容器技術基礎

2.1 ▶ 從行程開始說起

第 1 章詳細梳理了「容器」技術的來龍去脈。透過這些內容，希望你能理解下列三個事實：

- 容器技術的興起源於 PaaS 技術的普及；
- Docker 公司發布的 Docker 具有里程碑式的意義；
- Docker 透過「容器鏡像」解決了應用程式打包這個根本性難題。

緊接著，我們詳細介紹了容器技術圈在過去幾年的「風雲變幻」。透過這部分內容，希望你能理解這樣一個道理：

> 容器本身的價值非常有限，真正有價值的是「容器編排」。

也正因如此，容器技術生態才爆發了一場關於「容器編排」的「戰爭」。而這次「戰爭」最終以 Kubernetes 和 CNCF 社群的勝利而告終。所以接下來，我會以 Kubernetes 為核心，來詳細介紹容器技術的各項實踐與其中原理。

```
/ # ps
PID  USER   TIME COMMAND
  1 root   0:00 /bin/sh
  10 root   0:00 ps
```

可以看到，在 Docker 裡最開始執行的 /bin/sh 就是這個容器內部的第 1 號行程（PID=1），而這個容器裡共有兩個行程在執行。這就意味著，前面執行的 /bin/sh 以及剛剛執行的 ps，已經被 Docker 隔離在一個跟宿主機完全不同的世界當中。

這究竟是怎麼做到的呢？

本來，每當我們在宿主機上執行了一個 /bin/sh 程式，作業系統都會分配一個 PID（行程號）給它，例如 PID=100。這個編號是行程的唯一標識，就像員工的工號一樣。所以可以把 PID=100 粗略地理解為這個 /bin/sh 是公司的第 100 號員工，而第 1 號員工就自然是比爾·蓋茲這樣統領全域的人物。

現在，我們要透過 Docker 在一個容器當中執行這個 /bin/sh 程式。這時，Docker 就會在這個第 100 號員工入職時給他施一個「障眼法」，讓他永遠看不到前面的其他 99 位員工，更看不到比爾·蓋茲。這樣，他就會誤以為自己是公司的第 1 號員工。

這種機制其實就是對被隔離應用程式的行程空間動了手腳，使得這些行程只能「看到」重新計算過的 PID，例如 PID=1。可實際上，在宿主機的作業系統裡，它還是原來的第 100 號行程。

這種技術就是 Linux 中的 Namespace 機制。Namespace 的使用方式也非常有意思：它其實只是 Linux 建立新行程的一個選擇性參數。我們知道，在 Linux 系統中建立執行緒的系統呼叫是 clone()，例如：

```
int pid = clone(main_function, stack_size, SIGCHLD, NULL);
```

這個系統呼叫就會為我們建立一個新的行程，並且返回它的 PID。

而當我們用 clone() 系統呼叫建立一個新行程時，就可以在參數中指定 CLONE_NEWPID 參數，例如：

```
int pid = clone(main_function, stack_size, CLONE_NEWPID | SIGCHLD, NULL);
```

這時，新建立的這個行程將會「看到」一個全新的行程空間。在這個行程空間裡，它的 PID 是 1。之所以說「看到」，是因為這只是一個「障眼法」，在宿主機真實的行程空間裡，這個行程的 PID 還是真實的數值，例如 100。

當然，我們還可以多次執行上面的 clone() 呼叫，這樣就會建立多個 PID Namespace，而每個 Namespace 裡的應用程式行程都會認為自己是目前容器裡的第 1 號行程，它們既「看不到」宿主機裡真正的行程空間，也「看不到」其他 PID Namespace 裡的具體情況。

除了我們剛剛用到的 PID Namespace，Linux 作業系統還提供了 Mount、UTS、IPC、Network 和 User 這些 Namespace，用來對各種行程 context 施「障眼法」。例如，Mount Namespace 用於讓被隔離行程只「看到」目前 Namespace 裡的掛載點訊息，Network Namespace 用於讓被隔離行程「看到」目前 Namespace 裡的網路裝置和配置。

