

Composer 套件管理

范聖佑 Shengyou Fan
新北市樹林國小 (2015/07/06)

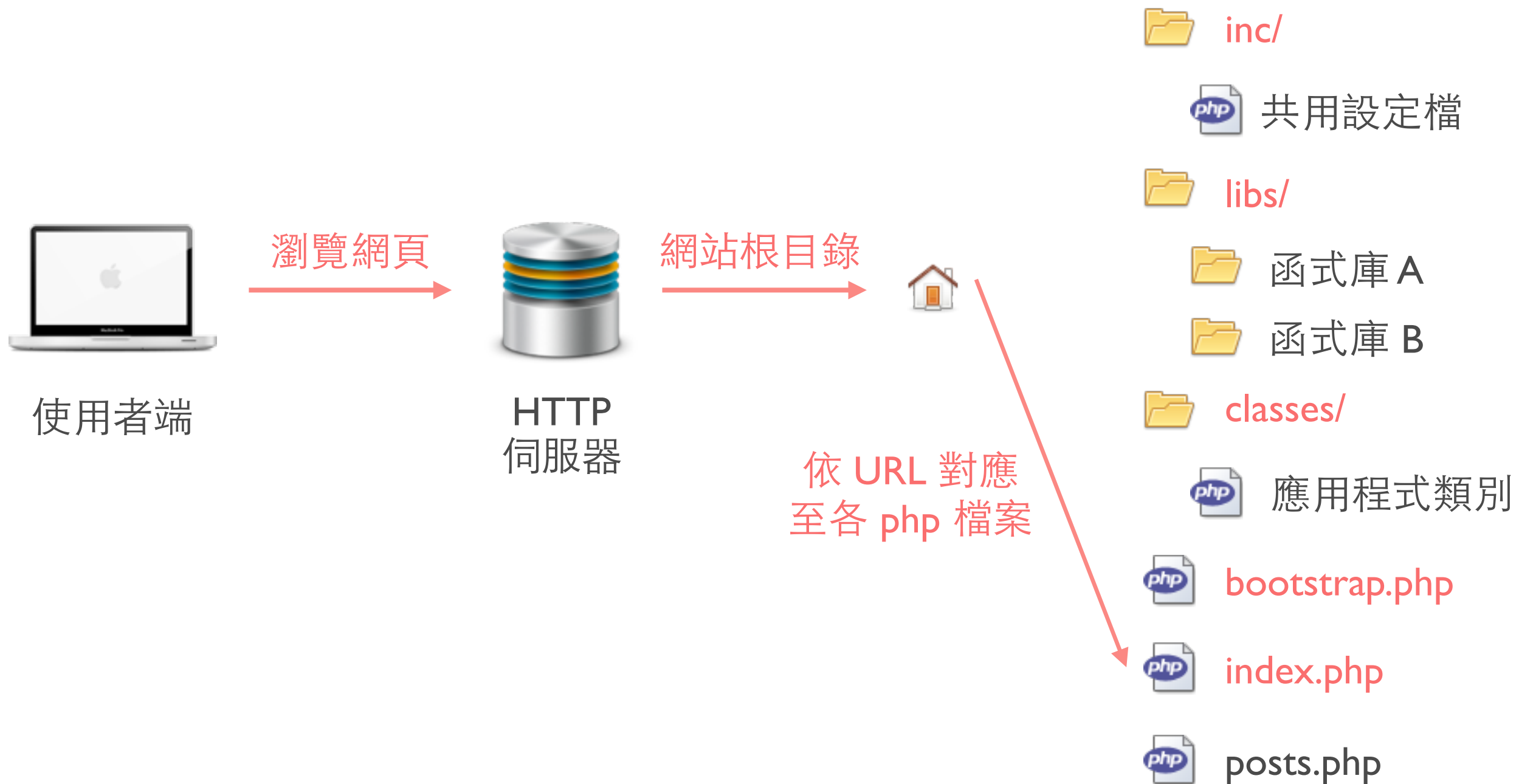
```
1  /**
2   * Display the specified resource.
3   *
4   * @param int $id
5   * @return Response
6   */
7  public function show($id)
8  {
9      $post = Post::with('comments')->where('id', $id)->first();
10
11      return View::make('post.show')->with('post', $post);
12  }
```

