例如，適配器可以確保應用程式實現一致的監視介面。或者它可以確保始終將日誌文件寫入標準輸出（`stdout`）或任何其他協議的。

真實世界的應用程式開發是種異質性的混合行動。應用程式的某些部分可能是由團隊從頭開始編寫的，部分由供應商提供，部分由現成的開源軟體和自行編譯的二進位檔組成。這種異質性的淨效應[譯註 1]，會讓你部署的任何實際應用程式，都將使用各種語言編寫，具有各種日誌記錄、監控和其他通用服務。

但是，要有效地監視和運作應用程式，需要有通用介面。當每個應用程式使用不同的格式和介面提供指標（Metrics）時，很難在一個地方收集所有指標，然後進行可視化和警報。這是適配器模式相關的地方。與其他單節點模式一樣，適配器模式由模組化容器組成。不同的應用程式容器可以呈現許多不同的監視介面，而適配器容器可以適應這種異質性結構，以呈現一致的介面。這讓你可以部署單一工具，用來提供單一介面。圖 4-1 說明了這種一般的模式。

譯註 1　net effect：淨效應。已經考慮過事務的正、反面作用後，所做的最後評論。

碁峰 www.gotop.com.tw

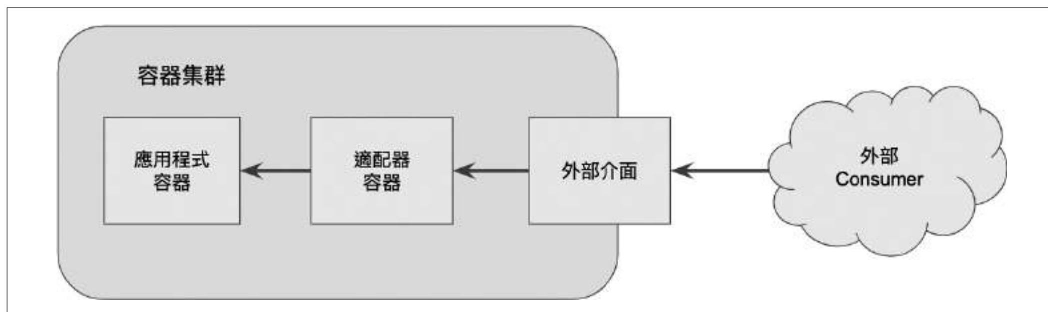

圖 4-1　一般適配器模式

本章節接下來的部分提供不同適配器的應用程式場景。

監控

在監控軟體時，你需要一個單一的方法，能夠自動探索和監控部署到環境中的應用程式。為了實現這一點，每個應用程式都必須實現相同的監視介面。有許多標準化監控介面的範例，例如 syslog、Windows 上的事件追蹤（etw）、JMX for Java 應用程式以及許多其他協議和介面。然而，這些每一個在通訊協議以及通訊方法（push 與 pull）方面都是獨一無二的。

可惜的是，分散式系統中的應用程式，可能涵蓋從你寫的程式碼，到現成的開源元件的全部範圍。因此，你將發現自己擁有各種不同的監控介面，你需要將這些介面彙整於一個易於理解的系統中。

還好大多數監控解決方案都知道它們需要廣泛應用，因此已經實作了各種擴充套件，可以將一種監控格式適用於通用介面。有了這套工具，要如何以彈性和穩定的方式部署和管理應用程式？適配器模式可以為我們提供答案。將適配器模式應用於監視，我們看到應用程式容器只是我們要監視的。適配器容器包含了用於將應用程式容器公開的監視介面，轉換為通用監視系統所期望的介面工具。

以這種方式解耦系統，使得系統更易於理解、可維護。推出新版本的應用程式不需要推出監控適配器。另外，監視容器可以與多個不同的應用程式容器一起重用。監視容器甚至可以由獨立於應用程式開發者的監視系統維護者提供。最後，將監視適配器部署為單獨的容器，以確保每個容器在 CPU 和記憶體方面都獲得自己的專用資源。這可確保行為不當的監控適配器不會導致客戶端服務出現問題。

