

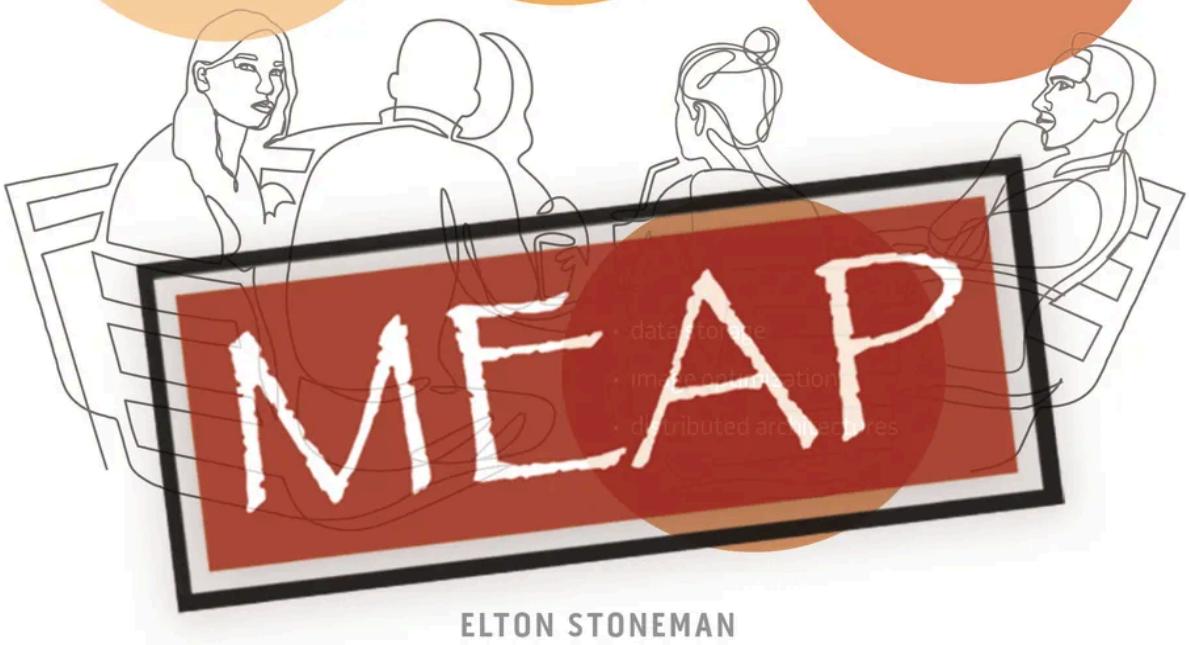
LEARN **DOCKER** IN A MONTH OF LUNCHES

SECOND EDITION

- running containers
- building images
- containers in the cloud

- Kubernetes primer
- Arm and Intel
- Linux and Windows

- containerized CI/CD
- legacy apps
- Docker Compose



MANNING



MEAP Edition Manning
Manning Early Access Program

在一個月的午餐時間學會 Docker, 第二版版本 4
Learn Docker in a Month of Lunches, Second Edition
Version 4

Copyright 2025 Manning Publications

For more information on this and other Manning titles go to manning.com.

welcome

你好呀！感謝您參與*Learn Docker in a Month of Lunches: Second Edition*的Meap。希望您發現這本書有用，我很高興聽到您的任何反饋。我叫Elton，我是自由職業者和教練。自從最早發行以來，我就一直在使用Docker，並在Docker工作了三年。從那以後，我在集裝箱旅程的每個階段都為組織提供了幫助。我很幸運能環遊世界，在會議上講話並在Docker上開展研討會，並且我正在蒸餾多年來使用和教Docker進入這本書。

所有最新技術的午餐中的第二版Learn Docker已在一個月的午餐中進行了更新 - 每個代碼樣本和Try-Now練習都已在Windows，Mac和Linux上進行了刷新和測試。更重要的是，第3部分中的所有章節都是新的：它們取代了不再相關的第一版的主題。多平台構建，雲容器服務，重建舊版Windows應用程序和Kubernetes都在進行中。Dockerswarm now of Out（它仍然存在，但我現在不知道有任何公司正在使用它，每個人都喜歡Kubernetes）。

在一個月的午餐中學習Docker的目標是針對新的和改進的Docker用戶。學習路徑通過包裝自己的應用程序將您的應用程序包裝到docker運行到生產級環境中的規模運行。這些章節開始簡單，逐漸變得更加複雜，節省了最後一個更高級的建築主題。如果您是集裝箱的新手，那麼這本書將帶您踏上邏輯和希望的旅程。如果您已經在Docker有一些經驗，那麼可以跳過章節是可以的 - 儘管我有信心您將學習一些新知識，甚至可以從第1章中學習。:)

我想說的是在一個月的午餐中學習Docker的重要一件事：我希望它盡可能訪問。太多的Docker書籍以為您是Linux Guru，它們只會為您提供僅在Intel機器上使用的練習，並且只有在您花了多年的時間為Sysadmin時才有意義。這本書是不同的。所有代碼樣本和練習都是跨平台，並且在Windows，Mac，Linux，Intel和Arm上工作。您應該能夠在桌面上的Windows 11，MacBook上的MacOS或Raspberry Pi上的Debian跟隨。我還試圖假設最少的背景知識 - Docker跨越了建築，開發和操作的界限，並且我嘗試這樣做。無論您的背景如何，這本書都應該為您服務。

午餐格式的月份非常適合學習Docker，該登山者將其分解為任務驅動的主題。重點是動手學習，因此我現在提供了很多嘗試，現在進行了很多嘗試，足夠的背景信息可以填充空白，並在每一章中供您完成實驗室。您應該找到從一章到另一章的邏輯進步。如果您認為我在章節之間取得了奇怪的飛躍，如果代碼樣本對您不起作用，或者如果某些內容沒有意義，請在Livebook討論論壇中標記它。

再次感謝您加入我。

- 埃爾頓

brief contents

PART 1: REVISED

- 1 Before you begin*
- 2 Understanding Docker and running Hello World*
- 3 Building your own Docker images*
- 4 Packaging applications from source code into Docker images*
- 5 Sharing images with Docker Hub and other registries*
- 6 Using Docker volumes for persistent storage*

PART 2: REVISED

- 7 Running multi-container apps with Docker Compose*
- 8 Supporting reliability with health checks and dependency checks*
- 9 Adding observability with containerized monitoring*
- 10 Running multiple environments with Docker Compose*
- 11 Building and testing applications with Docker and Docker Compose*

PART 3: NEW AND REVISED

- 12 Running containers on different platforms*
- 13 Replatforming the legacy: Packaging and running Windows apps in Docker*
- 14 Containers in the cloud with Microsoft Azure and Google Cloud*
- 15 Kubernetes: A primer*
- 16 CI/CD in the cloud with Docker and GitHub Actions*

PART 4: REVISED

- 17 Optimizing your Docker images for size, speed, and security*

- 18 Application configuration management in containers*
- 19 Writing and managing application logs with Docker*
- 20 Controlling HTTP traffic to containers with a reverse proxy*
- 21 Asynchronous communication with a message queue*
- 22 Never the end*

1開始之前

Docker是一個平台，用於在輕巧單元中運行應用程序，稱為*containers*。從雲中的無服務器功能到企業中的戰略規劃，容器已經佔據了無處不在的軟件。Docker現在是整個行業的運營商和開發人員的核心能力 - 自2019年以來的每項堆棧溢出調查中，Docker一直對人們最想要或最喜歡的技術（或兩者兼而有之）進行了調查。

Docker是一項簡單的學習技術。 您可以將這本書作為完整的初學者選擇，並且您將在第2章中運行容器，並在第3章中運行包裝應用程序。每一章都側重於實用任務，示例和實驗室在任何運行Docker的機器上都可以使用，即Windows，Mac和Linux用戶都在這裡歡迎使用。

在我教Docker的多年中，您將遵循的旅程得到了磨練。 每章都是動手的 - 除了這一章。 在開始學習Docker之前，重要的是要了解現實世界中的容器如何使用以及它們解決的問題類型 - 這就是我在這裡介紹的內容。本章還介紹了我將如何教Docker，因此您可以弄清楚這是否適合您。

現在，讓我們看看人們對容器的所作所為 - 我將涵蓋組織在Docker中看到巨大成功的五種主要情況。 您會看到可以使用容器解決的各種問題，其中一些肯定會映射到您自己的工作中的場景。到本章結尾，您將了解為什麼Docker是您需要知道的技術，並且您將看到這本書將如何將您帶到那裡。

1.1為什麼容器會佔領世界

我自己的Docker旅程始於2014年，當時我正在從事一個為Android設備提供API的項目。 我們開始使用Docker進行開發工具 - 源代碼並構建服務器。 然後，我們獲得了信心，並開始在容器中運行API以進行測試環境。 到項目結束時，每個環境都由Docker提供動力，包括生產，我們對可用性和規模都有嚴格的要求。

當我搬出項目時，向新團隊的交換是GitHub存儲庫中的一個讀數文件。在任何環境中構建，部署和管理應用程序的唯一要求是Docker。新開發人員剛剛抓住了源代碼，並運行了一個命令來構建並在本地運行所有內容。管理員使用完全相同的工具來部署和管理生產集群中的容器。

通常，在一個大小的項目上，交換需要兩個星期。新開發人員需要安裝特定版本的六個工具，管理員需要安裝六個完全不同的工具。Docker集中了工具鏈，並使所有人都變得更加容易，以至於我認為有一天每個項目都必須使用容器。

我在2016年至2020年之間在Docker，Inc。工作，並花了幾年的時間在成為自由職業顧問和教練，專門研究容器和雲。Docker正在接近無處不在，部分原因是它使交付變得如此容易，部分原因是它是如此靈活 - 您可以將其帶入所有新舊項目，Windows和Linux。讓我們看看這些項目中容器適合的位置。

1.1.1 遷移到雲

對於許多組織的領導者來說，將應用程序移至雲是最重要的。這是一個有吸引力的選擇 - LET Microsoft或Amazon或Google擔心服務器，磁盤，網絡，電源和碳中立性。在全球數據中心託管您的應用程序實際上無限的擴展潛力。幾分鐘之內部署到新環境，並僅用於您使用的資源。但是，您如何將應用程序轉移到雲中？

曾經有兩個選項將應用程序遷移到雲：基礎架構作為服務（IAAS）和平台作為服務（PAAS）。這兩個選項都不好。您的選擇基本上是妥協 - 選擇Paas並運行一個項目，將您的應用程序的所有部分遷移到雲中相關的託管服務。這是一個艱難的項目，它將您鎖定在一個雲中，但確實可以使您降低運行成本。替代方案是IaaS，您可以為應用程序的每個組件旋轉虛擬機。您可以在雲上攜帶可移植性，但運行成本要高得多。圖1.1顯示了典型的分佈式應用程序如何使用IaaS和Paas進行雲遷移。

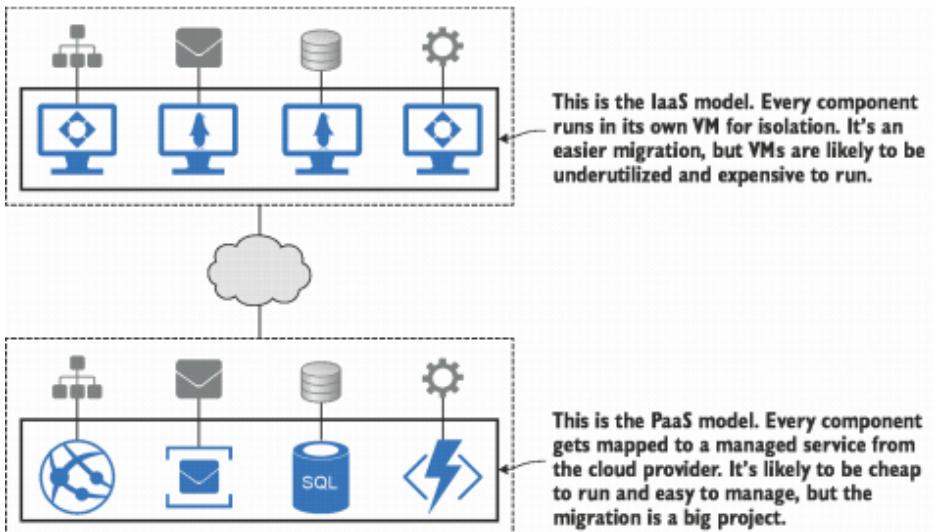


圖1.1遷移到雲的原始選項 - 使用IAAS並運行大量每月成本高的VM，或使用PAAS並獲得較低的運行成本，但在遷移上花費更多的時間。

Docker提供了第三種選擇，而無需妥協。您將應用程序的每個部分遷移到容器中，然後可以使用Azure Kubernetes Service或Amazon的Elastic Container Service或在數據中心中自己的容器平台上運行整個應用程序。您將在第7章中了解如何在容器中包裝和運行這樣的分佈式應用程序，在第14章和第15章中，您將看到如何在生產中進行大規模運行。圖1.2顯示了Docker選項，該選項可為您提供一個便攜式應用程序，您可以在任何云中以低成本運行，或者在數據中心或筆記本電腦上運行。

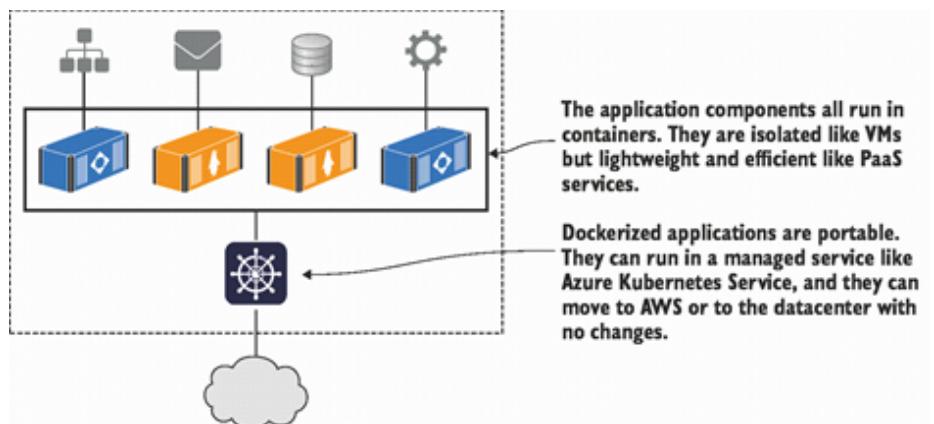


圖1.2同一應用程序，在移至雲之前遷移到Docker。該應用程序具有PAAS的成本優勢，並具有IaaS的可移植性優勢，並且您只能使用Docker獲得的易用性。

它確實需要一些投資才能遷移到容器中：您需要將現有的安裝步驟構建到稱為Dockerfiles的腳本中，並使用Docker Compose或Kubernetes格式將部署文檔構建為描述性應用程序表現。但是您無需更改代碼，最終結果使用每個環境（從筆記本電腦到雲）上的每個環境上的相同技術堆棧以相同的方式運行。

1.1.2 現代化的舊應用程序

您幾乎可以在容器中的雲中運行幾乎所有應用程序，但是如果使用較舊的單片設計，您將無法獲得Docker或Cloud Platform的全部價值。整體在容器中正常工作，但它們限制了您的敏捷性。您可以在30秒內使用容器進行自動舞台上的新功能進行新功能。但是，如果該功能是由200萬行代碼構建的巨石的一部分，那麼您可能不得不坐在兩週的回歸測試週期之前，然後才能發布。

將您的應用程序移至Docker是現代化體系結構，採用新模式而無需完整重寫應用程序的重要第一步。該方法很簡單 - 您首先將應用程序移至帶有Dockerfile的單個容器，並在本書中撰寫您將學習的語法。現在，您在一個容器中有一個整體。

容器在自己的虛擬網絡中運行，因此它們可以相互通信而不會暴露於外界。這意味著您可以開始分解應用程序，將功能移動到自己的容器中，因此逐漸地，您的整體可以演變為分佈式應用程序，其中整個功能集由多個容器提供。圖1.3顯示了示例應用程序體系結構的外觀。

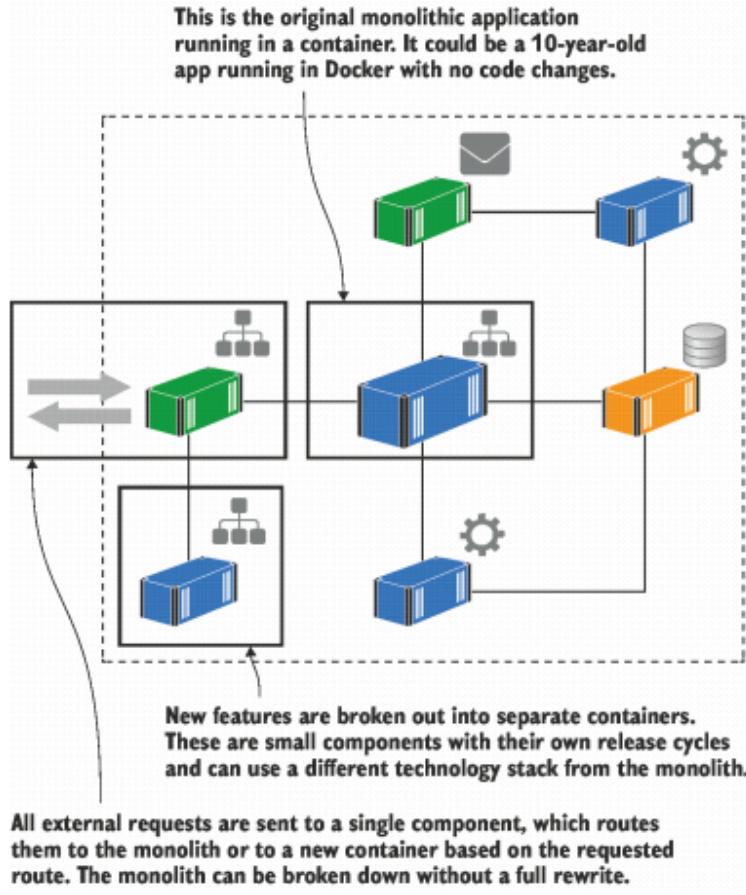


圖1.3將整體分解為分佈式應用程序，而無需重寫整個項目。所有組件都在Docker容器中運行，並且路由組件決定是否通過整體或新的微服務來滿足請求。

這為您帶來了微服務架構的許多好處。您的關鍵功能是可以獨立管理的小型，孤立的單元。這意味著您可以快速測試更改，因為您沒有更改整體，只有運行功能的容器。您可以向上和向下擴展功能，並且可以使不同的技術來滿足各種要求。

使用Docker對舊的應用程序架構進行現代化，您將使用第13、20和21章中的實際示例自己來做到這一點。您可以提供更敏捷，可擴展和彈性的應用程序，而您可以在階段進行此操作，而不是停止所有在18個月重寫項目上進行新功能的工作。

1.1.3 構建新的雲本地應用

Docker可以幫助您將現有的應用程序轉移到雲中，無論它們是分佈式應用程序還是整體。如果您有整體，則Docker可以幫助您將它們分解為現代體系結構，無論您是在雲中還是在數據中心中奔跑。Docker大大加速了基於雲原則的全新項目。

Cloud Native Computing Foundation (CNCF) 將這些新體系結構描述為“使用開源軟件堆棧將應用程序部署為微服務，將每個部分包裝到自己的容器中，然後動態策劃這些容器以優化資源利用率。”

圖1.4顯示了用於新的微服務應用程序的典型體系結構 - 這是社區的演示應用程序，您可以在<https://github.com/microvices>上找到該應用程序。

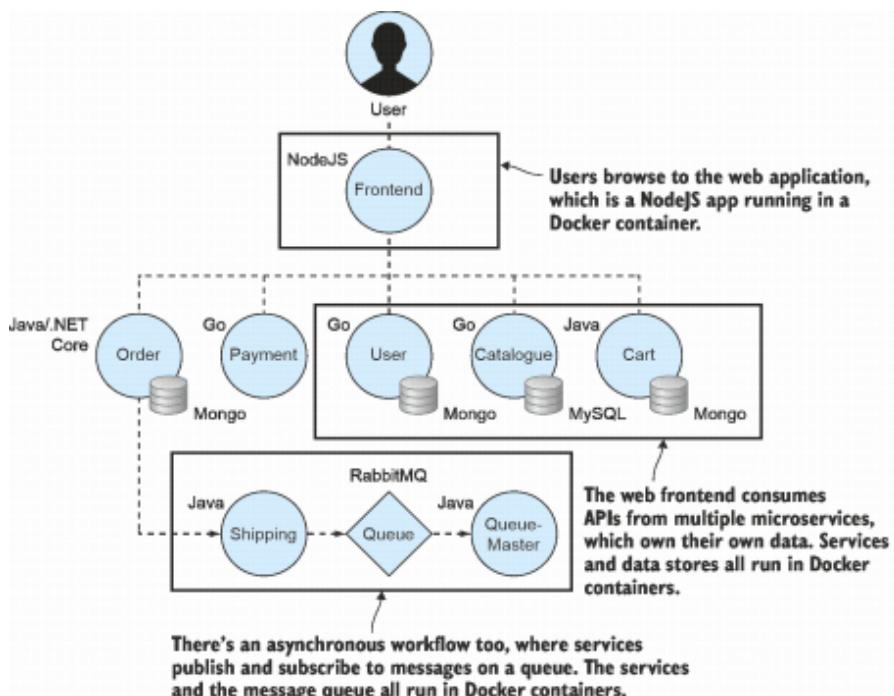


圖1.4雲本地應用程序是用微服務體系結構構建的，每個組件都在容器中運行。

如果您想查看如何實際實施微服務，這是一個很好的示例應用程序。每個組件都擁有自己的數據並通過API曝光。前端是一個消耗所有API服務的Web應用程序。演示應用程序使用各種編程語言和不同的數據庫技術，但是每個組件都有一個包裝dockerfile，並且整個應用程序在Docker組成的文件中定義。

您將在第4章中學習如何使用Docker來編譯代碼，作為包裝應用程序的一部分。這意味著您不需要安裝任何開發工具來構建和運行這樣的應用程序。開發人員只能安裝Docker，克隆源代碼，並使用一個命令構建並運行整個應用程序。

Docker還可以輕鬆地將第三方軟件帶入您的應用程序，在不編寫自己的代碼的情況下添加功能。Docker Hub是一項公共服務，團隊共享在容器中運行的軟件。CNCF發布了可以用於從監視到消息隊列的所有內容的開源項目地圖，並且它們都可以從Docker Hub免費使用。

1.1.4技術創新：無服務器等

現代的關鍵驅動力之一是一致性：團隊希望為所有項目使用相同的工具，流程和運行時。您可以使用Docker來做到這一點，從而將容器用於從Windows上運行的舊.NET MONOLITH到在Linux上運行的新GO應用程序的所有內容。您可以提供一個高容量容器平台來運行所有這些應用程序，因此您以相同的方式構建，部署和管理整個應用程序景觀。

技術創新不應與業務合理的應用程序分開。Docker是一些最大創新的核心，因此您可以繼續使用與探索新領域相同的工具和技術。最令人興奮的創新之一（當然是在容器之後）是無服務器功能。圖1.5顯示瞭如何在單個Docker群集中運行所有應用程序，即ge膠整體，新的雲本地應用程序和無服務器功能，該群集可以在雲或數據中心中運行。

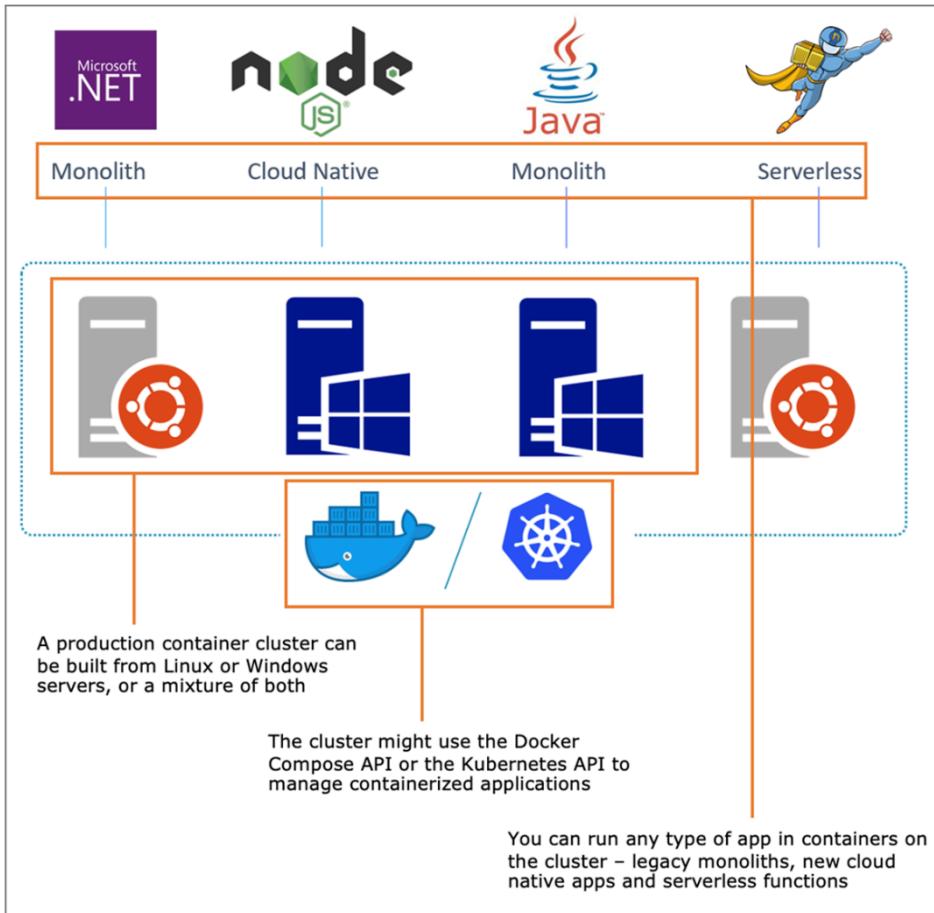


圖1.5運行Docker的單個服務器群可以運行每種類型的應用程序，並且無論使用哪種體系結構或技術堆棧，您都以相同的方式構建，部署和管理它們。

無服務器都是關於容器的。無服務器的目標是使開發人員編寫功能代碼，將其推到服務，並將服務構建和包裝代碼。當消費者使用該功能時，服務啟動了該功能的實例來處理請求。沒有構建服務器，管道或生產服務器可以管理；這一切都是由平台照顧的。

在引擎蓋下，所有無雲服務器選項都使用Docker（或兼容的容器運行時）包裝代碼和容器以運行功能。但是雲中的功能無法移植 - 您無法掌握您的AWS lambda功能並以Azure運行它，因為無服務器沒有開放標準。如果您想要無雲鎖定的無服務器，或者您在數據中心中運行，則可以使用nuclio，knative或openfaas在Docker中託管自己的平台，這些平台都是流行的無服務器框架。

其他主要創新（例如機器學習，區塊鍊和物聯網）受益於Docker的一致包裝和部署模型。您會發現所有部署到Docker Hub的主要項目 - Tensorflow和HyperLeDger是一個很好的例子。物聯網特別有趣，因為Docker與ARM合作，使容器成為Edge和IoT設備的默認運行時。

1.1.5 DevOps的數字轉換

所有這些方案都涉及技術，但是許多組織面臨的最大問題是運營的，尤其是對於大型和老年企業而言。團隊已被孤立地進入“開發人員”和“運營商”，負責項目生命週期的不同部分。發行時間的問題成為責備週期，並提出了高質量的大門，以防止未來的失敗。最終，您有這麼多優質的大門，您每年只能管理兩個或三個版本，它們具有風險和勞動力密集。

DevOps旨在通過將一個團隊擁有整個應用程序生命週期，將“DEV”和“OPS”結合到一個可交付的可交付中，將敏捷性帶入軟件部署和維護。DevOps主要是關於文化變革的，它可以將組織從巨大的季度發行到小型日常部署。但是，如果不更改團隊使用的技術，很難做到這一點。

運營商可能具有Bash，Nagios，PowerShell和System Center等工具背景。開發人員從事Make，Maven，Nuget和Msbuild的工作。當團隊不使用通用技術時，很難將團隊聚在一起，這是Docker真正幫助的地方。您可以通過轉移到容器的轉換來支持DevOps的轉換，突然間，整個團隊正在使用Dockerfiles和Docker撰寫文件，說相同的語言並使用相同的工具。

它也走得更遠。有一個強大的框架，用於實施稱為Calms的DevOps（文化，自動化，精益，指標和共享）。Docker在所有這些舉措上工作：自動化是運行容器的核心，分佈式應用程序建立在精益原則，生產應用程序中的指標以及部署過程中的指標，可以輕鬆發布，而Docker Hub都是關於共享而不是重複的工作。

1.2 這本書適合您嗎？

我在上一節中概述的五種情況涵蓋了IT行業中正在發生的所有活動，我希望很明顯，Docker是這一切的關鍵。如果您想讓Docker處理這種現實世界中的問題，這是您的書。它將帶您從零知識到生產級集群的容器中運行應用程序。

這本書的目的是教您如何使用Docker，因此我對Docker本身在引擎蓋下的工作方式沒有太多詳細介紹。我不會詳細討論集裝箱或下層詳細信息，例如Linux Cgroup和名稱空間或Windows Host Compute Service。如果您想要內部詞，傑夫·尼克洛夫（Jeff Nickoloff）和斯蒂芬·庫恩茲利（Stephen Kuenzli）的曼寧（Manning）的第二版是一個不錯的選擇。

本書中的樣本都是跨平台，因此您可以使用Windows，Mac或Linux（包括ARM處理器）進行工作，因此您也可以使用Raspberry Pi。我使用幾種編程語言，但是只有跨平台的語言，因此，我使用.NET而不是.NET Framework（僅在Windows上運行）。如果您想深入學習Windows容器，我的博客是一個很好的來源（<https://blog.sixeyed.com>）。

最後，這本書是關於Docker的，但是在生產部署方面，我將介紹Kubernetes，這是在雲中運行容器的最流行方式。Kubernetes是一項強大而復雜的技術，您可以使用Manning的

*Learn Kubernetes in a Month of Lunches*進行研究，但是從根本上說，這只是運行Docker容器的另一種方式，因此您在本書中學習的所有內容都適用。

1.3創建實驗室環境

現在讓我們開始。您需要與本書一起遵循的只是Docker和樣本的源代碼。

1.3.1安裝Docker

獲得Docker的最簡單方法是使用Docker桌面，該桌面可用於Windows，MacOS和Linux。這是一種商業產品，但是明確的許可使您可以在學習時免費安裝和使用它（即使在公司的機器上）。

免費的Docker社區版對開發甚至生產使用都很好。如果您正在運行Windows或MacOS的最新版本，那麼最好的選擇是Docker Desktop；舊版本可以使用Docker工具箱。Docker還為所有主要Linux發行版提供安裝軟件包。首先，使用最合適的選項安裝Docker，您需要為下載創建一個Docker Hub帳戶，該下載是免費的，可以讓您共享為Docker構建的應用程序。

在Windows 10上安裝Docker桌面10

您將需要最近版的Windows 10 Professional或Enterprise來使用Docker Desktop，並且需要確保已安裝了所有Windows更新。瀏覽到www.docker.com/products/docker-desktop，然後下載安裝程序並運行它，接受所有默認值。Docker Desktop運行時，您會在Windows時鐘附近的任務欄中看到Docker的鯨魚圖標。

在MacOS上安裝Docker桌面

您將需要最近發布MacOS，Docker Desktop支持英特爾和Apple Silicon CPS。瀏覽到www.docker.com/products/docker-desktop，然後下載安裝程序並運行它，接受所有默認值。Docker Desktop運行時，您會在時鐘附近的Mac菜單欄中看到Docker的鯨魚圖標。

在Linux上安裝Docker桌面

Docker桌面可用於所有主要Linux發行版的軟件包。發行版之間的安裝步驟不同 - 瀏覽到<https://docs.docker.com/desktop/install/linux-install/>並按照您自己的Linux系統的步驟操作。

Docker桌面替代方案

Docker Desktop將所有Docker組件都包裝到一個簡單的應用程序中，它為您提供了有用的GUI來幫助管理容器，但實際上您不需要它來運行容器。您可以在沒有桌面應用程序的情況下安裝Docker Engine和Docker命令行（瀏覽訪問<https://get.docker.com>以獲取將在Linux上執行此操作的腳本），或者您可以使用Rancher Desktop或colima等替代應用程序。

1.3.2 驗證您的Docker設置

幾個組件構成了Docker平台，但是對於本書，您只需要驗證Docker正在運行並使用了最新版本的Docker Compose。

首先，使用Docker版本命令檢查Docker本身：

```
> docker version
Client:
  Version:          27.0.3
  API version:     1.46
  Go version:      go1.21.11
  Git commit:      7d4bcd8
  Built:           Fri Jun 28 23:59:41 2024
  OS/Arch:         darwin/arm64
  Context:         desktop-linux

Server: Docker Desktop 4.32.0 (157355)
Engine:
  Version:          27.0.3
  API version:     1.46 (minimum version 1.24)
  Go version:      go1.21.11
  Git commit:      662f78c
  Built:           Sat Jun 29 00:02:44 2024
  OS/Arch:         linux/arm64
  Experimental:    false
```

您的輸出將與我的不同，因為版本將更改，並且您可能正在使用其他操作系統，但是只要您可以看到客戶端和服務器的版本號，Docker就可以正常工作。不必擔心客戶端和服務器暫時是什麼，您將在下一章中了解Docker的體系結構。

接下來，您需要測試Docker組成，這是與Docker一起使用的另一個命令。運行Docker撰寫版本以檢查：

```
>Docker撰寫版本Docker Compose版本v2.28.1-deskt  
op.1
```

同樣，您的確切輸出將與我的不同，但是只要您看到一個版本編號並且沒有錯誤消息，您就可以了。

1.3.3 下載本書的源代碼

本書的源代碼位於GitHub的公共GIT存儲庫中。如果您安裝了git客戶端，只需運行以下命令：

```
git克隆https://github.com/sixeyed/diamol.git cd diamol git chec  
kout 2e
```

如果您沒有git客戶端，請瀏覽<https://github.com/SIXEYED/DIAMOL/tree/2E>，然後單擊“代碼”按鈕將源代碼的郵政編碼下載到您的本地計算機上，然後展開存檔。

1.3.4 記住清理命令

Docker不會為您自動清理容器或應用程序包。退出Docker桌面（或停止Docker Service）時，您所有的容器都會停止，並且它們不使用任何CPU或內存，但是如果願意，您可以通過運行此命令在每章的結尾清理：

```
Docker容器RM -F $ (Docker Container LS -AQ)
```

而且，如果您想在練習後恢復磁盤空間，則可以運行此命令：

```
DocKER Image RM -F $ (Docker Image LS -F參考='Diamol/**' -Q)
```

Docker很明智地下載其需求，因此您可以隨時安全地運行這些命令。下次運行容器時，如果Docker找不到機器上的需求，它將為您下載。

1.4立即有效

“立即有效”是午餐叢書的另一個原則。在隨後的所有章節中，重點是學習技能並將其付諸實踐。

每章都以簡短的介紹為主題，然後是一試的練習，您可以在其中使用Docker付諸實踐。然後是一個回顧，還有一些細節，可以填補您潛水中可能遇到的一些問題。最後，有一個動手實驗室供您進入下一階段。

所有主題都圍繞在現實世界中真正有用的任務。您將學習如何在本章期間立即對該主題有效，並了解如何運用新技能。讓我們開始運行一些容器！

2了解Docker並運行Hello World

是時候與Docker動手實踐了，以便在Docker的核心功能上獲得很多經驗：在容器中運行應用程序。我還將介紹一些背景，這些背景將幫助您確切了解什麼是容器，以及為什麼容器是運行應用程序的輕量級方式。大多數情況下，您將遵循現在的練習，並運行簡單的命令，以了解這種新的使用應用程序的方式。

2.1在容器中運行Hello World

讓我們開始使用任何新計算概念的方式開始使用Docker：通過跑步Hello World。您在第1章中安裝了Docker，因此現在您應該啟動Docker桌面並打開您喜歡的終端 - 該終端可能是Mac上的終端或Linux上的Bash Shell，並且您應該在Windows中使用PowerShell。

您將向Docker發送命令，告訴它運行一個集裝箱，該容器打印出一些簡單的“Hello，World”文本。

現在嘗試

輸入此命令，該命令將運行Hello World容器：

```
Docker Run Diamol/CH02-Hello-Diamol :2e
```

完成本章後，您將確切了解這裡發生的事情。目前，只需看看輸出即可。它將像圖2.1一樣。

```

PS->docker run diamol/ch02-hello-diamol:2e
Unable to find image 'diamol/ch02-hello-diamol:2e' locally
2e: Pulling from diamol/ch02-hello-diamol
690e87867337: Pull complete
b2ba7d3da949: Pull complete
0b33baac0ae7: Pull complete
Digest: sha256:a2f8d536b77e0931b813c33e66f4ec1cb7ec4a17acfdbede5e07d98ad393969
Status: Downloaded newer image for diamol/ch02-hello-diamol:2e

-----
Hello from Chapter 2!
-----
My name is:
2f97e02a5924
-----
I'm running on:
Linux 6.6.32-linuxkit aarch64
-----
My address is:
inet addr:172.17.0.2 Bcast:172.17.255.255 Mask:255.255.0.0
-----
```

This run command starts a container from an application package called `diamol/ch02-hello-diamol:2e`

That package doesn't exist on this machine so Docker downloads it first

Docker runs a container from the package, and these logs are the output from the application

圖2.1運行Hello World容器的輸出。您可以看到Docker下載應用程序包（稱為圖像），在容器中運行應用程序並顯示輸出。

該輸出有很多。我將修剪未來的代碼列表以使它們簡短，但這是第一個，我想完整顯示它，以便我們可以剖析它。

首先，實際發生了什麼？ Docker Run命令告訴Docker在容器中運行一個應用程序。該應用程序已經包裝在Docker中運行，並已在公共網站上發布，任何人都可以訪問。容器軟件包（Docker稱為*image*）命名為 `Diamol/ch02-hello-diamol:2e`（我在本書中使用首字母縮寫 `diamol` - 它代表 *Docker In A Month Of Lunches*，而在第二版中則代表 `2e`）。您剛輸入的命令告訴Docker從該圖像運行一個容器。

Docker需要在圖像使用圖像運行容器之前，需要在本地進行本地的副本。第一次運行此命令時，您將沒有圖像的副本，並且可以在第一個輸出行中看到這一點：無法在本地找到圖像。然後Docker下載圖像（Docker調用 *pulling*），您可以看到圖像已下載。

現在，Docker使用該圖像啟動一個容器。該圖像包含應用程序的所有內容，以及告訴Docker如何啟動應用程序的說明。此圖像中的應用程序只是一個簡單的腳本，您會看到輸出，該輸出從第2章開始Hello！它寫下了有關正在運行的計算機的一些細節：

- 機器名稱 - 在此示例中，2F97E02A5924操作系統 - 在此示例中，Linux 6.6.6.
- 32-linuxkit aarch64網絡地址 - 在此示例中，172.17.0.2
-

我說您的輸出將是“這樣的東西” - 它不會完全一樣，因為容器獲取的某些信息取決於您的計算機。我在帶有Linux操作系統和64位ARM處理器的機器上運行了此操作。如果您使用Windows容器在Windows Server計算機上運行它，則從我運行的開始的線路將顯示這樣的內容：

```
-----  
我正在運行：Microsoft Windows [版本10.0.17763.557]  
] -----
```

如果您在較舊的Raspberry Pi上運行，則輸出將表明它正在使用其他處理器（ARMV7L是ARM 32位處理芯片的代號，AARCH64為64位ARM，X86_64是Intel 64位芯片的代碼）：

```
-----  
我正在運行：Linux 4.19.42-V7  
+ ARMV7L -----
```

這是一個非常簡單的示例應用程序，但它顯示了核心Docker工作流程。有人打包他們的應用程序以在容器中運行（我為此應用程序做了，但是您將在下一章中自己執行），然後發布它，以便其他用戶可以使用。然後，任何具有訪問權限的人都可以在容器中運行該應用程序。Docker稱此*build, share, run*。

這是一個非常強大的概念，因為無論應用程序多麼複雜，工作流程都是相同的。在這種情況下，這是一個簡單的腳本，但它可以是一個具有多個組件，配置文件和庫的Java應用程序。工作流將完全相同。而且，可以將Docker映像打包以在支持Docker的任何計算機上運行，這使該應用程序完全可移植 - 便攜式性是Docker的主要好處之一。

如果您使用同一命令運行另一個容器，會發生什麼？

現在嘗試

重複完全相同的Docker命令：

```
Docker Run Diamol/CH02-Hello-Diamol : 2e
```

您會看到與第一次運行相似的輸出，但是會有差異。Docker已經在本地擁有該圖像的副本，因此它不需要先下載圖像。它直接運行容器。容器輸出顯示相同的操作系統詳細信息，因為您使用的是同一台計算機，但是容器的名稱將有所不同，IP地址可能會更改：

```
-----您  
好，第2章！ -----  
-----我的名字是：D147  
4EFC0E67 -----  
-----
```

我在奔跑：

Linux 6.6.32 Linuxkit Aarch64

我的地址是：

```
INET ADDR : 172.17.0.2 broadcast : 172.17.255.255 mask : 255.255.0.0 -----  
-----
```

現在，我的應用程序正在使用名稱D1474EFC0E67的容器上運行，但它重複使用IP地址172.17.0.2。名稱每次都會更改，IP地址通常會更改，但是每個容器在同一台計算機上都運行，那麼這些不同的機器名稱和網絡地址來自何處？我們將深入研究一個小理論，然後再解釋這一點，然後回到了練習中。

2.2那麼，什麼是容器？

docker *container*與物理容器的想法相同，想像它的盒子裡有一個應用程序。在包裝盒內，該應用程序似乎擁有一台計算機：它具有自己的機器名稱和IP地址，並且還具有自己的磁盤驅動器（Windows Server容器也有自己的Windows註冊表）。圖2.2顯示了該應用程序如何由容器盒裝。

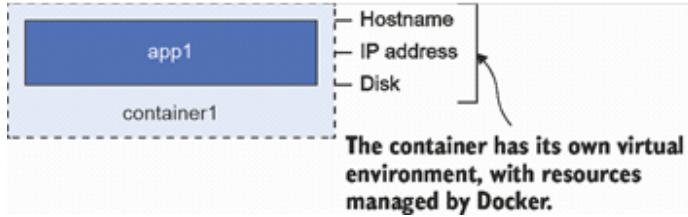


圖2.2容器環境內的應用程序

這些都是虛擬資源 - 主機名，IP地址和文件系統是由Docker創建的。它們是由Docker管理的邏輯對象，並且都將它們共同創建一個可以運行應用程序的環境。那就是容器的“盒子”。

盒子內的應用程序在框外面看不到任何內容，但是該框在計算機上運行，並且該計算機也可以運行許多其他盒子。這些框中的應用程序具有自己的獨立的虛擬環境（由Docker管理），但它們都共享計算機的CPU和內存，並且都共享了計算機的操作系統。您可以在圖2.3中看到同一計算機上的容器如何隔離。

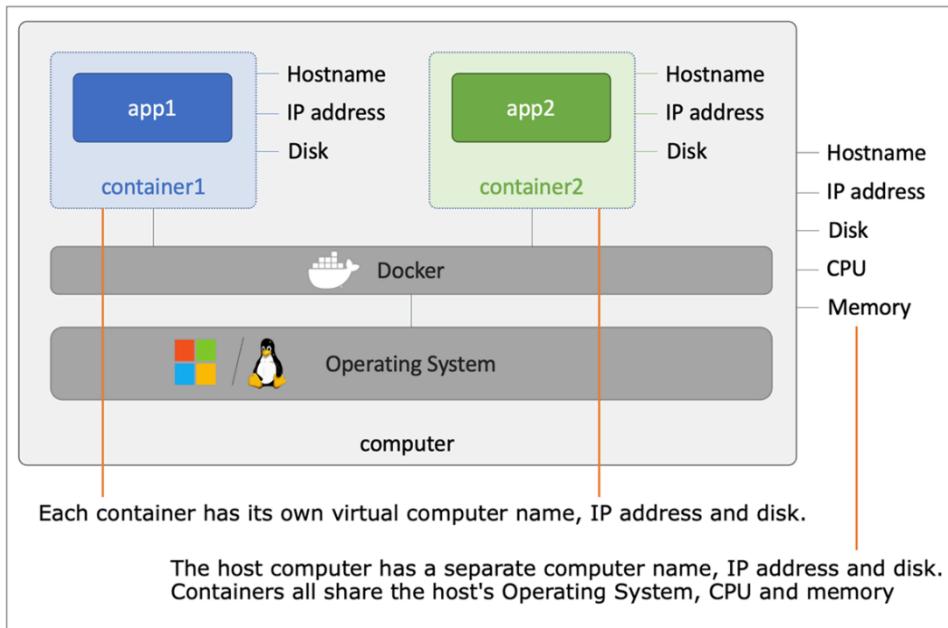


圖2.3一台計算機上的多個容器共享相同的操作系統，CPU和內存。

為什麼這麼重要？它解決了計算中的兩個衝突問題：隔離和密度。*Density*意味著在計算機上運行盡可能多的應用程序，以利用您擁有的所有處理器內核和內存。但是應用程序可能無法與其他應用程序合作 - 它們可能會使用不同版本的Java或.NET，它們可能會使用不兼容的工具或庫的版本，或者一個人可能具有繁重的工作量並餓死其他處理能力。應用程序確實需要彼此隔離，這使您無法在一台計算機上運行很多應用程序，因此您無法獲得密度。

解決該問題的最初嘗試是使用虛擬機（VM）。*Virtual machines*的概念與容器相似，因為它們為您提供了一個框來運行您的應用程序，但是VM的框需要具有自己的操作系統 - 它不共享VM運行的計算機OS。比較圖2.3（顯示多個容器，圖2.4），該圖2.4顯示了一台計算機上的多個VM。

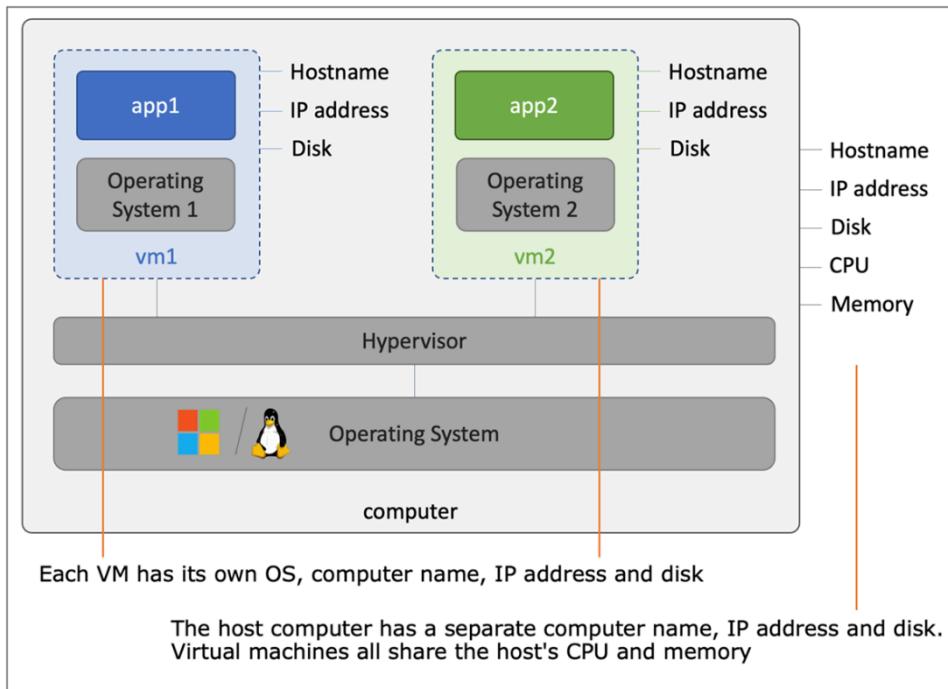


圖2.4一台計算機上的多個VM都有自己的操作系統。

這看起來像是圖表的很小差異，但它具有巨大的影響。每個VM都需要自己的操作系統，並且操作系統可以使用千兆字節的內存和大量CPU時間 - 促進了應為您的應用程序可用的計算功率。還有其他問題，例如OS的許可費用以及安裝操作系統更新的維護負擔。VM以密度成本提供隔離。

容器給你倆。每個容器共享運行該容器的計算機的操作系統，這使其非常輕巧。容器快速啟動並運行精益，因此您可以在同一硬件上運行比VM的容器更多的容器 - 通常是數量的五到十倍。您會得到密度，但是每個應用程序都有自己的容器，因此您也會得到隔離。這是Docker：效率的另一個關鍵特徵。

現在您知道Docker是如何做魔術的。在下一個練習中，我們將與容器更緊密地合作。

2.3連接到遠程計算機之類的容器

我們運行的第一個容器只做了一件事 - 應用程序打印出一些文本，然後結束。在很多情況下，您想做的一件事。也許您有一組自動化一些過程的腳本。這些腳本需要一組特定的工具來運行，因此您不能僅僅與同事共享腳本；您還需要共享一個文檔，以描述設置所有工具，並且您的同事需要花費數小時安裝它們。取而代之的是，您可以在Docker映像中包裝工具和腳本並共享圖像，然後您的同事可以在沒有額外的設置工作的情況下將腳本運行。

您也可以以其他方式使用容器。接下來，您將看到如何運行容器並連接到容器內部的終端，就像您連接到遠程計算機一樣。您使用同一Docker Run命令，但是您傳遞了一些其他標誌來運行具有連接終端會話的交互式容器。

現在嘗試

在終端會話中運行以下命令：

```
Docker Run-相互作用-TTY DIAMOL/基礎：2E
```

交互式標誌告訴Docker您要設置與容器的連接，而TTY標誌意味著您要連接到容器內的終端會話。輸出將顯示Docker拉動圖像，然後您將帶有命令提示符。如圖2.5所示，該命令提示是在容器內進行終端會話。

```
PS>docker run --interactive --tty diamol/base:2e
Unable to find image 'diamol/base:2e' locally
2e: Pulling from diamol/base
690e87867337: Already exists
ff40acd25796: Pull complete
Digest: sha256:27de1d9d65fb869b12122194ba309015518c60357e98ec5b05ddbb0a76bc95e9
Status: Downloaded newer image for diamol/base:2e
/ #
```

This command line is connected to a terminal session inside the container

This run command starts an interactive container container from an image called `diamol/base:2e`

圖2.5運行交互式容器並連接到容器的終端。

完全相同的Docker命令在Windows Server容器上以相同的方式工作，但是您會掉入Windows命令行會話中：

```
Microsoft Windows [版本10.0.17763.557] (C) 2018 Microsoft Corporation。版權所有。
```

```
C:\>
```

無論哪種方式，您現在都在容器中，並且可以運行通常可以在操作系統的命令行中運行的任何命令。

現在嘗試

運行命令主機名和日期，您會看到容器環境的詳細信息：

```
/# 主機名A41AD305D64D /# 日期
陽台8月11日15:50:27 UTC 2024
```

如果您想進一步探索，則需要熟悉命令行，但是您在這裡擁有的是連接到遠程計算機的本地終端會話 - 機器恰好是在計算機上運行的容器。如果您使用安全的Shell（SSH）連接到遠程Linux計算機或遠程桌面協議（RDP）連接到遠程Windows Server Core Machine，則您將獲得與Docker相同的體驗。

請記住，該容器正在共享計算機的操作系統，這就是為什麼如果您正在使用Windows，如果您正在運行Linux和Windows命令行，則可以看到Linux Shell的原因。有些命令對於兩者都是相同的（嘗試ping google.com），但其他命令具有不同的語法（您使用ls在linux中列出目錄內容，並在Windows中使用DIR）。

Docker本身俱有相同的行為，無論您使用哪種操作系統或處理器。它是容器內部的應用程序，它看到它正在基於英特爾的Windows機器或基於ARM的Linux One上運行。您以相同的方式管理使用Docker的容器。

現在嘗試

打開一個新的終端會話，您可以使用此命令看到所有運行容器的詳細信息：

Docker容器LS

輸出顯示有關每個容器的信息，包括使用的圖像，容器ID和命令Docker在啟動時在容器內運行 - 這是一些縮寫的輸出：

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
a41ad305d64d	diamol/base:2e	"/bin/sh"	3 minutes ago	Up 3 minutes

如果您的眼睛敏銳，您會注意到容器ID與容器內的主機名相同。Docker為其創建的每個容器分配一個隨機ID，該ID的一部分用於主機名。您可以使用許多Docker容器命令與特定容器進行交互，您可以使用所需的容器ID的前幾個字符識別。

現在嘗試

Docker容器頂部列出了在容器中運行的過程。您需要使用自己的容器ID-我使用A4作為ID A41AD305D64D的簡短形式：

```
> docker container top a4
UID      PID      STIME      TIME      CMD
root     670      15:48      0:00      /bin/sh
```

如果您在容器中運行多個進程，Docker將向他們展示全部。Windows容器將是這種情況，除了容器應用程序外，這些容器始終運行多個背景過程。

現在嘗試

Docker容器日誌顯示該容器收集的任何日誌條目：

```
>Docker容器日誌A4 / # 主機名A
41AD305D64D
```

Docker使用來自容器中應用程序的輸出收集日誌條目。在此終端會話的情況下，我會看到我運行的命令及其結果，但是對於真實的應用程序，您會看到代碼的日誌條目。例如，Web應用程序可以為所處理的每個HTTP請求編寫日誌條目，這些內容將顯示在容器日誌中。

現在嘗試

Docker Container Inspect向您顯示了容器的所有詳細信息：

```
>Docker容器檢查F8 [{" "ID" : " A41AD305D64D1BF8CA001B0038B2E543F78173780E0062D558EEE
F11360421BF" , "創建" : " 2024-08-11T15 :" 2024-08-11T15 :
```

完整的輸出顯示了許多低級信息，包括容器虛擬文件系統的路徑，容器內部運行的命令以及該容器已連接到的虛擬Docker網絡，如果您在應用程序中跟蹤問題，則可以很有用。它是大部分JSON，這有助於與腳本自動化，但對於書籍中的代碼列表不太好，因此我剛剛顯示了前幾行。

這些是您將一直使用的命令，當您使用容器，需要對應用程序問題進行故障排除，是否要檢查進程是否使用大量CPU時，或者您是否想查看網絡Docker已為容器設置設置時。

這些練習還有另一點，這是為了幫助您意識到，就Docker而言，容器看起來都一樣。Docker在每個應用程序的頂部都增加了一個一致的管理層。您可以在Linux容器中運行10年曆史的Java應用程序，在Windows容器中運行的15歲的.NET應用程序，以及在Raspberry Pi上運行的全新GO應用程序。您將使用完全相同的命令來管理它們 - 運行啟動應用程序，登錄以讀取日誌，頂部以查看過程並進行檢查以獲取詳細信息。

現在，您已經看到了更多的Docker可以做的事情；我們將完成一些練習，以進行更有用的應用程序。您可以關閉打開的第二個終端窗口（運行Docker容器日誌），返回到仍然連接到容器的第一個終端，然後運行出口以關閉終端會話。

2.4在容器中託管網站

到目前為止，我們運行了一些容器。第一對夫婦運行了打印一些文本然後退出的任務。下一個使用的交互式標誌將我們連接到容器中的終端會話，該終端會一直在運行，直到我們退出會話為止。Docker容器LS將顯示您沒有容器，因為命令僅顯示運行容器。

現在嘗試

運行Docker容器LS - ALL，該容器以任何狀態顯示所有容器：

```
> docker container ls --all
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
a41ad305d64d        diamol/base:2e      "/bin/sh"          11 minutes ago   Exited (0) 3 seconds ago
37cd349ca6da        diamol/ch02-hello-diamol:2e    "/bin/sh -c ./cmd.sh"  12 minutes ago   Exited (0) 12 minutes ago
2f97e02a5924        diamol/ch02-hello-diamol:2e      "/bin/sh -c ./cmd.sh"  18 minutes ago   Exited (0) 18 minutes ago
```

容器已退出狀態。這裡有幾個關鍵概念需要理解。

首先，僅在容器內的應用程序運行時才運行容器。申請過程結束後，容器將進入退出狀態。退出的容器不使用任何CPU時間或內存。腳本完成後，“Hello World”容器將自動退出。一旦我們退出終端應用，我們就可以連接到交互式容器。

其次，容器退出時不會消失。仍存在退出狀態中的容器，這意味著您可以再次啟動它們，檢查日誌並從容器的文件系統中複製文件。您只會看到帶有Docker容器LS的運行容器（除非您添加全標誌，否則Docker不會刪除退出的容器，除非您明確告訴它。退出的容器仍在磁盤上佔用空間，因為它們的文件系統保存在計算機的磁盤上）。

那麼，啟動留在後台並繼續運行的容器又如何呢？這實際上是Docker的主要用例：運行的服務器應用程序，例如網站，批處理過程和數據庫。

現在嘗試

這是一個簡單的例子，在容器中運行網站：

```
Docker Container Run-Detach-Publish 8088 : 80 Diamol/ch02-hello-diamol-web : 2e
```

這次，您唯一看到的輸出是一個長容器ID，然後您將返回命令行。容器仍在後台運行。

現在嘗試

運行Docker 容器LS，您會發現新容器具有

狀態提升：

```
> docker container ls
CONTAINER ID  IMAGE                               COMMAND      CREATED
          STATUS     PORTS
5c29724b22a1  diamol/ch02-hello-diamol-web:2e  "httpd-foreground"  14 seconds ago
Up 13 seconds   0.0.0.0:8088->80/tcp    relaxed_albattani
```

您剛剛使用的圖像是Diamol/CH02-Hello-Diamol-Web。 該圖像包括Apache Web服務器和簡單的HTML頁面。運行此容器時，您會運行完整的Web服務器，並託管一個自定義網站。坐在後台並收聽網絡流量的容器（在這種情況下為HTTP請求）需要在容器運行命令中進行幾個額外的標誌：