單元主題

- 比對傳統與現代網路應用程式開發方式的異同
- 了解現代網路應用程式在套件管理上的需求
- Composer 簡介、安裝與基礎指令
- Composer 錦囊妙計
- Composer Global 套件

現代 PHP 元件 與 Composer 套件管理

常見的 PHP 開發架構



libs 資料夾

- 通常放從 某來源 拿到的 某函式庫 (某外掛、某類包 ...)，也就是可以快速使用的工具包
- 將不同的函式庫依照資料夾分門別類放置。習慣好的話，會把該函式庫的版本號寫在資料夾上
- 把需要載入的函式庫路徑寫在上一層的 `bootstrap.php` 內統一管理
- 若需要更新函式庫版本時，就將舊版移除，把新版的資料夾移入，並手動更新 `bootstrap.php` 內的載入路徑

可能面臨的問題

- 雖然所有使用到的第三方函式庫看似整理在 `libs` 資料庫內，但卻沒有一套統一的管理機制：
 - 需要**手動**更新函式庫內容及載入設定，愈多需要人工的部份就愈容易有錯誤發生的風險
 - 沒有**相依性管理**機制，在新增、更新、移除第三方函式庫時，沒有辦法確保不會發生相依缺漏或衝突，增加開發與上線過程中的風險
 - 沒有統一的**元件載入**策略，因此每當需要新增一個元件時，就得手動修改 `bootstrap` 檔案，不但沒有效率且容易出錯

傳統的套件管理

- 為了解決這樣的問題，PHP 社群曾提出 PEAR/Pyru 專案來做 PHP 的套件管理，但經過數年的運作後宣告失敗，因為：
 - PEAR/Pyru 所提出的套件管理方案是針對整個系統而非專案
 - 套件設定檔編寫機制複雜，且官方的審核機制過嚴
 - 相較於其他程式語言的套件管理方式已太過陳舊，需要有更符合現代開發流程與工作習慣的套件管理方式
- PEAR/Pyru 的失敗造成大多數的 PHP 開發者常常重覆製造輪子且難以共享開發成果

現代 PHP 元件

- 把這些 libs 裡的函式庫視為 元件 (Component)，也就是一連串組合過的程式碼來幫助我們解決某個特定問題
- 從技術的角度來說，可以視為相關的 Classes、Interfaces、Traits 的組合
- 儘量以元件的概念來實作各種功能，將其他的開發者可以依需求更容易的散佈與取用
- 使用 PHP 元件做開發，可以減少框架獨佔性所造成的問題，也可以讓元件的適用性有更多的可能性

PHP-FIG

- 為了解決 PHP 元件管理及散佈的問題，在 2009 年 php|tek 大會上成立 PHP-FIG 組織，目的在於制定 PHP 社群在開發元件時的規範
- 由 PHP-FIG 所推動的通用元件規範，統一了所有 PHP 元件的 namespace、autoload 的作法，讓所有的元件開發者皆能依此遵循；也讓所有的元件使用者僅需學習一種方式就可以使用任何第三方元件
- 目前常聽到的 PSR-* 就是由 PHP-FIG 定義的規範

Composer 的興起

- 2012 年由 Nils Adermann 及 Jordi Boggiano 聯手開發了 Composer，特色包括：
 - 針對專案做套件相依性管理
 - 採用 JSON 做為設定檔編寫方式
 - 支援 SVN、Git 等現代常見的 VCS 系統做為套件原始碼管理及下載方式
- 目標在於取代傳統手動剪貼或 PEAR/Pyru 過時的管
理方式，讓所有的開發者都能透過指令很簡單的取
得/載入第三方元件

什麼是 Packagist ?

