虛擬機研究-docker

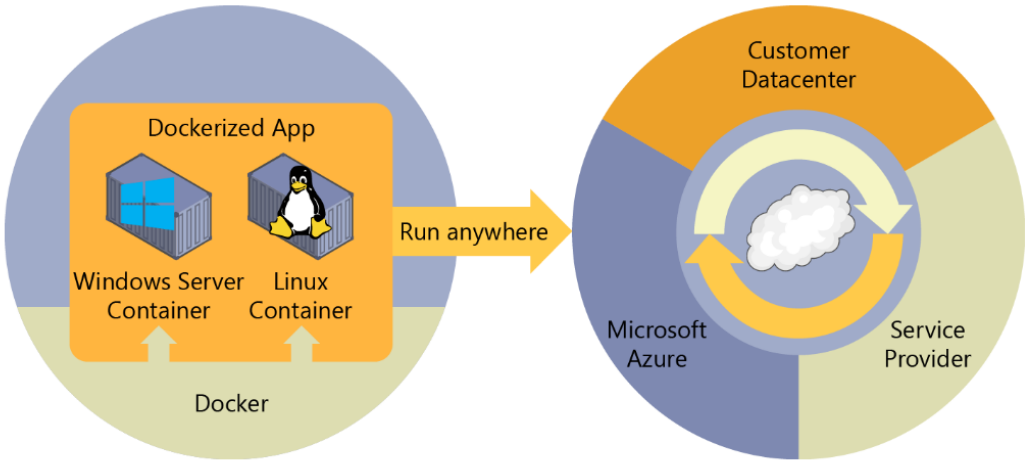
班級：資工二
姓名：鍾瑄
學號：110710520

內容

一、docker 是什麼？

Docker 是一個開放原始碼軟體平台，用於開發、交付、執行應用。其允許用戶將基礎設中的應用單獨分割出來，形成更小的顆粒（容器），從而提高速度。

- 使用 Go 語言進行開發，基於 Linux 內核的 cgroup，namespace，以及 AUFS 類的 UnionFS 等技術，進行封裝隔離。
- 能夠自動執行重複性任務，例如搭建和配置開發環境，讓開發人員能專注在真正重要的事情上。
- 用戶可以方便地創建和使用容器，把自己的應用放入容器。容器還可以進行版本管理、複製、分享、修改，就像管理普通的代碼一樣。
- 能將應用程式自動化部署為可攜式且可自足的容器，在雲端或內部部署上執行



圖：Docker 將容器部署在混合式雲端的所有圖層

** Go：Google 開發的程式語言*

** cgroup：是 Linux 核心的一個功能，用來限制、控制與分離一個行程群組的資源（如 CPU、記憶體、磁碟輸入輸出等）。*

** namespace：是 Linux 內核的一項功能，它對內核資源進行分區，以使一組進程看到一組資源，而另一組進程看到另一組資源。*

** AUFS：用於為 Linux 檔案系統實現聯合掛載*

** UnionFS：用於 Linux 的文件系統服務，它為其他文件系統實現聯合掛載*

二、容器介紹

容器就是將軟體變成一個標準化單元，以用來開發、交付和部署。用簡單一點的說法來說的話，容器就是一個存放東西的地方，就像書包可以裝文具、書本、水壺一樣。現在所說的容器存放的東西更偏向于應用比如網站、程式甚至是系統環境。

三、docker 與一般虛擬機之間的差別

Docker 容器與虛擬機器類似，但原理上，容器是將作業系統層虛擬化，虛擬機器則是虛擬化硬體，所以容器更具有可攜式性且更能高效地利用伺服器。且多個容器可以在同一台機器上運行，共用作業系統內核，但各自獨立的在使用者空間中運行，而虛擬機則是將一台伺服器變成多台伺服器。允許多個虛擬機在一台機器上運行。每個虛擬機都包含一整套作業系統、一個或多個應用、必要的二進位檔案和資源，因此佔用的空間較容器來的大很多。