- -Detach - 在背景中啟動容器，並顯示容器ID-發布 - 發布從容器到計算機的端口發行
-

運行一個獨立的容器只是將容器放在背景中，以便啟動並保持隱藏狀態，例如Linux守護程序或Windows服務。發布端口需要更多的解釋。安裝Docker時，它會將自己注入計算機的網絡層。進入計算機的流量可以被Docker攔截，然後Docker可以將該流量發送到容器中。

默認情況下，容器不會暴露於外界。每個人都有自己的IP地址，但這是Docker為Docker管理的網絡創建的IP地址，該容器未連接到計算機的物理網絡上。 發布容器端口意味著Docker在計算機端口上聆聽網絡流量，然後將其發送到容器中。在前面的示例中，發送到端口8088上的計算機的流量將發送到端口80上的容器中 - 您可以在圖2.6中看到流量流。

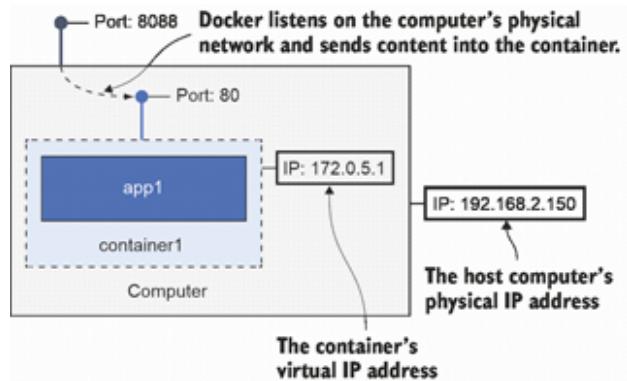


圖2.6計算機和容器的物理和虛擬網絡

在此示例中，我的計算機是運行Docker的計算機，它具有IP地址192.168.2.150。這是我物理網絡的IP地址，當我的計算機連接時路由器分配了它。Docker在該計算機上運行一個容器，並且該容器具有IP地址172.0.5.1。該地址由Docker分配給由Docker管理的虛擬網絡。我的網絡中沒有其他計算機可以連接到容器的IP地址，因為它僅存在於Docker中，但是由於端口已發布，它們可以將流量發送到容器中。

現在嘗試

瀏覽到`http://localhost:8088`在瀏覽器上。這是對本地計算機的HTTP請求，但是響應（見圖2.7）來自容器。（您絕對不會從這本書中學到的一件事是有效的網站設計。）

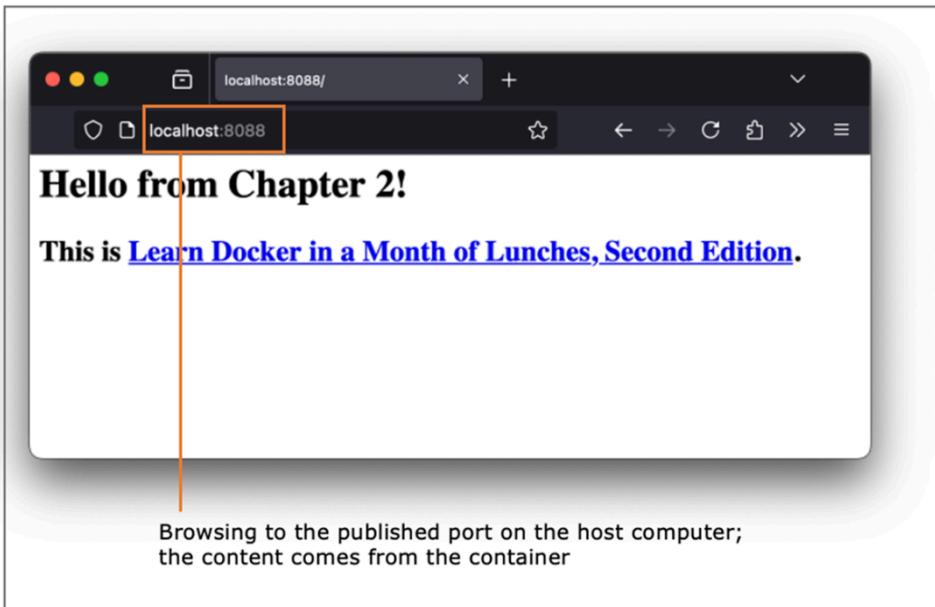


圖2.7 Web應用程序，從本地機器上的容器中提供

這是一個非常簡單的網站，但是即使如此，該應用程序仍然受益於Docker帶來的可移植性和效率。Web內容與Web服務器打包，因此Docker Image具有所需的一切。Web開發人員可以在其筆記本電腦上運行一個容器，整個應用程序（從HTML到Web服務器堆棧）將與運營商在生產中的服務器群集上在100個容器上運行該應用程序完全相同。

該容器中的應用程序無限期地運行，因此容器也將繼續運行。您可以使用我們已經用來管理它的Docker容器命令。

現在嘗試

Docker容器統計數據是另一個有用的：它顯示了容器正在使用的CPU，內存，網絡和磁盤的實時視圖。您需要添加容器的ID，對於Linux和Windows容器，輸出略有不同：

```
> docker container stats 5c2
CONTAINER ID   NAME          CPU %     MEM USAGE / LIMIT      MEM %     NET
I/O           BLOCK I/O      PIDS
5c29724b22a1   relaxed_albattani   0.01%    4.867MiB / 11.67GiB   0.04%
3.29kB / 1.4kB  233kB / 4.1kB    82
```

使用容器工作後，您可以使用Docker容器RM和容器ID將其卸下，如果容器仍在運行，則使用-force標誌將其強制刪除。

我們將以最後一個命令結束此練習，您將習慣於定期運行。

現在嘗試

運行此命令以刪除所有容器：

```
Docker Container RM - Force $ (Docker Container ls -All -quiet)
```

\$ () 語法將輸出從一個命令發送到另一個命令 - 它在Linux和Mac終端以及PowerShell上也同樣可以。 組合這些命令會產生計算機上所有容器ID的列表，並將其全部刪除。這是整理您的容器的好方法，但要謹慎使用它，因為它不會要求確認。

2.5了解Docker如何運行

盧比

現在，我們已經在本章中進行了很多嘗試，現在您應該對使用容器合作的基本知識感到滿意。

在本章中的第一個嘗試中，我討論了 *build*，*share*，*run* Docker核心的工作流。該工作流使分發軟件非常容易 - 我已經構建了所有示例容器圖像並共享它們，因為您知道您可以在Docker中運行它們，並且它們將為您與我一樣。現在，大量項目使用Docker作為發佈軟件的首選方法。您可以嘗試使用您在此處使用的相同類型的Docker Container run命令，嘗試使用新軟件（例如，彈性搜索或最新版本的SQL Server或Ghost Blogging引擎）。

我們將以更多的背景結束，以便您對與Docker運行應用程序的實際情況有深入的了解。安裝Docker和運行容器非常簡單 - 實際上涉及一些不同的組件，您可以在圖2.8中看到。

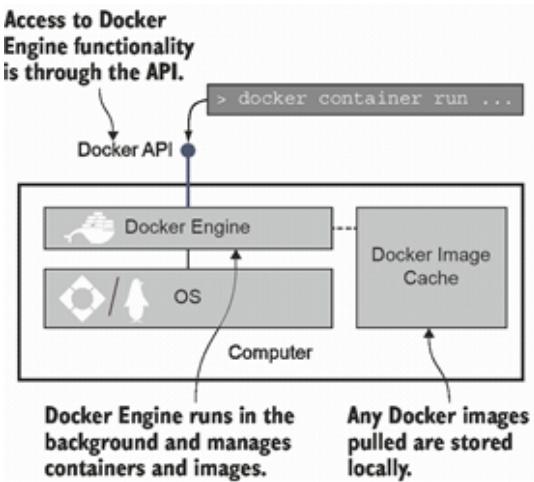


圖2.8 Docker的組件

這些是Docker的關鍵組成部分：

- *Docker Engine*是Docker的管理組件。它照顧了本地圖像緩存，在需要時下載圖像並在下載圖像時重複使用它們。它還可以與操作系統一起創建容器，虛擬網絡和所有其他Docker資源。引擎是一個始終運行的背景過程（例如Linux守護程序或Windows服務）。Docker引擎可以通過*Docker API*提供所有功能，該功能只是標準的基於HTTP的REST API。您可以配置引擎以使API僅從本地計算機（默認計算機）訪問，也可以使其可用於網絡上的其他計算機。*Docker command-line interface (CLI)*是Docker API的客戶端。當您運行Docker命令時，CLI實際上將它們發送到Docker API，並且Docker Engine可以完成工作。
-

理解Docker的建築是一件好事。與Docker Engine進行交互的唯一方法是通過API，您有不同的選擇來訪問API並保護它。CLI通過向API發送請求來工作。

到目前為止，我們已經使用CLI來管理Docker正在運行的同一台計算機上的容器，但是您可以將CLI指向該機器上的遠程計算機運行Docker和控制容器上的API，這就是您在不同環境中管理容器的操作，例如構建服務器，測試和生產。Docker API在每個操作系統上都是相同的，因此您可以在Windows筆記本電腦上使用CLI來管理Raspberry Pi上的容器或云中的Linux服務器上的容器。

Docker API具有已發布的規範，並且Docker CLI並不是唯一的客戶。有幾種連接到Docker API的圖形用戶界面，並為您提供一種與容器交互的視覺方式。API揭示了有關容器，圖像和其他資源Docker管理的所有詳細信息，以便它可以從Docker Desktop中的圖2.9中的儀表板供電。

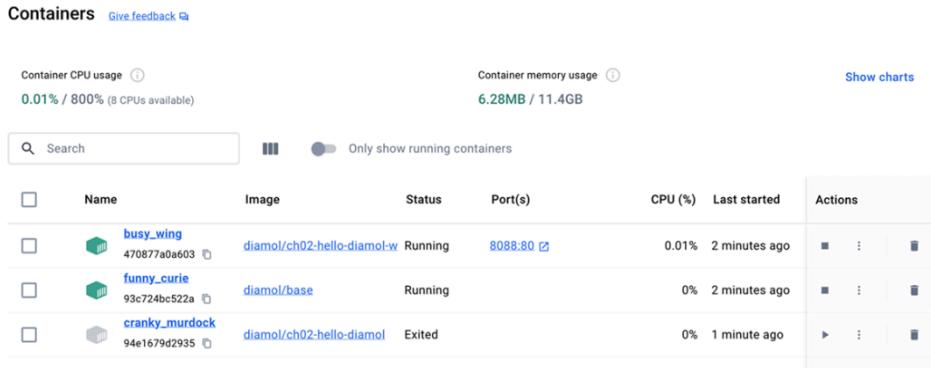


圖2.9 Docker桌面中的容器儀表板

我們不會比這更深入地深入研究碼頭建築。Docker Engine使用稱為Containerd的組件實際管理容器，並且集裝箱又利用操作系統功能來創建容器的虛擬環境。

您無需了解容器的低級詳細信息，但是很高興知道這一點：Containerd是由Cloud Native Computing Foundation監督的開源組件，並且運行容器的規範是開放和公共的；它稱為“開放容器倡議”（OCI）。

Docker是迄今為止最受歡迎，最易於使用的容器平台，但不是唯一的。您可以自信地投資於容器，而不必擔心自己被鎖定在一個供應商的平台中。

2.6 實驗室：探索容器文件系統

這是書中的第一個實驗室，所以這就是全部內容。 該實驗室為您設置了一個自己要完成的任務，這將幫助您鞏固本章中所學到的知識。我會給您一些指導和一些提示，但主要是，這是關於您現在進行的練習，並找到自己解決問題的方法。

每個實驗室都有本書的GitHub存儲庫中的示例解決方案。 值得花一些時間自己嘗試一下，但是如果想要檢查我的解決方案，可以在這裡找到它：<https://github.com/SIXEYED/DIAMOL/tree/2E/CH02/LAB>。

我們要去：您的任務是從本章中運行網站容器，但要替換index.html文件，以便當您瀏覽到容器時，您會看到其他主頁（您可以使用任何喜歡的內容）。 請記住，該容器具有自己的文件系統，並且在此應用程序中，該網站正在為容器文件系統上的文件提供服務。

這裡有一些提示可以讓您前進：

- 您可以運行Docker容器以獲取可以在容器上執行的所有操作的列表。 添加 - 支持任何Docker命令，您會看到更多詳細的幫助文本。 在Diamol/ch02-hello-diamol-web中，網站的內容來自目錄/usr/usr/local/apache2/htdocs（即c:\usr\usr\usr\usr\local\apache2\apache2\htdocs on Windows上）。
-

Good luck :)

3構建自己的docker映像

您在上一章中運行了一些容器，並使用Docker來管理它們。無論應用程序使用哪種技術，容器在應用程序中提供一致的體驗。到目前為止，您使用了我構建和共享的Docker圖像；現在，您將看到如何構建自己的圖像。在這裡，您將了解dockerfile語法以及當您將自己的應用程序集體化時始終使用的一些關鍵模式。

3.1 使用Docker Hub的容器圖像

我們將從您將在本章中構建的圖像的完成版本開始，以便您可以看到它的設計如何與Docker合作。現在，它使用一個名為Web-Ping的簡單應用程序進行了嘗試，該應用程序檢查網站是否已啟動。該應用程序將在容器中運行，並每三秒鐘向我的博客提供HTTP請求，直到停止容器為止。

您從第2章中知道，如果Docker容器運行尚未在計算機上，將在本地下載該容器圖像。這是因為軟件分佈已內置在Docker平台中。您可以離開Docker為您管理此操作，以便在需要時拉出圖像，也可以使用Docker CLI明確拉圖像。

現在嘗試

將Web-Ping應用程序的容器圖像拉動：

```
Docker Image Pul diamol/ch03-web-ping :2e
```

您會看到圖3.1中類似的輸出。

```
PS>docker image pull diamol/ch03-web-ping:2e
2e: Pulling from diamol/ch03-web-ping
0362ad1dd800: Already exists
b09a182c47e8: Already exists
39d61d2ed871: Already exists
b4e2115e274a: Already exists
4e6c11a417cf: Pull complete
e4df4747f674: Pull complete
Digest: sha256:648b2632b15c0be58c81cdd9032b3e101bb1a770f29ece1fc45bac8e508e2b74
Status: Downloaded newer image for diamol/ch03-web-ping:2e
docker.io/diamol/ch03-web-ping:2e
--
```

The name of the image to pull is `diamol/ch03-web-ping:2e`

One image is physically stored as many image layers

圖3.1從Docker Hub中拉出圖像

圖像名稱是Diamol/CH03-Web-Ping：2E，它存儲在Docker Hub上，Docker Hub是Docker尋找圖像的默認位置。 圖像服務器稱為*registries*，Docker Hub是可以免費使用的公共註冊表。 Docker Hub還具有Web界面，您會在<https://Hub>上找到有關此圖像的詳細信息。docker.com/r/diamol/ch03-web-ping。

Docker Image Pull命令中有一些有趣的輸出，它向您展示瞭如何存儲圖像。從邏輯上講，Docker映像是一件事 - 您可以將其視為包含整個應用程序堆棧的大型ZIP文件。 該圖像與我的應用程序代碼一起具有node.js運行時。

在拉動期間，您不會看到一個文件下載 - 您會看到許多正在進行的下載。 這些被稱為*image layers*。 Docker映像在物理上存儲為許多較小的文件，並且Docker將它們組裝在一起以創建容器的文件系統。 拉動所有圖層時，可以使用完整的圖像。

現在嘗試

讓我們從圖像中運行一個容器，看看應用程序的作用：

```
Docker容器運行-D -NAME Web-Ping Diamol/ch03-Web-Ping :2E
```

-d標誌是-Detach的短形式，因此該容器將在後台運行。該應用程序像沒有用戶界面的批處理作業一樣運行。與我們在第2章中分離的網站容器不同，該容器不接受傳入的流量，因此您不需要發布任何端口。

該命令中有一個新標誌，即 - 名稱。 您知道可以使用Docker生成的ID使用容器，但您也可以給他們一個友好的名字。 該容器稱為Web-Ping，您可以使用該名稱參考容器，而不是使用隨機ID。

我的博客現在被應用程序在您的容器中運行的應用程序所吸引。該應用程序以無盡的循環運行，您可以使用您熟悉的第2章中熟悉的同一docker容器命令看到它在做什麼。

現在嘗試

看看應用程序的日誌，這些日誌正在由Docker收集：

```
Docker容器日誌Web-Ping
```

您會在圖3.2中看到類似的輸出，顯示該應用程序向博客提出HTTP請求。 com。

```
PS>docker container run -d --name web-ping diamol/ch03-web-ping:2e  
077f519ed38cc7df67731a5c039e5d137f95f9830057834c46d077887df9e986  
PS>  
PS>  
PS>docker container logs web-ping  
*** web ping *** Pinging: blog.sixeyed.com; method: HEAD; 3000ms intervals  
Making request number: 1; at 1723393203540  
Got response status: 200 at 1723393203868; duration: 328ms  
Making request number: 2; at 1723393206548  
Got response status: 200 at 1723393206586; duration: 38ms  
Making request number: 3; at 1723393209553  
Got response status: 200 at 1723393209589; duration: 36ms  
--
```

This runs the `web-ping` application in a background container

Container logs show the app making HTTP requests to `blog.sixeyed.com`

圖3.2在操作中的Web-Ping容器，將不斷的流量發送到我的博客

一個提出Web請求並記錄響應多長時間的應用程序相當有用 - 您可以將其用作監視網站正常運行時間的基礎。但是這個應用程序看起來是使用我的博客的硬編碼，因此除了我以外，它都沒有用。

除了不是。實際上，該應用程序可以配置為使用不同的URL，請求之間的不同間隔，甚至是其他類型的HTTP調用。該應用讀取該應用程序應從系統環境變量中使用的配置值。

*Environment variables*只是操作系統提供的鍵/值對。它們以同樣的方式在Windows和Linux上工作，並且是存儲小型數據的一種非常簡單的方法。Docker容器也具有環境變量，但是它們不是來自計算機的操作系統，而是由Docker設置的，就像Docker創建容器的主機名和IP地址一樣。

Web-Ping映像具有為環境變量設置的一些默認值。運行容器時，這些環境變量將由Docker填充，這就是該應用程序用於配置網站URL的方法。創建容器時，您可以為環境變量指定不同的值，這將改變應用程序的行為。

現在嘗試

卸下現有容器，並運行一個新的容器，該容器具有為目標環境變量指定的值：

Docker RM -F網絡填充

docker容器運行-env target = google.com diamol/ch03-web-ping : 2e

您這次的輸出看起來像我的圖3.3中的我。

```
PS>docker container run --env TARGET=google.com diamol/ch03-web-ping:2e
** web-ping ** Pinging: google.com; method: HEAD; 3000ms intervals
Making request number: 1; at 1723393378191
Got response status: 301 at 1723393378191; duration: 90ms
Making request number: 2; at 1723393381103
Got response status: 301 at 1723393381162; duration: 59ms
Making request number: 3; at 1723393384109
```

The environment variable sets the URL the app will use

The same app from the same Docker image is now pinging google.com

圖3.3來自同一圖像的一個容器，將流量發送到Google

這個容器正在做一些不同的事情。首先，它是因為您沒有使用-DETACH標誌而進行交互式運行，因此該應用程序的輸出顯示在您的控制台上。容器將繼續運行，直到您按CTRL-C結束應用程序。其次，現在是pinging google.com，而不是blog.sixeyed.com。

這是您本章的主要要點之一：Docker Images可以用該應用程序的默認配置值打包，但是當您運行容器時，您應該能夠提供不同的配置設置。

環境變量是實現這一目標的一種非常簡單的方法。Web-Ping應用程序代碼尋找具有關鍵目標的環境變量。該鍵是圖像中使用blog.sixeyed.com的值設置的，但是您可以通過使用-env flag通過Docker Container Run命令提供不同的值。圖3.4顯示了容器如何具有自己的設置，彼此之間以及圖像不同。

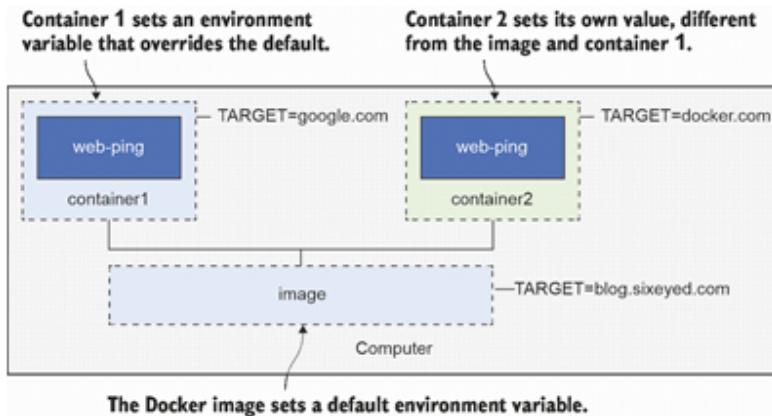


圖3.4碼頭圖像和容器中的環境變量

主機計算機也有自己的一組環境變量，但它們與容器分開。每個容器僅具有Docker填充的環境變量。圖3.4的重要方面是，每個容器中的Web-Ping應用程序都是相同的 - 它們使用相同的圖像，因此該應用程序運行的是完全相同的二進製文件集，但是由於配置，該行為是不同的。

這取決於Docker Image的作者提供了這種靈活性，並且您將在從Dockerfile構建第一個Docker映像時，現在就可以看到如何做到這一點。

3.2編寫您的第一個Dockerfile

Dockerfile是您編寫的簡單腳本來包裝應用程序 - 這是一組指令，而Docker Image是輸出。Dockerfile語法很容易學習，您可以使用Dockerfile包裝任何類型的應用程序。隨著腳本語言的流逝，它非常靈活。常見任務具有自己的命令，對於您需要進行的任何自定義，您都可以使用標準的shell命令（Linux上的Bash或Windows上的PowerShell）。列表3.1顯示了完整的Dockerfile，以包裝Web-Ping應用程序。

列出3.1網絡連接碼列

```
來自diamol /node : 2e env target = “ blog.sixeyed.com ” env方法= “ Head ” env In vesss Interval = “ 3000 ” WorkDir /Web-Pi ng copy copy app.js 。 cmd [ “ node ” ， / web-ping/app.js “ ]
```

即使這是您見過的第一個Dockerfile，您也可以很好地猜測這裡發生的事情。 Dockerfile的說明來自Env，WorkDir，Copy和CMD；他們是首都，但這是一個慣例，而不是必需的。這是每個指令的分解：

- 從 - 每個圖像都必須從另一個圖像開始。在這種情況下，Web-Ping圖像將使用Diamon/Node：2E圖像作為起點。該圖像已安裝了Node.js，這是Web-Ping應用程序需要運行的所有內容。 env-環境變量的設置值。語法是[鍵] = “[value]”，這裡有三個EVENT指令，設置了三個不同的環境變量。
- WorkDir - 在容器圖像文件系統中創建一個目錄，並將其設置為當前工作目錄。前斜線語法適用於Linux和Windows容器，因此這將在Windows上的Linux和C:\ Web-Ping上創建 /Web ping。
- 複製 - 從本地文件系統中的文件或目錄到容器映像中。語法是[源路徑] [目標路徑] - 在這種情況下，我將app.js從本地計算機複製到圖像中的工作目錄。
- CMD - 指定Docker從圖像啟動容器時運行的命令。這是運行node.js，在app.js中啟動應用程序代碼。
-
-

就是這樣。這些說明幾乎是您在Docker中打包自己的應用程序所需的全部，並且在這五行中，已經有一些好的做法。

現在嘗試

您無需複制和粘貼此Dockerfile；這是本書的源代碼中的全部，您在第1章中從Github克隆或下載。導航到下載的位置，並檢查是否有所有文件可以構建此圖像：

```
CD CH03/練習/Web-Ping LS
```

您應該看到您有兩個文件：

- Dockerfile（無文件擴展名），該內容與列表3.1 App.js的內容相同，該app.js具有Web-Ping應用程序的node.js代碼

您可以在圖3.5中看到這些。

```
PS>cd ch03/exercises/web-ping  
PS>ls  
Dockerfile app.js
```

This is the directory with the input files to build the image
It contains the Dockerfile and the application content

圖3.5您需要構建Docker圖像的內容

您不需要對Node.js或JavaScript的任何了解來包裝此應用並在Docker中運行。如果您確實查看app.js中的代碼，則會發現它是非常基本的，並且它使用標準node.js庫來進行HTTP調用並從環境變量中獲取配置值。

在此目錄中，您擁有為Web-Ping應用程序構建自己的圖像所需的一切。

3.3構建自己的容器圖像

Docker需要了解幾件事，然後才能從Dockerfile構建圖像：它需要圖像的名稱，並且需要了解將其包裝到圖像中的所有文件的位置。您已經在正確的目錄中有一個終端打開，因此您可以開始使用。

現在嘗試

通過運行Docker Image構建，將此Dockerfile變成Docker映像：

Docker Image Build-標籤Web-Ping。

-TAG參數是圖像的名稱，最終參數是Dockerfile和相關文件所在的目錄。Docker稱此目錄為*context*，該期間的意思是“使用當前目錄”。您會看到構建命令的輸出，執行Dockerfile中的所有指令。我的構建如圖3.6所示。

```
PS>docker image build --tag web-ping .
[+] Building 0.7s (9/9) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 194B
=> [internal] load metadata for docker.io/diamol/node:2e
=> [auth] diamol/node:pull token for registry-1.docker.io
=> [internal] load .dockerrcignore
=> => transferring context: 2B
=> [1/3] FROM docker.io/diamol/node:2e@sha256:6e2675af9ecb7662da4648d36e27969cbbd23
=> [internal] load build context
=> => transferring context: 288
=> CACHED [2/3] WORKDIR /web-ping
=> CACHED [3/3] COPY app.js .
=> exporting to image
=> => exporting layers
=> => writing image sha256:b171de8b94a2c2776b27ab2cf9221d827e116ea9365e0e9f0d2af04f
=> => naming to docker.io/library/web-ping
```

圖3.6從構建網絡木板碼頭圖像輸出

如果您從構建命令中遇到任何錯誤，則首先需要驗證Docker引擎的啟動。您需要在Windows或Mac上運行的Docker Desktop應用程序（在任務欄中查看鯨魚圖標）。然後驗證您是否處於正確的目錄。您應該在dockerrfile和app.js文件的CH03-WEB-PING目錄中。最後，驗證您是否正確輸入了構建命令 - 需要命令末尾的時期，以告訴Docker構建上下文是當前目錄。

如果您在構建過程中獲得有關文件權限的警告，那是因為您在Windows上使用Docker命令行來構建Linux容器，這要歸功於Docker Desktop的Linux容器模式。Windows不會像Linux那樣記錄文件權限，因此警告（您可以忽略）告訴您，從Windows機器中複制的所有文件均在Linux Docker Image中設置為完整的讀寫和寫入權限。

當您在輸出中看到“成功構建”和“成功標記”消息時，您的圖像就會構建。它在您的圖像緩存中存儲在本地，您可以使用Docker命令看到它以列出圖像。

現在嘗試

列出標籤名稱以“W”開頭的所有圖像：

Docker Image LS'W*

您會看到列出的Web-Ping映像：

```
>Docker Image LS'W*
存儲庫標籤圖像ID創建的尺寸Web-Ping最新B171DE8B94A2 7分鐘前15
4MB
```

您可以以與從Docker Hub下載的圖像完全相同的方式使用此圖像。應用程序的內容相同，並且可以使用環境變量應用配置設置。

現在嘗試

每五秒鐘將容器從您自己的圖像運行到Ping Docker的網站：

```
docker容器運行-e target = docker.com -e Interval = 5000 web -ping
```

您的輸出將像圖3.7中的我一樣，第一個日誌線證實了目標Web URL是docker.com，而ping間隔為5000毫秒。

```
PS>docker container run -e TARGET=docker.com -e INTERVAL=5000 web-ping
```

```
** web-ping ** Pinging: docker.com; method: HEAD; 5000ms intervals
```

```
Making request number: 1; at 1/23394013586
```

```
Got response status: 403 at 1723394013696; duration: 110ms
```

```
Making request number: 2; at 1723394018644
```

```
Got response status: 403 at 1723394018713; duration: 69ms
```

```
Making request number: 3; at 1723394023657
```

```
Got response status: 403 at 1723394023736; duration: 79ms
```

Environment variables specify the target
URL and the interval between pings

Container logs show the app is reading
configuration from the environment

圖3.7從您自己的圖像運行Web-Ping容器

該容器在前景中運行，因此您需要使用CTRL-C停止它。這將結束應用程序，並且容器將進入退出狀態。

您已經打包了一個簡單的應用程序以在Docker中運行，並且對於更複雜的應用程序，該過程完全相同。您可以使用包裝應用程序的所有步驟編寫Dockerfile，收集需要進入Docker映像的資源，並確定您如何希望圖像用戶配置應用程序的行為。

3.4 了解Docker的圖像和圖像層

在本書工作時，您將構建更多圖像。在本章中，我們將堅持使用此簡單的內容，並使用它更好地了解圖像的工作方式以及圖像和容器之間的關係。

Docker映像包含您包裝的所有文件，這些文件已成為容器的文件系統，並且還包含有關圖像本身的大量元數據。這包括有關圖像的構建方式的簡短歷史。您可以使用它來查看圖像的每一層和構建圖層的命令。

現在嘗試

檢查您的Web-Ping圖像的歷史記錄：

Docker圖像歷史網絡鑲嵌

您會看到每個圖像層的輸出線；這些是我的圖像中的前幾條（縮寫）線：

```
>Docker圖像歷史網絡鑲嵌
由F6CE8A61BA85創建的圖像9分鐘前CMD [ "node" "/web-ping/app.js" ]
<<缺少> 9分鐘前複製app.js。# buildkit <缺少> 9分鐘前工作dir /web-ping
```

命令創建的是Dockerfile指令 - 有一對一的關係，因此Dockerfile中的每一行都會創建一個圖像層。我們將在這裡介紹更多的理論，因為理解圖像層是您最有效利用Docker的關鍵。

Docker圖像是圖像層的邏輯集合。層是物理存儲在Docker引擎高速緩存中的文件。這就是為什麼這很重要的原因：圖像層可以在不同的圖像和不同的容器之間共享。如果您有很多運行node.js應用程序的容器，則它們都將共享包含node.js運行時的相同圖像層。圖3.8顯示瞭如何工作。

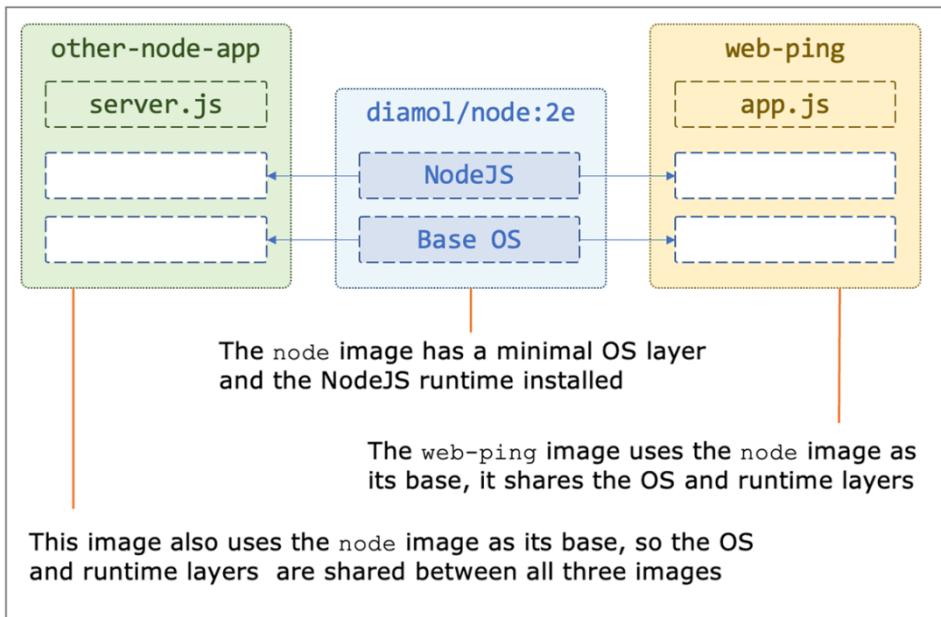


圖3.8圖像層如何邏輯構建到Docker圖像中

DIAMOL/節點：2E圖像具有纖細的操作系統層，然後是Node.js運行時。Linux圖像佔用約150 MB的磁盤（Windows容器的基本OS層更大，因此該圖像的Windows版本使用接近450 MB）。您的網絡圖像基於Diamol/Node：2E，因此從該圖像中的所有層開始 - 這就是Dockerfile中的指導給您的。您在基本圖像頂部包裝的App.js文件只有幾千字節的大小，那麼Web-ping圖像總共有多大？

現在嘗試

您可以使用Docker Image LS列出圖像，該圖像也顯示了圖像的大小。如果您不在命令中包含過濾器，則會看到所有圖像：

```
Docker Image Pull Diamol/節點：2E Docker Image LS
```

您的輸出將像我的圖3.9中一樣。

PS>docker image ls		TAG	IMAGE ID	CREATED	SIZE
REPOSITORY		latest	b171de8b94a2	16 minutes ago	★ 154MB
★web-ping		2e	144577951cdf	16 minutes ago	★ 154MB
★diamol/ch03-web-ping		2e	7ec33cf8bbf6	2 hours ago	72.9MB
diamol/ch02-hello-diamol-web		2e	40af35ca7334	2 hours ago	8.83MB
diamol/ch02-hello-diamol		2e	bf5017a855a2	3 hours ago	14.5MB
diamol/base		2e	3fd5b26294c4	4 days ago	★ 154MB
★diamol/node					

These three images all share the same NodeJS base layers

It looks as though they each use 154MB of disk space, but this is the **logical** size of the image, without accounting for shared layers

圖3.9列出圖像以查看其大小

看起來所有節點圖像都佔用了相同數量的空間 - 在Linux上每個圖像都有154 MB。其中有三個：Diamol/Node：2E，您從Diamol/CH03-Web-Ping中從Docker Hub刪除的原始示例應用程序：2E，以及您在Web-Ping中構建的版本。他們應該共享基本圖像層，但是Docker Image LS的輸出表明它們的大小為154 MB，因此總共 $154 * 3 = 462$ MB。

但不完全是。您看到的尺寸列是圖像的邏輯大小，即如果系統上沒有其他圖像，則圖像將使用多少磁盤空間。如果您確實有共享圖層的其他圖像，則磁盤太空碼頭的使用要小得多。您從圖像列表中看不到它，但是有一些Docker System命令可以告訴您更多。

現在嘗試

我的圖像列表總共顯示了558.23 MB的圖像，但這是總邏輯大小。系統DF命令準確顯示了Disk Space Docker使用的數量：

Docker系統DF

您可以在圖3.10中看到我的圖像緩存實際上正在使用223.6 MB，這意味著在圖像之間共享334 MB的圖像層，在磁盤空間上節省了60%。當您擁有大量應用程序圖像時，您通過重複使用節省的磁盤空間通常要大得多，所有這些圖像都共享運行時相同的基礎層。這些基礎層可能具有Java，.NET Core，PHP - 無論您使用哪種技術堆棧，Docker的行為都是相同的。

PS>docker system df				
TYPE	TOTAL	ACTIVE	SIZE	RECLAIMABLE
Images	6	0	223.6MB	223.6MB (100%)
Containers	0	0	0B	0B
Local Volumes	0	0	0B	0B
Build Cache	6	0	0B	0B

This is the physical disk space used to store all image layers

圖3.10檢查Docker的磁盤空間使用情況

最後的理論。如果圖像層在周圍共享，則無法編輯它們 - 否則，一個圖像的更改將層疊到共享更改層的所有其他圖像。Docker通過使圖像層僅閱讀來強制執行。一旦通過構建圖像創建圖層，該圖層就可以通過其他圖像共享，但不能更改。您可以利用這一點，使您的Docker圖像較小，並且通過優化Dockerfiles來更快地構建。

3.5 優化使用圖像層的Dockerfiles

快取

您的Web-Ping映像中包含應用程序的JavaScript文件。如果您更改該文件並重建圖像，則將獲得一個新的圖像層。Docker假設Docker圖像中的圖層遵循定義的序列，因此，如果您在該序列的中間更改一層，Docker不會在序列中重複使用後來的層。

現在嘗試

更改CH03-WEB-PING目錄中的app.js文件。不必更改代碼；只需在文件末尾添加一條新的空線即可。然後構建Docker映像的新版本：

Docker Image Build -T Web -Ping : V2。

您會看到與圖3.11中我相同的輸出。構建的步驟2使用緩存中的層，步驟3生成了一個新圖層。

```
PS>docker image build -t web-ping:v2 .
[+] Building 0.1s (8/8) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 194B
=> [internal] load metadata for docker.io/diamol/node:2e
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/3] FROM docker.io/diamol/node:2e
=> [internal] load build context
=> => transferring context: 883B
=> CACHED [2/3] WORKDIR /web-ping
=> [3/3] COPY app.js .
=> exporting to image
=> => exporting layers
=> => writing image sha256:8c4c8faa2690caf7c8029c82d112219
=> => naming to docker.io/library/web-ping:v2
```

Step 2 comes from the cache because
the input is unchanged

Step 3 executes because the copied file contents have
changed, so any steps following it would execute too

圖3.11構建圖像，其中可以從緩存中使用圖層

每個Dockerfile指令都會產生圖像層，但是如果指令在構建之間沒有變化，並且進入指令中的內容是相同的，那麼Docker知道它可以在緩存中使用上一層。這可以保存再次執行Dockerfile指令並生成重複層。輸入是相同的，因此輸出將相同，因此Docker可以使用緩存中已經存在的內容。

Docker通過生成哈希來計算輸入是否具有在緩存中的匹配，這就像代表輸入的數字指紋。哈希是由Dockerfile指令和任何要複制的文件的內容製成的。如果現有圖像層中的哈希沒有匹配，則Docker執行指令，並打破緩存。一旦緩存被打破，Docker即使沒有更改，Docker也會執行下面的所有說明。

即使在這個小樣本圖像中也有影響。自上次構建以來，app.js文件已更改，因此需要運行複制指令。此後的任何說明都需要再次執行。

您編寫的任何Dockerfile都應進行優化，以便通過更改的頻率來訂購指令 - 在Dockerfile開始時不太可能更改的說明，並且最有可能在末尾更改的說明。目標是使大多數構建需要僅執行最後一個指令，並將其用於其他所有內容。 當您開始共享圖像時，這可以節省時間，磁盤空間和網絡帶寬。

Web-Ping Dockerfile中只有七個說明，但仍然可以優化。 CMD指令不需要在Dockerfile的末尾； 它可以在指令之後的任何地方，並且仍然具有相同的結果。它不太可能改變，因此您可以將其移到頂部。 一個環境指令可用於設置多個環境變量，因此可以將三個獨立的Env指令組合在一起。 優化的Dockerfile在清單3.2中顯示。

清單3.2 優化的Web-Ping Dockerfile

來自Diamol/節點：2E

```
cmd [ "node" , /web-ping/app.js "]  
  
env target = " blog.sixeyed.com " \ meth  
od = " head " \ Interval = " 3000 "  
  
workdir /web-ping cop  
y app.js °
```

現在嘗試

優化的Dockerfile也在本章的源代碼中。切換到網絡優化的文件夾，然後從新的Dockerfile構建圖像：

```
CD .. / web-ping - 優化  
Docker Image Build - T Web - Ping : V3 °
```

您不會注意到與以前的構建相差太大。 現在有五個步驟而不是七個步驟，但是最終結果是相同的：您可以從該圖像中運行一個容器，並且它的行為就像其他版本一樣。但是現在，如果您更改app.js和Rebuild中的應用程序代碼，則所有步驟都來自緩存，除了最後一個步驟，這正是您想要的，因為這就是您更改的全部。

這就是在本章中構建圖像的全部。您已經看到了Dockerfile語法和需要知道的關鍵說明，並學會瞭如何與Docker CLI的圖像構建和合作。

從本章中可以採取兩個重要的概念，在您構建的每個圖像中都可以為您提供良好的服務：優化dockerfiles，並確保您的圖像是便攜式的，以便在部署到不同環境時使用相同的圖像。這確實意味著您應該注意如何構建Dockerfile指令，並確保應用程序可以從容器中讀取配置值。這意味著您可以快速構建圖像，當您部署到生產時，您使用的是在測試環境中質量批准的完全相同的圖像。

3.6 實驗室

好吧，這是實驗室的時間。這裡的目標是回答這個問題：如何在沒有Dockerfile的情況下製作Docker圖像？Dockerfile在那裡可以自動化應用程序的部署，但是您不能總是自動化所有內容。有時，您需要運行該應用程序並手動完成一些步驟，而這些步驟無法腳本進行腳本。

該實驗室是其中一個簡單的版本。您將從Docker Hub上的圖像開始：DIAMOL/CH03-LAB：2E。該圖像在路徑 /diamol/ch03.txt 上具有文件。您需要更新該文本文件並在最後添加您的名字。然後用更改的文件產生自己的圖像。您不允許使用Dockerfile。

如果您需要，該書的GitHub存儲庫中有一個示例解決方案。您會在這裡找到它：<https://github.com/SIXEYED/DIAMOL/tree/2E/CH03/LAB>。

這裡有一些提示可以讓您前進：

- 請記住，它的標誌使您可以交互跑到容器。放出容器的文件系統仍然存在。您
- 還沒有使用過很多命令。Docker容器 - 幫助您將向您展示兩個可以幫助您解決實驗
- 室的東西。

4包裝應用程序從源代碼 到Docker Images

構建Docker圖像很容易。此前，您了解到，您只需要在Dockerfile中使用一些說明即可打包應用程序以在容器中運行。 您需要知道另一件事才能打包自己的應用程序：您還可以在Dockerfiles中運行命令。

命令在構建過程中執行，並且命令中的任何文件系統更改都保存在圖像層中。這使得有關最靈活的包裝格式的Dockerfiles； 您可以擴展ZIP文件，運行Windows安裝程序，並執行其他操作。現在，您將使用該靈活性從源代碼打包應用程序。

4.1 Who有Docke時需要構建服務器 rfile ?

在筆記本電腦上構建軟件是您為本地開發做的事情，但是當您 在團隊中工作時，會有一個更嚴格的交付過程。有一個像GitHub這樣的共享源控制系統，每個人都在其中推動代碼更改，通常有一個單獨的服務器（或在線服務），當更改被推動時，可以構建軟件。

這個過程存在於早期遇到問題。如果開發人員在推出代碼時忘記了添加文件，則構建服務器將在構建服務器上失敗，並且團隊將被提醒。它使項目保持健康，但成本必須維護構建服務器。大多數編程語言都需要大量的工具來構建項目 - 圖4.1顯示了一些示例。

Tools needed to build software—could include Maven and the JDK for Java projects, NuGet MSBuild, and Visual Studio Build Tools for .NET.

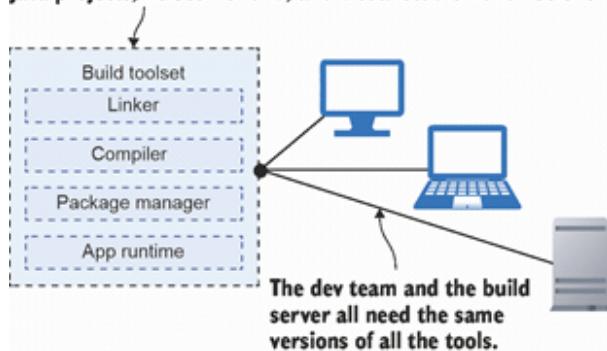


圖4.1每個人都需要相同的工具來構建軟件項目。

這裡有一個很大的維護費用。團隊中的一個新的入門者將花費整個第一天的時間安裝工具。如果開發人員更新其本地工具，以便構建服務器運行不同的版本，則構建可能會失敗。即使您使用的是託管構建服務，也有相同的問題，並且您可能會安裝一套有限的工具。

包裝構建工具集並共享它會更加干淨，這正是您可以與Docker一起做的事情。您可以編寫腳本腳本部署所有工具並將其構建到圖像中的Dockerfile。然後，您可以在應用程序DockerFiles中使用該圖像來編譯源代碼，最終輸出是您的打包應用程序。

讓我們從一個非常簡單的示例開始，因為在此過程中有幾個新事物需要理解。清單4.1顯示了帶有基本工作流程的Dockerfile。

列表4.1多階段的Dockerfile

```
從diamol /base : 2e作為構建階段運行迴聲'bu  
ilding ...'> /build.txt
```

```
來自diamol /base : 2e作為測試階段副本-from = build stage  
/build.txt /build.txt run迴聲
```

```
來自Diamol /base : 2e複製-from = test stage /build.txt /buil  
d.txt cmd [ “ cat ” , “ /build.txt ” ]
```

這稱為多階段Dockerfile，因為該構建有幾個階段。每個階段都以A從指令開始，您可以選擇為階段提供一個名稱為AS參數。清單4.1具有三個階段：建築階段，測試階段和最終未命名階段。儘管有多個階段，但輸出將是具有最後階段內容的單個Docker映像。

每個階段都獨立運行，但是您可以從以前的階段複製文件和目錄。我使用的是`-from`參數的複制指令，該參數告訴Docker從Dockerfile的較早階段複製文件，而不是從主機計算機的文件系統中複製文件。在此示例中，我在構建階段生成一個文件，將其複製到測試階段，然後將文件從測試階段複製到最後一個階段。

這裡有一個新的指令，運行，我用它來執行操作系統命令編寫文件。運行指令在構建過程中在容器內執行命令，該命令中的任何輸出都保存在圖像層中。您可以在運行指令中執行任何內容，但是您要運行的命令需要在您在“從指令中”中使用的Docker映像中存在。在此示例中，我使用Diamol/base : 2e作為基本圖像，並且包含Echo命令，因此我知道我的運行指令將起作用。

圖4.2顯示了當我們構建此Dockerfile時會發生什麼 - Docker將順序運行階段。

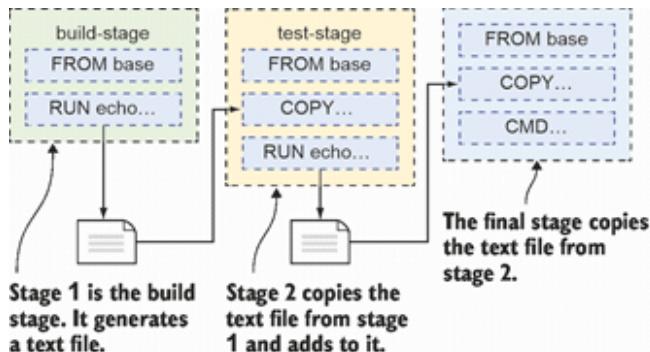


圖4.2執行多階段Dockerfile

重要的是要了解單個階段是孤立的。您可以在每個階段使用不同的基本圖像，並安裝了不同的工具集並運行您喜歡的任何命令。最後階段的輸出僅包含您從較早階段明確複製的內容。如果命令在任何階段失敗，則整個構建失敗。

現在嘗試

打開一個終端會話，向您存儲本書的源代碼的文件夾，並構建此多階段Dockerfile：

CD CH04/練習/多階段Docker Image Build -T 多階段。

您會看到構建按照Dockerfile的順序執行步驟，這使得通過圖4.3中可以看到的階段進行順序構建。

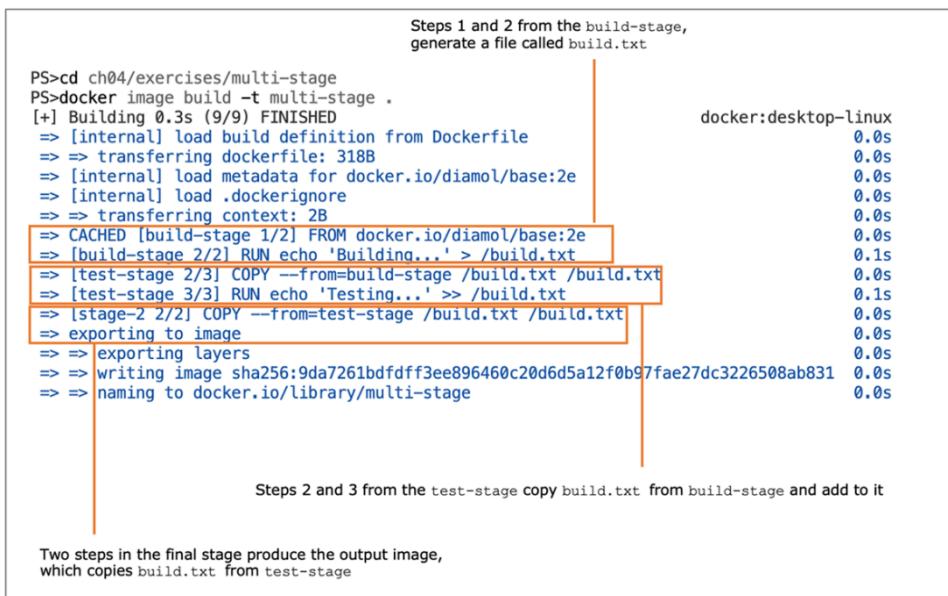


圖4.3 構建多階段的Dockerfile

這是一個簡單的示例，但是對於與單個Dockerfile的任何複雜性構建應用程序的模式相同。 圖4.4顯示了Java應用程序的工作流程。

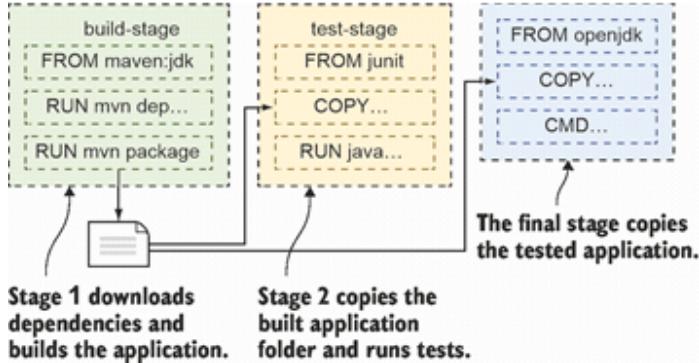


圖4.4 Java應用程序的多階段構建

在構建階段，您使用已安裝應用程序構建工具的基本圖像。您可以從主機計算機中複制源代碼，並運行構建應用程序所需的任何命令。您可以添加一個測試階段來運行單元測試，該測試將基本圖像與已安裝的測試框架一起使用，並從構建階段複製編譯的二進製文件，並運行測試。最後階段從基本圖像開始，僅安裝了應用程序運行時，它從構建階段中複制了在測試階段成功測試的二進製文件。

這種方法使您的應用程序真正可移植。您可以在任何地方的容器中運行該應用程序，但是您也可以在任何地方構建該應用程序 - Docker是唯一的先決條件。您的構建服務器只需要安裝Docker；新團隊成員在幾分鐘內設置了設置，並且構建工具都集中在Docker Images中，因此沒有機會離開同步。

所有主要的應用程序框架都已經在Docker Hub上具有公共圖像，並安裝了構建工具，並且在應用程序運行時有單獨的圖像。您可以直接使用這些圖像，也可以將它們包裹在自己的圖像中。您將獲得所有最新更新，並使用項目團隊維護的圖像。

4.2 應用程序演練：Java源代碼

我們現在將使用一個簡單的Java Spring Boot應用程序來進行現實生活中的示例，該應用程序將使用Docker構建和運行。您無需成為Java開發人員，也不需要在計算機上安裝任何Java工具來使用此應用；您需要的一切都將出現在Docker圖像中。如果您不使用Java，則仍然應該閱讀本節 - 它描述了一種適合其他編譯語言的模式，例如.NET和Rust。

源代碼位於本書的存儲庫中，在文件夾路徑CH04/練習/當天圖像中。該應用程序使用Java : Maven的相當標準的工具集，用於定義構建過程和獲取依賴關係，而OpenJDK則是可自由分佈的Java運行時和開發人員套件。Maven使用XML格式來描述構建，而Maven命令行稱為MVN。這應該是足夠的信息，可以理解清單4.2中的應用程序dockerfile。

Listing 4.2 Dockerfile for building a Java app with Maven

```
FROM diamol/maven:2e AS builder

WORKDIR /usr/src/iotd
COPY pom.xml .
RUN mvn -B dependency:go-offline

COPY . .
RUN mvn package

# app
FROM diamol/openjdk:2e

WORKDIR /app
COPY --from=builder /usr/src/iotd/target/iotd-service-0.1.0.jar .

EXPOSE 80
ENTRYPOINT ["java", "-jar", "/app/iotd-service-0.1.0.jar"]
```

這裡幾乎所有的Dockerfile說明都是您之前見過的，並且從您構建的示例中熟悉這些模式。這是一個多階段的Dockerfile，您可以說，因為指令有多個，並且將步驟得以從Docker的圖像層緩存中獲得最大收益。

第一階段稱為建築商。這是建造者階段發生的情況：

- 它使用Diamol/Maven：2E圖像作為基礎。該圖像安裝了OpenJDK Java開發套件，以及Maven Build Tool。構建器階段首先在圖像中創建一個工作目錄，然後在pom.xml文件中複制，這是Java構建的Maven定義。第一個運行語句執行maven命令，獲取所有應用程序依賴項。這是一個昂貴的操作，因此它具有自己的步驟來利用Docker層緩存。如果有新的依賴項，則XML文件將更改，步驟將運行。如果依賴項沒有更改，則使用層緩存。接下來，將其餘的源代碼複製在復制中。。意思是“從Docker構建跑入圖像中的工作目錄的位置複製所有文件和目錄”。構建器的最後一步是運行MVN軟件包，該軟件包編譯和包裝應用程序。輸入是一組Java源代碼文件，輸出是一個稱為JAR文件的Java應用程序包。
-
-

當此階段完成後，編譯的應用程序將存在於構建器階段文件系統中。如果Maven構建存在任何問題 - 如果網絡是離線和獲取依賴性失敗的，或者源中的編碼錯誤，則運行指令將失敗並且整個構建失敗。

如果建造者階段成功完成，Docker繼續執行最後階段，該階段產生了應用程序圖像：

- 它始於Diamol/OpenJDK : 2E，它用Java運行時打包，但沒有Maven構建工具。此階段創建一個工作目錄，並在構建器階段中復制編譯的JAR文件。Maven將應用程序及其所有Java依賴項包裝在此單個JAR文件中，因此這是構建器所需的全部。該應用程序是在端口80上聽取的Web服務器，因此該端口已在公開指令中明確列出，該端口告訴Docker可以發布此端口。入口點指令是CMD指令的替代方法 - 告訴Docker從圖像啟動容器時該怎麼辦 - 在這種情況下，請運行Java，並帶有通往應用程序JAR的路徑。
-
-

現在嘗試

瀏覽到Java應用程序源代碼並構建圖像：

```
CD CH04/練習/當日的Docker Image build -t-t-trimage  
e-Primage。
```

此構建有很多輸出，因為您會看到Maven，獲取依賴關係並貫穿Java build的所有日誌。圖4.5顯示了我構建的縮寫部分。

```

PS>cd ch04/exercises/image-of-the-day
PS>docker image build -t image-of-the-day .
[+] Building 12.3s (15/15) FINISHED
  => [internal] load build definition from Dockerfile
  => transferring dockerfile: 347B
  => [internal] load metadata for docker.io/diamol/openjdk:2e
  => [internal] load metadata for docker.io/diamol/maven:2e
  => [internal] load .dockerrcignore
  => => transferring context: 2B
  => CACHED [builder 1/6] FROM docker.io/diamol/maven:2e
  => CACHED [stage-1 1/3] FROM docker.io/diamol/openjdk:2e
  => [internal] load build context
  => => transferring context: 9.41kB
  => [builder 2/6] WORKDIR /usr/src/iotd
  => [stage-1 2/3] WORKDIR /app
  => [builder 3/6] COPY pom.xml .
  => [builder 4/6] RUN mvn -B dependency:go-offline
  => [builder 5/6] COPY .
  => [builder 6/6] RUN mvn package
  => [stage-1 3/3] COPY --from=builder /usr/src/iotd/target/iotd-service-0.1.0.jar .
  => exporting to image
  => => writing image sha256:30549c64ee78e78886fce2328fef23cd86afa3c9d4bf4a761c78e765
  => naming to docker.io/library/image-of-the-day

```

The last part of the builder stage uses Maven to build the app and package it into a Java application archive (JAR)

The final application stage copies the generated JAR file from the builder

圖4.5在Docker中運行Maven構建的輸出

那麼，您剛剛建造了什麼？這是一個簡單的REST API，它包裹了NASA的日常服務天文學圖片（<https://api.nasa.gov>）。Java應用程序從NASA獲取了今天的圖片的詳細信息，並將其存儲在本地緩存中，因此您可以反復呼叫此應用程序，而無需重複打入NASA的服務。

Java API只是您在本章中要運行的完整應用程序的一部分 - 它實際上將使用多個容器，它們需要相互通信。容器在虛擬網絡上互相訪問，使用Docker創建容器時分配的虛擬IP地址。您可以從命令行創建和管理虛擬Docker網絡。

現在嘗試

創建一個用於相互通信的容器的Docker網絡：

Docker網絡創建NAT

如果您從該命令中看到錯誤，那是因為您的設置已經具有名為NAT的Docker網絡，並且您可以忽略該消息。現在，當您運行容器時，您可以使用 - 網絡標誌將它們明確連接到該Docker網絡，並且該網絡上的任何容器都可以使用容器名稱彼此連接。

現在嘗試

從圖像中運行一個容器，將端口80發佈到主機計算機並連接到NAT網絡：

```
Docker續    ainer run -name iotd -d -p 800:80 - 網絡nat nat圖像      - 今天
```

現在，您可以瀏覽`http://localhost:800/image`，您將看到一些有關NASA當天圖像的JSON詳細信息。在我運行容器的那天，圖像是黑洞 - 圖4.6顯示了我的API的細節。