- PHP 的元件管理中心，開發者在自己的 **Github** 上儲存源始碼，並到 **Packagist** 上登記，就可以讓其他開發者取得該元件
- 在 **Packagist** 上面可以查詢到所有開放源始碼的第三方元件的**版本資訊**、**元件相依性**及**安裝/下載數**等統計資訊
- **Packagist** 讓 PHP 元件的散佈與更新變得容易且方便
- 簡而言之，**Packagist** 是尋找元件的地方；**Composer** 是安裝元件的工具

Composer 安裝

使用 Composer 的最低需求



+



+



php 5.3.2
(openssl extension)

composer.phar
執行檔

git
版本控制

*nix 環境安裝

```
curl -sS https://getcomposer.org/installer | php
```

若系統沒有 CURL 的話，可以用以下指令取代

```
php -r "readfile('https://getcomposer.org/installer');" | php
```

將 composer 變成系統全域指令

```
mv composer.phar /usr/local/bin/composer
```

installer 會檢查系統環境是否符合 Composer 的最低需求。若不符合的話會提示修正；若符合的話，就會自動下載 composer.phar 到當前工作目錄下

Windows 環境安裝

- Composer 提供了 Windows 版安裝程式，直接下載 <https://getcomposer.org/Composer-Setup.exe> 後，依照安裝程式步驟提示安裝即可
- 由於 Composer 相依於 PHP，因此在安裝過程中會尋問 php.exe 所在位置，請將本地端 PHP 安裝位置提供給安裝程式設定即可
- 請務必注意指定給 Composer 的 PHP 版本、php.ini 設定且正確掛載 Composer 所需的 extension

隨身版 (本地端安裝)

- 其實 Composer 是一個 PHP binary 檔，也就是所謂的 **PHAR** (PHP archive)。所以只要有 PHP Runtime，其實 Composer 本身可以做到隨身版 (免安裝) 的效果
- 可以直接將 **composer.phar** 放置在專案根目錄底下，只是在輸入指令時要改成：

```
php composer.phar {command}
```


composer [list]

- 顯示 composer 所有的指令
- 若在下指令的時候，突然忘了指令的全名時，可以先用這個指令查詢一下，再重新輸入指令即可
- 範例：
\$ composer
\$ composer list

composer {cmd} --help

- 部份 composer 指令可接受額外參數，若一時忘記參數的使用方式時，可先加 --help 於該指令後方，即顯示該指令的說明文件，待確認用法後再重新輸入指令

- 範例：

```
$ composer list --help  
$ composer install --help
```

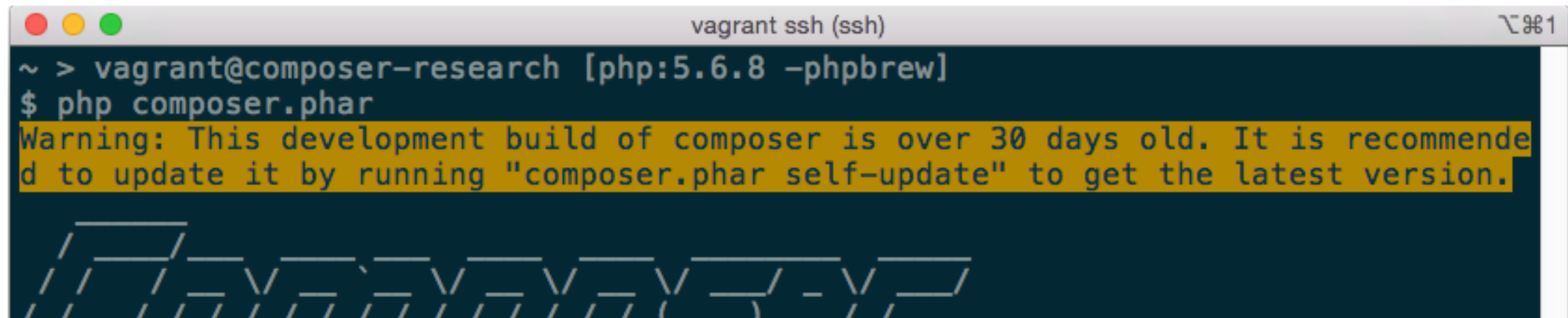
```
$ composer help list  
$ composer help install
```

composer self-update

- Composer 本身更新非常頻繁，除了增加新功能、修正錯誤外，也在逐步改善效能不彰的問題
- 請務必養成定期更新 Composer 的習慣，若超過 30 天沒有更新，在使用 Composer 指令時，也會有更
新提示
- 範例：

```
$ composer self-update
```

```
$ composer selfupdate
```