碁峰 www.gotop.com.tw

針對需求，我們將使用這個開源的 Web 快取：Varnish（*https://varnish-cache.org/*）。

部署快取服務

部署 Web 快取的最簡單方法是使用 Sidecar 模式，也在 Web 伺服器的每個實例旁邊（參見圖 5-5）。

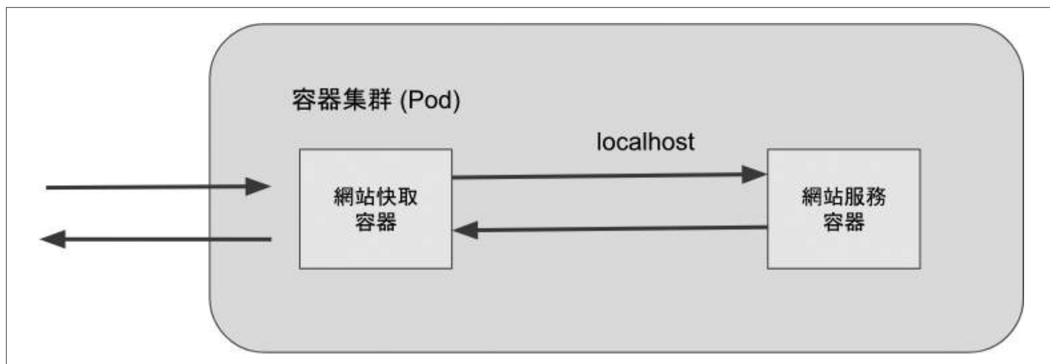

圖 5-5　以 Sidecar 模式新增 Web 快取服務

雖然這種方法很簡單，但它有些缺點，也就是必須使用與 Web 伺服器相同的比例擴展快取服務。對快取來講，這通常不是我們想要的，我們要的是少量的複本，每個複本提供最多的資源，因此，你會希望有兩個擁有 5GB 記憶體的複本，而不是十個擁有 1GB 記憶體的複本。這是因為如果每個網頁都儲存在每個複本中，使用十個複本，每個網頁將儲存 10 次，結果就是快取記憶體中能夠儲存的網頁變少，導致 *hit rate* 降低，即從快取中提供請求的時間的一小部分，這反過來又降低了快取的效用。

雖然你確實需要一些大型快取，但你可能還需要許多 Web 伺服器的小型複本。許多程式語言（例如 NodeJS）實際上只能使用單個 CPU 內核[譯註 1]，因此你希望許多複本能夠利用多個內核，即使在同一台機器上也是如此。所以，把快取層配置為 Web 服務層上方的第二個無狀態複本服務層是最有意義的，如圖 5-6 所示。

[譯註 1] NodeJS 是單執行緒，若要使用多核心 CPU 資源，需要透過原生的 Cluster Mode。

碁峰 www.gotop.com.tw

SSL 終止

除了執行效能快取之外，邊緣層（Edge Layer）執行的其他常見任務之一是 SSL 終止（SSL Termination）。即使打算在叢集系統不同層之間進行通訊，針對邊緣層和內部服務，你仍應使用不同 SSL 憑證。

實際上，每個單獨的內部服務都應該使用自己的憑證來確保每個層都可以獨立部署。可是 Varnish Web 快取不能用於 SSL 終止，還好 nginx 可以。因此，我們希望在我們的無狀態應用程式模式中增加第三層，該模式將是 nginx 伺服器的複本層，它將處理 HTTPS 流量的 SSL 終止並將流量轉發到 Varnish 快取。HTTP 流量繼續傳輸到 Varnish Web 快取，Varnish 將流量轉導到我們的 Web 應用程式，如圖 5-8 所示。