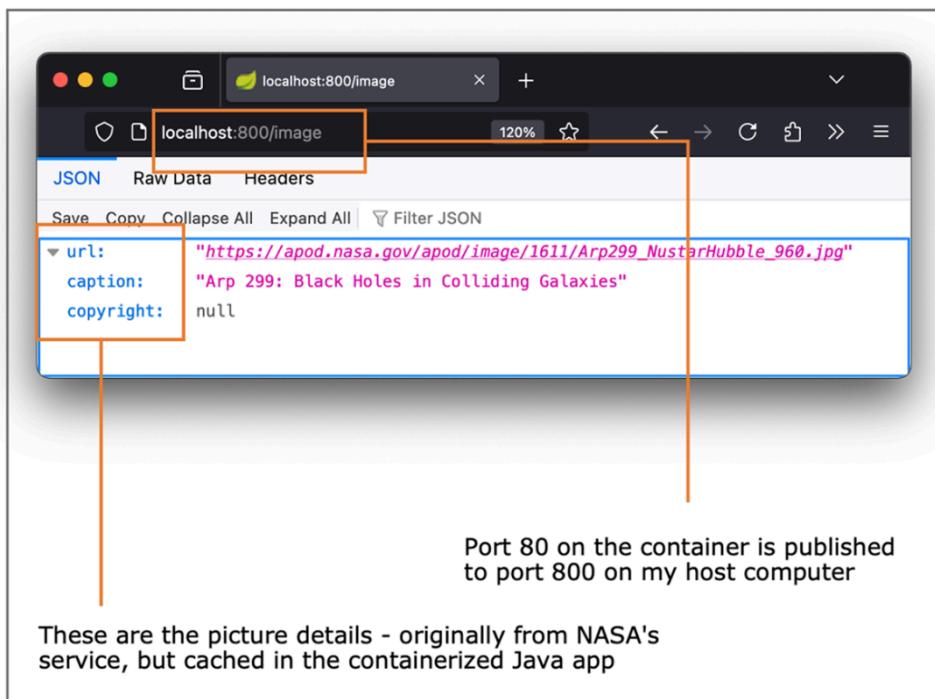


圖4.6在我的應用程序容器中，來自NASA的緩存細節

該容器中的實際應用並不重要（但請不要將其刪除 - 我們將在本章稍後使用它）。重要的是，您可以通過使用Dockerfile的源代碼副本在安裝Docker安裝的任何機器上構建此機器。您不需要安裝任何構建工具，也不需要特定版本的Java，您只是克隆代碼回購，而您可以使用幾個Docker命令來運行該應用程序。

這裡要清除的另一件事是：構建工具不是最終應用程序圖像的一部分。您可以從新的Docker映像中運行交互式容器，並且您會發現其中沒有MVN命令。只有Dockerfile中最後階段的內容才能進入應用程序圖像。您需要在最後階段中明確複製您想要的任何內容。

4.3 App演練：Node.js源代碼

這次，我們將瀏覽另一個多階段Dockerfile。組織越來越多地使用多種技術堆棧，因此了解Docker中不同構建的外觀是一件好事。node.js是一個有用的選擇，因為它的流行性，並且因為它是不同類型的構建的示例 - 這種模式還可以與Python，PHP和Ruby等其他腳本語言一起使用。此應用程序的源代碼位於文件夾路徑C H04/ercory/ Access-Log上。

彙編了Java應用程序，因此將源代碼複製到構建階段，並生成JAR文件。JAR文件是編譯的應用程序，並且將其複製到最終的應用程序映像中，但源代碼不是。與.net核心相同，其中編譯的文物為dlls（動態鏈接庫）。Node.js不同 - 使用JavaScript，這是一種解釋的語言，因此沒有彙編步驟。dockerized node.js應用需要node.js運行時和應用程序圖像中的源代碼。

但是，仍然需要多階段的Dockerfile：它優化了依賴性加載。Node.js使用一個名為NPM（Node Package Manager）的工具來管理依賴關係。清單4.3顯示了本章Node.js應用程序的完整Dockerfile。

清單4.3使用NPM構建Node.js應用的Dockerfile

```
來自Diamol/Node : 2e作為Builder WorkDir/SRC複製SRC/PACKAND.JSO  
N。從DIAMOL/NODE運行NPM安裝 # App : 2E公開80 cmd [ “ node ”  
，“ server.js ” ] workdir/app copy -from = builder/src/src/node_modules/app  
/nonode_modules/nonode_modules/copy src/copy src/。
```

此處的目標與Java應用程序相同 - 僅安裝了Docker即可打包和運行該應用程序，而無需安裝任何其他工具。兩個階段的基本圖像是Diamol/Node，它已安裝了Node.js運行時和NPM。dockerfile中的構建器階段在package.json文件中複制，其中描述了所有應用程序的依賴性。然後，它運行NPM安裝以下載依賴項。沒有彙編，因此需要做的一切。

此應用程序是另一個REST API。在最終應用階段，這些步驟將公開HTTP端口並將節點命令行指定為啟動命令。最後一件事是創建一個工作目錄並在應用程序工件中複制。下載的依賴項是從構建器階段複製的，並從主機計算機複製源代碼。SRC文件夾包含JavaScript文件，包括server.js，這是由node.js process啟動的輸入點。

我們在這裡有不同的技術堆棧，具有包裝應用程序的不同模式。Node.js應用程序的基本圖像，工具和命令都與Java應用程序不同，但是這些差異在DockerFile中捕獲。構建和運行應用程序的過程完全相同。

現在嘗試

瀏覽到Node.js應用程序源代碼並構建圖像：

```
CD CH04/練習/訪問記錄  
Docker Image Build -T訪問-LOG。
```

您會看到NPM的大量輸出（這也可能顯示出一些錯誤和警告消息，但您可以忽略這些消息）。圖4.7顯示了我構建的部分輸出。下載的軟件包將保存在Docker Image Layer Cache中，因此，如果您在應用程序上使用並進行代碼更改，那麼您運行的下一個構建將超級快。

```

PS>cd ch04/exercises/access-log
PS>docker image build -t access-log .
[+] Building 6.0s (12/12) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 271B
=> [internal] load metadata for docker.io/diamol/node:2e
=> [internal] load .dockerrcignore
=> => transferring context: 2B
=> [internal] load build context
=> => transferring context: 1.57kB
=> CACHED [builder 1/4] FROM docker.io/diamol/node:2e
=> [builder 2/4] WORKDIR /src
=> [stage-1 2/4] WORKDIR /app
=> [builder 3/4] COPY src/package.json .
=> [builder 4/4] RUN npm install
=> [stage-1 3/4] COPY --from=builder /src/node_modules/ /app/node_modules/
=> [stage-1 4/4] COPY src/
=> exporting to image
=> => exporting layers
=> => writing image sha256:37799a509e595bc95bd162cef71105452abbebe34d002ec744dee3f4
=> => naming to docker.io/library/access-log

```

The application stage copies downloaded dependencies from the builder stage

And the JavaScript files are copied from the `src` folder on the host computer

The builder stage fuses NPM to fetch the application's dependencies

圖4.7為節點構建多階段的dockerfile.js應用程序

您剛剛構建的Node.js應用程序根本不有趣，但是您仍然應該運行它以驗證其已正確打包。這是其他服務可以打電話以編寫日誌的REST API。有一個用於記錄新日誌的HTTP帖子端點，並且一個GET端點顯示已記錄了多少個日誌。

現在嘗試

從日誌API映像中運行一個容器，將端口80發佈到主機上的端口801，然後將其連接到同一NAT網絡：

```
Docker Container Run -name AccessLog -D -P 801 : 80 - 網絡nat access -log
```

現在，瀏覽到`http://localhost:801/stats`，您將看到該服務已記錄了多少個日誌。圖4.8顯示我到目前為止的日誌為零。FireFox很好地格式化了API響應，但是您可能會在其他瀏覽器中看到RAW JSON。

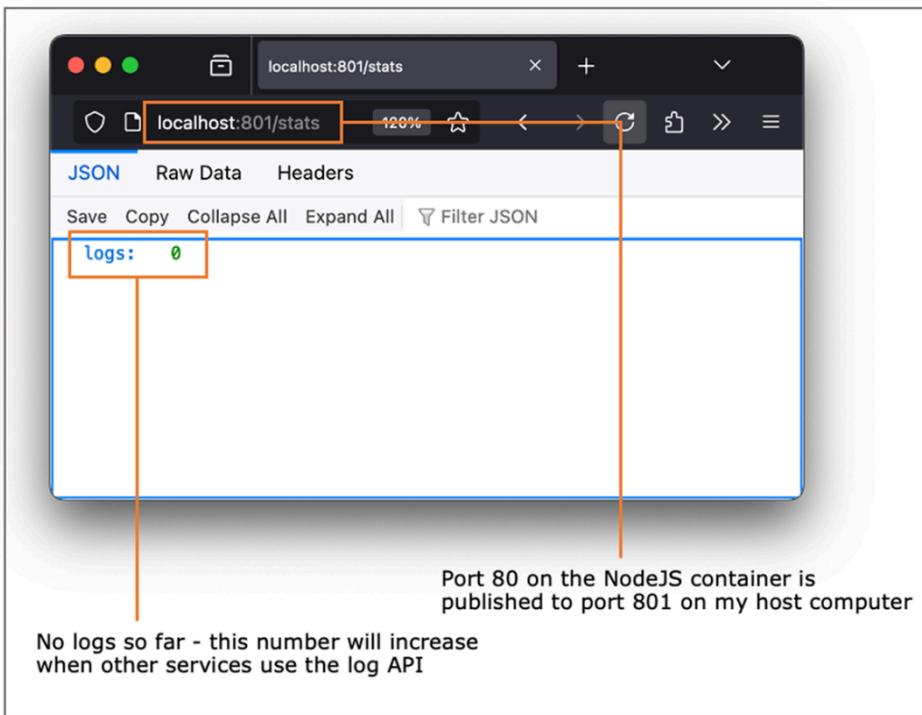


圖4.8在容器中運行node.js api

LOG API在Node.js版本22中運行，但是就像Java示例一樣，您不需要任何版本的node.js或安裝的任何其他工具來構建和運行此應用程序。此Dockerfile中的工作流下載依賴項，然後將腳本文件複製到最終圖像中。您可以使用Python完全相同的方法，使用PIP進行依賴項，也可以使用寶石使用Ruby。

4.4應用程序演練：GO源代碼

我們有一個多階段Dockerfile的最後一個示例 - 用於GO編寫的Web應用程序。Go是一種現代的跨平台語言，可編譯為本地二進製文件。這意味著您可以編譯應用程序以在任何平台（Windows，Linux，Intel或Arm）上運行，並且編譯的輸出是完整的應用程序。您不需要像Java，.net，node.js或Python一樣安裝單獨的運行時，這使得非常小的Docker圖像。

其他幾種語言也編譯為本地二進製文件（lust and Swift很受歡迎），但GO具有最廣泛的平台支持，並且它也是一種非常受歡迎的雲本地應用程序語言（Docker本身是用GO編寫的）。在Docker中構建GO應用程序意味著使用類似於您用於Java應用程序的多階段Dockerfile方法，但存在一些重要差異。清單4.4顯示了完整的Dockerfile。

Listing 4.4 Dockerfile for building a Go application from source

```
FROM diamol/golang:2e AS builder

COPY go.mod main.go /go/
RUN go build -o /server

# app
FROM diamol/base:2e

ENV IMAGE_API_URL="http://iotd/image" \
    ACCESS_API_URL="http://accesslog/access-log"
CMD ["/web/server"]

WORKDIR web
COPY index.html .
COPY --from=builder /server .
RUN chmod +x server
```

將編譯到本地二進製文件中，因此Dockerfile中的每個階段都使用不同的基本圖像。構建器階段使用Diamol/Golang：2E，它已安裝了所有GO工具。 GO應用程序通常不會獲取依賴項，因此此階段直接構建應用程序（這只是一個代碼文件main.go和Description File Go.Mod）。最終的應用階段使用最小圖像，該圖像只有一小層操作系統工具，稱為Diamol/base：2e。

DockerFile作為環境變量捕獲了一些配置設置，並將啟動命令指定為編譯的二進制。 應用程序階段通過在HTML文件中複制應用程序從主機服務和Web服務器二進制中的二進製文件結束。二進製文件需要在Linux中明確標記為可執行文件，這是最終CHMOD命令所做的（這對Windows沒有影響）。

現在嘗試

瀏覽到GO應用程序源代碼並構建圖像：

```
CD CH04/練習/圖像 - 美容  
Docker Image Build -T圖像 - 套裝。
```

這次，不會有很多編譯器的輸出，因為GO很安靜，並且僅在發生故障時寫入日誌。您可以在圖4.9中看到我的輸出 - 指出階段並行運行，因此在構建器階段完成之前，將復制HTML文件。

```
PS>cd ch04/exercises/image-gallery
PS>docker image build -t image-gallery .
[+] Building 3.7s (14/14) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 355B
=> [internal] load metadata for docker.io/diamol/base:2e
=> [internal] load metadata for docker.io/diamol/golang:2e
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [builder 1/3] FROM docker.io/diamol/golang:2e
=> [internal] load build context
=> => transferring context: 1.74kB
=> CACHED [stage-1 1/5] FROM docker.io/diamol/base:2e
=> [stage-1 1/5] WORKDIR /web
=> [builder 2/3] COPY go.mod main.go /go/
=> [stage-1 3/5] COPY index.html .
=> [builder 3/3] RUN go build -o /server
=> [stage-1 4/5] COPY --from=builder /server .
=> [stage-1 5/5] RUN chmod +x server
=> exporting to image
=> => exporting layers
=> => writing image sha256:713876e35c89470e9acea7d2749365404baed2f24e23b203e6362bab2
=> => naming to docker.io/library/image-gallery
```

HTML assets are copied from the host computer into the final image

The app is compiled in the builder stage. Go doesn't write output logs when the build is successful

And the web server binary is copied from the builder stage

圖4.9在多階段的Dockerfile中構建GO應用程序

此GO應用程序確實可以做一些有用的事情，但是在運行它之前，值得看看出來的圖像的大小。

現在嘗試

將GO應用程序圖像大小與GO Toolset圖像進行比較：

Docker 圖像ls -f參考=diamol/golang:2e -f reference=image-

許多docker命令允許您過濾輸出。此命令列出了所有圖像，並過濾輸出以僅包含帶有Diamol/Golang或Image-Gallery的引用的圖像。參考實際上只是圖像名稱。當您運行此操作時，您會發現為您的Dockerfile階段選擇正確的基本圖像有多重要：

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
image-gallery	latest	713876e35c89	5 minutes ago	35.1MB
diamol/golang	latest	400e76525fc9	5 days ago	230MB

在Linux上，安裝所有GO工具的圖像以230 MB為單位； 實際的GO應用程序圖像僅為35 MB。請記住：這是虛擬圖像大小，因此可以在不同圖像之間共享許多這些層。重要的節省不是磁盤空間，而是最終圖像中*isn't*的所有軟件。該應用程序在運行時不需要任何GO工具。 通過為應用程序使用最小的基礎圖像，我們節省了將近200 MB的軟件，這是潛在攻擊的表面積的巨大減少。

現在您可以運行該應用程序。這在本章中將您的工作聯繫在一起，因為GO應用程序實際上使用了您構建的其他應用程序中的API。您應該確保將這些容器運行，並在較早的嘗試中使用正確的名稱進行練習。如果運行Docker容器LS，則應從本章中看到兩個容器 - 名為AcscessLog的Node.js容器和稱為IOTD的Java容器。運行GO容器時，它將使用其他容器中的API。

現在嘗試

運行GO應用程序映像，發布主機端口並連接到NAT網絡：

```
Docker容器運行-D -P 802 : 80 -NETWORK NAT IMAGE -GALLERY
```

You can browse to <http://localhost:802> and you'll see NASA's Astronomy Picture of the Day. Figure 4.10 shows the image when I ran my containers.

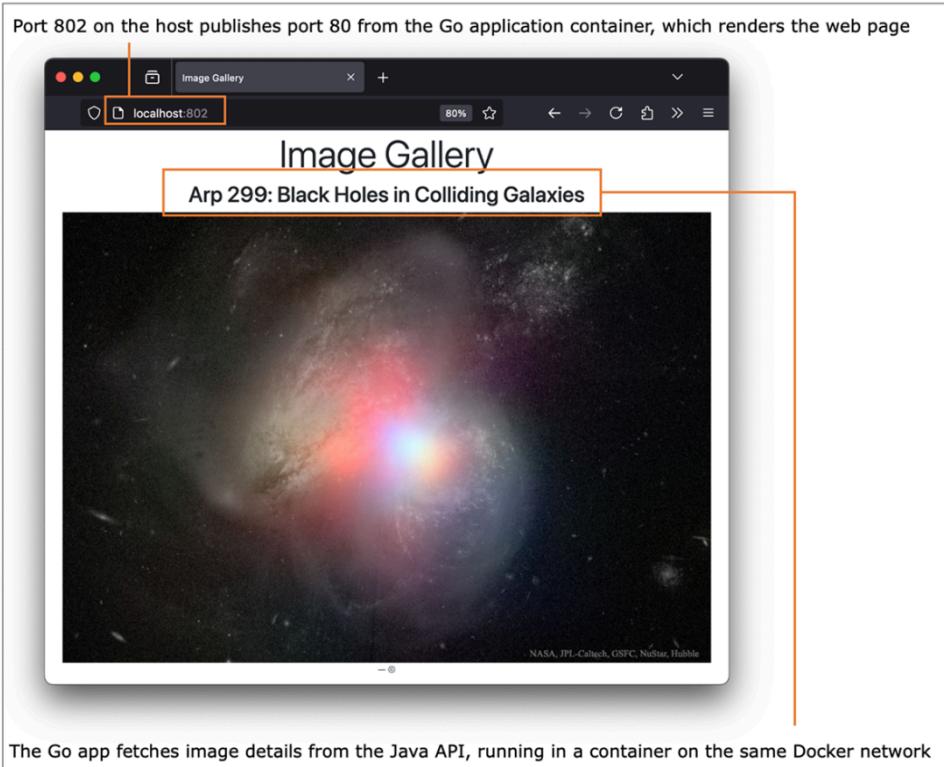


圖4.10 GO Web應用程序，顯示從Java API獲取的數據

目前，您正在三個容器上運行分佈式應用程序。GO Web應用程序調用Java API以獲取要顯示的映像的詳細信息，然後調用Node.js API來記錄已訪問該站點的訪問。您無需為這些語言中的任何一種工具安裝任何工具即可構建和運行所有應用程序；您只需要源代碼和Docker。

多階段Dockerfiles使您的項目完全便攜。您可以立即使用Jenkins來構建您的應用程序，但是您可以嘗試GitHub操作或Azure DevOps而無需編寫任何新的管道代碼 - 它們都支持Docker，因此您的管道只是Docker Image構建。

4.5了解多階段Dockerfiles

我們在本章中介紹了很多基礎，我將以一些關鍵點結尾，以便您清楚多階段的Dockerfiles的工作方式，以及為什麼在容器中構建應用程序的應用程序非常有用。

第一點是關於標準化。我知道，當您運行本章的練習時，您的構建將成功，並且您的應用程序將起作用，因為您使用的是我使用的完全相同的工具集。無論您擁有哪種操作系統或機器上安裝了什麼操作系統，所有的構建都在Docker容器中運行，並且容器映像具有所有正確的工具版本。在您的現實生活項目中，您會發現，對於新開發人員而言，這極大地簡化了入門，消除了構建服務器的維護負擔，並消除了用戶具有不同版本的工具的突破潛力。

第二點是性能。多階段構建中的每個階段都有自己的緩存。Docker在圖像層緩存中為每種說明尋找匹配；如果找不到一個，則緩存將被打破，其餘指令將執行，但僅在該階段。下一個階段再次從緩存開始。您將花費時間仔細構建Dockerfiles，完成優化後，您會發現90%的構建步驟使用緩存。

最後一點是，多階段的Dockerfiles讓您可以微調構建，以便最終的應用程序圖像盡可能瘦。這不僅是針對編譯器的，您需要在較早的階段隔離任何工具，因此該工具本身在最終圖像中不存在。一個很好的例子是捲曲 - 您可以使用Internet下載內容的流行命令行工具。您可能需要下載應用程序需求的文件，但是您可以在Dockerfile的早期階段執行此操作，以便在應用程序映像中安裝curl本身。這會使圖像大小降低，這意味著更快的啟動時間，但這也意味著您在應用程序圖像中使用的軟件較少，這意味著對攻擊者的潛在利用較少。

4.6 實驗室

實驗室時間！您將付諸實踐，了解有關多階段構建和優化Dockerfiles的知識。在本書的源代碼中，您可以在CH04/LAB上找到一個文件夾，這是您的起點。這是一個簡單的GO Web服務器應用程序，它已經具有Dockerfile，因此您可以在Docker中構建並運行它。但是Dockerfile迫切需要優化，這就是您的工作。

這個實驗室有具體的目標：

- 首先使用現有Dockerfile構建圖像，然後優化Dockerfile以生成新的圖像。當前圖像在Linux上的300 MB超過300 MB，Windows上的圖像為5.2 GB。您的優化圖像應在Linux上的25 MB或Windows上的500 MB低於25 MB。如果將HTML內容與當前Dockerfile更改，則再次編譯GO應用程序。如果您僅更改HTML，則優化的Dockerfile不應導致GO構建。
-

與往常一樣，本書的GitHub存儲庫有一個示例解決方案。但這是一個實驗室，您應該嘗試找到時間，因為優化Dockerfiles是您將在每個項目中使用的一項寶貴技能。但是，如果需要，我的解決方案在這裡：<https://github.com/SIXEYED/DIAMOL/BLOB/MASTER/CH04/LAB/DOCKERFILE>。最佳化

這次沒有提示，儘管我想說這個示例應用程序看起來與您在本章中已經構建的應用程序非常相似。

5與Docker Hub和其他註冊表 共享圖像

您已經花了最後幾章對*build*和*run*部分工作流程的*run*的一部分有很好的了解 - 現在是時候*share*了。 共享就是拍攝您在本地機器上構建的圖像，並使其可供其他人使用。 我認為這是Docker方程中最重要的部分。 包裝您的軟件及其所有依賴關係意味著任何人都可以輕鬆地在任何機器上使用它 - 環境之間沒有差距，因此不再有幾天的浪費設置軟件或跟蹤實際上是部署問題的錯誤。

5.1 使用註冊表，存儲庫和圖像標籤

軟件分配是內置在Docker平台中的。 您已經看到您可以從圖像運行一個容器，如果您在本地沒有該圖像，Docker將下載它。 集中存儲圖像的服務器稱為Docker *registry*. Docker Hub是最受歡迎的圖像註冊表，託管數百萬張圖像，每月下載數十億次。 這也是Docker Engine的默認註冊表，這意味著這是Docker首先尋找本地可用的圖像。

Docker圖像需要一個名稱，並且該名稱包含足夠的信息，可以為Docker找到您要尋找的確切圖像。 到目前為止，我們已經使用了一個或三個部分的非常簡單的名稱，例如圖像 - 美容或Diamol/Golang:2e。 實際上有四個部分的完整圖像名稱（正確稱為*image reference*）。 圖5.1顯示了Diamol/Golang的完整參考中的所有這些部分：2E：

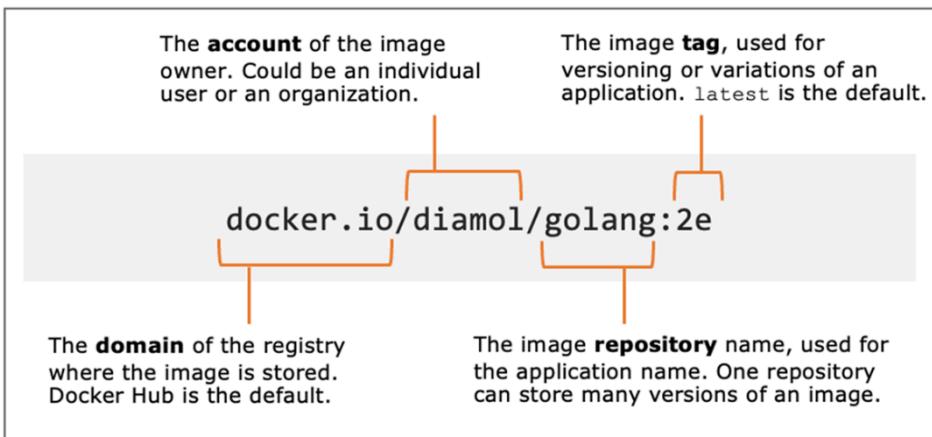


圖5.1 Docker圖像參考的解剖

當您開始管理自己的應用程序映像時，您將使用圖像參考的所有部分。在本地計算機上，您可以將圖像命名您喜歡的任何內容，但是當您想在註冊表上共享它們時，您需要添加更多詳細信息，因為圖像參考是註冊表上一個特定圖像的唯一標識符。

如果您不為圖像參考的一部分提供值，Docker使用幾個默認值。默認註冊表是Docker Hub，默認標籤是最新的。Docker Hub的域是Docker.io，所以我的圖像，Diamol/Golang：2e，是Docker.io/diamol/golang:2e的簡短版本（而且Diamol/Golang是Docker.io/diamol/golang：latest的簡短版本，這是這本書的第一版）。您可以使用其中的任何一個。Diamol帳戶是Docker Hub上的組織，Golang是該組織中的存儲庫。這是一個公共存儲庫，因此任何人都可以繪製圖像，但是您需要成為Diamol組織的成員才能推動圖像。

大公司通常在自己的雲環境或本地網絡中擁有自己的Docker註冊表。您可以通過在參考的第一部分中加入域名來定位自己的註冊表，因此Docker知道不使用Docker Hub。如果我在`r.diamol.net`主持自己的註冊表，我可以選擇自己的命名約定，以便可以將圖像存儲在`r.diamol.net/2e/golang`中。這一切都很簡單，但是圖像參考中最重要的部分是標籤。

到目前為止，您尚未使用圖像標籤，因為沒有它們就可以簡單地開始，但是當您開始構建自己的應用程序映像時，您應該始終標記它們。標籤用於識別同一應用程序的不同版本。NGINX Web服務器的官方Docker映像具有數百個標籤-NGINX：1.27是最新版本和NGINX：1.27.0-Alpine使用最小的Alpine操作系統，並且對於不同的Linux發行版和NGINX功能，還有更多。如果您在創建圖像時未指定標籤，則Docker使用默認標籤最新的標籤。這是一個誤導的名稱，因為標記為“最新”的圖像實際上可能不是最近的圖像版本。當您推動自己的圖像時，應始終使用明確的版本標記它們。

5.2 將自己的圖像推到Docker Hub

我們將開始將您在第4章中構建的圖像之一推到Docker Hub。您需要一個Docker Hub帳戶，如果您沒有一個帳戶，請瀏覽<https://hub.docker.com>並按照鏈接註冊一個帳戶（它是免費的，它不會吸引大量垃圾郵件到您的收件箱）。

您需要做兩件事將圖像推向註冊表。首先，您需要使用Docker命令行登錄註冊表，以便Docker可以檢查您的用戶帳戶是否有權推動圖像。然後，您需要給您的圖像一個參考，其中包括您有權推動的帳戶的名稱。

每個讀者都將擁有自己的Docker Hub用戶名，因此，為了使練習更容易跟隨，讓我們首先在終端會話中捕獲自己的Docker ID。之後，您可以復制並粘貼本章的其餘部分。

現在嘗試

打開終端會話，然後將Docker Hub ID保存在變量中。您的Docker ID是您的用戶名，而不是您的電子郵件地址。這是Windows和Linux上不同的命令，因此您需要選擇正確的選項：

```
# 使用PowerShell  
$ dockerid = '<你的docker-id-goes here >'  
  
# 在Linux或Mac上使用Bash  
導出dockerid = '<你的docker-id-goes here >'
```

目前我正在使用PowerShell，而我的Docker Hub用戶名已六眼，所以我運行的命令是\$ dockerid ='sixeyed';在Linux上，我將運行導出DockErid ='Sixeyed'。在任何系統上，您都可以運行Echo \$ dockerid，並且應該看到顯示的用戶名。從現在開始，您可以在練習中復制命令，它們將使用您的Docker ID。

首先登錄Docker Hub。實際上，它是推動和拉動圖像的Docker引擎，但是您可以使用Docker命令行進行身份驗證 - 當您運行登錄命令時，它將詢問您的密碼，這是您的Docker Hub密碼。

現在嘗試

驗證您是否設置了用戶名變量，然後登錄到Docker Hub。集線器是默認註冊表，因此您無需指定域名：

```
echo $ dockerId docker登錄 - 用戶$ dockerId
```

您會看到圖5.2中的輸出（例如，Docker輸入時不會顯示密碼）。

You don't have to use a variable for the username, it just makes it easier to copy commands from the rest of the chapter.

```
PS>echo $dockerId  
sixeyed  
PS>docker login --username $dockerId  
Password:  
Login Succeeded
```

You login to a registry using the Docker CLI. Docker Hub is the default registry so you don't need to specify a domain

The username is your registry username - your Docker Hub ID in this case.

圖5.2登錄到Docker Hub

現在您已經登錄了，您可以將圖像推到自己的帳戶或可以訪問的任何組織。我不認識您，但是如果我希望您的幫助照顧這本書的圖像，我可以將您的帳戶添加到Diamol組織中，您將能夠推動以Diamol/開頭的圖像。如果您不是任何組織的成員，則只能將圖像推向您自己的帳戶中的存儲庫。

您在第4章中構建了一個名為Image-Gallery的Docker映像。該圖像參考沒有帳戶名稱，因此您無法將其推向任何註冊表。但是，您無需重建圖像即可為其提供新的參考 - 圖像可以具有多個參考，就像有多個別名用於一個圖像一樣。

現在嘗試

為您現有圖像創建新的參考，將其標記為版本1：

```
docker圖像標籤圖像 - 件$ dockerid/image-gallery : v1
```

現在您有兩個參考；一個人有一個帳戶和版本號，但兩個引用都指向同一圖像。圖像還具有唯一的ID，您可以看到列出它們是否具有多個引用的列出。

現在嘗試

列出圖像 - 飾品圖像參考：

```
Docker Image LS - 濾波器參考=圖像 - 件 - 濾波器參考='*/image-gallery'
```

您會在圖5.3中看到與我相似的輸出，但標記的圖像將顯示您的Docker Hub用戶名而不是Sixeyed。

PS>docker image ls --filter reference=image-gallery --filter reference='*/image-gallery'			
REPOSITORY	TAG	IMAGE ID	CREATED
image-gallery	latest	713876e35c89	25 hours ago
sixeyed/image-gallery	v1	713876e35c89	25 hours ago
SIZE			
		35.1MB	35.1MB

There are two image references which match the filters I supplied - these could be two different images

But they have the same image ID, which means it's one image with two references

Both references have a virtual size of 35MB, but physically they share the same image layers

圖5.3一個帶有兩個參考的圖像

現在，您在帳戶名中具有帶有Docker ID的圖像參考，並且已登錄Docker Hub，因此您可以分享您的圖像！Docker Image Push命令是拉動命令的對應物；它將您的本地圖像層上傳到註冊表。

現在嘗試

列出圖像 - 飾品圖像參考：

Docker Image推送\$ Dockerid/Image-Gallery : V1

Docker Instristries以與本地Docker Engine相同的方式在圖像層的層面上工作。您推動圖像，但Docker實際上上傳了圖像層。在輸出中，您將看到一層ID列表及其上傳進度。在我的（縮寫）輸出中，您可以看到被推的層：

推動是指存儲庫[docker.io/sixeyed/image-gallery] 4E0FE95BD31A：推送B1B7E35
1754C：推送7BB737F39B61：推動...

V1：摘要：SHA256：EE332F8 ...尺寸：1574

註冊表與圖像層合作的事實是您需要花費時間優化Dockerfiles的另一個原因。僅當沒有現有的層匹配該層的哈希（Hash）時，只有在註冊表上進行物理上載。就像您當地的Docker Engine Cache一樣，但在註冊表上的所有圖像中都應用。如果您優化構建時90%的層來自緩存的地點，則在按下時，這些層中的90%已經在註冊表中。優化的Dockerfiles減少了構建時間，磁盤空間和網絡帶寬。

您可以立即瀏覽到Docker Hub並檢查您的圖像。Docker Hub UI使用與圖像引用相同的存儲庫名稱格式，因此您可以從帳戶名稱中算出圖像的URL。

現在嘗試

這個小腳本在Docker Hub上打印出圖像頁面的URL：

```
echo "https://hub.docker.com/r/qulydockerid/image-gallery/ta GS"
```

When you browse to that URL, you'll see something like figure 5.4, showing the tags for your image and the last update time.

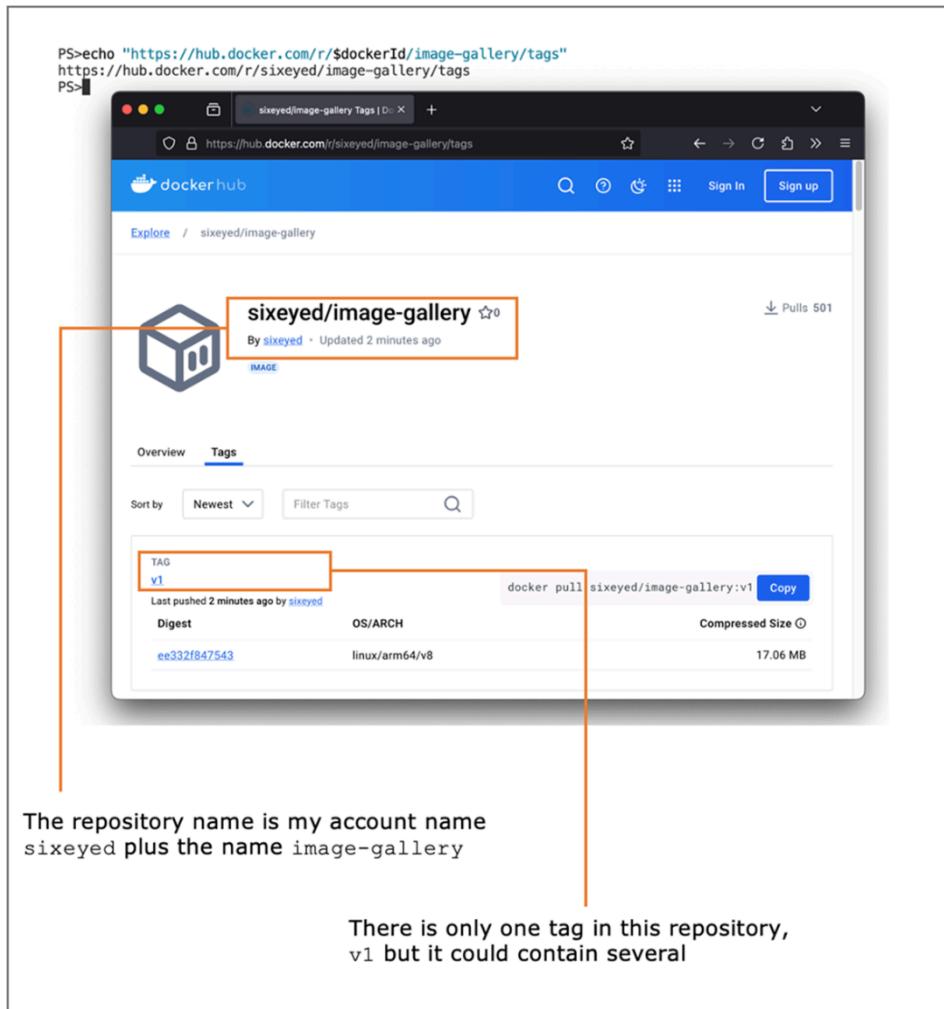


圖5.4 Docker Hub上的圖像列表

這就是推動圖像的全部。Docker Hub如果尚不存在的圖像為圖像創建一個新的存儲庫，默認情況下，存儲庫具有公共閱讀權。現在，任何人都可以找到，拉動和使用您的圖像 - 飾品應用程序。他們需要自己使用它如何使用它，但是您也可以將文檔放在Docker Hub上。

Docker Hub是最簡單的註冊表，它為您提供了零成本的大量功能，儘管您可以每月訂閱額外功能，例如私人存儲庫。也有很多替代註冊表。註冊表是一個開放的API規格，核心註冊服務器是Docker的開源產品。所有的雲都有自己的註冊表服務，您可以在數據中心使用商業產品（例如Artifactory或Nexus）管理自己的註冊表，也可以在容器中運行簡單的註冊表。

5.3運行並使用您自己的Docker註冊表

在本地網絡上運行自己的註冊表非常有用。它縮短了帶寬的使用和傳輸時間，它使您可以在環境中擁有數據。即使您不擔心這一點，也很高興知道您可以快速旋轉本地註冊表，如果您的主註冊表脫機，您可以用作備份選項。

Docker在源代碼存儲庫Docker/Distribution中的GitHub上維護核心註冊服務器。它為您提供了推動和拉圖像的基本功能，並且它使用與Docker Hub相同的層緩存系統，但是它並不能為您提供帶有輪轂的Web UI。這是一台我將Diamol圖像包裝到的超級輕量級服務器，因此您可以在容器中運行它。

現在嘗試

使用我的圖像在容器中運行Docker註冊表：

```
# 使用重新啟動標誌運行註冊表，以便每當您重新啟動Docker時，容器將#重新啟動：Doc  
ker容器運行-D -P 5010：5000 - 啟用始終diamol/registry：2e
```

現在，您在本地計算機上擁有註冊服務器。服務器的默認端口為5000，但可能在您的計算機上使用，因此該命令將其發佈到端口5010。您可以使用域localhost：5010標記圖像並將其推入此註冊表，但這並不是很有趣 - 您只能在本地計算機上使用註冊表。相反，最好給您的機器一個別名，以便您可以為註冊表使用適當的域名。

下一個命令會創建該別名。除了它擁有的任何其他網絡名稱外，它還將為您的計算機提供名稱註冊表。它通過寫入計算機的主機文件來做到這一點，這是一個簡單的文本文件，將網絡名稱鏈接到IP地址。

現在嘗試

Windows，Linux和MAC機器都使用相同的主機文件格式，但是文件路徑不同。GIT存儲庫中有一些有用的腳本，您可以用來添加值：

```
# 使用PowerShell（以管理員的身份運行）./scripts/add-to-hosts.ps1 registry.local 127.0.0.1

#Linux/MacOS
chmod +x ./scripts/add-to-hosts.sh
sudo ./scripts/add-to-hosts.sh registry.local 127.0.0.1
```

如果使用PowerShell從該命令中獲取權限錯誤，則需要在Windows上的高架會話中使用管理員特權登錄，或使用Sudo PWSH在Linux或Mac上啟動高升會話。成功運行命令後，您應該能夠運行ping registry.local，並在計算機的主頁IP地址（127.0.0.1）中查看響應，如圖5.5所示。

```
PS>./scripts/add-to-hosts.ps1 registry.local 127.0.0.1
PS>
PS>ping registry.local
PING registry.local [127.0.0.1] 56 data bytes
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.068 ms
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.159 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.171 ms
^C

registry.local is the new network
name for your own computer, which
you added to the hosts file           127.0.0.1 is your computer's
                                         "home" IP address
```

圖5.5為您的計算機添加新的網絡別名

現在，您可以使用域名註冊表。局部：圖像引用中的5010以使用您的註冊表。將域名添加到圖像中涉及您已經為Docker Hub完成的標記過程。這次，您只需在新圖像參考中包含註冊表域即可。

現在嘗試

使用您的註冊表域標記圖像 - 飾品圖像：

```
docker圖像標籤圖像 - 件註冊表。局部：5010/GALLERY/UI：V1
```

您的本地註冊表沒有任何身份驗證或授權設置。這顯然不是生產質量，但它可能適用於一個小型團隊，它確實可以讓您使用自己的圖像命名方案。第4章中的三個容器構成了NASA圖像的日常應用程序。您可以將所有圖像標記為使用畫廊作為項目名稱將它們組合在一起：

- 註冊表：5010/GALLERY/UI：V1 - GO WEB UI註冊表：5010/Gallery
- /API：V1 - Java API註冊表。5010/Gallery/logs/logs：v1 - node.js api
- api

您還需要做一件事，然後才能將此圖像推向本地註冊表。註冊表容器使用普通文本HTTP而不是加密的HTTP來推動圖像。Docker默認情況下不會與未加密的註冊表進行通信，因為它不安全。您需要在Docker允許您使用它之前將您的註冊表域明確添加到允許的不安全註冊表的列表中。

這使我們能夠配置Docker。Docker Engine對各種設置使用JSON配置文件，包括Docker將圖像層存儲在磁盤上的位置，Docker API在其中聆聽連接以及允許哪些不安全的註冊表。該文件稱為守護程序。您可以直接在這些系統上編輯文件，但是如果在Mac或Windows上使用Docker桌面，則需要使用UI訪問配置文件。

現在嘗試

右鍵單擊任務欄中的Docker鯨魚圖標，然後選擇設置。然後打開Docker Engine選項卡，您將看到JS ON配置。圖5.6顯示了默認設置。

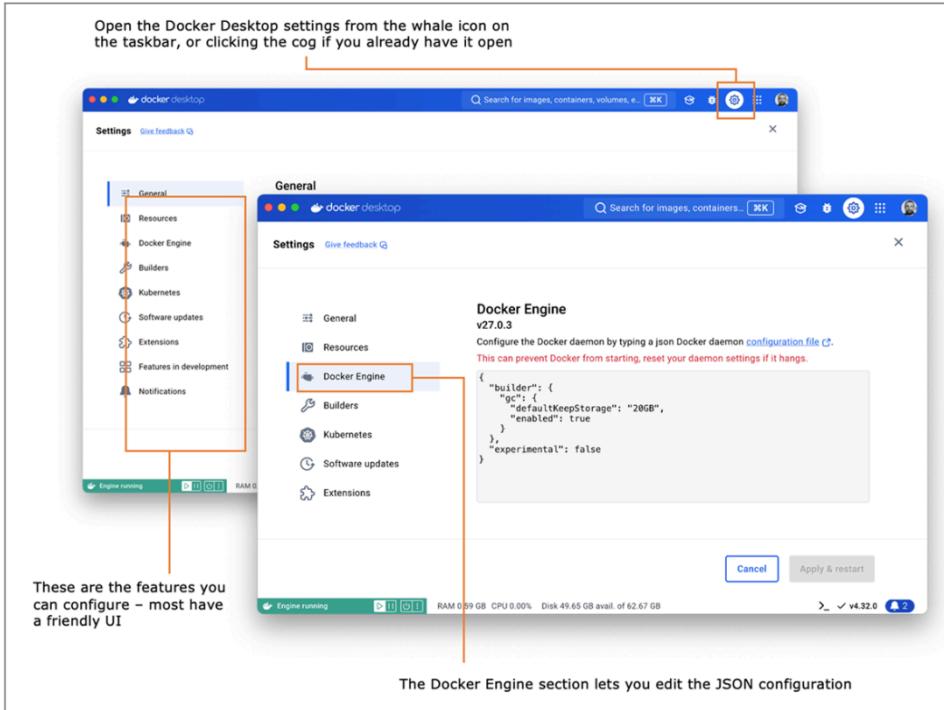


圖5.6編輯Docker引擎配置設置在Docker桌面中

如果您不運行Docker桌面，則需要手動編輯文件。首先在文本編輯器中打開守護程序文件，或者如果不存在，則創建它。現在，無論您以哪種方式運行Docker，都需要在JSON中添加一個部分，並告訴Docker您要使用的任何不安全註冊表的地址。配置設置將看起來像這樣 - 但是如果要編輯現有文件，請確保將原始設置留在其中：

```
{ "Insecure-Registries" : [ "igemistry.local:5010" ]}
```

您需要確保正確地完成JSON；否則，當您嘗試重新啟動時，Docker將失敗。如果您已經在JSON中具有設置，請在最後一個值之後添加逗號，然後插入不安全的註冊塊。您可以在圖5.7中看到使用Docker桌面的外觀：

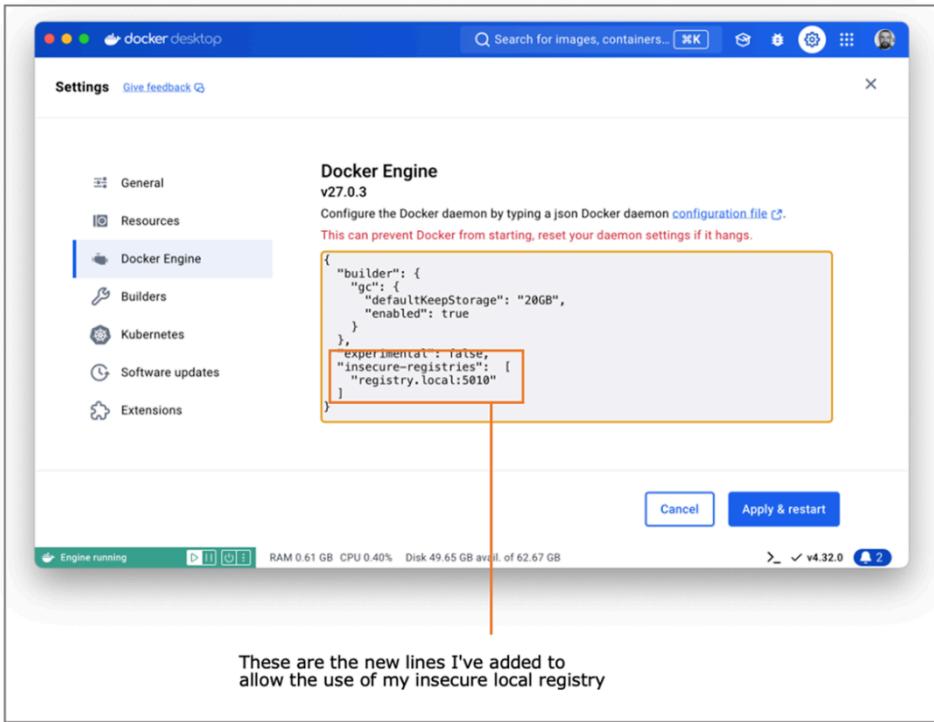


圖5.7將本地不安全註冊表添加到Docker引擎配置

需要重新啟動Docker引擎以加載任何新的配置設置，並且Docker Desktop在應用更改時為您做到這一點。否則，您需要在Windows Server上運行重新啟動服務Docker，或在Linux上運行Service Docker。 您可以使用Info命令檢查Docker Engine允許的Docker Engine允許使用哪些不安全的註冊表。

現在嘗試

列出有關您的Docker引擎的信息，並驗證您的註冊表是否在不安全的註冊表中：

Docker信息

在輸出結束時，您會看到註冊表配置，其中應包括您的不安全註冊表 - 您可以在圖5.8中看到我的註冊表。

```
PS>docker info
...
Experimental: false
Insecure Registries:
hubproxy.docker.internal:5555
registry.local:5010
127.0.0.0/8
Live Restore Enabled: false

Registries listed here can be used over HTTP;
otherwise Docker will only use HTTPS registries
```

圖5.8 Docker配置為允許的不安全註冊表

您應該謹慎將不安全的註冊表添加到Docker配置中。 您的連接可能會受到損害，並且推動圖像時攻擊者可以閱讀層。或者，更糟糕的是，當您提取圖像時，他們可以注入自己的數據。所有商業註冊表服務器在HTTPS上運行，您還可以配置Docker的開源註冊表以使用HTTPS。但是，要使用本地服務器進行演示，這是可以接受的風險。

現在，您可以將標記的圖像推向自己的註冊表。註冊表域是圖像參考的一部分，因此Docker知道使用Docker Hub以外的其他功能，並且您的HTTP註冊表在容器中運行的HTTP註冊表已在不安全的註冊表中清除。

現在嘗試

按下標記的映像 - 如果您看到錯誤說服務器給HTTP響應HTTPS客戶端，那麼您的Docker引擎沒有配置為使用您的不安全註冊表。再次檢查配置並確保已重新啟動引擎：

Docker Image Push Registry.Local : 5010/Gallery/UI : V1

運行第一個推送時，您的註冊表將完全空，因此您會看到上傳的所有層 - 我的輸出在圖5.9中：

```
PS>docker image push registry.local:5010/gallery/ui:v1
The push refers to repository [registry.local:5010/gallery/ui]
4e0fe95bd31a: Pushed
b1b7e351754c: Pushed
7bb737f39b61: Pushed
85959e5d9b71: Pushed
d30c266bf726: Pushed
9110f7b5208f: Pushing 9.11MB
```

My registry is empty, so all the layers get pushed. If I repeated the command, the registry would tell Docker it already has the layers stored

This is my local registry running in a container, but the domain name could refer to a remote registry service

圖5.9將一組新的圖像層推向Docker註冊表

現在，如果您重複推送命令，您會發現所有層已經存在，沒有任何上傳。這就是您在容器中運行自己的Docker註冊表所需要做的。您可以使用計算機的IP地址或真實域名在網絡上共享它。

5.4有效地使用圖像標籤

您可以將任何字符串放入Docker Image標籤中，而且，正如您已經看到的那樣，您可以為同一圖像具有多個標籤。您將使用它來版本在圖像中版本，並讓用戶對他們想要使用的內容做出明智的選擇，並在使用其他人的圖像時做出自己的知情選擇。

許多軟件項目都使用具有小數點的數字版本管理方案來指示版本之間的變化多大，並且您可以使用圖像標籤來做到這一點。基本思想是[主要]。[次要]。[patch]，它具有一些隱含的保證。僅遞增補丁編號的版本可能具有錯誤修復，但是它應該具有與最後一個版本相同的功能。縮小次要版本的發行版可能會添加功能，但不應刪除任何功能；主要版本可能具有完全不同的功能。

如果您在圖像標籤上使用相同的方法，則可以讓用戶選擇是堅持主要版本還是次要版本，或者始終具有最新版本。

現在嘗試

為您在圖像中包裝的GO應用程序創建一些新標籤，以指示專業，次要和補丁發布版本：

```
docker image tag image-gallery registry.local:5010/gallery/ui:latest docker image tag image-gallery registry.local:5010/gallery/ui:2 docker image tag image-gallery registry.local:5010/gallery/ui:2.1 docker image tag image-gallery registry.local:5010/gallery/ui:2.1.106
```

現在，想像一下該應用程序每月發行版，可以增加版本號。圖5.10顯示了圖像標籤如何從7月到10月的發行版發展。

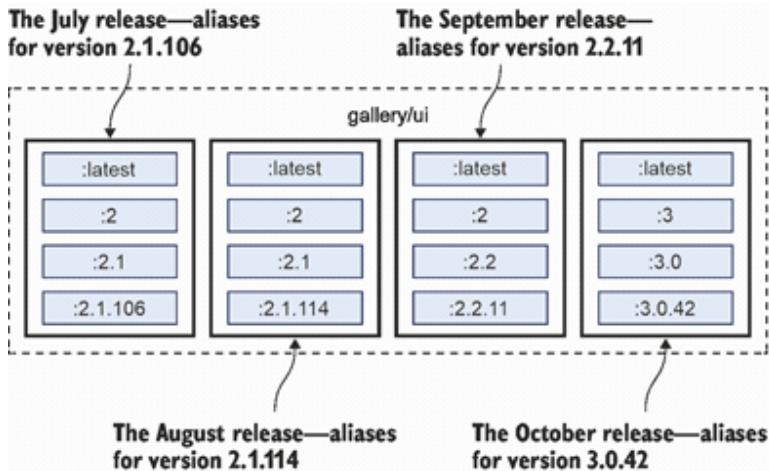


圖5.10軟件發布過程中圖像標籤的演變

您會看到其中一些圖像標籤是一個移動的目標。Gallery/UI : 2.1是7月份2.1.106版本的別名，但在8月，相同的2.1標籤是2.1.114版本的別名。畫廊/UI : 2也是7月份的2.1.106的別名，但到9月2日，標籤是2.2.11版本的別名。最新標籤的動作最多 - 在7月畫廊/UI是2.1.106的別名，但在10月，這是3.0.42的別名。

這是您將在Docker Images上看到的典型版本控制方案。 您應該收養自己，因為它可以讓您的圖像用戶選擇他們想要的最新狀態。他們可以將圖像拉命令或dockerfiles的“從指令”中的指令固定在特定的補丁版本中，並確保他們使用的圖像始終是相同的。 在此示例中，2.1.106標籤是從7月到10月的同一圖像。 如果他們想獲得補丁更新，可以使用2.1標籤，如果他們想獲得次要版本，則可以使用2個標籤。

這些選擇都可以。 這只是平衡風險的情況 - 使用特定的補丁版本意味著該應用程序在使用時將相同，但您不會獲得安全修復程序。使用主要版本意味著您將獲得所有最新的修復程序，但是可能會有意外的功能更改。

為自己的Dockerfiles中的基本圖像使用特定的圖像標籤尤為重要。使用產品團隊的構建工具映像來構建您的應用程序和他們的運行時映像來打包應用程序，這很有幫助，但是如果我不指定標籤中的版本，則將在將來為麻煩而設置自己。構建圖像的新版本可能會破壞您的Docker構建。 或更糟糕的是，運行時的新版本可能會破壞您的應用程序。

5.5 將官方圖像變成黃金圖像

當您查看Docker Hub和其他註冊表時，要了解的最後一件事要了解：您能相信在那裡找到的圖像嗎？ .NET 圖像發佈在Microsoft容器註冊表上，只有Microsoft才能推動，因此您可以對這些註冊表充滿信心，但是任何人都可以將圖像推到Docker Hub並公開可用。對於黑客來說，這是分發惡意軟件的好方法。您只需要給您的圖像一個無辜的名稱和虛假描述，然後等待人們開始使用它即可。Docker Hub通過經過驗證的發布者和官方圖像解決了該問題。

*Verified publishers*是亞馬遜，VMware和Atlassian等公司，他們在Docker Hub上發布圖像。 他們的圖像完成了一個批准過程，其中包括對漏洞的安全掃描。如果您想在容器中運行現成的軟件，則經過驗證的發布者的圖像是最好的選擇。

*Official images*是不同的 - 它們通常是由項目團隊和Docker共同維護的開源項目。 它們是安全掃描並定期更新的，並且符合Dockerfile的最佳實踐。官方圖像的所有內容都是開源的，因此您可以在Github上看到Dockerfiles。 大多數人開始使用官方圖像作為自己圖像的基礎，但在某個時候發現他們需要更多的控制。 然後，他們介紹了自己的首選基本圖像，稱為*golden images* - 圖5.11向您展示了它的工作原理。

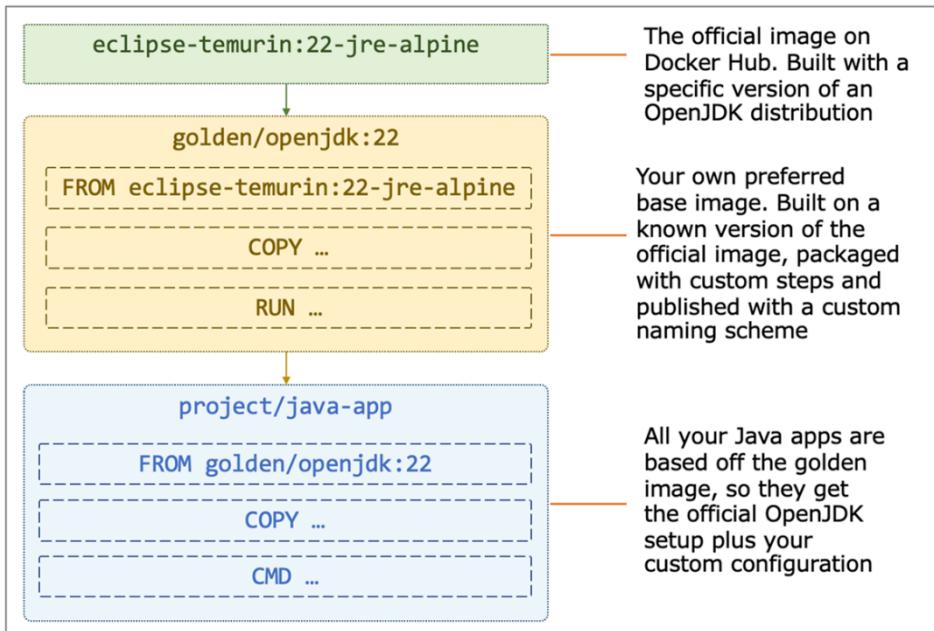


圖5.11使用黃金圖像封裝官方圖像

Golden Images使用官方圖像作為基礎，然後添加所需的任何自定義設置，例如安裝安全證書或配置默認環境設置。 黃金圖像生活在公司的Docker Hub或其自己的註冊表上的存儲庫中，所有應用程序圖像均基於黃金圖像。 這種方法提供了官方圖像的好處，即項目團隊的最佳實踐設置，但是藉助您需要的額外配置和自己選擇更新節奏。

現在嘗試

本章的源代碼中有兩個Dockerfiles可以作為.NET應用程序的黃金圖像構建。瀏覽每個文件夾並構建圖像：

```
CD CH05/練習/dotnet-sdk docker映像構建-t金/dotnet-sdk
: 2e °
```

```
CD ../aspnet-runtime
Docker Image Build -t Golden/Aspnet : 2e °
```

黃金圖像沒有什麼特別的。它們從Dockerfile開始，並使用您自己的參考和命名方案構建圖像。如果您查看構建的Dockerfiles，您會發現它們使用標籤指令在圖像中添加了一些元數據，並設置了一些常見的配置。現在，您可以在多階段Dockerfile中使用這些圖像進行.NET應用程序，看起來像列出5.1。

列出5.1使用.NET核心黃金圖像的多階段Dockerfile

來自Golden/dotnet-sdk：2e作為構建器副本。運行
dotnet Publish -o /out/app app.csproj

來自Golden/aspnet：2e複製-from = builder/out/app cmd [“dotnet”，“”/app/p/app.dll”]

該應用程序DockerFile具有與任何多階段構建的格式相同，但是現在您擁有基本圖像。官方圖像可能每個月都有新版本，但是您可以選擇將黃金圖像限制為季度更新。黃金圖像打開了另一種可能性 - 您可以在連續集成（CI）管道中使用工具來強制使用它們：可以掃描Dockerfiles，並且如果有人試圖在不使用金色圖像的情況下構建應用程序，則會構建失敗。這是鎖定源圖像團隊可以使用的好方法。

5.6 實驗室

這個實驗室將進行一些偵探工作，但最終值得。您需要在Docker註冊表API V2規範（<https://github.com/opencontainers/distribution-spec/blob/v1.0.1/spec.md>）周圍挖掘，因為REST API是您可以與本地Docker註冊表進行交互的唯一方法 - 您無法使用Docker Cli搜索或刪除docker cli cli。

該實驗室的目標是將您的畫廊/UI圖像的所有標籤推向您的本地註冊表，驗證它們都在那裡，然後刪除所有標籤，然後驗證它們已經消失了。我們將不包括畫廊/API或畫廊/日誌圖像，因為此實驗室重點是具有多個標籤的圖像，並且我們將其用於畫廊/UI。這裡有一些提示：

- 您可以使用單個圖像推動命令來推動所有這些標籤。您本地註冊表REST API的URL為`http://registry.local:5010/v2`。首先列出存儲庫的圖像標籤。然後，您需要獲得圖像清單。您可以通過API刪除圖像，但是您需要使用清單。閱讀文檔 - 您需要在頭部請求中使用的特定請求標頭。
-
-
-

解決方案是在本書的GitHub存儲庫中，這是一個難得的情況，可以作弊。前幾個步驟應該直接鍛煉身體，但是隨後變得有些尷尬，所以如果您最終前往這裡：<https://github.com/SIXEYED/DIAMOL/tree/2E/CH05/LAB>。

Good luck. And remember to read the docs.