這就是 Linux 容器最基本的實現原理。

所以，Docker 容器這個聽起來玄而又玄的概念，實際上是在建立容器行程時，指定了該行程所需要啟用的一組 Namespace 參數。這樣，容器就只能「看到」目前 Namespace 所限定的資源、檔案、裝置、狀態或者配置。而對於宿主機以及其他不相關的程式，它就完全「看不到」了。

可見，容器其實是一種特殊的行程而已。

小結

談到「為行程劃分一個獨立空間」的概念，自然會聯想到虛擬機。圖 2-1 可以看出虛擬機和容器的異同。

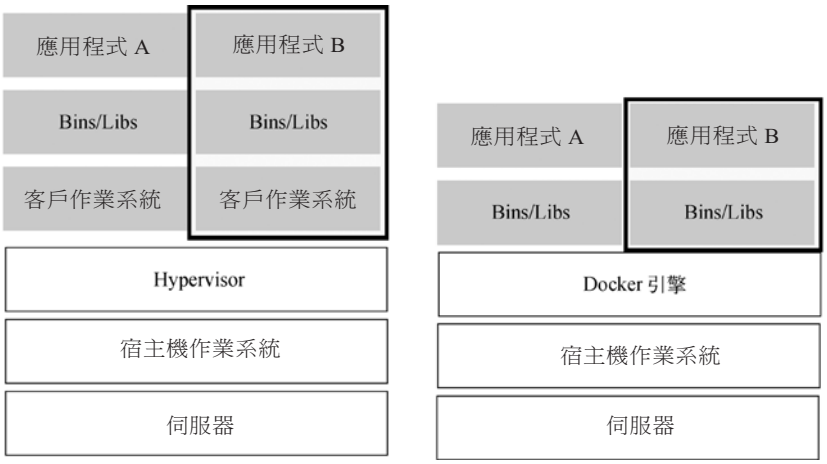

圖 2-1　虛擬機和容器的異同

圖 2-1 的左邊畫出了虛擬機的工作原理。其中，名為 Hypervisor 的軟體是虛擬機最主要的部分。它透過硬體虛擬化功能模擬出了執行一個作業系統所需要的各種硬體，例如 CPU、記憶體、I/O 裝置等。然後，它在這些虛擬硬體上安裝了一個新的作業系統 —— **客戶作業系統**（Guset OS）。

這樣，使用者的應用程式行程就可以在這個虛擬的機器中執行了，它能「看到」的自然也只有客戶作業系統的檔案和目錄，以及這台機器裡的虛擬裝置。這就是為什麼虛擬機也具有將不同的應用程式行程相互隔離的作用。

圖 2-1 的右邊則用一個名為 Docker 引擎的軟體取代了 Hypervisor。這也是很多人把 Docker 稱為「輕量級」虛擬化技術的原因，實際上就是把虛擬機的概念套用在容器上。

可是這樣的說法並不嚴謹。

在理解了 Namespace 的工作方式之後，你就會明白，和真實存在的虛擬機不同，在使用 Docker 時，並沒有一個真正的「Docker 容器」在宿主機中執行。Docker 幫助使用者啟動的還是原來的應用程式行程，只不過在建立這些行程時，Docker 為它們加上了各式各樣的 Namespace 參數。

更重要的是，由於執行這個掛載操作時「容器行程」已經建立了，也就意味著此時 Mount Namespace 已經開啟了，因此這個掛載事件只在該容器裡可見。在宿主機上看不到容器內部的這個掛載點，這就避免了 Volume 打破容器的隔離性。

注意，這裡提到的「容器行程」是 Docker 建立的一個容器初始化行程（`dockerinit`），而不是應用程式行程（`ENTRYPOINT+CMD`）。dockerinit 會負責完成根目錄的準備、掛載裝置和目錄、配置 hostname 等一系列需要在容器內進行的初始化操作。最後，它透過 `execv()` 系統呼叫讓應用程式行程取代自己成為容器裡 PID=1 的行程。