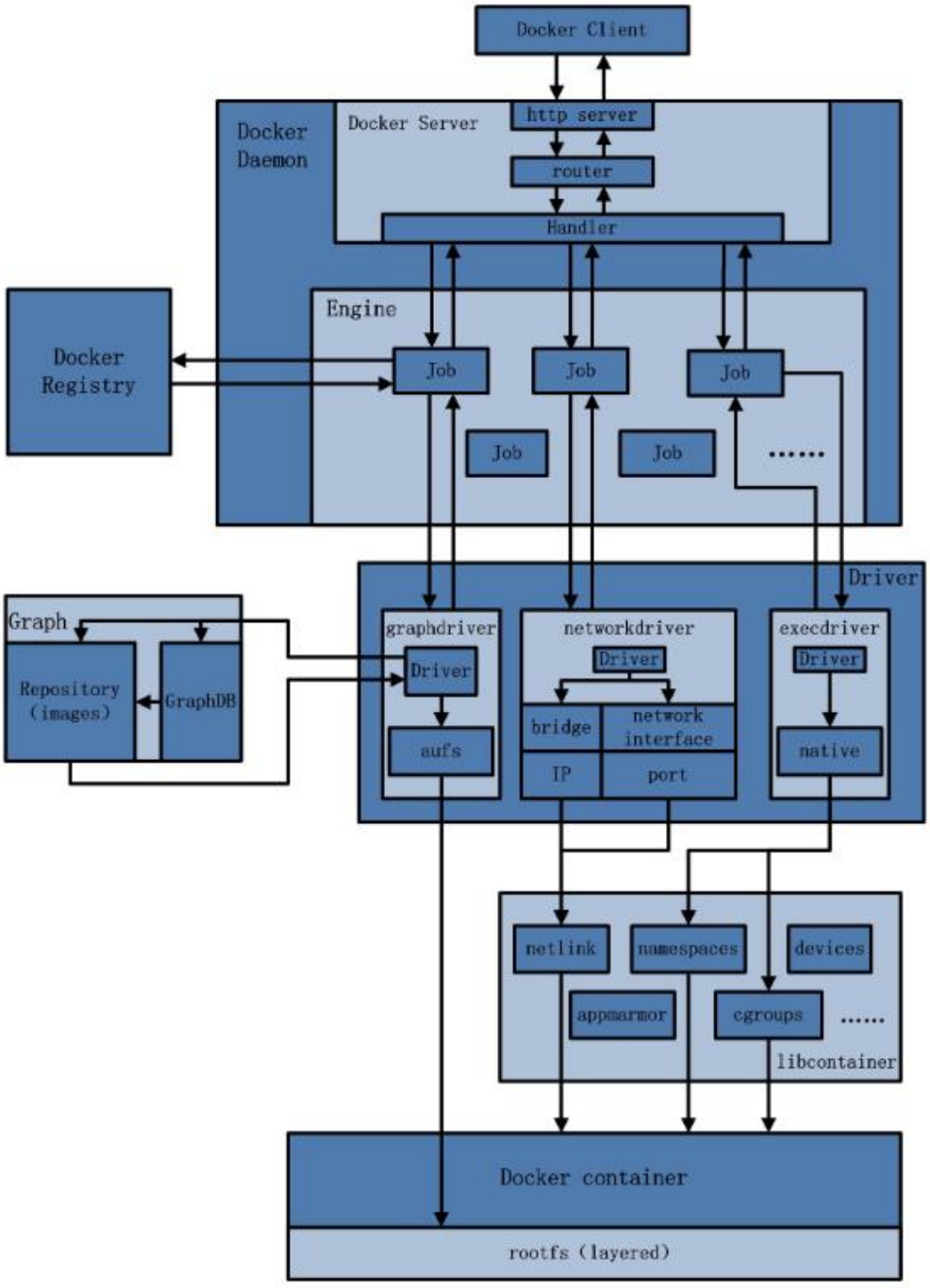
特性	容器	虛擬機
啟動時間	秒	分鐘
硬碟使用	MB	GB
系統支持量	上千個	幾十個

表：容器與虛擬機的比較

圖：虛擬機和容器比較的另類比喻

四、docker 總架構圖

Docker 並不像其他大型分散式系統那樣複雜。其主要的模組有: DockerC1ient、DockerDaemon、Docker Registry、Graph、Driver、libcontainer 以及 Docker Container。