6 使用Docker捲進行持久存儲

容器是無狀態應用程序的完美運行時。您可以通過在群集上運行多個容器來滿足需求增加，因為每個容器都會以相同的方式處理請求。 您可以通過自動滾動升級發布更新，從而使您的應用程序始終保持在線。

但是，並非您的應用程序的所有部分都是無狀態的。將有一些組件使用磁盤來提高性能或永久數據存儲。您也可以在Docker容器中運行這些組件。

存儲確實添加了並發症，因此您需要了解如何停止狀態應用程序。本章將引導您瀏覽Docker捲和安裝座，並向您展示容器文件系統的工作原理。

6.1 為什麼容器中的數據不是永久的

Docker容器具有帶有單個磁盤驅動器的文件系統，該驅動器的內容與圖像中的文件填充。您已經看到了這一點：當您在Dockerfile中使用複制指令時，從圖像運行容器時，您將複製到圖像中的文件和目錄就在那裡。而且您知道Docker圖像被存儲為多層，因此容器的磁盤實際上是一個虛擬文件系統，可以通過將所有圖像層合併在一起來建立Docker。

每個容器都有自己的文件系統，獨立於其他容器。 您可以從同一docker映像運行多個容器，它們都將從相同的磁盤內容開始。該應用程序可以更改一個容器中的文件，並且不會影響其他容器中的文件或圖像中的文件。 通過運行幾個寫入數據然後查看其輸出的容器，可以直接看到這一點。

現在嘗試

打開終端會話，並從同一圖像運行兩個容器。 圖像中的應用程序將隨機數寫入容器中的文件：

```
Docker Container Run -name RN1 Diamol/CH06-random-number : 2e
```

```
Docker Container Run-Name RN2 Diamol/ch06-random-number : 2e
```

該容器啟動時運行該容器，並且腳本將一些隨機數據寫入文本文件，然後結束，因此這些容器處於退出狀態。 兩個容器從同一圖像開始，但是它們將具有不同的文件內容。 您在第2章中了解到，Docker在退出時不會刪除容器的文件系統 - 保留它，因此您仍然可以訪問文件和文件夾。

Docker CLI具有Docker容器CP命令，可以在容器和本地計算機之間複製文件。您可以指定容器和文件路徑的名稱，並且可以使用它將這些容器中生成的隨機數文件複製到主機計算機上，以便您可以讀取內容。

現在嘗試

使用Docker容器CP從每個容器中複制隨機數文件，然後檢查內容：

```
Docker容器CP RN1 : /random/number.txt Number1.txt
```

```
Docker容器CP RN2 : /random/number.txt number2.txt
```

```
CAT Number1.txt
```

```
CAT NUMBER2.TXT
```

您的輸出將與我的輸出相似，如圖6.1所示。每個容器都在同一路徑，/random/number.txt上寫了一個文件，但是當將文件複製到本地計算機上時，您可以看到內容不同。 這是一種簡單的方法，即表明每個容器都有一個獨立的文件系統。在這種情況下，它是一個不同的文件，但是這些文件可能是數據庫容器，從同一SQL引擎運行但存儲完全不同的數據開始。

```

This runs two containers from the same image. rn1 and rn2 will
start with the exact same filesystem contents from the image

PS>docker container run --name rn1 diamol/ch06-random-number:2e
PS>docker container run --name rn2 diamol/ch06-random-number:2e
PS>
PS>docker container cp rn1:/random/number.txt number1.txt
Successfully copied 2.05KB to /Users/elton/scm/github/sixeyed/diamol/ch06/number1.txt
PS>docker container cp rn2:/random/number.txt number2.txt
Successfully copied 2.05KB to /Users/elton/scm/github/sixeyed/diamol/ch06/number2.txt
PS>
PS>cat number1.txt
11681
PS>cat number2.txt
21354

This copies files from containers to the host. rn1 and
rn2 both created a file at /random/number.txt when
they started, but the contents will be different

Reading the contents of the files copied from each
container. The random numbers are different

```

圖6.1運行寫數據的運行容器並檢查數據

容器內的文件系統似乎是一個磁盤：在Linux容器上可以 /dev /sda1和Windows容器上的C :\。但是，該磁盤是一個虛擬文件系統，該文件系統從多個來源構建，並作為一個單元作為一個單元構建。該文件系統的基本源是圖像層，可以在容器之間共享，並且容器的可寫入層是每個容器所獨有的。

圖6.2顯示瞭如何查看隨機數圖像和兩個容器。您應該從圖6.2中取出兩個重要的東西：圖像層共享，因此必須僅閱讀，並且每個容器中有一個可寫的層，該層具有與容器相同的生命週期。圖像層具有自己的生命週期 - 您拉的任何圖像都會保留在當地的緩存中，直到您刪除它們。但是，當容器啟動時，docker創建了容器可寫的層，當卸下容器時，docker將其刪除（停止容器不會自動將其刪除，因此仍然存在停止的容器文件系統）。

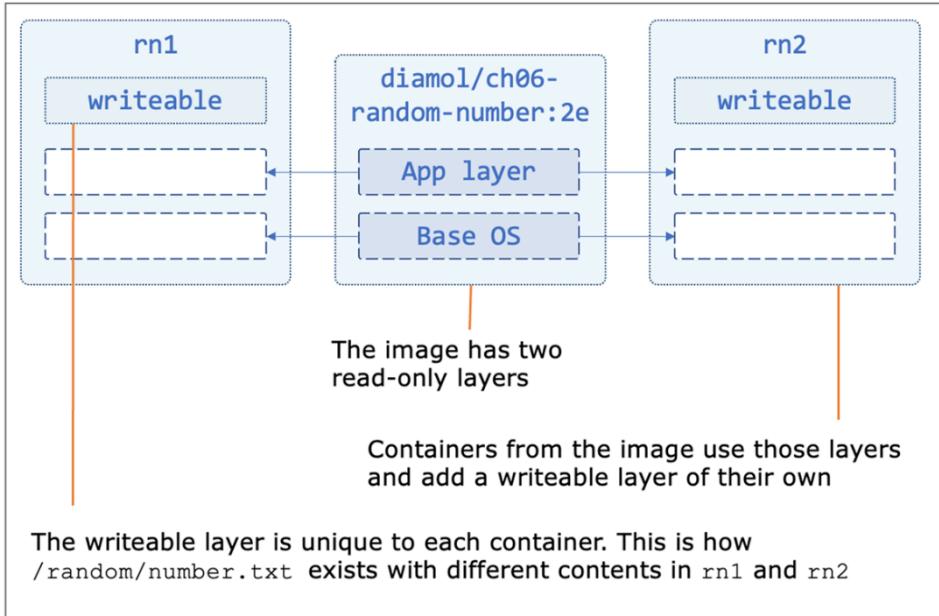


圖6.2容器文件系統是由圖像層和可寫入層構建的。

當然，可寫的層不僅用於創建新文件。容器可以從圖像層編輯現有文件。但是圖像層是只讀的，因此Docker做了一些特殊的魔術來實現這一目標。它使用*copy-on-write*進程允許編輯到來自僅讀取層的文件。當容器試圖在圖像層中編輯文件時，Docker實際上將該文件的副本填充到了可寫的層中，並且編輯發生在此處。對於容器和應用程序而言，這都是無縫的，但這是Docker超高效儲存的基石。

在我們繼續運行一些更有用的狀態容器之前，讓我們通過一個更簡單的示例來仔細研究。在此練習中，您將運行一個容器，該容器從圖像層打印出文件的內容。然後，您將更新文件內容，然後再次運行容器以查看發生了什麼變化。

現在嘗試

運行這些命令以啟動一個集裝箱，該容器打印出其文件內容，然後更改文件並再次啟動容器以打印出新的文件內容：

```
Docker容器運行 - 名稱F1 Diamol/CH06文件顯示：2
```

e

```
echo "https://blog.sixeyed.com" > url.txt
```

```
Docker容器CP url.txt f1 :/input.txt
```

```
Docker Container Start -Attach F1
```

這次，您使用Docker將主機計算機的文件複製到容器中，而目標路徑是容器顯示的文件。當您再次啟動容器時，同一腳本會運行，但是現在它打印出不同的內容 - 您可以在圖6.3中看到我的輸出。

This container prints out the contents of the `input.txt` file when it runs. This first run uses the `input.txt` contents from the image

```
PS>docker container run --name f1 diamol/ch06-file-display:2e  
https://www.manning.com/books/learn-docker-in-a-month-of-lunches  
PS>  
PS>echo "https://blog.sixeyed.com" > url.txt  
PS>  
PS>docker container cp url.txt f1:/input.txt  
Successfully copied 2.05kB to f1:/input.txt  
PS>  
PS>docker container start --attach f1  
https://blog.sixeyed.com
```

Now this command generates a new file on the local computer with different content, and uses that to overwrite the `input.txt` file in the container

The container exited after the first run. Starting it again means it runs the same command, but now it's reading the new file contents

圖6.3修改容器的狀態並再次運行

修改容器中的文件會影響該容器的運行方式，但不會影響圖像或該圖像中的任何其他容器。僅適用於一個容器，更改的文件將其生存在可寫的層中 - 一個新容器將使用圖像中的原始內容，並且當刪除容器F1時，更新的文件就消失了。

現在嘗試

啟動一個新容器，以檢查圖像中的文件是否不變。然後卸下原始容器，並確認數據已經消失：

Docker容器運行 - 名稱F2 Diamol/CH06文件顯示：2

e

Docker容器RM -F F1

Docker容器CP F1 : /input.txt。

您會看到與圖6.4中我相同的輸出。新容器使用圖像中的原始文件，當您刪除原始容器時，將刪除其文件系統，並且更改的文件將永遠消失。

Running a new container from the same image prints the original data, so the file edited in container f1 has not altered the image

```
PS>docker container run --name f2 diamol/ch06-file-display:2e  
https://www.manning.com/books/learn-docker-in-a-month-of-lunches  
PS>  
PS>docker container rm -f f1  
f1  
PS>  
PS>docker container cp f1:/input.txt .  
Error response from daemon: No such container: f1
```

Removing a container deletes its writeable layer. Any data in that layer is gone forever, so you should view the container's filesystem as transient storage

圖6.4修改容器中的文件不會影響圖像，並且容器的數據是瞬態的。

容器文件系統具有與容器相同的生命週期，因此，卸下容器時，卸下可寫入層並丟失了容器中的任何更改的數據。卸下容器是您會做很多事情的事情。在生產中，您可以通過構建新圖像，刪除舊容器並用更新圖像中的新圖像來升級應用程序。丟失了原始應用程序容器中的任何數據，替換容器始於圖像中的靜態數據。

在某些情況下，這很好，因為您的應用程序僅撰寫瞬態數據（可能要保留計算或檢索價格昂貴的本地數據緩存 - 替換容器可以從空的緩存開始。在其他情況下，這將是一場災難。您可以在容器中運行數據庫，但是在推出更新的數據庫版本時，您不會期望丟失所有數據。

Docker也為您覆蓋了這些情況。容器的虛擬文件系統始終是由圖像層和可寫入層構建的，但也可以提供其他來源。這些是*Docker volumes*和*mounts*。它們與容器有一個單獨的生命週期，因此可以用來存儲在容器更換之間持續存在的數據。

6.2 帶有碼頭量的容器運行容器

Docker音量是一個存儲單元 - 您可以將其視為容器的USB棒。數量獨立於容器並具有自己的生命週期，但可以將其附著在容器上。當數據需要持久時，您可以如何管理狀態應用程序存儲。您創建卷並將其附加到應用程序容器上；它在容器文件系統中作為目錄顯示為目錄。容器將數據寫入目錄，該目錄實際上存儲在卷中。當您使用新版本更新應用程序時，您將相同的捲附加到新容器上，所有原始數據都可以使用。

有兩種與容器一起使用卷的方法：您可以手動創建卷並將其連接到容器上，也可以在Docker-File中使用音量指令。這會構建一個啟動容器時會創建卷的圖像。語法只是卷< target-directory >。列表6.1顯示了圖像Diamol/ch06-todo-list：2e的多階段Dockerfile的一部分，這是一個使用卷的狀態應用程序。

使用卷列出多階段Dockerfile的6.1一部分

```
來自Diamol /dotnet-herpnet：2e workdir /app ent  
rypoint [ “dotnet” , “todolist.dll” ]
```

音量 /數據

複製 - 從= builder / out / 。

當您從此圖像運行容器時，Docker將自動創建卷並將其連接到容器上。該容器將在 /數據（或Windows容器上的C:\Data）中具有一個目錄，它可以從正常情況下讀取並寫入。但是，實際上，數據存儲在卷中，該卷將在刪除容器之後使用。您會看到，如果從圖像中運行容器，然後檢查卷。

現在嘗試

為待辦事項列表應用程序運行一個容器，並查看創建的捲碼頭：

```
Docker容器檢查-Format'{ json .mounts }'todo1
```

```
Docker卷LS
```

您會看到圖6.5中的輸出。Docker為此容器創建一個卷，並在容器運行時將其附加。我已經過濾了卷列表以顯示我的容器的捲。

```
This container run command doesn't mention volumes, but a volume is
declared in the Dockerfile so one will be created and attached to the container

PS>docker container run --name todo1 -d -p 8010:8080 diamol/ch06-todo-list:2e
32d016639226cbb8a9ef3e08f224bc09940764816bedcc8b95f14b63aea0ca0a
PS>
PS>docker container inspect --format '{{json .Mounts}}' todo1 | ConvertFrom-Json

Type      : volume
Name      : 7108b04dde10380fa48ad8b84ac175f91b7fc0919a683313375554b4c9480a6a
Source    : /var/lib/docker/volumes/7108b04dde10380fa48ad8b84ac175f91b7fc0919
             a683313375554b4c9480a6a/_data
Destination : /data
Driver     : local
Mode       :
RW         : True
Propagation :

Volumes are shown as "mounts" when you inspect a container. The
output is JSON but I'm using a PowerShell command to make it
user-friendly; it shows the volume ID, the physical source of the
volume on the host, and the target directory inside the container

PS>docker volume ls
DRIVER   VOLUME NAME
local    7108b04dde10380fa48ad8b84ac175f91b7fc0919a683313375554b4c9480a6a

Volumes are first-class citizens. You use docker volume commands to create, list,
inspect and remove them. This is the volume created by Docker for this container
```

圖6.5在Dockerfil中運行帶有音量的容器

ODO1 -D -P 8010 : 8080 Diamol/ch06 -todo -list : 2E

E Docker容器運行-Name T

©Manning Publications Co.要發表評論，請轉到Livebook

許可獲得Huan-lin Tsai < huanlin.tsai@gmail.com >

Docker量完全透明了在容器中運行的應用程序。瀏覽到`http://localhost:8010`，您會看到待辦事項應用程序。該應用程序將數據存儲在 /數據目錄處的文件中，因此，當您通過網頁添加項目時，它們將存儲在Docker卷中。圖6.6顯示了該應用程序中的應用程序 - 這是一個特殊的待辦事項清單，非常適合像我這樣的工作負載的人； 您可以添加項目，但您永遠無法刪除它們。

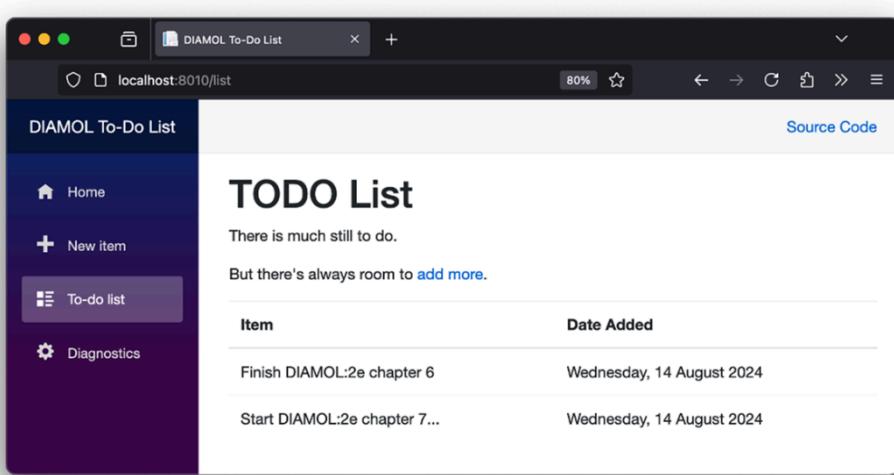


圖6.6永無止境的待辦事項列表，使用Docker卷在容器中運行

Docker圖像中聲明的捲是為每個容器的單獨卷創建的，但是您也可以在容器之間共享量。如果您啟動一個運行待辦事項應用程序的新容器，它將具有自己的捲，待辦事項列表將開始空置。但是，您可以運行帶有捲符號標誌的容器，該容器將附加另一個容器的量。在此示例中，您可以擁有兩個待辦事項應用程序容器共享相同的數據。

現在嘗試

運行第二個待辦事項列表容器，然後檢查數據目錄的內容。然後將其與另一個共享第一個容器中卷的新容器進行比較（Windows和Linux的EXEC命令略有不同）：

```

# this new container will have its own volume
docker container run --name todo2 -d diamol/ch06-todo-list:2e

# on Linux:
docker container exec todo2 ls /data

# on Windows:
docker container exec todo2 cmd /C "dir C:\data"

# this container will share the volume from todo1
docker container run -d --name t3 --volumes-from todo1 diamol/ch06-todo-list:2e

# on Linux:
docker container exec t3 ls /data

# on Windows:
docker container exec t3 cmd /C "dir C:\data"

```

輸出看起來像圖6.7（我在Linux上為此示例運行）。第二個容器以新卷開始，因此 / 數據目錄為空。第三個容器使用第一個容器，因此可以從原始應用程序容器中查看數據。

This starts a new container which will have a new volume created for it

The new volume is empty, there are no files at /data

```

PS> docker container run --name todo2 -d diamol/ch06-todo-list:2e
5abdc4375e5e9ebd2471035c40f48d7/cb681520e4/c828d2t58t633e6f0b2/b
PS>
PS> docker container exec todo2 ls /data
PS>
PS> docker container run -d --name t3 --volumes-from todo1 diamol/ch06-todo-list:2e
a9912eb07d0c59e885eb903da6e09d5cd8451bc88011398341250d658df410d0
PS>
PS> docker container exec t3 ls /data
todo-list.db
todo-list.db-shm
todo-list.db-wal

```

This container uses the volumes from the original application container, todo1. That means the same volume will be mounted at the same location /data

The new container shares the volume from todo1, so it can see the data written by the original application container

圖6.7運行具有專用和共享量的容器

容器之間共享量很簡單，但這可能不是您想做的。編寫數據的應用程序通常期望對文件的獨家訪問，並且如果另一個容器同時讀取和寫入同一文件，它們可能無法正常工作（或根本）。卷可以更好地用於保留應用升級之間的狀態，然後最好明確管理這些量。 您可以創建一個命名卷並將其附加到應用程序容器的不同版本。

現在嘗試

創建一個卷並將其用於待辦事項應用程序1的容器中。然後在UI中添加一些數據，然後將應用程序升級到版本2。容器需要匹配操作系統的文件系統路徑，因此我使用變量使復制和粘貼更容易：

```
# 將目標文件路徑保存在變量：linux容器$ target ='c
: \ data' # for Windows容器中

# 創建卷以存儲數據：Docker卷創建待辦事
項列表

# 運行V1應用程序，使用卷用於應用程序存儲：docker容器運行-d -p 8011 : 8080 -v todo -list
: $ target -name todo -v1
diamol/ch06-todo-list : 2e

# 在http://localhost:8011添加一些數據

# 刪除V1應用程序容器：Docker Cont
ainer RM -F TODO -V1

# 並使用相同卷的存儲卷運行V2容器：Docker容器運行-D -P 8011 : 8080 -V todo -list : $ targ
et -name todo -v2
DIAMOL/CH06-TODO-LIST : 2E-V2
```

圖6.8中的輸出顯示該卷具有其自己的生命週期。它存在於創建任何容器之前，並且將使用它的容器刪除時，它仍然存在。 該應用程序保留了升級之間的數據，因為新容器使用與舊容器相同的捲。

```

Create a new named volume. This is just an empty storage unit right now
PS>$target='/data'
Run a container using the -v flag to specify a volume - this will
mount the new empty volume to the path /data in the container
PS>
PS>docker volume create todo-list
todo-list
PS>
PS>docker container run -d -p 8011:8080 -v todo-list:$target --name todo-v1
diamol/ch06-todo-list:2e
a0566564c6d778c2951adfece505c34dc2d658767c6b627f5e1f0ab5dc4ccb3a
PS>
PS># add some data using the web app at http://localhost:8011
PS>
PS>docker container rm -f todo-v1
todo-v1
PS>
PS>docker container run -d -p 8011:8080 -v todo-list:$target --name todo-v2
diamol/ch06-todo-list:2e-v2
db9088ab321b8752855cea92d554bbb359ef5e6d4439be846ec11ae2c6b51416
PS>

Remove the container. The force flag -f removes it even
though the app is still running. This will delete the container's
writeable layer, but it will leave the volume intact
Start a new container from an updated version of the application image, mounting the same
volume to the same location. The v2 app will start with all the data written by the v1 app

```

圖6.8創建一個命名卷並使用它在容器更新之間持久數據

現在，當您瀏覽http://localhost:8011時，您會看到待辦事項應用程序的版本2，該版本已從昂貴的創意代理商中進行了UI改造（您可能需要在沒有緩存的情況下強制刷新，通常是CTRL-F5或CMD-SHIFT-SHIFT-R）。圖6.9顯示這已經準備好生產了。

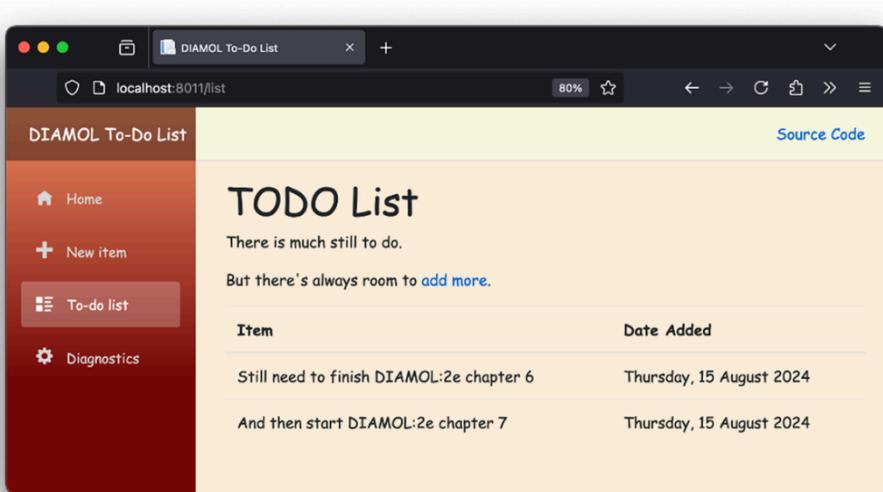


圖6.9全新的待辦事項應用UI

在我們繼續前進之前，關於Docker的數量有一件事。運行容器的Dockerfile和卷（或V）標誌中的音量指令是單獨的功能。如果運行命令中未指定音量，則使用音量指令構建的圖像將始終為容器創建卷。該卷將具有隨機的ID，因此您可以在容器消失後使用它，但是只有可以確定哪個卷具有數據。

無論圖像是否指定音量，音量標誌都將音量安裝到容器中。如果圖像確實具有捲，則音量標誌可以通過使用同一目標路徑的現有捲來覆蓋容器，因此不會創建新的捲。這就是待辦事項列表容器發生的事情。

您可以使用完全相同的語法，並為圖像中未指定音量的容器獲得相同的結果。作為圖像作者，您應該將捲指令用作狀態應用程序的故障安全選項。這樣，即使用戶未指定音量標誌，容器也將始終將數據寫入持續的捲。但是，作為圖像用戶，最好不要依靠默認值並使用命名卷。

6.3運行具有文件系統安裝的容器

卷對於分離存儲的生命週期很有用，並且仍然讓Docker為您管理所有資源。主機上存在卷，因此將它們與容器分離。Docker還提供了一種更直接的方法，可以使用*bind mounts*在容器和主機之間共享存儲。綁定的安裝座在主機上可作為容器內的路徑可用。綁定安裝座對容器是透明的，它只是容器文件系統的一部分的目錄。但這意味著您可以從容器中訪問主機文件，反之亦然，這解鎖了一些有趣的模式。

綁定安裝座，可讓您明確使用主機計算機上的文件系統進行容器數據。這可能是一個快速的固態磁盤，高度可用的磁盤，甚至是整個網絡中可訪問的分佈式存儲系統。如果您可以在主機上訪問該文件系統，則可以將其用於容器。我可以擁有一個帶有RAID數組的服務器，並將其用作我的待辦事項列表應用程序數據庫的可靠存儲。

現在嘗試

我確實確實有一個帶有RAID數組的服務器，但是您可能沒有，因此我們將在主機計算機上創建本地目錄並將其綁定到容器中。同樣，文件系統路徑需要匹配主機操作系統，因此我聲明了計算機上的源路徑和容器的目標路徑。注意Windows Server容器和Linux容器的不同：

```
# for Windows Server containers:  
$source="$(pwd)\databases".ToLower()  
$target="c:\data"  
  
# OR for Linux containers  
source="$(pwd)/databases"  
target='/data'  
  
mkdir ./databases  
  
docker container run --mount type=bind,source=$source,target=$target -d -p  
8012:8080 diamol/ch06-todo-list:2e  
  
curl -s http://localhost:8012  
  
ls ./databases
```

此練習使用curl命令（在Mac系統上的Linux上，在Windows上作為Curl.exe上）向待辦事項應用程序提出HTTP請求。這導致應用程序啟動，從而創建數據庫文件。最終命令列出了主機上本地數據庫目錄的內容，這將表明該應用程序的數據庫文件實際上在主機計算機上，如圖6.10所示。

Saving the source and target values for directory paths in variables makes the commands easier to work with

```

Run a container with a bind mount. Inside the container the app uses the directory /data, but this is a mount from the host directory

PS>$source=$(pwd)/databases
PS>$target=/data
PS>
PS>mkdir ./databases
PS>
PS>docker container run --mount type=bind,source=$source,target=$target -d -p 8012:8080 diamol/ch06-todo-list:2e
eb15114f2b2c2b10d48a42fa33e4d0a8126b0799bad34dca75965336bc407193
PS>
PS>curl -s http://localhost:8012 | Out-Null
PS>
PS>ls ./databases
todo-list.db          todo-list.db-shm        todo-list.db-wal
PS>

```

The container port 80 is published to port 8012 on the host. Sending an HTTP request to the container with curl starts the application, which creates the database file at /data - the Out-Null option silences the output of the command

These are the files created by the container. You can work with them directly from the host, and if you add files to the host directory they can be seen by the container

圖6.10使用綁定安裝座在容器上共享主機上的目錄

結合安裝是雙向的。您可以在容器中創建文件，並在主機上編輯它們，或在主機上創建文件並在容器中編輯它們。這裡有一個安全方面，因為容器通常應作為最小特權帳戶運行，以最大程度地減少攻擊者利用系統的風險。但是容器需要提高的權限才能在主機上讀取文件，因此該圖像是使用Dockerfile中的用戶指令構建的，以賦予容器管理權限 - 它使用Linux中的內置root用戶和Windows中的ContainerAdministrator用戶。

如果您不需要編寫文件，則可以在容器內部將主機目錄綁定為單位。這是從主機瀏覽器配置設置到應用程序容器中的一個選項。待辦事項應用程序圖像包裝使用默認配置文件，該文件將應用程序的日誌記錄級別設置為最低量。您可以從同一圖像運行容器，但可以將本地配置目錄安裝到容器中，並在不更改圖像的情況下覆蓋應用程序的配置。

現在嘗試

如果存在，則待辦事項應用程序將加載 /app/config路徑的額外配置文件。運行一個容器，該容器將安裝本地目錄綁定到該位置，該應用將使用主機的配置文件。首先導航到Diamol源代碼的本地副本，然後運行以下命令：

```

cd ./ch06/exercises/todo-list

# save the source path as a variable - on Windows:
$source="$(pwd)\config"; $target="c:\app\config"

# OR on Linux
source="$(pwd)/config" && target='/app/config'

# run the container using the mount:
docker container run --name todo-configured -d -p 8013:8080 --mount
type=bind,source=$source,target=$target,readonly diamol/ch06-todo-list:2e

# check the application:
curl -s http://localhost:8013

# and the container logs:
docker container logs todo-configured

```

主機目錄中的配置文件設置為使用更詳細的日誌記錄。當容器啟動時，它會映射該目錄，並且應用程序會看到配置文件並加載日誌記錄配置。在圖6.11所示的最終輸出中，有很多調試日誌線，應用程序不會使用標準配置編寫。

The source folder for the bind mount contains a configuration file set for additional logging.
The app in the container will read that config file from the mount and apply the settings

```

PS>cd ./ch06/exercises/todo-list
PS>
PS>$source="$(pwd)/config"; $target='/app/config'
PS>
PS>docker container run --name todo-configured -d -p 8013:8080 --mount type=bind,source=
$source,target=$target,readonly diamol/ch06-todo-list:2e
7d30eaaa0c448450c142c8a235f95eabd73418a90cd2c357d22a0da1d45d502a
PS>
PS>curl -s http://localhost:8013 | Out-Null
PS>
PS>docker container logs todo-configured
dbug: Microsoft.Extensions.Hosting.Internal.Host[1]
      Hosting starting
warn: Microsoft.AspNetCore.DataProtection.Repositories.FileSystemXmlRepository[60]

```

The application now creates hundreds of log entries which can be seen with the `container logs` command. The standard configuration baked into the image doesn't show these

圖6.11使用綁定安裝來將僅讀取的配置文件加載到容器中

您可以綁定主機計算機可以訪問的任何來源。 您可以使用Linux主機上安裝在 /MNT /NFS 上的共享網絡驅動器，也可以使用Windows主機上的X：驅動器映射。這些都可以是綁定安裝的來源，並以相同的方式浮出水面。這是獲取可靠甚至分佈式存儲的一種有用方法，用於在容器中運行的狀態應用程序，但是您需要了解一些局限性。

6.4知道文件系統安裝的局限性

要有效地使用綁定和卷，您需要了解一些關鍵方案和局限性，其中有些是微妙的，只會出現在容器和文件系統的異常組合中。

第一個方案很簡單：當您運行帶有安裝座的容器和安裝目標目錄的容器時會發生什麼情況，並且來自圖像層的文件會發生什麼？您可能會認為Docker會將源合併為目標。在容器內部，您希望該目錄具有來自圖像的所有現有文件以及MOUNT中的所有新文件。但事實並非如此。當您安裝已經具有數據的目標時，源目錄`replaces`目標目錄 - 因此，來自圖像的原始文件不可用。

您可以使用一個簡單的練習來看到此圖像，該圖像在運行時列出了目錄內容。 Linux和Windows容器的行為相同，但是命令中的文件系統路徑需要匹配操作系統。

現在嘗試

在沒有安裝座的情況下運行容器，它將列出圖像中的目錄內容。使用安裝座再次運行它，它將列出源目錄的內容（這裡再次有變量以支持Windows和Linux）：

```
cd ./ch06/cercises/bind-mount
```

```
# 設置路徑 - 適用於Windows Server容器：$ source = “ $(pwd  
 ) \ new; $ target =” c :\ init” # or for linux容器：
```

```
源= “ $(pwd) /new” && target ='/init'
```

```
Docker Container Run Diamol/CH06綁定安裝：2E
```

```
docker容器運行 - 安裝類型= bind , source = $ source , target = $ target diamol/ch06- bind-mount : 2e
```

您會看到，在第一個運行容器中，列出了兩個文件：abc.txt和def.txt。這些從圖像層加載到容器中。第二個容器將目標目錄替換為MOUD的源，因此未列出這些文件。僅顯示文件123.txt和456.txt，這些文件來自主機上的源目錄。圖6.12顯示了我的輸出。

```
When this container runs it lists the contents of the /init directory. Without a bind mount it shows the files which already exist from the Docker image

PS>cd ./ch06/exercises/bind-mount
PS>
PS>$source="$(pwd)/new"
PS>$target='/init'
PS>
PS>docker container run diamol/ch06-bind-mount:2e
abc.txt
def.txt
PS>
PS>docker container run --mount type=bind,source=$source,target=$target
diamol/ch06-bind-mount:2e
123.txt
456.txt
PS>

Run a container with a bind mount targeting /init, and the original directory contents are hidden. The source directory for the mount replaces the target, so only the source directory files are shown
```

圖6.12綁定安裝目錄陰影目標目錄（如果Exi）

sts。

第二種情況是對此的一種變化：如果將單個文件從主機掛載到容器文件系統中存在的目標目錄會發生什麼？這次，目錄內容~~are~~合併，因此您將從圖像中看到原始文件和主機的新文件，除非您正在運行Windows Server容器，否則根本不支持此功能。

容器文件系統是Windows容器與Linux容器不同的少數區域之一。有些事情確實以相同的方式工作。您可以使用DockerFiles內部的標準Linux風格路徑，因此 /數據適用於Windows容器，並成為C:\Data的別名。但這對於音量安裝和綁定安裝座不起作用，這就是為什麼本章中的練習使用變量為Linux用戶 /數據和Windows C:\Data提供的原因。

單文件安裝座的限制更加明確。如果您有可用的Windows Server和Linux機器，或者您在Windows上運行Docker Desktop，則可以自己嘗試此操作，該桌面支持Linux和Windows Server容器。

現在嘗試

Linux容器和Windows Server容器上的單文件安裝座的行為不同。如果您有Linux和Windows Server，則可以在行動中看到它：

```
cd ./ch06/cercises/bind-mount
```

```
# 帶有Linux容器：Docker容器運行-Mount類型= bind，source = “ $ (pwd) /new/123.txt” ，target = /Init/123.txt diamol/ch06-bind- mount : 2e
```

```
# 和Windows Server容器：docker容器運行 - 安裝類型= bind，source = “ $ (pwd) /new/123.txt” ，target = c :\ init \ init \ init \ 123.txt diamol/ch06-bind- mount : 2e : 2e : 2e
```

Docker映像是相同的，並且命令是相同的 - 從目標的特定文件系統路徑遵循。但是，當您運行此操作時，您會看到Linux示例可以按預期工作，但是您會在Windows上的Docker出現錯誤，如圖6.13所示。

You can bind-mount a single file with Linux containers. When the target directory already exists, the contents are merged with the bind mount.

```
PS> cd ./ch06/exercises/bind-mount  
PS>  
PS> docker container run --mount type=bind,source="$(pwd)/new/123.txt",  
target=/init/123.txt diamol/ch06-bind-mount  
123.txt  
abc.txt  
def.txt  
PS>  
PS> # switch to Windows containers  
PS>  
PS> docker container run --mount type=bind,source="$(pwd)/new/123.txt",  
target=C:\init\123.txt diamol/ch06-bind-mount  
C:\Program Files\Docker\Docker\Resources\bin\docker.exe: Error response  
from daemon: invalid mount config for type "bind": source path must be  
a directory.  
See C:\Program Files\Docker\Docker\Resources\bin\docker.exe run --help
```

You can't bind-mount a single file with Windows containers. The feature isn't supported, and you'll get an invalid mount config error, telling you the source for the mount must be a directory.

圖6.13用單個文件將安裝綁定為Linux上的源，但在Windows上不工作。

第三種情況不太常見。在不設置大量移動作品的情況下，很難複制，因此不會進行練習來掩蓋這一點 - 您必須相信我的諾言。場景是，如果將分佈式文件系統綁定到容器中，會發生什麼？容器中的應用程序仍然可以正常工作嗎？看，即使問題也很複雜。

分佈式文件系統可讓您從網絡上的任何機器訪問數據，並且通常使用操作系統本地文件系統中的不同存儲機制。它可能是您本地網絡，Azure文件或云中AWS S3上共享的SMB或NFS文件之類的技術。您可以將類似分佈式存儲系統的位置安裝到容器中。安裝座看起來像文件系統的正常部分，但是如果不能支持相同的操作，您的應用可能會失敗。

圖6.14中有一個具體的示例，該示例試圖使用用於容器存儲的Azure文件在雲上的容器中運行Postgres數據庫系統。Azure文件支持諸如讀寫之類的普通文件系統操作，但它不支持應用程序可能使用的一些更為不尋常的操作。在這種情況下，Postgres容器試圖創建一個文件鏈接，但是Azure文件不支持該功能，因此應用程序崩潰了。

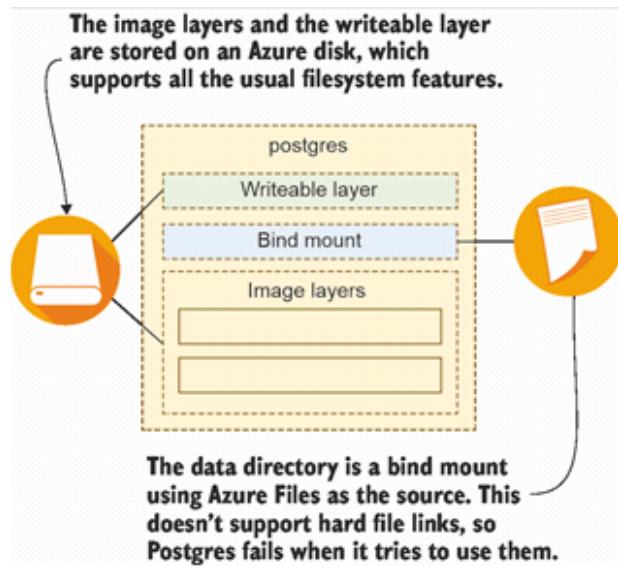


圖6.14分佈式存儲系統可能無法提供所有常用的文件系統功能。

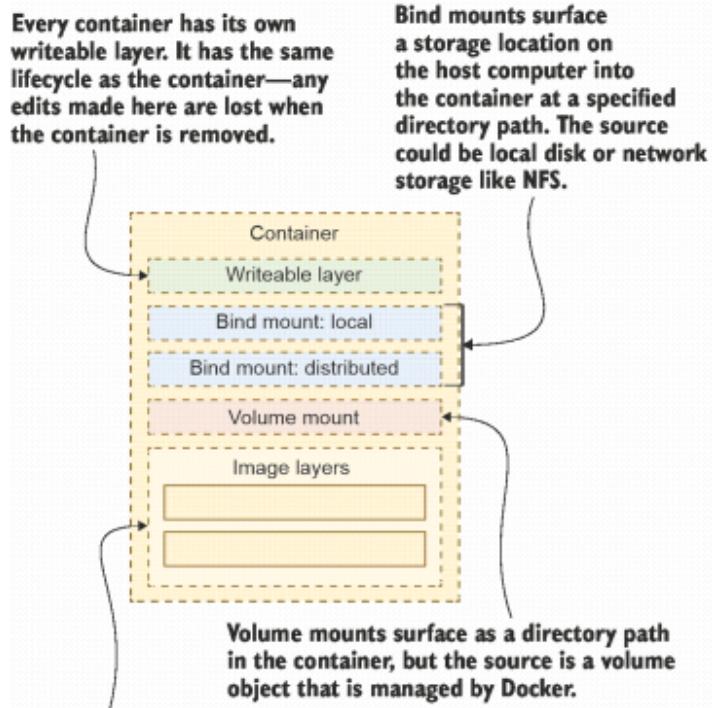
這種情況是一個異常值，但是您需要意識到這一點，因為如果發生這種情況，那真的沒有辦法。綁定安裝座的源可能不支持該應用程序在您的容器期望的所有文件系統功能。這是您無法計劃的 - 直到您使用存儲系統嘗試應用程序之前，您才知道。如果您想將分佈式存儲用於容器，則應意識到此風險，並且還需要了解，分佈式存儲與本地存儲的性能特徵將非常不同。如果您在帶有分佈式存儲的容器中運行每個文件，則使用大量磁盤的應用程序可能會磨碎。

6.5了解如何構建容器文件系統

我們在本章中介紹了很多。存儲是一個重要的主題，因為容器的選項與物理計算機或虛擬機上的存儲非常不同。我將結束整合我們所涵蓋的所有內容，並提供一些使用容器文件系統的最佳實踐指南。

每個容器都有一個單個磁盤，這是一個虛擬磁盤，可將幾個來源劃分在一起。Docker稱其為 *union filesystem*。我不會看Docker如何實現聯合文件系統，因為對於不同的操作系統有不同的技術。安裝Docker時，它為您的操作系統做出了正確的選擇，因此您無需擔心細節。

聯合文件系統使容器可以看到一個單個磁盤驅動器，並以相同方式使用文件和目錄，無論它們在磁盤上的位置。但是，如圖6.15所示，磁盤上的位置可以物理存儲在不同的存儲單元中。



The initial state of the container is provided by the image layers. Any files built into the image from the Dockerfile are here. The layers are read-only, but the container can edit files—Docker uses copy-on-write for this.

圖6.15容器文件系統是從多個來源的聯合創建的。

容器中的應用程序可查看單個磁盤，但是作為圖像作者或容器用戶，您可以選擇該磁盤的來源。可以有多個圖像層，多個音量安裝座和容器中的多個綁定安裝座，它們將始終具有一個可寫的層。以下是一些有關如何使用存儲選項的一般準則：

- **Writeable layer** - 完美用於短期存儲，例如緩存數據到磁盤以節省網絡呼叫或計算。這些是每個容器所獨有的，但是當卸下容器時，它們永遠消失了。
- **Local bind mounts** - 用於在主機和容器之間共享數據。開發人員可以使用綁定安裝座從計算機加載源代碼到容器中，因此，當他們對HTML或JavaScript文件進行本地編輯時，更改就會立即在容器中，而無需構建新圖像。

- *Distributed bind mounts* - 用於在網絡存儲和容器之間共享數據。這些很有用，但是您需要意識到網絡存儲將與本地磁盤具有相同的性能，並且可能無法提供完整的文件系統功能。它們可以用作配置數據或共享緩存的僅讀取源，也可以用作可讀取的數據來存儲可以在同一網絡上的任何計算機上使用的任何容器都可以使用的數據。
- *Volume mounts* - 用於在容器和由Docker管理的存儲對象之間共享數據。這些對於持久存儲很有用，該應用程序將數據寫入卷。當您使用新容器升級應用程序時，它將保留以前版本寫入卷的數據。
- *Image layers* - 這些呈現容器的初始文件系統。堆疊層，最新的一層覆蓋了較早的層，因此可以用寫入同一路徑的後續圖層覆蓋在Dockfile開頭的一層文件。圖層是只讀的，並且可以在容器之間共享。

•

6.6 實驗室

我們將將這些作品放在這個實驗室中。它又回到了一個舊的待辦事項清單應用程序，但這一次有了扭曲。該應用程序將在容器中運行，並從已經創建的一組任務開始。您的工作是使用相同的圖像運行應用程序，但具有不同的存儲選項，以便待辦事項列表開始空白，當您保存項目時，它們會存儲到Docker卷中。本章的練習應該使您到達那裡，但這裡有一些提示：

- 請記住，它是Docker RM -F \$（Docker PS -AQ）以刪除所有現有的容器。首先從DIAMOL/CH06-LAB：2E運行該應用程序以檢查任務。然後，您需要使用一些安裝座從同一圖像運行一個容器。該應用程序使用配置文件 - 其中有更多的設置對日誌級別。
-
-

如果您需要的話，我的示例解決方案是在書的GitHub存儲庫中，但是您應該嘗試通過此方法來處理此問題，因為如果您沒有太多的經驗，容器存儲會絆倒您。有幾種解決這個問題的方法。我的解決方案在這裡：<https://github.com/sixeyed/diamol/blob/2e/ch06/lab/readme.md>。

7 使用Docker組成

大多數應用程序不會在一個組件中運行。即使是大型舊應用程序也通常是作為前端和後端組件構建的，它們是在物理分佈的組件中運行的單獨邏輯層。 Docker非常適合運行分佈式應用程序 - 從n - Tier整體到現代微服務。 每個組件都以自己的輕質容器運行，並使用標準網絡協議將它們插入在一起。您可以使用Docker Compose來定義和管理這樣的多容器應用程序。

COMPOSE是用於描述分佈式Docker應用程序的文件格式，它是管理它們的工具。 讓我們重新審視書中早期的一些應用程序，看看Docker構成如何更容易使用它們。

7.1 Docker組成文件的解剖結構

您已經與許多Dockerfiles合作，並且您知道Dockerfile是包裝應用程序的腳本。但是對於分佈式應用程序，DockerFile實際上僅用於包裝應用程序的一部分。 對於具有前端網站，後端API和數據庫的應用程序，您可以擁有三個Dockerfiles，每個組件一個。您將如何在容器中運行該應用程序？

您可以使用Docker CLI依次啟動每個容器，並指定應用程序正確運行的所有選項。 這是一個可能成為故障點的手動過程，因為如果您弄錯了任何選項，則應用程序可能無法正常工作，或者容器可能無法通信。 相反，您可以使用Docker組成的文件來描述應用程序的結構。

Docker組成的文件描述了您的應用程序的*desired state* - 當所有內容運行時都應該看起來像什麼。這是一個簡單的文件格式，您將所有選項放入Docker Container Run命令中的所有選項中。然後，您使用Docker撰寫工具運行該應用程序。它可以算出所需的Docker資源，其中可能是容器，網絡或卷，並將請求發送到Docker API以創建它們。

清單7.1顯示了一個完整的Docker組成的文件 - 您將在本書的源代碼中的本章的練習文件夾中找到此文件。

列表7.1碼頭組合文件以從第6章運行待辦事項應用程序

```
服務：Todo-Web：圖片：Diamol/ch06-todo-list：2E端口：- “8020：80” 網絡：- 應用  
程序網絡網絡：App-netnets：app-nets：app-  
net：name：name：nat ofternal：true
```

該文件描述了一個簡單的應用程序，其中一個docker容器插入一個docker網絡。Docker撰寫使用YAML，這是一種可讀的文本格式，被廣泛使用，因為它可以輕鬆地轉化為JSON（這是API的標準語言）。空間在YAML中很重要 - 註明識別對象和對象的子屬性。

在此示例中，有兩個頂級語句：

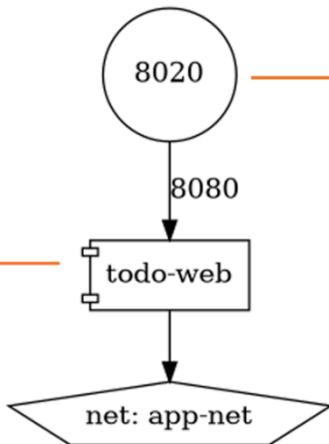
- 服務列出了組成應用程序的所有組件。Docker構成使用*services*而不是實際容器的想法，因為可以使用來自同一圖像的幾個容器進行大規模運行服務。 網絡列出了服務容器可以插入的所有Docker網絡。

•

您可以使用Compose運行此應用程序，它將啟動一個容器以達到所需的狀態。圖7.1顯示了應用程序資源的架構圖。

There is one service here called `todo-web`, which Docker Compose will run as a single container

The container publishes one port.
The port on the host is 8020,
publishing port 8080 on the container



The container is connected to a Docker network,
called `app-net` in the Compose specification

圖7.1簡單組合應用程序的體系結構帶有一個服務和一個網絡

在我們實際運行此應用程序之前，有幾件事需要更仔細地查看。稱為Todo-Web的服務將從Diamol/CH06-Todo-List：2E圖像中運行一個容器。它將在主機上發佈到容器上的端口80上的端口8020，並將容器連接到撰寫文件中所謂的App-net的docker網絡。最終結果將與運行Docker容器運行-P 8020 : 8080 -NAME TODO-WEB - 網絡NAT DIAMOL/CH06-TODO-LIST : 2E。

服務名稱下的屬性是屬性，它們是與Docker Container Run命令中選項的相當接近的地圖：圖像是要運行的圖像，端口是要發布的端口，網絡是要連接到的網絡。 服務名稱成為容器名稱和容器的DNS名稱，其他容器可以用來在Docker網絡上連接它們。服務中的網絡名稱是App-Net，但在網絡部分下，該網絡被指定為映射到稱為NAT的外部網絡。外部選項意味著組成期望NAT網絡已經存在，並且不會嘗試創建它。

您可以使用Docker組合使用Docker Compose Line來管理應用程序，該命令曾經與Docker CLI分開，但現在是內置的。 Docker組成的命令使用略有不同的術語，因此您可以使用UP命令啟動一個應用程序，該命令告訴Docker組成檢查撰寫文件並創建將應用程序提升到所需狀態所需的任何內容。

現在嘗試

打開終端並創建Docker網絡。然後使用清單7.1的撰寫文件瀏覽到文件夾，然後使用Docker-Compose命令行運行該應用：

```
# 忽略此命令中的任何錯誤：Docker網絡創建N  
AT
```

```
cd ./ch07/cercises/todo-list
```

Docker組成

您並不總是需要創建一個用於撰寫應用程序的Docker網絡，並且您可能已經在第4章中運行練習中的NAT網絡，在這種情況下，您將獲得一個可以忽略的錯誤。如果您使用Linux容器，Compose可以為您管理網絡，但是如果您使用Windows Server容器，則需要使用Docker在Windows上安裝時創建的名為NAT的默認網絡。我正在使用NAT網絡，因此無論您正在運行Linux還是Windows Server容器，相同的組合文件都適用於您。

Compose命令行期望在當前目錄中找到一個名為Docker-compose.yml的文件，因此在這種情況下，它加載了待辦事項列表應用程序定義。您將沒有任何匹配托多 - 章布服務所需狀態的容器，因此組合將啟動一個容器。當撰寫運行容器時，它會收集所有應用程序日誌並顯示由容器分組，這對於開發和測試非常有用。

我從運行上一個命令中輸出的輸出是在圖7.2中 - 當您自己運行時，您還會看到從Docker Hub拿起的圖像，但是在運行命令之前，我已經將它們拉到。

Networks defined as `external` in Compose files need to exist before you can run the app. This would create the `nat` network, except I've already created it

This starts the app. Docker Compose compares the resources in the Compose file with the resources currently running in Docker, and takes actions to get to the desired state

```

PS> docker network create nat
Error response from daemon: network with name nat already exists
PS>
PS> cd ./ch07/exercises/todo-list
PS>
PS> docker compose up
[+] Running 2/0
  ✓ Synchronized File Shares
  ✓ Container todo-list-todo-web-1  Creat...
                                                0.0s
                                                0.0s
Attaching to todo-web-1
todo-web-1 | warn: Microsoft.AspNetCore.DataProtection.Repositories.FileSystemXml
Repository[60]
todo-web-1 |      Storing keys in a directory '/root/.aspnet/DataProtection-Keys'
' that may not be persisted outside of the container. Protected data will be unava
ilable when container is destroyed. For more information go to https://aka.ms/aspn
et/dataprotectionwarning
todo-web-1 | info: Microsoft.Hosting.Lifetime[14]
todo-web-1 |      Now listening on: http://[::]:8080

```

Compose creates a single container for this app. It attaches to the standard out for that container and shows all the app logs. These are the startup logs for ASP.NET Core to-do app

圖7.2使用Docker Compose啟動應用程序，該應用創建Docker Resources

現在，您可以瀏覽到`http://localhost:8020`，並查看待辦事項列表應用程序。它以與第6章完全相同的方式工作，但是Docker組成為您提供了一種更強大的啟動應用方式。Docker組合文件將與應用程序和DockerFiles的代碼一起使用，並成為描述應用程序所有運行時屬性的單個地方。您無需記錄README文件中的圖像名稱或已發布的端口，因為它們都在撰寫文件中。

Docker組成格式記錄您需要配置應用程序所需的所有屬性，並且還可以記錄其他頂級Docker資源（例如捲和秘密）。該應用程序只有一個服務，即使在這種情況下，也可以使用一個可以用來運行該應用程序並記錄其設置的組合文件。但是，當您運行多包裝應用程序時，撰寫確實很有意義。

7.2與Compose一起運行多包裝應用程序

回到第4章中，我們構建了一個分佈式應用程序，該應用顯示了NASA的《Day API》中天文學圖片中的圖像。有一個Java前端網站，一個用GO編寫的REST API，以及用Node.js編寫的日誌收集器。我們通過依次啟動每個容器來運行應用程序，我們必須將容器插入同一網絡並使用正確的容器名稱，以便組件可以彼此找到。這正是Docker為我們撰寫的一種脆弱方法。

在列表7.2中，您可以看到描述圖庫應用程序的撰寫文件的服務部分。我已經刪除了網絡配置以關注服務屬性，但是就像在待辦事項應用程序示例中一樣，服務插入了NAT網絡。

```
列出7.2多容器圖庫應用程序的撰寫服務

accesslog:
  image: diamol/ch04-access-log:2e

iotd:
  image: diamol/ch04-image-of-the-day:2e
  ports:
    - "80"

image-gallery:
  image: diamol/ch04-image-gallery:2e
  ports:
    - "8010:80"
  depends_on:
    - accesslog
    - iotd
```

這是如何配置不同類型的服務的一個很好的例子。AccessLog Service不會發布任何端口或使用您將從Docker Container Run命令捕獲的任何其他屬性，因此記錄的唯一值是圖像名稱。IOTD服務是REST API - 撰寫文件記錄了圖像名稱，還將容器上的端口80發佈到主機上的隨機端口。圖像 - 飾品服務具有圖像名稱和一個特定已發布的端口：8010在容器中的端口80上的8010。它還具有一個依賴性_on部分，說此服務對其他兩個服務有依賴性，因此組合應確保在啟動此服務之前運行這些服務。

圖7.3顯示了此應用程序的體系結構。我已經從讀取撰寫文件並生成體系結構的PNG圖像的工具中生成了本章中的圖表。這是使文檔保持最新狀態的好方法 - 每次更改時，您都可以從撰寫文件中生成圖表。當然，該圖工具在Docker容器中運行 - 您可以在<https://github.com/pmsipilot/docker-compose-viz>上找到它。

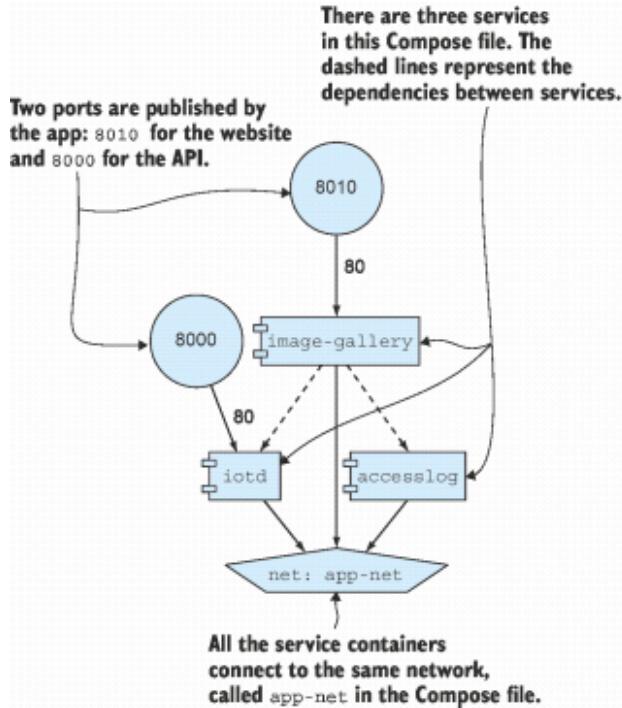


圖7.3 a mo RE複合物撰寫文件，指定連接到S的三個服務

AME網絡

我們將使用Docker組成來運行該應用程序，但是這次我們將在*detached mode*中運行。Compose仍然會為我們收集日誌，但是容器將在後台運行，因此我們將返回終端會話，我們可以使用更多的作曲功能。

現在嘗試

打開Diamol源代碼根的終端會話，然後導航到Image Gallery文件夾並運行應用程序：

```
CD ./ch07/cercises/image-of-the-day
```

```
Docker構成-Detach
```

您的輸出將像我的圖7.4中一樣。 您可以看到，由於撰寫文件中記錄的依賴項，因此訪問和IOTD服務是在圖像 - 飾面服務之前啟動的。

```
PS> cd ./ch07/exercises/image-of-the-day
PS>
PS> docker compose up --detach
[+] Running 4/4
  ✓ Synchronized File Shares
  ✓ Container image-of-the-day-iotd-1          Started
  ✓ Container image-of-the-day-accesslog-1       Started
  ✓ Container image-of-the-day-image-gallery-1   Started
```

Compose creates three containers - respecting the dependencies, so image-gallery is started after the accesslog and iotd services are running. --detach runs the containers in the background, so the logs aren't shown in the terminal session.

