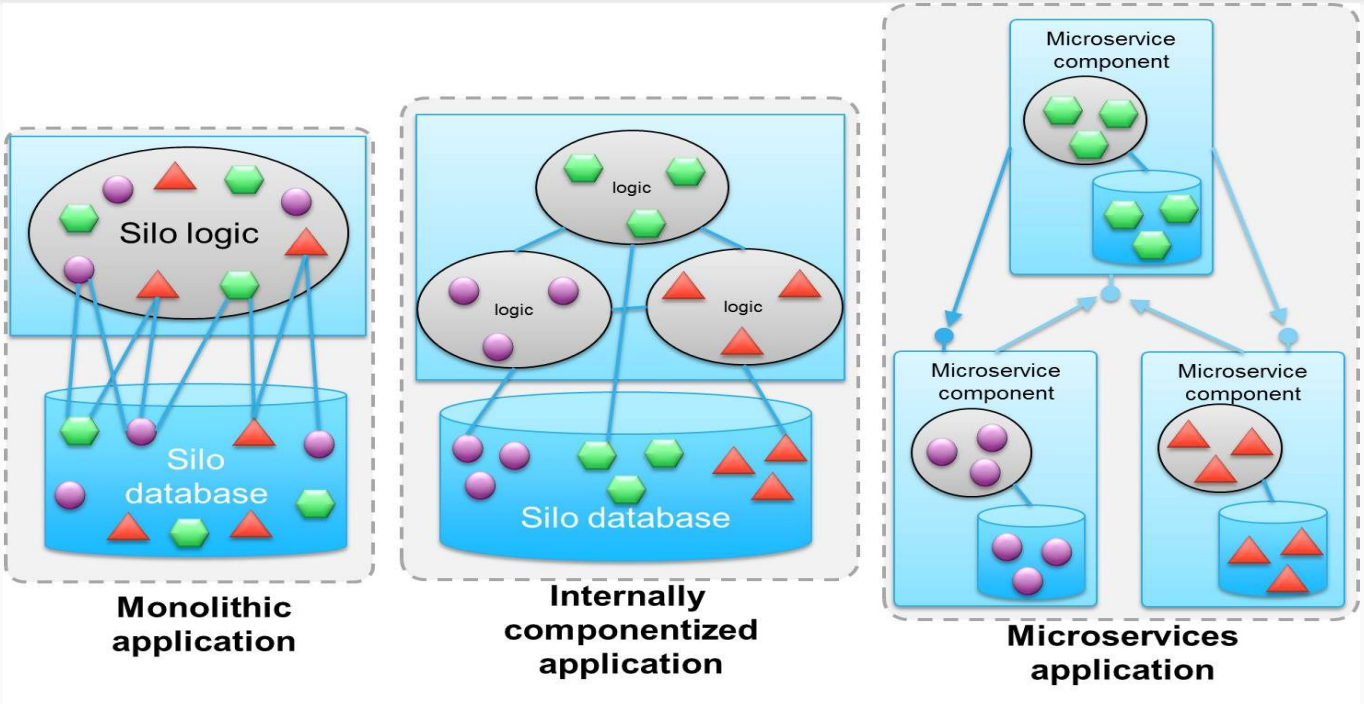
元件和相依性管理 - Part 1

Raix Lai

2018/12/13

單體式系統 V.S. 原件式系統 V.S. 微服務



什麼是元件呢?

元件

- 應用程式中具有一定規模的程式結構
- 有一套定義良好的 API
- 可以被另一種實作取代
- 具有鬆散耦合性，支援重複使用的設計
- 可獨立部屬
- 可以是程式內部溝通，也可以是網路溝通

什麼是微服務呢?

微服務

- 有一套定義良好的 API
- 可以被另一種實作取代
- 具有鬆散耦合性，支援重複使用的設計
- 小巧，並且專注於做好一件事
  - 單一責任原則
- 獨立自主的服務
  - 獨立變更
  - 獨立部屬
- 主要依賴網路溝通

維持應用程式可發布狀態

- 一般程式碼管理使用『主線開發模式』
- 挑戰:一邊要維持新功能開發，一邊要改動系統架構
- 一種方法是建立分支，待工作完成合併
  - 讓主線一直處於可發佈狀態，直到分支合併
  - 會導致測試期和發佈時間拉長