圖：docker 總架構圖

（一）Docker Client

Docker Client 是與 Docker Daemon 建立通信的客戶端。用戶可以使用可執行檔 docker 作為 Docker Client，發起眾多 Docker 容器的管理請求。Docker Client 主要透過以下三種方式和 Docker Daemon 建立通信，分別為: tcp://host: port、unix://path _ to _ socket 和 fd://socketfd 。

（二）Docker Daemon

Docker Daemon 是 Docker 架構中一個常駐在後臺的系統進程。所謂的"運行 Docker" ,即代表運行 Docker Daemon 。DockerDaemon 的作用主要有以下兩方面:

1、接收並處理 Docker Client 發送的請求。

2、管理所有的 Docker 容器。

Docker Daemon 運行時，會在後臺啟動一個 Server 負責接收 Docker Client 發送的請求;接收請求後，Server 通過路由與分發調度，找到相應的 Handler 來處理請求。其架構大致可以分為三部分: Docker Server、 Engine 和 Job。

Engine

Engine 是 Docker 架構中的運行引擎，同時也是 Docker 運行的核心模組。存儲著大量的容器資訊，同時管理著 Docker 大部分 Job 的執行。所以 Docker 中大部分任務的執行都需要 Engine 協助，並通過 Engine 匹配相應的 Job 完成執行。

Job

Job 是 Engine 內部最基本的工作執行單元。 DockerDaemon 完成的每一項工作都會呈現為一個 Job 。例如，在 Docker 容器內部運行一個進程、創建一個新的容器、在網路上下載一個文檔，這都是一個 Job。

Docker Registry

Docker Registry 是一個儲存容器鏡像(Docker Image) 的倉庫。容器鏡像是容器創建時用來初始化容器的檔案系統內容。其將大量的容器鏡像匯集在一起，並為分散的 Docker Daemon 提供鏡像服務。Docker 的運行過程中，有三種情況與 Docker Registry 通信，分別為搜索鏡像、下載鏡像、上傳鏡像。這三種情況所對應的 Job 名稱分別為 search、pull 和 push。

Graph Docker

Graph Docker 是容器鏡像的保管者。Docker 下載的鏡像和 Docker 構建的鏡像，都是由 Graph 統一化管理。由於 Docker 支援多種不同的鏡像存儲方式，如 aufs、devicemapper、Btrfs 等，故 Graph 對鏡像的儲存也會因以上種類而存在一些差異。

** Btrfs：一種支援寫入時複製的檔案系統。*

** devicemapper：設備映射器是 Linux 內核提供的框架，用於將物理塊設備映射到更高級別的虛擬塊設備。*

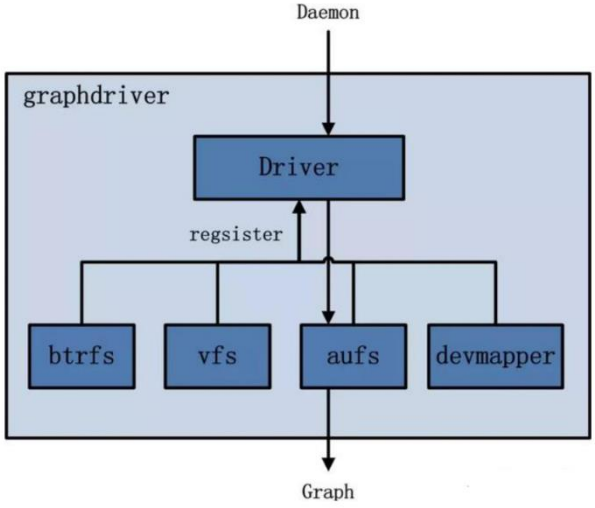
Driver

Driver Docker 架構中的驅動模組。通過 Driver 驅動，可以實現對容器運行環境的定制，定制的維度主要有網路環境、存儲方式以及容器執行方式。其實現可以分為以下三類驅動: graphdriver、networkdriver 和 execdriver。