這裡要用到的掛載技術就是 Linux 的**綁定掛載**（bind mount）機制。它的主要作用是，允許你將一個目錄或者檔案而不是整個裝置掛載到指定目錄上。並且，這時你在該掛載點上進行的任何操作，只是發生在被掛載的目錄或者檔案上，而原掛載點的內容會被隱藏起來且不受影響。

其實，如果你了解 Linux 核心，就會明白，綁定掛載實際上是一個 inode 取代的過程。在 Linux 作業系統中，可以把 inode 理解為存放檔案內容的「物件」，而 dentry（目錄項）就是訪問這個 inode 所使用的「指標」，見圖 2-4。

圖 2-4 綁定掛載示意圖

如圖 2-4 所示，`mount --bind /home /test` 會將 /home 掛載到 /test 上。這其實相當於將 /test 的 dentry 重定向到了 /home 的 inode。這樣當我們修改 /test 目錄時，實際上修改的是 /home 目錄的 inode。因此，一旦執行 `umount` 指令，/test 目錄原先的內容就會復原，因為修改實際發生在 /home 目錄裡。

沒錯。容器就是未來雲端運算系統中的行程，容器鏡像就是這個系統裡的 .exe 安裝包。那麼 Kubernetes 呢？

你應該也能立刻回答上來：Kubernetes 就是作業系統！

非常正確。

現在，我們登入一台 Linux 機器並執行如下指令：

```
$ pstree  g
```

這條指令的作用是展示目前系統中正在執行的行程的樹狀結構。它的返回結果如下所示：

```
systemd(1) + accounts daemon(1984) + {gdbus}(1984)
           |                       ` {gmain}(1984)
           | acpid(2044)
          ...
           | lxcfs(1936) + {lxcfs}(1936)
           |             ` {lxcfs}(1936)
           | mdadm(2135)
           | ntpd(2358)
           | polkitd(2128) + {gdbus}(2128)
           |                ` {gmain}(2128)
           | rsyslogd(1632) + {in:imklog}(1632)
           |                | {in:imuxsock) S 1(1632)
           |                ` {rs:main Q:Reg}(1632)
           | snapd(1942) + {snapd}(1942)
           |             | {snapd}(1942)
           |             | {snapd}(1942)
           |             | {snapd}(1942)
           |             | {snapd}(1942)
```

不難發現，在一個真正的作業系統裡，行程並不是「孤苦伶仃」地執行的，而是以行程組的方式「有原則」地組織在一起。例如，這裡有一個叫作 rsyslogd 的程式，它負責的是 Linux 作業系統中的日誌處理。可以看到，rsyslogd 的主程式 main，和它要用到的核心日誌模組 imklog 等，同屬於 1632 行程組。這些行程相互協作，共同履行 rsyslogd 程式的職責。

如此看來，一個有 A、B 兩個容器的 Pod，不就等同於一個容器（容器 A）共享另外一個容器（容器 B）的網路和 Volume 的做法嗎？

這好像透過 `docker run --net --volumes-from` 這樣的指令就能實現，例如：

```
$ docker run   net=B   volumes from=B   name=A image A ...
```

但是，你是否考慮過，如果真這樣做的話，容器 B 就必須比容器 A 先啟動，這樣一個 Pod 裡的多個容器就不是對等關係，而是拓撲關係了。

所以，在 Kubernetes 裡，Pod 的實現需要使用一個中間容器，這個容器叫作 Infra 容器。在 Pod 中，Infra 容器永遠是第一個被建立的容器，使用者定義的其他容器，則透過 Join Network Namespace 的方式與 Infra 容器關聯在一起。圖 5-1 展示了這樣的組織關係。