圖7.4 使用使用Docker指定的依賴項啟動多容器應用程序

當應用程序運行時，您可以瀏覽到`http://localhost:8010`。它的工作方式就像在第4章中一樣，但是現在您在Docker中有一個明確的定義，其中構成瞭如何將容器配置以使它們一起工作的文件。您還可以使用組合文件整體管理應用程序。API服務有效地無狀態，因此您可以將其擴展為在多個容器上運行。當Web容器請求來自API的數據時，Docker將在運行的API容器上共享這些請求。

現在嘗試

在同一終端會話中，使用Docker組合來增加IOTD服務的規模，然後刷新網頁幾次，然後檢查IOTD容器的日誌：

```
docker組成-d -scale iotd = 3

# 瀏覽到http://localhost:8010和刷新

Docker撰寫日誌 - -tail = 1 IOTD
```

您會在組合的輸出中看到兩個新的容器來運行Image API服務，因此它的比例為三個。當您刷新顯示照片的網頁時，Web應用程序請求來自API的數據，並且該請求可以由任何API容器處理。API處理請求時，將編寫日誌條目，您可以在容器日誌中看到。Docker組合可以向您顯示所有容器的所有日誌條目，也可以使用它來過濾輸出`--tail=1`參數僅從每個IOTD服務容器中獲取最後一個日誌條目。

我的輸出是在圖7.5中 - 您可以看到Web應用程序已使用了1和3的容器，但是到目前為止，容器2尚未處理任何請求。

The web app uses the `iotd` API containers, so sending traffic to the website will make requests to the `iotd` containers

```
PS> docker compose up -d --scale iotd=3
[+] Running 6/6
  ✓ Synchronized File Shares
  ✓ Container image-of-the-day-accesslog-1   Running
  ✓ Container image-of-the-day-iotd-1         Running
  ✓ Container image-of-the-day-iotd-3         Started
  ✓ Container image-of-the-day-iotd-2         Started
  ✓ Container image-of-the-day-image-gallery-1 Running
PS>
PS> # browse to http://localhost:8010 and refresh
PS>
PS> docker compose logs --tail=1 iotd
Scaling up the iotd service creates two new containers.
Along with the original container this gives the scale of three

iotd-3 | 2024-08-20 20:04:10.699 INFO 1 --- [           main] iotd.Application      : Started Application in 1.643 seconds
M running for 1.841)
iotd-1 | 2024-08-20 20:03:12.141 INFO 1 --- [p-nio-80-exec-1] iotd.ImageController : Fetched new APOD images from NASA
iotd-2 | 2024-08-20 20:04:17.601 INFO 1 --- [p-nio-80-exec-1] iotd.ImageController : Fetched new APOD images from NASA
PS>
```

This shows the most recent log entry from each of the `iotd` containers. You can see that `iotd_1` and `iotd_2` have received requests from the web app and fetched image data; `iotd_3` hasn't yet handled any requests

圖7 .5 擴展應用程序組件並使用Docker檢查其日誌

撰寫

Docker Compose現在正在為我管理五個容器。我可以使用Compose控制整個應用程序；我可以停止所有容器來節省計算資源，並在需要運行應用程序時再次啟動它們。但是，這些是普通的Docker容器，我也可以使用Docker CLI。COMPOSE是用於管理容器的單獨的命令行工具，但它使用Docker API的方式與Docker CLI一樣。您可以使用Compose來管理您的應用程序，但仍然使用標準Docker CLI與創建的組合的容器一起使用。

現在嘗試

在同一終端會話中，停止使用Docker組合命令啟動應用程序，然後用Docker CLI列出所有運行容器：

Docker構成停止

Docker構成開始

Docker容器LS

您的輸出將像我的圖7.6中一樣。 您會看到，當單個容器停止並再次啟動應用程序時，您會看到該列表以正確的依賴順序啟動。在容器列表中，您會看到撰寫已重新啟動了現有容器，而不是創建新容器。我所有的集裝箱都表明它們是幾分鐘前創建的，但是它們只播放了幾秒鐘。

Stopping the app with Compose stops all the containers. Stopped containers don't use any CPU or memory, but their filesystems still exist

Starting the app again restarts those same containers

```

PS>docker compose stop
[+] Stopping 5/5
✓ Container image-of-the-day-image-gallery-1 Stopped
✓ Container image-of-the-day-accesslog-1 Stopped
✓ Container image-of-the-day-iotd-1 Stopped
✓ Container image-of-the-day-iotd-2 Stopped
✓ Container image-of-the-day-iotd-3 Stopped
PS>
PS>docker compose start
[+] Running 5/5
✓ Container image-of-the-day-iotd-3 Started 0.1s
✓ Container image-of-the-day-accesslog-1 Started 0.1s
✓ Container image-of-the-day-iotd-2 Started 0.1s
✓ Container image-of-the-day-iotd-1 Started 0.1s
✓ Container image-of-the-day-image-gallery-1 Started 0.2s
PS>
PS>docker container ls
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
88ef2d1733aa diamol/ch04-image-of-the-day:2e "java -jar /app/iotd..." 3 minutes ago Up 13
seconds 0.0.0.0:49561->80/tcp image-of-the-day-iotd-3
4ab5efbe245d diamol/ch04-image-of-the-day:2e "java -jar /app/iotd..." 3 minutes ago Up 13
seconds 0.0.0.0:49562->80/tcp image-of-the-day-iotd-2
cf0c18403c5b diamol/ch04-image-gallery:2e "/web/server" 3 minutes ago Up 13
seconds 0.0.0.0:8010->80/tcp image-of-the-day-image-gallery-1
6b15feeacc57 diamol/ch04-image-of-the-day:2e "java -jar /app/iotd..." 3 minutes ago Up 13
seconds 0.0.0.0:49563->80/tcp image-of-the-day-iotd-1
756f40d0d63b diamol/ch04-access-log:2e "docker-entrypoint.s..." 3 minutes ago Up 13
seconds 80/tcp image-of-the-day-accesslog-1

```

Listing running containers shows that the app containers have only been up for a few seconds, even though they were created 3 minutes ago

圖7.6使用Docker撰寫並啟動多容器應用程序

還有更多的功能可以構成 - 跑步的Docker構成，沒有任何選項可以查看命令的完整列表，但是在進行進一步之前，您需要考慮一個非常重要的考慮。Docker Compose是客戶端工具。這是一條命令行，將指令根據撰寫文件的內容髮送到Docker API。Docker本身只是運行容器。尚不知道許多容器代表一個應用程序。僅構成知道這一點，並且僅通過查看Docker組成YAML文件來知道您的應用程序的結構，因此您需要擁有該文件來管理您的應用程序。

如果組合文件更改或更新運行應用程序，則可以使您的應用程序與組合文件同步。當您返回使用Compose管理應用程序時，這可能會導致意外行為。我們已經自己完成了此操作 - 我們將IOTD服務擴展到了三個容器，但是該配置未在撰寫文件中捕獲。當您將應用程序降低然後重新創建時，Compose將其歸還原始量表。

現在嘗試

在同一終端會話中（因為組合需要使用相同的YAML文件）使用Docker撰寫以將應用程序降低並備份。然後通過列出運行容器來檢查比例：

```
Docker撰写Docker构成-D  
Docker容器LS
```

down命令刪除了應用程序，因此構圖停止並卸下容器 - 如果將它們記錄在撰寫文件中，並且不標記為外部，則將刪除網絡和卷。然後UP啟動該應用程序，並且由於沒有運行的容器，因此創建所有服務，但是它在Compose文件中使用了App定義，該文件未記錄量表，因此API服務從一個容器開始，而不是我們先前運行的三個容器。

您可以在圖7.7的輸出中看到這一點。這裡的目標是重新啟動該應用程序，但我們也意外地將API服務縮減為降低。

This stops the app and removes all the resources managed by Docker Compose. All the containers are removed, but the network is defined as external in the Compose file, so it's not managed by Compose and isn't removed

```

PS>docker compose down
[+] Running 5/5
  ✓ Container image-of-the-day-image-gallery-1 Removed
  ✓ Container image-of-the-day-iotd-1 Removed
  ✓ Container image-of-the-day-accesslog-1 Removed
  ✓ Container image-of-the-day-iotd-3 Removed
  ✓ Container image-of-the-day-iotd-2 Removed
PS>
PS>docker compose up -d
[+] Running 3/3
  ✓ Container image-of-the-day-iotd-1 Started
  ✓ Container image-of-the-day-accesslog-1 Started
  ✓ Container image-of-the-day-image-gallery-1 Started
PS>
PS>docker container ls
CONTAINER ID        IMAGE               COMMAND      CREATED          STATUS          PORTS
d5c542677650        diamol/ch04-image-gallery:2e   "/web/server"  10 seconds ago  Up 9
seconds             0.0.0.0:8010->80/tcp    image-of-the-day-image-gallery-1
be2a037fc5a3        diamol/ch04-access-log:2e   "docker-entrypoint.s..."  10 seconds ago  Up 9
seconds             80/tcp                  image-of-the-day-accesslog-1
f91b4b61e45         diamol/ch04-image-of-the-day:2e  "java -jar /app/iotd..."  10 seconds ago  Up 9
seconds             0.0.0.0:49576->80/tcp   image-of-the-day-iotd-1

```

This starts the app again, creating all the resources needed to get to the desired state in the Compose file

The desired state in the Compose file is for a single container in the iotd service. We previously scaled up to three, but that's not in the Compose file so now we've returned to a single container

圖7.7刪除和重新創建應用程序將其重置為Docker組成文件中的狀態。

Docker Compose易於使用和功能強大，但是您需要注意它是客戶端工具，因此它取決於對應用程序定義YAML文件的良好管理。當您使用Compose部署應用程序時，它會創建Docker Resources，但是Docker Engine不知道這些資源是相關的，只要您擁有撰寫文件來管理它們，它們只是應用程序。

7.3 Docker如何將容器插在一起

分佈式應用程序中的所有組件都包含在Docker容器中，但是它們如何相互通信？您知道容器是具有自己網絡空間的虛擬化環境。每個容器都有Docker分配的虛擬IP地址，並且插入同一Docker網絡的容器可以使用其IP地址互相到達。但是，在應用程序生命週期期間（您升級到新的容器映像或更改配置時）將更換容器，並且新容器將具有新的IP地址，因此Docker還支持DNS的服務發現。

DNS是域名系統，將名稱鏈接到IP地址。它可以在公共互聯網和私人網絡上工作。當您將瀏覽器指向blog.sixeyed.com時，您正在使用一個域名，該名稱已解決到我託管博客的一台服務器的IP地址。您的計算機實際上使用IP地址獲取內容，但是您作為用戶使用域名，這更友好。

Docker擁有自己的DNS服務。在容器中運行的應用程序在嘗試訪問其他組件時會製作域查找。 Docker中的DNS服務執行查找 - 如果域名實際上是一個容器名稱，Docker返回容器的IP地址，並且消費者可以直接在Docker Network上工作。如果域名不是容器，則Docker將請求傳遞到Docker正在運行的服務器，因此它將製作標準的DNS查找以在組織的網絡或公共Internet上找到IP地址。

您可以通過圖像 - 壁畫應用在行動中看到該概念。 Docker的DNS服務的響應將包含一個單個IP地址，用於在單個容器中運行的服務，或者如果該服務跨多個容器規模運行，則多個IP地址。

現在嘗試

在同一終端會話中，使用Docker組成，將應用程序以三個比例為三個。然後連接到Web容器中的會話 - 選擇要運行的Linux或Windows命令 - 並執行DNS查找：

```
docker組成-d -scale iotd = 3

#對於Linux容器：
Docker Container exec- iT日期圖像 - 圖像gallery-1 sh

#對於Windows容器：
Docker Container exec -IT圖像 - 圖像 - 圖像 - 助手-1 CMD

#現在在集裝箱會話中運行：NSlookup AccessLog
```

g

出口

NSlookUp是一個小型實用程序，是Web應用程序的基本圖像的一部分，它可以為您提供的名稱執行DNS查找，並打印出IP地址。我的輸出在圖7.8中 - 您可以看到服務器地址已列出 - DNS服務器本身，由Docker管理 - 然後是容器的IP地址。我的AccessLog容器具有IP地址172.18.0.2。

This brings the app back up to the desired state, but with a specified scale of three for the `iotd` service

This runs an interactive shell in the first web app container, and connects the local session to that shell

```
PS>docker compose up -d --scale iotd=3
[+] Running 5/5
✓ Container image-of-the-day-iotd-1
✓ Container image-of-the-day-iotd-3
✓ Container image-of-the-day-accesslog-1
✓ Container image-of-the-day-iotd-2
✓ Container image-of-the-day-image-gallery-1
PS>
PS>docker container exec -it image-of-the-day-image-gallery-1 sh
/web #
/web # nslookup accesslog
Server:      127.0.0.11
Address:     127.0.0.11:53
```

Runn...
Star...
Running
Star...
Running

Non-authoritative answer:

```
Non-authoritative answer:
Name:   accesslog
Address: 172.18.0.2
```

The container image has the `nslookup` tool, which performs DNS name lookups. Querying the name of the `accesslog` service returns the container's IP address

圖7.8使用Docker構圖和執行DNS查找的服務擴展

插入同一Docker網絡的容器將在同一網絡範圍內獲取IP地址，並通過該網絡連接。 使用DNS意味著，當您 的容器被更換並更改IP地址時，您的應用程序仍然有效，因為Docker中的DNS服務將始終從域查找中返回 當前容器的IP地址。

您可以使用Docker CLI手動刪除AccessLog容器，然後使用Docker Compose再次備份應用程序。 Compose將看到沒有運行訪問的容器，因此它將啟動一個新的容器。 該容器可能會從Docker網絡中具有新的IP地址（對正在創建的其他容器的限制），因此當您運行域查找時，您可能會看到不同的響應。

現在嘗試

仍在同一終端會話中，使用Docker CLI刪除AccessLog容器，然後使用Docker Compose將應用程序帶回所需的狀態。然後，使用Linux或Windows中的CMD中的SH再次連接到Web容器，然後運行更多DNS查找：

Docker Container RM -F-tase-at-Accesslog-1圖像

```
docker組成-d -scale iotd = 3
```

#對於Linux容器：

```
Docker Container exec- iT日期圖像 - 圖像gallery-1 sh
```

#適用於Windows容器：Docker Container exec -It tage-ta-day-image-gallery-1 cm
d

#在容器會話中：NSlookup AccessLog
NSlookUp IoTD

出口

您可以在圖7.9中看到我的輸出。就我而言，沒有其他過程創建或刪除容器，因此相同的IP地址172.18.0.2用於新的AccessLog容器。 在DNS查找IOTD API中，您可以看到返回了三個IP地址，服務中的三個容器中的每個都有一個。

Forcibly removing the `accesslog` container means the Compose app is no longer in the desired state. Compose brings it back up by creating a new `accesslog` container

```
PS>docker container rm -f image-of-the-day-accesslog-1  
image-of-the-day-accesslog-1  
PS>  
PS>docker compose up -d --scale iotd=3  
[+] Running 5/5  
✓ Container image-of-the-day-iotd-1          Runn...  
✓ Container image-of-the-day-iotd-2          Runn...  
✓ Container image-of-the-day-iotd-3          Runn...  
✓ Container image-of-the-day-accesslog-1      Started  
✓ Container image-of-the-day-image-gallery-1  Running  
PS>  
PS>docker container exec -it image-of-the-day-image-gallery-1 sh  
/web #  
/web # nslookup accesslog  
Server:      127.0.0.11  
Address:     127.0.0.11:53
```

Non-authoritative answer:

Non-authoritative answer:

```
Name:  accesslog  
Address: 172.18.0.2
```

A DNS lookup for the `accesslog` service shows the new container has been allocated the IP address of the old container - that address became available as soon as the container was removed

```
/web # nslookup iotd  
Server:      127.0.0.11  
Address:     127.0.0.11:53
```

Non-authoritative answer:

Non-authoritative answer:

```
Name:  iotd  
Address: 172.18.0.3  
Name:  iotd  
Address: 172.18.0.5  
Name:  iotd  
Address: 172.18.0.6
```

The `iotd` API service is running at scale with three containers. A DNS lookup returns all three IP addresses

圖7.9帶有多個容器的服務量表 - 每個容器的IP地址都在查找中返回。

DNS服務器可以返回域名的多個IP地址。 Docker構成這種機制來簡單負載平衡，返回服務的所有容器IP地址。 這取決於應用程序如何處理多個響應的DNS查找；一些應用程序採用了使用列表中第一個地址的簡單方法。 為了嘗試在所有容器上提供負載平衡，Docker DNS每次以不同的順序返回列表。 您會看到，如果您重複使用IOTD服務的NSlookup呼叫，這是嘗試在所有容器周圍傳播流量的基本方法。

Docker在撰寫文件中為您的容器組成了所有啟動選項，並且可以在運行時照顧容器之間的通信。您也可以使用它為環境設置配置。

7.4 Docker中的應用程序配置

第6章的待辦事項應用程序可以以不同的方式使用。您可以將其作為一個容器運行，在這種情況下，將數據存儲在SQLite數據庫中，這只是容器內部的文件。您在第6章中看到瞭如何使用卷來管理該數據庫文件。SQLITE適用於小型項目，但是較大的應用程序將使用單獨的數據庫，並且可以將待辦事項應用程序配置為使用遠程PostgreSQL數據庫而不是本地SQLITE。

Postgres是一個強大而流行的開源關係數據庫。它可以在Docker中運行良好，因此您可以運行一個分佈式應用程序，該應用程序在一個容器中運行，並且數據庫在另一個容器中。待辦事項應用程序的Docker映像已符合本書的指導，因此它包裝了Dev環境的默認配置值集，但是可以應用配置設置，以便它們與其他環境一起使用。我們可以使用Docker Compose應用這些配置設置。

查看列表7.3中撰寫文件的服務，這些文件指定了Postgres數據庫服務和待辦事項應用程序。

列出7.3使用Postgres數據庫的待辦事項應用程序的撰寫服務

```
services:  
  
todo-db:  
    image: diamol/postgres:2e-12  
    ports:  
        - "5433:5432"  
    networks:  
        - app-net  
  
todo-web:  
    image: diamol/ch06-todo-list:2e  
    ports:  
        - "8020:80"  
    environment:  
        - DatabaseProvider=Postgres  
    depends_on:  
        - todo-db  
    networks:  
        - app-net  
    secrets:  
        - source: postgres-connection  
          target: /app/config/secrets.json
```

數據庫的規範很簡單 - 它使用Diamol/Postgres : 2E-12圖像，在容器中發布標準Postgres端口5342到主機上的端口5433，並使用服務名稱ToDo-DB，這將是該服務的DNS名稱。Web應用程序有一些新的部分可以設置配置：

- 環境設置了容器內部創建的環境變量。當此應用程序運行時，將有一個稱為數據庫的環境變量：在容器內部設置的提供商，並帶有Value Postgres。可以從運行時環境中讀取秘密，並在容器內部填充作為文件。該應用程序將在/app/config/secrets.json上有一個文件，其中包含稱為Postgres-connection的秘密內容。
-

秘密通常由集群環境中的容器平台提供 - Kubernetes具有類似的機制。它們存儲在群集數據庫中，可以加密，因此它們對於敏感配置數據（例如數據庫連接字符串，證書或API鍵）很有用。在運行Docker的一台計算機上，沒有用於秘密的集群數據庫，因此使用Docker Compose可以從文件中加載秘密。該撰寫文件的末尾有一個秘密部分，如列表7.4所示。

選單7.4從Docker中的本地文件加載秘密

秘密：

```
Postgres-connection : 文件 : ./config/secrets.json
```

這告訴Docker撰寫從主機上的文件中加載稱為Postgres-connection的秘密，稱為secrets.json。這種情況就像我們在第6章中介紹的綁定安裝座一樣，實際上，主機上的文件浮出水面到容器中。但是，將其定義為秘密，可以使您以相同的方式使用真實的，加密的秘密在聚類環境中進行建模。

將應用程序配置插入撰寫文件中，您可以以不同的方式使用相同的Docker映像，並在每個環境中明確說明設置。您可以為開發和測試環境提供單獨的撰寫文件，發布不同的端口並觸發應用程序的不同功能。此組合文件設置了環境變量和秘密，以在Postgres模式下運行待辦事項應用程序，並為其提供連接到Postgres數據庫的詳細信息。

運行應用程序時，您會看到它的行為方式相同，但是現在數據存儲在Postgres數據庫容器中，您可以與應用程序分開管理。

現在嘗試

在本書的代碼根部打開終端會話，然後切換到本練習目錄。在該目錄中，您將看到Docker組成文件以及包含加載到應用程序容器的秘密的JSON文件。使用Docker-Compose以通常的方式啟動應用程序：

```
cd ./ch07/exercises/todo-list-postgres

# 對於Linux容器：Docker組成-D

# 或Windows容器（使用不同的文件路徑）：docker compose -f docker-compose-windows.yml up -d
```

Docker組成PS

圖7.10顯示了我的輸出。除了Docker-Compose PS命令之外，沒有什麼新鮮事物，該命令列出了該撰寫應用程序一部分的所有運行容器。

```
Starting a new Compose application. This creates a container network and runs a PostgreSQL database and the to-do web app

PS>cd ./ch07/exercises/todo-list-postgres
PS>
PS>docker compose up -d
[+] Running 3/3
✓ Network todo-list-postgres_app-net      Created
✓ Container todo-list-postgres-todo-db-1   Started
✓ Container todo-list-postgres-todo-web-1   Started
PS>
PS>docker compose ps
NAME          IMAGE           COMMAND
CREATED      STATUS          PORTS
todo-list-postgres-todo-db-1  diamol/postgres:2e-12 "docker-entrypoint.s...
25 seconds ago Up 25 seconds  0.0.0.0:5433->5432/tcp
todo-list-postgres-todo-web-1  diamol/ch06-todo-list:2e   "dotnet ToDoList.dll"
25 seconds ago Up 25 seconds  0.0.0.0:8030->80/tcp
PS>

This shows the running containers which are part of this Compose application. Other containers from other apps are not shown.
```

圖7.10使用Docker組成並列出其容器的新應用程序

您可以在`http://localhost:8030`瀏覽此版本的待辦事項應用程序。 功能是相同的，但是現在數據已保存在Postgres數據庫容器中。 您可以使用數據庫客戶端進行檢查 - 我使用SQLectron，它是一個快速，開源，跨平臺UI，用於連接到Postgres，MySQL和SQL Server以及其他數據庫。服務器的地址是`localhost:5433`，它是容器發布的端口；該數據庫稱為TODO，用戶名和密碼都是郵政。 您可以在圖7.11中看到我在Web應用程序中添加了一些數據，我可以在Postgres中查詢它。

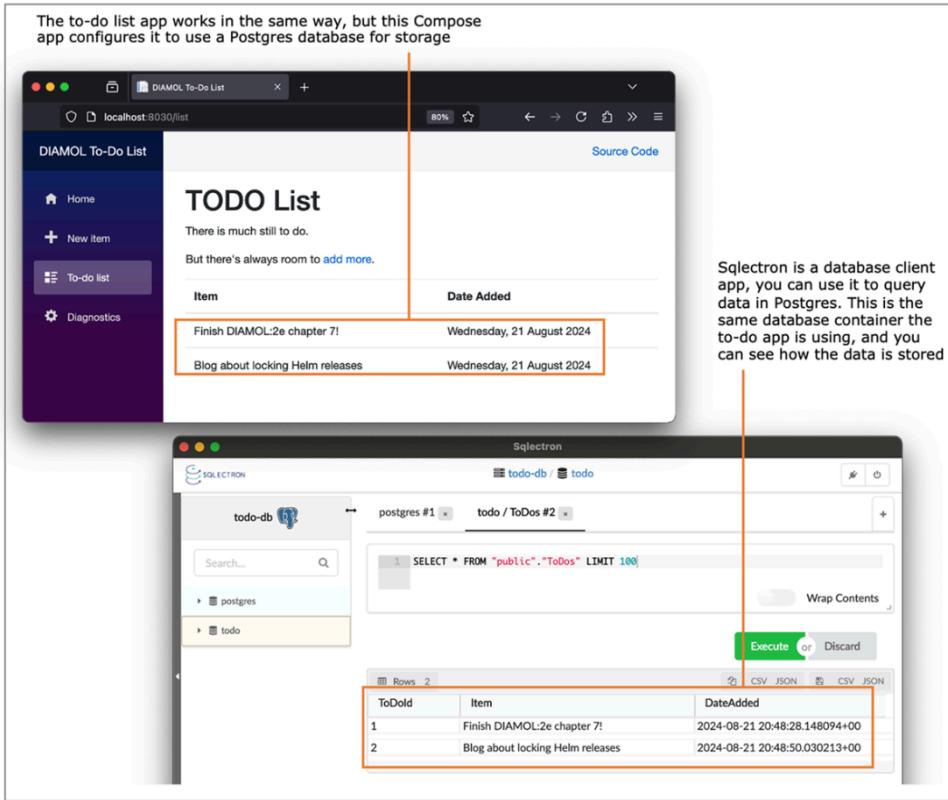


圖7.11在帶有Postgres數據庫的容器中運行待辦事項應用程序並查詢數據

將應用程序包與運行時配置分開是Docker的關鍵好處。 您的應用程序圖像將由您的構建管道生成，並且相同的圖像將在測試環境中進行，直到被確認為生產為止。 每個環境將使用環境變量或綁定安裝座或秘密應用自己的配置設置，在Docker組成的文件中易於捕獲。 在每個環境中，您都在使用相同的Docker圖像，因此您可以確信自己將與所有其他環境中測試的生產中完全相同的二進製文件和依賴項釋放到了生產中。

7.5 了解問題碼頭組成的解決方案

Docker Compose是一種非常整潔的方式，可以描述以小型清晰的文件格式描述複雜分佈式應用程序的設置。撰寫的YAML文件實際上是您應用程序的部署指南，但是它比寫為Word文檔的指南之前的數英里。在過去，這些Word文檔描述了應用程序發布的每個步驟，它們運行到了數十頁，其中包含不準確的描述和過時的信息。撰寫文件很簡單，並且可以操作 - 您使用它來運行您的應用程序，因此沒有過時的風險。

當您開始更多地使用Docker容器時，撰寫是工具包的有用組成部分。但是，重要的是要準確了解Docker構成的目的以及其局限性是什麼。撰寫使您可以定義應用程序，並將定義應用於運行Docker的單個計算機。它將該計算機上的實時Docker資源與撰寫文件中描述的資源進行了比較，並將向Docker API發送請求，以替換已更新的資源並在需要的地方創建新資源。

運行Docker-Compose時，您可以獲得所需的應用程序狀態，但這就是Docker撰寫的地方。它不是像Kubernetes這樣的完整容器平台，它不會不斷運行以確保您的應用程序保持其所需的狀態。如果容器失敗或手動刪除它們，則Docker組合將不會重新啟動或更換它們，直到您明確運行Docker再次組合為止。圖7.12使您可以很好地了解擬合應用程序生命週期的位置。

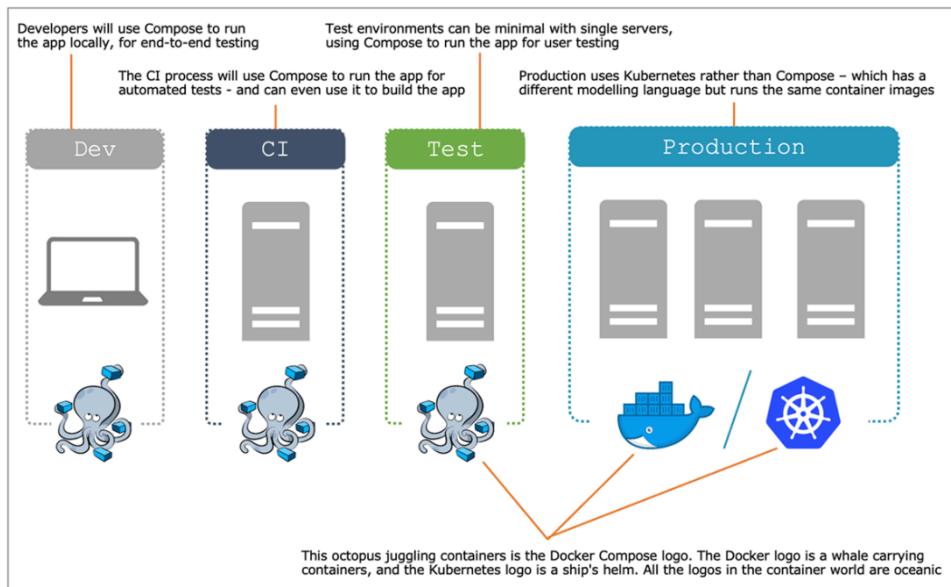


圖7.12您在應用程序生命週期中使用Docker在從開發到生產中構成的位置

這並不是說Docker組成不適合生產。如果您只是從Docker開始，並且將工作負載從單個VM遷移到容器，則可能是一個起點。您不會在該Docker機器上獲得高可用性，負載平衡或故障轉移，但是您也不會在單獨的App VM上得到它。 您將為您的所有應用程序獲得一套一致的工件，所有內容都有Dockerfiles和Docker撰寫文件 - 並且您將獲得一致的工具來部署和管理您的應用程序。這可能足以讓您開始在運行容器群集之前開始。

7.6 實驗室

Docker Compose具有一些有用的功能，可為運行應用程序增添可靠性。在這個實驗室中，我希望您創建一個撰寫定義，以在測試環境中更可靠地運行待辦事項Web應用程序，以便以下內容：

- 如果機器重新啟動或Docker Engine重新啟動，則應用程序容器將重新啟動。 數據庫容器將使用綁定安裝座來存儲文件，因此您可以將應用程序重新下來，但保留您的數據。 Web應用程序應在標準端口80上收聽測試。
-

這只是一個提示：

- 您可以在<https://docs.docker.com/compose/>找到Docker撰寫文件規範。<https://docs.docker.com/compose/>這定義了您可以捕獲的所有設置。

My sample solution is on the book's GitHub repository, as always. Hopefully, this one isn't too complex, so you won't need it: <https://github.com/sixeyed/diamol/blob/2e/ch07/lab/README.md>.

8 通過健康檢查和依賴性檢查 支持可靠性

我們正在旅途中，將軟件準備在容器中準備生產。您已經看到了在Docker Images中包裝應用程序，在容器中運行它們並使用Docker Compose定義多容器應用程序的簡單明了。在生產中，您將在像Kubernetes這樣的容器平台上運行應用程序，並且這些平台具有可幫助您部署自我修復應用程序的功能。 您可以將容器包裝在平台用來檢查容器中的應用程序是否健康的信息。 如果應用程序停止正常工作，則該平台可以刪除故障容器，並用新的容器替換。

在本章中，您將學習如何將這些支票包裝到容器圖像中，以幫助平台保持應用程序在線。

8.1 建築健康檢查到Docker圖像中

每次運行一個容器時，Docker都會在基本級別上監視您的應用程序的健康狀況。容器啟動時運行特定的過程，這可能是Java或.NET運行時，Shell腳本或應用程序二進制。 Docker檢查該過程仍在運行，如果停止，容器將進入退出狀態。

這為您提供了在所有環境中起作用的基本健康檢查。開發人員可以看到，如果過程失敗並且容器退出，他們的應用程序是不健康的。在聚類的環境中，容器平台可以重新啟動退出的容器或創建替換容器。但這是非常基本的檢查 - 確保該過程正在運行，但並非該應用程序實際上是健康的。容器中的Web應用程序可能會達到最大容量，並開始對每個請求返回http 503 “不可用” 響應，但是只要容器中的過程仍在運行，Docker也認為該容器是健康的，即使該應用程序停滯不前。

Docker為您提供了一種簡潔的方法，可以通過向Dockerfile添加邏輯來直接在Docker映像中構建真正的應用程序檢查。我們將使用一個簡單的HTTP API容器來完成此操作，但是首先，我們將在沒有任何健康檢查的情況下運行它，以確保我們了解問題。

現在嘗試

運行一個容納REST API的容器，該API返回隨機數。該應用程序有一個錯誤，因此在三個呼叫API之後，它變得不健康，隨後的每個電話都會失敗。打開終端，運行容器並使用API - 這是一個新圖像，因此您會在運行容器時看到Docker將其拉動：

```
# 啟動API容器Docker容器運行-D -P 8080 : 8080 diamol/ch08 -numbers -api : 2e
```

```
# 重複此三次 - 它返回一個隨機號碼curl http://localhost:8080/rng  
curl http://localhost:8080/rng curl http://localhost:8080/rng
```

```
# 從第四個電話開始，API總是失敗curl http://localhost:8080/rn  
g
```

```
# 檢查容器狀態碼頭容器LS
```

您可以在圖8.1中看到我的輸出。API在前三個調用中的行為正確，然後返回HTTP 500“內部服務器錯誤”響應。代碼中的錯誤意味著從現在開始，它將始終返回500。（實際上，這不是一個錯誤；該應用是故意編寫的。源代碼在第8章的存儲庫中，如果您想查看其工作原理。）在容器列表中，API容器具有狀態。容器內部的過程仍在運行，因此就Docker而言，它看起來不錯。容器運行時無法知道該過程中發生的事情以及應用程序是否仍正確地行為。

Runs a container which is a REST API for generating random numbers. The application has a bug which causes it to fail after three requests.

The first three calls to the API succeed and return random numbers.

```
PS> docker container run -d -p 8080:8080 diamol/ch08-numbers-api:2e  
a0bbbd618f516cf9dc9/5022df1e527/c866b661d89931/d3ed/d9c4c1a8b9a  
PS>  
PS>curl http://localhost:8080/rng  
59  
PS>curl http://localhost:8080/rng  
69  
PS>curl http://localhost:8080/rng  
60  
PS>curl http://localhost:8080/rng  
{"type":"https://tools.ietf.org/html/rfc9110#section-15.6.1","title":"An error occurred while processing your request.","status":500,"traceId":"00-f897b59435db7cd41653b84f56ed52da-1d693d56c2e06874-00"}  
PS>  
PS>docker container ls
```

CONTAINER ID	IMAGE	NAMES	COMMAND	CREATED	STATUS
a0bbbd618f516	diamol/ch08-numbers-api:2e	80/tcp, 0.0.0.0:8080->8080/tcp	"dotnet /app/Numbers.."	24 seconds ago	Up 23 seconds

The bug has kicked in and the app is broken. From now it will always return an HTTP 500 "Internal Server" error.

But the `entrypoint` process for the container is still running, so Docker thinks the app is healthy and the container status is Up.

圖8.1 Docker檢查了應用程序過程，因此即使應用程序處於失敗狀態，容器也已啟動。

輸入HealthCheck指令，您可以將其添加到Dockerfile中，以確切地告訴運行時間如何檢查容器中的應用程序是否仍然健康。HealthCheck指令指定了Docker在容器內運行的命令，該命令將返回狀態代碼。該命令可以是您需要檢查應用程序是否健康的任何內容。Docker將以定時的間隔在容器中運行該命令。如果狀態代碼說一切都很好，那麼容器就健康。如果狀態代碼連續幾次故障，則將容器標記為不健康。

列表8.1在一個隨機數API的新Dockerfile中顯示了HealthCheck命令，我將以版本2的形式構建該命令（完整文件位於CH08/ch08/ercory/nubmess/numbers/nyvels-api/dockerfile.v2的書籍源中。此健康檢查使用像主機上的捲曲命令一樣，但是這次它運行在容器內。/健康URL是應用程序中的另一個終點，該終點檢查了該錯誤是否已觸發；如果該應用程序正常工作，它將返回200個“確定”狀態代碼，而當該應用程序損壞時，它將返回500個“內部服務器錯誤”。

清單8.1 Dockerfile中的Healthcheck指令

來自Diomol/dotnet-Appnet：2e

```
entrypoint [ "dotnet" , "/app/numbers.api.dll" ] HealthCheck C  
MD curl -fail -fail http://localhost/health
```

WorkDir /App

複製 - 從= builder / out /。

其餘的Dockerfile非常簡單。這是一個.NET應用程序，因此入口處運行DotNet命令，而DotNet進程則是Docker監視的，以檢查該應用程序是否仍在運行。健康檢查對/健康端點進行了HTTP調用，API提供了該應用程序是否健康。使用-fault參數表示curl命令將狀態代碼傳遞給Docker - 如果請求成功，它將返回數字0，Docker讀取為成功的檢查。如果失敗，它將返回其他數字以外的數字，這意味著健康檢查失敗。

我們將構建該圖像的新版本，以便您可以看到構建命令如何與不同的文件結構一起工作。通常，您的應用程序源文件夾中有一個Dockerfile，Docker會找到並運行構建。在這種情況下，Dockerfile具有不同的名稱，並且位於與源代碼的單獨文件夾中，因此您需要明確指定構建命令中的路徑。

現在嘗試

運行一個終端並瀏覽到具有本書源代碼的文件夾。然後使用V2 Dockerfile使用V2標籤構建新圖像：

```
# 瀏覽到根路徑，該路徑具有用於源代碼和Dockerfiles的文件夾：CD ./ch08/exercises/numbers

# 使用-f標誌構建圖像來指定Dockerfile的路徑：Docker Image Build -T Diamol/ch08-numbers-api : 2e-v2 -f ./numbers-api/dockerfile.v2。
```

構建圖像後，您就可以使用健康檢查來運行該應用程序。您可以配置健康檢查運行頻率，以及多少個失敗檢查意味著該應用程序不健康。默認值是每30秒運行一次，對於連續三個失敗，以觸發不健康狀態。API 圖像的V2版本具有內置的健康檢查，因此，當您重複測試時，您會發現該容器的健康報告會報告。

現在嘗試

運行相同的測試，但使用V2映像標籤，並在命令之間留出一些時間，讓Docker在容器內開火。

```
# 啟動API容器，V2 Docker容器運行-D -P 8081 : 8080 DIAMOL/CH08-NUMBERS-
API : 2E-V2

# 等待30秒左右，並列出容器

# 重複此四次 - 它返回三個隨機數，然後失敗curl http://localhost:8081/rng curl http://localhost:8081/rng curl http://localhost:8081/rng curl http://localhost:8081/rng curl http://localhost:8081/rng : 8081/rng

# 現在，該應用處於失敗狀態 - 等待90秒並檢查Docker Container LS
```

我的輸出在圖8.2中。您可以看到，新版本的API容器最初顯示健康狀態 - 如果圖像內置了健康檢查，Docker顯示了運行容器的健康檢查狀態。觸發錯誤後的某個時候，容器顯示為不健康。

```
Version 2 of the API uses the same code but includes a health check in the Docker image

The status column now shows the container is up and it is healthy - Docker has
run the health check inside the container and it has returned a success code

PS>docker container run -d -p 8081:8080 diamol/ch08-numbers-api:2e-v2
238d8430fa7e4aa085135f178e06848b10112ubac007e7b8375bcd11731b52e1
PS>
PS>docker container ls
CONTAINER ID IMAGE COMMAND CREATED
STATUS PORTS NAMES
238d8430fa7e diamol/ch08-numbers-api:2e-v2 "dotnet /app/Numbers..." 39 seconds ago
Up 38 seconds (healthy) 0.0.0.0:8081->8080/tcp cool_currان
a0bbd618f516 diamol/ch08-numbers-api:2e "dotnet /app/Numbers..." 24 minutes ago
Up 24 minutes 80/tcp, 0.0.0.0:8080->8080/tcp quirky_buck
PS>
PS>curl http://localhost:8081/rng
93
PS>curl http://localhost:8081/rng
14
PS>curl http://localhost:8081/rng
64
PS>curl http://localhost:8081/rng
{"type":"https://tools.ietf.org/html/rfc9110#section-15.6.1","title":"An error occurred while processing your request.","status":500,"traceId":"00-1b08285a0b695e6adc1af3eee
a9d07f1-4277a0355d4b3699-00"}
PS>
PS>docker container ls
CONTAINER ID IMAGE COMMAND CREATED
STATUS PORTS NAMES
238d8430fa7e diamol/ch08-numbers-api:2e-v2 "dotnet /app/Numbers..." 2 minutes ago
Up 2 minutes (unhealthy) 0.0.0.0:8081->8080/tcp cool_currان
a0bbd618f516 diamol/ch08-numbers-api:2e "dotnet /app/Numbers..." 25 minutes ago
Up 25 minutes 80/tcp, 0.0.0.0:8080->8080/tcp quirky_buck
PS>

The API is broken from the fourth call onwards - now
the health check will return a failure status code

After three successive failures the container is flagged as unhealthy
- but it is still running, Docker doesn't stop unhealthy containers
```

圖8.2損壞的應用顯示為不健康的容器，但容器仍在啟動。

這種不健康的狀態是從Docker的API中發布的，因此通知運行該容器的平台可以採取行動來修復應用程序。Docker還記錄了最新的健康檢查結果，當您檢查容器時，您可以看到。您已經看到了Docker Container Inspector的輸出，該輸出顯示了Docker對容器的所有了解。如果運行健康檢查，也會顯示出來。

現在嘗試

我們有兩個API容器正在運行，創建它們時沒有給出一個名稱，但是我們可以使用帶有 -last Flag的容器LS找到最近創建的容器的ID。您可以將其饋送到容器檢查中，以查看最新容器的狀態：

```
docker容器檢查$ (docker容器ls - las -last 1 -format'{{.id}}')
```

JSON數據的頁面將返回此處，如果您滾動到州字段，您會發現有一個健康部分。其中包含健康檢查的當前狀態，“連勝失敗”，即連續失敗的數量以及最近的健康檢查日誌。在圖8.3中，您可以看到我的容器狀態的摘錄。健康檢查的失敗連續六個，這會觸發容器處於不健康狀態，您可以看到健康檢查命令中的日誌，當它們獲得500的HTTP狀態結果時，它們失敗了。

The output from docker container inspect includes a detailed State section

The health check for my container has failed 11 times in a row - the default is three times to trigger unhealthy status

```
"State": {  
    "Status": "running",  
    "Running": true,  
    "Paused": false,  
    "Restarting": false,  
    "OOMKilled": false,  
    "Dead": false,  
    "Pid": 2605,  
    "ExitCode": 0,  
    "Error": "",  
    "StartedAt": "2024-08-28T14:19:29.380199804Z",  
    "FinishedAt": "0001-01-01T00:00:00Z",  
    "Health": {  
        "Status": "unhealthy",  
        "FailingStreak": 11,  
        "Log": [  
            {  
                "Start": "2024-08-28T14:23:29.789046346Z",  
                "End": "2024-08-28T14:23:29.829134804Z",  
                "ExitCode": 22,  
                "Output": "  % Total    % Received % Xferd  
me   Time      Time Current\n          Dload  
Spent Left Speed\n0 0 0 0 0 0 0 0 0 0 0 0  
--:--:--:--:--:--:--:--:--:--:--:--:  
--:--:--:--:--:--:--:--:--:--:--:--:  
0\ncurl: (22) The requested URL returned error: 500\n"}  
},  
}  
}
```

Health check logs are shown too, so I can see the status code of the check and the log showing an HTTP 500 error code

圖8.3具有健康檢查的容器顯示了該應用程序的健康狀況和健康檢查日誌。

健康檢查正在做應有的事情：測試容器內的應用程序，並向Docker標記該應用不再健康。但是您還可以在圖8.3中看到，我的不健康容器具有“運行”狀態，因此即使Docker知道它無法正常工作，它仍然可以使用。為什麼Docker不重新啟動或更換該容器？

一個簡單的答案是，Docker無法安全地這樣做，因為Docker引擎在單個服務器上運行。 Docker可以停止並重新啟動該容器，但這意味著您的應用程序被回收時的停機時間。 或Docker可以刪除該容器並從同一設置啟動新的容器，但是您的應用程序可能會在容器內寫入數據，因此這意味著停機時間和數據丟失。 Docker不能確定採取行動修復不健康的容器不會使情況變得更糟，因此它廣播了該容器不健康，但會使其運行。 健康檢查也繼續進行，因此，如果故障是暫時的，下一個支票通過，容器狀態會再次變為健康。

健康檢查在集群中確實有用，多個服務器運行Docker由Kubernetes等容器平台管理。 然後，如果容器不健康，可以採取行動，將通知容器平台。（並非所有平台都尊重圖像中內置的健康檢查，但如果沒有，則它們具有類似的機制。）由於集群中還有額外的容量，因此在不健康的容器仍在運行的同時，可以啟動一個替換容器，因此不應有任何應用停機時間。

8.2 啟動具有依賴性檢查的容器

健康檢查是一項正在進行的測試，可幫助容器平台保持應用程序的運行。 帶有多個服務器的群集可以通過啟動新容器來處理臨時故障，因此即使您的某些容器停止響應，也不會損失服務。 但是，跨群集運行會給分佈式應用帶來新的挑戰，因為您無法再控制可能彼此依賴的容器的啟動訂單。

我們的隨機數生成器API有一個網站可以使用。 Web應用程序以自己的容器運行，並使用API容器生成隨機數。 在單個Docker服務器上，您可以確保在Web容器之前創建API容器，因此當Web應用程序啟動時，它具有所有可用的依賴項。 您甚至可以用Docker Compose明確捕獲這一點。 但是，在聚類的容器平台中，您不能規定容器的啟動順序，因此Web應用程序可能在API可用之前開始。

然後發生的事情取決於您的應用程序。 隨機數應用程序無法很好地處理。

現在嘗試

卸下所有運行的容器，因此現在您沒有API容器。 然後運行Web應用程序容器並瀏覽。 該容器已啟動並且該應用程序可用，但是您會發現它實際上行不通。

```
Docker容器RM -F $ (Docker容器LS -AQ) Docker容器運行-D -P 8082 : 8080 dia  
mol/ch08 -numbers -web : 2e docker容器LS
```

現在瀏覽到http://localhost:8082。您會看到一個看起來還不錯的簡單Web應用程序，但是如果單擊隨機數字按鈕，則會看到圖8.4中顯示的錯誤。

The web app uses the API to generate random numbers.
It doesn't check the API is available when it starts

PS>
PS>docker container rm -f \$(docker container ls -aq)
238d8430fa7e
a0bbd618f516
PS>
PS>docker container run -d -p 8082:8080 diamol/ch08-numbers-web:2e
f96d711d428dab77514f31ed91aa9b3799338b4b05c06013daa39452a962d0e
PS>
PS>docker container ls

CONTAINER ID	IMAGE	COMMAND	CREATED
f96d711d428d	diamol/ch08-numbers-web:2e	"dotnet /app/Numbers..."	9 seconds ago
Up 8 seconds	0.0.0.0:8082->8080/tcp		practical_jones

PS>

localhost:8082

Source Code

DIAMOL Random Number Generator

RNG service unavailable!

Get a random number

But the API service isn't available, so the web app doesn't work at all
- even though the server process is running and the container is up

圖8.4無法驗證其依賴項的應用程序看起來還可以，但狀態失敗。

這正是您不想發生的事情。 該容器看起來不錯，但是該應用程序無法使用，因為其密鑰依賴性不可用。某些應用程序可能內置了邏輯，以驗證它們啟動時所需的依賴項是否存在，但是大多數應用程序卻沒有，並且隨機號碼Web應用程序就是其中之一。它假設在需要時可以使用API，因此不會進行任何依賴性檢查。

您可以在Docker映像中添加該依賴項檢查。 依賴性檢查與健康檢查不同 - 它在應用程序啟動之前運行，並確保應用程序所需的所有內容。如果所有內容都存在，則依賴性檢查完成並啟動應用程序。如果依賴關係不存在，則支票失敗並退出容器。Docker沒有內置功能，例如HealthCheck指令進行依賴性檢查，但是您可以將該邏輯放入啟動命令中。

清單8.2顯示了Web應用程序的新Dockerfile的最終應用階段（完整文件是在CH08/ermities/numbers/numbers/web/dockerfile.v2） - CMD指令驗證API在啟動應用程序之前是否可用。

列出8.2 dockerfile，在啟動命令中具有依賴項檢查

```
來自diamol/dotnet-aspnet：2e env rngapi：url =
```

```
cmd curl-fab-http：// numbs-api/rng && \ dotnet numbers.  
web.dll
```

```
workdir /app copy -from = builder /o  
ut /。
```

此檢查再次使用捲曲工具，該工具是基本圖像的一部分。CMD指令在容器啟動時運行，並且對API進行HTTP調用，這是一項簡單的檢查，以確保其可用。Double-pampersand，&&，在Linux和Windows Command Shells中以相同的方式工作 - 如果左側的命令成功，它將在右側運行命令。

如果我的API可用，則curl命令將成功並啟動應用程序。這是一個.NET Web應用程序，因此Docker將監視Dotnet流程，以驗證該應用程序是否還活著（此Dockerfile中沒有健康檢查）。如果API不可用，則curl命令將失敗，dotnet命令將不會運行，並且在容器中什麼也不會發生，因此它退出。

現在嘗試

從隨機號碼Web圖像的V2標籤中運行一個容器。仍然沒有API容器，因此當該容器啟動時，它將失敗並退出：

```
Docker容器運行-D -P 8084：8080 diamol/ch08-numbers-web：2e-v2
```

```
Docker容器LS-
```

您可以在圖8.5中看到我的輸出。V2容器啟動後僅幾秒鐘就退出了，因為Curl命令未能找到API。原始的Web應用程序容器仍在運行，並且仍然無法使用。

```

Version 2 of the web application image includes a dependency
check, which makes sure the API is available before the app runs

PS>docker container run -d -p 8084:8080 diamol/ch08-numbers-web:2e-v2
51e6749c36a277fc56da105da8998476ub9e35bc7bb5a2010fdd5b82d6276b2a
PS>
PS>docker container ls --all
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
51e6749c36a2        diamol/ch08-numbers-web:2e-v2   "/bin/sh -c 'curl ..."   21 seconds ago    Exited (6) 21 seconds ago
f96d711d428d        diamol/ch08-numbers-web:2e       "dotnet /app/Numbers..."  12 minutes ago   Up 12 minutes
PS>

There's no API container running so the dependency check fails
and the app doesn't start - the container exits straight away

```

圖8.5如果檢查失敗，則具有依賴性檢查的容器。

它是違反直覺的，但是在這種情況下，要比運行的容器更好。這是失敗的行為，這是您在大規模運行時想要的。當容器退出時，該平台可以安排一個新容器以升級並更換它。也許API容器需要很長時間才能啟動，因此當Web容器運行時無法使用。在這種情況下，Web容器退出，安排了一個替換，並且到啟動時，API啟動並運行。

借助健康和依賴檢查，我們可以包裝應用程序在容器平台中成為一個好公民。到目前為止，我們使用的檢查是使用Curl非常基本的HTTP測試。這證明了我們想做的事情，但這是一種簡單的方法，最好不要依靠外部工具進行檢查。

8.3編寫用於應用程序檢查邏輯的自定義實用程序

Curl是測試Web應用程序和API的非常有用的工具。它是跨平台，因此它可以在Linux和Windows上使用，並且是我用作金色圖像的基礎的.NET運行時圖像的一部分，因此我知道它將在那裡進行檢查。我實際上並不需要圖像中的捲發才能運行我的應用程序，並且安全審查可能要求將其刪除。

我們介紹了第4章 - 您的Docker映像應具有最低限度，以便您的應用程序運行。任何額外的工具都會增加圖像大小，並且還會增加更新的頻率和安全攻擊表面。因此，儘管Curl是用於開始容器檢查的有用工具，但最好使用應用程序使用與應用程序使用的語言編寫自定義實用程序 - Java for Java Apps，Node.js for Node.js Apps等。

這有很大的優勢：

- 您可以減少圖像中的軟件需求 - 您不需要安裝任何額外的工具，因為檢查實用程序所需運行的所有內容都已用於應用程序。

- 您可以在檢查或分支的檢查中使用更複雜的條件邏輯，這些邏輯很難在Shell腳本中表達，尤其是如果您為Linux和Windows發布跨平台Docker映像。您的實用程序可以使用與應用程序使用的相同應用程序配置，因此您最終不會在多個地方指定諸如URL之類的設置，並有可能無法同步。您可以執行所需的任何測試，檢查數據庫連接或文件路徑是否存在您期望平台加載到容器中的證書 - 使用應用程序使用的相同庫。
-

在多種情況下，公用事業也可以通用。我在.NET中寫了一個簡單的HTTP檢查實用程序，可以在API映像中用於健康檢查和Web圖像中的依賴項檢查。每個應用都有多階段的Dockerfiles，一個階段在其中編譯了應用程序，另一個階段編譯了檢查實用程序，以及應用程序和實用程序中的最終階段副本。圖8.6顯示了它的外觀。

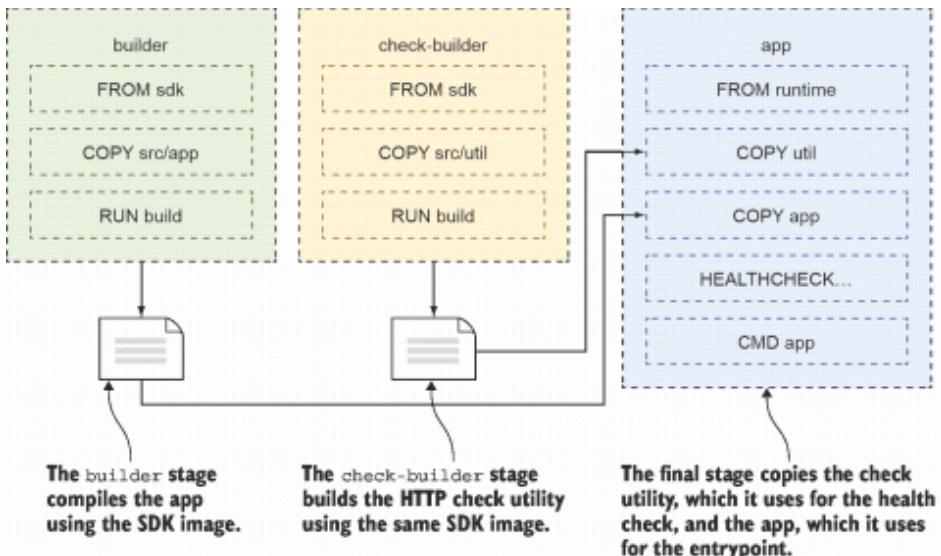


圖8.6使用多階段構建來編譯和包裝實用程序以及應用程序

清單8.3顯示了Dockerfile.v3的最後階段。現在，健康檢查的命令使用了.NET應用程序的Check Utility，因此檢查不再需要在圖像中安裝捲髮。

清單8.3使用自定義實用程序進行健康檢查以消除捲髮的需求

來自Diomol/dotnet-Appnet : 2e

```
entrypoint [ "dotnet" , "numbers.api.dll" ] HealthCheck cmd [ "dotnet" ,  
"utilities.httpcheck.dll" , "-U" , "http://localhost:8080/health" ]
```

WorkDir /App

複製-from = http-check-builder / out / 。複製 - 從= builder / out / 。

新的健康檢查的行為幾乎相同；與捲曲版本的唯一區別在於，當您檢查容器時，您不會在輸出中看到太多的詳細記錄。每次檢查只有一行，說這是成功還是失敗。該應用程序最初仍應保持健康；在您打電話給API後，它將被標記為不健康。

現在嘗試

刪除所有現有容器，並運行隨機數API的版本3。這次，我們將指定健康檢查間隔，以便更快地觸發它。檢查容器是否被列為健康，然後使用API並檢查容器是否將其翻轉為不健康：

```
# clear down existing containers
docker container rm -f $(docker container ls -aq)
# start the API container, v3

docker container run -d -p 8080:8080 --health-interval 5s diamol/ch08-numbers-
api:2e-v3

# wait five seconds or so and list the containers
docker container ls

# repeat this four times - it returns three random numbers and then fails
curl http://localhost:8080/rng
curl http://localhost:8080/rng
curl http://localhost:8080/rng
curl http://localhost:8080/rng

# now the app is in a failed state - wait 15 seconds and check again
docker container ls
```

Figure 8.7 shows my output. The behavior is the same as version 2, with the health check failing once the bug in the API is triggered, so the HTTP check utility is working correctly.