1、**graphdriver**：主要用於容器鏡像的管理。其包括：

- 從遠端 Docker Registry 上下載鏡像並進行儲存。
- 本地構建完鏡像後的儲存。

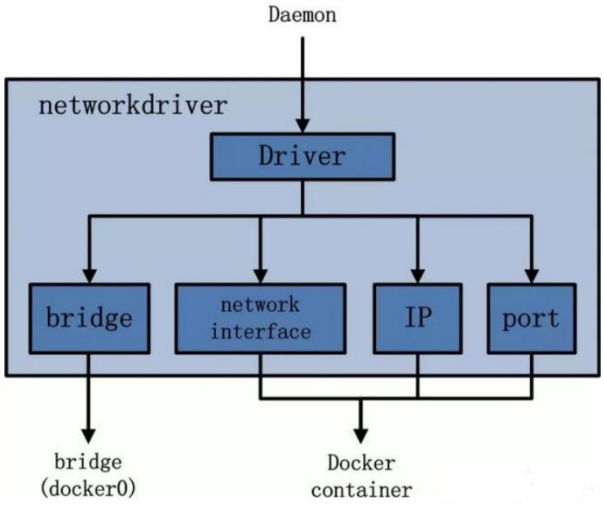
包含了四種系統(btrfs, vfs, aufs, devmapper)驅動 Drive 在 Docker Danmon 下完成註冊。



圖：graphdriver 架構圖

2、**networkdriver**：完成容器網路環境的配置。其中包括:

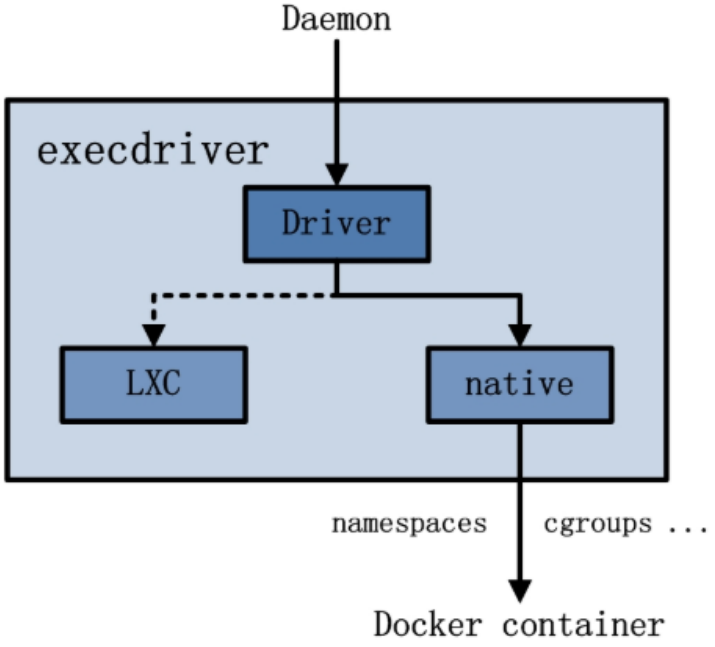
- Docker 啟動時為 Docker 環境創建網橋。
- Docker 容器創建時為其創建專屬虛擬網卡設備。
- Docker 容器分配 IP、埠並與宿主機做埠映射，設置容器防火牆策略等。