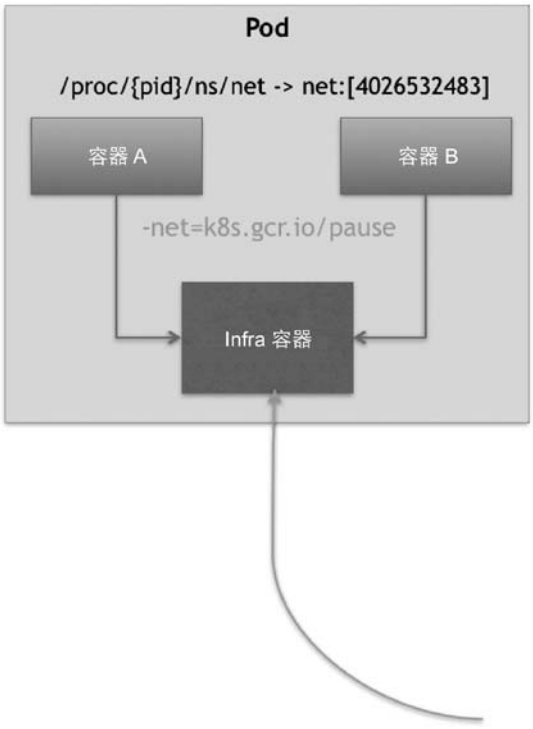

圖 5-1　組織關係示意圖

```
  rollingUpdate:
    maxSurge: 1
    maxUnavailable: 1
```

在 RollingUpdateStrategy 的配置中，maxSurge 指定的是除 DESIRED 數量外，在一次滾動更新中 Deployment 控制器還可以建立多少新 Pod；而 maxUnavailable 指的是在一次滾動更新中 Deployment 控制器可以刪除多少舊 Pod。這兩個配置還可以用前面介紹的百分比形式來表示，例如 maxUnavailable=50% 指的是一次最多可以刪除「50%*DESIRED 數量」個 Pod。

結合以上講述，下面圖 5-4 為擴展 Deployment、ReplicaSet 和 Pod 的關係圖。

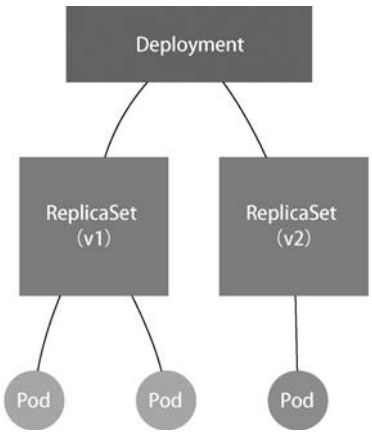

圖 5-4　Deployment、ReplicaSet 和 Pod 的關係圖

如圖 5-4 所示，Deployment 的控制器實際上控制的是 ReplicaSet 的數目，以及每個 ReplicaSet 的屬性。而一個應用程式的版本對應的正是一個 ReplicaSet，這個版本應用程式的 Pod 數量則由 ReplicaSet 透過它自己的控制器（ReplicaSet Controller）來保證。透過這樣的多個 ReplicaSet 物件，Kubernetes 就實現了對多個應用程式版本的描述。

明白了「應用程式版本和 ReplicaSet 一一對應」的設計理念之後，下面講解 Deployment 對應用程式進行版本控制的具體原理。

這一次，我會使用一個叫 kubectl set image 的指令，直接修改 nginx-deployment 所使用的鏡像。這個指令的好處就是，不用像 kubectl edit 需要打開編輯器。不過這一次，我把這個鏡像名字改為了一個錯誤的名字，例如 nginx:1.91。這樣，Deployment 就會出現一個升級失敗的版本。

下面實踐一下：

```
$ kubectl set image deployment/nginx deployment nginx=nginx:1.91
deployment.extensions/nginx deployment image updated
```

由於這個 nginx:1.91 鏡像在 Docker Hub 中並不存在，因此 Deployment 的滾動更新被觸發後會立刻會報錯誤並停止。

這時檢查一下 ReplicaSet 的狀態，如下所示：

```
$ kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
nginx deployment 1764197365   2         2         2       24s
nginx deployment 3167673210   0         0         0       35s
nginx deployment 2156724341   2         2         0       7s
```

返回結果顯示，新版本的 ReplicaSet（hash=2156724341）的水準擴展已經停止。而且，此時它已經建立了兩個 Pod，但是它們都沒有進入 READY 狀態。這當然是因為這兩個 Pod 都拉取不到有效的鏡像。與此同時，舊版本的 ReplicaSet（hash=1764197365）的水準收縮也自動停止了。此時，已經有一個舊 Pod 被刪除，還剩下兩個舊 Pod。