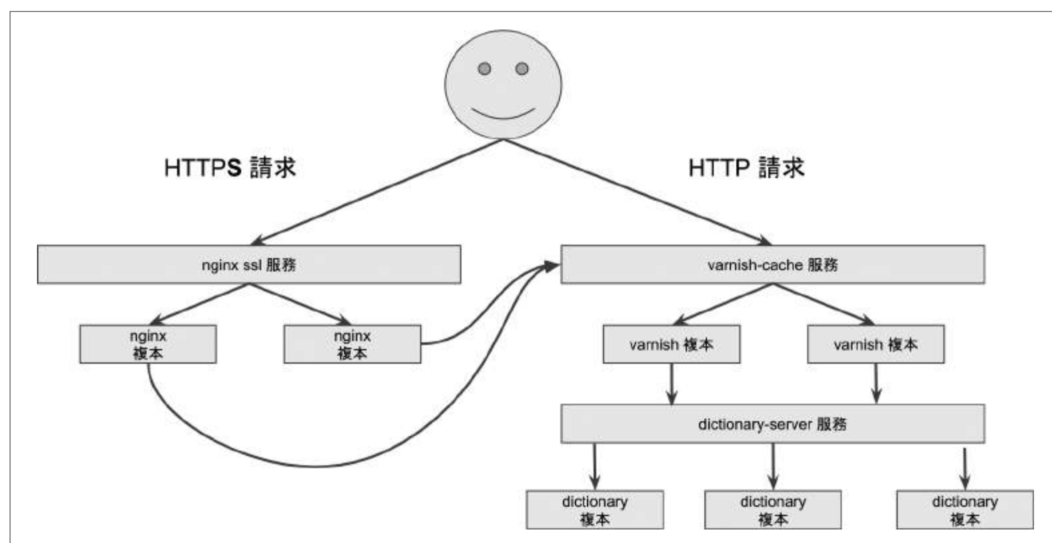

圖 5-8　完整複本無狀態服務範例

碁峯
www.gotop.com.tw

分配／聚集

到目前為止，我們已經檢驗這樣的系統：以每秒處理的請求數量（無狀態複本模式，Stateless Replicated Pattern），複本可擴展性的系統，以及資料大小（分片資料模式，Sharded Data Pattern）的可擴展性。本章將介紹**分配／聚集模式**（*Scatter/Gather Pattern*）[譯註1]，此模式在時間方面使用複本來實現可擴展性。具體來說，分配／聚集模式可以在服務請求中實現平行性（Parallelism），相較於需要依照次序執行的服務，可以更快地提供服務。

與複本和分片系統一樣，分配／聚集模式是一種樹狀模式（Tree Pattern），其根節點（Root）分配請求，並在葉節點（Leaf）處理這些請求。但是，與複本和分片系統相比，分配／聚集請求同時被分配到系統中的所有複本。每個複本執行少量處理，然後將結果的一部分返回到根節點。最後，根伺服器將各種部分結果組合在一起，形成對請求的單個完整回應，將此請求發送回客戶端。分配／聚集模式如圖 7-1 所示。

當處理特定請求所需的大量獨立處理時，分配／聚集模式非常有用。分配／聚集可以看作是對服務請求所需的計算進行分片，而不是分片資料（儘管資料分片也可能是其中的一部分）。

譯註1　Scatter/Gather 中文翻譯成「分配／聚集」，其中 Scatter 翻成「分配」，不翻分散（Distribute）。Gather 翻成「聚集」非「聚合」，「聚合」指的是 Aggregation，包含散、合兩個動作，換言之 Scatter/Gather 可以說是 Aggregation 的意思。Scatter/Gather 拆成兩個字有特別強調階段與分工之意，故文中不用「聚合」取代。