A screenshot of a terminal window titled 'vagrant ssh (ssh)'. The prompt is '~ > vagrant@composer-research [php:5.6.8 -phpbrew]'. The user has entered '\$ php composer.phar'. The output is a yellow warning message: 'Warning: This development build of composer is over 30 days old. It is recommended to update it by running "composer.phar self-update" to get the latest version.' Below the warning, there is a decorative ASCII art pattern of slanted lines.

```
vagrant ssh (ssh)
~ > vagrant@composer-research [php:5.6.8 -phpbrew]
$ php composer.phar
Warning: This development build of composer is over 30 days old. It is recommended to update it by running "composer.phar self-update" to get the latest version.
```

```
$ [sudo] composer self-update
```

現在就更新！

Composer 基礎操作

啟始一個 Composer 專案

- 在 Composer 的世界裡，所有的專案都是一個「套件」，所以即便我們只是要「使用」第三方套件，也必需先啟始一個套件專案
- 所有的 Composer 專案都依賴 `composer.json` 這個 `config` 檔來定義專案設定值。因此要啟始一個專案就得先寫 `composer.json` 檔
- Composer 本身有提供指令可以用互動問答的方式來幫助我們產生 `composer.json`。用這種方式可以比較輕鬆的撰寫專案設定檔

composer init

- 由 Composer 以互動問答的方式，幫助我們定義專案內各項必填設定值，快速地產生 `composer.json` 檔
- 一個 `composer.json` 至少需要填寫：套件名稱、套件描述、作者資訊、最低穩定度、套件類型、授權、相依性
- 範例：


```
$ composer init
```

如何尋找元件？

所有 PHP 元件都統一登記在 Packagist 上

Packagist *The PHP Package Repository*

[Browse](#) [Submit](#) [Create account](#) [Sign in](#)



Packagist is the main [Composer repository](#). It aggregates public PHP packages installable with Composer.

Getting Started

Define Your Dependencies

Put a file named `composer.json` at the root of your project, containing your project dependencies:

```
{
  "require": {
    "vendor/package": "1.3.2",
    "vendor/package2": "1.*",
    "vendor/package3": "^2.0.3"
  }
}
```

For more information about packages versions usage, see the [composer documentation](#).

Install Composer In Your Project

Run this in your command line:

```
1
```

Publishing Packages

Define Your Package

Put a file named `composer.json` at the root of your package, containing this information:

```
{
  "name": "your-vendor-name/package-name",
  "description": "A short description of what your package does",
  "require": {
    "php": ">=5.3.0",
    "another-vendor/package": "1.*"
  }
}
```

This is the strictly minimal information you have to give.

For more details about package naming and the fields you can use to document your package better, see the [about](#) page.

Commit The File

如何決定版本號？

- Composer 建議並希望所有的套件版本號都可以依循 SemVer 的規範。因此如何決定 composer.json 內的版本號要怎麼寫，就得先了解什麼是 SemVer？
- SemVer 的全名是 Semantic Versioning，其內容是提供開發者一個訂定軟體釋出時版本號的一種規範，讓版本號不再只是隨開發者高興來定義，而是讓版本號本身即具備意義
- 更多 SemVer 的全文可參考：<http://semver.org/>

X

.

Y

.