變更的同時保持應用程式的可發佈

將新功能隱藏，直到完成為止

透過抽象來虛擬分支

將大型修改變成一系列小型增量式修改

使用元件

將新功能隱藏，直到完成為止

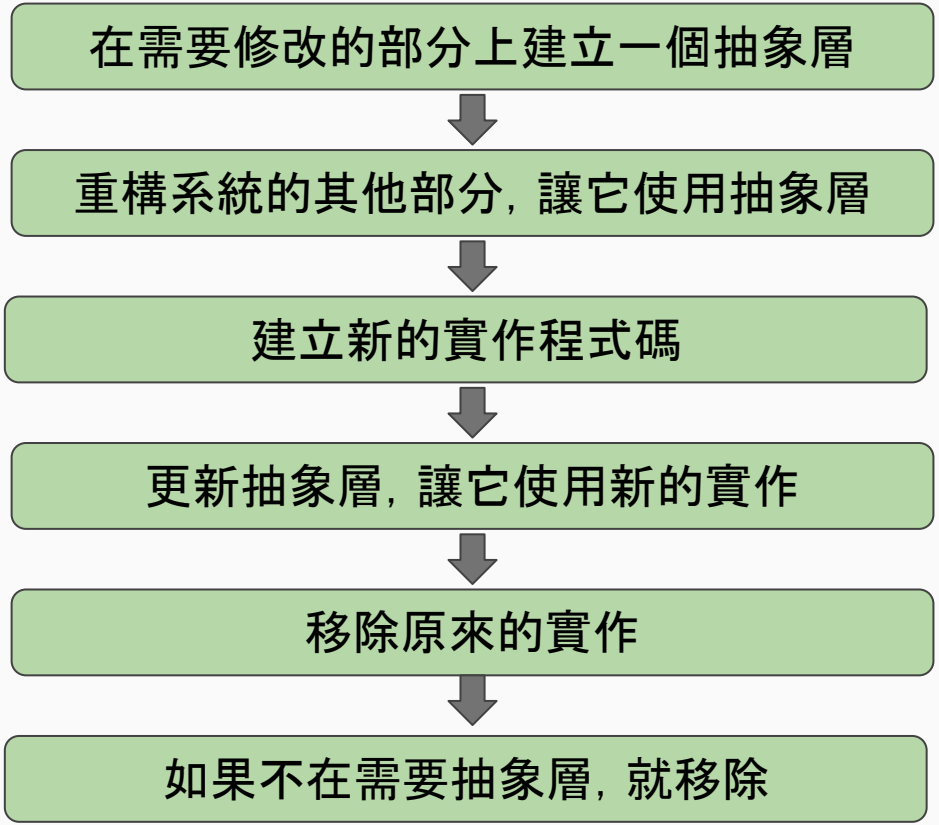
- 設置功能開關，只允許內部開發使用
  - Debug flag
  - Url redirecting
- 能確保功能開發一開始就與系統整合在一起

所有修改都是增量式的

- 最後階段才進行『合併所有東西』往往是最困難的
- 將大型的變更分解成一系列的小型修改
  - 同樣也是困難的工作
  - 維持了程式的可用性
  - 減少開發中斷造成的損失
- 可以『邊走邊兌現』，調整開發方向

透過抽象來虛擬分支

- 隔離程式碼庫中涉及修改的切入點
- 管理此部分開發中功能的所有修改



使用元件

- 幾乎現代的軟體系統都是由元件組成
- 具有一定的複雜度後才考慮將部分功能抽出成元件
- 元件開發的方式會讓軟體的開發流程更有效率
  - 將問題分成較小型且易於理解的程式碼區塊
  - 元件常能根據變化頻率的差異，呈現不同的生命週期
  - 鼓勵開發者使用清晰的職責描述來設計元件
  - 產生額外的自由度來最佳化建置和部屬流程

元件相依關係

元件 V.S.函式庫

- 由自己團隊或公司開發
- 可隨時異動
- 更新頻繁

- 由第三方開發
- 異動困難
- 更新頻率不高

管理元件相依

- 要一次編譯整個程式？ 還是當某個元件修改時，單獨編譯它?
- 建置時相依
  - 出現在程式編譯和連結時
- 執行時相依
  - 應用程式執行時需要依賴的元件操作