碁峰 www.gotop.com.tw

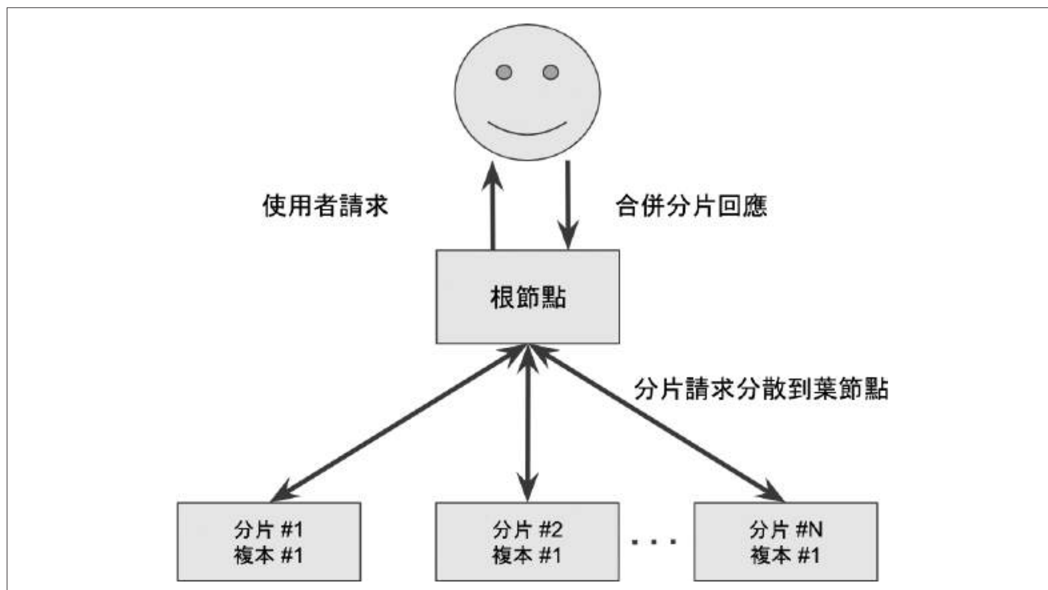

圖 7-1　分配／聚集模式

用根節點分配分配／聚集

最簡單的分配／聚集形式是每個葉節點完全同性質，工作被分配到許多不同的葉節點，以便改善請求的效能形式。這種模式相當於解決「尷尬平行」（Embarassingly Parallel）問題。問題可以分解成許多不同的部分，每個部分可以與所有其他部分一起重新組合以形成完整的答案。

為了更具體地理解這一點，假設需要為使用者請求 *R* 提供服務，並且單核心需要一分鐘來處理此請求的答案 *A*。如果我們編寫多執行緒應用程式，我們可以在單台機器，使用多核心平行化此請求。

由於這種方法和 30 核心處理器（是的，一般是 32 核處理器，數字 30 在數學上容易理解），我們可以將處理單個請求所需的時間減少到 2 秒（共 60 秒，拆分 30 個執行緒進行計算，得到 2 秒）。兩秒對於服務使用者的 Web 請求來講是相當慢的。此外，要真實達到在單核心做平行處理的加速是很棘手的，因為記憶體、網路、磁碟存取頻寬都是瓶頸。我們可以使用分配／聚集模式在多台不同的機器上，跨多個程序（process）平行化請求，而不是在單機器上跨核心平行化應用程序。

碁峰 www.gotop.com.tw

擴展分配 / 聚集的可靠性和規模

如同分片系統，具有分片分配 / 聚集系統的單個複本可能不是理想的設計選擇。單個複本意味著如果失敗，則所有分配 / 聚集請求將在分片不可用的持續時間內失敗，因為所有請求都需要由分配 / 聚集模式中的所有葉節點處理。同樣，升級將佔用分片的一定百分比，因此不可能在面對使用者負載下進行升級。最後，系統的計算規模將受到任何單個節點能夠實現的負載的限制。最終，這會限制你的擴展功能，正如我們在前面部分中看到的那樣，不能直接增加分片數量以提高分配 / 聚集模式的計算能力。

由於可靠性和擴展性的挑戰，正確的方法是複製每個單獨的分片，以便在每個葉節點處不是單個實例，而是存在實現每個葉節點分片的複本服務。此複製分片分配 / 聚集模式如圖 7-4 所示。

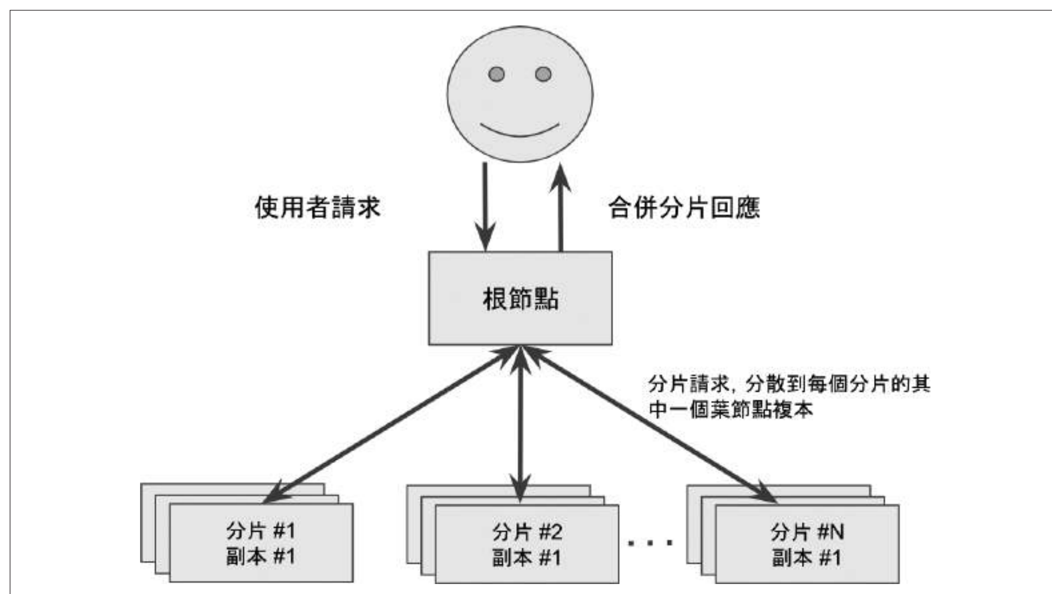

圖 7-4　一個分片、複本的分配 / 聚集系統

以這種方式建置，來自根節點的每個葉節點請求，實際上在分片的所有健康複本上進行負載平衡。這表示如果出現任何故障，使用者不會因為此故障而發現有異常。同時，可以安全地在負載下執行升級，因為每次複製的分片可以一次升級一個複本。實際上，可以同時執行多個分片的升級，具體取決於希望執行升級的速度。