Z

Major

Minor

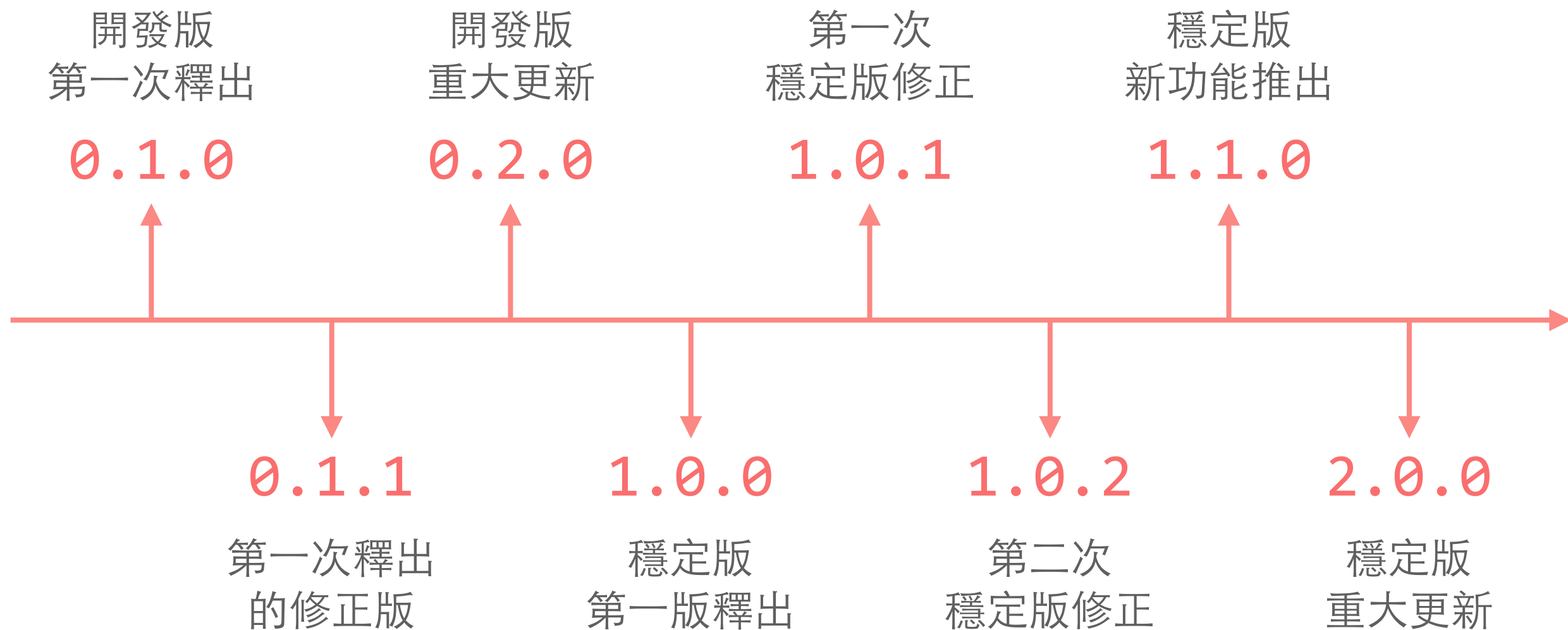
Patch

Breaks

Features

Fixes

版本號演進



Version Constraints

| | | |
|-----------------|--------------------------------|---|
| Exact Match | 1.2.3 | 1.2.3 |
| Wildcard Range | 1.0.* | ≥ 1.0 且 < 1.1 |
| Hyphen Range | 1.0 - 2.0 | $\geq 1.0.0$ 且 < 2.1 |
| Tilde Operator | ~ 1.2 | ≥ 1.2 且 < 2.0 |
| Caret Operator | $\wedge 1.2.3$ $\wedge 0.3$ | $\geq 1.2.3$ 且 < 2.0 $\geq 0.3.0$ 且 $< 0.4.0$ |
| Stability Flags | @stable @dev | dev > alpha > beta > RC > stable 預設抓 stable，但可用 @ 指定 |