圖：networkdriver 架構圖

3、**Execdriver**：為 Docker 容器的執行驅動。其負責：

- 創建容器運行命名空間。
- 容器資源使用的統計與限制。
- 器內部進程的真正運行等。



圖：Execdriver 架構圖

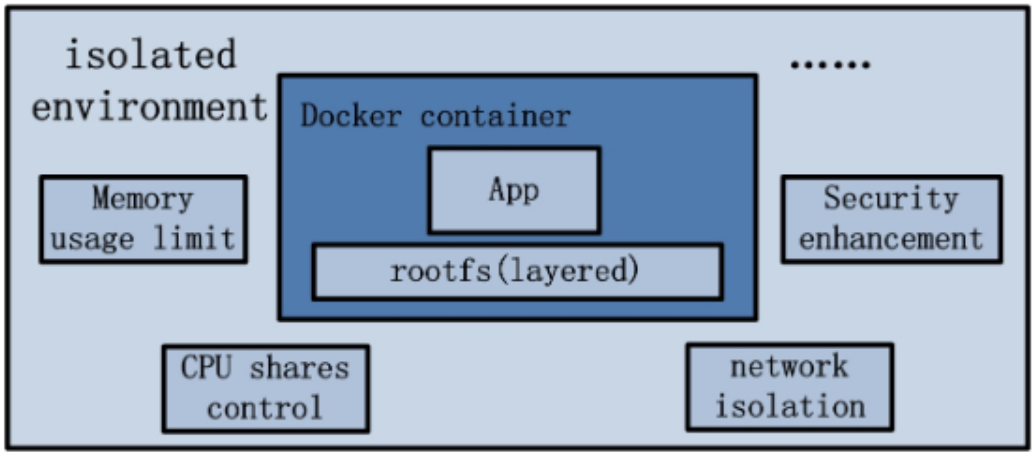
Libcontainer

Libcontainer 是 Docker 架構中一個使用 Go 語言設計的庫，設計初衷是希望該庫可以不依靠任何依賴，可以直接訪問內核中與容器相關的 API。Docker 可以直接調用 Libcontainer 來操縱容器的 Namespace、Cgroups、Apparmor、網絡設備以及防火牆規則等。且 Libcontainer 提供了一整套標準的接口來滿足上層對容器管理的需求。或者說 Libcontainer 屏蔽了 Docker 上層對容器的直接管理。

Docker Container

Docker Container 是 Docker 架構中服務交付的最終體現形式。Docker 按照用戶的需求與指令，訂製相應的 Docker 容器：

- 用戶通過指定容器鏡像，使得 Docker 容器可以自定義 rootfs 等文件系統。
- 用戶通過指定計算資源的配額，使得 Docker 容器使用指定的計算資源。
- 用戶通過配置網絡及其安全策略，使得 Docker 容器擁有獨立且安全的網絡環境。
- 用戶通過指定運行的命令，使得 Docker 容器執行指定的工作。



圖：Docker Container 架構圖

五、docker 部分源碼解析

（一）Docker 命令的 flag 參数解析

```
func main() {
    if reexec.Init() {
        return
    }
    flag.Parse()
    // FIXME: validate daemon flags here
    ...
}
```

首先判斷 reexec.Init( )，若為 true，則直接退出運行，否則將繼續執行。由於在 docker 運行之前沒有任何 Initializer 註冊，故該程式碼片段執行的返回值為 false。reexec 的作用是協調 execdriver 與容器創建時 dockerinit 這兩者的關係。

```
var (
    flVersion = flag.Bool([]string{"v", "-version"}, false, "Print version
    information and quit")
    flDaemon = flag.Bool([]string{"d", "-daemon"}, false, "Enable daemon mode")
    flDebug = flag.Bool([]string{"D", "-debug"}, false, "Enable debug mode")
    flSocketGroup = flag.String([]string{"G", "-group"}, "docker", "Group to
    assign the unix socket specified by -H when running in daemon modeuse '' (the
    empty string) to disable setting of a group")
    flEnableCors = flag.Bool([]string{"#api-enable-cors", "-api-enable-cors"},
    false, "Enable CORS headers in the remote API")
    flTls = flag.Bool([]string{"-tls"}, false, "Use TLS; implied by tls-verify
    flags")
    flTlsVerify = flag.Bool([]string{"-tlsverify"}, false, "Use TLS and verify
    the remote (daemon: verify client, client: verify daemon)")
```

```
    // these are initialized in init() below since their default values depend
    on dockerCertPath which isn't fully initialized until init() runs
    flCa     *string
    flCert   *string
    flKey    *string
    flHosts []string
    )

    func init() {
    flCa = flag.String([]string{"-tlscacert"}, filepath.Join(dockerCertPath,
    defaultCaFile), "Trust only remotes providing a certificate signed by the CA
    given here")
    flCert = flag.String([]string{"-tlscert"}, filepath.Join(dockerCertPath,
    defaultCertFile), "Path to TLS certificate file")
    flKey = flag.String([]string{"-tlskey"}, filepath.Join(dockerCertPath,
    defaultKeyFile), "Path to TLS key file")
    opts.HostListVar(&flHosts, []string{"H", "-host"}, "The socket(s) to bind to
    in daemon mode\nspecified using one or more tcp://host:port, unix:///path/
    to/socket, fd://* or fd://socketfd.")
}
```

以上源碼展示了 Docker 如何定義 flag 參數，以及在 init 函數中實現部分 flag 參數的初始化。 Docker main 函數執行前，這些變數創建以及初始化工作已經全部完成。這裡涉及了 Go 語言的一個特性，即 init 函數的執行。