碁峰 www.gotop.com.tw

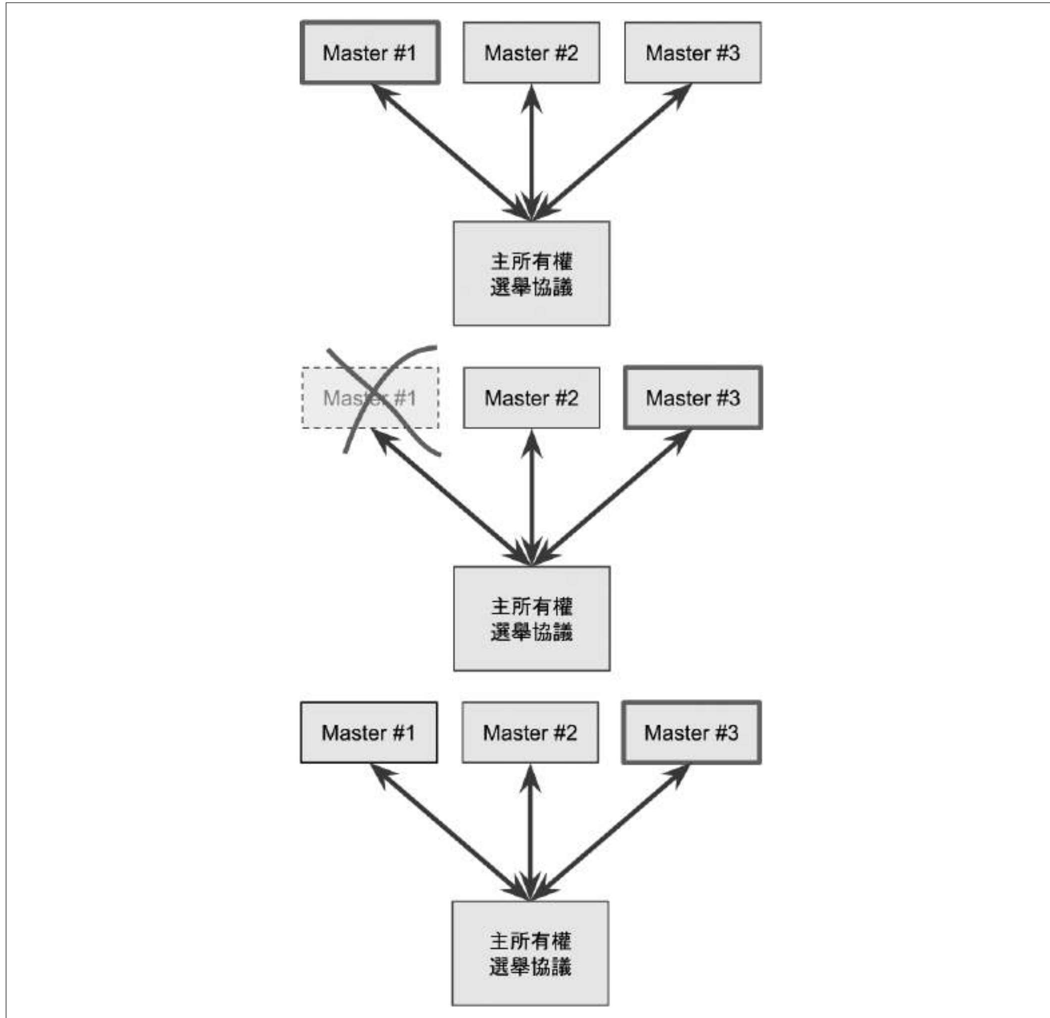

圖 9-1　運作中的主選舉協議：一開始選擇第一個複本作主要，當它故障時，第三個接管主要所有權的任務

通常，建立分散式所有權是設計一個可靠分散式系統中，最複雜也是最重要的。

碁峰
www.gotop.com.tw