填寫版本號的忠告

- 務必遵循 SemVer 規範
- 絕對不要用 dev-master 或是其他不穩定的版本
- 養成習慣上 Packagist 上查版本號，確認版本需求後再填 (若是該 Package 遵循 SemVer 的話，可使用 Caret Operator)
- 若對填寫版本號不是這麼有把握的話，可以先用 [Packagist Semver Checker](#) 驗證一下您的寫法
- 每次更動 composer.json 內容後，用 composer validate 檢查

composer validate

- 為了確保 `composer.json` 內的格式是正確的，Composer 提供了 `validate` 這個指令協助我們檢測
- 每次更動 `composer.json` 後，養成好習慣先用這個指令確保沒有錯誤後再執行其他指令
- 範例：
`$ composer validate`

composer install

- 套件安裝指令，`install` 指令會先檢查 `composer.lock` 是否存在？
 - 若不存在的話，就跑 `composer update` 來產生一個
 - 若存在的話，依照 `lock` 裡的 `package` 版本來安裝
- 所有的套件會下載至 `/vendor` 資料夾底下，並產生 Composer 的 `classmap` 做為套件載入的對照表
- 範例：
`$ composer install`

使用 Composer 自動載入

- 只要是透過 Composer 安裝的套件，就可以透過 Composer 自動載入，不再需要自行 `include` 或 `require` 了！
- 要讓 Composer 幫我們做自動載入，只需要在我們的程式最頂端加上一行，引入 Composer 的 `autoload.php` 即可：

```
require __DIR__.'/vendor/autoload.php';
```


Composer 進階技巧

composer.json

```
{
  "name": "{vendor}/{package-name}",
  "description": "{package description}",
  "keywords": ["{keyword1}", "{keyword2}"],
  "license": "{license}",
  "type": "project",
  "require": {
    "{vendor/package}": "{version}"
  },
  "require-dev": {
    "{vendor/package}": "{version}"
  },
  "autoload": {
    /* 套件載入設定 */
  },
  "autoload-dev": {
    /* 套件載入設定 */
  },
}
```

composer.lock

- 每一次 Composer 在解決完套件相依性後，會將最終決定安裝的版本號寫入 `composer.lock` 檔內，若把 `lock` 檔打開來看，會發現裡面是一個很大的套件對照表
- 務必把 `lock` 檔與 `json` 檔一同放進版本控制系統，隨著原始碼發佈給專案的其他開發者，而其他開發者拿到 `lock` 檔後，只要用 `composer install` 就可以安裝與自己一模一樣的套件版本，而不會有套件衝突或不一致的情況

composer show

- 可以查詢套件的版本及相關資訊
- `--installed (-i)` 可以列出目前專案裡安裝的所有相依套件資訊
- 範例：

```
$ composer show monolog/monolog  
$ composer show --installed  
$ composer show -i
```

scripts 區段

- Composer 在執行過程中有數個階段，為了讓開發者
在各階段可以有執行其他指令的機會，Composer 設
計了一系列的 event hook 供開發者使用
- 套件開發者可以設定在不同的事件發生時，可以呼
叫 Composer 執行指定的動作。這些動作都被紀錄
在 composer.json 裡的 scripts 區段
- 若因為任何原因而需要強制執行 scripts 區段裡的某
個事件指令時，可以透過 `composer run-script`
來呼叫 scripts 裡的動作

composer run-script

- 執行 composer.json 內 scripts 區段的各個指令，執行指令前需指定要執行的指令為何？
- 範例：

```
$ composer run-script post-root-package-install  
$ composer run-script post-create-project-cmd  
$ composer run-script post-install-cmd
```

自動載入的魔法

- 由於 `composer.json` 裡說明了該套件的載入方式，因此 Composer 就知道該怎麼載入對應的類別
- 為了加速 Composer 載入各類別的檔案，Composer 會預先將所有的類別與檔案的對照表，寫入 `vendor/composer/autoload_*.php` 的檔案內
- 所以當我們直接 `require __DIR__.' /vendor/autoload.php'`；時，就已經載入 Composer 的自動載入類別，後續的類別載入，就可以統一交給 Composer 處理