（二）處理 flag 資訊並收集 Docker Client 的配置資訊

- 處理的 flag 參數有: flVersion、flDebug、flDaemon、fltTlsVerify 以及 fltTls。
- Docker Client 收集的配置資訊有: protoAddrParts (通過 fltHosts 參數獲得，作用是提供 Docker Client 和 Docker Server 的通信協議以及通信地址)、tlsConfig (通過一系列 flag 參數獲得，如 *fltTls、*flTlsVerify，作用是提供安全傳輸層協議的保障)。

Main 函數的源碼分析：

```
if *flVersion {
      showVersion()
      return
}
```

此源碼代表著，若 Docker 發現 flag 參數 flVersion 為真，則說明 Docker 使用者希望查看 Docker 的版本資訊。此時 Docker 調用 showVersion( )顯示版本資訊，並從 main 函數退出;否則的話，繼續往下執行。

```
if *flDebug {
      os.Setenv("DEBUG", "1")
}
```

flDebug 參數為真的話，Docker 通過 os 包中的 Setenv 函數創建一個名為 DEBUG 環境變數，並將其值設為 "1"; 繼續往下執行。

```
if len(flHosts) == 0 {
      defaultHost := os.Getenv("DOCKER_HOST")
      if defaultHost == "" || *flDaemon {
           // If we do not have a host, default to unix socket
           defaultHost = fmt.Sprintf("unix://%s", api.DEFAULTUNIXSOCKET)
      }
      if _, err := api.ValidateHost(defaultHost); err != nil {
           log.Fatal(err)
      }
      flHosts = append(flHosts, defaultHost)
}
```

flHosts 的作用是為 Docker Client 提供所要連接的 host 物件，也就是為 Docker Sever 提供所要監聽的物件。首先判斷 flHosts 變數是否長度為 0。如果是，則說明用戶並沒有傳入地址，此時 Docker 通過 os 包獲取名為 DOCKER HOST 環境變數的值，將其賦值於 defaultHost。若 defaultHost 為空或者 flDaemon 為真，說明目前還沒有一個定義的 host 物件，則將其預設設置為 unix socket。驗證該 defaultHost 的合法性之後，將 defaultHost 的值追加至 flHost 的末尾，繼續往下執行。當然若 flHost 的長度不為 0，則說明用戶已經指定位址，同樣繼續往下執行。

```
if *flDaemon {
      mainDaemon()
      return
}
```

若 flDaemon 參數為真，則需求是啟動 Docker Daemono，Docker 隨即執行 mainDaemon 函數，實現 Docker Daemon 的啟動，若 mainDaemon 函數執行完畢，則退出 main 函數。

```
if len(flHosts) > 1 {
       log.Fatal("Please specify only one -H")
}
protoAddrParts := strings.SplitN(flHosts[0], "://", 2)
```

由於不執行 Docker Daemon 的啟動流程，故屬於 Docker Client 的執行邏輯。首先判斷 flHosts 的長度是否大於 1。若大於 1，則說明需要新創建的 Docker Client 訪問不止 1 個位址，顯然邏輯上行不通，故拋出錯誤，提醒使用者只能指定一 Docker Daemon 地址。接著，Docker flHosts 這個 string 陣列中的第一個元素進行分割，通過"://"來分割，分割出的兩個部分放入變數 protoAddrParts 陣列中。protoAddrParts 的作用是解析出 Docker Client 和 Docker Server 建立通信的協定與位址。

```
var (
     cli       *client.DockerCli
     tlsConfig tls.Config
)
tlsConfig.InsecureSkipVerify = true
```

如果假設 flDaemon 為假，可以認定 main 函數的運行是為了 Docker Client 的創建與執行。 Docker 在這裡創建了兩個變數，一個為類型是*c1ient. DockerCli 的對象 cli ，另一個為類型是 tls.Config 的對象 tlsConfig 。定義完變數之後，Docker tlsConfig InsecureSkip Veri 市屬性置為真。 tlsConfig 物件的創建是為了保障 cli 在傳輸資料的時候遵循安全傳輸層協議(TLS) 。安全傳輸層協定(TLS) 用於確保兩個通信應用程式之間的保密性與資料完整性。 tlsConfig 是 Docker Client 創建過程中可選的配置資訊。

```
// If we should verify the server, we need to load a trusted ca
if *flTlsVerify {
     *flTls = true
     certPool := x509.NewCertPool()
     file, err := ioutil.ReadFile(*flCa)
     if err != nil {
          log.Fatalf("Couldn't read ca cert %s: %s", *flCa, err)
     }
     certPool.AppendCertsFromPEM(file)
     tlsConfig.RootCAs = certPool
     tlsConfig.InsecureSkipVerify = false
}
```