Version 3 of the API uses the .NET HTTP utility app for the health check. This command specifies five seconds as the interval between health checks

```
PS>docker container rm -f $(docker container ls -aq)
9b540a74136a
PS>
PS>docker container run -d -p 8080:8080 --health-interval 5s diamol/ch08-
numbers-api:2e-v3
7b79fb9316a86a9a6b679952a53150af63bd355ab1e9618afe533164813b65bb
PS>
PS>docker container ls
CONTAINER ID IMAGE COMMAND C
CREATED STATUS PORTS NAMES
7b79fb9316a8 diamol/ch08-numbers-api:2e-v3 "dotnet Numbers.Api..." 1
1 seconds ago Up 10 seconds (healthy) 0.0.0.0:8080->8080/tcp eloquent_tharp
PS>
PS>curl http://localhost:8080/rng
12
PS>curl http://localhost:8080/rng
11
PS>curl http://localhost:8080/rng
66
PS>curl http://localhost:8080/rng
{"type":"https://tools.ietf.org/html/rfc9110#section-15.6.1","title":"An
error occurred while processing your request.","status":500,"traceId":"00
-57bb1eb31cda9a599bb5c4915c5cd759-6bb08a31a89c94d0-00"}
PS>
PS>docker container ls
CONTAINER ID IMAGE COMMAND C
CREATED STATUS PORTS NAMES
7b79fb9316a8 diamol/ch08-numbers-api:2e-v3 "dotnet Numbers.Api..." 4
0 seconds ago Up 39 seconds (unhealthy) 0.0.0.0:8080->8080/tcp eloquent_tharp
PS>
```

The API is still broken, so after four calls it enters the failed state

The HTTP check utility now returns a failure code, and after three failures in a row the container enters the unhealthy state

圖8.7使用包裝在Docker映像中的實用工具的容器健康檢查

HTTP檢查實用程序具有許多選項，可以使其適合不同的情況。在Web應用程序中，在Dockerfile.v3中，我在啟動時使用相同的實用程序進行依賴項檢查，以查看API是否可用。

清單8.4顯示了Dockerfile的最後階段。在這種情況下，我使用-t標誌來設置實用程序應等待響應的時間，並且-c標誌告訴實用程序將與應用程序加載相同的配置文件，並從應用程序配置中獲取API的URL。

使用實用程序清單8.4在容器啟動時進行依賴性檢查

來自Diomol/dotnet-Appnet : 2e

```
env rngapi : url = http://numbs-api:8080/rng
```

```
cmd dotnet utilities.httpcheck.dll -c rngapi : url -t 900 && \dotnet numbers.web.dll
```

```
workdir /app copy -from = http-check-builder /out /  
◦複製 - 從= builder / out / ◦ d
```

同樣，這消除了應用程序圖像中捲曲的要求，但是該行為與啟動命令中的HTTP實用程序大致相同。

現在嘗試

運行Web應用程序的版本3，您會發現容器幾乎立即退出，因為HTTP檢查實用程序進行API檢查時會失敗：

```
Docker集裝箱運行-D -P 8081 : 8080 diamol/ch08 -numbers -we B : 2E-V3
```

Docker容器LS-

您的輸出將像我的圖8.8中一樣。您會看到API容器仍在運行，但仍然不健康。 Web容器找不到它，因為它正在尋找DNS名稱數字API，並且在運行API容器時，我們沒有指定該名稱。如果我們為API容器使用了該名稱，則Web應用程序將連接並能夠使用它，儘管它仍然會顯示錯誤，因為API中的錯誤已被觸發，並且沒有響應。

```

Version 3 of the web app image uses the HTTP check utility for the dependency check at startup

PS>docker container run -d -p 8081:8080 diamol/ch08-numbers-web:2e-v3
c76b7dd1151cadfc3b8c03a349ed19b298udb4234a4917089a47d1166abf2240
PS>
PS>docker container ls --all
CONTAINER ID        IMAGE               COMMAND                  PORTS
CREATED             STATUS              NAMES
NAMES
c76b7dd1151c      diamol/ch08-numbers-web:2e-v3   "/bin/sh -c 'dotnet ..." "dotnet Numbers.Api..." "0.0.0.0:8080->8080/tcp
4 seconds ago     Exited (1) 3 seconds ago
bold_sammet
7b79fb9316a8      diamol/ch08-numbers-api:2e-v3    "dotnet Numbers.Api..." "dotnet Numbers.Api..." "0.0.0.0:8080->8080/tcp
6 minutes ago     Up 6 minutes (unhealthy)
eloquent_tharp

The dependency check in the web
app fails so the container exits

There is an API container running, but it doesn't have the name
numbers-api which is the DNS name the web app is looking for

```

數字 8.8使用包裝到Docker映像中的實用程序作為依賴關係

檢查工具

在實用程序中編寫自己的支票的另一個好處是，它使您的圖像可移植。不同的容器平台具有不同的聲明和使用健康檢查和依賴性檢查的方式，但是如果具有圖像中實用程序所需的所有邏輯，則可以與Docker，Docker Compose和Kubernetes以相同的方式工作。

8.4定義Docker中的健康檢查和依賴性檢查

如果您不相信容器在沒有依賴項時失敗和退出是一個好主意，那麼您將了解其工作原因。Docker Compose可以採取某種方式來維修不可靠的應用程序，但由於Docker Engine不得不替換不健康的容器，它不會替換不健康的容器：您正在一台服務器上運行，並且該修復程序可能會導致中斷。但是，如果它們退出，它可以設置容器以重新啟動，如果圖像中還沒有一個，則可以添加健康檢查。

清單8.5顯示了在Docker組成文件中聲明為服務的隨機數API（完整文件在CH08/ermites/numbers/numbers/docker-compose.yml中）。它指定了V3容器圖像，該圖像使用HTTP實用程序進行健康檢查，並添加設置以配置健康檢查應如何工作。

清單8.5在Docker組成文件中指定健康檢查參數

```
數字-API：圖像：Diamol/ch08-numbers-api：2e-v  
3端口： - “8087：8080：8087：8080” HealthC  
heck：Interval：5s超時：1S Retries：2 start_period  
：5S網絡： - App-Net
```

您對健康檢查有細粒度的控制。 我正在使用Docker映像中定義的實際健康檢查命令，但使用自定義設置來運行：

- 間隔是支票之間的時間 - 在這種情況下，五秒鐘。 超時是在被視為失敗之前應允許支票運行多長時間。 檢索是在將容器標記為不健康之前允許的連續故障數量。
- start_period是觸發健康檢查之前等待的時間，這使您可以在運行健康檢查之前給您的應用程序一些啟動時間。
-

對於每個應用程序和每個環境，這些設置可能會有所不同 - 在迅速發現您的應用程序失敗和允許臨時故障之間有一個平衡，因此您不會觸發有關不健康容器的錯誤警報。我對API的設置非常激進。 運行健康檢查需要CPU和內存，因此在生產環境中，您可能會以更長的間隔運行。

您還可以在撰寫文件中為圖像中沒有聲明的容器添加健康檢查。列表8.6在同一Docker組成的文件中顯示了Web應用程序的服務，在這裡，我正在為該服務添加健康檢查。我正在指定用於API服務的相同選項集，但是還有測試字段，它使Docker可以運行健康檢查命令。

列表8.6在Docker中添加健康檢查

```
numbers-web:  
  image: diamol/ch08-numbers-web:2e-v3  
  restart: on-failure  
  ports:  
    - "8088:8080"  
  healthcheck:  
    test: ["CMD", "dotnet", "Utilities.HttpCheck.dll", "-t", "150"]  
    interval: 5s  
    timeout: 1s  
    retries: 2  
    start_period: 10s  
  networks:  
    - app-net
```

最好在所有容器中添加健康檢查，但是此示例與圖像和重新啟動中的依賴關係檢查在一起：實用設置，這意味著，如果容器出乎意料地退出，Docker將重新啟動它（如果您還沒有這樣做，則可以為您重新啟動第7章實驗室的答案之一）。沒有依賴性設置，因此Docker組合可以按任何順序啟動容器。如果Web容器在API容器準備就緒之前開始，則依賴項檢查將失敗，Web容器將退出。同時，API容器將啟動，因此，當重新啟動Web應用程序容器時，依賴項檢查將成功，並且該應用程序將完全運行。

現在嘗試

清除運行的容器，並使用Docker Compose啟動隨機數字應用程序。列出您的容器，以查看Web應用程序是否確實啟動，然後重新啟動：

```
# 瀏覽撰寫文件CD ./ch08/ecercises/numbers # 清除現有容器
docker容器 docker rm -f $(docker容器ls -aq) # 啟動app d
ocker docker composs -d # 等待五秒鐘左右
```

我的輸出在圖8.9中，您的輸出應該非常相似。 撰寫同時創建兩個容器，因為沒有指定依賴關係。 當API容器啟動時，並且在應用程序準備好處理請求之前，Web容器的依賴項檢查運行。您可以在我的日誌中看到HTTP檢查帶有錯誤的退出，因此檢查失敗和容器退出。Web服務配置為在故障上重新啟動，因此同一容器再次啟動。 這次，API檢查在54毫秒內獲得了成功狀態代碼，因此支票通過，並且應用處於工作狀態。

I've removed all my containers so Docker Compose will start new containers for the API and for the web application - but I have no dependencies, so they could start at the same time

The API container is healthy. It has been running for 11 seconds, but it was created 12 seconds ago so it took one second for the app to start

```
PS>cd ./ch08/exercises/numbers
PS>
PS>docker compose up -d
[+] Running 2/2
  ✓ Container numbers-numbers-web-1  Started
  ✓ Container numbers-numbers-api-1  Started
PS>
PS>docker container ls
CONTAINER ID   IMAGE               COMMAND           CREATED          STATUS            PORTS          NAMES
0dcafe2dac97   diamol/ch08-numbers-api:2e-v3    "dotnet Numbers.Api..."   12 seconds ago   Up 11 seconds (healthy)   0.0.0.0:8087->8080/tcp   numbers-numbers-api-1
fe6b21f4e532   diamol/ch08-numbers-web:2e-v3     "/bin/sh -c 'dotnet ..."   12 seconds ago   Up 11 seconds (healthy)   0.0.0.0:8088->8080/tcp   numbers-numbers-web-1
PS>
PS>docker container logs numbers-numbers-web-1
HTTPCheck: error. Url http://numbers-api:8080/rng, exception Connection refused (numbers-api:8080)
HTTPCheck: status OK, url http://numbers-api:8080/rng, took 54ms
```

The web container is up - so the dependency check succeeded. It also took one second for the app to start and the container to be Up

But check the logs and we can see the container has been restarted. There are two logs from the HTTP check - the first check failed because the API wasn't running. The container exited, Docker restarted it and this time the HTTP check passed

圖8.9 Docker構成添加的彈性 - 首次檢查失敗後，Web容器將重新啟動。

瀏覽到http://localhost:8088，您最終可以通過Web應用程序獲取一個隨機號碼。至少，您可以單擊三次按鈕並獲得一個數字 - 在第四個按鈕，您將觸發API錯誤，此後您只會遇到錯誤。圖8.10顯示了罕見的成功之一。

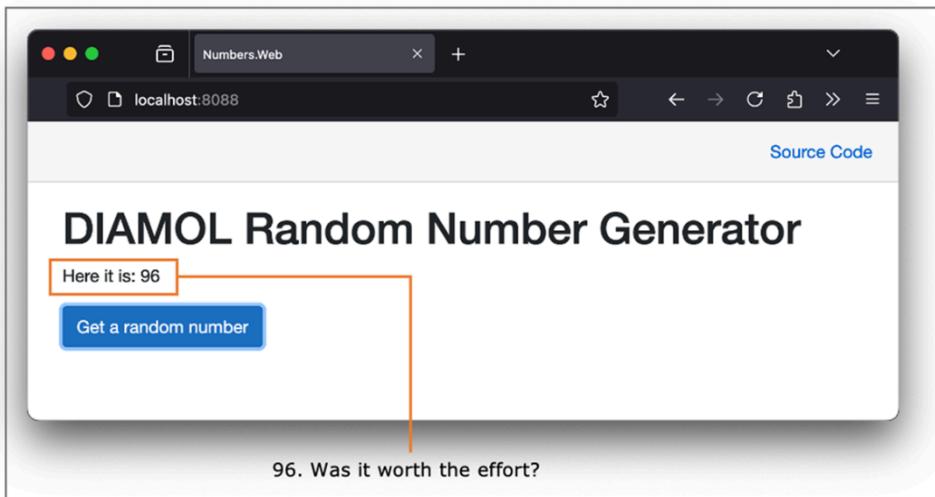


圖8.10該應用程序最終正常工作，並且在容器中進行了健康和依賴檢查。

您可能會問，當Docker組合可以使用`deweds_on`標誌為您完成時，為什麼要打擾到容器啟動的依賴項檢查呢？答案是，撰寫只能管理一台計算機上的依賴性，而應用程序在生產群集上的啟動行為遠遠低於可預測的。

8.5了解檢查電源自我修復應用程序

將您的應用程序構建為具有許多小組件的分佈式系統會提高您的靈活性和敏捷性，但確實使管理更複雜。組件之間將有很多依賴關係，並且很容易宣布要啟動組件的順序，以便您可以對依賴關係進行建模。但這確實不是一個好主意。

在一台計算機上，我可以告訴Docker組合我的Web容器取決於我的API容器，它將以正確的順序啟動它們。在生產中，我可能會在十二個服務器上運行Kubernetes，並且可能需要20個API容器和50個Web容器。如果我對啟動訂單進行建模，則容器平台會在啟動任何Web容器之前先啟動所有20個API容器嗎？如果19個容器開始良好，但是二十歲的容器有問題並需要5分鐘才能開始怎麼辦？我沒有Web容器，因此我的應用程序沒有運行，但是所有50個Web容器都可以運行，並且無法使用1個API容器可以正常運行。

這是依賴性檢查和健康檢查的地方。您不需要平台來保證啟動訂單 - 您會盡可能快地將其旋轉到盡可能多的服務器上。如果其中一些容器無法達到依賴項，則它們會迅速失效，並將重新啟動或替換為其他容器。大型應用程序以100%的服務運行，可能需要幾分鐘的改組，但是在那段時間內，該應用程序將在線並為用戶提供服務。圖8.11顯示了生產集群中容器的生命週期的示例。

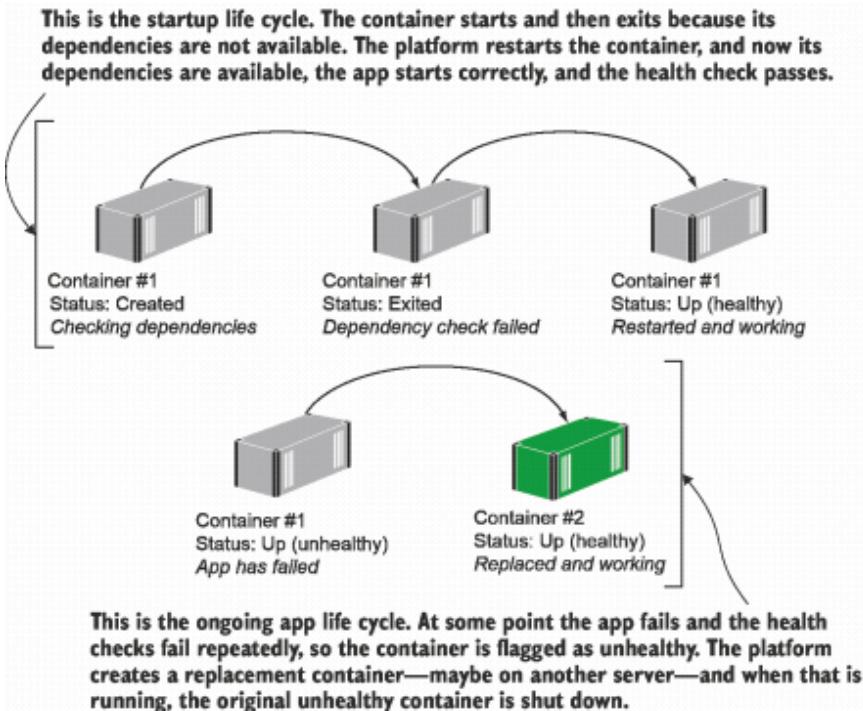


圖8.11 SEL 生產集群中的F-thealing應用程序 - 包含器可以重新啟動

D或更換。

自我修復應用程序的想法是，平台可以處理任何瞬態失敗。如果您的應用程序使其不可能用完存儲器，則該平台將關閉容器，並用具有新的內存分配的新替換。它無法修復錯誤，但可以使應用程序正常工作。

不過，您確實需要謹慎檢查。健康檢查定期進行，因此他們不應該做太多的工作。您需要找到餘額，因此檢查正在測試應用程序的關鍵部分，而無需花費太長運行或使用過多的計算資源。依賴性檢查僅在啟動時運行，因此您無需太擔心它們使用的資源，但是您需要小心檢查的內容。有些依賴項無法控制，如果平台無法修復問題，那麼如果您的容器失敗，則無濟於事。

弄清楚檢查中的邏輯是困難的部分。 Docker可以輕鬆捕獲這些支票並為您執行這些檢查，如果您正確地完成了這些檢查，則容器平台將為您提供運行。

8.6 實驗室

一些應用程序始終使用資源，以便初始依賴性檢查和正在進行的健康檢查正在測試同一件事。這個實驗室就是這樣。這是一個模擬內存豬的應用程序 - 只要它運行，它就會不斷分配並保持更多的內存。這是一個node.js應用程序，需要一些檢查：

- 在啟動時，它應該檢查是否有足夠的內存可以工作；如果沒有，它應該退出。 在運行時，應該檢查一次每5秒鐘，以查看其是否分配了比允許的更多的內存；如果有的話，它需要標記它不健康。 測試邏輯已經寫在Memory-Check.js腳本中。它只需要連接到Dockerfile。 腳本和初始Dockerfile在源文件夾CH08/實驗室中。
-
-

NOTE意，該應用程序並沒有真正分配任何內存。 容器中的內存管理因不同的環境而復雜 - Windows上由Docker Desktop管理的Linux容器與Linux上使用Docker Engine運行的容器不同。 對於此實驗室，該應用程序只是假裝使用內存。

This lab is pretty straightforward. I'll just point out that Node.js apps are not compiled, so you don't need multiple stages. My sample is in the same directory, called Dockerfile.solution, and you'll find the write-up in the book's GitHub repository: <https://github.com/sixeyed/diamol/blob/2e/ch08/lab/README.md>.

9 通過容器化監視 添加可觀察性

自主應用程序會自動擴大自身，以滿足傳入的流量，並且在間歇性故障時會自治。聽起來真是太好了，而且可能是真實的。如果您構建Docker Images或使用健康檢查建模應用程序，則容器平台可以為您完成許多操作，但是您仍然需要進行持續的監控和警報，以便在事情發生錯誤時可以參與其中。如果您對容器化應用程序沒有任何了解，那將是阻止您要生產的第一件事。

當您容器中運行應用程序時，可觀察性是軟件景觀的關鍵部分 - 它告訴您您的應用程序在做什麼以及它們的表現如何，並且可以幫助您查明問題的來源。在本章中，您將學習如何使用良好的方法與Docker進行監視：從應用程序容器中暴露指標，並使用Prometheus收集它們和Grafana，以在用戶友好的儀表板中可視化它們。這些工具是開源和跨平台，它們與您的應用程序一起在容器中運行。這意味著您可以在從開發到生產的每個環境中對應用程序性能進行相同的見解。

9.1 容器化應用程序的監視堆棧

當應用程序在容器中運行時，監視是不同的。在傳統環境中，您可能會有一個監視儀表板，顯示服務器及其當前利用率（盤空間，內存，CPU），並提醒您告訴您是否有過度勞累並可能停止響應。容器化的應用程序更具動態性 - 它們可能會跨越數十個或數百個簡短的容器，這些容器是由容器平台創建或刪除的。

您需要一種具有容器感知的監視方法，可以使用可以插入容器平台進行發現的工具，並找到所有運行的應用程序，而無需靜態的容器IP地址列表。Prometheus是一個開源項目，可以做到這一點。這是一種成熟的產品，由雲本機計算基礎（Kubernetes和Containerd Container Container Runtime的相同基礎）監督。Prometheus在Docker容器中運行，因此您可以輕鬆地在應用程序中添加監視堆棧。圖9.1顯示了堆棧的外觀。

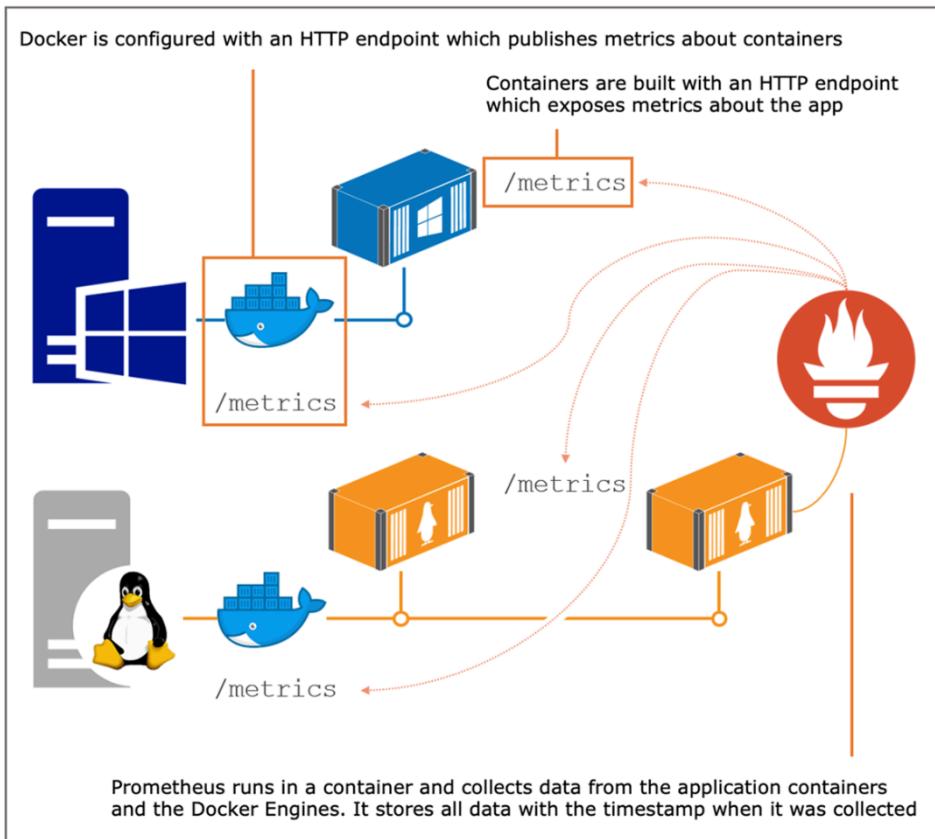


圖9.1在容器中運行Prometheus，以監視其他容器和Docker本身

普羅米修斯為監視帶來了一個非常重要的方面：一致性。 您可以為所有應用程序導出相同類型的指標，因此您有一種標準方法來監視它們是否是Windows容器中的.NET應用程序或Linux容器中的Node.js應用程序。您只有一種查詢語言可以學習，並且可以將其應用於整個應用程序堆棧。

使用Prometheus的另一個很好的理由是，Docker引擎還可以以這種格式導出指標，這也使您可以洞悉容器平台中發生的情況。您需要在Docker Engine配置中明確啟用Prometheus指標 - 您看到瞭如何在第5章中更新配置。您可以在Windows Server上直接編輯C:\ProgramData\ Docker\config在Windows Server上的c:\programData\ docker\config在Linux上的c:\programData\ docker\config。另外，在Docker桌面上，您可以右鍵單擊鯨魚圖標，選擇設置並在Docker Engine部分編輯配置。

現在嘗試

打開您的配置設置並添加一個新值：

```
"Metrics-Addr" : "0.0.0.0:9323"
```

此設置啟用監視並在端口9323上發布指標。

您可以在圖9.2中查看我的完整配置文件。

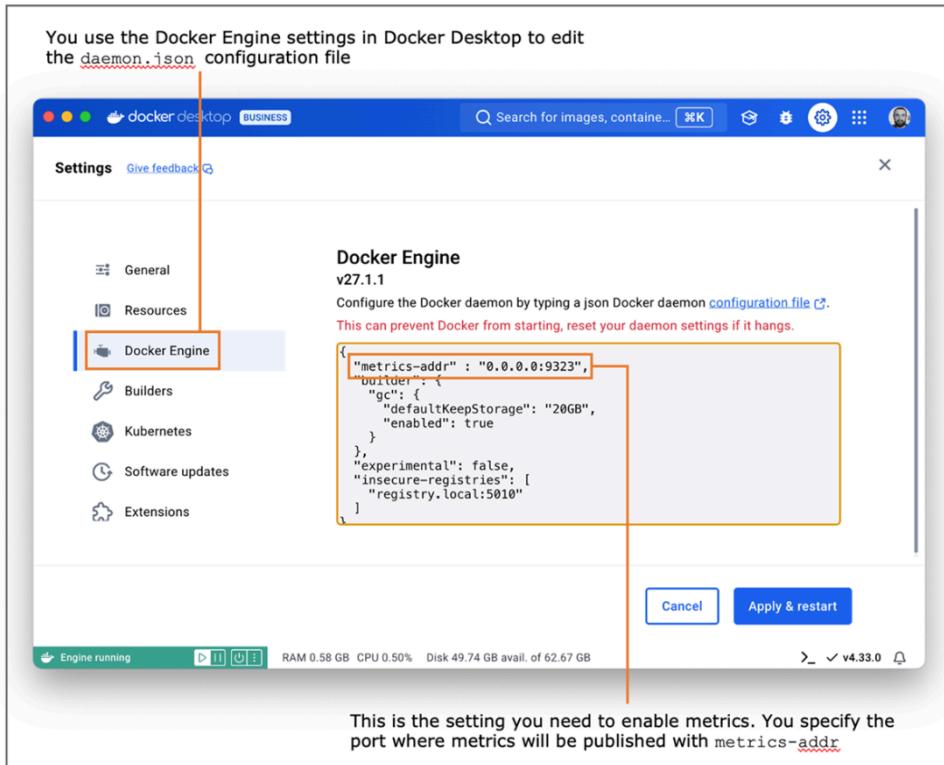


圖9.2將Docker引擎配置為Prometheus格式的導出指標

Docker Engine指標目前是一個實驗功能，這意味著它提供的詳細信息可能會改變。但這是長期以來的實驗功能，而且一直穩定。值得在您的儀表板上包括，因為它為系統的整體健康添加了另一層細節。現在，您已經啟用了指標，可以瀏覽到`http://localhost:9323/量表`，並查看Docker提供的所有信息。圖9.3顯示了我的指標，其中包括有關機器Docker的信息以及Docker正在管理的容器。

Enabling metrics gives you an HTTP endpoint where you can view data from the Docker Engine

```

localhost:9323/metrics
engine_daemon_container_actions_seconds_sum{action="start"} 0
engine_daemon_container_actions_seconds_count{action="start"} 1
# HELP engine_daemon_container_states_containers The count of containers in various states
# TYPE engine_daemon_container_states_containers gauge
engine_daemon_container_states_containers{state="paused"} 0
engine_daemon_container_states_containers{state="running"} 0
engine_daemon_container_states_containers{state="stopped"} 1
# HELP engine_daemon_engine_cpus_cpus The number of cpus that the host system of the engine has
# TYPE engine_daemon_engine_cpus_cpus gauge
engine_daemon_engine_cpus_cpus 10
# HELP engine_daemon_engine_info The information related to the engine and the OS it is running on
# TYPE engine_daemon_engine_info gauge
engine_daemon_engine_info{architecture="aarch64",commit="cc13f95",daemon_id="c68b7035-73d8-4db0-8af6-b707e05125cf",graphdriver="overlay2",kernel="6.10.0-linuxkit",os="Docker Desktop",os_type="linux",os_version="",version="27.1.1"} 1
# HELP engine_daemon_engine_memory_bytes The number of bytes of memory that the host system of the engine has
# TYPE engine_daemon_engine_memory_bytes gauge
engine_daemon_engine_memory_bytes 1.675051008e+10
# HELP engine_daemon_events_subscribers_total The number of current subscribers to events
# TYPE engine_daemon_events_subscribers_total gauge
engine_daemon_events_subscribers_total 8
# HELP engine_daemon_events_total The number of events logged
# TYPE engine_daemon_events_total counter
engine_daemon_events_total 0

```

Metrics include static information, like the amount of memory available on the machine running Docker

And also dynamic information, like the current number of containers in each state - paused, running stopped

圖9.3 Docker捕獲並通過HTTP API暴露的樣品指標

該輸出為普羅米修斯格式。這是一個簡單的基於文本的表示形式，其中每個指標都以其名稱和值顯示，並且該指標在一些幫助文本之前說明了指標是什麼和數據的類型。這些基本文本線是您的容器監控解決方案的核心。每個組件將公開一個提供當前指標的端點；當Prometheus收集它們時，它會在數據中添加時間戳，並將其存儲在所有先前的集合中，因此您可以隨著時間的推移查詢使用聚合或跟蹤更改的數據。

現在嘗試

您可以在容器中運行Prometheus，以閱讀Docker Machine的指標，但首先您需要獲取機器的IP地址。容器不知道他們正在運行的服務器的IP地址，因此您需要首先找到它並將其作為環境變量傳遞給容器：

```
# 將計算機的IP地址加載到變量中 - 在Windows : $ hostip = $(get -netipConfiguration | where-Object {$_ -like "IPV4DEFAULTGATEWAY-NE $ NEL $ NEULL})  
  
# 在Linux上：  
hostip = $(ip route | awk'{print $ nf; exit}')  
  
# 和Mac上：  
hostip = $(ifconfig en0 | grep -e'inet \s+' | awk'{print $ 2  
}'')  
  
# 將您的IP地址作為容器的環境變量傳遞：Docker容器運行-e Docker_host = $ HOSTIP -D -P 9090:90  
90 DIAMOL/PROMETHEUS :2E
```

Diamol/Prometheus Prometheus圖像中的配置使用Docker_主機IP地址與您的主機通信並收集您在Docker Engine中配置的指標。 您很少需要從容器內部訪問主機上的服務，如果這樣做，通常會使用服務器名稱，docker會找到IP地址。 在可能不起作用的開發環境中，IP地址方法應該很好。

普羅米修斯現在正在運行。 它做了幾件事：它運行一個計劃的作業以從Docker主機中提取指標，它將這些指標值與時間戳一起存儲在其自己的數據庫中，並且它具有基本的Web UI，您可以使用用於導航指標。 Prometheus UI顯示了Docker /指標端點中的所有信息，您可以過濾指標並將其顯示在表或圖表中。

現在嘗試

瀏覽到http://localhost:9090，您將看到Prometheus Web界面。您可以檢查Prometheus是否可以通過瀏覽狀態>目標菜單選項來訪問指標。 您的docker_host狀態應該是綠色的，這意味著普羅米修斯已經找到了它。然後切換到圖形菜單，您將看到一個下拉列表，顯示Prometheus從Docker收集的所有可用指標。其中之一是Engine_daemon_container_actions_seconds_sum，它是採取了不同的容器操作多長時間的記錄。選擇該指標並單擊執行，並且您的輸出將與圖9.4中的我相似，顯示了創建，刪除和啟動容器所花費的時間。

The Prometheus UI lets you run queries and quickly see the results.
Expressions can just be a metric name or can be complex queries in PromQL.

This dropdown shows the names of all the metrics that have been collected.

The screenshot shows the Prometheus UI interface. At the top, there's a navigation bar with tabs for Prometheus, Alerts, Graph, Status, and Help. Below the navigation bar, there's a search bar containing the text "engine_daemon_container_actions_seconds_sum". To the right of the search bar, it says "Load time: 12ms", "Resolution: 14s", and "Total time series: 5". Below the search bar is a button labeled "Execute". Underneath the search bar, there are two tabs: "Graph" and "Console". The "Console" tab is selected. In the console area, there's a table with two columns: "Element" and "Value". The table contains five rows of data. The last four rows are highlighted with a red border. The data is as follows:

Element	Value
engine_daemon_container_actions_seconds_sum[action="changes",instance="DOCKER_HOST:9323",job="docker"]	0
engine_daemon_container_actions_seconds_sum[action="commit",instance="DOCKER_HOST:9323",job="docker"]	0
engine_daemon_container_actions_seconds_sum[action="create",instance="DOCKER_HOST:9323",job="docker"]	2.3447196999999997
engine_daemon_container_actions_seconds_sum[action="delete",instance="DOCKER_HOST:9323",job="docker"]	0.2621732999999994
engine_daemon_container_actions_seconds_sum[action="start",instance="DOCKER_HOST:9323",job="docker"]	8.067746800000002

Console view shows the results of the query in a table, with just the most recent values. You can switch to Graph view and see the changing values over time.

圖9.4 Prometheus具有一個簡單的Web UI，您可以使用它來查找指標和運行查詢。

Prometheus UI是查看正在收集並運行一些查詢的簡單方法。環顧指標，您會看到Docker記錄了很多信息點。有些是高級讀數，例如每個州的容器數量以及失敗的健康檢查數量；其他人給出了低級細節，例如Docker Engine分配的記憶量；有些是靜態信息，例如CPU Docker的數量可用。這些是基礎架構級指標，所有這些都可能是在狀態儀表板中包含的有用項目。

您的應用程序將公開自己的指標，這也將記錄不同級別的詳細信息。目的是在每個容器中都有一個指標端點，並讓Prometheus按照時間表從他們那裡收集指標。Prometheus將存儲足夠的信息供您構建一個儀表板，以顯示整個系統的整體健康狀況。

9.2 從您的應用程序中暴露指標

我們已經研究了Docker Engine的指標，因為這是一種開始使用Prometheus的簡單方法。從每個應用程序容器中曝光一組有用的指標需要更多的努力，因為您需要代碼來捕獲指標並為Prometheus提供HTTP端點。聽起來不那麼工作，因為所有主要的編程語言都有Prometheus客戶庫可以為您做到這一點。

在本章的代碼中，我重新訪問了NASA Image Gallery應用程序，並在每個組件中添加了Prometheus指標。我正在使用Java和Go的Prometheus官方客戶以及Node.js的社區客戶庫。圖9.5顯示了現在如何將每個應用程序容器與Prometheus客戶端包裝，該客戶端收集和公開指標。

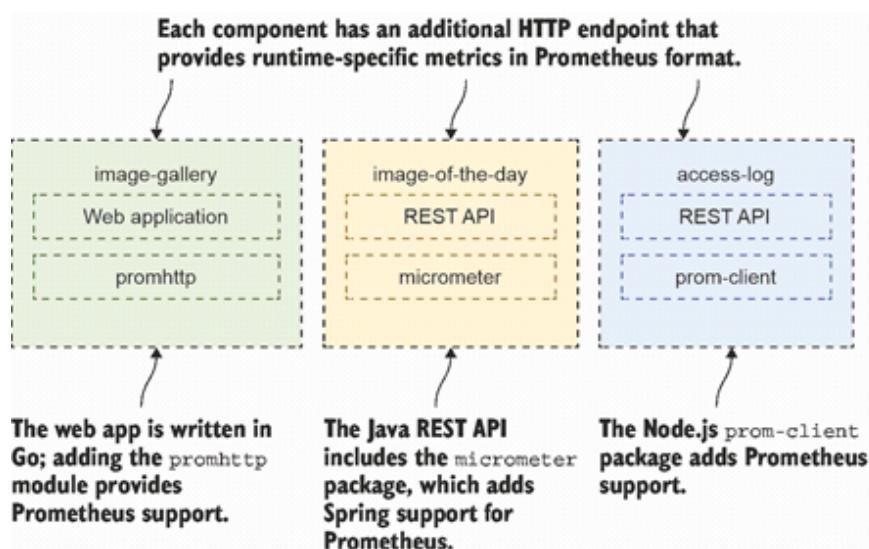


圖9.5 PROMETHEUS：您的應用程序中的Heus客戶端庫使指標端點可用

n容器。

從普羅米修斯客戶端庫中收集的信息點是運行時級指標。他們提供了有關您的容器在做什麼以及它的工作能力的關鍵信息，這與應用程序運行時有關。GO應用程序的指標包括活動goroutines的數量；Java應用程序的指標包括JVM中使用的內存。每個運行時都有其自己的重要指標，客戶庫在收集和導出這些方面做得很好。

現在嘗試

本章的練習中有一個Docker撰寫文件，該文件插入了圖庫應用程序的新版本，每個容器中都有指標。使用該應用，然後瀏覽到指標端點之一：

```
CD ./ch09/cercises
```

```
# 清除現有容器：Docker容器RM -F $ (Docker Container LS -  
AQ)
```

```
# 創建NAT網絡 - 如果您已經創建了它 # 您會收到一個可以忽略的警  
告：Docker Network創建NAT
```

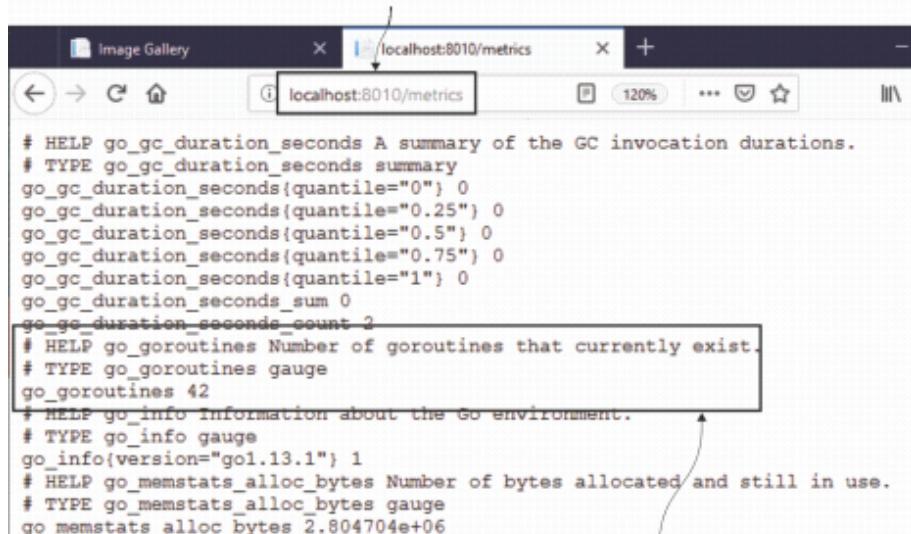
```
# 啟動項目docker -compo  
se -d
```

```
# 瀏覽到http://localhost:8010用於使用該應用
```

```
# 然後瀏覽到http://localhost:8010/量表
```

我的輸出在圖9.6中。這些是GO Frontend Web應用程序中的指標 - 生成此數據並不需要自定義代碼。您只需將GO Client庫中添加到您的應用程序中並設置它，就可以免費獲取所有這些數據。

Adding the Prometheus client library enables metrics in the application container.



The screenshot shows a web browser window with the URL `localhost:8010/metrics`. The page displays a list of Prometheus metrics in text format. The metrics include:

```
# HELP go_gc_duration_seconds A summary of the GC invocation durations.  
# TYPE go_gc_duration_seconds summary  
go_gc_duration_seconds{quantile="0"} 0  
go_gc_duration_seconds{quantile="0.25"} 0  
go_gc_duration_seconds{quantile="0.5"} 0  
go_gc_duration_seconds{quantile="0.75"} 0  
go_gc_duration_seconds{quantile="1"} 0  
go_gc_duration_seconds_sum 0  
go_gc_duration_seconds_count 2  
# HELP go_goroutines Number of goroutines that currently exist.  
# TYPE go_goroutines gauge  
go_goroutines 42  
# HELP go_info information about the Go environment.  
# TYPE go_info gauge  
go_info{version="go1.13.1"} 1  
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.  
# TYPE go_memstats_alloc_bytes gauge  
go_memstats_alloc_bytes 2.804704e+06
```

A callout arrow points from the text "Metrics are in the same Prometheus text format but the values are specific to the application runtime—this is a count of active Goroutines in the Go application." to the line `go_goroutines 42`.

Metrics are in the same Prometheus text format but the values are specific to the application runtime—this is a count of active Goroutines in the Go application.

圖9.6 Prometheus關於從圖庫Web容器中的關於GO運行時的Prometheus指標

如果您瀏覽到http://localhost:8011/cartuator/prometheus，您會看到Java Rest API的類似指標。指標端點是文本的海洋，但是所有關鍵數據點都在那裡構建一個儀表板，該儀表板將顯示是否使用了許多計算資源，例如CPU時間，內存或處理器線程，它們是否正在運行“熱”。

這些運行時指標是Docker的基礎架構指標之後您想要的下一個細節，但是這兩個級別並沒有告訴您整個故事。最終數據點是您明確捕獲以記錄有關應用程序的關鍵信息的應用程序指標。這些指標可能是針對操作的，顯示了組件已經處理的事件數量或處理響應的平均時間。或者他們可能以業務為中心，顯示當前活躍用戶的數量或註冊新服務的人數。

Prometheus客戶端庫也可讓您記錄這類指標，但是您需要明確編寫代碼以捕獲應用程序中的信息。這並不難。列表9.1使用node.js庫顯示了一個示例，該庫是Image Gallery應用程序中訪問-日誌組件的代碼中的示例。我不想向您投入一大堆代碼，但是隨著您在容器中進一步發展時，您肯定會花更多的時間與Prometheus一起使用，而該片段中的摘要則說明了一些關鍵的內容。

清單9.1在否中聲明並使用自定義Prometheus度量值

de.js

```
//聲明自定義指標：const AccessCounter = new Prom.Counter ({name : “access_log_total”，help：“access log -log -log log requests”})；const clientipgauge = new prom.gauge ({name：“access_client_ip_current”，help：“access log-當前唯一ip地址”})；//然後，更新指標值：accessCounter.inc ()；clientipgauge.set (countofipAddresses)；
```

在本章的源代碼中，您將看到我如何在GO編寫的Image-Gallery Web應用程序以及用Java編寫的“映像REST API”中添加指標。每個Prometheus客戶庫都以不同的方式工作。在main.go源文件中，我以與node.js應用相似的方式初始化計數器和儀表，但隨後使用客戶端庫中的儀器處理程序，而不是明確設置指標。java應用程序再次不同 - 在imagecontroller.java中，我使用@timed屬性並遞增源中的註冊表。每個客戶庫以最合乎邏輯的方式為語言工作。

普羅米修斯中有不同類型的指標 - 我在這些應用程序中使用了最簡單的指標：計數器和儀表。它們都是數字值。計數器具有增加或保持不變的值，並且儀表具有可以增加或減少的值。您或您的應用程序開發人員選擇度量類型並在正確的時間設置其價值；其餘的由Prometheus和客戶庫來照顧。

現在嘗試

您的圖像庫應用程序從上一個練習中運行，因此已經收集了這些指標。在應用程序中運行一些負載，然後瀏覽到Node.js應用程序的指標端點：

```
# 循環製作5 http獲取請求 - 在Windows上：for ($ i = 1; $ i -le 5; $ i++) {iwr -useb http http://localhost:8010 | of -null}

# 或linux上：
因為我在{1..5}中; curl http://localhost:8010 >/dev/null;完畢

# 現在瀏覽到http://localhost:8012/量表
```

您可以在圖9.7中看到我的輸出 - 我跑了一些循環以發送流量。前兩個記錄顯示了我的自定義指標，記錄收到的訪問請求數以及使用該服務的IP地址總數。這些是簡單的數據點（IP計數實際上是假的），但它們的目的是收集和顯示指標。Prometheus允許您記錄更複雜的指標類型，但是即使使用簡單的計數器和衡量，您也可以在應用程序中捕獲詳細的儀器。

The same metrics endpoint provides my custom values and the standard Node.js data.

```
# HELP access_log_count Access Log - count of log requests
# TYPE access_log_count counter
access_log_count 32

# HELP access_client_ip_gauge Access Log - unique IP addresses
# TYPE access_client_ip_gauge gauge
access_client_ip_gauge 15

# HELP process_cpu_user_seconds_total Total user CPU time spent in seconds.
# TYPE process_cpu_user_seconds_total counter
process_cpu_user_seconds_total 0.3130000000000017 1572378573914

# HELP process_cpu_system_seconds_total Total system CPU time spent in seconds.
# TYPE process_cpu_system_seconds_total counter
process_cpu_system_seconds_total 0.047 1572378573914

# HELP process_cpu_seconds_total Total user and system CPU time spent in seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 0.3600000000000001 1572378573914
```

Node.js runtime metrics are provided by the client library—this shows total amount of CPU time used.

圖9.7一個指標端點，包括自定義數據以及node.js運行時數據

您捕獲的內容取決於您的應用程序，但是以下列表提供了一些有用的準則 - 您可以在本月底返回這些指南，當您準備將詳細的監視添加到自己的應用程序中。

- 當您與外部系統交談時，請記錄電話需要多長時間以及響應是否成功 - 您將很快就可
以查看另一個系統是否正在減慢您的速度或破壞它。 任何值得記錄的東西都可能
值得錄製在度量標準中，而在內存，磁盤和CPU上可能會更便宜地增加計數器，而
不是編寫日誌條目，並且更容易地可視化發生的事情的頻率。 有關業務團隊想要報
告的應用程序或用戶行為的任何詳細信息都應記錄為指標 - 您可以構建實時儀表板而
不是發送歷史報告。
-

9.3運行普羅米修斯容器以收集指標

普羅米修斯使用拉動模型收集指標。 它沒有讓其他系統發送數據，而是從這些系統中獲取數據。 它調用此過程*scraping*，當您部署Prometheus時，您將配置要刮擦的端點。 在生產容器平台中，您可以配置Prometheus，以便自動找到整個集群上的所有容器。在Docker在單個服務器上撰寫中，您可以使用一個簡單的服務名稱列表，Prometheus通過Docker的DNS找到容器。

列表9.2顯示了我用於Prometheus的配置，用於刮擦我的Image Gallery應用程序中的兩個組件。有一個全局設置，它使用刮擦之間的默認10秒間隔，然後每個組件都有一個作業。該作業有一個名稱，配置指定了指標端點的URL路徑，以及Prometheus會查詢的目標列表。我在這裡使用兩種類型。首先，static_configs指定目標主機名，這對於一個容器來說很好。我還使用DNS_SD_CONFIGS，這意味著Prometheus將使用DNS服務發現 - 它將找到用於服務的多個容器，並且它支持按大規模運行。

清單9.2刮擦應用程序指標的Prometheus配置

```
global:
  scrape_interval: 10s

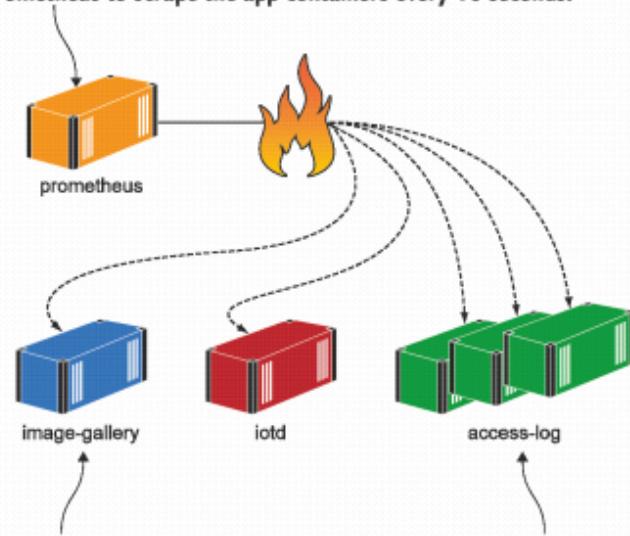
scrape_configs:
  - job_name: "image-gallery"
    metrics_path: /metrics
    static_configs:
      - targets: ["image-gallery"]

  - job_name: "iotd-api"
    metrics_path: /actuator/prometheus
    static_configs:
      - targets: ["iotd"]

  - job_name: "access-log"
    metrics_path: /metrics
    dns_sd_configs:
      - names:
          - accesslog
        type: A
        port: 80
```

這種配置將Prometheus設置為每10秒鐘一次對所有容器進行輪詢。它將使用DNS獲取容器IP地址，但是對於圖像 - 套件，它只希望找到一個容器，因此，如果擴展該組件，您將獲得意外的行為。如果DNS響應包含幾個，Prometheus始終使用列表中的第一個IP地址，因此當Docker負載將請求平衡到指標端點時，您將從不同的容器中獲得指標。AccessLog組件配置為支持多個IP地址，因此Prometheus將構建所有容器IP地址的列表，並按照同一時間表進行輪詢。圖9.8顯示了刮擦過程的運行方式。

The Docker image is packaged with configuration for Prometheus to scrape the app containers every 10 seconds.



There are single instances of the Go and Java components. Prometheus is using static_config so it only expects one IP address for the domain name.

The Node.js component is running at scale over three containers. Prometheus is using dns_sd_config, which means service discovery through DNS, so it will use all the IP addresses Docker returns from the DNS lookup.

圖9.8在容器中運行的Prometheus，配置為來自應用程序容器的刮擦度量

我為圖像庫應用程序構建了一個自定義的Prometheus docker映像。它基於Prometheus團隊在Docker Hub上發布的官方圖像，並在我自己的配置文件中複制（您可以在本章的源代碼中找到Dockerfile）。這種方法為我提供了一個預先配置的Prometheus圖像，我可以在沒有任何額外配置的情況下運行，但是如果需要的話，我總是可以在其他環境中覆蓋配置文件。

當許多容器運行時，指標會更有趣。我們可以將圖像庫應用程序的Node.js組件擴展到多個容器上運行，Prometheus將從所有容器中刮擦並收集指標。

現在嘗試

本章的練習文件夾中還有另一個Docker撰寫文件，該文件發布了用於訪問運行服務的隨機端口，以便可以大規模運行該服務。用三個實例運行它，並向網站發送更多負載：

```
docker-compose -f docker-compose-scale.yml up -dcale accessLog = 3

# 循環進行10 http獲取請求 - 在Windows上：for ($ i = 1; $ i -le 10; $ i++) {iwr -useb http http
:// localhost : 8010 | of -null}

# 或linux上：
因為我在{1..10}中; curl http : // localhost : 8010 >/dev/null;完畢
```

該網站在每次處理請求時都會撥打訪問量的服務 - 運行該服務的三個容器，因此呼叫應在所有這些服務中負載平衡。我們如何檢查負載平衡是否有效地工作？該組件的指標包括一個標籤，該標籤捕獲了發送指標的機器的主機名 - 在這種情況下，這是Docker容器ID。打開Prometheus UI並檢查Access-Log指標。您應該看到三組數據。

現在嘗試

瀏覽到http : // localhost : 9090/Graph。在“指標”下拉列表中，選擇access_log_total，然後單擊“執行”。

您會看到與圖9.9中的輸出相似的東西 - 每個容器中都有一個度量值，並且標籤包含主機名。每個容器的實際值將向您展示如何均勻地傳播負載平衡。在理想的情況下，數字是相等的，但是有很多網絡因素正在播放（例如DNS緩存和HTTP保持距離連接），這意味著您可能不會看到如果您在一台計算機上運行。

This is the custom counter in the Node.js application, which records how many requests it has processed.

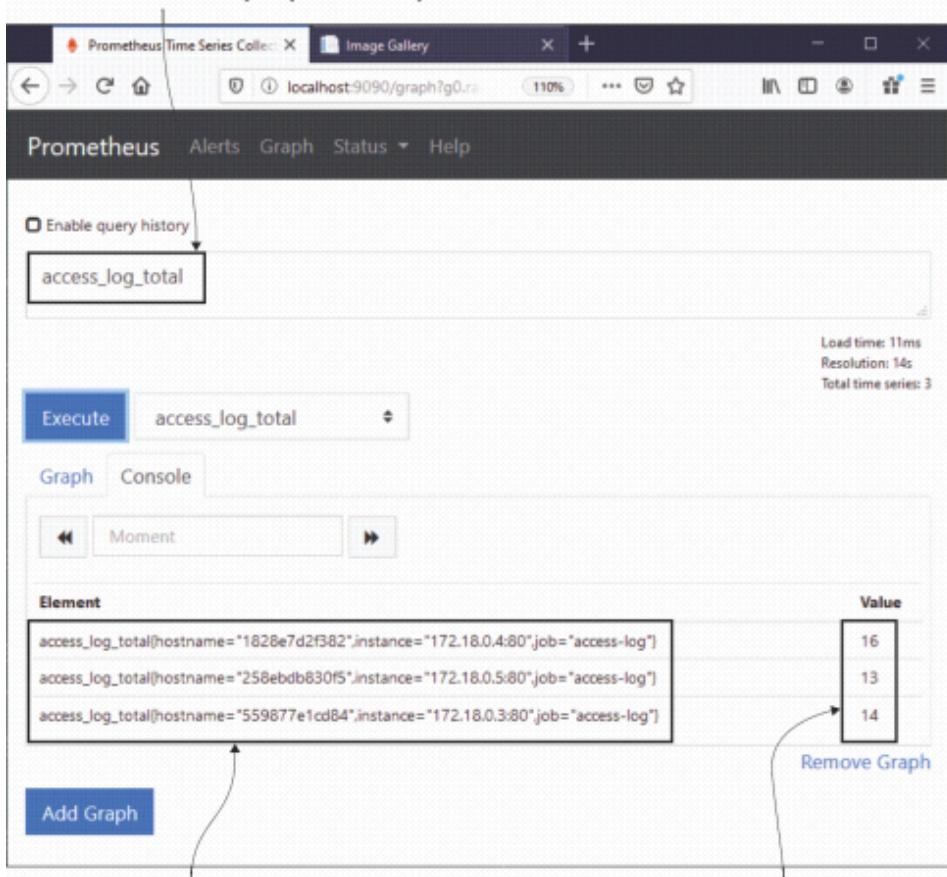


圖9.9處理指標可用於驗證請求是否正在負載平衡。

使用標籤錄製額外信息是普羅米修斯最有力的功能之一。它使您可以在不同水平的粒度上使用單個度量。目前，您正在看到指標的原始數據，每個容器中的一行中有一行顯示最新的度量值。您可以使用`sum()`查詢在所有容器上匯總所有容器，忽略單個標籤並顯示組合的總數，並且可以在圖中顯示，以查看隨著時間的推移使用量的增加。

現在嘗試

在Prometheus UI中，單擊“添加圖”按鈕以添加新查詢。在表達式文本框中，粘貼此查詢：

```
sum (access_log_total) 不帶 (主機名, 實例)
```

單擊執行，您將看到一個帶有時間序列的行圖，這就是Prometheus代表數據的方式 - 一組指標，每個指標都使用時間戳記錄。

I sent in some more HTTP requests to my local app before I added the new graph—you can see my output in figure 9.10.



圖9.10匯總了一個來自所有容器的總和值並顯示結果圖

`sum()` 查詢是用普羅米修斯自己的查詢語言編寫的，稱為*PromQL*。這是一種強大的語言，具有統計功能，可讓您查詢隨著時間和變化速率的變化，並且可以添加子查詢以關聯不同的指標。但是，您無需進入任何複雜性即可建造有用的儀表板。Prometheus格式的結構非常好，您可以通過簡單查詢可視化關鍵指標。您可以使用標籤來過濾值，並將結果匯總為匯總，只有這些功能將為您提供有用的儀表板。

圖9.11顯示了一個典型的查詢，該查詢將輸入儀表板。這匯總了所有image_gallery_request指標的值，過濾響應代碼為200的位置，而沒有實例標籤，因此我們將從所有容器中獲取指標。結果將是所有運行圖片庫Web應用程序的容器發送的200個“確定”響應的總數。

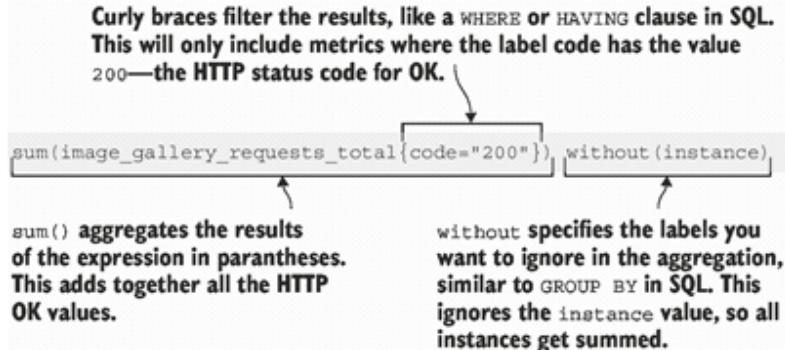


圖9.11簡單的Prometheus查詢。您不需要比這更多的promql學到更多。

Prometheus UI非常適合檢查您的配置，驗證所有刮擦目標都是可以達到的，並確定了查詢。但這並不意味著是儀表板，這就是Grafana進來的地方。

9.4運行格拉法納容器以可視化指標

我們在本章中涵蓋了很多基礎，因為監視是容器的核心主題，但是我們要迅速進行，因為更細節的細節都非常依賴於應用程序。您需要捕獲的指標取決於您的業務和運營需求，以及如何捕獲它們將取決於您使用的應用程序運行時以及該運行時Prometheus客戶端庫的機制。

一旦您將數據掌握在Prometheus中，事情就會變得更加簡單 - 它成為所有應用程序的相當標準方法。您將使用Prometheus UI來瀏覽所錄製的指標，並在查詢中進行查詢以獲取您想要查看的數據。然後，您將運行Grafana並將這些查詢插入儀表板中。每個數據點顯示為用戶友好的可視化，整個儀表板向您顯示了應用程序正在發生的事情。

通過本章，我們一直在為圖庫應用程序的Grafana儀表板努力，圖9.12顯示了最終結果。這是顯示所有應用程序組件和Docker運行時的核心信息的整潔方法。這些查詢也是為了支持刻度的構建，因此可以在生產群集中使用相同的儀表板。

The full Grafana dashboard for the application. Each visualization is populated by a simple Prometheus query. There is a dashboard row for each component, showing key metrics.



The final row shows information about the Docker Engine, including container metrics and configuration details.

圖9.12應用程序的Grafana儀表板。看起來很花哨，但實際上它的構建非常簡單。

Grafana儀表板在應用程序的許多不同級別上傳達了關鍵信息。它看起來很複雜，但是每個可視化都由一個promql查詢供電，並且沒有一個查詢比過濾和聚合更複雜。圖9.12中的縮水視圖並不能為您提供完整的圖片，但是我將儀表板打包到了自定義的Grafana映像中，因此您可以在容器中運行並探索。

現在嘗試

您需要再次捕獲計算機的IP地址，這次是撰寫文件的環境變量，並將其註入Prometheus容器中。然後使用Docker組成並生成一些負載：

```
#將計算機的IP地址加載到環境變量中 - 在Windows : $ enk : host_ip = $(get -netipconfiguration | where-object {$__.IPV4DEFAULTGATEWAY-NE -NE $ NULL}) 。
```

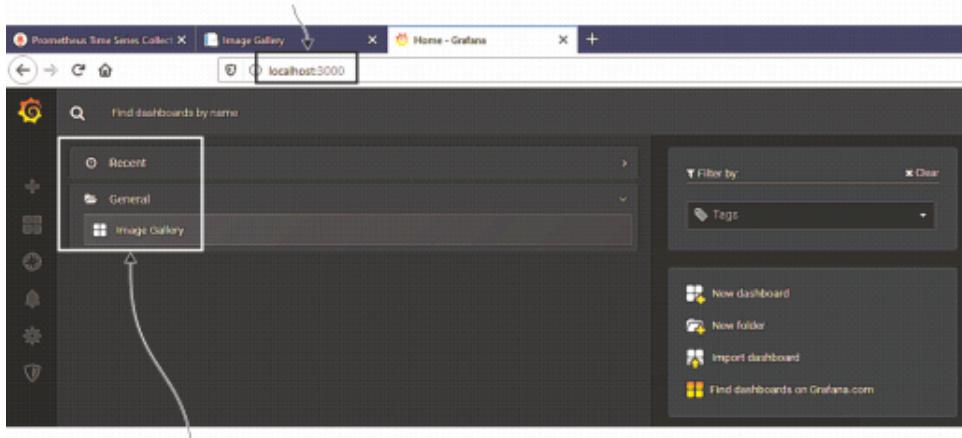
```
#在Linux上 :
```

```
導出host_ip = $(ip路由獲取1 | awk'{print $ nf; exit}') # # 使用包含grafana : docker-compose -f ./  
docker-compose-compose-compose-with-with-with-with-with-with-with-with-with-with-with-with-with-with-with-with-with-with-with-d-d-d-d-d-scale accessLog { ($ i = 1; $ i -le 20; $ i ++ ) {iwr -useb http://localhost:8010  
| of -null} #或on linux : for i in {1..20} in {1..20};請curl http://localhost:8010 >/dev/null;完畢
```

```
#並瀏覽到http://localhost:3000
```

Grafana將端口3000用於Web UI。首次瀏覽時，您需要登錄 - 憑據是用戶名管理員，密碼管理員。 您將被要求在第一次登錄時更改管理員密碼，但是如果您單擊跳過，我不會判斷您。 當UI加載時，您將在“家”儀表板中 - 單擊左上方的主頁鏈接，並且您會在圖9.13中看到儀表板列表。單擊圖像庫以加載應用程序儀表板。

The Docker image for this container is preconfigured with Grafana and the dashboard for the application.