PSR-4

- 雖然 PHP 的自動載入設計可以讓開發者自行決定類別該怎麼被載入，但若每一個套件的載入方式都不同的話，對套件的使用者來說是非常痛苦的
- 由 PHP-FIG 定義的 PSR 系列規範中，PSR-4 就定義了套件該怎麼命名 namespace，及各類別與資料夾的放置方式，透過統一的規範下，Composer 僅需實作相容於 PSR-4 的自動載入機制，就可以幫我們載入所有的類別
- PSR-4 的 autoloading 策略是同步 PHP namespace 及 Class 放置於資料夾的對應做載入，概念簡單且易懂！

更新/安裝新套件

- 專案所使用到的相依套件會不定期更新，若要取得最新版的套件，則需要執行更新套件指令
- 若專案需要使用到額外的套件時，也需要透過更新套件指令，除了升級現有套件外，也一併安裝新的套件
- Composer 提供 `update` 這個更新指令，這個指令會重新分析 `composer.json` 裡的套件相依性，除了安裝新的套件外，也會更新現有的套件。所有動作完成後，也會更新 `composer.lock` 檔並更新 `autoload` 裡的 `classmap`

composer update

- 套件更新指令，Composer 會先檢查 `composer.json` 檔的內容，從中取得版本設定，並依照相依性判定要安裝的最終版本：
 - 判定後就會安裝所決定的所有套件
 - 安裝完後會更新 `composer.lock`
- 範例：

```
$ composer update
```

該用 `install` 還是 `update`

- 只要專案內已經有 `composer.lock` 檔、而又不需要更新/安裝套件時，就是使用 `composer install`
- 只有在安裝新套件、需要更新套件版本時，才需要使用 `composer update` 指令。這通常都是專案核心開發/管理者才需要做的動作
- 由於 `composer install` 會直接使用 `lock` 檔內紀錄的套件直接安裝，跳過解析套件相依性的過程，且安裝時也會儘量用快取過的檔案來安裝。因此在記憶體使用、運作效能及安裝速度上都會有很大的提升！

Global 套件

composer global

- 大多數的 Composer 元件是獨立安裝於各別專案內，但 Composer 元件也可以做為散佈指令列工具的一種方式。
- 透過將元件以 global 的方式安裝，Composer 會將該元件安裝在 COMPOSER_HOME 目錄底下，將 COMPOSER_HOME/vendor/bin 加入 PATH 後，即可全域使用該指令
- 範例：

```
$ composer global require vender/package:version
```

```
$ composer global require laravel/installer:"^1.2"
```

(安裝 laravel-installer)

★ 套件會安裝至 COMPOSER_HOME :

Windows 位置 : C:\Users\USERNAME\AppData\Roaming\Composer\vendor\bin

*nix 位置 : ~/.composer/vendor/bin

設定環境變數 (PATH)

- 需要在作業系統裡將 Composer Global Package 的路徑設定至環境變數內，這樣當我們執行指令時，作業系統才能知道該去哪裡尋找我們要執行的程式
- 環境變數裡可以設定多個路徑位置，作業系統會依照填寫路徑的順序逐一去尋找指令程式。若都找不到才會回傳錯誤
- 各家作業系統設定方式略有不同，請參考各作業系統環境變數設定的教學。若使用 wagon 的話，在 cmd 啟動時會自動載入搭配預載 Composer 的 Global Package 位置進環境變數內

(*nix 作業系統在 .bashrc 裡加入 PATH)

```
export PATH=$PATH:~/composer/vendor/bin
```

(wagon 則是在 wagon\cmdr\vendor\init.bat 裡設定)

```
@set COMPOSER_HOME=%WAGON_ROOT%\composer
```

```
@set PATH=%COMPOSER_HOME%\vendor\bin
```



```
$ composer global update
```

(日後要更新 Global 套件時)

單元總結

- 在這個單元裡我們學到了些什麼？
 - PHP 元件與套件管理的需求
 - Composer 的安裝方式
 - Composer 基礎操作與指令
 - Composer 進階技巧
 - Composer Global 套件



歡迎提問討論