若 flT1sVerify 這個 flag 參數為真，則說明 Docker Client 和 Docker Server 一起驗證連接的安全性。此時，tlsConfig 物件需要載入一個接受信息的 ca 檔。ca 檔的路徑為*flCA 參數的值，最終完成 tlsConfig 物件中 RootCAs 屬性的賦值，並將 InsecureSkipVerify 屬性置為假。

```
// If tls is enabled, try to load and send client certificates
if *flTls || *flTlsVerify {
     _, errCert := os.Stat(*flCert)
     _, errKey := os.Stat(*flKey)
     if errCert == nil && errKey == nil {
          *flTls = true
          cert, err := tls.LoadX509KeyPair(*flCert, *flKey)
          if err != nil {
               log.Fatalf("Couldn't load X509 key pair: %s. Key encrypted?", err)
          }
          tlsConfig.Certificates = []tls.Certificate{cert}
     }
}
```

如果 flT1s 和 flT1sVerify 兩個 flag 參數中有一個為真，則說明需要載入並發送用戶端的證書。最終將證書內容交給 tlsConfig 的 Certificates 屬性。 至此，flag 參數已經全部處理完畢， DockerClient 也已經收集到所需的配置資訊。

（三）創建 Docker Client

```
if *flTls || *flTlsVerify {
     cli = client.NewDockerCli(os.Stdin, os.Stdout, os.Stderr,
     protoAddrParts[0], protoAddrParts[1], &tlsConfig)
} else {
     cli = client.NewDockerCli(os.Stdin, os.Stdout, os.Stderr,
       protoAddrParts[0], protoAddrParts[1], nil)
       }
```

若 flag 參數 flTls 為真或者 flTlsVerify 為真，則說明需要使用 TLS 協定來保障傳輸的安全性，故創建 Docker Client 的時候，將 t1sConfig 參數傳入，否則同樣創建 Docker Client, 只不過 tlsConfig 為 nil。

```
func NewDockerCli(in io.ReadCloser, out, err io.Writer, proto, addr string,
tlsConfig *tls.Config) *DockerCli {
var (
     isTerminal = false
     terminalFd uintptr
     scheme     = "http"
)

if tlsConfig != nil {
     scheme = "https"
}

if in != nil {
     if file, ok := out.(*os.File); ok {
          terminalFd = file.Fd()
          isTerminal = term.IsTerminal(terminalFd)
     }
}

if err == nil {
     err = out
}
return &DockerCli{
     proto:      proto,
     addr:       addr,
     in:         in,
     out:        out,
     err:        err,
     isTerminal: isTerminal,
     terminalFd: terminalFd,
     tlsConfig:  tlsConfig,
     scheme:     scheme,
     }
}
```

DockerCli 的屬性有 proto、DockerClient、Docker Server 的傳輸協議，addr、Docker Client 需要訪問的 host 目標位址，tlsConfig，安全傳輸層協定的配置。若 t1sConfig 不為空，則說明需要使用安全傳輸層協定，DockerCli 物件的 scheme 設置為" https"，另外還有關於輸入、輸出以及錯誤顯示的配置等。最終函數返回 DockerCli 物件。通過調用 client 包中的 NewDockerCli 函數，程式最終創建了 Docker Client ，返回 main 包中的 main 函數之後，程式繼續往下執行。

六、參考資料

（一）網路資料

1.https://docs.microsoft.com/zh-tw/dotnet/architecture/containerized-lifecycle/what-is-docker

2.https://juejin.im/post/5b260ec26fb9a00e8e4b031a

3. https://www.zhihu.com/question/28300645

4. https://www.itread01.com/content/1548105503.html

5. https://codertw.com/%E7%A8%8B%E5%BC%8F%E8%AA%9E%E8%A8%80/679879/

（二）書籍資料

Docker 源碼分析：https://www.tenlong.com.tw/products/9787111510727