相依性地獄

- **依賴過多**

  一個軟體包可能依賴於眾多的庫

- **多重依賴**

  所需軟體包到最底層軟體包之間的層級數過多。容易 產生依賴衝突和依賴迴圈。

- **依賴衝突**

  兩個軟體包無法共存的情況。

- **依賴迴圈**

  即依賴性關係形成一個閉合環路

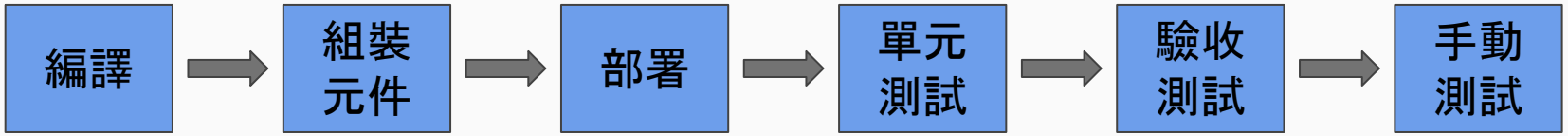
將元件流水線化

- 即使應用程式由多個元件組成，也不一定需要讓元件各自建置
- 使用一條建置流水線
  - 簡單有效
  - 容易追蹤到哪個部分破壞了建置

將元件流水線化

- 將系統分成多個不同建置流水線能讓系統獲益
  - 應用程式中的某些元件具有不同的生命週期
  - 應用程式的幾個功能由不同的團隊負責
  - 某些元件會使用不同的技術
  - 某些共用元件會不同的幾個專案使用
  - 元件相對較穩定，不需頻繁修改
  - 為元件建置較快，為整個系統建置較慢

元件建置流水線



- 快速回饋
- 提供建置狀態可視化

整合流水線

建置策略

1. 每當元件建置成功，就觸發整合流水線
2. 盡量建置整個應用程式，頻繁將最近版本的元件組裝起來

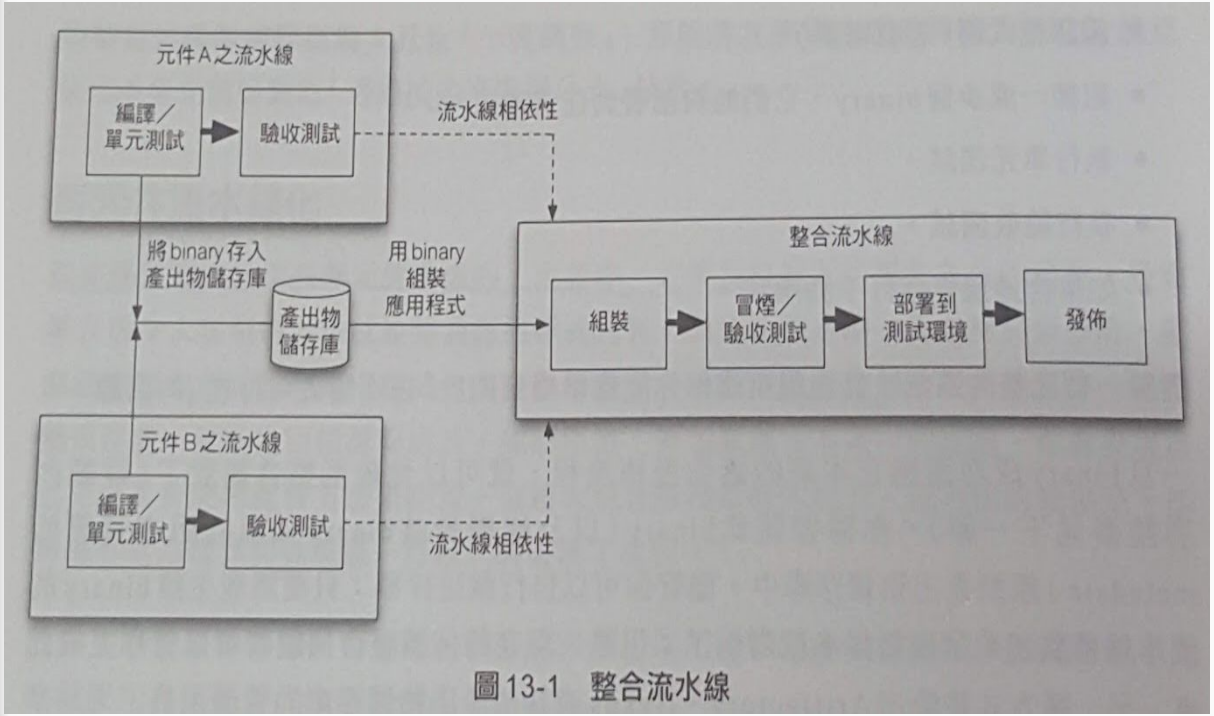


圖 13-1　整合流水線