那麼問題來了，如何讓 Deployment 的 3 個 Pod 都回滾到舊版本呢？只需要執行一條 kubectl rollout undo 指令，就能把整個 Deployment 回滾到上一個版本：

```
$ kubectl rollout undo deployment/nginx deployment
deployment.extensions/nginx deployment
```

實際操作時，Deployment 的控制器其實就是讓舊的 ReplicaSet（hash=1764197365）再次擴展成 3 個 Pod，並讓新的 ReplicaSet（hash=2156724341）

更進一步，這意味著 kube-apiserver 在響應指令式請求（例如 kubectl replace）時，一次只能處理一個寫入請求，否則可能產生衝突。而對於宣告式請求（例如 kubectl apply），一次能處理多個寫操作，並且具備 Merge 能力。

可能一開始覺得這種區別沒那麼重要。而且，正是由於要考慮這樣的 API 設計，做同樣一件事情，Kubernetes 需要的步驟往往要比其他產品多不少。但是，如果仔細思考 Kubernetes 的工作流程，就不難發現這種宣告式 API 的獨到之處。

接下來以 Istio 為例，說明宣告式 API 在實際使用時的重要意義。

2017 年 5 月，Google、IBM 和 Lyft 公司共同宣布了 Istio 開源專案的誕生。很快地，Istio 就在技術圈掀起了一波「微服務」的熱潮，把 Service Mesh 這個新的編排概念推到了風口浪尖。

Istio 實際上就是一個基於 Kubernetes 的微服務管理框架。它的架構非常清晰，如圖 5-7 所示。

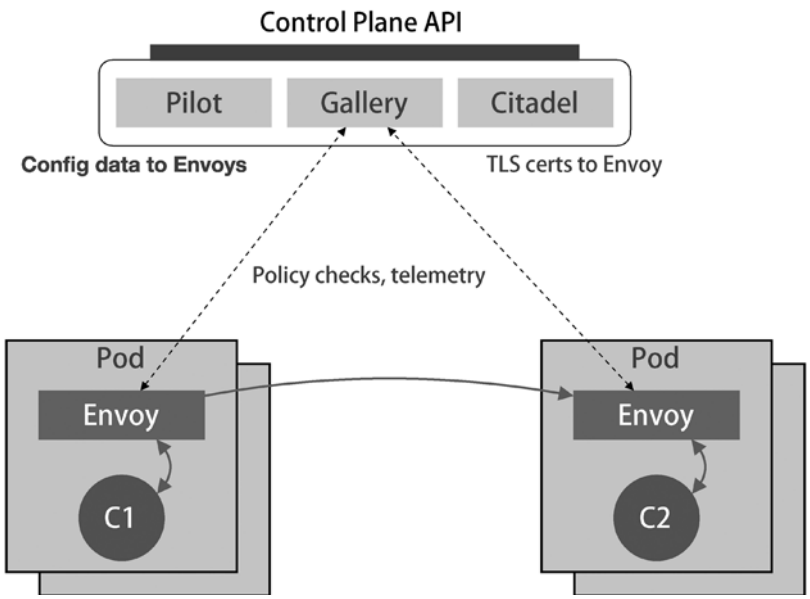

圖 5-7　Istio 架構示意圖

首先，Kubernetes 會匹配 API 物件的組。需要明確的是，對於 Kubernetes 裡的核心 API 物件，例如 Pod、Node 等，是不需要 Group 的（它們的 Group 是 ""）。所以，對於這些 API 物件來說，Kubernetes 會直接在 /api 這個層級進行下一步的匹配過程。

對於 CronJob 等非核心 API 物件來說，Kubernetes 就必須在 /apis 這個層級尋找它對應的 Group，進而根據「batch」（離線業務）這個 Group 的名字找到 /apis/batch。