Grafana can show many dashboards. From the Home link you can list all available dashboards.

圖9.13在Grafana中導航儀表板 - 此處顯示了使用的文件夾

我的應用儀表板是生產系統的合理設置。您需要一些關鍵數據點，以確保您正在監視正確的內容 - Google在*Site Reliability Engineering* book (<http://mng.bz/edzj>) 中討論了這一點。他們的重點是他們稱之為“黃金信號”的延遲，流量，錯誤和飽和度。

我將詳細介紹我的第一組可視化，以便您可以看到可以從基本查詢和正確的可視化選擇中構建智能儀表板。圖9.14顯示了圖庫Web UI的指標行 - 我將行切碎以使其更容易看到，但是這些顯示在儀表板上的同一條線上。

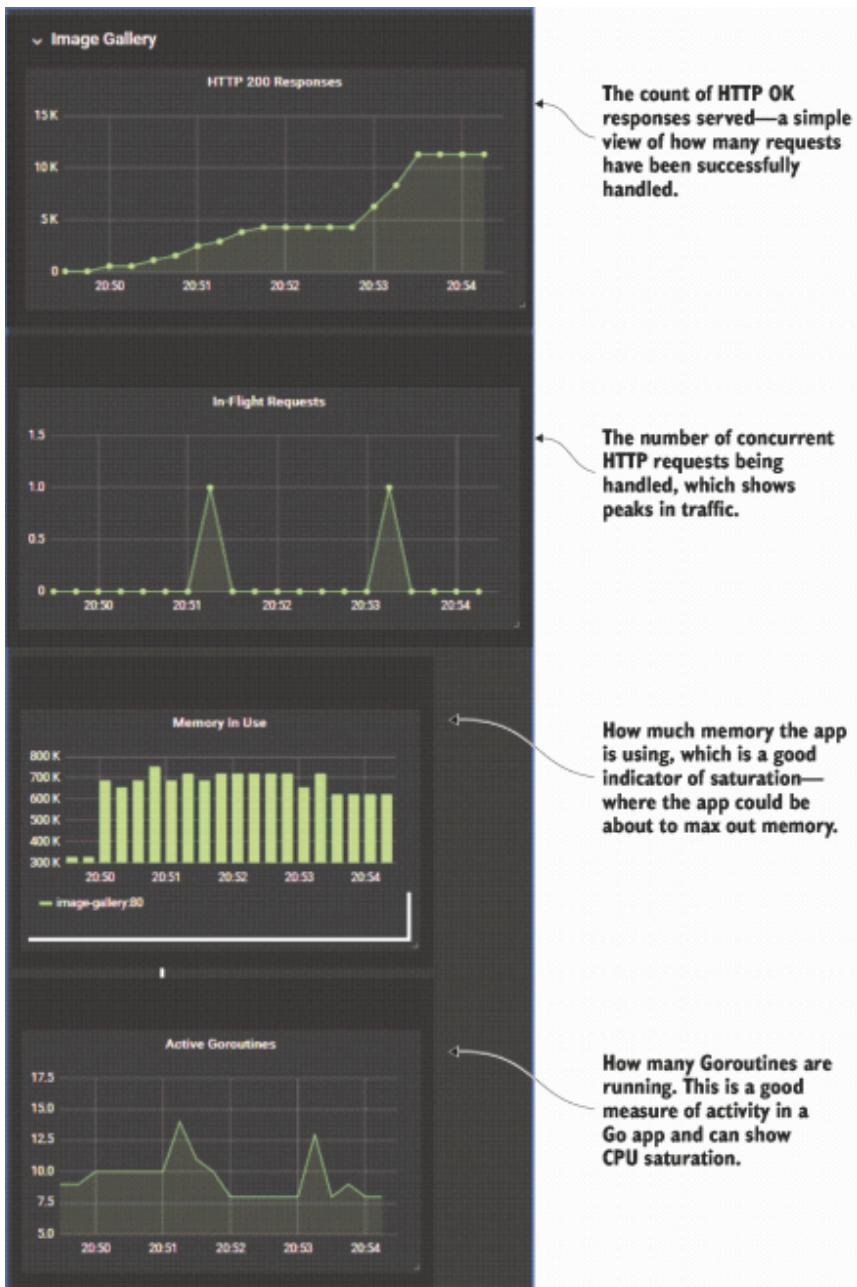


圖9.14仔細觀察應用儀表板以及可視化與金信號的關係

這裡有四個指標表明系統的使用程度如何，以及系統在支持該級別的使用方面的工作程度：

- *HTTP 200 Responses* - 這是一個簡單的計數，即該網站隨著時間的推移發送了多少http “確定”響應。PROMQL查詢是來自應用程序的計數器度量的總和：sum (image_gallery_requests_total {code = “200”}) 不帶（實例）。我可以在代碼=“500”上使用查詢過濾添加類似的圖表以顯示錯誤數。
- *In-Flight Requests* - 此顯示在任何給定點處的活動請求數。這是一個普羅米修斯的儀表，因此可以上下移動。沒有任何過濾器，該圖將顯示所有容器的總數，因此查詢是另一個總和：總和 (image_gallery_in_flight_requests) 不帶（實例）。
- *Memory In Use* - 此顯示圖像庫容器正在使用的系統內存多少。這是條形圖，此類型的數據更容易。當我擴大Web組件時，它將顯示每個容器的條。在作業名稱上的Promql查詢過濾器：go_memstats_stack_inuse_bytes {job = “image-gallery”}。我需要過濾器，因為這是一個標準的GO指標，並且Docker Engine的作業返回具有相同名稱的指標。
- *Active Goroutines* - 這是組件的努力工作的粗略指示 - goroutine是GO中的工作單位，許多人可以同時運行。該圖將顯示Web組件是否突然具有處理活動的激增。這是另一個標準go度量標準，因此promql查詢從Web作業中的統計數據並總結它們：sum (go_goroutines {job = \ “image-gallery”})，沒有（實例）。
-

儀表板其他行中的可視化都使用類似的查詢。不需要復雜的Promql-選擇要顯示的正確指標，並且真正需要的是正確的可視化來顯示它們。

在這些可視化中，實際值不如趨勢有用。無論我的Web應用程序平均使用200 MB的內存還是800 MB，這並不重要 - 當突然出現尖峰偏離常規時。組件的一組指標應幫助您快速查看異常並找到相關性。如果錯誤響應的圖表處於向上趨勢，並且活動goroutines的數量每隔幾秒鐘增加一倍，那麼很明顯，出現了問題 - 組件可能會飽和，因此您可能需要使用更多容器來擴展以處理負載。

Grafana是一種非常強大的工具，但它很容易使用。它是用於現代應用程序的最受歡迎的儀表板系統，因此值得學習 - 它可以查詢許多不同的數據源，並且也可以將警報發送到不同的系統。構建儀表板與編輯現有儀表板相同 - 您可以添加或編輯可視化（稱為*panels*），調整併移動它們，然後將儀表板保存到文件中。

現在嘗試

Google SRE方法說，HTTP錯誤計數是一個核心度量標準，儀表板中缺少它，因此我們現在將其添加到Image Gallery行中。如果您沒有運行，請再次運行整個圖像庫應用程序，請通過http://localhost瀏覽Grafana：3000，並使用用戶名管理員和密碼管理員登錄。

打開圖像庫儀表板，然後單擊屏幕右上角的添加面板圖標 - 這是帶有圖9.15中的加號的條形圖。



圖9.15用於添加面板，選擇時間段並保存儀表板的Grafana工具欄

現在，單擊“新面板”窗口中的添加查詢，您將看到一個屏幕，您可以在其中捕獲可視化的所有詳細信息。選擇Prometheus作為查詢的數據源，在指標字段中粘貼此Promql表達式：

```
sum (image_gallery_requests_total {code = "500"}) 不帶 (insta) NCE
```

您的面板應該看起來像我的圖9.16中的我。圖片庫應用程序在10%的時間左右返回錯誤響應，因此，如果您提出了足夠的請求，則會在圖表中看到一些錯誤。

按下逃鍵返回主儀表板。

The Grafana screen to add a new panel. You choose the type of visualization, add the title and the legend, and provide the query that Grafana runs to populate the panel.



圖9.16向Grafana儀表板添加一個新面板以顯示HTTP錯誤

您可以通過拖動右下角來調整面板大小，並通過拖動標題來移動它們。當您擁有儀表板時，您可以單擊工具面板中的共享儀表板圖標（請參見圖9.15），在那裡您可以選擇將儀表板導出為JSON文件。

Grafana的最後一步是包裝您自己的Docker映像，該圖像已經配置為Prometheus作為數據源和應用程序儀表板。我已經為Diamol/CH09-Grafana圖像做到了這一點。清單9.3顯示了完整的Dockerfile。

列表9.3包裝自定義Grafana圖像的Dockerfile

來自Diamol/Grafana：6.4.3

```
COPY datasource-prometheus.yaml ${GF_PATHS_PROVISIONING}/datasources/ COPY d  
ashboard-provider.yaml ${GF_PATHS_PROVISIONING}/dashboards/ COPY dashboard.json  
n /var/lib/grafana/dashboards/
```

該圖像從特定版本的Grafana開始，然後僅在一組YAML和JSON文件中複制。 Grafana遵循我在本書中已經宣傳的配置模式 - 內置了一些默認配置，但是您可以應用自己的配置模式。 當容器啟動時，Grafana在特定文件夾中尋找文件，並應用其找到的任何配置文件。 YAML文件設置了Prometheus連接並加載/var/lib/grafana/dashboards文件夾中的所有儀表板。 最後一行將我的儀表板json複製到該文件夾中，因此當容器啟動時它會加載。

您可以通過Grafana Presisioning做更多的工作，還可以使用API創建用戶並設置其偏好。 使用多個儀表板和只讀用戶構建Grafana映像並沒有更多的工作，可以訪問所有這些儀表板，這些儀表板可以放在Grafana播放列表中。 然後，您可以在辦公室的大屏幕上瀏覽Grafana，並使其自動循環通過所有儀表板。

9.5了解可觀察性水平

當您從簡單的概念驗證容器轉變為準備生產時，可觀察性是關鍵要求。 但是，我在本章中介紹了Prometheus和Grafana的另一個很好的理由：Learning Docker不僅與Dockerfiles和Docker組成的文件有關。 Docker的魅力的一部分是圍繞容器生長的巨大生態系統，以及該生態系統周圍出現的模式。

當容器首次流行時，監視是一個真正的頭痛。 當時我的生產發布與今天一樣容易構建和部署，但是我在運行時對應用程序沒有任何深刻的了解。 我不得不依靠像pingdom這樣的外部服務來檢查我的API是否仍在啟動，並且用戶報告以確保應用程序正常工作。 如今，監視容器的方法是經過久經考驗的路徑。 我們在本章中遵循了這一道路，圖9.17總結了該方法。

Grafana configured with a dashboard to visualize key metrics. It fetches the data from Prometheus using PromQL queries for each dashboard panel.

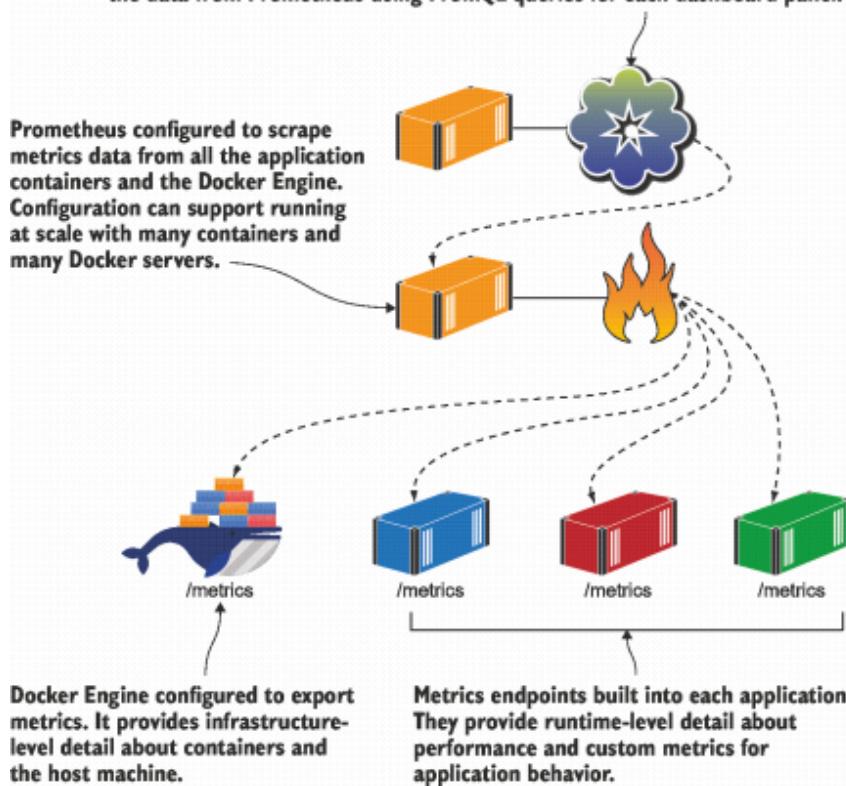


圖9.17在容器化應用程序中監視的體系結構 - prometheus位於中心。

我已經瀏覽了一個儀表板以供圖庫應用程序，這是該應用程序的總體視圖。在生產環境中，您還會有其他儀表板來挖掘額外的細節水平。將有一個基礎架構儀表板顯示免費的磁盤空間，可用的CPU以及所有服務器的內存和網絡飽和度。每個組件可能都有自己的儀表板顯示其他信息，例如用於服務Web應用程序的每個頁面或每個API端點的響應時間分解。

摘要儀表板是關鍵的儀表板。您應該能夠將所有最重要的數據點從應用程序指標中匯總到一個屏幕上，因此您可以一目了然地告訴您是否有錯誤的問題並在惡化之前採取迴避措施。

9.6 實驗室

本章將監視添加到Image Gallery應用程序中，該實驗室要求您對待辦事項列表應用程序進行相同的操作。您無需潛入源代碼，我已經構建了包含普羅米修斯指標的應用程序圖像的新版本。從Diamol /ch09-todo list運行一個容器，瀏覽到該應用程序，然後添加一些項目，您會在 /指標URL中看到可用的指標。對於實驗室，您希望將該應用程序與圖像庫相同的位置。

- 編寫一個Docker組成的文件，您可以用來運行該應用程序，該文件還啟動了Prometheus容器和Grafana容器。Prometheus容器應已配置為從待辦事項列表應用程序中刮擦指標。Grafana容器應配置使用儀表板，以顯示應用程序中的三個密鑰指標：創建的任務數，處理的HTTP請求總數以及當前正在處理的HTTP請求的數量。
-

這聽起來像是很多工作，但實際上不是 - 本章中的練習涵蓋了所有細節。這是一個很好的實驗室，因為它可以為您提供新應用程序的指標的經驗。

與往常一樣，您會在GitHub上找到我的解決方案，以及我的最終儀表板的圖形：<https://github.com/ixeyed/diamol/blob/master/ch09/lab/readme.md>

10 使用 Docker 組成 運行多個環境

我們在第7章中查看了Docker撰寫，您對如何使用YAML來描述多容器應用程序有了很好的了解，並使用撰寫命令行進行管理。從那以後，我們加強了我們的Docker應用程序，以通過健康檢查和監視為他們準備生產。現在是時候返回撰寫了，因為我們不需要每個環境中的所有這些生產功能。便攜性是Docker的主要好處之一。當您將應用程序打包以在容器中運行時，它以相同的方式運行，無論它在何處都可以使用，這很重要，因為它消除了在環境之間的漂移。

漂移是當使用手動流程來部署軟件時總是發生的。一些更新被遺漏或一些新的依賴性被遺忘，因此生產環境與用戶測試環境不同，這與系統測試環境再次不同。當部署失敗時，通常是由於漂移，並且需要大量的時間和精力才能追蹤缺失的碎片並將其正確。搬到Docker可以解決該問題，因為每個應用程序都已經包裝了其依賴項，但是您仍然需要靈活性來支持不同環境的不同行為。Docker Compose提供了我們將在本章中涵蓋的更高級功能。

10.1用Docker撰寫許多應用程序

Docker Compose是用於在單個Docker引擎上運行多容器應用程序的工具。這對開發人員很有幫助，並且也用於非生產環境。組織經常在不同的環境中運行多個應用程序的應用程序 - 同時版本1.5在生產中運行，版本1.5.1正在Hotfix環境中進行測試，版本1.6正在完成用戶測試，並且版本1.7在系統測試中。這些非生產環境不需要生產的規模和性能，因此對於Docker構成運行這些環境並從硬件中獲得最大利用的有用用例。

為此，環境之間需要存在一些差異。您不能有幾個容器試圖偵聽端口80上的流量，也不能試圖在服務器上寫入數據。您可以設計Docker撰寫文件以支持該文件，但是首先，您需要了解構圖如何標識哪些Docker資源是同一應用程序的一部分。它可以通過命名約定和標籤來做到這一點，如果您想運行同一應用程序的幾個副本，則需要圍繞默認設置工作。

現在嘗試

打開終端並瀏覽本章的練習。運行我們已經使用過的兩個應用程序，然後嘗試運行待辦事項列表應用程序的另一個實例：

```
CD ./ch10/cercises

# 從第8章運行隨機數應用程序：Docker Compose -f ./numbers/doc
# ker-compose.yml -D

# 從第6章運行待辦事項列表應用程序：docker compose -f ./todo-list/
# docker-compose.yml up -d

# 並嘗試另一個待辦事項列表的副本：docker compose -f ./todo-list/d
# ocker-compose.yml up -d
```

您的輸出將與圖10.1中的我的輸出相同。您可以從不同文件夾中的構成文件啟動多個應用程序，但是您無法通過從同一文件夾上運行來啟動第二個應用程序。Docker組成的認為，您要它運行已經運行的應用程序，因此它不會啟動任何新容器。

```

Runs the application specified in the
Compose file in the numbers directory
Docker Compose creates containers for
the apps, using the directory name as the
prefix for each container name

PS>cd ./ch10/exercises
PS>
PS>docker compose -f ./numbers/docker-compose.yml up -d
[+] Running 3/3
✓ Network numbers app-net      Created
✓ Container numbers-numbers-api-1 Started
✓ Container numbers-numbers-web-1 Started
PS>
PS>docker compose -f ./todo-list/docker-compose.yml up -d
[+] Running 2/2
✓ Network todo-list_app-net    Created
✓ Container todo-list-todo-web-1 Started
PS>
PS>docker compose -f ./todo-list/docker-compose.yml up -d
[+] Running 1/0
✓ Container todo-list-todo-web-1 Running
Runs the application specified in the todo-list
directory - Compose creates the application container
Repeating the command doesn't create a second copy of the app; Compose
sees there's already a container running which matches the app definition

```

圖10.1重複Docker Compose命令啟動應用程序不會運行第二份應用程序。

Docker構成使用*project*的概念來識別各種資源是同一應用程序的一部分，並且使用包含撰寫文件作為默認項目名稱的目錄名稱。在創建資源時構成前綴項目名稱，對於容器，它還添加了數字計數器作為後綴。因此，如果您的撰寫文件位於一個名為App1的文件夾中，並且它定義了一個名為Web的服務和一個名為Disk的捲，Compose將通過創建一個稱為App1-Disk的捲和一個稱為App1-Web-1的容器來部署它。容器名稱末尾的計數器支持刻度，因此，如果將其擴展到Web服務的兩個實例，則新容器將稱為App1-Web-2。 圖10.2顯示瞭如何為待辦事項列表應用程序構建的容器名稱。

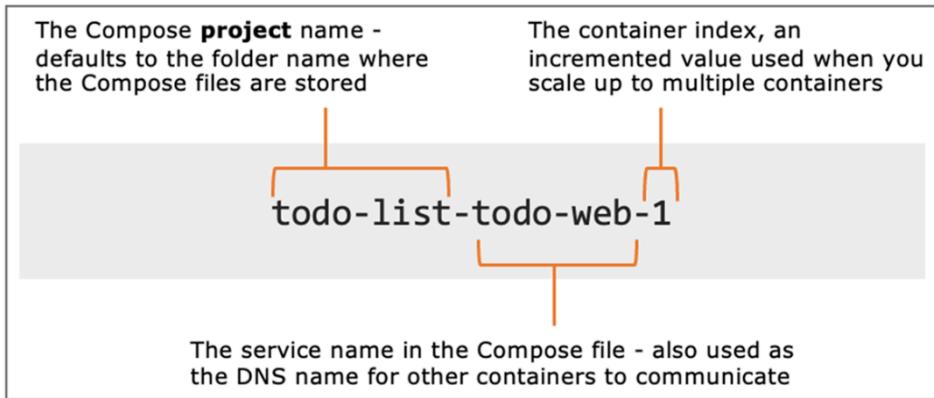


圖10.2 Docker構建了通過包括項目名稱來管理其管理的資源的名稱。

您可以覆蓋默認的項目名稱構成用途，這就是您可以在單個Docker引擎上的不同容器集中運行同一應用程序的許多副本。

現在嘗試

您已經有一個運行待辦事項應用程序的實例； 您可以通過指定其他項目名稱來開始另一個。該網站使用隨機端口，因此，如果您想實際嘗試這些應用程序，則需要查找分配的端口：

```
Docker撰寫-f ./todo-list/docker-compose.yml -p todo -test up -d
```

Docker容器LS

```
Docker Container Porto Todo-Test-Modo-Web-1 8080
```

我的輸出在圖10.3中。指定項目名稱意味著就撰寫而言，這是一個不同的應用程序，並且沒有資源與此項目名稱匹配，因此組合創建一個新容器。命名模式是可以預測的，因此我知道新容器將稱為托多測-WEBS-1。 Docker CLI具有容器端口命令可以找到已發布的容器的端口，我可以將其與生成的容器名稱一起使用以查找應用程序端口。

```

Specifying a project name means Compose sees this as a different
app from the existing one which used the folder name for the project

PS>docker compose -f ./todo-list/docker-compose.yml -p todo-test up -d
[+] Running 2/2
  ✓ Network todo-test_app-net      Cr...
  ✓ Container todo-test-todo-web-1 Started          0.0s
  ✓ Container todo-test-todo-web-1 Started          0.1s
PS>
PS>docker container ls
CONTAINER ID   IMAGE               COMMAND                  CREATED             STATUS              PORTS
0bd82312b726   diamol/ch06-todo-list:2e   "dotnet ToDoList.dll"   6 seconds ago
                Up 6 seconds           80/tcp, 0.0.0.0:56848->8080/tcp   todo-test-todo-web-1
a59b661e056d   diamol/ch06-todo-list:2e   "dotnet ToDoList.dll"   15 seconds ago
                Up 15 seconds          80/tcp, 0.0.0.0:56847->8080/tcp  todo-list-todo-web-1
9c477d51a256   diamol/ch08-numbers-api:2e-v3  "dotnet Numbers.Api..."  7 minutes ago
                Up 7 minutes (healthy)  8080/tcp                   numbers-numbers-api-1
PS>
PS>docker container port todo-test-todo-web-1 8080
0.0.0.0:56848

Now there are two copies of the application running. They use the exact same
Docker Compose file, but the project name makes them separate apps in Compose

Docker Compose creates containers with a consistent naming system, so you
can run commands like this to find the port the new container is publishing

```

圖10.3指定一個項目名稱，您可以使用一個組合文件運行同一應用的多個副本。

這種方法使您可以運行許多不同應用程序的副本。我也可以使用相同的撰寫文件，但指定其他項目名稱，也可以部署我的隨機數應用程序的另一個實例。這很有用，但是在大多數情況下，您需要更多的控制權 - 可以找出每個版本要使用的隨機端口並不是操作或測試團隊的絕佳工作流程。為了支持不同環境中的不同設置，您可以創建重複的撰寫文件並編輯需要更改的屬性，但是Compose具有更好的方法來通過覆蓋。

10.2 使用Docker撰寫覆蓋文件

團隊遇到了試圖使用Docker組成的不同應用程序配置的問題，並且通常最終得到許多撰寫文件，這是每個環境的一個。這是有效的，但是它是無法維護的，因為這些文件通常是90%重複的內容，這意味著它們將擺脫同步，並且您會回到漂移情況。覆蓋文件是一種更整潔的方法。Docker組成，可讓您將多個文件合併在一起，而後來的文件中的屬性則覆蓋了合併以前的文件。

圖10.4顯示瞭如何使用替代來構建一組易於維護的組合文件。您從包含應用程序的基本結構的核心Docker-compose.yml文件開始，其服務定義和配置為所有環境中常見的屬性。然後，每個環境都有自己的覆蓋文件，添加了特定的設置，但不會復制核心文件中的任何配置。

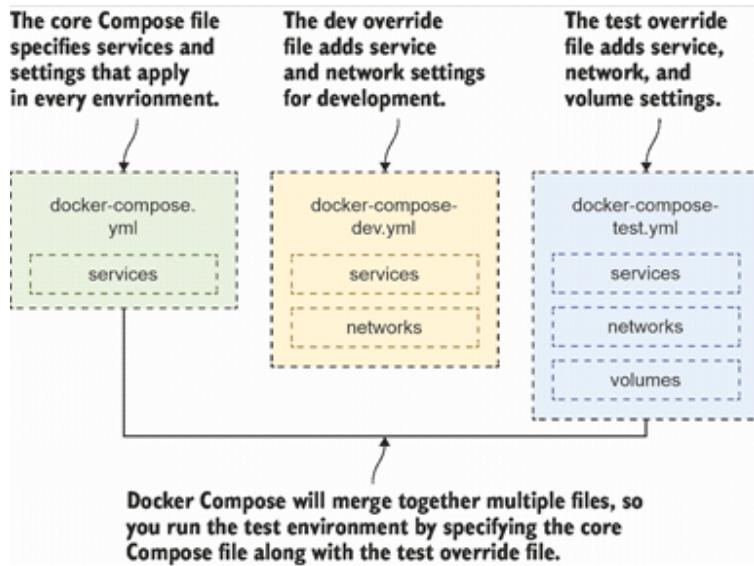


圖10.4用添加特定環境設置的覆蓋文件刪除重複

這種方法是可以維護的。如果您需要進行適用於所有環境的更改（例如，將圖像標籤更改以使用最新版本），您將在核心文件中進行一次，並且它會過濾到每個環境中。如果您只需要更改一個環境，則只需更改該文件即可。您在每個環境中擁有的覆蓋文件還可以作為環境之間差異的清晰文檔。

列表10.1顯示了一個非常簡單的示例，其中核心構成文件指定大多數應用程序屬性，並且覆蓋更改圖像標籤，以便此部署將使用待辦事項應用程序的v2。

列出10.1碼頭構成更新單個屬性的覆蓋文件

```
# from docker-compose.yml - the core app specification: services: todo-web: image: diamol/ch06-todo-list:2e ports: - 8080 environment: - DatabaseProvider=Sqlite networks: - app-net # and from docker-compose-v2.yml - the version override file: services:
```

全球：

圖像：Diamol/CH06-TODO列表：2E-V2

在Override文件中，您只需指定您關心的屬性，但是您需要保留主曲線文件的結構，以便Docker組合可以將定義鏈接在一起。此示例中的覆蓋文件僅更改圖像屬性的值，但是需要在服務塊下的托多德 - web塊中指定該值，因此組合可以與核心文件中的完整服務定義匹配。

當您Docker組合命令中指定多個文件路徑時，Docker將文件合併在一起。配置命令在此處非常有用 - 它驗證了輸入文件的內容，如果輸入有效，它將寫出最終輸出。您可以使用它來查看應用覆蓋文件時會發生什麼。

現在嘗試

在本章的練習文件夾中，使用Docker組合將列表10.1中的文件合併在一起並打印輸出：

```
docker compose -f ./todo-list/docker-compose.yml -f ./todo-list/docker-compose-v2.yml config
```

Config命令實際上並未部署應用程序； 它只是驗證配置。您會在輸出中看到兩個文件已合併。所有屬性均來自Core Docker組成的文件，除了從第二個文件覆蓋值的圖像標籤，您可以在圖10.5中看到。

The config command is useful for dry-runs - it merges the input files, validates the final output and displays it, if it is valid

```
PS>docker compose `>> -f ./todo-list/docker-compose.yml `>> -f ./todo-list/docker-compose-v2.yml `>> configname: todo-listservices: todo-web: environment: Database:Provider: Sqlite image: diamol/ch06-todo-list:2e-v2networks: app-net: nullports: - mode: ingress target: 8080 protocol: tcpenvironment:Database:Provider: Sqliteimage: diamol/ch06-todo-list:2e-v2networks: app-net: name: todo-list_app-net
```

The image tag is the only value which comes from the override file, the rest of the settings come from the base Compose file

圖10.5將撰寫文件與覆蓋文件合併並顯示輸出

Docker組成的應用程序按命令中列出的文件列出的順序，並在右側的右覆蓋文件中列出了文件。這很重要，因為，如果您弄錯了訂單，您將獲得意外的結果 - 配置命令在這裡是關鍵，因為它顯示了完整撰寫文件的干燥運行。輸出按字母順序排列所有內容，因此您會看到網絡和服務，這首先令人不安，但很有用。您可以將該命令自動化為部署過程的一部分，並提交合併的文件來源控制 - 然後字母順序使比較發行版易於比較。

使用覆蓋圖像標籤只是一個快速示例。數字文件夾中的隨機數應用程序有一組更現實的組合文件：

- docker-compose.yml - 核心應用程序定義。它指定沒有任何端口或網絡定義的Web和API服務。 docker-compose-dev.yml-用於開發該應用程序。它指定了Docker網絡，並為服務添加端口以發布和禁用健康和依賴性檢查。這就是開發人員可以迅速啟動並運行。 docker-compose-test.yml-用於在測試環境中運行。這指定了網絡，添加了健康檢查參數，並發布了Web應用程序的端口，但是通過不發布任何端口，它可以使API服務保持內部。 docker-compose-uat.yml-用於用戶接受測試環境。這指定了一個網絡，發布了網站的標準端口80，設置了始終重新啟動的服務，並指定了更嚴格的健康檢查參數。
-

列表10.2顯示了開發式文件的內容，很明顯，它不是完整的應用程序規範，因為沒有指定圖像。如果核心文件中有匹配的鍵，此處的值將合併到核心組成文件中，添加新屬性或覆蓋現有屬性。

Listing 10.2 An override file only specifies changes from the main Compose file

```
services:
  numbers-api:
    ports:
      - "8087:80"
    healthcheck:
      disable: true

  numbers-web:
    entrypoint:
      - dotnet
      - Numbers.Web.dll
    ports:
      - "8088:80"

networks:
  app-net:
    name: numbers-dev
```

其他覆蓋文件遵循相同的模式。每個環境都為Web應用程序和API使用不同的端口，因此您可以在一台計算機上運行它們。

現在嘗試

首先刪除所有現有容器，然後在多個環境中運行隨機數應用程序。每個環境都需要一個項目名稱和正確的組成文件集：

```
# 刪除任何現有的容器Docker Container RM -F $ (Docker Cont
ainer LS -AQ)

# 在Dev配置中運行該應用：Docker Compose -F ./numbers/docker-compose.yml -f ./numbers/docker-compose
-dev.yml -p numbers -dev up -d

# 和測試設置：docker撰寫-f ./numbers/docker-compose.yml -f ./numbers/docker-compose-
test.yml -P數字測試-D

# 和uat：
docker compose -f ./numbers/docker-compose.yml -f ./numbers/docker-compose-uat.yml -p numbers -uat -up -d
```

現在，您擁有運行應用程序的三個副本，它們都彼此隔離，因為每個部署都使用其自己的Docker網絡。在組織中，這些將在一台服務器上運行，團隊將通過瀏覽到正確的端口來使用他們想要的環境。例如，您可以將端口80用於UAT，端口8080進行系統測試，而端口8088用於開發團隊的集成環境。圖10.6顯示了我使用網絡和容器創建的輸出。

```

Runs the app using a project name. Compose creates containers with the
project name prefix, but the network doesn't have a prefix because it is
explicitly named in the Compose file

PS>docker compose -f ./numbers/docker-compose.yml -f ./numbers/docker-compose-dev.yml
-p numbers-dev up -d
[+] Running 3/3
✓ Network numbers-dev           Created      0.0s
✓ Container numbers-dev-numbers-web-1 Started   0.2s
✓ Container numbers-dev-numbers-api-1 Started   0.2s

PS>
PS>docker compose -f ./numbers/docker-compose.yml -f ./numbers/docker-compose-test.yml
-p numbers-test up -d
[+] Running 3/3
✓ Network numbers-test          Created      0.0s
✓ Container numbers-test-numbers-api-1 Started   0.2s
✓ Container numbers-test-numbers-web-1 Started   0.2s

PS>
PS>docker compose -f ./numbers/docker-compose.yml -f ./numbers/docker-compose-uat.yml
-p numbers-uat up -d
[+] Running 3/3
✓ Network numbers-uat          Created      0.0s
✓ Container numbers-uat-numbers-api-1 Started   0.2s
✓ Container numbers-uat-numbers-web-1 Started   0.2s

PS>

The test configuration uses a different override file
and specifies a different Docker network - so the
containers are isolated from the dev containers

The uat environment uses a different network again, so all
three versions of the app can be deployed on one server

```

圖10.6在一台計算機上的容器中運行多個孤立的應用程序環境

現在，您有三個部署可以用作單獨的環境：`http://localhost`是UAT，`http://localhost:8080`是系統測試，`http://localhost:8088`是開發環境。瀏覽任何一個，您會看到相同的應用程序
but each web container can see the API container only in its own network。這樣可以使應用程序分開，因此，如果您在開發環境中繼續獲取隨機數，則API會破裂，但是系統測試和UAT環境仍在工作。每個環境中的容器都使用DNS名稱進行通信，但是Docker限制了容器網絡中的流量。 圖10.7顯示了網絡隔離如何使您的所有環境分開。

Four containers are running on the same Docker Engine, but they connect to different Docker networks. Containers on the numbers-test network cannot see containers on the numbers-uat network.

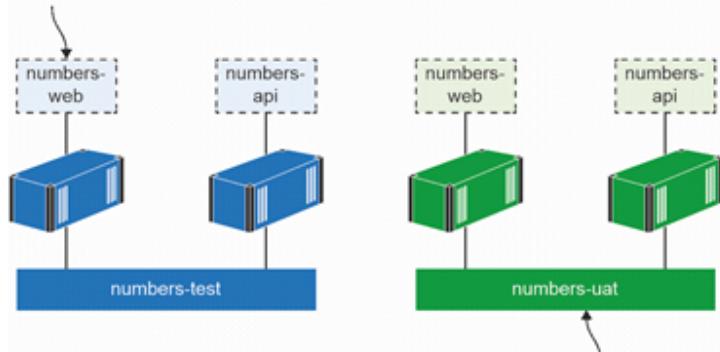


圖10.7 r 使用網絡FO，在一個Docker發動機上解開多個環境

r隔離

是時候提醒您Docker Compose是一種客戶端工具，您需要訪問所有撰寫文件以管理您的應用程序。您還需要記住所使用的項目名稱。如果您想清除測試環境，刪除容器和網絡，通常您只能運行Docker撰寫，但是對於這些環境而言，這對這些環境不起作用，因為組合需要所有在UP命令中使用的所有相同的文件和項目信息以匹配資源。

現在嘗試

讓我們刪除該測試環境。 您可以在down命令上嘗試不同的變體，但是唯一可以使用的變體是具有與原始UP命令相同的文件列表和項目名稱的一個變體：

```
#如果我們在當前目錄中使用了docker-compose.yml文件，這將有效
```

```
#如果我們使用沒有項目名稱的覆蓋文件：Docker compose -f ./numbers/docker-compose.yml -f ./numbers/docker-compose-test.yml
```

```
#但是我們指定了一個項目名稱，因此我們也需要包括：Docker撰寫-f ./numbers/docker-compos e.yml -f ./numbers/docker-compose-test.yml -p數字 -
```

您可以在圖10.8中看到我的輸出。您可能已經猜到，除非提供匹配的文件和項目名稱，否則構成無法識別應用程序的運行資源，因此在第一個命令中，它沒有刪除任何內容。在第二個命令中，Compose確實嘗試刪除容器網絡，即使有連接到該網絡的應用程序容器。

```
There's no Compose file in this directory and no -f flag to point  
to a Compose file, so the command has nothing to work with  
  
PS>docker compose down  
no configuration file provided: not found  
PS>  
PS>docker compose -f ./numbers/docker-compose.yml -f ./numbers/docker-compose-test.yml  
down  
[+] Running 1/0  
! Network numbers-test No resource found to remove  
PS>  
PS>docker compose -f ./numbers/docker-compose.yml -f ./numbers/docker-compose-test.yml  
-p numbers-test down  
[+] Running 3/2  
✓ Container numbers-test-numbers-api-1 Removed  
✓ Container numbers-test-numbers-web-1 Removed  
✓ Network numbers-test Removed  
PS>  
  
Without specifying the project name Compose  
doesn't find the resources it created  
  
This is the matching combination of files and project name for the  
original up command, so Compose finds and removes the resources
```

圖10.8您需要使用相同的文件和項目名稱來管理使用Compose的應用程序。

這些錯誤之所以發生，是因為網絡是在撰寫的覆蓋文件中明確命名的。我沒有在第二個down命令中指定項目名稱，因此它使用了默認值，即文件夾名稱號。Compose查找稱為數字的容器-Numbess-Web_1和數字-N umbers-api_1，但沒有找到它們，因為它們實際上是使用Project前綴數字測試創建的。組成的認為，這些容器已經消失了，只需要清理網絡，它確實找到了，因為在沒有項目前綴的情況下使用了組合文件中的顯式網絡名稱。撰寫試圖刪除該網絡，但幸運的是，Docker不會讓您刪除仍然附有容器的網絡。

這是向您展示您需要小心Docker組成的漫長方法。它是非生產環境的絕佳工具，它通過在單個計算機上部署數十或數百個應用程序從計算資源中獲得最大價值。覆蓋文件可重用應用程序定義並確定環境之間的差異，但是您需要了解管理開銷。您應該查看腳本和自動化的部署和拆卸。

10.3 向環境變量和秘密注入配置

您可以使用Docker網絡隔離應用程序，並捕獲具有組合替代的環境之間的差異，但是您還需要更改環境之間的應用程序配置。大多數應用程序都可以從環境變量或文件中讀取配置設置，並且組合對這兩種方法都有很好的支持。

我將介紹本節中的所有選項，因此，我們將比到目前為止更深入地撰寫。這將幫助您了解應用配置設置所具有的選擇，並且您可以選擇適合自己的選擇。

可以回到這些練習的待辦事項應用程序。該應用程序的Docker映像旨在讀取用於配置設置的環境變量和文件。環境之間需要有三個需要變化的項目：

- *Logging* - 如何詳細說明記錄級別。這將開始在開發環境中非常詳細，並且在測試和生產中變得越來越少。*Database provider* - 是在應用程序容器內使用簡單的數據文件，還是單獨的數據庫（可能在容器中運行也可能不會運行）。
- *Database connection string* - 如果應用程序不使用本地數據文件，則連接到數據庫的詳細信息。
-

我正在使用覆蓋文件為不同環境注入配置，並且我正在為每個項目使用不同的方法，因此我可以向您展示Docker Compose的選項。清單10.3顯示核心撰寫文件；這只是具有將配置文件設置為秘密的Web應用程序的基本信息。

列出10.3撰寫文件指定Web服務有一個秘密

```
services:  
  All-Web:  
    image: Diamol/CH06-Todo-List:2e  
    secrets:  
      - source: todo-db-connection  
        target: /app/config/secrets.json
```

秘密是注入配置的有用方法，在Docker組成和Kubernetes中都使用了類似概念。在撰寫文件中，您指定秘密的源和目標。源是從容器運行時加載秘密的地方，目標是秘密在容器內浮出水面的文件路徑。

該秘密指定為來自源todo-db連接的來源，這意味著需要在撰寫文件中定義該名稱的秘密。秘密的內容將在目標路徑/app/config/secrets.json上加載到容器中，這是應用程序搜索配置設置的位置之一。

前面的撰寫文件本身是無效的，因為沒有秘密部分，並且需要使用todo-db-connection秘密，因為它在服務定義中使用。列表10.4顯示了用於開發的覆蓋文件，該文件為服務設置了更多配置並指定了秘密。

Listing 10.4 The development override adds config settings and the secret setup

```
services:  
  todo-web:  
    ports:  
      - 8089:8080  
    environment:  
      - Database.Provider=Sqlite  
    env_file:  
      - ./config/logging.debug.env  
  
secrets:  
  todo-db-connection:  
    file: ./config/empty.json
```

該替代文件中有三個屬性，可以注入應用程序配置並更改容器中應用程序的行為。您可以使用它們的任何組合，但是每種方法都有好處：

- 環境在容器中添加了一個環境變量。該應用程序讀取此值並配置本地SQLITE數據庫，該數據庫是一個簡單的數據文件。這是設置配置值的最簡單方法，並且從撰寫文件中可以清楚地看出要配置的內容。`env_file`包含文本文件的路徑，文本文件的內容將作為環境變量加載到容器中。文本文件中的每一行被讀取為環境變量，其名稱和值則以平等符號隔開。該文件的內容設置了記錄配置。使用環境變量文件是一種在多個組件之間共享設置的簡單方法，因為每個組件都引用文件而不是複制環境變量列表。
- Secrets是組成YAML文件中的頂級資源，例如服務和網絡。它包含todo-db-connection的實際來源，該托入本地文件系統上的文件。在這種情況下，沒有單獨的數據庫可以連接到該應用程序，因此它使用一個空的JSON文件作為秘密。該應用程序將讀取文件，但是沒有配置設置可應用。
-

現在嘗試

您可以使用Compose File和todo-List-configured目錄中的覆蓋物在開發配置中運行該應用程序。用捲髮將請求發送到Web應用程序中，並檢查容器正在記錄大量細節：

```
#刪除現有容器：Docker容器RM -F $ (Docker Container LS -  
AQ)  
  
#帶有配置替代的應用程序 - 對於Linux容器：Docker Compose -f ./todo-list-configured/docker-co  
mpose.yml -f ./todo-list-配置/docker-compose-compose-dev.yml -p todo-p todo-dev up -d  
d-d  
  
#或用於Windows Server容器，這些容器使用不同的文件路徑作為秘密：Docker compose -f ./todo-list  
-configured/docker-compose.yml -f ./todo-list-配置/docker-compose-compose-compose.yml -f ./todo-list-lis  
t-list-configured-configured-compose-compose-compose-compose.yml-windev-dev-dev-dev-dev-dev--  
dev-dev-dev-dev--podo-podo-podo-dododod-podo tododod-dodo  
  
#將一些流量發送到應用程序：curl -s http  
:// localhost : 8089/list  
  
#檢查日誌：docker容器日誌 - 尾巴4 todo-dev-todo-web-1
```

您可以在圖10.9中看到我的輸出。 Docker組成始終為每個應用程序使用網絡，因此即使在組合文件中沒有指定網絡，它也會創建一個默認網絡並將容器連接到它。就我而言，最新的日誌行顯示了停止的ASP.NET服務器連接。 您的可能會顯示不同的內容，但是如果檢查整個日誌，您會看到這樣的條目，以及該應用程序發送到數據庫的SQL語句。這顯示了增強的日誌記錄配置。

```

Run the app in development configuration, using settings defined
in Compose override files, environment files and secrets:

PS>docker compose \
>> -f ./todo-list-configured/docker-compose.yml \
>> -f ./todo-list-configured/docker-compose-dev.yml \
>> -p todo-dev \
>> up -d
[+] Running 1/1
✓ Container todo-dev-todo-web-1 Started
PS>
PS>curl -s http://localhost:8089/list | Out-Null
PS>
PS>docker container logs --tail 4 todo-dev-todo-web-1
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished HTTP/1.1 GET http://localhost:8089/list - 200
      charset=utf-8 441.5436ms
debug: Microsoft.AspNetCore.Server.Kestrel.Connections[2]
      Connection id "0HN6G72KAN0FR" stopped.
PS>

Using the app causes log entries to be written

The development configuration writes lots of logs, including the web server
connection details - these aren't produced in the default configuration

```

圖10.9 c 通過在碼頭中應用配置設置來懸掛應用程序行為 是構成的

開發部署使用環境變量和秘密進行應用程序配置 - 撰寫文件中指定的值，而配置文件則加載到容器中。

還有一個測試部署，使用Compose支持的另一種方法：使用環境變量*on the host machine*為容器提供值。這使部署更加便攜，因為您可以更改環境而無需更改撰寫文件本身。如果您想在具有不同配置的其他服務器上旋轉第二個測試環境，這將很有用。列表10.5顯示了ToDo-Web服務規範中。

使用環境變量作為組合文件中的值列表10.5

```

todo-web:
  ports:
    - "${TODO_WEB_PORT}:8080"
  environment:
    - Database:Provider=Postgres
  env_file:
    - ./config/logging.information.env
  networks:
    - app-net

```

該端口的美元和賽車設置從環境變量中替換為該名稱。因此，如果我在計算機上有一個變量設置，其中我正在運行使用名稱todo_web_port和值8877的Docker組成，然後構成該值，而端口規範實際上變為“8877：80”。此服務規範在文件docker-compose-test.yml中，其中還包括數據庫的服務以及用於與數據庫容器連接的秘密。

您可以通過與開發環境相同的方式指定撰寫文件和項目名稱來運行測試環境，但是Compose的最終配置功能使事情變得更容易。如果在當前文件夾中查找一個稱為.env的文件，它將將其視為環境文件，並將內容讀為一組環境變量，並在運行命令之前將其填充它們。

現在嘗試

導航到已配置的待辦事項列表應用程序的目錄，並在沒有指定任何參數docker組成的情況下運行它：

```
CD ./todo-list配置
```

```
#或用於Windows Server容器 - 使用不同的文件路徑：CD ./todo-list-configured-windows
```

```
Docker組成-D
```

圖10.10顯示了Compose已創建了Web和數據庫容器，儘管COMPOSES並未指定數據庫服務。儘管我沒有指定名稱，但它還使用了項目名稱todo_ch10。.ENV文件設置了組合配置以默認運行測試環境，而無需指定測試覆蓋文件。

There are multiple Compose files and a directory containing configuration files. The `.env` file specifies the default configuration for Compose, including the files to use and the project name

```
PS>cd ./todo-list-configured
PS>
PS>ls -al
total 32
drwxr-xr-x  7 elton  staff  224 Sep  8 20:10 .
drwxr-xr-x  7 elton  staff  224 Jul 28  2022 ..
-rw-r--r--  1 elton  staff  252 Sep  8 20:09 .env
drwxr-xr-x  7 elton  staff  224 Jul 28  2022 config
-rw-r--r--  1 elton  staff  224 Sep  8 19:49 docker-compose-dev.yml
-rw-r--r--  1 elton  staff  450 Sep  8 19:51 docker-compose-test.yml
-rw-r--r--  1 elton  staff  153 Sep  8 19:50 docker-compose.yml
PS>
PS>docker compose up -d
[+] Running 3/3
  ✓ Network todo-test          Created
  ✓ Container todo-ch10-todo-web-1 Started      0.0s
  ✓ Container todo-ch10-todo-db-1 Started      0.2s
                                                0.2s
PS>
```

Running Compose commands without any argument uses the defaults from `.env`, so it merges the core Compose file with the test environment setup, creating a database container along with the web container

圖10.10使用環境文件指定Docker撰寫文件和項目名稱的默認值

您可以在此處使用一個簡單的命令而無需指定文件名，因為`.env`文件包含一組可用於配置Docker組成的環境變量。首次使用是用於容器配置設置，例如Web應用程序的端口； 第二個是為撰寫命令本身，列出要使用的文件和項目名稱。清單10.6完整顯示`.ENV`文件。

清單10.6配置容器並使用環境文件組合

```
# 容器配置 - 要發布的端口：todo_web_port = 8877 todo_
db_port = 5432

# 組成配置 - 文件和項目名稱：compose_path_separator =; compose_fil
e = docker-compose.yml; docker-compose-test.yml compose_project_name
= todo-ch10
```

環境文件將在測試配置中捕獲該應用程序的默認組合設置 - 您可以輕鬆地修改它，因此開發配置為默認值。將環境文件與您的撰寫文件一起保存有助於記錄哪些文件集表示哪個環境，但請注意，Docker僅查看一個稱為.env的文件。您無法指定文件名，因此您無法輕鬆地在具有多個環境文件的環境之間切換。

在Docker Compose中瀏覽配置選項已經花了一些時間。您將與Docker一起使用很多撰寫文件，因此您需要熟悉所有選項。我將在這裡總結所有這些，但是有些比其他人更有用：

- 使用環境屬性指定環境變量是最簡單的選項，它使您的應用程序配置易於從撰寫文件中讀取。但是，這些設置為純文本，因此您不應將它們用於敏感數據，例如連接字符串或API鍵。 帶有秘密屬性的加載配置文件是最靈活的選項，因為它是一種在許多容器平台中使用的建模配置的方法，可用於敏感數據。當您使用撰寫時，秘密的來源可能是本地文件，也可以是存儲在kubernetes群集中的加密秘密。無論源是什麼源，秘密的內容都會加載到容器中的文件中，以供讀取應用程序。 將設置存儲在文件中，並將其加載到具有環境屬性的容器中時，當您在服務之間擁有一許多共享設置時，將有用。撰寫本地讀取文件，並將單個值設置為環境屬性，因此您可以在連接到遠程Docker引擎時使用本地環境文件。 組合環境文件.ENV對於捕獲想要成為默認部署目標的任何環境的設置很有用。
-
-

10.4 通過擴展字段減少重複

在這一點上，您很可能會認為Docker組合具有足夠的配置選項來滿足任何情況。但這實際上是一個非常簡單的規範，並且在與之合作時會遇到一些局限性。最常見的問題之一是，當您擁有共享許多相同設置的服務時，如何減少組成文件的膨脹。在本節中，我將介紹Docker組成的最終功能，該功能可以解決此問題 - 使用*extension fields*在一個地方定義YAML的塊，您可以在整個撰寫文件中重複使用。擴展字段是組合的強大但未使用的功能。他們消除了很多重複和錯誤的潛力，一旦您習慣了YAML合併語法，它們就可以直接使用。

在本章練習的Image-Gallery文件夾中，有一個使用擴展字段的docker-compose-prod.yml文件。列表10.7顯示了您如何定義擴展字段，在任何頂級塊（服務，網絡等）之外宣布它們，並給他們一個帶有ampersand符號的名稱。

列表10.7定義Docker組成文件頂部的擴展字段

```
X標籤：&記錄記錄：選項：M  
ax-size：'100m'最大文件：'10'x-  
labels：&Labels app-name：ima-  
ge-gallery
```

擴展字段是自定義定義；在此文件中，有兩個稱為記錄和標籤。按照慣例，您將塊名為“X”前綴，因此X-Labels Block定義了一個稱為標籤的擴展名。日誌擴展名為容器日誌指定設置，並且可以在服務定義中使用。標籤擴展名為標籤指定一個鍵/值對，該標籤可以在服務定義中的現有標籤字段中使用。

您應該注意這些定義之間的區別 - 伐木字段包括日誌記錄屬性，這意味著它可以直接在服務中使用。標籤字段不包括標籤屬性，因此需要在現有的一組標籤中使用。清單10.8清楚地表明了使用兩個擴展名的服務定義。

Listing 10.8 Using extensions inside a service definition with YAML merge

```
services:  
  
  iotd:  
    ports:  
      - 8080:80  
    <<: *logging  
    labels:  
      <<: *labels  
    public: api
```

擴展字段與YAML合併語法<<：接著是字段名稱，該字段名稱帶有星號。因此，<<：*記錄將在YAML文件中此點的日誌擴展字段的值中合併。當撰寫此文件時，它將從日誌記錄擴展程序中將記錄部分添加到服務中，並將為現有標籤部分添加一個額外的標籤，並從標籤擴展字段中合併值。

現在嘗試

我們不需要運行此應用即可查看如何撰寫文件。剛剛運行config命令會這樣做。這驗證了所有輸入並打印出最終組合文件，擴展字段合併到服務定義中：

```
# 在CH10/練習下瀏覽圖像 - 加利爾文件夾：CD ..\ Image-Gallery
```

```
# 檢查生產替代配置：Docker Compose -f ./docker-compose.yml -f ./docker-compose-prod.yml配  
置
```

My output is in figure 10.11—I haven't shown the full output, just enough of the service definitions to show the extension fields being merged in.

The config command merges and processes the input files and validates the result. If it's valid the merged file is displayed

This warning tells you that there is a system environment variable used in the Compose file which doesn't have a value set. This is something to watch out for if you use variable substitution

```
PS>cd ../image-gallery
PS>
PS>docker compose -f ./docker-compose.yml -f ./docker-compose-prod.yml config
WARN[0000] The "HOST_IP" variable is not set. Defaulting to a blank string.
name: image-gallery
services:
  accesslog:
    image: diamol/ch09-access-log:2e
    labels:
      app-name: image-gallery
    logging:
      options:
        max-file: "10"
        max-size: 100m
    networks:
      app-net: null
  grafana:
    depends_on:
      - prometheus
        condition: service_started
        required: true
    image: diamol/ch09-grafana:2e
    labels:
      app-name: image-gallery
    logging:
      options:
        max-file: "10"
        max-size: 100m
```

Both services merge the logging and labels extension fields, so the contents are repeated - but they're not duplicated in the source Compose files

圖10.11使用config命令與擴展字段一起處理文件，並檢查結果

擴展字段是確保撰寫文件中最佳實踐的有用方法 - 使用相同日誌記錄設置和容器標籤是設置所有服務標準的一個很好的示例。這不是每個應用程序都可以使用的東西，但是在您要複制並粘貼大塊YAML的時候，可以在工具箱中使用它。現在您有了更好的方法。但是，有一個很大的限制：擴展字段不在多個撰寫文件中應用，因此您無法在核心組合文件中定義擴展名，然後將其在覆蓋中使用。這是對YAML而不是構成的限制，但這是值得注意的。

10.5 了解使用Docker的配置工作流程

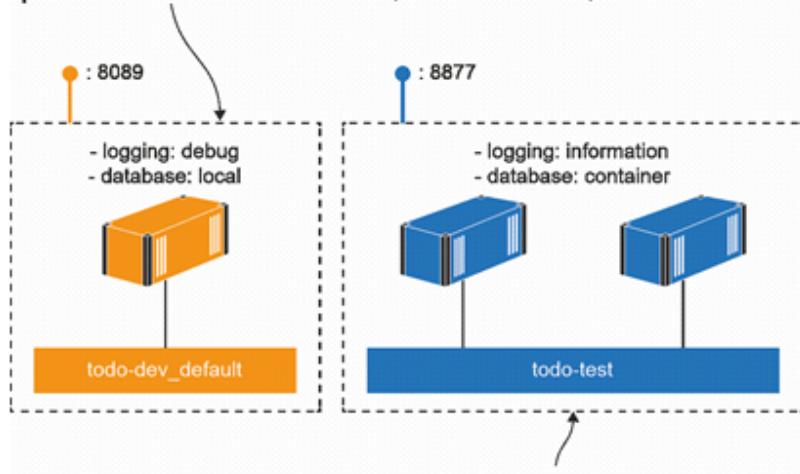
將整個部署配置用於在源控制中的一組工件中捕獲的系統的整個部署配置非常有價值。它允許您僅通過在該版本中獲取源並運行部署腳本來部署應用程序的任何版本。它還使開發人員可以通過本地運行生產堆棧並在自己的環境中重現錯誤來快速進行修復。

環境之間總是有意的變化，而Docker構成可以使您可以對環境之間的差異進行建模，同時仍可以為您提供生活在源控制中的一套部署偽像。在本章中，我們研究了Docker組成並專注於三個關鍵領域的不同環境：

- *Application composition* - 不每個環境都會運行整個堆棧。開發人員可能不使用監視儀表板之類的功能，或者應用程序可以在測試環境中使用容器化數據庫，而是將其插入生產中的雲數據庫中。覆蓋文件可讓您整齊地執行此操作，共享通用服務並在每個環境中添加特定服務。*Container configuration* - 專業需要更改以匹配環境的要求和功能。已發布的端口需要是唯一的，因此它們不會與其他容器相撞，並且音量路徑可能在測試環境中使用本地驅動器，而是在生產中共享存儲。覆蓋效果，以及每個應用程序的隔離Docker網絡，使您可以在單個服務器上運行多個環境。
- *Application configuration* - 容器內部應用程序的行為將在環境之間發生變化。這可能會更改記錄應用程序的數量，或者它用於存儲本地數據的緩存的大小，或者可以打開或關閉整個功能。您可以使用覆蓋文件，環境文件和秘密的任何組合來完成此操作。
-

圖10.12顯示，使用第10.3節中運行的待辦事項列表應用程序。開發和測試環境完全不同：在DEV中，該應用程序配置為使用本地數據庫文件，並且在測試中還運行一個數據庫容器，並將應用程序配置為使用該應用程序。但是每個環境都使用孤立的網絡和獨特的端口，因此可以在同一台計算機上運行，如果開發人員需要旋轉本地測試環境並查看其與其開發版本的比較，這是完美的。

The dev environment uses a local database file and enhanced logging, specified with environment variables, environment files, and secrets.



The test environment also runs a database container, specified in the Compose override file. It uses different ports and networks from dev, so both environments can be run on the same machine.