不難發現，這些 API Group 是以物件功能進行分類的，例如 Job 和 CronJob 都屬於「batch」這個 Group。

然後，Kubernetes 會進一步匹配 API 物件的版本號。

對於 CronJob 這個 API 物件來說，Kubernetes 在 batch 這個 Group 下匹配到的版本號就是 `v2alpha1`。

在 Kubernetes 中，同一種 API 物件可以有多個版本，這正是 Kubernetes 進行 API 版本化管理的重要手段。這樣，例如在 CronJob 的開發過程中，對於會影響使用者的變更，就可以透過升級新版本來處理，從而保證向後相容。

最後，Kubernetes 會匹配 API 物件的資源類型。

在前面匹配到正確的版本之後，Kubernetes 就知道我要建立的原來是一個 /apis/batch/v2alpha1 下的 CronJob 物件。此時，API Server 就可以繼續建立這個 CronJob 物件了。圖 5-9 總結了建立流程，以方便理解。

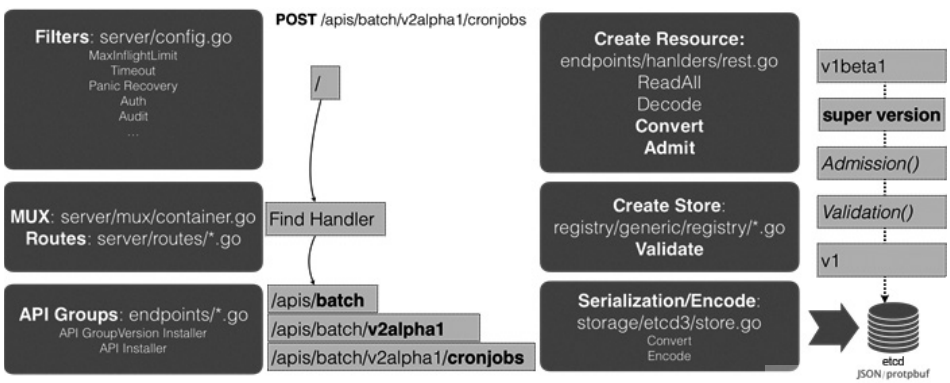

圖 5-9　CronJob 物件建立流程圖

首先，當我們發起了建立 CronJob 的 POST 請求之後，我們編寫的 YAML 的訊息就提交給了 API Server。

API Server 首先過濾這個請求，並完成一些前置性工作，例如授權、超時處理、審計等。

然後，請求會進入 MUX 和 Routes 流程。如果你編寫過 Web Server 就會知道，MUX 和 Routes 是 API Server 完成 URL 和 Handler 綁定的場所。而 API Server 的 Handler 要做的就是按照剛剛介紹的匹配過程，找到對應的 CronJob 類型定義。

接著，到了 API Server 最重要的職責：根據這個 CronJob 類型定義，使用使用者提交的 YAML 檔裡的欄位，來建立一個 CronJob 物件。

在此過程中，API Server 會進行一個 Convert 工作：把使用者提交的 YAML 檔轉換成一個名為 Super Version 的物件，它正是該 API 資源類型所有版本的欄位全集。這樣使用者提交的不同版本的 YAML 檔，就都可以用這個 Super Version 物件來進行處理了。

接下來，API Server 會先後進行 `Admission()` 和 `Validation()` 操作。例如，上一節提到的 Admission Controller 和 Initializer 就都屬於 Admission 的內容。

Validation 則負責驗證這個物件裡的各個欄位是否合法。經過驗證的 API 物件儲存在了 API Server 裡一個叫作 Registry 的資料結構中。也就是說，只要一個 API 物件的定義能在 Registry 裡查到，它就是有效的 Kubernetes API 物件。

最後，API Server 會把經過驗證的 API 物件轉換成使用者最初提交的版本，進行序列化操作，並呼叫 etcd 的 API 將其儲存。

由此可見，宣告式 API 對於 Kubernetes 來說非常重要。所以，API Server 這樣一個在其他專案中「平淡無奇」的元件，卻成了 Kubernetes 的重中之重。它不僅是 GoogleBorg 設計理念的集中體現，也是 Kubernetes 裡唯一被 Google 公司和 Red Hat 公司雙重控制、其他「勢力」根本無法參與其中的元件。