圖10.1 2使用Docker C為同一應用定義非常不同的環境

組成

最重要的收穫是配置工作流程在每個環境中都使用相同的Docker映像。構建過程將產生您的容器圖像的標記版本，該版本已通過所有自動測試。這是您使用撰寫文件中的配置部署到煙霧測試環境中的發布候選者。當它通過煙霧測試時，它將繼續進入下一個環境，該環境使用相同的圖像集並應用了撰寫的新配置。最終，如果測試全部通過，當您使用Docker Compose或Kubernetes部署清單將這些相同的容器圖像部署到生產中時，您將發布該版本。發布的軟件與通過所有測試的軟件完全相同，但是現在它具有從容器平台提供的生產行為。

10.6 實驗室

在這個實驗室中，我希望您為待辦事項應用程序構建自己的環境定義集。您將匯總開發環境和測試環境，並確保它們都可以在同一台機器上運行。

開發環境應該是默認環境，您可以使用Docker組成。設置應該

- 使用本地數據庫文件發佈到to-Do應用
- 用程序的端口8089運行V2
-

測試環境將需要使用特定的Docker組成文件和項目名稱運行。它的設置應該

- 使用單獨的數據庫容器使用卷用於數據庫存儲發佈到端口8080使用最新的to-Do應用程序圖像
-

這裡與本章的Todo-List結合練習中的撰寫文件有相似之處。 主要區別是卷 - 數據庫容器使用稱為PGDATA的環境變量，以設置應編寫數據文件的位置。您可以將其與撰寫文件中的捲規範一起使用。

正如您在本章中看到的那樣，有很多方法可以解決此問題。我的解決方案在此處的github上：<https://github.com/sixeyed/diamol/blob/2e/ch10/lab/readme.md>。

11 使用Docker和Docker組成的構建和測試應用程序

自動化是Docker的核心。您描述將組件包裝在Dockerfile中的步驟，並使用Docker命令行執行它們；您可以在Docker組合文件中對應用程序的體系結構進行建模，並使用命令行啟動和停止應用程序。命令行工具非常整潔地與自動化流程相吻合，例如按日程安排運行或開發人員推動代碼更改時的作業。您使用哪種工具來運行這些作業都沒關係；它們都讓您運行腳本命令，因此您可以輕鬆地將Docker Workflow與自動化服務器集成在一起。

現在，您將學習如何與Docker進行連續集成（CI）。CI是一個自動化過程，可定期運行以構建應用程序並執行一套測試。當CI工作健康時，這意味著該應用程序的最新代碼很好，已包裝並準備部署為候選人。建立和管理CI服務器和工作曾經是耗時且密集的。“構建經理”在大型項目中是人類的全職角色。Docker簡化了CI過程的每個部分，並為人們提供更多有趣的工作。

11.1 CI過程如何與Docker一起使用

CI進程是一條以代碼開頭，執行一組步驟並以經過測試的可部署工作結束的管道。CI面臨的挑戰之一是，每個項目的管道變得獨特 - 不同的技術堆棧在步驟中做不同的事情並產生不同類型的工作。CI服務器需要為所有這些獨特的管道工作，因此編程語言和構建框架的每種組合都可以安裝在服務器上，並且很容易變得難以管理。

Docker為CI過程帶來了一致性，因為每個項目都遵循相同的步驟並產生相同類型的工作。 圖11.1顯示了帶有Docker的典型管道，它是由代碼更改或定時時間表觸發的，並且會產生一組Docker映像。 這些圖像包含最新版本的代碼 - 對，測試，包裝並將其推入註冊表進行分發。

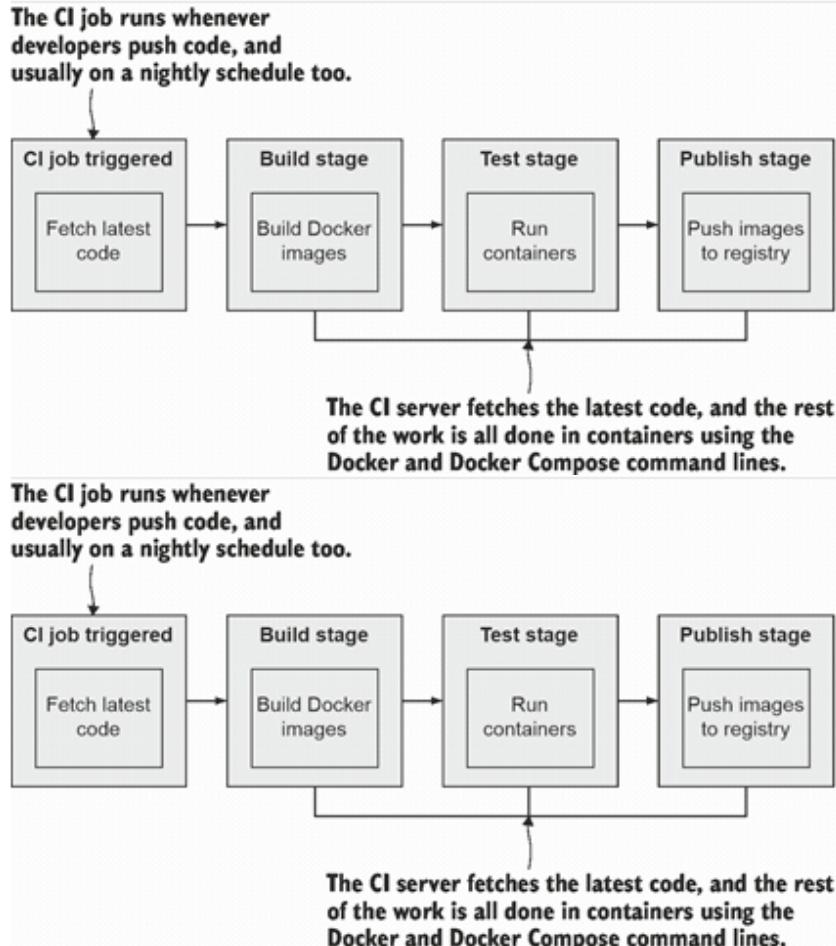


圖11.1 B

CI管道的ASIC步驟構建，測試和發布應用程序 - 所有執行

ITH Docker。

CI管道中的每個步驟均與Docker或Docker組成，所有工作都發生在容器內。您使用容器來編譯應用程序，因此CI服務器無需安裝任何編程語言或構建SDK。自動化單元測試作為圖像構建的一部分運行，因此，如果損壞了代碼，則構建失敗，CI作業將停止。您還可以通過使用Docker撰寫整個應用程序與一個單獨的容器一起運行測試以模擬用戶工作流程，從而運行更複雜的端到端測試。

在對接的CI過程中，所有艱苦的工作都發生在容器中，但是您仍然需要一些基礎架構組件來將所有內容放在一起：集中式源代碼系統，用於存儲圖像的Docker註冊表以及一個自動化服務器來運行CI作業。您可以從所有支持Docker中選擇的託管服務都有很大的選擇 - 您可以將Github與Azure Devops和Docker Hub混合使用，也可以使用GitLab，該GitLab提供了多合一解決方案。或者，您可以在Docker容器中運行自己的CI基礎架構。

11.2用Docker旋轉基礎設施

當您可以免費獲得可靠的託管服務時，沒有人願意運行自己的基礎架構組件，但是在Docker中運行構建系統是一個非常有用的替代方法。如果您想將源代碼和包裝的圖像完全保存在您自己的網絡中，這是理想的選擇 - 用於數據主權或傳輸速度，但是即使您為所有內容都使用外部服務，對於Github或Docker Hub具有公路或Internet連接的罕見場合，可以為罕見的情況提供一個簡單的備份選項。

您需要的三個組件可以輕鬆地使用企業級開源軟件在容器中運行。 使用一個命令，您可以使用名為GO GS的GIT服務器來運行自己的設置，用於源控制，開源Docker註冊表以及Jenkins作為自動化服務器。

現在嘗試

在本章的練習文件夾中，有一個Docker組成的文件來定義構建基礎架構。 對於Linux和Windows容器，設置的一部分是不同的，因此您需要選擇正確的文件。 您還需要在第5.3節中不執行DNS名稱註冊表中的主機文件中添加一個條目。

```
# add registry domain to local hosts file on Mac or Linux:  
4  
  
# OR using PowerShell (run as admin)  
../scripts/add-to-hosts.ps1 registry.local 127.0.0.1  
  
# switch to infrastructure folder:  
cd ch11/exercises/infrastructure  
  
# start the app with Linux containers:  
docker compose -f docker-compose.yml -f docker-compose-linux.yml up -d  
  
# OR start with Windows Server containers:  
docker compose -f docker-compose.yml -f docker-compose-windows.yml up -d  
  
# check containers:  
docker container ls
```

You can see my output in figure 11.2. The commands are different on Linux and Windows, but the outcome is the same—you’ll have the Gogs Git server published to port 3000, Jenkins published to port 8080, and the registry published to port 5010.

```

Adds a DNS name for the registry container to
the local machine, so I can push and pull images
PS>sudo ./scripts/add-to-hosts.sh registry.local 127.0.0.1
127.0.0.1 registry.local
PS>
PS>cd ch11/exercises/infrastructure
PS>
PS>docker compose -f docker-compose.yml -f docker-compose-linux.yml up -d
[+] Running 4/4
✓ Network build-infrastructure          Cre...
✓ Container infrastructure-registry.local-1 Started      0.0s
✓ Container infrastructure-gogs-1        Started      0.2s
✓ Container infrastructure-jenkins-1     Started      0.2s
PS>
PS>docker container ls
CONTAINER ID   IMAGE           COMMAND                  CREATED             STATUS              NAMES
0f70771ca1af   diamol/jenkins:2e    "/sbin/tini -- /usr/.."  6 seconds ago   Up 5 seconds   infrastructure-jenkins-1
fb22b8331fbb   diamol/gogs:2e      "/app/gogs/docker/st..."  6 seconds ago   Up 5 seconds   infrastructure-gogs-1
(health: starting)
13bcf1c78d25   diamol/registry:2e   "/app/registry serve..."  6 seconds ago   Up 5 seconds   infrastructure-registry.local-1
PS>

Starts the build infrastructure - Git, the Docker
registry and the Jenkins automation server. I'm using
Docker Desktop on Mac, so I have Linux containers

```

PORTS

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	NAMES
0f70771ca1af	diamol/jenkins:2e	"/sbin/tini -- /usr/.."	6 seconds ago	Up 5 seconds	infrastructure-jenkins-1
fb22b8331fbb	diamol/gogs:2e	"/app/gogs/docker/st..."	6 seconds ago	Up 5 seconds	infrastructure-gogs-1
(health: starting)		22/tcp, 0.0.0.0:3000->3000/tcp			
13bcf1c78d25	diamol/registry:2e	"/app/registry serve..."	6 seconds ago	Up 5 seconds	infrastructure-registry.local-1
		0.0.0.0:5010->5000/tcp			

All the containers have ports published, so
I can access them on the host machine

圖11.2用一個命令在容器中運行整個構建基礎架構

這三個工具很有趣，因為它們支持不同級別的自動化。註冊表服務器在沒有任何額外設置的情況下運行，因此現在您可以使用註冊表來推動和拉圖像。Local：5010作為圖像標籤中的域。Jenkins使用插件系統來添加功能，您可以手動設置該功能，也可以將Dockerfile中的一組腳本捆綁在一起以為您自動化設置。GO GS使您可以自動化系統設置，但是您無法創建用戶，因此您需要手動執行此操作。

現在嘗試

瀏覽到http://localhost:3000，您將看到Web UI的GOGS。第一頁是歡迎屏幕，如圖11.3所示。
在這裡，您只需單擊註冊即可創建新用戶 - 第一個用戶自動成為系統管理員：

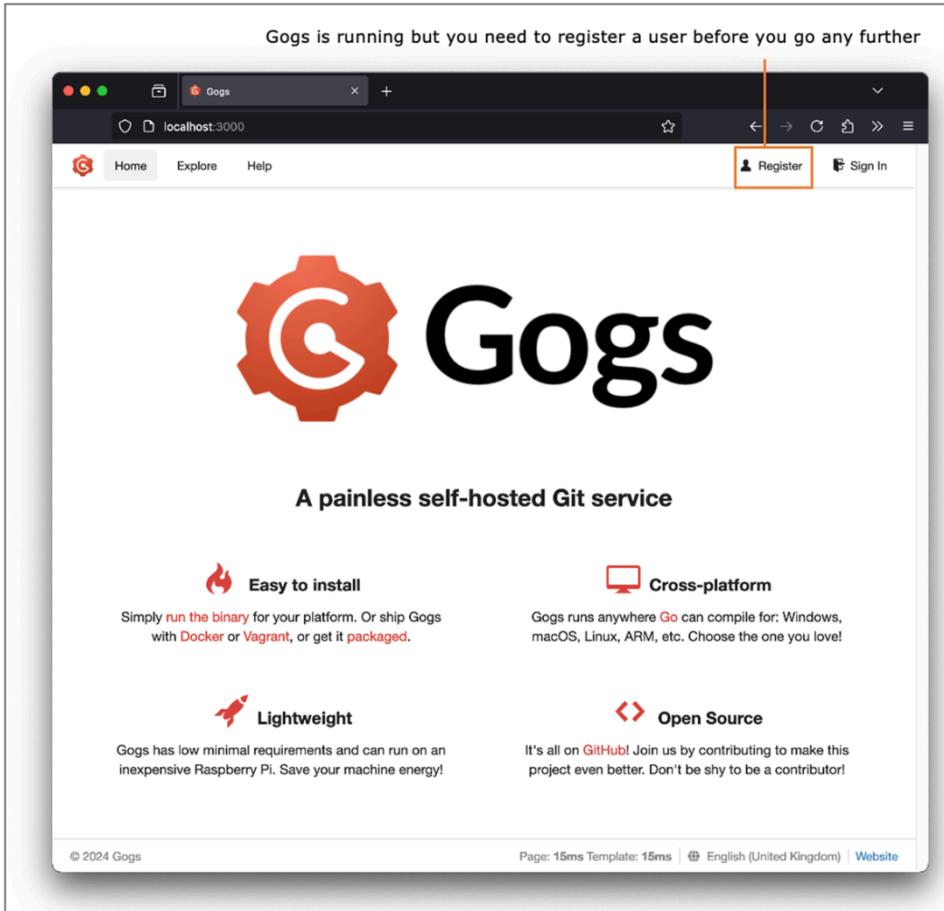


圖11.3在容器中運行GOG。這是一款需要一些手動設置的開源Git服務器。

用用户名Diamol創建用户，如圖11.4所示，您可以使用任何電子郵件地址或密碼，但是Jenkins CI作業期望GOGS用户稱為Diamol。

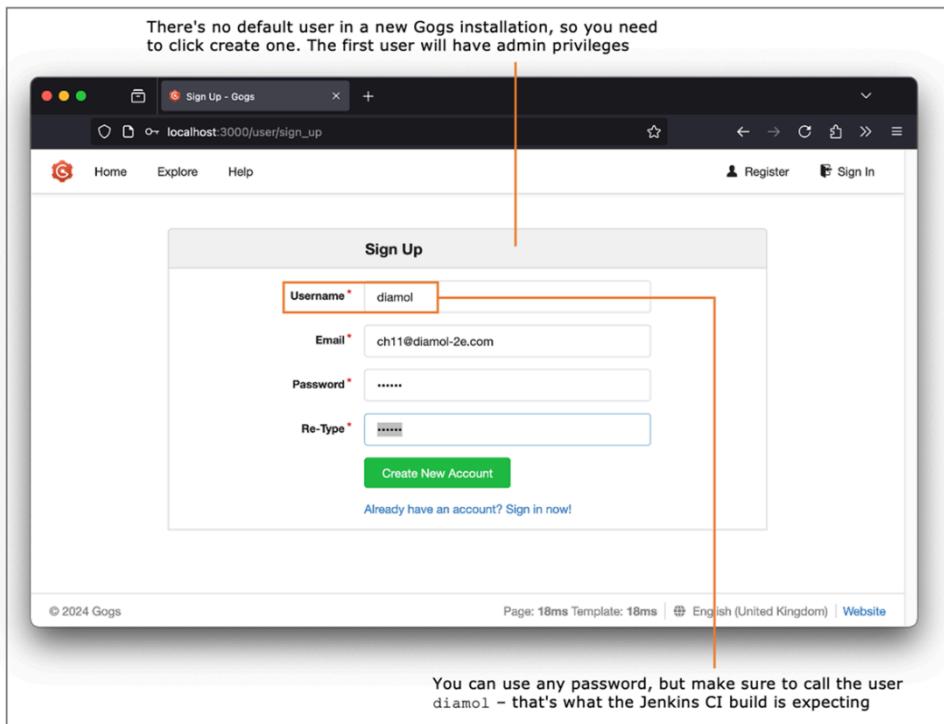


圖11.4在GOGS中創建一個新用戶，您可以將其用於將源代碼推向服務器

單擊創建新帳戶，然後使用Diamol用戶名和您的密碼登錄。最後一步是創建一個存儲庫，即我們將推出將觸發CI作業的代碼。瀏覽到`http://localhost:3000/repo/`創建並創建一個名為Diamol的存儲庫 - 其他詳細信息可以留空，如圖11.5所示。

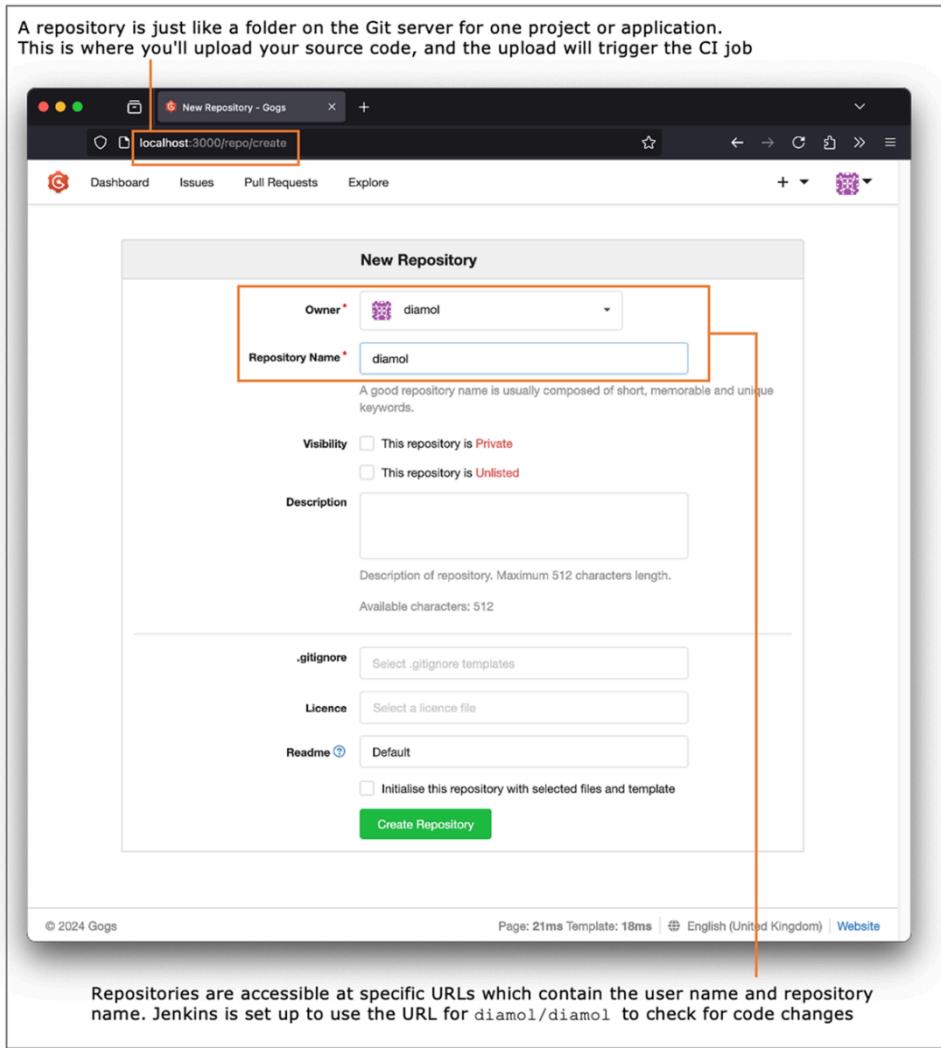


圖11.5 在GOGS中創建GIT存儲庫，您可以在其中上傳源代碼

為您的應用

當您Docker運行該軟件時必須手動配置軟件是非常令人沮喪的，因此必須將屏幕截圖複製到書中更令人沮喪，但是並非每個應用程序都可以使您完全自動化安裝。我本可以使用已經完成的設置步驟從我的容器中導出一個自定義圖像，但是對於您來說，重要的是要看到您不能始終很好地包裝到Docker容器運行工作流程中。

詹金斯是一個更好的體驗。Jenkins是一個Java應用程序，您可以將其作為Docker Image包裝，其中包含一組在容器啟動時運行的腳本。這些腳本幾乎可以做任何事情 - 安裝插件，註冊用戶並創建管道作業。這個Jenkins容器可以完成所有操作，因此您可以直接登錄並開始使用它。

現在嘗試

瀏覽到http://localhost:8080。您會在圖11.6中看到屏幕 - 已經有一個名為Diamol的作業，該屏幕處於失敗狀態。單擊右上角的日誌鏈接，並使用用戶名Diamol和密碼Diamol登錄。

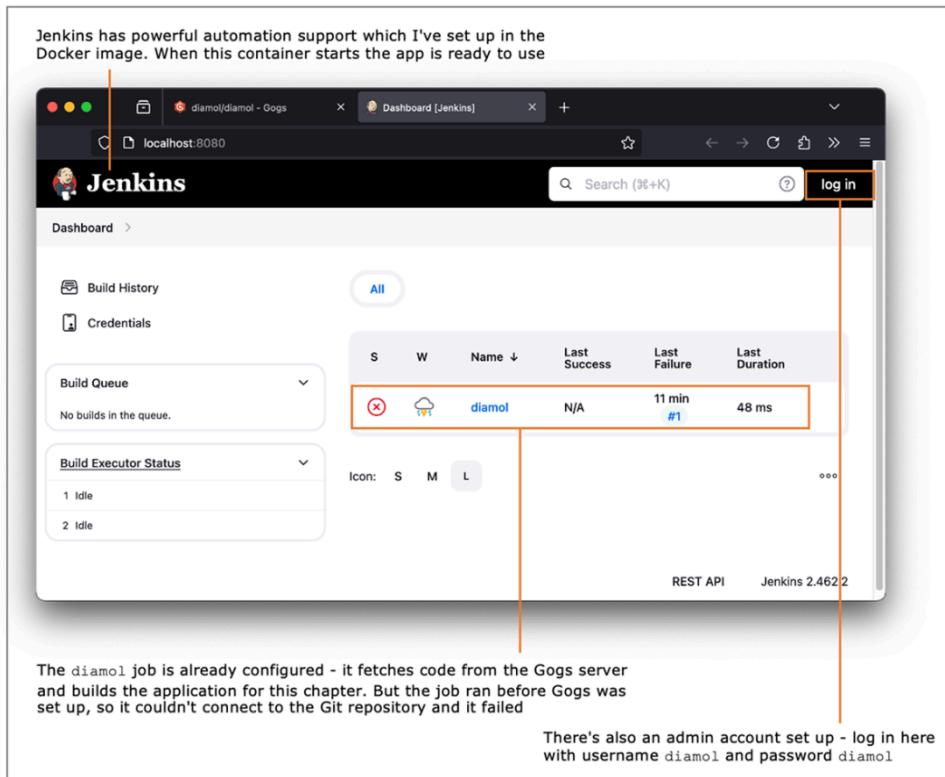


圖11.6在容器中運行Jenkins - 它已設置為已設置的用戶和CI作業。

Jenkins的作業失敗了，因為它配置為從GOOGS GIT服務器中獲取代碼，並且那裡還沒有代碼。本書的源代碼已經是您最初從GitHub克隆的GIT存儲庫。您可以將本地GOOGS容器添加為倉庫的另一台GIT服務器，並將書的代碼推向您自己的基礎架構。

現在嘗試

您可以使用Git Remote添加添加額外的Git服務器，然後推到遙控器。這將代碼從您的本地計算機上傳到GOGS服務器，這也恰好是計算機上的容器：

```
git遠程添加本地http://localhost:3000/diamol/diamol.git
```

```
Git推動本地2E
```

```
# GOGS會要求您登錄 - # 使用您在GOGS中註冊的Diamol用戶名和密碼
```

現在，您在本地GIT服務器中擁有整本書的源代碼。Jenkins作業配置為每分鐘尋找對代碼的更改，如果有更改，它將觸發CI管道。由於代碼存儲庫不存在，因此第一個工作運行失敗了，因此詹金斯（Jenkins）擋置了時間表。您現在需要手動運行工作，以啟動時間表再次工作。

現在嘗試

瀏覽到http://localhost:8080/job/diamol。您將在圖11.7中看到屏幕，然後可以在左側菜單中單擊“構建”以運行作業。如果您現在看不到構建選項，請確保已使用Diamol憑據登錄Jenkins。

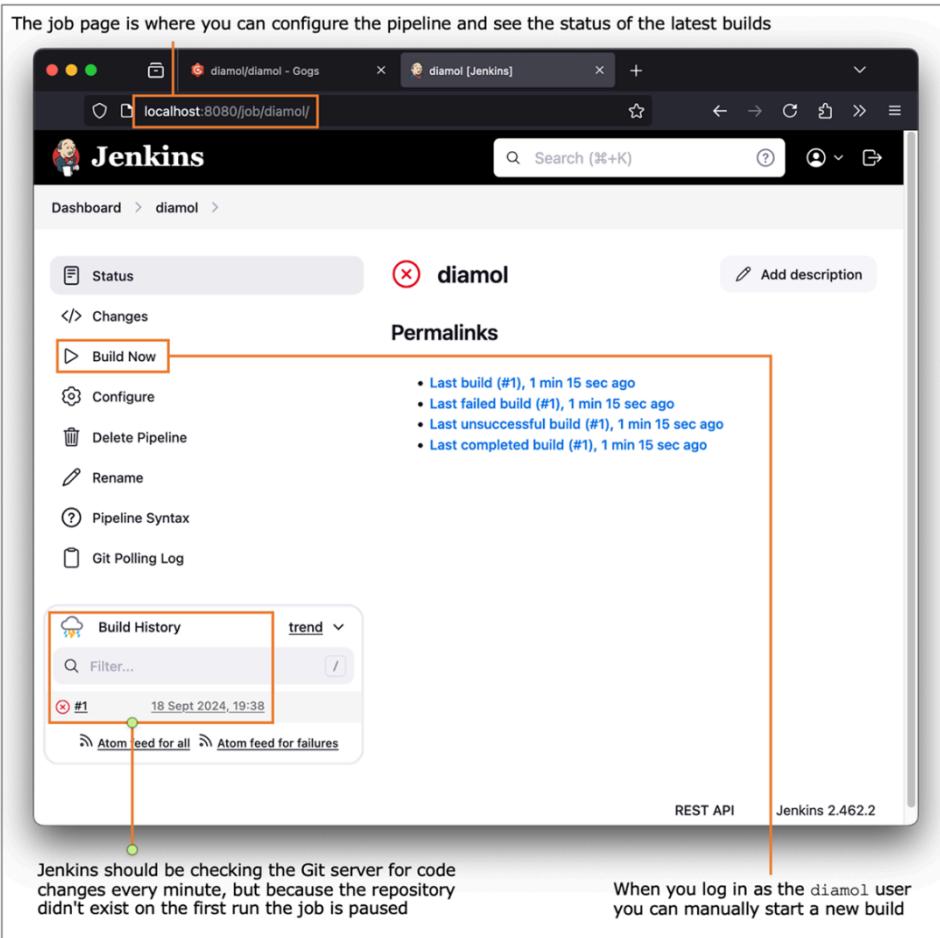


圖11.7 Jenkins作業頁面顯示了作業的當前狀態，並讓您手動啟動構建。

大約一分鐘後，構建將成功完成，網頁將刷新，您將在圖11.8中看到輸出。

The screenshot shows the Jenkins pipeline stage view for the 'diamol' pipeline. On the left, there's a sidebar with options like Status, Changes, Build Now, Configure, Delete Pipeline, Full Stage View, Rename, Pipeline Syntax, and Git Polling Log. Below that is the Build History section, which lists two builds: #2 (18 Sept 2024, 19:40) and #1 (18 Sept 2024, 19:38). The build history shows that build #2 was triggered manually and ran successfully. The main area is titled 'Stage View' and displays a table of stage times for build #2. The table has columns for Declarative: Checkout SCM, Verify, Build, Test, and Push. The total average stage time is 2min 9s. The 'Push' column is highlighted with an orange border. Below the table, there's a summary of the pipeline stages: Average stage times: (Average full run time: ~2min 15s). The pipeline stages are listed as follows:

Declarative: Checkout SCM	Verify	Build	Test	Push
317ms	632ms	2s	1s	2min 9s
#2 Sept 18 20:40 No Changes				
#1 Sept 18 20:38 No Changes				

Permalinks

- Last build (#2), 5 min 49 sec ago
- Last stable build (#2), 5 min 49 sec ago
- Last successful build (#2), 5 min 49 sec ago
- Last failed build (#1), 8 min 5 sec ago
- Last unsuccessful build (#1), 8 min 5 sec ago
- Last completed build (#2), 5 min 49 sec ago

The build history shows the most recent build status - the second build is the one I triggered manually, and it ran successfully

The pipeline stage view shows the steps which ran, along with the duration and status of each step. This is similar to the pipeline described in figure 11.1

圖11.8 該管道的每個部分都使用Docker容器運行，利用一個整潔的技巧：在Docker中運行的容器可以連接到Docker API，並在運行的同一Docker Engine上啟動新容器。Jenkins圖像已安裝了Docker CLI，並且組合文件中的配置設置了Jenkins，因此當它運行Docker命令時，它們會被發送到計算機上的Docker Engine。聽起來很奇怪，但實際上只是利用了Docker CLI呼叫Docker API的事實，因此來自不同地方的CLI可以連接到同一Docker引擎。圖11.9顯示了它的工作原理。

該管道的每個部分都使用Docker容器運行，利用一個整潔的技巧：在Docker中運行的容器可以連接到Docker API，並在運行的同一Docker Engine上啟動新容器。Jenkins圖像已安裝了Docker CLI，並且組合文件中的配置設置了Jenkins，因此當它運行Docker命令時，它們會被發送到計算機上的Docker Engine。聽起來很奇怪，但實際上只是利用了Docker CLI呼叫Docker API的事實，因此來自不同地方的CLI可以連接到同一Docker引擎。圖11.9顯示了它的工作原理。

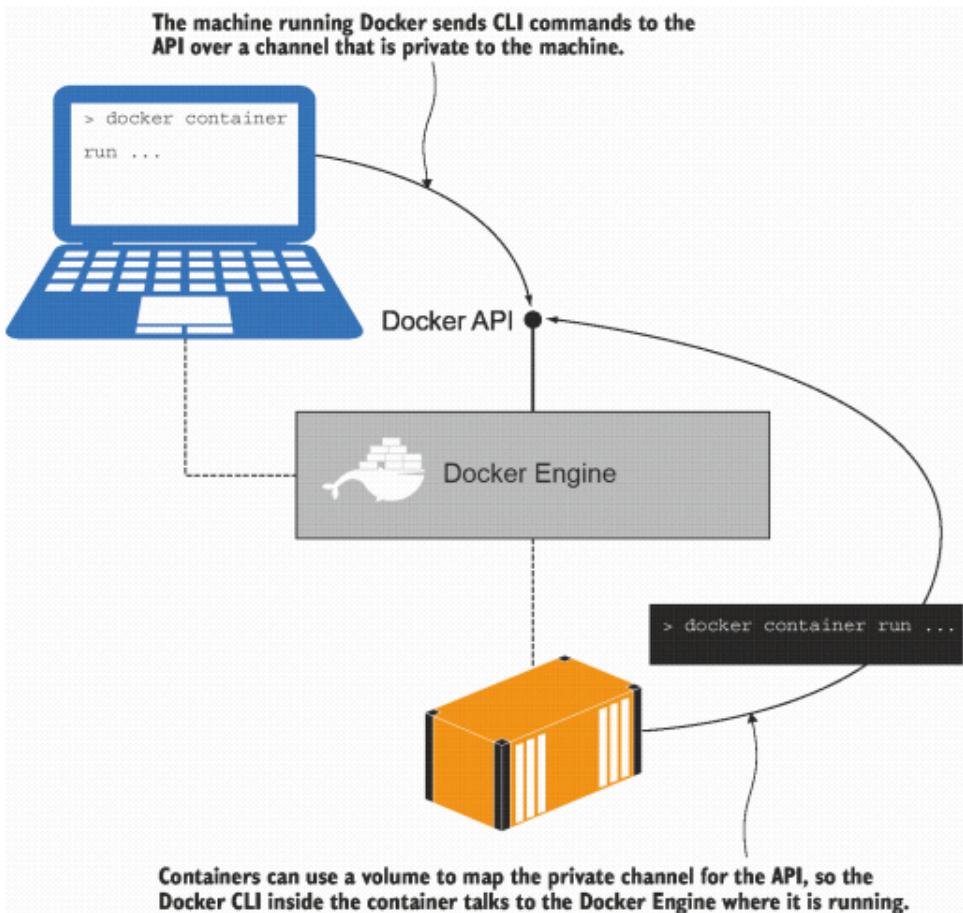


圖11.9運行容器帶有一個音量以綁定Docker API的私人通道

Docker CLI默認情況下，使用機器私有的通信通道（Linux上的插座或Windows上的命名管道）連接到本地Docker API。該通信通道可以用作容器的綁定安裝座，因此，當容器中的CLI運行時，它實際上是連接到機器上的插座或命名管道的。這解鎖了一些有用的方案，其中的應用程序內部的應用程序可以查詢Docker以查找其他容器，或者啟動和停止新容器。這裡還有一個安全問題，因為容器中的應用程序可以完全訪問主機上的所有Docker功能，因此您需要與您信任的Docker Images仔細使用此應用程序 - 當然，您可以相信我的Diamol Images。

清單11.1顯示了Docker的一部分組合您運行的文件以啟動基礎架構容器，重點是Jenkins規範。您可以看到卷與Linux版本中的Docker插座和Windows版本中的命名管結合在一起，這是Docker API的地址。

Listing 11.1 Binding the Docker CLI in Jenkins to the Docker Engine

```
# docker-compose.yml
services:
  jenkins:
    image: diamol/jenkins:2e
    ports:
      - "8080:8080"
    networks:
      - infrastructure

# docker-compose-linux.yml
jenkins:
  volumes:
    - type: bind
      source: /var/run/docker.sock
      target: /var/run/docker.sock

# docker-compose-windows.yml
jenkins:
  volumes:
    - type: npipe
      source: \\.\pipe\docker_engine
      target: \\.\pipe\docker_engine
```

這就是您需要的所有基礎架構。Jenkins連接到Docker Engine來運行Docker和Docker撰寫命令，並且可以通過DNS連接到Git Server和Docker註冊表，因為它們都是同一Docker網絡中的容器。CI進程運行一個命令來構建應用程序，並且構建的所有複雜性均在Dockerfiles和Docker組成文件中捕獲。

11.3 使用Docker構建建築設置

詹金斯（Jenkins）運行的作業已從第8章中構建了隨機數應用程序的新版本。您在第10章中看到瞭如何在多個撰寫文件中分解應用程序定義，並且該應用程序使用該方法來捕獲構建設置的詳細信息。列表11.2來自CH11/cerkises文件夾中的基本docker-compose.yml文件 - 它包含圖像名稱中的環境變量的Web和API服務定義。

列表11.2 使用圖像標籤中的變量構成文件

```
services:  
  numbers-api:  
    image: ${REGISTRY:-docker.io}/diamol/ch11-numbers-api:2e-v3-build-${BUILD_NUMBER:-  
local}  
    networks:  
      - app-net  
  
  numbers-web:  
    image: ${REGISTRY:-docker.io}/diamol/ch11-numbers-web:2e-v3-build-${BUILD_NUMBER:-  
local}  
    environment:  
      - RngApi__Url=http://numbers-api/rng  
    networks:  
      - app-net
```

環境變量語法此處包括一個默認值設置的設置：`- , ${註冊表:-docker.io}`告訴Compose在運行時替換該令牌，用稱為註冊表的環境變量的值。如果該環境變量不存在或為空，則將使用默認值Docker.io，這是Docker Hub的域。我在圖像標籤上使用相同的方法，因此，如果設置了環境變量build_number，則該值將進入標籤。否則將使用本地。

這是一個非常有用的模式，用於使用相同的工件支持CI過程和本地開發人員構建。當開發人員構建API映像時，它們不會設置任何環境變量，因此該圖像將稱為Docker.io/diamol/ch11-numbers-api：2e-v3-build-local。但是Docker.io是Docker Hub，這是默認域，因此該圖像將僅顯示為Diamol/ch11-numbers-api：2e-v3-build-build-local。當詹金斯（Jenkins）在詹金斯（Jenkins）中運行相同的構建時，變量將設置為使用本地碼頭註冊表和作業的實際構建號，詹金斯（Jenkins）將其設置為一個增量數字，因此圖像名稱將為註冊表：5010/diamol/diamol/ch11-numbers-api：2e-v3-build-api：2e-v3-build-api-api。

設置靈活的圖像名稱是CI設置的重要組成部分，但是關鍵信息是在“Override File Docker-Compose-Build.yml”中指定的，它告訴撰寫在哪裡可以找到DockerFiles。

現在嘗試

您可以使用與CI構建管道相同的步驟在本地構建應用程序。從終端會話開始，瀏覽本章目錄，然後用Docker組成：

CD CH11/練習

構建兩個圖像：

```
Docker Compose -f Docker-Compose.yml -F docker-compose-build.yml構建
```

檢查標籤的Web圖像：

```
Docker Image Inspert -F' {.config.labels} 'diamol/ch11-numbers-api : 2e-v3-build-本地
```

您可以在圖11.10中看到我的輸出。

```
This is the same build process that ran in Jenkins – using the same
Docker Engine, so the output is cached from the previous build
=> [numbers-web stage-2 4/4] COPY --from=builder /out/ .
=> [numbers-web] exporting to image
=> => exporting layers
=> => writing image sha256:b4c942985ca318cb55f6e828db7229a6d3e9b9eb375b64f 0.0s
=> => naming to docker.io/diamol/ch11-numbers-web:2e-v3-build-local 0.0s
=> [numbers-web] resolving provenance for metadata file 0.0s
PS>
PS>docker image inspect -f '{{.Config.Labels}}' diamol/ch11-numbers-api:2e-v3-buil
d-local
map[app_version:3.0 build_number:0 build_tag:local com.docker.compose.project:numb
ers-smoketest com.docker.compose.service:numbers-api com.docker.compose.version:2.
29.1]
PS>
This Dockerfile writes labels to the image, which is a useful way of building an audit
trail from a running container all the way back to the CI job that produced the image
```

圖11.10用Docker構建圖像並檢查圖像標籤

通過Docker構建應用程序，有效地為每個已指定的設置的服務運行Docker Image Build命令。那可能是十二張圖像或一個圖像，即使是一個圖像，也可以使用Compose構建的一個好習慣，因為您的撰寫文件指定構建圖像時所需的標籤。此構建中的更多內容是成功的CI管道的一部分 - 您可以在最終的Inspect命令中看到列出圖像標籤的命令。

Docker可讓您將標籤應用於大多數資源 - 統一器，圖像，網絡和卷。它們是簡單的密鑰/值對，您可以在其中存儲有關資源的其他數據。標籤在圖像上非常有用，因為它們被烘烤到圖像中並與圖像一起移動 - 當您推開或拉動圖像時，標籤也會持續下去。當您使用CI管道構建應用程序時，重要的是要有一條審核跟蹤，使您可以從運行的容器中追蹤到創建它的構建作業，並且圖像標籤可以幫助您做到這一點。

列表11.3顯示了隨機數API的Dockerfile的一部分（您將在本章的練習中找到完整的文件，網址為Numbers/Numbers-Api/Dockerfile.v4）。這裡有兩個新的Dockerfile說明：ARG和標籤。

列表11.3指定圖像標籤並在Dockerfile構建參數

```
# 應用圖像
來自Diomol/dotnet-Appnet:2e

arg build_number = 0 arg
build_tag =本地

Label App_version = "3.0" 標籤build_num
ber = ${build_number} label build_tag = ${b
uild_tag}

entrypoint [ "dotnet" , "nummess.api.dll" ]
```

標籤指令僅將鍵/值對從dockerfile應用於圖像時。您可以在Dockerfile中看到App_version = 3.0，並且與圖1.10中的標籤輸出匹配。其他兩個標籤指令使用環境變量來設置標籤值，而這些環境變量由ARG指令提供。

ARG與ENV指令非常相似，只是它在圖像上的構建時間工作，而不是在容器中的運行時間工作。他們都設置了環境變量的值，但是對於ARG指令，設置僅在構建過程中存在，因此您從圖像運行的任何容器都不會看到該變量。這是將數據傳遞到與運行容器無關的構建過程中的好方法。我在這裡使用它來提供圖像標籤中的值 - 在CI過程中，這些值記錄了構建的數量和完整的構建名稱。ARG指令還設置默認值，因此，當您在本地構建圖像而不傳遞任何變量時，您會看到build_number : 0 and build_tag : 圖像標籤中的本地。

您可以看到CI管道中的環境設置如何傳遞到撰寫覆蓋文件中的Docker build命令中。清單11.4顯示了所有構建設置的Docker-Compose-Build.yml文件的內容。

Listing 11.4 Specifying build settings and reusable arguments in Docker Compose

```
x-args: &args
args:
  BUILD_NUMBER: ${BUILD_NUMBER:-0}
  BUILD_TAG: ${BUILD_TAG:-local}

services:
  numbers-api:
    build:
      context: numbers
      dockerfile: numbers-api/Dockerfile.v4
      <<: *args

  numbers-web:
    build:
      context: numbers
      dockerfile: numbers-web/Dockerfile.v4
      <<: *args
```

除非您跳過第10章，在這種情況下，您應該返回並閱讀它，除非您跳過第10章。它不會比午餐時間更多。

構成規範中的構建塊有三個部分：

- 上下文 - 這是Docker將用作構建的工作目錄。這通常是當前目錄，您可以在Docker Image Build命令中通過一個週期傳遞，但這是數字目錄 - 路徑相對於組合文件的位置。
- Dockerfile - 相對於上下文的Dockerfile的路徑。
- Args - 任何要通過的參數，這些論點需要匹配Dockerfile中指定為ARG指令的密鑰。此應用程序的DockerFiles都使用相同的build_number和build_tag參數，因此我使用compense擴展字段來定義這些值一次，然後YAML合併將其應用於兩種服務。

您會看到在許多不同位置指定的默認值，這是為了確保對CI進程的支持不會破壞其他工作流程。您應該始終瞄準以相同方式構建的單個Dockerfile，但是運行構建。撰寫文件中的默認參數意味著當您在CI環境外運行它時構建成功，並且dockerfile中的默認值即使您不使用compose，圖像也正確構建。

現在嘗試

您可以使用普通圖像構建命令構建隨機數API映像，並繞過組合文件中的設置。您可以隨心所欲地調用圖像 - 構建成功，並且由於Dockerfile中的默認值而應用標籤：

```
# 更改為數字目錄#（這是通過撰寫的上下文設置完成的）：CD  
CH11/練習/數字
```

```
# 構建圖像，指定dockerfile路徑和一個構建參數：docker image builds-api/dockerfile.v4 -build-arg bu  
ild_tag = ch11 -t nubmess-api-api。
```

異

我的輸出在圖11.11中 - 您可以在標籤中看到值build_tag : ch11是從我的build命令中設置的，但是值build_nu
mber : 0是從dockerfile中的arg默認設置的。

```
The build command only specifies one argument, but the other argument  
has a default value set in the Dockerfile so the build succeeds  
PS>cd ch11/exercises/numbers  
PS>  
PS>docker image build -f numbers-api/Dockerfile.v4 --build-arg BUILD_TAG=ch11  
-t numbers-api -q . | Out-Null  
PS>  
PS>docker image inspect -f '{{.Config.Labels}}' numbers-api  
map[app_version:3.0 build_number:0 build_tag:ch11 com.docker.compose.project:  
build com.docker.compose.service:dotnet-aspnet com.docker.compose.version:2.2  
8.1]  
PS>  
The value for the build_tag label is specified in the build command; it gets supplied  
as an environment variable by Jenkins when the build runs in the CI pipeline
```

圖11.11包括構建參數的默認值支持開發人員構建工作流程。

這裡有很多級別的細節只是為了將標籤納入圖像，但要正確的事情是重要的事情。您應該能夠運行Docker Image檢查並確切查找該圖像的來源，將其跟蹤回到產生它的CI作業，然後將其跟蹤回到觸發構建的代碼的確切版本。這是從任何環境中運行容器回到源代碼的審核跟蹤。

11.4 編寫CI作業，沒有依賴項除了Docker

您一直在使用Docker和Docker組成的本章中愉快地為隨機數應用程序構建圖像，而無需在計算機上安裝任何其他工具。該應用程序有兩個組件，兩者都以.NET為單位，但是您不需要機器上的.NET SDK來構建它們。他們使用第4章中的多階段Dockerfile方法來編譯和包裝應用程序，因此您需要Docker和Compose。

這是集裝箱CI的主要好處，並且得到了所有託管構建服務（例如Docker Hub，Github Actions和Azure DevOps）的支持。這意味著您不再需要安裝了許多工具的構建服務器，並且要與所有開發人員保持最新的工具。這也意味著您的構建腳本變得非常簡單 - 開發人員可以在本地使用完全相同的構建腳本並獲得與CI管道相同的輸出，因此在不同的構建服務之間移動變得很容易。

我們正在使用Jenkins進行CI流程，並且可以使用簡單的文本文件配置Jenkins作業，該文件與應用程序代碼，Docker-Files和Rocots Files一起使用。清單11.5顯示了管道的一部分（來自文件CH11/hermities/jenkinsfile）以及管道步驟執行的批次腳本。

列出11.5描述CI作業的Jenkinsfile的構建步驟

#jenkinsfile中的構建階段 - 它切換目錄，然後運行兩個# shell命令 - 第一個設置腳本文件，以便可以執行#，第二個將其調用腳本：

```
stage ('build') {步驟{dir ('CH11/練習') {sh'chmod +x ./ci/01-build.bat.bat'sh'./ci/01-build.bat.bat'}}} # 構建-Pull
```

好吧，看看這一點 - 這是您本地運行的同一docker構建命令，除了添加了拉標誌，這意味著Docker將在構建過程中拉出所需的任何圖像的最新版本。無論如何，當您進行構建時，這是一個很好的習慣，因為這意味著您將始終從最新的基本圖像中構建圖像，並使用所有最新的安全修復程序。這在CI過程中尤其重要，因為您的Dockerfile使用的圖像可能會發生變化，這可能會破壞您的應用程序，並且您希望盡快找到它。

構建步驟運行一個簡單的腳本文件 - 文件名以.bat結束，因此它在Windows容器中的Jenkins下運行良好，但在Linux容器中也可以正常運行。此步驟運行構建，並且由於它是一個簡單的命令行調用，因此Docker組成的所有輸出（也是Docker的輸出）都可以捕獲並存儲在構建日誌中。

現在嘗試

您可以在Jenkins UI中查看日誌。瀏覽到`http://localhost:8080/job/diamol`以查看工作，在管道視圖中，單擊成功構建作業的構建步驟。然後單擊日誌。您可以擴展構建步驟，並會看到通常的Docker構建輸出；我的位於圖11.12中。

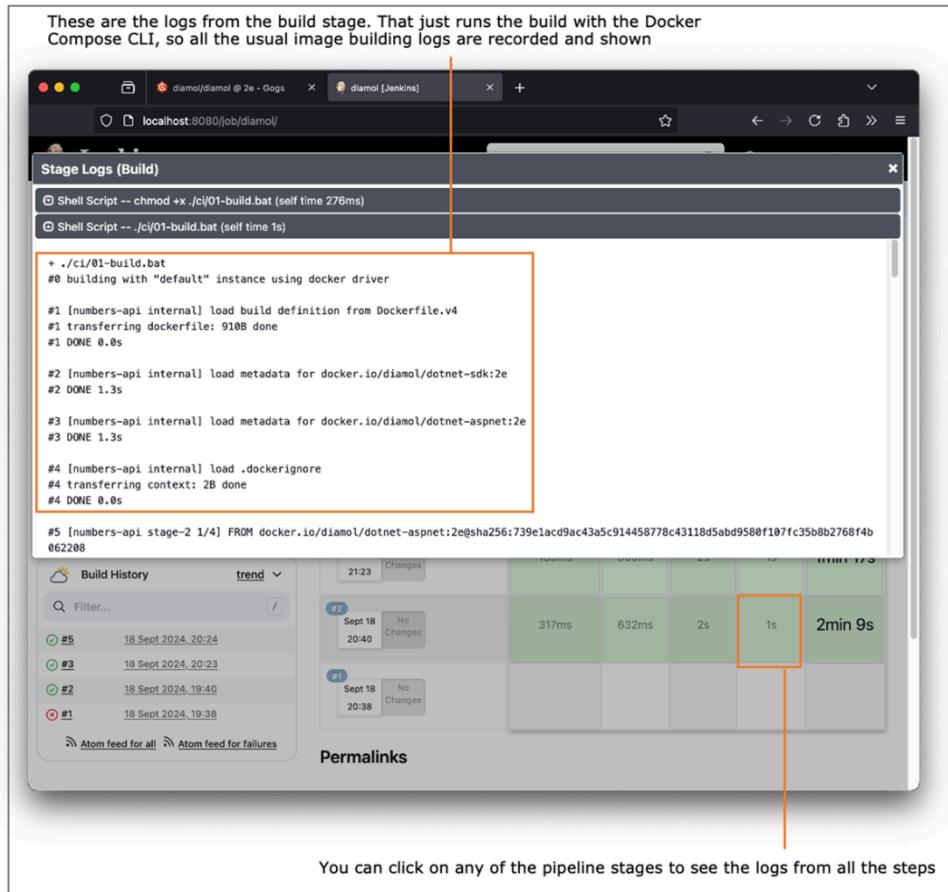


圖11.12在詹金斯（Jenkins）中查看管道構建的輸出顯示了通常的docker日誌。

構建管道中的每個步驟都遵循相同的模式。它只是調用一個批處理腳本，該腳本通過運行Docker組成命令來完成實際工作。這種方法使在不同的構建服務之間很容易切換；您沒有在專有管道語法中編寫邏輯，而是在腳本中編寫它並使用管道來調用腳本。我可以添加管道文件以在gitlab或github操作中運行構建，他們將調用相同的批次腳本。

Jenkins構建的階段全部由容器提供動力：

- **Verify** 調用腳本00-verify.bat，它只會打印出Docker和Docker組成的版本信息。這是啟動管道的有用方法，因為它驗證了Docker依賴項是否可用，並且記錄了構建圖像的工具的版本。**Build** 調用01-build.bat，您已經看到了；它使用Docker組成來構建圖像。註冊表環境變量在JenkinsFile中指定，因此將為本地註冊表標記圖像。
-

- *Test*調用02-test.bat，使用Docker組合來啟動整個應用程序，然後列出容器並將應用程序再次降低。這只是一個簡單的例證，但確實證明容器在沒有失敗的情況下運行。在一個真實的項目中，您將介紹該應用程序，然後在另一個容器中運行端到端測試。*Push*調用03-push.bat，它使用Docker組合來推動所有構建的圖像。圖像標籤具有本地註冊表域，因此，如果構建和測試階段成功，則將圖像推向註冊表。
-

CI管道中的階段是順序的，因此，如果在任何時候發生故障，則工作結束。這意味著註冊表僅存儲潛在釋放候選者的圖像 - 任何已將註冊表推向註冊表的圖像都必須成功通過構建和測試階段。

現在嘗試

您從詹金斯（Jenkins）獲得了一個成功的構建，因為沒有源代碼，因此數字1失敗了，然後構建數字2成功。 您可以使用REST API查詢本地註冊表容器，並且您應該只看到每個隨機數圖像的版本2標籤：

```
# 目錄端點顯示所有圖像存儲庫：curl http://registry.local:5010/v2/_catalog

# 標籤端點顯示一個存儲庫的單個標籤：curl http://registry.local:5010/v2/diamol/ch1
1-numbers-api/tags/list curl http://registry.local.local.local:5010/v2/diamol/diamol/diamol
/diamol/diamol/diambers web/lists/list/lists/list
```

您可以在圖11.13中看到我的輸出 - 我還進行了一些構建，因此Web和API映像存儲庫有多個標籤，但是都沒有構建1標籤，因為第一個構建失敗並且沒有推動任何圖像。

```
The Docker registry API is available from the domain name I added to  
my hosts file. The _catalog endpoint lists all image repositories  
  
PS>curl http://registry.local:5010/v2/_catalog  
{"repositories":["diamol/ch11-numbers-api","diamol/ch11-numbers-web"]}  
PS>  
PS>curl http://registry.local:5010/v2/diamol/ch11-numbers-api/tags/list  
{"name":"diamol/ch11-numbers-api","tags":["2e-v3-build-5","2e-v3-build-3","2e-v3-build-2"]}  
PS>  
PS>curl http://registry.local:5010/v2/diamol/ch11-numbers-web/tags/list  
{"name":"diamol/ch11-numbers-web","tags":["2e-v3-build-5","2e-v3-build-3","2e-v3-build-2","2e-v3-build-4"]}  
PS>  
  
There are multiple tags for the web application image – they were all built and pushed  
by the CI job, you can see the tag name ends with the Jenkins build number
```

圖11.13將Web請求發送到註冊表API以查詢存儲在容器中的圖像

這是一個非常簡單的CI管道，但它向您顯示了構建的所有關鍵階段和一些重要的最佳實踐。關鍵是讓Docker努力工作，並在腳本中構建管道的階段。然後，您可以使用任何CI工具，然後將腳本插入工具的管道定義。

11.5了解CI過程中的容器

在容器中編譯和運行應用程序只是您在CI管道中使用Docker可以做的事情的開始。Docker在所有應用程序構建的基礎上添加了一層一致性，您可以使用該一致性在管道中添加許多有用的功能。圖11.14顯示了一個更廣泛的CI過程，其中包括安全掃描的容器圖像，以確保已知漏洞和數字簽名圖像以主張其出處。

The early part of the pipeline is the same as the simple one we've used in this chapter, except that the registry is a private staging area, not the final repository for distribution.

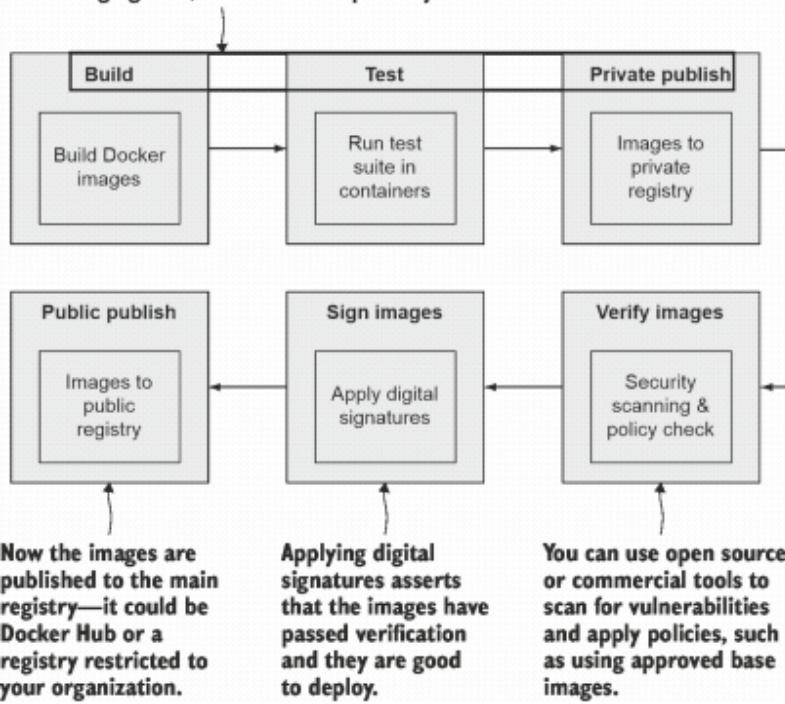


圖11.14生產級CI管道，該管道增加了安全門的階段

Docker稱這種方法為*secure software supply chain*，這對於所有組織尺寸都很重要，因為它使您有信心您將要部署的軟件是安全的。您可以在管道中運行工具以檢查已知的安全漏洞，並在存在問題時構建失敗。您可以將生產環境配置為僅通過數字簽名的圖像運行容器，這是在成功構建結束時發生的過程。當您容器被部署到生產中時，您可以確定它們是從您的構建過程中的圖像中運行的，並且它們包含通過所有測試並且沒有安全問題的軟件。

您在容器和圖像上添加的管道工作中添加的檢查和平衡，因此它們以相同的方式在所有應用程序平台上適用。如果您在項目中使用多種技術，則將在Dockerfiles中使用不同的基本圖像和不同的構建步驟，但是CI管道將全部相同。

11.6 實驗室

實驗室時間！ 您將建立自己的CI管道，但不要害怕。 我們將使用本章中的想法和練習，但是管道階段將要簡單得多。

在本章的實驗室文件夾中，您可以從第6章中找到待辦事項應用程序的源代碼的副本。該應用程序的構建幾乎可以使用 - `jenkinsfile`就在那裡，CI腳本在那裡，並且核心Docker組成的文件在那裡。 您只需要幾件事要做：

- 用構建設置編寫一個名為Docker-Compose-Build.yml的覆蓋文件。 創建一個詹金斯作業來運行管道。 將更改推向Diamol存儲庫中的GOOGS。
-
-

只有三個任務，但如果您的前幾個構建失敗，您需要檢查日誌並調整一些內容，請不要灰心。歷史上沒有人寫過詹金斯（Jenkins）的工作，這是第一次跑步的工作，所以這裡有一些提示：

- 您的組合替代將與練習中的替代相似 - 指定上下文和構建編號標籤的構建參數。
在Jenkins UI中，您單擊新項目以創建作業，您可以從現有的Diamol作業中復制。
- 除了通往JenkinsFile的路徑之外，新的作業設置將相同 - 您需要指定實驗室文件夾而不是練習文件夾。
-

如果您對此並不遙遠，可以在實驗室文件夾中的read-me文件中找到更多信息，並配有jenkins步驟的屏幕截圖，並為Docker組成的文件示例構建配置：<https://github.com/sixeyed/diamol/blob/2e/ch11/lab/readme.md>。
